Projects jan 2017

02312-14 Indledende programmering

Project name: *CDIO Final*

Group number: *14*

Due date: *Monday, 16/01 2017 at. 12:00*

Git repository: https://github.com/ldy985/14_final

Conceive Develop Implement Operate project (CDIO) Assignment Final: Matador game

Group 14

Rasmus Blichfeldt	s165205
Casper Bodskov	s165211
Lasse Dyrsted	s165240
Michael Klan	s144865
Mathias Larsen	s137055
Timothy Rasmussen	s144146

Danmarks Tekniske Universitet DTU

Supervisors:

Stig Høgh

Mads Nyborg

Ronnie Dalsgaard

Mathias Petersen

Time Table & Tasks

For detailed timetable see appendix 4.

Task	Sub total (h)	Name	Sub total (h)
Requirements	16.5	Timothy	56
Analysis	63	Lasse	79
Design	50.5	Rasmus	68.5
Implementation	143.5	Mathias	72.5
Test	35.5	Michael	59
Documentation	91	Casper	67
Total	402		402

The timetable serves to reflect the time allocated to the code and report by each individual member and in which tasks the time was spent. It also depicts the inadequate time spent by the different group members. This is caused by the disease and lack of time since it inhibited the indisposed from doing major amounts of work and therefore hasn’t been adding time to their total time spent.

Abstract

This rapport will serve as documentation of the development process of a Matador game that was set as requirement for the CDIO final project.

The board game will be made up of 40 fields, 2 dices and the option to be played by 2-6 players. Almost all the game mechanics from the actual Matador game is implemented in the finished program. The players will have the options to buy the different kinds of territories, buy houses/hotels, use chance cards and get thrown into jail. Other than that, there will be fields with preset effect meaning they will either add or subtract points (taxation or flat amount) from the player’s account, aswell landing on an owned field will have a rent scaled upon the amount of houses/Hotels owned by said player.

The entire assignment abided by the FURPS+, UP, and GRASP principles. The report also attempted to follow the assignment’s required formalities.

In conclusion the product was finished, the various parts of the program were tested and the requirements set from the assignment were met. However not all the desired iterations were completed since some features are missing from the fourth and fifth iteration.

Table of Content

1. Introduction

The purpose of the assignment is to show that we master the 4 phases of CDIO as well as the models and knowledge from lectures to fulfill a hypothetical customer's requested system, and be able to test and document the source code and program.
The 3 week project is solved by a 6-man developer team with the purpose to create a working "Matador" game in danish. The game consists of a gameboard with 40 fields, each with its unique game mechanic. The game does have an interface that serves as a border between user and source code which makes it possible for the user to interact with the program by drop-down menus and buttons, accomplished by the GUI we received for the earlier CDIO projects. The whole project is developed on Github with the GitKraken client. The code were written in object oriented Java with the IDE Intellij idea.

2. Analysis

The analysis were made as an iterative process therefore all sub chapters have been updated throughout the assignment with the exception of the diagrams. Due to time restrictions most of the diagrams were not updated to a final design version of the exact code. The analysis of crucial features were made at the beginning of the project. Then those features were moved to the design phase. Thereafter tests and code were written. After the first release that contained the features up to and including 2nd priority we started the next iteration of features. The analysis were primarily used to ensure that all members had a common vision and grasp of the project, and how it should proceed. The design and analysis were also a starting point for writing the code, but changes were made during the coding of program making the analysis obsolete for it's starting purpose but great for showing the iterative procedure. The rules of the game follows the rules outlined in the ruleset given with this project and can be found in appendix 5.

2.1. Requirements specification

1. **Player**
    - 1.1. The game shall be playable by 2-6 players.
    - 1.2. When a player has no more points he is out of the game.
    - 1.3. The player shall continue from the field he landed on the previous turn.
    - 1.4. The player shall have access to an account.
    - 1.5. The player shall have a name.

2. **Gameboard**
    - 2.1. The gameboard shall contain all the fields.

3. **Number of fields**
    - 3.1. The board shall include the following types of fields:
        - 3.1.1. 6 Chance fields
        - 3.1.2. 1 Jail field
        - 3.1.3. 1 Go to jail field
        - 3.1.4. 28 Ownable property fields:

3.1.4.1. 4 Fleet fields

3.1.4.2. 2 Brewery fields

3.1.4.3. 22 territory fields

3.1.5. 1 Start field

3.1.6. 2 Tax fields

3.1.7. 1 Free parking field

3.2. Field types

3.2.1. Jail

3.2.1.1. If the player is jailed, they shall not be able to move until he gets out of jail.

3.2.1.2. If the player lands on the jail field they are not jailed .

3.2.1.3. A player shall be able to get out of jail by either:

3.2.1.3.1. Paying 1000 points.

3.2.1.3.2. Rolling two of the same face value.

3.2.1.3.3. Use a get out of jail card.

3.2.1.4. When a player is jailed he shall not receive rent.

3.2.1.5. The jail field shall hold an array of players who is jailed.

3.2.2. Go the jail

3.2.2.1. When a player lands on the go to jail field, that player shall move to the jail field and be jailed.

3.2.3. Start

3.2.3.1. The start bonus shall be 4000 points

3.2.3.2. The player shall receive the start bonus when passing or landing on the start field.

3.2.3.3. All players begin the game on the start field.

3.2.4. Ownable

3.2.4.1. If the field is not owned, the player shall be able to buy the field for the price specified in appendix 2.

3.2.4.2. If the field is owned, the rent specified in appendix 2 shall be transferred to the owner .

3.2.4.3. Ownable fields shall be grouped with 2-4 properties in each group as specified in appendix 2.

3.2.5. Territory

3.2.5.1. If the player owns all fields in the group it shall be possible to buy a house on the field the player has landed on.

3.2.5.1.1. Houses and hotels shall be evenly distributed across the fields in a group.

3.2.5.1.2. If the field has 4 houses on it, it shall be possible to purchase a hotel.

3.2.5.1.3. When a hotel is purchased all the houses on the field shall be replaced with a hotel.

3.2.5.1.4. A player cannot buy more houses on a field that has a hotel.

3.2.5.2. If the player owns all fields of a group the initial rent (only) is doubled.

3.2.5.3. Groups of properties shall be color-coded.

3.2.6. Fleet

3.2.6.1. The rent is doubled for each additional fleet the player owns.

3.2.7. Brewery

3.2.7.1. Rent shall be paid by rolling the dices and multiplying that number by 100 points.

3.2.7.2. If both breweries are owned by the player the rent shall be multiplied with 200 points instead of 100.

3.2.8. Chance

3.2.8.1. When a player lands on a chance field, the player draws a chance card and follows the instructions on the card..

3.2.9. Free parking

3.2.9.1. When a player lands on free parking they shall receive an extra turn.

3.2.10. Tax

3.2.10.1. If a player lands on a tax field he shall pay the specified amount.

3.2.11. All fields shall have a name.

3.3. Pawning

3.3.1. It shall be possible to pawn of a field if a player otherwise would go bankrupt.

3.3.2. It shall be possible to sell a house or hotel at half the buying price if the player otherwise would go bankrupt.

3.3.3. The player cannot pawn off a field that has a house or hotel on it.

3.3.4. A player shall be able to buy back their pawned property for half the buying price plus 10% extra, only when landing on said property.

4. Dice

4.1. There shall be 2 six sided dice.

4.2. The player shall receive an extra turn if he rolls 2 of a kind.

4.3. If a player rolls 2 of a kind 3 times in a row, the player does not receive an extra turn but goes directly to jail.

4.4. The dice shall roll randomly according to statistics as calculated in appendix 1

5. Account

5.1. Each player shall have an account.

5.2. The account shall have a starting balance of 30.000 points.

5.3. The account shall store information about the player's points.

6. GUI

6.1. The provided GUI shall be used to display and play the game.

7. Chance deck

7.1. There shall be a chance card deck with the properties described in appendix 3.

7.2. The player shall be able to save “get-out-of-jail” card for later.

8. Non-functional requirements

8.1. The program shall be runnable on the Windows operating system.

8.2. The program shall run on the computers available in the databars on DTU campus Lyngby.

8.3. The program shall be playable by anyone with basic knowledge of computers and the capability to push a button.

8.4. Everything in the report as well as the code shall be written in English (UK).

8.5. The program shall be written in UTF-8.

8.6. The project shall be named 14_final

8.7. The source files shall be split up in appropriate sub folders.

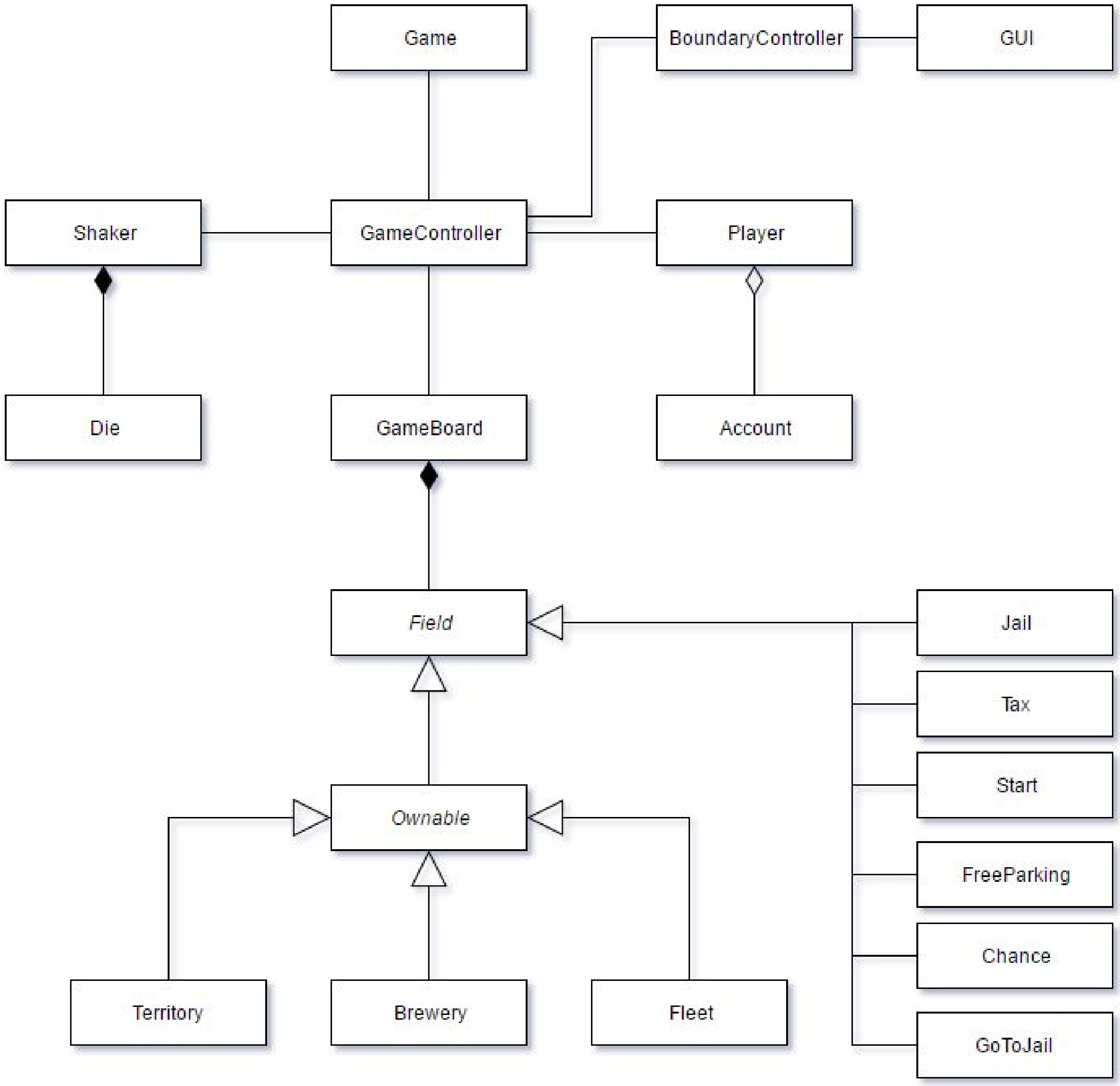
8.8. All work on the code of the program shall be done in coherence with Github and commits shall be done to the designated repository:

