# Rapport CDIO1

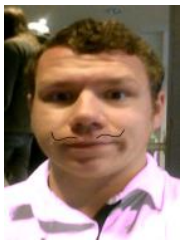

Mikkel Leth s153073

Sebastian Hjorth s153255

Senad Begovic s153349

Thomas Madsen s154174

Rasmus Olsen s153953



Nicki Rasmussen s15344


Due date: 17/10/2015

Institute: Danmarks Tekniske Universitet

Assignment type: CDIO Projekt

Course: 02312 + 02313 + 02315

Supervision: Ronnie Dalsgaard, Stig Høgh, Jacob Nordfalk og Henrik Tange

# Timesheet:

| **Timesheet** | **Ver. 2015-17-09** | | | | | |
|---|---|---|---|---|---|---|
| **Participant** | **Design** | **Implementation** | **Test** | **Document** | **Other** | **Total** |
| Mikkel | 4 | 7 | 1 | 6 | | 18 |
| Nicki | 3 | 6 | 1 | 2 | | 12 |
| Rasmus | 4 | 5 | 1 | 4 | | 14 |
| Sebastian | 4 | 5 | 2 | 5.5 | | 16.5 |
| Senad | 4 | 6 | 1 | 4 | | 15 |
| Thomas | 4.5 | 6 | 1 | 3.5 | | 15 |

# Table of Contests

# Introduction

## Corse goals

We are a group of students at the Technical University of Denmark, who have been given the task to make a dice-game for the game-company IOOuterActive, using our knowledge gained from the courses Introductory Programming and Developing methods for IT-systems. With this in our minds, we are going to implement the four terms (CDIO) to this assignment in the best way possible to satisfy our costumer. We will use our skills to analyze, design, implement and test our system, so it will fit the demands and requirements set down by the customer, which will be specified in the section below ('Requirements and demands').

## Requirements and demands

The requirements to the system is first of all, that it can be used on the computers around the University campus, which all are running Windows. The program has to be a dice-game between two people, who alternately rolls two dice and are shown the result right away. Furthermore, the sum of the dice are to be added to the current player's points. The winner of the game are the player who are first to reach 40 points.

To simplify the requirements, we have separated the requirements into functional and nonfunctional requirements in the table below:

| **Functional requirements** | **Non-functional requirements** |
|---|---|
| Has to be a game that consist of two people | Can be used on the campus machines |
| You roll with two dice | The dice result has to be shown within ½ second |
| The result of the dice are shown immediately | |
| The sum of the dice are added to the current player's point | |
| The winner is the player to first reach 40 points | |

The customer has as well given us some bonus demands in addition to the primary requirements, which we decided to implement as well. These demands were as follows: to give an extra roll in case the roll is a pair, to reset the current player's points in case of a pair of ones, and if a player rolls two six' twice in a row it will result in an automatic win to the current player. Also just getting to 40 points is not enough anymore, the player will have to roll a pair once the minimum of 40 points have been achieved.

Due to feelings of flaws in the demands, for example that some demands were overlapping each other, we decided to ask the customer the following questions:

Q. Will a pair of ones, in addition to resetting the points, also give the player an extra turn?
A. Yes.

Q. How many extra rolls can a player get by rolling pairs?
A. An infinite amount, although each player should at least be allowed to have one turn before a player can win.

Q. Should the players be able to choose a name for themselves?
A. Each player should be able to decide what they want to be called,

Q. Do the players have to roll to see who has the first turn?
A. Player 1 will always have the first turn.

We have been given a GUI (Graphical User Interface), from another project to help us. We were able to use this GUI to show the dice and possibly other functions, which we have been able to get working. The last demand was that the program had to be simple to use, thus a manual is not necessary, and the program should contain a test with 1000 rolls with a die, to show that the distribution of decays is more or less equally distributed.

Thereby, this is our final list of requirements and demands organized in functional and non-functional demands in the table below:

Requirement table

| **Functional requirements** | **Non-functional requirements** |
|---|---|
| Has to be a game that consist of two people | Can be used on the campus machines |
| You roll with two dice | The dice result has to be shown within ½ second |
| The result of the dice are shown immediately | A 1000 roll test |
| The sum of the dice are added to the current player's point | User friendly |
| Lose all points if a player rolls a pair of ones | |
| Extra turn to the player who rolls a pair of ones | |
| A pair of six' twice in a row results in a win | |
| A par is needed to win if points are at least 40 | |
| Each player should have at least one turn | |
| Player 1 always starts | |

# Theory Section

## Why Java?

We chose to use Java because we had to write a relatively simple program where memory and processing power wasn't a problem.

