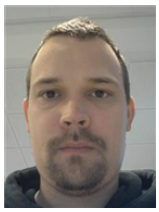
CDIO 3

Mathias Tværmose Gleerup, s153120,
Mikkel Geleff Rosenstrøm s124363,
Sofie Freja Christensen s153932
Morten V. Christensen s147300
Simon Lundorf s154008
Jonas Larsen, s136335,

Group nr. 15 Classes: 02312, 02313, 02315

Deadline : November 25th 2016

November 25, 2016



Contents

Table 1: Hour registration

Hours:	Jonas	Mathias	Mikkel	Morten	Simon	Sofie	Sum
Conceive	4,75	4,75	1	1	0,5	2,5	14,5
Design	8,5	7,25	0	5,25	5	10,25	36,25
Impl.	15	19,5	3	9	11	7	64,5
Test	0	3	1	0	3,5	0	7,5
Dok.	3	6	6	9,75	7	14	45,75
Andet.	0	0	0	0	0	0	0
Sum	31,25	40,5	11	25	27	33,75	168,5

1 Abstract

A program has been compiled in Java. The program is designed as a less complex version of the monopoly game. As such, it is played on a board with 21 squares, each with a specific feature with a possible effect on the players balance. The objective of the game is to be the "last man standing". You lose and exit the game by going bankrupt (balance reaches zero or below).
In addition, there has been compiled a GUI, to give a visual display of the game.

2 Foreword

This report takes its foundation in the program (CDIO 3), drawn from the following courses Indledende programmering (02312), Udviklingsmetoder til IT-systemer (02313) og Versionsstyring og testmetoder (02315). All of which takes place at DTU campus Lyngby, fall of 2016. The project is made in collaboration between the members of group 15:

- Jonas Larsen - SWT
- Mathias Tværmose Gleerup - SWT
- Mikkel Geleff Rosenstrøm - ITØ
- Morten Velin Christensen - SWT
- Simon Lundorf - SWT
- Sofie Freja Christensen - SWT

3 Customers Vision

3.1 In danish

Kundens vision: 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.

3.2 In english

Customers vision: Now we have the dice and the players set, but the squares still need some work. In this third game, we wish the former game to be expanded, with different types of squares, and also an actual game board. The players shall be able to land on a square, and then continue from that position on their next turn. They move in a circle on the board. There shall be 2-6 players. Every player begins with 30000£. The game ends, when all but one player has gone bankrupt. In the appendix, you can see an overview over which squares we wish, and a description of the different types.

4 Procedure

1. Started by listing the Requirements for the system.
2. Based on the Requirements, we made a Noun and verb analysis to identify possible entities and operations of the game.
3. Based on the Requirements we formulated use cases, describing sequences of proceeding operations.
4. A Domain model was made to visualize which entities we thought we needed in the game. This was based on the entities we identified in our noun analysis.
5. Created Analysis Sequence Diagrams based on use cases, to visualize the operations we need for the use cases to be possible.
6. Made documentation for our analysis sequence diagrams, along with abstract.
7. Decided on applying the GUI, at hand in the game.
8. Decided to make a seperate class to handle on commands to the GUI
9. Documented usecase diagrams and descriptions.
10. Made Analysis Class Diagram.
11. Began programming the individual classes.
12. Tested the classes in JUnit test environments.
13. Decided to devide different segments of the game into different packages.
14. Added an ID field to the Square classes.
15. Finished programming the individual classes, tested them and documented them.
16. Made the class Game to handle the logic behind the game, and the class GUIControl.
17. Made design sequence diagram according to use cases, design class diagram to document the classes, and System Sequence Diagram to visualize main flow.
18. Finished tests and documentation.

5 Requirements

5.1 Functional

1. The game has to be played by two-six players.
2. The game shall use two die
3. The game has to be executable on the machines in DTU's databars.
4. The players are to take turns to roll the die.
5. The player has to have an account balance.
   (a) The balance starts at 30.000 £.
6. The game must contain 5 types of squares.
   (a) *Territory* - When a player lands on this square, owned by another player. The player has to pay the owner rent.
   (b) *Refuge* - The player is awarded a bonus.
   (c) *Tax* - The player has to pay either a fixed amount, or 10 % of their fortune. The player chooses which.
   (d) *Labor camp* - The player has to roll the die, 100x this amount is what the player has to pay the owner. If the owner of the labor camp, owns more than one, the amount is multiplied by the amount of labor camps.
   (e) *Fleet* - The player has to pay the owner. Price determined by number of fleet, as such:
      i. Fleet: 500.
      ii. Fleet: 1000.
      iii. Fleet: 2000.
      iv. Fleet: 4000.
7. The game has to be played on a board with 21 squares developed from the former game CDIO2.
   (a) The following squares are to be used:
      1. **Tribe Encampment:** Territory, rent 100, price 1000.
      2. **Crater:** Territory, rent 300, price 1500.
      3. **Crater:** Territory, rent 300, price 1500.
      4. **Mountain:** Territory, rent 500, price 2000.
      5. **Cold Desert:** Territory, rent 700, price 3000.
      6. **Black cave:** Territory, rent 1000, price 4000.
      7. **The Werewall:** Territory, rent 1300, price 4300.
      8. **Mountain village:** Territory, rent 1600, price 4750.
      9. **South Citadel:** Territory, rent 2000, price 5000.
      10. **Palace gates:** Territory, rent 2600, price 5500.
      11. **Tower:** Territory, rent 3200, price 6000.
      12. **Castle:** Territory, rent 4000, price 8000.
      13. **Walled city:** Refuge, receive 5000.
      14. [21] **Monastery:** Refuge, receive 500.
      15. **Hut in the mountain:** Labor camp, pay 100xdice, price 2500.
      16. **The pit:** Labor camp, pay 100xdice, price 2500.
      17. **Goldmine:** Tax, pay 2000 or 10% of total assets (Whichever is lower).
      18. **Second Sail:** Fleet, Pay 500-4000, price 4000.
      19. **Sea Grover:** Fleet, pay 500-4000, price 4000.
      20. **The Buccaneers:** Fleet, pay 500-4000, price 4000.
      21 **Privateer armada:** Fleet, pay 500-4000, price 4000.
8. Each players position has to be saved
   (a) When the players get another turn, they continue from their latest position.
9. The game is won, when only one player is left in the game.

(a) A player loses the game, when they have gone bankrupt (account at 0 £ or under).

(b) The other players continue to play.

5.2 Non functional

1. There has to be a minimum system requirements for the game.
   (a) This guide has to include a description of how to import the code from a Git repository.
2. A text has to be displayed, describing the effects of the square a player lands on.
3. The game must display a board.
   (a) This will be achieved by using a Graphical User Interface (GUI).

6 Use case

6.1 Descriptions

Use case: Game.
ID: 1
Brief description: The game is called Monopoly. Two-six players play, until all but one player is bankrupt. Two die are thrown, which determines the field the player lands on and the effect on their account balance.
Primary actors: 1. Players.
Secondary actors: None.
Preconditions: Two-six players enter the game.
Main flow: 1. The game begins when players enter their names and are assigned a color. 2. Include(Roll). 3. The first player rolls the die. 4. Include(Move on the Board). 5. The board prints out the description for the current square. 6. The action of the square takes place. 7. Include(Change Account Balance). 8. The account prints out what the new balance is. 9. The players position is saved. 10. This is repeated for each player and looped. 11. When all but one player is bankrupt, the game terminates.
Postconditions: The winner is announced.
Alternative flows: 1. A player surrenders, the player is removed from the game.

Table 2: Game.

This use case description is an overview of our games flow. Its outlook is of a complete game sequence, from entering your name to winning or loosing. Since the use case is meant as an overview, the different actions of landing on a specific square is not included, but those will be fully described in seperate use cases.

Use case: Roll
ID: 2
Brief description: The die are thrown.
Primary actors: 1. Players.
Secondary actors: None.
Preconditions: The game is executing.
Main flow: 1. A player rolls the die. 2. The result of the roll is determined.
Postconditions: The board recieves the result of the roll.
Alternative flows: None.

Table 3: roll.
Use case description for the function roll. This diagram describes what happens when this function takes place. In Diagram 1, it shows which effect it has on the game.

Use case: Move on the Board.
ID: 3
Brief description: The player moves on the board.
Primary actors: 1. The die.
Secondary actors: None.
Preconditions: 1. The players position is fetched. 2. The board receives a roll from the die.
Main flow: 1. The board places the player on a field depending on the roll. 2. The board prints out which field the player landed on and the result of the roll.
Postconditions: The players position is saved.
Alternative flows: None.

Table 4: Move on the Board.
Use case description for the function of moving on the board. It shows, that the die are the primary actors, which they are because they determine whereto the player moves.

Use case: Change Account Balance.
ID: 4
Brief description: The account balance of a player is changed depending on which square the player lands on and if they choose to buy it, have to pay rent, gets a reward or has to pay a tax.
Primary actors: The board.
Secondary actors: None.
Preconditions: The account recieves information from the board, about which square the player landed on, and what action should be made to the account balance.
Main flow: 1. The amount is either added or subtracted from the players account balance.
Postconditions: The new balance is returned to the board.
Alternative flows: None.

Table 5: Change Account Balance.
Use case description, describes why and what happens when points are added or withdrawn from the point balance. The process requires information from the board.

Use case:
Land on fleet.

ID:
5

Brief description:
If the player lands on a fleet, they either buy it, pays rent or choose to do nothing.

Primary actors:

1. Players.

Secondary actors:
None.

Preconditions:

1. The player has landed on a fleet.
2. The fleet has not been bought yet.

Main flow:

1. The player buys the fleet.

Postconditions:
The board is informed, that the fleet has been purchased.

Alternative flows:

1. If the fleet is already owned by another player.
   (a) The current player has to pay a fee.
      i. The size of the fee is determined by how many fleets are owned by the same player.
         A. 1 fleet owned equals rent of 500 pounds.
         B. 2 fleets owned equals rent of 1000 pounds.
         C. 3 fleets owned equals rent of 2000 pounds.
         D. 4 fleets owned equals rent of 4000 pounds.
2. If the player chooses not to buy the fleet.
   (a) Nothing happens.

