Projektopgave maj 16

02362 Projekt i software-udvikling F16
Projekt navn: *CDIO*

Gruppe nr.: *02*

Afleveringsfrist: *fredag d. 13/5-2015 kl: 13:00*

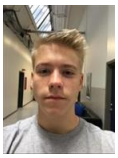
Denne rapport er afleveret via Campusnet (der skrives ikke under)

Denne rapport indeholder 40 sider inkl. denne side.

Studie nr.:	Efternavn, Fornavne	Underskrift
s144219,	*Gregersen, Vulpius Rasmus*	_____

Kontakt person (Projektleder)

s153256, *Moulvad, Rasmus Hannibal Barlach* _____



s143312, *Kristensen, David Josef* _____

Table of contents

Abstract 3
Introduction 4
Requirement Analysis 5
Functional Requirements 5
Non-Functional Requirements 5
System Analysis 6
GRASP Principles 6
3-layer model: 7
Use Cases 8
Use Case Diagram 8
State Machine Analysis 11
State Machine Table Overview 12
State Machine Diagram, Player and Field 12
Domain Model 13
Entity-Relationship Diagram 14
BCE Model 15
System Sequence Diagram 16
System Design and Implementation 17
Implementation 17
Design Sequence Diagram 18
Design Sequence Diagram, Set Pawned 19
Design Sequence Diagram, Roll Dice 20
Design Sequence Diagram, Buy Property 21
Design Class-Diagram with Named Relations 22
MySQL 24
Normalization 24
Views 24
Stored Procedures 24
Transaction Logic 25
Data Access Layer 25
JDBC 25
DAO Pattern 26
Database Structure 26
Enhanced Entity-Relationship Diagram 26
Testing 27
Testing Strategy 27
Manual Testing 27
Junit Test 27
Gameboard Test 28
Player Test 29
Player Options Test 30
Conclusion 31
Appendix 32
Game Rules 32
Configuration 35
Javadoc 37
Timesheet 40
Word- and Symbol Explanation, Abbreviation 40

Abstract

This report is composed in the courses 02327 Preliminary databases and database programming and 02362 Project in software development during our second term on the IT and Economics on DTU.

We were given the task of developing a complete Matador game with the additional features of saving and loading the game and all its corresponding states. We have extensively analyzed the functional and non functional requirements, with the help of software diagrams and models we managed to plan how to design a comprehensive game. Through various testing methods, developed a reliable and user friendly game.

Introduction

This game is far more advanced than our previous related projects. The Matador game requires database implementation to save and load games. The amount of rules has increased significantly and those new rules challenges us with many new functional requirements. In the production of the game our job as programmers is to design, construct and test the game. We intend to produce a reliable game that meets the requirements and reflect our development in programming and new knowledge of database implementation.

Requirement Analysis

The given requirements in the project description is to produce the final version of the Matador game. The main goal is to end up with a game with as many as the rules and functionalities as the original Matador game. The game should take objects states into account. We are to use the same GUI that were used in the previously projects. The last requirement in this project is to make it possible for players to save and load game in the specific state it's in.

Functional Requirements

- Game Rules[1]
  - Game board
    - 40 fields of different types.
  - A pair of dice
  - 2-6 players
- Save and Load game.

Non-Functional Requirements

- Is constructed after a multi layer model, and divided in relevant interfaces and classes.
- Avoid repetition of code and implementation of help methods and inheritance.
- Contains examples of Exceptions, collection, input validation with Regex or similar.
- Save and Load game, should utilize MySQL and JDBC.
- Implementation of the GUI from last semester.

[1] The complete rule set can be found in, Appendix, Game Rules.

System Analysis

GRASP Principles

GRASP is an abbreviation for General Responsibility Assignment Software Patterns.

We have made our project with regards to the 5 main principles of GRASP.

- Creator
- Information Expert
- Low Coupling
- Controller
- High Cohesion

Creator

To comply with the Creator principle. We have throughout the game initialized classes by it's logical dependent class. I.e.

- Our Game class creates the Gameboard, which there after creates all the individual fields, i.e. streets, shipping etc. and initializes them in the GUI and the Chance deck.
- Rules creates Player: And the Rules class keeps track of the Player and all its states.
- Chance deck initializes and keeps track of the 32 different individual chancecards.

Information Expert

We have assigned the responsibilities of our information to corresponding classes. All information regarding the player lies within the "Player class".

Low Coupling

We have met the low coupling measure I.e. our "Player class", which have no dependencies on other classes.

Controller

Our Game class is used as our Controller, since it deals and delegates the work to the other objects. Game organizes this primarily through our Gameboard, PlayerOptions and Rules classes.

High Cohesion

To obtain an effective and simple system, we made classes that have a clear and simple purpose, which relates to another class such as "ChanceDeck" and "Chance Card" class.

3-layer model:

In programming it is a beneficial to split up the subsystems in layers. This can be done with the 3-layer model. We had the model in mind when we coded our game. Basically the 3-layer model consists of:

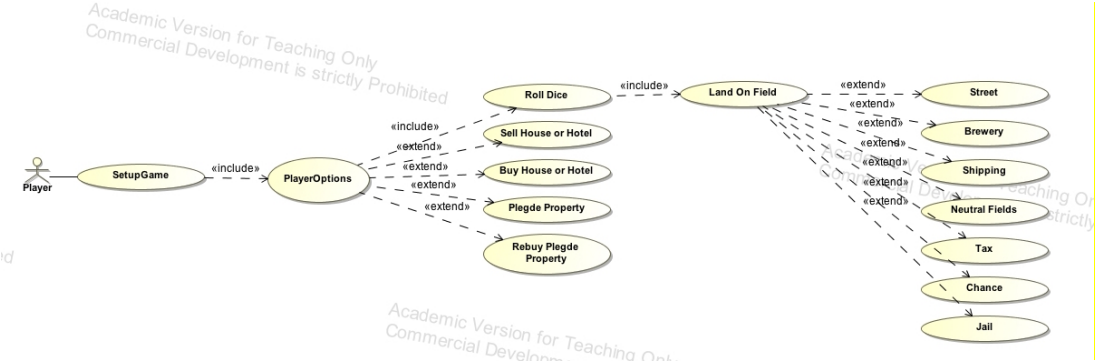
1. User Interface
2. Functionality
3. Data

We managed to comply with the model as a guideline for the program. Our user interface is the GUI provided in assignment, which is kept away from our data, and works through our functional classes. We did manage to implement interfaces in our SQL classes. But we excluded Java interfaces for the other Functionality and Data layers, due to a shortage of time.

Use Cases

Use Case Diagram

Use case diagrams shows the different use cases an actor can be presented for in our game. The diagram shows us how each use case leads to another use case. For a better overview we have included our use cases' extensions. We have chosen three of the main use cases to describe in detail.



Figur 1 This use case diagram describes the different actions and its relationship between the different actors.

Use Case: Setup Game
ID: 1
Brief description: The players choose either to start a new game or load a saved on. If a new game a chosen, the amount of players and names are required to proceed.
Primary actors: Players
Secondary actors: Database
Preconditions: Players has access to a computer in the DTU Databar.
Main flow: 1. Start new game. a. Enter amount of players. b. Enter the names of the players. 2. Load game

Post conditions:
Alternative flows:

Use Case: PlayerOptions
ID: 2
Brief description: This is main use case in the game, every turns starts here. And any invalid input in the main flow, returns the player to this use case. The player gets 5 different options to choose between. “Roll Dice”, “Buy House(s) or hotel”, “Sell House(s) or hotel”, “Pledge Property” &”Buy Property”. Each choice triggers a new use case.
Primary actors: Players
Secondary actors:
Preconditions: Player has setup the game.
Main flow: 1. Roll Dice a. The game rolls the dice and moves the player. b. Triggers Land On Field Use case. 2. Buy House a. Choose property group (color). b. Choose amount c. Place houses. 3. Sell House a. Choose property group (color). b. Choose amount c. Place houses. 4. Pledge property a. Type in the Field number you wish to pledge. 5. Rebuy pledged property a. Type in the Field number of the pledged property you wish to buyback.
Post conditions:
Alternative flows:

If the player is jailed a special turn with limited options is presented to the player:

1. If the player has a 'Get of Jail' chance card, afterwards his turn ends, he's able to use it and on his next turn the player is presented with main flow of Player Options.
2. The player can pay a fine of 1000, and end his turn. On his next turn he is presented with the main flow of Player Options.
3. The player gets 3 tries to roll a pair, if he's lucky and rolls it. He will move the eyes of the dice, and he's immediately given an additional turn (main flow of Player Option). If the player doesn't manage to roll a pair on the three turns (9 roll dices total), he's forced to pay the 1000 fine and pass the turn.

Use Case: Land On Field

ID: 3

Brief description:
The Land On Field is the method to retrieve the different conditions that each field represent from the field subclasses.

Primary actors:
Players

Secondary actors:

Preconditions:
Player has rolled the dice.

Main flow:
There are 5 different field types, which has different values and attributes.

- Street - Ownable field. If the field has no owner, the player can purchase the field. If the field already has a owner, the player must pay a fixed rent to the owner. When all Streets are owned by the same player, the rent is doubled and he'll be able to buy houses and eventually hotels, which drastically increases the rent.
- Brewery - Ownable field. If the field has no owner, the player can purchase the field. If the field already has a owner, the player must pay a variable rent to the owner depending of the sum of the dice roll. If both Breweries are owned by the same player, the rent is doubled on both fields.
- Shipping - Ownable field. If the field has no owner, the player can purchase the field. If the field already has a owner, the player must pay a rent to the owner which depends on the amount of Shipping fields owned.
- Tax - There are 2 tax fields, one with a fixed tax and another which gives the player a choose between paying a fixed tax or 10% of his total balance.
- Chance Card - The player receives a bonus.

