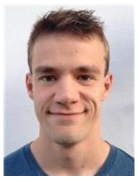
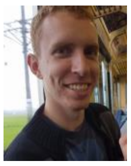
CDIO part 3

02312-14 Introductory programming, 02313 Development Methods for IT-Systems & 02315 Version Control and Test Methods.

Project title: CDIO part 3
Group number: 17
Deadline: Friday 25th November 2016 - 23:59
Version: 1.0

This report contains 52 pages, including this page.

Student nr., Surname, First name	**Signature**
S165248, Gadegaard, Theis 	____________________
S165232, Helstrup, Freya	____________________
S165208, Højsgaard, Tobias	____________________
S165209, Jønsson, Danny	____________________
S165227, Petersen, Gustav Hammershøi 	____________________
S165237, Poulsen, Joakim Thorum	____________________

Work distribution

Work distribution	Ver. 2016/11/25							
Date	**Participant**	**Analysis**	**Design**	**Impl.**	**Test**	**Doc.**	**Other**	**Total hours**
2016/11/25	Danny	0,5	0,5	4,5		4	1	**10,5**
2016/11/25	Freya	3	1	6,5	7,5	2	1,5	**21,5**
2016/11/25	Gustav	4	0,5	3	1	5	0,5	**14**
2016/11/22	Joakim	1		2,5				**3,5**
2016/11/25	Theis	1	0.5	4		1,5	4	**10,5**
2016/11/25	Tobias		3,5	7,5	1	6		**18**
	Sum	**9,5**	**5,5**	**28**	**9,5**	**18,5**	**7**	**78**

Resumé

The objective of this project is to create a software product in the form of a board game, based on the customer description given by IOOuterActive. This report’s purpose is to provide the documentation necessary to understand the thought process behind the software’s design and how it’s implemented.

Table of contents

1 Inception

1.1 Vision

Our task is to make a game in which the players move around a board and are able to land on different kinds of fields. The players lose if they end up having a balance of 0, and the last player remaining after all other players have lost will win.

2 Analysis

2.1 Requirements specification

2.1.1 Functional requirements

1.1	The game shall have two to six players.
1.2	Player 1 shall always start.
1.3	The game shall consist of a board with 21 fields, numbered from 1 to 21.
1.4	The players shall start with a balance of 30,000.
1.5	The players shall, in turns, throw two dice.
1.6	The player shall have a piece on the board, which will be moved around on the board.
1.7	The player shall move his piece the same number of fields as the sum of the dice thrown.
1.8	The game shall end after all players, except one, have lost all of their money.
1.9	The last remaining player, after all other players have lost their money, shall be the winner.
1.10	The board shall consist of the following fields having the specified type: 1. Walled city — Type: Refuge 2. Goldmine — Type: Tax 3. Tribe Encampment — Type: Territory 4. Second Sail — Type: Fleet 5. Crater — Type: Territory 6. Mountain — Type: Territory 7. Huts in the mountain — Type: Labor camp 8. Cold Desert — Type: Territory 9. Sea Grover — Type: Fleet 10. Black cave — Type: Territory 11. The Werewall — Type: Territory 12. Monastery — Type: Refuge 13. Caravan — Type: Tax 14. The Buccaneers — Type: Fleet 15. The pit — Type: Labor camp

	16. Mountain village Type: Territory 17. South Citadel Type: Territory 18. Palace gates Type: Territory 19. Privateer armada Type: Fleet 20. Tower Type: Territory 21. Castle Type: Territory
1.11	The player shall be able to buy one of the following fields for the price listed, if it is not already owned by another player: ● Tribe Encampment: 1000 ● Crater: 1500 ● Mountain: 2000 ● Cold Desert: 3000 ● Black cave: 4000 ● The Werewall: 4300 ● Mountain village: 4750 ● South Citadel: 5000 ● Palace gates: 5500 ● Tower: 6000 ● Castle: 8000 ● Huts in the mountain and The pit: 2500 ● Second Sail, Sea Grover, The Buccaneers and Privateer armada: 4000
1.12	The player shall pay an amount, as specified below, to the owner, if he lands on a field of the type territory and the field is owned by another player: ● Tribe Encampment: 100 ● Crater: 300 ● Mountain: 500 ● Cold Desert: 700 ● Black cave: 1000 ● The Werewall: 1300 ● Mountain village: 1600 ● South Citadel: 2000 ● Palace gates: 2600 ● Tower: 3200 ● Castle: 4000
1.13	The player shall pay the owner the sum of the dice thrown multiplied by 100 multiplied by the number of labor camps owned by the same owner, if he lands on a field of the type labor camp owned by another player.
1.14	The player shall pay a specified amount to the owner if he lands on a field of the type fleet owned by another player. The amount is based on how many fleets that the owner owns: ● The owner owns 1 fleet, the player pays 500 to the owner ● The owner owns 2 fleets, the player pays 1000 to the owner ● The owner owns 3 fleets, the player pays 2000 to the owner

	● The owner owns 4 fleets, the player pays 4000 to the owner
1.15	The player shall receive an amount, as listed below, if he lands on a field of the type refuge: ● Walled city: 5000 ● Monastery: 500
1.16	The player shall pay an amount, as listed below, if he lands on a field of the type tax: ● Goldmine: 2000 ● Caravan: 4000 or 10% of all assets* *) The player shall choose between the set amount and the percent (10% of the price of the player's owned fields + 10% of the player's balance).

2.1.2 Usability requirements

2.1	The game shall be playable by any average person without the need of instructions.

2.1.3 Reliability requirements

None

2.1.4 Performance requirements

4.1	The time between the input and result of a dice throw shall have a response time of no more than 333 milliseconds.

2.1.5 Supportability requirements

5.1	The game shall be installable and playable on Windows machines
5.2	The game shall be translatable by editing only one file.
5.3	The dice may be configurable, so the players can throw more than two dice or a dice with a different side-number than six.
5.4	The game shall be playable by two to six players.

2.1.6 Additional non-functional requirements

6.1	The game shall be developed using the IDE Eclipse and the Java Programming Language

2.2 Noun/verb analysis

We have used the client description to generate this list of nouns and verbs. These we can use in our further investigation and development of the game.

Nouns	Verbs
1. Dice game	1. Throw (dice)
2. Player	2. Play
3. Dice	3. Land on
4. Dice cup	4. Print out
5. Car/object	5. Translate
6. Board	6. Buy field
7. Field	7. Pay rent
8. Type	8. Set/get ownership
9. Territory	
10. Labor camp	
11. Fleet	
12. Refuge	
13. Tax	
14. Balance	
15. Positive/negative effect	
16. Ownership (of a specific field)	
17. Choice (between tax or amount)	
18. Bankruptcy	

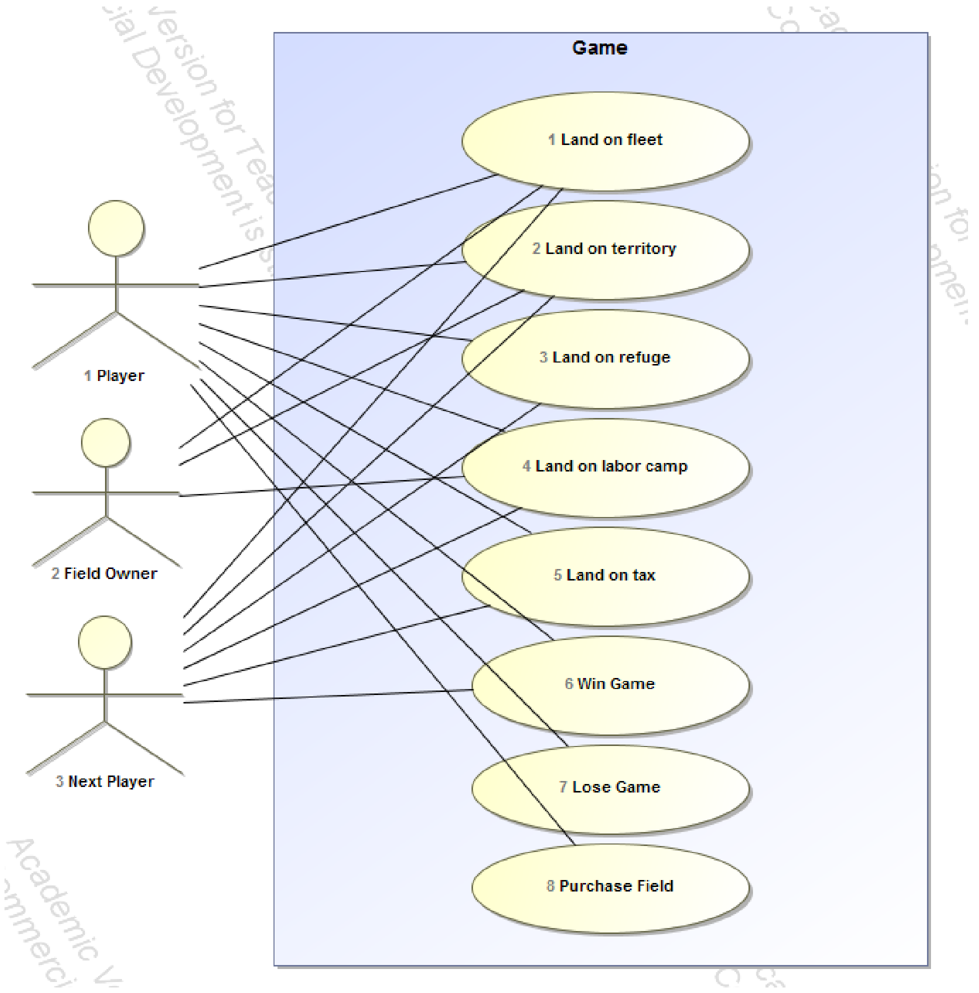
The ‘dice game’ itself consists of a ‘dice cup’, containing two ‘dice’, a ‘board’, made up of 21 ‘fields’, which can have 5 different ‘types’, and 2-6 ‘players’, who have ‘pieces’ on the board. A player shall ‘throw’ the dice and then ‘land on’ a field. A player has the opportunity to ‘buy a field’ and get ‘ownership’ of it.

The field can also have a ‘negative effect’ on the player’s balance if another player owns it, because he then must ‘pay rent’ to the owner.
The client would like the game to be easily ‘translated’, so this we have also noted.

3 Design

3.1 Use cases

3.1.1 Use case diagram



Above is our Use Case Diagram that shows all of our use cases and actors. We have three primary actors, which all are players in the game.

UC1: Land on fleet

Use case ID:	1
Use case name:	Land on fleet
Description:	When the current player lands on one of the four fleet fields, one of two things will happen. 1. If the the field is already owned by another player, the current player will have to pay 500-4000 to the owner, depending on how many fleets the owner has. 2. If no one is owner of the fleet, the current player will be given the option to buy the fleet.
Primary actors:	Player, Field owner, Next player
Secondary actors:	None
Preconditions:	1. It is the player's turn 2. The player rolls the dice and lands on a fleet field
Postconditions:	1. The player has either paid the owner or been given the option to buy the fleet. 2. It is the next player's turn.
Main flow:	1. The player lands on a fleet field 2. No other player has ownership of the field 3. The player is given the option to a. Buy the fleet b. Not buy the fleet 4. The player chooses to buy the fleet 5. The price of the fleet is subtracted from the players balance 6. The player gets ownership of the fleet
Alternate flows:	A2: Another player has ownership of the fleet field 1. The player pays an amount to whoever has ownership of the fleet field. A3: The players balance is less than the price of the fleet field 1. The player is not given the option to buy the field 2. The players balance remains unchanged A4: If the player chooses not to buy the fleet 1. The players balance remains unchanged.

