CDIO final

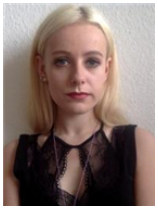
Emily Skovgaard Rasmussen, s153374
Mathias Tværmose Gleerup, s153120,
Sofie Freja Christensen s153932
Morten V. Christensen s147300
Simon Lundorf s154008
Jonas Larsen, s136335,
Group nr. 15 Class: 02312

Deadline : January 16th 2017

January 16, 2017

DTU

Danmarks Tekniske Universitet






Contents

1 Hour registration

CDIO Final	Emily	Jonas	Mathias	Morten	Simon	Sofie	Sum
Conceive	0	0	0	0	0	0	0
Design	13	13	11	13	9	11	70
Impl.	40	39	39,5	22	37,5	33	211
Test	4	6,5	2	5	2	6	25,5
Dok.	0	7	12,5	16	21	13	69,5
Andet.	8	0	0	0	0	0	8
Sum	65	65,5	65	56	69,5	63	**384**

1.1 Git location

https://github.com/Sofiefreja/CDIO—finals.git

2 Formalities

2.1 Abstract

A program has been compiled in java. The program is designed to be the Danish board game Matador. Most of the features are implemented, however it is compiled to be ready for further development, to include the remaining features. The board contains of 40 squares, each of these with a feature to effect the player and possibly balance. The objective of the game is to be the "last man standing". You lose and exit the game by going bankrupt (balance reaches below zero). Furthermore the possibility to switch between languages Danish and English is possible. This is all showcased by the use of a graphical user interface(GUI), to improve the players experience of the game and thereby increase usability.

2.2 Foreword

This report takes its foundation in the program (CDIO final), drawn from the following course Indledende programmering (02312). This stretches over 3 weeks, winter of 2017. Furthermore it is based on knowledge drawn from the courses: Udviklingsmetoder til IT-systemer (02313) og Versionsstyring og testmetoder (02315). All of which takes place at DTU campus Lyngby, fall of 2016. The project is made in collaboration between the members of group 15:

- Emily Skovgaard Rasmussen -SWT
- Jonas Larsen - SWT
- Mathias Tværmose Gleerup - SWT
- Morten Velin Christensen - SWT
- Simon Lundorf - SWT
- Sofie Freja Christensen - SWT

2.3 Git-strategy

In this project, the parallell configuration is expected to be handled as a combination of two types of patterns. These are the "Branch per change" and "Branch per environment" (These are seen in the appendix). This is used in a way, where there are three environments: The master, Develop and Release. The change environment however, is furthermore branched. This meaning, a new branch for every new feature being created and implemented. Documentation for executed is represented in the Appendix

2.4 Risk analysis

As part of the analysis process, we sat down and made a lists of the possible risks that we as a group could end up facing. To prepare us for the possible scenarios we made a prioritized list of what we felt was possible to implement if nothing went wrong, and how we would down prioritize the features in case something did. Our prioritized list of features was handed in as part of M1, how we planned to downgrade and tackle risks in case something went wrong can be seen here:

- Understaffed: downgrading of the red and yellow points, and longer days for the remaining in the group.
- Not enough time during M2: A solution could be downgrading the chancecards, and make them parking spaces.
- Underestimating the emphazis of the project: A solution could downgrading the chancecards and pawning.
- Complications with the GUI: This is not downgraded, therefore chancecards and pawning is downgraded.
- The merging between the patterns (branch per release and branch per environment).

3 Process

We started this project by doing a "matching of expectations", to define the internal rules for the group, e.g. being late, doing tasks you have signed up on, when we start and end the day. The result can be viewed in the appendix. The process of this production has been based on the Unified Process model, but also with elements of SCRUM. This means we have been developing incrementally, while still having a daily SCRUM meeting. That way everyone in the group was aware of our progress along the way. We divided the project into different releases, each implementing a new layer of features, so that it would be natural to evaluate at every release how much time we had to implement the next release.

4 Procedure

1. Release 1
   (a) Copying the classes Cup and Dice from CDIO-3.
   (b) Creation of the GUI board.
   (c) Making sure the cars move around on the board correctly
   (d) Making sure the players work correctly.
2. Release 2
   (a) Bankruptcy - a player exits the game is the balance is below zero.
   (b) Buying fields - a player can buy a square when he lands on it.
   (c) Rent - paying a rent when a player lands on a square which is already owned.
3. Release 3
   (a) Buying houses and hotels on Streets
   (b) Selling houses and hotels on Streets
   (c) Tax
   (d) Parking
4. Release 4
   (a) Bonus for passing the Start square
   (b) Jail
   (c) Effect of rolling two equals with the cup
   (d) Implementing Brewery and Shipping

5. Release 5.
   (a) Chancecard.
   (b) Pawn (+sale of houses and hotels to the bank) or sale to the bank
   (c) Tax 10%
   (d) Reading all strings from a file, improving the ability to change language.
6. Release 6
   (a) Trading between players
   (b) Buying houses evenly.
   (c) Button "Back" to backstep in the menus.
   (d) Pawned property building status (no building permit for same coloured streets when one or more is pawned).

The project reached release 5, meaning features listed in feature 6 has not been implemented.

4.1 Un-implemented features

As well as the Release 6, the following features has not been implemented.

1. Chance Card:
   (a) "Move three squares back"
   This Chance Card was not implemented, due to issues with compatibility with the Start bonus.
   (b) "Move to the next Shipping square. If the Shipping is owned, pay the owner double rent".
   The Chance Card was implemented without the double rent.
2. Selling properties back to the bank.
   This feature was not on the release schedule and we did not plan for it.

5 Systemrequirements

Functional

1. The game has to be played by 3-6 players.
2. The game shall use two die
3. The game has to be executable on the machines in DTU's databars.
4. The players are to take turns to roll the die.
   (a) If a player rolls two equal eyes, that player gets another turn.
   (b) If a player rolls doubles three times in a row, that player goes to jail.
5. The player has to have an account balance.
   (a) The balance starts at 30.000 kr.
6. The game must contain 8 types of squares.
   (a) *Street* - When a player lands on this square, owned by another player. The player has to pay the owner rent.
   (b) *Start* - The player is awarded a bonus of 4000 whenever, a player passes this square.
   (c) *Tax* - The player has to pay either a fixed amount, or 10 % of their fortune. This depends on which tax square, the player lands on.
   (d) *Brewery* - The player has to roll the die, 100x this amount is what the player has to pay the owner. If the owner of the labor camp, owns more than one, the amount is multiplied by the amount of breweries.
   (e) *Shipping* - The player has to pay the owner. Price determined by number of ships, as such:
      i. ship: 500.
      ii. ships: 1000.
      iii. ships: 2000.
      iv. ships: 4000.
   (f) *Parking* - Free zone, nothing happens.
   (g) *Jail* - The player can either choose to pay a fee of 1000 kr. or try to roll equal die. If the player cannot roll equal die, within 3 turns, the player automatically gets thrown out of jail, and has to pay the fee.
   (h) *Chance* - The player picks a chance card.
7. The game has to be played on a board with 40 squares developed from the former game CDIO3.
   (a) A table overview of the squares is listed in the appendix.
8. When a player owns all the streets of a certain color, this player can buy a house at the beginning of their turn
   (a) When 4 houses are owned on one lot, a hotel can be bought.
   (b) For every house and/or hotel built, the rent is increased.
9. Each players position has to be saved.
   (a) When the players get another turn, they continue from their latest position.
   (b) The player saves, how many rounds the player has been in jail.
10. The game is won, when only one player is left in the game.
    (a) A player loses the game, when they have gone bankrupt (account at 0 £ or under).
    (b) The other players continue to play.

Non functional

1. There has to be a minimum system requirements for the game.
   (a) This guide has to include a description of how to import the code from a Git repository.
2. A text has to be displayed, describing the effects of the square a player lands on.

3. The game must display a board.
    (a) This will be achieved by using a Graphical User Interface (GUI).

5.1 Noun and verb analysis

The customer want the game "Matador" to be development, due to time constraints, this will be a simplified version of the game. Extracted from the rules of the game, a requirement specification has been formed. From this a noun and verb analysis is created.

5.1.1 Noun analysis

A noun analysis has been included, to develop and retrieve an overview of prospectable classes.

Game
Players
Die
Account balance
Squares
Street
Start
Tax
Brewery
Shipping
Parking
Jail
Chance
Hotel
House
Rent
Board

5.1.2 Verb analysis

A verb analysis has been included, to develop the expected methods.

Roll
Lands
Owned
Bankrupt
Buys

These are the key methods. However a lot more is to be generated.

5.2 Use case descriptions

Use case: Sell building.
ID: 1
Brief description: What happens when you wish to sell a building.
Primary actors: 1. Player.
Secondary actors: None.
Preconditions: It's the player's turn (pre-rolling) The player has buildings to sell
Main flow: 1. The game presents the player with options. 2. The player presses sell buildings. 3. The player chooses which property to sell a building from.
Postconditions: 1. The building has been removed from the board. 2. The player's balance has been updated with the price of the sold house. 3. The player's turn continues.
Alternative flows:

Table 1: Sell house.
Describes what happens when a player wishes to sell one or more buildings. The use cases for buying buildings, pawning property or lifting the pawn on property would be nearly identical, as the system flow, and options presented, are available only when applicable

Use case: Get out of jail.
ID: 2
Brief description: How to get out of jail.
Primary actors: 1. The Player.
Secondary actors: None.
Preconditions: The Player has been jailed. It is the Players turn.
Main flow: 1. Player rolls doubles.
Postconditions: 1. The Player has been released from Jail. 2. The Player has been moved the amount of the doubles. 3. The Player gets another roll (bonus from rolling doubles)
Alternative flows: 1. If the Player does not roll doubles. • The Player stays in Jail [looped x3] • If the player does not roll doubles on the third turn, the Player is forced to post bail. – After posting bail the Player is moved the amount of the roll that wasn't a double. – If the player has insufficient funds to post bail, a menu will occur encouraging the player to sell assets or give up. 2. The Player posts bail (1000 kr.) (Pre-Rolling) • The Player is released from Jail and gets a roll. • The Player is moved out of Jail.

Table 2: Get out of Jail.
What has to happen for a Player to be released from Jail. The player can get lucky and roll doubles and be released, but if he doesn't he stays in jail for a maximum of 3 turns. If the player doesn't get doubles on turn 1,2 or 3 the player is forced to post bail. After posting forced bail, the Player will move the amount his previous roll showed. The Player also has the option to post bail instead of trying to roll doubles. If he wishes to do so, he will be released immediately and will get a roll.

Use case:
Draw Chance-card (move to nearest shipping)

ID:
3

Brief description:
What happens when you draw a chance card that sends you to the nearest shipping.

Primary actors:

1. Player.

Secondary actors:
None.

Preconditions:

The Player landed on "Prøv Lykken"-square and drew the specific chance card.

The nearest shipping company is not already owned.

Main flow:

1. The Player is moved to the nearest shipping company.
2. The game asks if the Player wants to buy the shipping company for 4000 kr.
3. The Player buys the shipping company.

Postconditions:

1. The Player is now registered as owner of the shipping company.
2. The Player balance has been updated.
3. The Players turn ends.

Alternative flows:

1. The Player has insufficient funds to buy the shipping company.
    - The Player is moved to the nearest shipping company.
    - The game will display a message saying that the Players funds are insuffcient to buy.
2. The Player opts to not buy the shipping company.
    - The Player is moved to the nearest shipping company.
    - The Player presses 'no' to buying. Ending his turn.
3. The shipping company is already owned.
    - The Player is moved to the nearest shipping company.
    - The Player pays rent of 500, 1000, 2000 or 4000 kr. depending on how many shipping companies the owner has (1-4).
        - If the player has insufficient funds to pay the rent, a menu will occur encouraging the player to sell assets or give up.
4. The shipping company is already owned, but the owner is currently in Jail.
    - The Player is moved to the nearest shipping company.
    - The Player is not charged rent.
5. The shipping company is currently pawned.
    - The Player is moved to the nearest shipping company.
    - The Player is not charged rent.

Table 3: Chance card: Move to nearest shipping.
A description of what happens when you draw a specific chance card, the move to nearest shipping card. It is relevant for all chance cards where the Player is moved to a specific location with a rent, though the calculation of rent will differ in the different card scenarios.

5.3 Use case reasoning

Since we have been hired to develop a digital version of the original Matador game, the rules and scenarios are pretty much set in stone from the beginning. We have made a few changes to the original game, to fit our expectations of what we will be able to implement for the release expected by the customer in the timeframe provided. To clarify those changes our usecases will reflect scanarios that we wanted thoroughly explained ourselves, to get an understanding of what we expect our game to do. For example the get out of Jail use case extends on the orginal rules to make it clear what and when we expect a specific thing to happen. Most of our use cases are also relevant for more than just the specific scenario described, which is one of the reasons why we don't have more of usecases in the rapport.

5.4 Use case diagram

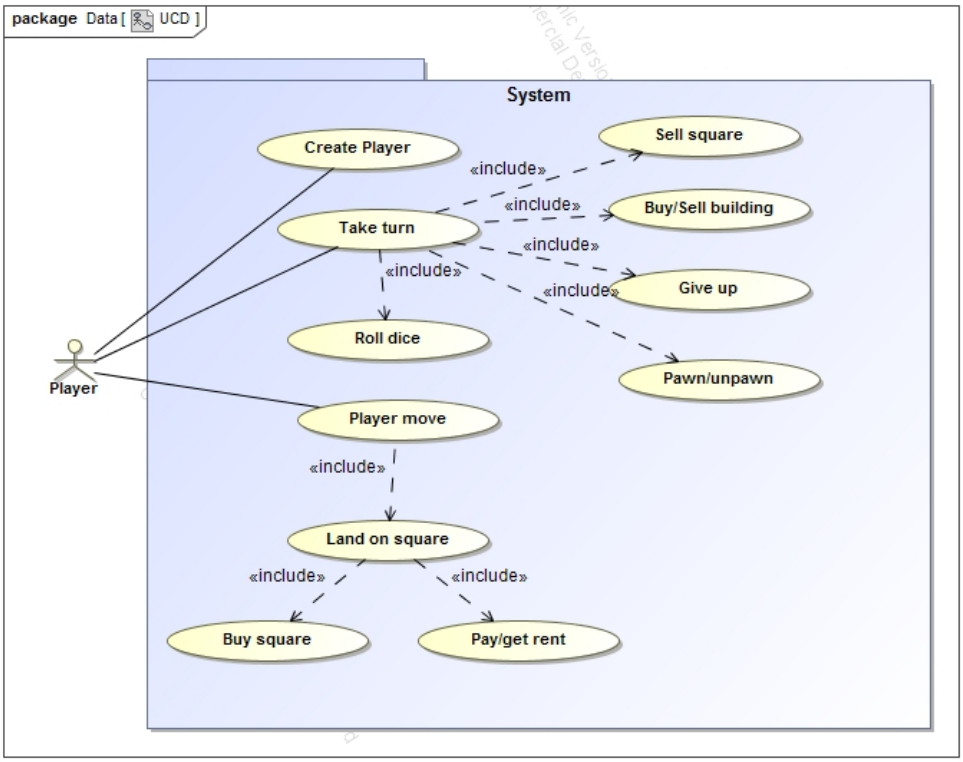


Figure 1: Use case diagram

The Use case diagram illustrates how the use cases are combined and invoked by the player or by another use case. The diagram includes use cases to which we have not written a use case description. This is to visualize by the name of the use caes how they would be invoked.

5.5 Domain model

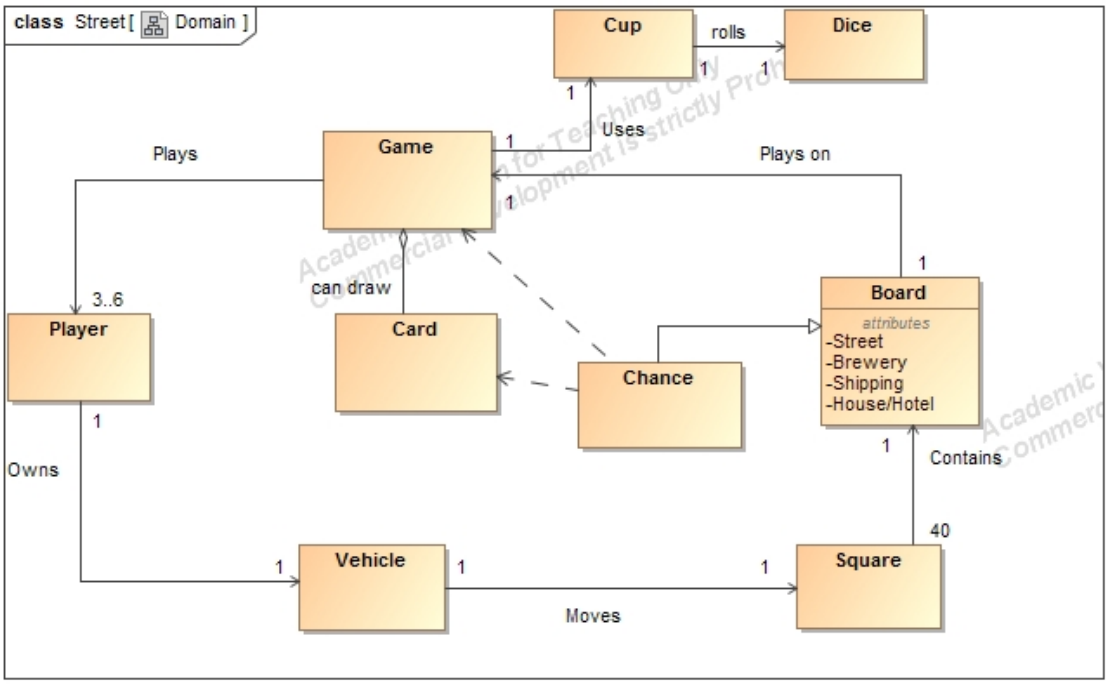


Figure 2: Domain model

Drawn from the prospectable classes, found in the Noun analysis, the following, has been picked out to be potential classes:

Game
Player
Card
Vehicle
Square
Board
Chance
Cup
Dice

These represent the basics of the game. Each player has an account balance, therefore the Account class is necessary. Furthermore for the visuals of the board, it is essential for the player to have a vehicle. This to see where the player lands, and where to move from next turn.
The square class, will contain different types of squares. Using polymorfism and creating classes for each type, thereby makes a inheritance hierarchy. This to encapsulate all the ownable squares and the needed methods for these.

6 Design

The following section contains the design of the developed program. This is described by the use of a design class diagram (DCD) and design sequence diagrams (DSD). These are two different types of diagram, one showing the final implemented classes

6.1 Design Class Diagram (DCD)

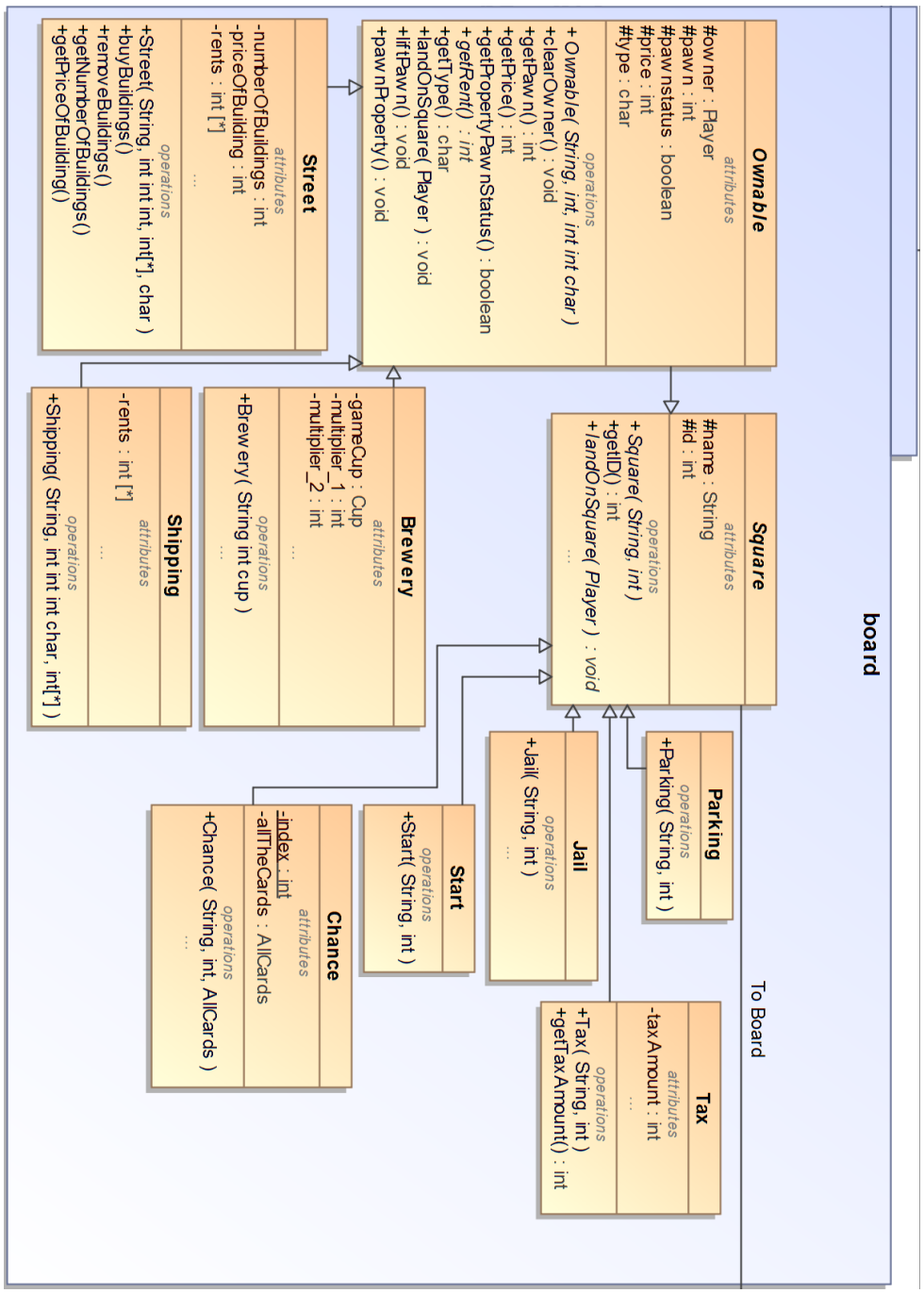


Figure 3: the package 'board'

