CDIO nr. 2

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 4th 2016

November 4, 2016

Danmarks Tekniske Universitet






	Jonas	Mathias	Mikkel	Morten	Simon	Sofie	Sum
Conceive	0,75	0,75	0	0,75	1	0,75	4
Design	1,5	2,5	2	4,5	4	3	17,5
Impl.	5,5	3	0	0	8	5	21,5
Test	5	3,5	0	1,5	0	1,5	11,5
Dok.	3,5	2,75	3	6,5	3,75	10,5	30
Andet.	0	0	0	0	0	0	0
Sum	16,25	12,5	5	13,25	16,75	20,75	84,5

Figure 1: Hour registration

Contents

1 Abstract

A program has been compiled in Java. The program is designed, as a race to reach 3000 points. It's a board game, with 11 fields. Each one of the fields has a feauture, that either subtracts, adds points or does nothing. 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 2), 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 by 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 Noun analysis

We've looked at the requirement text provided by the customer, and after discussing some areas where we found the provided text lacking, we made our own quick recap for analysis:

The game should consist of two players. The game should be playable on the DTU databar's computers. The game consists of a board with twelve squares. Each square has a unique value or event. To reach a field, you need to roll two dice and get a value from (2-12). Each player should start with a 1000 points. When landing on a square, the value from the event on the square should either be added or withdrawn from the point balance. The player that reaches 3000 points first wins. Consequently, if a player reaches 0 points, the player loses.

By doing a noun analysis on the above text, we find that the following nouns may become good examples of classes:

Player
Board
Square
Value
Points
Dice

From the verbs we might get the methods:

Should play (game)
Should start (with) (points)
Withdraw (points)
Add (points)
Roll (dice)

From the list of requirements given to us by our employer, we were told to incorporate an account class. The account class taking care of the points makes sense, so we can cross points off the list of potential classes. We have both a board and a square noun, but since we only need twelve squares and they each only have one event, we think that board can contain the squares. We were also told that it should be easy to replace the dice, therefor making them a class makes sense. The player should also be a class of its own.

From the noun/verb analysis, we end up with the following classes:
Player, Board, Account, Dice.

From the same analysis, we find the following potential methods;

Board

- Square

Account

- startValue
- withdrawValue
- addValue

Dice

-Roll
-showValue

4 Domain model

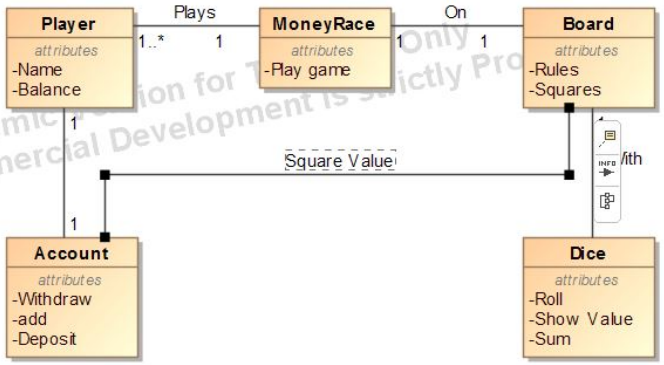

Figure 2: Domain model for Moneyrace

The domain model:

The domain model contains the classes that we identified as useful in the noun analysis, but also one extra, the MoneyRace class which contains 'main'. We would like to have our 'main' in a class for itself, without much more than one line of code, to avoid everything being directly connected to the main. In that way we avoid high coupling, and are in compliance with the principles in GRASP, while creating a game that is easier to reuse.

The player class should contain a name, and the individual players point balance. The player plays MoneyRace, which initializes the board. Our board will contain the rules of the game and of course contain the squares. The dice will be used, and the sum of the dice values will move the playing piece to the correct square. The squares value will be used by the account class to either withdraw or add to the players balance.