UC2: Land on territory

Use case ID:	2
Use case name:	Land on territory
Description:	Player can buy a field or pays a fee depending on field ownership
Primary actors:	Player, Field owner, Next player
Secondary actors:	None
Preconditions:	1. It is the player’s turn 2. The player rolls the dice and lands on a territory field
Postconditions:	1. The player has either paid the owner or been given the option to buy the field 2. It is the next player’s turn
Main flow:	1. No other player has ownership of the field 2. The player is given the option to 2.1. Buy the territory 2.2. Not buy the territory 3. The player chooses to buy the territory 4. The price of the territory is subtracted from the players balance 5. The player gets ownership of the territory
Alternate flows:	1A 1. The territory is owned by another player 2. Pay the fee defined by the field to the territory owner

UC3: Land on refuge

Use case ID:	3
Use case name:	Land on refuge
Description:	When a player lands on a refuge field the player will receive a bonus of either 500 or 5000 depending on which of the two refuges the player lands on.
Primary actors:	Player, Next player
Secondary actors:	None
Preconditions:	1. It is the player’s turn 2. The player rolls the dice and lands on a refuge field
Postconditions:	1. The player receives a bonus of either 500 or 5000. 2. It is the next player’s turn

Main flow:	1. The player lands on a refuge field 2. The player receives the bonus.
Alternate flows:	None

UC4: Land on labor camp

Use case ID:	4
Use case name:	Land on labor camp
Description:	When a player lands on a labor camp field, one of the two things will happen: 1. The player has to roll the dice and pay the sum of the dice roll times 100 times the amount of labor camps with the same owner, to the owner of the labor camp. 2. If there is no owner of the labor camp, the player will be given the option to buy the labor camp for the price of 2500.
Primary actors:	Player, Field owner, Next player
Secondary actors:	None
Preconditions:	1. It is the player's turn 2. The player rolls the dice and lands on a labor camp field
Postconditions:	1. The player has either paid the owner of the labor camp or been given the option to buy the labor camp. 2. It is the next player's turn
Main flow:	1. No other player has ownership of the field 2. The player is given the option to a. Buy the labor camp b. Not buy the labor camp 3. The player chooses to buy the labor camp 4. The price of the labor camp is subtracted from the player's balance 5. The player gets ownership of the labor camp
Alternate flows:	2A: Another player has ownership of the the labor camp field a. The player rolls the dice. b. The player times the sum of the dice with 100 and times the sum with the number of labor camps with the same owner. c. The player pays the result to the owner of the labor camp. 4A: The player's balance is less than the price of the labor camp field 3. The player is not given the option to buy the field 4. The player's balance remains unchanged

	4B: If the player chooses not to buy the labor camp 2. The player's balance remains unchanged.

UC5: Land on tax

Use case ID:	5
Use case name:	Land on tax
Description:	The player loses an amount of money when landing on a tax field
Primary actors:	Player, Next player
Secondary actors:	None
Preconditions:	1. It is the player's turn 2. The player rolls the dice and lands on a tax field
Postconditions:	1. The player has paid the required amount 2. It is the next player's turn
Main flow:	1. The player pays the amount required by the tax field
Alternate flows:	1A: The player lands on Caravan and gets to choose between paying 4000 or 10% of all his assets.

UC6: Win Game

Use case ID:	6
Use case name:	Win Game
Description:	A player wins if there are no other players remaining in the game.
Primary actors:	Player, Second player
Secondary actors:	None.
Preconditions:	1. There are only two players remaining in the game. 2. Second player loses the game. 3. It is the current player's turn.
Postconditions:	1. A message will be shown, gratulating the winner and giving the choice of exiting to main menu.
Main flow:	1. The current player wins the game.
Alternate flows:	None

UC7: Lose Game

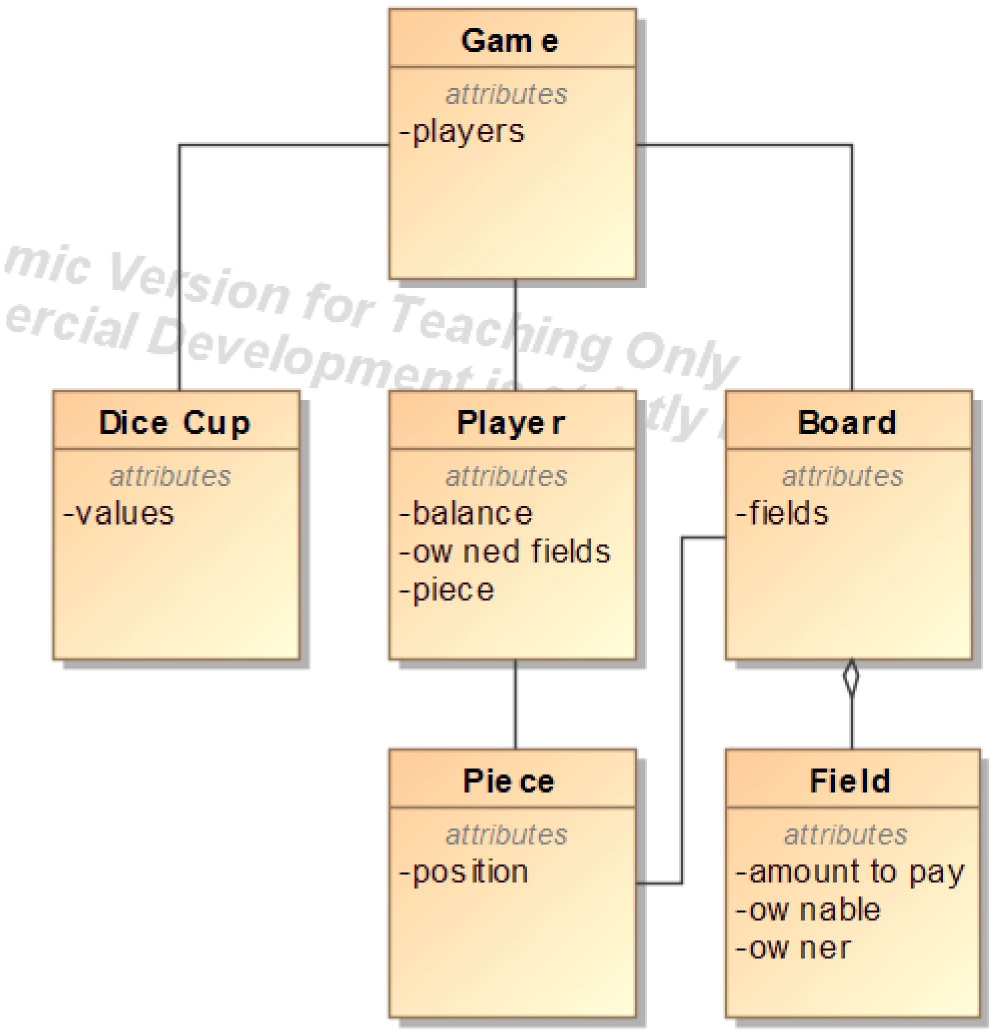
Use case ID:	7
Use case name:	Lose Game
Description:	When a player's balance gets reduced to less than zero, the player is declared bankrupt and loses the game. The games continues as long as there are at least two players in the game.
Primary actors:	Player
Secondary actors:	None
Preconditions:	1. It is the player’s turn 2. The player lands on a tax field or a field owned by another player. 3. The tax or rent reduces the player’s balance to zero
Postconditions:	1. The player is declared as having lost the game 2. The player will not receive rent from his owned fields anymore.
Main flow:	1. The current player lands on a tax field 2. The player’s balance is reduced to less than zero 3. The current player is declared as having lost the game.
Alternate flows:	1. The current player lands on a field owned by another player. 2. The current player does not have enough balance to pay the owner of the field. 3. The remainder of the current player’s balance is payed to the owner of the field. 4. The current player is declared as having lost the game.

UC8: Purchase field

Use case ID:	8
Use case name:	Purchase Field
Description:	A player has the opportunity to buy a field if it is not already owned by another player.
Primary actors:	Player
Secondary actors:	None
Preconditions:	1. It is the player’s turn. 2. The player has landed on a field, which is not owned.

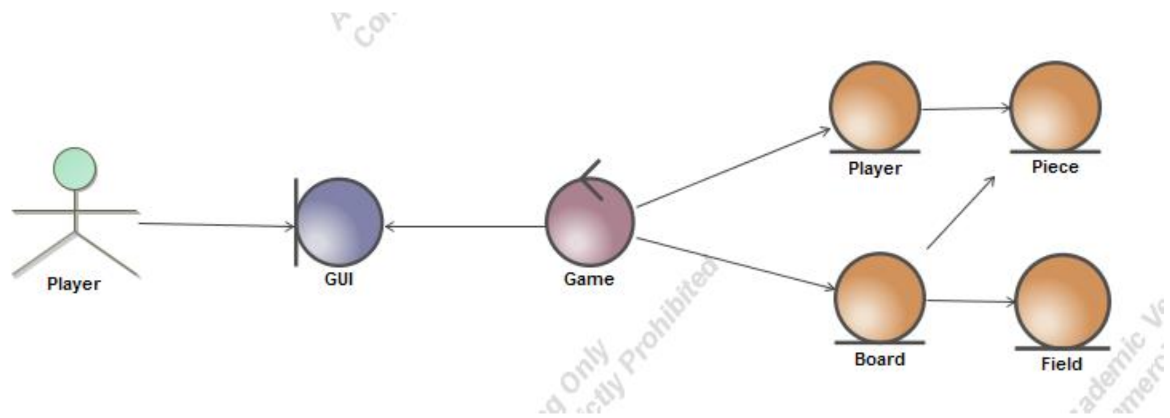
Postconditions:	1. The price of the field is subtracted from the player's balance. 2. The player owns the field.
Main flow:	1. The player gets the option to buy the field he landed on. 2. The player chooses to buy the field.
Alternate flows:	None.

3.2 Domain model



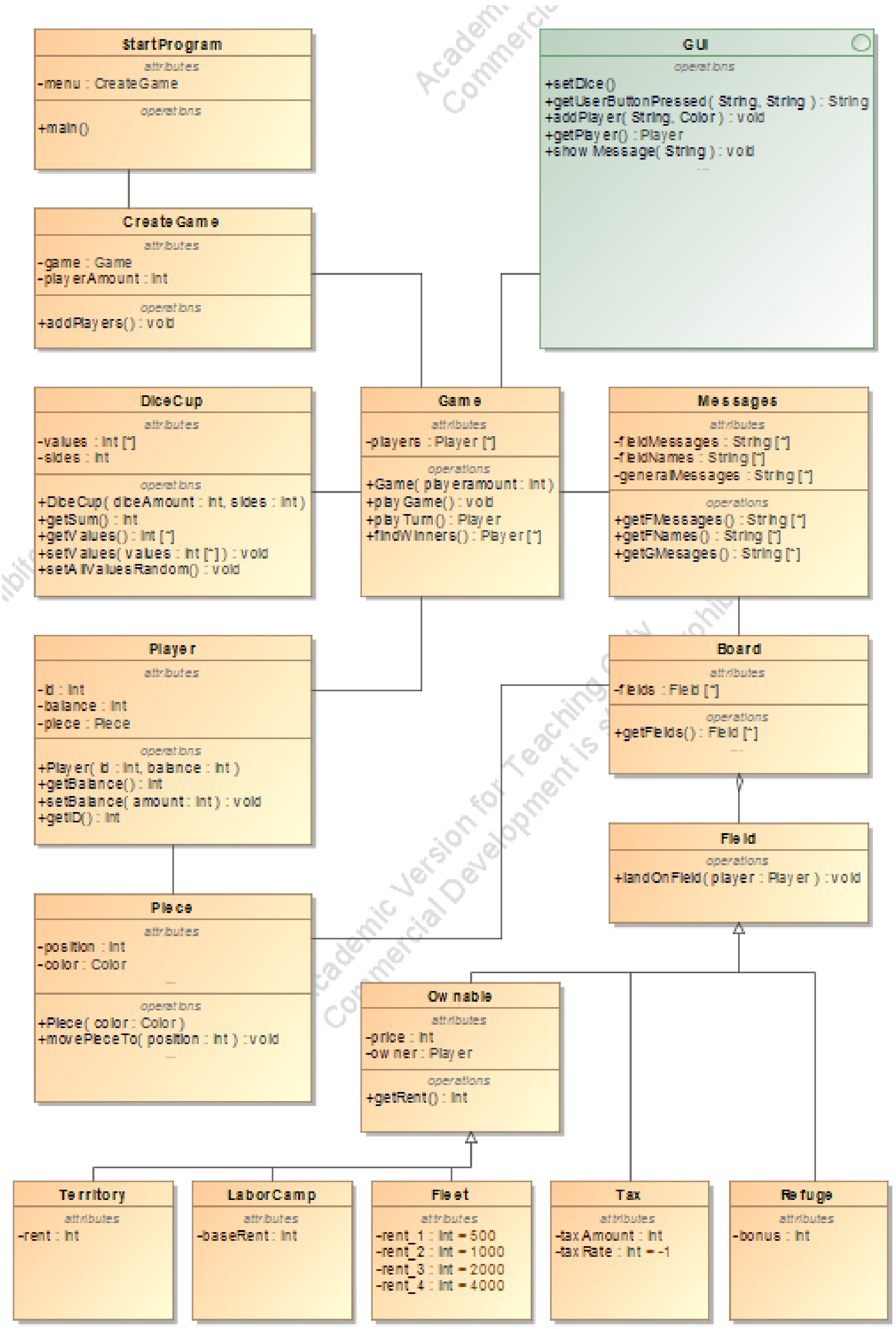
Based on the client description we have made the above definition of the domain. We have a game that consists of a board and a dice cup. It is played by players, that each have a piece. This piece is placed on the board. The board also has several fields.

