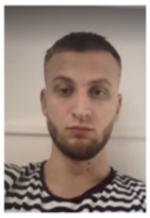
02312 Introductory Programming Fall 21
Milestone 1:

Projektgruppe: 11

January 4, 2022



Sofie Groth Dige	Mathilde Elia	Anshjyot Singh	Nick Tahmasebi	Marco Miljkov Hansen
s211917	s215811	s215806	s195099	s194302

Technical University of Denmark

Timetable

Nick	120 timer
Marco	100 timer
Ansh	130 timer
Sofie	115 timer
Mathilde	130 timer

Introduction:

Matador is the Danish version of the world renowned “Monopoly”. Matador, is a multiplayer board game, focusing on the economic aspects of the real world.

The aim of the game is that the players rotate around the board, purchasing properties and testing your luck with the chance cards. The winner (The Matador), is the last man standing, after the remaining players have been bankrupt. The game has a number of different rules, which makes the game Matador. We have specified these rules in our ‘Analyze’ Section, and furthermore prioritized them following the MoSCoW method.

We have been tasked with creating our own Matador board game, which includes the important features, such as purchasing properties, buying houses and hotels, and creating a number of different chance cards. All this has to be implemented in the given GUI, to show off our vast range of coding skills. Our starting point for our code is from CDIO3, where we also were tasked in making a less complex variant of Matador.

Customer vision:

The customer has a vision on developing a Matador game. The game should implement as many elements from the real game as possible. The game must be able to be played between 3-6 people.

Project planning:

Before we, in the group, started with the project itself, we sat together and made sure to prepare a plan for how we should approach this task. A way in which we could clarify the priorities in our project, as well as make an internal agreement which was based on which requirements had to be resolved first before we continued working on other requirements or queries for the project. We made use of the MoSCoW model, which describes the degree / order in which the requirements should be resolved. The “need to have” requirements were of course at the top of our priority list. We also believed that the planning for the work should be improved, improved in relation to areas of responsibility and better contact between the group members.

Analysis:

- Requirements:

In this part of the report we describe our requirements furthermore, with our risk analysis which focuses specifically on the risk that may come during the development of the project. The board we will base our game on is referred to on appendix nr. 1: board.

Functional Requirements with the MosCow Model

ID	Description	Status
M	**Must Have**	
M01	A game between 3-6 people, who are above the age of 10+	🟩
M02	Roll 2 dice, and ensure the outcome of the dice is correct. E.I (Described further in the extensions)	🟩
M03	The game must have 40 Properties that each have their own effect.	🟩
M04	Everytime the players pass “START”, they recieve 4.000 DKK from the bank.	🟩
M05	There are 6 “Chance” fields, where the player draws a chance card.	🟩
M06	The players must be able to land on a property and then continue from the spot.	🟩
S	**Should Have**	
S01	The Dice must have 6 sides.	🟩
S02	The players must start with 30.000 DKK.	🟩
S03	When a Player buys a property, the player must receive rent, when others land on the owned property.	🟩
S04	When everyone, except one player, goes bankrupt, the remaining player is then the games only “Matador”. E V - Edge case, if all players have the same amount of money. (Described below)	🟩
S05	The players must be able to increase the rent, by buying houses and hotels.	🟩
S06	To buy houses, the player must own every property of a specific colour.	🟩
S07	To buy hotels, the player must own 4 houses on the applicable property.	🟥
S08	The system must double the rent of applicable properties when all deeds in the	🟥

	same color group are owned by the same player.	🟥
S09	The players must start on “START”.	🟩
S10	The players must move clockwise around the board.	🟩
S11	Players must be able to pawn their deeds.	🟥
S12	A player must go directly to jail if they land on the field "Go to Prison".	🟩
S13	Players who are sent to jail must not receive money for passing the start-field.	🟩
S14	The system must give the player an extra turn if they roll a pair (eg 2 sixes).	🟥
S15	The system must send a player to jail if they roll a pair three times in a row.	🟥
S16	When a player goes bankrupt, all their plots must be sold at an auction.	🟥
S17	When a player owes more than they own, they become bankrupt.	🟥
S18	Bankrupt player’s values are to be transferred to their creditor.	🟥
C	**Could Have**	
C01	When a player lands on an amusement-property that hasn’t been purchased, it can be purchased from the bank.	🟩
C02	The system must handle the players' inbound and outbound payments.	🟩
C03	The system must support trading between players.	🟥
C04	If a player does not buy the deed for the field they land on, the deed must be auctioned.	🟥
C05	Every player must be able to bid during an auction.	🟥
C06	The system must, during an auction, give the deed to the highest bidder.	🟥
C07	The system must give the option of playing a normal and fast version of the game.	🟥
C07.1	During fast mode the system must deal two random deeds to each player.	🟥

C07.2	During fast mode the game must stop when chosen playtime is over.	🟥
C8	The chance-card deck is shuffled in the beginning of the game.	🟩
C9	The system must be able to calculate 10% of player values for the income tax.	🟥
W	**Wont Have**	🟥
W01	If the player forgets to charge rent, after someone lands on their property, they lose the right to the rent.	🟥
W02	The money must be distributed as follows: 2 x 5.000 DKK., 5 x. 2.000DKK., 7 x. 1.000 DKK., 5 x. 500 DKK., 4 x. 100 DKK. og 2 x. 50 DKK.	🟥
W03	The chance-card deck must be placed in the middle of the board, faced down.	🟥

Non-functional Requirements

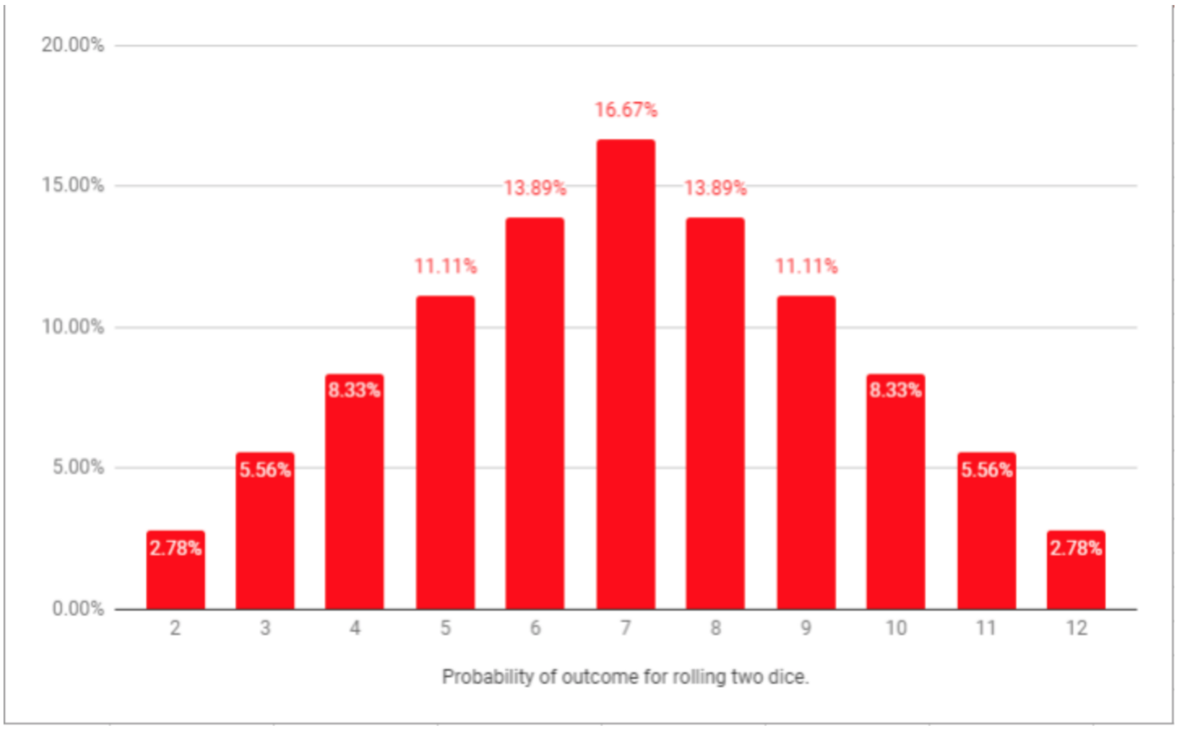
ID	Beskrivelse
IF01	Players must be able to play the game without instructions. (Ordinary people) E.II (Will be described in extensions in the next table)
IF02	The system must be able to run on the Windows machines in the databars at DTU.
IF03	There must be a link to our github repo, for the client to inspect the code.
IF04	Testing (JUnit) must be performed as it is a requirement for reliability.
IF05	Test-code confirming that the account balance can never be negative.
IF06	The game must be easy to translate into other languages
IF07	It should be easy to switch to other types of dice

There have been added extensions to our requirements, to make them measurable.

Extensions to our requirements

E.I.

Regarding requirement **M02** and **S01**, where the die must work correctly, it is believed that the die must be able to generate a random stroke. As well as follow the theoretical distribution for 2 dice, which is a distribution like the diagram below. This distribution is the probability of getting the different numbers from 2-12.



E.II.

By “Normal people” in requirement **IF01**, it means people who don't separate from the average. For example, people without striking abilities should be able to play the game without an instruction manual.

E.III

In requirement **IF06**, it is meant that the code can be written in several languages in the language class, so that others do not have to change the code, but only the text that is printed out. To make the requirement measurable, one could e.g. say that the program must be able to be translated within a certain time interval.

E.IV.

In requirement **IF07**, it is meant that one should change the cube from a 6-sided cube, to any x-sided triangle.

E.V.
In requirement **S04**, there is an edge case. This means that in the game a player does not have any money, the player with the most money wins. This can create a problem if the rest of the players have the same amount of money. This problem can be solved in different ways. For example, we could say that the players with the most money must roll the dice to decide who wins. So when all other players except one are bankrupt, the last player left wins.

E.VI.
An extension to requirement **S12** and **S14** are the different ways to get out of prison:

- By paying DKK 1,000 ,. before rolling the dice
- By using the “get out of jail” card (which can be deducted from the Chance-card pile)
- By rolling 2 identical dice. You move the number of fields forward, which the eyes show and you would still have an extra throw.

One can not stay in jail for more than three rounds. Ie. that you must pay the fine of DKK 1,000 and get out of prison after the 3 overturning rounds.

When you are in prison you can still buy land (by auction or trade between the players), but you can not charge rent from the other players.

You can get into the unfortunate situation of being in prison for more than 3 rounds, and end up having to pay a fine of DKK 1,000. But if you have no money to pay with, it means that you have lost the game and are bankrupt.

We have chosen to manage the functional requirements into the MoSCoW model. It makes it easier to manage and prioritize our requirements into four categories such as: ‘Must have’, ‘Should have’, ‘Could have’ and ‘Won't have’.

The ‘Must Have’ requirements are necessary to play the game, they are the most important and they need to be implemented. They are a part of the MVP , which is the ‘minimum viable product’, for the game to function as it should, to be playable. The ‘Should have’ requirements is the second most important, which is also going to be implemented in the project, because it is some of the features which makes the game more fun and challenging.

The ‘Could have’ requirements are going to be implemented if we have the time at the end of the project. We have chosen not to deal with requirement **W01**, because we believe that the system automatically charges rent for the players. So the players won't be able to forget to charge the other players themselves. Requirement **W02** will also not be included in the product, because the system handles the players balance as credit, and not with bills.

- Use-case diagram

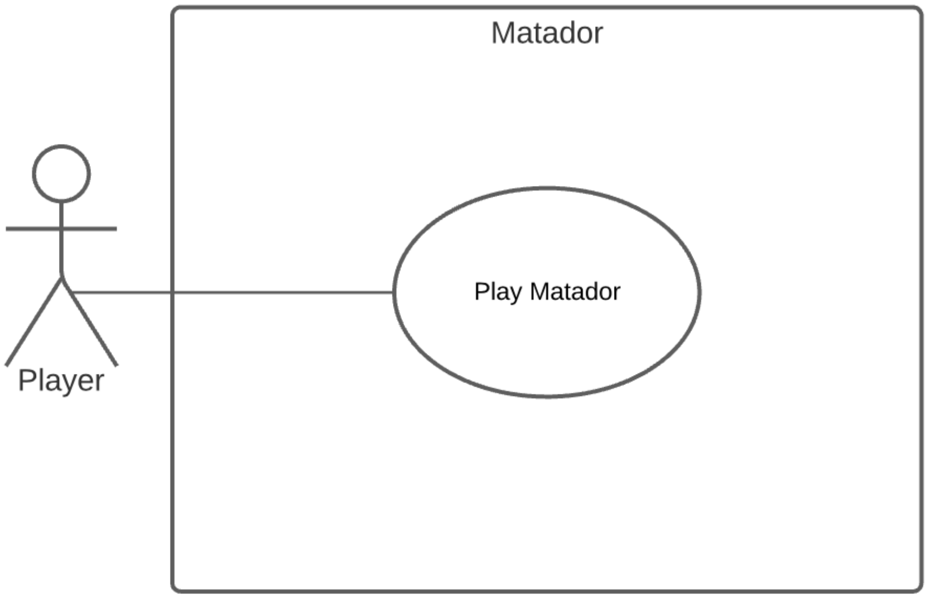


Figure.1 Use-case Diagram

A use case diagram is the primary form of system/software requirements for a new software program. Use cases specify the expected behavior (what), and not the exact method of making it happen (how). In this instance our use-case diagram includes a primary actor, which refers to our “player/s”, who has the task of playing the game, hence “Play Matador”. We have chosen to keep our use case diagram simple and elaborate on it, in our use case description.

After identifying the primary use case “Play Matador”, it has been written as a fully dressed use-case (see UC1). What appears under the main success scenario is what is needed for a plain functioning board game. The extensions are where all Matador functions have been identified and written down to what has been analysed as the game the client has asked us to make.

