

Project assignment 2017

02362 Project in Software development and 02327 Introductory databases

Deadline: Sunday the 7/5 2017 at 23:59 hours

Group: A

Group members:

s114750 Aleksander Wienziers-Madsen



s153063 Emil Frederik Rune Jørgensen



s165160 Ana-Maria Fischer Zokalj



s165153 Simon Christiansen



Index

Index

Index	2
Preface and introduction.....	3
Problem statement and background.....	3
Analysis and Design	4
Requirements	4
2.1 Functional requirements	4
2.2 non-functional requirements	6
Use Case	6
System Sequence Diagram	8
Domain model	9
State machine diagram.....	10
Class diagram.....	11
MVC (Model- view-controller):.....	12
Implementation.....	13
Software documentation.....	16
The design of the Java program	17
Use of JavaDoc.....	19
Database	19
Test	23
User's manual and example	24
Conclusion	25
Time card	26
Bibliography.....	26
Appendix.....	27
Appendix A – Use cases	27
Appendix B – System Sequence diagram	36

Preface and introduction

For our exam project in the courses “Project in Software Development” and “Introductory Databases”, we have been asked to finish a Monopoly (Matador) game with the possibility of saving and loading a game. We worked in a group of four, where we used unified process to complete this project. A Monopoly game consists of a board, a set of dice and between two and six players. In the game, the players roll the dice and move their tokens according to the face values to a field. Different things can occur on different fields. The player should be able to buy for example a Property, Utility or Railroad field, when they land on it and it is not already owned by another player. If the field is owned by another player, the current player should pay rent. A player also has a couple of choices, when it is their turn, for example trading with other players, buying houses and mortgaging their property. Our project’s goal is to make a simulation of a Monopoly game where these terms are met. Because a Monopoly game can last a long time, it should be possible to save the game and reload it later.

Author Ana-Maria Fischer Zokalj

Problem statement and background

Data structures

There are different requirements for this project, as expressed in the project description. For this project, we have made a program that simulates a Monopoly game. The players should be able to save and load a game. The game consists of a board, a set of dice and between two and six players. A board consists of different field types. We have designed the different data structures and classes that represent a board, a field, a player and a set of dice. We have made a UML diagram, a Class Diagram to illustrate this.

Choice of functionality

The idea is that we shall implement as many functionalities of Monopoly as possible. Because of the complexity, it may be appropriate to exclude some of the functions or set some constraints on them. We have chosen not to include the functionality of property auctions in our program, because of the limited time for the project.

Design state machine diagram

When it becomes a player’s turn, he/she has different choices, possibilities and constraints: A player that lands on a property field that isn’t owned by another player can for example buy the property field, mortgage their own property fields and buy the property field, or choose not to do anything. We have designed a state machine diagram to illustrate the player’s choices, possibilities and constraints.

Saving and loading a game

Because a Monopoly game can take a long time, it’s appropriate that the game can be saved and loaded. It is a requirement that this functionality is implemented by saving all relevant states in a local database (MySQL).

GUI

We have used the GUI, we got in the fall semester, so that the player should be able to:

1. Play Monopoly, which means rolling the dice, move their token, buy properties etc.
2. Save game

3. Load game
4. Quit game

We have used the graphic package from this fall, to show the game board etc.
(From the project description)

Author Ana-Maria Fischer Zokalj

Analysis and Design

For this project, we have used Unified Process. We have chosen to make time boxed iteration on two weeks' time where we made goals for each iteration. In the beginning we wanted to use Use Case driven development.

Therefore use cases were a central part of our project. We found scenarios and wrote many Use Cases, which gave us an overview of the different functionalities the game should contain. We ended up not using Use Case driven development, but just Unified Process. We have made system sequence diagrams for some of our Use Cases, to show how the player and the system interact. We found our requirements by reading the Monopoly rules and analyzing our use cases. We brainstormed on possible objects and classes of a Monopoly game. We did this by making a noun-analysis over which nouns we could actually use for objects and classes. We used this when we made our Domain model. We made a domain model, to show possible classes and their association to each other. We have made a Class diagram, so we define the classes' methods and attributes and how the classes are associated. This was essential for when we began coding our game, so all of us in the group had an idea of what we were coding and the overall design of the program.

Requirements

We have organized our requirements in functional and non-functional requirements; we deemed it sufficient to cover all the requirements:

2.1 Functional requirements

- | | |
|---------|--|
| RQ101: | The game shall be playable by 2-6 players. |
| RQ102: | The players move by rolling the dice and moving accordingly. |
| RQ103: | Each player shall have a fixed starting balance of 30.000 |
| RQ103: | The game shall consist of a circular-like playing field consisting of 40 individual fields. |
| RQ104: | The fields shall be of 8 different varieties, GO, Chance, Tax, Free Parking, Jail, Utility, Railroad and Property. |
| RQ105A: | The fields of the type Property, Railroad and Utility, shall be ownable by the players. |
| RQ105B: | All ownable fields shall have a fixed price, depending on its type and placement on the board. |
| RQ106: | The fields of the type Property shall, if owned, cost a fixed price if another player lands on the field, depending on its placement on the board. |
| RQ107A: | The properties shall be colour coded, with 2-3 fields in the same colour. |
| RQ107B: | If all fields in a colour series is owned by the same player, the price for any other player to land on the fields shall be doubled |

- RQ107C: If all fields in a colour series are owned by the same player, he shall be able to buy houses and hotels on the series.
- RQ108: The game shall have a fixed amount of 32 houses and 12 hotels available.
- RQ109A: A house shall have a fixed price for the colour series, and shall raise the cost for any other players to land on the field
- RQ109B: A property may not have more than 4 houses.
- RQ109C: The player may only buy a house if there are any houses available from the game.
- RQ109D: The player may only buy a house on a specific property in a series if it does not have more houses than any other property in the series.
- RQ109E: The player may sell houses back to the game for half their original price.
- RQ110A: A hotel shall have a fixed price, identical to the price for a house, for the colour series, and shall raise the cost for any other players to land on the field
- RQ110B: A property shall not have more than 1 hotel, and a hotel may only be purchased if the property has 4 houses.
- RQ110C: The player shall only buy a hotel if there are any hotels available from the game.
- RQ110D: The player may only buy a hotel on a specific property in a series if no other properties in the series have 3 houses or less.
- RQ110E: The player may sell hotels back to the game for half their original price.
- RQ111A: The fields of the type Utility shall, if owned by a player, cost a price depending on a die roll for another player that lands on the field.
- RQ111B: If a player owns both Utility fields the price for landing on the field for another player is doubled (according to RQ111A).
- RQ112: The fields of the type Railroad shall, if owned by a player, cost a price depending on the number of Railroads owned by the owner
- RQ113: If a player lands, or passes, GO, he shall be rewarded with 4000.
- RQ114A: If a player lands on Go To Jail, he shall be move to the Jail field.
- RQ114B: To get out of jail, the player may take three attempts per turn for up to 3 turns to roll equals, which will release the player and move on the board based on the roll.
- RQ114C: The player may also spend 1000 to get out of jail. If the player does not, over the 3 consecutive turns, roll equals he must pay 1000 to get out of jail.
- RQ114D: The player may spend a Get Out Of Jail Free card to get out of jail.
- RQ115: If a player lands on Chance, he shall draw a Chance Card, and action described on the card must be realized.
- RQ116: If a player lands on a Tax field, he shall pay a fixed amount, or if the field allows it, pay a percentage of his net worth.
- RQ117A: If a player is unable to pay his due fees, he shall hand over all owned properties (after selling all houses and hotels) to the creditor.
- RQ117B: If the creditor is the bank, all owned properties (subsequent to selling all houses and hotels) shall be set to un-owned, and may be purchased normally by other players.
- RQ118A: Fields of the types Railroad, Utility and Property, shall be mortgage able, meaning that for a fixed fee, the field can be rendered inactive.
- RQ118B: A field may not be mortgaged if any houses or hotels are present.
- RQ118C: The player may pay a price (10% added to the mortgage price) to unmortgage a field, and rendering it active.

2.2 non-functional requirements

RQ201:	The game shall be playable on a Windows computer.
RQ202:	The game shall be coded in the Java coding language
RQ203:	The game shall contain a database
RQ204:	The system shall contain exceptions
RQ205:	The system shall contain collections
RQ206:	The system shall contain input validation
RQ207:	The game shall contain interfaces

Use Case

We have made one main Use case over the whole game and then we made several smaller use cases. In this report we have chosen to include only the Game use case and add the others in the appendix. The main use case helped us figure out what the user would find important and what choices they would be faced with. In our smaller use cases we have matched the requirements with our use cases, so it is easier to trace.

Use case: The Game
Author: Emil Jørgensen – s153063 and Ana-Maria Zokalj s165160
ID: 01
Brief description: A main flow description of the possible actions being carried out during the turn of a single player.
Primary actor: Player
Secondary actor: N/A
Preconditions: The game is launched and a new game is started.
Main flow: <ol style="list-style-type: none">1. The turn is started.2. The player rolls the dice.3. The players token is moved according to the roll.4. The player may mortgage owned properties, railroad or utilities5. The player may unmortgage owned properties, railroads or utilities.6. The player may buy houses, if houses/hotels are available.7. The player may sell housing.8. A player may make deals with other players for owned properties (if no houses are on the property in question), utilities, railroad, “get out of jail free” cards or money.9. The player may forfeit the game, rendering him/herself inactive.10. The player may end his/her turn.
Postconditions: <ol style="list-style-type: none">1. The player’s set of owned properties, railroads or utilities may have been altered.2. The player’s account may have been altered.3. The state of properties, railroads and utilities may have been altered, according to housing and mortgage ability.4. The player’s position may have been altered.
Alternative flow:

A.3.1 The player lands on a property field

A.3.1.1 The property is unowned

A.3.1.1.1 The player chooses to buy the property.

A.3.1.1.1.1 If the player has the required means, these are deducted from his account, and the property is owned by the player.

A.3.1.1.1.2 If the player doesn't have the required means, he must mortgage, sell assets or negotiate to the required means before purchase.

A.3.1.1.2 The player chooses not to buy the property.

A.3.1.2 The property is owned by the active player and nothing happens.

A.3.1.3 The property is owned by another player.

A.3.1.3.1 Depending on the number of properties owned and housing on the field, the active player will pay the owed amount to the player owning the property.

A.3.1.3.2 If the property is mortgaged, the active player doesn't owe anything to the owner of the field.

A.3.2 The player lands on a railroad field.

A.3.2.1 The railroad field is unowned.

A.3.2.1.1 The player chooses to buy the railroad field.

A.3.2.1.1.1 If the player has the required means, these are deducted from his account, and the railroad is owned by him/her.

A.3.2.1.1.2 If the player doesn't have the required means, he/she must mortgage, sell assets or negotiate to the required means before purchase.

A.3.2.1.2 The player chooses not to buy the railroad and nothing happens.

A.3.2.2 The railroad field is owned by the actual player.

A.3.2.3 The railroad field is owned by another player.

A.3.2.3.1 Depending on the number of railroads field owned, the active player must pay the owed amount to the player who owns the railroad field.