3.3 BCE diagram



The Boundary Control Entity diagram shows the interaction between our system and the user. The user interacts with our Graphical User Interface and the Game from our problem domain acts as a controller that distributes work and fetches information from the Board, single Field and the Player entities in our domain. At this point the relations are still analytic in a broad sense and some of these relations have been changed slightly in our finished product.

3.4 Analysis class diagram



Based on our system requirements, our domain model and BCE diagram we have made this class diagram. This is our first sketch of how the program should be written.

3.4.1 Responsibilities

StartProgram is, as the name implies, responsible for starting the program and not much more. We use it to run the constructor for our CreateGame class. We have created this class to make our main() method as short as possible.

CreateGame is responsible for all pre-game setup necessary. This includes running the Game constructor, which creates a Game object and the GUI, determining the amount of players that should be in the game and resetting the Game with that amount of players when the user is ready to begin.

Game handles all logic related to game flow. It declares the winner and manages the single turns of each player.

DiceCup is responsible for getting the random values we need to move the players around the board.

Player is responsible for storing information related to each of the players. This means their balance, mainly, but also which piece on the board is theirs and which fields they own.

Piece is a small class that is used to hold the position of the player on the board and to move that position around during a turn.

Board stores all the necessary information on each of the fields we need for our game. It also has a public method used to create the GUI, since it has easy access to the individual fields. We can then run this method from another class.

Field is the superclass in a inheritance hierarchy containing each of our different field types. It serves as the general structure for each of the more specific classes, meaning that each class under it must have a landOnField() method. Each of the classes that have Field as their superclass is responsible for one type of field on the board.

Ownable is a subclass of Field that is worth noting as it also has several subclasses. It generalizes all of the field types that can be owned by the players, and as such contains methods that are relevant to ownership of a field. Each subclass of Ownable is a type of Field that can also be owned by a player.

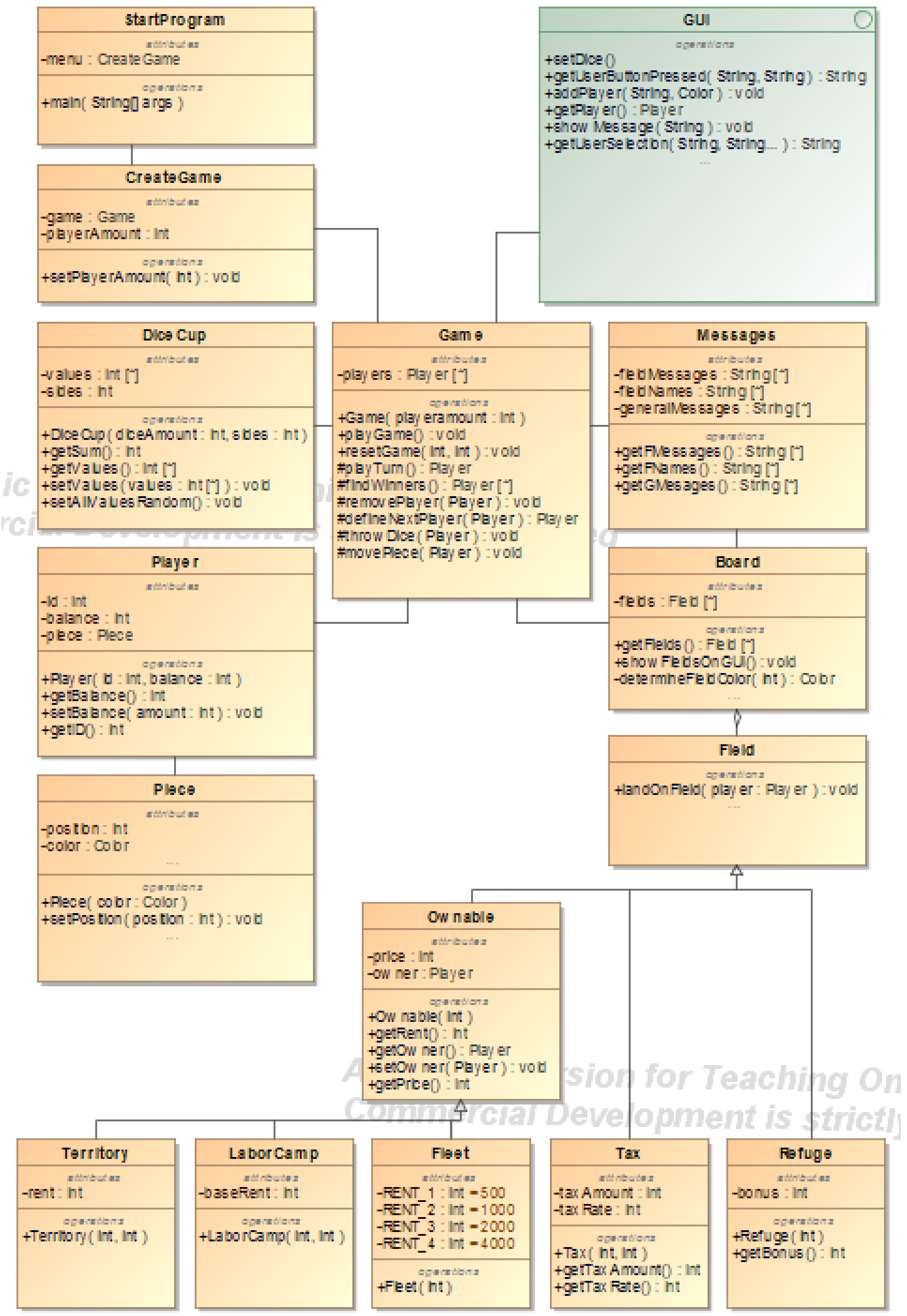
Messages is a class we use to ease translation of the game. It stores all messages that the GUI displays, including all the names of the fields. This ensures that a translator would only have to edit a single file in order to translate the game.

GUI is the graphical user interface. We use an already existing program with different classes and methods for our game, but represent this in our diagrams with a reference to “GUI”.

4 Implementation

4.1 Diagrams

4.1.1 Design class diagram



There are some things worth noting about this diagram.

One this is the inheritance heirarchy between Field and several specific types of field classes including Territory, Fleet and so on. Inheritance means that we can essentially specify a class in the same way that a car is a specific type of vehicle. There are some properties of all of our specific types of field that we can generalize. For example, that they are all on our Board and that they can all be landed on. We use this in practice in our Board class so that we only have to keep an array of fields to determine which field the player has landed on instead of having to make an array for each of the specific field types. Ownable fields can all be bought for a price and all have rent that other players have to pay when they land on one that is not owned by them. Therefore the Ownable class has the method getRent() as each of the specific field types all have different ways of determining this rent.

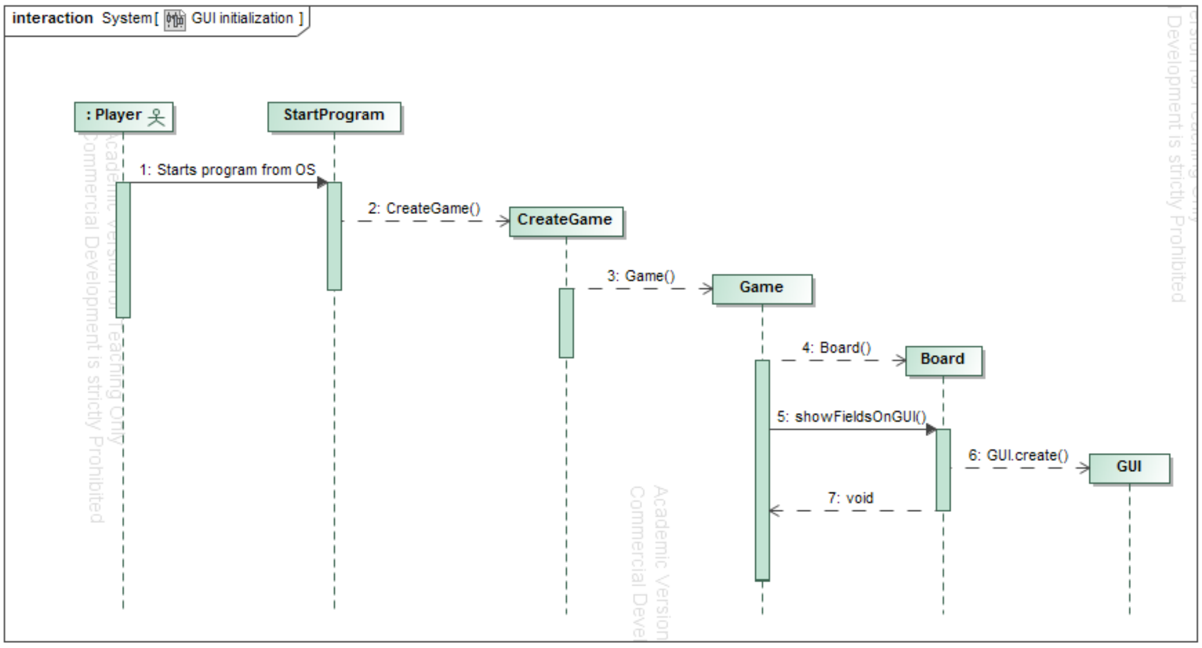
Another point is that Field and Ownable are both abstract classes, although this might not be apparent in the diagram. This means that, alone, they are incomplete and could not function. The landOnField() method in Field is an abstract method, which just means that every subclass of field must implement the method, not that it is a complete method that they can all use. The only reason we have implemented it in field is so that we are sure every subclass of it must have a landOnField() method. Ownable has a similar method, getRent(), which is also abstract and thus shares this property.

The concept that we have a landOnField() method and a getRent() method that is different depending on which field type we land on is called polymorphism and is a part of the GRASP patterns.