With the use case it becomes clearer what needs to be implemented, and what each function needs to contain.

UC2 is a brief format use-case for the fast version of Matador. The game setup and end differs from UC1 but they share the same game flow and logic.

Use case description

Scope: Matador
Level: User-goal
Primary actor: Player
Stakeholders:

- Player: Wants to play a digitized version of the board game Matador, that works and is intuitive.
- Client: Wants a digitized version of the board game Matador, that works and satisfies the player.

Preconditions: Player has started the game, chosen number of players and player names have been registered.
Success Guarantee: Game has been played and the system terminates.

UC1: Play Matador (Main success scenario)

1. System begins the player's turn.
2. System asks the player to throw the dice.
3. Player throws dice.
4. System moves the player the number of fields corresponding to face value.
5. Player lands on a field.
6. Player's turn ends.
7. System passes turn to the next player.
   Step 1 to 7 continues until there is one player left
8. System shows who won the game.
9. Player ends game.
10. System terminates.

Extensions (alternative flows)
*a At any time player can't afford chosen purchase or action.
    *a.1 System informs the player that they can't afford current action.

1.a At the beginning of each turn, System gives the player an option to trade possessions.
    1. Player chooses to trade.
    2. System shows all player possessions.
    3. Current player chooses what they want to trade.
    4.a Player, who the current player wants to trade with, chooses to accept.

                4.a.1 System completes the trade.
        4.b Player, who the current player wants to trade with, chooses to decline

1.b Player, that is in prison, has turn
        1. System gives the player the option to pay 1000 kr. or throw the dice.
                1.a Player has a “løsladelses” card.
                        1.a.1 System gives player the option to use card
                                1.a.1.a Player uses card
                                        1.a.1.a.1 System puts card at the bottom of the deck.
                                        1.a.1.a.2 Player is released from prison.
                                1.a.1.b Player does not use card
                                        1.a.1.b.1 System gives player the two other options

        2.b Player chooses to pay 1.000 kr.
                1 System withdraws 1.000 kr.
                2 Player is released from prison

        2.c Player chooses to throw dice.
                1.   Player throws dice
                2.a Player gets a pair.
                        2.a.1 Player is released from prison.

                2.b Player doesn’t get a pair.
                    Continues from step 6

1.c Player, who owns a deed / multiple deeds, has turn
        1. System gives the player the option to pawn a deed.
                1.a Player chooses to pawn a deed.
                        1.a.1 System shows the player’s owned deeds.
                        1.a.2 Player chooses what they want to pawn.
                        1.a.3 System pawns the deed and deposits the deeds pawn-value to the players account.
                        1.a.4.a Player owns more deeds.
                                Continue from step 1.c

1.d Player, who is eligible to buy houses and or hotels, has turn.
        1. System gives player the option to buy buildings
                1.a Player chooses to buy buildings
                        1.a.1 System shows where player can place buildings.
                        1.a.2 Player chooses where and what they want to place.
                        1.a.3 System withdraws amount from player’s account.
                        1.a.4 System places chosen building on chosen field.
                                1.a.4.a If building is the first hotel in the street color group.
                                        1.a.4.a.1 System removes all houses placed in color

group.
Continues from step 1.c.1

3.a Player throws a pair three consecutive times.
1 System moves the player's game piece to the prison field.
2 Player gets imprisoned.

5.a Player lands on a street / enterprise.
- 1.a Field isn't owned by another player.
  - 1.a.1 System gives the player an option to buy what's on the field.
    - 1.a.1.a Player chooses not to buy the deed.
      - 1.a.1.a.1 System sets deed on auction.
        Continues from step 6
    - 1.a.1.b Player chooses to buy the deed.
      - 1.a.1.b.1 System withdraws the amount from players account.
      - 1.a.1.b.2 System makes the player owner of the street / enterprise.
      - 1.a.1.b.3 Player receives the deed.

- 1.b Field is owned by another (not imprisoned) player.
  - 1.b.a.1 System transfers the rent amount from the player to the owner of the field.
  - 1.b.b Player lands on a Soda field that is owned by another player
    - 1 System asks the player to throw the dice.
    - 2 Player throws dice
    - 3 System determines the concerned amount from the dice's face value.
    - 4 System transfers the concerned amount from the player to the field owner.

5.b Player lands on or passes the "START" field.
1. Player receives 4.000 kr.

5.c Player lands on the "Prøv lykken" field.
1. System asks player to draw a card
2. Player draws a card.
3. The card's action is performed.
4. System places the card back at the bottom of the deck.

5.d Player lands on the "De fængsles" field.
1. System moves the player's game piece to the prison field.
2. Player gets imprisoned.

5.e Player lands on the "I fængsel" field.

5.f Player lands on the “Betal indkomstskat” field.

1. System gives the player an option to pay 4.000 kr. or 10% of their value.
2. Player chooses one of the payments
3. System withdraws the chosen amount from the player.

5.g Player lands on the “Ekstraordinær statsskat” field.

1. System withdraws 2.000 kr from the player.

1-6.a Player goes bankrupt (owes more than they own)

- 1.a Player’s creditor is another player
    - 1.a.1 System transfers money and deeds from bankrupt player to creditor.
        - 1.a.1.a Bankrupt player owns buildings
            - 1.a.1.a.1 System removes the players' buildings.
            - 1.a.1.a.2 System transfers the building's value to the creditor.
    - 1.a.2 System removes bankrupt player’s game piece
- 1.b Player’s creditor is the bank
    - 1.b.1 System removes player’s possessions and game piece from the board.
    - 1.b.2 System starts auktion with the possibly owned deeds.

1-6.b An auction has started.

1. System shows the auctioned deed with starting bid on 0 kr.
2. System gives the player options to bid or pass.
    - 2.a Player chooses to bid
        - 2.a.1 Player writes down bid (rounds to nearest 100)
            - 2.a.1.a Player writes invalid bid
                - 2.a.1.a.1 System informs player that the bid is invalid
                  *Continues from step *b.2*
        - 2.a.2 System updates the bid.
    - 2.b Player chooses to pass.
3. System proceeds to next player.
   Step 2 and 3 repeats until every player has chosen to pass.
4. System cashes the bid value from the highest bidder.
5. The highest bidder gets the deed.

6.a Player, that got a pair at the beginning of their turn, has ended their turn.
Continues from step 1

UC2: Play Matador (Fast version)

Primary actor: Player

Basic flow: Player starts the game and enters their desired number of players and the amount of time they want to play. The system creates the amount of players and deals two random deeds to each player, which they now own. The game follows UC1, until the chosen time has passed. The system identifies the wealthiest player and declares them as the winner of the game.

- Analysis-diagrams

System sequence diagram:

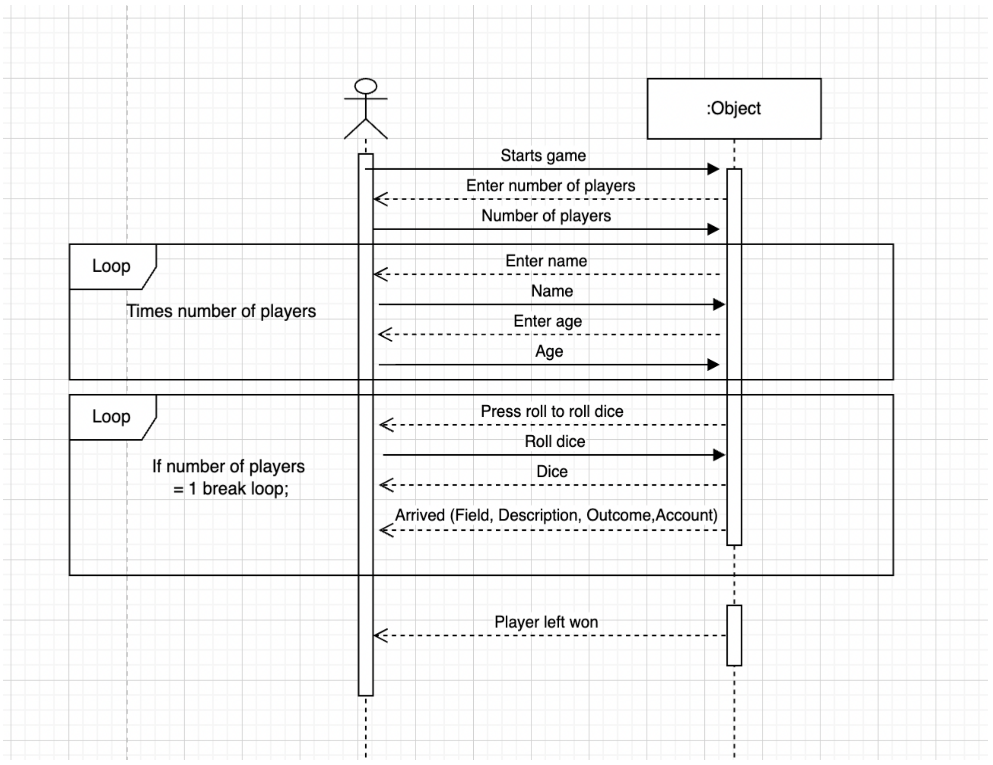


Figure 2. System Sequence Diagram

The system sequence diagram depicts our main success scenario.

The user starts the game, and the matador program will ask for the number of players which the user will enter. Then the program will ask for the name and age of a player, this part is looped by the number of players entered. The user will then be asked to roll the dice and after rolling the dice the game will return the value of the dice and will return the arrived function for the field landed on, a description of the field, an outcome which will trigger a choice depending on what you land on, and the effect on the players account. This is looped as long as there is no winner, and when there is one player left the loop will break and the system will return a message player left won.

Domain model

Our domain model consists of 8 classes, “Player”, “Dice”, “Cup”, “Account”, “Board” “Fields”, “Chancecard” and “Carddeck”. The cup holds 2 dice, and the players shake the cup, which is conceptual with behavior from real life. The player plays on the board and is connected to a bank account, because it holds the player's balance in the game. The Player is able to draw a card from the Card Deck, which is connected to 36 Chance Cards. At last the board consists of 40 Fields. It is incorporated with both behavior and data.

Our domain model looks as follows:

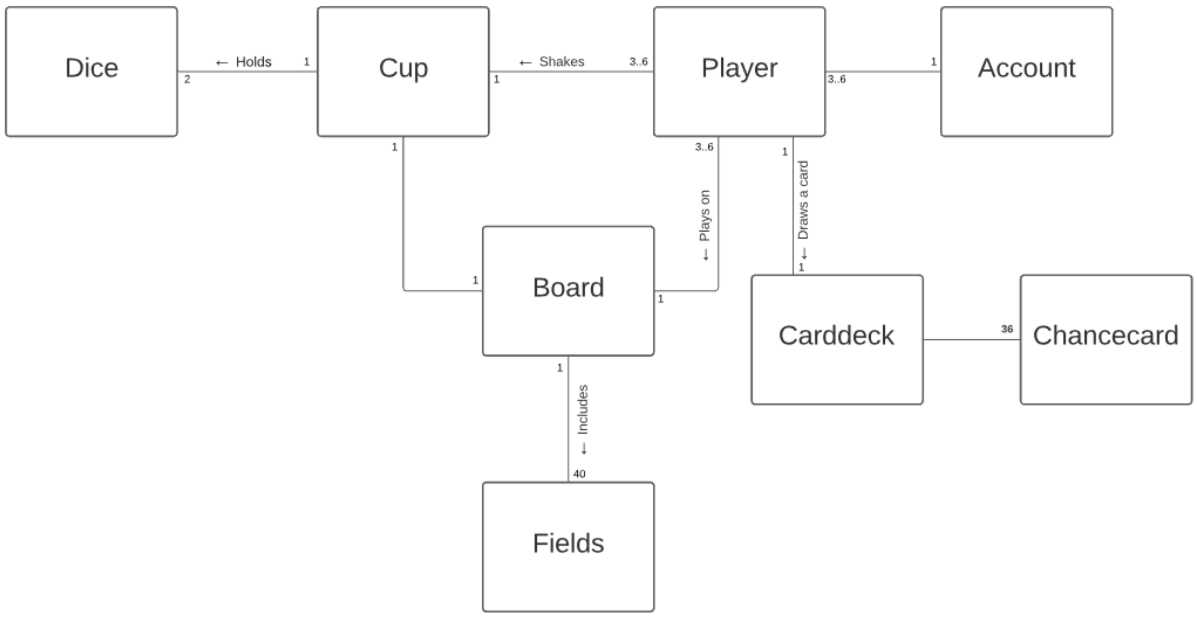


Figure. 3 Domain Model

Design:

The approach towards the project design has been to take the starting point in the previous project, Matador Jr, and add and adjust functions to make it fit the original Matador game. For these new functions to work with a clean and reusable code, there had to be some design changes.

Firstly, there was a problem with dependencies between multiple different classes. An example of this can be seen with the GUIController-class which was initiated in every field-class and so was the Player class. This created extremely high coupling for these two classes.

Before realizing the high coupling system, the plan was to make the ChanceField class responsible for creating and handling the chance cards. We later decided against that, since we wanted the ChanceField to function as only a model-class for the ChanceField itself and it would have added more couplings. It would go against the typical model-class structure to have it filled with game logic. Having the ChanceField be responsible for handling the chance cards would also go against the high cohesion principle which is to have classes have as small and clear responsibility range as possible.

The solution we decided upon instead was to implement the use of controllers into the design. These would work as an "indirection" of these couplings and support the low coupling principle to make it potentially more reusable. These controller classes will be associated with each other to free other classes of unnecessary coupling (see Figure 4).