• Jail - The player lands on jail, and is directly repositioned to the jail. The player is now in jail, and will be given the alternate flow of Player Options. • Neutral fields - The player can land on 3 different neutral fields. Which has effect on the player.
Post conditions:
Alternative flows:

State Machine Analysis

A state machine is used to control different states an object can enter. It's used to give an overview over which state and actions are used within the game. Every state within the state machine has one or several assigned actions which can be executed. Every action executed can lead to the same or another state depending on what impact the action has to the given state.

The following tables are made to give an overview of which states a player or field can enter and what their significance are, and furthermore which actions can be made within the state machine and a definition of the actions.

State/condition and actions overview for Field.			
State	**Significance**	**Actions**	**Significance**
0	Player is ready to be initialized.	O	Creates Player
1	Player is initialized and waits for name.	N	Player gets a name.
2	Player wait for his turn.	T	Player gets his turn
3	Player's turn.	ST	Player ends his turn
4	Player is bankrupt and is removed from the game.	B	Bankrupt
e	Error state	K	Unacceptable action

State/condition and actions overview for Player.			
State	**Significance**	**Actions**	**Significance**
0	Field waits for a owner.	B	Fields is now owned
1	Field is owned.	H	Field now has houses or hotels on it.
2	Field is owned and have houses.	S	Field subtracts houses.
3	Field is pawned.	SA	All houses removed from field.
4	Field is neutral	P	Field is pawned.
e	Error state	PB	Pawned field is bought back.
		N	Owner of the field is bankrupt.
		K	Unacceptable action

The state tables overview below shows the actions impact on any given state and what state it leads to. The definition of the symbols can be found in the tables above.

State Machine Table Overview

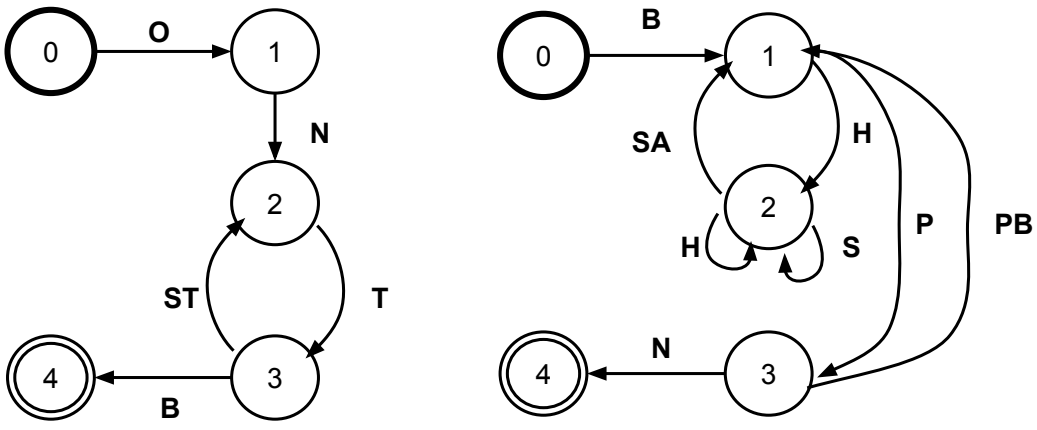
Player table overview of Actions state consequence						
	O	**N**	**T**	**ST**	**B**	**K**
0	1	e	e	e	e	e
1	e	2	e	e	e	e
2	e	e	3	e	e	e
3	e	e	e	2	4	e
4	e	e	e	e	e	e

Field table overview of Actions state consequence								
	B	**H**	**S**	**SA**	**P**	**PB**	**N**	**K**
0	1	e	e	e	e	e	e	e
1	e	2	e	e	3	e	e	e
2	e	2	2	1	e	e	e	e
3	e	e	e	e	e	1	4	e
4	e	e	e	e	e	e	e	e

The state machine diagram and the state machine table are closely related. But the state machine diagram only shows positive actions in the different states.

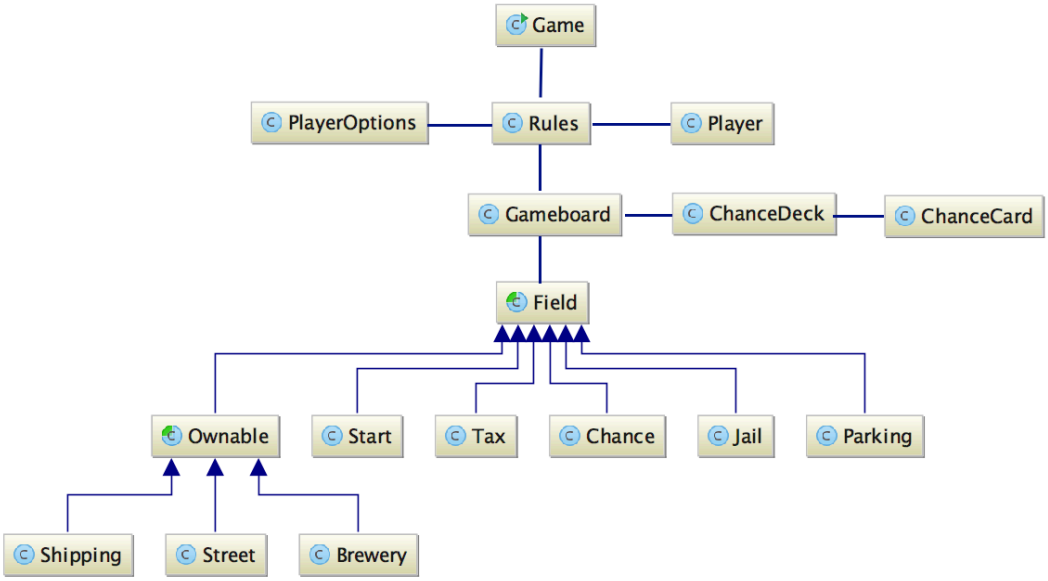
Both state machine diagrams show us the different states and actions that can be executed within them. Both state machines are deterministic, because there are no actions within a state that when executed leads to the same state.

State Machine Diagram, Player and Field



Domain Model

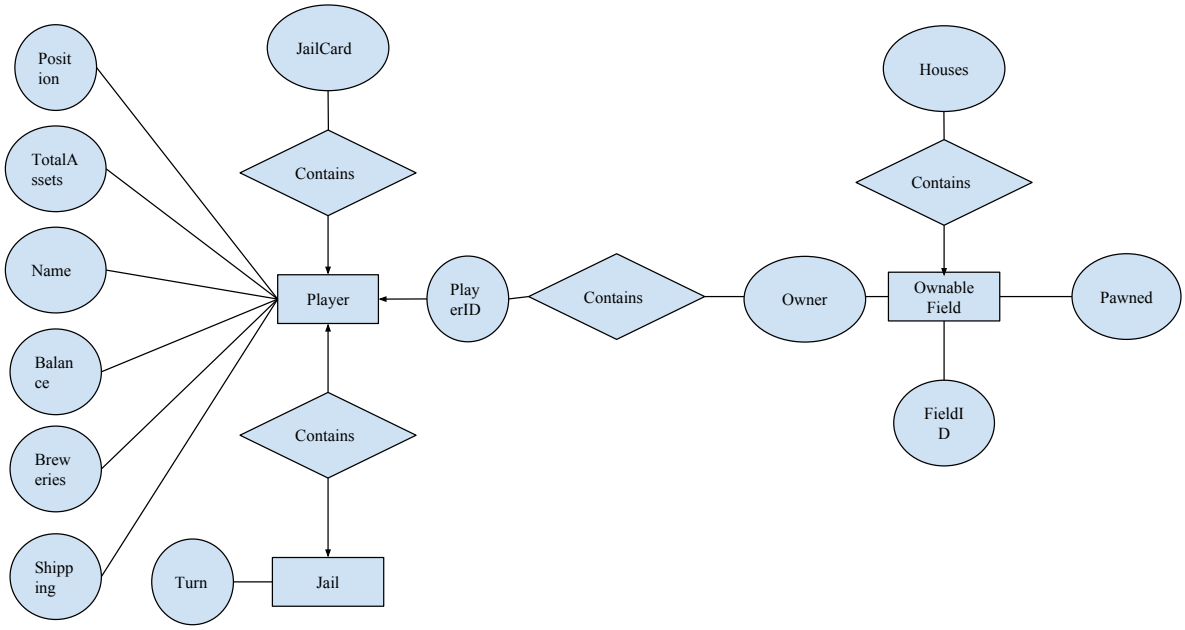
The Domain model describes the connection between the different classes and how they cooperate. This model is very meaningful for the project, since it provides us with knowledge of how the different classes interact with each other.



Entity-Relationship Diagram

Our ER diagram shows our entities, their attributes and their relations.

Our primary keys are: PlayerID for player and FieldID for field. The owner references the Field to the Player. The cardinality is Player to many → Fields. Because the Player can own several fields. But the Field can only contain one owner.