Table 6: Land on fleet.
The main flow for this use case is specific to the fleet being available, and the player wanting to buy it. Other options like the fleet already being owned, insufficient funds and declining to buy are alternative flows.

*See remaining use case descriptions in subsection 19.1 on page 30.

6.2 Use Case reasoning

The use cases above details the events possible in a given players turn. We included a big use case (ID:1 Game) that describes the overall flow of the game to give an overview of how we want the game to function. We think this is a good idea because it would let people with no previous knowledge of the game understand what we want quickly, and it can be used to makes sure that the customers vision of the overall game flow is met. The rest of the use case descriptions, are available to read in the Appendix.
Due to the changing instances of tax and rent calculations on owned fields, it was decided to place the owned event as alternative flows for the specific type square. We considered changing this to having a use case for each specific instance of a square being owned for clarity, but time constraints and the practicality of having the 'owned'-event as an alternative flow spoke against it, and in the end it was decided not to make the changes.

6.3 Use Case Diagram

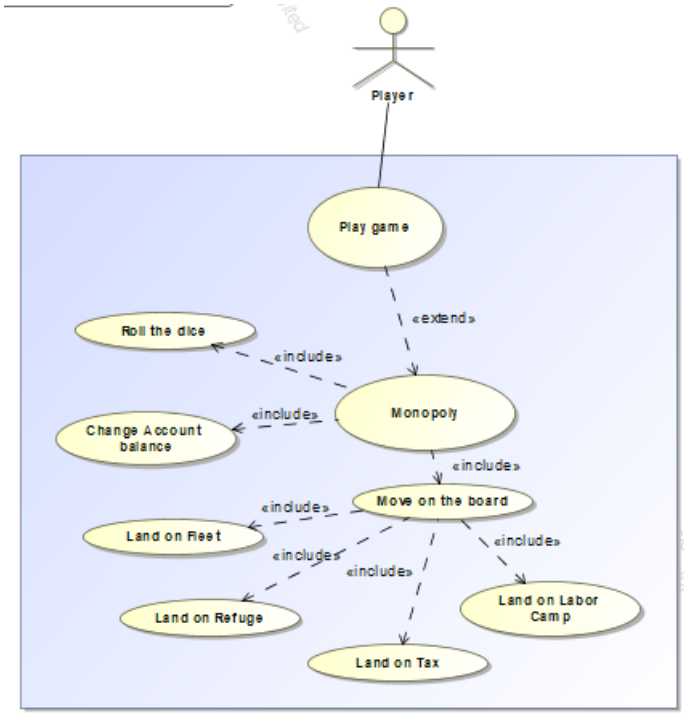


Figure 1: Use Case Diagram

Due to the relative similarity to the actual game of monopoly our Game Usecase was initially labelled as monopoly, we have moved on to the new label "Game" to avoid confusion. The usecase gives a brief overview of the sequence of events in the running of the game. Player access the game and plays → dice are rolled → the player is moved on the board, and a resulting square event transpires (labeled and described in their individual Use cases). We've decided to only include a use case diagram of our 'game flow' use case (ID: 1, Game), since we felt that a visual presentation of the other use cases didn't contribute to a deeper understanding of the game vision.

7 System Sequence Diagram

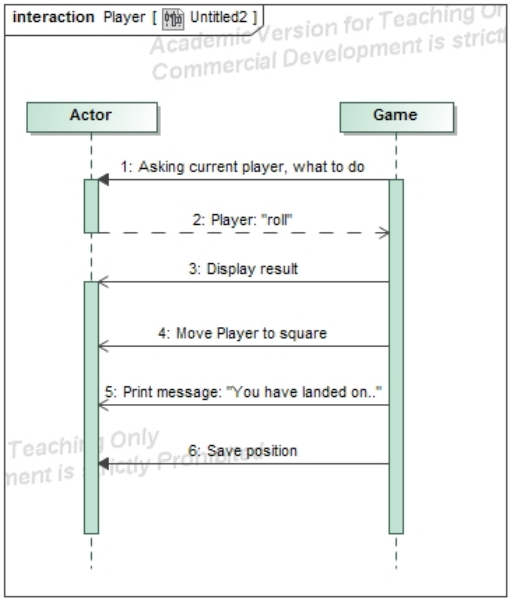


Figure 2: Roll and move.

The diagram above, is the SSD for rolling the dice, and moving on the board combined into one. As seen, first the player has to make a choice, what to do. When the player chooses to roll the dice, the result of the dice are displayed on the GUI. This results in the players piece, moving x amount of spaces, and landing on a square. The GUI displays which square the player landed on, and the belonging message from the square. The players position is now saved.

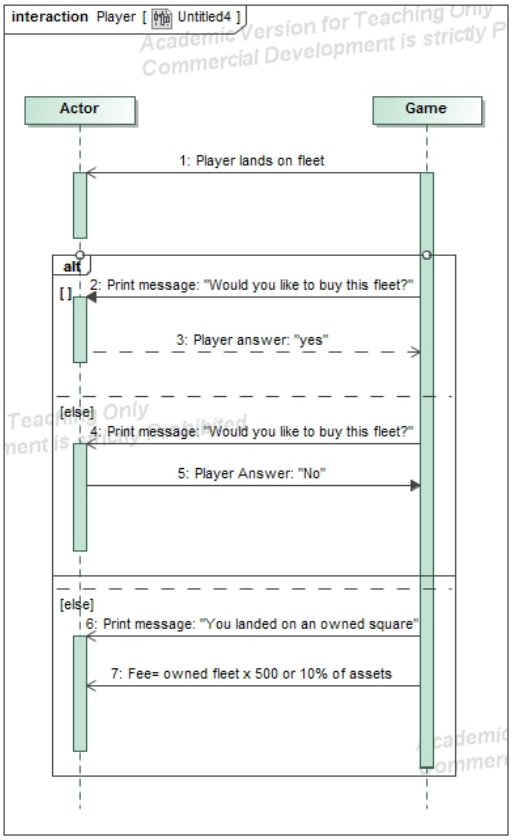


Figure 3: Land on fleet.

This SSD shows the sequence, for landing on a fleet. To begin with, the player lands on the fleet, afterwards there are three possible outcomes. If the fleet has not been purchased yet: the GUI displays a text, asking the player if wanting to buy it. The player either answers yes or no, following two different actions. Thirdly, if the fleet has already been purchased: The GUI displays a text, telling the player, the fleet is owned, and it's pay up time. Then the player has to pay the owner 500, 1000, 2000 or 4000 depending on fleets owned.

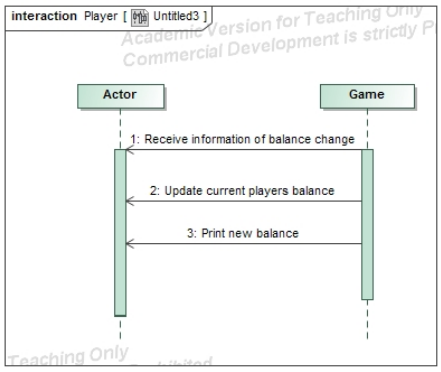


Figure 4: Change account balance.

This last SSD displays the sequence of changing a players account balance. First, the information about adding or subtracting a specific amount is received. Then the balance is updated and displayed on the GUI.

8 Noun analysis and verb analysis

We've reviewed the text of requirements provided by the customer. After careful consideration and discussion, we found the provided text missing key items for successful development. Therefore we have chosen to make a quick recap for analysis:
The game shall be an expansion of the former game requested. First of all, the game board shall contain different types of squares, and contain an actual board. The players has to land on a square, and then continue from there next turn. They have to circulate on the board. The game has to be able to be played by 2-6 players. Every player shall begin with a sum of 30.000 £. The game ends when all but one player, has gone bankrupt.

8.1 Noun analysis

A noun analysis has been included, to develop and retrieve an overview ofprospectable classes.

- Territory
- Refuge
- Tax
- Labor camp
- Fleet
- Messages
- Square
- Players
- Board
- Account
- Dice

8.2 Verb analysis

A verb analysis has been included, to develop the expected methods.

- Roll
- Buy
- Pay
- Move
- Printing
- Saving Position
- Enter name
- Choose color
- Start game
- Change Balance
- Bankrupt
- Ownership

9 Domain model

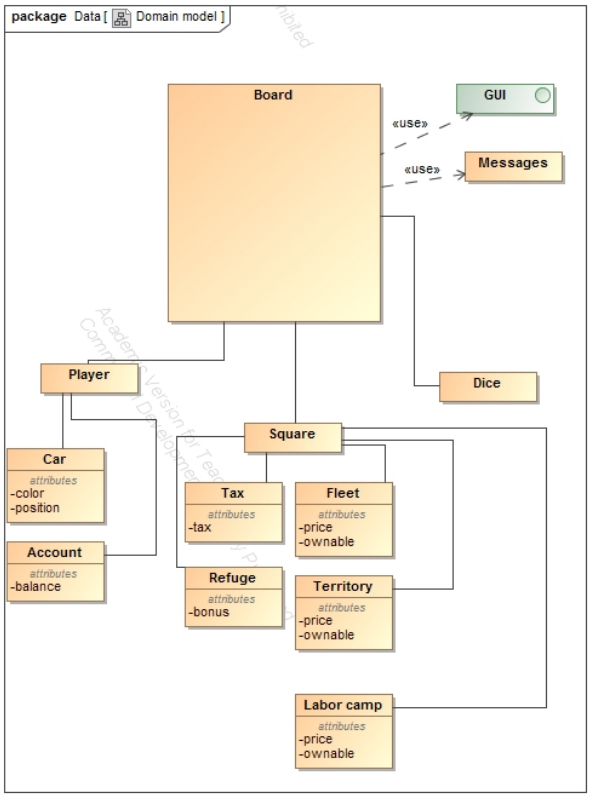


Figure 5: Domain model

From the prospectable classes that we found in our noun analysis, we picked out the following as potential classes in our build;

Board, Player, Square, Dice, Account.

