Projektopgave efterår 2014 - jan 2015

02312-14 Indledende programmering, 02313 Udviklingsmetoder til IT-Systemer og 02315 Versionsstyring og testmetoder.

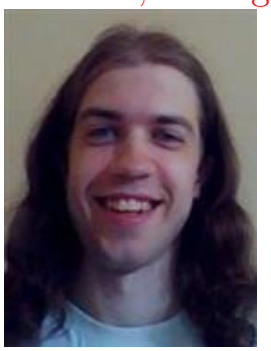
Projekt navn: del3
Gruppe nummer: 57
Afleveringsfirst: Søndag den 30/11 2014 Kl. 20:00

Denne rapport er afleveret via Campusnet (der skrives ikke under)
Denne rapport indeholder 31 sider incl. denne side.

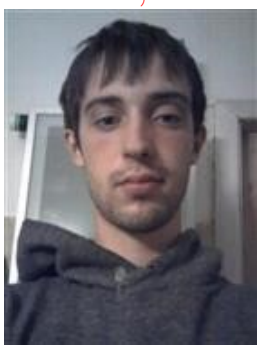
Studienummer, Efternavn, Fornavn Underskrift

S144852, Jørgensen, Kåre _____
Kontakt person (Projektleder)

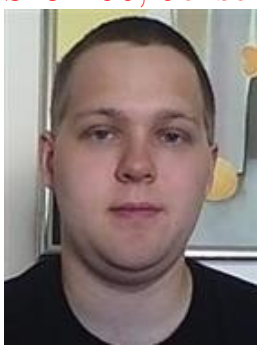
S144869, Bordig, Daniel _____



S144877, Hansen, Martin _____



S134296, Jensen, Nicklas _____



S144839, Dirir, Liban _____

S134834, Justesen, Mads _____

S133974, Hansen, Nicolai

CDIO 3							
Time-regnskab	Ver. 2014-30-11						
Dato	Deltager	Design	Impl.	Test	Dok.	Andet	Ialt
18/11/2014	Kåre	1					1
20/11/2014	Nicolai	2					2
22/11/2014	Nicolai		2				2
23/11/2014	Nicolai		1	1			2
24/11/2014	Nicolai		2				2
24/11/2014	Kåre		2				2
24/11/2014	Mads	2			1		3
24/11/2014	Nicklas		1				1
24/11/2014	Daniel		1				1
26/11/2014	Kåre		2		1		3
26/11/2014	Mads	1	2				3
27/11/2014	Daniel				2		2
28/11/2014	Kåre				2		2
28/11/2014	Daniel			2			2
29/11/2014	Nicolai	2					2
29/11/2014	Liban	2					2
29/11/2014	Daniel				2		2
27/11/2014	Martin	1					1
28/11/2014	Martin				2		2
28/11/2014	Nicklas			2			2
30/11/2014	Liban		1				1
30/11/2014	Kåre				2		2
30/11/2014	Daniel			2	1		3
30/11/2014	Nicklas			2			2
30/11/2014	Nicklas	2			1		3
30/11/2014	Nicolai			1	2		3
30/11/2014	Mads	1					1
	Sum	14	14	10	16	0	54

Indhold

1 Introduction

The purpose of this report is to provide our customer information and an overview about the program we have been asked to make. Through the report we will show that the customers' requirements and specifications are met, and ended up as the game the customer asked for.

The report is a review of the project's progress. We had the task of building further on a game we have previously made for the same customer. The game is a Monopoly-like game in a smaller format, with fewer rules. The requirements for the game as well as the system requirements and installation of the game will be gone through, so anyone can come to understand and get started with the game.

A domain- and BCE-model including a system-sequence diagram helps to provide a general overview of what the game can do, and what features are present. Next, the code's structure and design will appear in a design-sequencediagram and a design-classdiagram. There will be explanatory text for the diagrams, to elaborate further on what the diagrams show. To get an overview on how the code has been written and "come to life", there will be access to our repository.

At the end there will be a conclusion to pick up on project's progress and on what is being presented through the report.

2 Requirements

Functional requirements

- A game with two dice, played between 2-6 players
- Shall be able to simulate a dice cup with an easy changeable amount of dice, and sides on the dice
- The players start with a score of 30.000, and the game ends when one person is left with money on his/her account
- Shall simulate a board with 22 fields, with varying effects on the score of the players:
  - Territory
    - Can be bought by the player if the field is not owned
    - If the player hits a territory already bought, he/she has to pay a tax to the owner
  - Refuge
    - The player is paid a bonus
  - Tax
    - The player either loses a predetermined amount of cash, or 10 % of their entire score
  - Labor Camp
    - Can be bought by the player if the field is not owned
    - The player has to pay a fee to the owner of the camp if it is already bought
    - The player throws the dice, and that number multiplied by 100, and by the amount of labor camps owned by the same player, is what the player has to pay
  - Fleet
    - Can be bought by the player if the field is not owned
    - The player has to pay a fee to the owner of the fleet if it is already bought
    - The amount is determined by how many fleets the owner has:
      1. 500