BCE Model

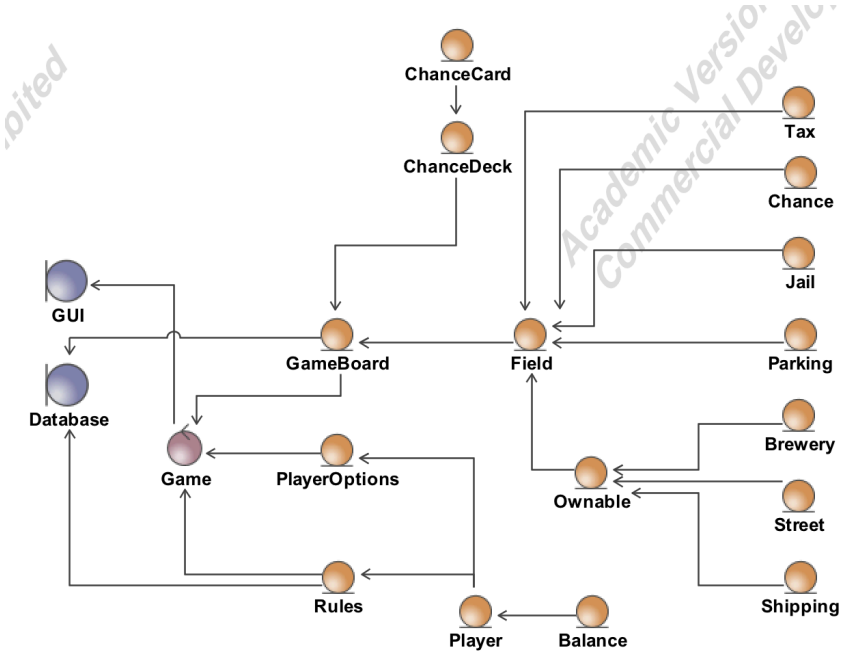
The different program classes can be categorized in three different types of classes. The different types are called the Boundary, Control, and Entity class.

- Boundary classes are those classes, which communicate with the system's surroundings.
- Control classes, share and gather the responsibility between other objects and determines the sequence in different actions.
- Entity classes, is the information which is used by the Boundary classes and shared with the control classes. These classes are the memory where the data is stored.

In our program we have two boundary classes, the GUI and the Database.

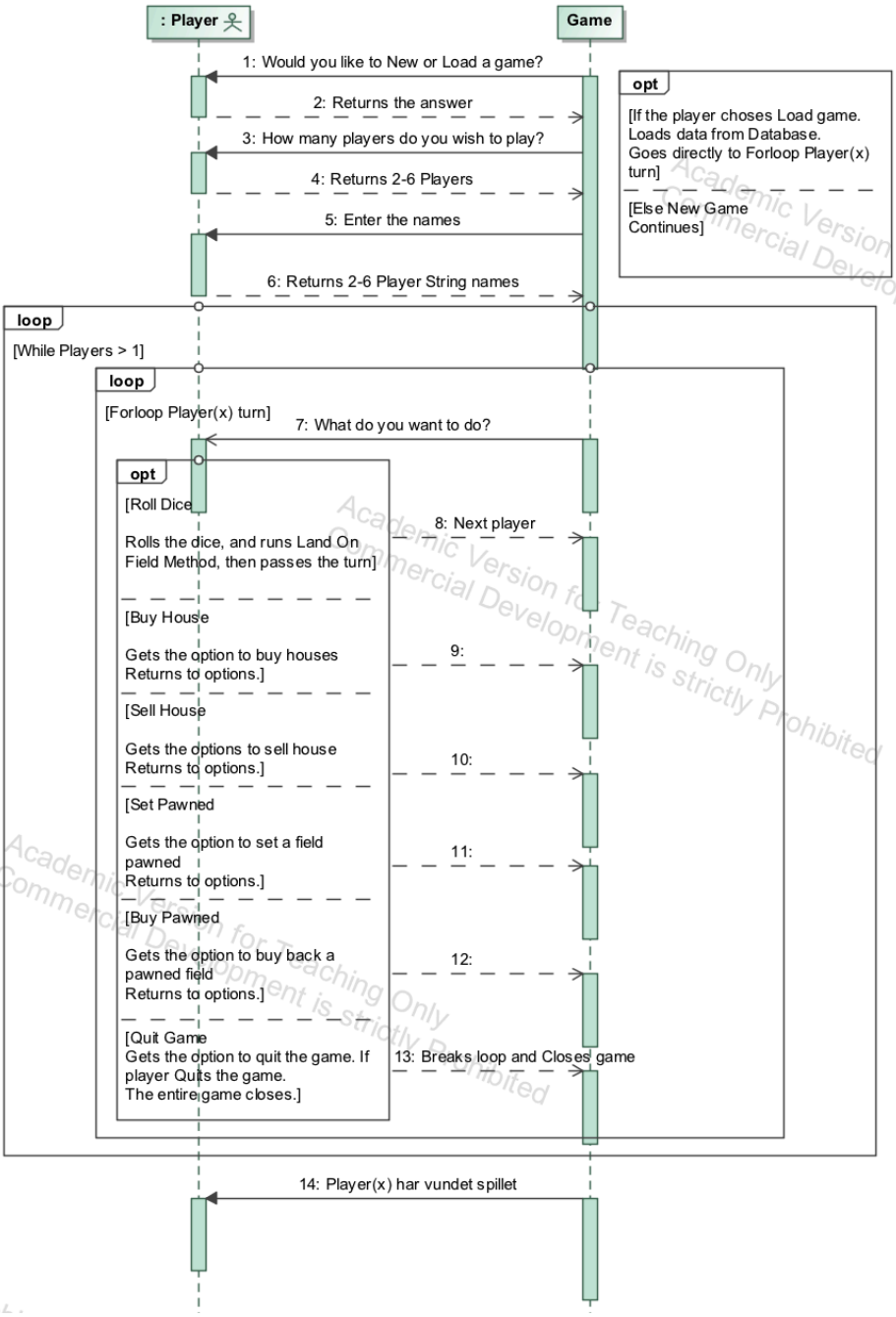
Our system exists of only one controller the Game class. This class have the control over every class and their methods. Our turn method that's basically the whole game is run from the Game class.

We have chosen to rank the responsibilities of the classes. So that the entities that are closets to the controller are the once with the most responsibility and the once furthest away are the once with the least.



System Sequence Diagram

The System Sequence-diagram is used to show the interactions between the actor and the game. In our case the SSD shows the scenario of creating and playing and entire game. All the actions are done by the actor in our scenario. The SSD shows all the use cases that the actor is given in our Matador game. Each time the actor triggers a use case the SSD shows us how the systems reacts and responds to the specific use case.



System Design and Implementation

Implementation

We based our assumptions and decisions on balancing the game and ultimately provide the most cohesive and functional game. In some of the the following highlighted topics we explain how we have chosen to implement the rules and other requirements. Our decisions are based on the limitations with the supplied GUI and shortage of time.

GUI: We faced multiple limitations with the GUI. We would've liked to add more information to the player, such as previous turn information, detailed information about properties owned and a specific property rent overview.

Rent: With houses and hotels added to a field the GUI couldn't update the rent in the centered box. This is only a graphical restriction, and this error have been reported to Ronnie. He has confirmed that the error would be fixed in newer version of the GUI.

Bank/Balance: In the description of rules. We are told that each player starts with a balance of 30.000. Since we aren't limited by psychical bills. We have removed the bank limitation from the game.

Auction: According to the rules. If a player lands on a field which are not owned by anyone and he and decides not to buy it, the field will go on an auction.
We've decided if the player doesn't want to buy a field. The field will stay purchasable. This means that an auction between the players aren't implemented.

Trade of properties: To keep the game fluid and limited GUI we've chosen not to implement the feature Trade of Properties. But instead instead pawned fields can be purchased if a player lands on the field.

Chance cards: If player lands on the Chance field and draws a Chance Card, it should be placed in the back of the card deck. We choose to handle this in a more unpredictable way. When all 33 cards have been used and each time a game is loaded through our database, we use a shuffle method. Which randomly places all the Chance cards in our Chance deck. If we had time, we would've like to save the state of the Jail cards.

House/Hotel limitation: We have removed the limitation of 32 houses and 12 hotels in the game.

Jail card: The GUI can't show if a player owns a jail card. If a player is thrown in jail, while it has a jail card in its possession, the GUI will ask if the player would like to use it.

Houses: If player buys houses on a group of property, the rules tell us to share the spread the houses evenly. We decided to make an automatic algorithm that prioritizes

the most profitable combination We did this to sustain a simple game that fits the limited GUI.

Error: We experienced exceptions with GUI, that we couldn't catch from our class. We spoke with Ronnie about this, and he told us that the problem would be fixed in a newer version of the GUI. The exception occurs when letters, symbols or other illegal numbers given within the statement are written in the 'GUI.GetUserInteger' method. Never the less, the exception doesn't crash the game.

Input validation: All user inputs are validated i.e. when the player types its name. The name is processed by multiple checks, before being accepted. We could have used Regex as our method to validate the strings. But we overlooked the requirement and used our prior method for name validation.