   8.8.1. Once every hour

   8.8.2. Every time a smaller objective or piece of code is finished.

8.9. The program shall be importable to eclipse.

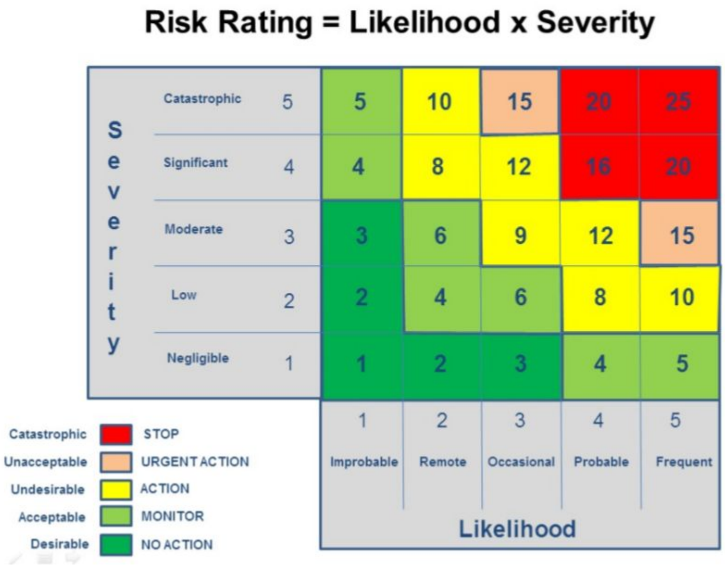
2.2. Domain model



We used the domain model which gives us an overview of the classes and their relations to each other. Throughout the initial analysis we had deducted that the classes presented in the diagram is the best logical way to distribute responsibilities in our program compared to the requirements analysis. We had tried to base our design on sub controllers and one main controller to merge the features and functionalities of our program.

2.3. Risk Analysis

Before we started programming, we had to make a risk analysis of the different objects that had to be added to our source code. It helps prioritizing the more difficult and important parts of the program to a achieve a working source code.



Risk Rating = Likelihood x Severity

Severity						
Catastrophic	5	5	10	15	20	25
Significant	4	4	8	12	16	20
Moderate	3	3	6	9	12	15
Low	2	2	4	6	8	10
Negligible	1	1	2	3	4	5
		1	2	3	4	5
		Improbable	Remote	Occasional	Probable	Frequent
		Likelihood				

Catastrophic	🟥	STOP
Unacceptable	🟧	URGENT ACTION
Undesirable	🟨	ACTION
Acceptable	🟩	MONITOR
Desirable	🟩	NO ACTION

We will be using this model to rate the different code assignments as well how high the impact will be on the finished product.

- **C** - Critical (Affects all project functionalities and baselines).
- **H** - High (Affect stakeholders needs and major product functionalities).
- **M** - Medium (These risks are subject to contingency but most of the times, a mitigation plan will be established in order to avoid the risk).
- **L** - Low (Generally these are risk for which Risk Acceptance strategies will be held or quick mitigation plan will be implemented. Usually the team will decide to live with the risk as a contingency).

<RSK-01> - Gameboard

Magnitude	Description	Impact	Mitigation Strategy / Contingency Plan
8	If we don’t have a working game board, we won’t have a working game.	C	Develop this in the early stage of the project, so we have time to test the functionality of the program.

<RSK-02> - Player Information & currency storage

Magnitude	Description	Impact	Mitigation Strategy / Contingency Plan
1	Players won’t be able to create an account, and won’t be able to win the game.	H	We will migrate the source code from CDIO Player and account.

<RSK-03> - GUI integration failure

Magnitude	Description	Impact	Mitigation Strategy / Contingency Plan
12	We won’t have an graphical interface for for the game.	M	We will need to try implement the GUI as soon as possible to be able to test and fix bugs.

<RSK-05> - Shaker & Die not implemented

Magnitude	Description	Impact	Mitigation Strategy / Contingency Plan
1	No way to move the players on the board.	C	We will reuse the shaker and die class from the CDIO 3 project, and add the new needed functionalities.

<RSK-06> - Missing Field Types data

Magnitude	Description	Impact	Mitigation Strategy / Contingency Plan
5	This will have a big impact on the flow of the game.	C	We will reuse many of the fields class from the CDIO 3 project, and add the new needed functionalities. The fields that was not part of CDIO 3 will have high priority and be written early in the process.

<RSK-07> - Chance Field not implemented

Magnitude	Description	Impact	Mitigation Strategy / Contingency Plan
5	This will have a big impact on the actual game, because of the missing information of the actual field.	C	Chance cards will be implemented in later in the process as the game is playable without any chance cards.

2.4. Feature list

The following is a prioritised list of features. The list was made on the basis of the analysis of the game mechanics that was carried out during the creation of the requirement specification and the risk analysis. All features of the 1st property were implemented before work was started on the 2nd priority features. This pattern was repeated until there were no more priorities.

We created this feature list as a more comprehensive way of showing what attributes the programmers should focus on making first. This is an effective way to limit the program if there’s a lack of time or if there’s points in the process where it’s necessary to have a product to show off. In our case we had milestone 2 where we had to show a functioning program. Limiting the code after each priority (freezing feature development when a priority is finished) made it possible to show the program's development in smaller chunks of functioning code instead of a few lines of “soon-to-be-implemented” or “TODO”s.

The 1st priority features were corrected and reused from CDIO 3. They are also critical in the sense that without them the program would not be a game. The 2nd priority features completes the game board and makes it possible to run the game. The completion of these marked our 1st release. The 3rd priority features makes the game recognizable as a Matador game. 4th and 5th priority features are rules that many players do not use or change to their own custom rules when playing the board game version.

Feature	Priority	implemented		Feature	Priority	implemented
Die	1	Yes		Chance field	2	Yes
Shaker	1	Yes		Jail field	2	Yes
Players	1	Yes		Go to jail field	2	Yes
Gameboard	1	Yes		Start field	2	Yes
Account	1	Yes		Free parking	2	Yes
Ownable fields	1	Yes		GameController	2	Yes
Territory fields	1	Yes		GUI	2	Yes
Tax fields	1	Yes		Language	2	Yes

Brewery	1	Yes
Fleet field	1	Yes

Boundary Controller	2	Yes

Feature	Priority	implemented
Chance deck	3	Yes
Buying house/hotels	3	Yes
Houses evenly distributed	3	Yes
Rolling double gives extra turn	3	Yes
Pass start bonus	3	Yes
Limit houses and hotels	3	Yes
No rent when jailed	3	Yes

Feature	Priority	implemented
Auction	5	No

Feature	Priority	implemented
Selling houses/hotels	4	No
Pawning territories	4	No
Rolling 2 of the same value 3 times jails the player	4	Yes
Bankrupt transfers all owned fields to the creditor but they are still pawned	4	No
If a player owns all territories in a group, base rent is doubled	4	Yes
Color-coded fields	4	Yes

2.5. System requirements

Because of our program's simple nature, the program only requires that the systems specifications matches up with the specified system requirements for java 8, Java JRE (Java Runtime Environment), and the supplementing files. Java 8 and Java JRE must be installed.

Version of windows	Windows vista SP1 or newer.
RAM	128 MB
Disk Space	126 MB
Processor	Pentium 2 266 MHz or newer.

2.6. The need of new features

Since the software development in CDIO 3 our skills have improved. We now have new creative ways for solving the problems in our our project. This and the increased complexity of the program force us to change some of the classes and program structure from the CDIO 3 project. Some of our old classes have had simple solutions that wouldn't allow expanding the program. This resulted in some major changes to some classes and minor or no changes to classes with little responsibility like the die class. Other than the changed classes from CDIO we also realized that a lot of the new features we would like to implement required more classes. Our coding speed also increased resulting in our classes getting completed faster and more efficiently meaning we can spend more time on more delicate parts of the game (eg. Chance cards and the ability to buy houses).

2.7. Thoughts / process

Class responsibility: We debated the distribution of the responsibility of methods and attributes among the classes in our program. Ultimately, we decided, for the most part, to leave the responsibility of what happens when landing on the field to the fields (and therefore also the classes) themselves. An example of this would be the Territory fields. They are the only fields that can buy houses, however we decided to leave the responsibility to calculating whether or not a house should be bought or not, to the GameBoard class, because that ties in with it's responsibility to handle what happens on the board, behind the scenes. This made us rewrite the landOnField() method in the Territory class, and to add the appropriate methods in our GameBoard class.

Making field/card objects: An issue we ran into early on in our last project was how to make an array of Field objects, by not hardcoding every object instantiation in the code. Our solution was to use Gson, a modified version of Json, to parse a text file with all the Field's data in it, and then make the objects, saving a lot of code-lines. This worked, and we wanted to use the same feature when we wanted an array of Card objects in this project. We realised, however, that we were not capable of explaining how Gson made objects without using constructors. So to avoid using code we couldn't explain ourselves, we decided to not use a filereader, and instantiated every field and every card object in the code. An example of this in code can be seen below. (Copied from GameBoard.class)

```
board[0] = new Start("Start", 1, Color.red);
board[1] = new Territory("Rødovervej", 2, Color.blue, 1200, new int[]{50, 250, 750, 2250, 4000, 600}, 1000);
board[2] = new Chance("Prøv lykken", 3, Color.black);
board[3] = new Territory("Hvidovervej", 2, Color.blue, 1200, new int[]{50, 250, 750, 2250, 4000, 600}, 1000);
board[4] = new Tax("Indkomstskat", 4, Color.lightGray, 4000, 0.1f);
board[5] = new Fleet("Scandlines", 5, Color.gray, 4000, new int[]{500, 1000, 2000, 400});
board[6] = new Territory("Roskildevej", 6, Color.orange, 2000, new int[]{100, 600, 1800, 5400, 8000, 1100}, 1000);
```

As seen in the our code above, we "manually" instantiate the objects in an array of Fields.