Java's built-in garbage collection, great standard library and wide support for OO would speed up the development process.

## Terminology

**CDIO**

CDIO is an abbreviation of the 4 design principles: Conceive — Design — Implement — Operate.

**Constraint**

Constraints describe the limits of the final product.

**Requirements**[1]

Requirements play a big role in the Design and Conceive phase. Requirements describe the specific needs for the program and what it should be able to do. A common way of dealing with the requirements is dividing the specific needs into functional and nonfunctional requirements and as use cases and actors. We can further elaborate on the requirements by dividing them even further with "MoSCoW": *Must have, Should have, could have and Want to have*. These four categories may have attributes, which means they can contain information associated with each requirement.

**Multiplicity**

Multiplicity is a limitation to the number of Classes a specific object can work in coherence with at any given time. Multiplicity is a great way to document how each of your classes depend on each other and can help reduce coupling related errors.

**Association**[2]

A link between two classes is called an "Association". For two classes to work together there must be an association between the objects in question.

**Aggregation**[3]

Aggregation is a kind of relationship, and is commonly referred to as a whole/part relationship. It consists of one object (called the "whole" or the "aggregate") and another object (called the part). The "whole" uses the "part" to do different operations.

The part is independent of the whole, and can live by itself, without a whole.
But, the whole may not work optimally without a part.

[1] Jim Arlow & Ila Neustadt, UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design: Chapter 3.
[2] Jim Arlow & Ila Neustadt, UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design: Chapter 9.
[3] Jim Arlow & Ila Neustadt, UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design: Chapter 9.

**Composition**[4]

Composition is kinda like aggregation, but the bond between the whole and the part is stronger. In composition the part is dependent of the whole, it cannot be without it. This is the key difference between aggregation and composition. Besides a part can only relate to 0 or 1 whole, in aggregation, the part can relate to more.

**Comparison**

A comparison is when two variables or constants are compared. E.g we roll two dice and then we need to be able to see if they rolled the same result on both die.

**Visibility**

The visibility of a certain part of a program is determined whether it is private or public. As a rule of thumb: Variables are private and methods are public. This way we keep a good order and methods from a certain class can't change a variable in a different class.

**Classes**[5]

A class is a part of the program that deals with its immediate area of responsibility. So that when we need to do a specific thing, then we can call the class responsible for it. This way we divide the program up into smaller chunks, which can be very useful when dealing with a large program. If the program can consist of a single class if you so desire, but it will be difficult in the longer run.

**Domain model**

Showcases the specific behavior and data that the program uses and its outputs.

**Use Case**[6]

Use cases are used to picture a certain situation a user of the program will be situated in, thus we can prepare the program for what it might encounter from the user. Use cases can be broken down into system sequence diagrams which also can be broken down to system operations.

**Use case specification**

A more precise way of describing a user interaction with the program.

[4] Jim Arlow & Ila Neustadt, UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design: Chapter 9.
[5] Jim Arlow & Ila Neustadt, UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design: Chapter 7.
[6] Jim Arlow & Ila Neustadt, UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design: Chapter 4 & 5.

**Class diagram**[7]
Showcases the variables and methods of a class in specifics, including type, return type and the name of the method or variable itself. Lastly it shows the visibility.

**BCE-model**
Shows the relation between the user and the program, furthermore it showcases how the classes themselves interact with each other. The classes are divided into three categories: Boundary, entity and control. Boundary communicates interaction between the actor and the system. Control classes confer responsibilities to other classes. Entities has memory and reflects what the system is about.

**Noun- and verb analysis**
Noun- and verb analysis is a way of finding classes, attributes, responsibility and methods. When you work with this way of analysis, you read through the requirements. The nouns and noun phrases indicate a class or the class attributes. Verbs and verb phrases indicates responsibilities or methods.
By doing this analysis you can get a good overview of the different classes you have to work with, alone by looking at the requirements.