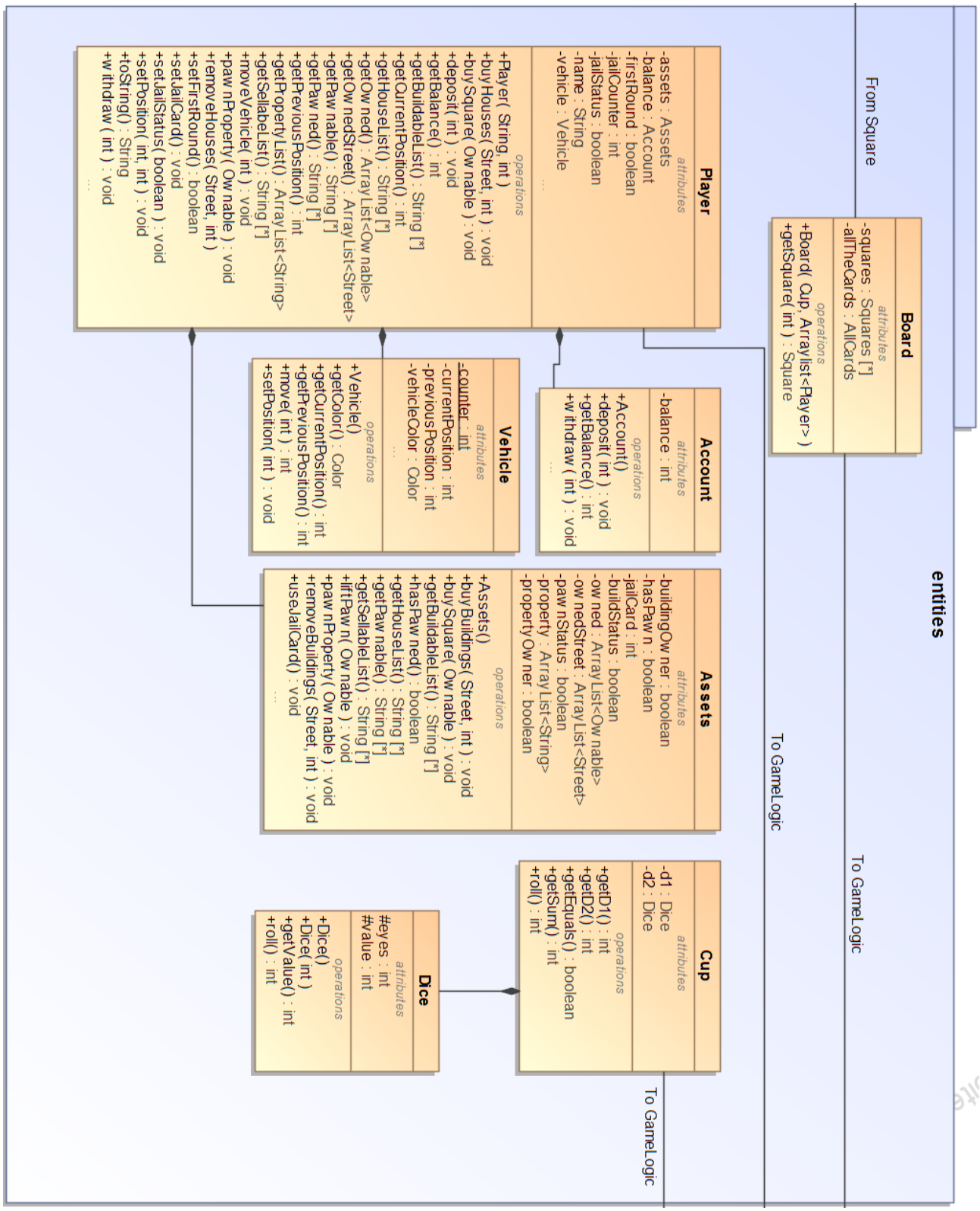


Figure 4: The package 'entities'

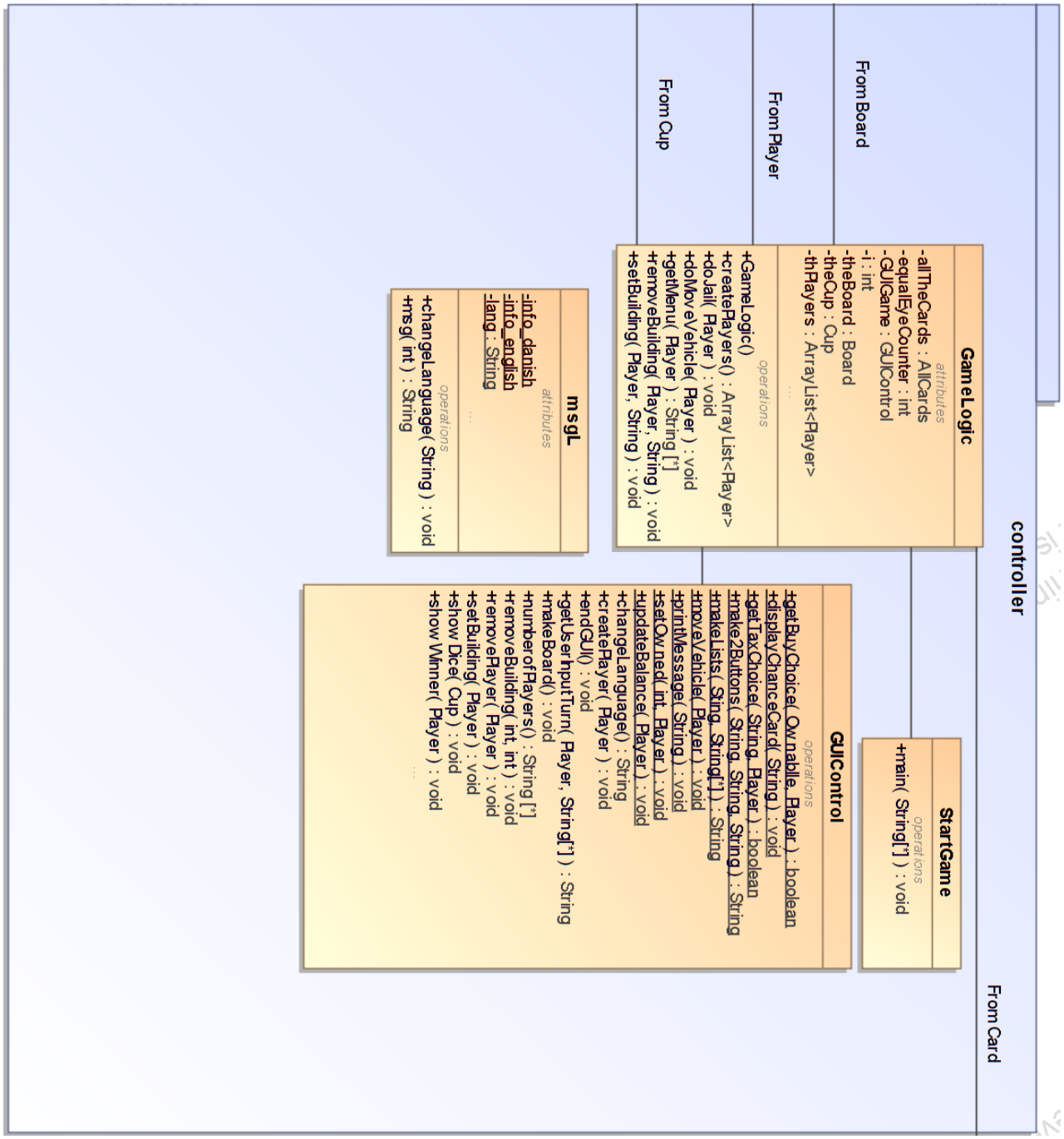


Figure 5: The package 'controller'

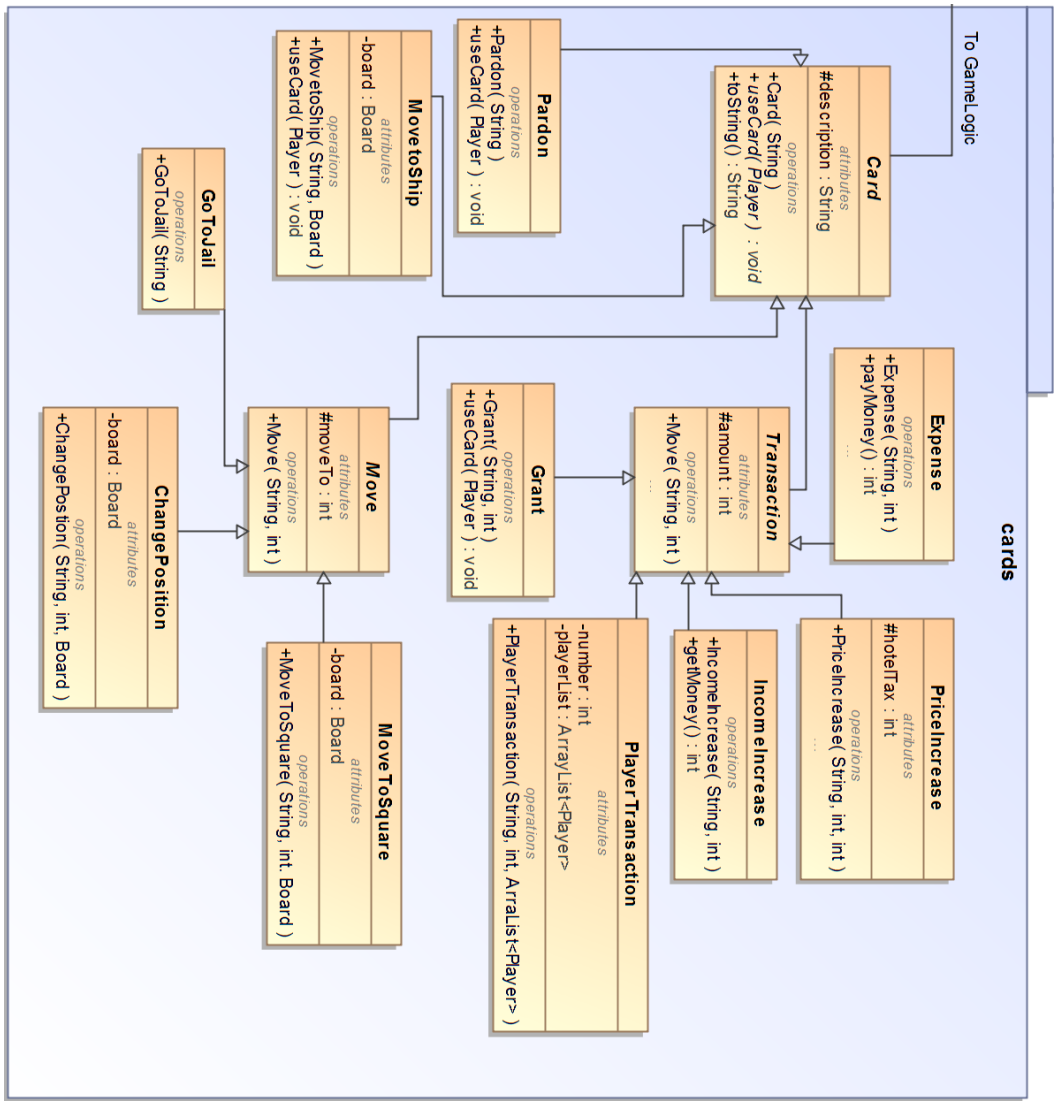


Figure 6: The package 'cards'

The above standing figures represents the unified developed product. This consists of four different packages, to separate the different kinds of classes. The board package represent all the squares on the gameboard, and which types they are. Separated into two types, Ownable and non-ownables. However both types inherit the abstract method landOnSquare from the super-class Square, defining it in the individual sub-class. By way of polymorfism, the method can be called on any subclass of Square, but the method varies depending on the object landOnSquare is called upon. Street, Shipping and Brewery furthermore inherits the abstract method getRent from the class Ownable. This is because landOnSquare is common for all Ownable subclasses, the only this that varies is the way the rent is calculated. The package entities lays its foundation in classes containing memory. Here Player and Assets are the central classes, due to them containing all information about the players status in the game, what they own etc. The package controller, includes GUIControl, GameLogic, StartGame and msgL. GUIControl initializes the graphical user interface(GUI) to make an actual visual board and increases usability. msgL is an added feature, which makes it possible to choose between danish and english. GameLogic is the central class in the game. This defines how the game should play out and how a players turn should be presented.The cards pack is dedicated entirely to the implemented feature of "chance cards". Here they are separated in the different types of cards. This is transaction and move. These are specified in the individual classes. Inheriting the abstract method useCards from the class 'Card'. This method is specified for individual use in every subclass, which again is a usage of polymorphism.

6.2 Design Sequence Diagram (DSD)

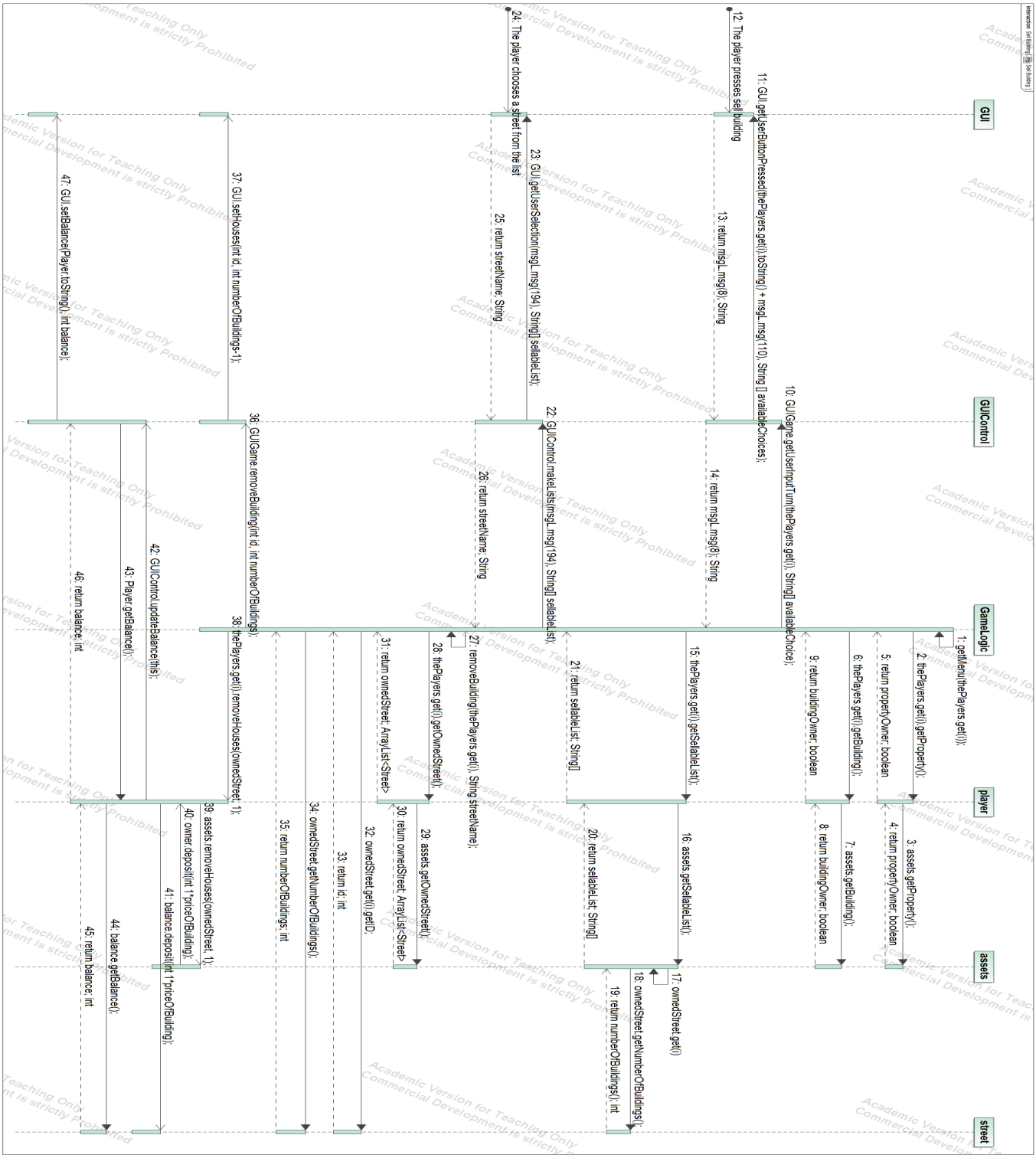


Figure 7: Sell Building DSD

The previous figure represents a use case where a player wants to sell a building. Firstly, the system checks whether or not the player even has buildings to sell. This is done from sequence 2 to 9, which represents if-statements that checks the instance of assets belonging to the player. In this sequence, it is chosen that they are elevated to true, as this triggers a button appearing on the GUI during the player turn. This will allow the player to click the "Sell building" button, which then triggers a check for which properties the player owns, with buildings on. This happens from sequence 15-21, after which the system pushes this list to the GUI. The player can then choose a property, after which a building is sold. Finally, the system automatically deposits the price of the building to the player, and updates the balance to the GUI. This all happens in a for-loop controlling a player's turn, meaning the player will still have their turn after this has happened. For this DSD, it was decided not to show the call to the msgL class, as this code is explained in the appendix.

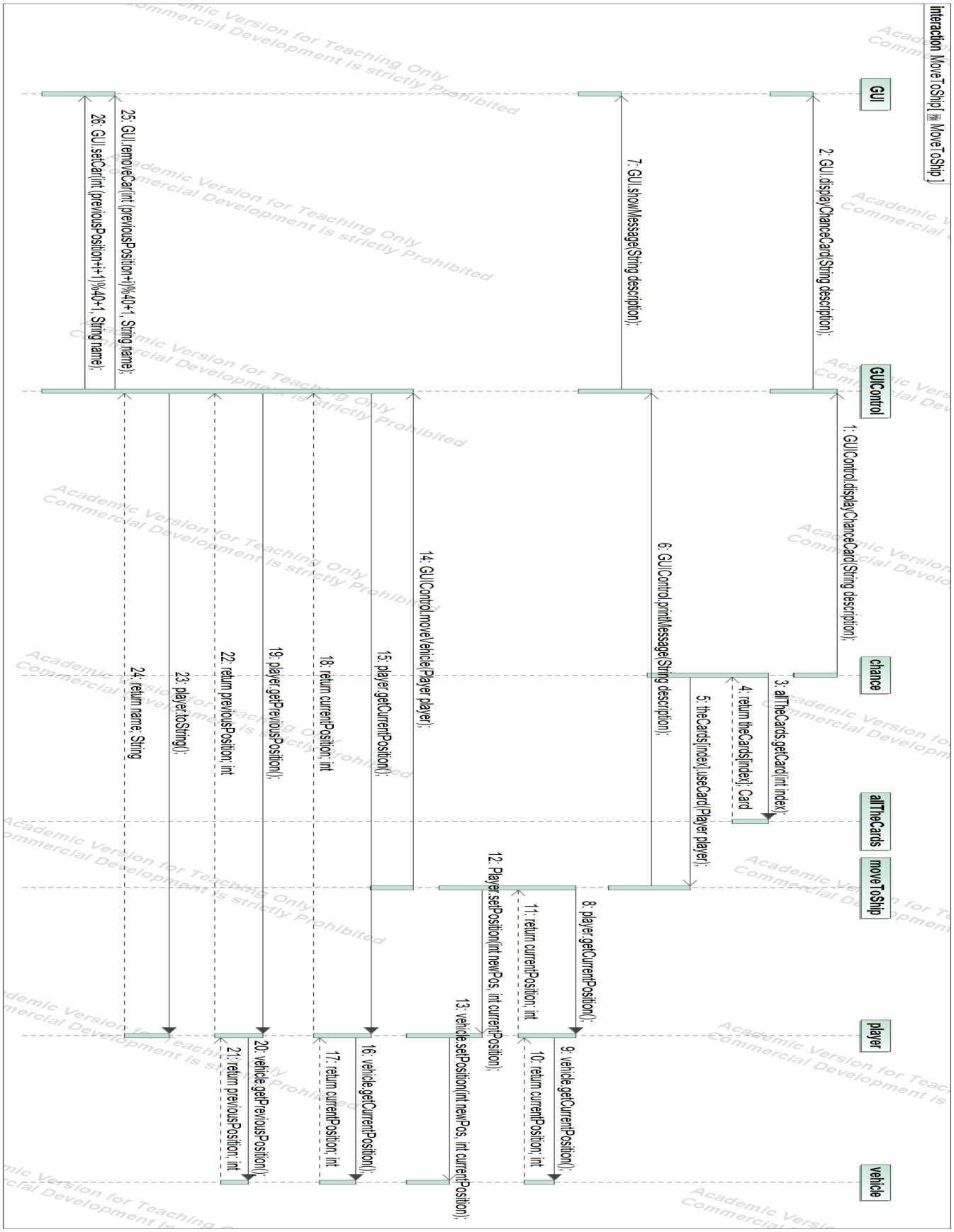


Figure 8: MoveToShip DSD (Movement)