They are all basics in our game and were obvious choices. Since each player has a point balance, we've chosen that an account class is needed. Each player also needs a visual object to follow their progress on our playing board, hence the car class.

The board class will be our main with responsibility of controlling the GUI, creating players, squares and dice. It will also control the game flow since it contains the rules.

The Square class will contain different kinds of squares, we feel that creating a class for each type and using polymorphism seems like the way to go, since we only have two scenarios on most of the squares, buy or pay a tax, but each type of square has a different way of calculating the two.

We've also envisioned a class just for messages for easy translation of all displayed textmessages.

10 BCE diagram

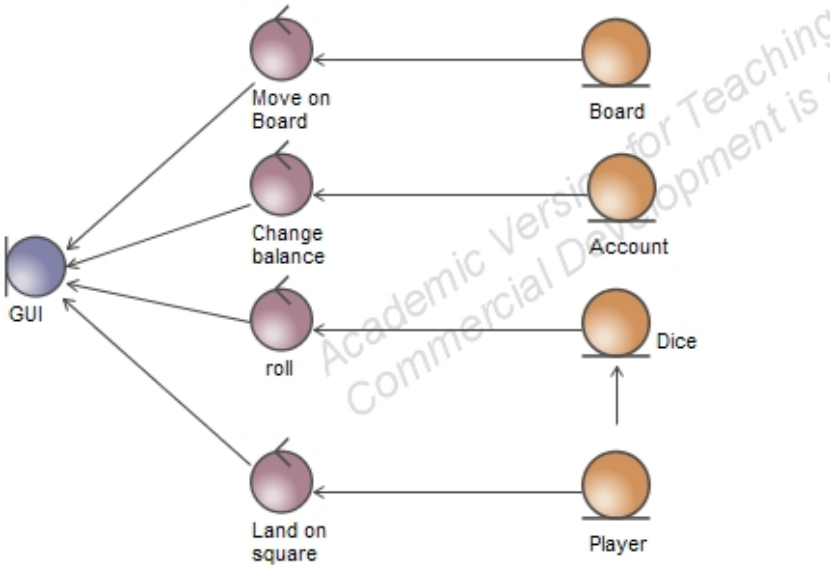


Figure 6: BCE diagram

Creating a BCE for each use case. The diagram above, illustrates the BCE diagram of this project. The diagram is built with classes found in the domain model as entity objects. The four entities included are: Board, Account, Dice and Player. in the model, the players starts their turn, by rolling the dice. From here, two values are generated, these are displayed on the GUI. Afterwards, the board receives this information, so it is able to move the piece on the board. The account receives information from the square landed on, to subtract or add money to the current players account. The control called "Land on square" determines, what happens, when landed on the square. Here they get the opportunity to buy it, if available, pay rent, tax or receive money. All entities here lead to the same boundary, the GUI. The GUI displays everything that happens in the game, visually.

Due to the extent of this game, it has been chosen to make one large, but simplified BCE model. This is instead of one BCE for each use case.

11 Analysis Sequence Diagrams

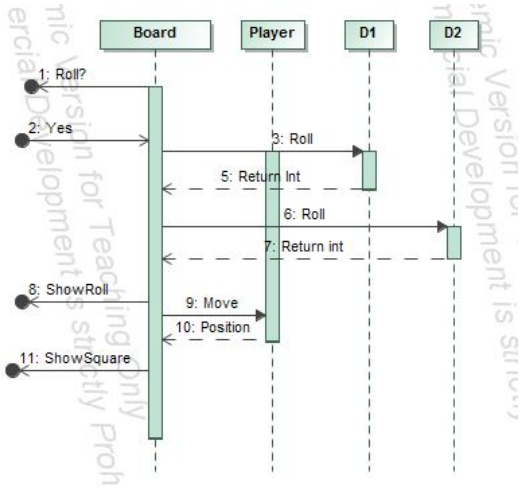


Figure 7: Move on Board

Figure 1 displays the sequence executed, when a player, moves on the board. Four elements are displayed: The board, player and the two dice (D1 and D2). When a player gets a turn, they are asked if they want to roll the dice. When this happens, through the player, the dice are informed. Then the dice each return a value between 1-6. The result is now displayed on the screen, here the GUI. Now, the player moves on the board, and the square landed on is displayed.

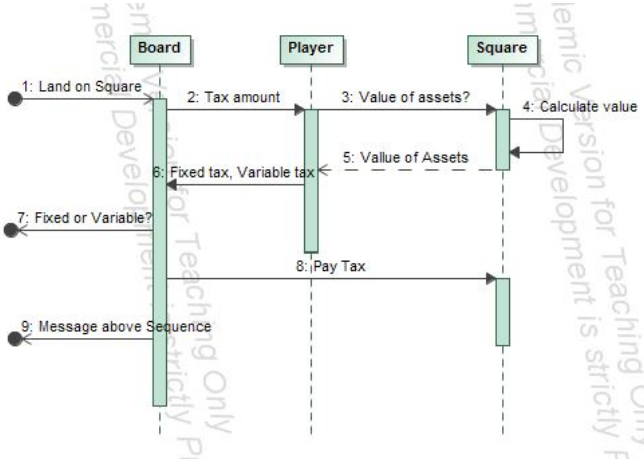


Figure 8: Tax

Figure 2 is a brief description, of what happens, when a player lands on a tax square. In this sequence, there are 3 elements involved: Board, player and square. First of all, the player lands on the square. This square identifies as a tax square. The rate of the tax is sent to the player. After this the total value of the players assets is sent to square. Here 10 % is calculated from the total amount of assets, sent back to the player. This information is sent to the board, and displayed on the GUI. Now the player chooses which to pick.

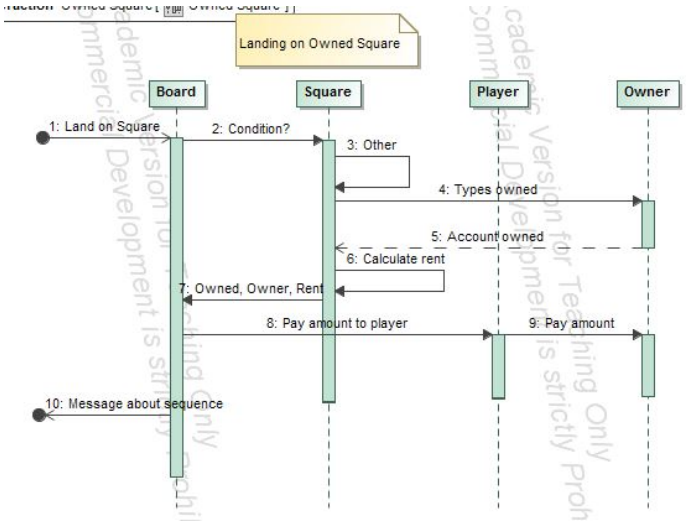


Figure 9: Owned

Figure 3 is a sequence of the what happens, when the square landed on, is already owned. Four elements are included in this: Board, square, player and owner. First condition: The player lands on the square. Now the conditions of the square is evaluated (Is it owned or not?). Then the owner is returned to the square. Here the rent is is calculated, and both the name of the owner and the rent is returned to the board, and displayed on the GUI. After this, the current player, has to pay the rent to the owner.

*The remaining two ASD's are available in Figure 19.2 of the Appendix on page 34.

12 Analysis Class Diagram

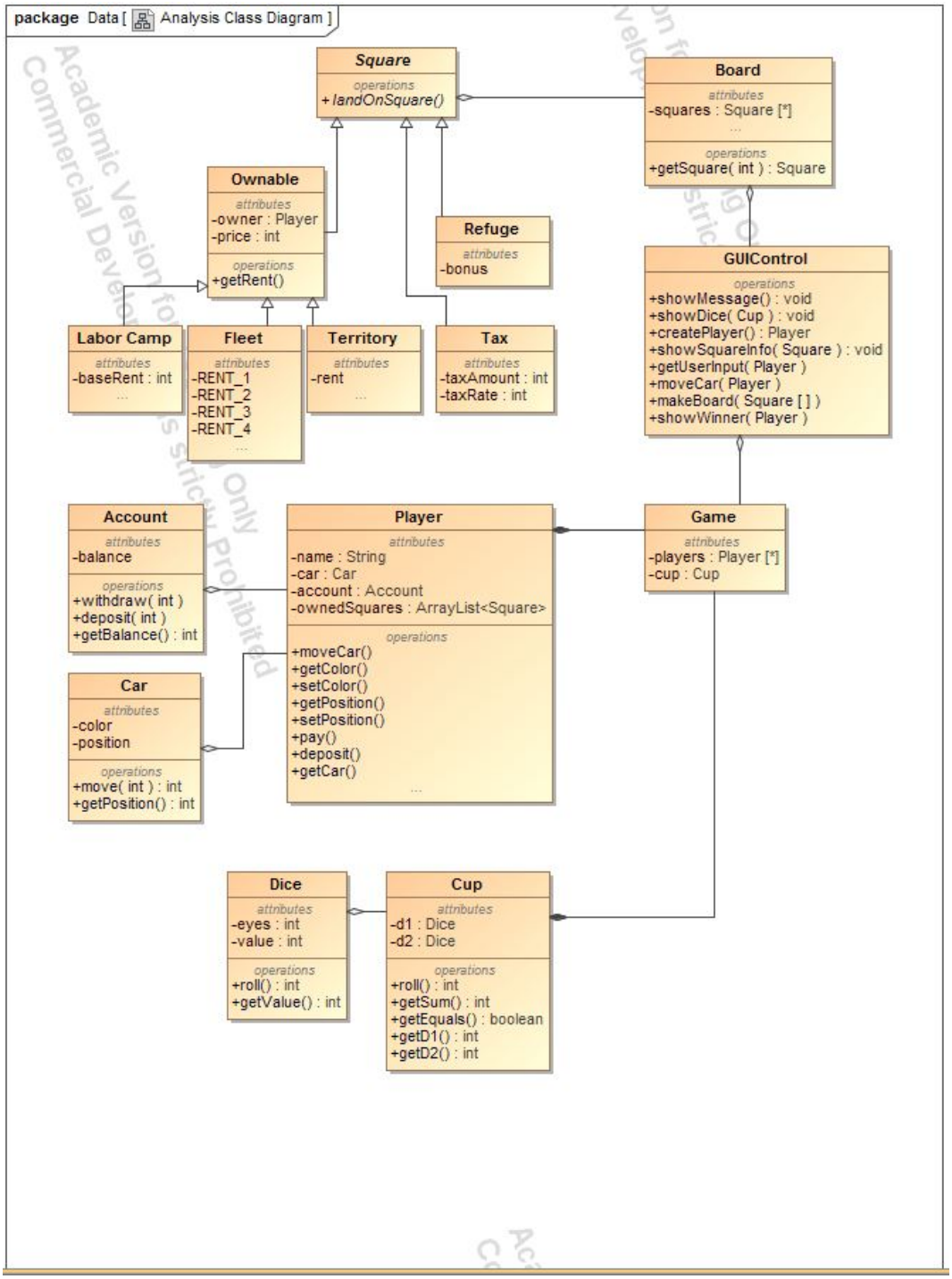


Figure 10: Analysis Class Diagram

The first extensive look at how we expect the game to be designed. The thought behind our diagram and class design is that every class has to extend from a super class that functions as a controller, making the code more accessible and easier to debug and adjust.