4.1.2 Design sequence diagram

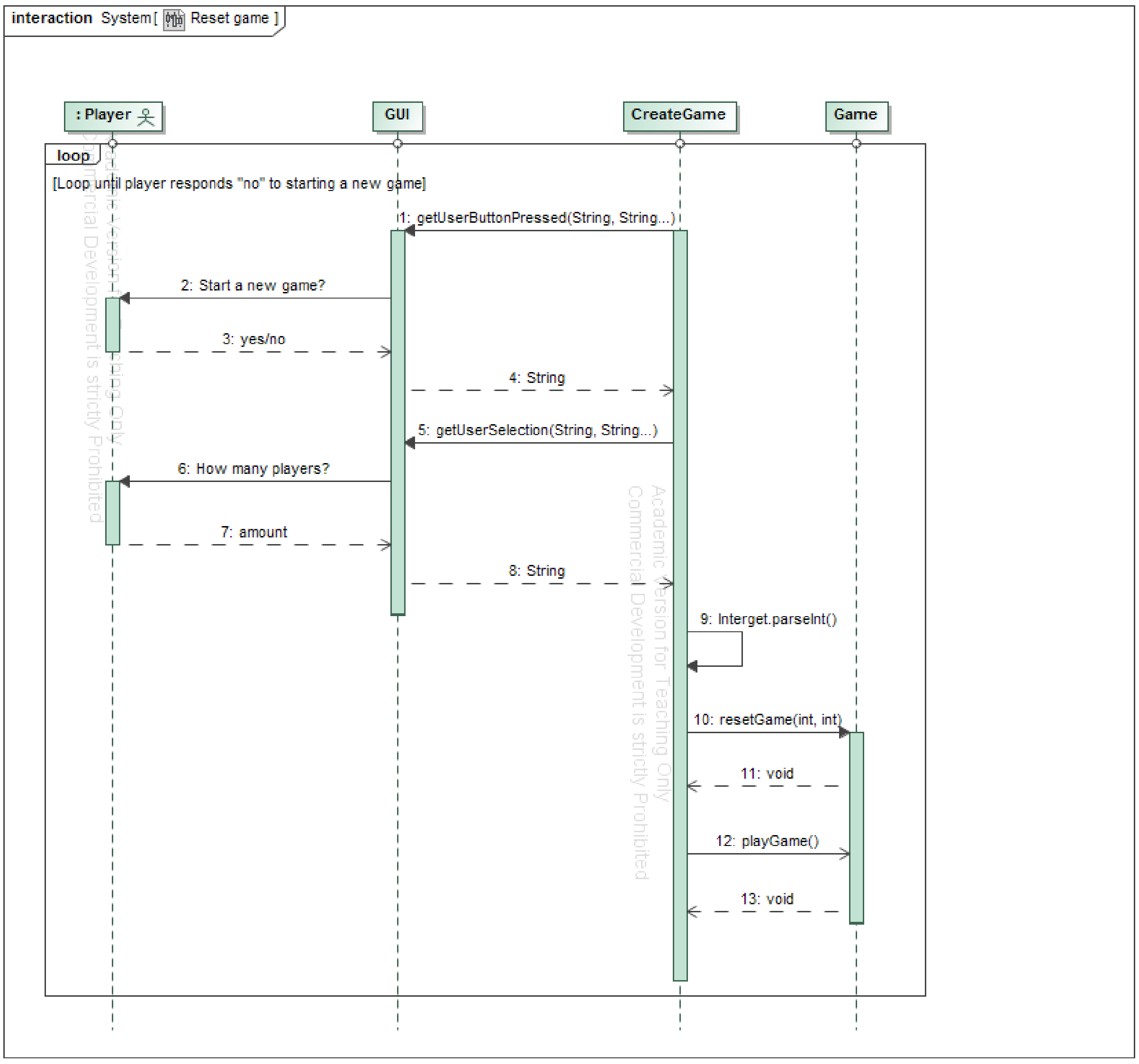
We have split our sequence diagrams up in order to make them easier to comprehend.

Let us start with what happens in the beginning of our program when a user starts it up from the operating system.

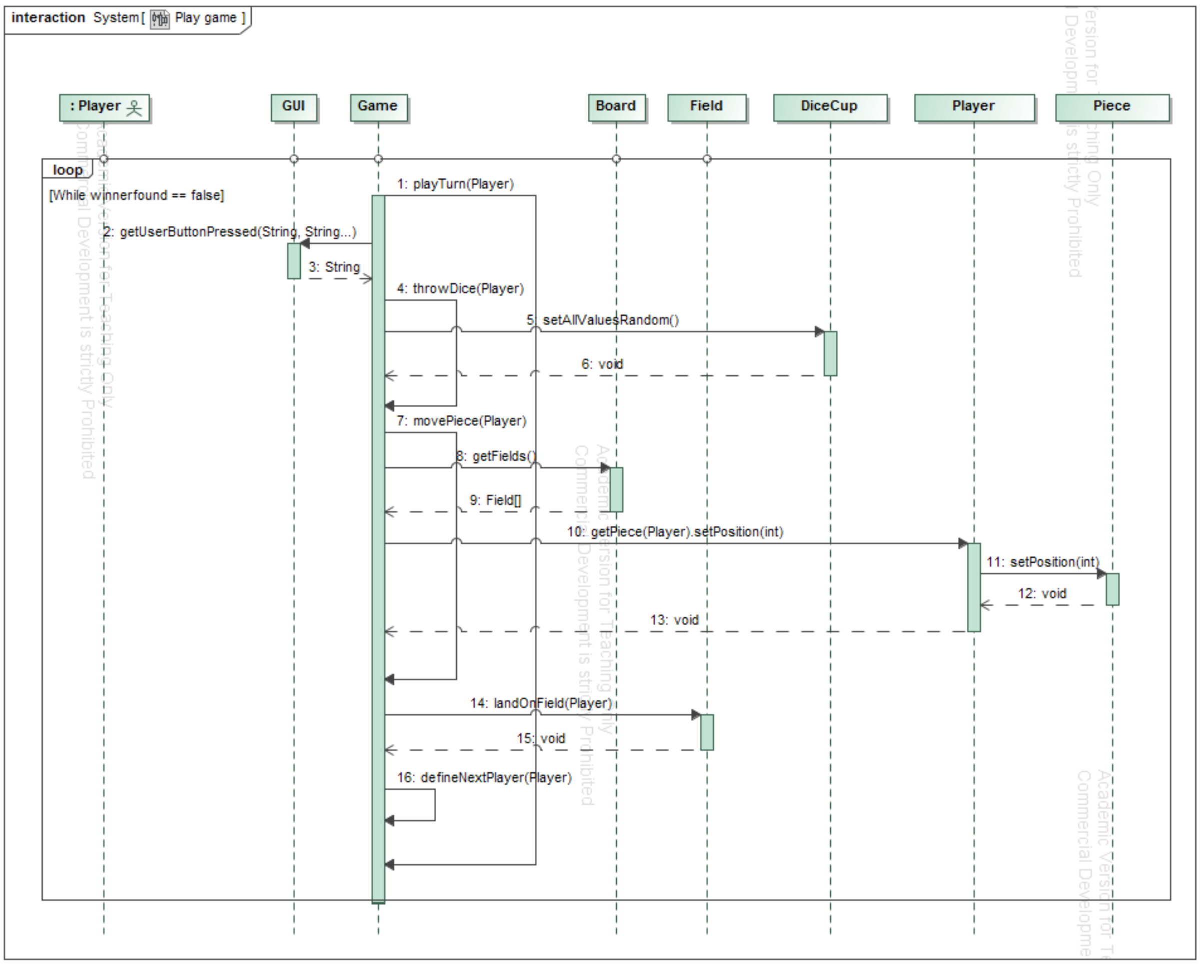


When the player clicks the program, we create several objects before the user is even able to see the graphical user interface. We create objects of the CreateGame, Game and Board class so we have them available for our next user interaction. Since the GUI needs to have the fields from the board in order to be created, we have to initialize both the Board and the Game, which controls this Board.

Once the GUI is created, we can ask the player how many players they want to play the game with. This is illustrated with the following sequence diagram:


interaction System[Reset game]
: Player
GUI
CreateGame
Game
loop
[Loop until player responds "no" to starting a new game]
1: getUserButtonPressed(String, String...)
2: Start a new game?
3: yes/no
4: String
5: getUserSelection(String, String...)
6: How many players?
7: amount
8: String
9: Interget.parseInt()
10: resetGame(int, int)
11: void
12: playGame()
13: void
Academic Version for Teaching Only
Commercial Development is strictly Prohibited

We first ask the user if they want to start a new game, then we ask how many players they want to play the game with. The result of this is that we know the amount of players we should reset the game to accommodate. After we have done this, we start the game. Once the game is over and we return void, we return to the start of this sequence diagram and keep looping until the player no longer wants to play our game. This is another reason we do not pass the amount of players as a parameter of the Game constructor, as we do not need to create a new Game and Board every time we want to restart the game. We simply keep this in a method, which makes the loop easier to implement.



We only show the most significant parts of playGame() in the above diagram. In the most basic sense, the game consists of a loop of turns where we keep going through the array of remaining players in the game.

We start a turn by asking the player to roll the dice. This prompts the throwDice() method, which sets the values of the dice to be random and sets the dice on the GUI.

After this we move the player's piece to the relevant position, and updates the necessary information. We also get the field that the player is currently landed on through the Board class which allows us to call the landOnField() method for that field.

We end the turn by defining which player will have the next turn.

This loop keeps on going until there is only 1 player left in the players array. Players get removed from the players array when they are unable to pay rent or tax for the field they have landed on, which means that their balance is reduced to 0 or less. When we define the next player, we check if the player who has just gone through their turn needs to be removed and adjust the size of our player array accordingly.

4.1.3 GRASP

We have based many of our design decisions around the patterns of GRASP.

Low coupling. Most of our classes have only one or two associations and do their communication with the rest of the program through their associated class. Doing this causes low coupling and means that we only end up having a few clear connections between different classes instead of having millions of methods that call to every single class in the program.

High cohesion. The classes we have focus around doing approximately one task. For example, the Player class stores information on one player, meaning their balance, ID, position on the board and so on. The CreateGame starts the game with a set amount of players. We have tried to keep these tasks as simple as possible, and every method and attribute in the classes are then centered around that task.

Controller. The Game class is the first class in our system that handles inputs from the GUI. It manages most of the work distribution of our other classes, and as such is the central class of the program. In theory, it might have been possible to separate the Game class into a Game controller and a Game logic class, one to handle the inputs and another to handle the actual logic. But since this project is small, we found it appropriate to have a single class in charge of managing the game flow and displaying the appropriate button messages that are necessary for the game to play out properly. We risk ending up with a bloated controller, but for the sake of simplicity, we have kept it this way.

Creator. In our system, we mostly use the creator patterns where:

- Creator contains the object class
- Creator closely uses the object class
- Creator has the necessary information for initialization of the object class

This is relevant when the Game *has* several Player(s) and *uses* Dice or when the Board is *composed* of Fields. We use the last pattern when we need to find out how many players the game should be played with. We don’t determine it immediately on initialization, but the class that creates the Game object still ends up being the one to set the amount of players.

Information expert. Board is a good example of the application of information expert. The Board contains information on all of the fields in our game, and as such is used to access individual fields and fetch their information. Game also acts as an information expert for all the players in our game in a way.

Pure fabrication. Pure fabrication is a class that does not present one of the problem domain concepts. An example of use of this pattern in our system would be the Messages class. This class is only there to contain the messages we display on the GUI. It is not something that can be represented physically, or something you would even think of if it was not because we are dealing with software that we want to translate easily.

Polymorphism. We use this in our inheritance between Field and the specific types of field. This means we have landOnField() and getRent() methods that take on different forms depending on what specific type of field we land on.

Indirection. We have used this pattern when making the Board class. It supports the low coupling of the system and the reusability of certain parts of the system by making sure that the Game class has to get through an intermediate object in order to get the information on each of the fields.

We have not used the protected variations pattern for this project.

4.2 Code documentation

4.2.1 StartProgram

StartProgram() is our main class with the only purpose to run our program as its only responsibility. We have tried to have this class manage as little code as possible and therefore the only thing it does is to instantiate the CreateGame() class which handles the games “startup-code”.