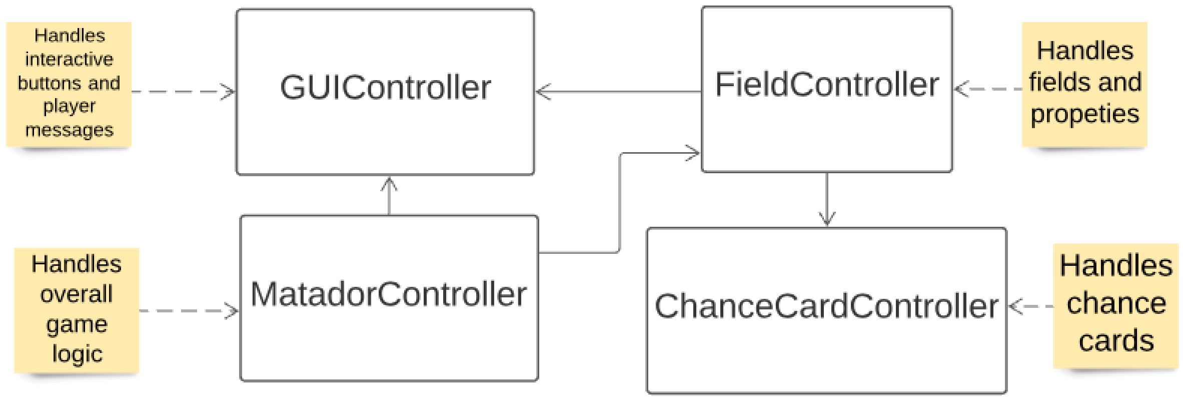

Handles interactive buttons and player messages
GUIController
FieldController
Handles fields and propeties
Handles overall game logic
MatadorController
ChanceCardController
Handles chance cards

Figure. 4 Interaction between Controllers

Since all fields have the clear common ground that they are all fields, it would be an advantage to use inheritance. We therefore want to implement an abstract Field-class that functions as a parent class for all the model Field-classes. The same goes with the chance cards, where classes representing different types of chance cards can extend from an abstract parent class called "ChanceCard".

Sequence diagram:

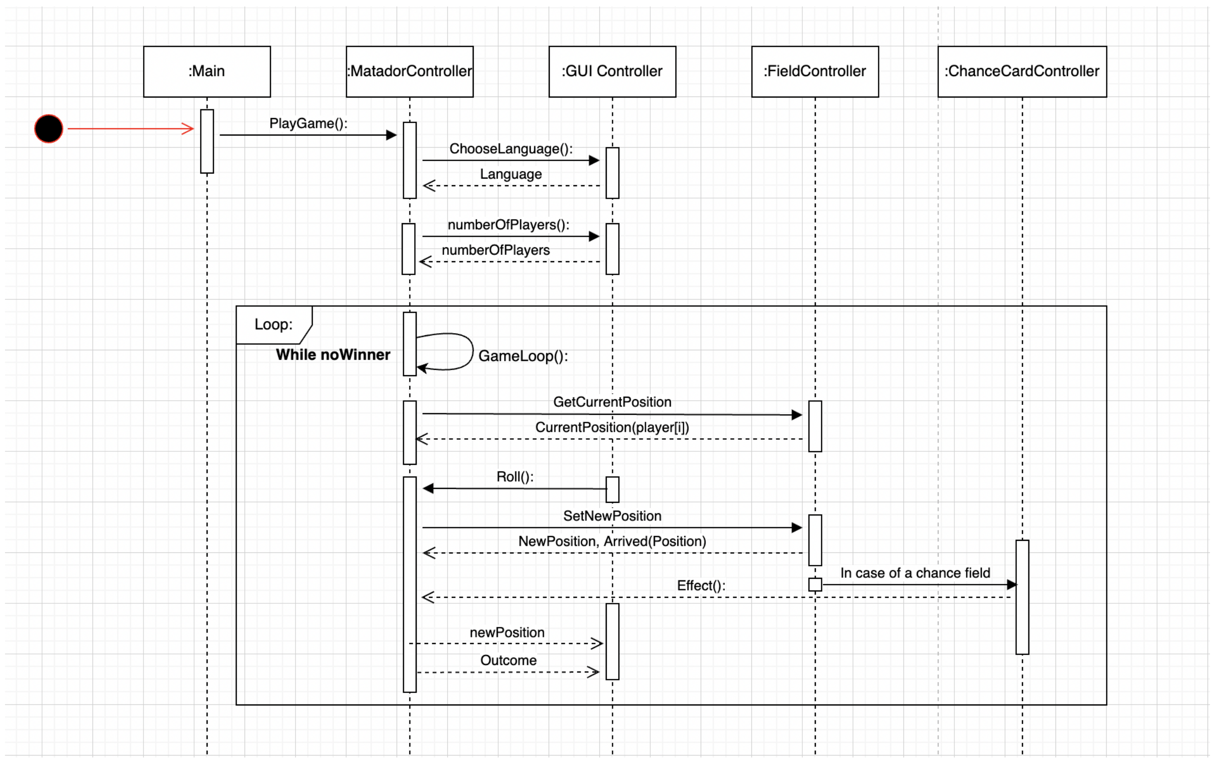


Figure. 5 Sequence Diagram

Our Sequence diagram depicts the communication between all our controllers. We have decided to leave out all other classes since the diagram quickly would become unreadable.

When the game is started our main class will call the play game function that resides inside the MatadorController. The matador controller begins by initializing the board and the GUI controller will ask the user to choose a language and return what language will be used to our matador controller, (at the moment the only language in our language class is english). The number of players function is then called to the GUIcontroller and the number is returned.

The Game loop function is called, this makes the MatadorController get the positions of the players from the FieldController and the positions are returned. Then the player will press roll on their screen, and the value of the roll will be returned to the Matadorcontroller which will make the field controller set the new position of the player. The FieldController will return an “arrived” function based on what kind of field the player has landed on. In case of it being a ChanceCardField the chance card controller will draw a card from the chance deck, and its effect will be returned to the MatadorController. The controller will return the new position and the outcome of either the chance card or the outcome of the player landing on another field to the GUIcontroller.

Also, with the goal of implementing all chance cards specified for the project, we decided it would be more appropriate to make the cards in an array, instead of a switch case, which was used previously. The switch case would become too long, unmanageable and there would be an excess of repeated code. There will also be added a parent class called ChanceCard that classes representing the different types of chance cards can extend from (see Figure 6).

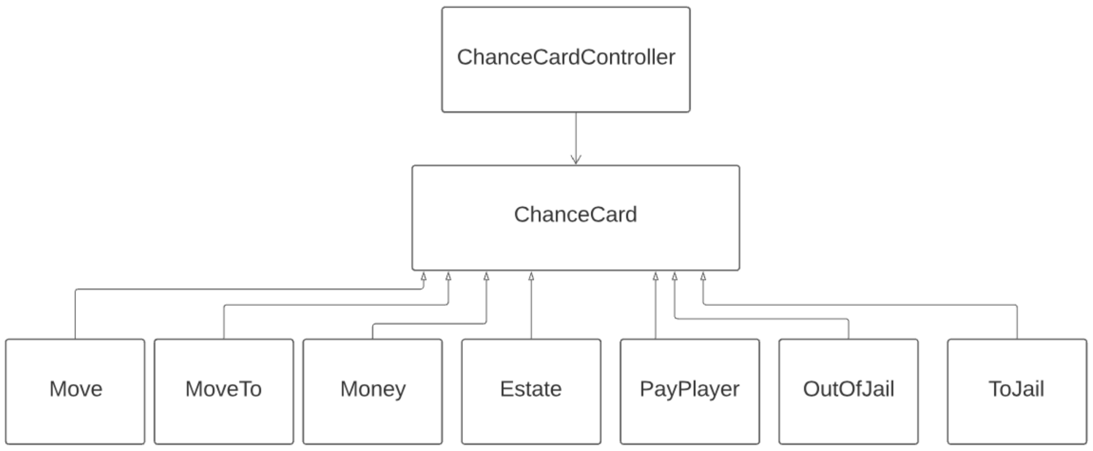
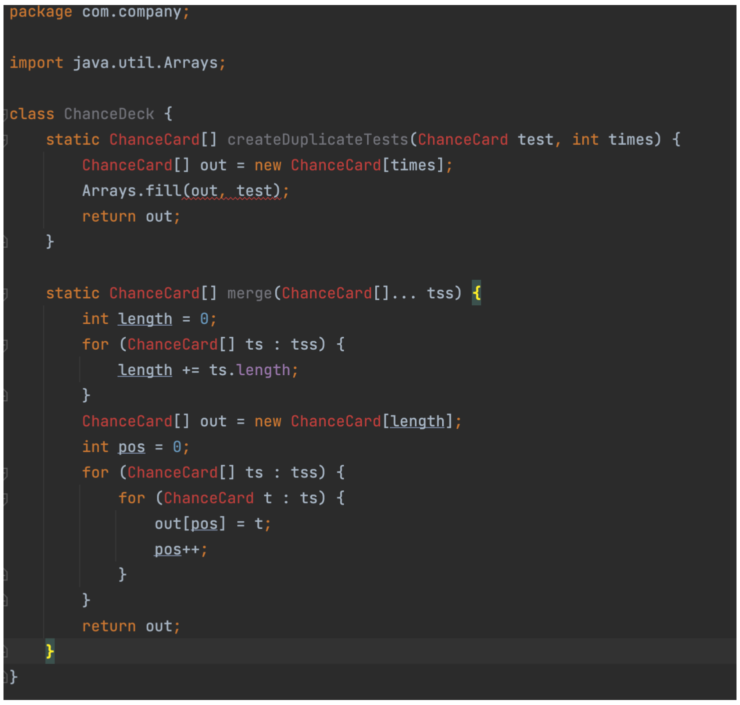


Figure. 6 How our Chance Cards are implemented

Each chance card has their own action, which is activated when drawn. Depending on these actions the cards have been grouped into types, to gather cards with the same card-logic.

With these grouped classes, we avoid having to copy the same card-logic for each card and instead simply check if the drawn card is an instance of one of the sub-classes who all behave the same. Inheritance is also used here for creating a "template" chance card through the parent class, which will make all the child classes at least contain a message and some sort of value.

We fell upon the dilemma of if hardcarding the different amount of chance cards was necessary. This was carefully thought about, due to our vision of reusing our code for further instances. In the instance of a certain card occuring a heavy number of times, it would be easier to tell the program how many of each card was required.



```java
package com.company;

import java.util.Arrays;

class ChanceDeck {
    static ChanceCard[] createDuplicateTests(ChanceCard test, int times) {
        ChanceCard[] out = new ChanceCard[times];
        Arrays.fill(out, test);
        return out;
    }

    static ChanceCard[] merge(ChanceCard[]... tss) {
        int length = 0;
        for (ChanceCard[] ts : tss) {
            length += ts.length;
        }
        ChanceCard[] out = new ChanceCard[length];
        int pos = 0;
        for (ChanceCard[] ts : tss) {
            for (ChanceCard t : ts) {
                out[pos] = t;
                pos++;
            }
        }
        return out;
    }
}
```

Figure. 7 Duplicate Chance Cards

This is where our idea of an array of arrays occured. Our vision was to create a ChanceDeck class. ChanceDeck would use the Array.fill function which, as suggested, filles the array given the information above. We also included static ChanceCard, being static this has the advantage and allows many classes access to behavior that isn't dependent on an instance of any of them, This would lead to every instance of a chance card would create an array[x] with x being the number of times the card occured. We then would merge the arrays together, so we would have a deck of cards, which would be adjustable to every game. This code was implemented in our ChanceCard branch, but due to lack of time it wasn't implemented into the final result.

- Design patterns

GRASP:

Creator: Is a GRASP-pattern that tells us which class is responsible for the creation of another class. In our project our “Field” class can be seen as a creator, because of our different classes such as IncomeTaxField Class and JailField Class. They all extend the field class. Thereby they would be non-existent without the Field-class as it is their superclass.

Information expert: Is another GRASP pattern that is included in our project, as. Information expert is used to decide which method should be assigned and to where.

Our MatadorController class could also be seen as an information expert as it holds a lot of information from other classes which also means that it holds a higher responsibility for the fact that the program runs.

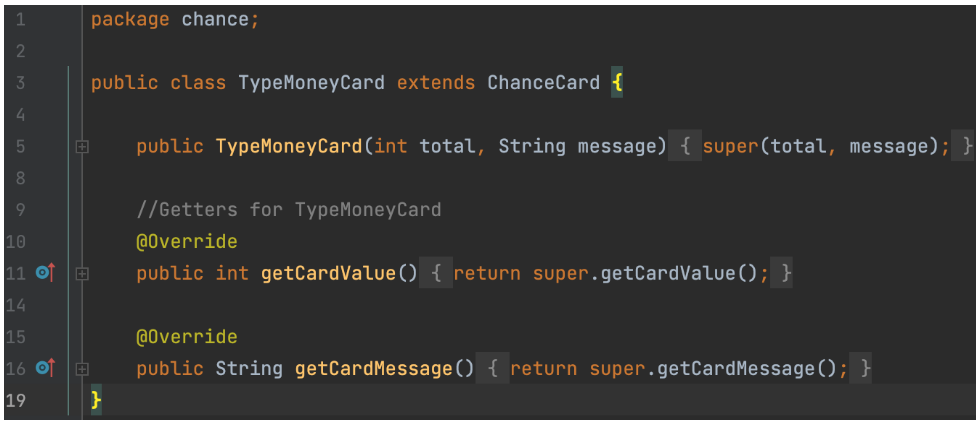
Polymorphism:

Polymorphism is the ability for an entity to take up several different forms. To elaborate, this means that objects from different classes, with the same attributes, can be implemented differently, in different instances. There are 2 main types of polymorphism, static, and dynamic.

Polymorphism that is resolved during compiling a project is called Static polymorphism. Method overloading, is an example of polymorphism being resolved during compiling.

Dynamic polymorphism on the other hand, is resolved at runtime. This is a process in which a call to an overridden method is resolved. Method overriding (@Overide), happens when the programmer declares a method in a subclass, which is already present in a parent class. This is done so that the subclass can give its own implementation of the given method.