We have build on our domain model and used some of the same classes, but after putting in more thought on how we want our program to be, we have evolved on the domain model to create a class diagram with new classes and changes to the old ones. First, we have taken a lot of responsibilities away from the board class, since we felt that the board class in the domain model got too large and did not comply with the rules of GRASP. To do that, we have created the GUIControl class to take care of all the GUI commands. We also feel that managing things like player names and controlling the turns of the battle could be a class of its own, hence the Game class is now a thing.

With the addition of a GUIControl class we now feel that the class "Messages" from the domain model is obsolete, since we will try to keep most messages gathered in the GUIControl class. The original thought behind the messages class was to allow easy translation by gathering all text commands in one class. Creating the GUIControl class also makes it easier for a potential visual update of the game, if the customer should wish to do that in the future. We have also added the Cup class, again with GRASP principles in mind. The Cup will initialize dice objects and have the responsibility of rolling them and calculating the sum.

13 Design Class Diagram

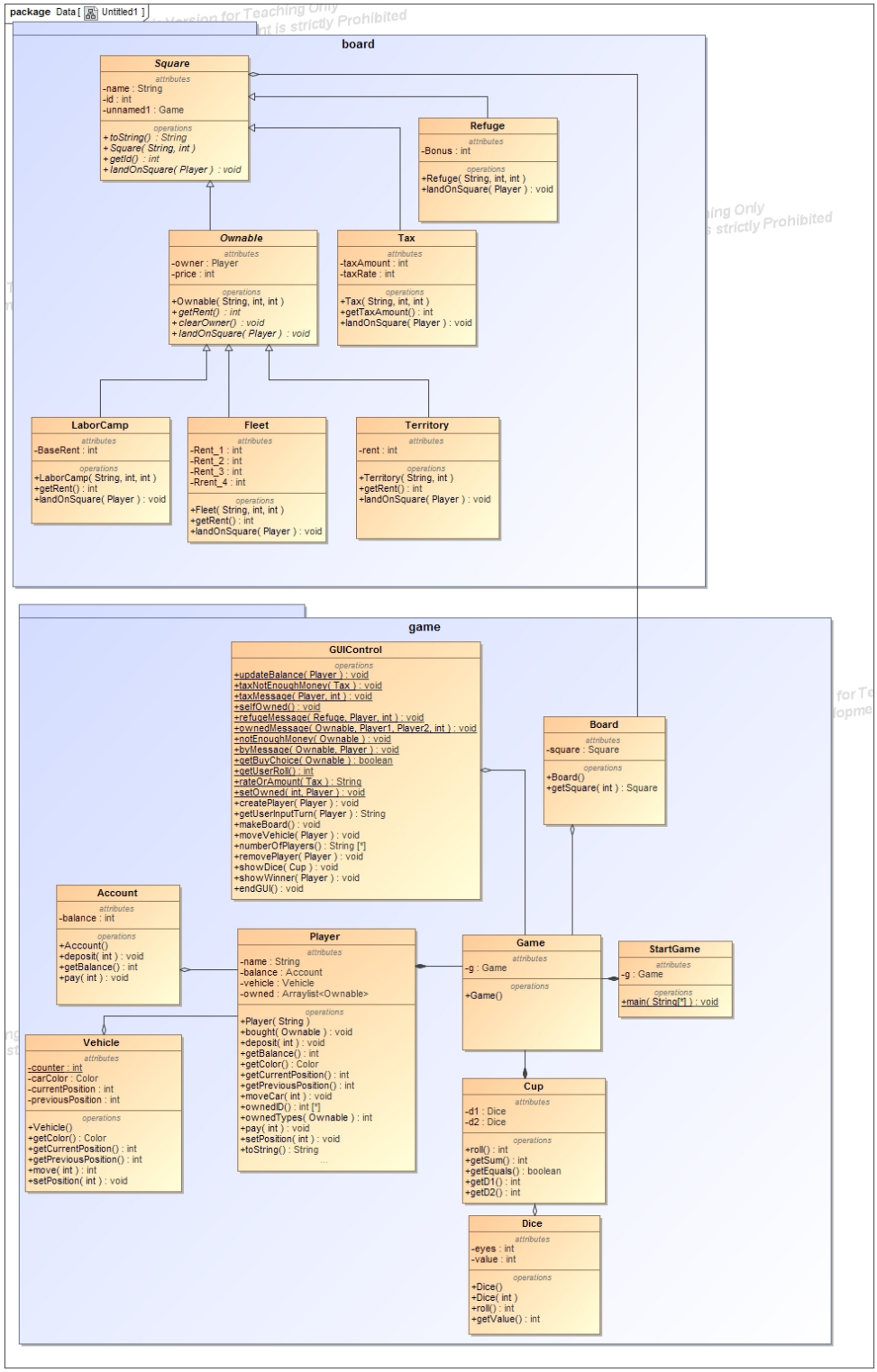


Figure 11: DCD

The diagram above shows, how the classes turned out in the end. This class diagram, can be compared to the Analysis Class Diagram. Doing this, it's seen that they're similar, but there are differences. Looking at the classes, there are only few changes. These include the class car, being changed to be called Vehicle. Furthermore, another class has been added, StartGame. This is the main, that only starts the game. Connection wise, it's basically the same, however, in the actual game/this diagram, Board is not connected to Game through GUIControl. It's is directly connected to Game, and GUIControl is alone connected to Game. StartGame is also connected to Game. Besides these, the biggest change in the diagrams, are the many methods added to the different classes.

14 Classes

14.1 Inheritance, abstraction and polymorfism

In this game we have used inheritance to create the classes Tax, Refuge, LaborCamp, Territory and Fleet. This is a brief explanation of the term inheritance within Java programming, applied to this project.

In Java, a class can inherit attributes and methods from one other class. the concept of inheritance is build on generalization. So when you have classes that have a lot in common, you can create a class, a so called super class, which all the classes in question can inherit from. They will be subclasses of the more general super class. That way, attributes and methods in the superclass can be applied in all subclasses.
Below is a cutout from our class diagram, where inheritance is visualized.

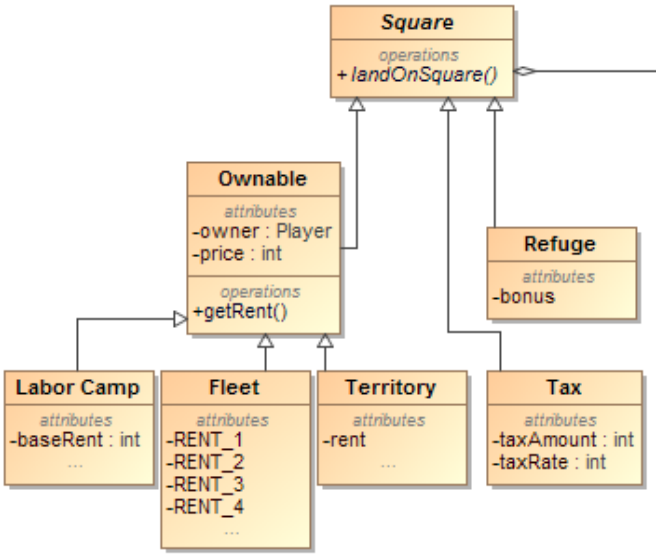


Figure 12: Cutout of Analysis Class Diagram

In the requirements we identified the 5 different types of squares, which generally had the same action performed on it. A player landed on the square, and an action is performed on that player. Therefore this method (landOnSquare()) is a part of the super class Square, and is to be implemented in the sub classes. This is called polymorfism. A good way to explain polymorfism, is to go into the class Ownable. Ownable is created because the subclasses, lowest in the hierarchy, can be divided into squares that can be owned and squares that can't. So, in ownable, the method landOnSquare is implemented, and is general for each of the ownable subclasses. However, if a square is owned, its rent is calculated differently. So we create an abstract method getRent(), implement it in the subclasses and us it in landOnSquare.
The keen observer will now notice that a new word came in to play. Abstract is a keyword applied to classes and methods which we need to extend in subclasses.
Back to Ownable. The method landOnSquare uses the method getRent(), which returns the rent of an owned square. So when landOnSquare is called on a subclass, maybe Fleet, it calls the landOnSquare of its superclass,

Ownable, which then again applies the specific getRent() in Fleet.
That is what is known as polymorfism.

14.2 ”board” package

14.2.1 Square

We made a Square class to be the super class for all types of Squares in the game.
This class is abstract, with the abstract method landOnSquare, to be implemented in every subclass. The method landOnSquare accepts an instance of type Player as a parameter, to simulate that the player lands on the field. Square also has the method getID() which returns the integer id of that square. The id value is assigned in the constructor.