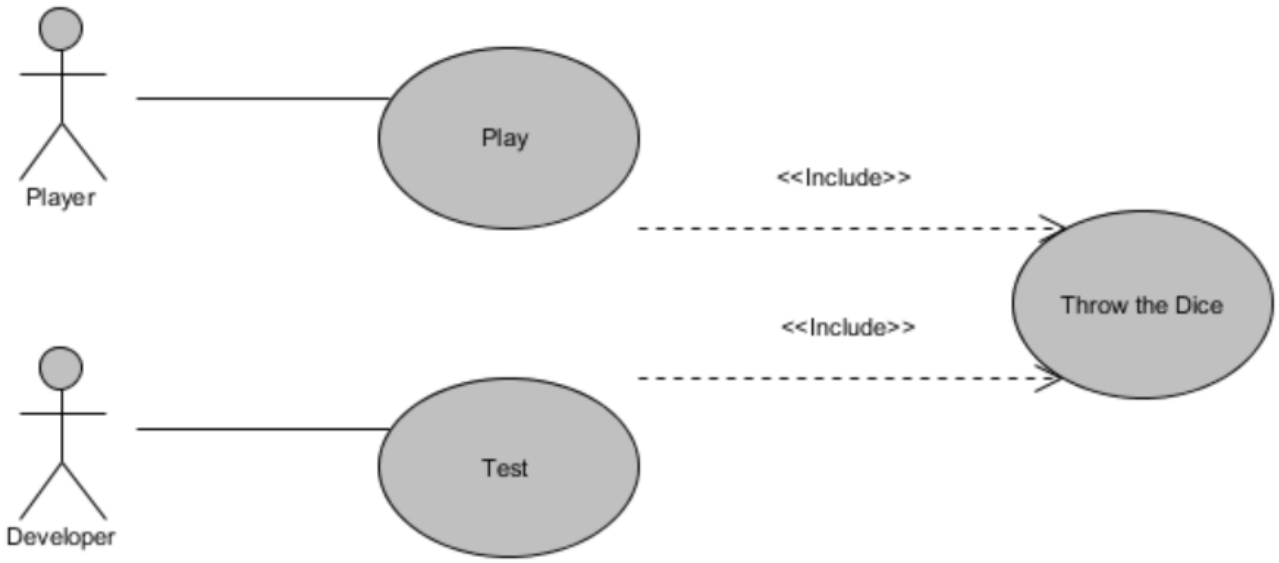
2. 100
3. 2000
4. 4000

- The players move in circles around the board
- Shall be easy to translate to other languages

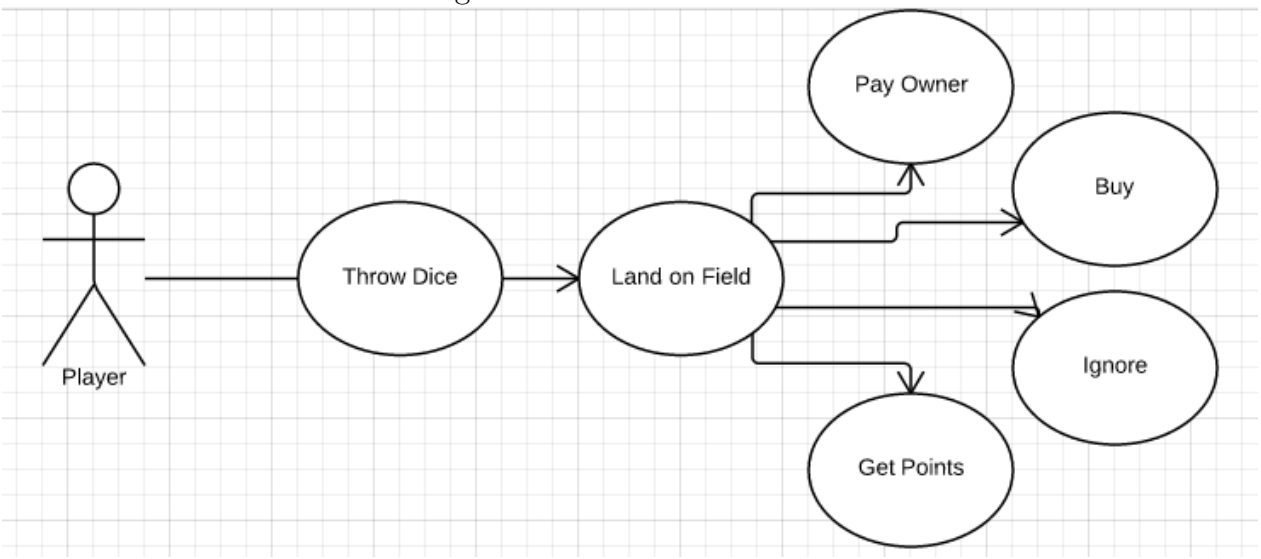
Non-functional requirements

- Shall be able to run on the machines (Windows) in the databars on DTU
- The result of throws must be shown immediately, ignoring the first throw (faster than 333 ms.)
- Shall utilize an already developed GUI
- Normal people must be able to play the game without instructions

3 Use Case Diagrams



Figur 1: UC01 StartGame



Figur 2: UC02 LandOnField

4 Actors

Actor	Player
ID	A01
Semantics	
This actor represents two players playing the game.	

Actor	Developer
ID	A02
Semantics	
This actor represents a developer developing the game.	