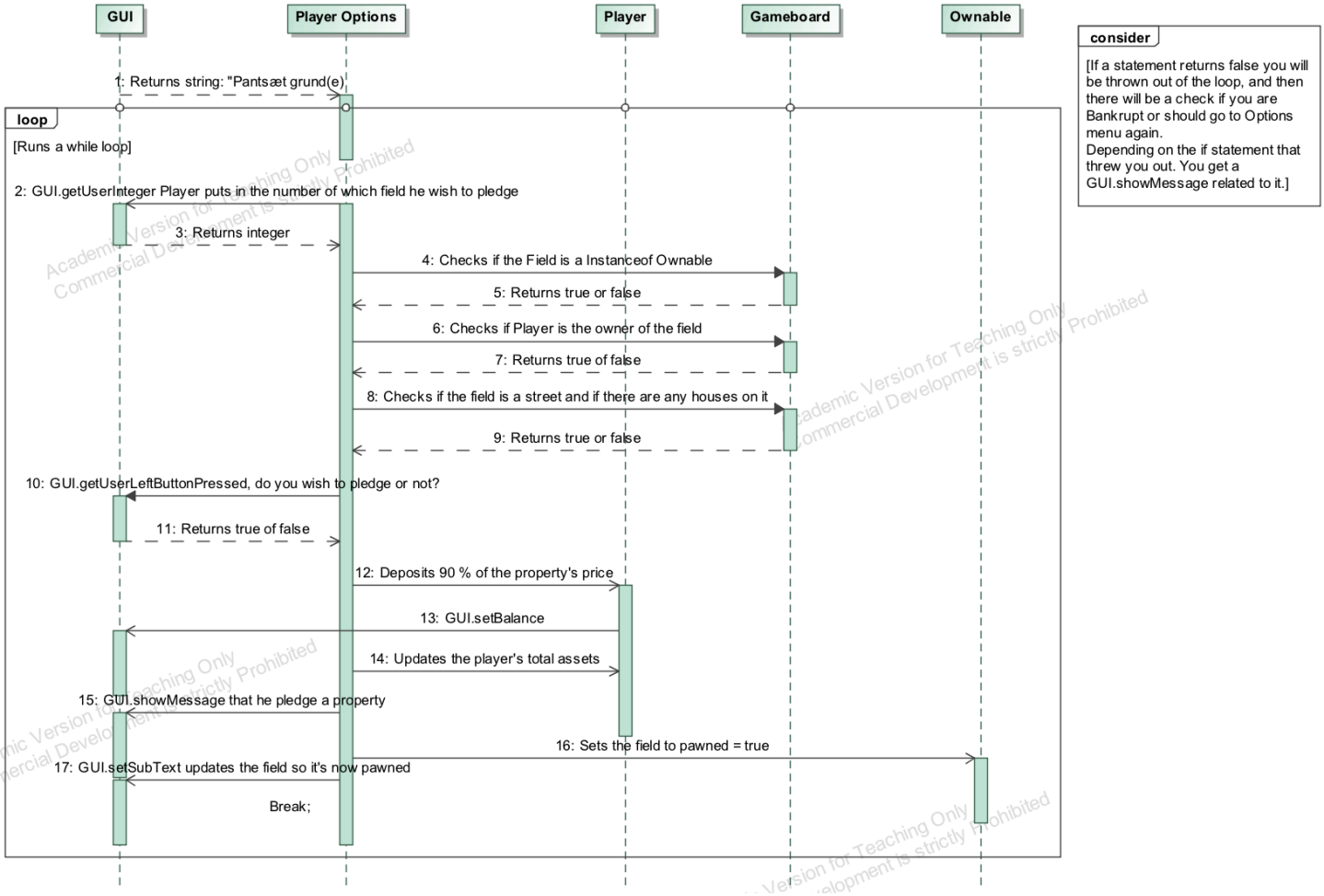
Design Sequence Diagram

The Design sequence-diagram and the System sequence-diagram are bit different from each other. The System sequence-diagram focuses on how the actor and the system interact. The Design sequence-diagram focuses on our methods, and how the classes interact with each other.

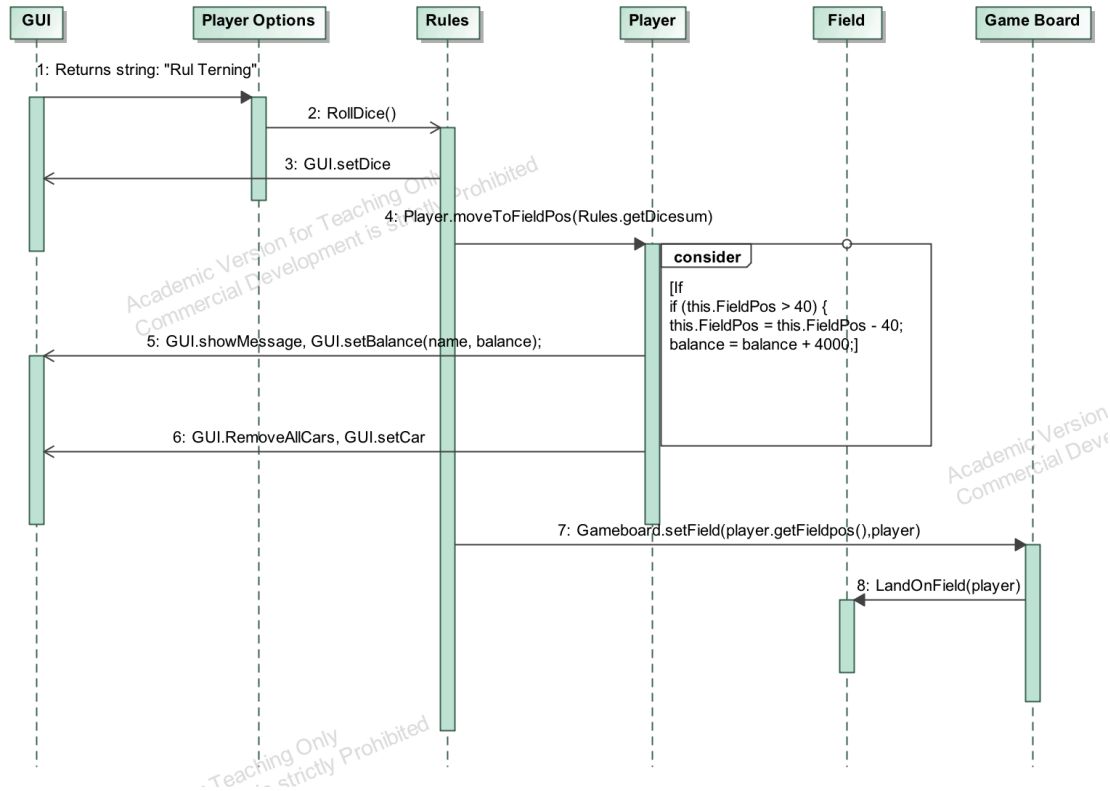
We analyzed how we wanted the game to run and what options in the menu that the player should be able to pick from. Based on the analysis we made three design sequence-diagrams, Set Pawned, Roll Dice and Buy Property. These design sequence-diagrams helped us understand how we wanted to classes to interact and from our design sequence-diagrams we were able to code the different options.

Our three design sequence-diagrams below all starts from the Options method. The player is given six options. Three of these six options starts one of the design sequence-diagram. In the diagram it's shown which option triggered the method.

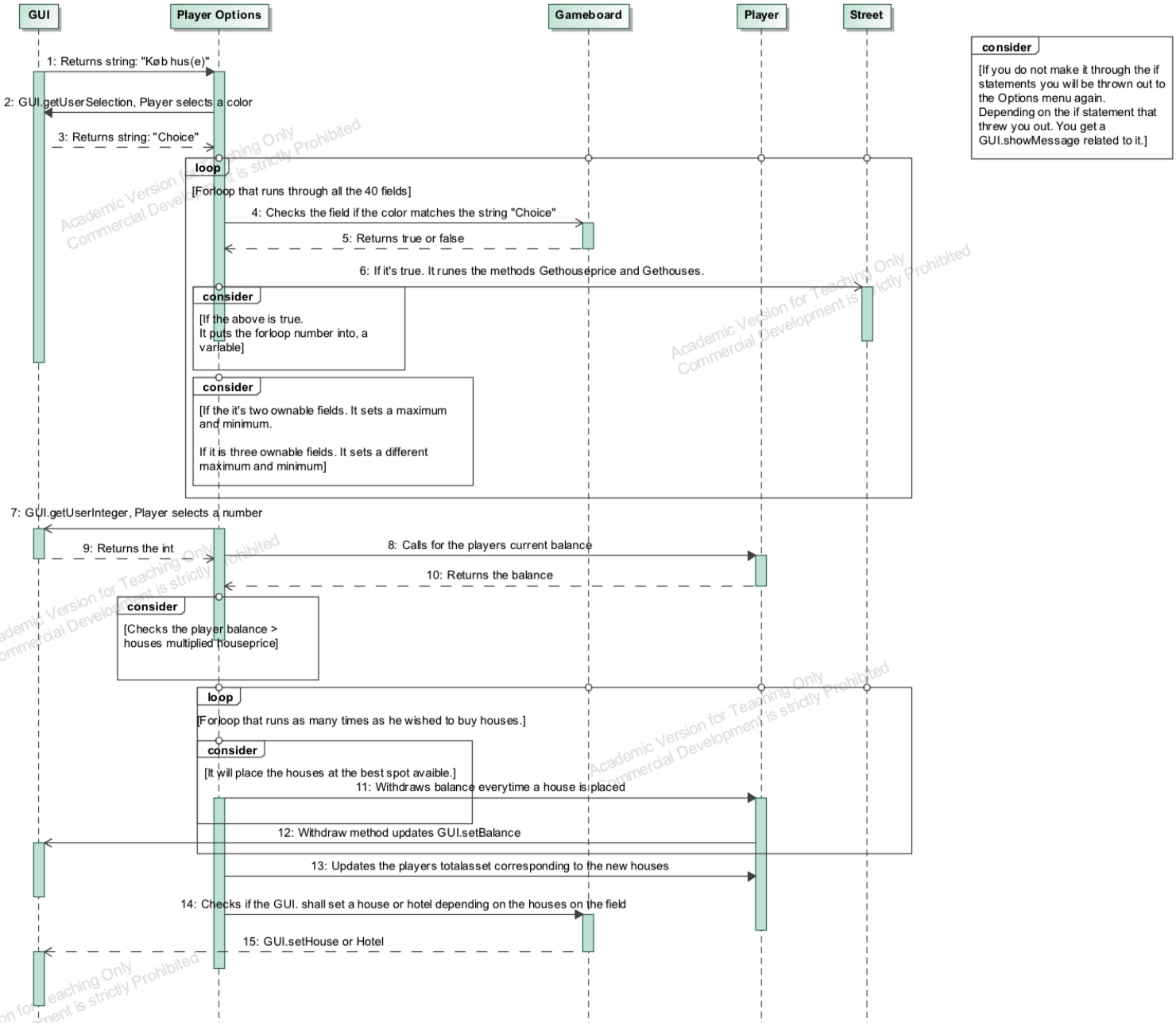
Design Sequence Diagram, Set Pawned



Design Sequence Diagram, Roll Dice



Design Sequence Diagram, Buy Property



Design Class-Diagram with Named Relations

The classes in a class diagram represent both the main objects, interactions in the application and the classes to be programmed.