A.3.2.3.2 If the railroad is mortgaged, the active player doesn't owe anything to the player owning the field.

A.3.3 The player lands on a utility.

A.3.3.1 The utility is unowned.

A.3.3.1.1 The player chooses to buy the utility field.

A.3.3.1.1.1 If the player has the required means, these will be deducted from his/her account.

A.3.3.1.1.2 If the player doesn't have the required means, he/she must mortgage, sell assets or negotiate to the required means before purchase.

A.3.3.1.2 The player chooses not to buy the utility field and nothing happens.

A.3.3.2 The utility is owned by the active player.

A.3.3.3 The utility is owned by another player.

A.3.3.3.1 Depending on the number of utility fields owned, and the dice value rolled by the active player, the active player pays the owed amount to the player owning the utility field.

A.3.3.3.2 If the utility is mortgaged, the active player doesn't owe anything to the player.

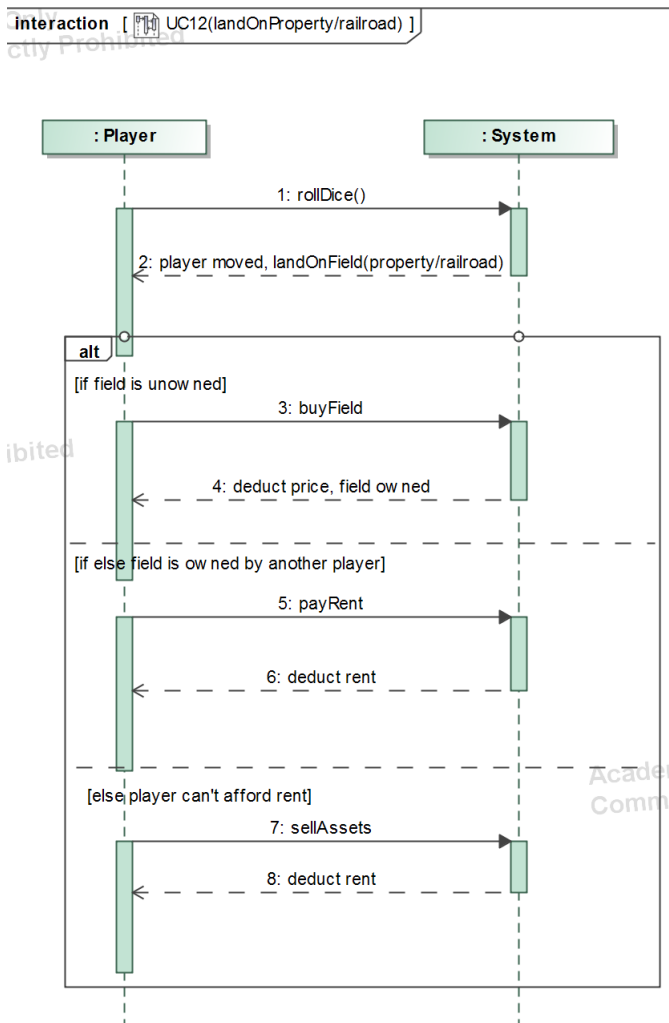
A.3.4 The player lands on Parking and nothing happens.

A.3.5 The player lands on chance

- A.3.5.1 A random card is drawn, the effects of the card is carried out.
- A.3.6 The player lands on Tax.
 - A.3.6.1 Depending on the field, either a fixed amount or a percentage of their capital is deducted from the player's account.
- A.3.7 The player lands on jail.
 - A.3.7.1 The player is put directly in jail.
- A.3.8 The player lands on or passes start
 - A.3.8.1 The player gets an amount of 4000 added to his account.
- A.4.1 The active property, railroad or utility is rendered "mortgaged".
- A.4.2 A fixed amount based on the property, railroad or utility is added to the player's amount.
- A.5.1 The mortgaged property, railroad or utility is rendered "unmortgaged".
- A.5.2 A fixed amount based on the property, railroad or utility is deducted from the players account.
- A.6.1 A house is added to the desired property
- A.6.2 If the property contains 4 houses, they may be replaced by a hotel for a fee.
- A.6.3 A fixed cost, based on the property, is deducted from the players account.
- A.7.1 The hotel is added to the bank.
- A.7.2 If a hotel is sold, it is replaced by four houses, if there aren't enough houses available, additional houses must be sold until a correct amount is available from the bank.
- A.7.3 A fixed amount, based on the property, is added to the players account.
- B.1 The player begins his turn in prison.
 - B.1.1 The player may pay 1000 to get directly out of jail and initiate a regular turn.
 - B.1.2 The player may spend a "get out of jail free" card and get directly out of jail and initiate a regular turn.
 - B.1.3 The player may roll the dice, and if the roll results in equals the player gets out of jail and initiates a regular turn, with the roll before acting as the roll for the turn.
- B.2 The player begins his turn in prison for the third turn in a row.
 - B.2.1 The player may spend a "get out of jail free" card and get directly out of jail and initiate a regular turn.
 - B.2.2 The player may pay 1000 to get directly out of jail and initiate a regular turn. If the player can't manage to pay 1000 he/she is rendered bankrupt.

System Sequence Diagram

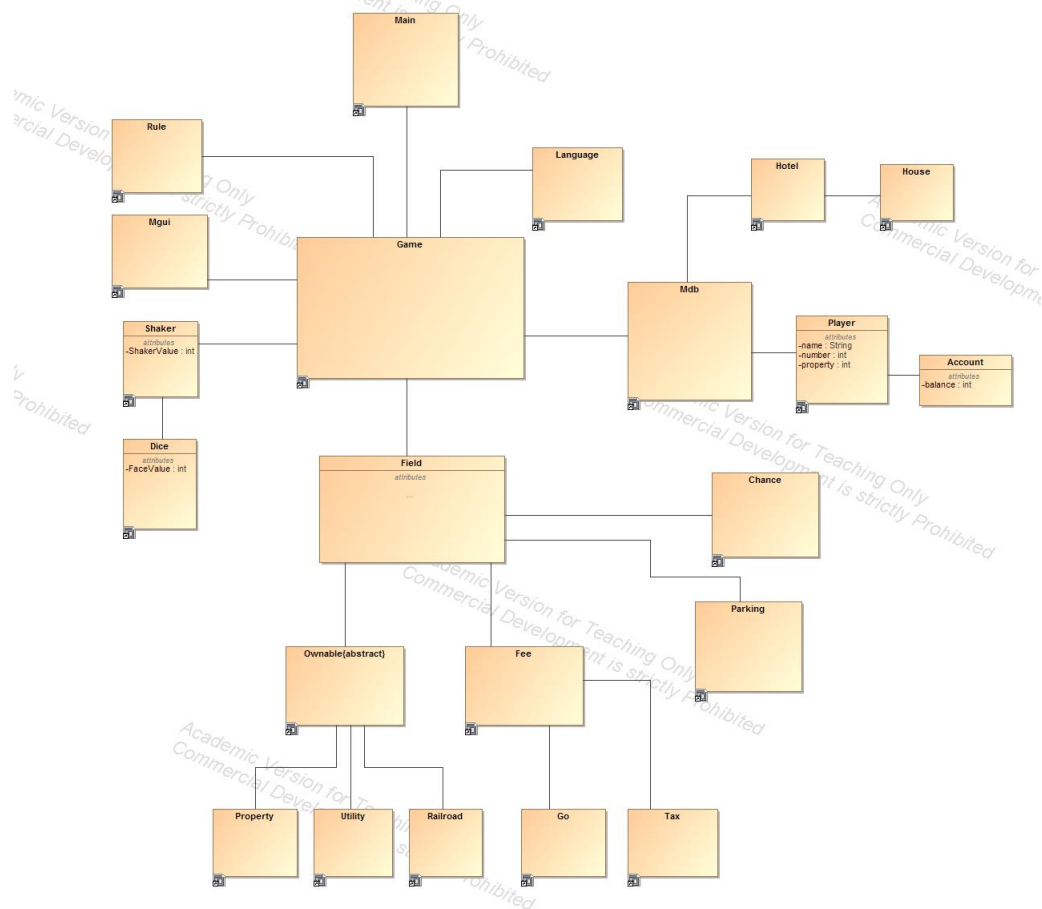
To show an example of one of our SSD's which is made for the land on property/railroad field use case. The SSD shows the most common things that happen if a player lands on property/railroad field. The player starts by rolling the dice. The system moves the player and in this case they land on a property/railroad field. If the field is unowned then the player can buy it. The field can be owned by another player and the current player must pay rent. If the player can't afford the rent, they must sell some assets and then pay the rent.



Domain model

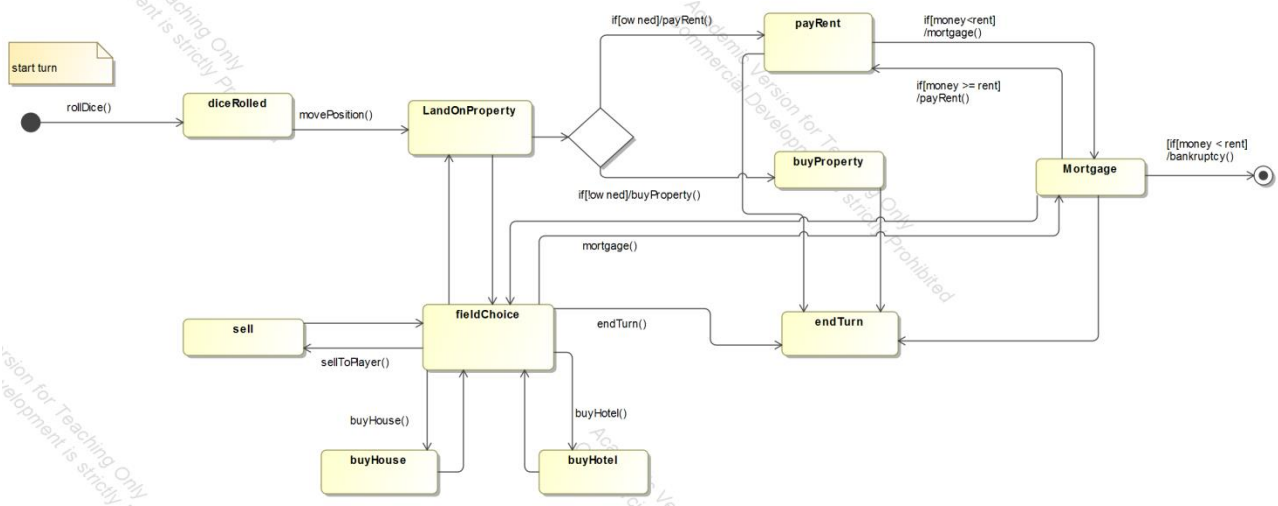
In the beginning of the project, we made a domain model to figure out the classes and how they associate with each other. Some of the classes have changed over time in the final code, because we gained new knowledge, so this is the domain model from the beginning of the project. We have a game class that controls everything. We have the main class that runs the game. We have a language class, so you can choose between English and Danish. There is a dice class with an integer attribute called `faceValue` for the `faceValue` of the dice. The Shaker class that uses dice class to create two dice objects. The shaker class has an integer attribute called `shakerValue`. The field class is an interface, `ownable` implements the field class. `Ownable` is an abstract class and the `property`, `utility` and `railroad` classes inherit from `ownable`. `Fee` implements the Field class, and `Go` and `Tax` class inherit from the `Fee` class. The `chance` and `parking` class implements the Field class. We have a controller called `Mdb`, that works as link between the database and

game. The database contains player, account, hotel and houses.