5 Requirements

1. The game has to be played by two players.
2. The game has to be executable on the machines in DTU's databars.
3. The players are to take turns to roll the dice.
4. The game has to be played on a board with 11 squares.
5. The game shall use two die.
   (a) The die has to be capable of further developement.
6. Each player has to have an account balance.
   (a) When the game starts each player shall have an initial balance of 1000.
7. Every square has to affect the account balance of the player who lands on it.
8. The game is won when a player reaches an account balance of 3000 or over.
9. The game is lost if a player achieves an account balance of under 0.
10. The following squares are to be used:
    (a) Tower: +250
    (b) Crater: -100
    (c) Palace gates: +100
    (d) Cold Desert: +100
    (e) Walled City: +180
    (f) Monastary: 0
    (g) Black Cave: -70
    (h) Huts in the mountain: +60
    (i) The Werewall: -80
        i. The player gets another turn if he lands on this field.
    (j) The Pit: -50
    (k) Goldmine: +650
11. A text has to be displayed describing the effects of the square a player lands on.
12. The game has to be easily translated into other languages.
13. It must be possible to apply the player and their account balances to other games.
14. The costumer has to be able to view all the system changelogs.
    (a) All changelogs has to be properly documented.
    (b) Only essential changes has to be documented. Non-essential changes does not.
15. The costumer must be able to access and replicate the test process.
16. There has to be a minimum system requirements for the game.
17. There has be a guide on how to compile, install and execute the source code.
    (a) This guide has to include a description of how to import the code from a Git repository.