4.2.2 CreateGame

CreateGame() is the class which handles creating a new game. When instantiated this class will (through its constructor) instantiate the Game() class and creates a fresh game with a set amount of players (users choice) through it.

4.2.3 Game

Game() is the class we use to run our game. Since the Game() class’s responsibility is to run the game all the data required to run the game is stored in other classes. The Game class consist of the following methods:

- resetGame()
    - Instantiates a new fresh game - assigns amount of players playing and giving them a set balance, besides that, this method also manipulates the establishes the game interface
- playGame()
    - Checks if a player has won the game yet, if no player has won the game this method will call upon the playTurn() method to play a turn for a player. This method will run in a loop until a winner has been found
- playTurn(Player currentPlayer)
    - Uses the methods “defineNextPlayer(), throwDice() and movePiece()” to make a roll for the current player, move the current player according to the said roll and sets itself up to do the same for the next player when this method gets called again.
- removePlayer(Player player)
    - Removes a player from the player array
- defineNextPlayer(Player currentPlayer)
    - Defines what player has already played a turn and can therefore find the next player in row, to play their turn, which is what it returns
- throwDice(Player currentPlayer)

    - Simulates the throwing of dices and returns the result of said throw when called upon
- movePiece(Player currentPlayer)
    - Removes a player piece from its tile and move it to another tile according to what has been rolled by the piece's master

4.2.4 Board

We use Board to store the information on the fields we use in our game.

We use a Field type array to do this.

```
Field[] fields = new Field[21];
```

We can then use the constructors of the subclasses to instantiate an object in this array.

```
fields[0] = new Refuge(5000); //Walled city
fields[1] = new Tax(2000, -1); //Goldmine
```

When running our landOnField() method, it will then run the implementation that we have used for the appropriate subclass.

We also have another method in the Board, which we have called showFieldsOnGUI(). This method works with the GUI we have been given for the project and sets it up with the appropriate fields.

```
for(int i = 0; i < fields.length; i++){
        graphicfields[i] = new Street.Builder()
                .setBgColor(determineFieldColor(i))
                .setTitle(Messages.getFNames()[i])
                .setDescription(Messages.getFNames()[i])
                .setSubText(determineSubText(i))
                .build();
}
GUI.create(graphicfields);
GUI.displayChanceCard();
```

This syntax uses several helping methods that we have created, but they all take information from the fields inside board and use that to create every 'Street' in the array based on which subclass of Field they are an instance of.

Board also has a get method which returns the array of fields inside of it, which we use in our Game class.

4.2.5 Field

Field is an abstract class that we use to generalize all of the fields we have on our board. Having this general class is what even allows us to have an array of fields in the Board class that we can access.

All subclasses of Field must have an implementation of the landOnField(Player) method, which returns void. The reason is that this method is abstract and is not implemented on the super class level.

Field is only used as a generalization of our different field types to store it in Board and to ensure that all subclasses of it must have landOnField() as one of their methods.

4.2.5.1 Tax

Tax is a subclass of Field. When the player lands on this field, landOnField(Player) is called.

In this class, if the tax rate was instantiated as -1, we only give the player an option to pay a flat amount in tax. We then set the player's score to be equal to the score at the beginning of the turn minus the flat rate from the tax field.

If the tax rate was set to a positive value, we give the player the option to either pay a flat amount in tax or pay a percentage of all assets they own.

We have a list of all the fields that the player owns inside of the player and use the method getAllAssets() to calculate the total value of everything that the player owns. If the player choose to pay a percentage, we subtract a percentage of this amount from their balance equal to the tax rate.

4.2.5.2 Refuge

Refuge is like tax, but instead gives the player a flat amount of money when they land on it. When landOnField(Player) is called, the system adds a bonus to the player's score.

The bonus is determined on instantiation with the constructor for this class.

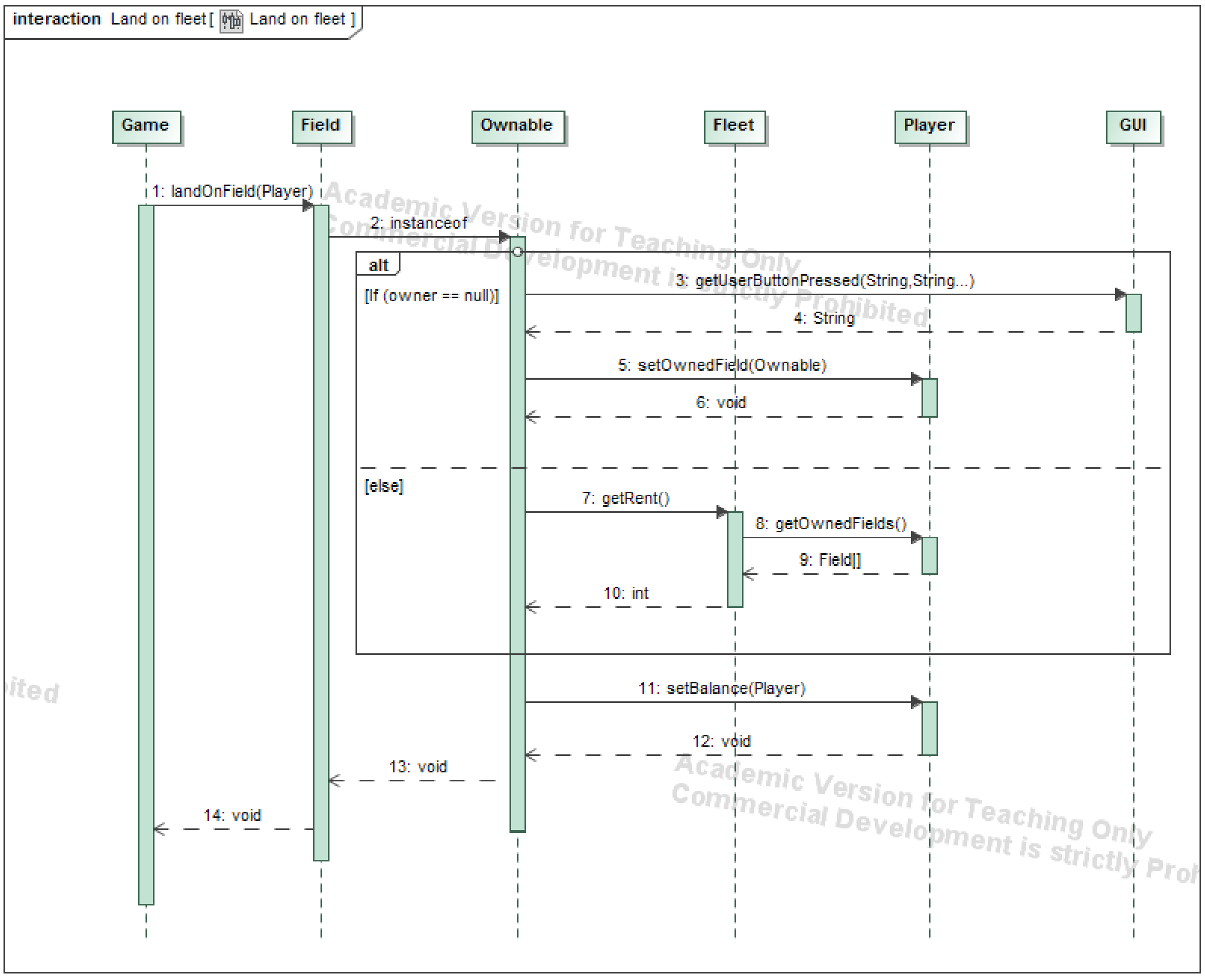
4.2.5.3 Ownable

Ownable is a subclass of Field that the player gains the option to buy when they land on. When other players land on a field owned by a player, the player who lands on the field then has to pay rent to the owner.

It is an abstract class, which means that it has a method that is, in a sense, incomplete. We have implemented this method (getRent()) in each of its subclasses.

Its relation with Field and its own subclasses has already been explained in our documentation diagrams and in our GRASP documentation and will also be further elaborated in the documentation for Fleet.

4.2.5.3.1 Fleet



When we land on a fleet, our system first receives a command from the Game object that landOnField() should be called. This method exists only in Field as an abstract method, which means that the system will check which subclass we have an instance of. In this case, it would be Ownable, first and foremost, as this is where our landOnField() method is implemented.

Ownable checks whether the field is already owned.

If it is not owned, the player gets an option to buy the field in the GUI. They can also decline this offer. If they do not have enough money to buy the field with their current balance, they are not given the option at all.

If the field is owned, Ownable calls getRent(), which is an abstract method. The system again checks which subclass the field is an instance of. This time it is the fleet class. This getRent() implementation looks at the owned fields of a player, and counts how many of these have the instance of Fleet. It then determines which of the values from RENT_1 to RENT_4 it needs to return and does so. This gives the Ownable class an int, which it can then use to set the balance of the field owner and the balance of the player whose turn it is, to the appropriate amounts.

The other ownable fields have a similar sequence of commands. The difference lies in their implementation of getRent().

4.2.5.3.2 Territory

Territory simply uses a flat rent, which is defined on instantiation, and returns this value. Otherwise it is exactly similar to Fleet in terms of function.

4.2.5.3.3 LaborCamp

LaborCamp uses a base rent times the dice roll that the player get in the beginning of their turn. In all other aspects it is similar to both Territory and Fleet.

4.2.6 DiceCup

DiceCup is the class that manages our dice rolls and make sure said dice rolls are random. When called the DiceCup class will require a set amount of dices it should and the amount of dice sides the dice should have, in our program this amount is a constant of 2 dice with 6 sides each. The DiceCup class have the following methods:

- getValues()
  - returns the individual results of each die roll in a array
- getSum()
  - returns the sum of all the rolls that have been made
- setAllValuesRandom()
  - Simulates a roll using the given amount of dice and the sides of each die

Bear in mind though that these methods does not give any real results except for setValuesRandom() unless there have already been made a roll. To counter the problem of not having made a roll we (in the constructor) make a first roll before any of the methods gets called.

4.2.7 Messages

Messages class contains all the messages throughout the game. The messages is currently in danish but can easily be translated to english whenever it is needed to.
We have placed the messages in order, so that it is easy for us, and for others, to figure out what specific situation the messages are used.

4.2.8 Player

Player class stores references to information related to the specific player. Here is the player's 'name', 'id', 'balance', 'Piece', 'ownedFields' and 'diceSum'. The references are private, but a simple public 'get' method can be used to reach out for these information.

```
public Player(String name,int id, int balance, Piece piece){..}
```

The constructor takes a name, id, balance and a Piece (color and position on board). Name and ID will never change throughout the game and therefore there are no methods to set those, as it'll be set in the constructor.

Three set methods has been added to the class.

```
        public void setBalance(int balance){..}
```

This method sets the balance of the player. It is important to note, that the balance is located both in the GUI and the player's own balance variable.

```
        public void setOwnedField(Ownable field){..}
```

The setOwnedField adds a field into the private ownable[] ownedFields array that can contain up to 17 fields.

```
        public void setDiceSum(int diceSum){..}
```

Just like any of the other methods we have in this class, the setDiceSum is also important. We need to store DiceSum of the players' throw in case the player lands on a Labor Camp, where it requires to multiply the sum with 100.

The above three set methods are of the type void, as it shall not return anything, but simply just change the private stored references to new information.

4.2.8.1 Piece

Piece class has the only responsibility of storing references to all the players pieces. It will store a reference to the specific piece's position and it's color.
By calling a player's Piece, you will be able to change it's position throughout the game using its method setPosition(integer).

4.3 Version control

We have used Github for our version control during this project. Our strategy for version control involved having a secondary branch in addition to the main ‘master’ branch. We called this our ‘development’ branch.

In the development branch, we made new commits every time we had completed both large and miniscule tasks in our development of the program. Some of us updated more than others, but for the most part, we had at least one commit for every ten minutes of coding.