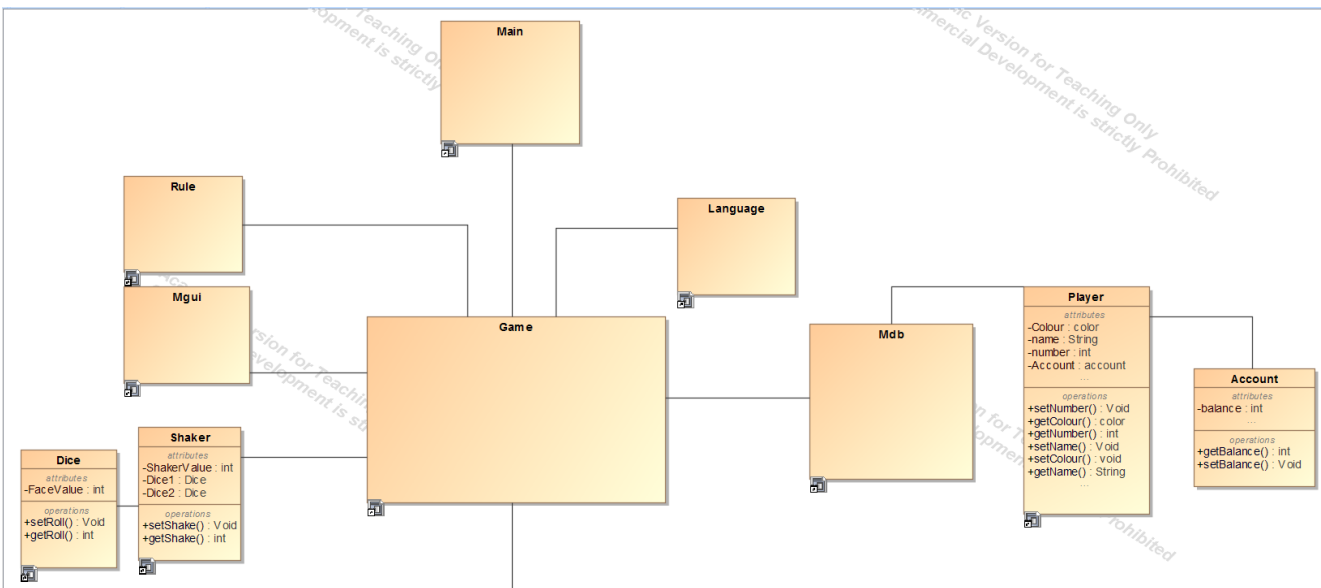
State machine diagram

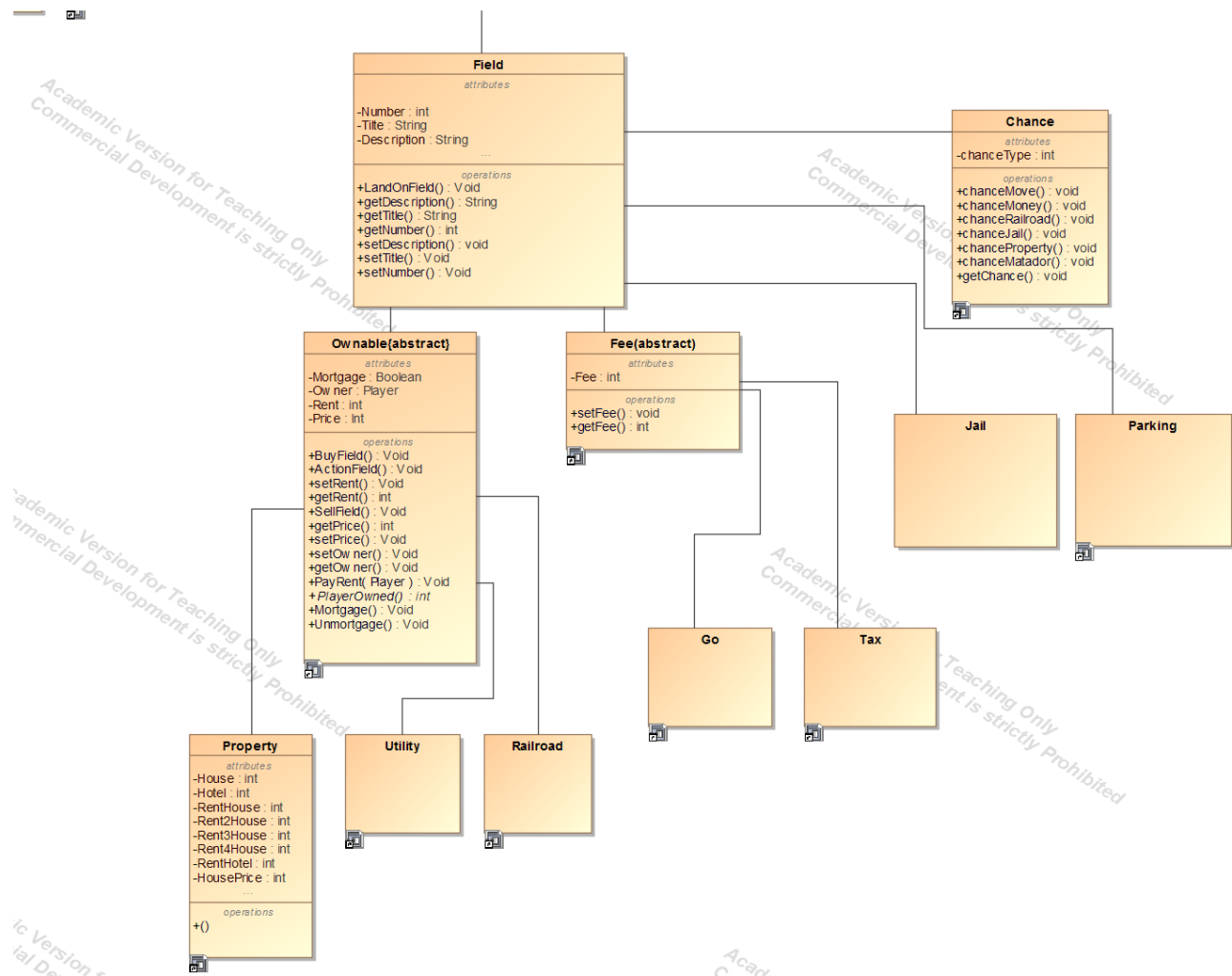
We have designed a state machine diagram to show the players choices, possibilities and limits when it's the players turn and the player lands on a property field. The state machine diagram shows a players turn. First the dice are rolled, and then the player's position is moved. In this case the player lands on a property field. The field is owned by another player, the current player needs to pay rent. If the player doesn't have enough money to pay the rent, the player needs to mortgage some assets until they have the right amount. If the player then has enough money, they must pay the rent. Otherwise they go bankrupt. If the field is unowned, the player can buy the property. Then the turn ends. The player also have a couple of other choices each turn, namely mortgage property fields, sell to other players or buy houses or hotels. The player can also choose to not do anything, then the turn just ends.



Class diagram

We made class diagrams to get an overview over the different classes, their attributes, methods and relationship. The dice class has an integer attribute called `faceValue`, `getRoll/setRoll` method. The shaker class has the attributes `shakerValue` which is an integer, `dice1` and `dice2` that are objects created from the dice-class. It has `setShake/getShake` methods. The player class has the attributes; `colour`, `name` which is a String, `number` which is an integer that correspond to the `playerID`, and `account` which is an object from the account class. Player has `get/set` methods for all its attributes. Account class has the attribute `balance` which is an integer, which is where the player's money is saved.





The field class is an interface class. It has the attributes; number which corresponds to FieldID, title which is the fields name, it has a description of the field. The ownable class is abstract and implements the field class. Ownable has the boolean attribute Mortgage which tells us, if the ownable field is mortgaged or not. It has the attribute owner, which is which of the players owns the field. It has an integer attribute called Rent, which is the field's rent and an integer attribute called price which is the price of the field. Ownable has the method buyField, so the player can buy the field they land on, if it's ownable. It has a method called PayRent, so if the current player lands on an owned field, they must pay rent. Ownable has the methods mortgage and unmortgage, so the player can mortgage or unmortgage their properties. We have a property class which inherits from the Ownable class. The Property class has the integer attributes house, hotel, renthouse, rent2House, Rent3House, Rent4House, RentHotel and housePrice. These attributes describe how many houses there is on the field, what the rent is according to the amount of houses and what the rent is if there is a hotel on the property. The program contains an abstract class called Fee which implements the Field class. It has an integer attribute called fee. There is a Chance class for the Chance Cards, which has an attribute called chanceType. It has methods for each of the chance card types.

MVC (Model-view-controller):

MVC is a three tier model, with view, control and model. View is what the user can see, like a user interface. Controller acts as a link between the Model layer and the View layer components to process all

the business logic and incoming requests, manipulating data using Model component and interact with the Views to render the final output. The Model component corresponds to all the data-related logic that the user works with.

The MVC model is parallel with the BCE model.

View – Boundary

Model – entities

Control – controller

We have divided the packages in our Java code into an entities package with all data related logic, boundary with a graphical user interface and a controller package that works as a link between the boundary layer and the entity layer.

(https://www.tutorialspoint.com/mvc_framework/mvc_framework_introduction.htm)

Central problems:

We have two databases, one for the game and one for the chance cards. We choose to have the chance cards in a database so we can save which cards have been drawn for when you reload the game.

In the beginning of the project, we wanted the player's car color to be saved in the database, so that the player would have the same color car when they reloaded the game. We ended up deciding that each playerID has a fixed car color, so that it didn't need to be saved in the database.

In the beginning we wanted to make sure the game could be played in both Danish and English where we would use I/O with text files to do so, which we later decided against because of limited time.

Author Ana-Maria Fischer Zokalj

Implementation

In this section we have focused on the part of our code that meets the learning goals in the course 02362 – project in software development. This includes how and where we have used JDBC, I/O with text files, exception handling, interfaces, thread programming, and so on.

I/O with text files

For chance cards we use I/O with text files to get the text for the cards. For the board fields, we also used I/O with text file in order to get description, subtext and title.

We are concurrently using try/catch exceptions for loading the text files.