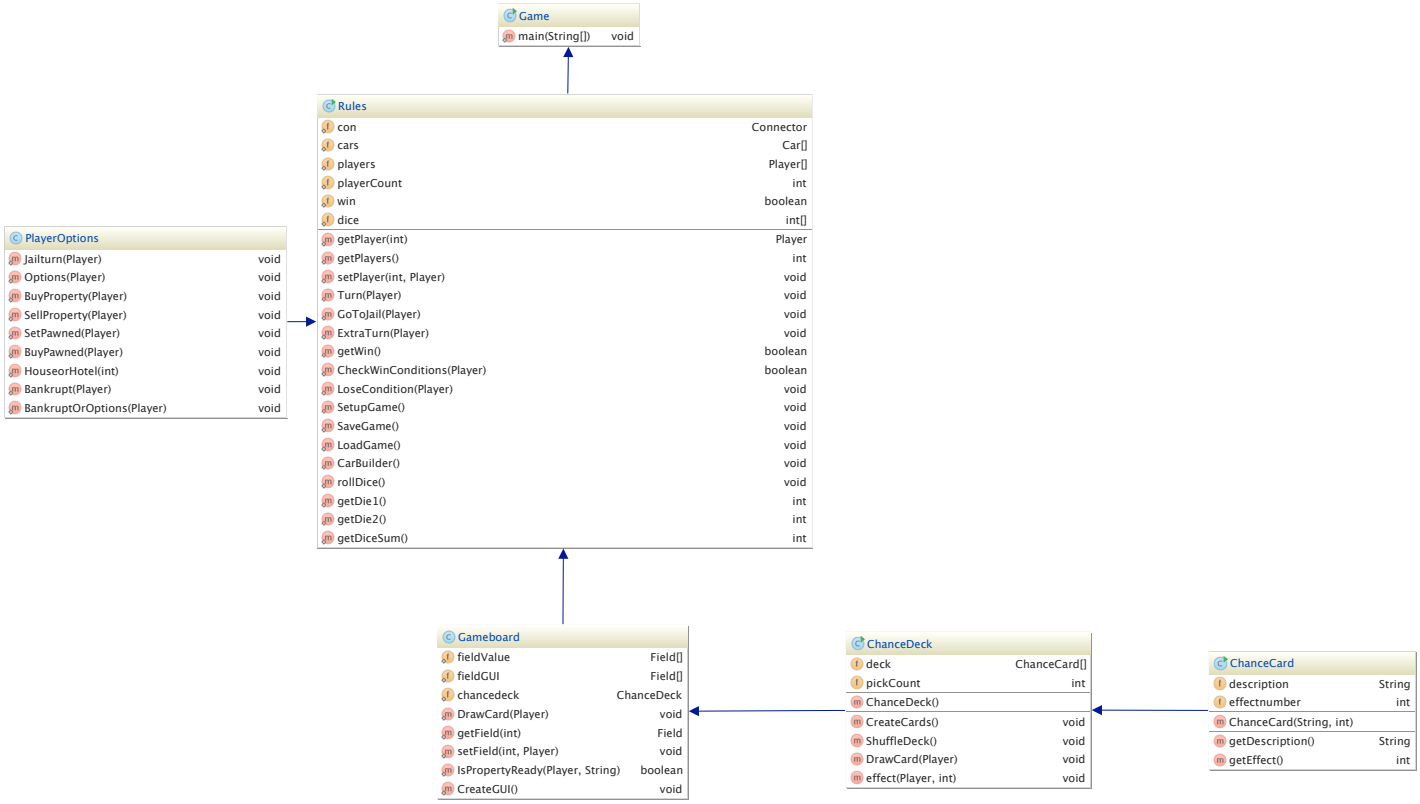
In the diagram, classes are represented with boxes, which contain three parts:

The top part contains the name of the class.

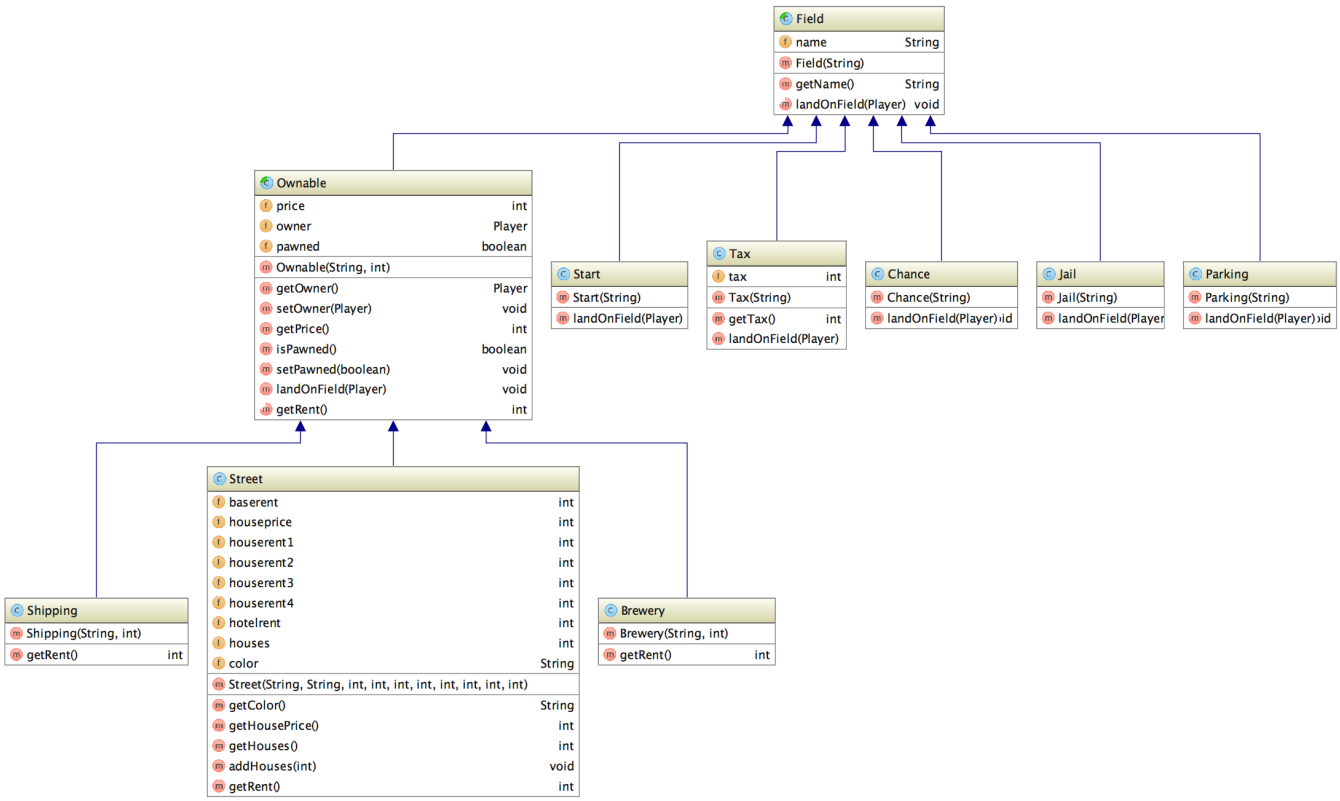
The middle part contains the attributes of the class.

The bottom part contains the methods the class can execute.

The upperpart of our Design class-diagram of the Matador project



Lower part of our Design class-diagram with the field and subclasses



Normalization

Normalization is the process of breaking tables down into smaller tables, and thereby reducing redundancy and allowing easy retrieval of information.

Normalization is non-destructive and tables can be joined again if necessary.

Normalization forms are additive, meaning to reach a normalization of 3NF it must also live up to the requirements in 1NF and 2NF.

In practice, 3NF is the most used normalization form, which are reached when:

- 1NF: Every tuple contains exactly one value for each attribute.
- 2NF: Every non key attribute is irreducible dependent on the Primary Key.
- 3NF: Every non key attribute is non-transitively dependent on the Primary Key.

Due to time pressure, we weren't able to normalize our tables.

