Projects fall 2016 – jan 2017

02312-14 Indledende programmering, 02313 Udviklingsmetoder til IT-Systemer and 02315 Versions Styring og test-metoder.

Project name: *CDIOdel2*

Group number.: *14*

Due date: *Fredag, 4/11 2016 at. 23:59*

Conceive Develop Implement Operate project (CDIO) Assignment 2: Dice Game with Fields

Group 14

Rasmus Blichfeldt	s165205
Casper Bodskov	s165211
Lasse Dyrsted	s165240
Michael Klan	s144865
Mathias Larsen	s137055
Timothy Rasmussen	s144146

Danmarks Tekniske Universitet DTU

Supervisor:

Agner Fog

Henrik Tange

Stig Høgh

Time Table & Tasks

Dato	Initials	Requirements	Analysis	Design	Implementation	Test	Documentation	Total
12/10/2016	RB	0.5	3.5					4
12/10/2016	CB	1	2				1	4
12/10/2016	LD	0.25	3.75					4
12/10/2016	MK							0
12/10/2016	ML	0.25	3.75					4
12/10/2016	TR	0.25	3.75					4
14/10/2016	RB			4				4
14/10/2016	CB		4					4
14/10/2016	MK		4					4
14/10/2016	ML		4					4
14/10/2016	TR			4				4
15/10/2016	LD				0.5			0.5
25/10/2016	RB							0
25/10/2016	CB				2			2
25/10/2016	LD				1			1
25/10/2016	MK				1			1
25/10/2016	ML				2			2
25/10/2016	TR						0.5	0.5
26/10/2016	RB				2			2
26/10/2016	CB						2	2
26/10/2016	LD				1		1	2
26/10/2016	MK							0
26/10/2016	ML						2	2
26/10/2016	TR				1	0.5		1.5
28/10/2016	CB						2.5	2.5

28/10/2016	RB				3			3
28/10/2016	TR				3			3
28/10/2016	MK				0.5			0.5
28/10/2016	ML					3		3
28/10/2016	LD			1	2			3
30/10/2016	ML						2	2
31/10/2016	RB				2	1.5	3	6.5
31/10/2016	CB		1				4	5
31/10/2016	ML						6.5	6.5
31/10/2016	LD		1	1	2.5	1	1	6.5
31/10/2016	MK				3			3
31/10/2016	TR						3	3
01/11/2016	RB		1				1.5	2.5
01/11/2016	CB			0.5		1	1	2.5
01/11/2016	LD		0.5				2	2.5
01/11/2016	MK		0.5				2	2.5
01/11/2016	ML		0.5	2				2.5
01/11/2016	TR						0.5	0.5
03/11/2016	TR			0.5			0.5	1
04/11/2016	TR						2	2
04/11/2016	ML						2	2
04/11/2016	LD						2	2
04/11/2016	RB						2	2
04/11/2016	CB						2	2
		2.25	33.25	13	26.5	7	46	128

Name	Hours	Objectives
Rasmus Blichfeldt	24	10
Casper Bodskov	24	13
Lasse Dyrsted	21.5	13
Michael Klan	11	2
Mathias Larsen	28	12
Timothy Rasmussen	19.5	4

Assignment:	Made by	Assisted by
Abstract	Casper	
Introductory	Casper	
Objectives	Casper	
Requirements analysis	Casper	Mathias
System requirements	Casper	Rasmus, Mathias
Word analysis	Casper	
Use case diagram	Timothy	
Use case	Casper	
Domain model	Rasmus	Lasse, Casper
Robustness diagram	Mathias	
Probability calculation for a die	Mathias	
Probability calculation for two dice	Mathias	
Flowchart	Michael	
SSD	Lasse	
Analysis class diagram	Rasmus	Lasse, Casper

Design class diagram	Rasmus	Lasse,Timothy
Design sekvens diagram	Michael	Mathias, Casper
Class responsibility	Mathias	
GRASP	Casper	
toString method	Rasmus	
Test	Mathias	
Import to eclipse	Lasse	
Results	Mathias	
Requirements fulfillments	Mathias	
Conclusion	Casper	
Code GameController	Rasmus	Lasse
Code Player	Lasse	Rasmus
Code Account	Casper	Rasmus, Lasse
Code Die	Lasse	Rasmus
Code Shaker	Lasse	Rasmus
Code Game	Rasmus	Lasse
Implement GUI	Timothy	Rasmus, Lasse
Code PlayerTest	Lasse	
Code AccountTest	Mathias	
Code DieTest	Mathias	
Code ShakerTest	Mathias	
Report Structure	Timothy	

Abstract

This paper documents the process of completing the assignment in which we made a program in java which functions as a board game, in which the player has to accumulate the most gold (balance) to win. The board game includes a shaker with two dice, each capable of getting a face value of 1-6 as well as a board GUI with tiles which all has different effects on the game instance. As shown, by the final tests the program is functional and addresses all the demands and requirements of the assignment. The dice uphold the theoretical probabilities that govern physical dice. In conclusion, the game is working as intended and has no major bugs or other inhibitors prior to possible consumer operation. The final version of the program is written in java and is runnable on DTU’s computers and any pc with the specific requirements are written in the paper.

Table of Content

1. Introduction

The purpose of this assignment is to learn the 4 phases of CDIO as well as utilize the models and knowledge from lectures to fulfill a hypothetical customer’s requested system. This rapport will serve as documentation of our work and how we have used models and tables to visualize our ideas and work schedule. Throughout the work progress we will describe the models to show that we have a certain understanding of the use of said models. Furthermore we will calculate the probabilities of dice throws and the final game mechanics (E.g. how likely are you to get a negative account balance).
The code will also have explanatory text in the report or as comments within the code itself unless we deem the code is self explanatory to our peers.

The following objectives have been outlined for this project as seen in appendix 1.

1. Write a code for a game in Java that simulates 2 dice being thrown and a token moving between tiles.
2. The game needs to be playable by two people.
3. The game needs to be playable without training.
4. The code needs to document the work progress behind the making of the game.
5. The code needs to be easy to expand upon.
6. The code needs to be easy to change, both the language in the game and the dice should be easy to swap to more or less sided die.

2. Analysis

2.1 Requirements specification

1. Users
   - 1.1. There shall be two players
   - 1.2. The players shall be assigned a balance (point-pool)
     - 1.2.1. It shall have a starting value of 1000
     - 1.2.2. It shall not be possible for the balance to have a negative amount of points
   - 1.3. It shall be possible to assign each player a name
2. Shaker/dice cup
   - 2.1. It shall contain 2 dice
     - 2.1.1. The dice shall be six-sided.
     - 2.1.2. The sum of the dice shall decide which of the preset tiles the player “lands” on
     - 2.1.3. Each of the faces of the dice shall have an equal probability of being thrown
3. Tiles
   - 3.1. The player shall be able to see which tile he/she “lands” on
   - 3.2. The tiles shall be named as follows and landing on them shall give the amount listed to the right of the names
     - 3.2.1. Tower +250
     - 3.2.2. Crater -100
     - 3.2.3. Palace gates +100
     - 3.2.4. Cold Desert -20
     - 3.2.5. Walled city +180
     - 3.2.6. Monastery 0
     - 3.2.7. Black cave -70
     - 3.2.8. Huts in the mountain +60
     - 3.2.9. The Werewall (werewolf-wall) -80
       - 3.2.9.1. Landing on “The Werewall” gives the players an extra turn
     - 3.2.10. The pit -50
     - 3.2.11. Goldmine +650