An example from our GameBoard class:

```
public class GameBoard {
    public ArrayList<Field> FieldList = new ArrayList<Field>();
    private Text TitleFile = new Text("FieldTitles.txt");
    private Text DescriptionFile = new Text("FieldDescription.txt");
    private Text SubtextFile = new Text("FieldSubtext.txt");
    private String[] TitleList = null;
    private String[] DescriptionList = null;
    private String[] SubtextList = null;
    private int noOwner = 0;

    /**
     * Creates the Array of boards
     */
    public void CreateBoard(){
        try {
```

```

        TitleList = TitleFile.OpenFile();
        DescriptionList = DescriptionFile.OpenFile();
        SubtextList = SubtextFile.OpenFile();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

```

Exception handling

We used try/catch exceptions, to catch any errors that could occur. An example from the code, DBCreator class:

```

try{
    Class.forName("com.mysql.jdbc.Driver"); //Register JDBC Driver

    System.out.println("Creating a connection...");
    conn = DriverManager.getConnection(DB_URL, USER, PASS); //Open a connection

    ResultSet resultSet = conn.getMetaData().getCatalogs();

    while (resultSet.next()) {

        String databaseName = resultSet.getString(1);
        if(databaseName.equals(dbName)){
            return true;
        }
    }
    resultSet.close();
}
catch(Exception e){
    e.printStackTrace();
}
}

```

Interfaces

Our field class is an interface class:

```

/**
 * @author Emil Jørgensen, editet by Aleksander
 */
package entities;

import java.awt.Color;

import controllers.*;
import controllers.mGUI;
import entities.Shaker;

public interface Field {

    //METHODS

    /**
     * Method for the action landing on a field.
     */
    public void landOnField(Game game, GameBoard gameboard, int b, int p, mGUI mui, Shaker shake);

    /**
     * Method for getting description
     * @return Description of the current field
     */
    public String getDescription();

    /**
     * Method for setting description
     */
    public void setDescription(String desc);

    /**
     * Method for getting title
     * @return Title on the field
     */
}

```

```

    */
    public String getTitle();
    /**
     * Method for setting title
     */
    public void setTitle(String titl);

    /**
     * Method for getting number
     * @return Number of the field
     */
    public int getNumber();
    /**
     * Method for setting number
     */
    public void setNumber(int numb);

    /**
     * Method for setting the Colour of a field
     * @param colour the Java.awt.Color
     */
    public void setColour(Color colour);

    /**
     * Method for getting the Colour of a field
     */
    public Color getColour();
}

```

Thread programming

We used thread programming in our Game class and made the threads all share the object lock. This ensures that the threads have a common “lock for the cpu”:

```

    /**
     * Creates the different threads for the game.
     * @param playersInGame
     */
    public void createPlayerThreads(int playersInGame)
    {
        for(int x = 1; x <= playersInGame; x++){
            synchronized(lock){
                }
                PlayTurn thread = new PlayTurn("x", playerList.get(x).getID(), this,
this.board);
                thread.start();
            }
            synchronized(lock){
                id = 1;
                lock.notifyAll();
            }
        }
    }

```

JDBC

We used JDBC to create and connect to our two databases, here is an example:

```

public DBconnector() {
    }
    /**

```

```

    * The connector method. Connects to a SQL Dataase
    * @param host The host name, "localhost" if run locally
    * @param port The port ID, "3306" if run locally
    * @param database The name of the database you want to connect to
    * @param user The username to the server
    * @param pass The password to the server
    */
    public void Connect(String DB) {
        try {

            Class.forName("com.mysql.jdbc.Driver");
            String url = "jdbc:mysql://" + HOST + ":" + PORT +

            connection = DriverManager.getConnection(url,

        } catch (ClassNotFoundException | SQLException e) {
            e.printStackTrace();
            System.exit(1);
        }

    }

    /**
     * @return connection
     */
    public Connection getConnection(){
        return connection;
    }

    /**
     * Method for SQL Data Querys (NOT DATA MANIPULATION)
     * @return Returns the data as a ResultSet
     */
    public ResultSet doQuery(String DB,String query) throws SQLException{
        Connect(DB);
        Statement stmt = connection.createStatement();
        ResultSet res = stmt.executeQuery(query);
        return res;
    }

    /**
     * Method for SQL Data Manipulation
     */
    public void doUpdate(String DB, String query) throws SQLException{
        Connect(DB);
        Statement stmt = connection.createStatement();
        stmt.executeUpdate(query);
    }

    /**
     * Method for closing the connection after use
     */
    public void close() throws SQLException{
        if(connection!=null){
            connection.close();
        }
    }
}

```

Software documentation











To show our overall design of the program, we have made a design class diagram. We started with a more simple class diagram in the beginning of the project to give us an idea of the overall design.

The design of the Java program

We have used the Boundary – controller – entity model for how we divided our java program into packages as described earlier in the MVC section.

The boundary package would contain our GUI class, which is what the user can see and interact with. The GUI is placed in the Build Path instead, but had we created it, it would be a boundary. The mGui class controls the GUI. The GUI shows our Monopoly gameboard with Fields, player cars and dice.

The “controllers” package contains:

- ✓  controllers
 - >  aDB.java
 - >  Chance.java
 - >  DBconnector.java
 - >  DBcreator.java
 - >  Game.java
 - >  GameBoard.java
 - >  Main.java
 - >  mGUI.java
 - >  PlayTurn.java

The aDB class is an abstract class which gets the current database and sets the name of the database. It contains common values for the database classes; DBconnector and DBCreator.

Chance class has a method that creates all the chance cards. It has a method that shuffles the chance cards. It has a method called drawChance, which draws a chance card. It has a loadChance, which loads all the chance cards which haven't been deleted.

DBconnector class inherits from the aDB class. It has the responsibility of connecting to the database using the values from the aDB class. Its responsibility is also to alter and view the database.

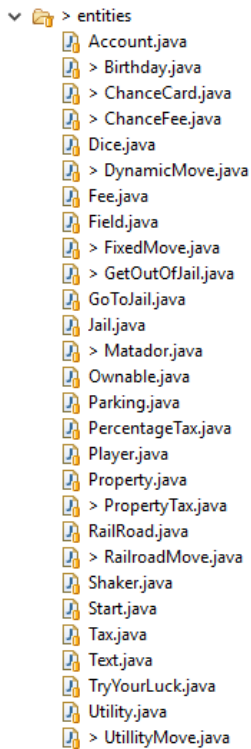
DBCreator class inherits from the aDB class and uses the DBconnector. It has the responsibility to create the Game and Chance databases with tables. It can also delete the database.

Game class is responsible for creating the objects; the player threads etc.

GameBoard class creates all the fields in the code.

PlayTurn class is responsible for each players turn. Each player runs in their own thread, which is on wait when it's not the players turn. It is responsible for the player's possibility to sell to other players, mortgage and buying houses/hotels or properties. It contains the method sellOfStuff, which sells your assets if your balance is less than 0. It starts with selling your hotels and houses and continues on to sell properties.

The entities package contains:



The Account class is responsible for the players balance and net worth.

The Player class is responsible for the player and the player's position on the board. The player has an account, so the player class creates an account object.

The Dice class is responsible for the dice and the shaker class, shakes the two dice.

The Text class has a text file attached which it opens and reads.

Chance

The classes; Birthday, ChanceFee, DynamicMove, FixedMove, GetOutOfJail, Matador, PropertyTax, Railroad move inherits from the abstract class ChanceCard. They are different types of chance cards and are saved in different tables in the database and therefore they are made in their own classes.

ChanceCard is an abstract class with get/set methods for the shared variables.

Field

Field class is an interface class which defines what a field basically is and makes it simple to create the GameBoard list. Ownable is an abstract class that implements the Field class. Ownable is responsible for defining the Ownable fields – property, utility and railroad.

The Property class inherits from the Ownable class, and defines the property fields.

The Utility class inherits from the Ownable class, and defines the utility fields.

The Railroad class inherits from the Ownable class, and defines the Railroad fields.

The Fee class is an abstract class which implements the Field class. The Fee class is responsible for defining the Fee field – which is the Tax class.

The Tax class inherits from the Fee class, and defines the Tax fields.

The Start class implements the Field class, and defines the Start field.

The GoToJail class implements the Field class, and defines the Go to Jail field.

The Parking class implements the Field class, and defines the Parking field.

The Jail class implements the Field class, and defines the Jail Field.

Use of Javadoc

An example from the Field class that shows the use of Javadoc in our Java code:

```
24     public String getDescription();
25     /**
26      * Method for setting description
27      */
28     public void setDescription(String desc);
29
30     /**
31      * Method for getting title
32      * @return Title on the field
33      */
34     public String getTitle();
35     /**
36      * Method for setting title
37      */
38     public void setTitle(String titl);
39
40     /**
41      * Method for getting number
42      * @return Number of the field
43      */
44     public int getNumber();
45     /**
46      * Method for setting number
47      */
48     public void setNumber(int numb);
49
50     /**
51      * Method for setting the Colour of a field
52      * @param colour the Java.awt.Color
53      */
54     public void setColour(Color colour);
55
56     /**
57      * Method for getting the Colour of a field
58      */
59     public Color getColour();
60 }
```

Database

Both our databases are created in our Java code using JDBC, here is an example of the code used to create our Game database in the DBCreator class:

```
public void tbCreatorGame(){
    //CREATION of tables
    try {
        GameConnector.doUpdate("Game","CREATE TABLE Player(PlayerID INTEGER(1), Name VARCHAR(20), Position INTEGER(2), GetOutOfJail INTEGER(1), Jailtries INTEGER(1), PRIMARY KEY ( PlayerID ));");
        GameConnector.doUpdate("Game","CREATE TABLE Account(PlayerID INTEGER(1), Money INTEGER(10), Networth INTEGER(10), PRIMARY KEY ( PlayerID ));");
        GameConnector.doUpdate("Game","CREATE TABLE Field(FieldID INTEGER(2), Name VARCHAR(20), Description VARCHAR(140), PRIMARY KEY ( FieldID ));");
        GameConnector.doUpdate("Game","CREATE TABLE Ownable(FieldID INTEGER(2), Owner INTEGER(2), Price INTEGER(2), Mortgage INTEGER(4), PRIMARY KEY( FieldID ), FOREIGN KEY ( FieldID ) REFERENCES Field(FieldID ));");
        GameConnector.doUpdate("Game","CREATE TABLE Property(FieldID INTEGER(2), Rent INTEGER(4), Rent1 INTEGER(4), Rent2 INTEGER(4), Rent3 INTEGER(4), Rent4 INTEGER(4), Hotel INTEGER(1), PRIMARY KEY ( FieldID ));");
        GameConnector.doUpdate("Game","CREATE TABLE Utility(FieldID INTEGER(2), StartFee INTEGER(5), PRIMARY KEY ( FieldID ));");
        GameConnector.doUpdate("Game","CREATE TABLE Railroad(FieldID INTEGER(2), Rent1 INTEGER(4), Rent2 INTEGER(4), Rent3 INTEGER(4), Rent4 INTEGER(4), PRIMARY KEY ( FieldID ));");
        GameConnector.doUpdate("Game","CREATE TABLE Jail(FieldID INTEGER(2), Turns INTEGER(2), PRIMARY KEY ( FieldID ), FOREIGN KEY ( Player ) REFERENCES Player(PlayerID ));");
        GameConnector.doUpdate("Game","CREATE TABLE Tax(FieldID INTEGER(2), TaxType INTEGER(1), Tax INTEGER(4), PRIMARY KEY ( FieldID ));");
        GameConnector.doUpdate("Game","CREATE TABLE ProcentageTax(TaxID INTEGER(1), procentageTax INTEGER(2), PRIMARY KEY ( TaxID ));");
        System.out.println("Tables in Game created successfully...");
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
//CLOSURE
try{
    if(GameConnector!=null)
        GameConnector.close();
} catch (SQLException se){
    se.printStackTrace();
}
}
```

