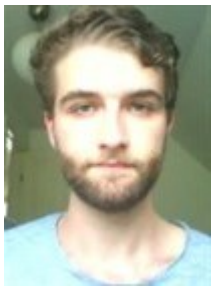
Technical University of Denmark

02312 Indledende Programmering
02313 Udviklingsmetoder til IT-systemer
02315 Versionstyring og testmetoder

CDIO Final

Gruppe 13

Nicolai Kammersgård
s143780



Simon Fritz Hansen
s175191

Mathias M. Thejsen
s175192

Magnus S. Koch
s175189

Christian Stahl Andersen
s143780

Zeth Adrian Benjamin Damore
s155891

January 15, 2018

Time sheet

Person	Date	Hours
Whole group	02/01-2018	5
Whole group	03/01-2018	5
Whole group	04/01-2018	5
Mathias	05/01-2018	5
Christian	05/01-2018	4
Mathias, Simon, Christian and Nicolai	08/01-2018	8
Benjamin and Magnus	08/01-2018	5
Whole group	09/01-2018	7
Whole group	10/01-2018	7
Whole group	11/01-2018	7
Whole group	12/01-2018	7
Whole group	13/01-2018	4
Simon	14/01-2018	4
Whole group	15/01-2018	4

Abstract

The purpose of this project is to construct a full replication of the board game Matador. We have decided to implement all the features possible, which goes as follows:
House building, property management, pawning ability, auction, and trading. To achieve this we tried to work using an agile/iterative method, more specifically the unified process.
We managed to do everything on time, and incorporate all the features we planned. Architecture wise, we planned everything out first as mvc, later to rework it for BCE.

På Dansk
Formålet med dette projekt er at konstruere en komplet replikation af spillet Matador. Vi har valgt at implementere alle egenskaber, hvilket betyder følgende:
At bygge huse, eje grunde, pantsætning, auktionering af grunde og handel med andre spillere. For at opnå dette har vi forsøgt at arbejde med agile/iterative metoder, mere specifikt Unified Process.
Vi har formået at nå alt til tiden og har fået implementeret alle vores planlagte egenskaber. Vores arkitektur var først planlagt efter MVC modellen, men blev senere omlagt til BCE modellen.

Goals

The main goal of this project is for us to rework the old code we have created more specifically re-factor and improve it to make sure we have a consistent high code quality. As implementing Matador as opposed to previous projects is quite complex we have decided to redo almost all our of code, but based of previous projects.

Parallel with our main objective we strive to work in iterations and make use of the GRASP patterns, while utilizing the tools learned in 02312, 13 and 15.

Contents

1 Introduction

In this CDIO we are tasked with creating a Matador board game. We have decided to implement all functions in a regular Matador game.
Working and testing in a top down approach allows us to first implement and test the most crucial parts, and then expand to our "sub" modules, thus we can work in a iterative approach, where we can re evaluate what we create constantly. Additionally we once again work using the UP, where we work in iterations, and utilize use case oriented development.

På Dansk
I denne CDIO har vi fået til opgave at lave et Matador spil. Vi har valgt at implementere alle regler og funktioner, så det fungerer ligesom et Matador spil som spilles på bræt.
Vi arbejder med en top-down metoden, hvilket giver os mulighed for først at implementere og teste de vigtigste dele, og derefter arbejde ned af med vores mere specifikke egenskaber. Dette betyder at vi kan arbejde på en iterativ metode, og der ved konstant genevaulere hvad vi skriver i koden. Vi arbejder derudover efter UP metoden, hvor vi arbejder i iterationer og benytter use case drevet development.

2 Analysis

2.1 Feature List

Feature List table

Priority (1-5)	Risk (1-5)	Feature Name	Feature Description
5	1	Game Movement	Ability for the player to move around the board
4	3	Ability to purchase property	The player must be able to buy and sell property
2	5	Chance cards	Implementation of chance cards
5	2	Property names and descriptions	Having the right property names, description and values.
4	5	Implementation of special fields	Implementing the special fields, such as prison, ferry and parking.
4	4	Ability to build on property	Ability to build houses and hotels on your own property
3	3	Ability to pawn property	The player should be able to pawn his properties, if he is in need of money
1	4	Auction house	Ability to start an auction on a property if the player did not wish to buy it
1	4	Trade between players	Possible for players to trade cards and money between each other

2.2 Requirements

Requirement specification table

Functional Requirements	**Non Functional Requirements**
REQ1: The game must support 2-6 players	NREQ1: The software must be written in Java
REQ2: The Player must be able to land on all fields	NREQ2: Use the provided GUI
REQ3: Some fields must either add or take money from the Player's balance	NREQ3: Reuse as much code as possible from last cdio
REQ4: The Balance may not be negative	NREQ4: Utilize GRASP patterns
REQ5: The Players must start with 30000 kroner	NREQ5: Use git for source/version control and maven for dependency management
REQ6: The Player should move along the fields on the board	NREQ6: The system must be easily translatable/configurable
REQ7: Some fields must cause the player to skip a turn	NREQ7: The system must function on the windows machines at the DTU datalabs.
REQ8: Some fields must function as a jail	
REQ9: Some fields must function as a chance	
REQ10: One field must function as a parking field, a safe field	
REQ11: The fields which are not prison, start, parking or chance must be own-able	
REQ11: Landing on a another player field should affect both players balance in positive/negative ways	
REQ12: Players must have the ability to pawn their owned properties	
REQ13: Properties must be able to go on auction	
REQ14: Players must be able to trade properties between each other	
REQ15: Players must be able to build on some properties	
REQ16: Chance fields must draw a chance card for the Player	

2.3 Stakeholders

The following table shows the stakeholders of the program.
We have two different stakeholders: The User and the Company.

Stakeholders

Name / Role	Requirements	Type
User	To play the game without any issues	Primary
Company	To have a functioning game	Secondary

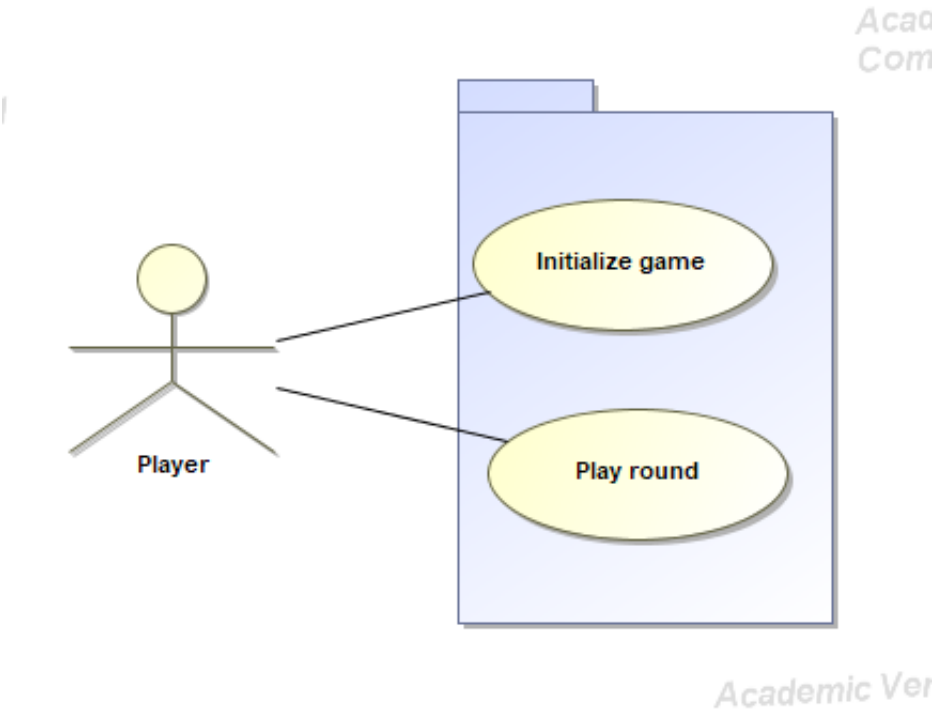
2.4 Actors

This table shows the actors in the Use Cases. Only the players are actors in this game

Actors

Name / Role	Use cases	Use Case ID
Players	To be able to play the game	1

2.5 Use Case Diagram



Use Case Diagram

Whole group

2.6 Use Case

Brief: Play Round

Use Case ID: 1

The player starts the game and throws dice until he/she wins or loses

Brief: Initialize Game

Use Case ID: 2

Gameboard is created, Players and Accounts are initialized, and players are placed on the board.

Use Case ID 1

Use Case: Play Round

Mainflow	1. Player A throws Dice 2. The Player lands on the field equal to the sum of the dice values 3. The Player's Balance may be changed 4. The next Player's turn starts 5. Start from 1, until it is player A's turn again 6. Go to 2 a. If player A passes start, then their balance is updated 7. Repeat until all but one player has gone bankrupt
Alternative Flows	2a: Player lands on a property that is not owned (a) The player buys the field (A) The players balance is deducted the field value (B) The next player starts their turn (C) Start from 1 (b) The player does not buy the field (A) The property is set on auction (B) The next player starts their turn (C) Start from 1 2b: The Player lands on a field that is owned by another Player (a) The players balance is deducted the field value (b) The field owners balance is updated (c) The next Player starts their turn (d) Start from 1 2c: The player lands on a prison field (a) The Player is visiting the prison (b) The next Player starts their turn (c) Start from 1

Alternative Flows Continued

2d The player lands on the ”Go To Prison” field

(a) The Player must go to prison

(b) The Player can escape prison by:

(i) Paying 1000 before they throw the dice

(ii) Throwing a pair

(iii) Using a ”Get Out of Jail Free” chance card

(iv) Spending 3 turns in prison, after which they have to pay the fine of 1000

(c) The next Player starts their turn

(d) Start from 1

2e The Player lands on a chance field

(a) The Player draws a chance card

(b) The Player follows the directions of the card

(c) The next Player starts their turn

(d) Start from 1

Fully Dressed Use Case ID 1

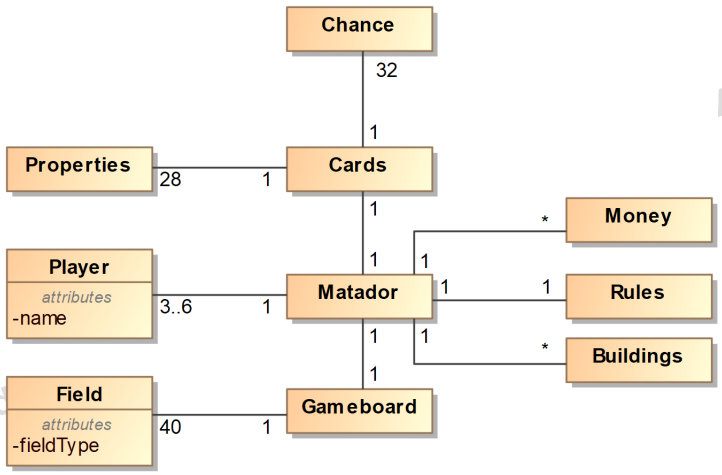
Fully Dressed Use Case ID 1

Referring to UC	1
Scope	Our game
Primary Actors	Player
Secondary Actors	
Preconditions	The game is started, both Players start with 30000 points
Postconditions	The Game has been won by one Player
Fully Dressed	(1) The Player throws the Dice (I) Player lands on a property field (A) The player buys the field (i) The players balance is deducted the field value (ii) The next player starts their turn (iii) Start from 1 (iv) Postcondition: The Player now owns the field (B) Another Player owns the Field (i) The current Player pays X amount to the other Player (ii) The current Player's Balance is changed (iii) The other Player's Balance is changed (iv) The next Player's turn begins (v) Start from 1 (C) The Player already owns the field (i) The next Player starts their turn (ii) Start from 1 (D) The Player chooses not to buy the field (i) The property goes on auction (ii) The next Player starts their turn (iii) Start from 1

Fully Dressed Continued	(II) The Player lands on a chance field (A) The Player draws a random chance card (i) The Player moves to X field (a) The field is not owned - Go to (I) (b) The field is owned by another player - Go to (I).B (c) The Player is sent to prison (ii) The Player has to pay X amount (iii) The Player receives X amount (III) The Player lands on a prison field (A) The Player is sent to prison (i) The Player pays 1000 (a) The Player gets out of jail (ii) The Player utilizes a ”Get Out of Jail Free” chance card (a) The Player gets out of jail (iii) The Player throws a pair (a) The Player gets out of jail (iv) If 3 turns have passed, the Player has to pay 1000 and can then move out of prison (IV) The Player lands on the parking field (A) Nothing happens (V) The Player passes the start field (A) The Player is given 4000 (B) The Player’s balance is updated
Frequency	Always, unless a bug is occurring

2.7 Domain Model

This domain model shows how the different classes work together.
It also shows which main attributes each class have.



Domain Model

2.8 Risk

This part of the document have been designed to capture risks in our project. It captures the risks starting from the one with highest priority, meaning the one which can have the highest negative impact. The risks are organized in a table along with their description, impact, magnitude, indicator and mitigation/contingency strategy.

Risk Table

Magnitude: Risk are ranked from 1 - 10 where 1 is the lowest risk and 10 is the highest risk. This ranking is based upon the criticality of the risk and the probability of it occurring.
Impact: Ranked from 1 - 10, with 1 being Negligible and 10 being Critical

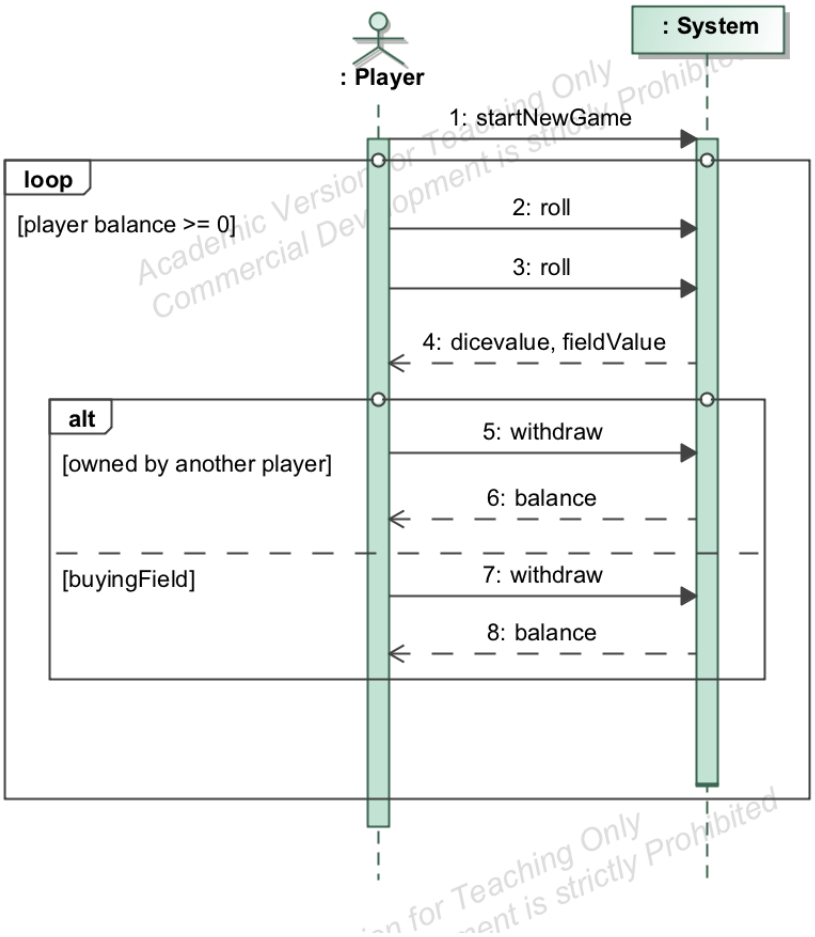
Risks
Simon

Risk	Magnitude	Impact
Issues with implementation of old code	5	2
Sickness	2	1

As we are reusing the dice from last project, we know that our dice works accordingly to probability theory and therefore it is not a relevant risk to list anymore.

2.9 System sequence diagram

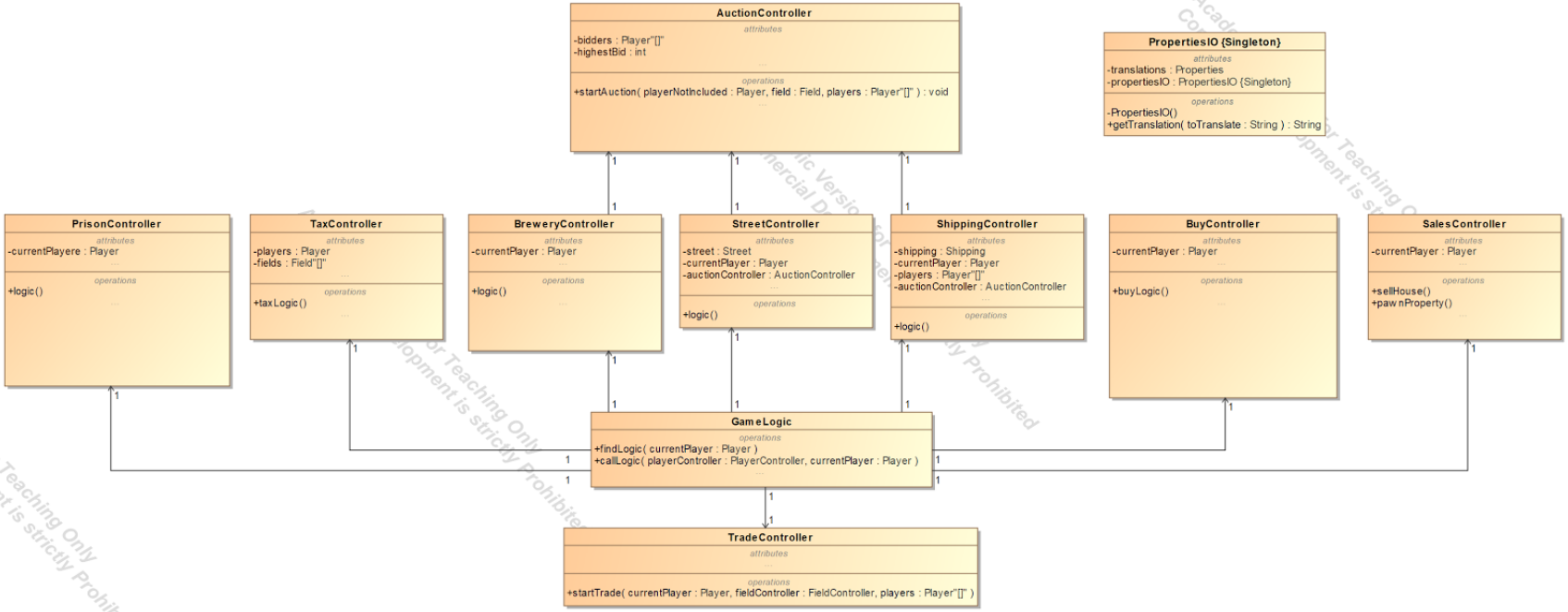
The following diagram shows the events for the specific use case. In this project there is only one use case therefore we only have one system sequence diagram.



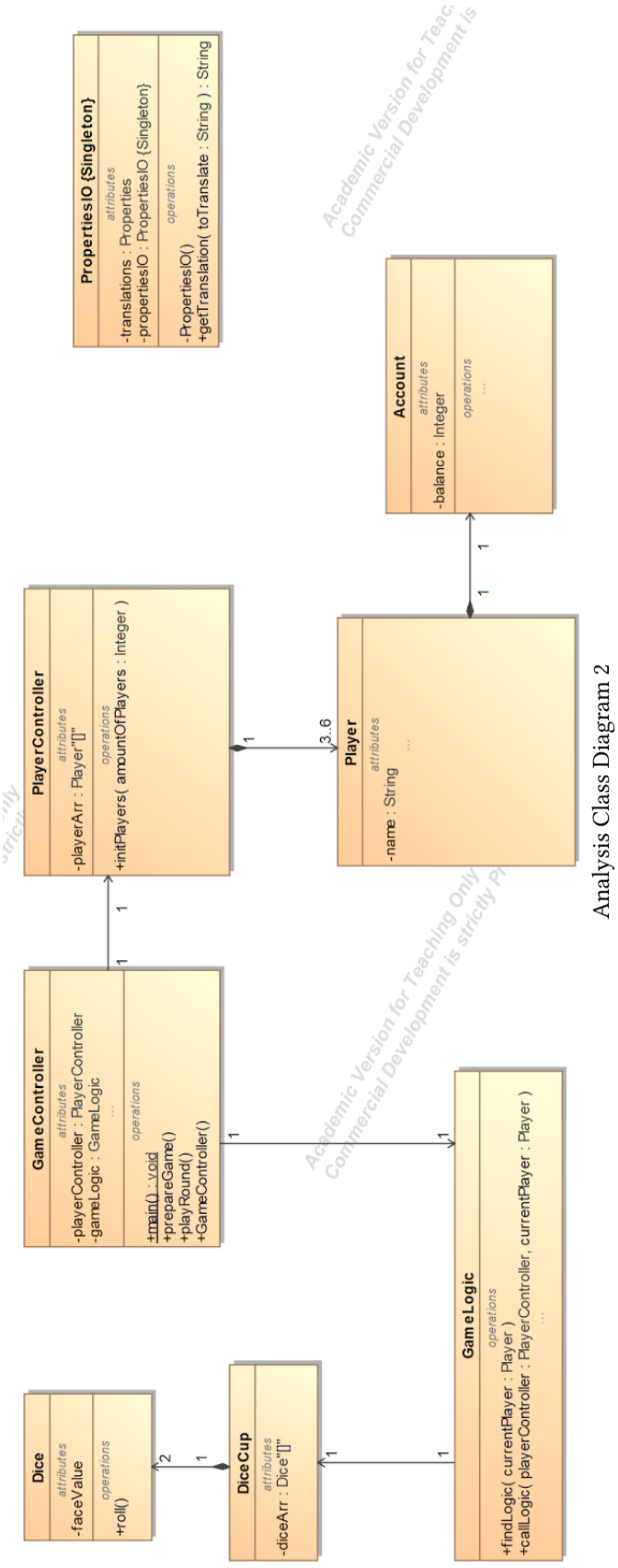
System Sequence Diagram

2.10 Analysis class diagram

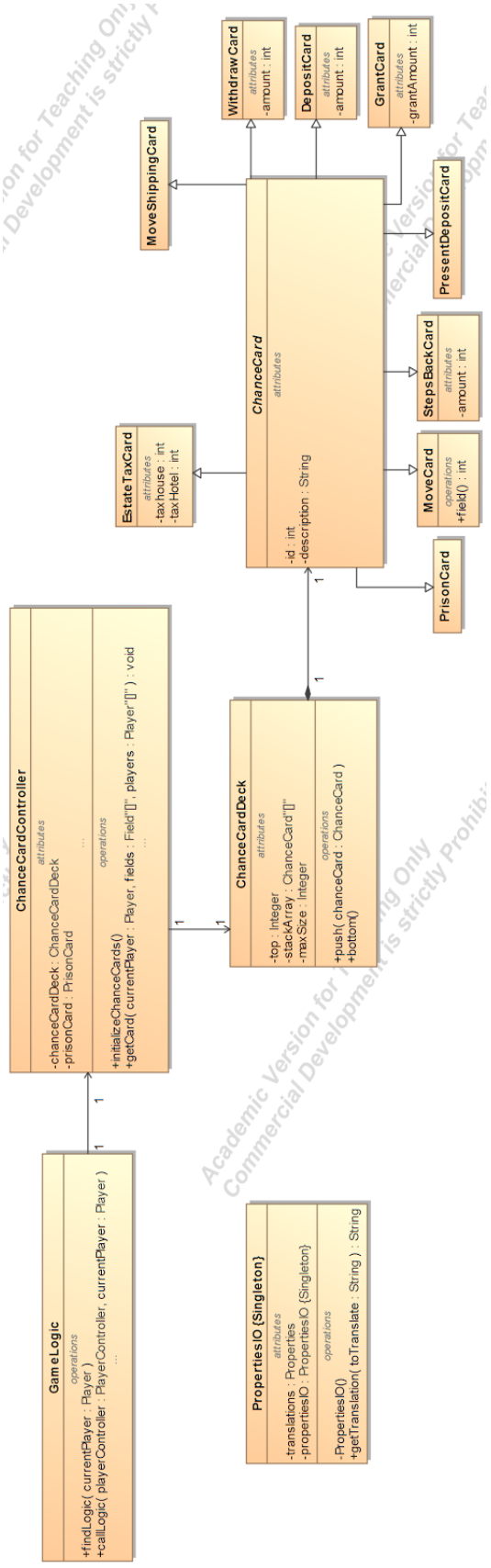
The diagram below shows the major classes with the necessary methods and states for the program.



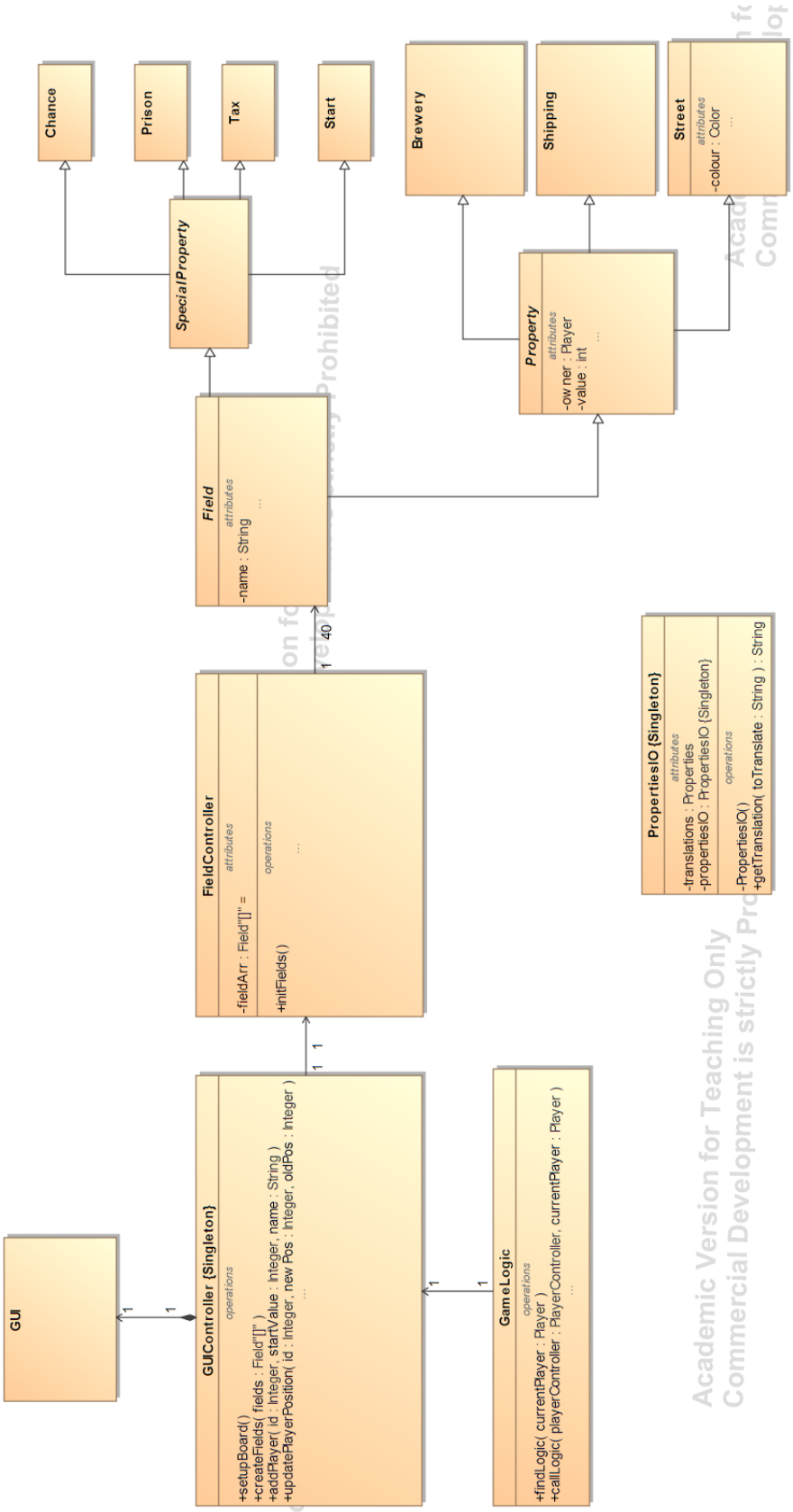
Analysis Class Diagram 1



Analysis Class Diagram 2



Analysis Class Diagram 3



Analysis Class Diagram 4

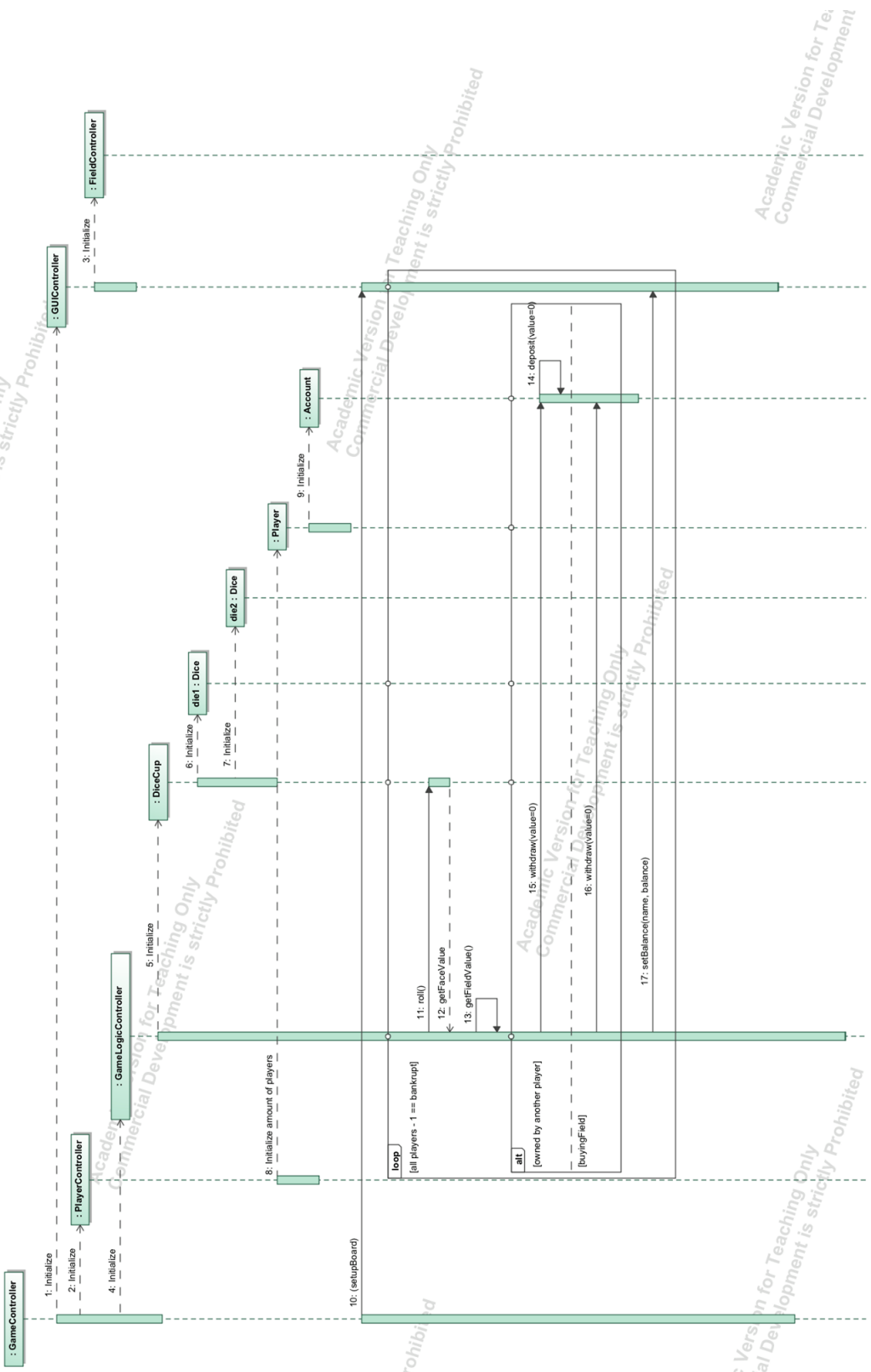
2.11 Used Gameboard and rules

To gain a more cohesive experience we decided to utilize the Matador version from 1936 by BRIO Scanditoy. In the attachments/bilag section a scan of the gameboard, rules, and cards can be found, which is the version we have utilized and implemented.