2.8. Class responsibility

2.8.1. Game folder

Class	Responsibility	Connections
Account	Keeps track and alters the balance of the players.	Player

Class	Responsibility	Connections
Boundary-Controller	Chooses whether the program is run in a test mode or with the GUI.	GUI GameController

Class	Responsibility	Connections
Die	Generate a random value between 1 - 6.	Shaker

Class	Responsibility	Connections
Game	Selects the language and starts the game.	GameController

Class	Responsibility	Connections
GameBoard	Holds all the fields used in the game. Creates the chance card deck. Determines if the player can buy houses and hotels.	Field Shaker ChanceDeck

Class	Responsibility	Connections
Game Controller	Controls the main flow of the game.	BoundaryController GameBoard Language Player Game

Class	Responsibility	Connections
Language	Fetches the selected language.	GameController

Class	Responsibility	Connections
Player	Stores the players’ names. Creates an account object for each player. Stores a boolean for an extra turn Stores how many “Get out of jail”-cards the player has. Stores the players’ positions.	GameController Account

Class	Responsibility	Connections
Shaker	Creates two die objects. Store how many doubles that has been rolled in a row. Store their facevalues.	Die GameController

2.8.2. ChanceCard folder

Class	Responsibility	Connections
ChanceCard	Defines common attributes and methods for all chance card types.	ChanceDeck MoveCard MoneyCard JailCard ToJailCard GoToNearestFleet TotalValueCard

Class	Responsibility	Connections
ChanceDeck	Stores all the chance card array.	ChanceCard GameController

Class	Responsibility	Connections
MoneyCard	Stores the amount of money that will be transferred to or from the player if specific cards are drawn.	ChanceCard

Class	Responsibility	Connections
MoveCard	Stores the amount of fields that the player will move if a specific card type is drawn.	ChanceCard

Class	Responsibility	Connections
ToJailCard	Moves the player to jail	ChanceCard

Class	Responsibility	Connections
JailCard	Its existence serves as a get-out-of-jail card	ChanceCard

Class	Responsibility	Connections
TotalValueCard	Gives the player 40000 if their total assets are below 15000.	ChanceCard

Class	Responsibility	Connections
GoToNearestFleet	Moves the player to the nearest fleet field	ChanceCard

2.8.3. Fields folder

Class	Responsibility	Connections
Brewery	Determines the rent of a specific field	Ownable

Class	Responsibility	Connections
Chance	Makes the player draw a chance card.	Field ChanceDeck

Class	Responsibility	Connections
Field	Defines common attributes and methods for all the field types.	Ownable FreeParking Start Chance Jail GoToJail Tax

Class	Responsibility	Connections
Fleet	Determines the rent of a specific field	Ownable

Class	Responsibility	Connections
FreeParking	Gives the player an extra turn.	Field

Class	Responsibility	Connections
GoToJail	Puts the player in jail.	Field

Class	Responsibility	Connections
Jail	Stores which players are jailed and adds players if they are jailed.	Field

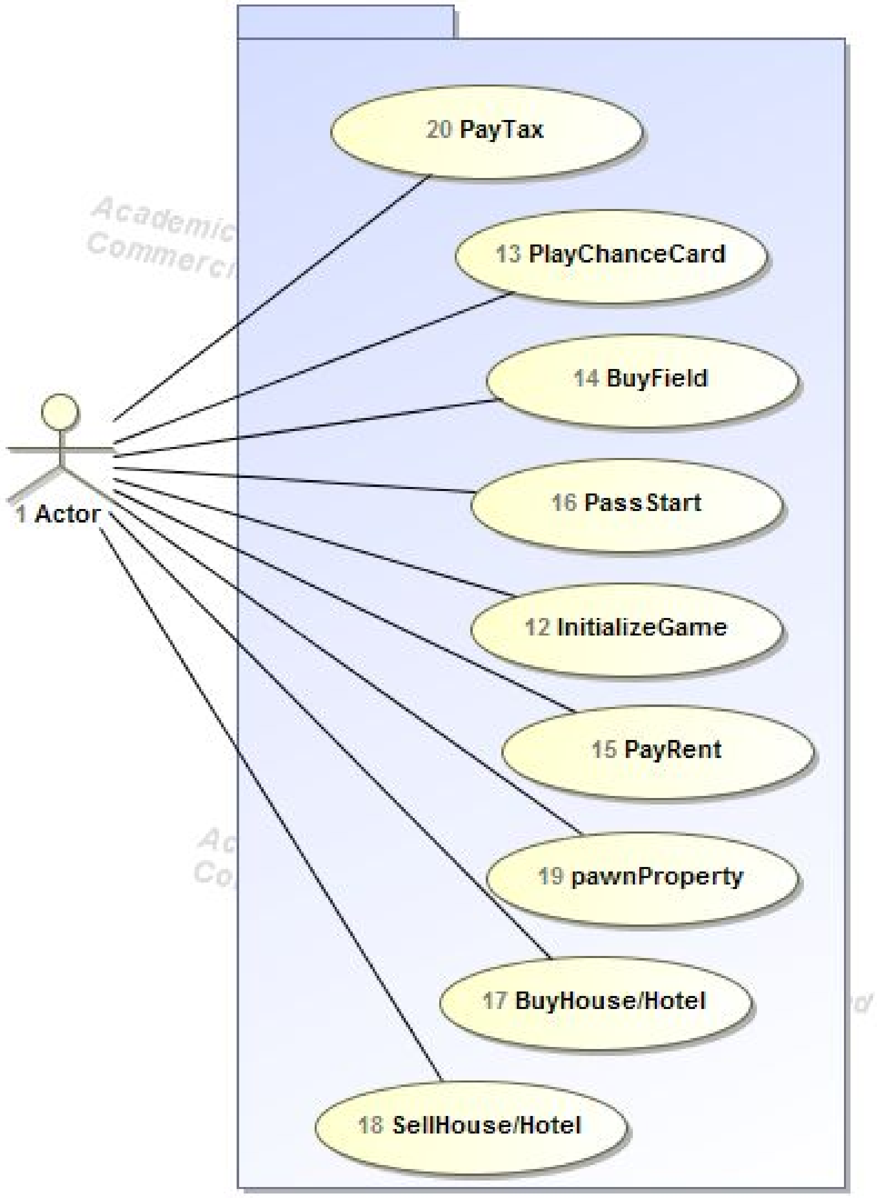
Class	Responsibility	Connections
Ownable	Handles the owner, rent and price of fields as well as pawn pricing.	Field Brewery Fleet Territory

Class	Responsibility	Connections
Start	Gives the player 4000 points when passed.	Field

Class	Responsibility	Connections
Tax	Determines the rent of a specific field.	Field

Class	Responsibility	Connections
Territory	Stores number of houses/hotels and group number.	Ownable

2.9. Use case diagram



The use case diagram shows how interactive the program is and how the user affects the program. (The actor’s interaction with the program) This particular diagram is relatively simple since there is only a single actor. The use case descriptions will explain more in-depth as to what happens during the interaction whereas the diagram is more simple and only serves to give an overview. The use case diagram is a static model since it has no runtime depiction and only shows how the user interacts with the program. The use case descriptions however has a flow and could be described as dynamic despite not being models.

2.10. Use case descriptions

Name	Buy Field
Identifier	UC1
Description	How a player buys an unowned field
Primary actors	Player
Secondary actors	None
Preconditions	Player lands on an ownable field that is not owned by other players.
Main flow	1. The system validates the the player have enough points to buy the field. 2. The system asks the player if she/he want to buy the field she/he has just landed on. 3. The player presses the buy button. 4. The system transfers the amount the field costs from current players account. 5. The field stores that it is now owned by the player. 6. The system prints the players points and owned fields. 7. Player presses the OK button.
Postconditions	The next player's turn start.
Alternative flow	None.

Name	Initialize game
Identifier	UC2
Description	Initializes and sets up the game.
Primary actors	Player 1-6
Secondary actors	None
Preconditions	A player executes an instance of the Matador Game
Main flow	1. System prints “How many players in this game?”. 2. A player selects the amount of players in a dropdown menu. 3. System prints “the players name?” 4. A player enters his/her name. 5. Repeat step (3.) and (4.) for all players.
Postconditions	The game begins with player 1’s turn.
Alternative flow	None.

Name	Buy house/hotel
Identifier	UC3
Description	What happens when a player chooses to buy a house.
Primary actors	Player.
Secondary actors	None.
Preconditions	Player lands on a field they owns.
Main flow	1. The system checks if the player owns every property in the group. 2. The system checks if the player can afford to buy a house. 3. The system checks if there is houses or hotels available. 4. The system checks which fields in the group have fewest houses. 5. System prints “do you want to buy a house on this field?” 6. Player chooses yes. 7. System prints “Where do you want to buy a house?” thereafter presenting a button for each field that have the lowest amount of houses. 8. Player chooses one. 9. Points are deducted from the player’s account and a house is added to the field. 10. Repeat step 1-9 until the player choses not to buy any more houses or no more houses/hotels are available. 11. Player presses the OK button.
Postconditions	The next player's turn starts.
Alternative flow	None.

Name	Play chance card
Identifier	UC4
Description	What happens when a player has drawn a chance card
Primary actors	Player
Secondary actors	None
Preconditions	Player has landed on a chance field.
Main flow	1. The system draws a card for the player and displays it. 2. The system follows these instructions (example: moves to a field, is sent to jail, receives money, pays money.) 3. The system then prints out what has happened 4. The player presses the “OK” button.
Postconditions	The next player's turn start.
Alternative flow	None.

Name	Pass start
Identifier	UC5
Description	What happens when a player passes or lands on the start field
Primary actors	Player
Secondary actors	None
Preconditions	Player lands on, or passes the start field.
Main flow	1. The system checks if the field number the player will land on is higher than 40. 2. If It is, then 40 is subtracted from the number. 3. The player moves to the field corresponding with the now calculated number. 4. The player receives 4000 points
Postconditions	The player follows the instructions for the field they are now standing on.
Alternative flow	None.

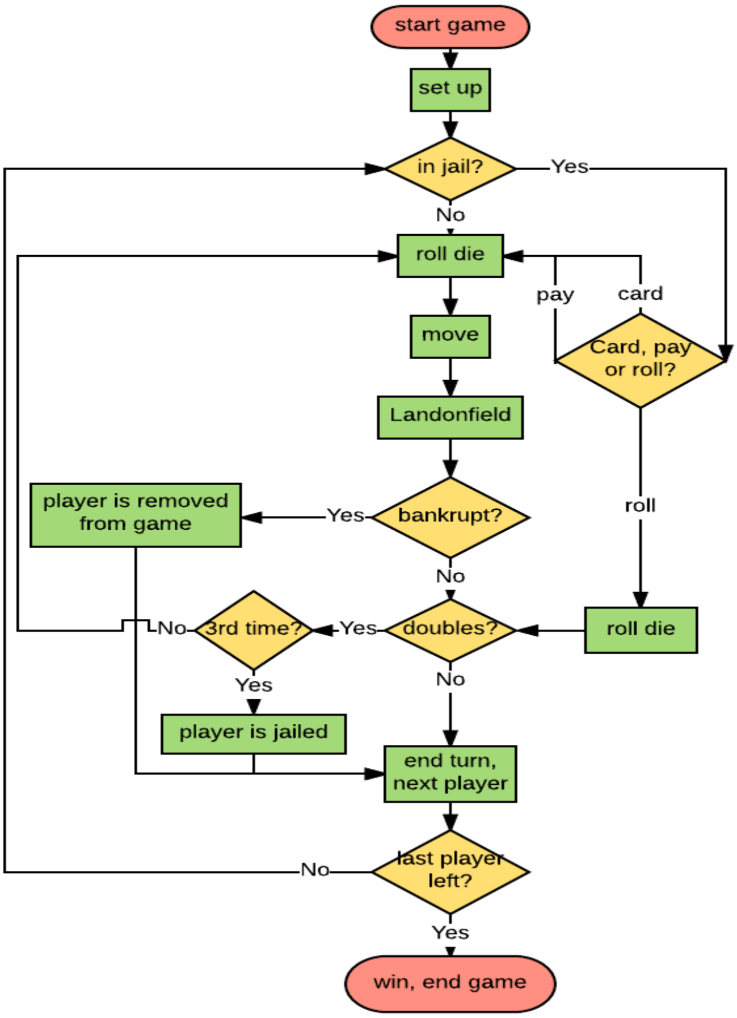
Name	Pay Rent
Identifier	UC6
Description	How a player transfers points to another player, when the player lands on an owned field.
Primary actors	Player.
Secondary actors	None.
Preconditions	The player lands on an owned field.
Main flow	1. System transfers the rent amount to the owner of the field. 2. System prints the rent amount, owner of the field and balance before and after the transfer. 3. If current player's account is less than 0 3.1. Player is removed from game. 4. Player presses the OK button.
Postconditions	The next player's turn starts.
Alternative flow	None.