3.3. The player shall receive a text explaining the current tile and the effects of landing thereupon

4. Win condition

4.1. The first player to reach a balance of 3000 or more shall win.

5. Non-functional requirements

5.1. The program shall run on “Windows” operating system

5.2. The program shall run on the computers available in the DATA bars on DTU campus Lyngby

5.3. Every part of the program shall be tested, including...

5.3.1. The dice shall be tested for probability issues and general code

5.4. The program shall be playable by anyone with basic knowledge of computers and the capability to push a button

5.5. (GUI)

5.5.1. The provided GUI shall be used as a starting point

5.6. Everything shall be written in English

5.7. The program shall be written in UTF-8

5.8. The project shall be named 14_del2

5.9. The test classes shall be in another package than the game

5.9.1. One package shall include the game files

5.9.2. The second package shall include the test files

5.10. The game text shall be easily translatable through code

5.11. The code shall make it possible to change the dice

5.12. All work on the code shall be done in a Github repository and “commits” shall be done

5.12.1. Once every hour

5.12.2. Every time a smaller objective or piece of code is written

5.13. The program shall be importable to eclipse

5.14. The GRASP principles shall be followed

5.15. There shall be a declaration which describes the minimal computer specifications required to run the program and what programs which needs to be installed prior to running the program

2.2 System requirements

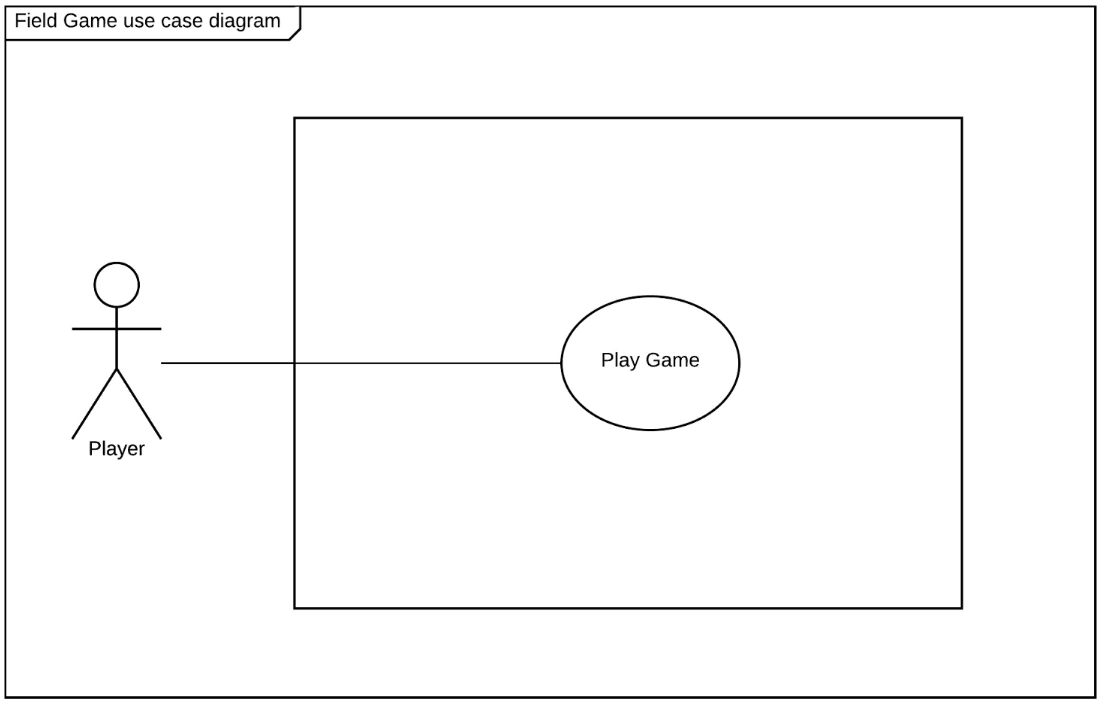
Because of our program’s simple nature, the program only requires that the systems specifications matches up with the specified system requirements for java 8 and Java JRE (Java Runtime Environment) and the supplementing files. Java 8 and Java JRE must be installed.

Version of windows	Windows vista SP1 or newer.
RAM	128 MB
Disk Space	126 MB
Processor	Pentium 2 266 MHz or newer.

2.3 Word analysis resumé

We need a class (Player) which holds attributes and methods for both player objects. This class needs methods for communicating with another class called Account. The Player class then gets the balance information from the Account and it mustn’t go below 0. All transactions between the Player and Account should be written for the user. The code must also end in a return statement for the transactions so that the user will know if it was successfull or not. The game mechanics deducted from the assignment are simple. Save the tiles and assign them a number and how they affect the Account. The account starts with a balance of 1000. Then call them in another class we could call Game Controller which checks whether or not the amount is above 3000 and then tells the Player class which player wins. Which tile the Player lands on is decided by dice throws. The Dice must have a class and there should be a Shaker class as well putting the 2 dice throws together and then ultimately sending that information to the Game Controller.

2.4 Use case diagram

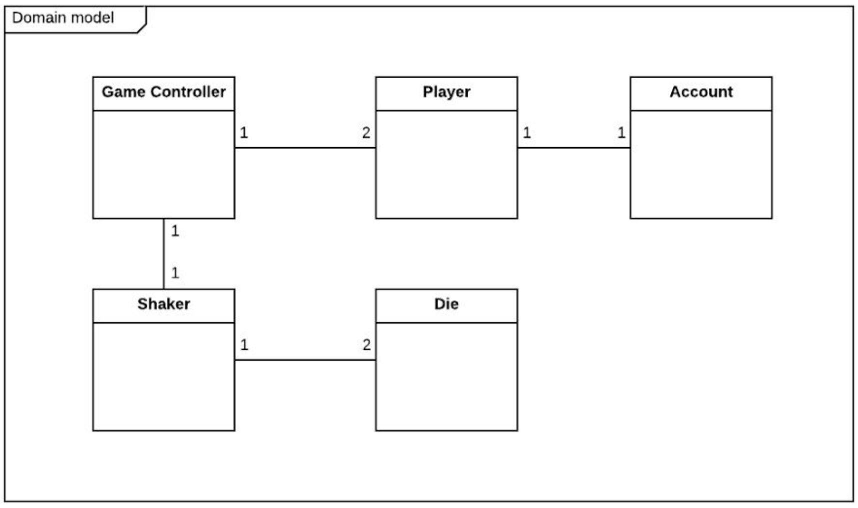


The use case diagram for the project is pretty simple. The reason for this is that the only interaction between the user and the program, is the option to play the game.