6 Class Diagram

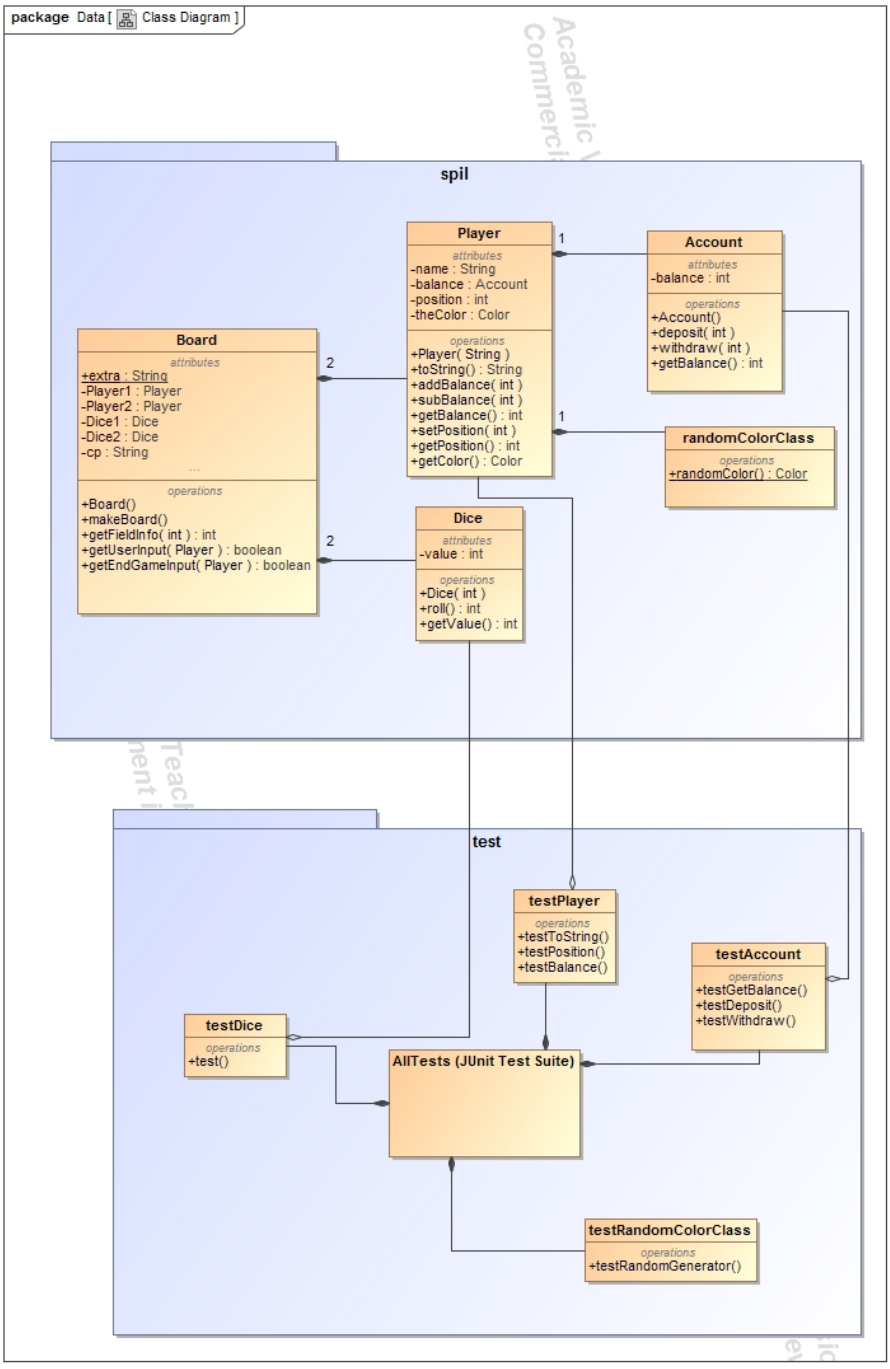

Figure 3: Class Diagram

Comments on class diagram
The class diagram shows the correlation between the classes. Each class has a specific responsibility which, along with using encapsulation. Ensures low coupling and compliance with the principles of GRASP.
Some of these responsibilities has been explained below.

The Dice class has the responsibility of computing a random value between 1 and the desired number of sides on the dice (the constructor receives an integer which is the highest value on the dice and also the number of sides on the dice).
The class Account is only used by Player, so the setters and getters in Player class were made to be in compliance with GRASP (Low Coupling).
The Player class has the responsibility of tracking a players name, balance, position of the player and the color of the players car.
The Board constructor has the responsiblity of running the game. It is instantiated in the 'main' (MoneyRace), which is the only line of code in the 'main' MoneyRace.
The method makeBoard has the responsibility of constructing the fields in the GUI.
getFieldInfo has the responsibility of evaluating a roll of the dices and returning the change in the Players balance to Board.
getUserInput has the responsibility of receiving and evaluating a response from the user. It uses the GUI.getUserButtonPressed method to print a selection of buttons for the user to puch.

7 Use case diagrams

Use case: Money Race
ID: 1
Brief description: The game is called Money Race. Two players are in a race to reach 3000 points. Two die are thrown, which determines the field the player lands on and the effect on their point balance.
Primary actors: 1. Players.
Secondary actors: 1. Board. 2. Points. 3. Dice.
Preconditions: Two players enter the game.
Main flow: 1. The game begins when both players enter their names. 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 field, the player landed on. 6. Include(Change Account Balance) 7. The account prints out what the new balance is. 8. The players position is then reset to the starting position. 9. This is repeated for each player. 10. When one of the players reach 3000 points, that player wins and the game terminates.
Postconditions: The winner is announced.
Alternative flows: 1. If a player reaches negative points, that player loses and the game terminates. 2. If a player lands on the "werewall-field", that player recieves an extra turn.

Table 1: Money Race.
This use case diagram, is an overall description on the basics of how the game works. It's used to show, how the program functions as a whole.

Use case: roll
ID: 2
Brief description: The die are thrown.
Primary actors: 1. Players.
Secondary actors: 1. The die.
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 2: roll.
Use case diagram 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: 1. The board. 2. Players.
Preconditions: 1. The player is positioned on the starting position 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 player is reset to the starting position.
Alternative flows: None.

Table 3: Move on the Board.
Use case diagram 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 field the player lands on.
Primary actors: The board.
Secondary actors: 1. The players. 2. The account.
Preconditions: The account recieves information from the board, about which field the player landed on and what should happen to the point balance.
Main flow: 1. If the field has positive points. (a) The account balance has points added to it. 2. If the field has negative points. (a) The account balance has points subtracted from it. 3. Else (a) The account balance remains the same.
Postconditions: None.
Alternative flows: None.