Name	Pay tax
Identifier	UC7
Description	How a player pays tax
Primary actors	Player
Secondary actors	None.
Preconditions	Player has landed on a tax field.
Main flow	1. The system prints “do you want to pay 4000 or 10% of what you own?” 2. The player chooses 10% 3. The system calculates 10% of the players asset value and deducts it from the player’s account. 4. Player presses the OK button.
Postconditions	The next player's turn starts.
Alternative flow	None.

Name	Pawn property
Identifier	UC8
Description	How the players can pawn off their property to avoid going bankrupt
Primary actors	Player
Secondary actors	None.
Preconditions	The player has run out of points and has sold the houses/hotel on the field(s) they wish to pawn
Main flow	1. The player has to pay an amount of points and can not afford it. 2. The system asks the player which of he's properties he wants to pawn off in order not to go bankrupt. 3. The player chooses 1 or more properties to pawn off, until their account balance is no longer negative. 4. The player receives the money and the selected properties are marked as pawned. 5. The player presses the "OK" button.
Postconditions	The next player's turn starts.
Alternative flow	None.

Name	Sell house/hotel
Identifier	UC9
Description	How the players can sell their houses/hotels to avoid going bankrupt
Primary actors	Player
Secondary actors	None.
Preconditions	The player has run out of points
Main flow	1. The player has to pay either the bank or another player and can’t afford it. 2. The system asks the player which of their houses/hotels they wish to sell in order not to go bankrupt. 3. The player chooses 1 or more house or hotel to sell, until their account balance is no longer negative. 4. The player receives the money and the selected houses/hotels are removed from the board and returned to the bank. 5. The player presses the “OK” button.
Postconditions	The next player's turn starts.
Alternative flow	None.

2.11. Flowchart



The flowchart shows the flow of all possible branches in the program. The flowchart thereby outlines the logic of the program in a clear manner. The game start is presented as a problem and reaching the end (In our case “winning the game”) is the solution. The red general circle are the “terminators” and are where the flowchart begin and ends. The green square are the actions that affect the player in the game performed by the system. The yellow squares represents the if-statements in our program. The flowchart is a dynamic model since it shows the runtime of the system.

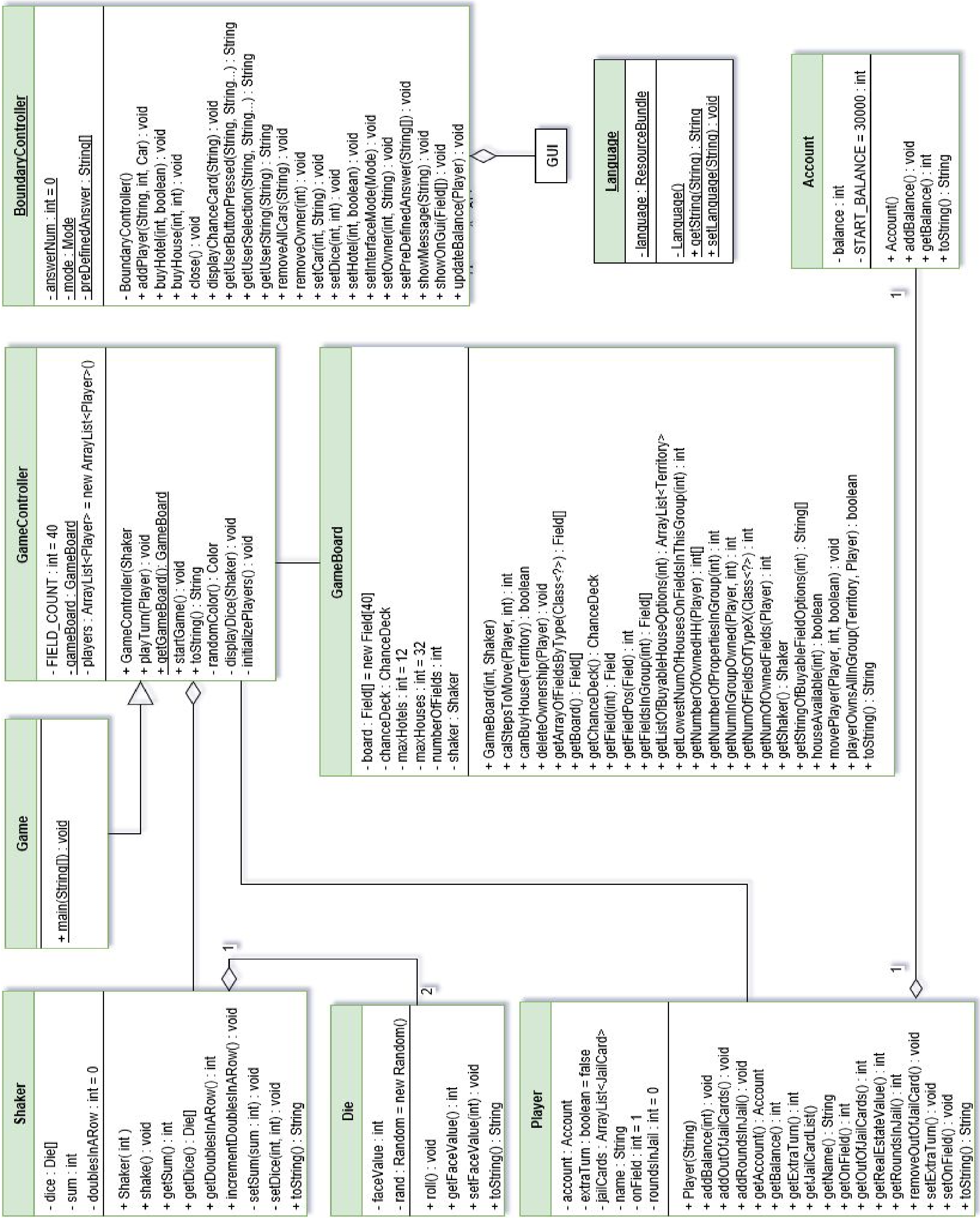
3. Design

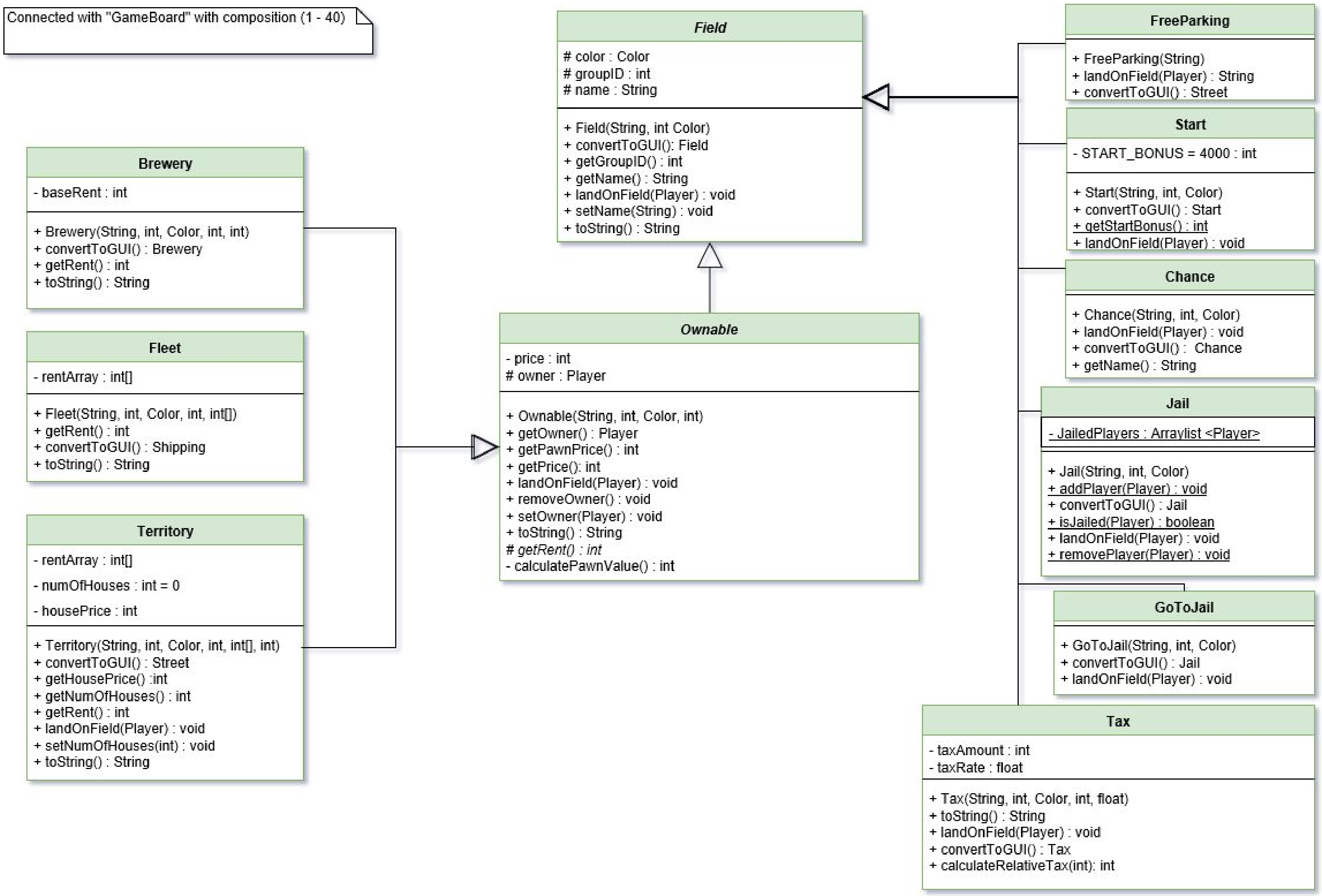
Content in the design chapter reflect the finished code, as the program is not completed and due to time restraints, this chapter is rather short. We chose to focus our efforts on implementing as many features a possible, therefore it was decided to not make a lot of the content that would normally be found in the design chapter, f.ex. Design sequence diagrams.