It was our goal, that the project leader should decide when we had hit a version of the program that had a satisfactory degree of functionality, and then merge the development branch and the master branch.

In other words, we wanted to have one branch for all of our updates and a second branch that was only updated once every couple of days when we had achieved certain levels of functionality.

This, anyhow, was not achieved as the development and master branch didn’t merge before our project was finished.

The commits worked well while different members of the group were working in different parts of the program, but became a little bit problematic when two or three people were working together on a single class.

We experienced a few file conflicts, but not so many that the project was brought to a halt, and this strategy of version control probably ended up being the fastest in the end over other alternatives like having different development branches for each of our group members, which would put a larger burden on the project leader, who would then have to juggle 6 different branches in total, which would probably still create merge conflicts at some point.

5 Test

5.1 Unit tests

5.1.1 DiceCup

We have tested the following on the DiceCup class by creating a JUnit test case. It is called DiceCupTest, and can be found in the package DiceGameTest.

1. Whether the thrown value is within a predefined parameter (between 1 and the number of dice sides)
2. Whether the dice is random enough

We have tested this by “throwing a dice” with 6 sides 30000 times and tested if the dice went over or under our predefined values as well as tested if the result of the dice throws are spread out evenly among the 6 possibilities.
The test was successful.

5.1.2 Field

We have tested the method landOnField() in the classes Fleet, Territory, Refuge, Labor Camp and Tax by creating a JUnit test case. It is called fieldTest, and can be found in the package DiceGameTest.
In Fleet we have made a full test of all possibilities, but four of them are the same for all Ownable classes. Therefore they are only present in the method fleetTest. These situations are:

1. The field has no owner and the player has a high enough balance to buy the field.
2. The field has no owner and the player does not have a high enough balance to buy the field.
3. The owner of the field is the player itself.
4. The owner of the field has a balance of 0.

The last situation is when the owner of the field has a balance higher than 0, then it will depend on whether the Ownable field is a Labor Camp or not.
The test was successful.

5.2 Use case tests

TC01: Land on fleet

Test case ID	TC01
Summary	Testing if the use case UC1, land on fleet, works as intended.
Requirements	None
Preconditions	1. It is the player’s turn 2. The player rolls the dice and lands on a fleet field
Test procedure	1. We use a program to generate a game with our preconditions met. 2. We choose “Yes” to buy the field. 3. We check if the next player is the second player in the players array. 4. We check if the price of the field has been subtracted from the player’s startbalance. 5. We check if the next player’s balance is still the startbalance. 6. We then continue the game and let the next player land on the same field as the first player. 7. We click “Ok” to pay rent to the owner. 8. We check if the rent has been subtracted from the player’s balance and added to the owner’s balance. 9. We check if the next player is the first player in the players array.
Test data	1. Number of players = 2 2. Startbalance of players = 5000 3. Values of dice = {2,2} 4. Price of field = 4000 5. Rent of field = 500
Expected result	1. Next player = Player 2 2. Player 1 balance = 1000 3. Player 2 balance = 5000 4. Next player = Player 1 5. Player 1 balance = 1500 6. Player 2 balance = 4500
Actual result	1. Next player = Player 2 2. Player 1 balance = 1000 3. Player 2 balance = 5000 4. Next player = Player 1 5. Player 1 balance = 1500 6. Player 2 balance = 4500
Status	Passed
Tested by	Freya Helstrup

Date	24/11/2016
Test environment	Eclipse Neon 1a Release (4.6.1)

TC02: Land on territory

Test case ID	TC02
Summary	Testing if the use case UC2, land on territory, works as intended.
Requirements	None
Preconditions	1. It is the player’s turn 2. The player rolls the dice and lands on a territory field
Test procedure	1. We use a program to generate a game with our preconditions met. 2. We choose “Yes” to buy the field. 3. We check if the next player is the next player in the players array. 4. We check if the price of the field has been subtracted from the player’s startbalance. 5. We check if the next player’s balance is still the startbalance. 6. We then continue the game and let the next player land on the same field as the first player. 7. We click “Ok” to pay rent to the owner. 8. We check if the rent has been subtracted from the player’s balance and added to the owner’s balance. 9. We check if the next player is the next player in the players array.
Test data	1. Number of players = 2 2. Startbalance of players = 5000 3. Values of dice = {2,1} 4. Price of field = 1000 5. Rent of field = 100
Expected result	1. Next player = Player 2 2. Player 1 balance = 4000 3. Player 2 balance = 5000 4. Next player = Player 1 5. Player 1 balance = 4100 6. Player 2 balance = 4900
Actual result	1. Next player = Player 2 2. Player 1 balance = 4000 3. Player 2 balance = 5000 4. Next player = Player 1 5. Player 1 balance = 4100 6. Player 2 balance = 4900

Status	Approved.
Tested by	Freya Helstrup
Date	24/11/2016
Test environment	Eclipse Neon 1a Release (4.6.1)

TC03: Land on refuge

Test case ID	TC03
Summary	Testing if the use case UC3, land on refuge, works as intended.
Requirements	None
Preconditions	1. It is the player’s turn 2. The player rolls the dice and lands on a refuge field
Test procedure	1. We use a program to generate a game with our preconditions met. 2. We check if the next player is the next player in the players array. 3. We check if the bonus of the field has been added to the player’s startbalance. 4. We check if the next player’s balance is still the startbalance.
Test data	1. Number of players = 2 2. Startbalance of players = 5000 3. Values of dice = {6,6} 4. Bonus = 500
Expected result	1. Next player = Player 2 2. Player 1 balance = 5500 3. Player 2 balance = 5000
Actual result	1. Next player = Player 2 2. Player 1 balance = 5500 3. Player 2 balance = 5000
Status	Approved.
Tested by	Freya Helstrup
Date	24/11/2016
Test environment	Eclipse Neon 1a Release (4.6.1)

TC04: Land on labor camp

Test case ID	TC04
Summary	Testing if the use case UC4, land on labor camp, works as intended.
Requirements	None
Preconditions	1. It is the player’s turn 2. The player rolls the dice and lands on a labor camp field
Test procedure	1. We use a program to generate a game with our preconditions met. 2. We choose “Yes” to buy the field. 3. We check if the next player is the next player in the players array. 4. We check if the price of the field has been subtracted from the player’s startbalance. 5. We check if the next player’s balance is still the startbalance. 6. We then continue the game and let the next player land on the same field and pay rent to the first player. 7. We check if the rent has been subtracted from the player’s balance and added to the owner’s balance. 8. We check if the next player is the next player in the players array.
Test data	1. Number of players = 2 2. Startbalance of players = 5000 3. Values of dice = {3,4} 4. Price of field = 2500 5. Rent of field = 100*dice roll
Expected result	1. Next player = Player 2 2. Player 1 balance = 2500 3. Player 2 balance = 5000 4. Next player = Player 1 5. Player 1 balance = 3200 6. Player 2 balance = 4300
Actual result	1. Next player = Player 2 2. Player 1 balance = 2500 3. Player 2 balance = 5000 4. Next player = Player 1 5. Player 1 balance = 3200 6. Player 2 balance = 4300
Status	Approved.
Tested by	Freya Helstrup
Date	24/11/2016
Test environment	Eclipse Neon 1a Release (4.6.1)

TC05: Land on tax

Test case ID	TC05
Summary	Testing if the use case UC5, land on tax, works as intended.
Requirements	None
Preconditions	1. It is the player's turn 2. The player rolls the dice and lands on a tax field
Test procedure	1. We use a program to generate a game with our preconditions met. 2. We choose "Ok" to pay the tax. 3. We check if the next player is the next player in the players array. 4. We check if the tax has been subtracted from the player's startbalance. 5. We check if the next player's balance is still the startbalance. 6. We then continue the game and let the same player land on another tax field. 7. We choose "Betal 10% af alle ejendele" to pay the taxRate. 8. We check if the tax has been subtracted from the player's balance. 9. We check if the next player's balance is still the startbalance. 10. We check if the next player is the next player in the players array.
Test data	1. Number of players = 2 2. Startbalance of players = 5000 3. Values of dice in first turn = {1,1} 4. Tax amount in first turn = 2000 5. Values of dice in second turn = {5,6} 6. Tax amount in second turn = 300
Expected result	1. Next player = Player 2 2. Player 1 balance = 3000 3. Player 2 balance = 5000 4. Next player = Player 2 5. Player 1 balance = 2700 6. Player 2 balance = 5000
Actual result	1. Next player = Player 2 2. Player 1 balance = 3000 3. Player 2 balance = 5000 4. Next player = Player 2 5. Player 1 balance = 2700 6. Player 2 balance = 5000
Status	Approved.
Tested by	Freya Helstrup
Date	24/11/2016

Test environment	Eclipse Neon 1a Release (4.6.1)

TC06: Win Game

Test case ID	TC06
Summary	Testing if the use case UC6, win game, works as intended.
Requirements	None
Preconditions	1. There are two players with a balance of less than 2000 remaining in the game. 2. The first player lands on field that sets his balance to 0.
Test procedure	1. We use a program to generate a game with our preconditions met. 2. We choose “Ok” to pay the tax. 3. We check if the next player is the first player in the players array. 4. We check if the balance of the first player is 0. 5. We check if the length of the players array is 1.
Test data	1. Number of players = 2 2. Startbalance of players = 1000 3. Values of dice = {1,1}
Expected result	1. Next player = Player 2 2. Player 1 balance = 0 3. Length of players array = 1
Actual result	1. Next player = Player 2 2. Player 1 balance = 0 3. Length of players array = 1
Status	Approved.
Tested by	Freya Helstrup
Date	25/11/2016
Test environment	Eclipse Neon 1a Release (4.6.1)

TC07: Lose Game

Test case ID	TC07
Summary	Testing if the use case UC7, lose game, works as intended.
Requirements	None
Preconditions	1. It is the player’s turn 2. The player rolls the dice and lands on a field, so his balance is set to 0.
Test procedure	1. We use a program to generate a game with our preconditions met. 2. We choose “Ok” to pay the tax. 3. We check if the next player is the first player in the players array. 4. We check if the balance of the player is 0. 5. We check if the length of the players array has been reduced with 1.
Test data	1. Number of players = 3 2. Startbalance of players = 2000 3. Values of dice = {1,1}
Expected result	1. Next player = Player 2 2. Player 1 balance = 0 3. Length of players array = 2
Actual result	1. Next player = Player 2 2. Player 1 balance = 0 3. Length of players array = 2
Status	Approved.
Tested by	Freya Helstrup
Date	25/11/2016
Test environment	Eclipse Neon 1a Release (4.6.1)

TC08: Purchase field

Test case ID	TC08
Summary	Testing if the use case UC8, purchase field, works as intended.
Requirements	None
Preconditions	1. It is the player's turn 2. The player lands on a field, which is not owned.
Test procedure	1. We use a program to generate a game with our preconditions met. 2. We choose "Ja" to buy the field. 3. We check if the next player is the second player in the players array. 4. We check if the balance of the player is reduced with the price of the field. 5. We check if the player is the owner of the field.
Test data	1. Number of players = 3 2. Startbalance of players = 5000 3. Values of dice = {4,1} 4. Price of field = 1500
Expected result	1. Next player = Player 2 2. Player 1 balance = 3500 3. Owner of field 5 = Player 1
Actual result	1. Next player = Player 2 2. Player 1 balance = 3500 3. Owner of field 5 = Player 1
Status	Approved.
Tested by	Freya Helstrup
Date	25/11/2016
Test environment	Eclipse Neon 1a Release (4.6.1)

5.3 Integration test

Test case ID	TC101
Summary	Testing if it is possible to run a game in its entirety and start a new game afterwards.
Requirements	None
Preconditions	None
Test procedure	Run the game.
Test data	None
Expected result	The program goes through an entire game without crashing and resets the GUI with new players.
Actual result	● The program successfully completes the entire game ● It successfully manages to create new players with the correct score after saying yes to a new game ● A bug appears if the game is made with fewer players than the first game. The player remains on screen, but does not participate in the game.*
Status	Approved.
Tested by	Tobias
Date	25/11/2016
Test environment	Eclipse Neon 1a Release (4.6.1)

*: Visual documentation for this test can be found in appendix 8.3.4

5.4 Test conclusion

5.4.1 Unit tests

The unit tests were successful, meaning that our dice is random and our landOnField method works as intended.