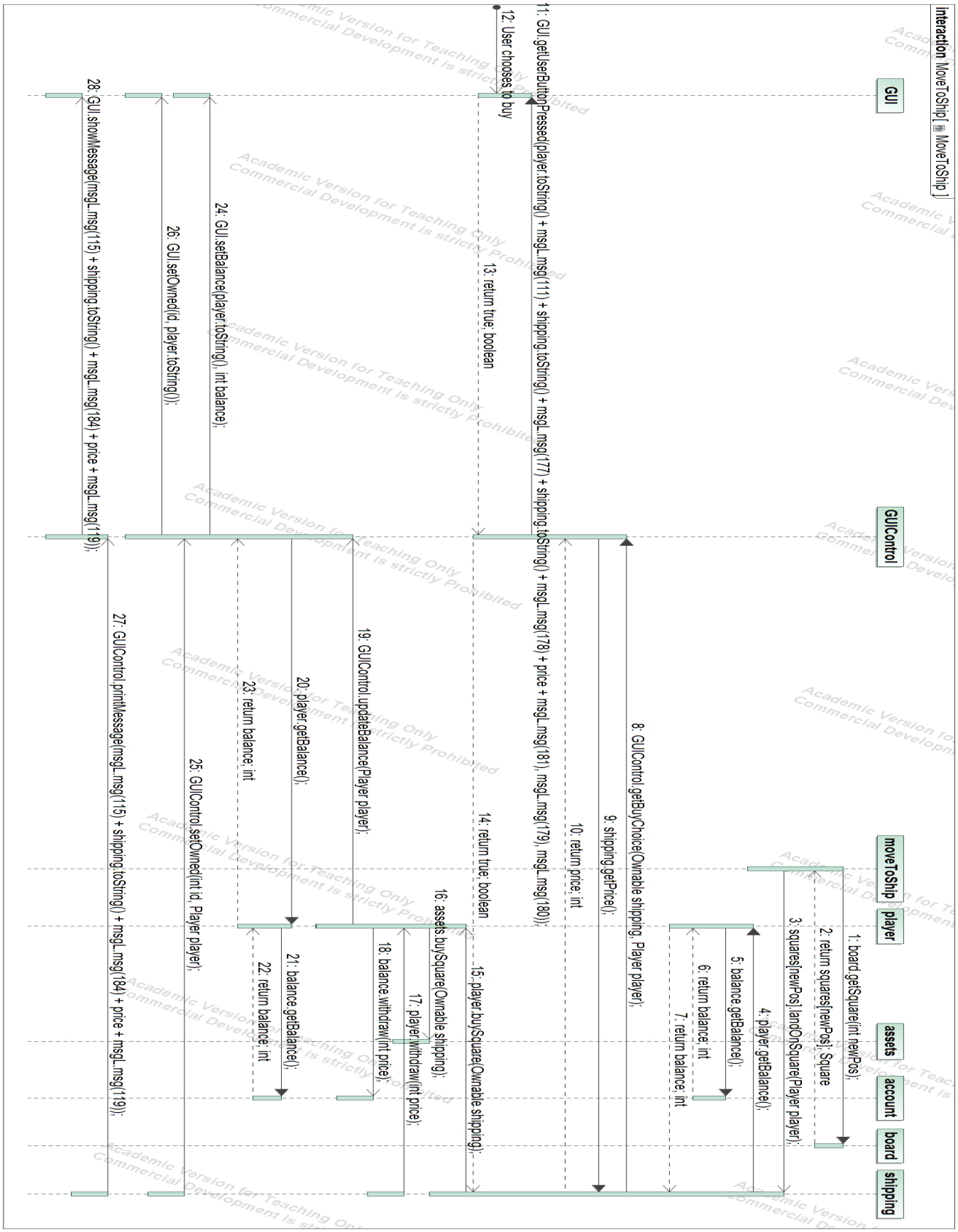


Figure 9: MoveToShip DSD (Landing on square)

The two MoveToShip DSDs visualizes what happens when a player draws the "Move to nearest shipping" chance card, which is shown in the (Movement) part. Here it is seen how the player draws a chance card and has the description of the card displayed through the GUI. After this, the system evaluates where the player stands on the board, and moves the player to the nearest shipping. This is done by an if-statement that happens right before sequence 8. Here, the system checks if the player's current position matched a certain square, and moves the player accordingly. At the end of the sequence, the player's piece on the board is moved. Sequence 25-26 represents this, and is explained in more detail in the appendix. In the (Land on square) part, the player has been moved to the nearest shipping. This sequence represents a use case where the shipping isn't already owned, and where the player decides to buy the property. This use case was chosen as it accurately displays not only how the system handles landing on a square, but also how it handles when a player choose to buy a property.

7 Implementation

The following section contains the implementation part of the project. During the process of development, certain choices have been made, to improve the experience and functionality of the game. Therefore distinctive methods will be explained and reasoned by the use of code examples and descriptive text.

7.1 moveVehicle

The method moveVehicle is used in the project, as a combined method, with two responsibilities. These are a visual movement of the players vehicle on the board, and managing the bonus, when passing start.

Listing 1: Try-catch

```
try {
          Thread.currentThread();
          if (value > 12) {
            Thread.sleep(0);//for test modes, the vehicle teleports.
          } else {
            Thread.sleep(150);//the thread sleeps for 150 milliseconds
          }
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
```

What makes this method interesting, is the included 'try-catch' method. Due to the use of inheritance in the project, it is possible to run classes as threads. This makes it possible, to execute more than one event at one time. The 'try-catch' method contains two blocks. The 'try' block contains an if statement. As seen above there are two incidents. One of them used for test modes, which makes the vehicle teleport, and optimizes the testing of the entire project. The 'else' tells us, that the thread sleeps for 150 milliseconds.

```
            Thread.sleep(150);//the thread sleeps for 150
            //milliseconds
```

The statement 'the thread sleeps for 150 miliseconds' is explainable by the thread having to "rest", this way it is able to execute again. Otherwise it would consume all the CPU time for process. This would lead to the other methods and threads not being able to execute.

The 'catch' block executes if the 'try' block exception is called. The print Stack is included as an assisting tool to help identify the exception.

Visually speaking, the costumer will see the influence of this code, by it making the vehicle move from square to square, one by one. The 'try' blocks sleep function makes sure it rests between the sqares. The 'catch' block "awakens" the vehicle, by using InterruptedException.

7.2 msgL - static

The static block is used in the project for reading text from a file and inserting it into a String array.

Listing 2: static block

```
1   static {
2     String fileName = "language2";
3     File file = new File(fileName);
4     String tempString;
5     int Danish = 1;
6     int English = 2;
7     int i = 0;
8     int j = 0;
9     try {
10      Scanner inputStream = new Scanner(file);
11      while (inputStream.hasNextLine()) {
12        tempString=inputStream.nextLine();//stores the next line in a temporary string
13        info_danish[i++] = tempString.split("\t")[Danish];//splits the text by tabs
14        info_english[j++] = tempString.split("\t")[English];
15      }
16      inputStream.close();
17
18    } catch (FileNotFoundException e) {
19      System.out.println("Forkert filnavn");
20    }
21  }
```

The static block is executed as soon as the JVM loads the class, which means we don't have to initialize the block using a constructor.
In this static block we read a a number of strings from a file, seperate the lines by "^" and insert column 1 in the Danish array and colomn 2 in the english array.

7.3 AllCards - shuffle()

To be able to use our cards as a normal card game, we have to make a shuffle method in the class AllCards.

Listing 3: Shuffle()

```
1   public void shuffle(){
2       Random r = new Random();
3
4       for(int last = theCards.length-1; last > 0; last--){
5         int i = r.nextInt(last+1);
6         Card tmp = theCards[last];
7         theCards[last] = theCards[i];
8         theCards[i] = tmp;
9       }
10  }
```

The algorithm works by taking the bottom card, and remember it as tmp

```
last = theCards.length-1
Card tmp = theCards[last]
```

Then we generate a random number between 0 and 43 with:

```
int i = r.nextInt(last+1)
```

Then switching the bottom card with a random card from the pile

```
theCards[last] = theCards[i]
theCards[i] = tmp
```

That is the main function.
Then we have to repeat this with all the cards in the pile from the bottom to the top.
To do that we use a for loop, that start from last and count down `i--`

8 Test

8.1 TestMode testcases

We have created a TestMode in our project, giving you the choice of playing a normal game or loading into a test mode at the beginning of the game. We did this because we felt it was a good way to show, that the features we have been working on implementing actually works as intended. The TestMode has build in test case scenarios that are scripted to fit the outcome we wanted to test. You can see the test scripts in the following tables, including expected results per turn. Other than being a good way of showing off the games features, it helped us tweak a few things and find a few small bugs. For the preset test cases to make sense, the scripts needs to be followed.

Test case ID	TC 1
TestMode testcase:	1
Summary	Testing implemented features: Buy property, house, hotel. Sell property, house, hotel. Pawn property. Double rent (when all of one kind of street is owned) Gone Broke (loosing)
Requirements	TestMode Active. TC1 Chosen.
Test Procedure	Using TestMode
Test data	Roll (0,37): Player 1 Lands on Frederiksberggade and buys it. Roll (0,32): Player 2 lands on Vimmelskaftet and opts not to buy. Roll (0,19): Player 3 lands on Strandvejen and buys it. Roll (1,1): Player 1 lands on Rådhuspladsen and buys it. He then buys 5 houses(hotel) on Rådhuspladsen. Roll (0,21): Player 1 lands on Helle. Collects startbonus. Roll (4,1): Player 2 lands on Frederiksberggade. Roll (0,20): Player 3 lands on Rådhuspladsen, Does not have enough money, pawns owned property (Strandvejen), still not enough cash, has to give up. Roll (n/a): Player 1 sells hotel and houses on Rådhuspladsen.
Expected result	Player 1 should get option to buy Frederikberggade (Game should register Player 1 as owner and withdraw the price of the street) Player 2 should get option to buy, declines. (Game should register the decline, turn should end) Player 3 should get option to buy Strandvejen (Game should register Player 3 as owner and withdraw the price of the street) Player 1 should get option to buy Rådhuspladsen (Game should register Player 1 as owner and withdraw the price of the street) Player 1 should get extra turn from rolling double 1's. (Game should offer him to buy houses because he owns both streets of same type. Then should register the bought houses on the street and withdraw 4000 kr. pr. house.) Player 1 should get 4000 kr. from passing Start and his turn should end when he lands on Helle. Player 2 should be witdrawn 1400 kr. (double rent) (Game should inform of rent amount.) Player 3 should pay rent of 40.000, balance should go to -9000, then -7000 after pawning Strandvejen (half the cost of buying the property), (Game should register that the Player wishes to give up and remove him as owner from the pawned street and remove his vehicle.) Player 1 should get option to sell houses because he has at least one. His balance should be added with 2000 kr. pr. house sold. (Game should remove hotel/houses on the board accordingly)
Actual result	All as expected.
Status	Passed
Tested by	Morten Velin Christensen
Date	15/1-2017
Test environment	Eclipse Neon, v. 4.6.0

Test case ID	TC 2
TestMode nr:	2
Summary	Testing implemented features: Land on go to jail. Then move to jail Player lands on owned street. The player doesn't pay rent Rolling doubles three times in a row. Then jail. Rolling doubles in jail. Then the player is released from jail. Pay bail. When a player is in jail, and the player doesn't roll doubles for the third time. Then the player has to pay bail and move.
Requirements	TestMode Active. TC2 Chosen.
Test Procedure	Using TestMode
Test data	Roll (3,3): Jens lands on Roskildevej and opt to buy. He gets another turn. Roll (23,1): Jens lands on Go to jail and moves to jail. Roll (2,2): Peter lands on Indkomstskat and opt to pay 4000 kr. He gets another turn. Roll (4,4): Peter lands on Tuborg and opt not to buy. He gets another turn. Roll (5,5): Peter rolls doubles for the third time and he moves directly to jail. Roll (3,3): Line lands on Roskildevej and the owner is in jail, so she doesn't have to pay him. She gets another turn. Roll (2,2): Line lands on visit jail and she gets another turn. Roll (1,2): Line lands on Bülowsvej and opt not to buy. Roll (4,5): Jens is still in jail. Roll (4,4): Peter is released from jail and lands on Hellerupvej. He opt not to buy. He gets another turn. Roll (3,4): Peter lands on Scandlines Gedser-Rostock and opt not to buy. Roll (10,7): Line lands on go to jail and moves to jail. Roll (5,4): Jens didn't roll doubles is he is sill in jail. Roll (4,2): Peter lands on Amagertorv and opt not to buy. Line opt to pay bail. Roll (6,1): Line lands on Prøv lykken Roll (5,6): Jens did not roll doubles for the third time and has to pay 1000 kr. He lands on Trianglen and opt not to pay.
Expected result	Jens lands on Roskildevej, and he should get option to buy it. He rolled doubles so he should get another turn Jens lands on go to jail, he should move to jail. Peter lands on Indkomstskat and should get option to pay 4000 kr or 10 % tax of all his assets. He rolled doubles so he should get another turn Peter lands on Tuborg and should get option to buy it. He rolled doubles for the second time so he should get another turn Peter rolls doubles for the third time so he should move directly to jail. Line lands on Roskildevej, which is owned by Jens. But Jens is in jail, so she shouldn't pay rent. Line lands on Bülowsvej and should get option to buy it. Jens is in jail, he didn't roll doubles so he should not get out of jail. Peter rolls doubles so he should be released from jail and move the rolled eyes on the board. He lands on Hellerupvej and should get option to buy it. He rolled doubles so he should get another turn. Peter lands on Scandlines Gedser-Rostock and should get option to buy it. Line lands on go to jail, and should move to jail. Jens didn't roll doubles, so he should still be in jail. Peter lands on Amagertorv and should get option to buy it. Line opt to pay bail, she should get another turn and move out of jail. Line lands on Prøv lykken. She should get a changecard. Jens did not roll doubles for the third time and he should pay 1000 kr and move out of jail. Jens lands on Trianglen and should get option to buy it.

Actual result	All as expected.
Status	Passed
Tested by	Jonas Larsen
Date	14/1-2017
Test environment	Eclipse Neon, v. 4.6.0

Test case ID	TC 3
TestMode nr:	3
Summary	Testing implemented features: Lands on Prøv lykken, and the player draws a card. Pardon Card. Go to jail card. Player Transaction card. Players get start bonus when passing the first square.
Requirements	TestMode Active. TC2 Chosen.
Test Procedure	Using TestMode
Test data	Roll (21,1): Jens lands on Prøv lykken. He draws a pardon card. Roll (16,1): Peter lands on Prøv lykken. He draws a go to jail card and moves to jail. He doesn't get the start bonus. Roll (33,3): Line Lands on Prøv lykken. She draws Player Transaction card and receives 200 kr from each player. Roll (4,4): Jens lands on go to jail, but he has a pardon card so just visits the jail. He rolled doubles so he gets another turn. Roll (3,1): He lands on a street and opt not to buy. Roll (3,3): Peter rolls doubles and gets out of jail. He lands on a street and opt not to buy. He rolled doubles so he gets another turn Roll (5,2): Peter lands on a street and opt not to buy. Roll (6,6): Line passes the first square and receive 4000 kr.
Expected result	Jens lands on Prøv lykken, he should draw a pardon card. Peter lands on Prøv lykken, he should draw a go to jail card and not receive 4000 kr from start bonus. Line lands on Prøv lykken, she should draw a player transaction card and receive 200 kr from each player Jens lands on go to jail, but he has a pardon card. He should move to jail for a visit. He rolled doubles so he should get another turn. Jens lands on a street and should get option to buy it. Peter roll doubles and should be released from jail and get another turn Peter lands on a street and should get option to buy it. He rolled doubles so he should get another turn. Peter lands on a street and should get option to buy it Line passes the first square and should receive 4000 kr.
Actual result	All as expected.
Status	Passed
Tested by	Jonas Larsen
Date	14/1-2017
Test environment	Eclipse Neon, v. 4.6.0

8.2 JUnit tests

8.2.1 TestBonus.java

The TestBonus JUnit test case uses the same for loop as used in the moveVehicle method in GUIControl.
The starting conditions are set up in the @Before block, before every @Test is run. This before block sets the balance, name, orgCurrentPosition and orgPreviousPosition. Every test modifies the position by a rollValue, and then is it evaluated whether the player gets a Start bonus. The test covers when your roll didn't get you over the Start square, when you landed exactly on the Start square and when your roll moved you past the Start square.
The final test covers your first roll of the game, where you are starting on the start square, but when you move past it, you should not be awarded a Startbonus.

8.2.2 TestTaxPercent.java

The TestTaxPercent test takes a player, lets him buy some Streets and build buildings on them. It then calculate an extra house tax for the houses and hotels that are built on them. It withdraws this extra tax from the players account, and tests to see if the account balance is as expected.

8.3 User test

The game was tested by an electrotechnology student from DTU. The student was observed while running the game through. The following was observed:

- It is not possible to see on the board, if a property is pawned or not.
- If you click the button to pawn a property, it is not possible to go back, without pawning.
- It is not possible to see all the text in the middle of the screen. when reading a square

8.4 Test conclusion

All in all, the testing procedure was executed as a combination of testcases, unit testing, user tests and us running through the program multiple times. By doing this, many of the flaws were detected. Thereby we were able to fix many of them, the rest were taken into strong consideration, and is decided to be fixed for future releases.

9 Conclusion

We successfully developed a program compiled in Java. Some features were not implemented, but this was included in our time schedule. We found several small issues, which we did not have the time to implement correctly, f.x. "Move three squares back" was not implemented because it conflicted with the Start bonus feature.
In hindsight, there wasn't used enough time on the analysis stage of the development, but we simply didn't have the time in the first place. Apart from that, our process worked well. The structure with releases, implementing a number of features, gave us an opportunity to see the bigger picture after every release. That way, almost every feature was implemented fully before moving on to the next release.
The tests conducted on the program revealed a number of issues but were succesfully corrected in the source code. Apart from the test cases we also ran the game a couple of times in Normal Mode, to reveal unforseen issues.

10 Appendix

10.1 Squares

Table 4: List of squares and their properties

Feltnavn	Pris	Leje	m/1 hus	2 huse	3 huse	4 huse	hotel	Hus pr	Hotel pr	Pant	udgifter
Start											+4.000
Rødovrevej	1200	50	250	750	2250	4000	6000	1000	1000	600	
Prøv lykken											
Hvidovrevej	1200	50	250	750	2250	4000	6000	1000	1000	600	
Indkomstskat											-4000
Scandlines H-H	4000	500	1000	2000	4000					2000	
Roskildevej	2000	100	600	1800	5400	8000	11000	1000	1000	1000	
Prøv lykken											
Valby Langgade	2000	100	600	1800	5400	8000	11000	1000	1000	1000	
Allégade	2400	150	800	2000	6000	9000	12000	1000	1000	1200	
Fængsel - besøg											
Frederiksberg Allé	2800	200	1000	3000	9000	12500	15000	2000	2000	1400	
Tuborg Squash	3000	x100	x200							1500	
Bülowsvej	2800	200	1000	3000	9000	12500	15000	2000	2000	1400	
Gl. Kongevej	3200	250	1250	3750	10000	14000	18000	2000	2000	1600	
Mols-Linien	4000	500	1000	2000	4000					2000	
Bernstorffsvej	3600	300	1400	4000	11000	15000	19000	2000	2000	1800	
Prøv lykken											
Hellerupvej	3600	300	1400	4000	11000	15000	19000	2000	2000	1800	
Strandvejen	4000	350	1600	4400	12000	16000	20000	2000	2000	2000	
Parkering											
Trianglen	4400	350	1800	5000	14000	17500	21000	3000	3000	2200	
Prøv lykken											
Østerbrogade	4400	350	1800	5000	14000	17500	21000	3000	3000	2200	
Grønningen	4800	400	2000	6000	15000	18500	22000	3000	3000	2400	
Scandlines G-R	4000	500	1000	2000	4000					2000	
Bredgade	5200	450	2200	6600	16000	19500	23000	3000	3000	2600	
Kgs. Nytorv	5200	450	2200	6600	16000	19500	23000	3000	3000	2600	
Coca Cola	3000	x100	x200							1500	
Østergade	5600	500	2400	7200	17000	20500	24000	3000	3000	2800	
Fængslet											
Amagertorv	6000	550	2600	7800	18000	22000	25000	4000	4000	3000	
Vimmelskaftet	6000	550	2600	7800	18000	22000	25000	4000	4000	3000	
Prøv lykken											
Nygade	6400	600	3000	9000	20000	24000	28000	4000	4000	3200	
Scandlines R-P	4000	500	1000	2000	4000					2000	
Prøv lykken											
Frederiksberggade	7000	700	3500	10000	22000	26000	30000	4000	4000	3500	
Ekstraordinær statsskat											-2000
Rådhuspladsen	8000	1000	4000	12000	28000	34000	40000	4000	4000	4000	

10.2 Git Strategy

10.2.1 Patterns

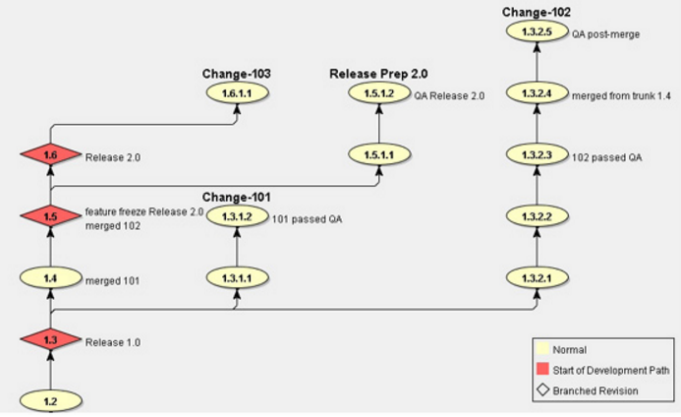


Figure 10: Branch per change

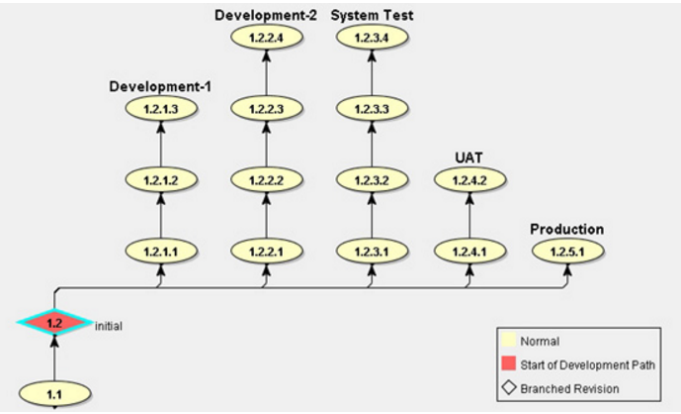


Figure 11: Branch per environment

10.2.2 Documentation

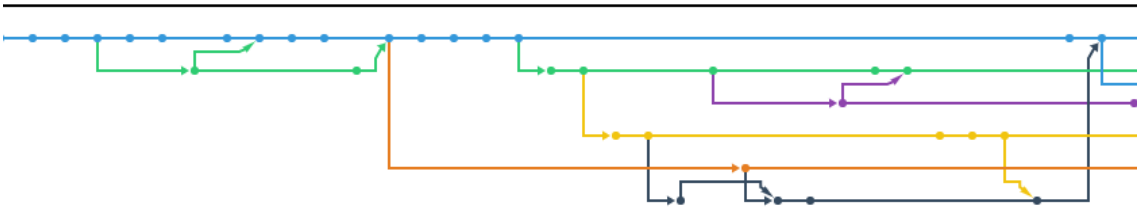


Figure 12: Representation of the strategy outcome

10.3 Use cases

Use case: Roll
ID: 2
Brief description: The die are thrown.
Primary actors: 1. Players.
Secondary actors: None.
Preconditions: The game is executing.
Main flow: 1. A player rolls the die. 2. The result of the roll is determined.
Postconditions: The board recieves the result of the roll.
Alternative flows: None.