3 Design

3.1 Design Sequence Diagram

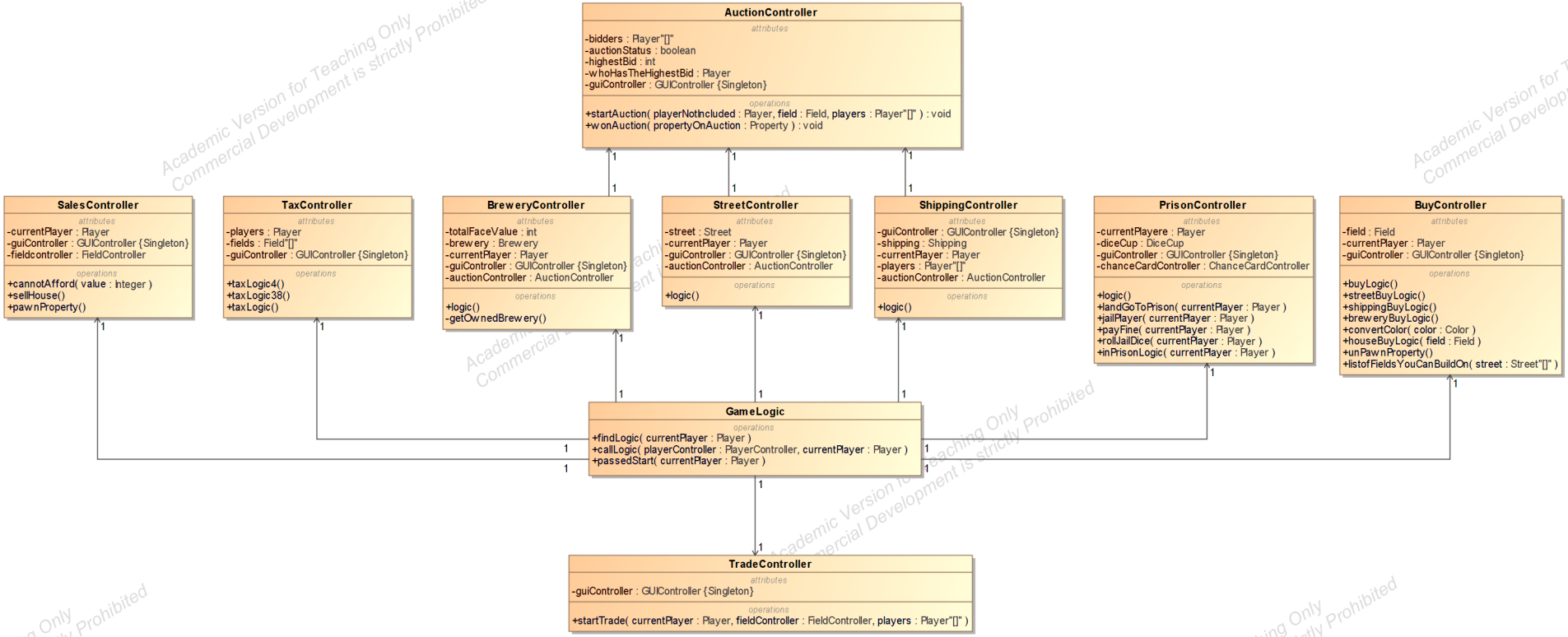
The diagram illustrated below shows the processes operating with each other and how they operate between the classes and objects.



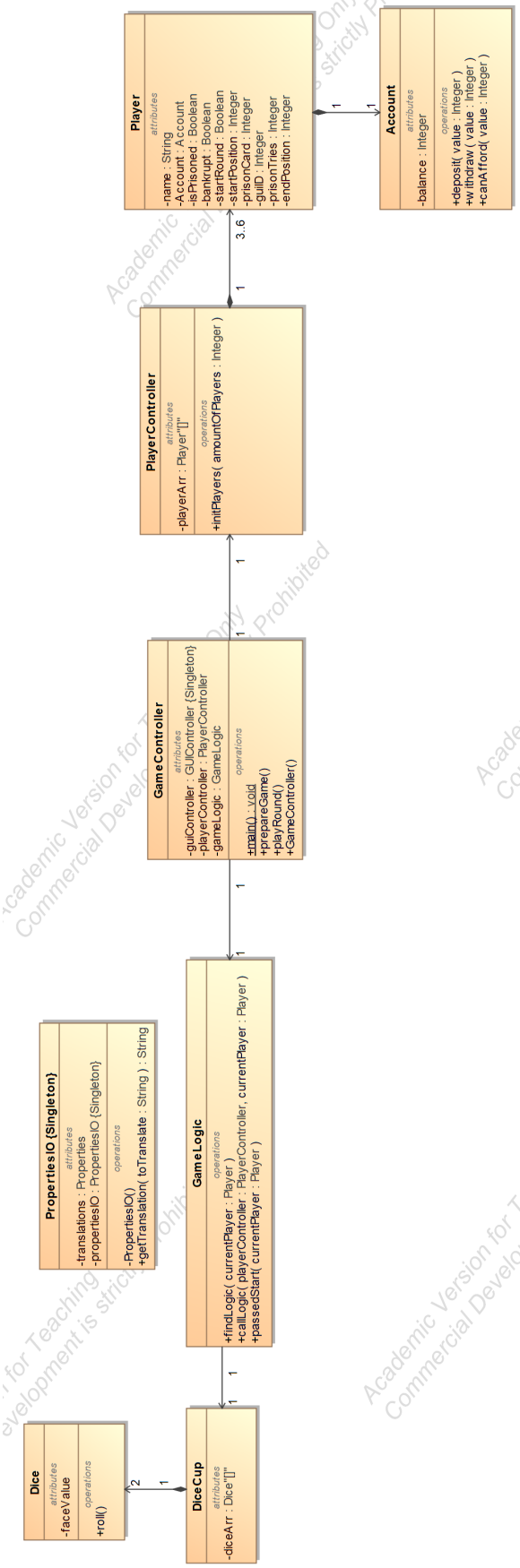
Design Sequence Diagram

3.2 Design Class Diagram

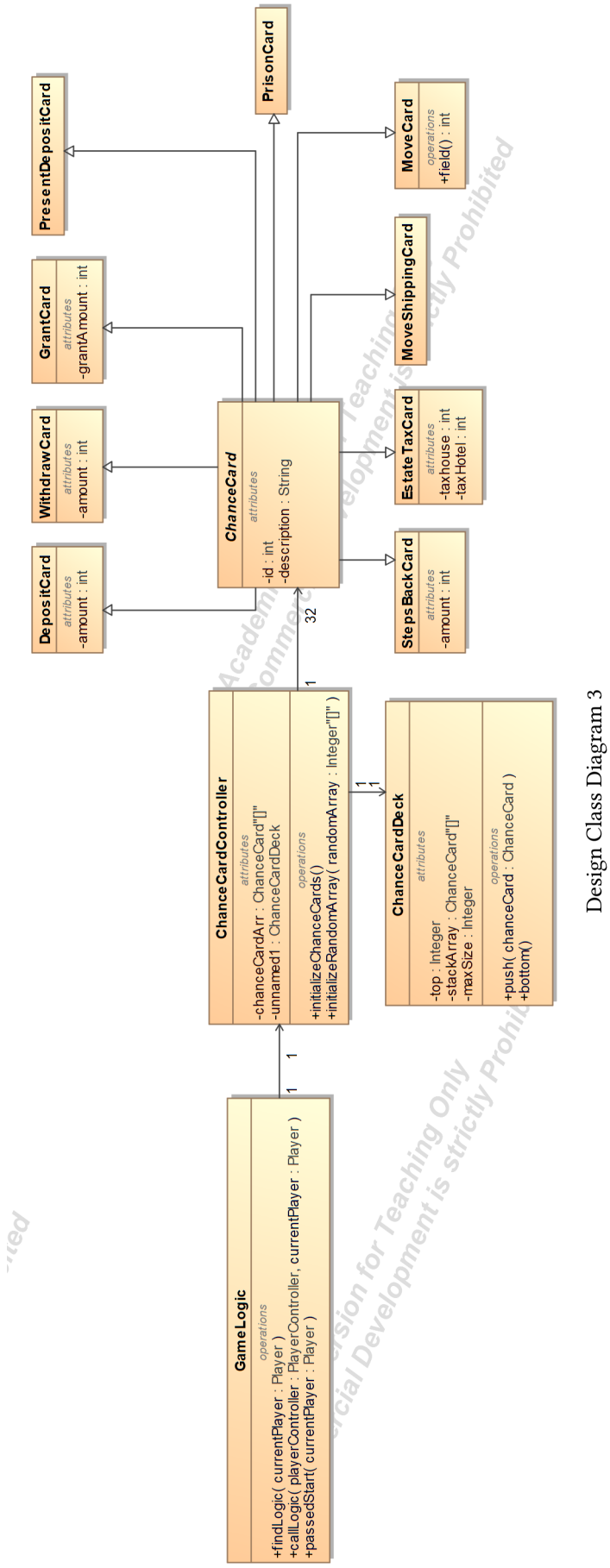
This diagram shows the static structure of the system which include classes, attributes, methods and the relations between the objects.



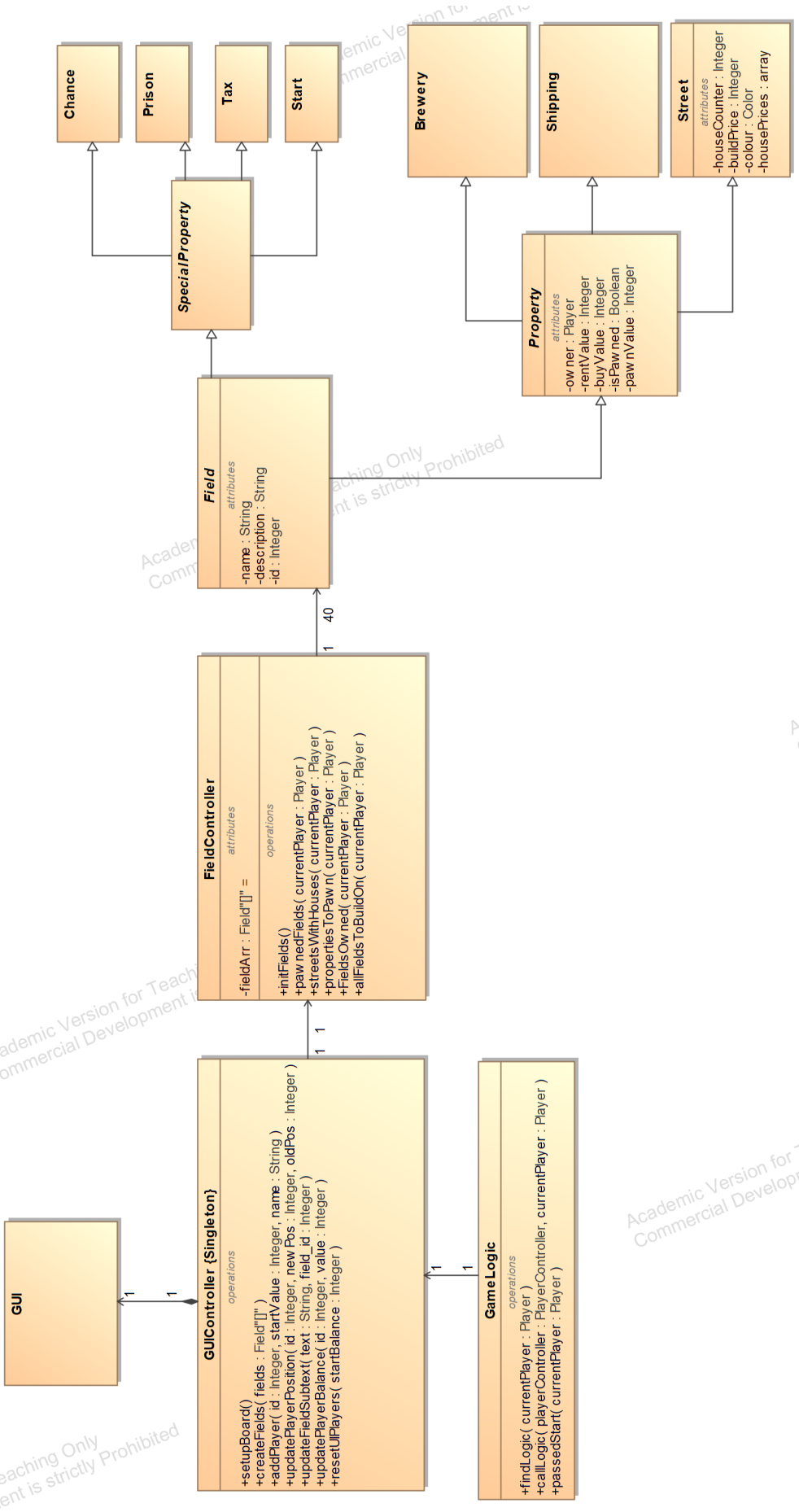
Design Class Diagram 1



Design Class Diagram 2



Design Class Diagram 3



Design Class Diagram 4

3.3 Description of used GRASP patterns

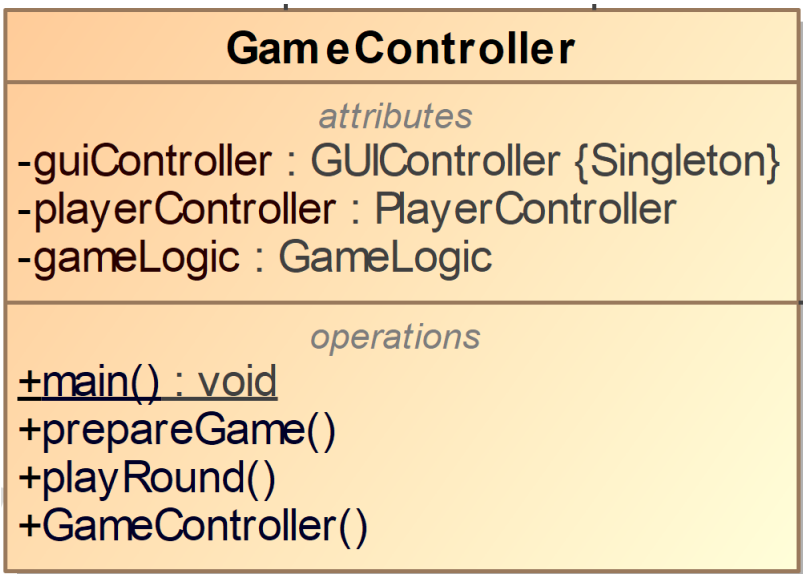
In this project, we have decided to completely restructure our code and way of thinking compared to the previous cdio projects. The only code we have reused from the previous CDIO projects is the player, account, and dice class. As a result our utilised grasp patterns have changed.
We now have a single usecase controller which is our GameController, which implements and recognises our usecases by initialising the game and keeping track of whose turn it is, alternating the turns. The GameController functions as our information expert and creator, as it keeps track of who has the references to each other and it initialises the game, more specifically the main components.
Additionally unlike previous projects, we now have multiple controllers. We now have a main logic controller called GameLogic, which decides the outcome of the player, and can call additional subcontrollers, such as BreweryController, BuyController or PrisonController. This also means that the GameLogic functions as a creator by initialising the subcontrollers.
We also have a FieldController, which acts as a creator aswell, the responsibility of fieldcontroller is to initialise the fields and control stuff that has todo with the field itself, but not the logic of the specific field.
We utilise polymorphism in our Field class, as we have our superclass Field, which passes down some base values to it's childs Property and SpecialProperty, which also both pass down methods, and values to the subfields to its own child classes. Both field, property, and SpecialProperty are abstract classes and thus never instantiated. As a result form utilizing polymorphism. We are able to make more redundant code, that is easier to maintain and expand. The guiController is initialised/kept as a singleton, to make sure we never get more than a single instance of it. To keep up high cohesion and low coupling we have tried to make sure each class has a well defined responsibility.

4 Implementation

In this chapter the most important and complex classes are discussed and their purpose/functionality is explained.

4.1 GameController

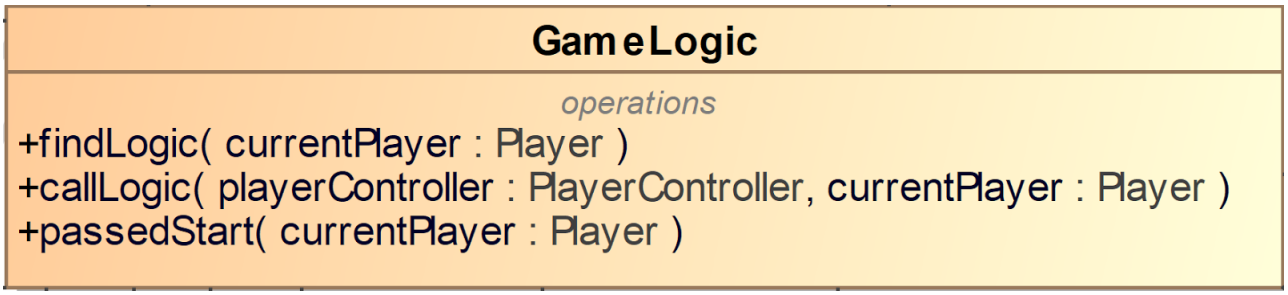
The main class where the game is started is the GameController. The GameController has the responsibility to initialize the gui, prepare the game, and keep track of whose turn it is aswell as wherever the game is over or not.



- GameController()
  - The constructor initializes the GUIController, the PlayerController, and the gamelogic.
- prepareGame()
  - Prepares the game by calling a method in the GUIController that adds the players to the board, and then loops over the received players, filling out the playerArray.
- playRound()
  - playRound has an outer while loop that runs as long as the game is live. A for loop is then initialized that runs through the player array switching player, while checking if the player is bankrupt, if the player is, then the player is skipped automatically. There is then a secondary loop that finds the winner, and then displays the winner message.

4.2 GameLogic

Each round a method called showOptions which generates a menu for the player, depending on what the player should be able todo, it will generate options such as “Roll dice “, “Buy house/hotels”, “Pawn property”, and “Trade”. So if the player does not own any properties “Buy house/hotel” or “Pawn property” will not appear, checks are made when generating the menu, to make sure that the player fulfills certain conditions. Depending on which option is chosen a respective controller is called. If the player should choose to roll dice, the resolveField is called, which re-



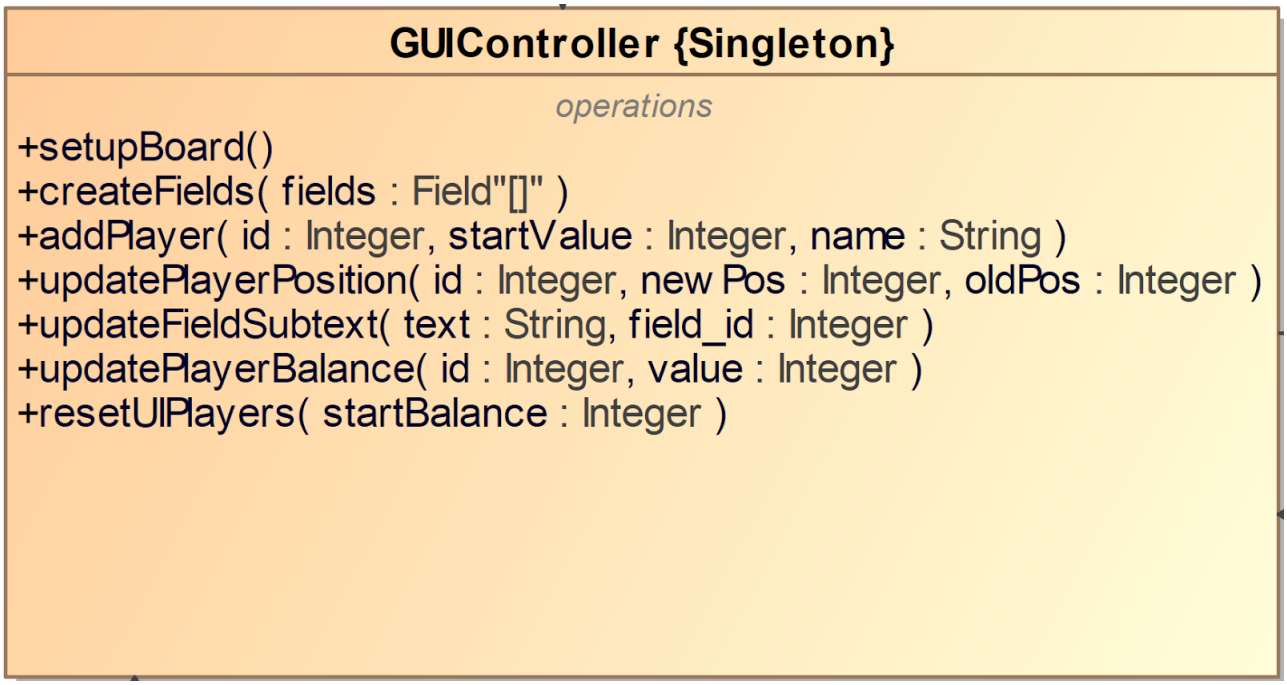
solves what kind of field we landed on and thereby call it’s respective field controller which then takes care of what should happen, depending if the player can afford landing there, if he wishes to buy the property, if it’s a chance card field or not. Additionally the roll diceF option takes care of checking if the player should have passed start, and updates their player position.

- GameLogic()
  - The constructor initializes a chancecardcontroller, a dicecup with two dice and a tradecontroller
- ShowOptions(PlayerController playerController, Player currentPlayer)
  - This method has the responsibility to figure out what options the player should have. It does that by checking various things such as if the player is jailed, if the player already has rolled,if the player owns any properties. Depending on the outcome of the above a list is generated for the player that will display the player options.
  - It then contains a switch statement that then depending on what button the player presses does various things, if the player chooses to roll dice, then his position is updated, and logic is then called.
  - If the player decides to buy houses/hotels, a list is generated of the properties he can build on, and displayed.
  - If the player chooses to pawn or unpawn, buy or sales controller is called.
  - If the player chooses to trade, the tradecontroller is started

- resolveField(Player currentPlayer, DiceCup diceCup)
  - This method is called in the showoptions method if the player chooses to roll, it then take the players new position and figures out which kind of field the player has landed on and then call it's respective controller.
- updatePos(Player currentPlayer)
  - The purpose of this method is to update the players position, it first figures out wherever the player has passed start, and then updates the visual/gui position.
- passedStart(Player currentPlayer)
  - The purpose of this method is to check wherever the player has passed start in order to give the player his passed start money.

4.3 GUIController

The GUIController has the responsibility to setup the board, by interacting with the FieldController which contains a method to "populate" the board and our fieldarray. The GUIController is a singleton, and is interacted with in all controllers, whenever something should happen on the board such as updating the balance, position or displaying a message.

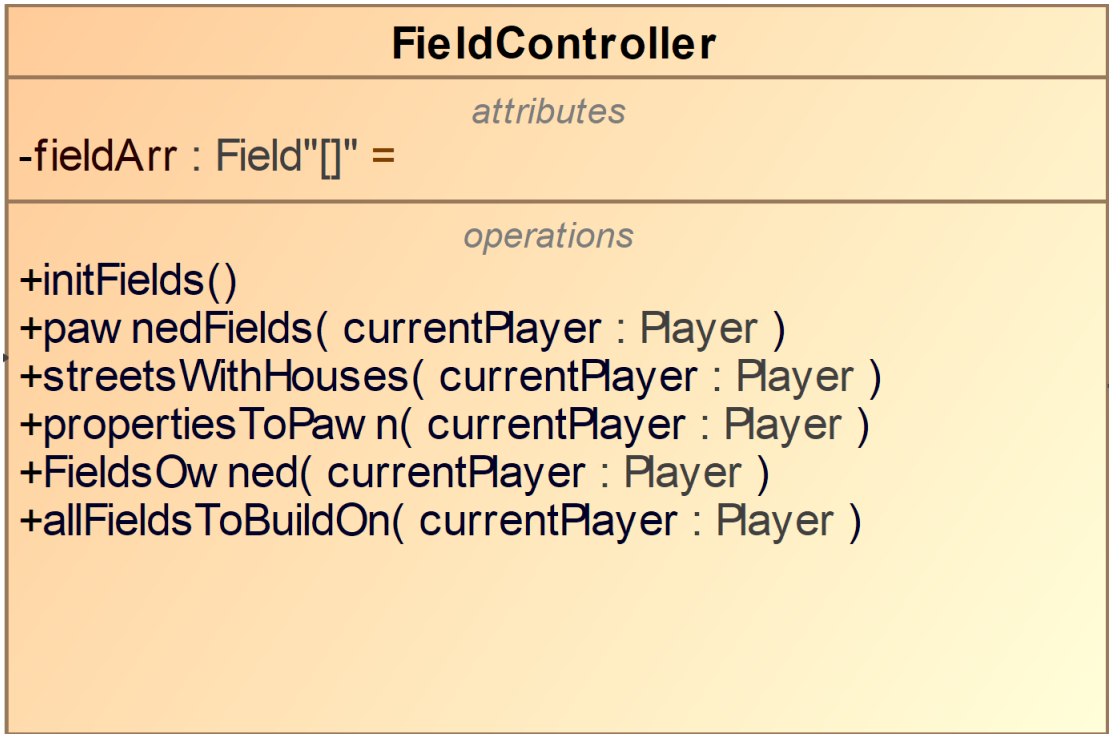


- setupBoard()
  - Prepares the board by adding players to the board

- createFields(Field[] fields)
  - Populates the board by generating the right fields based on the field array, filling out the board with the right colour, name and value
- updatePlayerPosition(id int, newPos int, oldPos int)
  - Animates the player as the player moves

4.4 FieldController

As mentioned above the main purpose of the FieldController is to initialize the board/fields, but it also contains some other interesting methods. It contains methods used to find out who owns what properties, which is used when building houses, and also methods to generate a list of pawnable properties.



- initFields()
  - The purpose of this method is to fill out the field array, by first initializing an empty array of the size 40 of the type Field, corresponding to the board. It then proceeds to initialize the corresponding subfields such as shipping or street, and then fills out the array with the right references.
- allFieldsToBuildOn(Player currentPlayer)

  - The purpose of this method is to generate an array of the type street which contains references to all the streets that houses can be built on. We loop over all fields, and filter out all the fields which are not the type street. We then check if the current street we are checking has the same owner as our currentPlayer and make sure that the housecounter is less than 5, as you can't have more than 5 houses. We then grab the colour of that field, and make sure that the other streets with the same colour is owned by the same owner.
  - When that is done we then create a new array, as the previous one had a default size of 40, the new array will now be the actual size of how many fields the player can build on, and references are copied over.

- FieldsOwned(Player currentPlayer)
  - The purpose of this method, is to generate an array that contains all the field titles, that the player owns. It does this quite simply by looping over all the fields, making sure they are ownable fields and that they are owned by the player. It then fills out an array containing all the field titles/names

- propertiesToPawn(Player currentPlayer)
  - The purpose of this method is to generate a string array which contains all the properties, that the player can pawn. It does this easily by looping over the field array; checking if they are streets, if the owner is the currentPlayer, if the house counter is 0, and if the street is already not pawned. As done in the two previous methods we make a temp array first, with a default size, and then later create an actual array with the right size.

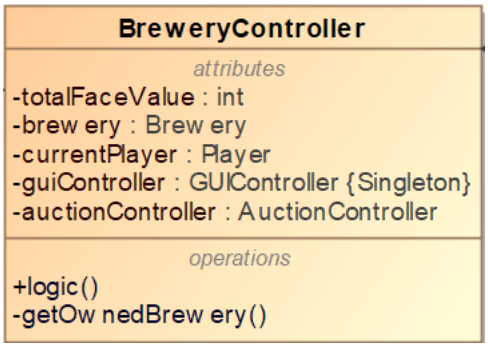
- streetsWithHouses(Player currentPlayer)
  - The purpose of this method is to generate a list of streets that contains houses, so the player has the ability to sell the houses. It does this by looping over the fieldarray, making sure they are streets, checking if currentPlayer is the street owner, and if the housecounter is less than 5 but greater than 0. We then fill out the temporary array. Afterwards the real array is filled out and made the right size.

- pawnedFields(Player currentPlayer)
  - This method is used when the player wishes to unpawn a field. Once again we loop over all the fields, making sure that the currentPlayer is the owner, that the field is pawned and that the player can afford to unpawn his property.

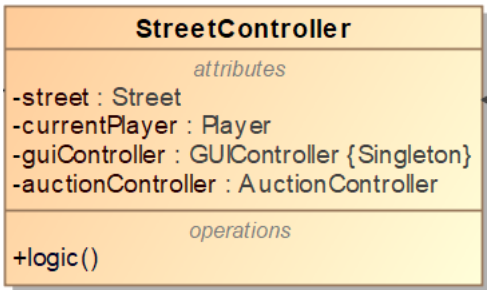
4.5 Shipping/Brewery/StreetController

The shipping, brewery, and street controller all serve the purpose of handling the logic when landing on the respective fields, they check if the fields are owned, if not then the player may be asked to purchase the property. However the actual purchase is handled in the BuyController. The

Shipping.logic() method checks who owns the fields. If the field is not owned by anyone, it gives the option to buy the field. If the player cannot afford the field, AuctionController.startAuction() is called, making all other players, who can afford it, able to bid on the field.
If the player already owns the field nothing happens, and the Player's turn continues as normal.
If the field is already owned by another player, the method check if the player can afford the rent. If they are unable, SalesController.cannotAfford() is called. This method is described further down.



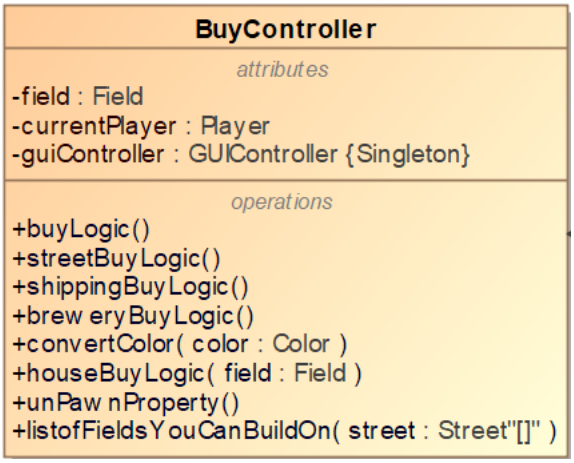
The Brewery.logic() method checks who owns the fields. If the field is not owned by anyone, it gives the option to buy the field. If the player cannot afford the field, AuctionController.startAuction() is called, making all other players, who can afford it, able to bid on the field.
If the player already owns the field nothing happens, and the Player's turn continues as normal.
If the field is already owned by another player, the method checks if the player can afford the rent. If they are unable, SalesController.cannotAfford() is called. This method is described further down.
The Street.logic() method checks who owns the fields. If the field is not owned by anyone, it gives the option to buy the field. If the player cannot afford the field, AuctionController.startAuction() is called, making all other players, who can afford it, able to bid on the field.



If the player already owns the field nothing happens, and the Player's turn continues as normal. If the field is already owned by another player, the method checks if the player can afford the rent. If they are unable, SalesController.cannotAfford() is called. This method is described further down.

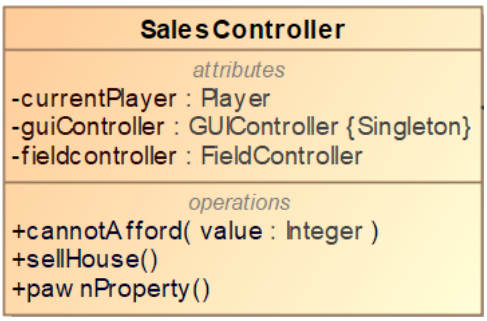
4.6 Buy/Sales Controllers

The BuyController and SalesController has the responsibility of handling purchase and selling of properties. The buycontroller is utilized when a player wishes to buy a property and the sales is used when a player needs to sell/pawn their property. buyLogic()/streetBuyLogic()/shippingBuyLogic()/brewer



- When a player wishes to buy a street buyLogic is called, buyLogic figures out what kind of field we are dealing with, then calls either street, shipping or brewery buyLogic.
- streetBuyLogic() sets the owner of the field, and updates the player balance
- shippingBuyLogic() as the shipping landing price is dependant on how many a players own, it first figures out how many a player owns by looping through all the fields.
- breweryBuyLogic() as the brewery landing price is dependant on how many a player owns, it also figures out how many a player owns

- houseBuyLogic(Field field) Is called when a player wishes to purchase a house on his field, it figures out the build price, and then adds a houes to the field.
- unPawnProperty() is called when a player is able to unpawn one of their pawned fields. It calls the FieldController.pawnedFields() which returns a String array of all pawned fields owned by the Player. It only returns fields that the player can afford to unpawn. The player then chooses which field to unpawn and the players balance is updated. An update to GUIController is also send.
- listOfFieldsYouCanBuildOn(Street[] street)
  - This method is used when a player wishes to build on a property, it makes sure that building is done evenly by looping over all our fields, then grabbing the colour of a field, and the current amount of houses.
  - It then compares it to the other fields with the same colour, and makes sure that the other fields don't have less houses than the current one, if it does it is not added to the list.