2.5 Use case

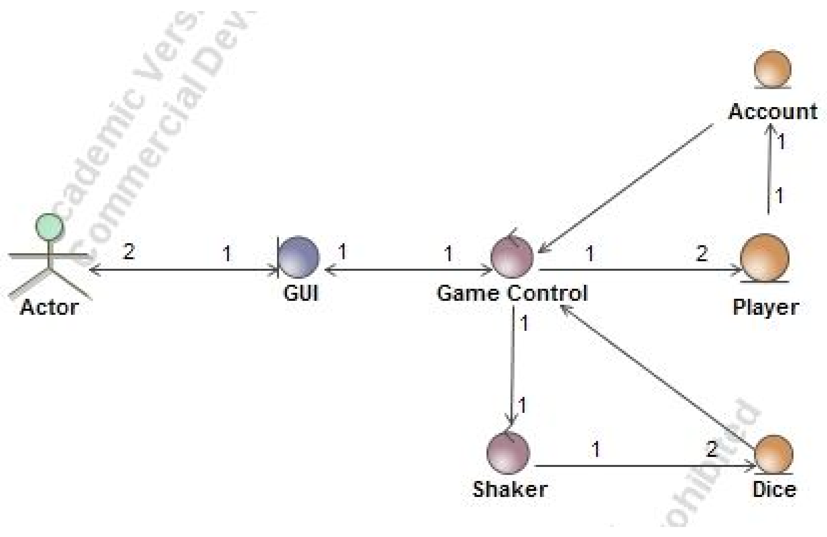
Name	Tile game
Identifier	UC1
Description	Playing the Tile game
Primary actors	Player 1, Player 2
Secondary actors	None
Preconditions	A player runs an instance of Tile game
Main flow	1. System asks for player 1's name. 2. Player 1 enters name. 3. System stores name. 4. Repeat step 1-3 for player 2. 5. A player presses the start button. 6. While there is no winner 6.1. Current player pushes the OK button to roll the dice. 6.2. System rolls dice and the sum decide which tile the player will “land” on. 6.3. System prints the tile and its effects. 6.4. If current player’s balance is 3000 or more 6.4.1. Current player wins. 6.5. Current player pushes the OK button. 6.6. System moves turn to next player. 7. System prints “congratulation” + currentPlayerName + “ you won” and presents a play again button.
Postconditions	The game is terminated.
Alternative flow	1. If a player lands on field 10 (The Werewall) the player gets an extra turn

2.5 Domain model



The domain models serves to give an overview of the system as we see it. We have deducted from the word analysis that these are at least some of the classes we need to make the system operational. This domain model was very bland and without much detail while our later models had sufficient details to make describe the system precisely.

2.6 Robustness diagram



The purpose of the robustness diagram, or BCE (boundary, control, entity), is to analyze the steps of the use case, and thereby validate the business logic. It also tests the robustness of the whole system, by ensuring that the terminology is consistent across all the use cases, and that the usage requirements are fulfilled. The terminology determined in the use cases and robustness diagrams is then used in other UML diagrams.
Actors are the same as in other UML diagrams.
Boundary elements represents interfaces such as GUI's and TUI's.
Control elements connects boundary elements and entity elements. They also implement the logic required to manage elements and their interactions.
Entity elements are often the entities found in the domain model, in OOAD they will be classes. However some classes can be control elements.

In this project there is only one use case therefore the BCE is relatively simple. It is clear how the classes interact with each other it is also aprent that the design pattern The controller is being used.

2.7 Theoretical probability calculation

2.7.1 Theoretical probability calculation for a die

The frequency of a certain value being rolled was calculated by taking the number of possible combinations that could result in that value being rolled divided by the total number of combinations multiplied by 100%.

$$\frac{Tally}{observations} * 100\%$$

Example

The total number of observations (total tally) was 6

So the Frequency of rolling any value were: $\frac{1}{6}$*100%=16,667%

The mean value was calculated by the formula $\overline{x} = \sum_{i=1}^{k} f_i * x_i$

$$\overline{x} = 0,1667 * 1 + 0,1667 * 2 + 0,1667 * 3 + 0,1667 * 4 + 0,1667 * 5 + 0,1667 * 5 = 3,5$$

The variance was calculated by the formula $var(x) = \sum_{i=1}^{k} f_i (x_i - \overline{x})^2$

$$var(x) = 0,167 * (1-3,5)^2 + 0,167 * (2-3,5)^2 + 0,167 * (3-3,5)^2 + 0,167 * (4-3,5)^2 +$$
$$0,167 * (5-3,5)^2 + 0,167 * (6-3,5)^2 +$$
$= 2,922$

To get the spread over 60000 rolls the variance was multiplied by 60000.

$v = 2,922 * 60000 = 175320$

Then the spread was calculated by the formula $\sigma(x) = \sqrt{v}$

$\sigma(x) = \sqrt{175320} = 418,71 \approx 419$

Our calculations was done using statistic. The decision to test over 60000 rolls is due to the nature of statistic as the spread becomes relatively smaller as the number of rollers increase. This in turn increases the precision of the test. The probability of rolling each value was calculated to 16,67% with a spread of 419 over 60000 rolls.

2.7.2 Theoretical probability calculation for 2 dice

	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

The frequency of a certain value being rolled was calculated by the formula. $\frac{Tally}{observations} * 100\%$

Example

The total number of observations (total tally) was 6*6= 36

Combinations that gave the value 4 (tally for the value 4) was 3

So the Frequency of rolling the value of 4 was: $\frac{3}{36}$*100%=8,334%

The mean value was calculated by the formula $\overline{x} = \sum_{i=1}^{k} f_i * x_i$

$\overline{x} = 0,028 * 2 + 0,056 * 3 + 0,083 * 4 + 0,111 * 5 + 0,139 * 6 + 0,167 * 7 + 0,139 * 8 + 0,111 * 9 +$

$0,083 + 0,056 + 0,028 * 12 = 6,91 \approx 7$

The variance was calculated by the formula $var(x) = \sum_{i=1}^{k} f_i(x_i - \overline{x})^2$

$var(x) = 0,028 * (2-7)^2 + 0,056 * (3-7)^2 + 0,083 * (4-7)^2 + 0,111 * (5-7)^2 + 0,139 * (6-7)^2 +$

$0,167 * (7-7)^2 + 0,139 * (8-7)^2 + 0,111 * (9-7)^2 + 0,083 * (10-7)^2 + 0,056 * (11-7)^2 + 0,028 * (12-7)^2$

$=$5,852

Then the spread was calculated by the formula $\sigma(x) = \sqrt{v}$

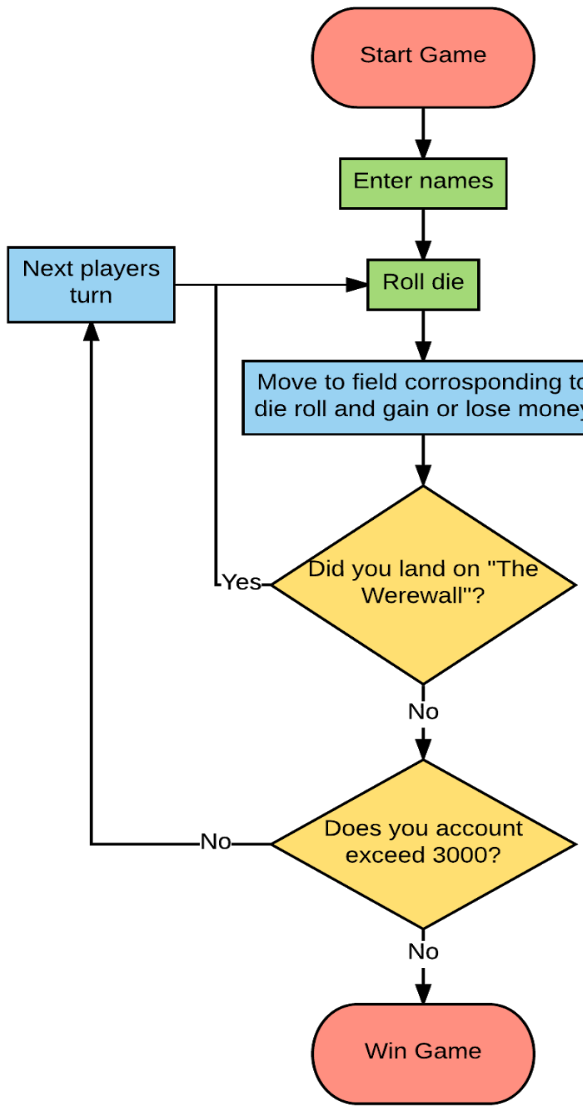
Over 60000 rolls the spread was.