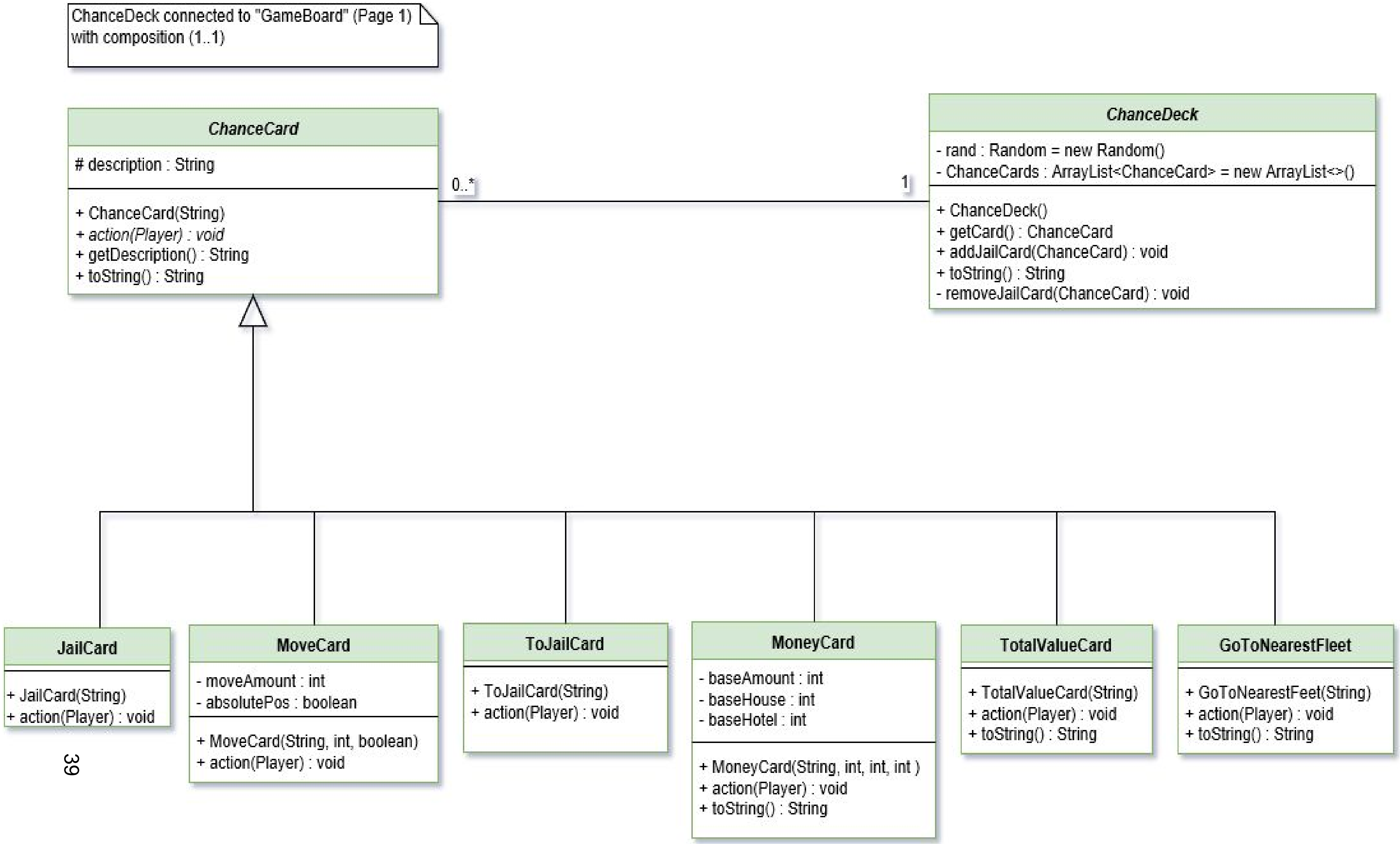
3.1. Boundary controller

We made the boundary controller to overcome the problem of testability. When we want to test the game, some of the game mechanics are dependent on user action and that is not possible in an automated test. The boundary acts like a wall between the interface and the program, which enables us to intercept a call to the UI and return a predefined answer. In the test, it makes us able to set what the the UI-calls return without anything popping up during the test. The boundary controller also makes it easy to change to another UI at a later point.

3.2. Design Class Diagram


BoundaryController
- answerNum : int = 0
- mode : Mode
- preDefinedAnswer : String[]
+ BoundaryController()
+ addPlayer(String, int, Car) : void
+ buyHotel(int, boolean) : void
+ buyHouse(int, int) : void
+ close() : void
+ displayChanceCard(String) : void
+ getUserButtonPressed(String, String...) : String
+ getUserSelection(String, String, String...) : String
+ getUserString(String) : String
+ removeAllCars(String) : void
+ removeOwner(int) : void
+ setCar(int, String) : void
+ setDice(int, int) : void
+ setHotel(int, boolean) : void
+ setInterfaceMode(Mode) : void
+ setOwner(int, String) : void
+ setPreDefinedAnswer(String[]) : void
+ showMessage(String) : void
+ showOnGui(Field[]) : void
+ updateBalance(Player) : void
GUI
Language
- language : ResourceBundle
- Language()
+ getString(String) : String
+ setLanguage(String) : void
Account
- balance : int
- START_BALANCE = 30000 : int
+ Account()
+ addBalance() : void
+ getBalance() : int
+ toString() : String
1
GameController
- FIELD_COUNT : int = 40
- gameBoard : GameBoard
- players : ArrayList<Player> = new ArrayList<Player>()
+ GameController(Shaker
+ playTurn(Player) : void
+ getGameBoard() : GameBoard
+ startGame() : void
+ toString() : String
- randomColor() : Color
- displayDice(Shaker) : void
- initializePlayers() : void
GameBoard
- board : Field[] = new Field[40]
- chanceDeck : ChanceDeck
- maxHotels : int = 12
- maxHouses : int = 32
- numberOfFields : int
- shaker : Shaker
+ GameBoard(int, Shaker)
+ calStepsToMove(Player, int) : int
+ canBuyHouse(Territory) : boolean
+ deleteOwnership(Player) : void
+ getArrayOfFieldsByType(Class&lt;?&gt;) : Field[]
+ getBoard() : Field[]
+ getChanceDeck() : ChanceDeck
+ getField(int) : Field
+ getFieldPos(Field) : int
+ getFieldsInGroup(int) : Field[]
+ getListOfBuyableHouseOptions(int) : ArrayList<Territory>
+ getLowestNumOfHousesOnFieldsInThisGroup(int) : int
+ getNumberOfOwnedHH(Player) : int[]
+ getNumberOfPropertiesInGroup(int) : int
+ getNumInGroupOwned(Player, int) : int
+ getNumOfFieldsOfTypeX(Class&lt;?&gt;) : int
+ getNumOfOwnedFields(Player) : int
+ getShaker() : Shaker
+ getStringOfBuyableFieldOptions(int) : String[]
+ houseAvailable(int) : boolean
+ movePlayer(Player, int, boolean) : void
+ playerOwnsAllInGroup(Territory, Player) : boolean
+ toString() : String
Game
+ main(String[]) : void
Shaker
- dice : Die[]
- sum : int
- doublesInARow : int = 0
+ Shaker(int)
+ shake() : void
+ getSum() : int
+ getDice() : Die[]
+ getDoublesInARow() : int
+ incrementDoublesInARow() : void
- setSum(sum : int) : void
- setDice(int, int) : void
+ toString() : String
1
2
Die
- faceValue : int
- rand : Random = new Random()
+ roll() : void
+ getFaceValue() : int
+ setFaceValue(int) : void
+ toString() : String
Player
- account : Account
- extraTurn : boolean = false
- jailCards : ArrayList<JailCard>
- name : String
- onField : int = 1
- roundsInJail : int = 0
+ Player(String)
+ addBalance(int) : void
+ addOutOfJailCards() : void
+ addRoundsInJail() : void
+ getAccount() : Account
+ getBalance() : int
+ getExtraTurn() : int
+ getJailCardList()
+ getName() : String
+ getOnField() : int
+ getOutOfJailCards() : int
+ getRealEstateValue() : int
+ getRoundsInJail() : int
+ removeOutOfJailCard() : void
+ setExtraTurn() : void
+ setOnField() : void
+ toString() : String
1


Connected with "GameBoard" with composition (1 - 40)
Field
# color : Color
# groupID : int
# name : String
+ Field(String, int Color)
+ convertToGUI(): Field
+ getGroupID() : int
+ getName() : String
+ landOnField(Player) : void
+ setName(String) : void
+ toString() : String
Ownable
- price : int
# owner : Player
+ Ownable(String, int, Color, int)
+ getOwner() : Player
+ getPawnPrice() : int
+ getPrice(): int
+ landOnField(Player) : void
+ removeOwner() : void
+ setOwner(Player) : void
+ toString() : String
# getRent() : int
- calculatePawnValue() : int
Brewery
- baseRent : int
+ Brewery(String, int, Color, int, int)
+ convertToGUI() : Brewery
+ getRent() : int
+ toString() : String
Fleet
- rentArray : int[]
+ Fleet(String, int, Color, int, int[])
+ getRent() : int
+ convertToGUI() : Shipping
+ toString() : String
Territory
- rentArray : int[]
- numOfHouses : int = 0
- housePrice : int
+ Territory(String, int, Color, int, int[], int)
+ convertToGUI() : Street
+ getHousePrice() :int
+ getNumOfHouses() : int
+ getRent() : int
+ landOnField(Player) : void
+ setNumOfHouses(int) : void
+ toString() : String
FreeParking
+ FreeParking(String)
+ landOnField(Player) : String
+ convertToGUI() : Street
Start
- START_BONUS = 4000 : int
+ Start(String, int, Color)
+ convertToGUI() : Start
+ getStartBonus() : int
+ landOnField(Player) : void
Chance
+ Chance(String, int, Color)
+ landOnField(Player) : void
+ convertToGUI() : Chance
+ getName() : String
Jail
- JailedPlayers : Arraylist <Player>
+ Jail(String, int, Color)
+ addPlayer(Player) : void
+ convertToGUI() : Jail
+ isJailed(Player) : boolean
+ landOnField(Player) : void
+ removePlayer(Player) : void
GoToJail
+ GoToJail(String, int, Color)
+ convertToGUI() : Jail
+ landOnField(Player) : void
Tax
- taxAmount : int
- taxRate : float
+ Tax(String, int, Color, int, float)
+ toString() : String
+ landOnField(Player) : void
+ convertToGUI() : Tax
+ calculateRelativeTax(int): int


ChanceDeck connected to "GameBoard" (Page 1)
with composition (1..1)
ChanceCard
# description : String
+ ChanceCard(String)
+ action(Player) : void
+ getDescription() : String
+ toString() : String
0..*
1
ChanceDeck
- rand : Random = new Random()
- ChanceCards : ArrayList<ChanceCard> = new ArrayList<>()
+ ChanceDeck()
+ getCard() : ChanceCard
+ addJailCard(ChanceCard) : void
+ toString() : String
- removeJailCard(ChanceCard) : void
JailCard
+ JailCard(String)
+ action(Player) : void
MoveCard
- moveAmount : int
- absolutePos : boolean
+ MoveCard(String, int, boolean)
+ action(Player) : void
ToJailCard
+ ToJailCard(String)
+ action(Player) : void
MoneyCard
- baseAmount : int
- baseHouse : int
- baseHotel : int
+ MoneyCard(String, int, int, int)
+ action(Player) : void
+ toString() : String
TotalValueCard
+ TotalValueCard(String)
+ action(Player) : void
+ toString() : String
GoToNearestFleet
+ GoToNearestFeet(String)
+ action(Player) : void
+ toString() : String

The design class diagram is an exact representation of the program. It is static like the domain model but far more detailed and contains all the information necessary to code the program from scratch. It is used to explain or show the program code more visually than looking in the source code without losing information about the specifics of the code. We had to divide the diagram since we had to print it on an A4 sheet. However the connections are showed as notes between the pages and is logically set up so that no information is lost.