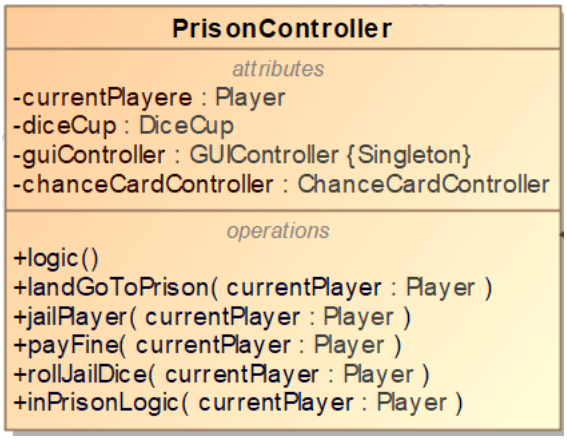
- cannotAfford(int value)
  - The main method called when a player is unable to pay the rent or tax, and takes an integer value equal to the rent/tax needed to be payed. The method keeps looping until the player's balance is above the value which is send when cannotAfford is called. Two other methods are called within cannotAfford, sellHouses, and pawnProperty. These two method returns true or false depending on whether the player is able to pawn or sell anything.
    If the player balance is not above the value after selling all houses and pawning all fields, the player is declared bankrupt and the loop exits.
- sellHouses()

  - Checks whether or not the player has any fields with houses built on them, using the FieldController.streetWithHouses() method. This method returns a String array of all streets with houses owned by the player. The player can then choose to sell houses from any of the streets in the array. The method handles everything on the bank side and sends an update to the GUIController.

- pawnProperty()

  - Checks whether or not the player has any properties to pawn. It uses the FieldController.propertiesToPawn() method to return a String array of all the properties the player own and are able to pawn. The player can the choose to pawn any of the streets in the array. The methods handles everything on the bank side and sends an update to the GUIController.

4.7 Prison Controller



The PrisonController is used when a player lands on the prison field, and handles all the prison logic, keeps track of how long the player has been in jail, and where ever the player has a get out of prison card.

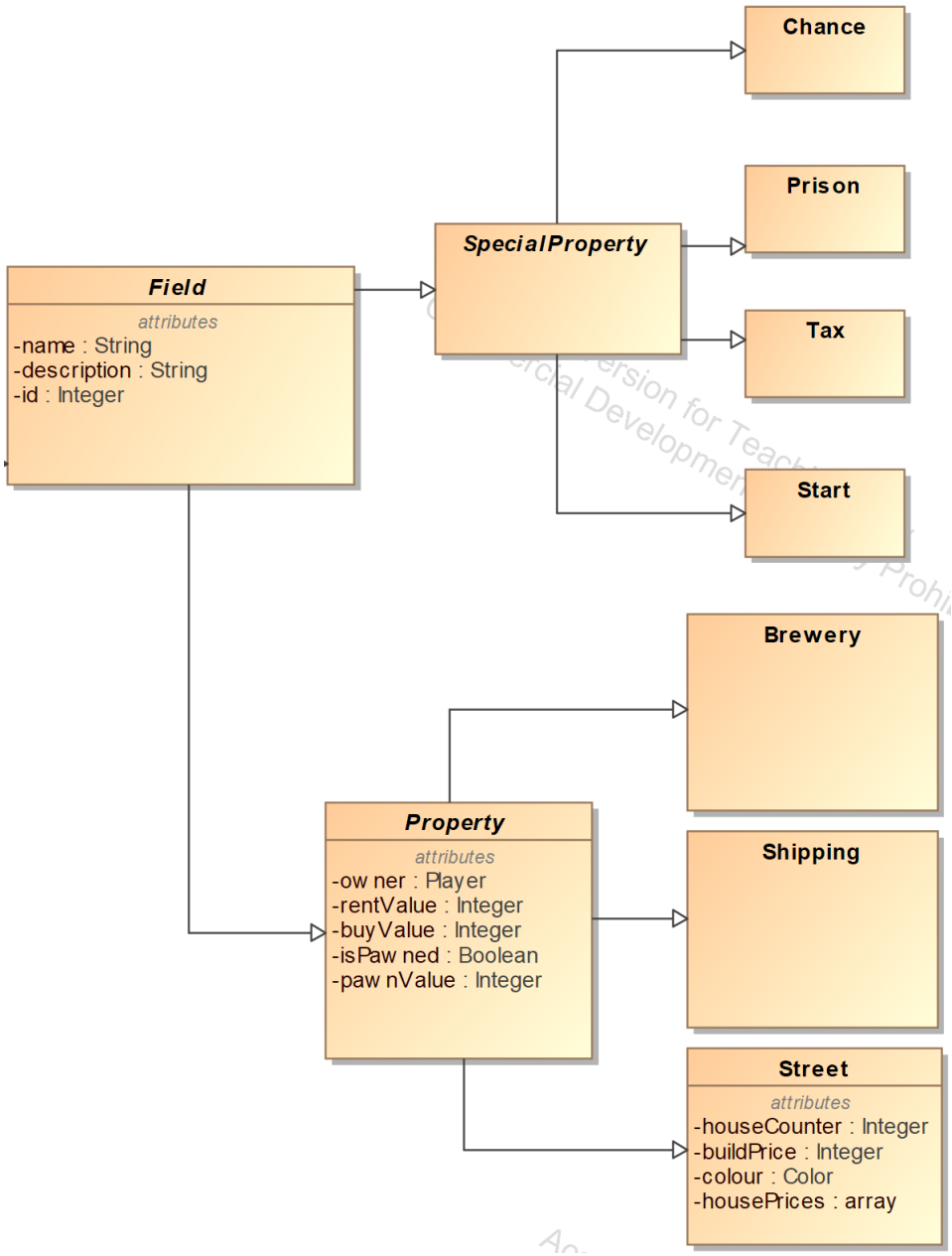
4.8 Explanation of Field and the Subclasses

Field functions as our mother class by having a name, id, and description. This is passed down to two subclasses Property and SpecialProperty

- Property

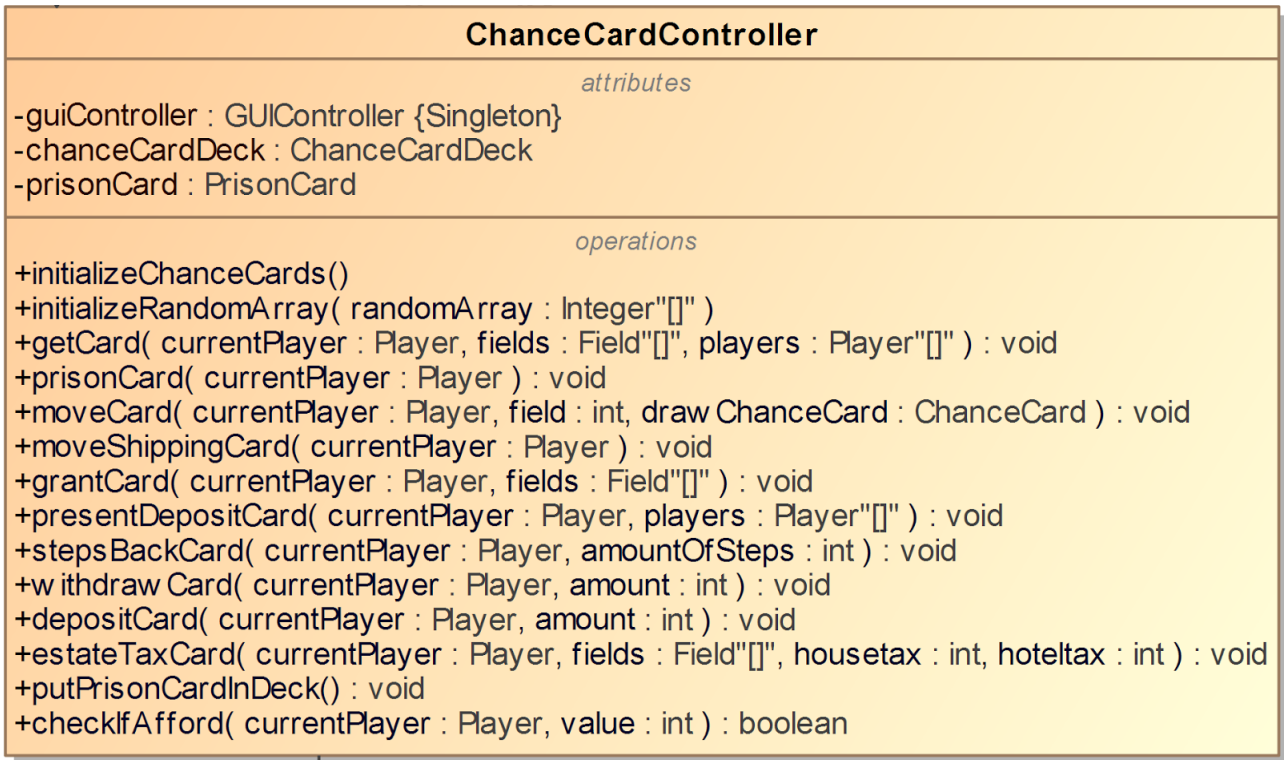
  - Property has 3 subclasses. Shipping, brewery, and street. Property passes down values such as owner, buy, rent, and pawn value.

- SpecialProperty



– SpecialProperty does not pass down any values but still has child classes, tax, start, parking, and prison. The reason all these are not a subclass of property is because they are non ownable fields.

4.9 ChanceCardController



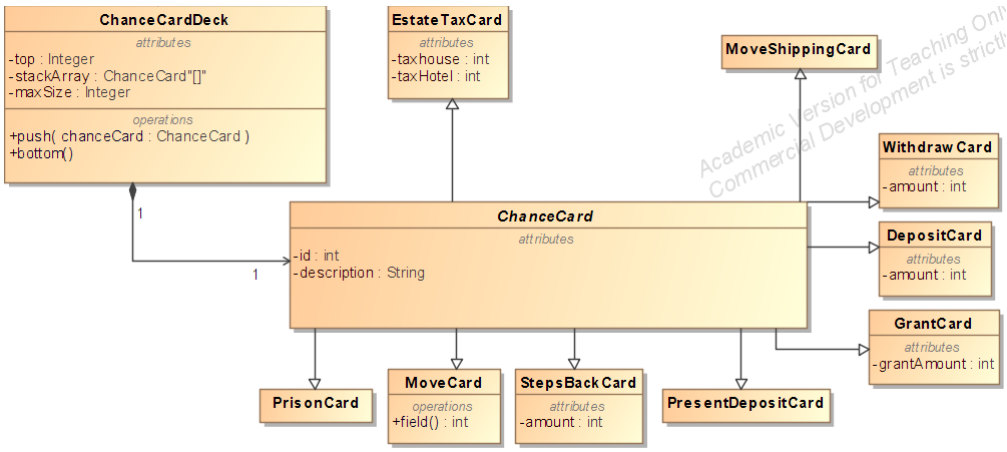
The chance card controller has the responsible for all the chance cards and handles all the logic. The logic from each chance card are divided into individual functionality and method.

- getCard() this is the main function in the controller and its also called when someone lands on a chance-card field. When the function is called it will check which chance card are the next to be picked. The picked card will call the function which will perform the logic of the card.
- prisonCard() is utilised when the prison card is drawn from the deck. The function adds a prison card to a player, which then later gets used to get out of prison.
- moveCard() is utilised when the move card is drawn from the deck. The function moves the player to a specific position depending on which card is picked.
- moveShippingCard() is utilised when the move shipping card is drawn from the deck. The function moves the player to the closest shipping property, depending on which chancefield is drawn.
- grantCard() is utilised when the grant card is picked from the deck. The function checks if the player's net worth is lower than 15.000. If it is lower than the player gets a grant on 40.000.

- presentDepositCard() is utilised when the present deposit card is picked from the deck. The function then takes 200 money from each player, other then the current player and gives it to the current player.
- stepsBackCard() is utilised when the steps back card is drawn from the deck. The function then moves the player an amount of steps back depending on which card is picked.
- withdrawCard() is utilised when the withdraw card is drawn from the deck. The function then gives the current player an amount of money depending on which card is picked.
- depositCard() is utilised when the deposit is drawn from the deck. The function hereby takes an amount of money from the current player depending on which card is picked.
- estateTaxCard() is when the estateTaxCard is picked from the deck. The function checks how many houses and hotels the player owns on their properties. Hereafter takes an amount of money depending taxes.
- putPrisonCardInDeck() this function is called when the prison-card is used and therefore should be shuffled back in the deck.
- checkIfAfford() this function checks if player has the money to afford something. It uses the sales controller for this purpose.

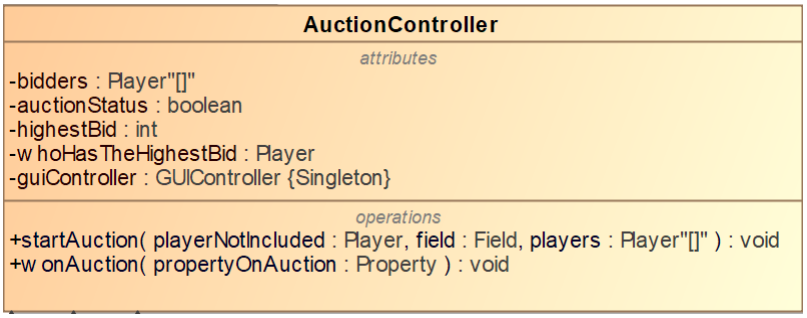
4.9.1 Explanation of ChanceCardDeck and the subclasses of ChanceCard

ChanceCardDeck is the upper class in the diagram, which contains all the different chance cards. Instead of having 32 different chance cards, we decided to divide it in categories and thereby split it up in objects. Therefore the ChanceCard class is an abstract class which is inheriting to the subclasses.



4.10 AuctionController

The auction controller is used when a player chooses not to buy a property the player lands on and thereby starts an auction for the specific property, so the additional players get a chance to buy it. When the auction is started all players can bid on the property and hereby will the highest bidder wins.



- startAuction() is the function which all the property controller classes (BreweryController, ShippingController, StreetController) calls when the player choose not to the buy property. The function creates an array which include all bidders, then starts a loop where all bidders gets a chance to bid on the property.
- wonAuction() is called by the startAuction() function when the bet loop is over and has the responsible for giving the property to the winner of the auction.

4.11 TradeController

The trade controllers purpose is to give the player the option to trade with each other. At the start of the players turn the player gets the option to trade with other with his money.

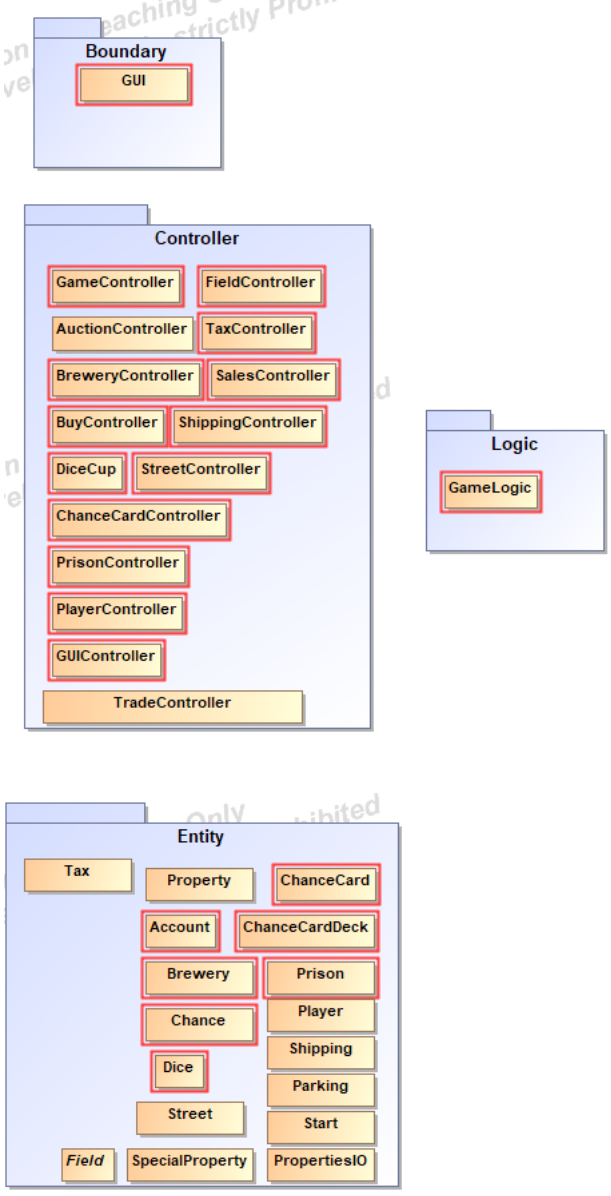
- startTrade() is the function that is called when a player wants to trade. The function then shows all players who has an property to offer. Hereby the player gets the opportunity to choose which property he wishes and hereafter can offer some money for the property.

4.12 PropertiesIO

The PropertiesIO class is the class which contains access to the properties in the config file. The config file contains all translations and variables which can be changed in the game.

- getTranslation() this function is called every time access to the config file is needed. It has the parameter name which gives the parameter for giving access to a specific property in the config file.

Package diagram/Project structure



5 Test

5.1 Test Cases

We have compiled a few test cases for our program. These test are simple and documented so that others will have an easy time redoing the tests and getting the same results. These are the tests:

- Check if the correct throw of dice is assigned to the correct player
- Do you get to buy a field if the field is ownable
- Do you get skipped two turns if you land in prison?
- Can you get a chance card
- Do you get prompted to pay if you land on another players property
- Do you get 4000 when you pass start

See attached pdf for test results and instructions

5.2 Junit Test

Using Junit, we perform unit tests. These are all located in the package called "test", and the test results can be seen in the junit folder which contains .html files of the results.

testFieldController
As mentioned previously the fieldcontroller initializes the field array and also has a few methods that generates lists or checks things within the field array. 3 test have been made that first checks if the fieldarray is generated properly, secondly the two methods AllFieldsToBuildOn and FieldsOwned are tested.
In the test of AllFieldsToBuildOn a player is set to own all streets and then we check if the array generated by the method corresponds to our initialized fieldarray.
In the test of FieldsOwned, we set the player to own the first 10 properties, and then check the length of the list generated by the method.

testAccount
The main purpose of these test, is to test the deposit/withdraw functionality. We test the deposit method by initializing a player, depositing money, and then see if the balance is now equal to the sum of the start value and the deposited value. We test the withdraw in the same way, seeing if the player balance is equal to the withdrawn value subtracted from the start balance.

We then test wherever the balance can ever be negative, by trying to withdraw and deposit a negative value.

testDice
Just as in the last CDIO, we test if our dice works accordingly to probability theory. We perform 60000 rolls, and see if the frequency of the face values is equal to 1/6.

testPlayer
In the test of the player class, we first test the constructor, as the Player class should initialize an account object attached to the player object. Additionally we test the the functionality of the getters and setters for account and player name.
This time we also tests the canAfford method, by checking if a player can afford 32000 after the player has been initialized which it should not be able to as the player starts with 30000.

testStreetController
These tests simulate the logic of the streetcontroller buying and selling a property, and checking if you can buy a property if you dont have enough money.

testPrisonController
These tests check the methods used in the various parts of the PrisonController, to make sure the flow of the methods, which is testable, works correctly.

testSalesController
These tests check if the constructor is made correctly. Additionally we also test the functionallity of the cannot afford function.

5.3 Appraisal of the system in terms of quality

The system is built based on the BCE model and grasp patterns. As a result we have ended out with a result that is very modular and flexible. This means that the logic and controllers that define the boundary, rules and gameplay of Matador is completely disconnected from the core functionality of the game and thus the logic can be replaced with new a logic and the game should still work without any issues. This is achieved by trying to maintain high cohesion and low coupling, making sure that our code is of as high standard as possible.

Not only is the game modular in terms of switching over to a different game, but it is also quite easy to just make a few modifications. Firstly, all translations and configs are stored in a properties file, so changing the start amount of money, the landing prices or the amount of fields is quite

easy. Secondarily, if a developer wished to introduce new kinds of fields, it could be done without any issues, by simply adding a new field class and controller, and adding it into the gamelogic. Our classes have as little as possible dependencies on each other, and as a result introducing a new tax field is a very simple task anyone without previous background in our project should be able to handle.

Although we have been placed various limitations in terms of what can be utilized we still feel as if our end result is of great quality, easy to maintain and easy to re use for further projects.

6 Project Planning

In this project the unified process (UP) is used as development process framework. This means that the project is divided into iterations. In this project, due of the size the development cycle consists of two iterations.

Iteration Table

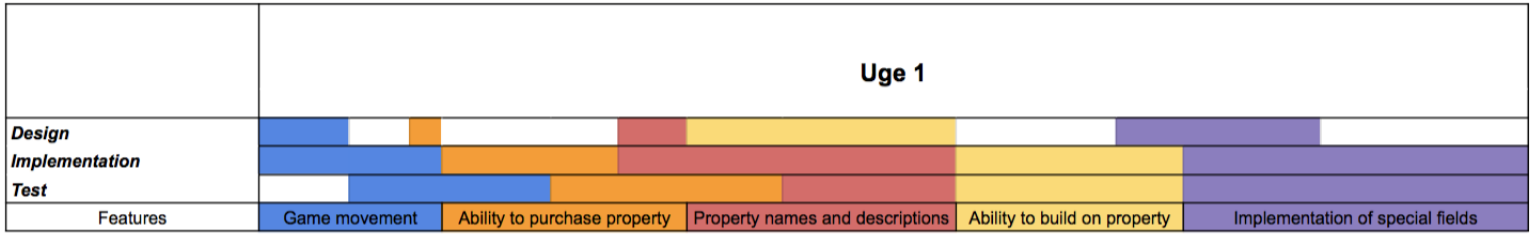
Phases	Iterations	Period (week)
Elaboration	1	1
Construction	1	2
Transition	0	

Each iteration has its own development process, therefore also its own plan. The documentation shows what each iteration inherents, including tasks and process in order which process/task comes first.

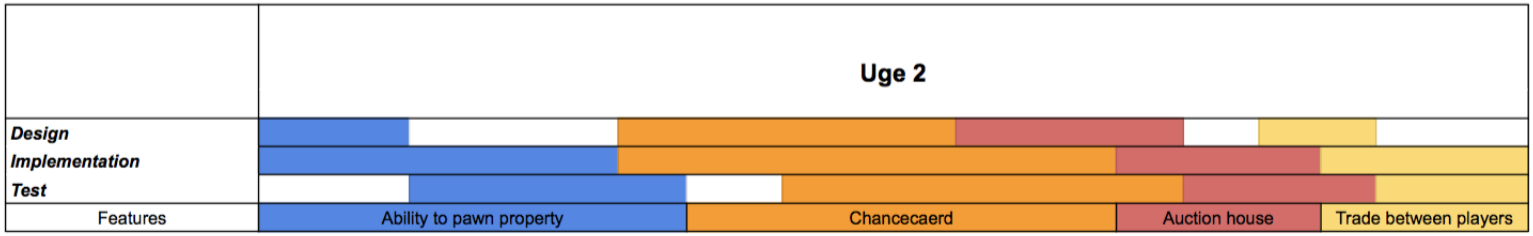
Iteration Table Full

Iterations	Phases	Tasks/Processes
1#	Elaboration	• Read and understand the project description • Plan how the development process should go • Define the requirements and analyze the use cases • Figure out all risks in the project • Modelling diagrams which describe the user interaction with the system • Start defining the functions and designing the class diagrams • Start defining the simple test cases • Check if the design of the system will work with small parts of implementation and testing
2#	Construction	• Finalize all use-cases • Implement all the classes that has been designed in the class diagram • Implement the test cases as unit tests • Test how all systems performance

Because the mass amount of features. We decided to working down to the bottom starting with the main features first. The following chart shows the workflow.



Iteration Table



Iteration Table

The following chart shows more specific how the workflows of the project progressed and what work flow belongs to what iteration.

	Uge 1					Uge 2					
	Tuesday	Wednesday	Thursday	Friday	Homework	Monday	Tuesday	Wednesday	Thursday	Friday	Homework
Requirements											
Analysis											
- Use case											
- Domain model											
- Risk											
- System sequence diagram											
Design											
- Design sequence diagram											
- Design class diagram											
Implementation											
Test											
- Test case											
- Testing code											
7. Project planning											
8. Conclusion											
Iterations	1#					2#					

Iteration Table

7 Conclusion

As seen in the requirements traceability matrix, that all our requirements have been met, using our use cases.

	Use cases	
Requirements	U1	U2
R1	■	■
R2		■
R3		■
R4		■
R5	■	
R6		■
R7		■
R8		■
R9		■
R10		■
R11	■	■
R12		■
R13		■
R14		■
R15		■
R16		■

Traceability Matrix diagram

7.1 Product oriented conclusion

We ended up being able to implement all the features we planned, which is a complete replication of the board game.
The major features are, ability to buy property and build, ability to build houses and hotel while following the original building rules of matador. Ability to pawn and unpawn properties. Implementation all fields including chance cards, that are shuffled at the beginning, and then when used put at the bottom of the card pile. Ability to trade properties within players, and to auction properties if no one wishes to buy a property.

We have not made any compromises and we are overall happy with the result. It is quite clear that we worked in an iterative approach as we have redone parts thought the project as we realised things could be done in a smarter and better way. We are also very happy with the code quality, we have tried to maintain high cohesion and low coupling as much as possible and follow the grasp patterns to make our code higher quality, easier to maintain and re-use.

Dansk:
Vi endte ud med at have implementeret alle de egenskaber og features vi planlagde, hvilket er en fuldstændig kopi af brætspillet.

De mest mærkværdige funktioner er følgende. Mulighed for at købe grunde, samt mulighed for bebyggelse i form af huse og hoteller. Alle felter er funktionelle inklusiv chance card, hvor kortbunken bliver blandet i starten, og når der bliver trukket et kort så bliver det lagt i bunden. Vi har også implementeret mulighed for at pantsætte samt bytning af grunde. Til sidst har vi også implementeret muligheden for auktion af grunde.

Vi har ikke lavet nogle kompromier og er overordnet glad med resultatet. Det står ret klart at vi har arbejdet på en iterativ måde hvor vi har lavet nogle dele om og konstant prøvet på at gøre vores produkt så godt som muligt. Vi er også meget glade ift. Kode kvalitet, ved at følge grasp principperne og gå efter så høj samhørighed og lav kobling er vi kommet ud med et godt resultat der er af høj kvalitet, nemt at genbruge og vedligeholde.

7.2 Process oriented conclusion

Everything went great, using UP we managed to reevaluate our work and keep on schedule. We got a great insight into why it's good to write good and reusable code as we were able to re-use some parts of our previous cdio projects and not just skip the analysis and design part of the project. Additionally we could easily refactor and restructure our code. There is no doubt that this project has been challenging in it's own way, the constraints placed by the project in terms

of what could be used and the gui has made certain things challenging to implement, but we still managed to get it all done in an alright manner.

Dansk:
Alt er gået godt, ved brug af Unified process har vi formået at gen evaluere vores arbejde og holde os til tidsplanen konstanten. Vi har også fået en dybere og vigtig indsigt i hvorfor det er vigtigt at skrive god kode fra starten af og ikke bare springe design eller analysedelen over. Et resultat af overståede er at vi nemt har kunne refactor og restrukturere vores kode. Der er ingen tvivl om at dette projekt har været udfordrende på sin egen vis, de begrænsninger der er sat af projekter i forhold til hvad der kan benyttes og guien har gjort at visse ting har været udfordrende at få implementeret på ordentlig vis. Alligevel følger vi at vi er kommet helskinnet i mål.

7.3 Thoughts for future projects

We've had some issues with encoding which could be improved for future projects.

Dansk:
Vi har haft nogle få problemer med encoding, hvilket er noget vi kunne arbejde på.

Glossary

Definition of the glossary

The glossary contains a complete list of the definitions of the technical terms and the abbreviations. The definitions will have a specific important for the system and the report.

Definitions, acronyms and abbreviations

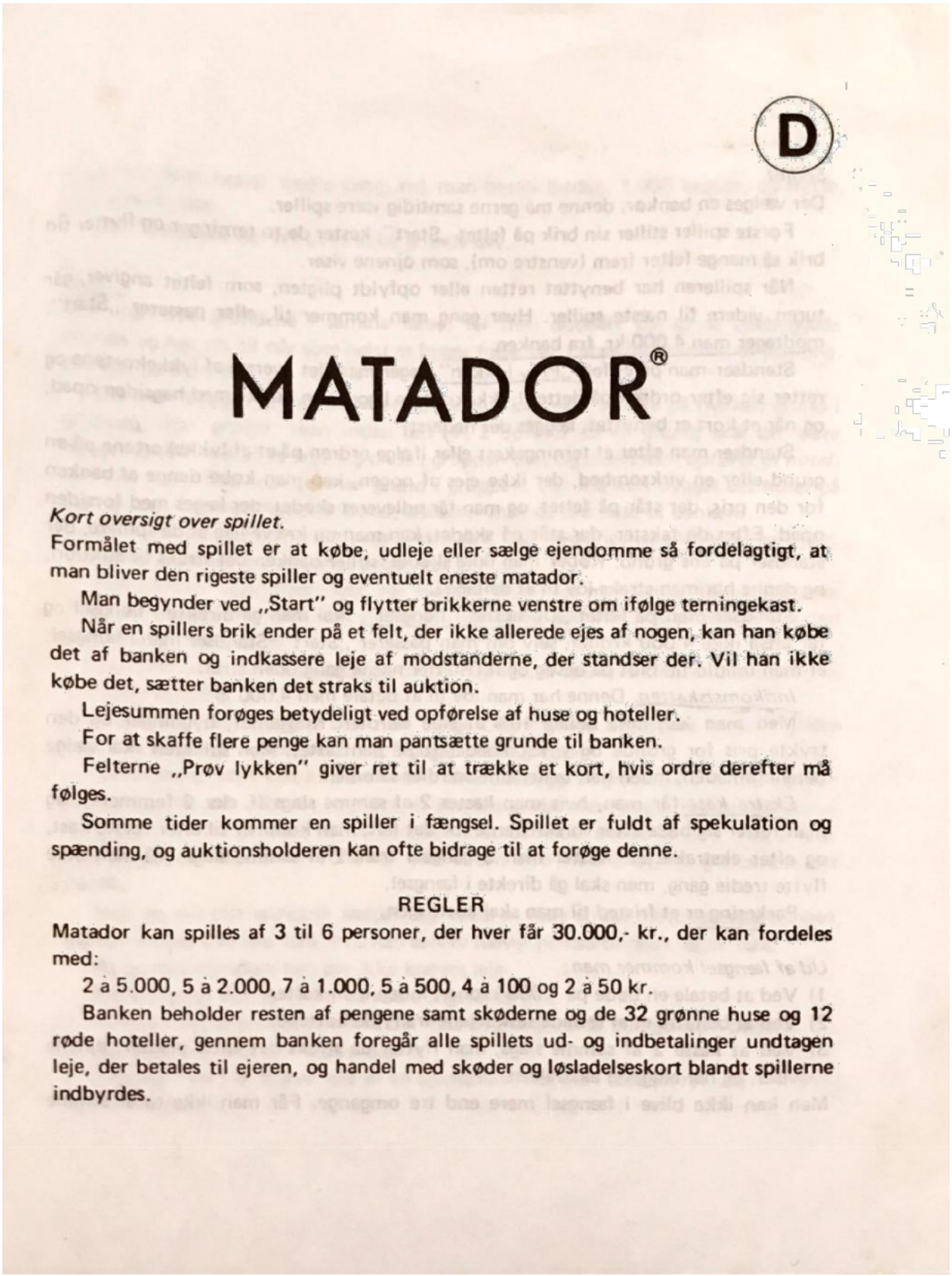
Glossary

Name	Definition
Java**	Java is a ***OOP*** language that will run on any ***operating system***.
GUI	GUI stands for graphical user interface, which allow the users to interact with the program through graphical icons and animation.
Eclipse	Eclipse is an ***IDE*** which includes extra tools for programming with Java.
IDE	IDE stands for integrated development environment, which is an software program where developers write, test and run their code. Tools are included to help developers be more productive.
OOP	OOP stands for object-oriented programming. It's programming with ***object***.
Object**	Objects is something you can find in the real-world which both have behaviours and states. Some examples could be a dog, car, mountain bike and so on.
Class**	Classes are used as placeholders for fields and methods. They are also used as templates for manufacturing ***objects***.

* The references are in italic

** Some of the definitions are reused from previous projects (CDIO 0, CDIO 1 and Roskilde projekt)

Appendix



D

MATADOR®

Kort oversigt over spillet.

Formålet med spillet er at købe, udleje eller sælge ejendomme så fordelagtigt, at man bliver den rigeste spiller og eventuelt eneste matador.

Man begynder ved „Start" og flytter brikkerne venstre om ifølge terningekast.

Når en spillers brik ender på et felt, der ikke allerede ejes af nogen, kan han købe det af banken og indkassere leje af modstanderne, der standser der. Vil han ikke købe det, sætter banken det straks til 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 må følges.