14.2.2 Refuge

Refuge inherits from Square, and implements the abstract method landOnSquare.
This method adds a bonus to the players balance.

14.2.3 Tax

Tax inherits from Square, and implements the abstract method landOnSquare.
This method subtracts a tax rate or fixed tax amount from the players balance. The player who landed on the field has a choice

14.2.4 Ownable

Ownable is also an abstract class which extends Square.
It is the super class to all three ownable types of squares.
It has an integer price and an instance of Player, owner. This way, if a square is owned, the square knows who owns it. When another player lands on an owned square, that square has access to the owner, and the player that landed on it. That way it can operate the two players account in a method landOnSquare which is inherited to every ownable subclass.
Ownable has an abstract method getRent, which is used in landOnSquare. This method needs to be implemented differently in the three subclasses, because the rent is calculated differently for each type of ownable square.

14.2.5 Fleet

The class Fleet extends Ownable, and has 4 constant values, which are the rent levels. When Fleet is owned, the rent depends on how many instances of Fleet the owner owns.

14.2.6 LaborCamp

The class LaborCamp extends Ownable, and has a baseRent integer. When an instance of LaborCamp is owned and a player lands on it the getRent method calculates the rent by rolling the cup and multiplying the result of that roll with the number of labor camps the owner owns, and the baseRent.

14.2.7 Territory

Territory is extended from Ownable. Each instance of Territory has a different rent, which is payed if the square is owned.

14.3 ”game” package

14.3.1 Dice

We copied the Dice class from CDIO—2. In this project we also made a class FakeDice, which inherited from the Dice class. This FakeDice is only used to test the methods of the Dice Class, and is located in the test package.

14.3.2 Cup

A class cup was made. It instantiates two Dice, and has methods for rolling, showing the sum of the current value of the two dices, showing the current value of each dice, getting a boolean value of whether the two Dices has the same value.
We also made a class FakeDice, which inherited from the Dice class. This FakeDice is only used to test the methods of the Dice Class, and is located in the test package.

14.3.3 Player

The player class is the class that controls the identity of the player itself. It is the controller for the vehicle and account classes. It also holds a list of every square that the player owns. Upon initializing a player, the class first accepts a name, then uses the deposit method from account to deposit the starting the balance. Finally it initializes a vehicle for the player. The majority of the methods in the player class call into account and vehicle. Within the player class itself, there are several methods for adding and checking what squares a player owns. There's a method for adding squares to the list, to return the ones owned, to return the IDs of the ones owned, and finally for checking how many of each specific type a player owns.

14.3.4 Vehicle

The vehicle class first and foremost calculates and controls the position of the player. It is able to calculate modulo, such that the player position "restarts" once the final square on the board is reached. It holds both the previous and the current position of the player. Furthermore, when the vehicle class is initiated, it will initiate with a color depending on the number of the player initiating it. This will give each vehicle a different color.

14.3.5 Account

The account class holds the balance of each player. It initiates with a balance of 0. Using the deposit method, money is able to be deposited into the player's account. The pay method withdraws money from the account. Using the getBalance method, the current balance is able to be reached when other classes need it.

14.3.6 GUIControl

The GUIControl class does much as the name implies. It is this class that controls all messages that are pushed to the GUI, it initializes the board and it controls all interactions the player has with the board. The class has methods for initializing a GUI representation of the board and every square on it, the players and the dice. It has methods that pushes a request for input to the players and methods that pushes a GUI message depending on the square.

14.3.7 Game

The Game class is the class the runs the game. The game is run in the constructor, so the only thing needed in the StartGame(main), is the Game constructor. In game, the logic behind the turn of the players is controlled by this class.

15 Design Sequence Diagram

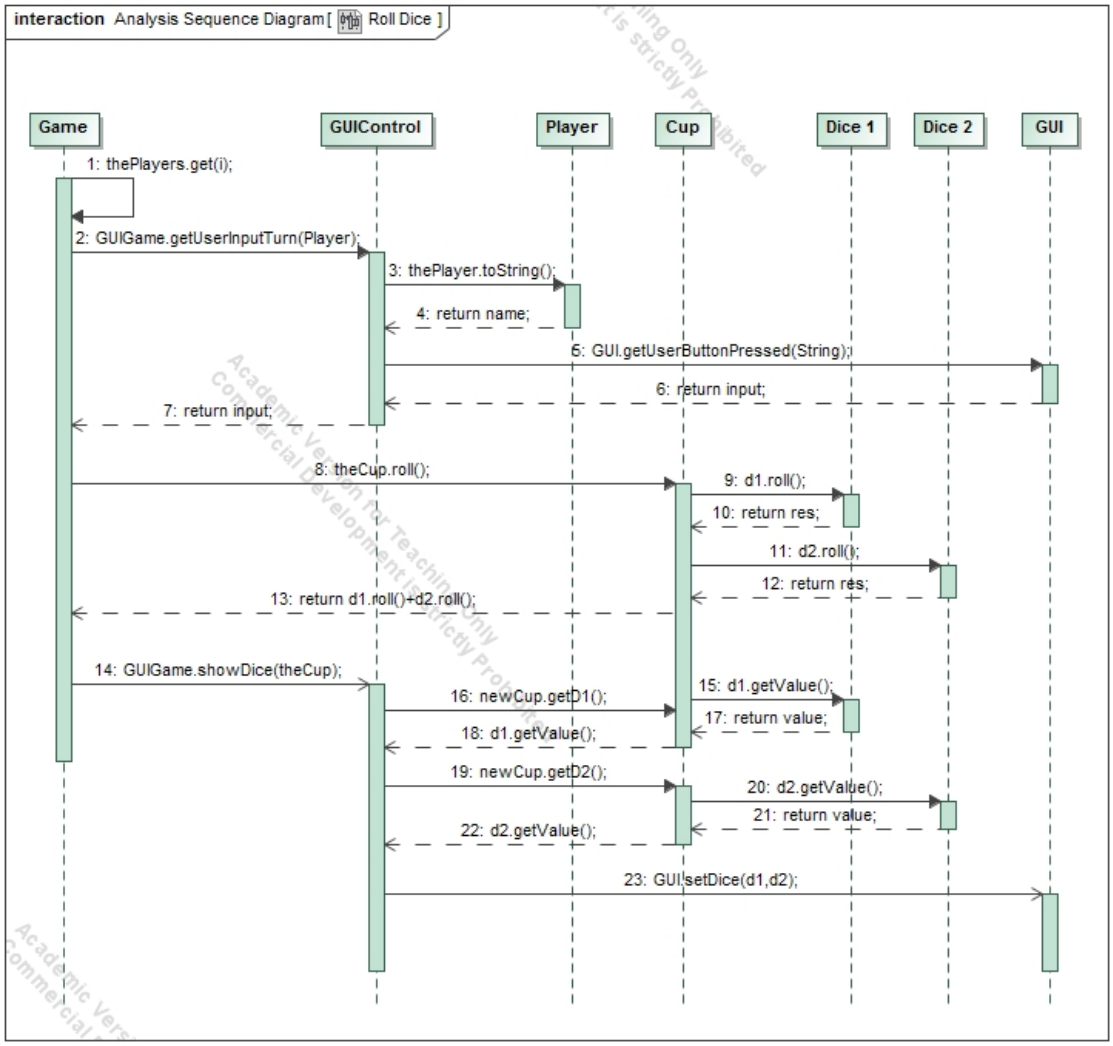


Figure 13: Roll

This figure is a representation of how the code acts when a player rolls the dice during their turn. The ASD "Move on Board" was split into two different DSD's, namely this one, and the DSD for "Move on Board". It was concluded that the actual code behind both rolling the dice and moving on the board were too big to be included in a single DSD, and that the use cases should be split up into two. This DSD includes the Game, GUIControl, Player, Cup and two Dice classes. The overall sequence of events are the same from the ASD depicting a roll. The main difference, and the reason a DSD was deemed necessary over this sequence, was the inclusion of the GUIControl and Vehicle class. Where it was assumed during the analys phase, that a "Board"-class would control the GUI, it was decided to create an entire class for managing the GUI, which on the DSD is represented by the "GUIControl"-class. Furthermore, a "Cup"-class was created to create a class that would handle the resulting die rolls, instead of letting the "Game"-class handle this.