But to reach 3NF, we would have to split up the Field table, into at least 2 smaller tables. To live up the requirement under 1NF. This is due to our Field tables contains are column 'Houses', which is only used by Fields of the type 'Street'.

Views

A view is a virtual relation. They are pre-defined select expressions, which can join tables, hide columns and a lot more, to show only the data that is needed. A view is non-destructive, meaning they cannot manipulate or destroy data, it can only show data from the tables.

In our project we haven't used views, as we have a very simple database design, which doesn't require any special select expression to show the relevant data.

Stored Procedures

A stored procedure are functions and procedures saved on the database server. It provides the ability to store complicated queries on the database server, or repetitive procedures to be run from the server, instead of being parsed by a program, or triggered manually. It also provides some security, as the inner workings of the tables doesn't have to be known to run the procedures. A trigger could simply be a fixed time interval, or when tables are updated.

In our project we have used Prepared Statements instead of Stored Procedures. We could just as well have used Stored Procedures, which in theory should've increased the performance.

Transaction Logic

Transactions provides reliability and security of the data in the database.

Transactions groups all the readings and writings records into one transaction, if any of them fail. All commands are rolled back. And therefore secures the database against errors and failures.

ACID is an acronym for the 4 requirements for a transaction.

- Atomicity: Ensures all commands are reflected in the database, or none are.
- Consistency: Ensures the database is initially consistent.
- Isolation: Ensure concurrently executing transactions are isolated from one another.
- Durability: Ensure once a transaction has been committed, that transactions updates do not get lost, even if upon a system failure.

Data Access Layer

JDBC

JDBC stands for Java Database Connectivity. JDBC is an application programming interface (API) for the programming language Java, which provides access to relational databases or other persistent storage. Most database vendors have developed a JDBC driver for their particular Database system, like MySQL, Microsoft SQL or Oracle Database.

We have used MySQL's JDBC driver for our project.

The JDBC contains 3 steps to define or manipulate data on the database.

- The JDBC driver, which has to be loaded into the Java Class.
- The connection to the relational database, which is initialized thru the DriverManager.
- Statement, Prepared Statement or Callable Statement (for Stored Procedures). Which contains the actual database queries or commands.

When querying the database, the JDBC returns the data into a resultset, which then has to be processed.

If a database operation fails, JDBC raises an SQLException. When you call JDBC statements, it requires you to catch potential SQLException in the code, to avoid the program crashing by having the Exception raised to the OS layer.

Another alternative to JDBC is the ODBC.

ODBC stands for Open Database Connectivity and provides much the same as JDBC, but is platform independent. If the programming language was i.e. C++, ODBC would be the choice. But for Java the JDBC simply outperforms the ODBC.

DAO Pattern

The DAO pattern is used to separate low level data accessing API (Application Programming Interface) or operations from high level business services. The DAO pattern consist of:

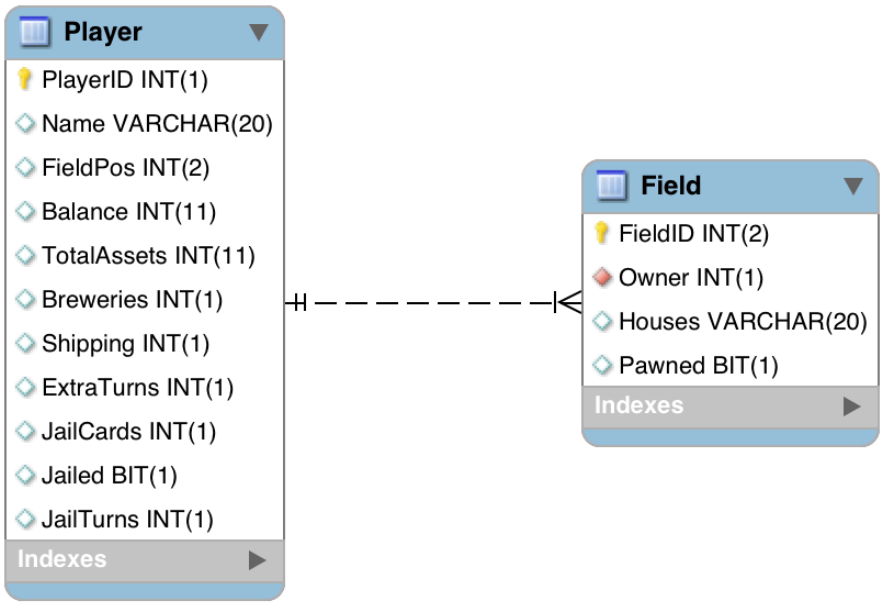
The DAO implementation, which is an abbreviation for Data Access Object. The DAO object is responsible for the access and manipulation of data in the relational database. When using DAO classes, it is recommended to create an interface for the class, which defines the standard operations to be performed on the objects.

DTO is an abbreviation for Data Transfer Object. DTO is used to transfer the data between classes and modules of your application. DTO helps limit the number of methods and calls made to the database. The alternative to DTO is using a lot of parameters, which limits the code readability and potentially over complicates the coding.

Database Structure

Enhanced Entity-Relationship Diagram

The EER diagram shows the tables' attributes and its limits furthermore the relations with other tables. The primary-, foreign keys and cardinalities are also shown in the diagram.



Testing

Testing Strategy

Testing was a crucial part of our project. While making the project we used black- and white box test tools for testing.

In our black box testing, we did several manual positive- and negative tests. This was done as a routine every time a new method was added or changed. When we were finalizing the game. We performed extensive manual tests of the game.

We used white box testing as a tool to test our methods. We used Junit tests to run our white box tests in our projects. We created white box tests of methods that we found relevant. The Junit tests are easily tweaked to fit other similar tests.

Manual Testing

Besides from our Junit test. We have done a lot to make the game as secure when it comes to handling the actors input. We used the GUI to restrict the actor's choices. Each time we implemented new method to handle a certain scenario. We would do a positive and negative manual test of the specific scenario.

An example of our manual test securing can be seen in the 'BuyProperty' method. We have designed the method to calculate a minimum and maximum value dependent on the player's current house count. These variables are used within the 'GUI.GetUserInteger' method to secure. When the player is asked to fill in the desired house amount. If the choice contradicts the houses he is able to buy the GUI will not accept the input.

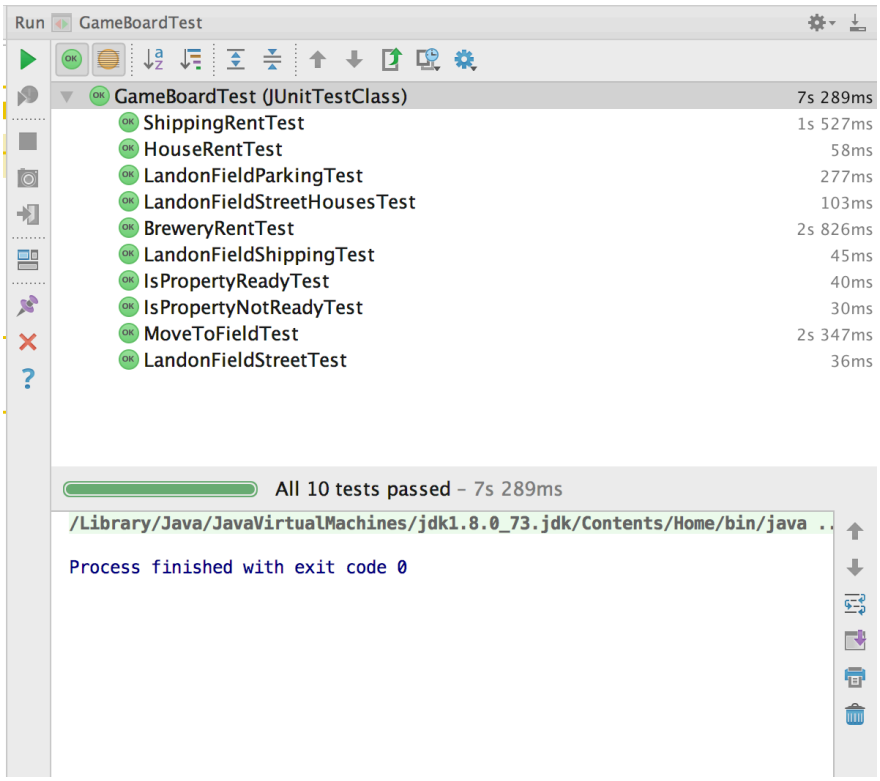
Junit Test

Junit is a regression testing framework which we've used to implement unit testing in Java and increase the quality of our projects code. We wanted to secure that our methods within our project works correctly and without throwing any exceptions that are not handled.

We have made three different Junit test-classes, GameboardTest -, PlayerTest – and PlayerOptionsTest-class. Within these three Junit test-classes we have picked out the mostly used methods. Whenever we have been making a change to our source code. We would run the Junit test-classes.