Somme tider kommer en spiller i fængsel. Spillet er fuldt af spekulation og spænding, og auktionsholderen kan ofte bidrage til at forøge denne.

REGLER

Matador kan spilles af 3 til 6 personer, der hver får 30.000,- kr., der kan fordeles med:

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

Banken beholder resten af pengene samt skøderne og de 32 grønne huse og 12 røde hoteller, gennem banken foregår alle spillets ud- og indbetalinger undtagen leje, der betales til ejeren, og handel med skøder og løsladelseskort blandt spillerne indbyrdes.

Spillet.
Der vælges en bankør; denne må gerne samtidig være spiller.

Første spiller stiller sin brik på feltet „Start", kaster de to terninger og flytter sin brik så mange felter frem (venstre om), som øjnene viser.

Når spilleren har benyttet retten eller opfyldt pligten, som feltet angiver, går turen videre til næste spiller. Hver gang man kommer til, eller passerer „Start", modtager man 4.000 kr. fra banken.

Standser man på et felt „Prøv lykken", tager man det øverste af lykkekortene og retter sig efter ordren på dette. Lykkekortene ligger i en bunke med bagsiden opad, og når et kort er benyttet, lægges det nederst.

Standser man efter et terningekast eller ifølge ordren på et af lykkekortene på en grund eller en virksomhed, der ikke ejes af nogen, kan man købe denne af banken for den pris, der står på feltet, og man får udleveret skødet, der læges med forsiden opad. Efter de takster, der står på skødet, kan man nu kræve leje af de spillere, der standser på ens grund. Køber man ikke skødet, stiller banken det straks til auktion, og denne har man straks lov til at deltage i.

Standser man på feltet „De sættes i fængsel", skal man gå direkte i fængsel og får altså ikke 4.000 kroner, selv om man passerer „Start". Standser man i fængsel, er man imidlertid blot på besøg og fortsætter næste gang uden straf.

Indkomstskatten. Denne 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). Men spilleren skal vælge betalingsmåden, inden han tæller sine værdier sammen.

Ekstra kast får man, hvis man kaster 2 af samme slags (f. eks. 2 femmere), og man retter sig både efter forskrifterne for det felt, man kommer til efter første kast, og efter ekstrakastet. Kaster man 3 gange i træk 2 af samme slags, må man ikke flytte tredie gang, men skal gå direkte i fængsel.

Parkering er et fristed til man skal kaste igen.

Ud af fængsel kommer man:
1) Ved at betale en bøde på 1.000 kroner, inden man kaster.
2) Ved at benytte et af løsladelseskortene fra lykkekortene.
3) Ved at kaste 2 af samme slags; man flytter da straks det antal pladser, øjnene viser, og har alligevel ekstrakast.

Man kan ikke blive i fængsel mere end tre omgange. Får man ikke to af samme slags, når man kaster tredie gang, må man som øjnene viser.

Den fængslede har ret til at købe grunde o

Huse.
Ejer man alle grundene i samme farve, få grunde og har ret til når som helst at bygge h der står på skøderne.

Der skal bygges jævnt, dvs. det første h gruppen, man ønsker; men inden hus nr. 2 bygget eet på hver af de andre grunde i grupp *skal der være fire huse på hver grund i grup* hver grund. Når man køber et hotel, afleverer

Banken skal, når som helst man ønsker de Prisen for et hotel er fem gange prisen for et h

Har banken ingen bygninger, når man vil nogle tilbage. Er der flere, der vil købe, og b dem, der er, til auktion.

Indbyrdes handel med *ubebyggede* grund kan blive enig om.

N.B.! Har man bygget, skal man sælge man kan afhænde nogen grund i den pågælde

Pantsætning. Man kan kun pantsætte sine det beløb, der står påtrykt bagsiden af skøde skal man først sælge disse til banken. Spillere bagsiden opad. Renten er 10%, der betales s hæves.

Hvis en pantsat ejendom sælges, og købe må han alligevel betale 10%, hvis han senere h

Af pantsat ejendom kan der ikke kræves le

Banken giver kun løn mod pantsætningssik

Pantsætning af grunde samt handel med by

Spillerne må ikke låne indbyrdes.

Glemmer man at kræve leje af en medspil ham har kastet.

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 går ud af spillet.

Er det banken, der er kreditor, sælger han straks modtagne grunde ved auktion.

HURTIGT SPIL

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

Der bestemmes en spilletid, og når tiden er omme, har den vundet, der har størst formue.

Nr. 18091/239-240


MATADOR
START
Hvergang De passerer modtag kr. 4.000
Rådhus-pladsen
kr. 8.000
Ekstra-ordinær statsskat betal kr. 2.000
Frederiksberg-gade
kr. 7.000
Prøv lykken
Skandinavisk Linietrafik A/S
kr. 4.000
Nygade
kr. 6.400
Vimmelskaftet
kr. 6.000
Amagertorv
kr. 6.000
De Fængsles
Østergade
kr. 5.600
Faxe Bryggeri
kr. 3.000
Kgs. Nytorv
kr. 5.200
Bredgade
kr. 5.200
Mols-Linien A/S
kr. 4.000
Grønningen
kr. 4.800
Østerbrogade
kr. 4.400
Trianglen
kr. 4.400
Parkering
Strandvej
kr. 4.000
Hellerupvej
kr. 3.600
Bernstorffsvej
kr. 3.600
A/S Grenå Hundested Færgefart
kr. 4.000
Gl. Kongevej
kr. 3.200
Bülowsvej
kr. 2.800
Allégade
kr. 2.400
Roskildevej
Hvidovrevej
kr. 1.200
Rødovrevej
kr. 1.200
POLITI


FREDERIKSBERGGADE
Leje af grund kr. 700
m/ 1 hus » 3.500
» 2 huse » 10.000
» 3 huse » 22.000
» 4 huse » 26.000
» hotel » 30.000
Hvis en spiller ejer alle grundene i een farvegruppe, fordobles lejen af de ubebyggede grunde i den pågældende gruppe
Hvert hus koster kr. 4.000
Et hotel koster » 4.000 + 4 huse
Pantsætningsværdi kr. 3.500
RÅDHUSPLADSEN
Leje af grund kr. 1.000
m/ 1 hus » 4.000
» 2 huse » 12.000
» 3 huse » 28.000
» 4 huse » 34.000
» hotel » 40.000
Hvis en spiller ejer alle grundene i een farvegruppe fordobles lejen af de ubebyggede grunde i den pågældende gruppe
Hvert hus koster kr. 4.000
Et hotel koster » 4.000 + 4 huse
Pantsætningsværdi kr. 4.000
RØDOVREVEJ
Leje af grund kr. 50
m/ 1 hus » 250
» 2 huse » 750
» 3 huse » 2.250
» 4 huse » 4.000
» hotel » 6.000
Hvis en spiller ejer alle grundene i een farvegruppe, fordobles lejen af de ubebyggede grunde i den pågældende gruppe
Hvert hus koster kr. 1.000
Et hotel koster » 1.000 + 4 huse
Pantsætningsværdi kr. 600
HVIDOVREVEJ
Leje af grund kr. 50
m/ 1 hus » 250
» 2 huse » 750
» 3 huse » 2.250
» 4 huse » 4.000
» hotel » 6.000
Hvis en spiller ejer alle grundene i een farvegruppe, fordobles lejen af de ubebyggede grunde i den pågældende gruppe
Hvert hus koster kr. 1.000
Et hotel koster » 1.000 + 4 huse
Pantsætningsværdi kr. 600


PANTSAT
FREDERIKSBERGGADE
Pantsat for kr. 3.500
Kortet skal vende denne side opad, hvis ejendommen er pantsat.
PANTSAT
RÅDHUSPLADSEN
Pantsat for kr. 4.000
Kortet skal vende denne side opad, hvis ejendommen er pantsat.
PANTSAT
HVIDOVREVEJ
Pantsat for kr. 600
Kortet skal vende denne side opad, hvis ejendommen er pantsat.
PANTSAT
RØDOVREVEJ
Pantsat for kr. 600
Kortet skal vende denne side opad, hvis ejendommen er pantsat.

VIMMELSKAFTET

Leje af grund	kr.	550
m/ 1 hus	»	2.600
» 2 huse	»	7.800
» 3 huse	»	18.000
» 4 huse	»	22.000
» hotel	»	25.000

Hvis en spiller ejer **alle** grundene i een farvegruppe, fordobles lejen af de ubebyggede grunde i den pågældende gruppe

Hvert hus koster	kr.	4.000
Et hotel koster	»	4.000
	+	4 huse

Pantsætningsværdi **kr. 3.000**

NYGADE

Leje af grund	kr.	600
m/ 1 hus	»	3.000
» 2 huse	»	9.000
» 3 huse	»	20.000
» 4 huse	»	24.000
» hotel	»	28.000

Hvis en spiller ejer **alle** grundene i een farvegruppe, fordobles lejen af de ubebyggede grunde i den pågældende gruppe

Hvert hus koster	kr.	4.000
Et hotel koster	»	4.000
	+	4 huse

Pantsætningsværdi **kr. 3.200**

AMAGERTORV

Leje af grund	kr.	550
m/ 1 hus	»	2.600
» 2 huse	»	7.800
» 3 huse	»	18.000
» 4 huse	»	22.000
» hotel	»	25.000

Hvis en spiller ejer **alle** grundene i een farvegruppe, fordobles lejen af de ubebyggede grunde i den pågældende gruppe

Hvert hus koster	kr.	4.000
Et hotel koster	»	4.000
	+	4 huse

Pantsætningsværdi **kr. 3.000**

VALBY LANGGADE

Leje af grund	kr.	100
m/ 1 hus	»	600
» 2 huse	»	1.800
» 3 huse	»	5.400
» 4 huse	»	8.000
» hotel	»	11.000

Hvis en spiller ejer **alle** grundene i een farvegruppe, fordobles lejen af de ubebyggede grunde i den pågældende gruppe

Hvert hus koster	kr.	1.000
Et hotel koster	»	1.000
	+	4 huse

Pantsætningsværdi **kr. 1.000**

ROSKILDEVEJ

Leje af grund	kr.	100
m/ 1 hus	»	600
» 2 huse	»	1.800
» 3 huse	»	5.400
» 4 huse	»	8.000
» hotel	»	11.000

Hvis en spiller ejer **alle** grundene i een farvegruppe, fordobles lejen af de ubebyggede grunde i den pågældende gruppe

Hvert hus koster	kr.	1.000
Et hotel koster	»	1.000
	+	4 huse

Pantsætningsværdi **kr. 1.000**

ALLÉGADE

Leje af grund	kr.	150
m/ 1 hus	»	800
» 2 huse	»	2.000
» 3 huse	»	6.000
» 4 huse	»	9.000
» hotel	»	12.000

Hvis en spiller ejer **alle** grundene i een farvegruppe, fordobles lejen af de ubebyggede grunde i den pågældende gruppe

Hvert hus koster	kr.	1.000
Et hotel koster	»	1.000
	+	4 huse

Pantsætningsværdi **kr. 1.200**

GL. KONGEVEJ

Leje af grund	kr.	250
m/ 1 hus	»	1.250
» 2 huse	»	3.750
» 3 huse	»	10.000
» 4 huse	»	14.000
» hotel	»	18.000

Hvis en spiller ejer **alle** grundene i een farvegruppe, fordobles lejen af de ubebyggede grunde i den pågældende gruppe

Hvert hus koster	kr.	2.000
Et hotel koster	»	2.000
	+	4 huse

Pantsætningsværdi **kr. 1.600**

FREDERIKSBERG ALLÉ

Leje af grund	kr.	200
m/ 1 hus	»	1.000
» 2 huse	»	3.000
» 3 huse	»	9.000
» 4 huse	»	12.500
» hotel	»	15.000

Hvis en spiller ejer **alle** grundene i een farvegruppe, fordobles lejen af de ubebyggede grunde i den pågældende gruppe

Hvert hus koster	kr.	2.000
Et hotel koster	»	2.000
	+	4 huse

Pantsætningsværdi **kr. 1.400**

BÜLOWSVEJ

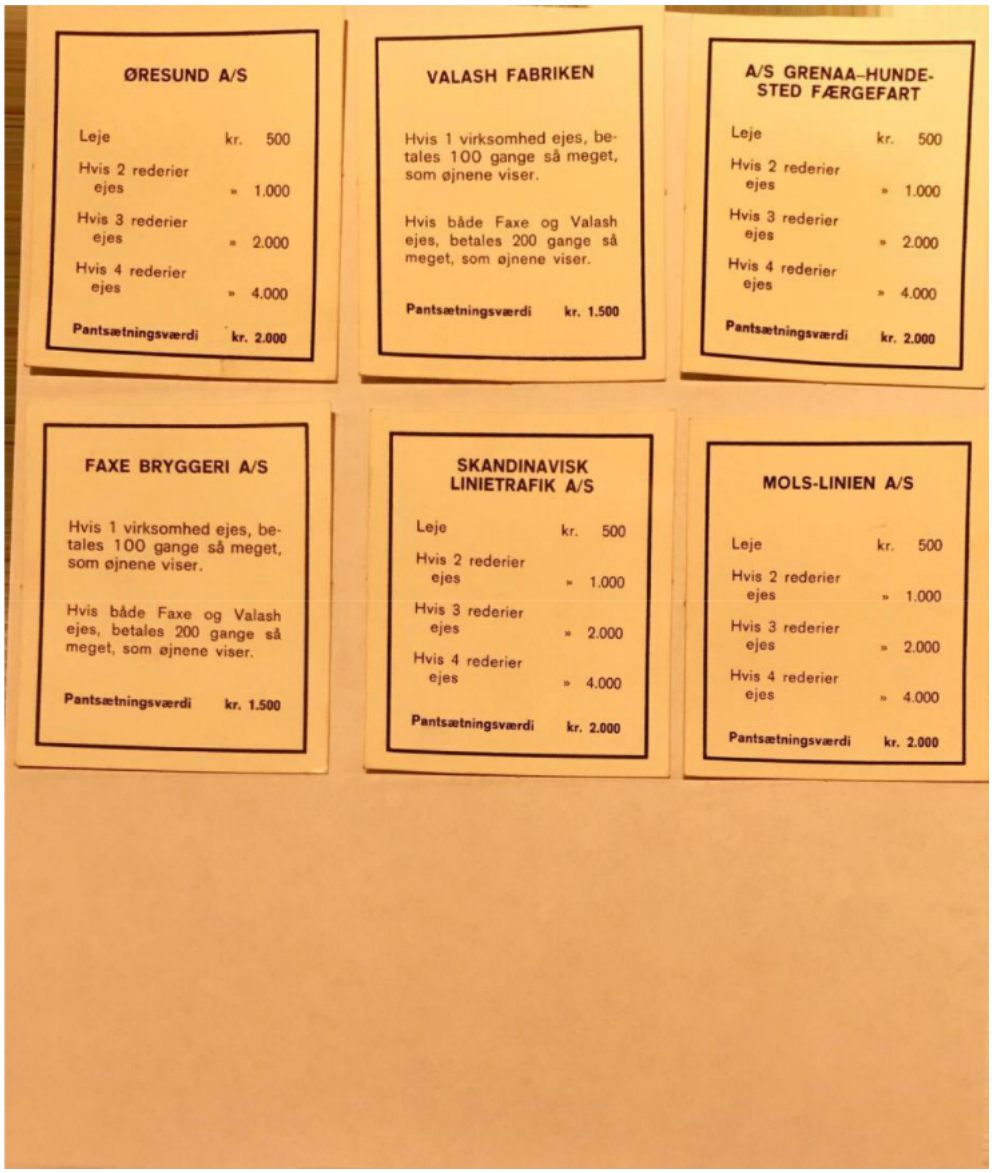
Leje af grund	kr.	200
m/ 1 hus	»	1.000
» 2 huse	»	3.000
» 3 huse	»	9.000
» 4 huse	»	12.500
» hotel	»	15.000

Hvis en spiller ejer **alle** grundene i een farvegruppe, fordobles lejen af de ubebyggede grunde i den pågældende gruppe

Hvert hus koster	kr.	2.000
Et hotel koster	»	2.000
	+	4 huse

Pantsætningsværdi **kr. 1.400**


PANTSAT
VIMMELSKAFTET
Pantsat for kr. 3.000
Kortet skal vende denne side opad, hvis ejendommen er pantsat.
PANTSAT
NYGADE
Pantsat for kr. 3.200
Kortet skal vende denne side opad, hvis ejendommen er pantsat.
PANTSAT
AMAGERTORV
Pantsat for kr. 3.000
Kortet skal vende denne side opad, hvis ejendommen er pantsat.
PANTSAT
VALBY LANGGADE
Pantsat for kr. 1.000
Kortet skal vende denne side opad, hvis ejendommen er pantsat.
PANTSAT
ROSKILDEVEJ
Pantsat for kr. 1.000
Kortet skal vende denne side opad, hvis ejendommen er pantsat.
PANTSAT
ALLÉGADE
Pantsat for kr. 1.200
Kortet skal vende denne side opad, hvis ejendommen er pantsat.
PANTSAT
GL. KONGEVEJ
Pantsat for kr. 1.600
Kortet skal vende denne side opad, hvis ejendommen er pantsat.
PANTSAT
FREDERIKSBERG ALLÉ
Pantsat for kr. 1.400
Kortet skal vende denne side opad, hvis ejendommen er pantsat.
PANTSAT
BÜLOWSVEJ
Pantsat for kr. 1.400
Kortet skal vende denne side opad, hvis ejendommen er pantsat.


ØRESUND A/S
Leje kr. 500
Hvis 2 rederier ejes » 1.000
Hvis 3 rederier ejes » 2.000
Hvis 4 rederier ejes » 4.000
Pantsætningsværdi kr. 2.000
VALASH FABRIKEN
Hvis 1 virksomhed ejes, betales 100 gange så meget, som øjnene viser.
Hvis både Faxe og Valash ejes, betales 200 gange så meget, som øjnene viser.
Pantsætningsværdi kr. 1.500
A/S GRENAA–HUNDESTED FÆRGEFART
Leje kr. 500
Hvis 2 rederier ejes » 1.000
Hvis 3 rederier ejes » 2.000
Hvis 4 rederier ejes » 4.000
Pantsætningsværdi kr. 2.000
FAXE BRYGGERI A/S
Hvis 1 virksomhed ejes, betales 100 gange så meget, som øjnene viser.
Hvis både Faxe og Valash ejes, betales 200 gange så meget, som øjnene viser.
Pantsætningsværdi kr. 1.500
SKANDINAVISK LINIETRAFIK A/S
Leje kr. 500
Hvis 2 rederier ejes » 1.000
Hvis 3 rederier ejes » 2.000
Hvis 4 rederier ejes » 4.000
Pantsætningsværdi kr. 2.000
MOLS-LINIEN A/S
Leje kr. 500
Hvis 2 rederier ejes » 1.000
Hvis 3 rederier ejes » 2.000
Hvis 4 rederier ejes » 4.000
Pantsætningsværdi kr. 2.000


PANTSAT
ØRESUND A/S
Pantsat for kr. 2.000
Kortet skal vende denne side opad, hvis selskabet er pantsat.
PANTSAT
VALASH FABRIKEN
Pantsat for kr. 1.500
Kortet skal vende denne side opad, hvis virksomheden er pantsat.
PANTSAT
A/S GRENAA-HUNDE-STED FÆRGEFART
Pantsat for kr. 2.000
Kortet skal vende denne side opad, hvis selskabet er pantsat.
PANTSAT
FAXE BRYGGERI A/S
Pantsat for kr. 1.500
Kortet skal vende denne side opad, hvis virksomheden er pantsat.
PANTSAT
SKANDINAVISK LINIETRAFIK A/S
Pantsat for kr. 2.000
Kortet skal vende denne side opad, hvis selskabet er pantsat.
PANTSAT
MOLS-LINIEN A/S
Pantsat for kr. 2.000
Kortet skal vende denne side opad, hvis selskabet er pantsat.

TRIANGLEN

Leje af grund	kr.	350
m/ 1 hus	»	1.800
» 2 huse	»	5.000
» 3 huse	»	14.000
» 4 huse	»	17.500
» hotel	»	21.000

Hvis en spiller ejer **alle** grundene i een farvegruppe, fordobles lejen af de ubebyggede grunde i den pågældende gruppe

Hvert hus koster	kr.	3.000
Et hotel koster	»	3.000
	+	4 huse
Pantsætningsværdi	**kr.**	**2.200**

ØSTERBROGADE

Leje af grund	kr.	350
m/ 1 hus	»	1.800
» 2 huse	»	5.000
» 3 huse	»	14.000
» 4 huse	»	17.500
» hotel	»	21.000

Hvis en spiller ejer **alle** grundene i een farvegruppe, fordobles lejen af de ubebyggede grunde i den pågældende gruppe

Hvert hus koster	kr.	3.000
Et hotel koster	»	3.000
	+	4 huse
Pantsætningsværdi	**kr.**	**2.200**

GRØNNINGEN

Leje af grund	kr.	400
m/ 1 hus	»	2.000
» 2 huse	»	6.000
» 3 huse	»	15.000
» 4 huse	»	18.500
» hotel	»	22.000

Hvis en spiller ejer **alle** grundene i een farvegruppe, fordobles lejen af de ubebyggede grunde i den pågældende gruppe

Hvert hus koster	kr.	3.000
Et hotel koster	»	3.000
	+	4 huse
Pantsætningsværdi	**kr.**	**2.400**

ØSTERGADE

Leje af grund	kr.	500
m/ 1 hus	»	2.400
» 2 huse	»	7.200
» 3 huse	»	17.000
» 4 huse	»	20.500
» hotel	»	24.000

Hvis en spiller ejer **alle** grundene i een farvegruppe, fordobles lejen af de ubebyggede grunde i den pågældende gruppe

Hvert hus koster	kr.	3.000
Et hotel koster	»	3.000
	+	4 huse
Pantsætningsværdi	**kr.**	**2.800**

BREDGADE

Leje af grund	kr.	450
m/ 1 hus	»	2.200
» 2 huse	»	6.600
» 3 huse	»	16.000
» 4 huse	»	19.500
» hotel	»	23.000

Hvis en spiller ejer **alle** grundene i een farvegruppe, fordobles lejen af de ubebyggede grunde i den pågældende gruppe

Hvert hus koster	kr.	3.000
Et hotel koster	»	3.000
	+	4 huse
Pantsætningsværdi	**kr.**	**2.600**

KGS. NYTORV

Leje af grund	kr.	450
m/ 1 hus	»	2.200
» 2 huse	»	6.600
» 3 huse	»	16.000
» 4 huse	»	19.500
» hotel	»	23.000

Hvis en spiller ejer **alle** grundene i een farvegruppe, fordobles lejen af de ubebyggede grunde i den pågældende gruppe

Hvert hus koster	kr.	3.000
Et hotel koster	»	3.000
	+	4 huse
Pantsætningsværdi	**kr.**	**2.600**

STRANDVEJ

Leje af grund	kr.	350
m/ 1 hus	»	1.600
» 2 huse	»	4.400
» 3 huse	»	12.000
» 4 huse	»	16.000
» hotel	»	20.000

Hvis en spiller ejer **alle** grundene i een farvegruppe, fordobles lejen af de ubebyggede grunde i den pågældende gruppe

Hvert hus koster	kr.	2.000
Et hotel koster	»	2.000
	+	4 huse
Pantsætningsværdi	**kr.**	**2.000**

BERNSTORFFSVEJ

Leje af grund	kr.	300
m/ 1 hus	»	1.400
» 2 huse	»	4.000
» 3 huse	»	11.000
» 4 huse	»	15.000
» hotel	»	19.000

Hvis en spiller ejer **alle** grundene i een farvegruppe, fordobles lejen af de ubebyggede grunde i den pågældende gruppe

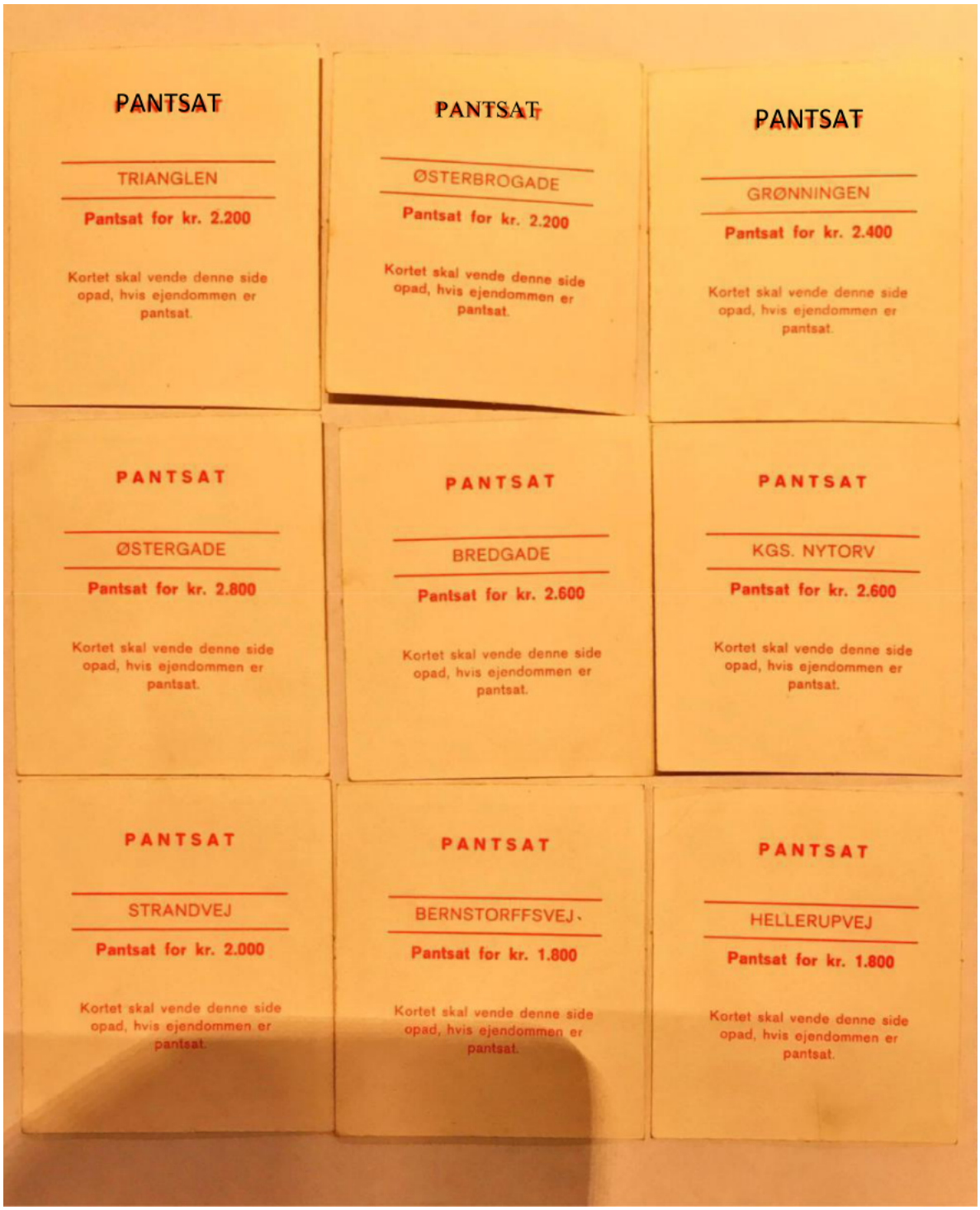
Hvert hus koster	kr.	2.000
Et hotel koster	»	2.000
	+	4 huse
Pantsætningsværdi	**kr.**	**1.800**

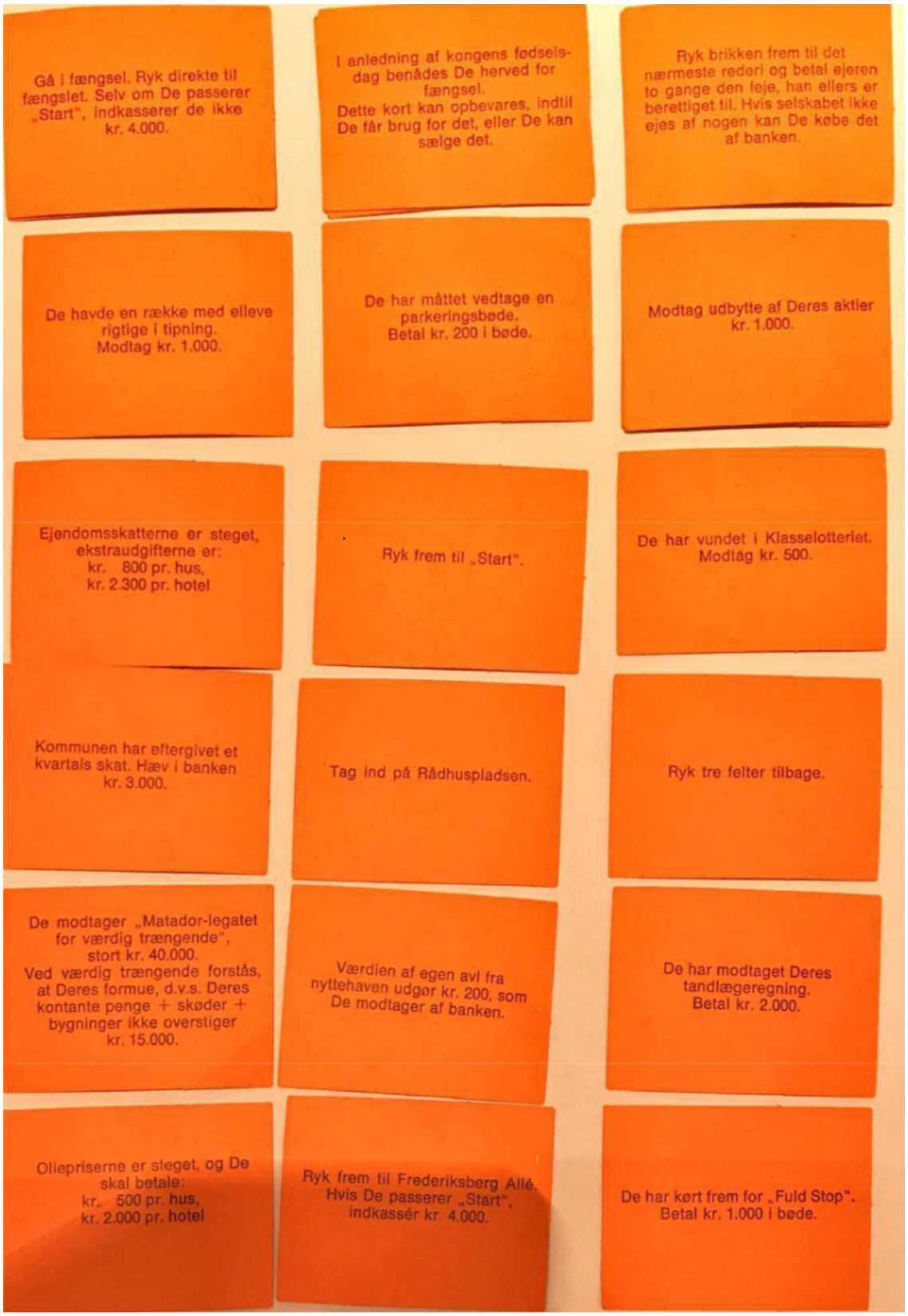
HELLERUPVEJ

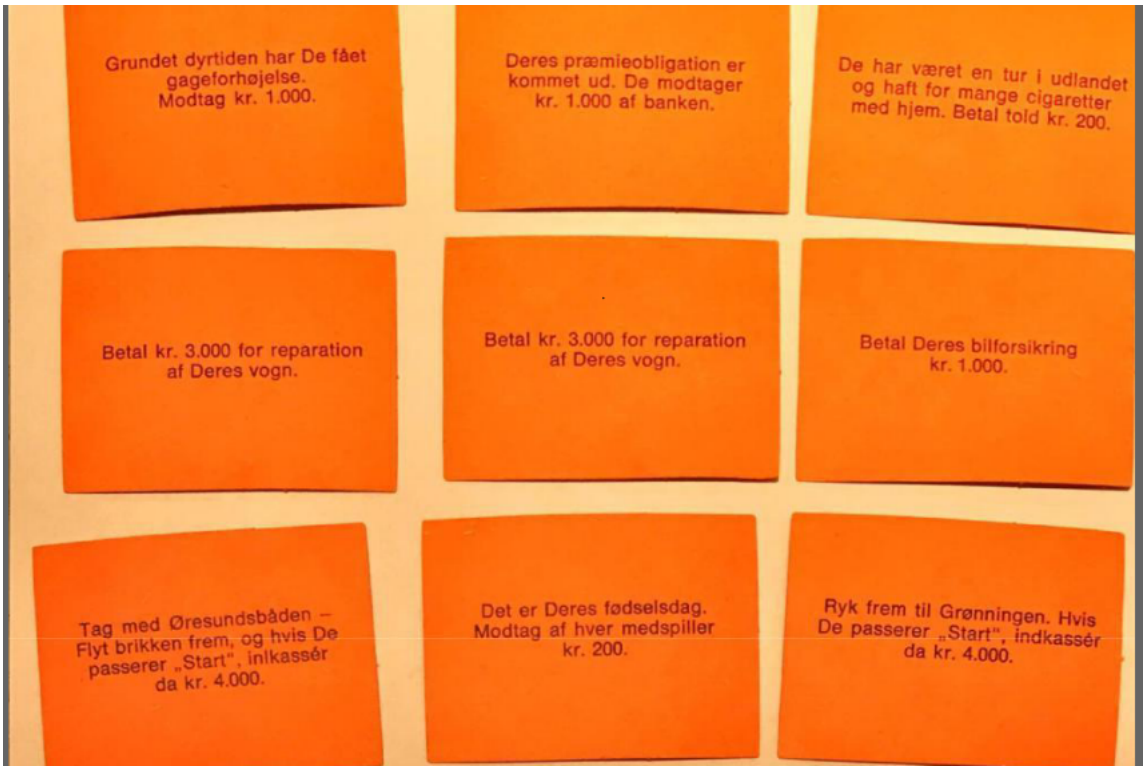
Leje af grund	kr.	300
m/ 1 hus	»	1.400
» 2 huse	»	4.000
» 3 huse	»	11.000
» 4 huse	»	15.000
» hotel	»	19.000

Hvis en spiller ejer **alle** grundene i een farvegruppe, fordobles lejen af de ubebyggede grunde i den pågældende gruppe

Hvert hus koster	kr.	2.000
Et hotel koster	»	2.000
	+	4 huse
Pantsætningsværdi	**kr.**	**1.800**


PANTSAT
TRIANGLEN
Pantsat for kr. 2.200
Kortet skal vende denne side opad, hvis ejendommen er pantsat.
PANTSAT
ØSTERBROGADE
Pantsat for kr. 2.200
Kortet skal vende denne side opad, hvis ejendommen er pantsat.
PANTSAT
GRØNNINGEN
Pantsat for kr. 2.400
Kortet skal vende denne side opad, hvis ejendommen er pantsat.
PANTSAT
ØSTERGADE
Pantsat for kr. 2.800
Kortet skal vende denne side opad, hvis ejendommen er pantsat.
PANTSAT
BREDGADE
Pantsat for kr. 2.600
Kortet skal vende denne side opad, hvis ejendommen er pantsat.
PANTSAT
KGS. NYTORV
Pantsat for kr. 2.600
Kortet skal vende denne side opad, hvis ejendommen er pantsat.
PANTSAT
STRANDVEJ
Pantsat for kr. 2.000
Kortet skal vende denne side opad, hvis ejendommen er pantsat.
PANTSAT
BERNSTORFFSVEJ
Pantsat for kr. 1.800
Kortet skal vende denne side opad, hvis ejendommen er pantsat.
PANTSAT
HELLERUPVEJ
Pantsat for kr. 1.800
Kortet skal vende denne side opad, hvis ejendommen er pantsat.