In this example, we are looking at our TypeMoneyCard subclass. When designing our chance cards, we wanted to use and put emphasis on *inheritance.* Our primary aim was to create code that could be reused very easily. This meant creating the correct infrastructure to enable it to implement features on top of it in the future, if needed.



```java
package chance;

public class TypeMoneyCard extends ChanceCard {

    public TypeMoneyCard(int total, String message) { super(total, message); }

    //Getters for TypeMoneyCard
    @Override
    public int getCardValue() { return super.getCardValue(); }

    @Override
    public String getCardMessage() { return super.getCardMessage(); }
}
```

Figure. 8 Usage of subclass and @Overide

We created a parent class (ChanceCard), which has the subclass (TypeMoneyCard) which then further has the methods (getCardValue & getCardMessage). These methods are also subject (present) in our parent class. When creating the methods in subclass, this makes sure we override it. The @Override is simply used to help us with any human logical errors that might occur in the instance of using it.

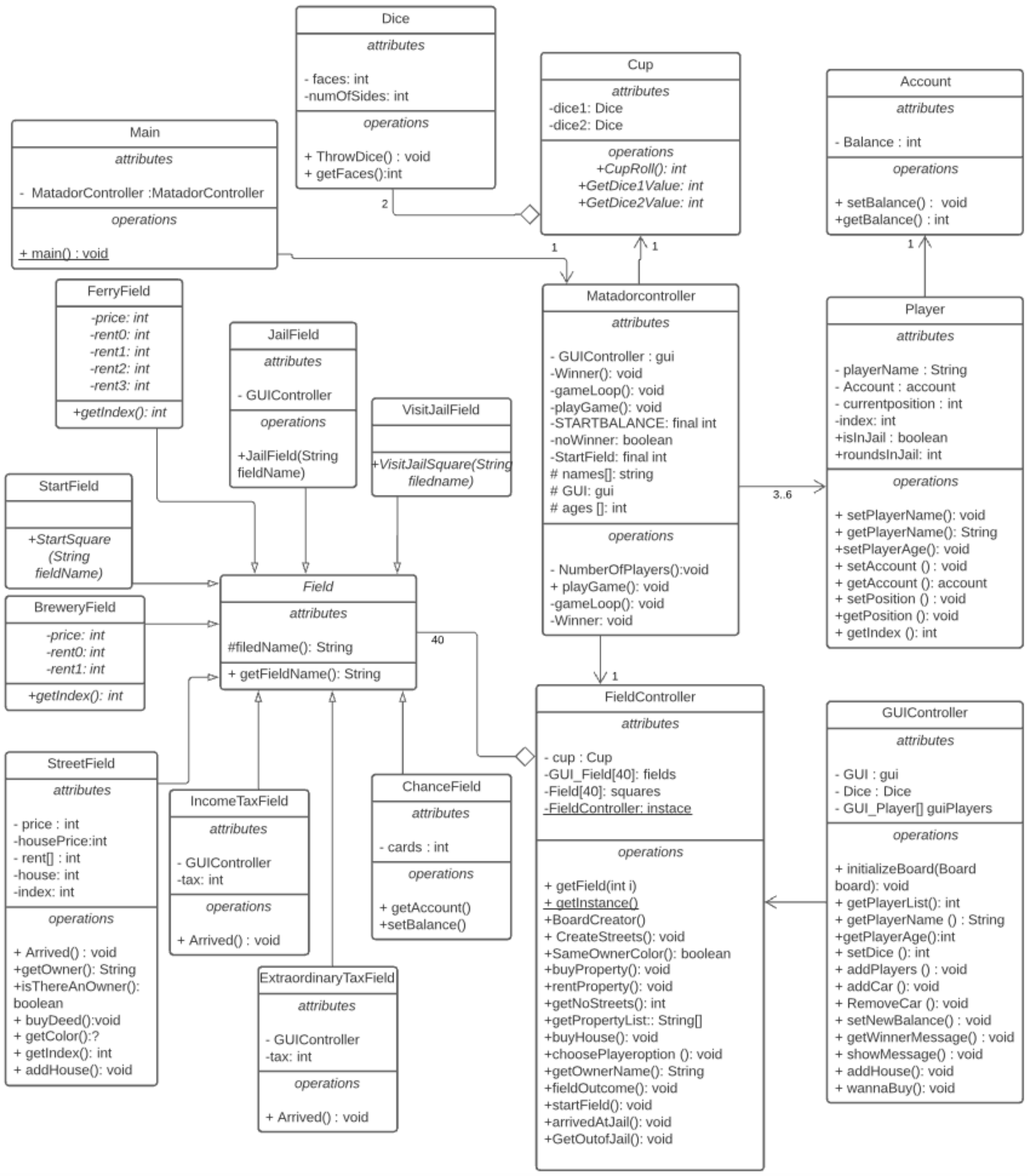
Polymorphism is also used in our fields. Here we have our different “landing fields” which extends our field class.

Low-coupling: Low coupling is a pattern that we would have wished to have implemented even more, low coupling is essential for making good reusable code. And it has been a bit of a hassle since big parts of our code are from previous projects where we didn’t code with that in mind. We have decided to make controller classes that take a lot of the methods out of our other classes and thereby the coupling of each class becomes lower since they are handled by the controllers. So you see the low coupling in the classes without any methods in them. But it is because of the controllers that the coupling can be lowered in each class.

Design Class Diagram

Our Design class diagram (based on the actual implementation) is built up around our MatadorController, which is associated with ‘FieldController’, ‘ChanceController’, ‘Cup’, ‘main’, and ‘Player’. It is shown that the ‘Player’ class is 3-6 players, which is associated with the account-class, which has an account to each player. The

‘FieldController' class consists of 40 fields, as well as all of the field-descriptions, which is associated with the ‘Field’ class. The Field class works as a parent class, with 9 children. We have 9 classes who inherit from the Field-class, these nine fields can be seen on the diagram. They have been created, because each special field has some specific qualities. We have made an aggregation between ‘Dice’ and ‘Cup’, because it refers to the classes to be built as a collection. It means that ‘Dice’ is a part of ‘Cup’. The same applies from ‘Field’ to ‘Board’, because the ‘Field’ class is a part of the ‘Board’ class.



24

Figure. 9 Design Class Diagram

Design Class Diagram over the different chance cards:

In our design class diagram, we have a ChanceCard-class, which is the parent class of 6 child-classes. ChanceController consists of all the different chance cards… The ChanceCard parent class is abstract, because it shouldnt be initiated, but forms all the child classes below. The Field Controller and the MatadorController are initiating the Chance-Controller, but we have separated them to make a smaller diagram, to make it more visible.

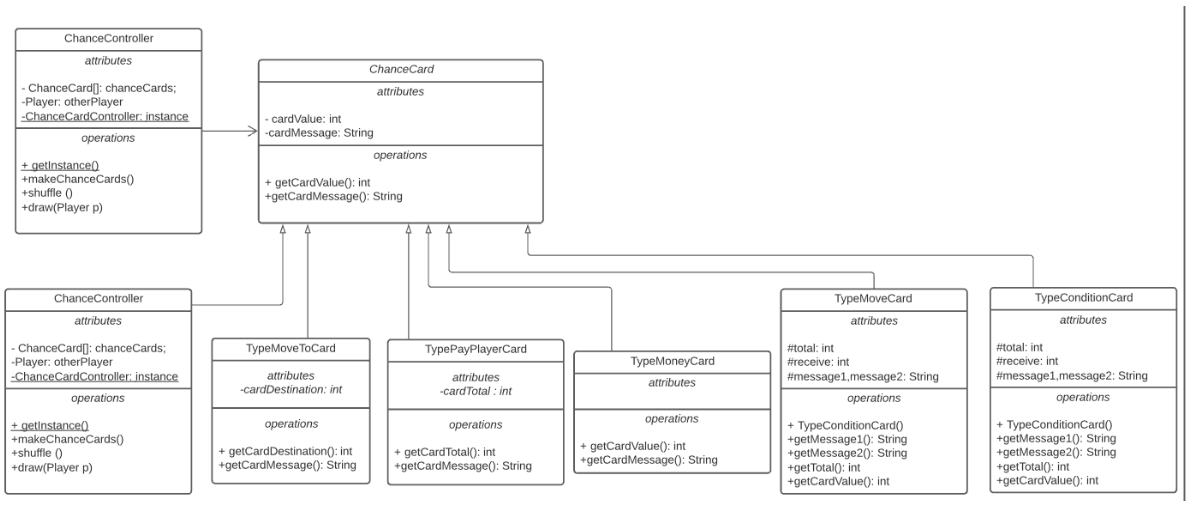


Figure. 10 Design Class Diagram over different chance cards

Implementation:

- Code

We have taken as our starting point the MVC (Model View Controller) model in relation to the code structure. The MVC model is a software design pattern a way of organizing code, which is used to divide a program logic into three elements
Model: This code holds our data and defines the essential components of our application.
View: View contains all the functions that directly interact with the user (which can be put into perspective with our GUI-class).
Controller: The controller code receives user input and decides what to do with it, handles functions and actions based on inputs and therefore can be seen as the logic/brains of the application and which ties together the Model and the View.

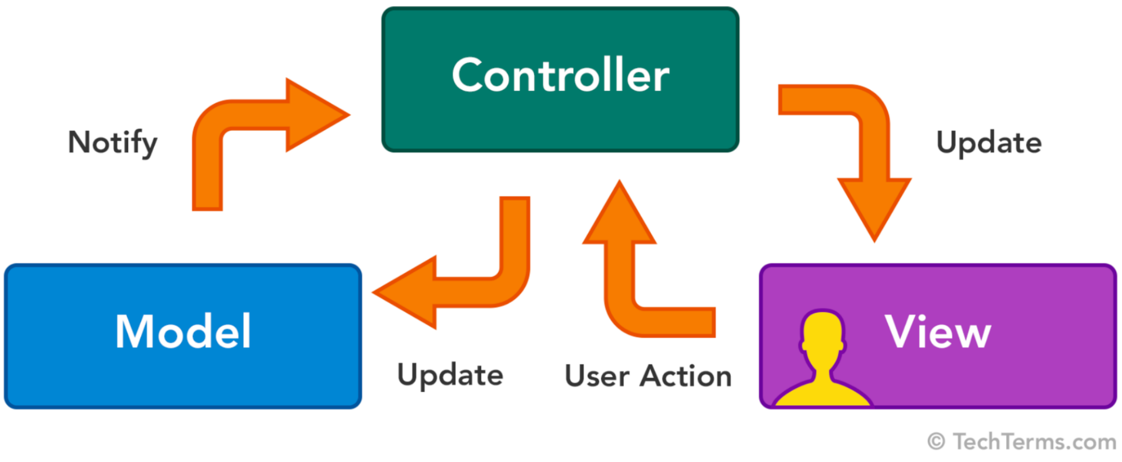
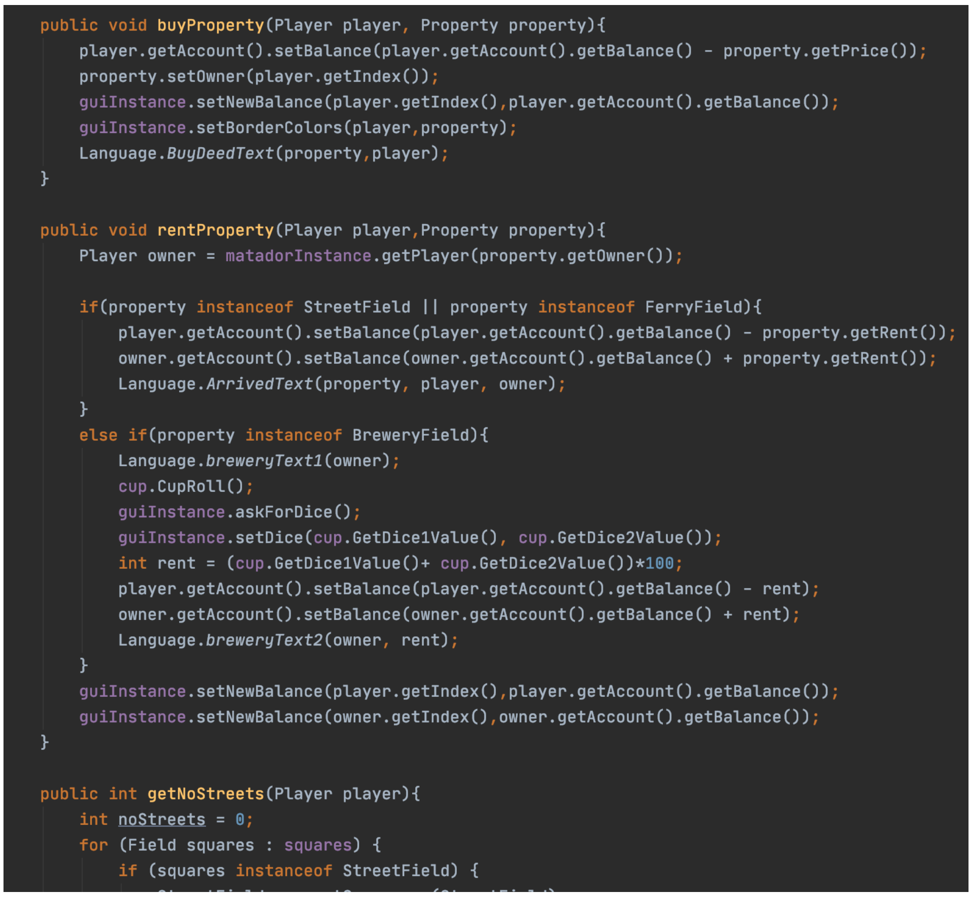


Figure. 11 : MVC (Model View Controller)

6 packages are used in our project.

Controller-package
The controller-package is a collection of all controllers and a “main” class which is our game launcher. The controller package is used to control the logic behind the initiation of our objects. The methods of the various objects regarding the game itself are defined, and their execution is performed. The controller package consists of the 3 controllers: MatadorController, FieldController, ChanceController.