5.4.2 Use case tests

The test cases for our use cases were successful.

5.4.3 Integration test

During our time of testing the game, we noticed a minor bug, that does not have to do with the gameplay at all, but has an impact on the interface.
After playing a game of 3 or more players, you will get the choice of restarting the game. If you pick a lesser amount of players than the previous game, the players from previous game will still exist on the board. This can be seen in Appendix 8.3.4.
Therefore, they won’t be a part of the game, but will be shown on the players list.
Besides that, the game runs as intended and fulfills our requirement specification.

6 Conclusion

Based on IOOuterActive’s vision, we have made and fulfilled a requirement specification for a board game. We have succeeded in designing and creating such a board game, in which players move around a board and land on fields with different effects. The game's structure has been documented, and the game has been tested sufficiently, so that we can guarantee a finished and working product.

7 Literature

John Lewis, William Loftus: *Java software solutions, Seventh Edition*
Published 2012
Pearson Education, Inc., publishing as Addison-Wesley, Boston, Massachusetts
ISBN 13: 978-0-13-214918-1

Craig Larman: *Applying UML and Patterns: An introductions to object-oriented analysis and design and iterative development*
Published: 30. October 2004
Pearson Education, Inc. publishing as John Wait
ISBN: 0-13-148906-2

8 Appendix

8.1 Appendix 1: Vision document / Customer description

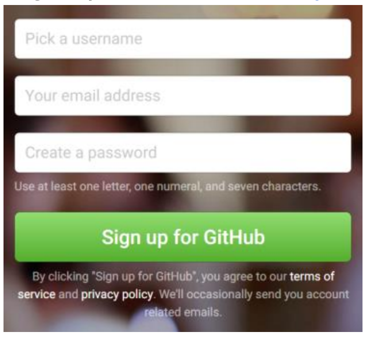
Nu har vi terninger og spillere på plads, men felterne mangler stadig en del arbejde. I dette tredje spil ønsker vi derfor, at forrige del bliver udbygget med forskellige typer af felter, samt en decideret spilleplade.
Spillerne skal altså kunne lande på et felt og så fortsætte derfra på næste slag. Man går i ring på brættet.
Der skal nu være 2-6 spillere.
Man starter med 30.000.
Spillet slutter når alle, på nær én spiller, er bankerot.

I bilag kan I se en oversigt over de felter vi ønsker, samt en beskrivelse af de forskellige typer.

8.2 Appendix 2: Importing Git repositories in Eclipse Java Neon

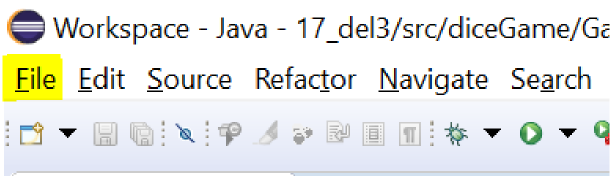
We have used GitHub as our main code sharing tool - making it possible for us to merge our code and download fixes or updated code from each other. This chapter will guide you to find our git repository and import it to Eclipse Java Neon (from now on just referred to as Eclipse) - therefore if you want this guide to work we would recommend you have Eclipse installed.

0. Install Eclipse (if not installed)
1. Sign up as a user at: https://github.com/ (If not already signed up)



Pick a username
Your email address
Create a password
Use at least one letter, one numeral, and seven characters.
Sign up for GitHub
By clicking "Sign up for GitHub", you agree to our terms of service and privacy policy. We'll occasionally send you account related emails.

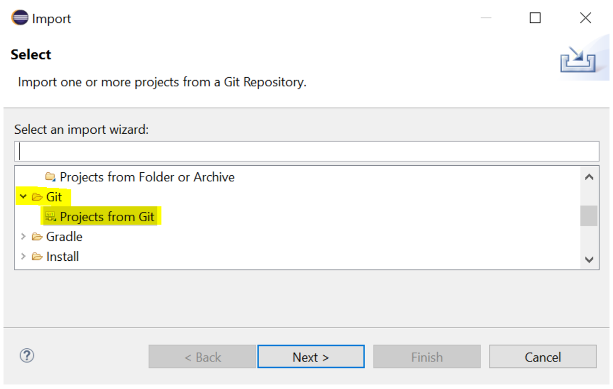
2. Open up Eclipse
3. Press file (Shortcut: Alt + f) up in the left corner



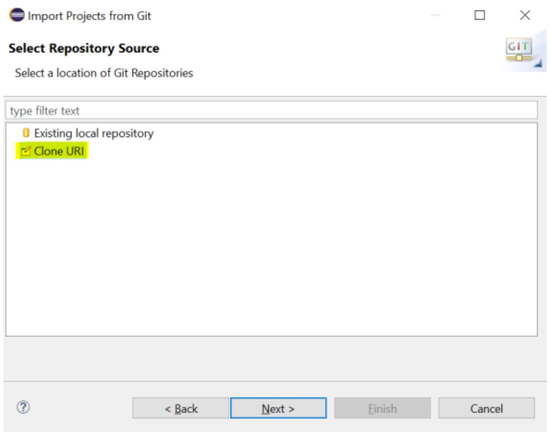
Workspace - Java - 17_del3/src/diceGame/Ga
File Edit Source Refactor Navigate Search

4. Press Import in the dropdown menu (Shortcut: i when dropdown open)

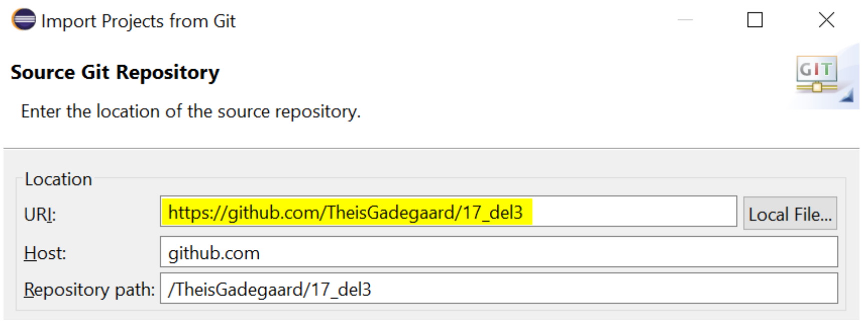
5. In the new window open up the Git folder and highlight “Projects from Git”



6. Press Next
7. Highlight “Clone URI”

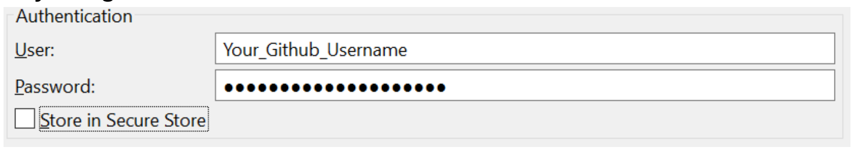


8. Press Next
9. Copy (Ctrl + c) and paste (Ctrl + v) the following link into the URI input: https://github.com/TheisGadegaard/17_del3



URI: https://github.com/TheisGadegaard/17_del3
Host: github.com
Repository path: /TheisGadegaard/17_del3

10. The rest should fill out automatically except for authentication where you should write in your github user information

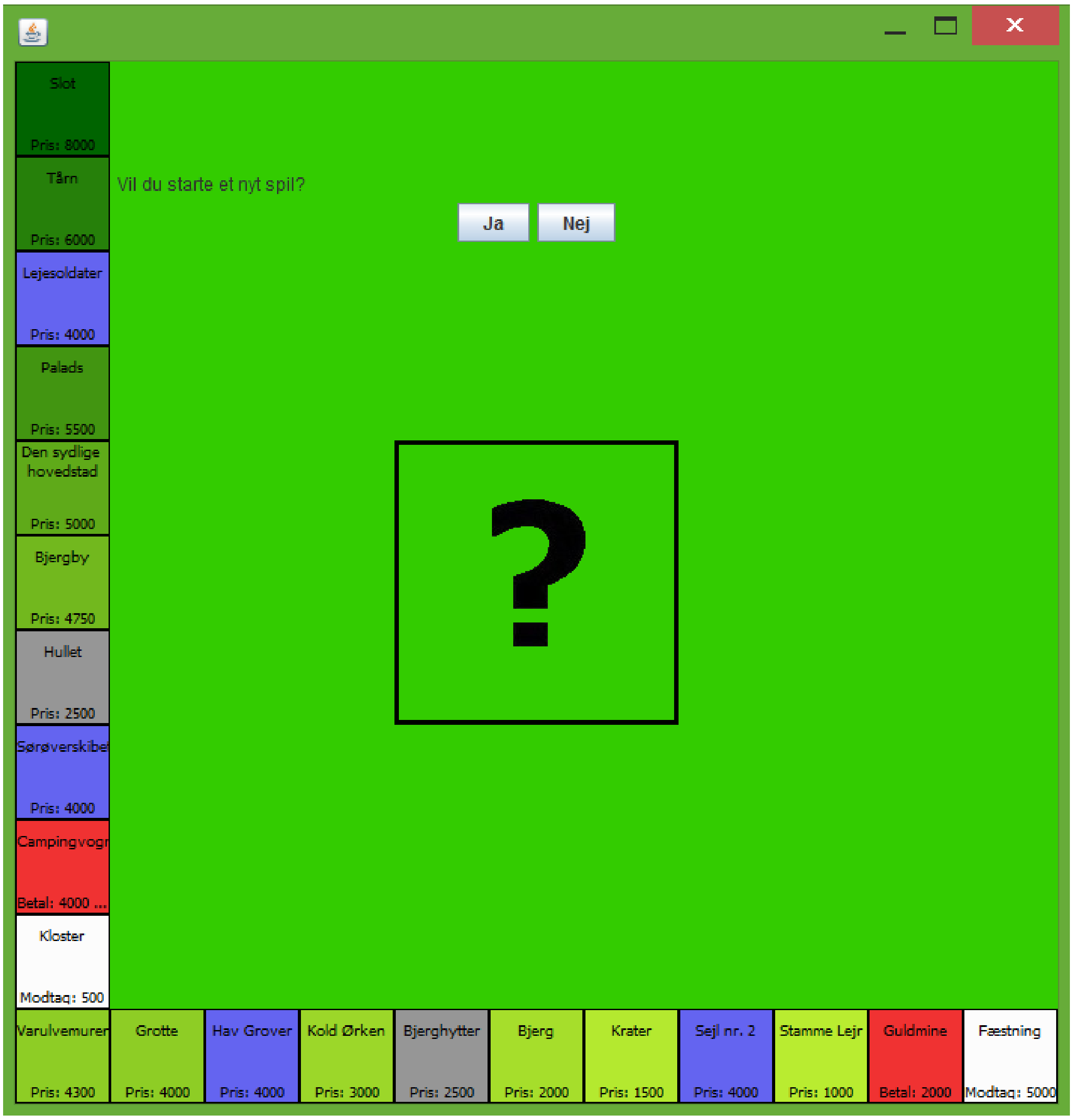


User: Your_Github_Username
Password: ●●●●●●●●●●●●●●●●●●●●
☐ Store in Secure Store

10.1. To skip this step in the future check the “Store in Secure Store” box, this will save your username and password to use in the future