Gå i fængsel. Ryk direkte til fængslet. Selv om De passerer „Start", indkasserer de ikke kr. 4.000.
I anledning af kongens fødselsdag benådes De herved for fængsel.
Dette kort kan opbevares, indtil De får brug for det, eller De kan sælge det.
Ryk brikken frem til det nærmeste rederi og betal ejeren to gange den leje, han ellers er berettiget til. Hvis selskabet ikke ejes af nogen kan De købe det af banken.
De havde en række med elleve rigtige i tipning. Modtag kr. 1.000.
De har måttet vedtage en parkeringsbøde. Betal kr. 200 i bøde.
Modtag udbytte af Deres aktier kr. 1.000.
Ejendomsskatterne er steget, ekstraudgifterne er: kr. 800 pr. hus, kr. 2.300 pr. hotel
Ryk frem til „Start".
De har vundet i Klasselotteriet. Modtag kr. 500.
Kommunen har eftergivet et kvartals skat. Hæv i banken kr. 3.000.
Tag ind på Rådhuspladsen.
Ryk tre felter tilbage.
De modtager „Matador-legatet for værdig trængende", stort kr. 40.000. Ved værdig trængende forstås, at Deres formue, d.v.s. Deres kontante penge + skøder + bygninger ikke overstiger kr. 15.000.
Værdien af egen avl fra nyttehaven udgør kr. 200, som De modtager af banken.
De har modtaget Deres tandlægeregning. Betal kr. 2.000.
Oliepriserne er steget, og De skal betale: kr. 500 pr. hus, kr. 2.000 pr. hotel
Ryk frem til Frederiksberg Allé. Hvis De passerer „Start", indkassér kr. 4.000.
De har kørt frem for „Fuld Stop". Betal kr. 1.000 i bøde.


Grundet dyrtiden har De fået gageforhøjelse. Modtag kr. 1.000.
Deres præmieobligation er kommet ud. De modtager kr. 1.000 af banken.
De har været en tur i udlandet og haft for mange cigaretter med hjem. Betal told kr. 200.
Betal kr. 3.000 for reparation af Deres vogn.
Betal kr. 3.000 for reparation af Deres vogn.
Betal Deres bilforsikring kr. 1.000.
Tag med Øresundsbåden – Flyt brikken frem, og hvis De passerer „Start", inkassér da kr. 4.000.
Det er Deres fødselsdag. Modtag af hver medspiller kr. 200.
Ryk frem til Grønningen. Hvis De passerer „Start", indkassér da kr. 4.000.

Account Source Code

```
1
2  /**
3   *
4   */
5  package core;
6
7  /**
8   * @author Magnus Stjernborg Koch — s175189
9   *
10  * This class is an Account object which contains the player's balance
11  */
12 public class Account {
13
14     private int balance;
15
16     /**
17      * Constructor which sets the account value to the default value
18      */
19     public Account() {
20         this.balance = 30000;
21     }
22
23     /**
24      * Getter for the balance
25      * @return the balance
26      */
27     public int getBalance() {
28         return this.balance;
29     }
30
31     /**
32      * Setter for the balance
33      * @param balance
34      */
35     public void setBalance(int balance) {
36         this.balance = balance;
37     }
38
39     /**
40      * Deposit function for balance
41      * @param value
42      */
43     public void deposit (int value) {
44         //Makes sure that a negative value would not effect the balance
45          if (value >= 0)
46             this.balance += value;
47     }
48
49     /**
50      * Withdraw function for balance
51      * @param value
52      */
53     public void withdraw(int value) {
54         //Ensure that that a negative value would function
55         this.balance -= Math.abs(value);
56         //Makes sure that that we do not end up with a negative balance
57         if (this.balance < 0)
58             this.balance = 0;
59     }
60
61     /**
62      *
63      * Checks if the player has the balance required to buy a specific field
64      * @param value
65      * @return boolean
66      */
67     public boolean canAfford(int value) {
68         if(this.balance - value>= 0) {
69             return true;
70         }else {
71             return false;
72         }
73     }
74 }
75
```

AuctionController Source Code

```
package core;
/**
 * @author Magnus Stjernborg Koch — s175189
 *
 */
public class AuctionController {
    private Player[] bidders;
    private boolean auctionStatus;
    private int highestBid;
    private Player whoHasTheHighestBid;
    private GUIController guiController = GUIController.getInstance();
    AuctionController() {
        whoHasTheHighestBid = null;
    }
    /**
     * This function is called when a new auction needs to start. Its public so other controllers can call the function
     */
    public void startAuction(Player playerNotIncluded, Field field, Player[] players) {
        //auction is now started
        auctionStatus = true;
        //amount of players in auction
        int playersInAuction = 0;
        //The property on auction
        Property propertyOnAuction = (Property) field;
        // initialising an array for all bidders in the current auction
        bidders = new Player[players.length];
        // initialising the highest bid, so the default highest bid depends on the property price
        highestBid = propertyOnAuction.getBuyValue()-1;
        //Loop is going through players and checking who can bet in this auction
        for (int i = 0; i < players.length; i++) {
            if (!(players[i] == playerNotIncluded))
                if (players[i].getAccount().canAfford(highestBid)) {
                    playersInAuction++;
                    bidders[i] = players[i];
                }
        }
        //The auctions starts only if there are at least one that can bet
        if (!(playersInAuction == 0)) {
            guiController.writeMessage(PropertiesIO.getTranslation("auctionon")+" "+field.getName());
            //Runs auction until the auctionstatus is changed
            while (auctionStatus) {
                boolean noMoreRounds = true;
                //Goes through all bidders
                for (Player bidder : bidders) {
                    if (!(bidder == null)) {
                        // if player has the highest bet
                        if (!(bidder == whoHasTheHighestBid)) {
                            //checks if bidder can afford highest bet
                            if (bidder.getAccount().canAfford(highestBid + 1)) {
                                switch (guiController.requestPlayerChoiceButtons(PropertiesIO.getTranslation("auctionplayer")+" " + bidder.getName() + "
                                        "+PropertiesIO.getTranslation("auctionbid")+"?", PropertiesIO.getTranslation("yesbutton"), PropertiesIO.getTranslation("nobutton"))) {
                                    case "Ja":
                                        //The amount the player bets
                                        int bid = guiController.requestIntegerInput(bidder.getName() + PropertiesIO.getTranslation("auctiontopbid") + " "+ highestBid +
                                                PropertiesIO.getTranslation("auctionyourbid"),0, bidder.getAccount().getBalance());
                                        //can player afford the bet
                                        if (bidder.getAccount().canAfford(bid)) {
                                            // is bet higher then highest bet
                                            if (bid > highestBid) {
                                                highestBid = bid;
                                                whoHasTheHighestBid = bidder;
                                                noMoreRounds = false;
                                            }
                                        }else {
                                            bid = guiController.requestIntegerInput(bidder.getName() + " " + PropertiesIO.getTranslation("auctiontoohigh") +" " + highestBid
                                                    + PropertiesIO.getTranslation("auctionyourbid"),0, bidder.getAccount().getBalance());
                                            if (bid > highestBid) {
                                                highestBid = bid;
                                                whoHasTheHighestBid = bidder;
                                                noMoreRounds = false;
                                            }
                                        }
                                        break;
                                    case "Nej":
```

```
77                                          break;
78                                      }
79                                  }
80                              } else {
81                                  guiController.writeMessage(PropertiesIO.getTranslation("auctionalreadyhighestbid"));
82                              }
83                          }
84                      }
85                      if (noMoreRounds)
86                          auctionStatus = false;
87                  }
88                  //Checks if there is a winner
89                  if (!(whoHasTheHighestBid == null)) {
90                      wonAuction(propertyOnAuction);
91                      guiController.writeMessage(whoHasTheHighestBid.getName() + " "+ PropertiesIO.getTranslation("auctionhighestbid")+" "+ propertyOnAuction.getName());
92                  } else
93                      guiController.writeMessage(PropertiesIO.getTranslation("auctionnobids"));
94              }
95          }
96          /**
97           * When someone has won the auction house the player will be set has owner of the property
98           */
99          private void wonAuction(Property propertyOnAuction) {
100             //Takes amount which the player bed from his account
101             whoHasTheHighestBid.getAccount().withdraw(highestBid);
102             //Set him as the new owner of the property
103             propertyOnAuction.setOwner(whoHasTheHighestBid);
104             guiController.updatePlayerBalance(whoHasTheHighestBid.getGuiId(),whoHasTheHighestBid.getAccount().getBalance());
105             guiController.setOwner(whoHasTheHighestBid.getGuiId(),propertyOnAuction.getId());
106         }
107 }
108
```

Brewery Controller Source Code

```
1
2  package core;
3
4  /**
5   * @author Mathias Thejsen s175192 && Simon Hansen s175191
6   *
7   */
8  public class BreweryController {
9      private int totalFaceValue;
10     private Brewery brewery;
11     private Player currentPlayer;
12     private GUIController guiController = GUIController.getInstance();
13     private AuctionController auctionController;
14
15     private Player[] players;
16
17     public BreweryController(Player currentPlayer, int totalFaceValue, Field[] fields, Player[] players) {
18         this.currentPlayer = currentPlayer;
19         this.brewery = (Brewery) fields[currentPlayer.getEndPosition()];
20         this.totalFaceValue = totalFaceValue;
21         this.players = players;
22         auctionController = new AuctionController();
23     }
24
25     /**
26      * Logic method for brewery
27      */
28     protected void logic() {
29         // Field is not owned
30         if(brewery.getOwner() == null) {
31
32             // Check if the player can afford
33             if(currentPlayer.getAccount().canAfford(brewery.getBuyValue())) {
34
35                 // Give the player choices
36                 String[] choices = {PropertiesIO.getTranslation("yesbutton"), PropertiesIO.getTranslation("nobutton")};
37                 String result = guiController.requestPlayerChoiceButtons(PropertiesIO.getTranslation("streetlandonbuy")+" "+brewery.getName(), choices);
38
39                 // If they choose Yes run buyLogic method
40                 if(result.equals(PropertiesIO.getTranslation("yesbutton"))) {
41                     BuyController buyController = new BuyController(currentPlayer, brewery);
42                     buyController.buyLogic();
43                 }
44
45                 else { // If the player does not wish to buy the field
46                     auctionController.startAuction(currentPlayer, brewery, players);
47                 }
48             }
49             else { // If the player cannot afford the field
50                 auctionController.startAuction(currentPlayer, brewery, players);
51             }
52         }
53         else{
54             // Field is owned by the player
55             if(brewery.getOwner() == currentPlayer || brewery.isPawned()) {
56
57             }else {
58                 // Set the rentPrice
59                 int rentPrice = brewery.getRentValue()*totalFaceValue;
60
61                 //We check if the landing player can afford the rent
62                 if(currentPlayer.getAccount().canAfford(rentPrice)) {
63
64                     // Notify the user
65                     guiController.writeMessage("You landed on.."+brewery.getName() + ".. 's field and have to pay.."+brewery.getRentValue()+" to "+brewery.getOwner().getName());
66
67                     // Withdraw the rentPrice from the player
68                     currentPlayer.getAccount().withdraw(rentPrice);
69
70                     // Deposit the rentPrice to the owner
71                     brewery.getOwner().getAccount().deposit(rentPrice);
72
73                     // Send updates to the GUI
74                     guiController.updatePlayerBalance(brewery.getOwner().getGuiId(), brewery.getOwner().getAccount().getBalance());
75                     guiController.updatePlayerBalance(currentPlayer.getGuiId(), currentPlayer.getAccount().getBalance());
```

```
76                  }else {
77                      guiController.writeMessage("You cannot afford the rent, you have to pawn or sell something");
78                      //  Initialize the SalesController
79                      SalesController salesController = new SalesController(currentPlayer);
80
81                      // Run cannotAfford method
82                      boolean response = salesController.cannotAfford(rentPrice);
83                      if(response) {
84                          guiController.writeMessage("You can now pay the rent of "+rentPrice);
85
86                          // Withdraw rentValue from the player
87                          currentPlayer.getAccount().withdraw(rentPrice);
88
89                          // Deposit rentValue to the owner
90                          brewery.getOwner().getAccount().deposit(rentPrice);
91
92                          // Sends updates to GUIController
93                          guiController.updatePlayerBalance(brewery.getOwner().getGuiId(), brewery.getOwner().getAccount().getBalance());
94                          guiController.updatePlayerBalance(currentPlayer.getGuiId(), currentPlayer.getAccount().getBalance());
95                      }
96                  }
97              }
98          }
99      }
100 }
101
```

Buy Controller Source Code

```
1
2  package core;
3
4  import java.awt.Color;
5
6  /**
7   *
8   * @author Mathias Thejsen s175192 && Simon Hansen s175191
9   *
10  */
11 public class BuyController {
12     private Field field;
13     private Player currentPlayer;
14     private GUIController guiController = GUIController.getInstance();
15
16
17     public BuyController(Player currentPlayer) {
18         this.currentPlayer = currentPlayer;
19     }
20     /**
21      * Constructor for the buy logic
22      * @param id
23      */
24     public BuyController(Player currentPlayer, Field field) {
25         this.currentPlayer = currentPlayer;
26         this.field = field;
27     }
28
29     /**
30      * Logic for buying a property
31      * @param currentPlayer
32      * @return
33      */
34     protected void buyLogic() {
35         if(field instanceof Street) {
36             this.streetBuyLogic();
37         }else if(field instanceof Brewery) {
38             this.breweryBuyLogic();
39         }else if(field instanceof Shipping) {
40             this.shippingBuyLogic();
41         }
42     }
43
44
45     /**
46      * Logic for buying a street
47      * @param currentPlayer
48      * @param field
49      */
50
51     private void streetBuyLogic() {
52         Street street = (Street) field;
53         currentPlayer.getAccount().withdraw(street.getBuyValue()); // Withdraw money form player based on the property base value
54         street.setOwner(currentPlayer); // Set the owner
55         guiController.setOwner(currentPlayer.getGuiId(), currentPlayer.getEndPosition());
56         guiController.updatePlayerBalance(currentPlayer.getGuiId(), currentPlayer.getAccount().getBalance());
57     }
58
59     /**
60      * Logic for buying a shipping field
61      * @param currentPlayer
62      * @return
63      */
64     private void shippingBuyLogic() {
65         Shipping shipping = (Shipping) field;
66         Field[] fields = guiController.getFieldController().getFieldArr();
67         int counter = 0; // How many the player owns
68         currentPlayer.getAccount().withdraw(shipping.getBuyValue()); // Withdraw the basevalue from the player
69         guiController.updatePlayerBalance(currentPlayer.getGuiId(), currentPlayer.getAccount().getBalance());
70         guiController.setOwner(currentPlayer.getGuiId(), field.getId());
71         shipping.setOwner(currentPlayer); // Set the owner
72         for (int i = 0; i < fields.length; i++) { // First loop and find out how many we own
73             if(fields[i] instanceof Shipping) {
74                 Shipping shipping2 = (Shipping) fields[i];
75                 if (shipping2.getOwner() == currentPlayer) {
```

```
76                  counter++; // Update how many we own
77                  for (int j = 0; j < fields.length; j++) {
78                      shipping2.setRentValue(getShippingValue(counter)); // Set the value on them all
79                  }
80              }
81          }
82      }
83  }
84
85
86  /**
87   * Calculates the price of the shipping fields
88   * @param i How many we own
89   * @return The value
90   */
91  private int getShippingValue(int i) {
92      switch (i) {
93      case 1:
94          return 500;
95      case 2:
96          return 1000;
97      case 3:
98          return 2000;
99      case 4:
100         return 4000;
101     default:
102         return 0;
103     }
104 }
105
106 /**
107  * The logic for buying a brewery field
108  * @param currentPlayer
109  * @return
110  */
111 private void breweryBuyLogic() {
112     Brewery brewery = (Brewery) field;
113     Field[] fields = guiController.getFieldController().getFieldArr();
114     int counter = 0;
115     currentPlayer.getAccount().withdraw(brewery.getBuyValue());
116     guiController.updatePlayerBalance(currentPlayer.getGuild(), currentPlayer.getAccount().getBalance());
117     guiController.setOwner(currentPlayer.getGuild(), field.getId());
118     brewery.setOwner(currentPlayer);
119     for (int i = 0; i < fields.length; i++) { // First loop and find out how many we own
120         if(fields[i] instanceof Brewery) {
121             Brewery brewery2 = (Brewery) fields[i];
122             if (brewery2.getOwner() == currentPlayer) {
123                 counter++; // Update how many we own
124                 for (int j = 0; j < fields.length; j++) {
125                     brewery.setRentValue(getBreweryValue(counter)); // Set the value on them all
126                 }
127             }
128         }
129     }
130 }
131
132
133 /**
134  * Calculates the price of the brewery fields
135  * @param i How many we own
136  * @return The value
137  */
138 private int getBreweryValue(int i) {
139     switch (i) {
140     case 1:
141         return 100;
142     case 2:
143         return 200;
144     default:
145         return 0;
146     }
147 }
148 /**
149  * Converts a color to a string
150  * @param color
151  * @return a string color
152  */
```

```
153     private String convertColor(Color color) {
154         if(color == Color.yellow) {
155             return "yellow";
156         }else if(color == Color.blue) {
157             return "blue";
158         }else if(color == Color.orange) {
159             return "orange";
160         }else if(color == Color.white) {
161             return "white";
162         }else if(color == Color.green) {
163             return "green";
164         }else if(color == Color.magenta) {
165             return "purple";
166         }else if(color == Color.red) {
167             return "red";
168         }else if(color == Color.gray) {
169             return "gray";
170         }else {
171             return "Error";
172         }
173     }
174 
175     /**
176      * Buying a house logic
177      * @param currentPlayer
178      * @return
179      */
180     protected void houseBuyLogic(Field field) {
181         if(field instanceof Street) { // We are only dealing with fields of the type normal, so only check for those
182             Street normal = (Street) field; // Instantiate a new Normal object casting fieldsid normal
183             currentPlayer.getAccount().withdraw(normal.getBuildPrice());
184             normal.setHouseCounter(normal.getHouseCounter() + 1);
185             normal.setRentValue(calcHousePrice(normal.getHouseCounter()));
186             guiController.updatePlayerBalance(currentPlayer.getGuild(), currentPlayer.getAccount().getBalance());
187             guiController.setHouse(normal.getId(),normal.getHouseCounter());
188             guiController.setOwner(currentPlayer.getGuild(), normal.getId());
189         }
190     }
191 
192 
193     /**
194      * Calculates the new price of rent when a new house has been build
195      * @param houses
196      * @return the landing price
197      */
198     private int calcHousePrice(int houses) {
199         if(field instanceof Street) {
200             Street normal = (Street) field;
201             switch (houses) {
202             case 1:
203                 return normal.getHousePrices()[1];
204             case 2:
205                 return normal.getHousePrices()[2];
206             case 3:
207                 return normal.getHousePrices()[3];
208             case 4:
209                 return normal.getHousePrices()[4];
210             case 5:
211                 return normal.getHousePrices()[5];
212             default:
213                 return field.getId();
214             }
215         }
216         return field.getId();
217     }
218 
219 
220 
221 
222 
223 
224     /**
225      * Logic for unpawning a property
226      */
227     protected void unPawnProperty() { // This is only called when the player can afford to unpawn a field
228 
229         // Gets the field array from the GUIController
```

```
230         Field [] fieldArr = guiController.getFieldController().getFieldArr();
231
232         // Initializes the property variable
233         Property property = null;
234
235         // Calls and hold the String array of all the fields that the player owns and can afford to unpawn
236         String [] temp = guiController.getFieldController().pawnedFields(currentPlayer);
237
238         // Makes the String array to hold all the fields and a return option
239         String [] pawnedProperties = new String[temp.length+1];
240
241         // Parses all the fields from the temp array into the pawnedProperties array
242         for(int i = 0; i<temp.length;i++) {
243             pawnedProperties[i] = temp[i];
244         }
245
246         // Adds the return option to the array
247         pawnedProperties[pawnedProperties.length-1] = PropertiesIO.getTranslation("returnbutton");
248
249         // Prompts the user to choose a field to unpawn or return
250         String result = guiController.requestPlayerChoiceDropdown(PropertiesIO.getTranslation("unpawnpick"), pawnedProperties);
251
252         // If the choice is not the return option
253         if(!result.equals(PropertiesIO.getTranslation("returnbutton"))) {
254
255             // Loops over the field array to get the field information
256             for(int i = 0; i < guiController.getFieldController().getFieldArr().length; i++) {
257
258                 // Checks if the field name matches the choice of the player
259                 if(fieldArr[i].getName().equals(result)) {
260                     property = (Property) fieldArr[i];
261                 }
262             }
263
264             // Sets the property to not pawned
265             property.setPawned(false);
266
267             // Saves the price of unpawning which is the pawnValue + 10% of the pawnValue
268             int value = property.getPawnValue()+(int)((property.getPawnValue()*0.10));
269
270             // Withdraws from the Players account
271             currentPlayer.getAccount().withdraw(value);
272
273             // Sends an update to the GUIController
274             guiController.updatePlayerBalance(currentPlayer.getGuiId(), currentPlayer.getAccount().getBalance());
275         }
276     }
277
278
279     /**
280      * Returns a string array containing all the fields a player can build on
281      * @param street
282      * @return
283      */
284     protected String[] listOfFieldsYouCanBuildOn(Street[] street) {
285         String[] properties = new String[40];
286         String[] propertiesSorted = null;
287         int index = 0;
288
289         //Loop over all streets in input array
290         for (int counter = 0; counter < street.length; counter++) {
291             //Get color from street in string format
292             String color = convertColor(street[counter].getColour());
293             int amountOfHouses = street[counter].getHouseCounter();
294             boolean mayBuild = true;
295             for (Street streetTemp : street) {
296                 String color2 = convertColor(streetTemp.getColour());
297                 if(color == color2) {
298                     if(amountOfHouses > streetTemp.getHouseCounter()) {
299                         mayBuild = false;
300                     }
301                 }
302             }
303
304             if(mayBuild)
305                 properties[index++] = street[counter].getName();
306         }
```

```
307
308         //add to return array.
309         propertiesSorted = new String[index];
310         for (int i = 0; i < propertiesSorted.length; i++) {
311             propertiesSorted[i] = properties[i];
312         }
313         return propertiesSorted;
314     }
315 }
316
```

Chance Source Code

```
1
2  package core;
3  /**
4   * @author Mathias Thejsen s175192 && Simon Hansen s175191
5   */
6  public class Chance extends Field {
7      public Chance(int id, String name, String description) {
8          super(id, name, description);
9      }
10 }
11
```

Dice Source Code

```
1
2  package core;
3  import java.util.concurrent.ThreadLocalRandom;
4  /**
5   * Class created in CDIO 1
6   * @author Simon Fritz (s175191) and Mathias Thejsen (s175192)
7   */
8
9  public class Dice {
10     private int faceValue;
11
12     /**
13      * Constructor for dice
14      */
15     public Dice() {
16     }
17
18     /**
19      * Handles the roll
20      * @return, returns a random integer value between 1 and 6.
21      */
22     public int roll() {
23         faceValue = ThreadLocalRandom.current().nextInt(1, 6 + 1);
24         return faceValue;
25     }
26
27     /**
28      * @return returns the value of the dice
29      */
30     public int getFaceValue() {
31         return faceValue;
32     }
33
34     /**
35      * @param The dice value to set
36      */
37     public void setFaceValue(int faceValue) {
38         this.faceValue = faceValue;
39     }
40 }
41
```

DiceCup Source Code

```
1
2  package core;
3  /**
4   *
5   * @author Whole group
6   *
7   */
8  public class DiceCup {
9      private Dice diceArr[];
10     /**
11      * The dicecup contains a dice arr, which is filled with refrences to dices
12      * @param amoutOfDice
13      */
14     public DiceCup(int amoutOfDice) {
15         diceArr = new Dice[amoutOfDice];
16         for (int i = 0; i < diceArr.length; i++) {
17             diceArr[i] = new Dice();
18         }
19     }
20     /**
21      * Rolls all the dices contained within the dice arr
22      */
23     public void roll() {
24         for (int i = 0; i < diceArr.length; i++) {
25             diceArr[i].roll();
26         }
27     }
28     /**
29      * Returns true if the dices are pairs
30      * @return
31      */
32     public boolean isPair() {
33         int tempValue = 0;
34         for (Dice dice : diceArr) {
35             if(tempValue == 0)
36                 tempValue = dice.getFaceValue();
37             if(tempValue != dice.getFaceValue())
38                 return false;
39         }
40         return true;
41     }
42
43     public int getTotalFaceValue() {
44         int total = 0;
45         for (Dice dice : diceArr)
46             total += dice.getFaceValue();
47         return total;
48     }
49
50     public Dice[] getDiceArr() {
51         return diceArr;
52     }
53 }
54
```

Field Source Code

```
1
2  package core;
3  /**
4   * Basic field
5   * @author Mathias Thejsen s175192 && Simon Hansen s175191
6   *
7   */
8  public abstract class Field {
9      private String name;
10     private int id;
11     private String description;
12     /**
13      * Constructor for field
14      * @param id
15      * @param name
16      * @param description
17      */
18     public Field(int id, String name, String description) {
19         this.id = id;
20         this.name = name;
21         this.description = description;
22     }
23
24     public String getName() {
25         return name;
26     }
27
28     public void setName(String n) {
29         name = n;
30     }
31
32     public int getId() {
33         return id;
34     }
35
36     public void setId(int i) {
37         id = i;
38     }
39
40     public String getDescription() {
41         return description;
42     }
43 }
44
```