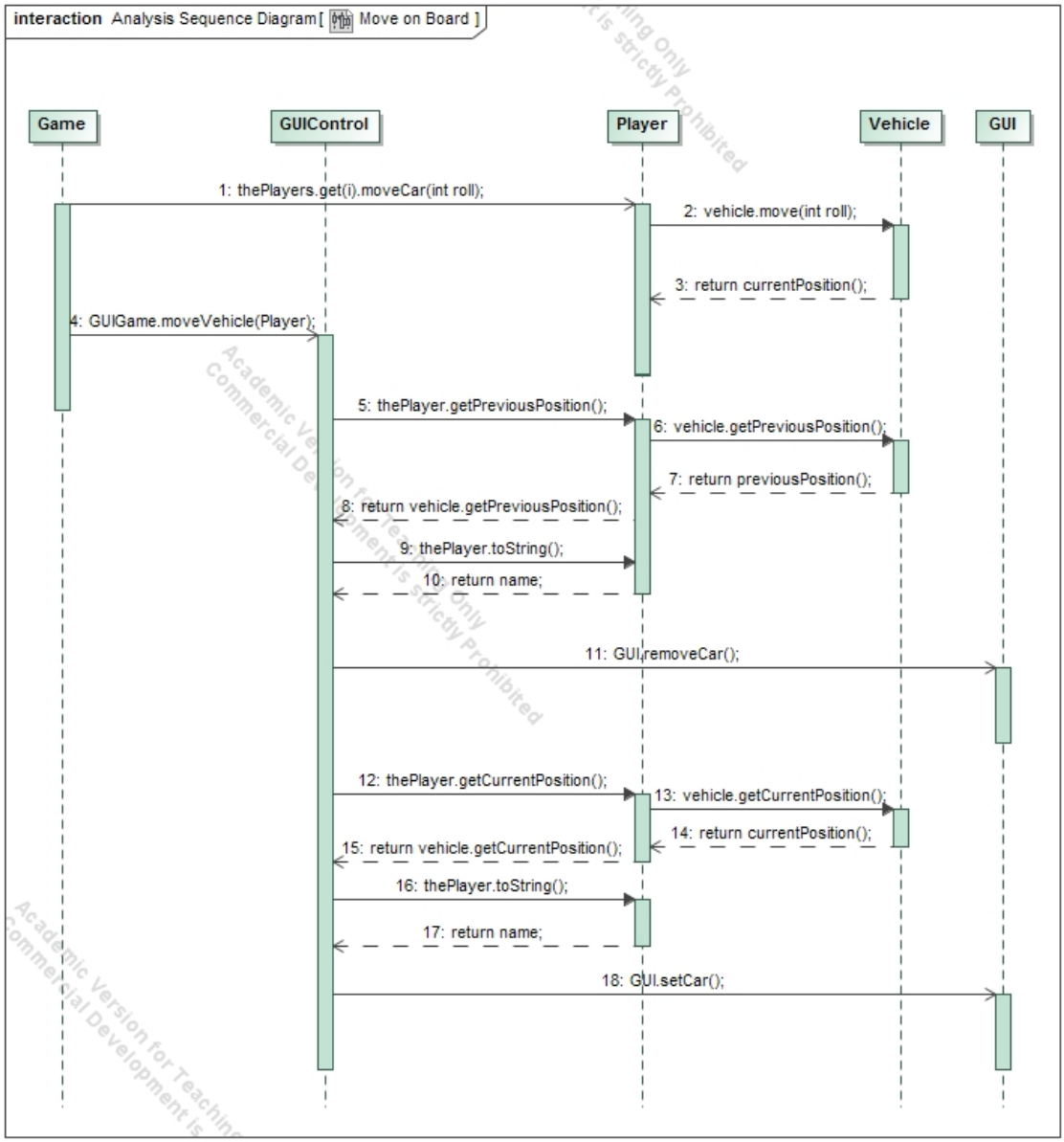


Figure 14: Move on Board

This DSD is the second part of the original "Move on Board"-ASD. This DSD assumes that the player has already rolled the die and that the player is about to move his or her piece on the board. Where it differs from the "Move on Board"-ASD is once more in the inclusion of the GUIControl, but also the "Vehicle"-class. Originally it was thought that the "Player"-class would control the movement. This turned out to be a bigger task, and the responsibility was given to the "Vehicle"- and "GUIControl"-class in conjunction. Once the player has rolled the dice, it is send to the "Player"-class, that then sends to the "Vehicle"-class that calculates a new current position. Herefter, the "Game"-class sends to the "GUIControl"-class that the piece is to be moved. This calls to the "Player"-class, then the "Vehicle"-class for the new and old position. The piece is then removed from the old position in the GUI, and placed on the new position.

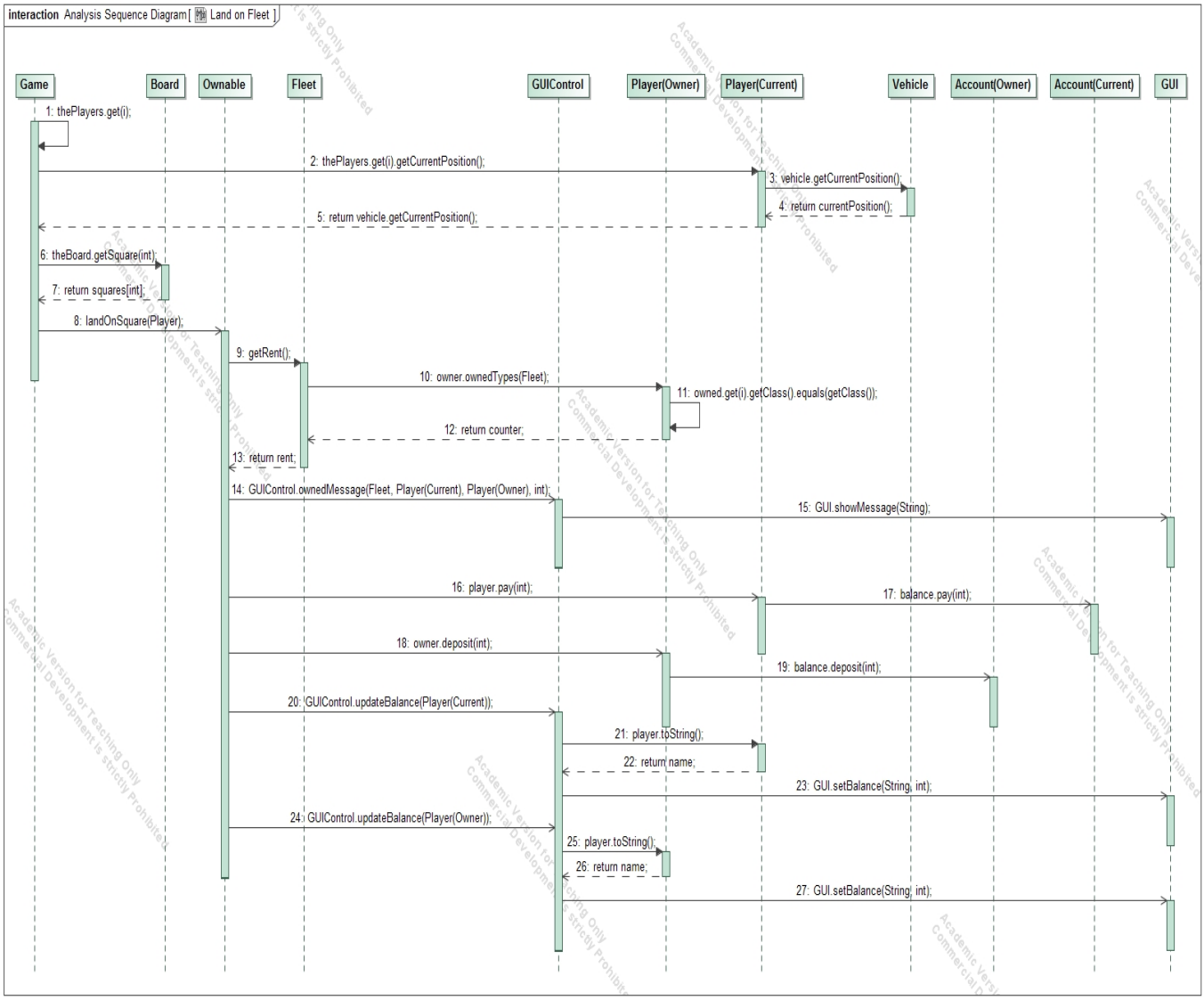


Figure 15: Land on Fleet

The final DSD included is one that looks at the sequence when a player lands on a square of the type "Fleet". During the analysis phase, an ASD was designed for landing on a general ownable square. Landing on "Fleet" was chosen as this would also show the money interaction between players. Once it is determined what square the player has landed on, and that it is owned, the "Ownable"-class takes control of most of the sequence. It calls to the specific type of square for the rent, which then calls to the owner of the square, to determine how many squares of the type that the player controls. The "Ownable"-class the sends to GUIControl to print to the GUI, and begins the withdraw and deposit money from the current player to the player who owns the square, until the resulting account balances are printed to the GUI.

16 Test

16.1 Summary

This chapter descripes the way the game has been tested, to ensure it works, and catch any flaws there might be. To this, JUnit testing has been used, to test the individual classes eligible to this type of testing. The rest, including the gathered game, is tested by a simple user test.

16.2 Documentation

16.2.1 Tests

Table 7: Methods in Dice

Test case ID	TC 1
Git-Hub Location	TestDice
Summary	Testing the operations which can be used on an object of the type Dice.
Requirements	
Test Procedure	The test uses a FakeDice, extended from Dice, which always shows the same value. The methods roll() and getValue is then used, and if the methods returns the expected values the test is passed
Test data	The FakeDice is initiated with the value 3.
Expected result	getValue() will return 3. roll() will return 3.
Actual result	getValue() did return 3. roll() did return 3.
Status	Passed.
Tested by	Mathias Tværmose Gleerup
Date	21/11-2016
Test environment	Eclipse Neon, v. 4.6.0

Table 8: Probability test

Test case ID	TC 2
Git-Hub Location	TestDice
Summary	This test uses a mean value of 3.5 (the mean of 1,2,3,4,5,6), to determine whether the dice is fair.
Requirements	This test requires that the status of TC1 is 'Passed'.
Test Procedure	A dice is rolled 100000 times, and the mean is calculated. If the mean is within the acceptable range of the mean value (3.5), the test is passed.
Test data	Rolled 100000 times. Upper threshold: 3.55. Lower threshold: 3.45.
Expected result	The mean value of the 100000 rolls is expected to be between the established thresholds.
Actual result	Mean value was between the thresholds.
Status	Passed
Tested by	Mathias Tværmose Gleerup
Date	21/11-2016
Test environment	Eclipse Neon, v. 4.6.0

Table 9: Methods in Cup

Test case ID	TC 3
Git-Hub Location	TestCup
Summary	Testing the methods in Cup, using a class FakeCup which extends Cup.
Requirements	The Cups probability test is mute, because the Cup uses two Dices. Therefore, the Dices probability test needs to be 'Passed'.
Test Procedure	The test creates a FakeCup, with two FakeDices with the same value. The test asserts the value of return methods in FakeCup. Then a new FakeCup is created with different values, and the methods are tested again.
Test data	The first FakeCup has the values (5,5) The second FakeCup has the values (4,6).
Expected result	Depends on the method.
Actual result	All results were correct.
Status	Passed
Tested by	Mathias Tværmose Gleerup
Date	21/11-2016
Test environment	Eclipse Neon, v. 4.6.0