5 Use Cases

Abstract Use Case	StartGame
Use Case ID	UC01
Actors	Player Developer
Preconditions	
1. The game has not been started	
Flow of Events	
1. The Use case begins when the player/developer starts the game 2. <UC02 ThrowDice>	
Postconditions	
1. The game has been started	

Use Case	LandOnField
Use Case ID	UC02
Scope	Windows machines with Java installed
Description	The player throws the dice one time, lands on a field, and a specific action is executed based on which field the player landed on
Primary actor	Player
Stakeholders and interests	The player has an interest in playing the game successfully.

Preconditions

1. The game has been started

Flow of Events

1. The Use case begins when the player/Developer throws the dice
2. The dice thrown is added to the position of the player
3. If the player lands on field 1-11
    a. If the field is not owned already
        i. The player can buy the field for a set amount
    b. If the field is already owned
    c. The player pays a fee to the owner
4. If the player lands on field 12 or 13
    a. The player gets paid a bonus
5. If the player lands on field 14 or 15
    a. If the field is not owned already
        i. The player can buy the field for a set amount
    b. If the field is already owned
    c. The player pays a fee, calculated by a throw of the dice, multiplied by 100, and by the number of this type of field the owner has
6. If the player lands on field 16 or 17
    a. The player has to either pay a set amount, or 10 % of his/her entire wealth
7. If the player lands on field 18-21
    a. If the field is not owned already
        i. The player can buy the field for a set amount
    b. If the field is owned
        i. The player pays a fee based on the number of this type of field the owner has

Postconditions

1. The player has had one turn, and the dice is handed to the next player either with
    a. A score above 0, meaning the player is still in the game
    b. A score of 0, meaning the player has gone bankrupt, and is no longer in the game
2. The player is the last one in the game with a score above 0, and wins the game