FieldController Source Code

```
1
2  package core;
3  import java.awt.Color;
4  /**
5   *
6   * @author Mathias && Simon && Nicolai
7   *
8   */
9  public class FieldController {
10     private Field fieldArr [];
11     public FieldController () {
12         initFields ();
13     }
14     /**
15      * Fills out the fields array
16      */
17     private void initFields () {
18         fieldArr = new Field [40]; // We create an array of the lenght 40 of the type field
19         for (int i = 0; i < fieldArr.length ; i++) { // Loop through all our fields
20             String description = "";
21             String [] descriptionSplit ;
22             if(i == 0) {
23                 // START FIELD "id, name, description"
24                 description = PropertiesIO.getTranslation("startdescription"); // As we have to put something inbetween the sentence we split it
25                 descriptionSplit = description.split(",");
26                 description = (descriptionSplit [0]+PropertiesIO.getTranslation("startpassedvalue")+ descriptionSplit [1]);
27                 fieldArr[i] = new Start(i,
28                         PropertiesIO.getTranslation("field "+(i+1)),
29                         description);
30             } else if(i == 5 || i == 15 || i == 25 || i == 35) {
31                 // SHIPPING FIELDS "id, name, owner, basevalue, pawnvalue, description)
32                 description = PropertiesIO.getTranslation("shippingdesc")+PropertiesIO.getTranslation("field "+(i+1)+"pant");
33                 description = description.replace("[1]", ""+PropertiesIO.getTranslation("field "+(i+1)+"leje"));
34                 description = description.replace("[2]", ""+PropertiesIO.getTranslation("field "+(i+1)+"rederi2"));
35                 description = description.replace("[3]", ""+PropertiesIO.getTranslation("field "+(i+1)+"rederi3"));
36                 description = description.replace("[4]", ""+PropertiesIO.getTranslation("field "+(i+1)+"rederi4"));
37                 fieldArr[i] = new Shipping(i,
38                         PropertiesIO.getTranslation("field "+(i+1)),
39                         description,
40                         null,
41                         Integer.parseInt(PropertiesIO.getTranslation("field "+(i+1)+"value")),
42                         Integer.parseInt(PropertiesIO.getTranslation("field "+(i+1)+"pant")),
43                         Integer.parseInt(PropertiesIO.getTranslation("field "+(i+1)+"leje")));
44             } else if(i == 12 || i == 28) {
45                 // BREWERY FIELDS "id, name, owner, basevalue, pawnvalue, description"
46                 description = PropertiesIO.getTranslation("brewerydesc")+PropertiesIO.getTranslation("field "+(i+1)+"value");
47                 fieldArr[i] = new Brewery(i,
48                         PropertiesIO.getTranslation("field "+(i+1)),
49                         description,
50                         null,
51                         Integer.parseInt(PropertiesIO.getTranslation("field "+(i+1)+"value")),
52                         Integer.parseInt(PropertiesIO.getTranslation("field "+(i+1)+"pant")),
53                         2000);
54             } else if(i == 2 || i == 7 || i == 17 || i == 22 || i == 33 || i == 36) {
55                 // CHANCE CARD FIELDS "id, name, description"
56                 description = PropertiesIO.getTranslation("chancedesc");
57                 fieldArr[i] = new Chance(i,
58                         PropertiesIO.getTranslation("field "+(i+1)),
59                         description);
60             } else if(i == 4 || i == 38) {
61                 //TAX FIELDS "id, name, taxvalue, description"
62                 fieldArr[i] = new Tax(i,
63                         PropertiesIO.getTranslation("field "+(i+1)),
64                         "");
65             } else if(i == 10 || i == 30) {
66                 //PRISON FIELD "id, name, description"
67                 if (i == 10) description = PropertiesIO.getTranslation("prisondesc1");
68                 else description = PropertiesIO.getTranslation("prisondesc2");
69                 fieldArr[i] = new Prison(i,
70                         PropertiesIO.getTranslation("field "+(i+1)),
71                         description);
72             } else if(i == 20) {
73                 //Parking FIELD "id, name, description"
74                 description = PropertiesIO.getTranslation("parkingdesc");
75                 fieldArr[i] = new Parking(i,
```

```
76                      PropertiesIO.getTranslation("field "+(i+1)),
77                      description);
78              } else {
79                  // Switch tot ranslate colour from PropertiesIO to gui colour.
80                  Color color;
81                  switch(PropertiesIO.getTranslation("field "+(i+1)+"color")) {
82                  default : color = Color.black; break;
83                  case "yellow" : color = Color.yellow; break;
84                  case "blue" : color = Color.blue; break;
85                  case "pink" : color = Color.orange; break;
86                  case "white" : color = Color.white; break;
87                  case "green" : color = Color.green; break;
88                  case "purple" : color = Color.magenta; break;
89                  case "red" : color = Color.red; break;
90                  case "grey" : color = Color.gray; break;
91                  }
92                  // Get house prices from PropertiesIO and save to array
93                  int housePrices[] = { // Get all the house prices
94                          Integer.parseInt(PropertiesIO.getTranslation("field "+(i+1)+"leje")),
95                          Integer.parseInt(PropertiesIO.getTranslation("field "+(i+1)+"hus1")),
96                          Integer.parseInt(PropertiesIO.getTranslation("field "+(i+1)+"hus2")),
97                          Integer.parseInt(PropertiesIO.getTranslation("field "+(i+1)+"hus3")),
98                          Integer.parseInt(PropertiesIO.getTranslation("field "+(i+1)+"hus4")),
99                          Integer.parseInt(PropertiesIO.getTranslation("field "+(i+1)+"hotel"))
100                 };
101                 // Split description of field, made up of 10 parts
102                  descriptionSplit = new String[10];
103                 for (int j = 0; j < descriptionSplit.length; j++) {
104                     descriptionSplit[j] = PropertiesIO.getTranslation("propertydesc"+(j+1)); // Loop through the PropertiesIO and fill out array
105                 }
106                 description = ( // put everything in the right order
107                         descriptionSplit[0]+PropertiesIO.getTranslation("field "+(i+1)+"leje")+"\n"+
108                         descriptionSplit[1]+PropertiesIO.getTranslation("field "+(i+1)+"hus1")+"\n"+
109                         descriptionSplit[2]+PropertiesIO.getTranslation("field "+(i+1)+"hus2")+"\n"+
110                         descriptionSplit[3]+PropertiesIO.getTranslation("field "+(i+1)+"hus3")+"\n"+
111                         descriptionSplit[4]+PropertiesIO.getTranslation("field "+(i+1)+"hus4")+"\n"+
112                         descriptionSplit[5]+PropertiesIO.getTranslation("field "+(i+1)+"hotel")+"\n"+
113                         descriptionSplit[6]+"\n"+
114                         descriptionSplit[7]+PropertiesIO.getTranslation("field "+(i+1)+"build")+"\n"+
115                         descriptionSplit[8]+PropertiesIO.getTranslation("field "+(i+1)+"build")+"\n"+
116                         descriptionSplit[9]+PropertiesIO.getTranslation("field "+(i+1)+"pant")+"\n"
117                         );
118                 // STREET(int id, String name, String description, Player owner, int buyValue, int pawnValue, int rentValue, int[] housePrices, int buildPrice, Color colour)
119                 fieldArr[i] = new Street(i,
120                         PropertiesIO.getTranslation("field "+(i+1)),
121                         description,
122                         null,
123                         Integer.parseInt(PropertiesIO.getTranslation("field "+(i+1)+"value")),
124                         Integer.parseInt(PropertiesIO.getTranslation("field "+(i+1)+"pant")),
125                         housePrices[0],
126                         housePrices,
127                         Integer.parseInt(PropertiesIO.getTranslation("field "+(i+1)+"build")),
128                         color);
129             }
130         }
131     }
132     public Field[] getFieldArr() {
133         return fieldArr;
134     }
135     public void setFieldArr(Field[] fieldArr) {
136         this.fieldArr = fieldArr;
137     }
138     /**
139      * Generates a street array with all the fields that houses can be built on
140      * @param currentPlayer
141      * @return
142      */
143     public Street[] allFieldsToBuildOn(Player currentPlayer) {
144         Street streetArr[] = new Street[40];
145         int val = 0;
146         boolean exists = false;
147         Color colour;
148         // We loop over all our fields
149         for (int i = 0; i < fieldArr.length; i++) {
150             //Bool to stay
151             boolean canBeBuildOn = true;
152             // We find the fields which are an instance of Normal
```

```
153             if(fieldArr[i] instanceof Street) {
154                 // Casting
155                 Street street = (Street) fieldArr[i];
156                 // We check if the current field is owned by the player and if there are too many houses already
157                 if(street.getOwner() == currentPlayer && street.getHouseCounter() <5) {
158                     // If the player can afford the buildprice
159                     if(currentPlayer.getAccount().canAfford(street.getBuildPrice())) {
160                         // Grab the colour
161                         colour = street.getColour();
162                         // Start an inner loop
163                         for(int j = 0; j < fieldArr.length; j++) {
164                             // Once again only want to look at the fields which are of the type normal
165                             if(fieldArr[j] instanceof Street) {
166                                 // casting
167                                 Street street2 = (Street) fieldArr[j];
168                                 // Making sure that the fields of the same colour and the same owner, if not the same owner we return false
169                                 if(street2.getColour() == colour && street2.getOwner() != currentPlayer) {
170                                     canBeBuildOn = false;
171                                     break;
172                                 }
173                             }
174                         }
175                         // If property can be build on
176                         if(canBeBuildOn == true) {
177                             streetArr[val++] = street;
178                         }
179                     }
180                 }
181             }
182         }
183         /**
184          * We create a new array which only contains the streets we are allowed to build on with the right size
185          */
186         Street[] sortedStreetArr = new Street[val];
187         for(int i = 0; i < sortedStreetArr.length; i++) {
188             sortedStreetArr[i] = streetArr[i];
189         }
190         return sortedStreetArr;
191     }
192     /**
193      * Generate an array of strings which contains that field title of the fields that the player calling owns
194      * @param currentPlayer
195      * @return
196      */
197     public String[] FieldsOwned(Player currentPlayer) {
198         // Makes a temporary String array with the length of all properties
199         String[] temp = new String[28];
200         // Counter
201         int counter = 0;
202         // Loops over all the fields
203         for(int i = 0; i < fieldArr.length; i++) {
204             // Checks if the field is an instance of the Property class
205             if(fieldArr[i] instanceof Property) {
206                 // Casts the field to Property
207                 Property property = (Property) fieldArr[i];
208                 // Checks if the field is owned by the Player
209                 if(property.getOwner() == currentPlayer) {
210                     // Adds the field to the temp array
211                     temp[counter] = fieldArr[i].getName();
212                     // Adds one to the counter
213                     counter++;
214                 }
215             }
216         }
217         // Makes new array with the length of counter
218         String[] propertyOwned = new String[counter];
219         // Loops over the temp array and adds the filled spots to the propertyOwned array
220         for(int j = 0; j < counter; j++) {
221             propertyOwned[j] = temp[j];
222         }
223         return propertyOwned;
224     }
225     public String[] propertiesToPawn(Player currentPlayer) {
226         // Make a temporary String array with the length of all properties
227         String[] temp = new String[28];
228         // Counter
229         int counter = 0;
```

```
230         // Loops over all fields
231         for(int i = 0; i < fieldArr.length; i++) {
232             // Checks if the field is an instance of the Street class
233             if(fieldArr[i] instanceof Street) {
234                 // Casts the field to street
235                 Street street = (Street) fieldArr[i];
236                 // Checks if the Street is owned by the player, checks if no houses are built on it and if it is already pawned
237                 if(street.getOwner() == currentPlayer && street.getHouseCounter() == 0 && !street.isPawned()) {
238                     // Adds the field to the temp array
239                     temp[counter] = fieldArr[i].getName();
240                     // Adds one to the counter
241                     counter++;
242                 }
243                 // Checks if the field is an instance of the Shipping Class
244                 if(fieldArr[i] instanceof Shipping) {
245                     // Adds the field to the temp array
246                     temp[counter] = fieldArr[i].getName();
247                     // Adds one to the counter
248                     counter++;
249                 }
250                 // Checks if the field is an instance of the Brewery Class
251                 if(fieldArr[i] instanceof Brewery) {
252                     // Adds the field to the temp array
253                     temp[counter] = fieldArr[i].getName();
254                     // Adds one to the counter
255                     counter++;
256                 }
257             }
258         }
259         // Makes a new array with the length of counter
260         String[] propertyOwned = new String[counter];
261         // Loops over the temp array and adds the filled spots to the propertyOwned array
262         for(int j = 0; j < counter; j++) {
263             propertyOwned[j] = temp[j];
264         }
265         return propertyOwned;
266     }
267     public String[] streetsWithHouses(Player currentPlayer) {
268         // Makes a temporary array with the length of all Street fields
269         String[] temp = new String[22];
270         // Counter
271         int counter = 0;
272         // Loops over all fields
273         for(int i = 0; i < fieldArr.length; i++) {
274             // Checks if the field is an instance of the Street class
275             if(fieldArr[i] instanceof Street) {
276                 // Casts the field to street
277                 Street street = (Street) fieldArr[i];
278                 // Checks if the field is owned by the Player and there are not too many houses
279                 if(street.getOwner() == currentPlayer && street.getHouseCounter() <= 5 && street.getHouseCounter() > 0) {
280                     // Add the field to the temp array
281                     temp[counter] = fieldArr[i].getName();
282                     // Adds one to the counter
283                     counter++;
284                 }
285             }
286         }
287         // Males a new array with the length of counter
288         String[] propertyOwned = new String[counter];
289         // Loops over the temp array and adds the filled spots to the propertyOwned array
290         for(int j = 0; j < counter; j++) {
291             propertyOwned[j] = temp[j];
292         }
293         return propertyOwned;
294     }
295     /**
296      * returns a list of pawned fields
297      * @param currentPlayer
298      * @return
299      */
300     protected String[] pawnedFields(Player currentPlayer) {
301         String[] temp = new String[28];
302         int counter = 0;
303         // Looping over all fields
304         for(int i = 0; i < fieldArr.length; i++) {
305             if(fieldArr[i] instanceof Property) { // Check if it's a property
306                 Property property = (Property) fieldArr[i]; // casting
```

```
307                 if(property.getOwner() == currentPlayer) { // Find the fields the player owns
308                     if(property.isPawned()) { // check if it's pawned
309                         if(currentPlayer.getAccount().canAfford((int) (property.getPawnValue()+(property.getPawnValue()*0.10)))) { // check if he can afford to unpawn
310                             temp[counter] = fieldArr[i].getName();// add it
311                             counter++; // add it
312                         }
313                     }
314                 }
315             }
316         }
317         String[] propertyOwned = new String[counter];
318         for(int j = 0; j < counter; j++) {
319             propertyOwned[j] = temp[j];
320         }
321         return propertyOwned;
322     }
323     /**
324      * Used to check if the player should have the possibility to trade with someone
325      * @param currentPlayer
326      * @return true if possible else false
327      */
328     protected boolean tradePossible(Player currentPlayer) {
329         for (int i = 0; i < fieldArr.length; i++) { // loop over all fields
330             if(fieldArr[i] instanceof Property) { // Find those who are properties
331                 Property property = (Property) fieldArr[i]; // Initialize and cast
332                 if(property.getOwner() != currentPlayer && property.getOwner() != null) { // Make sure it's owned by someone that is not currentPlayer
333                     return true; // Return true, no need to look any futher
334                 }
335             }
336         }
337         return false; // Not possible to trade so no need to show option
338     }
339 }
340
```

GameController Source Code

```
1
2  package core;
3  public class GameController {
4      private GUIController guiController;
5      private PlayerController playerController;
6      private GameLogic gameLogic;
7      public static void main(String Args[]) {
8          GameController gameController = new GameController();
9          gameController.prepareGame();
10         gameController.playRound();
11     }
12     public GameController() {
13         guiController = GUIController.getInstance();
14         playerController = new PlayerController();
15         gameLogic = new GameLogic();
16     }
17     public void prepareGame() {
18         String playerNames[] = guiController.setupBoard();
19         playerController.initPlayers(playerNames.length);
20         for(int i = 0; i < playerNames.length; i++) {
21             Player player = new Player(playerNames[i], i);
22             playerController.getPlayers()[i] = player;
23         }
24     }
25
26     public void playRound() {
27         boolean gameIsLive = true;
28
29         while(gameIsLive) {
30             for (Player player : playerController.getPlayers()) {
31                 boolean switchPlayer = true;
32                 do {
33                     //Check if game is still live
34                     int amountBankrupt = 0;
35                     for (Player otherPlayer : playerController.getPlayers())
36                         if(otherPlayer.isBanktrupt())
37                             amountBankrupt++;
38
39                     if(amountBankrupt >= playerController.getPlayers().length-1) {
40                         gameIsLive = false;
41                         break;
42                     }
43
44                     if(!player.isBanktrupt()) {
45                         switchPlayer = gameLogic.showOptions(playerController, player);
46                     }
47                 } while(!switchPlayer);
48                 player.setRolled(false);
49                 player.setPairs(0);
50             }
51         }
52         for (int i = 0; i < playerController.getPlayers().length; i++) {
53             if(!playerController.getPlayers()[i].isBanktrupt()) {
54                 guiController.writeMessage(playerController.getPlayers()[i].getName()+PropertiesIO.getTranslation("winnerstr"));
55             }
56         }
57         System.exit(0);
58     }
59 }
60
```

GameLogic Source Code

```
1
2  package core;
3  import core.ChanceCardLogic.ChanceCardController;
4  /**
5   *
6   * @author Mathias Thejsen s175192 && Simon Hansen s175191
7   *
8   */
9  public class GameLogic {
10     private DiceCup diceCup;
11     private GUIController guiController = GUIController.getInstance();
12     private Field[] fields;
13     private PrisonController prisonController;
14     private FieldController fieldController;
15     private ChanceCardController cardController;
16     private PlayerController playerController;
17     private TradeController tradeController;
18     /**
19      * Constructor for gamelogic
20      */
21     public GameLogic() {
22         cardController = new ChanceCardController();
23         diceCup = new DiceCup(2);
24         tradeController = new TradeController();
25     }
26     /**
27      * Generates a list of options for the player
28      * @param playerController
29      * @param currentPlayer
30      * @return true if their turn is over
31      */
32     public boolean showOptions(PlayerController playerController, Player currentPlayer) {
33         fieldController = guiController.getFieldController();
34         fields = fieldController.getFieldArr();
35         BuyController buyController = new BuyController(currentPlayer);
36         SalesController salesController = new SalesController(currentPlayer);
37         this.playerController = playerController;
38         int counter = 0;
39         String choicesArr[] = new String[5];
40         if(currentPlayer.isPrison()) { // Check if player is prisoned
41             prisonController.prison(currentPlayer);
42             if(!currentPlayer.isPrison()) {
43                 updatePos(currentPlayer);
44                 passedStart(currentPlayer);
45                 checkIfExtraRound(currentPlayer);
46                 resolveField(currentPlayer, diceCup);
47                 return false;
48             }
49         } else {
50             // Add end turn if player has rolled
51             if(currentPlayer.isRolled()) {
52                 choicesArr[counter++] = "Afslut tur";
53             }
54             // Add buy house or hotel
55             Street[] buildablestreets = fieldController.allFieldsToBuildOn(currentPlayer);
56             if(buildablestreets.length > 0) {
57                 choicesArr[counter++] = "Køb huse/hoteller";
58             }
59             // Add Roll dice if not already rolled
60             if(!currentPlayer.isRolled()) {
61                 choicesArr[counter++] = "Rul terningerne";
62             }
63
64             if(fieldController.propertiesToPawn(currentPlayer).length > 0) { // Check if the player has anything to pawn
65                 choicesArr[counter++] = "Pantsæt grund";
66             }
67             if(fieldController.pawnedFields(currentPlayer).length > 0) { // Check if the player has any pawned property
68                 choicesArr[counter++] = "Fjern pantsætning";
69             }
70             if(fieldController.tradePossible(currentPlayer)) { // Check if there are any propreties to trade with
71                 choicesArr[counter++] = "Byt grunde";
72             }
73             // Move to new array
74             String choices[] = new String[counter];
75             for(int i = 0; i < counter; i++)
```

```
76                  choices[i] = choicesArr[i];
77              do {
78                  switch(guiController.requestPlayerChoiceButtons(PropertiesIO.getTranslation("turn1") + currentPlayer.getName() + PropertiesIO.getTranslation("turn2"), choices))
79                          {
80                  case "Rul terningerne" : {
81                      diceCup.roll();  //  roll  dices
82                      guiController.showDice(diceCup); // show the dices on the gui
83                      updatePos(currentPlayer); // update pos
84                      passedStart(currentPlayer); // check if we passed start if so give money
85                      resolveField(currentPlayer, diceCup); // call some logic on the field
86                      if(currentPlayer.isMoved()) { // check if the player was moved with a chance card
87                          resolveField(currentPlayer, diceCup);
88                          currentPlayer.setMoved(false);
89                      }
90                      if(currentPlayer.isBanktrupt() || currentPlayer.isPrison() || checkIfExtraRound(currentPlayer)) { // Check if player is bankrupt, prisoned or if he deserves
91                              an extra round
92                          return true;
93                      }
94                      return false;
95                  }
96                  case "Køb huse/hoteller" : {
97                      buildablestreets = fieldController.allFieldsToBuildOn(currentPlayer); // Grab list of buildable streets
98                      buyController = new BuyController(currentPlayer, fields[currentPlayer.getEndPosition()]); // initialize buycontroller
99                      String houseList = guiController.requestPlayerChoiceDropdown(PropertiesIO.getTranslation("chooseproperty"),
100                             buyController.listOfFieldsYouCanBuildOn(buildablestreets)); // The property a player chose
101                     for (int j = 0; j < fields.length; j++) {
102                         if(fields[j].getName() == houseList) { // Find the specificed house
103                             buyController.houseBuyLogic(fields[j]); // call housebuylogic which adds the house
104                             break;
105                         }
106                     }
107                     guiController.writeMessage(PropertiesIO.getTranslation("boughthouseon")+houseList);
108                     return false;
109                 }
110                 case "Pantsæt grund":{
111                     salesController.pawnProperty(); // Call pawn property
112                     return false;
113                 }
114                 case "Afslut tur":{
115                     return true; // end turn
116                 }
117                 case "Fjern pantsætning":{
118                     buyController.unPawnProperty();
119                     return false;
120                 }
121                 case "Byt grunde": {
122                     tradeController.startTrade(currentPlayer, fieldController, playerController.getPlayers());
123                     return false;
124                 }
125                 }
126             } while (diceCup.isPair());
127         }
128         return true;
129     }
130     /**
131      * Called by the gamecontroller, this switch checks what kind of field we land on, and then calls a respective logic switch
132      * @param currentPlayer
133      * @return A message to the gamecontroller
134      */
135     public void resolveField(Player currentPlayer, DiceCup diceCup) {
136         int id = currentPlayer.getEndPosition();
137         if (fields[id] instanceof Street) {
138             StreetController streetController = new StreetController(currentPlayer, fields[id], playerController.getPlayers());
139             streetController.logic();
140         } else if (fields[id] instanceof Brewery) {
141             BreweryController breweryController = new BreweryController(currentPlayer, diceCup.getTotalFaceValue(), fields, playerController.getPlayers());
142             breweryController.logic();
143         } else if (fields[id] instanceof Chance) {
144             cardController.getCard(currentPlayer, fields, playerController.getPlayers());
145         } else if (fields[id] instanceof Shipping) {
146             ShippingController shippingController = new ShippingController(currentPlayer, diceCup.getTotalFaceValue(), fields, playerController.getPlayers());
147             shippingController.logic();
148         } else if (fields[id] instanceof Prison) {
149             prisonController = new PrisonController(currentPlayer, diceCup, cardController);
150             prisonController.logic();
151         } else if (fields[id] instanceof Parking) {
152             guiController.writeMessage(PropertiesIO.getTranslation("landonparking"));
```

```
153         } else if (fields[id] instanceof Tax) {
154             TaxController taxController = new TaxController(currentPlayer, fields);
155             taxController.taxLogic();
156         }
157     }
158     private void updatePos(Player currentPlayer) {
159         //save start position and set new end position
160         currentPlayer.setStartPosition(currentPlayer.getEndPosition());
161         if((currentPlayer.getEndPosition() + diceCup.getTotalFaceValue()) > 39) {
162             currentPlayer.setEndPosition(currentPlayer.getEndPosition() + diceCup.getTotalFaceValue() - 40); // Player passed start
163         }else {
164             currentPlayer.setEndPosition(currentPlayer.getEndPosition() + diceCup.getTotalFaceValue()); // Player did not pass start
165         }
166         guiController.updatePlayerPosition(currentPlayer.getGuiId(), currentPlayer.getEndPosition(), currentPlayer.getStartPosition()); // Update position on gui
167     }
168     private boolean checkIfExtraRound(Player currentPlayer) {
169         //check if player will get another turn because of pairs
170         if(!diceCup.isPair()) {
171             currentPlayer.setRolled(true);
172         }else {
173             currentPlayer.setPairs(currentPlayer.getPairs()+1);
174             if(currentPlayer.getPairs() >= 3) { // Check if he rolled 3 pairs in a row
175                 currentPlayer.setPairs(0);
176                 guiController.writeMessage(PropertiesIO.getTranslation("rolled3inarow"));
177                 PrisonController prisonController = new PrisonController(currentPlayer, diceCup, cardController);
178                 prisonController.jailPlayer(currentPlayer); // jail the player
179                 return true;
180             }
181         }
182         return false;
183     }
184     /**
185      * Updates balance if we passed start, called everytime we roll dices
186      * @param currentPlayer
187      */
188     public void passedStart(Player currentPlayer) {
189         boolean passed = false;
190         if(!currentPlayer.isStartRound()) {
191             if(((diceCup.getTotalFaceValue() + currentPlayer.getStartPosition()) > 40) || currentPlayer.getStartPosition() == 0) { // Check if he passed start
192                 currentPlayer.getAccount().deposit(4000);
193                 guiController.updatePlayerBalance(currentPlayer.getGuiId(), currentPlayer.getAccount().getBalance());
194                 passed = true;
195             }
196         } else {
197             currentPlayer.setStartRound(false);
198             passed = false;
199         }
200         //update balance if start is passed
201         if(passed)
202             guiController.updatePlayerBalance(currentPlayer.getGuiId(), currentPlayer.getAccount().getBalance());
203     }
204     public PrisonController getPrisonLogic() {
205         return prisonController;
206     }
207 }
208
```

GUIController Source Code

```
1
2  package core;
3  import gui_fields.*;
4  import gui_main.GUI;
5  import java.awt.*;
6  import java.util.concurrent.TimeUnit;
7  /**
8   * @author Christian S. Andersen
9   */
10 public class GUIController {
11     private GUI_Field[] fields_GUI;
12     private FieldController fieldController;
13     private GUI gui;
14     private GUI_Player[] players_GUI;
15     private static final GUIController guiController = new GUIController();
16     private GUIController() {}
17     public static GUIController getInstance() {
18         return guiController;
19     }
20     /**
21      * Prepares the board. Lets the players choose amount of players and player names.
22      * @return String array of the players names.
23      */
24     public String[] setupBoard() {
25         fieldController = new FieldController();
26         Field fields[] = fieldController.getFieldArr();
27         fields_GUI = new GUI_Field[fields.length];
28         createFields(fields);
29
30         int amountOfPlayers = guiController.requestNumberOfPlayers();
31         String playerNames[] = new String[amountOfPlayers];
32
33         for (int i = 0; i < amountOfPlayers; i++) {
34             String name = guiController.requestStringInput(PropertiesIO.getTranslation("playernames")+(i+1));
35             if (name.equals(""))
36                 name = "player"+(i+1);
37             addPlayer(i, 30000, name);
38             playerNames[i] = name;
39         }
40
41         return playerNames;
42     }
43     /**
44      * Creates the GUI_Field objects necessary information to create the board.
45      * @param fields the array of fields objects, with the necessary information.
46      */
47     public void createFields(Field[] fields) {
48         for (int i = 0; i < fields_GUI.length; i++) { // Iterates over every field, and creates the GUI_Field objects, corresponding to their Field part.
49             if(fields[i] instanceof Start) {
50                 Start start = (Start) fields[i];
51                 fields_GUI[i] = new GUI_Start("Start", "Kr. 4000", start.getDescription(), Color.RED, Color.BLACK);
52             }else if(fields[i] instanceof Street) {
53                 Street normal = (Street) fields[i];
54                 fields_GUI[i] = new GUI_Street(normal.getName(), "kr. "+normal.getBuyValue(), normal.getDescription(), ""+normal.getBuyValue(), normal.getColour(),
55                         Color.BLACK);
56             }else if(fields[i] instanceof Brewery) {
57                 Brewery brewery = (Brewery) fields[i];
58                 fields_GUI[i] = new GUI_Brewery("default", brewery.getName(), "kr. "+brewery.getBuyValue(), brewery.getDescription(), ""+brewery.getBuyValue(), Color.WHITE,
59                         Color.BLACK);
60             }else if(fields[i] instanceof Shipping) {
61                 Shipping shipping = (Shipping) fields[i];
62                 fields_GUI[i] = new GUI_Shipping("default", shipping.getName(), "kr. "+shipping.getBuyValue(), shipping.getDescription(), ""+shipping.getBuyValue(),
63                         Color.WHITE, Color.BLACK);
64             }else if(fields[i] instanceof Chance) {
65                 Chance chance = (Chance) fields[i];
66                 fields_GUI[i] = new GUI_Chance("?", PropertiesIO.getTranslation("takeachancecard"), PropertiesIO.getTranslation("takeachancecard"), Color.WHITE, Color.BLACK);
67             }else if(fields[i] instanceof Prison) {
68                 Prison prison = (Prison) fields[i];
69                 fields_GUI[i] = new GUI_Jail("default", prison.getName(), prison.getName(), prison.getDescription(), Color.WHITE, Color.BLACK);
70             }else if(fields[i] instanceof Parking) {
71                 Parking parking = (Parking) fields[i];
72                 fields_GUI[i] = new GUI_Refuge("default", parking.getName(), "", parking.getDescription(), Color.WHITE, Color.BLACK);
73             }else if(fields[i] instanceof Tax) {
74                 Tax tax = (Tax) fields[i];
75                 fields_GUI[i] = new GUI_Tax(tax.getName(), "kr. 2000", tax.getDescription(), Color.WHITE, Color.BLACK);
```

```
76              }else {
77                  fields_GUI[i] = new GUI_Empty();
78              }
79          }
80          gui = new GUI(fields_GUI);
81      }
82      /**
83       * Used when adding a new player to the board.
84       * @param id the id associated with the Player object.
85       * @param startValue value the player starts with.
86       * @param name name of the player.
87       */
88      public void addPlayer(int id, int startValue, String name) {
89          players_GUI[id] = new GUI_Player(name, startValue, createCar(id));
90          fields_GUI[0].setCar(players_GUI[id], true);
91          gui.addPlayer(players_GUI[id]);
92      }
93      /**
94       * Removes a player from the board, and changes his name.
95       * @param id id associated with the player.
96       * @param pos position of the playere.
97       */
98      public void removePlayer(int id, int pos) {
99          fields_GUI[pos].setCar(players_GUI[id], false);
100         players_GUI[id].setName("TODO Fallit");
101     }
102     /**
103      * Creates the GUI vehicle for a player.
104      * @param id the id associated with the player.
105      * @return object type GUI_Car.
106      */
107     private GUI_Car createCar(int id) {
108         Color color = getVehicleColor(id);
109         return new GUI_Car(color, color, GUI_Car.Type.CAR, GUI_Car.Pattern.FILL);
110     }
111     /**
112      * Displays a message in the middle of the board, with black text on white background.
113      * @param message the message you would like displayed on the card.
114      */
115     public void displayChanceCard(String message) {
116         gui.displayChanceCard(message);
117     }
118     /**
119      * Updates the player position on the GUI.
120      * @param id the id associated with the player.
121      * @param newPos the position to move the player to.
122      * @param oldPos the position the player is currently on.
123      */
124     public void updatePlayerPosition(int id, int newPos, int oldPos) {
125         if (oldPos > newPos) { //Handles if the player moves past start, where his old position will be bigger than the new one.
126             for (int i = oldPos; i < fields_GUI.length-1; i++) {
127                 fields_GUI[i].setCar(players_GUI[id], false);
128                 fields_GUI[i+1].setCar(players_GUI[id], true);
129                 try {
130                     //Waits between every step, to simulate player movement.
131                     TimeUnit.MILLISECONDS.sleep(100);
132                 } catch (InterruptedException e) {
133                     e.printStackTrace();
134                 }
135             }
136             //When the player is at the end of the board, move his position to the start position.
137             fields_GUI[fields_GUI.length-1].setCar(players_GUI[id], false);
138             fields_GUI[0].setCar(players_GUI[id], true);
139             oldPos = 0;
140             try {
141                 TimeUnit.MILLISECONDS.sleep(100);
142             } catch (InterruptedException e) {
143                 e.printStackTrace();
144             }
145         }
146         // Iterates over every field between the players current position and his destination. Moves the player to the every field between them to animate the player walking.
147         for (int i = oldPos; i < newPos;i++) {
148             fields_GUI[i].setCar(players_GUI[id], false);
149             fields_GUI[i+1].setCar(players_GUI[id], true);
150             try {
151                 TimeUnit.MILLISECONDS.sleep(100);
152             } catch (InterruptedException e) {
```

```
153                 e.printStackTrace();
154             }
155         }
156     }
157     /**
158      * Updates the subtext of a field.
159      * @param text the text to change to.
160      * @param field_id the number of the field to change.
161      */
162     public void updateFieldSubtext(String text, int field_id) {
163         fields_GUI[field_id].setSubText(text);
164     }
165     /**
166      * Changes the balance of the player onn the GUI.
167      * @param id the id associated with the player.
168      * @param value the value to update to.
169      */
170     public void updatePlayerBalance(int id, int value) {
171         players_GUI[id].setBalance(value);
172     }
173     /**
174      * Moves the players vehicle from the GUI to the jail field.
175      * @param id the id associated with the player.
176      * @param playerPos position of the player.
177      * @param jailPos position of the jail.
178      */
179     public void jailPlayer(int id, int playerPos, int jailPos) {
180         fields_GUI[playerPos].setCar(players_GUI[id], false);
181         fields_GUI[jailPos].setCar(players_GUI[id], true);
182     }
183     /**
184      * Instantly moves the player from his current field to a destination.
185      * @param id the id of the player to move.
186      * @param currentPos the position the player is standing on.
187      * @param destination the destination for the player.
188      */
189     public void teleport(int id, int currentPos, int destination) {
190         fields_GUI[currentPos].setCar(players_GUI[id], false);
191         fields_GUI[destination].setCar(players_GUI[id], true);
192     }
193     /**
194      * Changes the border of an ownable field, to the color of the player.
195      * @param player_id the location of the specific GUI_Player object in the array of GUI_Player objects.
196      * @param field_id the location of the specific GUI_Field object in the array of GUI_Field objects.
197      */
198     public void setOwner(int player_id, int field_id) {
199         ((GUI_Ownable)fields_GUI[field_id]).setBorder(players_GUI[player_id].getPrimaryColor());
200     }
201     /**
202      * Visually changes the amount of houses on the board.
203      * @param field_id the id associated with the field.
204      * @param amount amount of houses to set. 5 houses is a hotel.
205      */
206     public void setHouse(int field_id, int amount) {
207         if (amount >= 5) {
208             ((GUI_Street) fields_GUI[field_id]).setHotel(true);
209         } else {
210                 ((GUI_Street) fields_GUI[field_id]).setHouses(amount);
211         }
212     }
213     /**
214      * Lets the user input an integer the system can interprit. A pre defined message will be displayed alongside the text field.
215      * @return int
216      */
217     public int requestNumberOfPlayers() {
218         int input = gui.getUserInteger("      "+PropertiesIO.getTranslation("amountofplayers"), 3, 6);
219         while(input < 3 || input > 6) {
220             input = gui.getUserInteger("      "+PropertiesIO.getTranslation("amountofplayers"), 3, 6);
221         }
222         players_GUI = new GUI_Player[input];
223         return input;
224     }
225     /**
226      * Takes input from the player between two integers.
227      * @param message the String to write on the GUI.
228      * @param min minimum amount the player can input.
229      * @param max maximum amount the player can input.
```

```
230      * @return returns the input from the user.
231      */
232     public int requestIntegerInput(String message, int min, int max) {
233         return gui.getUserInteger(message, min, max);
234     }
235     /**
236      * Takes integer input from the player.
237      * @param message message to write.
238      * @return int value.
239      */
240     public int requestIntegerInput(String message) {
241         return gui.getUserInteger(message);
242     }
243     /**
244      * Lets the user input a String message the system can interpret.
245      * @param message — Message the GUI will display along with the text field.
246      * @return String
247      */
248     public String requestStringInput(String message) {
249         return gui.getUserString("  "+message);
250     }
251     /**
252      * Creates a bottom and dropdown menu on the board, with the message provided, and the options provided. Returns a String for the users choice.
253      * @param message the message which will be written to the players.
254      * @param options the list of options for the dropdown menu.
255      * @return a String with the players choice.
256      */
257     public String requestPlayerChoiceDropdown(String message, String[] options) {
258         return gui.getUserSelection("  "+message, options);
259     }
260     /**
261      * Creates 'x' amount of buttons, along with a message, and returns the selected button.
262      * @param message the message to display to the player.
263      * @param options the various buttons to display.
264      * @return string
265      */
266     public String requestPlayerChoiceButtons(String message, String... options) {
267         return gui.getUserButtonPressed("  "+message, options);
268     }
269     /**
270      * Writes a message on the GUI.
271      * @param message the String which will be written.
272      */
273     public void writeMessage(String message) {
274         gui.showMessage("  "+message);
275     }
276     /**
277      * Changes the dice on the board.
278      */
279     public void showDice(DiceCup diceCup) {
280         gui.setDice(diceCup.getDiceArr()[0].getFaceValue(), diceCup.getDiceArr()[1].getFaceValue());
281     }
282     /**
283      * Returns the color depending on what number the player is.
284      * @return Color
285      */
286     public Color getVehicleColor(int id) {
287         switch (id) {
288         case 0:
289             return Color.BLACK;
290         case 1:
291             return Color.RED.darker();
292         case 2:
293             return Color.GREEN.darker();
294         case 3:
295             return Color.YELLOW.darker();
296         case 4:
297             return Color.MAGENTA.darker();
298         case 5:
299             return Color.CYAN.darker();
300         default:
301             return Color.BLUE.brighter();
302         }
303     }
304     /**
305      * Used when starting new game. Will reset all players position to start and set all their money to the start money.
306      */
```

```
307     public void resetUIPlayers(int startBalance) {
308         for (GUI_Field field : fields_GUI) {
309             field.removeAllCars();
310         }
311         for (GUI_Player player : players_GUI) {
312             fields_GUI[0].setCar(player, true);
313             player.setBalance(startBalance);
314         }
315     }
316     public FieldController getFieldController() {
317         return fieldController;
318     }
319 }
320
```