*See remaining tests in subsection 19.3 of the Appendix on page 35

Table 10: User Test

Test case ID	TC 9
Summary	testing if the game can be started and a turn can be taken by all players (in this case 3)
Requirements	a full turn can be run as expected each player can be created and named. each player can take a turn. the results of the turn is shown correctly. each take a turn. player 1 and player 2 surrender. player3 wins.
Preconditions	
Test Procedure	set player count to 3 name players, player1, player2, player3 press roll to start turn. choose to buy or not (in case of tax choose either fixed amount or percentage) press ok once the message is read. do this until all players have had one turn. player1 and player 2 surrender
Test data	see code
Expected result	3 players created and named. one turn run without problems. two players able to surrender, remaining player wins.
Actual result	the results fit the expected results.
Status	Passed.
Tested by	Mikkel Geleff Rosenstrøm
Date	25/11/2016
Test environment	Windows 10

16.3 Test Conclusion

An error in the Account class was found and was identified by testing the squares. The error was fixed and the testing as a whole was above expectations. Due to similarities of the square tests only two types are included (Refuge and Tax)in the test documentation although the tests can be found on github. The testing went as expected although it took longer than planned

16.4 How to import code from GitHub

1. Open Eclipse.
   (a) In the top left corner click File.
   (b) Fourth from the bottom, click Import.
   (c) Find the folder called Git.
   (d) Choose the subfile Projects from Git.
   (e) Click Clone URL.
2. Go to your browser and open GitHub.
   (a) Sign in to your account and double click on the wanted project.
   (b) On the right side of the screen, is a green button with the text "Clone or download
   (c) Copy the link in the box.
3. Go back to Eclipse.
   (a) Under Location, paste the URL, in the top box "URL".
   (b) Enter your GitHub account information in the "Authentication" section.
   (c) Check the box "Store in secure store".
      i. This is to make it wasier for yourself, while working in the java project.
      ii. If you choose not to do this, you will have to enter your account information, every time you want to push(upload) an alteration to GitHub.
   (d) Click next two times.
   (e) Ensure the box "Import existing Eclipse projects" is checked, and click next.
   (f) You have now implemented the project. Click finish.

16.5 System requirement

Minimum

1. Java version 8 update 111
2. Command language interpreter (CLI) for running the java file.

17 Conclusion

The creation of the program was concluded as expected and on time to specifications.
We looked at the relations between the different classes and concluded through testing, both JUnit and user tests, that the program worked as outlined by the specifications detailed in the requirements section.
Despite scheduling conflicts there has been little, to no friction between members and the cooperation has resulted in a correct program. We clearly made progress in accommodating different skill levels and pre-existing knowledge. Furthermore, we feel that we underestimated the extent of the documentation of the tests, and the implementation of the GUI.

18 Litteraturliste

References

[1] Craig Larman, Applying UML and Patterns 2004.

[2] Lewis and Loftus Java Software solutions 7th ed.

19 Appendix

19.1 Use Case Descriptions

Use case: Land on Territory
ID: 6
Brief description: If the player lands on a Territory, they either pay, buy or nothing.
Primary actors: 1. Players.
Secondary actors: None.
Preconditions: 1. The player has landed on a Territory 2. The Territory has not been bought yet.
Main flow: 1. The player buys the Territory.
Postconditions: The board is informed, that the Territory has been purchased.
Alternative flows: 1. If the Territory is already owned by another player. (a) The current player has to pay a fee. (b) The fee is determined, by the number of Territories owned. (c) If the player cannot afford the fee. The player loses. 2. If the player has insufficient funds. (a) The opportunity to buy the Territory is still displayed. (b) The purchase will be denied. 3. If the player chooses not to buy the Territory. (a) Nothing happens.

Table 11: Land on Territory.
This use case description, is case specific in the main flow, to the Territory being available and the player buys it. Again, all other cases for the field is described in the alternative flows section.

Use case:
Land on labor camp.

ID:
7

Brief description:
If the player lands on a labor camp, they either pay, buy or nothing.

Primary actors:

1. Players.

Secondary actors:
None.

Preconditions:

1. The player has landed on a labor camp.
2. The labor camp has not been bought.

Main flow:

1. The player buys the Labor camp.

Postconditions:
The board is informed, that the fleet has been purchased.

Alternative flows:

1. If the labor camp is already owned by another player.
   (a) The current player has to pay a fee. The fee is determined by:
      i. The player rolls the dice.
      ii. The value of the dice is multiplied by 100.
      iii. If the owner possesses more than one labor camp, the fee is multiplied by the number owned.
   (b) The player cannot afford the fee. The player loses.
2. If the player has insufficient funds.
   (a) The opportunity to buy the labor camp is still displayed.
   (b) The purchase will be denied.
3. The player chooses not to buy the labor camp. Nothing happens.

Table 12: Labor camp.
This use case description, is also case specific in the main flow, so the player buys the sqaure. Here also, all other options is described in the alternative flows section.

Use case: Land on tax.
ID: 8
Brief description: If the player lands on tax, they have to pay a tax.
Primary actors: 1. Players.
Secondary actors: None.
Preconditions: 1. The player has landed on tax.
Main flow: 1. The player has to pay a tax, by choice: (a) A fixed amount. (b) 10 % of all assets.
Postconditions: 1. The board is informed, that the fleet has been purchased.
Alternative flows: 1. If the player cannot pay the tax. The player loses.

Table 13: Land on tax.
In this use case description, the player chooses what to pay. They can choose a fixed amount or 10 % of all assets.

Use case: Land on Refuge.
ID: 9
Brief description: If the player lands on a refuge, they receive money.
Primary actors: 1. Players.
Secondary actors: None.
Preconditions: The player has landed on a Refuge.
Main flow: 1. The player is awarded money.
Postconditions: The board receives information.
Alternative flows: None.

Table 14: Refuge.
In this use case description, the player receives a bonus.

19.2 ASD

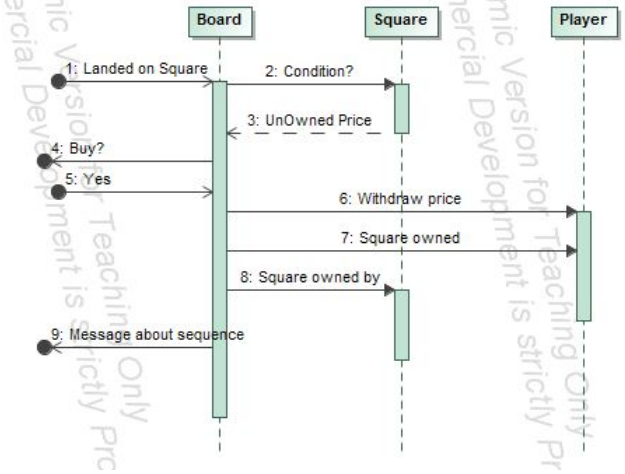


Figure 16: Unowned

Figure 4: When the square is not owned. Three elements take part in this: Board, square and player. The playe rlands on the square, and again here, the condition of the square is evaluated. Here the square returns to the board, that the square is unowned. Now the player get the opportunity to buy the square. In this figure, the player chooses to buy the square. Then the board send information to the player, to withdraw the price, and that the square now is owned. Also the square receives information, about whom owns the square now.

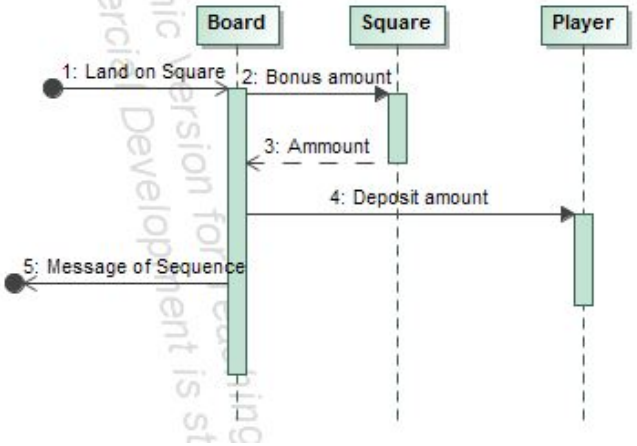


Figure 17: Refuge

Figure 5 briefly describes what happens, when you land on a refuge. A sequence including three elements: Board, square and player. When landed on the square. The board asks the square, how big a bonus, the player sahll receive. The amount is then returned to the board. Then the amount is deposited to the player.

19.3 Test Cases

Table 15: Methods in Account

Test case ID	TC 4
Git-Hub Location	TestAccount
Summary	Tests the methods in the class Account by creating an account. Then an amount is deposited and withdrawn, checking the balance in between.
Requirements	
Test Procedure	The test initiates by creating an account. The integer 30000 is then deposited to the account. Afterwards the deposited amount was checked with getBalance. This procedure was repeated with the pay method, which withdrew 10000. Finally it was tested if the correct amount was present in the account with getBalance.
Test data	Deposits the integer 30000 Withdraws the integer 10000.
Expected result	First test expects the integer 30000. Second test expects the integer 20000.
Actual result	The results fit the expected results.
Status	Passed.
Tested by	Simon Lundorf.
Date	22/11/2016
Test environment	Eclipse Neon, v. 4.6.0

Table 16: TestBoard

Test case ID	TC 5
Git-Hub Location	testBoard
Summary	Tests the method in the class Board, by creating a board. Then, the content of a specific index in the array is tested, by the toString method.
Requirements	
Test Procedure	The test is initiated, by creating a Board. The method getSquare is then tested, by choosing the square 16. Using assertEquals, calling the specific square, and afterwards using toString. This way, it checks the content.
Test data	Checking string ”Palace Gates”.
Expected result	The expected result, was that the square contained the string ”Palace Gates”.
Actual result	The result of the test, is as expected.
Status	Passed.
Tested by	Sofie Freja Christensen.
Date	22/11/2016
Test environment	Eclipse Neon