4. Test

4.1. Test strategy

In the elaboration phase for each class a test scenario was written in great detail. In the implementation and test phases for the classes we have given each of the group members individual classes to create and another member was responsible for writing the test. This way the programmers have little partiality for their classes and therefore can make more critic and in-depth tests. By reducing the bias for our code pieces we make better and more thorough tests. Other than making the tests objectively better we also force the members to enhance their understanding of the code and the specific individual classes before they eventually review them in the classes' absolute final stage. We will continue this practise in future projects.

4.2. Test scenarios

Name	ShakerTest
Identifier	TS1
Description	A unit test of the class Shaker
Primary actors	System
Preconditions	Create a shaker and two die objects
Test scenario	Validate that... 1. the entities have been created and is of the right type 2. the shake method rolls the two die randomly according to statistisk. 3. sum of the dice are calculated correctly 4. two dice objects can be returned 5. the doublesInARow increments when the dice roll the same face value 6. doubelsInARow can be returned 7. the sum can be returned
Test case	In test folder class name ShakerTest
Postconditions	None

Name	PlayerTest
Identifier	TS2
Description	A unit test of the class Player
Primary actors	System
Preconditions	Create a player and an account object
Test scenario	Validate that the... 1. entities have been created and are of the correct type 2. real estate value can be changed and returned 3. player’s position can be changed and returned 4. player's name can be returned 5. account balance can be changed and returned
Test case	In test folder class name PlayerTest
Postconditions	None

Name	DieTest
Identifier	TS3
Description	A unit test of the class Die
Primary actors	System
Preconditions	Create a die object
Test scenario	Validate that the… 1. entities have been created and are of the correct type 2. die rolls randomly according the statistics 3. face value can be set and returned
Test case	In test folder class name DieTest
Postconditions	None

Name	AccountTest
Identifier	TS4
Description	A unit test of the class Account
Primary actors	System
Preconditions	Create an account object
Test scenario	Validate that the... 1. entity have been created and are of the correct type 2. starting balance is 30.000 points 3. balance can be changed and returned
Test case	In test folder class name AcccountTest
Postconditions	None

Name	GameBoardTest
Identifier	TS5
Description	A unit test of the class GameBoard
Primary actors	System
Preconditions	Create a gameboard including all the fields, a card deck and two player objects
Test scenario	Validate that the… 1. entity have been created and are of the correct type 2. fields are in the correct order according the the list in appendix 2 3. all variables for the fields corresponds to the description in appendix 2 and can be returned 4. field objects can be returned 5. field array can be returned 6. houses and hotels can be set and returned. 7. rent increments and returns the correct value according to to appendix 2. 8. all properties can remove ownership from a given player and only that player 9. number of properties in a group owned by a certain player can be set and returned 10. If the player owns all properties in a group 11. If the player needs to collect points from passing start 12. pawn price can be calculated, corresponding to the buy price and returned 13. all the chance cards does what is described in appendix 3
Test case	In test folder class name GameBoardTest
Postconditions	None

The GameBoardTest have not been fully implemented but we left it because what is written shows the process we would use the make most of the test i.e. using arrays with predifined test data.

Name	BreweryTest
Identifier	TS6
Description	A unit test of the class
Primary actors	System
Preconditions	Create a player, a shaker with two dice and two brewery objects
Test scenario	Validate that the... 1. entity have been created and are of the correct type 2. field name can be set and returned 3. buying price can be returned 4. owner can be set and returned 5. rent can be calculated and returned 6. pawn price can be calculated correctly and is half of the buy price
Test case	In test folder class name BreweryTest
Postconditions	None

Name	TerritoryTest
Identifier	TS7
Description	A unit test of the class Territory
Primary actors	System
Preconditions	Create a player and 3 territory objects
Test scenario	Validate that the... 1. entity have been created and are of the correct type 2. field name can be set and returned 3. buying price can be returned 4. house price can be returned 5. pawn price can be calculated correctly and is half of the buy price
Test case	In test folder class name TerritoryTest
Postconditions	none

Name	FleetTest
Identifier	TS8
Description	A unit test of the class Fleet
Primary actors	System
Preconditions	Create two player and 4 fleet objects
Test scenario	Validate that the... 1. entity have been created and are of the correct type 2. field name can be set and returned 3. buying price can be returned 4. owner can be set and returned 5. rent can be returned and corresponds to how many fleets the owner of the fleet that have been landed on owns 6. pawn price can be calculated correctly and returned
Test case	In test folder class name FleetTest
Postconditions	none

Name	TaxTest
Identifier	TS9
Description	A unit test of the class Tax
Primary actors	System
Preconditions	Create a player and two tax fields
Test scenario	Validate that the... 1. entity have been created and are of the correct type 2. field name can be set and returned 3. tax rate can be returned 4. relative tax amount can be calculated, dependent on the tax rate and player real estate value, and returned 5. tax amount can be returned
Test case	In test folder class name TaxTest
Postconditions	none

Name	StartTest
Identifier	TS10
Description	A unit test of the class
Primary actors	System
Preconditions	Create a start and player object
Test scenario	Validate that the... 1. entity have been created and are of the correct type 2. field name can be set and returned 3. start bonus is 4000 points and can be returned
Test case	In test folder class name StartTest
Postconditions	none

4.3. Test coverage analysis

Due to time restriction we chose to put effort into implementing features. Because of this only a few tests were written. This have resulted in only 25 of 56 functional requirements have been tested bringe to coverage percentage to 44.6%. However when each individual feature had been written they were debugged, validating that the code would compile and that the code did what it was supposed to do. After all 2nd priority features had been completed it was also validated that the whole program could run. When the program was runnable, the 2nd phase of debugging started where the game was played several times testing for bugs. Although most of the testing other than unit-testing were manual, we are confident that our game contains no game breaking bugs and most minor bugs have been found and corrected.

4.3.1. Traceability matrix

The traceability matrix only tracks automatic tests.

✓ = tested.

✓/ = partially tested

Requerment\Test	TS1	TS2	TS3	TS4	TS5	TS6	TS7	TS8	TS9	TS10
Player										
1.1										
1.2	✓									
1.3										
1.4	✓									
1.5	✓									
Gameboard										
2.1					✓					
Number of fields										
3.1.1					✓					
3.1.2					✓					
3.1.3					✓					
3.1.4.1					✓					
3.1.4.2					✓					
3.1.4.3					✓					
3.1.5					✓					
3.1.6					✓					
3.1.7					✓					
Jail										
3.2.1.1										
3.2.1.3.1										
3.2.1.3.2										
3.2.1.3.3										

3.2.1.4										
3.2.1.5										
Go to jail										
3.2.2.1										
Start										
3.2.3.1										✓
3.2.3.2										
3.2.3.3										
Ownable										
3.2.4.1							✓			
3.2.4.2										
3.2.4.3										
Territory										
3.2.5.1										
3.2.5.1.1										
3.2.5.1.2										
3.2.5.1.3										
3.2.5.1.4										
3.2.5.2							✓			
Fleet										
3.2.6.1								✓		
Brewery										
3.2.7.1						✓				
3.2.7.2						✓				
Chance										
3.2.8.1										
Free parking										
3.2.9.1										
Tax										

3.2.10.1									✓	
Fields										
3.2.11						✓/	✓/	✓/	✓/	
Pawning										
3.3.1										
3.3.2										
3.3.3										
3.3.4										
Dice										
4.1			✓							
4.2										
4.3										
4.4	✓		✓							
Account										
5.1		✓								
5.2		✓		✓						
5.3		✓		✓						
GUI										
6.1										
Chance deck										
7.1										
7.2										

4.4. Known bugs

1. It is possible to buy 6 houses instead of 5.
2. The method canBuyHouse in GameBoard class returns the wrong value.
3. When a player is jailed and rolles a double he is not released, but the land on field action of the field he was supposed to land on is executed.
4. If a player draws a chance card that moves the player backwards and the move makes the player pass start in the counterclockwise direction the game crashes.

4.5. Results

All automatic tests passed and the manuel playthroughs revealed the bugs described in the known bugs chapter. The program is still far from thoroughly tested and more bugs are certain to appear when/if more tests are created.

4.6. Import to eclipse

To Import:

1. Extract the “14_final” folder to a location of preference
2. Open eclipse
3. Goto File > Import > General and choose Projects from Folder or archive
4. Click next
5. Press the Directory button and choose the “14_final” folder you extracted

4.7. Run the game

1. Doubleclick the “start game.bat” file in the root of the extracted archive.

5. Conclusion

The assignment was completed to the required specifications as shown by the results and requirements fulfillment chapters. The tests run on and in the program were successful and showed little to no errors while it was under development. However the tests are severely lacking and doesn't validate all our requirements especially in the GameBoard class. This was caused by a lack of time since we have already started on it early in development but since we've made changes we hadn't had time to update it. Other than this we lack a few "chance cards" from the original game.

Nevertheless we mostly overcame any major hurdles with proper file management, enabling us to effectively backtrack to earlier builds that didn't include whichever error was displayed. We also utilized branch management following a specific model we agreed upon early in the planning phase, but we did have some problems with updating the files to the correct versionnumber just like our last project. Other than this the coding went swift and the integration between the code pieces had little major problems and only required a bit of time and discussions within the group.

We had planned to implement the core features of the game as the first iteration and implement extra features as later iterations. Most of the iterations were made however we could've stopped and turned in the core features which is why we decided to part the features into iterations after their importance for the game. We only missed 4 features from our list which were all in the lowest priorities.

The program runs smoothly and the report features all the models we've had to use to show the process and sufficiently explain the program, so overall we are really content about our finished product and report.

6. Appendix

6.1. Appendix 1 Theoretical probability calculation

6.1.1. Theoretical probability calculation for a die

The frequency of a certain value being rolled was calculated by taking the number of possible combinations that could result in that value being rolled divided by the total number of combinations multiplied by 100%.

$$\frac{Tally}{observations} * 100\%$$

Example

The total number of observations (total tally) was 6

So the Frequency of rolling any value were: $\frac{1}{6}$*100%=16,667%

The mean value was calculated by the formula: $\overline{x} = \sum_{i=1}^{k} f_i * x_i$

$$\overline{x} = 0,1667 * 1 + 0,1667 * 2 + 0,1667 * 3 + 0,1667 * 4 + 0,1667 * 5 + 0,1667 * 5 = 3,5$$

The variance was calculated by the formula: $var(x) = \sum_{i=1}^{k} f_i(x_i - \overline{x})^2$

$$var(x) = 0,167 * (1-3,5)^2 + 0,167 * (2-3,5)^2 + 0,167 * (3-3,5)^2 + 0,167 * (4-3,5)^2 +$$

$$0,167 * (5-3,5)^2 + 0,167 * (6-3,5)^2$$=2,922

To get the spread over 60000 rolls the variance was multiplied by 60000.

$$v = 2,922 * 60000 = 175320$$

Then the spread was calculated by the formula: $\sigma(x) = \sqrt{v}$

$$\sigma(x) = \sqrt{175320} = 418,71 \approx 419$$

Our calculations was done using statistics. The decision to test over 60000 rolls is due to the nature of statistic as the spread becomes relatively smaller as the number of rollers increase. This in turn increases the precision of the test. The probability of rolling each value was calculated to 16,67% with a spread of 419 over 60000 rolls.