Table 5: roll.
Use case description for the function roll. This diagram describes what happens when this function takes place. In Diagram 1, it shows which effect it has on the game.

10.4 System requirement

Minimum

1. Processor: Pentium
2. Memory: 128 MB
3. Additional comments:
4. Compiler program, such as Eclipse
5. Java version 8 update 111
6. Command language interpreter (CLI) for running the java file.

10.5 Test cases

Test case ID	TC 4
Summary	Testing Brewery rent is calculated correct when a player owns both of them.
Requirements	FakeDice initialized. CupFile updated with preset rolls.
Test Procedure	Roll (6,6): Player 1 lands on brewery Tuborg and buys it. Get extra turn for rolling doubles. Roll (10,6): Player 1 lands on Brewery Carlsberg and buys it. Ending his turn. Roll (6,6): Player 2 lands on brewery Tuborg.
Test data	See FakeCup and CupFile.
Expected result	Player 2 is charged eyes * 2 * 100 in rent.
Actual result	Double 6's rent charged 2400.
Status	Passed
Tested by	Morten Velin Christensen
Date	
Test environment	Eclipse Neon, v. 4.6.0

Test case ID	TC 5
Summary	Testing player options when balance is under 0 and no property is owned.
Requirements	FakeDice initialized. CupFile updated with preset rolls. Player initialized with balance of 3000 instead of 30.000
Test Procedure	Roll (2,2): Player 1 lands on income tax and chooses to pay 4000
Test data	See FakeCup and CupFile.
Expected result	Players only option should be to give up (not enough money and nothing to sell)
Actual result	Player options limited to give up.
Status	Passed
Tested by	Morten Velin Christensen
Date	
Test environment	Eclipse Neon, v. 4.6.0

Test case ID	TC 6
Summary	Testing player options when balance gets under 0 and property is owned.
Requirements	FakeDice initialized. CupFile updated with preset rolls. Player initialized with balance of 2000 instead of 30.000
Test Procedure	Roll (3,3): Player 1 lands on Roskildevej and buys it (2000 kr.) Roll (16,16): Player lands on extraordinary tax and has to pay 2000 kr.
Test data	See FakeCup and CupFile.
Expected result	Player balance is updated to 0 after buying Roskildevej. Player gets extra turn for rolling doubles (3,3) Player lands on extraordinary tax and has to pay 2000 kr. Not enough money, option to sell property or give up should be displayed.
Actual result	Player lands on Roskildevej and buys it, but does not get extra turn. Error regarding extra turn when balance equals 0 found.
Status	Failed
Tested by	Morten Velin Christensen
Date	10/1-2017
Test environment	Eclipse Neon, v. 4.6.0

Test case ID	TC 7
Summary	Testing player options when balance gets under 0 and property is owned.
Requirements	FakeDice initialized. CupFile updated with preset rolls. Player initialized with balance of 2000 instead of 30.000
Test Procedure	Roll (3,3): Player 1 lands on Roskildevej and buys it (2000 kr.) Roll (16,16): Player lands on extraordinary tax and has to pay 2000 kr.
Test data	See FakeCup and CupFile.
Expected result	Player balance is updated to 0 after buying Roskildevej. Player gets extra turn for rolling doubles (3,3) Player lands on extraordinary tax and has to pay 2000 kr. Not enough money, option to sell property or give up should be displayed.
Actual result	Player lands on Roskildevej and buys it, balance is updated to 0. Player gets another roll (from doubles roll) Player lands on extraordinary tax and has to pay 2000 kr. Player balance is -2000 and menu forces him to sell or give up.
Status	Passed
Tested by	Morten Velin Christensen
Date	10/1-2017
Test environment	Eclipse Neon, v. 4.6.0

10.6 Revised Rules

The following is the revised rules, based on the official "Matador" rules. These have been changed to reflect that the game is now on a computer, and has been rewritten to only include implemented features:

10.6.1 Purpose of the game

- The purpose of the game is to buy, rent or sell property in an adventageous way, such that only one person has monopoly.
- Pieces are placed on "START" and is moved clockwise around on the board, according to dice rolls. When a player's piece lands on a property that isn't already owned by another participant, the player can buy the property from the bank, and collect rent from opponents when they land on the property.
- The rent of the property is increased by building houses and hotels.
- To get more money, property can be pawned to the bank.
- The squares "Chance card" gives a player the right to draw a card, after which the order on the card has to be followed.
- Occassionally a player is sent to jail.

10.6.2 Preparations

- The system itself acts as the bank. Each player recieves 30000 kr. at the start of the game.
- Payment of rent, tax collection and collection of various fees will be handled automatically by the system.
- It won't be possible to trade between players.
- Chance cards are generated by the system.
- The dice are likewise generated by the system.

10.6.3 The game itself

- At the start of the game, the players will be asked to input how many players is going to player.
- After this, a name is entered for each player, which decides the order in which the players take their turn.
- The first player throws the die, after which the system will move the player's piece an amount of squares, according to the eyes of the dice. The square that the player lands on will then take effect, after which the turn passes to the next player.
- If a player passes "START", a bonus of 4000 kr. will be awarded.
- If a player lands on a "Chance card" square, the system will draw a chance car. The effect of the card will be displayed by the system and the card will take effect.
- At the start of the game, the chance cards will be generated in random order by the system. After the "pile" is empty, the system will generate them again, in a new random order.
- If a player lands on a property already owned by another participant, the system will collect rent from the player that landed on the square and pay the rent to the owner. This is handled by the system.
- If a player lands on the square "Go to jail", the player will be moved straight to jail. The start bonus of 4000 kr. will not be collected. However, if a player lands on the "In jail" square, the player in question is only visiting and will continue their next turn as normal.
- If a player lands on the "Income taxes" square, the player can pay either 4000 kr. or 10 percent of their total assets, calculated as the total value of their cash, buildings and price of property, including the pawned property. The player will not be notified about the total value until after they decide.
- Players will recieve an extra turn if they roll doubles. The effect of the square after the first roll, as well as the effect of the square after the second roll, will take effect. If a player rolls doubles three times in a row, the player will be sent directly to jail.
- The square "Parking" is a sanctuary, where nothing happens.

10.6.4 Getting out of jail

- A player can get out of jail in the following way:
  - By paying a fine of 1000 kr. before the die are rolled.
  - By using one of the "Get out of jail free" cards from the pile of chance cards.
  - By rolling a double, after which the player immediately moves the amount of squares indicated by the eyes of the roll, and recieves an extra turn as normal.
- A player can't be jailed for more than three turns straight. If a player doesn't roll a double by their third turn, the player immediately pays the fine of 1000 kr. and will immediately move the amount of squares indicated by the eyes of the roll.
- While in jail, a player can't collect rent from other players, but can still build houses and hotels.

10.6.5 Houses and hotels

- If a player owns all the property of the same color, the player will collect double rent on the undeveloped property and can start building houses. The houses are bought from the bank.
- A player can build up to four houses on each property, after which a hotel can be build. Only one hotel can be build, and the system will automatically remove the houses, if a hotel is build. Money is not returned for the four houses.
- Buildings can be sold back to the bank. The price of a hotel is five times that of a house.
- A player can only build and sell houses and hotels at the start of the player's own turn.

- There is no limit on the amount of houses and hotels that can be build, except for the one set by the squares.
- Buildings has to be sold back to the bank before a property can be pawned.

10.6.6 Pawning

- Only undeveloped property can be pawned to the bank for the amount listed on the deeds. If there are buildings on the property, these has to be sold to the bank. The player keeps ownership of the property while pawned.
- If a player wishes to lift the pawn, they will have to pay interests of 10 percent on top of the loan, meaning that if a property is pawned for 1000 kr., then 1100 kr. is to be paid to lift the pawn.
- Buildings can not be constructed on pawned property.
- It is only the player that has pawned the property that can buy it back.
- Rent is not collected on pawned property.
- Pawning of a property happens through the bank.

10.6.7 Bankruptcy

- If a player owes more cash than he owns, the bank will withdraw the money anyway. After this, the player will recieve the oppotunity to sell buildings and pawn property in an attempt to take their balance out of debt. If this doesn't happen, the player will be removed from the game.
- When a player is removed from the game, all the player's property will be returned to the bank, after which they can be bought following normal game rules by landing on the proper square.

11 JavaCode

Listings

11.1 Package controller

Listing 4: GameLogic

```
1   package controller;
2
3   import java.util.ArrayList;
4
5   import entities.Board;
6   import entities.Cup;
7   import entities.Player;
8   import test.FakeCup;
9   import board.Ownable;
10  import board.Street;
11
12  public class GameLogic {
13
14     private GUIControl GUIGame;
15     private Board theBoard;
16     private Cup theCup;
17     public ArrayList<Player> thePlayers;
18     private int equalEyeCounter;
19     private int i;
20     private boolean testMode = false;
21     private int startAmount;
22
23     /**
24      * GameLogic controls the gameflow
25      */
26
27     public GameLogic() {
28
29        // Initializing the game.
30
31        GUIGame = new GUIControl();
32        GUIGame.makeBoard();
33
34        // How to start the game.
35        GameOrTestMode();
36
37        // The players are initialized
38        thePlayers = createPlayers(testMode, startAmount);
39
40        theBoard = new Board(theCup, thePlayers, testMode);
41
42        // Beginning the game.
43        equalEyeCounter = 0;
44
45        // The game should run until one player remains in the ArrayList.
46        while (thePlayers.size() > 1) {
47
48           // The for loop running through all the players.
49           for (i = 0; i < thePlayers.size(); i++) {
50
51              // If only one player is left, the 'else' runs and breaks the
52              // for loop
53              if (thePlayers.size() != 1) {
54
55                 // Creating menues for the player based on property or
56                 // status.
57                 String[] availableChoices = getMenu(thePlayers.get(i));
58                 String turn = GUIGame.getUserInputTurn(thePlayers.get(i), availableChoices);
59
60                 // All the if statements for Player choices on start of
61                 // turn.
62                 if (thePlayers.get(i).getBalance() >= 0) {
63
64                    if (turn.equals(msgL.msg(1))) {//player chose "Roll"
65                       theCup.roll();
66                       GUIGame.showDice(theCup);
67                       // All the if statements for different scenarios a
68                       // player can be in when rolling.
69
70                       if (thePlayers.get(i).getJailStatus()) {
71                          //When a player is
72                          doJail(thePlayers.get(i));
73                       } else {
74
75                          // If the dice is equals turn resets
76                          if (theCup.getEquals() == true && equalEyeCounter != 2) {
77                             doMoveVehicle(thePlayers.get(i));
78                             if (thePlayers.get(i).getJailStatus() == false) {
79                                // player gets another turn if not in
80                                // jail.
81                                i--;
82                                equalEyeCounter++;
83                             }
84                          } else if (theCup.getEquals() && equalEyeCounter == 2) {
85                             // Puts the player in jail if
86                             // equalEyeCounter hits 3.
87                             GUIControl.printMessage(msgL.msg(189));
88                             thePlayers.get(i).setPosition(10, thePlayers.get(i).getCurrentPosition());
89                             GUIControl.moveVehicle(thePlayers.get(i));
90                             thePlayers.get(i).setJailStatus(true);
91                             equalEyeCounter = 0;
92                          } else {
93                             equalEyeCounter = 0;
94                             doMoveVehicle(thePlayers.get(i));
95                             if (thePlayers.get(i).getBalance() < 0) {
96                                i--;
97                             }
98                          }
99                       }
100                      //Roll option ends.
101                   } else if (turn.equals(msgL.msg(4))) {//player chose "Pawning"
102                      Ownable pawned = null;
103                      String pawnName = GUIControl.makeLists(msgL.msg(5), thePlayers.get(i).getPawnable());
```

```
                for (int j = 0; j < thePlayers.get(i).getOwned().size(); j++) {

                   if (thePlayers.get(i).getOwned().get(j).toString().equals(pawnName)) {

                      pawned = thePlayers.get(i).getOwned().get(j);

                   }

                }

                thePlayers.get(i).pawnProperty(pawned);
                i--;
             } else if (turn.equals(msgL.msg(190))) {//player chose "Remove pawn status"
                Ownable hasPawned = null;
                String hasPawnName = GUIControl.makeLists(msgL.msg(191), thePlayers.get(i).getPawned());
                for (int j = 0; j < thePlayers.get(i).getOwned().size(); j++) {

                   if (thePlayers.get(i).getOwned().get(j).toString().equals(hasPawnName)) {
                      if(thePlayers.get(i).getBalance()>=thePlayers.get(i).getOwned().get(j).getPawn()){
                         hasPawned = thePlayers.get(i).getOwned().get(j);
                      }
                   }

                }

                if(hasPawned!=null){
                   thePlayers.get(i).liftPawn(hasPawned);
                }else{
                   GUIControl.printMessage(msgL.msg(213));
                }
                i--;
             } else if (turn.equals(msgL.msg(6))) {//player chose "Buy houses or hotels
                if (thePlayers.get(i).getBuildStatus() == true) {
                   String streetName = GUIControl.makeLists(msgL.msg(192),
                         thePlayers.get(i).getBuildableList());
                   setBuilding(thePlayers.get(i), streetName);
                } else {
                   GUIControl.printMessage(msgL.msg(193));
                }
                i--;
             } else if (turn.equals(msgL.msg(8))) {//player chose "Sell houses or hotels"
                if (thePlayers.get(i).getBuilding() == true) {
                   String streetName = GUIControl.makeLists(msgL.msg(194),
                         thePlayers.get(i).getSellableList());
                   removeBuilding(thePlayers.get(i), streetName);
                }
                i--;
             } else if (turn.equals(msgL.msg(10))) {//player chose "Give up"
                GUIGame.removePlayer(thePlayers.get(i));
                thePlayers.remove(i);
                i--; //Because the array shrinks
             } else if (turn.equals(msgL.msg(166))) {//player chose "Pay bail of 1000 kr."
                GUIControl.printMessage(msgL.msg(167));
                thePlayers.get(i).withdraw(1000);
                thePlayers.get(i).setJailStatus(false);
                i--;
             }
          } else if (thePlayers.get(i).getBalance() < 0 && thePlayers.get(i).getProperty() == true) {
             //the players options if the balance is below 0 and he still has a property.
             if (turn.equals(msgL.msg(10))) {//player chose "Give up"
                GUIGame.removePlayer(thePlayers.get(i));
                thePlayers.remove(i);
                i--;//Because the array shrinks
             }else if (thePlayers.get(i).getProperty()) {
                getMenu(thePlayers.get(i));
                if (turn.equals(msgL.msg(4))) {//player chose "Pawning"
                   Ownable pawned = null;
                   String pawnName = GUIControl.makeLists(msgL.msg(5), thePlayers.get(i).getPawnable());
                   for (int j = 0; j < thePlayers.get(i).getOwned().size(); j++) {
                      if (thePlayers.get(i).getOwned().get(j).toString().equals(pawnName)) {
                         pawned = thePlayers.get(i).getOwned().get(j);
                      }

                   }

                   thePlayers.get(i).pawnProperty(pawned);
                   if (thePlayers.get(i).getBalance() < 0) {
                      i--;
                   }
                }else if (turn.equals(msgL.msg(8))) {//player chose "Sell houses or hotels"
                   if (thePlayers.get(i).getBuilding() == true) {
                      String streetName = GUIControl.makeLists(msgL.msg(194),
                            thePlayers.get(i).getSellableList());
                      removeBuilding(thePlayers.get(i), streetName);
                   }
                   if (thePlayers.get(i).getBalance() < 0) {
                      i--;
                   }
                }
             }
          } else if (thePlayers.get(i).getBalance() < 0 && thePlayers.get(i).getProperty() == false) {
             //player is removed if he has negative balance and no properties.
             GUIGame.removePlayer(thePlayers.get(i));
             thePlayers.remove(i);
             i--; //Because the array shrinks.

          }
       } else { // Breaks the forloop because winner is found.
          break;
       }
    }
  }
  GUIGame.showWinner(thePlayers.get(0)); // Shows the winner.
  GUIGame.endGUI();

}

/**
 * Run the game in test mode or just run the game.
 */
```

```
214    private void GameOrTestMode() {
215
216       String startUp = GUIControl.make2Buttons(msgL.msg(197), msgL.msg(198), msgL.msg(199));
217       if (startUp == msgL.msg(198)) {
218          String testCase = GUIControl.make3Buttons(msgL.msg(200), msgL.msg(201), msgL.msg(202), msgL.msg(203));
219
220          if (testCase == msgL.msg(201)) {
221             theCup = new FakeCup(0);
222             startAmount = 35000;
223          } else if (testCase == msgL.msg(202)) {
224             theCup = new FakeCup(1);
225             startAmount = 30000;
226          } else if (testCase == msgL.msg(203)) {
227             theCup = new FakeCup(2);
228             startAmount = 30000;
229          }
230          testMode = true;
231
232       } else if (startUp == msgL.msg(199)) {
233          theCup = new Cup();
234          startAmount = 30000;
235          if (GUIGame.changeLanguage().equals(msgL.msg(176))) {
236             GUIGame.endGUI();
237             GUIGame.makeBoard();
238          }
239
240       }
241    }
242
243    /**
244     * Asks for number of players, their names and creates an arraylist of
245     * players.
246     */
247
248    private ArrayList<Player> createPlayers(boolean testMode, int startAmount) {
249       String[] playerNames;
250       if (testMode == false) {
251          playerNames = GUIGame.numberOfPlayers();// Ask how many players
252                                      // there are in the game.
253
254       } else {
255
256          playerNames = new String[3];
257          playerNames[0] = msgL.msg(204);
258          playerNames[1] = msgL.msg(205);
259          playerNames[2] = msgL.msg(206);
260       }
261
262       /*
263        * Creating the players. By using an array, the names are saved in each
264        * of their own index'es in the array.
265        */
266       thePlayers = new ArrayList<Player>();
267
268       // Save all the players in a ArrayList.
269
270       for (int i = 0; i < playerNames.length; i++) {
271          thePlayers.add(new Player(playerNames[i], startAmount)); // Creating
272          // player
273          // objects for the game
274          GUIGame.createPlayer(thePlayers.get(i)); // Creating Player objects
275                                         // for the GUI
276
277       }
278       return thePlayers;
279    }
280
281    /**
282     * Menues for the player based on what assets he currently holds and
283     * balance.
284     *
285     * @param theplayer
286     *           Type: Player
287     * @return options Type: String[] with player options.
288     */
289    private String[] getMenu(Player theplayer) {
290       ArrayList<String> choices = new ArrayList<>();
291
292       if (theplayer.getJailStatus() == true) {
293          choices.add(msgL.msg(166));
294       }
295       if (!(theplayer.getBalance() < 0)) {
296          choices.add(msgL.msg(1));
297       }
298       if (theplayer.getProperty()) {
299          if (theplayer.getPawnStatus() == true) {
300             choices.add(msgL.msg(4));
301          }
302          if (theplayer.hasPawned() == true) {
303             if(theplayer.getBalance()>0)
304                choices.add(msgL.msg(190));
305          }
306          if (theplayer.getBuilding() == true) {
307             choices.add(msgL.msg(8));
308          }
309          if (theplayer.getBuildStatus() == true) {
310             choices.add(msgL.msg(6));
311          }
312       }
313       choices.add(msgL.msg(10));
314
315       // }
316
317       return choices.toArray(new String[choices.size()]); // Converting the
318                                            // Arraylist to
319       // an Array.
320
321    }
322
323    /**
```

```
    * Moving vehicle.
    *
    * @param theplayer
    *           type: Player
    */

   private void doMoveVehicle(Player theplayer) {
      theplayer.moveVehicle(theCup.getSum());
      GUIControl.moveVehicle(theplayer);
      int newPosition = theplayer.getCurrentPosition();
      theBoard.getSquare(newPosition).landOnSquare(theplayer);
   }

   /**
    * Method for a turn while in prison.
    *
    * @param theplayer
    *           type: Player.
    */
   private void doJail(Player theplayer) {
      if (theCup.getEquals()) {
         GUIControl.printMessage(msgL.msg(170));
         doMoveVehicle(theplayer);
         theplayer.setJailStatus(false);
         i--;
      } else if (theplayer.getJailCounter() == 2) {
         GUIControl.printMessage(msgL.msg(171));
         theplayer.withdraw(1000);
         theplayer.setJailStatus(false);
         doMoveVehicle(theplayer);
      } else {
         GUIControl.printMessage(msgL.msg(172));
         theplayer.addToJailCounter();
         // der sker ikke noget.
      }
      equalEyeCounter = 0;
   }

   /**
    * Sets a building on the Street and in the GUI.
    *
    * @param thePlayer
    * @param streetName
    */
   private void setBuilding(Player thePlayer, String streetName) {
      int i, position = 0, numberOfBuildings = 0;
      Street theStreet = null;

      for (i = 0; i < thePlayer.getOwnedStreet().size(); i++) {
         if (thePlayer.getOwnedStreet().get(i).toString().equals(streetName)) {
            position = thePlayer.getOwnedStreet().get(i).getID();
            numberOfBuildings = thePlayer.getOwnedStreet().get(i).getNumberOfBuildings();
            theStreet = thePlayer.getOwnedStreet().get(i);
         }
      }
      GUIGame.setBuilding(position, numberOfBuildings);
      thePlayer.buyBuildings(theStreet, 1);

   }

   /**
    * Removes a building from the Street and the player.
    *
    * @param thePlayer
    * @param streetName
    */
   private void removeBuilding(Player thePlayer, String streetName) {
      int i, position = 0, numberOfBuildings = 0;
      Street theStreet = null;

      for (i = 0; i < thePlayer.getOwnedStreet().size(); i++) {
         if (thePlayer.getOwnedStreet().get(i).toString().equals(streetName)) {
            theStreet = thePlayer.getOwnedStreet().get(i);
            position = thePlayer.getOwnedStreet().get(i).getID();
            numberOfBuildings = thePlayer.getOwnedStreet().get(i).getNumberOfBuildings();
         }

      }
      GUIGame.removeBuilding(position, numberOfBuildings);
      thePlayer.removeBuildings(theStreet, 1);

   }
}
```