Table 17: TestVehicle

Test case ID	TC 6
Git-Hub Location	TestVehicle
Summary	Tests the methods in the class Vehicle. Creates a new vehicle, sets and tests the position, then tests moving the vehicle. Finally tests that the color of the vehicle is properly initialized.
Requirements	
Test Procedure	The test initiates by creating a vehicle. A position for the vehicle is then set, and it is tested if the current position fits the set position. The vehicle is then moved using the move method, and the new current position and the previous position are tested. The vehicle is then moved again, and the position is checked to see if modulo is calculated correctly. Finally it is tested if the color of the vehicle is correct, and whether or not a new vehicle is assigned the next correct color.
Test data	The position is set to integer 10. The vehicle is then moved by integer 10. A new position is set to integer 20 and the vehicle is moved by integer 10. Finally the color of the vehicles are set.
Expected result	The first test expects the integer 10. The second test expects the integers 20 and 10. The third test expects integer 9. The fourth test expects the color code RGB[0, 0, 255] and RGB[0, 255, 0]
Actual result	The test results fit the expected results.
Status	Passed.
Tested by	Simon Lundorf.
Date	22/11/2016
Test environment	Eclipse Neon, v. 4.6.0

Table 18: TestPlayer

Test case ID	TC 7
Git-Hub Location	TestPlayer
Summary	Testing that Balance start out as required and it is possible to add and subtract. Whether the player can move to a new square and whether the players piece can move and is assigned a colour. Finally there is a test if the player can buy squares and whether those squares are a part of the right hierarchy
Requirements	
Test Procedure	The test initiates by creating a player "Simon" and five squares. two of Fleet and three LaborCamp respectively. It then asserts that the created player is indeed simon and the players balance is indeed 30000. The balance is then changed up to 40000 and down to 20000 to confirm that pay and depost works. The players vehicle is then moved to another square from square 0 to 5 and 20 from 5 respectively Next we test that the colour of the car is indeed blue. Thus confirming that the switch assigning colour works. The player the purchases 2 squares from the fleet group and 3 squares form the LaborCamp group and we test if those squares are indeed from the LaborCamp Supergroup despite being labeled differently.
Test data	A player called Simon is created Two Fleet squares are created , labelled Boat and Carrier. Three LaborCamp squares are created, labelled Slave Pit, Quarry, Mine. Playername Simon is then tested as a string. The player balance is set to integer 30000 Increased by integer 10000 Decreased by integer 20000 The player position moved by integer 5 The player position moved by integer 15 The players vehicle is set as Blue The player account buys two tiles Fleet tile: boat and carrier The player account buys three tiles: , slavepit , quarry , mine. the owned squares are set in an array The arrayed squares are tested for ownership.
Expected result	The first test expects a player with string Simon as name the second tests expects a balance change of integer 30000 to integer 40000 the third test expects a balance change from integer 40000 to integer 20000 The fourth test expects the position set to integer 5 from integer 0 the fifth tests expects the position set to integer 20 from integer 5 the sixth test expects the vehicle to be blue. the seventh test expects the purchase of five squares. two Fleet, Boat and Carrier. Three LaborCamp, slavepit,quarry, mine. The eighth test expects to find the 5 tiles in an array. The ninth tests expects the arrayed squares to be identified as owned.
Actual result	The actual tests fit the expected results
Status	Passed
Tested by	Simon Lundorf
Date	22/11/2016
Test environment	Eclipse Neon

19.3.1 Test Dice

Table 19: TestRefuge

Test case ID	TC 8
Git-Hub Location	TestRefuge
Summary	Testing the properties of the refuge Square. Namely that it is possible for a player to incur three specific actions. add, subtract or nothing. We also determine if the Square are in fact the refuge Class.
Requirements	
Test Procedure	The test initiates a player named Anders And (setUp). and the players balance is set at 1000 Three squares from the Refuge class are then created All the squares are labeled with a text (Helle 1200, Helle 0 and Helle -200) It the tests the entities player, refuge200, refuge0 and refuge-200 to determine whether they contain information and whether refuge200, refuge0 and refuge-200 are indeed a part of the refuge class. It then performs two related group tests, two for each Square (Refuge200, refuge0, refuge-200) and tests what happens what player with a balance of 1000, lands on a specific tile one and twice. It then resets the player Anders And(tearDown)
Test data	Player name Anders and is created with integer 1000 to balance Three squares of the refuge Class are created. Testing whether the player contains data Testing whether the Squares contain data Testing whether the Squares are indeed from the Refuge Class Testing Refuge200 with expected integer 1000 and actual integer equals balance Testing whether expected and actual are indeed even. Testing Refuge200 with expected integer 1000+200 and actual integer equals balance Testing whether expected and actual are indeed even. Testing Refuge200 with expected integer 1000+200+200 and actual integer equals balance Testing whether expected and actual are indeed even. Testing Refuge0 with expected integer 1000 and actual integer equals balance Testing whether expected and actual are indeed even. Testing Refuge0 with expected integer 1000+0 and actual integer equals balance Testing whether expected and actual are indeed even. Testing Refuge0 with expected integer 1000+0+0 and actual integer equals balance Testing whether expected and actual are indeed even. Testing Refuge-200 with expected integer 1000 and actual integer equals balance Testing whether the Refuge Class allows a negative deposit once and twice

Expected result	first test expects the player to contain data the second test expects the square refuge200 to contain data the third test expects the square refuge0 to contain data the fourth test expects the square refuge-200 to contain data the fifth test expects the square refuge200 to be of the refuge class the sixth test expects the square refuge0 to be of the refuge class the seventh test expects the square refuge-200 to be of the refuge class the eight test expects the balance of integer 1000 to change to integer 1200 the ninth test expects the balance of integer 1000 to change to integer 1400 the tenth test expects the balance of integer 1000 to change to integer 1000 the eleventh test expects the balance of integer 1000 to change to integer 1000 the thirteenth test expects the balance of integer 1000 to change to integer 1000 the fourteenth test expects the balance of integer 1000 to change to integer 1000
Actual result	the actual results fit the expected results
Status	Passed
Tested by	Mathias Tværmose Gleerup
Date	25/11/2016
Test environment	Eclipse Neon

Table 20: TestTax

Test case ID	TC 9
Git-Hub Location	TestTax
Summary	Testing the properties of the tax Square. Namely that it is possible for a player to incur three specific actions. add, subtract or nothing. We also determine if the Square are in fact the tax Class.
Test Procedure	The test initiates a player named Anders And (setUp). and the players balance is set at 1000 Three squares from the Tax class are then created All the squares are labeled with a text (Helle 1200, Helle 0 and Helle -200) It the tests the entities player, tax200, tax0 and tax-200 to determine whether they contain information and whether tax200, tax0 and tax-200 are indeed a part of the tax class. It then performs two related group tests, two for each Square (tax200, tax0, tax-200) then test what happens what player with a balance of 1000, lands on a specific tile one and twice. It tests if a square can be set to have a custom tax (-1000/10) It then resets the player Anders And(tearDown)
Test data	Player name Anders and is created with integer 1000 to balance Three squares of the tax Class are created. Testing whether the player contains data Testing whether the Squares contain data Testing whether the Squares are indeed from the tax Class Testing tax200 with expected integer 1000 and actual integer equals balance Testing whether expected and actual are indeed even. Testing tax200 with expected integer 1000+200 and actual integer equals balance Testing whether expected and actual are indeed even. Testing tax200 with expected integer 1000+200+200 and actual integer equals balance Testing whether expected and actual are indeed even. Testing tax0 with expected integer 1000 and actual integer equals balance Testing whether expected and actual are indeed even. Testing tax0 with expected integer 1000+0 and actual integer equals balance Testing whether expected and actual are indeed even. Testing tax0 with expected integer 1000+0+0 and actual integer equals balance Testing whether expected and actual are indeed even. Testing tax-200 with expected integer 1000 and actual integer equals balance Testing whether the tax Class allows a negative deposit once and twice Testing whether the tax rate can be set as a division of the whole.

Table 21: TestTax

Expected result	Player name Anders and is created with integer 1000 to balance. Three squares of the tax Class are created. Testing whether the player contains data. Testing whether the Squares contain data. Testing whether the Squares are indeed from the tax Class. Testing tax200 with expected integer 1000 and actual integer equals balance. Testing whether expected and actual are indeed even. Testing tax200 with expected integer 1000+200 and actual integer equals balance. Testing whether expected and actual are indeed even. Testing tax200 with expected integer 1000+200+200 and actual integer equals balance. Testing whether expected and actual are indeed even. Testing tax0 with expected integer 1000 and actual integer equals balance. Testing whether expected and actual are indeed even. Testing tax0 with expected integer 1000+0 and actual integer equals balance. Testing whether expected and actual are indeed even. Testing tax0 with expected integer 1000+0+0 and actual integer equals balance. Testing whether expected and actual are indeed even. Testing tax-200 with expected integer 1000 and actual integer equals balance. Testing whether the tax Class allows a negative deposit once and twice. Testing whether the tax class can be set to a custom amount.
Actual result	the actual results fit the expected results
Status	Passed
Tested by	Mathias Tværmose Gleerup
Date	25/11/2016
Test environment	Eclipse Neon

19.4 How to import code from GitHub

1. Open Eclipse.
   (a) In the top left corner click File.
   (b) Fourth from the bottom, click Import.
   (c) Find the folder called Git.
   (d) Choose the subfile Projects from Git.
   (e) Click Clone URL.
2. Go to your browser and open GitHub.
   (a) Sign in to your account and double click on the wanted project.
   (b) On the right side of the screen, is a green button with the text "Clone or download
   (c) Copy the link in the box.
3. Go back to Eclipse.
   (a) Under Location, paste the URL, in the top box "URL".
   (b) Enter your GitHub account information in the "Authentication" section.
   (c) Check the box "Store in secure store".
      i. This is to make it wasier for yourself, while working in the java project.
      ii. If you choose not to do this, you will have to enter your account information, every time you want to push(upload) an alteration to GitHub.
   (d) Click next two times.
   (e) Ensure the box "Import existing Eclipse projects" is checked, and click next.
   (f) You have now implemented the project. Click finish.

19.5 System requirement

Minimum

1. Java version 8 update 111
2. Command language interpreter (CLI) for running the java file.