6 Word Analysis

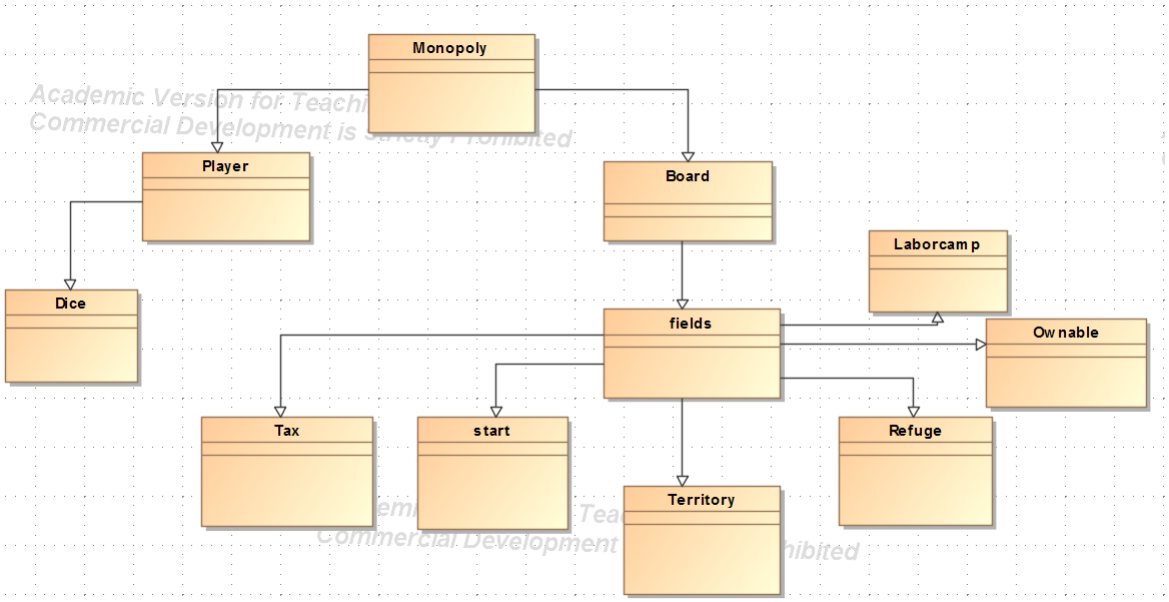
Nouns

- player1-6
- dice1
- dice2
- diceSum
- won
- fieldPoints
- fieldText
- account

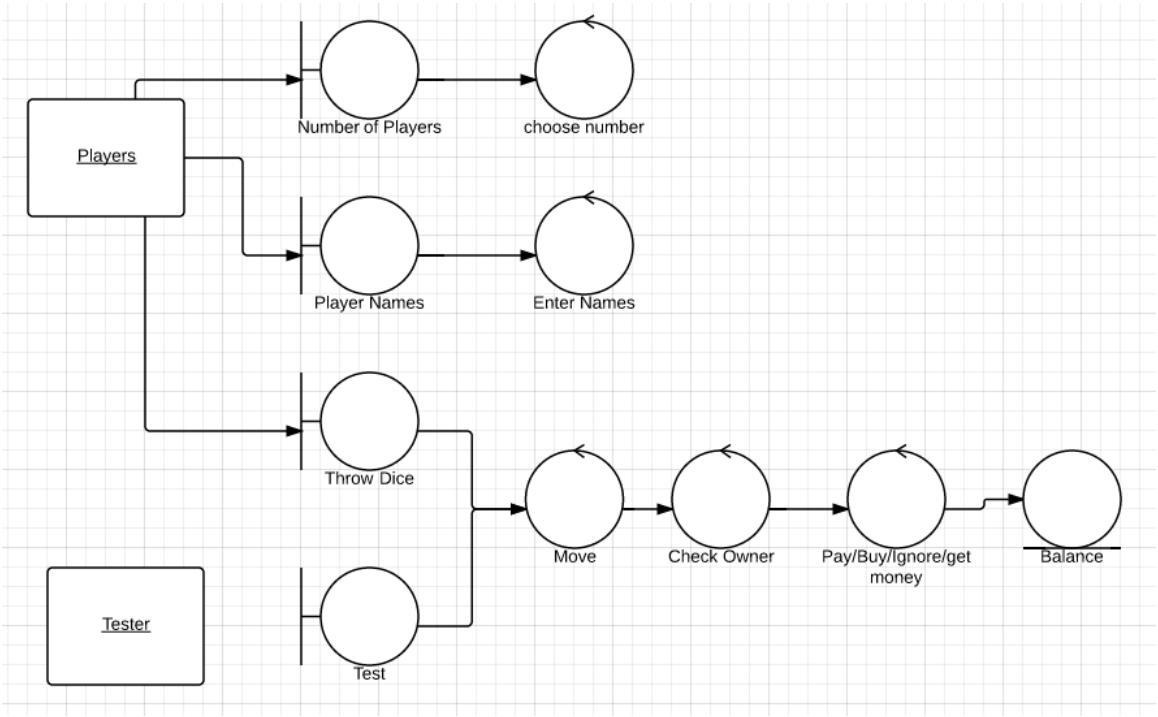
Verbs

- addPoints
- throwDice
- getSum
- setDice (GUI)
- showMessage (GUI)
- addPlayer (GUI)
- setCar (GUI)
- removeCar (GUI)
- setBalance (GUI)
- getPosition
- setPosition
- getScore
- getFieldText
- getFieldPoints