$\sigma(x)\sqrt{5,852 * 60000} = 592,55 \approx 593$

Value	Tally	Frequency	$\bar{x}$	var(x)	$\sigma(x)$ over 60000 rolls	Expected observations
2	1	2,778%				1667
3	2	5,556%				3334
4	3	8,333%				5000
5	4	11,110%				6666
6	5	13,889%				8333
7	6	16,667%				10000
8	5	13,889%				8333
9	4	11,110%				6666
10	3	8,333%				5000
11	2	5,556%				3334
12	1	2,778%				1667
Total	36	100%	$6,91 \approx 7$	$5,852 \approx 6$	593	60000

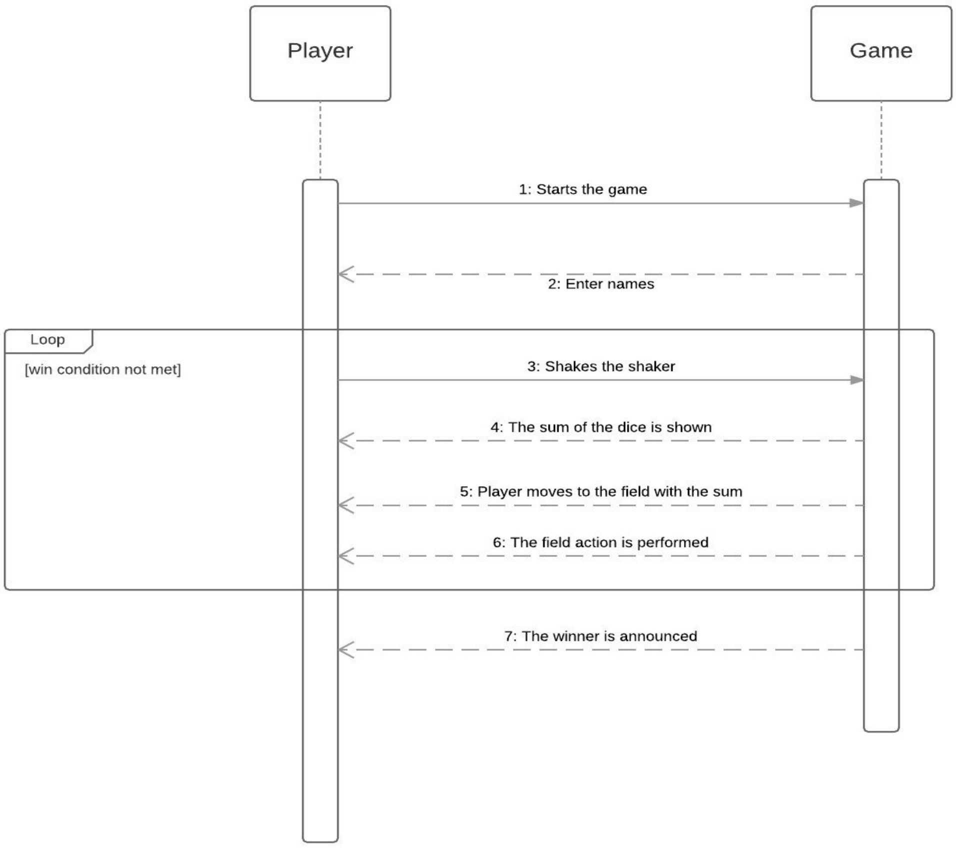
Due the the nature of statistics, it was chosen that the test of the probability of rolling a certain value with two dice would be done over 60000 rolls to achieve a precise result. The probability of rolling each value calculated and presented in the table above in the frequency cullom. From those results it was possible to calculate how many observations was expected, the results were presented in the observations cullom in the table above. Finally the spread was calculated to 593 over 60000 rolls.

2.8 Flowchart


Start Game
Enter names
Next players turn
Roll die
Move to field corrosponding to die roll and gain or lose money
Did you land on "The Werewall"?
Yes
No
Does you account exceed 3000?
No
No
Win Game

The flowchart is a walkthrough of all the possible outcomes the players can be presented with. It presents the game start as a problem, and the solution being ending the game, and the steps and choices being made in order to reach the solution.
Flowcharts are generally used to give an overview of the workflow and how it behaves in every possible situation.
In relation to the program; the terminators represent the beginning and ending of the game. The green processes represent the actions that are executed by the players, the blue processes represent the actions executed by the system, and the decisions represent the handling of the account and
Turns.

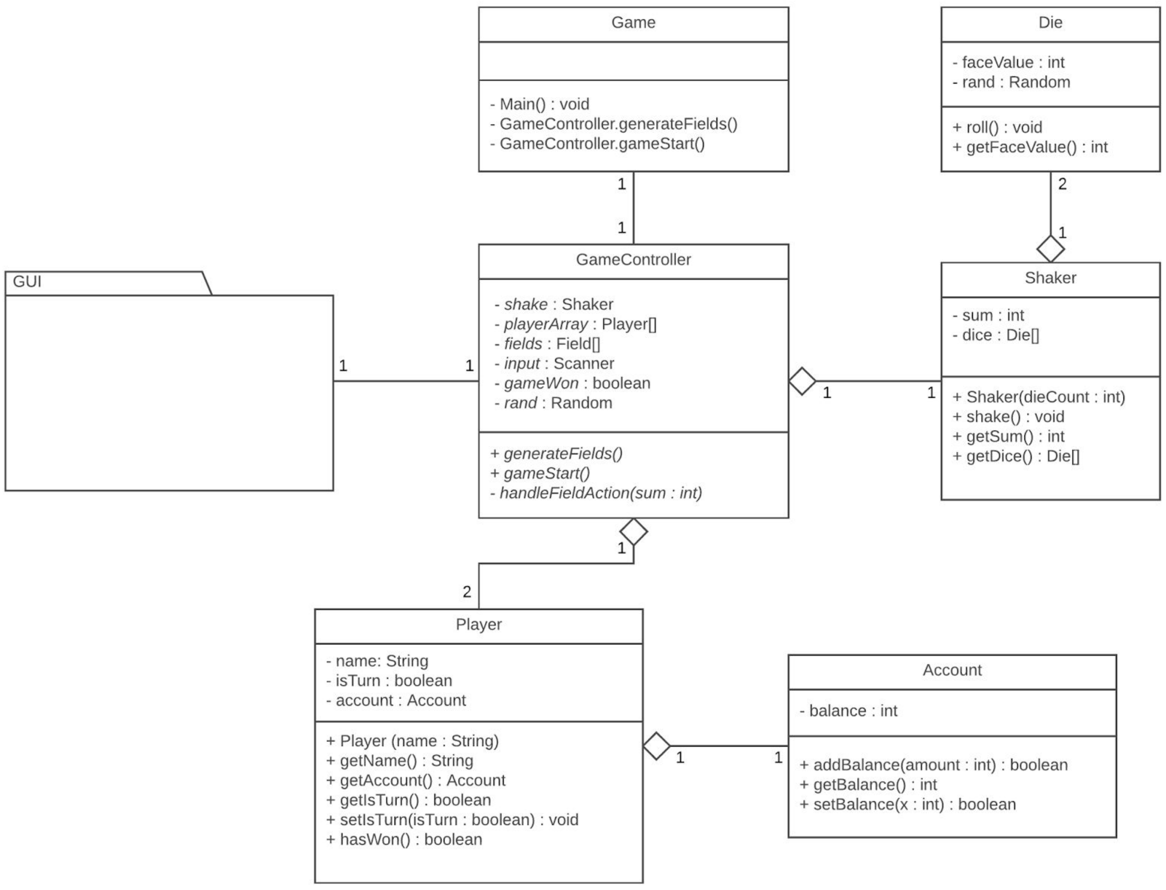
2.9 SSD


Player
Game
1: Starts the game
2: Enter names
Loop
[win condition not met]
3: Shakes the shaker
4: The sum of the dice is shown
5: Player moves to the field with the sum
6: The field action is performed
7: The winner is announced

A SSD's (System Sequence Diagram) job is to give an overview of the relationship between the main actor using the system and the system itself. It simplifies the code itself and doesn't get into technical detail about how the code works. The SSD focuses on the input and output of a given system and leaves the actual functionality as a "black box" where the inner logic and object/class/method relationship is left out.