**System sequence diagram**
The system sequence diagram (SSD) gives a good time course of how the code is runned, and is the closest we come to the final code in the analysis part of the work.

The SSD is a dynamic model which shows the input and output events in relation to the system. Basicly the SSD shows a visual demonstration of a use case flow (a use case scenario).

The SSD have two purposes:

- Describes what the system must be able to do
- Divides the system into system operations, so we know what each part should be capable of

**Design sequence diagram**
The design sequence diagram is very similar to the system sequence diagram, except that the system sequence diagram describes the communication and cooperation in the system, while the design sequence diagram is more about the code's course, how it runs etc.

[7] Jim Arlow & Ila Neustadt, UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design: Chapter 7.

## Summary

In this project we were given a set of requirements from a customer who had a vision for a dice game, which were to run on the computers on DTU. From these requirements we had to work our way through analysis, implementation, design and test. In the analysis part we used different work methods such as the noun- and verb analysis to identify the unspecified requirements from the vision, the domain model to obtain the different relationships between the classes and the system sequence diagram to design the flow of the program.

After this part we moved on to the implementation phase. This is the code writing part of the work where we implemented all the classes and their mutually relationship we got from the analysis phase. After the code was written, we had to make a model for the code. This helps get a clear view of the code, this was done with a design sequence model.
At last we wrote some unit tests to check possible errors in the program. This is done to detect any errors that might be introduced, should the code be changed later on.


# Main section

## Analysis

The program consists of four classes: Board, Player, Dice og DiceResult. Player and DiceResult both functions as data containers. We found those classes by analysing the requirements. First we made a noun- and verb analysis[8] to decide which components we had to use. We proceeded onto turning our noun- and verb analysis into use case models in which we discovered the actors and subjects. We came up with two use-cases: “New Game”[9] and “Roll The Dice”[10]. New Game is the part where the actor can name new players and should always be the first use case to occur .Roll The Dice is the use case for when the actor decides to roll the dice. Now we had our use cases and together with the noun- and verb analysis we could move on to work with performing a design analysis on the classes.

First, we discovered that the user was the only actor on the system. Secondly, we discovered that we needed some way of storing the player data(name and points) and the dice data across the use cases. Which concludes the analysis.

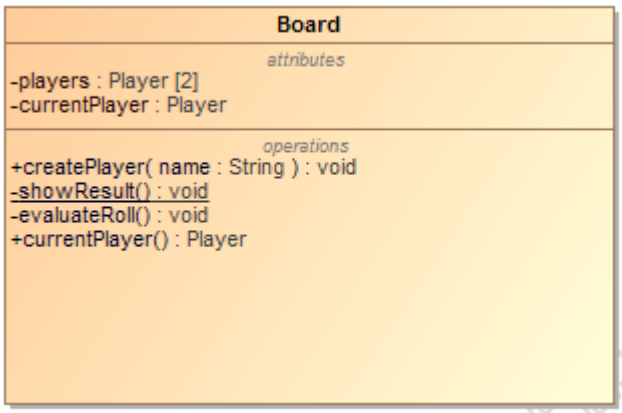
## Design

Here we have our first three classes, “Player”[11], “Dice”[12] and “DiceResult”[13]. Dice represents the actual dice and has an operation to roll the dice, whereas DiceResult is an utility class for storing the result of the roll. We also needed a class that could confer responsibilities to the three classes. For that we made a class called board.

[8] Appendix 7
[9] Appendix 5
[10] Appendix 6
[11] Appendix 3
[12] Appendix 1
[13] Appendix 2

The board class contains the main method, and it is from here the game is played. This class controls the different player objects and add points to the correct player, after the dice have been rolled. It also evaluate the roll i.e. it checks if the dice are alike, and if the current player meets the win conditions. In this way, we can control if the player needs an extra roll or not, and whether the player can win if he rolls two of a kind. It also checks the previous roll to see if it was two 6's, so that if the same roll is made, the player also wins. The way we track this is by giving each player their own set of dice which then keeps track of the previous RollResult.