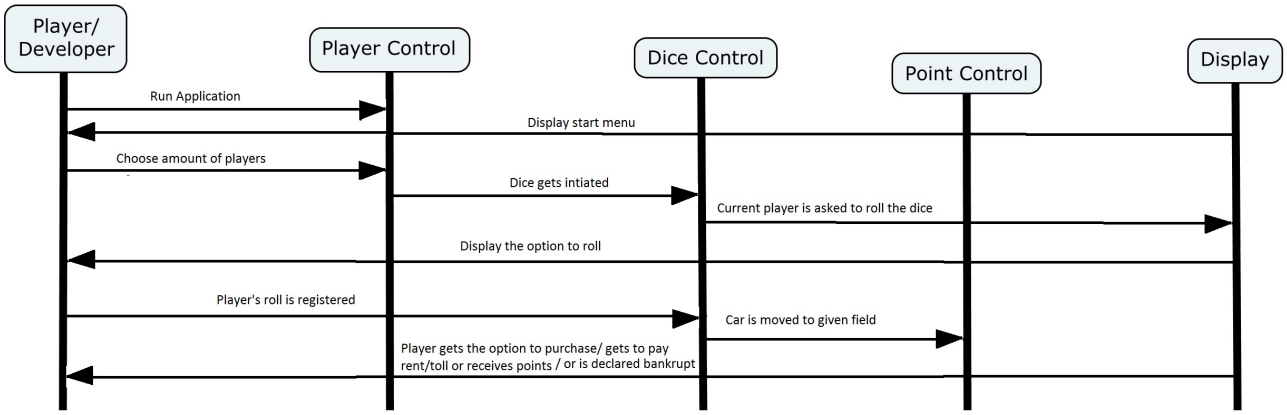
7 UML Diagrams and Models



Figur 3: Domain Model

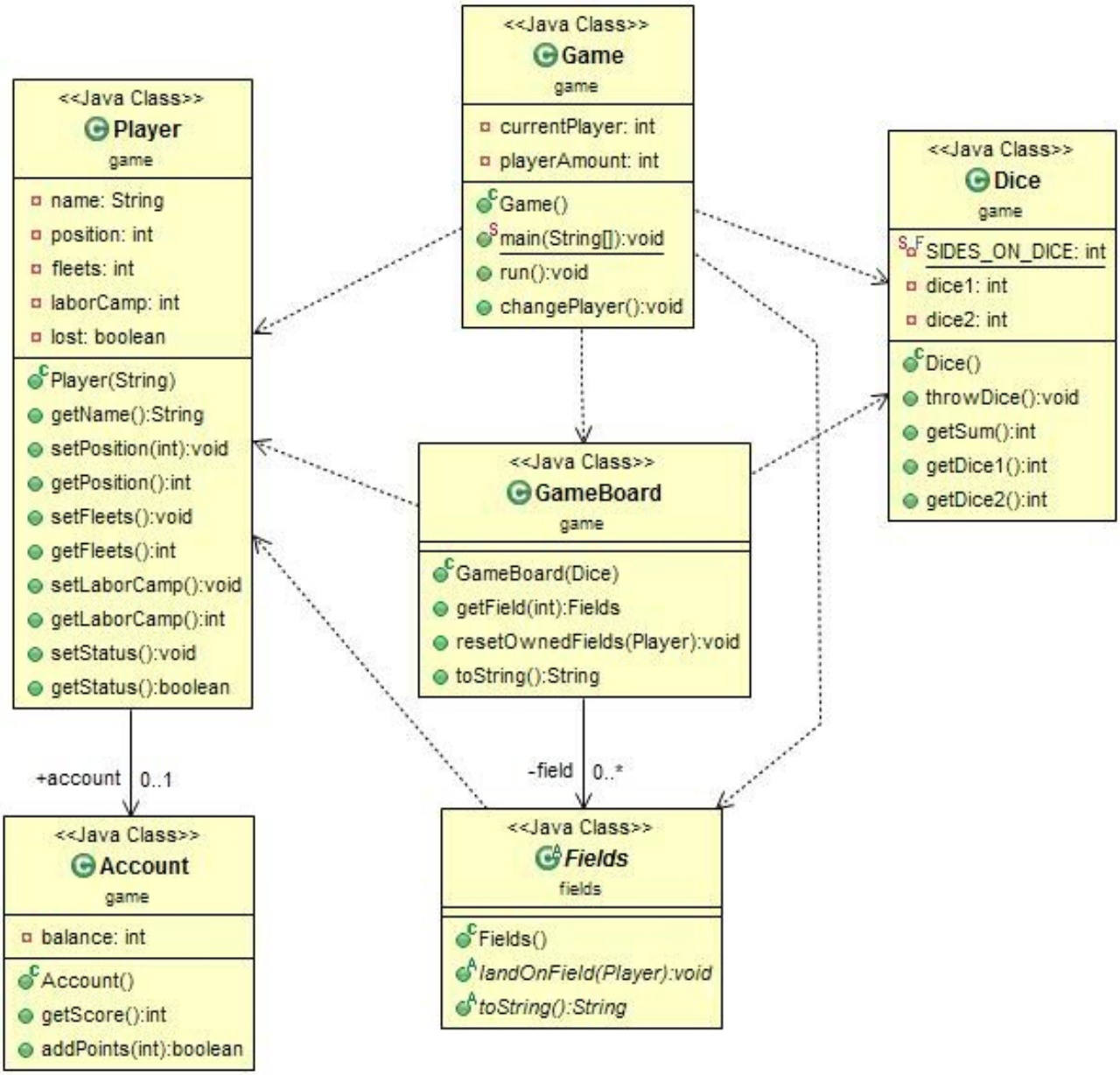


Figur 4: BCE Model



Figur 5: System Sequence Diagram

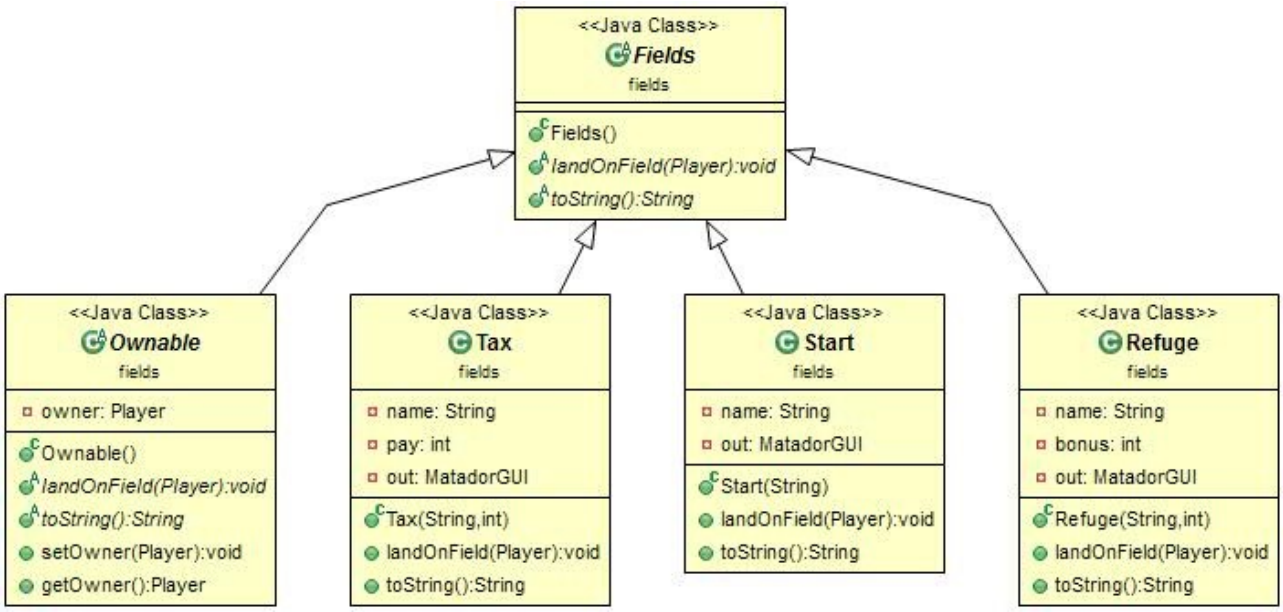
The diagram illustrates how the program runs when a player uses it. The player starts the game with a certain amount of points. When the player throws the dice, changes will occur to his points, and when all but one player has 0 points, the program tells that player that he has won.