Table 4: Change Account Balance.
Use case diagram describes why and what happens when points are added or withdrawn from the point balance. The process requires information from the field.

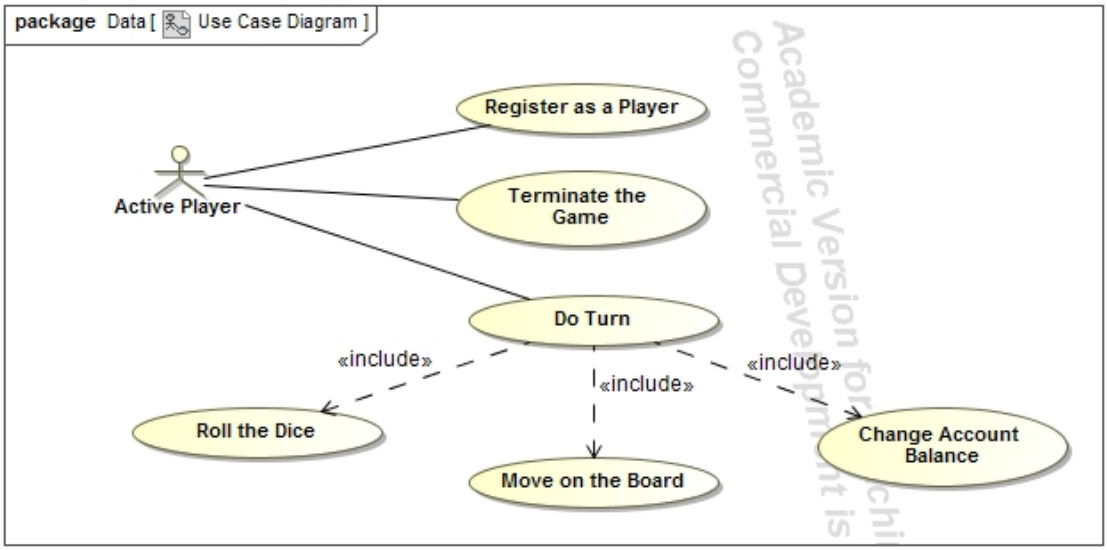

Figure 4: Use Case Diagram.

We have chosen to only do one big use case diagram instead of one for each use case. The reasoning behind this is, that our use cases are still rather small and we feel that one big diagram gives the best graphical overview for the customer. We have also chosen to place the use case model after the use cases, because we feel that a visual display of the games flow will create a better understanding of the game, and in that way help the customer to provide us with better feedback.