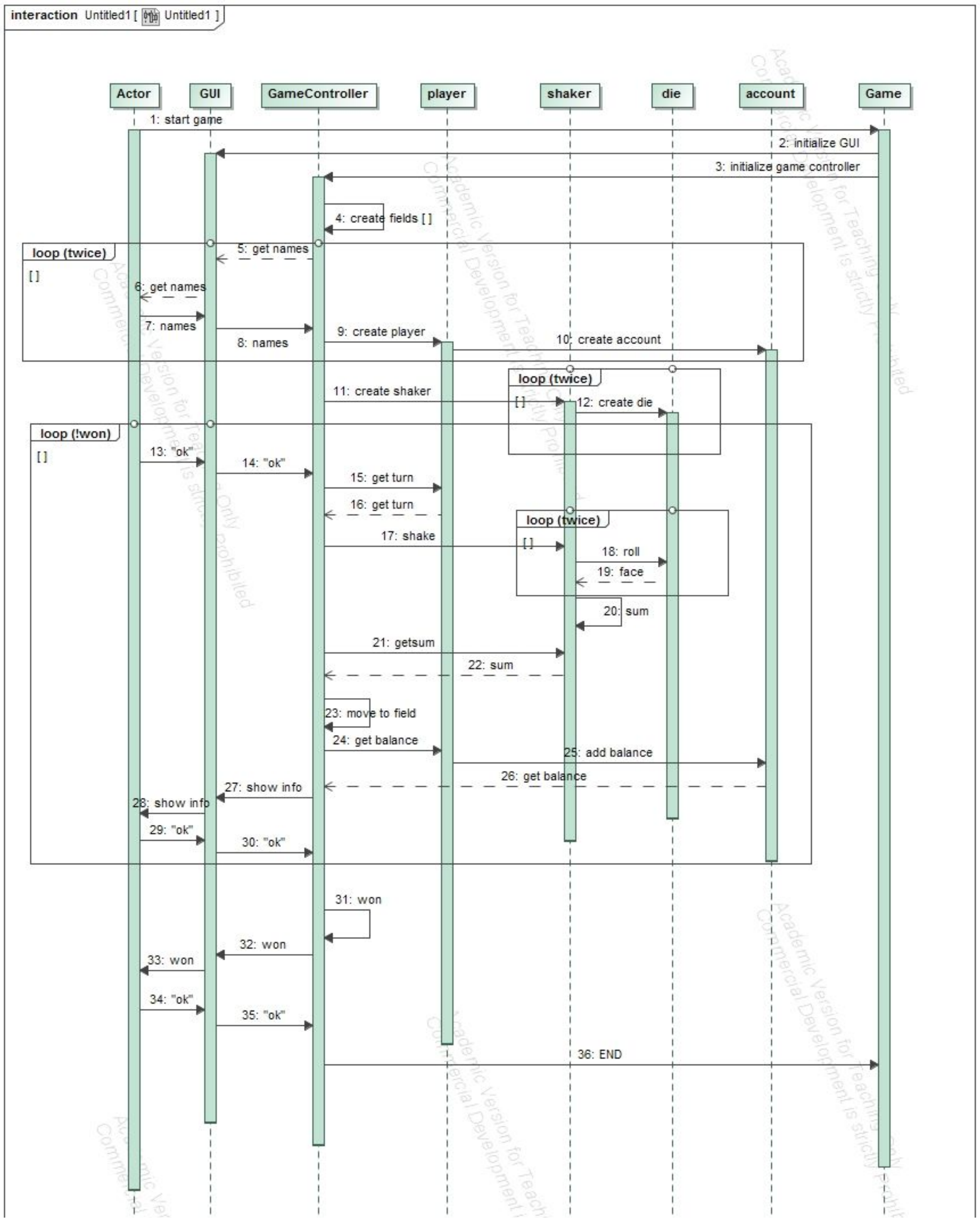
Our main actor in this project is the player. The player actor is, in reality, two different people, but to keep it simple, one actor was used to symbolize the two players since they use the same system. The SSD follows our main use case and shows the basic flow of the program.

2.10 Analysis Class diagram



The analysis class diagram is where we start the development for the code. This is essential for basic code and isn't displaying the final code correctly. This is done later in the design class diagram. The analysis class diagram is more detailed than the domain model but only serves to help show how the classes will interact and what they will contain.

2.11 Design Sequence Diagrams



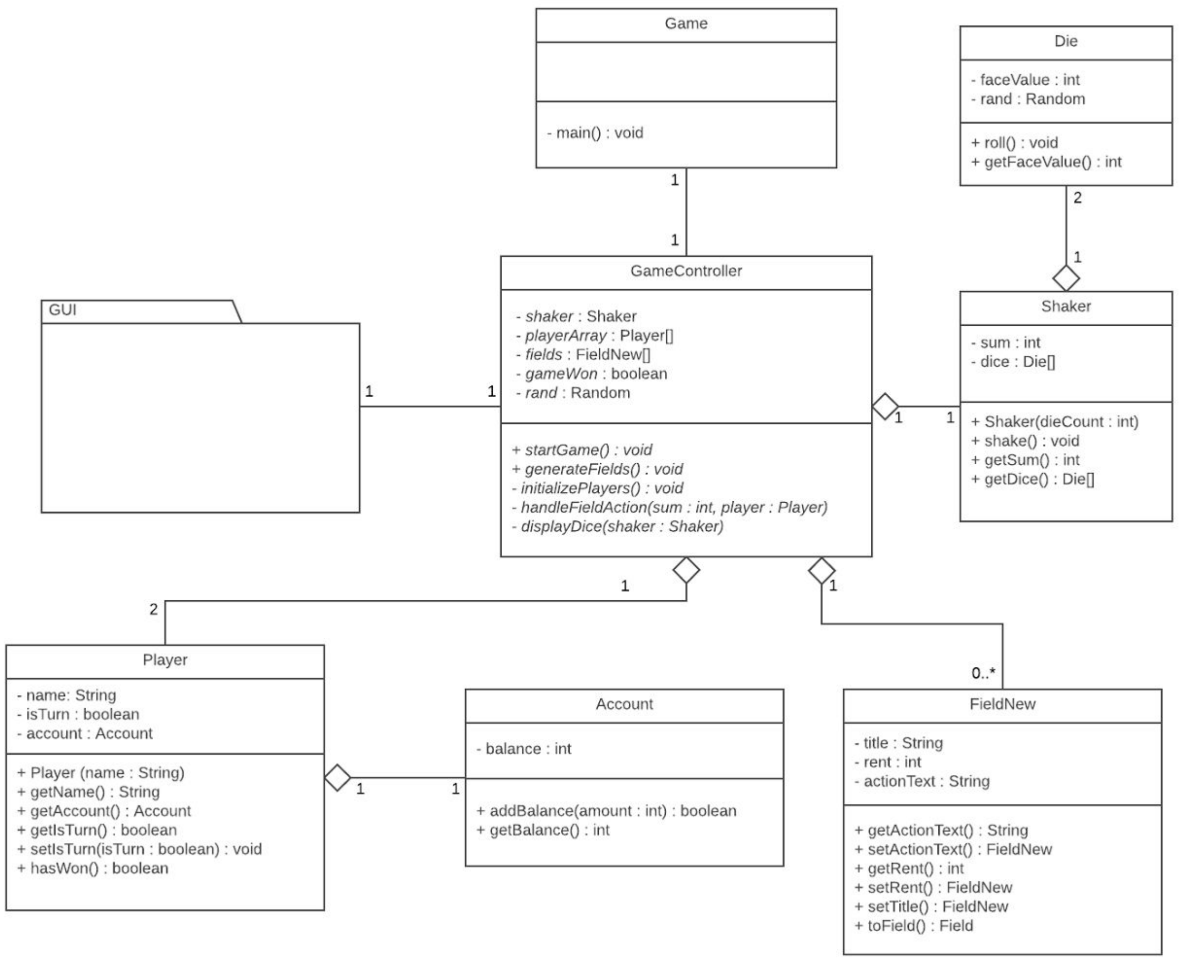
Our design sequence diagram shows how the objects interact with each other in the system. It also shows in what order the actions are executed. Unlike the system sequence diagram the design sequence diagram has more details and serves another purpose. It lies close to a visual use case in the sense that we see all the actions in the program instead of the user input only. We use it to visualize the system operations in a time sequence.