Figur 6: Class Diagram 1: Game

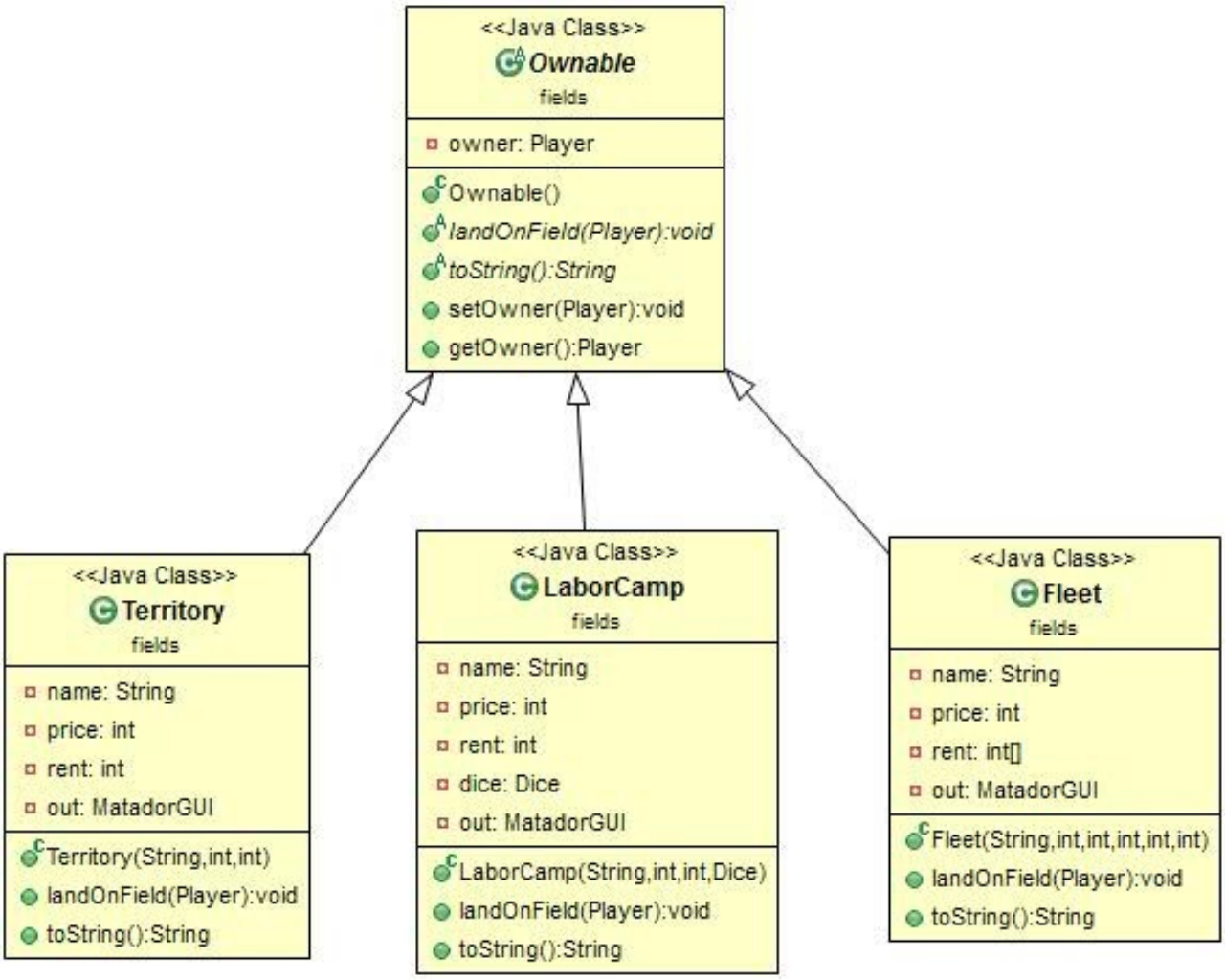
We have chosen to split the class diagram in 3 pieces to make it less cluttered and more readable. The first diagram shows the classes of the game-package. We've taken the Player-class from CDIO2 and extended it with some more attributes and methods. We've added two integers, one to hold the number of owned fleets and one to hold the number of owned labor camps. For each of these integers, we have created two associated method. The method setFleets() increases the number of owned fleets by one, while getFleets() returns the number of owned fleets. Same goes for setLaborCamp() and getLaborCamp() just with owned Labor camps

instead. The Fields-class is and abstract class and contains the abstract methods landOnField() and toString() which are specified in the inherited classes shown in the two next diagrams.



Figur 7: Class Diagram 2: Fields

Ownable, Tax, Start and Refuge are all classes inherited by the Fields-class and each of these classes inherit the landOnField() method which specifies what happens when a player lands on that type of field.



Figur 8: Class Diagram 3: Fields

In the same manner as before, Territory, LaborCamp and Fleet are inherited by Ownable. What separates these type of fields from the others is that they have an owner in terms of a player. The owner can be assigned with the setOwner() method and returned with the getOwner() method.

8 Test

Our first test is to see if a player moves the right amount of spaces on the game board. The player is supposed to move the correct number of space, equal to the dice sum, and is also supposed to move back to the starting field when passing field 21.The test will show if the player landed on the right field.