You can see in the diagram below that different attributes and methods come with the Board class. When we wrote the board class we did so with extensibility in mind. This is why we use an ArrayList to keep track of the players instead of an array. This allows us to easily extend the amount of players without too much of a hassle later on, should the requirements change. For now only two players can participate, even though more are supported.[14]




Another thing to notice is the function evaluateRoll. This function has the responsibility to check if any of the game rules has been fulfilled and take appropriate action if so. Our thoughts during the design phase was that it would get the dice results to evaluate directly from currentPlayer. During the implementation phase we discovered that it would be much more convenient if the dice results were passed to the function, as this allowed us to easily test the logic through unit tests.

For each of our four classes we made a class description followed by a class diagram. We made our class description to give ourselves an idea of the different classes, what they should contain, the variables, methods that belonged to them and how they were related to each other. By doing so, it was much easier to write the code, when we got to the implementation phase.

After that we continued to the domain model[15], in order to find the different relationships between the classes and the multiplicity as well. When making these models it is good to have in mind that the more connections a class have with other classes, the more you have to edit in your coding because a class have influence on other classes. You can see the domain model in appendix 8. Here we discovered that Board would depend on the Player class, and the Player class would depend on the Dices, as seen in the diagram. Neither Player, nor Dice had to know about the Board class, as the Board class controlled the logic for the gameplay. We decided to go with composition because the game wouldn't function without Dices to increase score and Players to roll the dice.

[14] Appendix 4
[15] Appendix 8

Then we had to make the multiplicity between the classes clear. The Board class plays the game, and in each game, two players battle each other. Therefore the multiplicity for Board is 1 and for Player it is 2 because two players is always a part of oné board. Likewise one player is connected with one Dice. Therefore the multiplicity is 1 for Player and 1 for Dice. The last multiplicity is between Dice and DiceResult. DiceResult contains the results of one roll, but in the Dice class we need to keep track of two rolls, the current roll and the previous roll. In order for that to happen, we need to have two DiceResults for each Dice. The multiplicity is by that 2 for Diceresult and 1 for Dice.

When the domain model was finished we started on the BCE-model[16]. The BCE-model helps us get a better overview of each class' responsibility. We concluded that the player interacts with the GUI and therefore the GUI is, cf. the terminology, a boundary class. This is depicted in appendix 9. The GUI gets its information from the Board which controls the game. And the board gets the information from the Player class and the Dice class and stores the advancements of the game in these two classes. The Board class is by that our control class, and the Player and Dice class are both entity classes. At last we used all our prior diagrams to create a system sequence diagram[17], which allowed us to model how our code should advance and how the classes should communicate with each other and when.

At last we made a design sequence diagram[18] where we modelled how the code should run. Here we designed it so that the code would run in a loop until one of the winning conditions were met.

[16] Appendix 9
[17] Appendix 10
[18] Appendix 11

# Conclusion

## Overall conclusion

We are very satisfied with our results. We started out by reading carefully all the requirements through both the basic ones and the additional requirements that have been given, and even made contact with the client, to secure, that he was satisfied with the upcoming program. After that was over, we started out by designing many successful diagrams, which then described how our thought process was to the design of our program. Eventually we were able to meet all the requirements in the most optimal way that we could think of, which made the overall project very satisfying. The product-oriented and process-oriented conclusion will be described below as well as future opportunities for the program.

## Product-oriented conclusion

The program has been developed to satisfy our understanding of the requirement specification. All requirements and extra requirements have been fulfilled, which we also listed in our 'Requirements and Demands'-section[19]. All rules of the game have been implemented and they are easy to understand by the user. We have used the GUI that was given to us to show the result of the dice and to inform the player about the progress of the game. We made sure that the dice are random every time they are rolled.

## Process-oriented conclusion

We began development of the program by understanding and analysing the requirements specification. By doing so we got a clear view of what our program is supposed to do. We made sure with the customer that there were no misunderstandings or anything left out. Finally we came up with a list of functional and nonfunctional requirements for the program[20].

Now that we had the task laid out we could begin getting a rough understanding of the program. We started out by making use cases to get an understanding of how the user would interact with the program. From there on we made domain models, BCE models, sequence diagrams and class diagrams to get a proper understanding of the programs structure and inner workings. Progressing with each model the final product became clearer.

With our final product visualized we could begin coding it. Every member of the group had to make his own solution with roots in the models we made. The group chose one of the solutions to keep and work on. By doing this every member got experience in programming. When the program was done we moved on to testing. We performed positive tests to see if the rules of the game worked and the dice rolled correctly. We tested the dice to make sure that their results were statistically correct, which was part of the requirements for the program.