Listing 5: GUIControl

```
package controller;

import java.awt.Color;
import java.util.ArrayList;

//import board.Tax;
//import board.Refuge;
import board.Ownable;
//import desktop_fields.Refuge;
//import desktop_fields.Tax;
import desktop_codebehind.Car;
//import board.*;
import desktop_fields.Brewery;
import desktop_fields.Chance;
import desktop_fields.Field;
import desktop_fields.Jail;
import desktop_fields.Refuge;
import desktop_fields.Shipping;
import desktop_fields.Start;
import desktop_fields.Street;
```

```java
import desktop_fields.Tax;
import desktop_resources.GUI;
import entities.Cup;
import entities.Player;

/**
 * Controller class which handles all contact with the GUI.
 *
 */
public class GUIControl {
   /**
    * Makes the visual board.
    */
   public void makeBoard() { // Method that creates the board for the GUI and
                     // sets the squares with their descriptions,
                     // colouring and number
      // Street colors
      Field[] fields = new Field[40];
      fields[0] = new Start.Builder().setTitle(msgL.msg(11)).setDescription(msgL.msg(12)).setSubText(msgL.msg(173))
            .setBgColor(Color.getHSBColor(241, 196, 15)).setFgColor(Color.black).build();
      fields[1] = new Street.Builder().setTitle(msgL.msg(13)).setDescription(msgL.msg(14)).setSubText(msgL.msg(15))
            .setBgColor(Color.blue).setFgColor(Color.black).build();
      fields[2] = new Chance.Builder().setBgColor(Color.getHSBColor(241, 196, 15)).setFgColor(Color.black).build();
      fields[3] = new Street.Builder().setTitle(msgL.msg(16)).setDescription(msgL.msg(17)).setSubText(msgL.msg(15))
            .setBgColor(Color.blue).setFgColor(Color.black).build();
      fields[4] = new Tax.Builder().setTitle(msgL.msg(18)).setDescription(msgL.msg(19)).setSubText(msgL.msg(19))
            .setBgColor(Color.getHSBColor(241, 196, 15)).setFgColor(Color.black).build();
      fields[5] = new Shipping.Builder().setTitle(msgL.msg(20)).setDescription(msgL.msg(21)).setSubText(msgL.msg(22))
            .setBgColor(Color.white).setFgColor(Color.black).build();
      fields[6] = new Street.Builder().setTitle(msgL.msg(23)).setDescription(msgL.msg(24)).setSubText(msgL.msg(25))
            .setBgColor(Color.getHSBColor(127, 140, 141)).setFgColor(Color.black).build();
      fields[7] = new Chance.Builder().setBgColor(Color.getHSBColor(241, 196, 15)).setFgColor(Color.black).build();
      fields[8] = new Street.Builder().setTitle(msgL.msg(26)).setDescription(msgL.msg(27)).setSubText(msgL.msg(28))
            .setBgColor(Color.getHSBColor(127, 140, 141)).setFgColor(Color.black).build();
      fields[9] = new Street.Builder().setTitle(msgL.msg(29)).setDescription(msgL.msg(30)).setSubText(msgL.msg(31))
            .setBgColor(Color.getHSBColor(127, 140, 141)).setFgColor(Color.black).build();
      fields[10] = new Jail.Builder().setTitle(msgL.msg(32)).setDescription(msgL.msg(33)).setSubText(msgL.msg(32))
            .setBgColor(Color.getHSBColor(241, 196, 15)).setFgColor(Color.black).build();
      fields[11] = new Street.Builder().setTitle(msgL.msg(34)).setDescription(msgL.msg(35)).setSubText(msgL.msg(36))
            .setBgColor(Color.green).setFgColor(Color.black).build();
      fields[12] = new Brewery.Builder().setTitle(msgL.msg(37)).setDescription(msgL.msg(38)).setSubText(msgL.msg(39))
            .setBgColor(Color.white).setFgColor(Color.black).build();
      fields[13] = new Street.Builder().setTitle(msgL.msg(40)).setDescription(msgL.msg(41)).setSubText(msgL.msg(42))
            .setBgColor(Color.green).setFgColor(Color.black).build();
      fields[14] = new Street.Builder().setTitle(msgL.msg(43)).setDescription(msgL.msg(44)).setSubText(msgL.msg(45))
            .setBgColor(Color.green).setFgColor(Color.black).build();
      fields[15] = new Shipping.Builder().setTitle(msgL.msg(46)).setDescription(msgL.msg(47)).setSubText(msgL.msg(22))
            .setBgColor(Color.white).setFgColor(Color.black).build();
      fields[16] = new Street.Builder().setTitle(msgL.msg(49)).setDescription(msgL.msg(50)).setSubText(msgL.msg(54))
            .setBgColor(Color.gray).setFgColor(Color.black).build();
      fields[17] = new Chance.Builder().setBgColor(Color.getHSBColor(241, 196, 15)).setFgColor(Color.black).build();
      fields[18] = new Street.Builder().setTitle(msgL.msg(52)).setDescription(msgL.msg(50)).setSubText(msgL.msg(54))
            .setBgColor(Color.gray).setFgColor(Color.black).build();
      fields[19] = new Street.Builder().setTitle(msgL.msg(55)).setDescription(msgL.msg(56)).setSubText(msgL.msg(48))
            .setBgColor(Color.gray).setFgColor(Color.black).build();
      fields[20] = new Refuge.Builder().setTitle(msgL.msg(58)).setDescription(msgL.msg(59)).setSubText(msgL.msg(60))
            .setBgColor(Color.white).setFgColor(Color.black).build();
      fields[21] = new Street.Builder().setTitle(msgL.msg(61)).setDescription(msgL.msg(56)).setSubText(msgL.msg(63))
            .setBgColor(Color.getHSBColor(224, 130, 131)).setFgColor(Color.black).build();
      fields[22] = new Chance.Builder().setBgColor(Color.getHSBColor(241, 196, 15)).setFgColor(Color.black).build();
      fields[23] = new Street.Builder().setTitle(msgL.msg(64)).setDescription(msgL.msg(65)).setSubText(msgL.msg(63))
            .setBgColor(Color.getHSBColor(224, 130, 131)).setFgColor(Color.black).build();
      fields[24] = new Street.Builder().setTitle(msgL.msg(67)).setDescription(msgL.msg(68)).setSubText(msgL.msg(69))
            .setBgColor(Color.getHSBColor(224, 130, 131)).setFgColor(Color.black).build();
      fields[25] = new Shipping.Builder().setTitle(msgL.msg(70)).setDescription(msgL.msg(71)).setSubText(msgL.msg(48))
            .setBgColor(Color.white).setFgColor(Color.black).build();
      fields[26] = new Street.Builder().setTitle(msgL.msg(73)).setDescription(msgL.msg(44)).setSubText(msgL.msg(75))
            .setBgColor(Color.white).setFgColor(Color.black).build();
      fields[27] = new Street.Builder().setTitle(msgL.msg(76)).setDescription(msgL.msg(74)).setSubText(msgL.msg(78))
            .setBgColor(Color.white).setFgColor(Color.black).build();
      fields[28] = new Brewery.Builder().setTitle(msgL.msg(79)).setDescription(msgL.msg(80)).setSubText(msgL.msg(81))
            .setBgColor(Color.white).setFgColor(Color.black).build();
      fields[29] = new Street.Builder().setTitle(msgL.msg(82)).setDescription(msgL.msg(83)).setSubText(msgL.msg(84))
            .setBgColor(Color.white).setFgColor(Color.black).build();
      fields[30] = new Jail.Builder().setTitle(msgL.msg(85)).setDescription(msgL.msg(86)).setSubText(msgL.msg(86))
            .setBgColor(Color.getHSBColor(241, 196, 15)).setFgColor(Color.black).build();
      fields[31] = new Street.Builder().setTitle(msgL.msg(87)).setDescription(msgL.msg(88)).setSubText(msgL.msg(89))
            .setBgColor(Color.yellow).setFgColor(Color.black).build();
      fields[32] = new Street.Builder().setTitle(msgL.msg(90)).setDescription(msgL.msg(91)).setSubText(msgL.msg(92))
            .setBgColor(Color.yellow).setFgColor(Color.black).build();
      fields[33] = new Chance.Builder().setBgColor(Color.getHSBColor(241, 196, 15)).setFgColor(Color.black).build();
      fields[34] = new Street.Builder().setTitle(msgL.msg(93)).setDescription(msgL.msg(94)).setSubText(msgL.msg(95))
            .setBgColor(Color.yellow).setFgColor(Color.black).build();
      fields[35] = new Shipping.Builder().setTitle(msgL.msg(96)).setDescription(msgL.msg(47))
            .setBgColor(Color.white).setFgColor(Color.black).build();
      fields[36] = new Chance.Builder().setBgColor(Color.getHSBColor(241, 196, 15)).setFgColor(Color.black).build();
      fields[37] = new Street.Builder().setTitle(msgL.msg(97)).setDescription(msgL.msg(98)).setSubText(msgL.msg(99))
            .setBgColor(Color.orange).setFgColor(Color.black).build();
      fields[38] = new Tax.Builder().setTitle("").setDescription(msgL.msg(100)).setSubText(msgL.msg(100))
            .setBgColor(Color.getHSBColor(241, 196, 15)).setFgColor(Color.black).build();
      fields[39] = new Street.Builder().setTitle(msgL.msg(101)).setDescription(msgL.msg(102))
            .setSubText(msgL.msg(103)).setBgColor(Color.orange).setFgColor(Color.black).build();

      GUI.create(fields);
   }
   /**
    * Method for choosing language of the messages.
    * @return language chosen
    */
   public String changeLanguage() {
      String language = GUI.getUserButtonPressed(msgL.msg(174), msgL.msg(175), msgL.msg(176));
      msgL.changeLanguage(language);
      return language;
   }
   /**
    * Shows the winner in the GUI
    * @param winner to be announced
    */
   public void showWinner(Player winner) {
      GUI.showMessage(msgL.msg(104) + winner + msgL.msg(207) + msgL.msg(105));
```

```
131     }
132     /**
133      * Creates a player on the board.
134      * @param newPlayer to enter
135      */
136     public void createPlayer(Player newPlayer) {
137
138        // Creating the car.
139        Car car = new Car.Builder().typeUfo().patternHorizontalDualColor().primaryColor(Color.WHITE)
140              .secondaryColor(newPlayer.getColor()).build();
141        // Creating player on board.
142        GUI.addPlayer(newPlayer.toString(), newPlayer.getBalance(), car);
143        GUI.setCar(1, newPlayer.toString());
144     }
145
146     /**
147      * Visual representation of the dices.
148      * @param newCup reference
149      */
150     public void showDice(Cup newCup) {
151        int d1 = newCup.getD1();
152        int d2 = newCup.getD2();
153        if (d1 > 6 || d2 > 6 && d2 < 0 || d2 < 0) //in test mode som dicevalues will exceed 6
154                                    //which isn't compatible with the GUI.
155           GUI.setDice(1, 1);
156        else
157           GUI.setDice(d1, d2);
158     }
159
160     /**
161      * Getting a string array with the names of the players.
162      * @return playerNames String[]
163      */
164     public String[] numberOfPlayers() {
165        int numberOfPlayers = GUI.getUserInteger(msgL.msg(106), 3, 6);
166        String[] playerNames = new String[numberOfPlayers];
167        String[] tempArray = new String[numberOfPlayers];
168        for (int u = 0; u < numberOfPlayers; u++) {
169           tempArray[u] = msgL.msg(107);
170           playerNames[u] = msgL.msg(107) + u;
171        }
172        String tempName;
173        boolean sameName = false;
174        int number;
175
176        for (int i = 0; i < numberOfPlayers; i++) {
177           number = i + 1; // The Array[] have to start at index 0, but the
178                           // player is number 1.
179           sameName = false;
180           tempName = GUI.getUserString(msgL.msg(108) + String.valueOf(number));
181           for (int j = 0; j < numberOfPlayers; j++) {
182              if (tempArray[j].equals(tempName)) {
183                 sameName = true;
184                 i--;
185                 GUI.showMessage(msgL.msg(109));
186              }
187           }
188           if (sameName == false) {
189              playerNames[i] = tempName;
190              tempArray[i] = tempName;
191           }
192        }
193        return playerNames;
194     }
195
196     /**
197      *
198      * @param thePlayer in question
199      * @param choices in String[]
200      * @return input
201      */
202     public String getUserInputTurn(Player thePlayer, String[] choices) {
203        String input;
204        input = GUI.getUserButtonPressed(thePlayer.toString() + msgL.msg(110), choices);
205        return input;
206     }
207
208     /**
209      * Moves vehicle on the board
210      *
211      * @param thePlayer
212      *           type: Player
213      */
214     public static void moveVehicle(Player thePlayer) {
215
216        int value = thePlayer.getCurrentPosition() - thePlayer.getPreviousPosition();
217        while (value < 0)
218           value += 40;
219
220        for (int i = 0; i < value; i++) {//Sets the player on one spot after another to simulate a piece moving on the board
221           GUI.removeCar((thePlayer.getPreviousPosition() + i) % 40 + 1, thePlayer.toString());
222           GUI.setCar((thePlayer.getPreviousPosition() + i + 1) % 40 + 1, thePlayer.toString());
223           if ((thePlayer.getPreviousPosition() + i + 1) % 40 + 1 == 2) {//if the player has passed the Start field
224              if (thePlayer.getFirstRound() == false && thePlayer.getCurrentPosition() != 10) {
225                 thePlayer.deposit(4000);
226              } else {
227                 thePlayer.setFirstRound(false);
228              }
229           }
230           try {
231              Thread.currentThread();
232              if (value > 12) {
233                 Thread.sleep(0);//for test modes, the vehicle teleports.
234              } else {
235                 Thread.sleep(150);//the thread sleeps for 150 miliseconds
236              }
237           } catch (InterruptedException e) {
238              e.printStackTrace();
239           }
240        }
```

```
241     }
242
243     /**
244      * Player choice of buying the square he landed on.
245      * @param field the player landed on
246      * @param player in question
247      * @return boolean
248      */
249     public static boolean getBuyChoice(Ownable field, Player player) {
250
251        String input = GUI.getUserButtonPressed(player.toString() + msgL.msg(111) + field.toString() + msgL.msg(177)
252              + field.toString() + msgL.msg(178) + field.getPrice() + msgL.msg(181), msgL.msg(179), msgL.msg(180));
253        if (input.equals(msgL.msg(179)))
254           return true;
255        else
256           return false;
257     }
258     /**
259      * Player chooses which way he wants to pay taxes.
260      * @param name of the Tax field.
261      * @param player to get choice
262      * @return boolean
263      */
264     public static boolean getTaxChoice(String name, Player player) {
265
266        String input = GUI.getUserButtonPressed(player.toString() + msgL.msg(111) + name + msgL.msg(182), msgL.msg(173),
267              msgL.msg(183));
268        if (input.equals(msgL.msg(173)))
269           return true;
270        else
271           return false;
272     }
273
274     /**
275      * Prints message in GUI
276      *
277      * @param message
278      *           type: String
279      */
280     public static void printMessage(String message) {
281        GUI.showMessage(message);
282     }
283
284     /**
285      * Removing player from playing board when player surrenders or looses.
286      * @param thePlayer to be removed
287      */
288     public void removePlayer(Player thePlayer) {
289
290        // Remove the players owned squares.
291        int[] list = thePlayer.getOwnedID();
292        ArrayList<Ownable> arr = thePlayer.getOwned();
293        for (int i = 0; i < list.length; i++) {
294           GUI.removeOwner(list[i]);
295           GUI.setHouses(arr.get(i).getID(), 0);
296           arr.get(i).liftPawn();
297           arr.get(i).clearOwner();
298        }
299        // Remove Car
300        GUI.removeCar(thePlayer.getCurrentPosition() + 1, thePlayer.toString());
301        GUI.setBalance(thePlayer.toString(), 0);
302     }
303
304     /**
305      * Marks a square as owned buy a player.
306      * @param squareNumber of the Ownable
307      * @param thePlayer to own
308      */
309     public static void setOwned(int squareNumber, Player thePlayer) {
310
311        GUI.setOwner(squareNumber, thePlayer.toString());
312     }
313     /**
314      * Updates the balance of the player in the GUI
315      * @param player in question
316      */
317     public static void updateBalance(Player player) {
318        GUI.setBalance(player.toString(), player.getBalance());
319     }
320     /**
321      * Closes the GUI.
322      */
323     public void endGUI() {
324        GUI.close();
325     }
326     /**
327      * Sets a number of buildings on the specified square.
328      * @param position of the Street
329      * @param numberOfBuildings to be set
330      */
331     public void setBuilding(int position, int numberOfBuildings) {
332
333        if (numberOfBuildings == 4) {
334           GUI.setHotel(position, true);
335        } else if (numberOfBuildings >= 0) {
336           GUI.setHouses(position, numberOfBuildings + 1);
337        } else {
338           GUI.showMessage(msgL.msg(195));// error message is displayed.
339        }
340     }
341     /**
342      * Removes a number of houses from a street.
343      * @param position of the Street
344      * @param numberOfBuildings to be removed
345      */
346     public void removeBuilding(int position, int numberOfBuildings) {
347        GUI.setHouses(position, numberOfBuildings - 1);
348     }
349     /**
350      * Makes two buttons and returns a string representation of what was pressed.
```

```
351     * @param message to be printed
352     * @param button1 text
353     * @param button2 text
354     * @return button pressed
355     */
356    public static String make2Buttons(String message, String button1, String button2) {
357
358       return GUI.getUserButtonPressed(message, button1, button2);
359    }
360    /**
361     * Makes three buttons and returns a string representation of what was pressed.
362     * @param message to be printed
363     * @param button1 text
364     * @param button2 text
365     * @param button3 text
366     * @return button pressed
367     */
368    public static String make3Buttons(String message, String button1, String button2, String button3) {
369       return GUI.getUserButtonPressed(message, button1, button2, button3);
370    }
371    /**
372     * Makes a list for the player to choose from.
373     * @param message to be printed
374     * @param options to choose from
375     * @return selection String
376     */
377    public static String makeLists(String message, String[] options) {
378       return GUI.getUserSelection(message, options);
379    }
380    /**
381     * Displays a text in the chanceCard field in the middle.
382     * @param txt of the chance card
383     */
384    public static void displayChanceCard(String txt) {
385       GUI.displayChanceCard(txt);
386    }
387 }
```

Listing 6: msgL

```
1  package controller;
2
3  import java.io.File;
4  import java.io.FileNotFoundException;
5  import java.util.Scanner;
6
7  public class msgL {
8     private static String[] info_danish = new String[1000];
9     private static String[] info_english = new String[1000];
10    static String lang="Dansk";
11
12    //the static block runs as soon as a method in this class has been initiated.
13    static {
14       String fileName = "language2";
15       File file = new File(fileName);
16       String tempString;
17       int Danish = 1;
18       int English = 2;
19       int i = 0;
20       int j = 0;
21       try {
22          Scanner inputStream = new Scanner(file);
23          while (inputStream.hasNextLine()) {
24             tempString=inputStream.nextLine();//stores the next line in a temporary string
25             info_danish[i++] = tempString.split("\t")[Danish];//splits the text by tabs
26             info_english[j++] = tempString.split("\t")[English];
27          }
28          inputStream.close();
29
30       } catch (FileNotFoundException e) {
31          System.out.println("Forkert filnavn");
32       }
33    }
34    /**
35     * Getter for the String arrays with messages.
36     * @param index of message
37     * @return string
38     */
39    public static String msg(int index){
40       if(lang.equals("English")){
41          return info_english[index];
42       }
43       else{
44          return info_danish[index];
45       }
46    }
47    /**
48     * Changes the language of the strings msg() returns
49     * @param language chosen
50     */
51    public static void changeLanguage(String language){
52       if(language.equals("English")){
53          lang="English";
54       }else if(language.equals("Dansk")){
55          lang="Dansk";
56       }
57    }
58 }
```

Listing 7: StartGame

```
1  package controller;
```

```
2
3  public class StartGame {
4
5     public static void main(String[] args) {
6        @SuppressWarnings("unused")
7        GameLogic startGame = new GameLogic();//simply starts the game.
8     }
9  }
```