The MatadorController is our main controller, and contains all game logic. Whereas the FieldController and ChanceController only controls certain objects initiation. FieldController controls methods used in the different fields can be seen below, methods such as buyProperty and rentProperty which are used a lot as part of the controller.



```java
public void buyProperty(Player player, Property property){
    player.getAccount().setBalance(player.getAccount().getBalance() - property.getPrice());
    property.setOwner(player.getIndex());
    guiInstance.setNewBalance(player.getIndex(),player.getAccount().getBalance());
    guiInstance.setBorderColors(player,property);
    Language.BuyDeedText(property,player);
}

public void rentProperty(Player player,Property property){
    Player owner = matadorInstance.getPlayer(property.getOwner());

    if(property instanceof StreetField || property instanceof FerryField){
        player.getAccount().setBalance(player.getAccount().getBalance() - property.getRent());
        owner.getAccount().setBalance(owner.getAccount().getBalance() + property.getRent());
        Language.ArrivedText(property, player, owner);
    }
    else if(property instanceof BreweryField){
        Language.breweryText1(owner);
        cup.CupRoll();
        guiInstance.askForDice();
        guiInstance.setDice(cup.GetDice1Value(), cup.GetDice2Value());
        int rent = (cup.GetDice1Value()+ cup.GetDice2Value())*100;
        player.getAccount().setBalance(player.getAccount().getBalance() - rent);
        owner.getAccount().setBalance(owner.getAccount().getBalance() + rent);
        Language.breweryText2(owner, rent);
    }
    guiInstance.setNewBalance(player.getIndex(),player.getAccount().getBalance());
    guiInstance.setNewBalance(owner.getIndex(),owner.getAccount().getBalance());
}

public int getNoStreets(Player player){
    int noStreets = 0;
    for (Field squares : squares) {
        if (squares instanceof StreetField) {
```

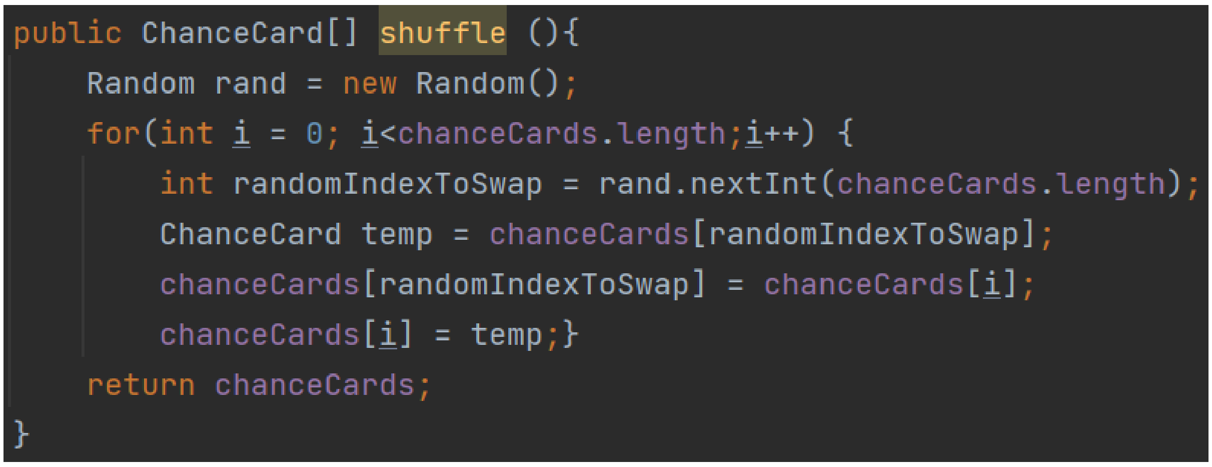
Figure. 12: FieldController

The property type fields the FieldController contains every field's outcome when landed on. An example of this is the ‘arrivedAtJail’ and ‘GetOutOfJail’ methods that deal with the JailField itself. ArrivedAtJail is the method that places the player back to ‘VisitJailField’ and ‘GetOutOfJail’ which holds a switch case of the following ways to get out of jail. Before the switch case it has a loop, so the player doesn't stay in jail for more than 3 rounds.

Besides having responsibility for the fields the Fieldcontroller also contains the game logic revolving houses.

As the name implies, FieldController controls all types of fields involved in the game while ChanceController controls all chance-cards.

The ChanceCardController creates all chance cards in an array and has methods to draw a card, find which instance the drawn card is, and then perform the cards logik. The drawn card is then placed at the last index of the chance card array to simulate putting the card at the bottom of the deck. It also has a method to “shuffle” all cards in the array. As seen beneath, it finds a random index in the chance card array using java’s Random class, and then swaps it with the card in index i.



```
public ChanceCard[] shuffle (){
    Random rand = new Random();
    for(int i = 0; i<chanceCards.length;i++) {
        int randomIndexToSwap = rand.nextInt(chanceCards.length);
        ChanceCard temp = chanceCards[randomIndexToSwap];
        chanceCards[randomIndexToSwap] = chanceCards[i];
        chanceCards[i] = temp;}
    return chanceCards;
}
```

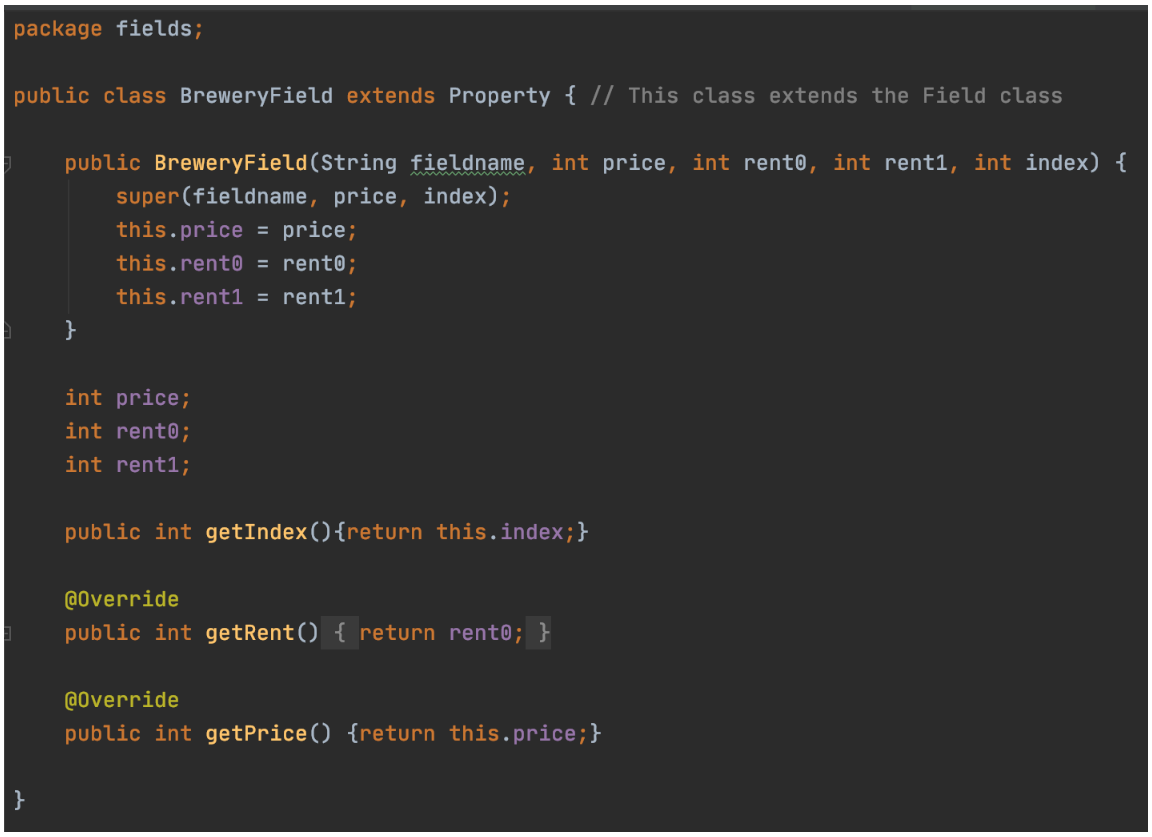
Figure. 13: ChanceCardController

The shuffle method is implemented in the playGame() method in the MatadorController class, right before the game loop to make sure it is shuffled a single time and keeps it’s order during the game loop.

Fields-package

The field-package contains every fieldtype. We have implemented the use of inheritance in this package, this allows us to create new classes that are built upon existing classes. The Field class is abstract and forms the basis of the ten types of fields that the board contains. A field consists of a field name (String). The idea of Inheritance is to create new classes (child class or derived or subclass) from an existing class (parent class or superclass). When you inherit from an existing class, you can reuse methods and fields of the parent class. You can even add new methods and fields in your current class also. This way, we implement the concept of reusability, creating better connections between different classes, and achieve method overriding.

The usage of inheritance regarding extending, the super-method, Override can be seen below. The BreweryField inherits “fieldname, price and index”, and adds a rent method.

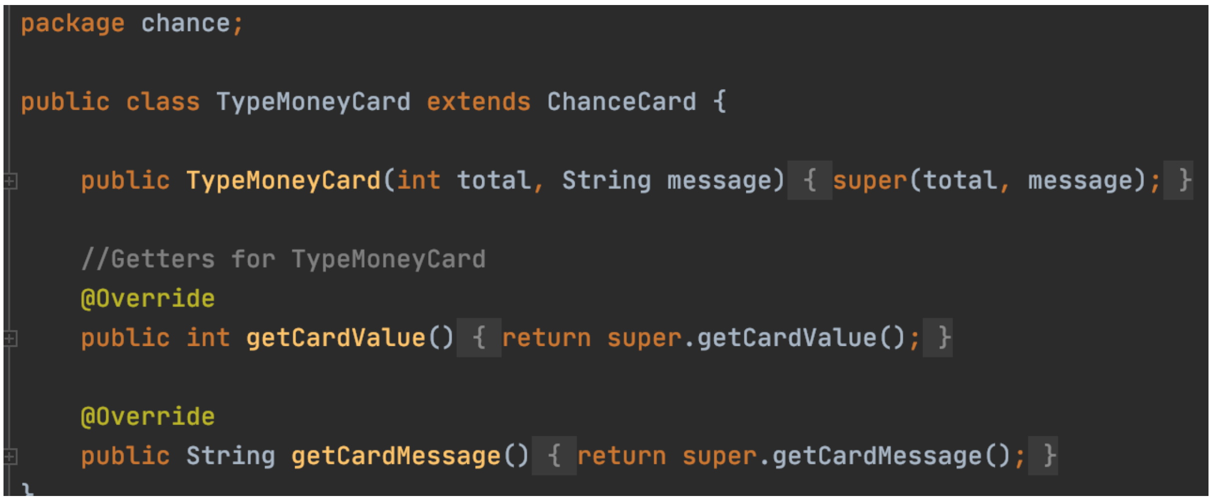


```java
package fields;

public class BreweryField extends Property { // This class extends the Field class

    public BreweryField(String fieldname, int price, int rent0, int rent1, int index) {
        super(fieldname, price, index);
        this.price = price;
        this.rent0 = rent0;
        this.rent1 = rent1;
    }

    int price;
    int rent0;
    int rent1;

    public int getIndex(){return this.index;}

    @Override
    public int getRent() { return rent0; }

    @Override
    public int getPrice() {return this.price;}

}
```

Figure 14. BreweryField

Chance-package

Our chance package includes all types of chance cards which can be chosen throughout the game, and we have also implemented the use of inheritance in this package as well. As we are creating different types of chance-cards, it made sense to use inheritance in this part also. As the child class acquires all the properties and behaviors of the parent class, and as our chance cards share mostly the same methods, the usage of inheritance fits well.



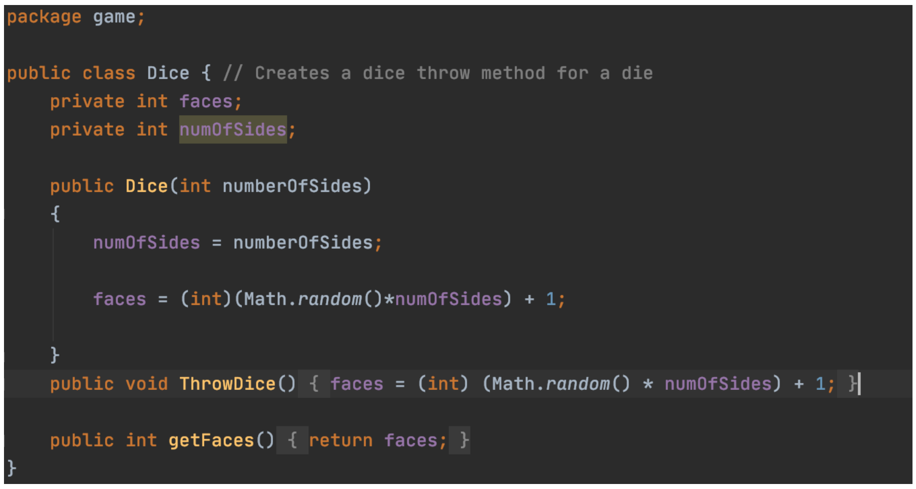
```java
package chance;

public class TypeMoneyCard extends ChanceCard {

    public TypeMoneyCard(int total, String message) { super(total, message); }

    //Getters for TypeMoneyCard
    @Override
    public int getCardValue() { return super.getCardValue(); }

    @Override
    public String getCardMessage() { return super.getCardMessage(); }
```

Figure. 15: TypeMoneyCard

Game-package

Our game package contains four classes that are essential for the game to work, it's basic game features and therefore we created a whole package for them. The four classes are Player, Account, Dice and Cup.