2.12 Later additions

One class isn’t accounted for in the diagrams because we didn’t realize it was necessary before testing the GUI. There was an additional class that was required for the program to function with the GUI which we called FieldNew. But since we didn’t realize till it was very late in the process we decided to ignore the design diagrams meaning the following diagrams will not include this new feature/class since it was only essential because of GUI implementation problems.

3. Design

Design Class diagram



The class diagram builds on the design of the domain model. Its purpose is to directly reflect the written code. By doing this, we get a detailed overview of the program’s structure, and therefore the code. This ensures that we have implemented the parts that should be implemented.

3.1 Class responsibility

3.1.1 Game folder

Class	Responsibility	Connections
Die	Generate random value in the range 1-6.	Shaker.

Class	Responsibility	Connections
Shaker	Create two die. Roll the die. Calculate the sum of the dice.	Die. GameController.

Class	Responsibility	Connections
Account	Keeping track of the players points. Changing the player’s points.	Player.

Class	Responsibility	Connections
Player	Storing the players name. Determining if it is the player's turn. Creating an account object.	Account. GameController.

Class	Responsibility	Connections
GUI	Handling input from the actors.	GameController.

Class	Responsibility	Connections
GameController	Creates the game board. Creates the players. Handles input from GUI. Handles win condition. Controls the flow of the game	GUI. Game. Player. Shaker.

3.1.2 Test folder

Class	Responsibility	Passed
DieTest	Tests if the probability of rolling each value is the same.	Yes

Class	Responsibility	Passed
ShakerTest	Tests if the probability of rolling each value is the same. Tests if the sum is calculated correctly.	Yes

Class	Responsibility	Passed
AccountTest	Tests if a value can be add to the account, including negative value. Tests if the balance can be returned. Tests if the balance can be set to a desired value.	Yes

Class	Responsibility	Passed
PlayerTest	Tests if the player's turn can be set true/false. Tests if the player’s account can be returned. Tests if the player’s win condition can be returned. Tests if the player’s placement on the gameboard can be returned.	Yes

3.2 Documentation for use of GRASP

Grasp is an abbreviation of General responsibility assignment software patterns (or principles). We abide by these principles by dividing the system into specialize classes keeping their cohesion high. By making a controller class we assign the responsibility of dealing with system events to a non-UI class. There's minimal coupling in the sense that each class creates one other class each. Player creates Account and only draws information from Account and gives to the Controller. Only the controller connects all the other classes. Most of the classes doesn't have much responsibility outside their main purpose/methods. This can be seen on the design class diagram as well as the code itself.
The patterns:

1. The controller pattern can be directly correlated to our GameController class. It deals with system events.
2. The creator pattern can be applied to most of the classes since they create objects.
3. High cohesion is a pattern that keeps objects understandable and specified. The relation between Shaker-Die and Player-Account is basically a subsystem which have a clear focused objective.
4. The indirection pattern is used in our program with the model-view-controller pattern. The user sees the View then uses the Controller to manipulate the Model. In our case the user uses the GameController, which manipulates the Game/Shaker/Player classes then shows the results in the GUI.
5. Information expert is a principle where you assign responsibilities to methods, computed fields. We used this to set the roll command in the Die class instead of the Shaker since it is the class with the most information required to complete the method.
6. Low coupling lowes dependency between classes and reduce impact on other classes when changing one class. This was done by having individual people make the different classes meaning the programmers were forced to make their class as closed as possible.
7. The polymorphism principle means using polymorphic operations. Our specific code isn't large enough for polymorphism to be utilized to our benefit more than it would be difficult to try and implement.
8. The protected variations pattern protects elements by using polymorphism to create various implementations of this interface. Our GUI don't change when some variations occur in objects or subsystems behind the logic of the GameController.

9. A pure fabrication is a class that doesn’t do anything specific for the programs functionality but is made to achieve low coupling, high cohesion and reuse potential. We have a class for this but since the program is so little, we also have “toString” methods so further development of the language or flavour text or tiles is easier as well as the Game class which only serves to start the GameController.

3.3 The toString-method

We disregarded the use of the toString method, as we deemed it more time consuming and less intuitive code-wise. We could have returned names and points from the player class, through a toString method. Instead we use accessors (get methods) from the GameController class, where everything GUI related is located. This is to bundle everything GUI related to one place, all to create code that is easier to read.

3.4 Test

1. DiceTest
    - **roll()**
        - This method was tested according to the theoretical probability of rolling each value. A for-loop rolled the dice 60000 times, the value incremented a variable assigned for each possible value (1-6) through a switch statement.
        - Then 6 assertEquals() statements tested if the values were within the theoretical probabilities with a spread of 400.
2. ShakerTest
    - **getSumTest()**
        - A variable was created and given the value of the sum using shake.getDice()[].getFaceValue() method for each die and the values were added.
        - This value was then tested against the getSum() method.

    - **probTest()**
        - This method was tested according to the theoretical probability of rolling each value. A for-loop rolled the dice 60000 times, the value incremented a variable assigned for each possible value (2-12) through a switch statement.
        - Then 11 assertEquals() statements tested if the values were within the theoretical probabilities with a spread of 593.

3. accountTest
    - **addBalanceTest()**
        - The addBalance() method was used to add 100 points.
        - Then the assertEquals() method was used to along with getBalance() method to test if the balance was 1100 points.
    - **getBalanceTest()**
        - Tested if the starting balance was 1000 points, and if a desired amount could be added to the balance.
        - The starting balance was tested by using the getBalance method to store the starting balance to a variable.

        - An assertEquals() method was used to test if the starting balance was 1000 points.
    - **setBalanceTest()**
        - The setBalance() method was used to change the balance to 20 points.
        - Then the assertEuqals() method and the getBalance() method were used to test if the balance was 20 points.