11.2 Package entities

Listing 8: Board

```
1  package entities;
2
3  import java.util.ArrayList;
4  import board.*;
5  import board.Tax;
6  import controller.msgL;
7  import board.Parking;
8
9  /**
10  * Keeps track of all the squares, in an array
11  *
12  */
13 public class Board {
14    Square[] squares = new Square[40];
15    private AllCards allTheCards;
16
17    /**
18     * Constructor for a Board
19     * @param theCup of the game
20     * @param thePlayers of the game
21     * @param testMode value of test mode
22     */
23    public Board(Cup theCup, ArrayList<Player> thePlayers, boolean testMode) {
24
25       allTheCards = new AllCards(thePlayers, this);
26
27       if (testMode == false)
28          allTheCards.shuffle();
29
30       squares[0] = new Start(msgL.msg(11), 1);
31       squares[1] = new Street(msgL.msg(13), 2, 1200, 600, 1000, 50, 250, 750, 2250, 4000, 6000, 'A');
32       squares[2] = new Chance(msgL.msg(208), 3, allTheCards);
33       squares[3] = new Street(msgL.msg(16), 4, 1200, 600, 1000, 50, 250, 750, 2250, 4000, 6000, 'A');
34       squares[4] = new Tax(msgL.msg(18), 5, 4000);
35       squares[5] = new Shipping(msgL.msg(209), 6);
36       squares[6] = new Street(msgL.msg(23), 7, 2000, 1000, 1000, 100, 600, 1800, 5400, 8000, 11000, 'B');
37       squares[7] = new Chance(msgL.msg(208), 8, allTheCards);
38       squares[8] = new Street(msgL.msg(26), 9, 2000, 1000, 1000, 100, 600, 1800, 5400, 8000, 11000, 'B');
39       squares[9] = new Street(msgL.msg(29), 10, 2400, 1200, 1000, 150, 800, 2000, 6000, 9000, 12000, 'B');
40       squares[10] = new Parking(msgL.msg(33), 11);
41       squares[11] = new Street(msgL.msg(34), 12, 2800, 1400, 2000, 200, 1000, 3000, 9000, 12500, 15000, 'C');
42       squares[12] = new Brewery(msgL.msg(37), 13, theCup);
43       squares[13] = new Street(msgL.msg(40), 14, 2800, 1400, 2000, 200, 1000, 3000, 9000, 12500, 15000, 'C');
44       squares[14] = new Street(msgL.msg(43), 15, 3200, 1800, 2000, 250, 1250, 3750, 10000, 14000, 18000, 'C');
45       squares[15] = new Shipping(msgL.msg(46), 16);
46       squares[16] = new Street(msgL.msg(49), 17, 3600, 1800, 2000, 300, 1400, 4000, 11000, 15000, 19000, 'D');
47       squares[17] = new Chance(msgL.msg(208), 18, allTheCards);
48       squares[18] = new Street(msgL.msg(52), 19, 3600, 1800, 2000, 300, 1400, 4000, 11000, 15000, 19000, 'D');
49       squares[19] = new Street(msgL.msg(55), 20, 4000, 2000, 2000, 350, 1600, 4400, 12000, 16000, 20000, 'D');
50       squares[20] = new Parking(msgL.msg(59), 21);
51       squares[21] = new Street(msgL.msg(61), 22, 4400, 2200, 3000, 350, 1800, 5000, 14000, 17500, 21000, 'E');
52       squares[22] = new Chance(msgL.msg(208), 23, allTheCards);
53       squares[23] = new Street(msgL.msg(64), 24, 4400, 2200, 3000, 350, 1800, 5000, 14000, 17500, 21000, 'E');
54       squares[24] = new Street(msgL.msg(67), 25, 4800, 2400, 3000, 400, 2000, 6000, 15000, 18500, 22000, 'E');
55       squares[25] = new Shipping(msgL.msg(210), 26);
56       squares[26] = new Street(msgL.msg(73), 27, 5200, 2600, 3000, 450, 2200, 6600, 16000, 19500, 23000, 'F');
57       squares[27] = new Street(msgL.msg(76), 28, 5200, 2600, 3000, 450, 2200, 6600, 16000, 19500, 23000, 'F');
58       squares[28] = new Brewery(msgL.msg(79), 29, theCup);
59       squares[29] = new Street(msgL.msg(82), 30, 5600, 2800, 3000, 500, 2400, 7200, 17000, 20500, 24000, 'F');
60       squares[30] = new Jail(msgL.msg(85), 31);
61       squares[31] = new Street(msgL.msg(87), 32, 6000, 3000, 4000, 550, 2600, 7800, 18000, 22000, 25000, 'G');
62       squares[32] = new Street(msgL.msg(90), 33, 6000, 3000, 4000, 550, 2600, 7800, 18000, 22000, 25000, 'G');
63       squares[33] = new Chance(msgL.msg(208), 34, allTheCards);
64       squares[34] = new Street(msgL.msg(93), 35, 6400, 3200, 4000, 600, 3000, 9000, 20000, 24000, 28000, 'G');
65       squares[35] = new Shipping(msgL.msg(211), 36);
66       squares[36] = new Chance(msgL.msg(208), 37, allTheCards);
67       squares[37] = new Street(msgL.msg(97), 38, 7000, 3500, 4000, 700, 3500, 10000, 22000, 26000, 30000, 'H');
68       squares[38] = new Tax(msgL.msg(100), 39, 2000);
69       squares[39] = new Street(msgL.msg(101), 40, 8000, 4000, 4000, 1000, 4000, 12000, 28000, 34000, 40000, 'H');
70
71    }
72
73    /**
74     * Method for returning a square from the array in this instance.
75     *
76     * @param index message
77     * @return Square
78     */
79    public Square getSquare(int index) {
80       return squares[index];
81    }
82
83 }
```

Listing 9: AllCards

```java
package entities;
import java.util.ArrayList;

import java.util.Random;

import cards.Card;
import cards.ChangePosition;
import cards.Expense;
import cards.GoToJail;
import cards.Grant;
import cards.IncomeIncrease;
import cards.MoveToSquare;
import cards.MoveToShip;
import cards.Pardon;
import cards.PlayerTransaction;
import cards.PriceIncrease;
import controller.msgL;
/**
 * Class for holding all Cards
 *
 */
public class AllCards {
   Card theCards[];

   /**
    * Creating an AllCards instance
    * @param thePlayers of the game
    * @param theBoard of the game
    */
   public AllCards(ArrayList<Player> thePlayers, Board theBoard) {
      theCards = new Card[44];

      theCards[0] = new Pardon(msgL.msg(156));
      theCards[1] = new GoToJail(msgL.msg(147));
      theCards[2] = new PlayerTransaction(msgL.msg(132), 200, thePlayers);
      theCards[3] = new ChangePosition(msgL.msg(133), 3, theBoard);
      theCards[4] = new IncomeIncrease(msgL.msg(134),1000);
      theCards[5] = new IncomeIncrease(msgL.msg(135), 1000);
      theCards[6] = new Expense(msgL.msg(136), 300);
      theCards[7] = new IncomeIncrease(msgL.msg(137), 1000);
      theCards[8] = new IncomeIncrease(msgL.msg(137), 1000);
      theCards[9] = new Grant(msgL.msg(138), 40000);
      theCards[10] = new MoveToSquare(msgL.msg(139), 0, theBoard);
      theCards[11] = new MoveToSquare(msgL.msg(139), 0, theBoard);
      theCards[12] = new PlayerTransaction(msgL.msg(140), 500, thePlayers);
      theCards[13] = new MoveToSquare(msgL.msg(141), 39 ,theBoard);
      theCards[14] = new MoveToSquare(msgL.msg(142),15,theBoard);
      theCards[15] = new Expense(msgL.msg(143), 1000);
      theCards[16] = new Expense(msgL.msg(144), 200);
      theCards[17] = new Expense(msgL.msg(145), 200);
      theCards[18] = new MoveToShip(msgL.msg(146),theBoard);
      theCards[19] = new MoveToSquare(msgL.msg(131), 11, theBoard);
      theCards[20] = new GoToJail(msgL.msg(147));
      theCards[21] = new PriceIncrease(msgL.msg(148), 500, 2000);
      theCards[22] = new IncomeIncrease(msgL.msg(149), 1000);
      theCards[23] = new IncomeIncrease(msgL.msg(149), 1000);
      theCards[24] = new MoveToShip(msgL.msg(150),theBoard);
      theCards[25] = new MoveToShip(msgL.msg(150), theBoard);
      theCards[26] = new MoveToSquare(msgL.msg(186), 19 ,theBoard);
      theCards[27] = new Expense(msgL.msg(151), 1000);
      theCards[28] = new IncomeIncrease(msgL.msg(152), 4000);
      theCards[29] = new IncomeIncrease(msgL.msg(153), 500);
      theCards[30] = new IncomeIncrease(msgL.msg(153), 500);
      theCards[31] = new MoveToSquare(msgL.msg(154), 32,theBoard);
      theCards[32] = new ChangePosition(msgL.msg(155), 3,theBoard);
      theCards[33] = new ChangePosition(msgL.msg(155), 3,theBoard);
      theCards[34] = new PriceIncrease(msgL.msg(130), 800, 2300);
      theCards[35] = new Pardon(msgL.msg(156));
      theCards[36] = new Expense(msgL.msg(157), 3000);
      theCards[37] = new Expense(msgL.msg(157), 3000);
      theCards[38] = new IncomeIncrease(msgL.msg(158), 1000);
      theCards[39] = new Expense(msgL.msg(159), 200);
      theCards[40] = new MoveToSquare(msgL.msg(160), 24,theBoard);
      theCards[41] = new IncomeIncrease(msgL.msg(161), 3000);
      theCards[42] = new PlayerTransaction(msgL.msg(162), 500, thePlayers);
      theCards[43] = new Expense(msgL.msg(163), 1000);
   }
   /**
    * Method which takes an int as a parameter
    * and returns that index from the 'theCards' array of this instance
    * @param index [0:43]
    * @return Card
    */
   public Card getCard(int index) {
      return theCards[index];
   }
   /**
    * Shuffles the cards
    */
   public void shuffle(){
      Random r = new Random();

      for(int last = theCards.length-1; last > 0; last--){
         int i = r.nextInt(last+1);
         Card tmp = theCards[last];
         theCards[last] = theCards[i];
         theCards[i] = tmp;
      }
   }
}
```

Listing 10: Cup

```java
package entities;

/**
```

```
 * Class Cup, for operating two Die at once.
 *
 */
public class Cup {
   private Dice d1 = new Dice(); // initiating dices to use in the cup.
   private Dice d2 = new Dice();

   /**
    * Method for rolling the dices
    *
    * @return sum of the two dices
    */
   public int roll() {
      return d1.roll() + d2.roll();

   }

   /**
    * Method for getting the result of the roll.
    *
    * @return sum of the two dices.
    */
   public int getSum() {
      return d1.getValue() + d2.getValue();
   }

   /**
    * Method for identifying if the two die show the same value.
    *
    * @return true if the two dices shows the same eyes.
    */
   public boolean getEquals() {
      boolean res = false;
      if (d1.getValue() == d2.getValue()) {
         res = true;
      }
      return res;
   }

   /**
    * Method for getting the value of dice 1.
    *
    * @return value of dice 1.
    */
   public int getD1() {
      return d1.getValue();
   }

   /**
    * Method for getting the value of dice 2.
    *
    * @return value of dice 2.
    */
   public int getD2() {
      return d2.getValue();
   }
}
```

Listing 11: Dice

```
package entities;
import java.util.Random;
/**
 * Class Dice, for getting random value between 1 and 6.
 *
 */
public class Dice {
   protected int value;
   protected int eyes;

   /*
    * A constructor with a defualt of 6 sides.
    */
   public Dice(){
      this.eyes=6;
   }

   /*
    * A constructor which makes a dice with n sides.
    */
   public Dice(int eyes) {
      if (eyes > 0) {
         this.eyes = eyes;
      }
      else{
         this.eyes=6;
      }
   }

   /**
    * Method for rolling the die.
    * @return value of dice (from 1 to eyes).
    */
   public int roll() {
      Random dice = new Random();
      int res = dice.nextInt(eyes) + 1;
      value = res;
      return res;
   }
   /**
    * Method for getting the current value of the dice.
    * @return value of the dice.
    */
   public int getValue() {
      return value;
   }
```

```
}
```

Listing 12: Player

```
package entities;

import java.awt.Color;

import java.util.*;

import board.Ownable;
import board.Street;
import controller.GUIControl;


public class Player {

   private String name;
   private Account balance = new Account();
   private Vehicle vehicle;
   private Assets assets = new Assets(this);
   private int jailCounter;
   private boolean jailStatus;
   private boolean firstRound=true;

   /**
    * Constructor for a Player, that initiates the player with an account balance, a vehicle, a jail status and a jail counter
    *
    * @param name String with the player name
    * @param balance Int with the player's starting balance
    */
   public Player(String name, int balance) {
      this.name = name;
      this.balance.deposit(balance);
      vehicle = new Vehicle();
      jailStatus = false;
      jailCounter = 0;

   }

   /**
    * Method for getting an integer array of the ID of the Ownable Squares this
    * player owns.
    *
    * @return array of integers containing the ID of owned squares.
    */
   public int[] getOwnedID() {
      return assets.getOwnedID();
   }
   /**
    *
    * @return ArrayList of the streets, this player owns.
    */
   public ArrayList<Street> getOwnedStreet() {

      return assets.getOwnedStreet();

   }

   /**
    * Method for buying an Ownable Square.
    *
    * @param square in question
    */
   public void buySquare(Ownable square) {
      assets.buySquare(square);
   }

   /**
    * Method for adding to this players jailCounter.
    */
   public void addToJailCounter() {
      jailCounter++;
   }

   /**
    * Method for resetting this players jailCounter.
    */
   public void resetJailCounter() {
      jailCounter = 0;
   }

   /**
    * getter method for the players jailCounter.
    *
    * @return jailCounter
    */
   public int getJailCounter() {
      return jailCounter;
   }
   /**
    * Setting the value of firstRound
    * @param b value of firstRound
    */
   public void setFirstRound(boolean b){
      firstRound=b;
   }
   /**
    *
    * @return a boolean value of whether this is the players first round.
    */
   public boolean getFirstRound(){
      return firstRound;
   }
   /**
    * Method for returning and ArrayList of the squares this player owns.
```

```
     *
     * @return ArrayList of ownables
     */
    public ArrayList<Ownable> getOwned() {
       return assets.getOwned();
    }

    /**
     * Method for setting the jail status of a player
     *
     * @param jailStatus of the player
     */
    public void setJailStatus(boolean jailStatus) {
       this.jailStatus = jailStatus;
    }

    /**
     * Method for checking the jail status of a player
     *
     * @return Boolean value true or false depending on the jail status of the
     *         player
     */
    public boolean getJailStatus() {
       return jailStatus;
    }

    /**
     * Method that returns whether or not the player has a get out of jail free
     * card
     *
     * @return Boolean value true or false depending on whether the player has a
     *         card
     */
    public boolean getJailCard() {
       return assets.getJailCard();
    }

    /**
     * Method for using a get out of jail free card
     */
    public void useJailCard() {
       assets.useJailCard();
    }

    /**
     * Method for receiving a get out of jail free card
     */
    public void setJailCard() {
       assets.setJailCard();
    }

    /**
     * Method for returning the name of the player
     *
     * @return Player name of the type string
     */
    public String toString() {
       return name;
    }

    /**
     * Method for depositing an amount from the player account
     *
     * @param amount of money
     */
    public void deposit(int amount) {
       balance.deposit(amount);
       GUIControl.updateBalance(this);
    }

    /**
     * Method for withdrawing an amount from the player account
     *
     * @param amount of money
     */
    public void withdraw(int amount) {
       balance.withdraw(amount);
       GUIControl.updateBalance(this);;
    }

    /**
     * Method for getting the current account balance of a player
     *
     * @return Account balance of the type integer
     */
    public int getBalance() {
       return balance.getBalance();
    }

    /**
     * Method for calculating where the player's vehicle lands after rolling the
     * dice
     *
     * @param roll value of the cup
     */
    public void moveVehicle(int roll) {
       vehicle.move(roll);
    }

    /**
     * Method for getting the previous position of the player's vehicle
     *
     * @return Previous player position of the type integer
     */
    public int getPreviousPosition() {
       return vehicle.getPreviousPosition();
    }

    /**
     * Method for getting the current position of the player's vehicle
```

```
     *
     * @return Player position of the type integer
     */
    public int getCurrentPosition() {
       return vehicle.getCurrentPosition();
    }

    /**
     * Method for setting the position of the player's vehicle
     *
     * @param currentPosition Type: int
     * @param previousPosition Type: int
     */
    public void setPosition(int currentPosition,int previousPosition) {
       vehicle.setPosition(currentPosition,previousPosition);
    }

    /**
     * Method for generating a color used by the
     *
     * @return A color of the type Color
     */
    public Color getColor() {
       return vehicle.getColor();
    }

    /**
     * Returns a list of the names of the properties owned by this player.
     *
     * @return String ArrayList
     */
    public ArrayList<String> getPropertyList() {

       return assets.getPropertyList();

    }

    /**
     * Method for getting a list of the properties with houses built on them.
     *
     * @return String[]
     */
    public String[] getHouseList() {
       return assets.getHouseList();
    }

    /**
     * Method for getting a list of the properties with hotels built on them.
     *
     * @return String []
     */
    public String[] getHotelList() {
       return assets.getHotelList();
    }

    /**
     * Returns a boolean value of whether the player owns a building
     * @return boolean
     */
    public boolean getBuilding() {
       return assets.getBuilding();
    }

    /**
     *
     * @return Returns a boolean value of whether the player owns a property or not.
     */
    public boolean getProperty() {

       return assets.getProperty();

    }
    /**
     * Method for buying houses on a Street.
     * @param street in question
     * @param amount of houses
     */
    public void buyBuildings(Street street, int amount) {

       assets.buyBuildings(street, amount);

    }
    /**
     * Method for removing houses on a Street
     * @param street in question
     * @param amount of houses
     */
    public void removeBuildings(Street street, int amount) {

       assets.removeBuildings(street, amount);

    }
    /**
     * Method for getting a boolean value of whether the player can build houses
     * @return boolean of has a building.
     */
    public boolean getBuildStatus() {

       return assets.getBuildStatus();

    }
    /**
     * Returns a string array of streets which can be built on.
     * @return String[] buildable squares
     */
    public String[] getBuildableList() {
       return assets.getBuildableList();
    }
    /**
     *
```

```
    * @return string array of sellable squares.
    */
   public String[] getSellableList() {

      return assets.getSellableList();

   }
   /**
    *
    * @return boolean value of whether the player can pawn something.
    */
   public boolean getPawnStatus() {

      return assets.getPawnStatus();

   }
   /**
    *
    * @return string array of pawnable squares
    */
   public String[] getPawnable() {

      return assets.getPawnable();

   }
   /**
    * Pawns a property.
    * @param ownable square
    */
   public void pawnProperty(Ownable ownable) {

      assets.pawnProperty(ownable);
   }
   /**
    * Lifts the pawn of the property.
    * @param ownable square
    */
   public void liftPawn(Ownable ownable) {

      assets.liftPawn(ownable);

   }
   /**
    *
    * @return String[] of pawned properties.
    */
   public String[] getPawned() {

      return assets.getPawned();

   }
   /**
    *
    * @return boolean value of whether the player has a pawned square.
    */
   public boolean hasPawned() {

      return assets.hasPawned();

   }

}
```

Listing 13: Vehicle

```
package entities;

import java.awt.Color;

/**
 * Class which keeps track of the player's position on the board and creates a
 * piece that the player moves with
 *
 */

public class Vehicle {

   Color vehicleColor = null;
   static int counter = 0;
   private int currentPosition = 0;
   private int previousPosition = 0;

   /**
    * Constructor that initializes a vehicle with a counter and a color for a
    * player
    */

   public Vehicle() {

      counter++;
      setColor();

   }

   /**
    * Method for calculating and returning the new position of a player's
    * vehicle while also saving the previous position
    *
    * @param value of movement
    * @return currentPosition
    */

   public int move(int value) {

      previousPosition = currentPosition;

```

```
42          currentPosition = (currentPosition + value) % 40;
43
44          return currentPosition;
45
46      }
47
48
49
50      /**
51       * Method for setting a new position of the player's vehicle
52       *
53       * @param currentPosition [0-39]
54       * @param previousPosition [0-39]
55       */
56      public void setPosition(int currentPosition,int previousPosition) {
57          this.currentPosition = currentPosition;
58          this.previousPosition = previousPosition;
59      }
60
61      /**
62       * Method for getting the previous position of a player's vehicle
63       *
64       * @return The previous position of the player vehicle, of the type integer
65       */
66
67      public int getPreviousPosition() {
68          return previousPosition;
69      }
70
71      /**
72       * Method for getting the current position of a player's vehicle
73       *
74       * @return The current position of the player vehicle, of the type integer
75       */
76
77      public int getCurrentPosition() {
78          return currentPosition;
79      }
80
81      /**
82       * Method for deciding the color of a player's vehicle, depending on number
83       * of players
84       *
85       * @return The color of a player's vehicle of the type color
86       */
87      private void setColor() {
88          switch (counter) {
89          case 1:
90              vehicleColor = Color.orange;
91              break;
92
93          case 2:
94              vehicleColor = Color.magenta;
95              break;
96
97          case 3:
98              vehicleColor = Color.pink;
99              break;
100
101         case 4:
102             vehicleColor = Color.cyan;
103             break;
104
105         case 5:
106             vehicleColor = Color.black;
107             break;
108
109         case 6:
110             vehicleColor = Color.GRAY;
111             break;
112
113         }
114
115     }
116
117     /**
118      * Method for returning the color of the player's vehicle
119      *
120      * @return Color of the type Color
121      */
122
123     public Color getColor() {
124         return vehicleColor;
125     }
126 }
```