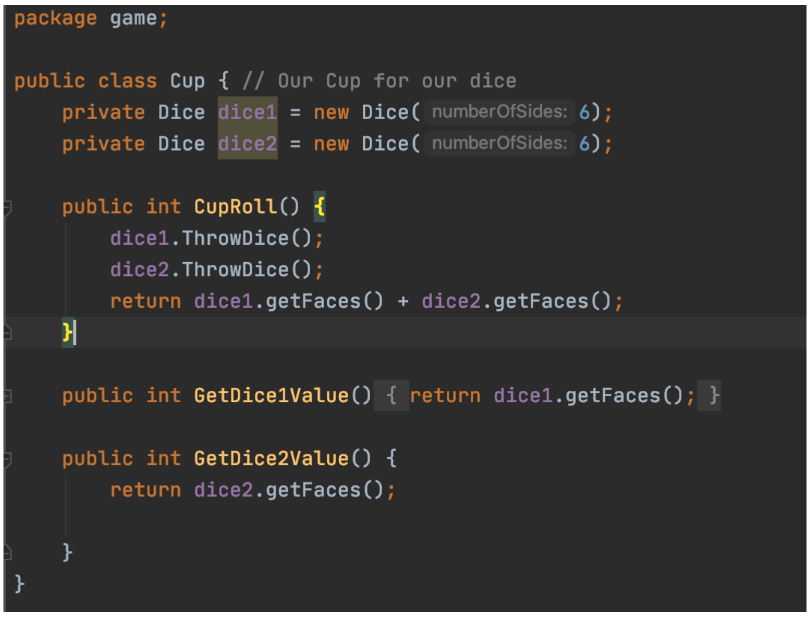
A Dice class is defined which generates a random dice roll in the Dice class. Below, use is made of Math.random () which shows the mathematics behind the randomly generated stroke.



```java
package game;

public class Dice { // Creates a dice throw method for a die
    private int faces;
    private int numOfSides;

    public Dice(int numberOfSides)
    {
        numOfSides = numberOfSides;

        faces = (int)(Math.random()*numOfSides) + 1;

    }
    public void ThrowDice() { faces = (int) (Math.random() * numOfSides) + 1; }

    public int getFaces() { return faces; }
}
```

Figure 16. Dice Class

The Dice method is then transferred to our Cup class, which puts our two dice in one method. The relationship can thus also be combined into GRASP patterns regarding Creator.

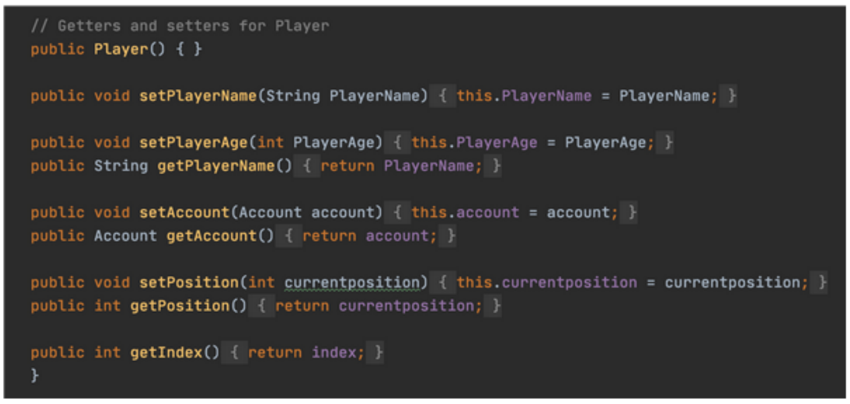


```java
package game;

public class Cup { // Our Cup for our dice
    private Dice dice1 = new Dice( numberOfSides: 6);
    private Dice dice2 = new Dice( numberOfSides: 6);

    public int CupRoll() {
        dice1.ThrowDice();
        dice2.ThrowDice();
        return dice1.getFaces() + dice2.getFaces();
    }

    public int GetDice1Value() { return dice1.getFaces(); }

    public int GetDice2Value() {
        return dice2.getFaces();

    }
}
```

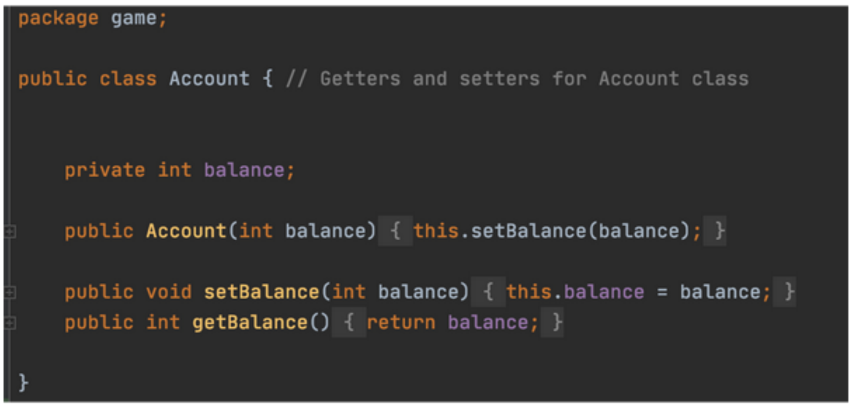
Figure. 17 Cup Class

In our Account and Player class, we are using a lot of Getters and Setters. They play an important role in retrieving and updating the value of a variable outside the specific class.

The getter and setter method gives you control on how a particular field is initialized. This also makes debugging easier.



```java
// Getters and setters for Player
public Player() { }

public void setPlayerName(String PlayerName) { this.PlayerName = PlayerName; }

public void setPlayerAge(int PlayerAge) { this.PlayerAge = PlayerAge; }
public String getPlayerName() { return PlayerName; }

public void setAccount(Account account) { this.account = account; }
public Account getAccount() { return account; }

public void setPosition(int currentposition) { this.currentposition = currentposition; }
public int getPosition() { return currentposition; }

public int getIndex() { return index; }
}
```



```java
package game;

public class Account { // Getters and setters for Account class


    private int balance;

    public Account(int balance) { this.setBalance(balance); }

    public void setBalance(int balance) { this.balance = balance; }
    public int getBalance() { return balance; }

}
```

Figure 18. Player and Account class.

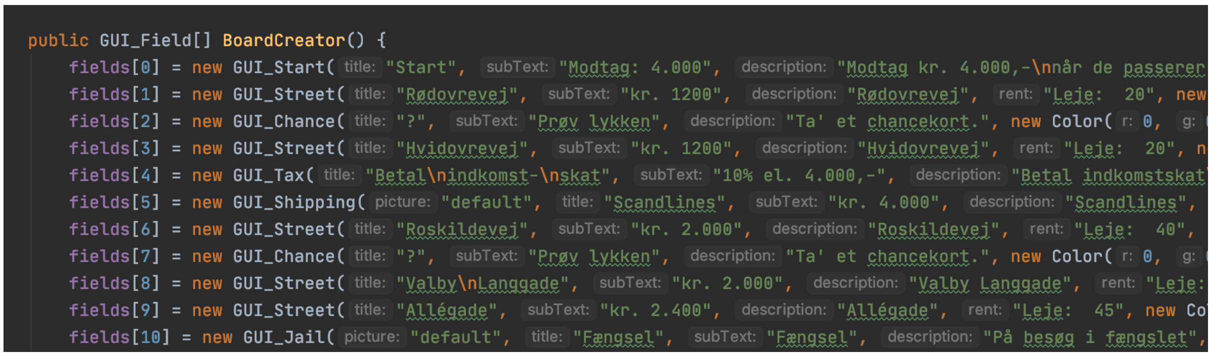
GUI-package
Our project contains one more controller, the GUI-controller which as previously written is a part of the “view” element in the MVC-model, and that’s why we excluded it from the controller-package as it is not a part of the game-logic but the user-interface. The GUI-controller controls every element that can be interacted with, by the user. It’s the logic behind our board display and the viewable-functions happening upon them. We have been using the ’setdice’ method, which makes two dice visible on the board. This controller is very important for the development of our Matador-game, because it shows a visual construction of the game for the viewers. We would have liked to have much more low-coupling in the GUI-class, as the code is quite complex, and intertwined relationships are difficult to understand.

Language-package
We make use of a language-class that contains every “showMessages”, every outprints which can be seen on the GUI. The purpose of this package was to make it easy to translate every message to different languages, though we have only implemented english.

- GUI-Implementation

A GUI (Graphic User Interface) has been implemented as part of this project. The GUI displays a game board resembling the Matador game. We have implemented the matadorgui-3.1.6.jar file which constructs the layout of our board and contains the underlying structure to be used.
As mentioned previously, the GUIController controls everything visually, and can thus be considered to be one of our most important classes. A class that constructs our dice, players, visualizes general methods from other classes such as getPlayerList, getPlayerName, setBalance, etc.

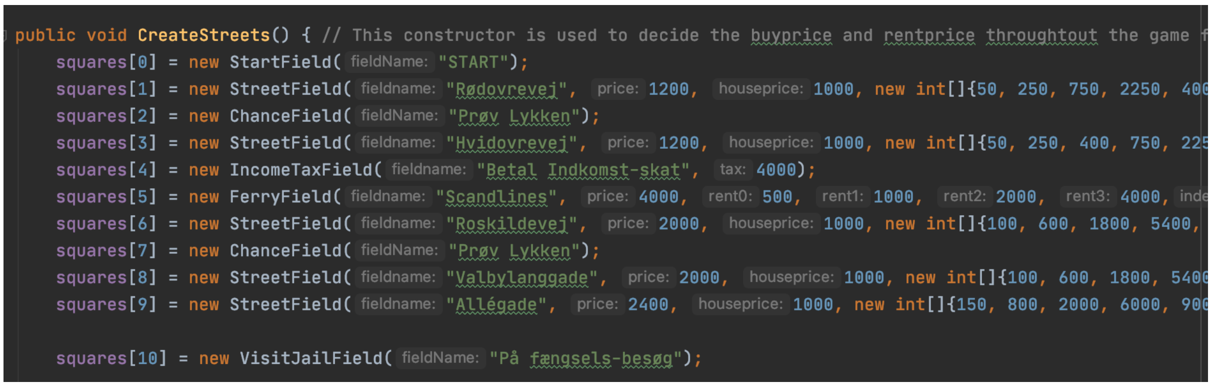
We use arrays to display our Board, the first array is used to implement the GUI-fields. We use the GUI-commands which have been given to us, this way we can get the colors and images on the GUI.



```java
public GUI_Field[] BoardCreator() {
    fields[0] = new GUI_Start( title: "Start",  subText: "Modtag: 4.000",  description: "Modtag kr. 4.000,-\nnår de passerer
    fields[1] = new GUI_Street( title: "Rødovrevej",  subText: "kr. 1200",  description: "Rødovrevej",  rent: "Leje:  20", new
    fields[2] = new GUI_Chance( title: "?",  subText: "Prøv lykken",  description: "Ta' et chancekort.", new Color( r: 0,  g:
    fields[3] = new GUI_Street( title: "Hvidovrevej",  subText: "kr. 1200",  description: "Hvidovrevej",  rent: "Leje:  20", n
    fields[4] = new GUI_Tax( title: "Betal\nindkomst-\nskat",  subText: "10% el. 4.000,-",  description: "Betal indkomstskat
    fields[5] = new GUI_Shipping( picture: "default",  title: "Scandlines",  subText: "kr. 4.000",  description: "Scandlines",
    fields[6] = new GUI_Street( title: "Roskildevej",  subText: "kr. 2.000",  description: "Roskildevej",  rent: "Leje:  40",
    fields[7] = new GUI_Chance( title: "?",  subText: "Prøv lykken",  description: "Ta' et chancekort.", new Color( r: 0,  g:
    fields[8] = new GUI_Street( title: "Valby\nLanggade",  subText: "kr. 2.000",  description: "Valby Langgade",  rent: "Leje:
    fields[9] = new GUI_Street( title: "Allégade",  subText: "kr. 2.400",  description: "Allégade",  rent: "Leje:  45", new Co
    fields[10] = new GUI_Jail( picture: "default",  title: "Fængsel",  subText: "Fængsel",  description: "På besøg i fængslet",
```

Figure. 19: BoardCreator

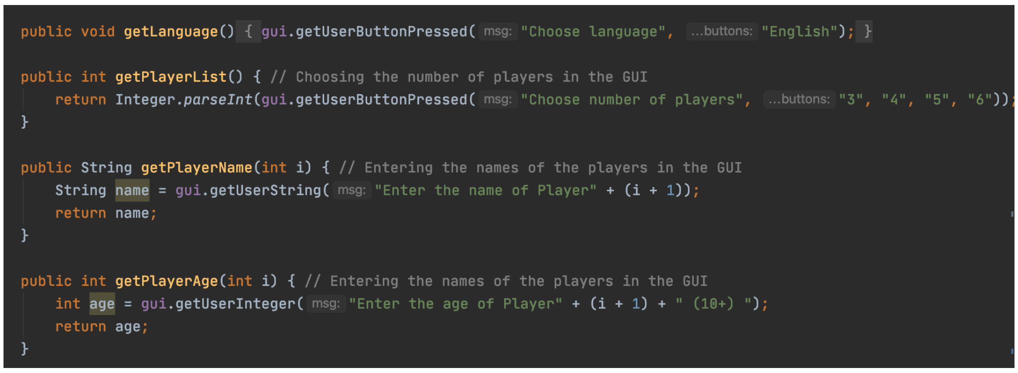
The array beneath is used to implement the methods regarding the specific fields. Methods containing the field prices, rent prices ect. to the GUI-fields. As the two arrays share the same index, the methods are placed exactly to the wanted field.



```java
public void CreateStreets() { // This constructor is used to decide the buyprice and rentprice throughtout the game f
    squares[0] = new StartField( fieldName: "START");
    squares[1] = new StreetField( fieldname: "Rødovrevej",  price: 1200,  houseprice: 1000, new int[]{50, 250, 750, 2250, 400
    squares[2] = new ChanceField( fieldName: "Prøv Lykken");
    squares[3] = new StreetField( fieldname: "Hvidovrevej",  price: 1200,  houseprice: 1000, new int[]{50, 250, 400, 750, 225
    squares[4] = new IncomeTaxField( fieldname: "Betal Indkomst-skat",  tax: 4000);
    squares[5] = new FerryField( fieldname: "Scandlines",  price: 4000,  rent0: 500,  rent1: 1000,  rent2: 2000,  rent3: 4000,  inde
    squares[6] = new StreetField( fieldname: "Roskildevej",  price: 2000,  houseprice: 1000, new int[]{100, 600, 1800, 5400,
    squares[7] = new ChanceField( fieldName: "Prøv Lykken");
    squares[8] = new StreetField( fieldname: "Valbylanggade",  price: 2000,  houseprice: 1000, new int[]{100, 600, 1800, 5400
    squares[9] = new StreetField( fieldname: "Allégade",  price: 2400,  houseprice: 1000, new int[]{150, 800, 2000, 6000, 900

    squares[10] = new VisitJailField( fieldName: "På fængsels-besøg");
```