8 BCE diagram

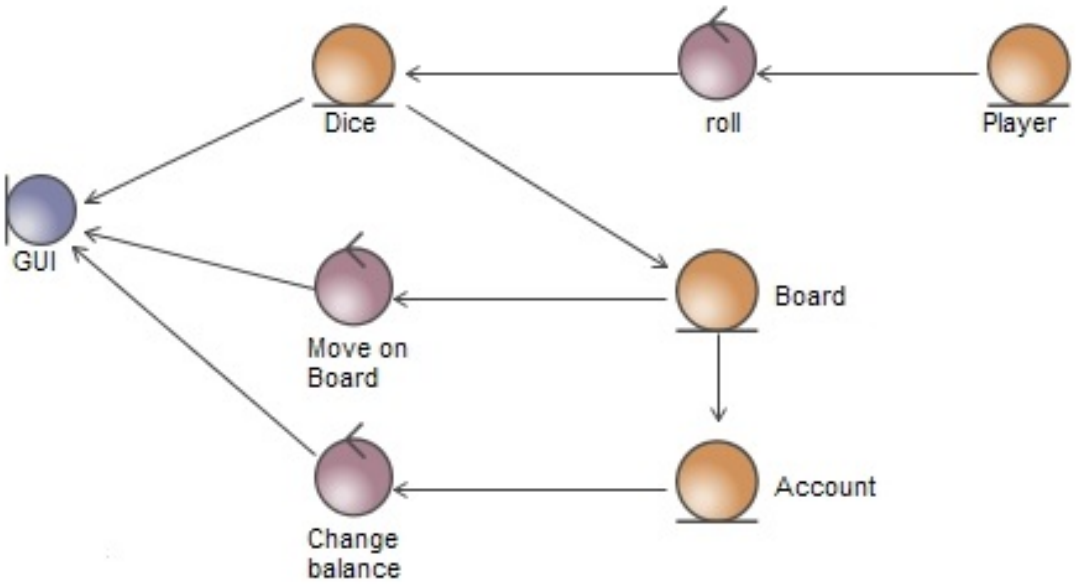

Diagram 5: BCE

Our BCE model is built with classes found in the domain model as entity objects. We have four entities, our player, board, dice and account. In our model, the player starts their turn by rolling the dice. The dice generates values, which the board use to move the playing piece to the square equal to the value of the dice. The account class takes the value from the square on the board, and either adds or withdraws points to the balance of the account.
The three entities dice, board and account talk to the GUI, which serves as our boundary object.

We have chosen to only do one large BCE-model, instead of one for each use case, since the use cases are still rather small and we wouldn't get much help from creating a BCE for each use case.
Creating a BCE for each use case.