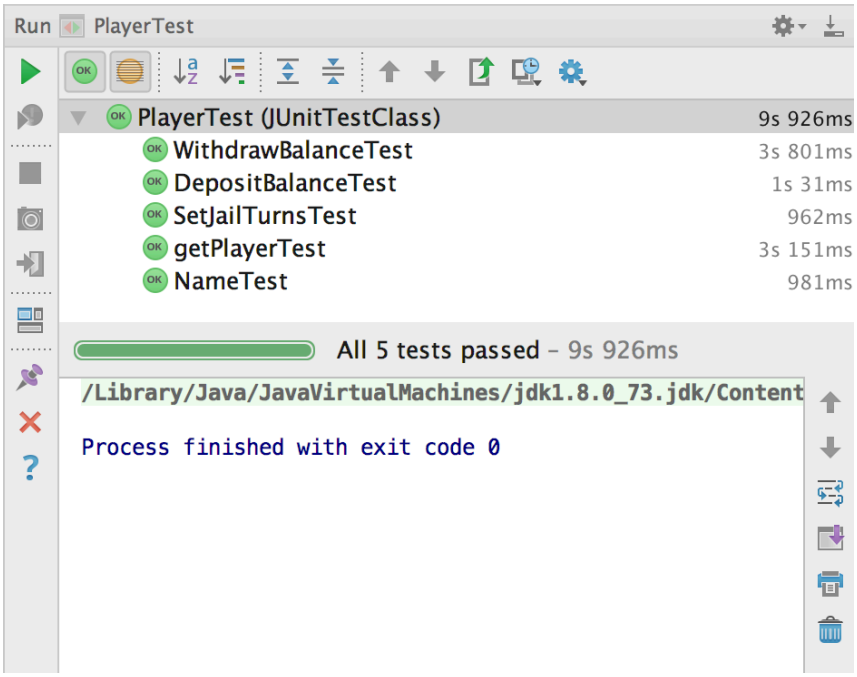
Gameboard Test

The GameboardTest-class handles basic Gameboards tests such as our 'HouseRent' method and 'LandOnField' method. The Junit test runs without any player input.



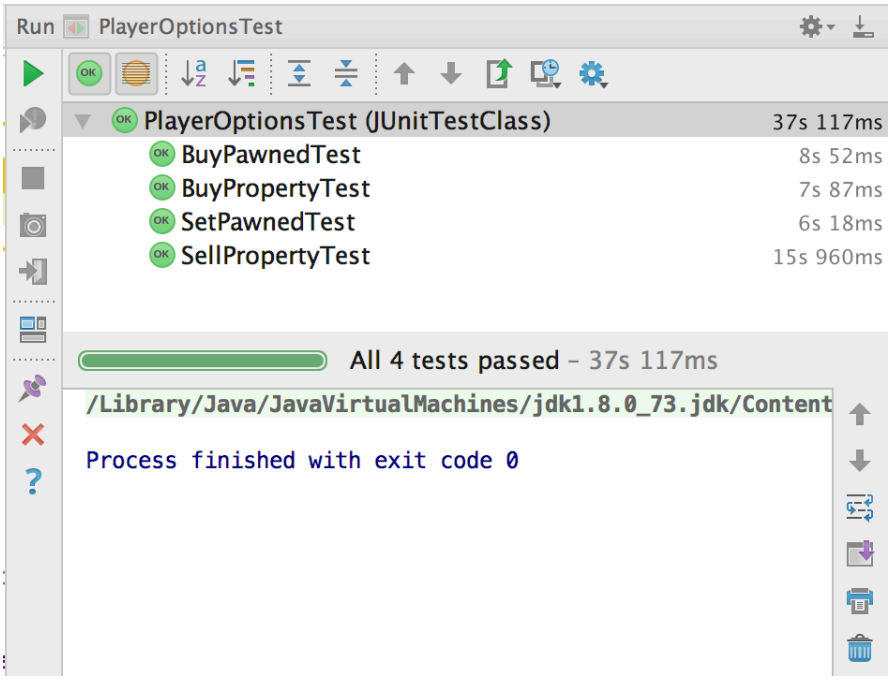
Player Test

The PlayerTest-class handles Player tests and a database test. Methods that are tested in the Junit test can be viewed on the picture below. The Junit test runs without any player input.



Player Options Test

The Player Option Junit test is different from the two other Junit test-classes. As the PlayerOption test requires the input of an actor. For the test to run successfully the actor most run the prewritten conditions and choices. The Junit test can therefore be executed to run as a positive - or negative test.



As a conclusion for our Testing section. We have prevented exceptions through input restrictions. And we have therefore been able to make a reliable and stabile program.

Conclusion

With help from our analysis, we have managed to code a board game which is very close to the original Matador Game, and even improved it, by removing some of the restrictions and limits, associated with the board game.

This game is much more comprehensive than our earlier projects that were related to the Matador game. In our analysis section we discussed our assumptions and decisions that we dealt with to make the game and its functionality as simple and fluid as possible within the period of time we had to produce the game.

In our analysis process we were obligated to take the GUI into consideration regarding methods and functionality. Some of the limitations are mentioned in our implementation sector on page 17, but one of the major concerns was the lack of game information for the player. It was difficult to give each player visual information that were easily accessible. We would have preferred a general status area for each player with information about owned and pawned fields and the rent based on the properties.

Since the last project our group has been reduced from 6 to 3. There is many pros and cons with this decrease of members. Earlier we found it very difficult to involve all group members in i.e. coding. But in this project all 3 members have been involved with all aspects.

We look forward to more experiences with MySQL. We learned a lot while writing the Database section, and we can already see numerous ways to streamline the interaction between Java programming and the database.

We have experienced that manual testing throughout the project have been very time demanding, and in the future we will definitely develop more Unit tests as we develop the project.

The lectures in software development have given us a deeper understanding of the different states our objects enters and how actions can change their state throughout our game. This was an important lesson to help us further develop our programming skills.

Appendix

Game Rules

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 betale bøden, 1.000 kroner, og flytte, som øjnene viser.

Den fængslede har ret til at købe grunde osv.

Huse.

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

Der skal bygges jævnt, dvs. det første hus kan man opføre på hvilken grund i gruppen, man ønsker; men inden hus nr. 2 opføres på en grund, skal der være bygget eet på hver af de andre grunde i gruppen osv. Og *inden man opfører et hotel, skal der være fire huse på hver grund i gruppen.* Der må kun bygges eet hotel på hver grund. Når man køber et hotel, afleverer man de fire huse til banken.

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

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

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

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

Pantsætning. Man kan kun pantsætte sine ubebyggede grunde etc. til banken for det beløb, der står påtrykt bagsiden af skøderne. Har man bygninger på grundene, skal man først sælge disse til banken. Spilleren beholder skødekortene, men vender bagsiden opad. Renten er 10%, der betales sammen med lånet, når pantsætningen hæves.

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

Af pantsat ejendom kan der ikke kræves leje.

Banken giver kun løn mod pantsætningssikkerhed.

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

Spillerne må ikke låne indbyrdes.

Glemmer man at kræve leje af en medspiller, har man tabt sin ret, når nr.2 efter ham har kastet.

Configuration

Installation

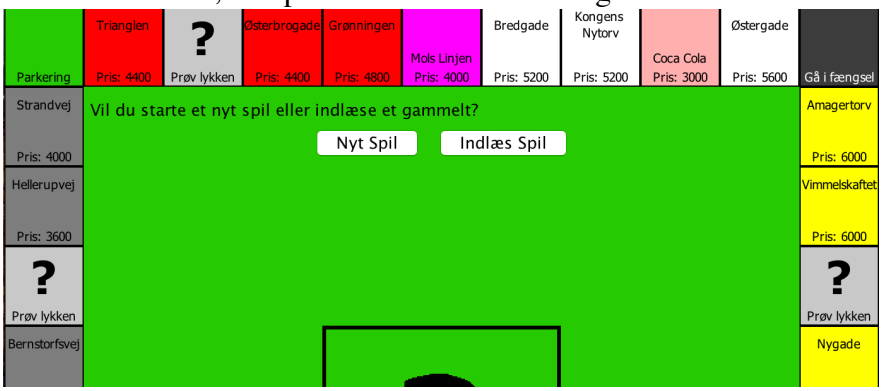
Windows:
1: Unzip the folder Matador_Gr02.zip
2: Open command prompt and run the following command: Java home path/bin/javaw.exe" -jar file path
3: Select New game or Load game
4: Play

Mac:
1: Unzip the folder Matador_Gr02.zip
2: Open terminal and write following command: Java -jar file path
3: Select New game or Load game.
4: Play

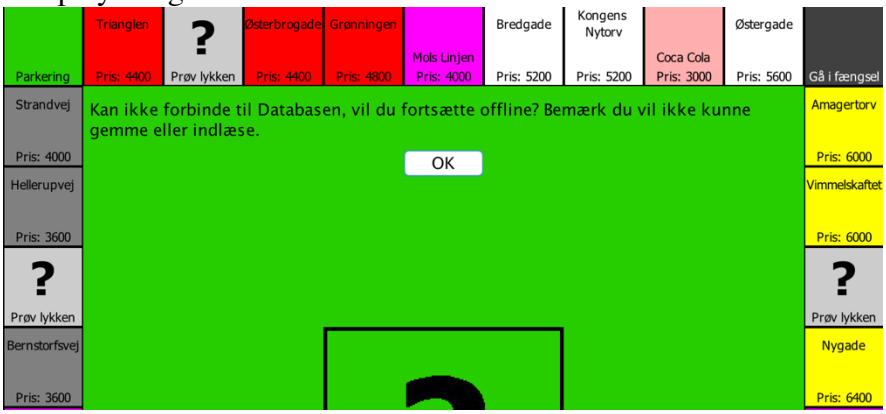
User guide

When game is launched player input is required to proceed in the game progress. Here is short game walkthrough, that will show to show you the functionality and possibilities in the game.