[19] See Requirement table (page 4)
[20] See Requirement table (page 4)

## Future opportunities

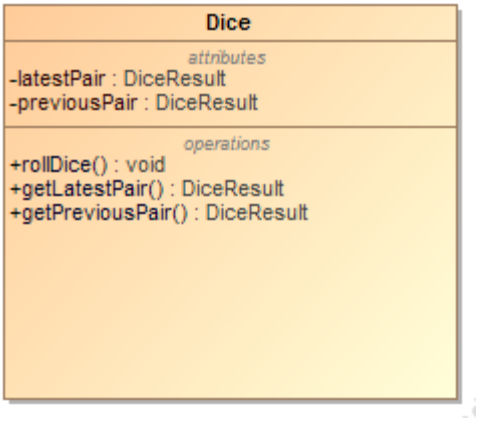
For our future opportunities with the program, may be implementation of more specific rules. For example, that could be, if you roll two 3's then you lose half of your points or something similar to that. We also thought that you could easily add more than two players in the game. Therefore, as mentioned before, we had implemented an array list for the players, so you can easily change it to be more than two players. With more players, we could perhaps turn the game to an online one, or make some small fixes to the board with advertising or a better design over.

## Literature and source list

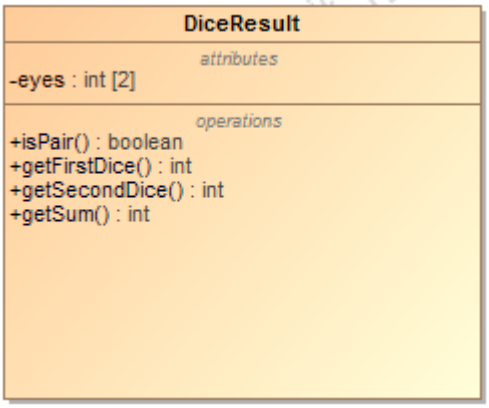
Jim Arlow & Ila Neustadt, 2005, U*ML 2 and the Unified Process: Practical Object-Oriented Analysis and Design.* Udgivelsessted: Addison-Wesley Professional. Second edition.