4. PlayerTest
    - **getName()**
        - The assertEquals() and getname() methods were used to test if the name “Player1” was returned.
    - **setIsTurn()**
        - The players turn was set to true using the setTurn() method.
        - Then the assertEquals() and getIsTurn() methods were used to test if the correct value was returned.
        - The steps were repeated for for false value.
    - **getAccountTest()**
        - Tested if the player's account could be returned using the assertNotNull() and getAccount() methods.
    - **hasWon()**
        - Tested if hasWon() method returned false if the balance was under 3000 points.
        - Then the balance was changed to 3000 points and the assertEquales medhod tested if the hasWon() method returned true.

3.4.1 Manual test of GameController

The program follows the use-case and has no functional errors if played as you're supposed to. However the GUI supports "unlimited" characters for the player name which results in some wacky bugs and strange visual breakdowns but it was said in the description of the GUI that the user shouldn't input extremely long names. The code itself doesn't give any errors and the program runs fast and smooth. Some GUI polish is all that's needed for a fully fledged product. Another minor error is when you enter the same value as a name it plays as if there's only one player making the game obsolete. A fix would be to either give an error message when a wrong name or repeated name is entered or save the names as randomised variables instead of their String value.

3.4.2 Import to eclipse

To Import:

1. Extract the folder to a location of preference
2. Open eclipse
3. Goto File > Import > General and choose Projects from Folder or archive
4. Click next
5. Press the Directory button and choose the location you extracted the project folder.

To run:

6. Right-click on the imported project and choose Run as and then choose Java Application.

To test:

7. Right-click on the imported project and choose Run as and then choose JUnit Test

3.4.3 Run the game

1. Doubleclick the jar file in the root of the extracted archive.

3.5 Results

All the test classes passed and the manual test of the whole program followed the main/alternative flow described in the use case.

3.6 Requirements fulfillment

Requirements	Documentation
1.1	Two players could play the game, as documented in the design class diagram artifact.
1.2	Each player had a “point-pool”/account, as documented in the design class diagram artifact.
1.2.1	The players started with 1000 point, as documented in the GameController classes source code line 178.
1.2.2	The accounts could not become negative, as documented in the Account source code line 22-24.
1.3	The players could enter their names, as documented in the SSD artifact and the GameController classes source code line 173.
2.1	The shaker contained two dice as documented, in the design class diagram artifact.
2.1.1	The dies were six-sided, as documented in the Dice class source code 19.
2.1.2	The sum of the two dice were calculated in the Shaker, as documented in the source code for this class line 23-30, and the player landed on the corresponding tile, as documented in the source code of the GameController class line 203.
2.1.3	Each of the faces of the die had an equal probability of being rolled, as documented in the Theoretical probability calculation for a die chapter and in the DieTest class test results.
3.1	The player’s could see what tile they landed on, as documented in the GameController source code line 203.

3.2.1 - 3.2.11 and 3.3	The tiles were named and had the correct properties, as documented in the GameController source code line 33 - 106.
4.1	The win condition was to reach 3000 points, as documented in the flowchart artifact and the Player source code 48.
5.1	The program could run on windows os, due to the fact that it was made in Java.
5.2	The program could run on the computers in the DATA bars, as they use windows os.
5.3	All parts of the program were tested, as documented in the test classes and the test chapter.
5.4	The game was playable with very basic knowledge of computers, as it only requires the player to press one button.
5.5	The provided GUI was used, as documented in the design class diagram and the GUI source code.
5.6	The only place where danish is used is in the appendix, as the original assignment was in danish.
5.7	UTF-8 is the standard character set in IntelliJ and the settings were not changed.
5.8	The project was named 14_del2
5.9.1 - 5.9.2	All test classes were placed in a folder named Test, as documented in the source code folder structure.
5.10	The language of the game can be changed easily as few lines of code needs to be changed.
5.11	The die could be changed by editing one line of code in the Die class line 19.
5.12.1 - 5.12.2	The Github repository was used as, documented on Github https://github.com/ldy985/14_del2

5.13	The program was imported to eclipse, as documented in the test chapter.
5.14	The GRASP principles were used, as documented in the GRASP chapter.
5.15	The system requirements were made, as documented in the system requirements chapter.

4. Conclusion

The assignment was completed to the required specifications as shown by the results and requirements fulfillment chapters. The tests run on and in the program were successful and showed little to no errors while it was under development. We mostly overcame any major hurdles with proper version management, enabling us to effectively backtrack to earlier builds that didn’t include whichever error was displayed. As shown in the report we also abided to the GRASP principles, and applied it to the project structure & Development.

5. Appendix

CDIO del opgave 2

Indledning

IOOuterActive har fået endnu en opgave, og rollerne er de samme som sidst. I denne opgave skal I bl.a. vise, at I kan bruge klasser, relationer og GRASP.

Kundens vision:

Vi blev imponeret over jeres terningespil og vil derfor gerne have at i udvikler et nyt spil til os. Ligesom tidligere skal det være et spil mellem 2 personer, der kan spilles på maskinerne i DTU's databarer, uden bemærkelsesværdige forsinkelser.
Spillerne slår på skift med 2 terninger og lander på et felt med numrene fra 2 - 12. At lande på hvert af disse felter har en positiv eller negativ effekt på spillernes pengebeholdning. (Se den følgende feltoversigt), derudover udskrives en tekst omhandlende det aktuelle felt. Når en spiller lander på Goldmine kan der f.eks. udskrives: "Du har fundet guld i bjergene og sælger det for 650, du er rig!".
Spillerne starter med en beholdning på 1000.
Spillet er slut når en spiller har 3000.
Spillet skal let kunne oversættes til andre sprog.
Det skal være let at skifte til andre terninger.
Vi vil gerne have at I holder jer for øje at vi gerne vil kunne bruge spilleren og hans pengebeholdning i andre spil.

Feltliste

1. Tower	+250
2. Crater	-100
3. Palace gates	+100
4. Cold Desert	-20

5. Walled city	+180	
6. Monastery	0	
7. Black cave	-70	
8. Huts in the mountain	+60	
9. The Werewall (werewolf-wall)	-80	men spilleren får en ekstra tur.
10. The pit	-50	
11. Goldmine	+650	

I kan ligesom sidst benytte GUI'en hvis I har overskud til det. Husk at tilrette feltnavne i GUI'en vha. dennes metoder.

Versionering

Kunden ønsker efterfølgende at kunne arbejde videre på systemet og ønsker at kunne følge med i, hvordan udviklingen er foregået, herunder se de enkelte gruppemedlemmers bidrag.
I afleveringen skal I derfor også levere et GIT repository (mappen .git) hvor kunden kan se hvilke commits, der er foretaget, og af hvem. Alternativt kan afleveres et link til et repository på nettet eks. *https://github.com/cbudtz/TestRepo42.git*.
Der skal committes hver gang en delopgave er løst eller forbedret (dvs mindst én gang i timen når der arbejdes på projektet).
Med hver commit gives en kort beskrivelse, der forklarer, hvad der er blevet lavet. Det skal tilstræbes at committet kun indeholder relevante ændringer (andre ændringer, såsom formatering eller omstrukturering skal ske i et separat commit).
NB:Husk at kontrollere at repositoriet kan importeres i eclipse! Skal projektet kunne importeres i eclipse (uden for mange krumspring), er det vigtigt at .classpath og -project også versionsstyres.

Afprøvning

Kunden ønsker at programmet er overbevisende afprøvet og ønsker at kunne gentage afprøvningen.

I skal have afprøvningskoden med i projektet (vi foreslår et antal JUnit tests).

Projektlederens bemærkninger:

Det er tilladt at lave udskrifter på engelsk, men vær konsekvent.