SpecialProperty Source Code

```
1
2  package core;
3  /**
4   *
5   * @author Mathias Thejsen s175192 && Simon Hansen s175191
6   *
7   */
8  public class Parking extends SpecialProperty {
9      public Parking(int id, String name, String description) {
10         super(id, name, description);
11     }
12 }
13
```

Player Source Code

```
1
2  package core;
3  /**
4   * @author Nicolai Kammersgård <s143780@student.dtu.dk>
5   */
6  public class Player{
7      private String name;
8      private Account account;
9      private boolean isPrisoned, bankrupt,startRound, rolled, isMoved;
10     private int prisonCard;
11     private int guiId;
12     private int startPosition;
13     private int endPosition;
14     private int prisontries;
15     private int pairs;
16
17     /**
18      * Constructor
19      */
20     public Player() {
21     }
22
23     /**
24      * Constructor with params
25      * @param name The players name
26      */
27     public Player(String name, int idGui) {
28         this.name = name;
29         this.account = new Account();
30         this.isPrisoned = false;
31         this.isMoved = false;
32         this.bankrupt = false;
33         this.rolled = false;
34         this.startRound = true;
35         this.startPosition = 0;
36         this.prisonCard = 0;
37         this.guiId = idGui;
38         this.prisontries = 0;
39         this.endPosition = this.startPosition;
40         this.pairs = 0;
41     }
42     public int getPrisonCard() {
43         return prisonCard;
44     }
45     public void setPrisonCard(int prisonCard) {
46         this.prisonCard = prisonCard;
47     }
48     public void addPrisonCard() {
49         this.prisonCard++;
50     }
51     public void removePrisonCard() {
52         this.prisonCard--;
53     }
54     public int getGuiId() {
55         return guiId;
56     }
57     public void setGuiId(int id_GUI) {
58         this.guiId = id_GUI;
59     }
60     public int getEndPosition() {
61         return endPosition;
62     }
63     public void setEndPosition(int endPosition) {
64         this.endPosition = endPosition;
65     }
66     public Account getAccount() {
67         return this.account;
68     }
69
70     public void setAccount(Account account) {
71         this.account = account;
72     }
73     public String getName() {
74         return this.name;
75     }
```

```
76      public void setName(String name) {
77          this.name = name;
78      }
79      public int getStartPosition() {
80          return this.startPosition;
81      }
82      public void setStartPosition(int fieldNumber) {
83          this.startPosition = fieldNumber;
84      }
85
86      public void setPrison(boolean prison) {
87          this.isPrisoned = prison;
88      }
89      public boolean isPrison() {
90          return this.isPrisoned;
91      }
92
93      public void setPrisontries(int prisontries) {
94          this.prisontries = prisontries;
95      }
96
97      public int getPrisontries() {
98          return prisontries;
99      }
100
101     public boolean isBanktrupt() {
102         return this.bankrupt;
103     }
104
105     public void setBanktrupt(boolean bankrupt) {
106         this.bankrupt = bankrupt;
107     }
108
109     public boolean isStartRound() {
110         return startRound;
111     }
112     public void setStartRound(boolean startRound) {
113         this.startRound = startRound;
114     }
115
116     public boolean isRolled() {
117         return rolled;
118     }
119
120     public int getPairs() {
121         return pairs;
122     }
123
124     public void setPairs(int pairs) {
125         this.pairs = pairs;
126     }
127
128     public void setRolled(boolean rolled) {
129         this.rolled = rolled;
130     }
131
132     public boolean isMoved() {
133         return isMoved;
134     }
135
136     public void setMoved(boolean isMoved) {
137         this.isMoved = isMoved;
138     }
139 }
140
```

PlayerController Source Code

```
1
2  package core;
3  /**
4   *
5   * @author Mathias Thejsen
6   *
7   */
8  public class PlayerController {
9      private Player playerArr[];
10
11     public PlayerController() {
12
13     }
14     public void initPlayers(int amountOfPlayers) {
15         playerArr = new Player[amountOfPlayers];
16     }
17     public Player[] getPlayers() {
18         return playerArr;
19     }
20     public void setPlayers(Player[] playerArr) {
21         this.playerArr = playerArr;
22     }
23 }
24
```

Prison Source Code

```
1
2  /**
3   *
4   * @author Mathias Thejsen s175192 && Simon Hansen s175191
5   *
6   */
7  class Prison extends SpecialProperty {
8      Prison(int id, String name, String description) {
9          super(id, name, description);
10     }
11 }
12
```

PrisonController Source Code

```
1
2  package core;
3  import core.ChanceCardLogic.ChanceCardController;
4  /**
5   *
6   * @author Christian Stahl Andersen s164150
7   *
8   */
9  public class PrisonController {
10     private Player currentPlayer;
11     private DiceCup diceCup;
12     private GUIController guiController = GUIController.getInstance();
13     private ChanceCardController chanceCardController;
14     /**
15      * Constructor for the prisonController.
16      * @param currentPlayer the player who is to be handled at the moment.
17      * @param diceCup the diceCup for the project.
18      * @param chanceCardController the controller for the chance cards, to add support for prison cards.
19      */
20     PrisonController(Player currentPlayer, DiceCup diceCup, ChanceCardController chanceCardController) {
21         this.currentPlayer = currentPlayer;
22         this.diceCup = diceCup;
23         this.chanceCardController = chanceCardController;
24     }
25     /**
26      * Determines if the player is visiting prison, in prison or about to be sent to prison.
27      */
28     public void logic() {
29         int pos = currentPlayer.getEndPosition();
30         if (pos == 10 && !currentPlayer.isPrison()) {
31             guiController.writeMessage(PropertiesIO.getTranslation("prisonvisit"));
32         } else {
33             prison(currentPlayer);
34         }
35     }
36     /**
37      * basic logic for the player, gets an input, and selects the method to use.
38      * @param currentPlayer the player who will choose an action.
39      */
40     public void prison(Player currentPlayer) {
41         String choice = getPlayerChoice();
42         if (choice.equals(PropertiesIO.getTranslation("throwdice"))) {
43             rollJailDice(currentPlayer);
44         } else if (choice.equals(PropertiesIO.getTranslation("prisonchoice1"))) {
45             payFine(currentPlayer);
46         } else if (choice.equals(PropertiesIO.getTranslation("prisonchoice2"))) {
47             usePrisonCard(currentPlayer);
48         } else if (choice.equals(PropertiesIO.getTranslation("prisongoto"))) {
49             jailPlayer(currentPlayer);
50         }
51     }
52     /**
53      * Jails the player.
54      * @param currentPlayer the player to be moved and jailed.
55      */
56     public void jailPlayer(Player currentPlayer) {
57         guiController.jailPlayer(currentPlayer.getGuiId(), currentPlayer.getEndPosition(), 10);
58         currentPlayer.setEndPosition(10);
59         currentPlayer.setPrison(true);
60     }
61     /**
62      * Pays 1000 to release the player from prison.
63      * @param currentPlayer the playere to be released
64      */
65     public void payFine(Player currentPlayer) {
66         currentPlayer.getAccount().withdraw(1000);
67         guiController.updatePlayerBalance(currentPlayer.getGuiId(), currentPlayer.getAccount().getBalance());
68         diceCup.roll();
69         if (currentPlayer.isPrison()) {
70             releasePrison(currentPlayer);
```

```
71          }
72      }
73      /**
74       * Uses a prison card for the player.
75       * @param currentPlayer the player to be released from prison.
76       */
77      public void usePrisonCard(Player currentPlayer) {
78          chanceCardController.putPrisonCardInDeck();
79          currentPlayer.setPrisonCard(currentPlayer.getPrisonCard()-1);
80          guiController.writeMessage(PropertiesIO.getTranslation("prisonusecard")+currentPlayer.getPrisonCard());
81          if (currentPlayer.isPrison()) {
82              releasePrison(currentPlayer);
83          }
84      }
85      /**
86       * If the player is in prison, he will be released properly.
87       * @param currentPlayer the active player.
88       */
89      private void releasePrison(Player currentPlayer) {
90          currentPlayer.setPrison(false);
91          guiController.writeMessage(PropertiesIO.getTranslation("prisonrollmessage"));
92          diceCup.roll();
93          guiController.showDice(diceCup);
94      }
95      /**
96       * Rolls the dice in the jail and releases him, if he gets a pair.
97       * @param currentPlayer the player to roll for.
98       */
99      public void rollJailDice(Player currentPlayer) {
100         diceCup.roll();
101         guiController.showDice(diceCup);
102         currentPlayer.setPrisontries(currentPlayer.getPrisontries()+1);
103         System.out.println(currentPlayer.getPrisontries());
104         if (diceCup.isPair()) {
105             guiController.writeMessage(PropertiesIO.getTranslation("prisonpairroll"));
106             currentPlayer.setPrison(false);
107             currentPlayer.setPrisontries(0);
108         } else {
109             guiController.writeMessage(PropertiesIO.getTranslation("prisonroll")+(3-currentPlayer.getPrisontries()));
110         }
111     }
112     /**
113      * Creates a menu with valid choices for the player.
114      * @return String with the text the player choose.
115      */
116     public String getPlayerChoice() {
117         SalesController salesController = new SalesController(currentPlayer);
118         String choices = "";
119         String[] choiceArr;
120         if (currentPlayer.isPrison()) {
121             if (currentPlayer.getPrisontries() < 3) {
122                 choices = choices + "," + PropertiesIO.getTranslation("throwdice");
123             }
124             if (currentPlayer.getAccount().canAfford(1000) && currentPlayer.getPrisontries() == 0 || currentPlayer.getPrisontries() == 3) {
125                 choices = choices + "," + PropertiesIO.getTranslation("prisonchoice1");
126             }
127             if (currentPlayer.getPrisonCard() > 0) {
128                 choices = choices + "," + PropertiesIO.getTranslation("prisonchoice2");
129             }
130             while (!currentPlayer.getAccount().canAfford(1000) && currentPlayer.getPrisontries() == 3) {
131                 salesController.cannotAfford(1000);
132             }
133             if (choices.startsWith(",")) choices = choices.substring(1);
134             choiceArr = choices.split(",");
135             return guiController.requestPlayerChoiceButtons(PropertiesIO.getTranslation("prisongenericmessage"), choiceArr);
136         } else {
137             if (currentPlayer.getAccount().canAfford(1000) || currentPlayer.getPrisontries() == 3) {
138                 choices = choices + "," + PropertiesIO.getTranslation("prisonchoice1");
139             }
140             if (currentPlayer.getPrisonCard() > 0) {
141                 choices = choices + "," + PropertiesIO.getTranslation("prisonchoice2");
142             }
143             choices = choices + "," + PropertiesIO.getTranslation("prisongoto");
144             if (choices.startsWith(",")) choices = choices.substring(1);
145             choiceArr = choices.split(",");
146             return guiController.requestPlayerChoiceButtons(PropertiesIO.getTranslation("prisonlandgotomessage"), choiceArr);
147         }
```

```
148     }
149 }
150
```

PropertiesIO Source Code

```
1
2  package core;
3  import java.io.FileNotFoundException;
4  import java.io.IOException;
5  import java.util.Properties;
6  /**
7   * @author Magnus Stjernborg Koch — s175189 && Mathias
8   * Translator is being handled using a properties file.
9   */
10 public class PropertiesIO {
11     private static Properties translations = new Properties();
12     private static PropertiesIO propertiesIO;
13     /**
14      * Translator constructor
15      * @throws IOException may throw exception if file is not found
16      */
17     private PropertiesIO() {
18         try {
19             translations.load(ClassLoader.getSystemResourceAsStream("config.properties"));
20         } catch (FileNotFoundException e) {
21             System.out.println("File not found: "+e);
22         } catch (IOException e) {
23             System.out.println("IO error: "+e);
24         }
25     }
26     /**
27      * Gets the translation of the given key
28      * @param toTranslate the key to look for
29      * @return The attached string to the key
30      */
31     public static String getTranslation (String toTranslate) {
32         if (propertiesIO == null){
33             propertiesIO = new PropertiesIO();
34             }
35         return translations.getProperty(toTranslate);
36     }
37 }
38
```

Property Source Code

```
1
2  package core;
3  /**
4   *
5   * @author Mathias Thejsen s175192 && Simon Hansen s175191
6   *
7   */
8  public abstract class Property extends Field {
9      private Player owner;
10     private int buyValue;
11     private int rentValue;
12     private int pawnValue;
13     private boolean isPawned;
14     /**
15      * Constructor for the property field
16      * @param id
17      * @param name
18      * @param owner
19      * @param baseValue
20      * @param pawnValue
21      * @param description
22      */
23     public Property(int id, String name, String description, Player owner, int buyValue, int pawnValue, int rentValue) {
24         super(id, name, description);
25         this.buyValue = buyValue;
26         this.pawnValue = pawnValue;
27         this.owner = owner;
28         this.isPawned = false;
29         this.rentValue = rentValue;
30     }
31     public Player getOwner() {
32         return owner;
33     }
34     public void setOwner(Player owner) {
35         this.owner = owner;
36     }
37     public int getBuyValue() {
38         return buyValue;
39     }
40     public void setBuyValue(int buyValue) {
41         this.buyValue = buyValue;
42     }
43     public int getRentValue() {
44         return rentValue;
45     }
46     public void setRentValue(int rentValue) {
47         this.rentValue = rentValue;
48     }
49     public int getPawnValue() {
50         return pawnValue;
51     }
52     public void setPawnValue(int pawnValue) {
53         this.pawnValue = pawnValue;
54     }
55     public boolean isPawned() {
56         return isPawned;
57     }
58     public void setPawned(boolean isPawned) {
59         this.isPawned = isPawned;
60     }
61 }
62
```

SalesController Source Code

```
1
2  package core;
3  /**
4   *
5   * @author Mathias Thejsen s175192 && Simon Hansen s175191
6   *
7   */
8  public class SalesController {
9      private Player currentPlayer;
10     private GUIController guiController = GUIController.getInstance();
11     private FieldController fieldcontroller = guiController.getFieldController();
12     /**
13      * Constructor for salescontroller
14      * @param currentPlayer
15      */
16     public SalesController(Player currentPlayer) {
17         this.currentPlayer = currentPlayer;
18     }
19      /**
20      * Logic when player cannot afford something
21      * @param value The value we can't afford
22      */
23     public boolean cannotAfford(int value) {
24         boolean housesToSell = true;
25         boolean propertyToPawn = true;
26         Field[] fields = fieldcontroller.getFieldArr();
27
28         // While loop until the player can afford the rent/pay
29         while(currentPlayer.getAccount().getBalance() < value && !currentPlayer.isBanktrupt()) {
30             String[] streets = fieldcontroller.streetsWithHouses(currentPlayer); // Get an array of streets with houses
31             String[] properties = fieldcontroller.propertiesToPawn(currentPlayer); // Get an array of properties we can pawn
32
33              // Check if the Player has anything to sell, if not they go bankrupt.
34             if(streets.length == 0 && properties.length == 0 && currentPlayer.getAccount().getBalance() < value) {
35                 guiController.writeMessage(PropertiesIO.getTranslation("banktrupt"));
36                 currentPlayer.setBanktrupt(true);
37                 return false;
38             }
39
40             // Make array to hold options
41             String[] temp = new String[2];
42             int counter = 0;
43             // Check if the streets array is empty (There will always be one due to the Return Option)
44             if(streets.length > 0) {
45                 temp[counter] = PropertiesIO.getTranslation("sellhouses");
46                 counter++;
47             }
48             // Check if the properties array is empty (There will always be one due to the Return Option)
49             if(properties.length > 0) {
50                 temp[counter] = "PropertiesIO.getTranslation(\"pawnproperty\")";
51                 counter++;
52             }
53             String[] options = new String[counter];
54             for(int i = 0; i < options.length; i++) {
55                 options[i] = temp[i];
56             }
57             // Ask user for choice
58             String result = guiController.requestPlayerChoiceDropdown(PropertiesIO.getTranslation("salescontrollerrequest"), options);
59
60             // Runs the sellHouse method if chosen
61             if(result.equals(PropertiesIO.getTranslation("sellhouses"))) {
62                 housesToSell = sellHouse();
63             }
64
65             // Runs the pawnProperty method if chosen
66             if(result.equals(PropertiesIO.getTranslation("pawnproperty"))) {
67                 propertyToPawn = pawnProperty();
68             }
69
70             // If the player has no houses to sell, properties to pawn and still cannot afford, the player is declared bankrupt and false is returned
71             if(!housesToSell && !propertyToPawn && currentPlayer.getAccount().getBalance() < value) {
72                 currentPlayer.setBanktrupt(true);
73                 return false;
74             }
75         }
```

```
76          return true;
77      }
78
79      /**
80       * Logic for selling a house
81       * @return a boolean that indicates if we have any houses to sell or not
82       */
83      public boolean sellHouse() {
84          // Method returns a String array with all fields that the current player owns that haves houses built on them
85          String[] temp = fieldcontroller.streetsWithHouses(currentPlayer); // A temporary array
86          String[] streets = new String[temp.length+1]; // A street array that is one size bigger than the previous, because we need the return option
87          for(int i = 0; i<temp.length;i++) {
88              streets[i] = temp[i]; // Fill the streets array
89          }
90          streets[streets.length-1] = PropertiesIO.getTranslation("returnbutton");
91          // Returns false if there is no fields with houses
92
93
94          // Prompts the user for a choice
95          String response = guiController.requestPlayerChoiceDropdown(PropertiesIO.getTranslation("sellhouse"), streets);
96          if(response.equals(PropertiesIO.getTranslation("returnbutton"))) { // We can go a menu back
97              return true;
98          }
99          for(int i = 0; i < fieldcontroller.getFieldArr().length; i++) {
100             // Finds the field that the player have chosen
101             if(response.equals( fieldcontroller.getFieldArr()[i].getName())) {
102
103                 // hold the field as a street class
104                 Street street = (Street) fieldcontroller.getFieldArr()[i];
105
106                 // We get the current player and deposit the house price back into the players account
107                 currentPlayer.getAccount().deposit(street.getBuildPrice());
108
109                 // Removes a house from the Street field
110                 street.setHouseCounter(street.getHouseCounter()-1);
111
112                 // Send all the updates to GUIController
113                 guiController.updatePlayerBalance(currentPlayer.getGuiId(), currentPlayer.getAccount().getBalance());
114                 guiController.setHouse(street.getId(), ((street.getHouseCounter())-1));
115                 guiController.writeMessage(PropertiesIO.getTranslation("soldhouse")+street.getBuildPrice());
116             }
117         }
118         return true;
119     }
120     /**
121      * Logic for pawning a property
122      * @return Returns true if we have anything at all to pawn, and false if not
123      */
124     public boolean pawnProperty() {
125
126         // Holds a String array of all the properties owned by the current Player
127         String[] temp = fieldcontroller.propertiesToPawn(currentPlayer);
128         String[] properties = new String[temp.length+1];
129         for(int i = 0; i<temp.length;i++) {
130             properties[i] = temp[i];
131         }
132         properties[properties.length-1] = PropertiesIO.getTranslation("returnbutton");
133         // Returns false if the player owns no properties to pawn
134
135         // Prompts the user for a choice
136         String response = guiController.requestPlayerChoiceDropdown(PropertiesIO.getTranslation("pawnpick"), properties);
137         if(response.equals(PropertiesIO.getTranslation("returnbutton"))) {
138             return true;
139         }
140         for(int i = 0; i < fieldcontroller.getFieldArr().length; i++) {
141
142             // Finds the field that the player have chosen
143             if(response.equals( fieldcontroller.getFieldArr()[i].getName())) {
144
145                 // Hold the field as a property class
146                 Property property = (Property) fieldcontroller.getFieldArr()[i];
147
148                 // We set the pawn bool to true
149                 property.setPawned(true);
150
151                 // We get the pawn value and deposit it into the owners account
152                 currentPlayer.getAccount().deposit(property.getPawnValue());
```

```
153
154                 // Send all updates to GUIController
155                 guiController.updatePlayerBalance(currentPlayer.getGuiId(), currentPlayer.getAccount().getBalance());
156                 guiController.writeMessage(PropertiesIO.getTranslation("pawnstr") + " "+property.getName()+" for "+property.getPawnValue());
157             }
158         }
159         return true;
160     }
161     public Player getCurrentPlayer() {
162         return currentPlayer;
163     }
164 }
165
```

Shipping Source Code

```
1
2  package core;
3  /**
4   *
5   * @author Mathias Thejsen s175192 && Simon Hansen s175191
6   *
7   */
8  public class Shipping extends Property {
9      /**
10      * Constructor for shipping field
11      * @param id The id of the field
12      * @param name The name of the field
13      * @param owner The owner of the field
14      * @param baseValue Base value(board value)
15      * @param pawnValue Pawn value
16      * @param description Description of the field
17      */
18     public Shipping(int id, String name, String description, Player owner, int buyValue, int pawnValue, int rentValue) {
19         super(id, name, description, owner, buyValue, pawnValue, rentValue);
20     }
21 }
22
```

ShippingController Source Code

```
1
2  package core;
3  /**
4   * Logic for landing on a shipping field
5   * @author Mathias Thejsen s175192 && Simon Hansen s175191
6   *
7   */
8  public class ShippingController {
9      private GUIController guiController = GUIController.getInstance();
10     private Shipping shipping;
11     private Player currentPlayer;
12     private Player[] players;
13     private AuctionController auctionController;
14     /**
15      * Constructor for shipping logic
16      * @param currentPlayer The player that landed on the field
17      * @param totalFaceValue The total dice value of both dices
18      * @param fields The fields array
19      */
20     public ShippingController(Player currentPlayer, int totalFaceValue, Field[] fields, Player[] players) {
21         this.currentPlayer = currentPlayer;
22         this.shipping = (Shipping) fields[currentPlayer.getEndPosition()];
23         this.players = players;
24         auctionController = new AuctionController();
25     }
26     /**
27      * Returns the outcome of landin gon the field
28      * @param currentPlayer The current player
29      * @return
30      */
31     public void logic() {
32
33         // We check if the field is owned
34         if(shipping.getOwner() == null) {
35
36             // Check if the Player can afford it
37             if(currentPlayer.getAccount().canAfford(shipping.getBuyValue())) {
38
39                 // Prompt user for choice
40                 String[] choices = {PropertiesIO.getTranslation("yesbutton"), PropertiesIO.getTranslation("nobutton")};
41                 String result = guiController.requestPlayerChoiceButtons(PropertiesIO.getTranslation("streetlandonbuy")+" "+shipping.getName() + " for " +
42                         shipping.getBuyValue(), choices);
43
44                 // Check if the choice is Yes
45                 if(result.equals(PropertiesIO.getTranslation("yesbutton"))) {
46
47                     // Initialize BuyController
48                     BuyController buyController = new BuyController(currentPlayer, shipping);
49
50                     // Run buyLogic method
51                     buyController.buyLogic();
52                 }
53                 else { // If the player does not wish to buy the field
54                     auctionController.startAuction(currentPlayer, shipping, players);
55                 }
56             }
57             else {
58                 // If the player cannot afford the field
59                 auctionController.startAuction(currentPlayer, shipping, players);
60             }
61         }
62         else{
63
64             // Field owned by the player landing on it
65             if(shipping.getOwner() == currentPlayer || shipping.isPawned()) {
66                 // Nothing should happen
67             }
68
69             // If it is owned by another player
70             else {
71
72                 // We get the field rent/price
73                 int rentPrice = shipping.getBuyValue();
74
75                 // Check if the player can afford the rent
```

```
76                  if(currentPlayer.getAccount().canAfford(rentPrice)) {
77                      // Send update to player
78                      guiController.writeMessage(PropertiesIO.getTranslation("streetlanddon")+" "+shipping.getName()+" " + PropertiesIO.getTranslation("streetlanddon2") + "
79                              "+shipping.getRentValue()+" til "+shipping.getOwner().getName());
80
81                      // Withdraw rentPrice from player
82                      currentPlayer.getAccount().withdraw(rentPrice);
83
84                      // Deposit rentPrice to owner
85                      shipping.getOwner().getAccount().deposit(rentPrice);
86
87                      // Send updates to GUIController
88                      guiController.updatePlayerBalance(shipping.getOwner().getGuiId(), shipping.getOwner().getAccount().getBalance());
89                      guiController.updatePlayerBalance(currentPlayer.getGuiId(), currentPlayer.getAccount().getBalance());
90
91                  }
92                  else { // We can't afford landing here
93                      guiController.writeMessage(PropertiesIO.getTranslation("streetcantafford"));
94                      // Initialize the SalesController
95                      SalesController salesController = new SalesController(currentPlayer);
96
97                      // Run cannotAfford method
98                      boolean response = salesController.cannotAfford(rentPrice);
99                      if(response) {
100                         guiController.writeMessage(PropertiesIO.getTranslation("streetcanpayrent")+rentPrice);
101
102                         // Withdraw rentValue from the player
103                         currentPlayer.getAccount().withdraw(rentPrice);
104
105                         // Deposit rentValue to the owner
106                         shipping.getOwner().getAccount().deposit(rentPrice);
107
108                         // Sends updates to GUIController
109                         guiController.updatePlayerBalance(shipping.getOwner().getGuiId(), shipping.getOwner().getAccount().getBalance());
110                         guiController.updatePlayerBalance(currentPlayer.getGuiId(), currentPlayer.getAccount().getBalance());
111                     }
112                 }
113             }
114         }
115     }
116 }
117
```

SpecialProperty Source Code

```
1
2  package core;
3  /**
4   *
5   * @author Mathias s175192 og Simon
6   *
7   */
8  public class SpecialProperty extends Field {
9      public SpecialProperty(int id, String name, String description) {
10         super(id, name, description);
11     }
12 }
13
```

Start Source Code

```
1
2  package core;
3  /**
4   *
5   * @author Mathias Thejsen s175192 && Simon Hansen s175191
6   *
7   */
8  public class Start extends SpecialProperty {
9      /**
10      * Constructor for start field
11      * @param id Field number
12      * @param name Field name
13      * @param description Field description
14      */
15     public Start(int id, String name, String description) {
16         super(id, name, description);
17     }
18 }
19
```

Street Source Code

```
1
2  package core;
3  import java.awt.*;
4  /**
5   *
6   * @author Mathias Thejsen s175192 && Simon Hansen s175191
7   *
8   */
9  public class Street extends Property {
10     private int houseCounter;
11     private int[] housePrices = new int[6];
12     private int BuildPrice;
13     private Color colour;
14     /**
15      * Constructor for street
16      * @param id
17      * @param name
18      * @param description
19      * @param owner
20      * @param buyValue
21      * @param pawnValue
22      * @param rentValue
23      * @param housePrices
24      * @param buildPrice
25      * @param colour
26      */
27     public Street(int id, String name, String description, Player owner, int buyValue, int pawnValue, int rentValue, int[] housePrices, int buildPrice, Color colour) {
28         super(id, name, description, owner, buyValue, pawnValue, rentValue);
29         this.houseCounter = 0; // Amount of house on the field
30         this.housePrices = housePrices; // The rent price when houses are build
31         this.BuildPrice = buildPrice; // The price for building a house/hotel
32         this.colour = colour; // The color for the field when owned by a player
33     }
34
35     /**
36      * Getter and Setters
37      */
38     public int getHouseCounter() {
39         return houseCounter;
40     }
41
42     public void setHouseCounter(int houseCounter) {
43         this.houseCounter = houseCounter;
44     }
45     public int getBuildPrice() {
46         return BuildPrice;
47     }
48     public void setBuildPrice(int buildPrice) {
49         BuildPrice = buildPrice;
50     }
51     public int[] getHousePrices() {
52         return housePrices;
53     }
54     public void setHousePrices(int[] housePrices) {
55         this.housePrices = housePrices;
56     }
57     public Color getColour() {
58         return colour;
59     }
60     public void setColour(Color type) {
61         colour = type;
62     }
63 }
64
```

StreetController Source Code

```
1
2  package core;
3  /**
4   * @author Mathias Thejsen s175192 && Simon Hansen s175191
5   *
6   */
7  public class StreetController {
8      private Street street;
9      private Player currentPlayer;
10     private GUIController guiController = GUIController.getInstance();
11     private AuctionController auctionController;
12     private Player[] players;
13     /**
14      * Constructor for normal logic
15      * @param currentPlayer
16      * @param field
17      */
18     public StreetController(Player currentPlayer, Field field, Player[] players) {
19         this.currentPlayer = currentPlayer;
20         this.street = (Street) field;
21         this.players = players;
22         auctionController = new AuctionController();
23     }
24     /**
25      * The logic when landing on a normal/property field (a field where you can put houses on)
26      */
27     public void logic() {
28         // Check if field is owned
29         if(street.getOwner() == null) {
30             // Check if we can afford it
31             if(currentPlayer.getAccount().canAfford(street.getBuyValue())) {
32                 // Prompt the player for a choice
33                 String[] choices = {PropertiesIO.getTranslation("yesbutton"), PropertiesIO.getTranslation("nobutton")};
34                 String result = guiController.requestPlayerChoiceButtons(PropertiesIO.getTranslation("streetlandonbuy")+" "+street.getName()+" for " + street.getBuyValue(),
35                         choices);
36
37                 // Check if choice is Yes
38                 if(result.equals(PropertiesIO.getTranslation("yesbutton"))) {
39
40                     // Initialize buyController
41                     BuyController buyController = new BuyController(currentPlayer, street);
42
43                     // Run the buyLogic method
44                     buyController.buyLogic();
45                 }
46                 else { // If the player does not wish to buy the field
47                     auctionController.startAuction(currentPlayer, street, players);
48                 }
49             }
50             else { // If the player cannot afford the field
51                 auctionController.startAuction(currentPlayer, street, players);
52             }
53         }
54         else{ // The field is owned
55
56             // Check if the player landing there is the owner itself
57             if(street.getOwner() != currentPlayer || street.isPawned()){
58                 // If field is owned by someone else, we check if they can afford landing there
59                 if(currentPlayer.getAccount().canAfford(street.getRentValue())) {
60                     // Show information to player
61                     guiController.writeMessage(PropertiesIO.getTranslation("streetlanddon")+" "+street.getName()+" " + PropertiesIO.getTranslation("streetlanddon2") + "
62                             "+street.getRentValue()+" til "+street.getOwner().getName());
63                     // Withdraw rentValue from the player
64                     currentPlayer.getAccount().withdraw(street.getRentValue());
65
66                     // Deposit rentValue to the owner
67                     street.getOwner().getAccount().deposit(street.getRentValue());
68
69                     // Sends updates to GUIController
70                     guiController.updatePlayerBalance(street.getOwner().getGuiId(), street.getOwner().getAccount().getBalance());
71                     guiController.updatePlayerBalance(currentPlayer.getGuiId(), currentPlayer.getAccount().getBalance());
72
73                 }
74                 else { // Can't afford
75                     guiController.writeMessage(PropertiesIO.getTranslation("streetcantafford"));
```

```
76                  //  Initialize  the  SalesController
77                  SalesController  salesController  = new SalesController(currentPlayer);
78
79                  // Run cannotAfford method
80                  boolean response =  salesController.cannotAfford(street.getRentValue());
81                  if(response) {
82                      guiController.writeMessage(PropertiesIO.getTranslation("streetcanpayrent")+ " "+street.getRentValue());
83
84                      // Withdraw rentValue from the player
85                      currentPlayer.getAccount().withdraw(street.getRentValue());
86
87                      // Deposit rentValue to the owner
88                      street.getOwner().getAccount().deposit(street.getRentValue());
89
90                      // Sends updates to GUIController
91                      guiController.updatePlayerBalance(street.getOwner().getGuiId(), street.getOwner().getAccount().getBalance());
92                      guiController.updatePlayerBalance(currentPlayer.getGuiId(), currentPlayer.getAccount().getBalance());
93                  }
94              }
95          }
96      }
97  }
98  }
```