Figure. 20: CreateStreets

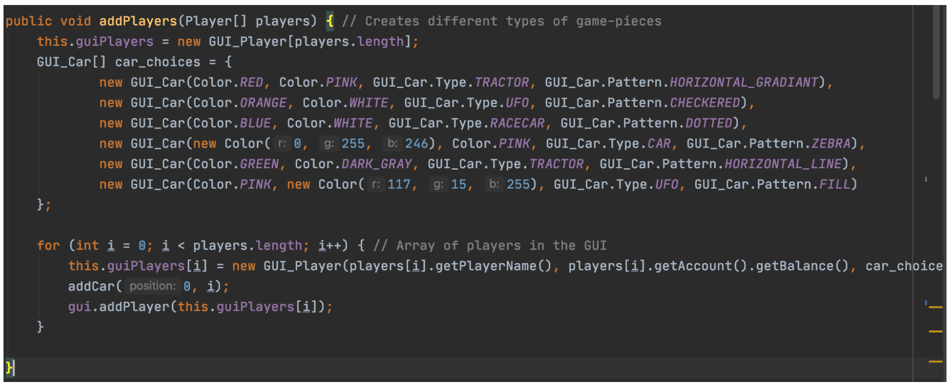
A GUI is a form of user interface that allows users to interact with the system. Now that Matador is a highly interactive program, the interactivity elements should also be done graphically through our GUI. In this case we have implemented different methods in the intro of our game, where the user can interact with the system. By the use of “getUserButtonPressed”, “getUserString” and “getUserInteger” we are able to display buttons and writable sections. We wanted our game to be more than just a simulation, that's we have several buttons and interactive elements and not just a single button that continues the game.



```java
public void getLanguage() { gui.getUserButtonPressed( msg: "Choose language",  ...buttons: "English"); }

public int getPlayerList() { // Choosing the number of players in the GUI
    return Integer.parseInt(gui.getUserButtonPressed( msg: "Choose number of players",  ...buttons: "3", "4", "5", "6"));
}

public String getPlayerName(int i) { // Entering the names of the players in the GUI
    String name = gui.getUserString( msg: "Enter the name of Player" + (i + 1));
    return name;
}

public int getPlayerAge(int i) { // Entering the names of the players in the GUI
    int age = gui.getUserInteger( msg: "Enter the age of Player" + (i + 1) + " (10+) ");
    return age;
}
```

Figure 21. GUI-buttons

The game consists of 3-6 players, and that's why we used an array to display the game pieces in correlation with the amount of players in the game.



```java
public void addPlayers(Player[] players) { // Creates different types of game-pieces
    this.guiPlayers = new GUI_Player[players.length];
    GUI_Car[] car_choices = {
            new GUI_Car(Color.RED, Color.PINK, GUI_Car.Type.TRACTOR, GUI_Car.Pattern.HORIZONTAL_GRADIANT),
            new GUI_Car(Color.ORANGE, Color.WHITE, GUI_Car.Type.UFO, GUI_Car.Pattern.CHECKERED),
            new GUI_Car(Color.BLUE, Color.WHITE, GUI_Car.Type.RACECAR, GUI_Car.Pattern.DOTTED),
            new GUI_Car(new Color( r: 0,  g: 255,  b: 246), Color.PINK, GUI_Car.Type.CAR, GUI_Car.Pattern.ZEBRA),
            new GUI_Car(Color.GREEN, Color.DARK_GRAY, GUI_Car.Type.TRACTOR, GUI_Car.Pattern.HORIZONTAL_LINE),
            new GUI_Car(Color.PINK, new Color( r: 117,  g: 15,  b: 255), GUI_Car.Type.UFO, GUI_Car.Pattern.FILL)
    };

    for (int i = 0; i < players.length; i++) { // Array of players in the GUI
        this.guiPlayers[i] = new GUI_Player(players[i].getPlayerName(), players[i].getAccount().getBalance(), car_choice
        addCar( position: 0, i);
        gui.addPlayer(this.guiPlayers[i]);
    }

}
```

Figure 22. GUI addPlayers method in GUIController.

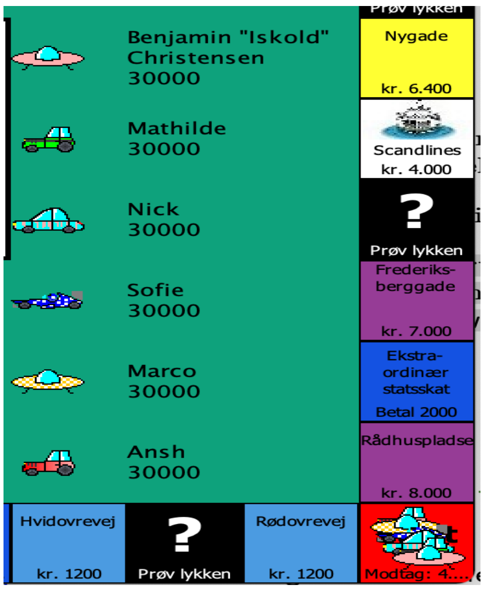
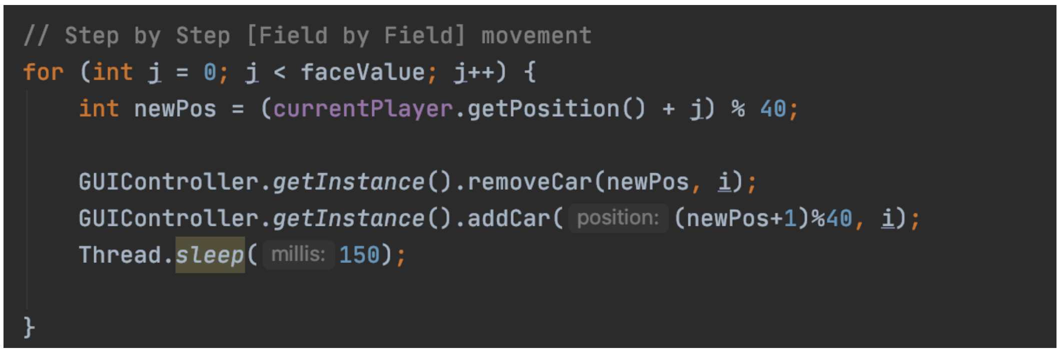


Figure. 23 Cars and account

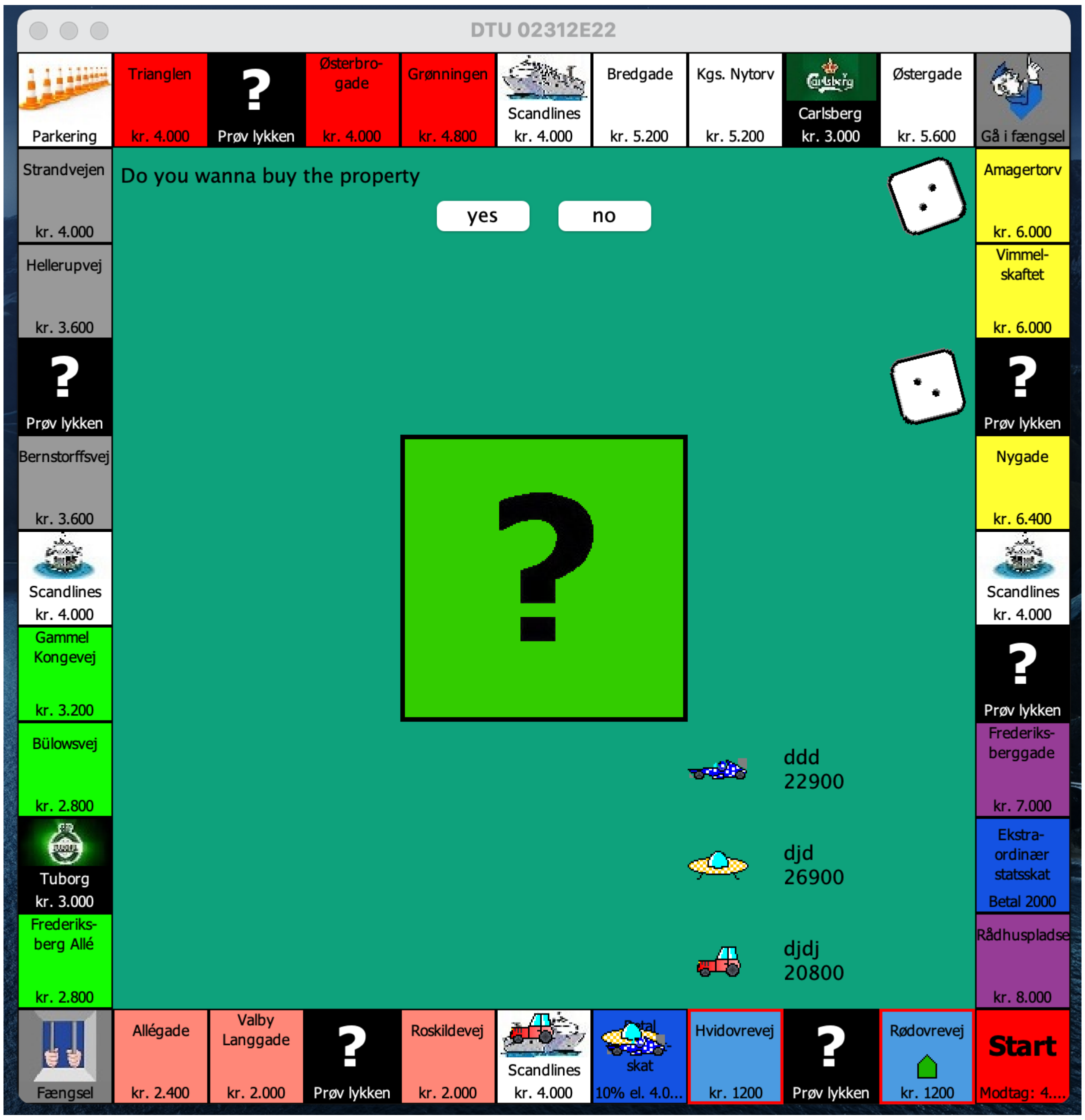
We also implemented some cool features, which make the game more dynamic and real. The pieces move step by step (field to field), this feature really adds to the game. The use of for-loops and modulus went into this display.



```
// Step by Step [Field by Field] movement
for (int j = 0; j < faceValue; j++) {
    int newPos = (currentPlayer.getPosition() + j) % 40;

    GUIController.getInstance().removeCar(newPos, i);
    GUIController.getInstance().addCar( position: (newPos+1)%40, i);
    Thread.sleep( millis: 150);

}
```

Figure 24. Step by step movement

The picture below displays our Matador-game with every field, players, dice, in-game interactions, house ect.



DTU 02312E22

Do you wanna buy the property

yes   no

ddd
22900

djd
26900

djdj
20800

Figure 25: Displaying our GUI

Additional information:

- Guide for Github repository

Our link for our project in github: https://github.com/Anshjyot/11_Matador.git

We are going to show you two methods on how to import the project from github.

Method 1) URL from Github

1.a) When opening github you should find the right repository with the name 11_Matador, then you click on the green button “Code” og copies the shown URL.
1.b) Then you open up “IntelliJ IDEA”, and click on “Get from VCS (Version Control)” and insert the copied link from 1.a). Here you need to double check that a file with the same name doesn't already exist. Then you click “Clone”. Then the project should be implemented correctly.

Method 2) Connecting Github to IntelliJ

2.a) If we press “Get from VCS (Version Control)”, and next “Github”, you get the chance to login to your Github profile.
2.b) When clicking “Login”, you get assigned to a homepage that wants you to “Authorize in Github”. After you have correctly signed in to your profile, you get a message “You can close the page”.
2.c) When you return to IntelliJ you can see the following projects that have been shared with your account. Here we choose “11_Matador”, and the project opens as required.

- Configuration Control

Maven

We have used the GUI that was already assigned to us, hence we use Maven to drive the GUI as a dependency. We also have a dependency to our Junit, that we use to test our codes through. Through our JUnit we will run with code-coverage as described above. With the use of Maven, all developers have the chance to download a project, where it isn't needed to add a special version of some external-files, as in our case the Matador-GUI-file or the Junit test, because it is included through Maven. We ran into this problem in CDIO 2, but this time we have constructed a Maven-project, where our used files were added through Maven. The project can easily be accessed through Github, with "Download Zip" by following the Github Guide in the beginning of the report. When the project is running through github, it is very important that we know how to use the functions: merge, commit and push. These 3 functions are responsible for former versions of our project that have been saved in git. This also includes merge-conflicts, where developers have edited in the same class and then commited. Thats why, responsibility-areas below the different classes are important so you don't end up in these types of problems.