Test Case 1: Moves right amount of spaces (positive test)

Preconditions:

- Set players position to field 18.

Test:

- Move player 10 spaces.

Postconditions:

- Check that the player is on the right field (field 6).

```
//Preconditions
Player player = new Player("Test");
player.setPosition(18);
System.out.println("player.getPosition(): " + player.getPosition());

//Test
player.setPosition(10);

//Postconditions
if (player.getPosition() == 6)
    System.out.println("Postcondition 1: OK");
else
    System.out.println("Postcondition 1: Fejl");
```

The players starting position is set to field 18, which is one of the last fields on the board and so the player should move back to the start field and on, if the next throw is over 3.
We set the player to move 10 spaces, and so we expect the player to end up on field 6. The test will print the players' position before the player is

moved, and then print the result of the test.

```
player.getPosition(): 18
Postcondition 1: OK
```

Test report 1	
Test Case 1: Moves right amount of spaces	Date: 19-11-2014 Commit: e997472
Precondition: Set position	OK
Test: Move player	OK
Postcondition: Check if the player is on the right field	OK

Our second test is to check if the balance of an account can be negative or stops a 0. The account will be set to a low score and we will then try to take more money away from the account than it has.

Test Case 2: Can the balance be negative (negative test)

Preconditions:

- Set account score to 400.

Test:

- Subtract 500 from the account.

Postconditions:

- Check that the account score has been set to 0.

```
//Preconditions
Account account = new Account();
account.addPoints(-29600);
System.out.println("account: " + account.getScore());

//Test
account.addPoints(-500);

//Postconditions
if (account.getScore() == 0)
    System.out.println("Postcondition 1: OK");
else
    System.out.println("Postcondition 1: Fejl");
```

The account is set to 400 so that -500 would make the score go negative. We subtract the 500 from the account to see what happens. The test will print the score before we subtract the 500 and then print the result of the test.

```
account: 400
Postcondition 1: OK
```

Test report 2	
Test Case 2: Balance can't be negative	Date: 30-11-2014 Commit: 86522c4
Precondition: Set balance	OK
Test: Subtract amount	OK
Postcondition: Check that the account is at 0	OK

Our third test is to check if the dice sum will be from 2-12, and not less or more. The dices will be thrown a 100 times and any amount that's too high or low will be counted.

Test Case 3: Dice sum results (positive test)

Preconditions:

- Create a dice.
- Declare 0 wrong throws

Test:

- Throw dices a 100 times.
- Keep track of wrong throws

Postconditions:

- Check that no dice sum was wrong.

```
//Preconditions
Dice dice = new Dice();
int WrongThrow = 0;

//Test
for (int i = 0; i < 100; i++){
dice.throwDice();
if (dice.getSum() < 1 || dice.getSum() > 12){
    WrongThrow++;
    }
}
//Postconditions
if (WrongThrow == 0)
    System.out.println("Postcondition 1: OK");
else
    System.out.println("Postcondition 1: Fejl");
```

Before the dices are thrown there is of course no wrong throw. The dices will then be thrown a 100 times and then a check will see if any of dice

sums are lower than 2 or higher than 12. We expect the dice sum to be from 2-12 at all times, and the test will print the result.

```
Postcondition 1: OK
```

Test report 3	
Test Case 3: Dices throws correctly	Date: 30-11-2014 Commit: 142b2ff
Precondition: Create dice	OK
Test: Throw dice and check the sum	OK
Postcondition: All dice sums was from 2-12	OK

Test Case 4: Reset owned fields

Preconditions:

- Set field 1 (Tribe encampment) owner to player 1.
- Set field 16 (The pit) owner to player 1.
- Set field 18 (Privateer armada) owner to player 1.

Test:

- Reset player 1 owned fields.

Postconditions:

- Verify that field 1 owner is null.
- Verify that field 16 owner is null.
- Verify that field 18 owner is null.

```
// Preconditions
((Ownable) gameboard.getField(1)).setOwner(player);
System.out.println("gameboard.getField(1).getOwner().getName: "
        + ((Ownable) gameboard.getField(1)).getOwner().getName());
((Ownable) gameboard.getField(16)).setOwner(player);
System.out.println("gameboard.getField(16).getOwner().getName: "
        + ((Ownable) gameboard.getField(16)).getOwner().getName());
((Ownable) gameboard.getField(18)).setOwner(player);
System.out.println("gameboard.getField(18).getOwner().getName: "
        + ((Ownable) gameboard.getField(18)).getOwner().getName());

// Test
System.out.println("gameboard.resetOwnedFields(player 1)");
gameboard.resetOwnedFields(player);

// Postconditions
if (((Ownable) gameboard.getField(1)).getOwner() == null)
    System.out.println("Postcondition 1: OK");
else
    System.out.println("Postcondition 1: Fejl");
if (((Ownable) gameboard.getField(16)).getOwner() == null)
    System.out.println("Postcondition 2: OK");
else
    System.out.println("Postcondition 2: Fejl");
if (((Ownable) gameboard.getField(18)).getOwner() == null)
    System.out.println("Postcondition 3: OK");
else
    System.out.println("Postcondition 3: Fejl");
```