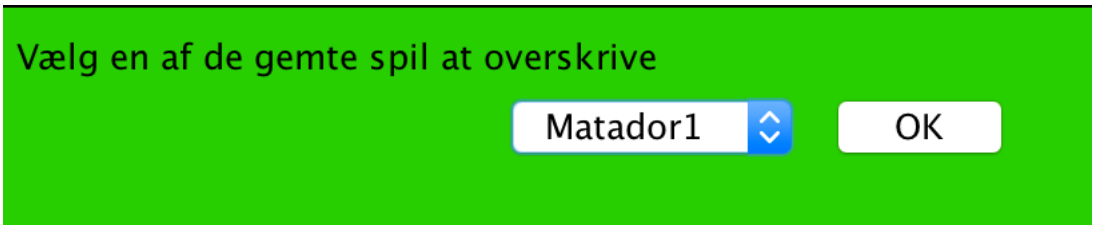
1: This is the Setup options for the game. Since this game connects to a hosted database service, it is possible to load a saved game.

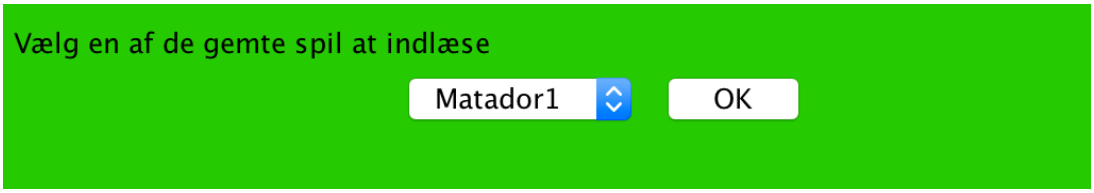


2: If you aren't connected to the internet, this message will be shown. If you press ok, the program will continue to progress 1, where it will be possible to select "Nyt Spil" and play the game without the save function.

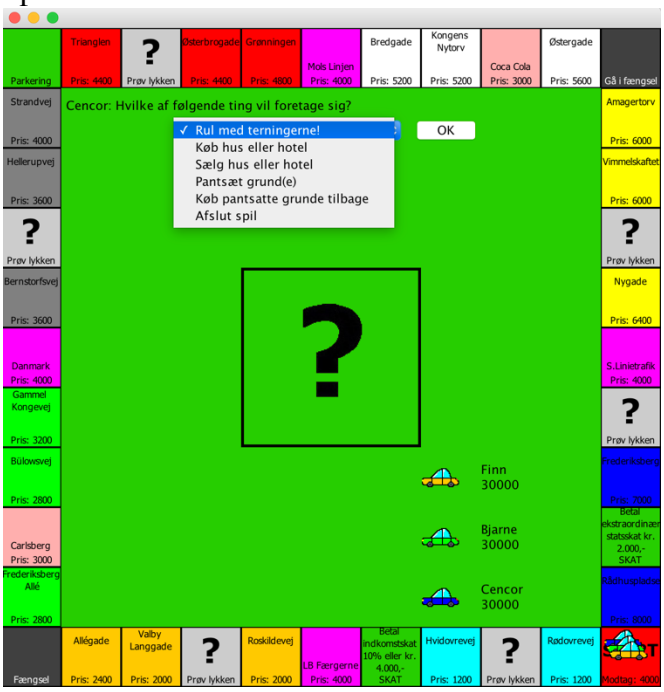


3: Setup Continued: If you select ”Nyt Spil”, you will be asked to overwrite one out of five exisiting games in the database, before the game starts.
If you select ”Indlæs Spil”, you will be asked which game you would like to load from the database.

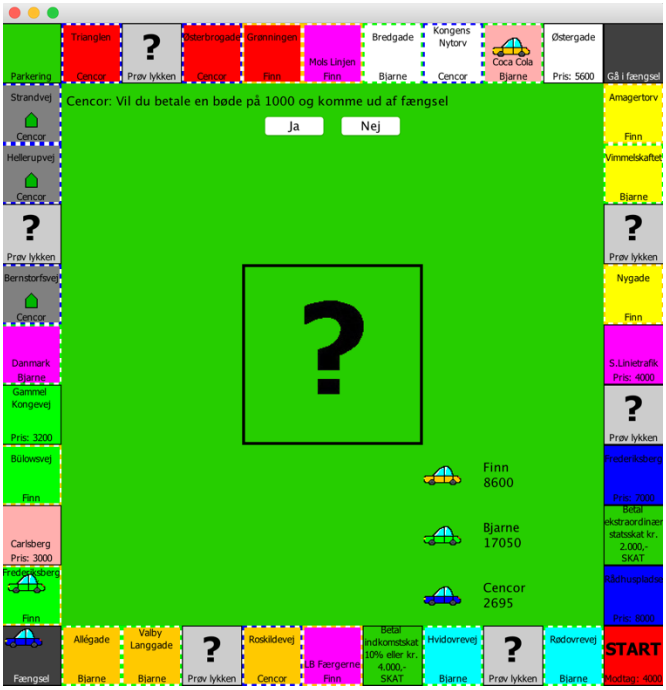




4: The game will now start. This example is new game. At this picture we see how 3 players have been added to game and the first player will be presented with some options.



5: In this example we see a loaded game from database



Javadoc

Javadoc is used to run over the code to make documentation of the classes. Documentation is important in coding, for other people to understand your code. Another advantage is when you look back on an old project. It will be manageable because your code is explained by Javadoc.

Javadoc contain tags and general comments which are implemented in I a HTML file.

An example:

The blue text which are surrounded by /**text */ is the comments and tags made by the developer. In this example the Javadoc is made to a Constructor.

```java
/**
 * The player constructor is used in the Rules class where numbers between 2-6 player constructors are selected
 *
 *@param name The name of the actual player
 *@param balance The balance of the actual player
 *@param TotalAssets Value of all house, hotel and properties. Used for specific chancecard
 *@param FieldPos Players position on the gameboard
 *@param breweries Number of breweries
 *@param shipping Number of owned shipping properties
 *@param jailcard Number of jailcard in player possession
 *@param Jailturns Number of many times player has been in jail
 *@param jailed Jailed boolean which it set either true or false. False as standard
 *
 * */
public Player(String name, int balance, int TotalAssets, int FieldPos, int breweries, int shipping, int jailcard, int Jailturns, boolean jailed) {
    this.name = name;
    this.balance = balance;
    this.TotalAssets = TotalAssets;
    this.FieldPos = FieldPos;
    this.breweries = breweries;
    this.shipping = shipping;
    this.jailcard = jailcard;
    this.Jailturns = Jailturns;
    this.jailed = jailed;
}
```

This picture shows the Javadoc information given in Eclipse for the Constructor.

entity.Player.Player(String name, int balance, int TotalAssets, int FieldPos, int breweries, int shipping, int jailcard, int Jailturns, boolean jailed)

The player constructor is used in the Rules class where numbers between 2-6 player constructors are selected

Parameters:

name The name of the actual player
balance The balance of the actual player
TotalAssets Value of all house, hotel and properties. Used for specific chancecard
FieldPos Players position on the gameboard
breweries Number of breweries
shipping Number of owned shipping properties
jailcard Number of jailcard in player possession
Jailturns Number of many times player has been in jail
jailed Jailed boolean which it set either true or false. False as standard

If Javadoc is made in an entire class or project, Javadoc can produce a HTML file to give developers or other involved people a detailed overview of the source code. In our project folder we have added a HTML file that represent the player class in our project.

Here are some screenshots of the Javadoc HTML file which provide information about Methods and a specific Constructor.

Methods:

Modifier and Type	Method and Description
void	**depositBalance**(int i) Method to deposit from the balance
int	**getBalance**() Return specific player balance
int	**getBreweries**() Return number of breweries
int	**getExtraTurns**() Getters and setters for extra turn
int	**getFieldPos**() Return Field position
int	**getJailcard**() Return Field position
int	**getJailturns**() Returns numbers of Jailturns
java.lang.String	**getName**() name to return player name
int	**getShipping**() Sets number of shipping properties
int	**getTotalAssets**() Getters for TotalAssets Return totalAssets amount
boolean	**isJailed**() Getters and setters to seek information about the constructed players Returns if player is jailed
void	**moveToFieldPos**(int FieldPos) Method to move player Checks if player has passed the start position

Constructor:

Constructor Detail

Player

```
public Player(java.lang.String name,
              int balance,
              int TotalAssets,
              int FieldPos,
              int breweries,
              int shipping,
              int jailcard,
              int Jailturns,
              boolean jailed)
```

The player constructor is used in the Rules class where numbers between 2-6 player constructors are selected

Parameters:

`name` - The name of the actual player

`balance` - The balance of the actual player

`TotalAssets` - Value of all house, hotel and properties. Used for specific chancecard

`FieldPos` - Players position on the gameboard

`breweries` - Number of breweries

`shipping` - Number of owned shipping properties

`jailcard` - Number of jailcard in player possession

`Jailturns` - Number of many times player has been in jail

`jailed` - Jailed boolean which it set either true or false. False as standard

Timesheet

Timesheet						
Navn	Design	Impl	Test	Dok	Andet	I alt
David	20	37	22	21	5	105
Hannibal	17	37	28	15	8	105
Rasmus	19	43	22	16	5	105
Sum	56	117	72	52	18	315

Word- and Symbol Explanation, Abbreviation

Abbreviation explanation:
CDIO = Conceive, Design, Implement and Operate

GUI = Graphical user interface

UML = Unified Modeling Language

BCE = Boundary Control Entity

SSD = System Sequence Diagram

DSD = Design System Diagram

DCD = Design Class Diagram

GRASP = General Responsibility Assignment Software Patterns/Principles

FURPS = Functional, Usability, Reliability, Performance and Supportability requirements.