9 Sequence diagram

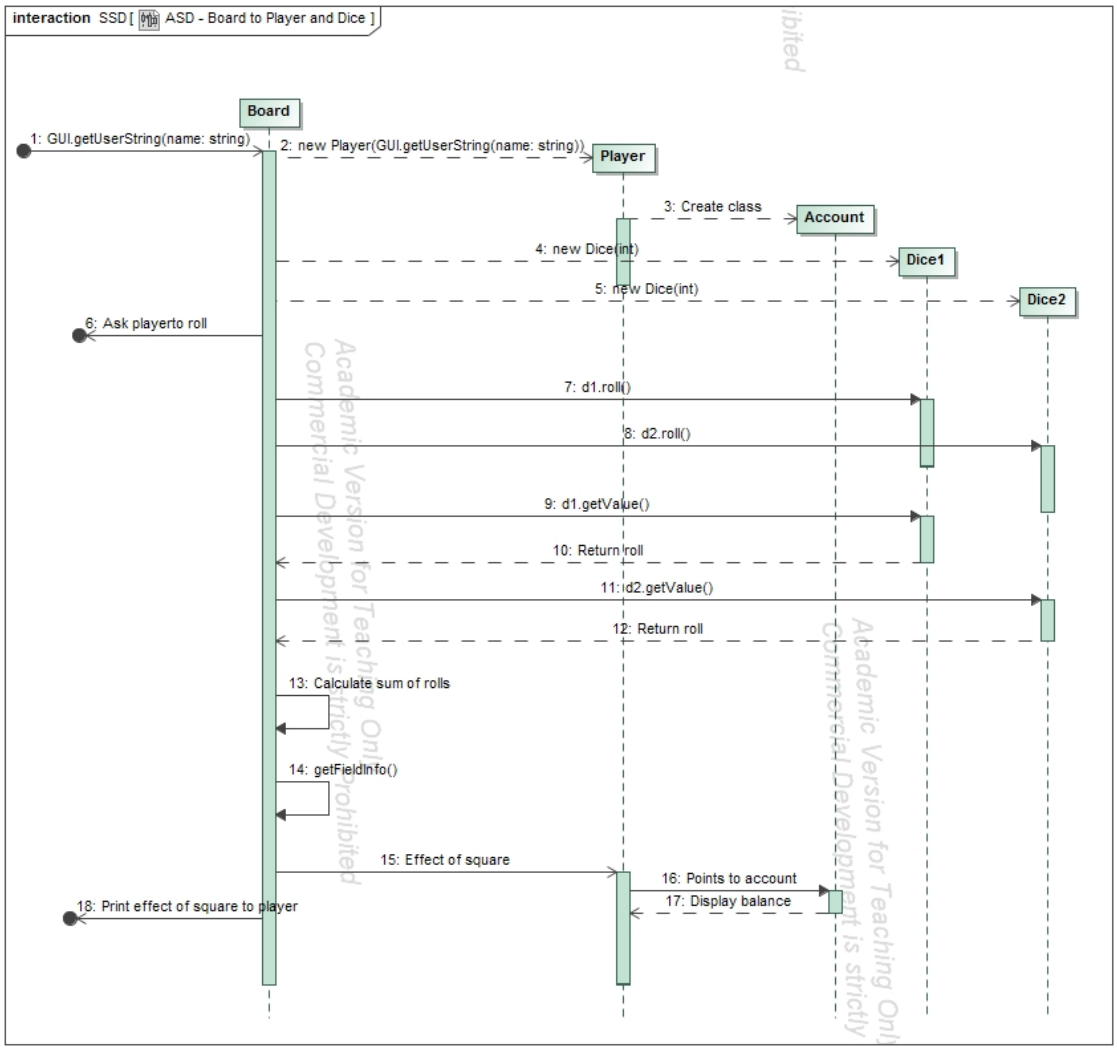

Figure 5: ASD
Brief overview of how we envision a turn played out.