# Appendix

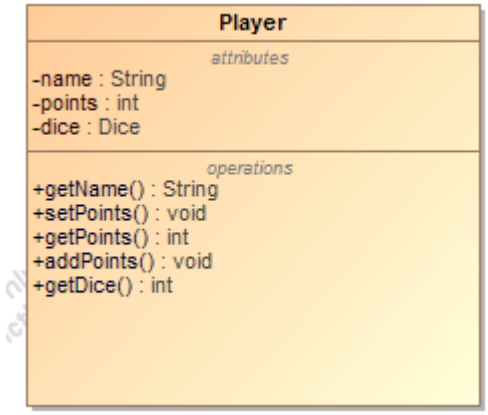
## Appendix 1




## Appendix 2




## Appendix 3

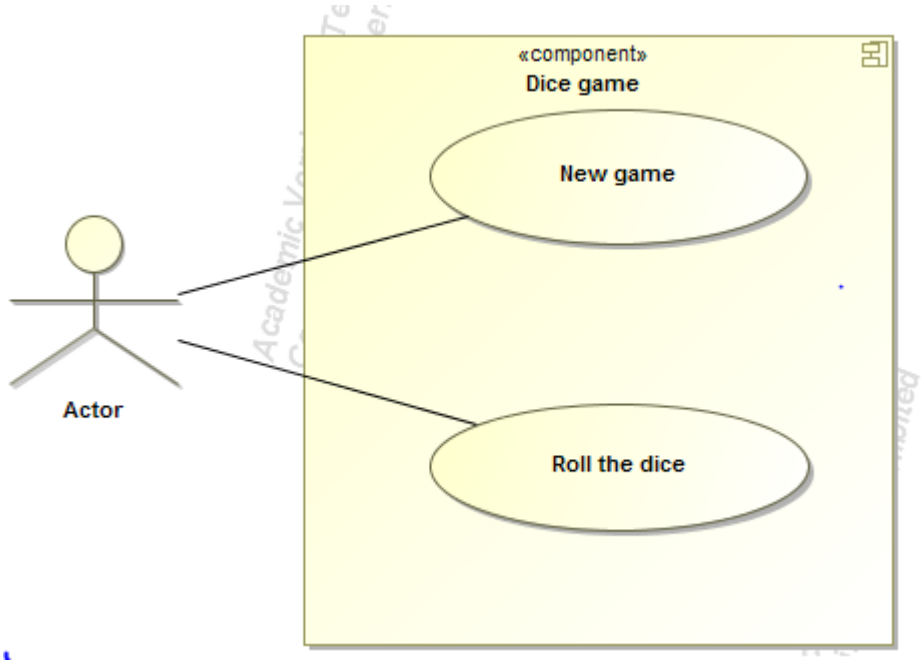


## Appendix 4


## Appendix 5

**Use cases:**

- Begin game – choose player 1 and 2
- Roll dice
- A player fulfills the win-conditions (rolling  a pair of 6 twice in a row or when  the player is above 40 points and rolls a pair)



## Appendix 6

**Use Case Description "Roll the dice"**

**ID:**
1

**Brief description:**
When it's the player's turn and he needs to roll with the dice

**Primary actors:**
Player

**Secondary actors:**
None

**Preconditions:**
When it's the player's turn

**Main flow:**

1. Player rolls the dice
2. The system shows the number of eyes
3. Add number of eyes to point
4. If player rolls to of a kind
    a. If two 1's
        i. Set points to 0
    b. Else, if the player has 40 points
        i. If true, player wins
    c. Else
        i. If player rolls two 6's
            1. If last roll was two 6's
                a. Player wins
    d. New turn

**Post conditions:**
None

**Alternative flows:**
None

## Appendix 7

**Noun- and verb analysis:**

**Nouns:**

First player and second player

- Synonyms: Player 1 and Player 2.

The sum of eyes

- Synonyms: Result and outcome.

Points

- Synonyms: Sum and score

**Verbs:**

To win, [Player] wins

- Synonyms: Victory

To start (A player rolls the dice)