Listing 14: Assets

```
1  package entities;
2
3  import java.util.ArrayList;
4
5  import board.Ownable;
6  import board.Street;
7
8  /**
9   * Class which keeps track of a Player's assets (Squares, buildings and
10  * jailCards).
11  *
12  */
13
14 public class Assets {
15
16     ArrayList<Ownable> owned = new ArrayList<Ownable>();
17     ArrayList<Street> ownedStreet = new ArrayList<Street>();
18     ArrayList<String> property = new ArrayList<String>();
```

```
19      private int jailCard;
20      private Player assetOwner;
21      private boolean propertyOwner;
22      private boolean buildingOwner;
23      private boolean buildStatus;
24      private boolean pawnStatus;
25      private boolean hasPawn;
26
27      /**
28       * Constructor for Assets
29       * @param player owner of this asset
30       */
31      public Assets(Player player) {
32         jailCard = 0;
33         propertyOwner = false;
34         buildingOwner = false;
35         buildStatus = false;
36         assetOwner = player;
37      }
38
39      /**
40       * Method for adding a get out of jail free card to the player
41       */
42      public void setJailCard() {
43         jailCard++;
44      }
45
46      /**
47       * Method for checking if the player has a get out of jail free card
48       *
49       * @return Boolean value true or false depending on whether or not the
50       *         player has a card
51       */
52
53      public boolean getJailCard() {
54         if (jailCard >= 1) {
55            return true;
56         } else {
57            return false;
58         }
59      }
60
61      /**
62       * Method for removing a get out of jail free card after it is used by the
63       * player
64       */
65      public void useJailCard() {
66         jailCard--;
67      }
68
69      /**
70       * Method for adding a bought square to list of squares a player owns
71       *
72       * @param square bought
73       */
74
75      public void buySquare(Ownable square) {
76         owned.add(square);
77         property.add(square.toString());
78         propertyOwner = true;
79
80         if (square instanceof Street) {
81
82            Street isStreet = (Street) square;
83            ownedStreet.add(isStreet);
84
85         }
86
87      }
88
89      /**
90       * Method for determining the square IDs of the squares a player owns
91       *
92       * @return An integer array with the square IDs
93       */
94      public int[] getOwnedID() {
95
96         int[] squareID = new int[owned.size()];
97
98         for (int i = 0; i < owned.size(); i++) {
99
100           squareID[i] = owned.get(i).getID();
101        }
102
103        return squareID;
104
105     }
106
107     /**
108      * Method for getting an ArrayList of owned squares
109      *
110      * @return ArrayList of Ownables
111      */
112     public ArrayList<Ownable> getOwned() {
113        return owned;
114     }
115     /**
116      *
117      * @return property String ArrayList
118      */
119     public ArrayList<String> getPropertyList() {
120
121        return property;
122
123     }
124     /**
125      *
126      * @return ArrayList Street of the owned streets.
127      */
128     public ArrayList<Street> getOwnedStreet() {
```

```
129
130         return ownedStreet;
131
132     }
133
134     /**
135      * Method for getting an array of the names of properties with houses built
136      * on them
137      *
138      * @return String[]
139      */
140     public String[] getHouseList() {
141
142         ArrayList<String> houseList = new ArrayList<String>();
143
144         for (int i = 0; i < ownedStreet.size(); i++) {
145
146             if (ownedStreet.get(i).getNumberOfBuildings() >= 1 && ownedStreet.get(i).getNumberOfBuildings() < 5) {
147
148                 houseList.add(ownedStreet.get(i).toString());
149
150             }
151
152         }
153
154         return houseList.toArray(new String[houseList.size()]);
155     }
156
157     /**
158      * Method for getting an array of the names of properties with hotels built
159      * on them
160      *
161      * @return String[]
162      */
163     public String[] getHotelList() {
164
165         ArrayList<String> hotelList = new ArrayList<String>();
166
167         for (int i = 0; i < ownedStreet.size(); i++) {
168
169             if (ownedStreet.get(i).getNumberOfBuildings() == 5) {
170
171                 hotelList.add(ownedStreet.get(i).toString());
172
173             }
174
175         }
176
177         return hotelList.toArray(new String[hotelList.size()]);
178     }
179     /**
180      * @return list of sellable squares.
181      */
182     public String[] getSellableList() {
183
184         ArrayList<String> sellableList = new ArrayList<String>();
185
186         for (int i = 0; i < ownedStreet.size(); i++) {
187
188             if (ownedStreet.get(i).getNumberOfBuildings() >= 1 && ownedStreet.get(i).getNumberOfBuildings() <= 5) {
189
190                 sellableList.add(ownedStreet.get(i).toString());
191
192             }
193
194         }
195
196         return sellableList.toArray(new String[sellableList.size()]);
197     }
198
199     /**
200      * Method for returning a boolean value of whether the player owns a
201      * building.
202      *
203      * @return boolean hasBuilding
204      */
205     public boolean getBuilding() {
206         return buildingOwner;
207     }
208
209     /**
210      * Method for returning a boolean value of whether the player owns a a
211      * property.
212      *
213      * @return boolean hasProperty
214      */
215     public boolean getProperty() {
216
217         return propertyOwner;
218
219     }
220
221     /**
222      * Method for buying an amount of buildings on a specific Street.
223      *
224      * @param street in question
225      * @param amount of buildings
226      */
227     public void buyBuildings(Street street, int amount) {
228
229         street.buyBuildings(amount);
230         buildingOwner = true;
231
232     }
233
234     /**
235      * Method for removing an amount of houses from a specific Street.
236      *
237      * @param street in question
238      * @param amount of buildings
```

```
239     */
240    public void removeBuildings(Street street, int amount) {
241
242       boolean noBuilding = true;
243       buildingOwner = false;
244
245       street.removeBuildings(amount);
246
247       for (int i = 0; i < ownedStreet.size(); i++) {
248
249          if (ownedStreet.get(i).getNumberOfBuildings() >= 1) {
250
251             noBuilding = false;
252
253          }
254
255          if (noBuilding == false) {
256
257             buildingOwner = true;
258
259          }
260
261       }
262    }
263
264    /**
265     * Method for returning a boolean value of whether the player can build a
266     * house.
267     *
268     * @return boolean
269     */
270    public boolean getBuildStatus() {
271
272       if (this.getBuildableList().length != 0) {
273
274          buildStatus = true;
275
276       } else {
277
278          buildStatus = false;
279
280       }
281
282       return buildStatus;
283
284    }
285
286    /**
287     * Method for getting a String array of streets which there can be built
288     * buildings on.
289     *
290     * @return String[]
291     */
292    public String[] getBuildableList() {
293
294       ArrayList<String> buildableList = new ArrayList<String>();
295       int counterA = 0;
296       int counterB = 0;
297       int counterC = 0;
298       int counterD = 0;
299       int counterE = 0;
300       int counterF = 0;
301       int counterG = 0;
302       int counterH = 0;
303
304       for (int i = 0; i < ownedStreet.size(); i++) {
305
306          if (ownedStreet.get(i).getType() == 'A') {
307             counterA++;
308
309          } else if (ownedStreet.get(i).getType() == 'B') {
310             counterB++;
311
312          } else if (ownedStreet.get(i).getType() == 'C') {
313             counterC++;
314
315          } else if (ownedStreet.get(i).getType() == 'D') {
316             counterD++;
317
318          } else if (ownedStreet.get(i).getType() == 'E') {
319             counterE++;
320
321          } else if (ownedStreet.get(i).getType() == 'F') {
322             counterF++;
323
324          } else if (ownedStreet.get(i).getType() == 'G') {
325             counterG++;
326
327          } else if (ownedStreet.get(i).getType() == 'H') {
328             counterH++;
329          }
330
331       }
332
333       for (int i = 0; i < ownedStreet.size(); i++) {
334
335          if (counterA == 2 && ownedStreet.get(i).getType() == 'A'
336                || counterB == 3 && ownedStreet.get(i).getType() == 'B'
337                || counterC == 3 && ownedStreet.get(i).getType() == 'C'
338                || counterD == 3 && ownedStreet.get(i).getType() == 'D'
339                || counterE == 3 && ownedStreet.get(i).getType() == 'E'
340                || counterF == 3 && ownedStreet.get(i).getType() == 'F'
341                || counterG == 3 && ownedStreet.get(i).getType() == 'G' || counterH == 2
342                      && ownedStreet.get(i).getType() == 'H' && ownedStreet.get(i).getNumberOfBuildings() < 5) {
343
344             if (ownedStreet.get(i).getPropertyPawnStatus() == false
345                   && assetOwner.getBalance() >= ownedStreet.get(i).getPriceOfBuilding()
346                   && ownedStreet.get(i).getNumberOfBuildings() < 5) {
347
348                buildableList.add(ownedStreet.get(i).toString());
```

```
            }
         }
      }

      return buildableList.toArray(new String[buildableList.size()]);

   }
   /**
    *
    * @return pawnstatus
    */
   public boolean getPawnStatus() {

      if (this.getPawnable().length != 0) {

         pawnStatus = true;

      } else {

         pawnStatus = false;

      }

      return pawnStatus;

   }
   /**
    *
    * @return list of pawnable squares.
    */
   public String[] getPawnable() {

      ArrayList<String> pawnable = new ArrayList<String>();

      for (int i = 0; i < ownedStreet.size(); i++) {

         if (ownedStreet.get(i).getPropertyPawnStatus() == false && ownedStreet.get(i).getNumberOfBuildings() == 0) {

            pawnable.add(ownedStreet.get(i).toString());
         }

      }

      for (int j = 0; j < owned.size(); j++) {

         if (owned.get(j).getPropertyPawnStatus() == false && !(owned.get(j) instanceof Street)) {

            pawnable.add(owned.get(j).toString());

         }
      }

      return pawnable.toArray(new String[pawnable.size()]);

   }
   /**
    *
    * @return list of the pawned squares.
    */
   public String[] getPawned() {

      ArrayList<String> pawned = new ArrayList<String>();

      for (int i = 0; i < owned.size(); i++) {

         if (owned.get(i).getPropertyPawnStatus() == true) {

            pawned.add(owned.get(i).toString());

         }

      }

      return pawned.toArray(new String[pawned.size()]);

   }
   /**
    *
    * @return boolean value of whether Assets contain a pawned square.
    */
   public boolean hasPawned() {

      if (owned.size() == 0) {
         hasPawn = false;
         return hasPawn;
      } else {
         for (int i = 0; i < owned.size(); i++) {
            if (owned.get(i).getPropertyPawnStatus() == true) {
               hasPawn = true;
            }
         }
         return hasPawn;
      }
   }
   /**
    * Sets the pawnstatus to true of this square
    * @param ownable in question
    */
   public void pawnProperty(Ownable ownable) {

      ownable.pawnProperty();
      pawnStatus = false;
      hasPawn = true;

   }
   /**
    * Unpawns a square.
    * @param ownable in question
    */
```

```
459     public void liftPawn(Ownable ownable) {
460
461        ownable.liftPawn();
462        pawnStatus = true;
463        hasPawn = false;
464     }
465
466  }
```

Listing 15: Account

```
1   package entities;
2   /**
3    * Class for creating an Account, which keeps track of a players balance.
4    *
5    */
6   public class Account {
7
8      private int balance;
9
10     /**
11      * Constructor the initializes the player's account with a balance of 0
12      */
13
14     public Account() {
15        balance = 0;
16     }
17
18     /**
19      * Method for depositing money into a player's account
20      *
21      * @param value of money
22      */
23
24     public void deposit(int value) {
25        if (value > 0) {
26           balance += value;
27        }
28     }
29
30     /**
31      * Method for withdrawing money from a player's account
32      *
33      * @param value of money
34      */
35
36     public void withdraw(int value) {
37        if (value > 0) {
38           balance -= value;
39        }
40     }
41
42     /**
43      * Method for checking the amount on a player's account balance
44      *
45      * @return The amount of money on an account, of the type integer
46      */
47
48     public int getBalance() {
49        return balance;
50     }
51
52  }
```

11.3 Package board

Listing 16: Square

```
1   package board;
2   import entities.Player;
3   /**
4    * Abstract class Square, superclass to all Squares.
5    *
6    */
7   public abstract class Square {
8      protected String name;
9      protected int id;
10     /**
11      * Constructor
12      * @param name of this instance
13      * @param id [1:40]
14      */
15     public Square( String name, int id){
16        this.name=name;
17        this.id=id;
18     }
19     /**
20      * Method which determines what happens to a player when he lands on this instance.
21      * @param player landed
22      */
23     public abstract void landOnSquare(Player player);
24     /**
25      * Returns the name of this instance
26      * @return name
27      */
28     public String toString(){
29        return name;
30     }
31     /**
```

```
    * Returns the id (int) of this instance.
    * @return id [1:40]
    */
   public int getID(){
      return this.id;
   }

}
```

Listing 17: Start

```
package board;

import controller.GUIControl;
import controller.msgL;
import entities.Player;

/**
 * Class Start extends Square
 *
 */

public class Start extends Square {
   /**
    * Constructor
    * @param name of this instance
    * @param id [1:40]
    */
   public Start(String name, int id) {
      super(name, id);

   }

   @Override
   public void landOnSquare(Player player) {
      GUIControl.printMessage(msgL.msg(185));
   }

}
```

Listing 18: Parking

```
package board;
import entities.Player;
import controller.GUIControl;
/**
 * Class Parking extends Square
 *
 */
public class Parking extends Square {
   /**
    * Constructor
    * @param name of this instance
    * @param id [1:40]
    */
   public Parking(String name, int id) {
      super(name, id);
   }
   /**
    * When landing on a Parking square, a message is printet.
    * @param player who landed on this square
    */
   public void landOnSquare(Player player){
      GUIControl.printMessage(name);
   }
}
```

Listing 19: Jail

```
package board;

import entities.Player;
import controller.GUIControl;
import controller.msgL;

/**
 * Class Jail extends Square
 *
 */

public class Jail extends Square {

   /**
    * Constructor
    * @param name of this instance
    * @param id [1:40]
    */
   public Jail(String name, int id) {
      super(name, id);
   }

   @Override
   public void landOnSquare(Player player) {
      if(player.getJailCard()== false){
      GUIControl.printMessage(msgL.msg(164));
      player.setJailStatus(true);
      player.setPosition(10,player.getCurrentPosition());
      GUIControl.moveVehicle(player);
      } else {
```

```
        GUIControl.printMessage(msgL.msg(187));
        player.setPosition(10,player.getCurrentPosition());
        GUIControl.moveVehicle(player);
        player.useJailCard();
      }
   }
}
```

Listing 20: Tax

```
package board;

import entities.Player;

import java.util.ArrayList;

import controller.GUIControl;
import controller.msgL;

public class Tax extends Square {
   private int taxAmount; // amount defined in constructor

   /**
    * Constructor
    *
    * @param name
    *           of this instance
    * @param id
    *           [1:40]
    * @param taxAmount
    *           to be withdrawed if chosen.
    */
   public Tax(String name, int id, int taxAmount) {
      super(name, id);
      this.taxAmount = taxAmount;
   }

   @Override
   public void landOnSquare(Player player) {

      if (id == 5) {
         Boolean choice = GUIControl.getTaxChoice(name, player);
         if (choice == true) {
            player.withdraw(taxAmount);
         } else {

            this.taxPercent(player);

         }
      } else {
         GUIControl.printMessage(player.toString() + msgL.msg(116) + taxAmount + msgL.msg(119) + msgL.msg(212));
         player.withdraw(taxAmount);
      }
   }

   /**
    * Method for getting taxAmount.
    *
    * @return taxAmount
    */
   public int getTaxAmount() {
      return taxAmount;
   }

   /**
    * Method for calculating and withdrawing 10% of a player's total assets
    * @param player
    */

   public void taxPercent(Player player) {

      ArrayList<Integer> propertyPrice = new ArrayList<Integer>();
      ArrayList<Integer> buildingPrice = new ArrayList<Integer>();
      int sumProperty = 0;
      int sumBuilding = 0;

      for (int i = 0; i < player.getOwned().size(); i++) {
         propertyPrice.add(player.getOwned().get(i).getPrice());
      }
      for (int j = 0; j < player.getOwnedStreet().size(); j++) {
         buildingPrice.add(player.getOwnedStreet().get(j).getNumberOfBuildings()
               * player.getOwnedStreet().get(j).getPriceOfBuilding());
      }
      for (int k = 0; k < propertyPrice.size(); k++) {
         sumProperty += propertyPrice.get(k);
      }
      for (int l = 0; l < buildingPrice.size(); l++) {
         sumBuilding += buildingPrice.get(l);
      }
      int taxAmount = (player.getBalance() + sumProperty + sumBuilding) / 10;
      GUIControl.printMessage(msgL.msg(165) + taxAmount + msgL.msg(119));
      player.withdraw(taxAmount);

   }

}
```

Listing 21: Chance

```
package board;

import controller.GUIControl;
```

```
import entities.Player;
import entities.AllCards;

/**
 *
 * Class Chance extends Squar
 *
 */

public class Chance extends Square {

   AllCards allTheCards;
   static int index = 0;

   /**
    * Constructor
    * @param name of this instance
    * @param id [1:40]
    * @param allTheCards of the game
    */
   public Chance(String name, int id, AllCards allTheCards) {
      super(name, id);
      this.allTheCards = allTheCards;
   }

   @Override
   public void landOnSquare(Player player) {
      GUIControl.printMessage(name); // Text "Prøv lykken" is printet.
      String description = allTheCards.getCard(index).toString();
      GUIControl.displayChanceCard(description);
      allTheCards.getCard(index).useCard(player);

      if (index == 43) {
         index = 0;
         allTheCards.shuffle();
      } else {
         index++;
      }
   }
}
```

Listing 22: Ownable

```
package board;

import entities.Player;
import controller.GUIControl;
import controller.msgL;

/**
 * Abstract class Ownable, extended from Square.
 * Superclass to all ownable subclasses of Square.
 *
 */
public abstract class Ownable extends Square {
   protected int price;
   protected int pawn;
   protected char type;
   protected Player owner;
   protected boolean pawnStatus;

   /**
    * Constructor
    * @param name of this instance
    * @param id [1:40]
    * @param price of this instance
    * @param pawn price
    * @param type [A:J]
    */
   public Ownable(String name, int id, int price, int pawn, char type) {
      super(name, id);
      this.price = price;
      this.pawn = pawn;
      this.type = type;
      this.pawnStatus = false;
   }

   @Override
   public void landOnSquare(Player player) {
      if (owner != null) {// if the square is owned, the following happens.
         if (player.toString().equals(owner.toString()) == false) {// if the player is not the owner of
                                                   // this field, the following happens
            if(owner.getJailStatus()==true){
               GUIControl.printMessage(msgL.msg(127));
            } else if (this.pawnStatus == true) {

               GUIControl.printMessage(msgL.msg(188));

            }else{
               int amount = getRent();// the rent is calculated, depending on the subclass.
               GUIControl.printMessage(player.toString() + msgL.msg(111) + this.toString() + msgL.msg(112) + owner.toString()
                     +msgL.msg(113) + amount + msgL.msg(119) + msgL.msg(114) + owner.toString());
               player.withdraw(amount);
               owner.deposit(amount);
            }
         } else { //if the player is the owner of this square
            GUIControl.printMessage(msgL.msg(120));
         }
      } else if (player.getBalance() >= this.price) {// if the field is'nt owned and the player
                                    // has enough money, he has the choice of buying it.
         if (GUIControl.getBuyChoice(this, player) == true) {// if player chooses to buy it, the following happens
            GUIControl.printMessage(msgL.msg(115) + this.toString() + msgL.msg(184) + this.getPrice() + msgL.msg(119));
            player.buySquare(this);
            player.withdraw(this.price);
            owner = player;
            GUIControl.setOwned(this.getID(), player);
```

```
        }
      } else {// if the square is not owned but the player can't afford it, a
            // message is printed.
        GUIControl.printMessage(msgL.msg(121) + this.toString());
      }
   }

   /**
    * Method for getting the rent of the instance this method is called upon.
    *
    * @return rent
    */
   public abstract int getRent();

   /**
    * Sets pawn status to false and withdraws an amount to the owners Account.
    */
   public void liftPawn() {
      int amount = pawn+(pawn/10);
      owner.withdraw(amount);
      this.pawnStatus = false;
   }

   /**
    *Sets pawn status to true and deposits the pawn price to the owners Account.
    */
   public void pawnProperty() {
      owner.deposit(pawn);
      this.pawnStatus = true;
   }

   /**
    * Sets owner to null.
    */
   public void clearOwner() {
      this.owner = null;
   }
   /**
    *
    * @return boolean value of the pawn status.
    */
   public boolean getPropertyPawnStatus() {
      return pawnStatus;
   }

   /**
    *
    * @return price of this instance
    */
   public int getPrice() {
      return price;
   }

   /**
    *
    * @return the type of this instance.
    */
   public char getType() {
      return type;
   }

   /**
    *
    * @return pawn amount.
    */
   public int getPawn() {
      return pawn;
   }

   /**
    * Test method for setting the owner
    * @param player
    */

   public void setOwner(Player player) {

      owner = player;

   }

   /**
    * Test method for getting the owner
    * @return Owner of the square of the type Player
    */

   public Player getOwner() {

      return owner;

   }


}
```

Listing 23: Street

```
package board;

import java.util.ArrayList;

import controller.GUIControl;
import controller.msgL;

/**
 * Class Street extends Ownable
 *
```

```
 */
public class Street extends Ownable {
   int priceOfBuilding;
   int[] rents = new int[6];
   int numberOfBuildings = 0;

   /**
    * Constructor
    * @param name of this instance
    * @param id [1:40]
    * @param price of this instance
    * @param pawn price
    * @param priceOfBuilding price of a building
    * @param rent0 base
    * @param rent1 house1
    * @param rent2 house2
    * @param rent3 house3
    * @param rent4 house4
    * @param rentHotel hotel
    * @param type [A:H]
    */
   public Street(String name, int id, int price, int pawn, int priceOfBuilding, int rent0, int rent1, int rent2,
         int rent3, int rent4, int rentHotel, char type) {
      super(name, id, price, pawn, type);
      this.priceOfBuilding = priceOfBuilding;
      rents[0] = rent0;
      rents[1] = rent1;
      rents[2] = rent2;
      rents[3] = rent3;
      rents[4] = rent4;
      rents[5] = rentHotel;
   }

   @Override
   public int getRent() {
      int maxAmount;
      ArrayList<Ownable> arr = owner.getOwned();
      int counter = 0;
      for (int i = 0; i < arr.size(); i++) {
         if (arr.get(i).getType() == this.type) {
            counter++;
         }
      }
      if (this.type == 'A' || this.type == 'H') {
         maxAmount = 2;
      } else {
         maxAmount = 3;
      }
      if (counter == maxAmount && numberOfBuildings == 0) {
         return rents[0] * 2;
      } else {
         return rents[numberOfBuildings];
      }
   }

   /**
    * Method for buying an amount buildings on a Street
    * @param amount of buildings
    */
   public void buyBuildings(int amount) {
      int maxAmount;
      ArrayList<Ownable> arr = owner.getOwned();
      int counter = 0;
      for (int i = 0; i < arr.size(); i++) {
         if (arr.get(i).getType() == this.type) {
            counter++;
         }
      }
      if (this.type == 'A' || this.type == 'H') {
         maxAmount = 2;
      } else {
         maxAmount = 3;
      }
      if (counter == maxAmount) {
         numberOfBuildings += amount;
         owner.withdraw(amount * priceOfBuilding);
         GUIControl.printMessage(msgL.msg(168) + amount + msgL.msg(169) + (amount * priceOfBuilding));
      }
   }


   /**
    * Method for removing an amount of buildings on this instance.
    * @param amount to be removed
    */
   public void removeBuildings(int amount) {
      numberOfBuildings -= amount;
      owner.deposit(amount * (priceOfBuilding/2));
   }

   /**
    *
    * @return Number of buildings of this instance
    */
   public int getNumberOfBuildings() {
      return numberOfBuildings;
   }

   /**
    *
    * @return The cost of building a building
    */
   public int getPriceOfBuilding() {
      return priceOfBuilding;
   }

   /**
    * Test method for setting a number of buildings on the street
    * @param amount
    */
```

```
121
122    public void setBuilding(int amount) {
123
124       numberOfBuildings = amount;
125
126    }
127 }
```

Listing 24: Shipping

```
1  package board;
2  /**
3   * Class Shipping extends Ownable
4   *
5   */
6  public class Shipping extends Ownable {
7     int [] rents={500,1000,2000,4000};
8     int numberOfShips;
9     /**
10     * Constructor
11     * @param name of this instance
12     * @param id [1:40]
13     */
14    public Shipping(String name, int id) {
15       super(name, id, 4000, 2000, 'J');  // the price, pawn-price and type for a Shipping field
16                                          // is always the same.
17    }
18    @Override
19    public int getRent() {
20       numberOfShips=-1;
21       for(int i=0; i<owner.getOwned().size();i++){
22          if(owner.getOwned().get(i).getType()==this.getType()){
23             numberOfShips++;
24          }
25       }
26       return rents[numberOfShips];
27    }
28 }
```