Overhold GRASP principperne Creator, Controller, Høj binding (High Cohesion), Information Expert, Lav kobling (Low Coupling).

Rapporten kunne have samme struktur som CDIO_del1

Benyt de terninger I allerede har lavet.

Jeg vil foreslå, at I løser opgaven i følgende rækkefølge:

Krav / Analyse

Lav følgende artefakter:

- Kravspecificering
- Use-case diagram
- Use-case beskrivelser
- Domæne-model
- SSD'er.
- BCE

Såfremt nogle af disse undlades, skal der være gode begrundelser for dette.

Design-dokumentation

Lav følgende artefakter:

- Design-klasse-diagram
- DSD'er (Design Sekvens Diagrammer)
- Argumentation for test (evt. en klasse i java, der tester programmet)
- Dokumentation for overholdt GRASP

Kode

- Lav en klasse Spiller der indeholder en spillers attributter og funktioner.
- Lav tilsvarende en klasse Konto der beskriver Spillerens pengebeholdning.

- Overvej hvilke typer attributterne i Spiller, samt i Konto skal have. Lav passende
- konstruktører.
- Lav passende get og set metoder.
- Lav passende toString metoder.
- Tilføj metoder til at indsætte og hæve penge på en Konto.
- Ændr nu Konto-klassen således at der ikke kan komme en negativ balance,
- ligeledes skal metoderne fortælle om transaktionen er blevet gennemført (***Hint***: brug statementet **return** til at returnere denne information).
- Lav det spil kunden har bedt om med de klasser I nu har.
- Hvis I vælger at bruge GUI'en kan I evt. benytte metoderne i Bilag 1.
- Husk at skrive en oversigt over pakkerne og deres klasser - klassernes ansvarsområder og evt. spændende funktioner.

Test

- Lav en test der sandsynliggør at balancen aldrig kan blive negativ, uanset hvad hæv og indsæt metoderne bliver kaldt med.

Konfiguration

Kunden ønsker at vide hvilke krav, der er til styresystemet og installerede programmer.
I skal derfor udfærdige en beskrivelse af minimumskrav og en vejledning i hvordan kildekoden compiles, installeres og afvikles. Dette inkluderer en beskrivelse af hvorledes koden importeres fra et git repository.

Bilag 1 Metoder i GUI'en (teksten er fra javadoc, derfor er den på engelsk)

- **public static void setTitleText(int fieldNumber, String title)**

  Sets the title of a field on the board.

  fieldNumber : int [1:40]

  title : String (Mind the length!)

- **public static void setSubText(int fieldNumber, String subText)**

  Sets the subText of a field on the board.

  fieldNumber : int [1:40]

  subText : String (Mind the length!)

- **public static void setDescriptionText(int fieldNumber, String description)**

  Sets the Description (The text shown in the center when mouse hovers) of a field on the board.

  fieldNumber : int [1:40]

  description : String (Mind the length!)

- **public static void addPlayer(String name, int balance, int r, int g, int b)**

  Adds a player to the board.

  Max. 6 players.

  name : String (Mind the length!) (Unique identifier of the player - no duplicates)

  balance : int

  r The RED part of the player color

  g The GREEN part of the player color

  b The BLUE part of the player color

- **public static void addPlayer(String name, int balance, Color color)**

  Adds a player to the board.

  Max. 6 players.

name : String (Mind the length!) (Unique identifier of the player - no duplicates)

balance : int

color : Color

- **public static void setBalance(String name, int newBalance)**

  Sets the balance of a player if the player has been added.

  name The name of the player

  newBalance : int

- **public static void setCar(int fieldNumber, String name)**

  Places a car on the field.

  All cars can be placed on the same field.

  A car can only be placed if the corresponding player has been added.

  If a car is placed on the same field multiple times, nothing more happens.

  A car can not be placed on multiple fields simultaneously.

  fieldNumber : int [1:40]

  name The name of the player

- **public static void removeCar(int fieldNumber, String name)**

  Removes a car from the board.

  If the car is not on the board, nothing happens.

  fieldNumber : int [1:40]

  name The name of the player

- **public static void removeAllCars(String name)**

  Removes all cars belonging to this player.

  name The name of the player.

Appendix 2 Word analysis (Danish)

Ord analyse

Vi har givet to stykker tekst. Det ene afsnit (Kode) beskriver nødvendige stykker kode, som ikke kræver fortolkning. Hertil vil vi bruge afsnittet Kundens vision til at finde andre metoder eller uddybende kode.

Kode

- Lav en klasse Spiller der indeholder en spillers attributter og funktioner.
- Lav tilsvarende en klasse Konto der beskriver Spillerens pengebeholdning.
- Overvej hvilke typer attributterne i Spiller, samt i Konto skal have. Lav passende
- konstruktører.
- Lav passende get og set metoder.
- Lav passende toString metoder.
- Tilføj metoder til at indsætte og hæve penge på en Konto.
- Ændr nu Konto-klassen således at der ikke kan komme en negativ balance,
- ligeledes skal metoderne fortælle om transaktionen er blevet gennemført (***Hint***: brug statementet **return** til at returnere denne information).
- Lav det spil kunden har bedt om med de klasser I nu har.
- Hvis I vælger at bruge GUI'en kan I evt. benytte metoderne i Bilag 1.
- Husk at skrive en oversigt over pakkerne og deres klasser - klassernes ansvarsområder og evt. spændende funktioner.

Analyse of "Kode"

Vi skal altså have en klasse Spiller og en tilsvarende klasse Konto, der holder styr på Spillerens pengebeholdning. Vi skal bruge get, set og toString metoder, samt metoder der gør det muligt at hæve eller indsætte penge på en Konto. Derefter skal det ikke være muligt at have en negativ saldo på Kontoen. Der skal også laves en return statement som informere spilleren om transaktionen er blevet gennemført.

Kundens vision:

Vi blev imponeret over jeres terningespil og vil derfor gerne have at i udvikler et nyt spil til os. Ligesom tidligere skal det være et spil mellem 2 personer, der kan spilles på maskinerne i DTU's databarer, uden bemærkelsesværdige forsinkelser.
Spillerne slår på skift med 2 terninger og lander på et felt med numrene fra 2 - 12. At lande på hvert af disse felter har en positiv eller negativ effekt på spillernes pengebeholdning. (Se den følgende feltoversigt), derudover udskrives en tekst omhandlende det aktuelle felt. Når en spiller lander på Goldmine kan der f.eks. udskrives: "Du har fundet guld i bjergene og sælger det for 650, du er rig!".
Spillerne starter med en beholdning på 1000.
Spillet er slut når en spiller har 3000.
Spillet skal let kunne oversættes til andre sprog.
Det skal være let at skifte til andre terninger.
Vi vil gerne have at I holder jer for øje at vi gerne vil kunne bruge spilleren og hans pengebeholdning i andre spil.

Analyse af "Kundens vision"

Vi skal have en "set tile" metode til at opskrive felterne 2-12. Hertil skal vi have en class til et bæger (Shaker) og nogle terninger (Dice). Terningerne skal ku slås og vise resultaterne i shakeren, som kan sende informationer til en Game Controller eller main class. Den positive eller negative effekt skal så gemmes i et array eller måske objekter, som så kan hentes igen, når man slår tallet til et bestemt felt. Teksten til det felt bliver kaldt og skal udskrives.
"Spillerne starter med en beholdning på 1000". Så spillet starter ved at deres Account/Konto indeholder 1000 penge og slutter først når det tal når 3000. Det kunne gøres ved et while statement, som holder programmet kørende, så længe beholdningen er under 3000.