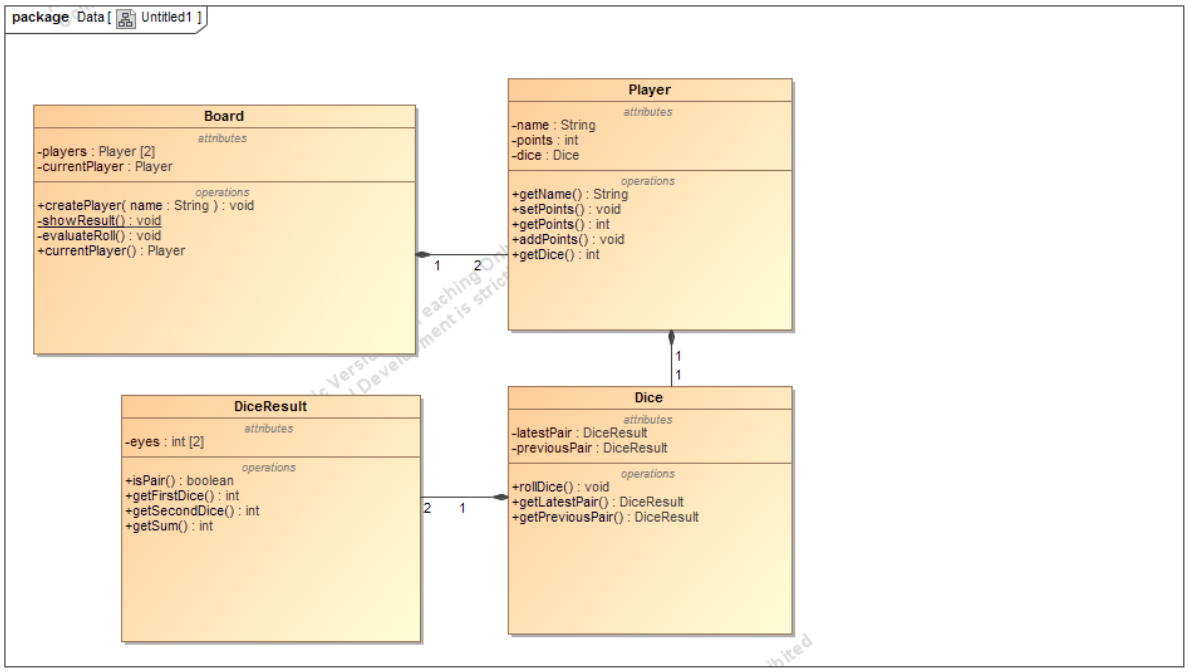
- Synonyms: Roll first

To roll the dice

- Synonyms: Throw the dice

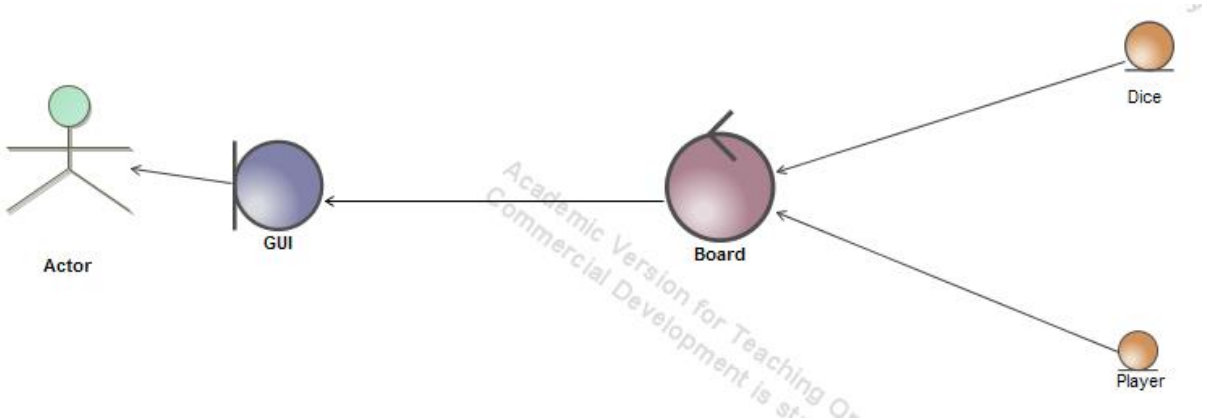
## Appendix 8

**Domain model**



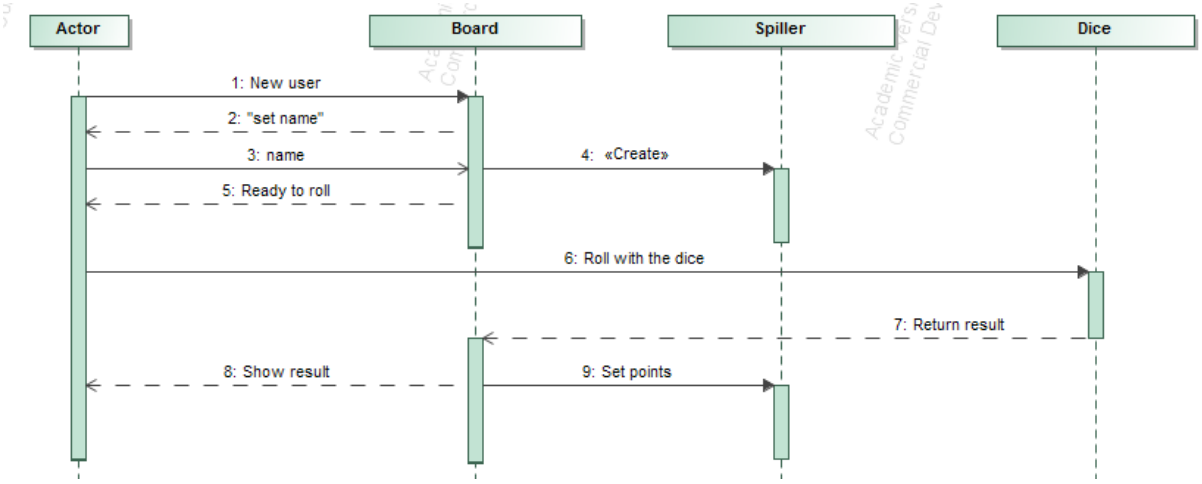

## Appendix 9

**BCE-model**



## Appendix 10

**System sequence diagram**




## Appendix 11

**Design sequence diagram**