Tax Source Code

```
1
2  package core;
3  public class Tax extends SpecialProperty {
4      public Tax(int id, String name, String description) {
5          super(id, name, description);
6      }
7
8  }
9
```

TaxController Source Code

```
1
2  package core;
3  /**
4   *
5   * @author Mathias Thejsen s175192 && Simon Hansen s175191
6   *
7   */
8  public class TaxController {
9      private Player currentPlayer;
10     private Field[] fields;
11     private GUIController guiController = GUIController.getInstance();
12     /**
13      * Constructor for tax logic
14      * @param currentPlayer
15      * @param fields a field array
16      */
17     public TaxController(Player currentPlayer, Field[] fields) {
18         this.currentPlayer = currentPlayer;
19         this.fields = fields;
20     }
21     /**
22      * If we land on field 39, we have to pay 2000
23      * @return depends on outcome
24      */
25     public void taxLogic38() {
26         // Check if the player can afford the tax
27         if (currentPlayer.getAccount().canAfford(Integer.parseInt(PropertiesIO.getTranslation("statetaxvalue")))) {
28
29             // Withdraw the value from the players account
30             currentPlayer.getAccount().withdraw(Integer.parseInt(PropertiesIO.getTranslation("statetaxvalue")));
31
32             // Send updates to the GUIController
33             guiController.updatePlayerBalance(currentPlayer.getGuiId(), currentPlayer.getAccount().getBalance());
34             guiController.writeMessage(PropertiesIO.getTranslation("statetatxpaymentstr"));
35         }
36         else { // If the player cannot afford the tax
37
38             // Notify the player
39             guiController.writeMessage(PropertiesIO.getTranslation("statetaxnotpaidstr"));
40
41             // Make SalesController
42             SalesController salesController = new SalesController(currentPlayer);
43
44             // Calls the cannotAfford method, which returns a boolean depending on whether or not the player can afford the tax after selling/pawning
45             boolean response = salesController.cannotAfford(Integer.parseInt(PropertiesIO.getTranslation("statetaxvalue")));
46             if(response) { // If we can afford the tax after selling/pawning
47
48                 // Withdraw the tax from the player
49                 currentPlayer.getAccount().withdraw(Integer.parseInt(PropertiesIO.getTranslation("statetaxvalue")));
50
51                 // Send update to the GUIController
52                 guiController.updatePlayerBalance(currentPlayer.getGuiId(), currentPlayer.getAccount().getBalance());
53                 guiController.writeMessage(PropertiesIO.getTranslation("statetatxpaymentstr"));
54             }
55         }
56     }
57     /**
58      * If we land on field 5, you can either pay 10% of income tax or 4000
59      * @return Depends on outcome
60      */
```

```
61  public void taxLogic4() {
62      int buildingvalue = 0; // Variable used to hold value of houses
63      int playervalue = currentPlayer.getAccount().getBalance(); // The players current balance
64      int propertyvalue = 0; // Variable used to hold value of properties
65      SalesController salesController = new SalesController(currentPlayer); // Initialize a SalesController
66
67      // Make the two choices for the player
68      String[] choices = {"4000", "10%"};
69
70      // Prompt user for a choice
71      String choice = guiController.requestPlayerChoiceButtons(PropertiesIO.getTranslation("taxoptions"), choices);
72
73      // If they chose the 10%
74      if (choice.equals("10%")) {
75          // Calculates 10% of income tax
76
77          // looping over all fields
78          for (int i = 0; i < fields.length; i++) {
79
80              // Only dealing with fields that are property -> can be owned
81              if (fields[i] instanceof Property) {
82
83                  // Casting & initializing a new property as we know fields[i] represents a property
84                  Property property = (Property) fields[i];
85
86                  // Find those that the player owns
87                  if (property.getOwner() == currentPlayer) {
88
89                      // The property value is a sum of all the basevalues
90                      propertyvalue = propertyvalue + property.getBuyValue();
91                  }
92
93              }
94              // If we are dealing with streets ->Properties that can have houses
95              else if(fields[i] instanceof Street) {
96                  Street street = (Street) fields[i];
97
98                  // Loop over all the houses
99                  for (int j = 0; j < street.getHouseCounter(); j++) {
100
101                     // adding up all the house prices
102                     buildingvalue = buildingvalue + street.getHousePrices()[i];
103                 }
104             }
105         }
106         // Take 10% of it
107         int value = (int) ((buildingvalue + playervalue + propertyvalue) * 0.10);
108         // Check if we can afford
109         if(currentPlayer.getAccount().canAfford(value)) {
110
111             // If we can, withdraw the value from the player
112             currentPlayer.getAccount().withdraw(value);
113
114             // Send update to the GUIController
115             guiController.updatePlayerBalance(currentPlayer.getGuiId(), currentPlayer.getAccount().getBalance());
116             guiController.writeMessage(PropertiesIO.getTranslation("statetaxtotal")+value);
117         }
118         // If we cannot afford it
119         else {
120
121             //Notify the player
122             guiController.writeMessage(PropertiesIO.getTranslation("incometaxcantafford"));
123
124             // Call the cannotAfford method, which keeps running until the player can either afford the tax or have gone bankrupt
125             boolean response = salesController.cannotAfford(value);
126
127             // If the player can afford the tax after selling
128             if(response) {
129
130                 // Withdraw the value from the player
131                 currentPlayer.getAccount().withdraw(value);
132
133                 // Send update to the GUIController
134                 guiController.updatePlayerBalance(currentPlayer.getGuiId(), currentPlayer.getAccount().getBalance());
135                 guiController.writeMessage(PropertiesIO.getTranslation("statetaxtotal")+value);
136             }
137         }
```

```
138
139         }else{ // If the choice is to pay 4000
140
141             // Check if the player can afford the tax
142             if (currentPlayer.getAccount().canAfford(Integer.parseInt(PropertiesIO.getTranslation("incometaxvalue")))) {
143
144                 // If the player can afford it, withdraw the tax from the players account
145                 currentPlayer.getAccount().withdraw(Integer.parseInt(PropertiesIO.getTranslation("incometaxvalue")));
146
147                 // Send update to the GUIController
148                 guiController.updatePlayerBalance(currentPlayer.getGuiId(), currentPlayer.getAccount().getBalance());
149             }
150             // If the player cannot afford the tax
151             else {
152
153                 // Notify the player
154                 guiController.writeMessage(PropertiesIO.getTranslation("incometaxcantafford"));
155
156                 // Call the cannotAfford method, which keeps running until the player can either pay the tax or have gone bankrupt
157                 boolean response = salesController.cannotAfford(Integer.parseInt(PropertiesIO.getTranslation("incometaxvalue")));
158
159                 // If the player can afford the tax after selling/pawning
160                 if(response) {
161
162                     // Notify the player
163                     guiController.writeMessage(PropertiesIO.getTranslation("incometaxnowavailable"));
164
165                     // Withdraw the tax from the player
166                     currentPlayer.getAccount().withdraw(Integer.parseInt(PropertiesIO.getTranslation("incometaxvalue")));
167
168                     // Send an update to the GUIController
169                     guiController.updatePlayerBalance(currentPlayer.getGuiId(), currentPlayer.getAccount().getBalance());
170                 }
171             }
172         }
173     }
174     /**
175      * Method that checks which tax field the player has landed on
176      */
177     public void taxLogic() {
178
179         // Check if the player landed on the tax field on field 38
180         if(currentPlayer.getEndPosition() == 38) {
181             taxLogic38();
182         }
183         // If the player has landed on the tax field on field 4
184         else {
185             taxLogic4();
186         }
187     }
188 }
```

TradeController Source Code

```
1
2  package core;
3  /**
4   * This class handles the trading of properties within the game
5   * @author Magnus Stjernborg Koch s175189 & Nicolai Kammersgård s143780
6   *
7   */
8  public class TradeController {
9      private GUIController guiController = GUIController.getInstance();
10     /**
11      * This method initialises the trade
12      * @param currentPlayer
13      * @param fieldController
14      * @param players
15      */
16     public void startTrade(Player currentPlayer, FieldController fieldController, Player[] players) {
17         String[] choices = new String[players.length-1];
18         boolean playerempty = true;
19         int counter = 0;
20         for (int i = 0; i < players.length; i++) { // Looping over all the players
21             if (!(players[i] == currentPlayer) && fieldController.FieldsOwned(players[i]).length > 0) { // We don't want the player to be able to trade to himself and we want
22                     to make sure the players actually owns some fields
23                 choices[counter++] = players[i].getName(); // Add the players to a list
24                 playerempty = false; // Boolean to keep track if we actually have any players that can trade something
25             }
26         }
27         String[] actualchoices = new String[counter]; // make a new array and fill out
28         for (int i = 0; i < actualchoices.length; i++) {
29             actualchoices[i] = choices[i];
30         }
31         if(!playerempty) { // if we dont have any players that can trade, then don't start the trading stuff
32             String response = guiController.requestPlayerChoiceButtons(PropertiesIO.getTranslation("tradewhototradewith"), actualchoices);
33             for (int i = 0; i < players.length; i++) { // Loop over all players
34                 if (players[i].getName().equals(response)) { // find the player the currentPlayer choose to trade with
35                     if (fieldController.FieldsOwned(players[i]).length > 0) { // Make sure he owns some fields
36                         response = guiController.requestPlayerChoiceButtons(PropertiesIO.getTranslation("tradewhototradewith"), fieldController.FieldsOwned(players[i])); //Pick a
37                                 field
38                     }
39                 }
40             }
41             Field[] fields = fieldController.getFieldArr();
42             for (int i = 0; i < fields.length; i++) { // Loop over all fields
43                 if (fields[i].getName().equals(response)) { // find the field the player chose
44                     Property property = (Property) fields[i];
45                     int amountOffered = 0;
46                     boolean cannotAfford = true;
47                     do {
48                         amountOffered = guiController.requestIntegerInput(PropertiesIO.getTranslation("tradehowmuchoffer"));
49                         if(currentPlayer.getAccount().canAfford(amountOffered)) {
50                             cannotAfford = false;
51                         }
52                     } while(cannotAfford);
53                     // Ask the player if he wishses to acccept the offer
54                     response = guiController.requestPlayerChoiceButtons(property.getOwner().getName()+PropertiesIO.getTranslation(" tradeacceptoffer ")+" "+amountOffered+" for
55                             "+property.getName()+"?",PropertiesIO.getTranslation("yesbutton"),PropertiesIO.getTranslation("nobutton"));
56                     if (response.equals(PropertiesIO.getTranslation("yesbutton"))) {
57                         currentPlayer.getAccount().withdraw(amountOffered);
58                         property.getOwner().getAccount().deposit(amountOffered);
59                         guiController.updatePlayerBalance(currentPlayer.getGuiId(), currentPlayer.getAccount().getBalance());
60                         guiController.updatePlayerBalance(property.getOwner().getGuiId(), property.getOwner().getAccount().getBalance());
61                         property.setOwner(currentPlayer);
62                         guiController.setOwner(currentPlayer.getGuiId(), property.getId());
63                     }
64                 }
65             }
66         }
67     }
68 }
69
```

ChanceCard Source Code

```
1
2  package core.ChanceCardLogic;
3  /**
4   * @author Magnus Stjernborg Koch — s175189
5   *
6   */
7  public abstract class ChanceCard {
8      // Field id, goes easier together with other controllers
9      private int id;
10     //For GUI purposes
11     private String description;
12     public ChanceCard(int id, String description) {
13         this.id = id;
14         this.description = description;
15     }
16     public int getId() {
17         return id;
18     }
19     public void setId(int id) {
20         this.id = id;
21     }
22     public String getDescription() {
23         return description;
24     }
25     public void setDescription(String description) {
26         this.description = description;
27     }
28 }
29
```

ChanceCardController Source Code

```
1
2  package core.ChanceCardLogic;
3  import core.*;
4  import java.util.concurrent.ThreadLocalRandom;
5  /**
6   * @author Magnus Stjernborg Koch — s175189
7   *
8   */
9  public class ChanceCardController {
10     private GUIController guiController;
11     //Deck to hold the different kind of cards
12     private ChanceCardDeck chanceCardDeck;
13     //Saving the prisonCard for discard purposes
14     private PrisonCard prisonCard;
15     public ChanceCardController() {
16         guiController = GUIController.getInstance();
17         // initialising chancecarddeck with the amount of 32 ChanceCards
18         chanceCardDeck = new ChanceCardDeck(32);
19         // Initialising chancards with the specifications from config/PropertiesIO
20         initializeChanceCards();
21     }
22     private void initializeChanceCards() {
23         ChanceCard[] chanceCardArray = new ChanceCard[32];
24         //Randomarray for making a unsorted deck
25         int[] randomArray = new int[32];
26         // Initialising the random array, creating 32 different ints from 0 to 31, so each card has its own id based on field id
27         initializeRandomArray(randomArray);
28         //Loop goes through all chancecards and adding them to the array of chanceCards
29         for (int i = 0; i < chanceCardArray.length; i++) {
30             switch (i) {
31             case 0:
32             case 1:
33                 chanceCardArray[i] = new PrisonCard(i, PropertiesIO.getTranslation("chance"+(i+1)));
34                 break;
35             case 2:
36             case 3:
37                 chanceCardArray[i] = new MoveCard(i, PropertiesIO.getTranslation("chance"+(i+1)), 10);
38                 break;
39             case 4:
40             case 5:
41                 chanceCardArray[i] = new MoveShippingCard(i, PropertiesIO.getTranslation("chance"+(i+1)));
42                 break;
43             case 6:
44             case 7:
45             case 8:
46             case 9:
47             case 10:
48             case 11:
49                 chanceCardArray[i] = new DepositCard(i, PropertiesIO.getTranslation("chance"+(i+1)),1000);
50                 break;
51             case 12:
52                 chanceCardArray[i] = new MoveCard(i, PropertiesIO.getTranslation("chance"+(i+1)),0);
53                 break;
54             case 13:
55                 chanceCardArray[i] = new DepositCard(i, PropertiesIO.getTranslation("chance"+(i+1)),500);
56                 break;
57             case 14:
58                 chanceCardArray[i] = new DepositCard(i, PropertiesIO.getTranslation("chance"+(i+1)),3000);
59                 break;
60             case 15:
61                 chanceCardArray[i] = new MoveCard(i, PropertiesIO.getTranslation("chance"+(i+1)), 39);
62                 break;
63             case 16:
64                 chanceCardArray[i] = new StepsBackCard(i, PropertiesIO.getTranslation("chance"+(i+1)),3);
65                 break;
66             case 17:
67                 chanceCardArray[i] = new GrantCard(i, PropertiesIO.getTranslation("chance"+(i+1)));
68                 break;
69             case 18:
70                 chanceCardArray[i] = new DepositCard(i, PropertiesIO.getTranslation("chance"+(i+1)),200);
71                 break;
72             case 19:
73                 chanceCardArray[i] = new WithdrawCard(i, PropertiesIO.getTranslation("chance"+(i+1)),2000);
74                 break;
75             case 20:
```

```
76                  chanceCardArray[i] = new EstateTaxCard(i, PropertiesIO.getTranslation("chance"+(i+1)),500,2000);
77                  break;
78              case 21:
79                  chanceCardArray[i] = new EstateTaxCard(i, PropertiesIO.getTranslation("chance"+(i+1)),800,2300);
80                  break;
81              case 22:
82                  chanceCardArray[i] = new WithdrawCard(i, PropertiesIO.getTranslation("chance"+(i+1)),1000);
83                  break;
84              case 23:
85              case 24:
86                  chanceCardArray[i] = new WithdrawCard(i, PropertiesIO.getTranslation("chance"+(i+1)),200);
87                  break;
88              case 25:
89                  chanceCardArray[i] = new MoveCard(i, PropertiesIO.getTranslation("chance"+(i+1)),11);
90                  break;
91              case 26:
92              case 27:
93                  chanceCardArray[i] = new WithdrawCard(i, PropertiesIO.getTranslation("chance"+(i+1)),3000);
94                  break;
95              case 28:
96                  chanceCardArray[i] = new WithdrawCard(i, PropertiesIO.getTranslation("chance"+(i+1)),1000);
97                  break;
98              case 29:
99                  chanceCardArray[i] = new MoveCard(i, PropertiesIO.getTranslation("chance"+(i+1)),5);
100                 break;
101             case 30:
102                 chanceCardArray[i] = new PresentDepositCard(i, PropertiesIO.getTranslation("chance"+(i+1)));
103                 break;
104             case 31:
105                 chanceCardArray[i] = new MoveCard(i, PropertiesIO.getTranslation("chance"+(i+1)),24);
106                 break;
107             }
108         }
109         //Saving the prisonCard so it easier can be removed later.
110         prisonCard = (PrisonCard) chanceCardArray[0];
111         //Using the randomarray to push each card in the deck unsorted and random
112         for (int i = 0; i < chanceCardArray.length; i++) {
113             //Pushing card to deck (queue structure)
114             chanceCardDeck.push(chanceCardArray[randomArray[i]]);
115         }
116     }
117     /**
118      * Function for creating the random array to be used to making a random order in the ChanceCardDeckArray
119      *
120      */
121     private void initializeRandomArray (int[] randomArray) {
122         for (int i = 0; i < randomArray.length; i++) {
123             int rndNumber;
124             boolean newCard;
125             do {
126                 newCard = true;
127                 rndNumber = ThreadLocalRandom.current().nextInt(0, randomArray.length);
128                 for (int j = 0; j < i; j++)
129                     if (randomArray[j] == rndNumber) {
130                         newCard = false;
131                         break;
132                     }
133             } while (!newCard);
134             randomArray[i] = rndNumber;
135         }
136     }
137     /**
138      * The function called by other controllers when landed on the ChanceCard field, then it handles both the logic and
139      * the user interface related to chance cards
140      *
141      */
142     public void getCard (Player currentPlayer, Field[] fields, Player[] players) {
143         //picks a new card from the bottom of the deck.
144         ChanceCard drawnChanceCard = chanceCardDeck.bottom();
145         // getting the description for user interface
146         guiController.displayChanceCard(drawnChanceCard.getDescription());
147         guiController.writeMessage(PropertiesIO.getTranslation("takeachancecard"));
148         //Checks with type of chanceCard has been drawned, and then calls a function related to that type of chance card
149         if (drawnChanceCard instanceof PrisonCard) {
150             prisonCard(currentPlayer);
151         }else if (drawnChanceCard instanceof MoveCard) {
152             moveCard(currentPlayer, ((MoveCard) drawnChanceCard).getField(), drawnChanceCard);
```

```
153         }else if(drawnChanceCard instanceof MoveShippingCard) {
154             moveShippingCard(currentPlayer);
155         }else if(drawnChanceCard instanceof GrantCard) {
156             grantCard(currentPlayer, fields);
157         }else if(drawnChanceCard instanceof PresentDepositCard) {
158             presentDepositCard(currentPlayer, players);
159         }else if(drawnChanceCard instanceof StepsBackCard) {
160             stepsBackCard(currentPlayer, ((StepsBackCard) drawnChanceCard).getAmount());
161         }else if(drawnChanceCard instanceof WithdrawCard) {
162             withdrawCard(currentPlayer, ((WithdrawCard) drawnChanceCard).getAmount());
163         }else if(drawnChanceCard instanceof DepositCard) {
164             depositCard(currentPlayer, ((DepositCard) drawnChanceCard).getAmount());
165         }else if(drawnChanceCard instanceof EstateTaxCard) {
166             estateTaxCard(currentPlayer, fields, ((EstateTaxCard) drawnChanceCard).getTaxHouse(), ((EstateTaxCard) drawnChanceCard).getTaxHotel());
167         }
168         // If the drawn card were a prison card dont push it back.
169         if(!(drawnChanceCard instanceof PrisonCard)) {
170             chanceCardDeck.push(drawnChanceCard);
171         }
172     }
173     /**
174      * This function adds a prison card to a user which drawn the card
175      *
176      */
177     private void prisonCard(Player currentPlayer) {
178         currentPlayer.addPrisonCard();
179     }
180     /**
181      * This function is moving the player depending on which moveCard were drawn
182      *
183      */
184     private void moveCard(Player currentPlayer, int field, ChanceCard drawnChanceCard) {
185         currentPlayer.setStartPosition(currentPlayer.getEndPosition());
186         currentPlayer.setEndPosition(field);
187         // if move to prisonfield
188         if(drawnChanceCard.getId() == 2 && drawnChanceCard.getId() == 3) {
189             guiController.teleport(currentPlayer.getGuiId(), currentPlayer.getStartPosition(), currentPlayer.getEndPosition());
190         } else {
191             guiController.updatePlayerPosition(currentPlayer.getGuiId(), currentPlayer.getEndPosition(), currentPlayer.getStartPosition());
192             currentPlayer.setMoved(true);
193             if(currentPlayer.getEndPosition() < currentPlayer.getStartPosition() && currentPlayer.getEndPosition() != 0) {
194                 currentPlayer.getAccount().deposit(4000);
195                 guiController.updatePlayerBalance(currentPlayer.getGuiId(), currentPlayer.getAccount().getBalance());
196             }
197         }
198     }
199     /**
200      * This function moves to the closes shipping field depending on which chance field the player landed on
201      *
202      */
203     private void moveShippingCard(Player currentPlayer) {
204         currentPlayer.setStartPosition(currentPlayer.getEndPosition());
205         if(currentPlayer.getEndPosition() == 2) {
206             currentPlayer.setEndPosition(5);
207         }
208         if(currentPlayer.getEndPosition() == 7) {
209             currentPlayer.setEndPosition(15);
210         }
211         if(currentPlayer.getEndPosition() == 17 || currentPlayer.getEndPosition() == 22) {
212             currentPlayer.setEndPosition(25);
213         }
214         if(currentPlayer.getEndPosition() == 33) {
215             currentPlayer.setEndPosition(35);
216         }
217         if(currentPlayer.getEndPosition() == 36) {
218             currentPlayer.setEndPosition(5);
219             currentPlayer.getAccount().deposit(4000);
220         }
221         guiController.updatePlayerPosition(currentPlayer.getGuiId(), currentPlayer.getEndPosition(), currentPlayer.getStartPosition());
222         guiController.updatePlayerBalance(currentPlayer.getGuiId(), currentPlayer.getAccount().getBalance());
223         currentPlayer.setMoved(true);
224     }
225     /**
226      * This function is checking if the player's networth is higher or lower then 15.000 if yes then ads 40000 to balance
227      *
228      */
229     private void grantCard(Player currentPlayer, Field[] fields) {
```

```
230         int playerValue = 0;
231         for (int i = 0; i < fields.length; i++) {
232             if (fields[i] instanceof Property) {
233                 Property property = (Property) fields[i];
234                 if (property.getOwner() == currentPlayer) {
235                     playerValue += property.getBuyValue();
236                     if (property instanceof Street) {
237                         Street street = (Street) property;
238                         if (street.getHouseCounter() > 0) {
239                             for (int j = 0; j < street.getHouseCounter(); j++) {
240                                 playerValue += street.getBuildPrice();
241                             }
242                         }
243                     }
244                 }
245             }
246         }
247         playerValue += currentPlayer.getAccount().getBalance();
248         if (playerValue <= 15000) {
249             currentPlayer.getAccount().deposit(40000);
250             guiController.updatePlayerBalance(currentPlayer.getGuiId(), currentPlayer.getAccount().getBalance());
251         }
252     }
253     /**
254      * This function takes 200 money from each player (not the currentplayer) and give all the total money to the currentplayer)
255      *
256      */
257     private void presentDepositCard(Player currentPlayer, Player[] players) {
258         int present = 0;
259         for (int i = 0; i < players.length; i++) {
260             if (!(players[i] == currentPlayer)) {
261                 players[i].getAccount().withdraw(200);
262                 present += 200;
263                 guiController.updatePlayerBalance(players[i].getGuiId(), players[i].getAccount().getBalance());
264             }
265         }
266         currentPlayer.getAccount().deposit(present);
267         guiController.updatePlayerBalance(currentPlayer.getGuiId(), currentPlayer.getAccount().getBalance());
268     }
269     /**
270      * This function moves the player some steps back depending on the card
271      *
272      */
273     private void stepsBackCard(Player currentPlayer, int amountOfSteps) {
274         currentPlayer.setStartPosition(currentPlayer.getEndPosition());
275         if ((currentPlayer.getEndPosition() - amountOfSteps) < 0)
276             currentPlayer.setEndPosition(39);
277         currentPlayer.setEndPosition(currentPlayer.getEndPosition() - amountOfSteps);
278         guiController.teleport(currentPlayer.getGuiId(), currentPlayer.getStartPosition(), currentPlayer.getEndPosition());
279         currentPlayer.setMoved(true);
280     }
281     /**
282      * This function is withdrawing a specific amount of money from the current player depending on which card
283      *
284      */
285     private void withdrawCard(Player currentPlayer, int amount) {
286         int currentPlayerBalance = currentPlayer.getAccount().getBalance();
287         if (checkIfAfford(currentPlayer, amount))
288             currentPlayer.getAccount().withdraw(amount);
289         else {
290             currentPlayer.getAccount().withdraw(currentPlayerBalance);
291             currentPlayer.setBanktrupt(true);
292         }
293         guiController.updatePlayerBalance(currentPlayer.getGuiId(), currentPlayer.getAccount().getBalance());
294     }
295     /**
296      * This function is depositing a specific amount of money to the current player depending on which card
297      *
298      */
299     private void depositCard(Player currentPlayer, int amount) {
300         currentPlayer.getAccount().deposit(amount);
301         guiController.updatePlayerBalance(currentPlayer.getGuiId(), currentPlayer.getAccount().getBalance());
302     }
303     /**
304      * This function is checking which fields is owned by the currentplayer and how many houses and hotels are placed on them
305      * then withdrawing a tax from each house and hotel.
306      *
```

```
307         */
308     private void estateTaxCard (Player currentPlayer, Field[] fields, int housetax, int hoteltax) {
309         int estateTax = 0;
310         for (int i = 0; i < fields.length; i++) {
311             if (fields[i] instanceof Street) {
312                 Street street = (Street) fields[i];
313                 if (street.getOwner() == currentPlayer) {
314                     if (street.getHouseCounter() == 5) {
315                         estateTax += hoteltax;
316                     } else {
317                         estateTax += (housetax * (street.getHouseCounter()));
318                     }
319                 }
320             }
321         }
322         currentPlayer.getAccount().withdraw(estateTax);
323         guiController.updatePlayerBalance(currentPlayer.getGuiId(), currentPlayer.getAccount().getBalance());
324     }
325     /**
326      * This function is called when the prisoncard is used and needs to be return to the deck, so it can be drawn by others
327      *
328      */
329     public void putPrisonCardInDeck() {
330         chanceCardDeck.push(prisonCard);
331     }
332     /**
333      * This function is using the SalesController to check if the player can afford the different types of cards.
334      *
335      */
336     private boolean checkIfAfford (Player currentPlayer, int value) {
337         SalesController salesController = new SalesController(currentPlayer);
338         return salesController.cannotAfford(value);
339     }
340 }
341
```

ChanceCardDeck Source Code

```
1
2  package core.ChanceCardLogic;
3  /**
4   * @author Magnus Stjernborg Koch — s175189
5   *
6   */
7  public class ChanceCardDeck {
8      //When all cards are added
9      private int maxSize;
10     //The array of the chance cards
11     private ChanceCard[] stackArray;
12     private int top;
13     public ChanceCardDeck(int s) {
14         maxSize = s;
15         stackArray = new ChanceCard[maxSize];
16         top = -1;
17     }
18     //Pushing a chancecard into the deck
19     public void push(ChanceCard j) {
20         stackArray[++top] = j;
21     }
22     //Takes the card from the bottom of the deck and then reorder the deck so all cards moves one down in the order.
23     public ChanceCard bottom() {
24         ChanceCard[] tempStackArray = new ChanceCard[maxSize];
25         ChanceCard returnCard = stackArray[0];
26
27         --top;
28         for(int i = 0 ; i <= top ; i++) {
29             tempStackArray[i] = stackArray[i+1];
30         }
31         stackArray = tempStackArray;
32
33         return returnCard;
34     }
35 }
36
```

DepositCard Source Code

```
1
2  package core.ChanceCardLogic;
3  /**
4   * @author Magnus Stjernborg Koch — s175189
5   *
6   */
7  public class DepositCard extends ChanceCard {
8      //Amount of money to deposit
9      private int amount;
10     public DepositCard(int id, String description, int amount) {
11         super(id, description);
12         this.amount = amount;
13     }
14     public int getAmount() {
15         return amount;
16     }
17     public void setAmount(int amount) {
18         this.amount = amount;
19     }
20 }
21
```

EstateTaxCard Source Code

```
1
2  package core.ChanceCardLogic;
3  /**
4   * @author Magnus Stjernborg Koch — s175189
5   *
6   */
7  public class EstateTaxCard extends ChanceCard {
8      //The specific tax for houses
9      private int taxHouse;
10     //The specific tax for hotels
11     private int taxHotel;
12     public EstateTaxCard(int id, String description, int taxHouse, int taxHotel) {
13         super(id, description);
14         this.taxHouse = taxHouse;
15         this.taxHotel = taxHotel;
16     }
17     public int getTaxHouse() {
18         return taxHouse;
19     }
20     public void setTaxHouse(int taxHouse) {
21         this.taxHouse = taxHouse;
22     }
23     public int getTaxHotel() {
24         return taxHotel;
25     }
26     public void setTaxHotel(int taxHotel) {
27         this.taxHotel = taxHotel;
28     }
29 }
30
```

GrantCard Source Code

```
1
2  package core.ChanceCardLogic;
3  /**
4   * @author Magnus Stjernborg Koch — s175189
5   *
6   */
7  public class GrantCard extends ChanceCard {
8      //What the grant amount is
9      private int grantAmount;
10     public GrantCard(int id, String description) {
11         super(id, description);
12         this.grantAmount = 40000;
13     }
14     public int getGrantAmount() {
15         return grantAmount;
16     }
17     public void setGrantAmount(int grantAmount) {
18         this.grantAmount = grantAmount;
19     }
20 }
21
```

MoveCard Source Code

```
1
2  package core.ChanceCardLogic;
3  /**
4   * @author Magnus Stjernborg Koch — s175189
5   *
6   */
7  public class MoveCard extends ChanceCard {
8      // Specific field to place the player on
9      private int field;
10     public MoveCard(int id, String text, int field) {
11         super(id, text);
12         this.field = field;
13     }
14     public int getField() {
15         return field;
16     }
17     public void setField(int field) {
18         this.field = field;
19     }
20 }
21
```

MoveShippingCard Source Code

```
1
2  package core.ChanceCardLogic;
3  /**
4   * @author Magnus Stjernborg Koch — s175189
5   *
6   */
7  public class MoveShippingCard extends ChanceCard {
8      public MoveShippingCard(int id, String description) {
9          super(id, description);
10     }
11 }
12
```

PresentDepositCard Source Code

```
1
2  package core.ChanceCardLogic;
3  /**
4   * @author Magnus Stjernborg Koch — s175189
5   *
6   */
7  public class PresentDepositCard extends ChanceCard {
8      public PresentDepositCard(int id, String description) {
9          super(id, description);
10     }
11 }
12
```

PrisonCard Source Code

```
1
2  package core.ChanceCardLogic;
3  /**
4   * @author Magnus Stjernborg Koch — s175189
5   *
6   */
7  public class PrisonCard extends ChanceCard {
8      public PrisonCard(int id, String text) {
9          super(id, text);
10     }
11 }
12
```

StepsBackCard Source Code

```
1
2  package core.ChanceCardLogic;
3  /**
4   * @author Magnus Stjernborg Koch — s175189
5   *
6   */
7  public class StepsBackCard extends ChanceCard{
8      //The amount of steps going back
9      private int amount;
10     public StepsBackCard(int id, String description, int amount) {
11         super(id, description);
12         this.amount = amount;
13     }
14     public int getAmount() {
15         return amount;
16     }
17     public void setAmount(int amount) {
18         this.amount = amount;
19     }
20 }
21
```

WithdrawCard Source Code

```
1
2  package core.ChanceCardLogic;
3  /**
4   * @author Magnus Stjernborg Koch — s175189
5   *
6   */
7  public class WithdrawCard extends ChanceCard {
8      //The amount to withdraw
9      private int amount;
10     public WithdrawCard(int id, String description, int amount) {
11         super(id, description);
12         this.amount = amount;
13     }
14     public int getAmount() {
15         return amount;
16     }
17     public void setAmount(int amount) {
18         this.amount = amount;
19     }
20 }
21
```