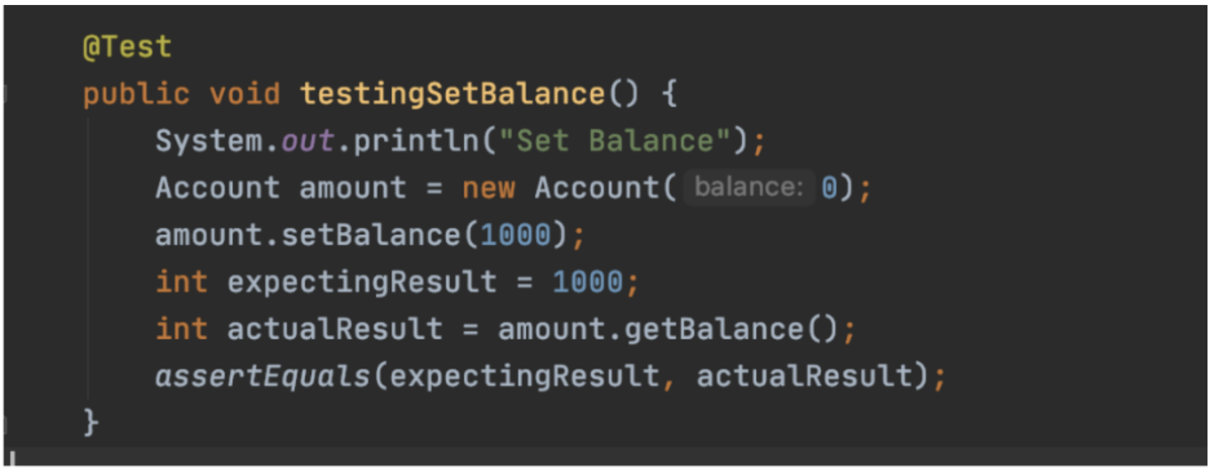
- Test

Account test:

In our 3 tests, JUnit is used with imports regarding assertEquals which are used to compare our “expected” and “actual” results, and the “fail” function.

Test T1: Three different Junit tests have been performed, Account Test, Cup Test and the other is a PlayerPosition Test.

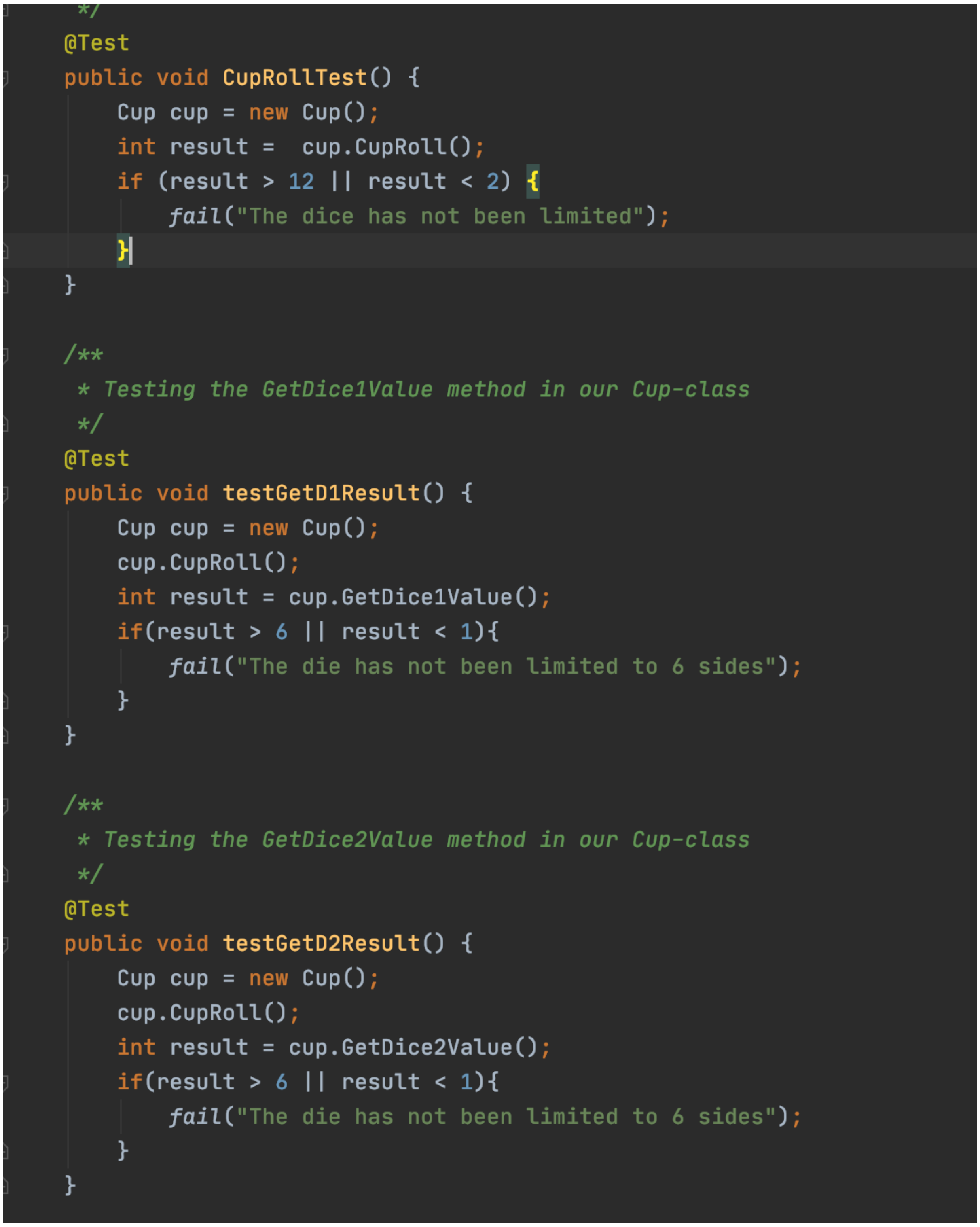
Account-Test specifically tests our setBalance method - where an amount (a sum of money) is added to one's account - where of course our setBalance method is used. As well as a method that tests that one's amount does not go below 0. It ends up that my expectingResult and actualResult is the same and thus the test is correct.



```java
@Test
public void testingSetBalance() {
    System.out.println("Set Balance");
    Account amount = new Account( balance: 0);
    amount.setBalance(1000);
    int expectingResult = 1000;
    int actualResult = amount.getBalance();
    assertEquals(expectingResult, actualResult);
}
```

Figure. 26: Account test

Cup test:
Test T2:
We also used a Cup-test, which shows if the die has been limited or not - both singular and inside the cup.

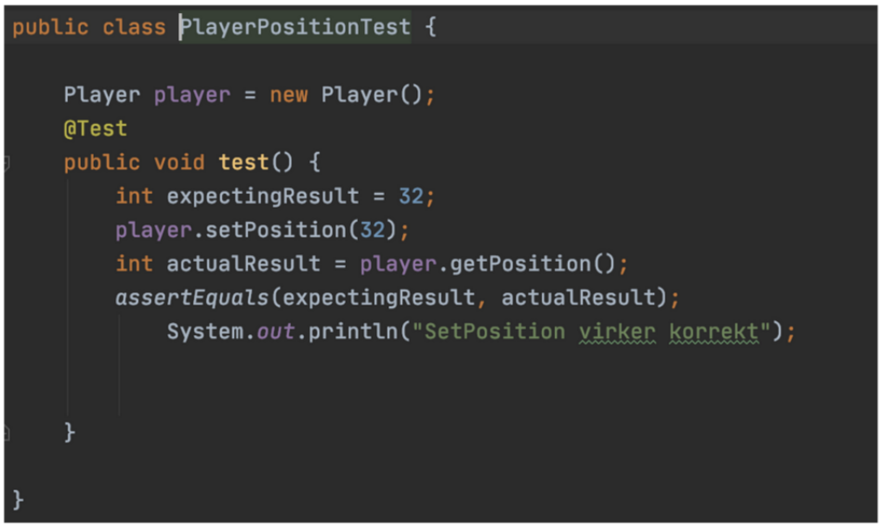


```
     */
    @Test
    public void CupRollTest() {
        Cup cup = new Cup();
        int result =  cup.CupRoll();
        if (result > 12 || result < 2) {
            fail("The dice has not been limited");
        }
    }

    /**
     * Testing the GetDice1Value method in our Cup-class
     */
    @Test
    public void testGetD1Result() {
        Cup cup = new Cup();
        cup.CupRoll();
        int result = cup.GetDice1Value();
        if(result > 6 || result < 1){
            fail("The die has not been limited to 6 sides");
        }
    }

    /**
     * Testing the GetDice2Value method in our Cup-class
     */
    @Test
    public void testGetD2Result() {
        Cup cup = new Cup();
        cup.CupRoll();
        int result = cup.GetDice2Value();
        if(result > 6 || result < 1){
            fail("The die has not been limited to 6 sides");
        }
    }
```

Figure 27. Cup test code.

Player position Test:

Test T3: In our PlayerPosition Test, we use our setPosition and getPosition method from our program. A method that positions the player at a specific field. It can be seen below



```java
public class PlayerPositionTest {

    Player player = new Player();
    @Test
    public void test() {
        int expectingResult = 32;
        player.setPosition(32);
        int actualResult = player.getPosition();
        assertEquals(expectingResult, actualResult);
            System.out.println("SetPosition virker korrekt");


    }

}
```

Figure. 28 PlayerPositionTest

Databar test T4: Furthermore we have tested our program on the windows computers on DTU as it is a requirement that the program runs on them. As seen on the picture the program runs without any failures.

We have imported and tested our program on the window computers on DTU. It has been a requirement that in the earlier in the earlier projects, (CDIO1 → CDIO3)

User Test

Throughout our usertest, we learned that the customer wanted more visible buttons to interact with throughout the game.

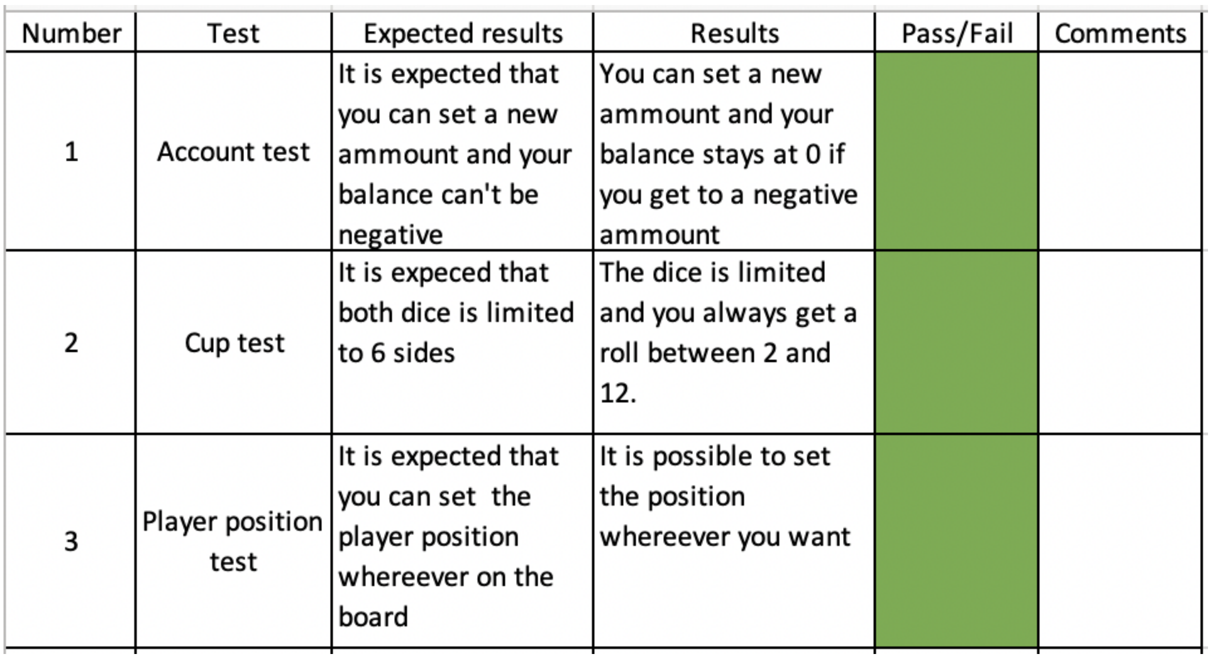
The user thought it was a nice feature that the cars were moving step-by-step, because it made it easier to follow and the game became more realistic with the dynamic effect.

Traceability matrix:

Test case	Functional Requirements									Non-Functional Requirements	
	M02	M06	S01	S02	S09	S10	S12	S13	C02	IF02	IF07
T1				x					x		

T2	x		x								x
T3		x			x	x	x	x			
T4										x	

Test case:



Number	Test	Expected results	Results	Pass/Fail	Comments
1	Account test	It is expected that you can set a new ammount and your balance can't be negative	You can set a new ammount and your balance stays at 0 if you get to a negative ammount	🟩	
2	Cup test	It is expeced that both dice is limited to 6 sides	The dice is limited and you always get a roll between 2 and 12.	🟩	
3	Player position test	It is expected that you can set the player position whereever on the board	It is possible to set the position whereever you want	🟩	

Figure. 29: Test Cases

Conclusion

Throughout the project's duration, we have lived up to the main project requirements, and fulfilled all our “Must haves” and most of our “Should haves”.

In general the group agrees that the best way to tackle this project again, was to restart and rebuild. By doing this, we would have ensured better infrastructure by creating lower coupling and higher cohesion, so that the implementation process of new features would have been easier. However despite our problems across the last couple of weeks, we have managed to create a runnable and fairly interactive program on top of our patchy infrastructure. Our only wish was to include the player-to-player trading system which would have added interactivity. We managed to include as many GRASP-Patterns, showcasing our java-abilities in a number of scenarios. Overall it was an enjoyable and instructive course where we learned the key to creating a good reusable code, Low Coupling and High Cohesion.