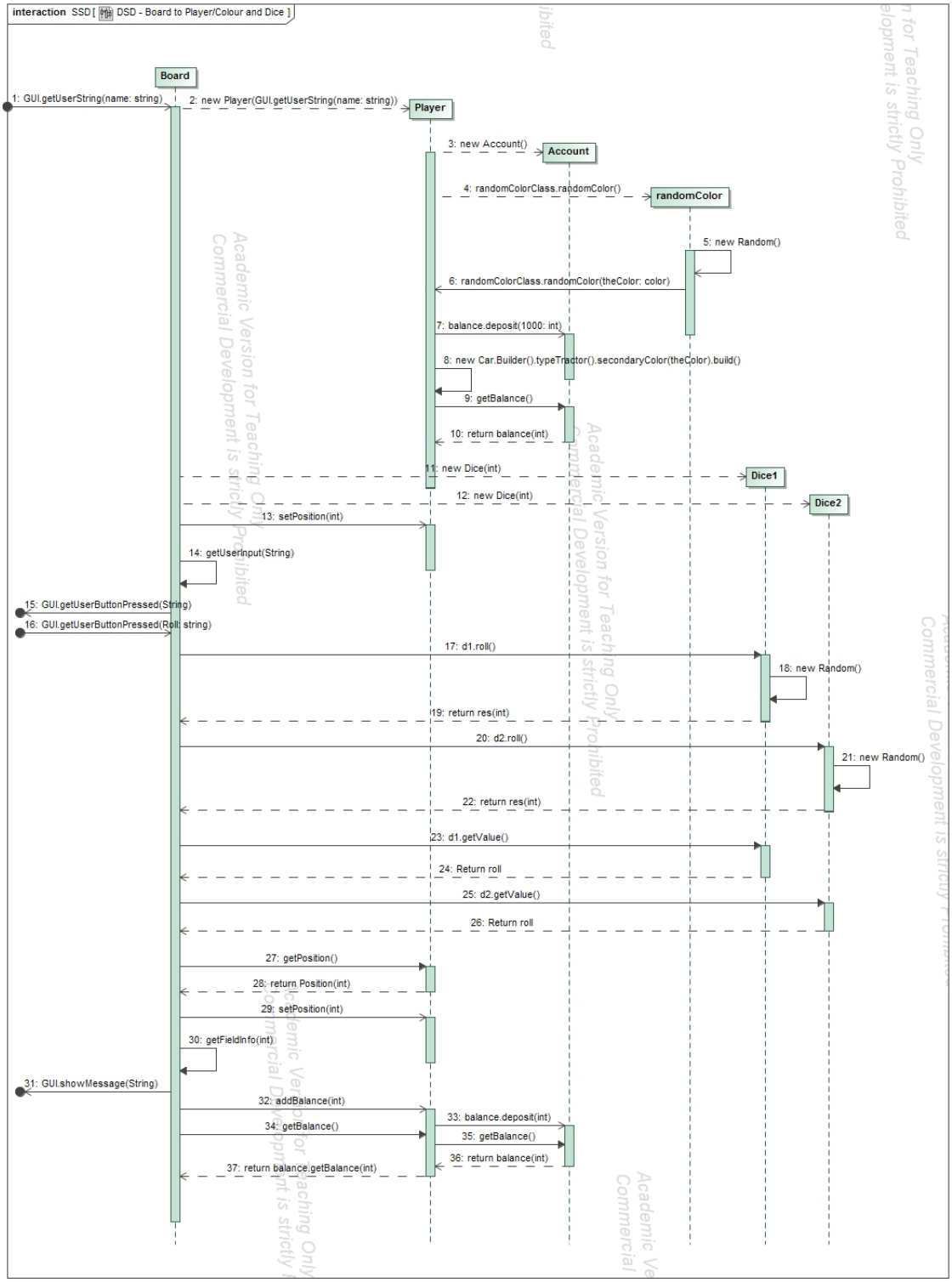

Figure 6: DSD
how a turn ended out running once the code had been written and finalized.

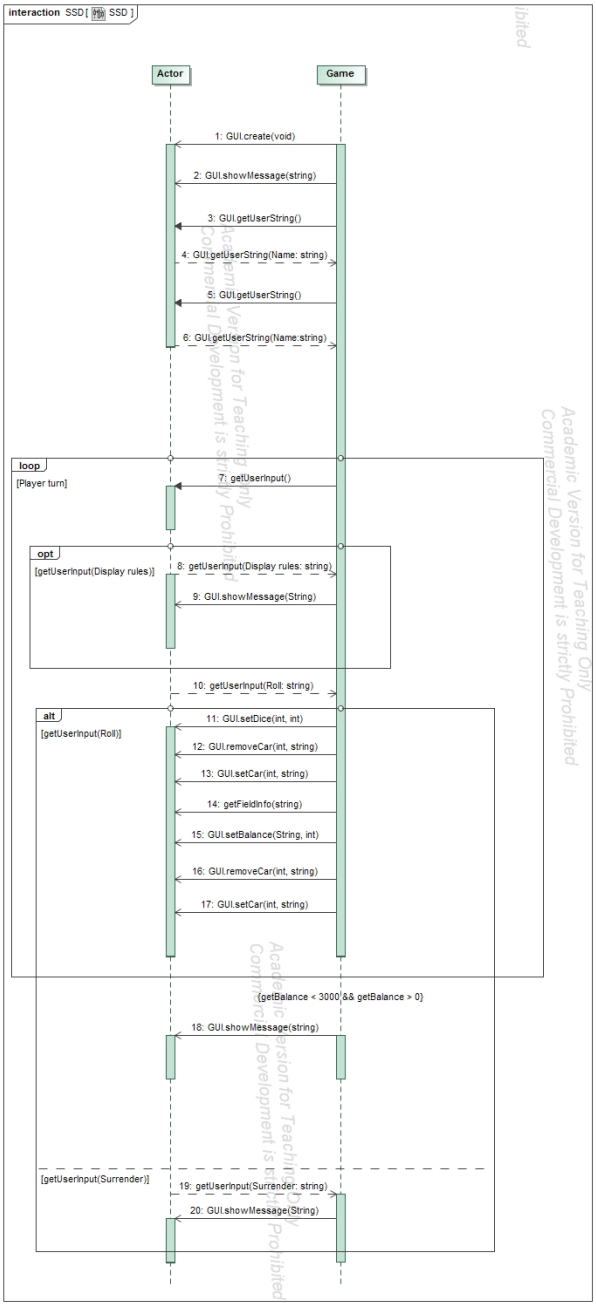

Figure 7: SSD
Shows the complete game being run as one player with all alternative flows.