6.1.2. Theoretical probability calculation for 2 dice

	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

The frequency of a certain value being rolled was calculated by the formula: $\frac{Tally}{observations} * 100\%$

Example

The total number of observations (total tally) was 6*6= 36

Combinations that gave the value 4 (tally for the value 4) was 3

So the Frequency of rolling the value of 4 was: $\frac{3}{36}$*100%=8,334%

The mean value was calculated by the formula: $\overline{x} = \sum_{i=1}^{k} f_i * x_i$

$$\overline{x} = 0,028 * 2 + 0,056 * 3 + 0,083 * 4 + 0,111 * 5 + 0,139 * 6 + 0,167 * 7 + 0,139 * 8 + 0,111 * 9 +$$

$$0,083 + 0,056 + 0,028 * 12 = 6,91 \approx 7$$

The variance was calculated by the formula: $var(x) = \sum_{i=1}^{k} f_i(x_i - \overline{x})^2$

$$var(x) = 0,028 * (2-7)^2 + 0,056 * (3-7)^2 + 0,083 * (4-7)^2 + 0,111 * (5-7)^2 + 0,139 * (6-7)^2 +$$

$$0,167 * (7-7)^2 + 0,139 * (8-7)^2 + 0,111 * (9-7)^2 + 0,083 * (10-7)^2 + 0,056 * (11-7)^2 + 0,028 * (12-7)^2$$

$=$5,852

Then the spread was calculated by the formula: $\sigma(x) = \sqrt{v}$

Over 60000 rolls the spread was.

$$\sigma(x)\sqrt{5,852 * 60000} = 592,55 \approx 593$$

Value	Tally	Frequency	$\bar{x}$	var(x)	$\sigma(x)$ over 60000 rolls	Expected observations
2	1	2,778%				1667
3	2	5,556%				3334
4	3	8,333%				5000
5	4	11,110%				6666
6	5	13,889%				8333
7	6	16,667%				10000
8	5	13,889%				8333
9	4	11,110%				6666
10	3	8,333%				5000
11	2	5,556%				3334
12	1	2,778%				1667
Total	36	100%	$6,91 \approx 7$	$5,852 \approx 6$	593	60000

Due to the nature of statistics, it was chosen that the test of the probability of rolling a certain value with two dice would be done over 60000 rolls to achieve a precise result. The probability of rolling each value calculated and presented in the table above in the frequency cullom. From those results it was possible to calculate how many observations was expected, the results were presented in the observations cullom in the table above. Finally the spread was calculated to 593 over 60000 rolls.

6.2. Appendix 2: Fields

```
Start("Start", 1, Color.red);
Territory("Rødovervej", 2, Color.blue, 1200, new int[]{50, 250, 750, 2250, 4000, 600}, 1000);
Chance("Prøv lykken", 3, Color.black);
Territory("Hvidovervej", 2, Color.blue, 1200, new int[]{50, 250, 750, 2250, 4000, 600}, 1000);
Tax("Indkomstskat", 4, Color.lightGray, 4000, 0.1f);
Fleet("Scandlines", 5, Color.gray, 4000, new int[]{500, 1000, 2000, 400});
Territory("Roskildevej", 6, Color.orange, 2000, new int[]{100, 600, 1800, 5400, 8000, 1100}, 1000);
Chance("Prøv lykken", 3, Color.black);
Territory("Valby Langgade", 6, Color.orange, 2000, new int[]{100, 600, 1800, 5400, 8000, 1100}, 1000);
Territory("Allégade", 6, Color.orange, 2400, new int[]{150, 800, 2000, 6000, 9000, 1200}, 1000);
Jail("Fængsel", 7, Color.black);
Territory("Frederiksberg Allé", 8, Color.cyan, 2800, new int[]{200, 1000, 3000, 9000, 12500, 1500}, 2000);
Brewery("Tuborg", 9, Color.green, 3000, 100);
Territory("Büowsvej", 8, Color.cyan, 2800, new int[]{200, 1000, 3000, 9000, 12500, 1500}, 2000);
Territory("Gl. Kongevej", 8, Color.cyan, 3200, new int[]{250, 1250, 3750, 10000, 14000, 1800}, 2000);
Fleet("Mols-Linien", 5, Color.gray, 4000, new int[]{500, 1000, 2000, 400});
Territory("Bernstorffsvej", 10, new Color(190, 123, 252), 3600, new int[]{300, 1400, 4000, 11000, 15000, 1900}, 20
Chance("Prøv lykken", 3, Color.black);
Territory("Hellerupvej", 10, new Color(190, 123, 252), 3600, new int[]{300, 1400, 4000, 11000, 15000, 1900}, 2000)
Territory("Strandvejen", 10, new Color(190, 123, 252), 4000, new int[]{350, 1600, 4400, 12000, 16000, 2000}, 2000)
FreeParking("Parkering", 11, Color.pink);
Territory("Trianglen", 12, Color.red, 4400, new int[]{350, 1899, 5000, 14000, 17500, 2100}, 3000);
Chance("Prøv lykken", 3, Color.black);
Territory("Østerbrogade", 12, Color.red, 4400, new int[]{350, 1800, 5000, 14000, 17500, 2100}, 3000);
Territory("Grønningen", 12, Color.red, 4800, new int[]{400, 2000, 6000, 15000, 18500, 2200}, 3000);
Fleet("Scandlines", 5, Color.gray, 4000, new int[]{500, 1000, 2000, 400});
Territory("Bredgade", 13, Color.white, 5200, new int[]{450, 2200, 6600, 16000, 19500, 2300}, 3000);
Territory("Kgs. Nytorv", 13, Color.white, 5200, new int[]{450, 2200, 6600, 16000, 19500, 2300}, 3000);
Brewery("CocaCola", 9, Color.green, 3000, 100);
Territory("Østergade", 13, Color.white, 5600, new int[]{500, 2400, 7200, 17000, 20500, 2400}, 3000);
GoToJail("De fængsles", 10, Color.black);
Territory("Amagertorv", 14, Color.yellow, 6000, new int[]{550, 2600, 7800, 18000, 22000, 2500}, 4000);
Territory("Vimmelskaftet", 14, Color.yellow, 6000, new int[]{550, 2600, 7800, 18000, 22000, 2500}, 4000);
Chance("Prøv lykken", 3, Color.black);
Territory("Nygade", 14, Color.yellow, 6400, new int[]{600, 3000, 9000, 2000, 24000, 2800}, 4000);
Fleet("Scandlines", 5, Color.green, 4000, new int[]{500, 1000, 2000, 400});
Chance("Prøv lykken", 3, Color.black);
Territory("Frederiksberg gade", 15, Color.magenta, 7000, new int[]{700, 3500, 1000, 22000, 26000, 3000}, 4000);
Tax("Indkomstskat", 4, Color.lightGray, 2000, 1.0f);
Territory("Rådhuspladsen", 15, Color.magenta, 8000, new int[]{1000, 4000, 12000, 28000, 34000, 4000}, 4000);
```

6.3. Appendix 3: Chance cards

```
MoneyCard("pay3000car1", -3000, 0, 0));
MoneyCard("pay3000car2", -3000, 0, 0));
MoneyCard("payperhousehotel1", 0, 500, 2000));
MoneyCard("pay200parking", -200, 0, 0));
MoneyCard("pay1000carinsurance", -1000, 0, 0));
MoneyCard("pay1000redlight", -1000, 0, 0));
MoneyCard("pay2000dentist", -2000, 0, 0));
MoneyCard("pay200cigarette", -200, 0, 0));
MoneyCard("payperhouhotel2", 0, 800, 2300));
MoneyCard("get1000stock1", 1000, 0, 0));
MoneyCard("get1000stock2", 1000, 0, 0));
MoneyCard("get1000stock3", 1000, 0, 0));
MoneyCard("get1000raise", 1000, 0, 0));
MoneyCard("get200produce", 200, 0, 0));
MoneyCard("get1000premium1", 1000, 0, 0));
MoneyCard("get1000premium2", 1000, 0, 0));
MoneyCard("get3000quarterlytax", 3000, 0, 0));
MoneyCard("get500lottery", 500, 0, 0));
MoneyCard("get1000tip", 1000, 0, 0));
MoveCard("move3forward", 3, false));
MoveCard("move3backwards", -3, false));
MoveCard("movetostart", 1, true));
MoveCard("movetogroeningen", 25, true));
MoveCard("movetoalle", 12, true));
MoveCard("movetoraadshuspladsen", 40, true));
TotalValueCard("legate"));
JailCard("getoutofjail"));
JailCard("getoutofjail"));
ToJailCard("gotojail"));
ToJailCard("gotojail"));
GoToNearestFleet("nearestfleet"));
GoToNearestFleet("nearestfleet"));
```

6.4. Appendix 4: Timetable

Casper							
Dato	Requirements	Analysis	Design	Implementation	Test	Documentation	Total
02/01/2017	3	3					6
03/01/2017				2	6		8
04/01/2017						0.5	0.5
05/01/2017				4	4		8
06/01/2017				6		1.5	7.5
09/01/2017			8				8
10/01/2017						7	7
11/01/2017			2	4		2	8
12/01/2017						1	1
13/01/2017						9	9
16/01/2017						4	4
total	3	3	10	16	10	25	67

Micheal							
Dato	Requirements	Analysis	Design	Implementation	Test	Documentation	Total
02/01/2017	2	3					5
03/01/2017		8					8
04/01/2017	1	5	2				8
05/01/2017							0
06/01/2017							0
09/01/2017		4				2	6
10/01/2017						8	8
11/01/2017						4	4
12/01/2017				3		5	8
13/01/2017		5				3	8
16/01/2017						4	4
total	3	25	2	3	0	26	59