Listing 25: Brewery

```
1  package board;
2
3  import entities.Cup;
4
5  /**
6   * Class Brewery extends Ownable
7   *
8   */
9  public class Brewery extends Ownable {
10
11    private int multiplier_1=100;
12    private int multiplier_2=200;
13    private Cup gameCup;
14    /**
15     * Constructor
16     * @param name of this instance
17     * @param id [1:40]
18     * @param gameCup of the game
19     */
20    public Brewery(String name, int id, Cup gameCup) {
21       super(name, id, 3000, 1500, 'I'); // the price, pawn-price and type for a Shipping field
22                                         // is always the same.
23       this.gameCup = gameCup;
24    }
25
26    @Override
27    public int getRent(){
28       int typeCounter=0;
29       //the for loop evaluates the number of breweries the owner has
30       for(int i = 0; i<owner.getOwned().size();i++){
31          if(owner.getOwned().get(i).getType()==this.getType()){
32             typeCounter++;
33          }
34       }
35       if(typeCounter==2){//Condition for doubling the rent.
36          return gameCup.getSum()*multiplier_2;
37       }else{
38          return gameCup.getSum()*multiplier_1;
39       }
40    }
41
42 }
```

11.4 Package cards

Listing 26: Card

```
1  package cards;
2
3  import entities.Player;
4
5  /**
6   * Abstract class Card, superclass to all Cards.
7   *
```

```
 */
public abstract class Card {
   protected String description;


   /**
    * Super constructor which takes a String name as a parameter.
    *
    * @param description of the card
    */
   public Card(String description) {
      this.description = description;
   }

   /**
    * Returns the name of the Card
    * @return description
    */
   @Override
   public String toString() {
      return description;
   }

   /**
    * Method which determines what happens to a player when the specific card
    * is picked.
    * @param player to use the card
    */
   public abstract void useCard(Player player);
}
```

Listing 27: Move

```
package cards;


/**
 * Abstract class for Card subclasses which changes the position of the player.
 *
 */
public abstract class Move extends Card {
   protected int moveTo;

   /**
    * Constructor
    * @param description of the cards
    * @param moveTo a number
    */
   public Move(String description, int moveTo) {
      super(description);
      this.moveTo = moveTo;
   }
}
```

Listing 28: Transaction

```
package cards;


/**
 * abstract. amount
 *
 */
public abstract class Transaction extends Card {
   int money;

   /**
    * Constructor for a Transaction card
    *
    * @param description of the card
    * @param money to be transferred
    */
   public Transaction(String description, int money) {
      super(description);
      this.money = money;
   }
}
```

Listing 29: Grant

```
package cards;



import board.Street;
import controller.GUIControl;
import controller.msgL;
import entities.Player;

/**
 * Class Grant extends Transaction
 *
 */
public class Grant extends Transaction {

   /**
    * Constructor for Grant card
    *
    * @param description of the card.
```

```
    * @param money to be granted
    */
   public Grant(String description, int money) {
      super(description, money);
   }

   @Override
   public void useCard(Player player) {
      Street a;
      int total = player.getBalance();
      for (int i = 0; i < player.getOwned().size(); i++) {
         total += player.getOwned().get(i).getPrice();
         if (player.getOwned().get(i) instanceof Street) {
            a = (Street) player.getOwned().get(i);
            total += a.getNumberOfBuildings() * a.getPriceOfBuilding();
         }
      }
      if (total <= 15000) {
         GUIControl.printMessage(msgL.msg(122));
         player.deposit(40000);
      } else {
         GUIControl.printMessage(msgL.msg(123));
      }
   }
}
```

Listing 30: Pardon

```
package cards;

import controller.GUIControl;
import entities.Player;

/**
 * Class which awards the Player with a 'get out of jail for free card'.
 *
 */
public class Pardon extends Card {

   /**
    * Constructor for PrisonBreak card
    *
    * @param description of the card
    */
   public Pardon(String description) {
      super(description);

   }

   @Override
   public void useCard(Player player) {
      player.setJailCard();
      GUIControl.printMessage(description);
   }

}
```

Listing 31: MoveToShip

```
package cards;
import entities.Player;
import controller.GUIControl;
import entities.Board;

/**
 * Class for cards which moves a player to nearest Shipping square
 *
 */
public class MoveToShip extends Card {
   Board board;
   /**
    * Constructor for MoveToShip card
    * @param description oif the card
    * @param board of the game
    */
   public MoveToShip(String description, Board board) {
      super(description);
      this.board=board;
   }

   @Override
   public void useCard(Player player) {
      GUIControl.printMessage(description);
      if (player.getCurrentPosition()==2 || player.getCurrentPosition()==36) {
         player.setPosition(5,player.getCurrentPosition());
         GUIControl.moveVehicle(player);
         board.getSquare(5).landOnSquare(player);

      }else if (player.getCurrentPosition()==7) {
         player.setPosition(15,player.getCurrentPosition());
         GUIControl.moveVehicle(player);
         board.getSquare(15).landOnSquare(player);

      }else if (player.getCurrentPosition()==17 || (player.getCurrentPosition()==22)) {
         player.setPosition(25,player.getCurrentPosition());
         GUIControl.moveVehicle(player);
         board.getSquare(25).landOnSquare(player);

      }else if (player.getCurrentPosition()==33) {
         player.setPosition(35,player.getCurrentPosition());
         GUIControl.moveVehicle(player);
         board.getSquare(35).landOnSquare(player);
```

```
   }

   }
}
```

Listing 32: ChangePosition

```
package cards;

import controller.GUIControl;
import entities.Board;
import entities.Player;

/**
 * Class ChangePosition extends Move
 *
 */
public class ChangePosition extends Move {
   Board board;
   /**
    * Constructor
    * @param description of the card.
    * @param moveTo the position the player shall be moved by.
    * @param board of the game
    */
   public ChangePosition(String description, int moveTo, Board board) {
      super(description, moveTo);
      this.board=board;
   }
   @Override
   public void useCard(Player player) {
      GUIControl.printMessage(description);
      player.setPosition(moveTo+player.getCurrentPosition(),player.getCurrentPosition());
   if(player.getCurrentPosition() < 0){
      player.setPosition(39,player.getPreviousPosition());
   }
   GUIControl.moveVehicle(player);
   board.getSquare(player.getCurrentPosition()).landOnSquare(player);
   }
}
```

Listing 33: MoveToSquare

```
package cards;

import entities.Player;
import controller.GUIControl;
import entities.Board;

/**
 * Class for moving a Player to a specific Square
 *
 */
public class MoveToSquare extends Move {

   Board board;
   /**
    * Constructor for MoveToSquare
    * @param description of the card
    * @param moveTo the square ID the player should move to
    * @param board in the game
    */
   public MoveToSquare(String description, int moveTo,Board board){
      super(description,moveTo);
      this.board=board;
   }
   @Override
   public void useCard(Player player){
      GUIControl.printMessage(description);
      player.setPosition(moveTo,player.getCurrentPosition());
      GUIControl.moveVehicle(player);
      board.getSquare(moveTo).landOnSquare(player);
   }
}
```

Listing 34: GoToJail

```
package cards;

import controller.GUIControl;
import entities.Player;

/**
 * Class GoToJail extends Move
 *
 */
public class GoToJail extends Move{
   /**
    * Constructor for GoToJail card.
    *
    * @param description of the card
    */
    public GoToJail(String description) {
    super(description,10);
   }

   @Override
   public void useCard(Player player) {
      GUIControl.printMessage(description);
```

```
      player.setJailStatus(true);
      player.setPosition(10,player.getCurrentPosition());
      GUIControl.moveVehicle(player);
   }



}
```

Listing 35: PriceIncrease

```
package cards;


import entities.Player;
import java.util.ArrayList;
import controller.GUIControl;
import controller.msgL;

/**
 * Class for card which charges the Player with a fee for houses and hotels.
 *
 */
public class PriceIncrease extends Transaction {
   protected int houseTax;
   protected int hotelTax;

   /**
    * Constructor for a PriceIncrease card
    *
    * @param description of the card
    * @param houseTax increase
    * @param hotelTax increase
    */
   public PriceIncrease(String description, int houseTax, int hotelTax) {
      super(description, houseTax);
      this.houseTax = houseTax;
      this.hotelTax = hotelTax;

   }

   @Override
   public void useCard(Player player) {

      ArrayList<Integer> housePrice = new ArrayList<Integer>();
      ArrayList<Integer> hotelPrice = new ArrayList<Integer>();
      int sumHouse = 0;
      int sumHotel = 0;

      GUIControl.printMessage(description);

         if (player.getBuilding() == true) {

            for (int i = 0; i < player.getOwnedStreet().size(); i++) {

               if (player.getOwnedStreet().get(i).getNumberOfBuildings() >= 1
                     && player.getOwnedStreet().get(i).getNumberOfBuildings() < 5) {

                  housePrice.add(player.getOwnedStreet().get(i).getNumberOfBuildings() * houseTax);

               } else if (player.getOwnedStreet().get(i).getNumberOfBuildings() == 5) {

                  hotelPrice.add(hotelTax);

               }

            }

            for (int j = 0; j < housePrice.size(); j++) {

               sumHouse += housePrice.get(j);

            }

            for (int k = 0; k < hotelPrice.size(); k++) {

               sumHotel += hotelPrice.get(k);

            }

            player.withdraw(sumHouse + sumHotel);

         } else {

            GUIControl.printMessage(msgL.msg(126));

         }
   }
}
```

Listing 36: IncomeIncrease

```
package cards;

import controller.GUIControl;
import controller.msgL;
import entities.Player;

/**
 * Class IncomeIncrease extends Transaction
 *
 */
```

```java
public class IncomeIncrease extends Transaction {
   /**
    * Constructor
    *
    * @param description of the card
    * @param amount to be rewarded.
    */
   public IncomeIncrease(String description, int amount) {
      super(description, amount);
   }

   /**
    * @return pay amount
    */
   public int getMoney() {
      return money;
   }

   /**
    * Player receives award, and balance is updated.
    *
    * @param player to use the card
    */
   public void useCard(Player player) {
      GUIControl.printMessage(msgL.msg(118)+ money);
      player.deposit(money);
   }
}
```

Listing 37: PlayerTransaction

```java
package cards;

import java.util.ArrayList;
import controller.GUIControl;
import entities.Player;


public class PlayerTransaction extends Transaction{
   ArrayList<Player> playerList;

   int number = 0;

   /**
    * Constructor for MobilePay Card
    * @param description of the card
    * @param money to be payed
    * @param playerList a list of the players
    */
   public PlayerTransaction(String description, int money, ArrayList<Player> playerList) {
      super(description, money);
      this.playerList = playerList;
   }

   @Override
   public void useCard(Player player) {


   int amount = playerList.size();

        GUIControl.printMessage(description);

        for (int i=0; i<playerList.size();i++) {
           playerList.get(i).withdraw(money);
        }
      player.deposit(money*amount);
      }
   }
```

Listing 38: Grant

```java
package cards;



import board.Street;
import controller.GUIControl;
import controller.msgL;
import entities.Player;

/**
 * Class Grant extends Transaction
 *
 */
public class Grant extends Transaction {

   /**
    * Constructor for Grant card
    *
    * @param description of the card.
    * @param money to be granted
    */
   public Grant(String description, int money) {
      super(description, money);
   }

   @Override
   public void useCard(Player player) {
      Street a;
      int total = player.getBalance();
      for (int i = 0; i < player.getOwned().size(); i++) {
         total += player.getOwned().get(i).getPrice();
```

```
        if (player.getOwned().get(i) instanceof Street) {
           a = (Street) player.getOwned().get(i);
           total += a.getNumberOfBuildings() * a.getPriceOfBuilding();
        }
      }
      if (total <= 15000) {
         GUIControl.printMessage(msgL.msg(122));
         player.deposit(40000);
      } else {
         GUIControl.printMessage(msgL.msg(123));
      }
   }
}
```

11.5 Package test

Listing 39: FakeCup

```
package test;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

import entities.Cup;

public class FakeCup extends Cup{
   String[] cupArray = new String[100];
   int[] d1= new int[100];
   int[] d2= new int[100];
   private static int counter=0;

   public FakeCup(int testCaseNumber) {
      //System.out.println(new File("ajknsfhijoa").getAbsolutePath()); // FInd bib
      String fileName = "cupFile.csv";
      File file = new File(fileName);
      int i = 0;
      try {
         Scanner inputStream = new Scanner(file);
         while (inputStream.hasNextLine()) {
            cupArray[i++] = inputStream.nextLine().split("\t")[testCaseNumber];
         }
         inputStream.close();
      } catch (FileNotFoundException e) {
         System.out.println("Forkert");
      }

      for(i=0;i<cupArray.length;i++){
         if(cupArray[i] == null)
            break;
         d1[i]=Integer.valueOf(cupArray[i].split(",")[0]);
         d2[i]=Integer.valueOf(cupArray[i].split(",")[1]);
      }
   }
   @Override
   public int roll(){
      counter++;
      int value=d1[counter]+d2[counter];
      return value;
   }
   /**
    * Method for getting the result of the roll.
    *
    * @return sum of the two dices.
    */
   public int getSum() {
      return d1[counter] + d2[counter];
   }

   /**
    * Method for identifying if the two die show the same value.
    *
    * @return true if the two dices shows the same eyes.
    */
   public boolean getEquals() {
      boolean res = false;
      if (d1[counter] == d2[counter]) {
         res = true;
      }
      return res;
   }

   /**
    * Method for getting the value of dice 1.
    *
    * @return value of dice 1.
    */
   public int getD1() {
      return d1[counter];
   }

   /**
    * Method for getting the value of dice 2.
    *
    * @return value of dice 2.
    */
   public int getD2() {
      return d2[counter];
   }
}
```

Listing 40: TestBonus

```
package test;


import entities.Player;
import static org.junit.Assert.*;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;


public class TestBonus {

   Player p1;
   String name="Simon";
   int initBalance=0;
   int orgCurrentPosition=33;
   int orgPreviousPosition=27;

   @Before
   public void setUp(){
      p1= new Player(name, initBalance);
      p1.setPosition(orgCurrentPosition, orgPreviousPosition);
      assertTrue(p1.toString().equals(name));
      assertFalse(p1.getJailStatus());
      assertEquals(p1.getBalance(),initBalance);
      assertEquals(p1.getCurrentPosition(),orgCurrentPosition);
      assertEquals(p1.getPreviousPosition(),orgPreviousPosition);
   }
   @After
   public void tearDown(){
      p1 = new Player(name,initBalance);
      p1.setPosition(orgCurrentPosition, orgPreviousPosition);
   }
   @Test
   public void conditions1() {
      int rollValue = 4;
      p1.setFirstRound(false);
      p1.setPosition(p1.getCurrentPosition()+rollValue, p1.getCurrentPosition());
      //uses the same algorithm as in GUIControl.moveVehicle()
      for (int i = 0; i < rollValue; i++) {//Sets the player on one spot after another to simulate a piece moving on the board
         if ((p1.getPreviousPosition() + i + 1) % 40 + 1 == 2) {//if the player has passed the Start field
            if (p1.getFirstRound() == false && p1.getCurrentPosition() != 10) {
               p1.deposit(4000);
            } else {
               p1.setFirstRound(false);
            }
         }
      }
      assertFalse(p1.getBalance()>initBalance);
   }
   @Test
   public void conditions2(){
      int rollValue=7;
      p1.setFirstRound(false);
      p1.setPosition(p1.getCurrentPosition()+rollValue,p1.getCurrentPosition());
      for (int i = 0; i < rollValue; i++) {//Sets the player on one spot after another to simulate a piece moving on the board
         if ((p1.getPreviousPosition() + i + 1) % 40 + 1 == 2) {//if the player has passed the Start field
            if (p1.getFirstRound() == false && p1.getCurrentPosition() != 10) {
               p1.deposit(4000);
            } else {
               p1.setFirstRound(false);
            }
         }
      }
      assertFalse(p1.getBalance()>initBalance);
   }
   @Test
   public void conditions3(){
      int rollValue=10;
      p1.setFirstRound(false);
      p1.setPosition(p1.getCurrentPosition()+rollValue,p1.getCurrentPosition());
      for (int i = 0; i < rollValue; i++) {//Sets the player on one spot after another to simulate a piece moving on the board
         if ((p1.getPreviousPosition() + i + 1) % 40 + 1 == 2) {//if the player has passed the Start field
            if (p1.getFirstRound() == false && p1.getCurrentPosition() != 10) {
               p1.deposit(4000);
            } else {
               p1.setFirstRound(false);
            }
         }
      }
      assertTrue(p1.getBalance()>initBalance);
   }
   @Test
   public void conditions4(){
      p1.setPosition(0, 0);
      int rollValue=10;
      p1.setPosition(10, 0);
      p1.setPosition(p1.getCurrentPosition()+rollValue,p1.getCurrentPosition());
      for (int i = 0; i < rollValue; i++) {//Sets the player on one spot after another to simulate a piece moving on the board
         if ((p1.getPreviousPosition() + i + 1) % 40 + 1 == 2) {//if the player has passed the Start field
            if (p1.getFirstRound() == false && p1.getCurrentPosition() != 10) {
               p1.deposit(4000);
            } else {
               p1.setFirstRound(false);
            }
         }
      }
      assertFalse(p1.getBalance()>initBalance);
   }
}
```

Listing 41: TestTaxPercent

```java
package test;

import static org.junit.Assert.*;

import java.util.ArrayList;
import java.util.stream.IntStream;

import org.junit.Test;

import board.Street;
import entities.Board;
import entities.Cup;
import entities.Player;

public class TestTaxPercent {

   private Cup theCup = new Cup();
   private Board theBoard = new Board(theCup, null, false);
   ArrayList<Integer> housePrice = new ArrayList<Integer>();
   ArrayList<Integer> hotelPrice = new ArrayList<Integer>();
   int sumHouse;
   int sumHotel;
   ArrayList<Integer> propertyPrice = new ArrayList<Integer>();
   ArrayList<Integer> buildingPrice = new ArrayList<Integer>();
   int sumProperty = 0;
   int sumBuilding = 0;

   Player p1 = new Player("Test", 100000);

   @Test
   public void test() {

      // The test player buys squares and has the price withdrawn from their account

      p1.buySquare((Street) (theBoard.getSquare(1))); // Price 1200
      p1.buySquare((Street) (theBoard.getSquare(3))); // Price 1200
      p1.buySquare((Street) (theBoard.getSquare(6))); // Price 2000
      p1.buySquare((Street) (theBoard.getSquare(8))); // Price 2000
      p1.buySquare((Street) (theBoard.getSquare(11))); // Price 2800
      p1.buySquare((Street) (theBoard.getSquare(13))); // Price 2800
      p1.buySquare((Street) (theBoard.getSquare(14))); // Price 3200

      for (int i = 0; i < p1.getOwned().size(); i++) {

         p1.withdraw(p1.getOwned().get(i).getPrice());
         p1.getOwned().get(i).setOwner(p1);

      }

      // A total of 15200 should have been withdrawn from the test player's balance

      assertEquals(84800, p1.getBalance());

      // 7 pieces of property should be registered as owned by the test player

      assertEquals(7, p1.getOwnedStreet().size());

      // The test player constructs buildings on some of the streets

      ((Street) theBoard.getSquare(1)).setBuilding(5);
      ((Street) theBoard.getSquare(3)).setBuilding(2);
      ((Street) theBoard.getSquare(13)).setBuilding(4);
      ((Street) theBoard.getSquare(14)).setBuilding(5);

      // Checks the total number of buildings the player now owns

      int[] buildings = new int[p1.getOwnedStreet().size()];

      for (int i = 0; i < p1.getOwnedStreet().size(); i++) {

         buildings[i] = p1.getOwnedStreet().get(i).getNumberOfBuildings();

      }

      int totalBuildings = IntStream.of(buildings).sum();

      // The total amount should be 16

      assertEquals(16, totalBuildings);

      p1.withdraw(((Street) theBoard.getSquare(1)).getPriceOfBuilding()*5); // Price of 1 building: 1000
      p1.withdraw(((Street) theBoard.getSquare(3)).getPriceOfBuilding()*2); // Price of 1 building: 1000
      p1.withdraw(((Street) theBoard.getSquare(13)).getPriceOfBuilding()*4); // Price of 1 building: 2000
      p1.withdraw(((Street) theBoard.getSquare(14)).getPriceOfBuilding()*5); // Price of 1 building: 2000

      // A total of 25000 should have been withdrawn from the player as payment for the buildings

      assertEquals(59800, p1.getBalance());

      // The following code is taken from the method taxPercent from the class Tax. The calls to GUI are excluded

      for (int i = 0; i < p1.getOwned().size(); i++) {
         propertyPrice.add(p1.getOwned().get(i).getPrice());
      }
      for (int j = 0; j < p1.getOwnedStreet().size(); j++) {
         buildingPrice.add(p1.getOwnedStreet().get(j).getNumberOfBuildings()
               * p1.getOwnedStreet().get(j).getPriceOfBuilding());
      }
      for (int k = 0; k < propertyPrice.size(); k++) {
         sumProperty += propertyPrice.get(k);
      }
      for (int l = 0; l < buildingPrice.size(); l++) {
         sumBuilding += buildingPrice.get(l);
      }

      // The property value should be 15200 and the building value should be 25000

      assertEquals(15200, sumProperty);
      assertEquals(25000, sumBuilding);

```

```
111         int taxAmount = (p1.getBalance() + sumProperty + sumBuilding) / 10;
112
113         // As the test player started with 100000 in cash and have only bought property and buildings, the tax amount should be 10000
114
115         p1.withdraw(taxAmount);
116
117         // The test player's remaining funds should now be 49800
118
119         assertEquals(49800, p1.getBalance());
120
121     }
122 }
```

12 Litterature

References

[1] Craig Larman, Applying UML and Patterns 2004.

[2] Lewis and Loftus Java Software solutions 7th ed.

[3] Jonas Larsen, Mathias Tværmose Gleerup, Mikkel Geleff Rosenstrøm, Morten V. Christensen, Simon Lundorf og Sofie Freja Christensen, CDIO3.