This test is supposed to show that each type of ownable fields can be reset when necessary. We've picked field 1 (territory), field 16 (labor camp) and field 18 (fleet). Each of mentioned fields has been assigned player 1 as their owner. Resetting player 1's field should result in their owner being null. If that isn't the case, a bug need to be located and fixed.

```
gameboard.getField(1).getOwner().getName: Player 1
gameboard.getField(16).getOwner().getName: Player 1
gameboard.getField(18).getOwner().getName: Player 1
gameboard.resetOwnedFields(player 1)
Postcondition 1: OK
Postcondition 2: OK
Postcondition 2: OK
```

Test report 4		
Test Case 4: reset owned fields	Date: 28/11-2014	Date: 29/11-2014
Pre 1: set owner field 1	Ok	Ok
Pre 2: set owner field 16	Ok	Ok
Pre 3: set owner field 18	Ok	Ok
Test 1: reset owned fields	Ok	Ok
Post 1: verify owner field 1 is null	Error	Ok
Post 2: verify owner field 16 is null	Error	Ok
Post 3: verify owner field 18 is null	Error	Ok

The test report shows that we had some errors with the postconditions. When we executed the test script, the test would crash and the console would print a NullPointerException. It wasn't a bug in the main code but the test script itself and the bug wasn't too complicated to fix.

9 Inheritance

Inheritance allow us to reduce the amount of code in each class. For instance if we, for example have two very similar classes named Labrador and Pug, and we have similar methods in these two classes, we can reduce the amount of duplicate code by creating a superclass called dog. We can then derive the two classes Labrador and Pug from Dog, meaning we could write all of the methods that are common for both breeds in Dog and all of the methods that are individual in their own class. Each class can only have one super class, whereas a superclass can have an unlimited amount of subclasses.

We have utilized inheritance in our program when creating the fields for our game, because they naturally are very alike.

10 Abstract

Abstract is a type of either method or class. An abstract class is a class, which may or may not contain abstract methods, that cannot be instantiated, but can be subclassed. In the abstract class, abstract methods can be declared which are methods that are not implemented, but simply declared. Usually the subclass implements the methods.

11 GRASP

How responsibility is assigned between the objects in the program

Expert/ Information Expert:

This is accomplished in several, if not all, of the classes, by simply assigning the methods to the methods containing the most relevant data. For example is addPoints, which add points to the account of the player, located in account.java. Likewise is all methods relating to the player located in the player.java file.

Creator:

Our creator is game.java, which starts the game, and holds all objects. To manage the fields in the game, we have fields.java, which contains the abstract object landOnField.

Low coupling/High cohesion:

Low coupling is achieved by focusing most of the objects (Player, Dice, Gameboard) in game, and the rest in fields and related classes. Admittedly the coupling between the fields classes and our class MatadorGUI containing all calls to the GUI could be lowered, but we have made a conscious decision to sacrifice some coupling to achieve high cohesion.

We have done this, because we feel it is important to focus all calls to GUI in one class, and it also makes our game that much more easy to translate, due to the text printed to the player all being located in that class. Other than making MatadorGUI to achieve high cohesion, we have also created several other classes to achieve this, among others Player, Account and Dice.

Controller:

Our controller is MatadorGUI, which makes sure that all calls to the GUI does not directly interact with any other classes.

Pure fabrication:

We have made use of pure fabrication several times, among others to create Account, instead of storing all methods related to the player in Player. Other than that, MatadorGUI was also created due to us not having a class to store the calls to the GUI.

Indirection:

Not a lot of indirection is used in our program, due to the simplicity of it, but we have created Fields, which ensures that all calls made to the fields of the game, is not done directly in our main class Game.

12 Fieldclass and landOnField

A fieldclass are the specific fields. In the game, no 2 fields are the same. Every single field, have a purpose different from the other fields. For example, there is a "tax"field. That field have two main purposes. Either you can pay 10% of your currency, or you can pay 4000 points. No other field does this.

The landOnField part makes sure that, the computer knows witch field the player has landed on. So the fieldclass can begin. Without the landOnField, the fieldclass wouldn't work.

13 Configuration

System requirements for the provided executable jar file:
Windows xp/vista/7/8
Java
Java SE Development Kit 7

Additional system requirements, and guide for compiling the source code:
Eclipse Luna

- Open Eclipse
- Click File
- Import
- Existing projects into workspace
- Browse
- Choose CDIO3 provided in the archive
- Click finish
- Select CDIO3 in Project explorer
- Select src
- Select game
- Double click game.java
- Run (ctr+f11)

14 Conclusion

We have made the matador-like game, with the asked requirements by the customer, and the game works fine. All of the demanded features are present, so the game is similar to the product that was asked for. We had a problem towards the end of the project, when a player goes bankrupt still owned the fields he had bought, and therefore also got money from other players without being part of the game. We didn't have a clear definition of what happens to a player when he goes bankrupt. Our last mail with questions to the customer did not get answered before we felt that we had to solve the problem and focus on the rest of the tasks.

We didn't use the time schedule as much as intended. Later on it was hard to remember for each person what day you did what, and how much time was spent on it. Therefore, the schedule is not 100% accurate, and might be missing some entries.

As for the game going forward it will be possible to extend it to a whole game of Monopoly, with 40 fields and the remaining rules. You could also go a different route, and make some other rules or expansions to the game, to create a more unique game.