10 Procedure

1. Requirements
   (a) Started with the costumers description of what they wanted the system to do.
   (b) Listed the requirements as we saw them
   (c) Conferred with a representative from the costumer group.
   (d) Adjusted the requirements.
2. Design and structure.
   (a) Decided that the GUI had to be implemented from the start and to design the classes so that they would output correctly to the GUI.
   (b) With the concept of GRASP in mind, we started forming some conceptual class diagrams.
   (c) We decided to write the code for the game in the Board class, so that the game would not be run in a static environment (main).
   (d) Decision was made to recycle the class Dice from CDIO #1, and include an extra constructor for initializing a dice with N sides.
   (e) Wrote down the conceptual classes in class diagrams.
3. Use cases
   (a) Discussed use cases to establish the boundaries of the system.
   (b) Documented the use cases.
4. Design and structure
   (a) Adjusted the class diagrams according to boundaries and limits found in the use cases.
   (b) Discussed the adjusted class diagrams according to GRASP.
   (c) Created UML diagrams of classes in MagicDraw.
5. Requirements
   (a) Revisited the requirements to the system to ensure that they were coherent with the use cases and classes.
6. Programming
   (a) Created GitHub repository.
   (b) Created the classes Player, Dice, Account.
   (c) Created Board.java and MoneyRace.java(main).
7. GUI
   (a) Based on the GUI design, we decided to give players in the game a Car, to visualize which field the players has landed on.
   (b) We then decided to expand the player class to also have an attribute theColor of type Color. On this basis, the decision was made to make another class, randomColorClass, with a static method randomColor, which would be used to generate a random color for each instance of type Player.
8. Revisited class diagrams to include randomColorClass
9. Created class randomColorClass
10. Began forming the Board class
    (a) Made methods for getting input from the players via the GUI, making the fields in the GUI and a method for ending the game.

(b) Wrote the game in the constructor of Board class.

11. Tests
    (a) Wrote JUnit test cases for testing the classes individually, and made a JUnit test suite to execute tests quickly.

11 Guide

11.1 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.

11.2 System requirement

Minimum

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

12 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 brute force methods, 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 been fruitful and enjoyable. We clearly made progress in accommodating different skill levels and pre-existing knowledge. We experienced during the process, that we did not use the optimal order of development. We bring this knowledge with us, to the next CDIO. Furthermore, we feel that we underestimated the extent of the report. However we experienced, that we felt we did not have the necessary knowledge, to compute a BCE diagram and the difference between ASD and DSD.

13 Test

13.1 Test of the Player class

To verify that the system is fully operational, with few problems, if any, the project has been tested using JUnit. Due to costumers request, the testing has been documented. The testing has been executed after the project was finished, but furthermore during the process of making.

13.1.1 Testing the classes

1. Testing the class Account with the JUnit test called testAccount, using three methods.
   (a) *testDeposit:* This method checks, if the deposit begins at zero.
   (b) *testGetBalance:* Here it's checked, if the right amount is paid to the account.
   (c) *testWithdraw:* This checks if the right amount is subtracted from the account.
2. Testing the Dice class - to ensure the dice is fair. Executed the following way:
   (a) Rolls the die 100000 times, counting each outcome and calculating the mean.
      i. If the mean is between 3.45 and 3.55 the die is fair.
3. Testing the class Player with testPlayer, using three methods.
   (a) testBalance tests if:
      i. When a player object is created, the balance begins at 1000
      ii. *subBalance:* This method should be able to subtract the given amount, from the current amount.
      iii. *addBalance:* Here it should be able to add the given amount, to the current amount.
   (b) testPosition checks if:
      i. The position chenges every time a new position is given.
   (c) testToString checks if:
      i. When calling the method toString, the name of the player must be returned.
4. Testing the RandomColorClass.
   (a) Like the dice the randomColorClass must be distributed uniformly. Therefore, we generate 100000 outcomes and check if every outcome is about the same number. After we had tested our classes we made the class board. The board class is the centre of all the classes, and is running the game. To test if the game and GUI are working correctly, we ran the game multiple times and continuously made corrections until everything was working as intended. We also have to check what happens if the account gets negative. First we created a class preSetDice that extends dice, where it's possible to change the dice outcome to a pre-set value. Then we replace the dice class with the preSetDice class and starts the game. If the games end after a player gets a negative balance, the test is successful.

14 Litteraturliste

References

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

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