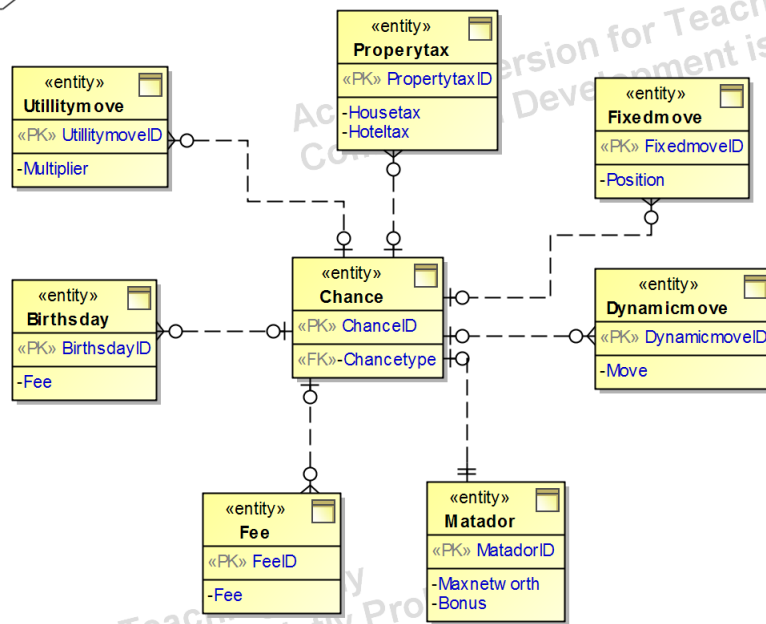
If we created the Game database using MySQL workbench, it would look like this:

```
1 • create database Game;
2 • use game;
3
4 • create table player(
5   PlayerID integer(1),
6   Name varchar(20),
7   Colour varchar(10),
8   Position integer(2),
9   GetOutOfJail integer(1),
10  primary key (PlayerID)
11 );
12
13 • create table Account(
14   PlayerID integer(1),
15   Money integer(1),
16   Networth integer(10),
17   primary key (PlayerID)
18 );
19
20 • create table Field(
21   FieldID integer(2),
22   Name varchar(20),
23   Description varchar(140),
24   Primary key (FieldID)
25 );
26
27 • create table Ownable(
28   FieldID integer(2),
29   Owner integer(2),
30   Price integer(2),
31   Mortgage integer(4),
32   Primary key(FieldID),
33   Foreign key (Owner) references Player(PlayerID)
34 );
```

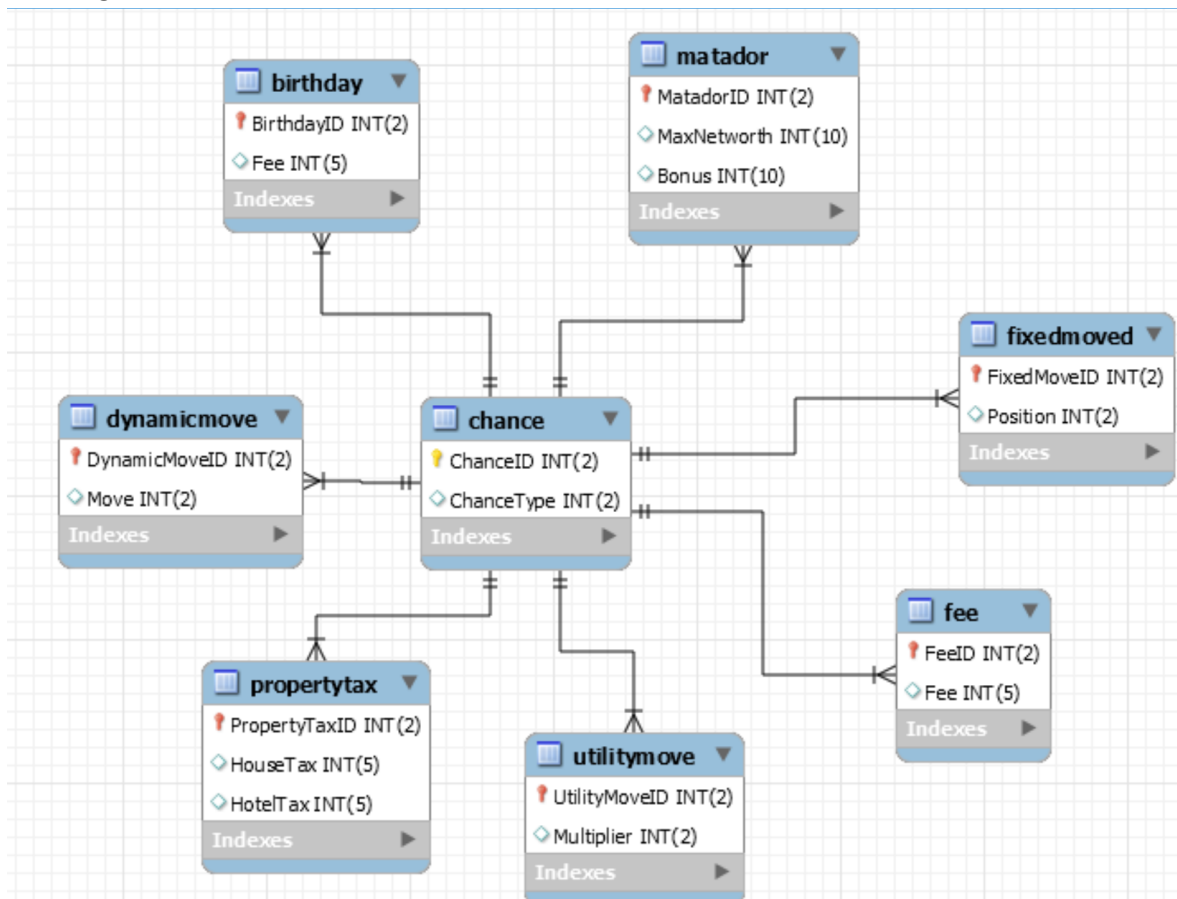
And so on.

For our database, we first programmed the Game and Chance database in SQL workbench using MySQL. We wanted to make the database in MySQL workbench and then connect to that database using JDBC in our java code. We later decided that the databases should be created in our java program using JDBC instead. Our ER diagram for our game database shows that our database is in 3rd normal form. Our EER diagram is created in MySQL workbench and is essentially an expansion of our ER diagram. One of the main differences from our ER-diagram and our EER diagram is that the EER diagram shows what foreign keys we have used in the different tables.

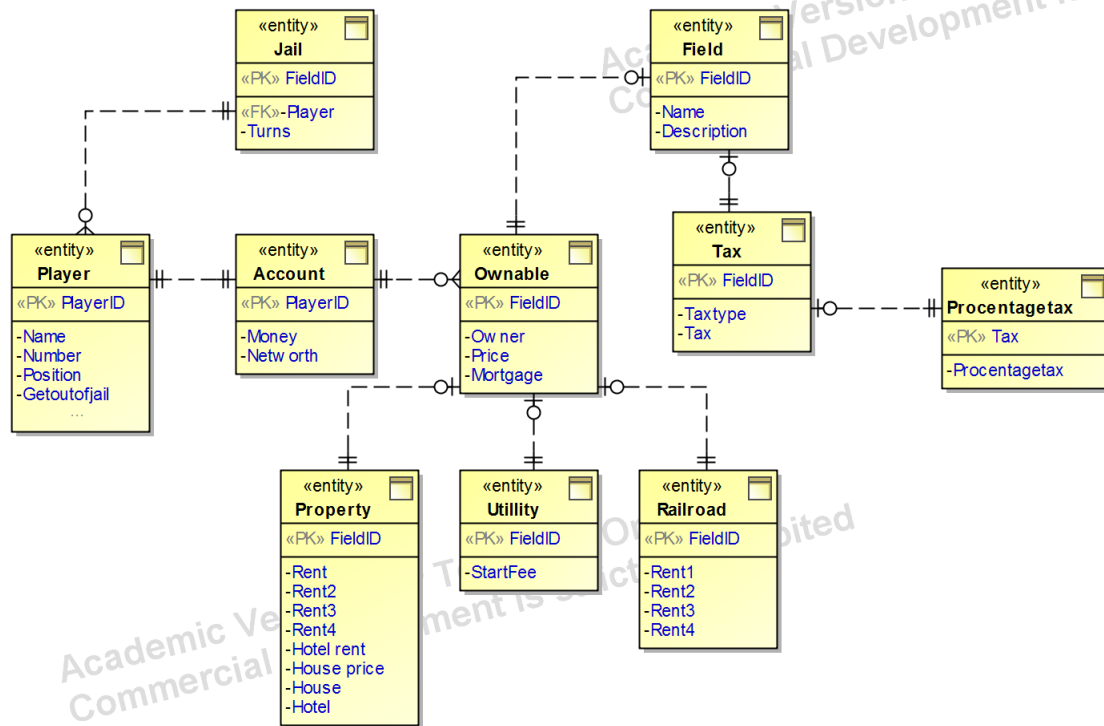
ER – diagram for the Chance database:



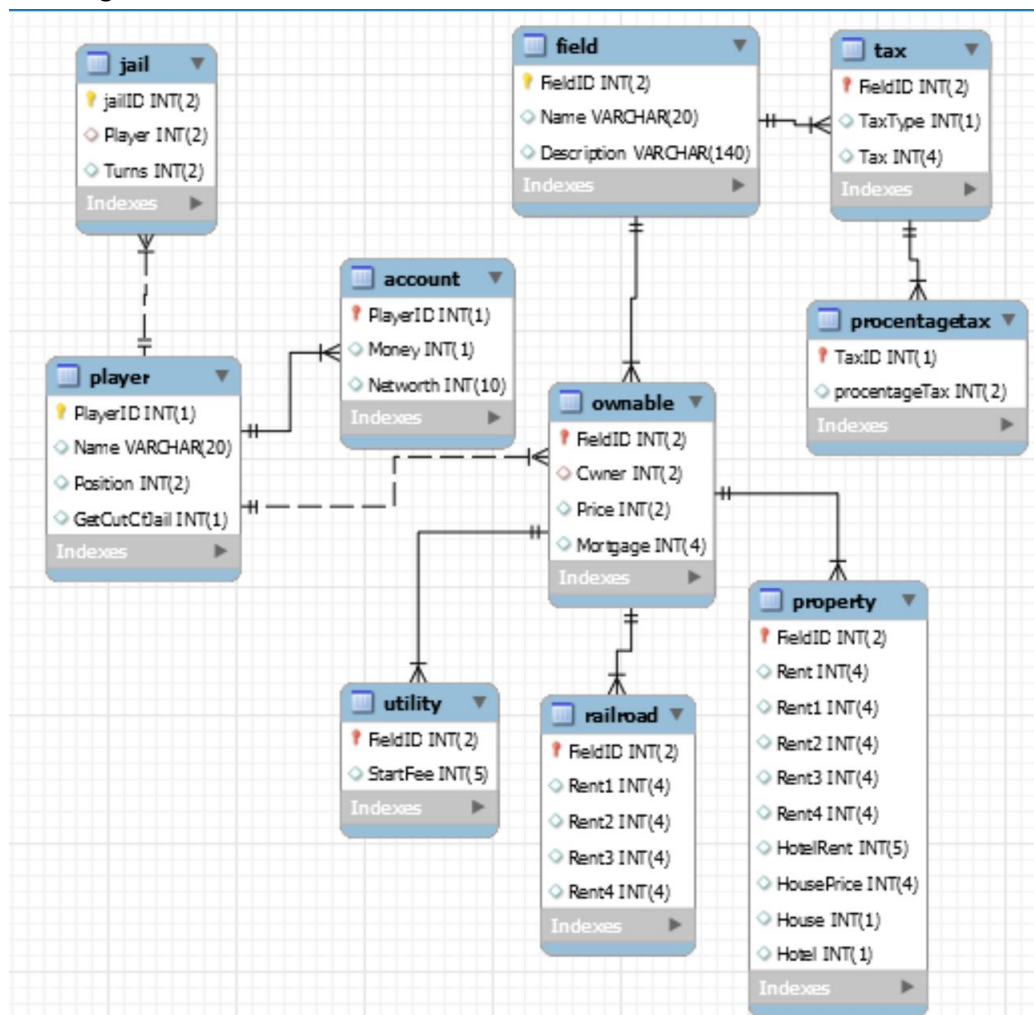
EER Diagram of the Chance card Database:



ER-Diagram for the Game Database:



EER Diagram for the Game database



Author Ana-Maria Fischer Zokalj

Test

We have continuously tested our code, but unfortunately the time ran out and we didn't have the time to make any actual test cases. If we had more time, we would have tested our program throughout.

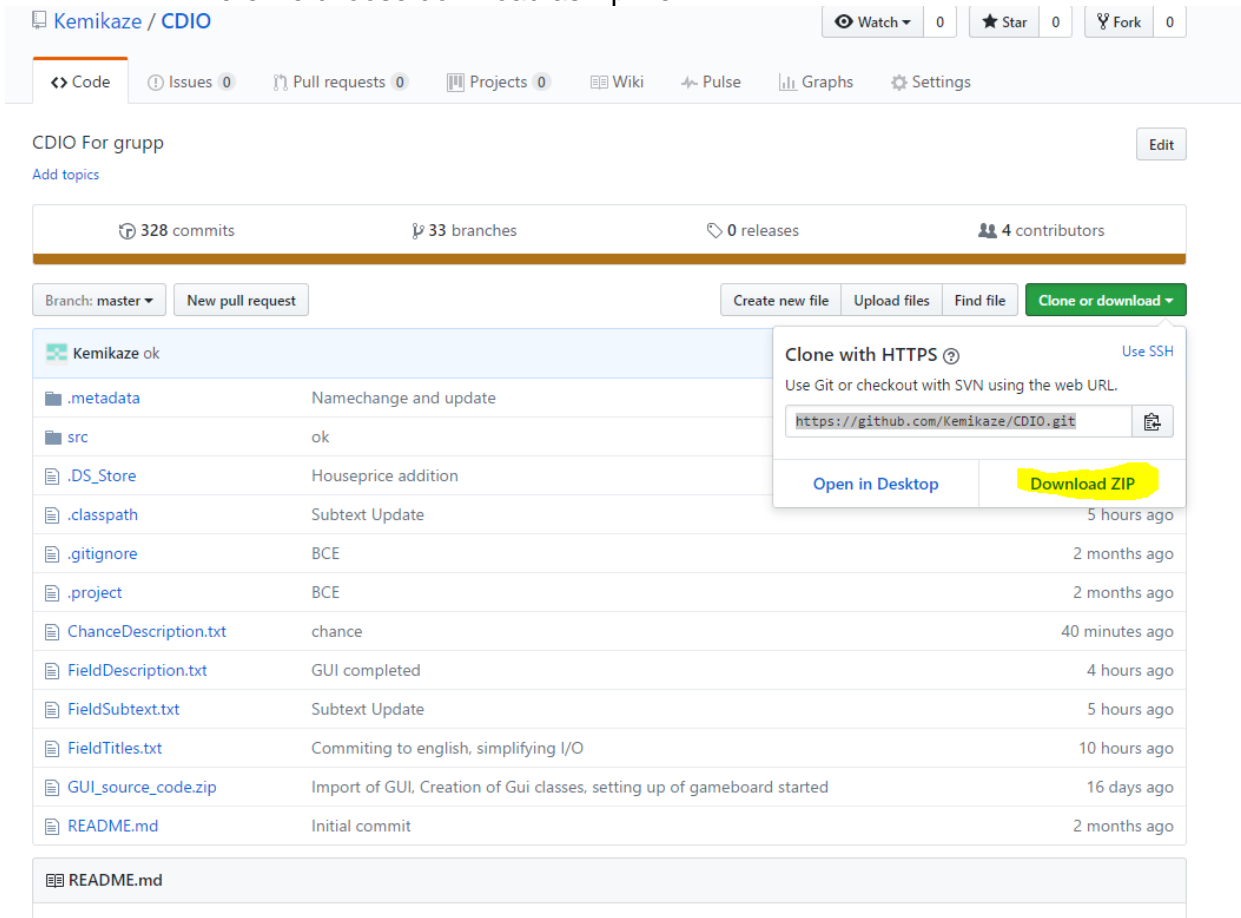
Author Ana-Maria Fischer Zokalj

User's manual and example

Manual for installing and playing

Installing the repository from github:

1. The gitrepository is fetched from github on the following url:
<https://github.com/Kemikaze/CDIO>
2. Here we choose download as zip file.



Kemikaze / CDIO

Watch 0 Star 0 Fork 0

Code Issues 0 Pull requests 0 Projects 0 Wiki Pulse Graphs Settings

CDIO For grupp Edit

Add topics

328 commits 33 branches 0 releases 4 contributors

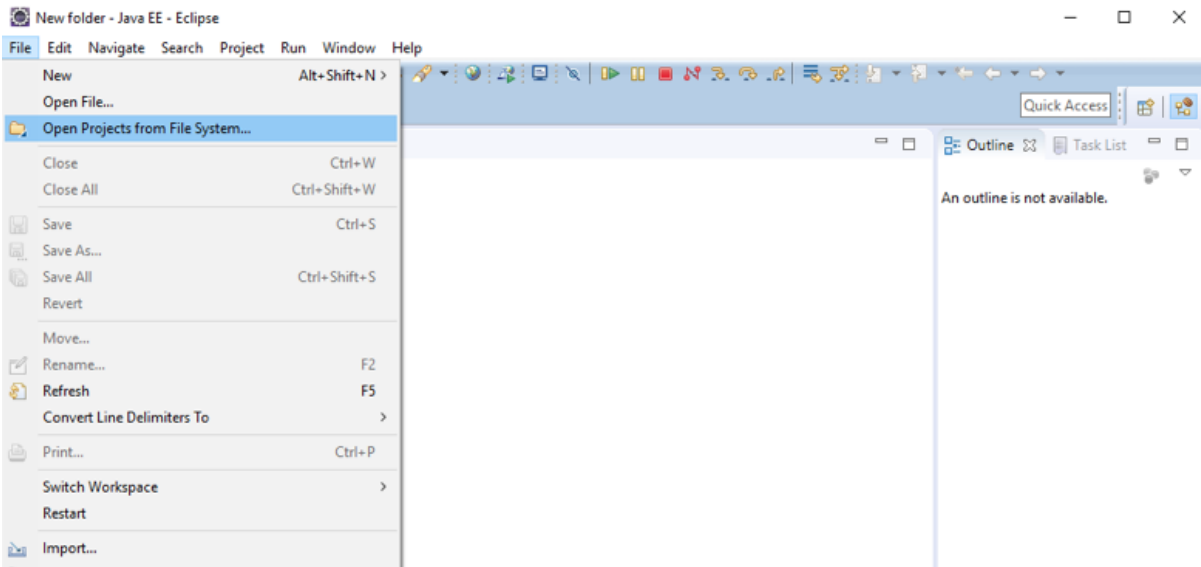
Branch: master New pull request Create new file Upload files Find file Clone or download

Kemikaze ok

.metadata	Namechange and update	
src	ok	
.DS_Store	Houseprice addition	
.classpath	Subtext Update	5 hours ago
.gitignore	BCE	2 months ago
.project	BCE	2 months ago
ChanceDescription.txt	chance	40 minutes ago
FieldDescription.txt	GUI completed	4 hours ago
FieldSubtext.txt	Subtext Update	5 hours ago
FieldTitles.txt	Committing to english, simplifying I/O	10 hours ago
GUI_source_code.zip	Import of GUI, Creation of Gui classes, setting up of gameboard started	16 days ago
README.md	Initial commit	2 months ago

README.md

3. The zip file is then extracted, and the default folder that windows chooses is selected
4. Eclipse is now to be opened, and in here we choose, "File" and "Open Projects from File System"



5. A popup occurs, and here we find our path to where we extracted our zip file.
6. After this, we press Finish, and the repository is loaded into our eclipse.
7. To start the game, we now need to find the main class,

Author Aleksander Wienziers-Madsen

Conclusion

Our group has created a program that simulates a Monopoly Game (Matador spil) where the game can be saved in a database. Unfortunately we ran out of time and the player isn't able to load the game. The player can roll the dice, move around on the game board and buy unowned properties. For each turn the player has some choices, which include buy houses/hotels, trade with other players, mortgage their properties or simply not do anything. If the player lands on a property, utility or railroad field that is owned by another player, the current player has to pay rent. If the current player doesn't have enough money to pay the rent, the player must sell their assets until they have enough money to pay the rent.

Our Java code meets several of the learning goals for the course "Project in Software Development". We have used thread programming in the playTurn class which is responsible for the players turn. We have used I/O with text files in the Text class that has a text file attached which it opens and reads. We have used JavaDoc to document much of our code. We have an interface class, called Field which is implemented in our Ownable class. We have used the BCE model to divide our java program into packages of boundary, controller and entities. The program includes JDBC that connects and creates our two databases in our Java program. One database where the essential objects from the game are saved and another database where the Chance cards are saved. We didn't use views or stored procedures, which we could have included if we had time, so we could have met more of the learning goals for Introductory Databases.

Because of limited time, we have not been able to Test the program as much as we would have liked.