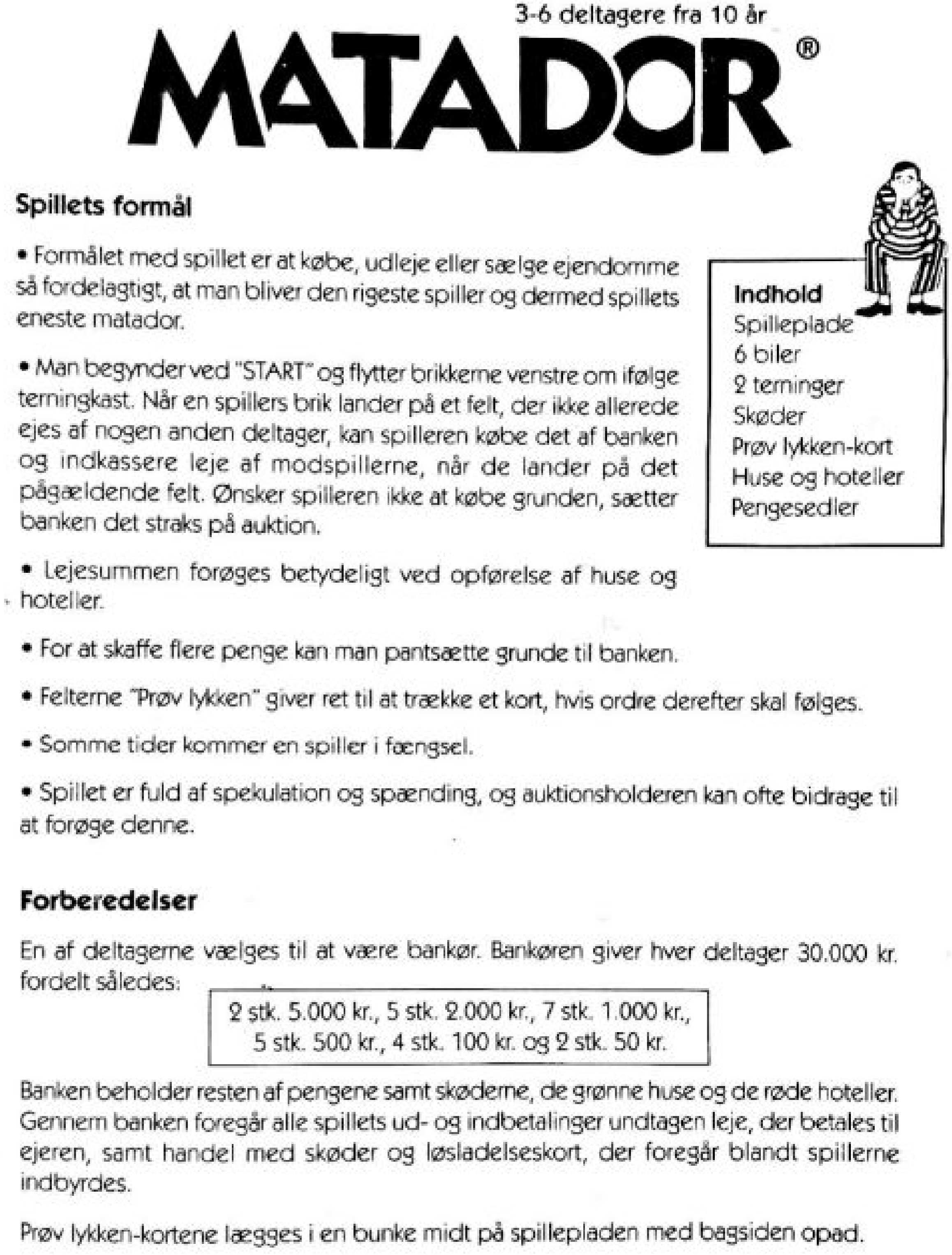
Mathias							
Dato	Requirements	Analysis	Design	Implementation	Test	Documentation	Total
02/01/2017	3	3					6
03/01/2017	1	4	1			2	8
04/01/2017			3		4.5		7.5
05/01/2017				4	2	2	8
06/01/2017				3	4		7
09/01/2017							0
10/01/2017				3	6		9
11/01/2017				8			8
12/01/2017				7			7
13/01/2017						8	8
16/01/2017						4	4
total	4	7	4	25	16.5	16	72.5

Rasmus							
Dato	Requirements	Analysis	Design	Implementation	Test	Documentation	Total
02/01/2017	3	3					6
03/01/2017		4	4				8
04/01/2017		3	5			0.5	8.5
05/01/2017							0
06/01/2017				1			1
09/01/2017				9			9
10/01/2017				8			8
11/01/2017				8			8
12/01/2017				8			8
13/01/2017		1	3	4			8
16/01/2017						4	4
total	3	11	12	38	0	4.5	68.5

Lasse							
Dato	Requirements	Analysis	Design	Implementation	Test	Documentation	Total
02/01/2017	2	3	1				6
03/01/2017		4	4				8
04/01/2017		3	2			2	7
05/01/2017		1	1	3	3		8
06/01/2017			3	4			7
09/01/2017			2	5			7
10/01/2017			1	7			8
11/01/2017			1	6	1		8
12/01/2017				7	1		8
13/01/2017				2	2	4	8
16/01/2017						4	4
total	2	11	15	34	7	10	79

Timothy							
Dato	Requirements	Analysis	Design	Implementation	Test	Documentation	Total
02/01/2017							0
03/01/2017	1	3	2			1.5	7.5
04/01/2017	0.5	3				4	7.5
05/01/2017			1.5	4.5	2		8
06/01/2017							0
09/01/2017				10			10
10/01/2017			4	6			10
11/01/2017				7			7
12/01/2017							0
13/01/2017							0
16/01/2017						4	4
total	1.5	6	7.5	27.5	2	9.5	54

6.5. Appendix 5: Rules



3-6 deltagere fra 10 år

MATADOR®

Spillets formål

- Formålet med spillet er at købe, udleje eller sælge ejendomme så fordelagtigt, at man bliver den rigeste spiller og dermed spillets eneste matador.
- Man begynder ved "START" og flytter brikkerne venstre om ifølge terningkast. Når en spillers brik lander på et felt, der ikke allerede ejes af nogen anden deltager, kan spilleren købe det af banken og indkassere leje af modspillerne, når de lander på det pågældende felt. Ønsker spilleren ikke at købe grunden, sætter banken det straks på auktion.
- Lejesummen forøges betydeligt ved opførelse af huse og hoteller.
- For at skaffe flere penge kan man pantsætte grunde til banken.
- Felterne "Prøv lykken" giver ret til at trække et kort, hvis ordre derefter skal følges.
- Somme tider kommer en spiller i fængsel.
- Spillet er fuld af spekulation og spænding, og auktionsholderen kan ofte bidrage til at forøge denne.

Indhold
Spilleplade
6 biler
2 terninger
Skøder
Prøv lykken-kort
Huse og hoteller
Pengesedler

Forberedelser

En af deltagerne vælges til at være bankør. Bankøren giver hver deltager 30.000 kr. fordelt således:

2 stk. 5.000 kr., 5 stk. 2.000 kr., 7 stk. 1.000 kr.,
5 stk. 500 kr., 4 stk. 100 kr. og 2 stk. 50 kr.

Banken beholder resten af pengene samt skøderne, de grønne huse og de røde hoteller. Gennem banken foregår alle spillets ud- og indbetalinger undtagen leje, der betales til ejeren, samt handel med skøder og løsladelseskort, der foregår blandt spillerne indbyrdes.

Prøv lykken-kortene lægges i en bunke midt på spillepladen med bagsiden opad.

Selve spillet

Deltagerne stiller deres bil på feltet "START" og bliver enige om, hvem der begynder. Spillet fortsætter derefter i urets retning.

Første spiller kaster begge teminger og flytter sin bil så mange felter frem, som øjnene viser. Når spilleren har benyttet retten eller opfyldt pligten, som feltet angiver, går turen videre til næste spiller. Hver gang man passerer "START", modtager man 4.000 kr. fra banken.

Lander man på et felt med "**Prøv lykken**", skal man tage det øverste kort i bunken med Prøv lykken-kort og rette sig efter ordlyden på det. Når et kort er benyttet, lægges det tilbage nederst i bunken.

Lander man efter et temingkast eller ifølge ordren på et af Prøv lykken-kortene på en grund eller virksomhed, der ikke ejes af nogen anden deltager, kan man købe denne af banken for den pris, der står på feltet, og man får så udleveret skødet, der lægges med forsiden opad foran spilleren. Efter de takster der står på skødet, kan man nu opkræve leje af de spillere, der lander på ens grund. Køber man ikke skødet, sætter banken det straks på auktion, og denne har alle lov til at deltage i.

Lander man på feltet "De fængsles", skal man gå direkte i fængsel og modtager ikke de 4.000 kr. for at passere "START". Lander man derimod på feltet "I fængsel", er man blot på besøg og fortsætter næste gang uden straf.

Indkomstskatten har man lov til at betale med 4.000 kr. Men man kan også betale 10% af sine værdier: Kontanter, bygninger og den trykte pris for grunde og virksomheder (også pantsatte). Spilleren skal vælge betalingsmåden, inden han tæller sine værdier sammen.

Man får et **ekstrakast**, hvis man kaster 2 af samme slags (f.eks. 2 femmere), og man skal rette sig både efter forskrifterne for det felt, man lander på efter første kast og efter ekstrakastet. Kaster man 3 gange i træk 2 af samme slags, må man ikke flytte tredje gang, men skal gå direkte i fængsel.

Feltet med **Parkering** er et fristed, indtil man skal kaste igen.

Man kommer ud af fængslet på en af følgende måder:

1) Ved at betale en bøde på 1.000 kr., inden man kaster terningerne.

2) Ved at benytte et af løsladelseskortene fra bunken med Prøv lykken-kort.

3) Ved at kaste 2 af samme slags. Man flytter så straks det antal felter frem, som øjnene viser, og har alligevel ekstrakast.

Man kan ikke blive i fængslet mere end tre omgange. Får man ikke to af samme slags, når man kaster tredje gang, må man betale bøden på 1.000 *kr.* og flytte, som øjnene viser. Er man i fængsel, har man stadig ret til at købe grunde (ved auktion eller handel spillerne imellem), men man kan ikke opkræve leje af de andre spillere.

Huse og hoteller

Ejer man alle grundene i samme farve, får man dobbelt leje af de ubebyggede grunde og har ret til når som helst at bygge huse, der købes hos banken til den pris, der står på skøderne.

Der skal bygges jævnt, dvs. at man kan opføre det første hus på den grund i gruppen, man ønsker, men inden hus nr. 2 opføres på en grund, skal der være bygget ét hus på hver af de andre grunde i gruppen osv.

Inden man opfører et hotel, skal der være fire huse på hver grund i gruppen.

Der må kun bygges ét hotel på hver grund. Når man køber et hotel, afleverer man de fire huse til banken.

Banken skal, når som helst man ønsker det, tage bygningerne tilbage til halv pris. Prisen for et hotel er fem gange prisen for et hus.

Har banken ingen bygninger, når man vil købe, må man vente, til der kommer nogle tilbage. Er der flere, der vil købe, og har banken ikke nok til alle, sætter banken de huse, der er, på auktion.

Indbyrdes handel med ubebyggede grunde etc. er spillerne tilladt til den pris, de kan blive enige om.

NB! Har man bygget, skal man sælge bygningerne tilbage til banken, inden man kan afhænde nogen grund i den pågældende gruppe.

Pantsætning

- Man kan kun pantsætte sine ubebyggede grunde etc. til banken for det beløb, der står trykt på skøderne. Har man bygninger på grundene, skal man først sælge disse til banken. Spilleren beholder skødekortene, men vender bagsiden opad. Renten er 10% (der rundes op til nærmeste 100 kr.), og renten betales sammen med lånet, når pantsætningen hæves.

- Hvis en pantsat ejendom sælges, og køberen ikke straks hæver pantsætningen, skal han alligevel betale 10%, hvis han senere hæver pantsætningen.

- Af pantsat ejendom kan der ikke opkræves leje.

- Banken giver kun lån mod pantsætningssikkerhed.

- Pantsætning af grunde samt handel med bygninger sker kun gennem banken.

- Spillerne må ikke låne indbyrdes.

Glemmer man at opkræve leje af en medspiller, har man tabt sin ret, når spiller nr. 2 efter vedkommende har kastet terningerne.

Fallit. Skylder en spiller mere, end han ejer, skal han overdrage alt til sin kreditor efter at have solgt eventuelle bygninger tilbage til banken, og han må derefter udgå af spillet.

Er det banken, der er kreditor, sælger bankøren straks modtagne grunde på auktion.

Hurtigt spil

Bankøren blander skødekortene og giver hver spiller to skøder, for hvilke han modtager den trykte pris.

Der bestemmes en spilletid, og når tiden er gået, har den spiller vundet, som har størst formue.

God fornøjelse!

BRIO®

Markedsføres af BRIO® A/S
3600 Frederikssund