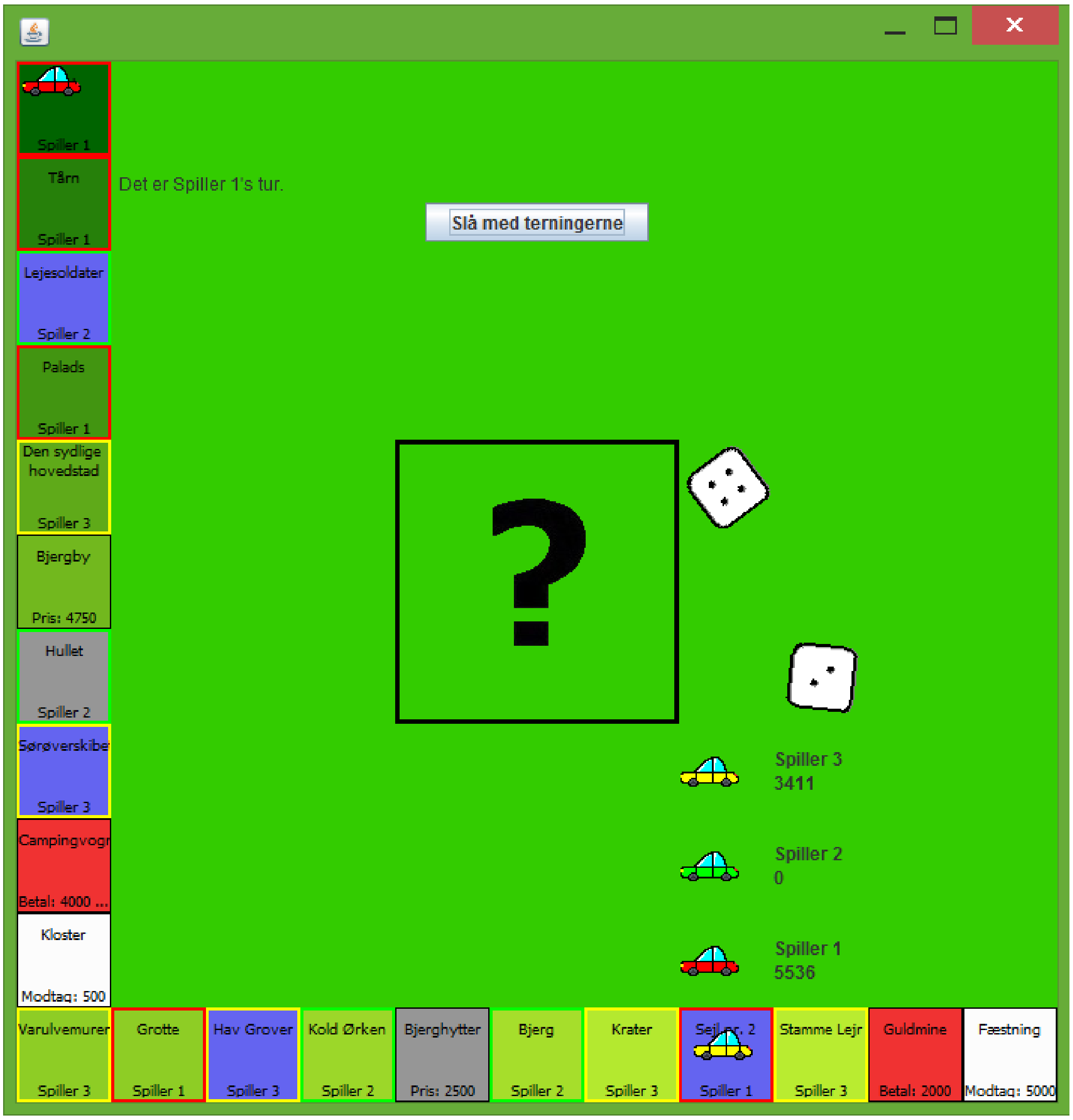
11. Press next until the window disappears
12. You have now imported our GitHub repository to Eclipse

8.3 Appendix 3: Visual documentation



Vil du starte et nyt spil?

8.3.1 Game start

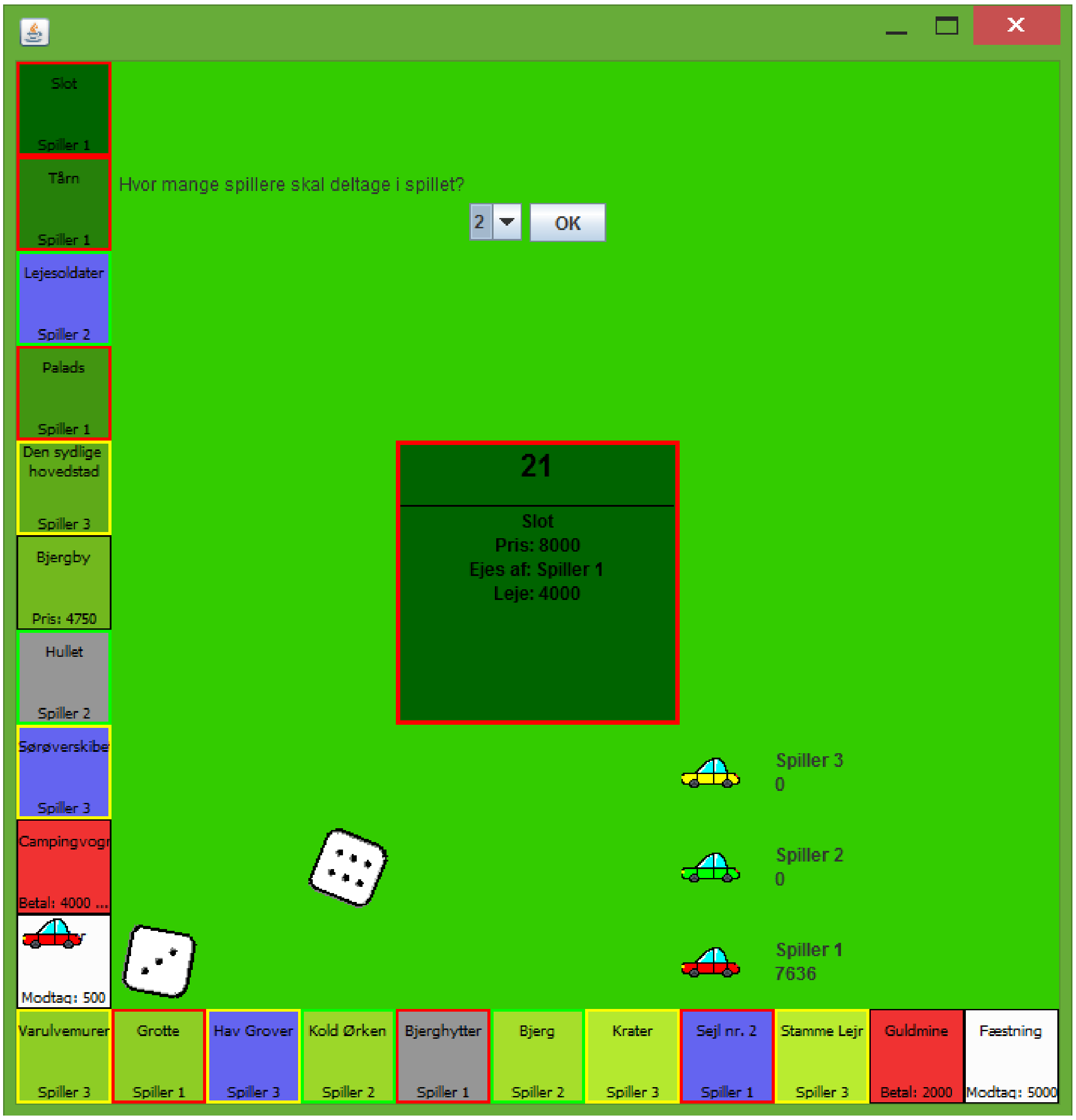

Det er Spiller 1's tur.

Spiller 3
3411

Spiller 2
0

Spiller 1
5536

8.3.2 Game in progress

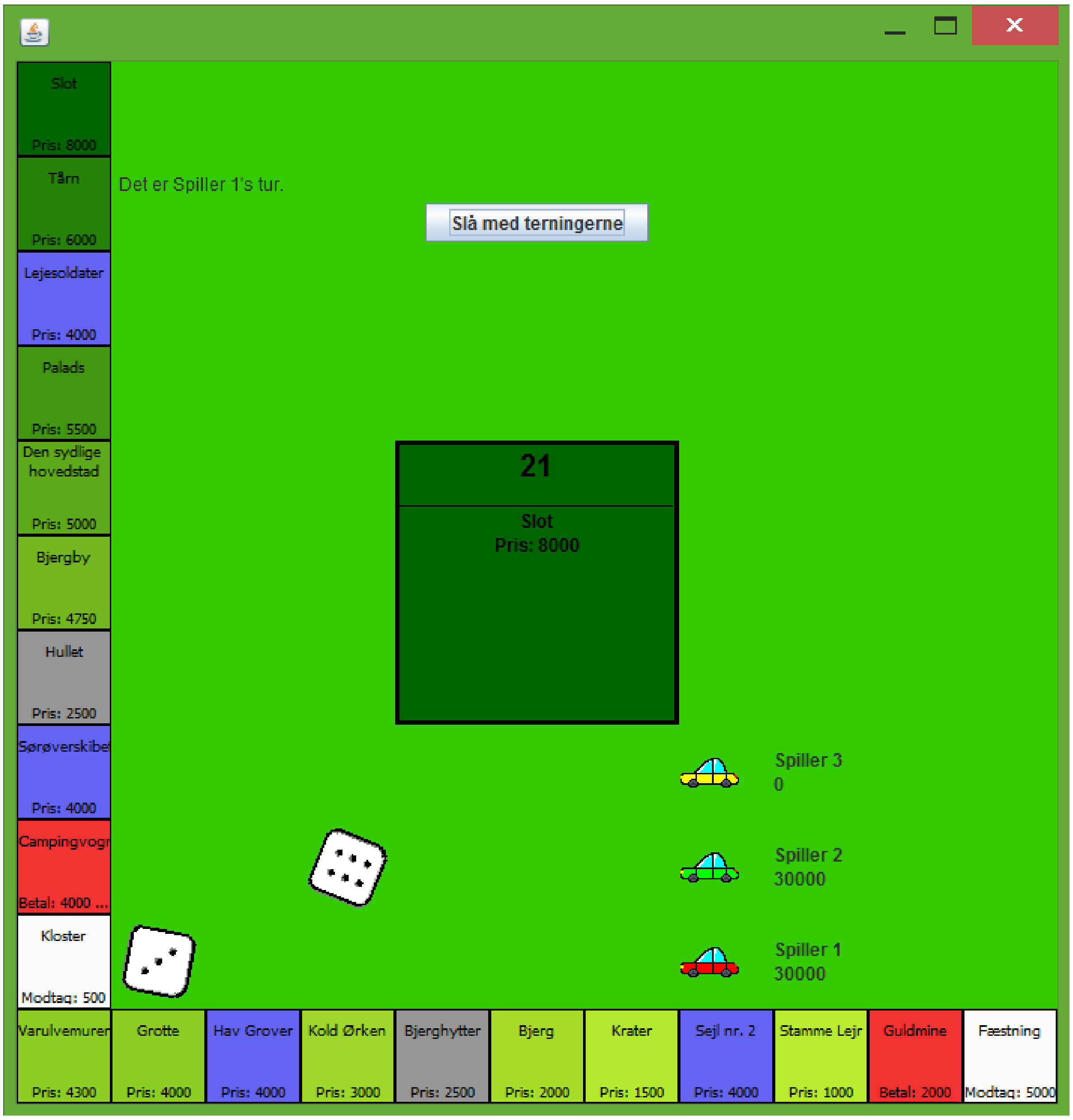


Hvor mange spillere skal deltage i spillet?

21

Slot
Pris: 8000
Ejes af: Spiller 1
Leje: 4000

8.3.3 Start new game after first



8.3.4 Visual bug

8.4 Appendix 4: Source code

The source code for this project can be found in the .zip file that was turned in.