Author Ana-Maria Fischer Zokalj

Time card

CDIO						
Deltagernavn	Analyse	Design	Implementation	Test	Dokumentation	Ialt
Aleksander	5.5	12.5	30	4	2	54
Emil	10	5	50	5	5	75
Simon	10	12	40	1	2	65
Ana	10	15	2	0	25	52
Ialt	35.5	44.5	122	10	34	

Bibliography

- www.stackoverflow.com
- Applying UML and Patterns – third edition – by Craig Larman
- Database system concepts, sixth edition – by Silberschatz, Korth and Sudarshan.
- https://www.tutorialspoint.com/mvc_framework/mvc_framework_introduction.htm

Appendix

Appendix A – Use cases

Use Case: Roll Dice
Author: Emil Jørgensen - s153063
ID: 02
Requirements: RQ102
Brief description: A player begins his turn by rolling the dice and moving accordingly
Primary actors: Player
Secondary actors: N/A
Preconditions: N/A
Main flow: <ol style="list-style-type: none">1. The player clicks "roll dice"2. The dice is rolled, and showed to the player3. The player is moved accordingly to the roll
Postconditions: <ol style="list-style-type: none">1. The player's position is moved.
Alternative flows: N/A

Use Case: Buy Property
Author: Emil Jørgensen - s153063
ID: 03
Requirements: RQ105
Brief description: The player lands on an ownable field and chooses to buy it.
Primary actors: Player
Secondary actors: N/A
Preconditions: The player has rolled the dice and landed on an ownable field that is currently unowned.
Main flow: <ol style="list-style-type: none">1. The Player clicks "Buy Property"2. The cost of the field is deducted from the Players account

3. The property is set to be owned by the player
Postconditions: 1. The cost has been deducted from the player 2. The property is owned by the player
Alternative flows: A1. The player does not have the required funds, and is informed that he cannot buy the property

Use Case: Buy House
Author: Emil Jørgensen - s153063
ID: 04
Requirements: RQ109
Brief description: When the player owns all the properties in a series, he/she can buy a house on the properties
Primary actors: Player
Secondary actors: N/A
Preconditions: 1. The player owns all the properties in a series 2. The property does not have more houses than any other property in the series
Main flow: 1. The player clicks on "Buy a House" 2. The price of the house is deducted from the players account 3. The house is added to the property
Postconditions: 1. The price of the house is deducted from the player 2. A house is added to the property
Alternative flows: A1. The player does not have the required funds to buy a house, and is informed by this A2. The field cannot get another house (according to the preconditions) and the player is informed by this

Use Case: Buy Hotel
Author: Aleksander
ID: 05

Requirements: RQ110
Brief description: When the player owns all the properties in a series, and has built 4 houses on each, he can upgrade them to a hotel
Primary actors: Player
Secondary actors: N/A
Preconditions: 1. The player owns all the properties in a series. 2. The properties have 4 houses each on them.
Main flow: 1. The player clicks on "Actions" 2. The player clicks on "Buy a Hotel" 3. The hotel is added to the property, and the houses are removed.
Postconditions: 1. The price of the hotel is deducted from the player 2. A hotel is added to the property
Alternative flows: A1. The player does not have the required funds to buy a hotel, and is informed about this. A2. The field cannot get a hotel, because there are no more hotels left, and the player is informed about this.

Use Case: Sell House
Author: Ana-Maria Zokalj, s165160
ID: 06
Requirements: RQ109E
Brief description: A player can sell their houses.
Primary actors: Player
Secondary actors: N/A
Preconditions: 1. The player owns at least one house. 2. The property does not have less houses than any other property in the series.

Main flow: <ol style="list-style-type: none"> 1. The player clicks on "Actions". 2. The player clicks on "Sell a House". 3. Half of the original price of the house is added to the players account. 4. The house is removed from the property.
Postconditions: <ol style="list-style-type: none"> 1. Half of the original price of the house is added to the players account. 2. A house is removed from the property.
Alternative flows: N/A

Use Case: Sell Hotel
Author: Ana-Maria Zokalj, s165160
ID: 07
Brief description: A player can sell their hotels.
Requirements: RQ110E
Primary actors: Player
Secondary actors: N/A
Preconditions: <ol style="list-style-type: none"> 1. The player owns at least one hotel
Main flow: <ol style="list-style-type: none"> 1. The player clicks on "Actions". 2. The player clicks on "Sell a Hotel". 3. Half of the original price of the hotel is added to the players account. 4. The hotel is removed from the property. 5. When the hotel is sold, it is replaced by four houses, from the bank.
Postconditions: <ol style="list-style-type: none"> 1. Half of the original price of the hotel is added to the players account. 2. A hotel is removed from the property.

Alternative flows:

A1. If there aren't enough houses available to replace the hotel, additional houses must be sold until a correct amount is available.

Use Case: Go to Jail
Author: Simon
ID: 08
Requirements: RQ114
Brief description: The player lands on the go to jail field
Primary actors: Player
Secondary actors: N/A
Preconditions: The player has rolled the dice and landed on the go to jail field
Main flow: 1. The lands on the go to jail field 2. The player is put directly to jail
Postconditions: 1. The player is now in jail
Alternative flows: N/A

Use Case: Get out of jail
Author: Simon Christiansen, s165153
ID: 09
Brief description: A player gets out of jail
Requirements: RQ114B - RQ114D
Primary actors: Player
Secondary actors: N/A

Preconditions: 1. The player is in jail
Main flow: 1. The player roll the dice <div style="margin-left: 40px;"> 1.1 If the dice shows a pair the player is released from jail <div style="margin-left: 40px;"> 1.1.1 The player moves the number of eyes shown on the dice </div> </div> <div style="margin-left: 40px;"> 1.2 if the dice doesn't show a pair the players turn is over <div style="margin-left: 40px;"> 1.2.1 The player doesn't roll a pair three turn in a row, the player is released </div> </div>

Use Case: Draw Chance
Author: Simon Christiansen, s165153
ID: 10
Brief description: The player draws a chance card
Requirements: RQ115
Primary actors: Player
Secondary actors: N/A
Preconditions: 1. The landed on a Draw chance field
Main flow: 1. The player draws a chance card 2. Depending on the chance card drawn the player is forced to do something or given a "Get out of jail free card"
Postconditions: 1. Postcondition depends on the chance card drawn
Alternative flows: N/A

Use Case: Pass Go
Author: Aleksander

ID: 11
Requirements: RQ113
Brief description: Player passes GO and receives money from the bank in doing so.
Primary actors: Player
Secondary actors: N/A
Preconditions: Player is starting his/her turn and has yet to roll the dice.
Main flow: 1. The player rolls the dice. 2. The value of the dices are enough to make him land/pass GO 3. The player receive 4000 for passing GO
Postconditions: 1. The players account reflects that he/she passed GO
Alternative flows: N/A

Use Case: Land on property/railroad
Author: Ana-Maria Zokalj - s165160
ID: 12
Requirements: RQ112/RQ106
Brief description: The player rolls the dice and land on a property or a railroad field.
Primary actors: Player
Secondary actors: N/A
Preconditions: 1. The player lands on a property or a railroad field.

Main flow:

1. The player rolls the dice.
2. The player's car is moved according to the roll.
3. The player lands on a property or a railroad field.
4. The field is unowned.
5. The player chooses to buy the field
6. The player has the required means. The price is deducted from his/hers account, and the field is now owned by him/her.

Postconditions:

1. The player buys the property field and can claim rent from other players landing on this field.
2. The field is already owned by another player, and the active player has to pay rent to the player owning the field.

Alternative flows:

- 4.1 The field is already owned by the active player and nothing happens.
- 4.2. The field is owned by another player.
 - 4.2.1 Depending on the number of fields owned, the active player must pay the owed amount to the player who owns the field.
 - 4.2.2 If the field is mortgaged, the active player doesn't owe anything to the player owning the field.
 - 4.2.3 If the active player can't afford paying the rent for landing on the field.
 - 4.2.3.1 The active player must mortgage or sell assets in order to pay the rent.
 - 4.2.3.1.1 If the active player is still unable to pay the owed amount, he/she goes bankrupt and loses the game.
- 5.1 The player can choose not to buy the field.
- 6.1 If the player doesn't have the required means, he/she must mortgage, sell assets or negotiate to the required means before purchase.

Use Case: Land on Utility**Author:** Emil Jørgensen - s153063**ID:** 13**Requiriements:** RQ111**Brief description:**

What happens when a player lands on Utility

Primary actors:

Player

Secondary actors:

N/A

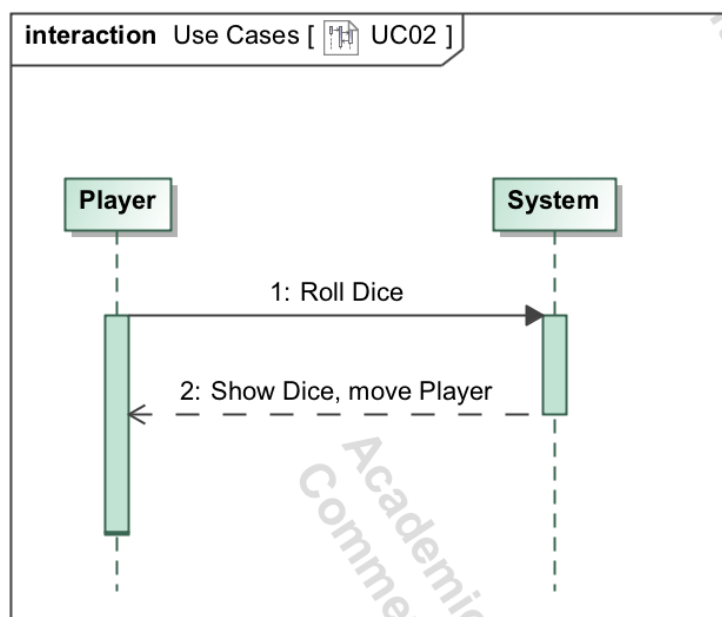
Preconditions: 1. The player lands on a utility
Main flow: 1. The utility is unowned <ul style="list-style-type: none"> 1.1. The player chooses to buy the utility <ul style="list-style-type: none"> 1.1.1. If the player has the required means, these are deducted from his account, and the utility is owned by the player 1.1.2. If the player doesn't have the required means, he must mortgage, sell assets or negotiate to the required means before purchase 1.2. The player chooses not to buy the utility 2. The utility is owned by the active player 3. The utility is owned by another player <ul style="list-style-type: none"> 3.1. Depending on the number of utilities owned, and the dice value rolled by the active player, the active player pays the owed amount to the player owning the utility 3.2. If the utility is mortgaged, the active player doesn't owe anything to the player
Postconditions: 1. If Main flow 1.1.1. is used, the player owns the utility, and the price has been deducted 2. If Main flow 3.1. is used, the correct rent is transferred from the player to the owner
Alternative flows: N/A

Use Case: Pay Tax
Author: Ana-Maria Zokalj - s165160
ID: 14
Requirements: RQ116
Brief description: Player lands on a tax field and has to pay a fee.
Primary actors: Player
Secondary actors: N/A
Preconditions:

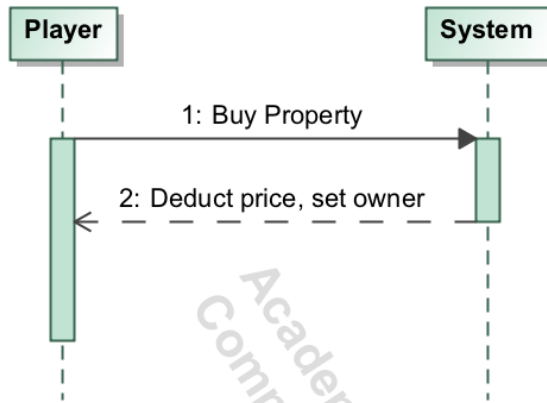
1. Player lands on a tax field
Main flow: <ol style="list-style-type: none"> 1. The player rolls the dice. 2. The player lands on a tax field. 3. The player may either pay a factored amount of 10% of all the player's assets or a fixed amount of 4000. 4. The amount chosen is deducted from the players account.
Postconditions: <ol style="list-style-type: none"> 1. The fee (either fixed or factored) is deducted from the player's account.
Alternative flows: N/A

Appendix B – System Sequence diagram

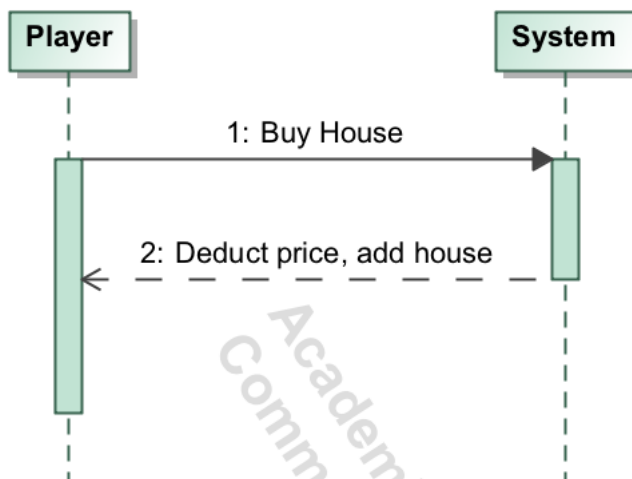
The person who made the Use Case, also made the corresponding SSD to that Use Case.



interaction Use Cases [UC03]



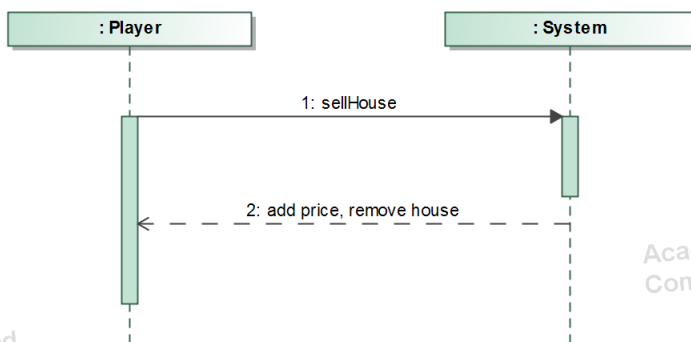
interaction Use Cases [UC04]



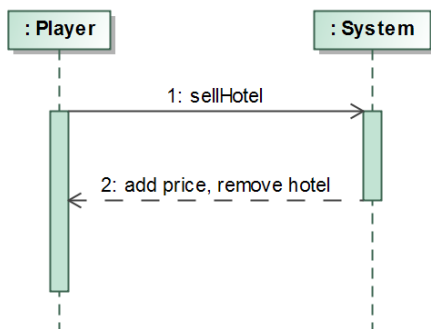
interaction UC05(buyHotel) [UC05(buyHotel)]



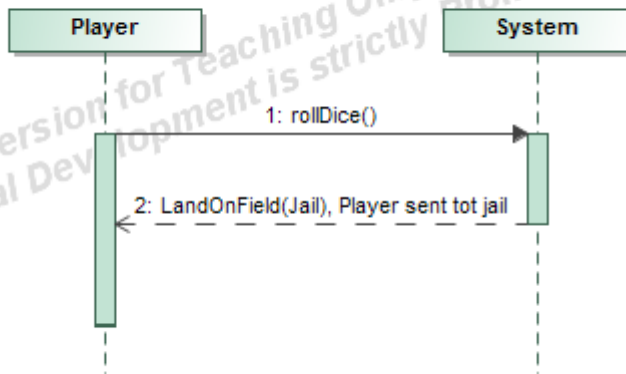
interaction UC06(sellHouse) [UC06(sellHouse)]



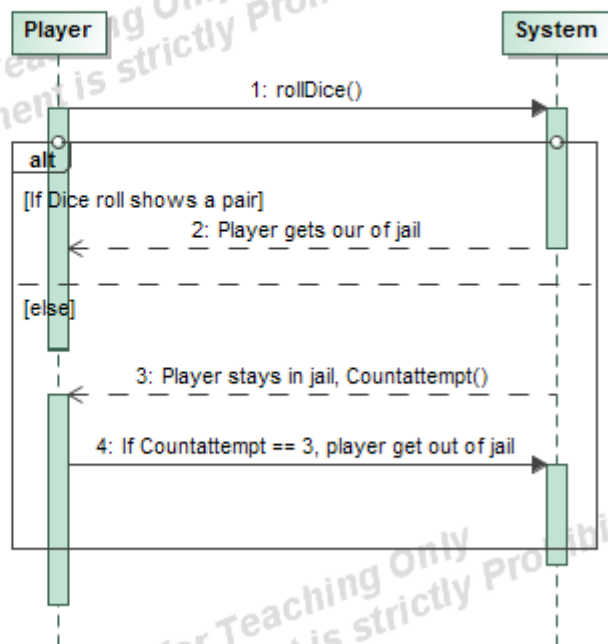
interaction UC07(sellHotel) [UC07(sellHotel)]



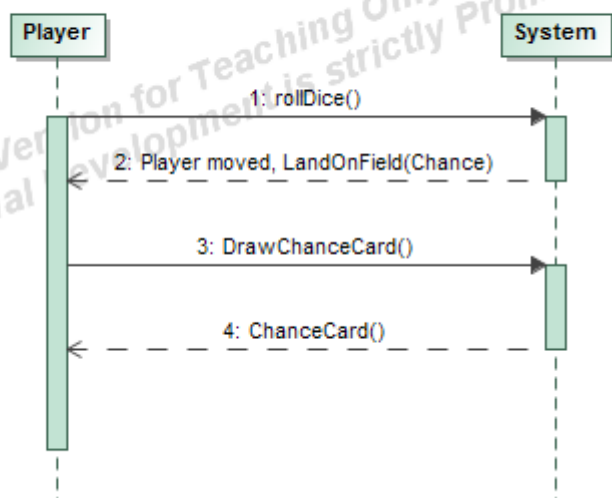
interaction SSD [ SSD08(Go to Jail)]



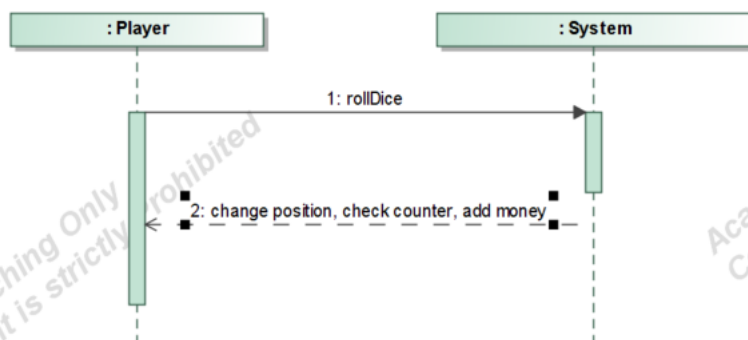
interaction SSD [ SSD09 (Get out of jail)]



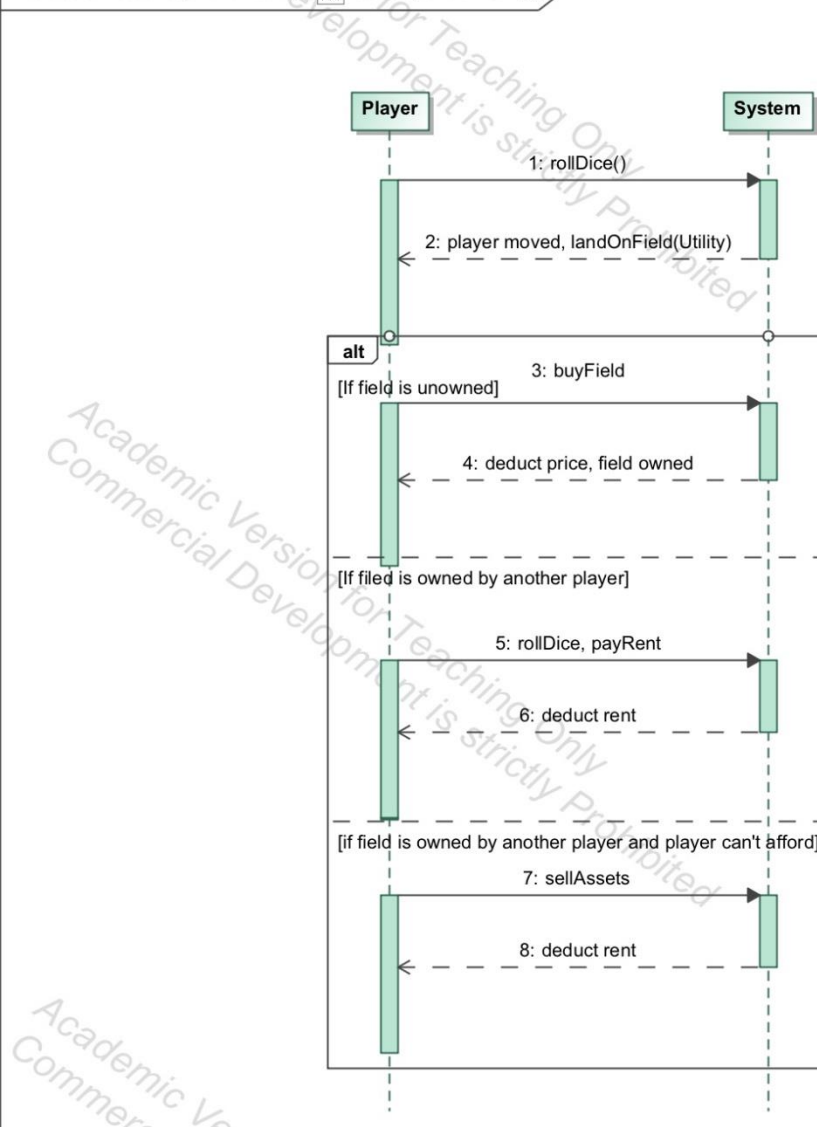
interaction SSD [SSD10 (Draw Chance)]



interaction UC11(passGO) [UC11(passGO)]



interaction USC13(land OnUtility) [1 USC13(land OnUtility)]



interaction [1 UC14 (payTax)]

