Projektopgave efterår 2015 – jan 2016

02312-14 Indledende programmering, 02313 Udviklingsmetoder til IT-Systemer og 02315 Versionsstyring og testmetoder.

Projekt navn: *CDIOdel2*

Gruppe nr.: *22.*

Afleveringsfrist: *lørdag d. 07/11-2015 kl: 04:59*

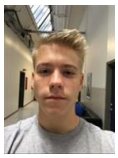
Denne rapport er afleveret via Campusnet (der skrives ikke under)

Denne rapport indeholder 43 sider inkl. denne side.

Studie nr.:	Efternavn, Fornavne	Underskrift
s144219,	*Gregersen, Vulpius Rasmus*	_____

Kontakt person (Projektleder)

s153256, *Moulvad, Rasmus Hannibal Barlach* _____



s133010, *Samuelsen, Alexander* _____

s154067, *Andresen, Alexandra Boel* _____

s143312, *Kristensen, David Josef* _____

s153431, *Steensen, Mark Sloth* _____

Timesheets

CDIO Del 2							
Time-regnskab	**Total**						
Dato	Deltager	Design	Impl.	Test	Dok.	Andet	Ialt
	Alexander	6	1	1,5	6	0	26,5
	Alexandra	4	0	2	2	0	23
	David	6	1	2,5	6	0	27,5
	Hannibal	6	1	2,5	6	0	30,5
	Mark	3	1	2,5	5	0	21,5
	Rasmus	6	1	1,5	4	0	25,5
	Sum	31	5	12,5	29	0	154,5
						Control:	154,5

Abstract

This report examines the creation of a game from the very first steps of the process until completion. Beginning with the design, then implementation, testing and finally documentation. The conclusion reviews what we have learned during the process.

Table of contents

Introduction

This report is composed in the courses 02312-14 Introductory Programming, 02315 Version control and Test methods and 02313 Development methods for IT-Systems, during first term on the IT and Economics study on DTU. The assignment is part two of three CDIO-projects, which will be completed during the first term. While working on this assignment, we will get introduced to the basic important concepts of programming. The overall purpose of the three parts is to create a Monopoly-game.

Main section

In this assignment we are a working for the game-firm IOOuterActive. The firm has received an order to create a computer game. As employees of the firm, it is our job as programmers to design, construct, test and develop the game. The customer has shared requirements for game-rules and has asked for program testing.

Requirement analysis

The first and most important things to be discussed before starting programming are the requirement specifications. The client has given us a description of how the game should operate, and which rules they want included in the game-rules.

The request is a 2-player game played by throwing 2 dice. To obtain an overview of the game we have chosen to structure the requirements in functional and non-functional requirements.

Functional requirements

Beneath is a description from the customer, which we have chosen as the documentation for the functional requirements.

"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 -200

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 -90

11. Goldmine +650"

We have sorted these requirements and listed them under different categories:

- Functionality:
  - 2-player game.
  - The game it to contain 12 fields and 2 bricks (cars).
  - The game must function when you throw the dice-cup with two dice and the result is to be shown immediately after.
    - The dice-cup must contain two dices with six sides, and the outcome will be two random numbers between 1 and 6.
  - Game rules:
    - Both players start on a field named StartPosition.
    - Both players start with 1000 points.
    - The values shown on the dice determines how many fields the player is to move forward.
    - Each field contains a unique condition which affects the player's Balance.
    - The game ends when a player reaches 3000 points.
- External interface:
  - A GUI.
- Additional features
  - Field Overview:
    1. Tower, + 250 points.
    2. Crater, - 200 points.
    3. Palace Gates, - 100 points.
    4. Cold Desert, - 20 points.
    5. Walled City, + 180 points.
    6. Monastery, + 0 points.
    7. Black Cave, - 70 points.
    8. Huts in The Mountain, - 60 points.
    9. The Werewall, - 80 points and receives an extra turn.
    10. The Pit, - 90 points.
    11. Goldmine, + 650 points.
    12. StartPosition, + 0 points.

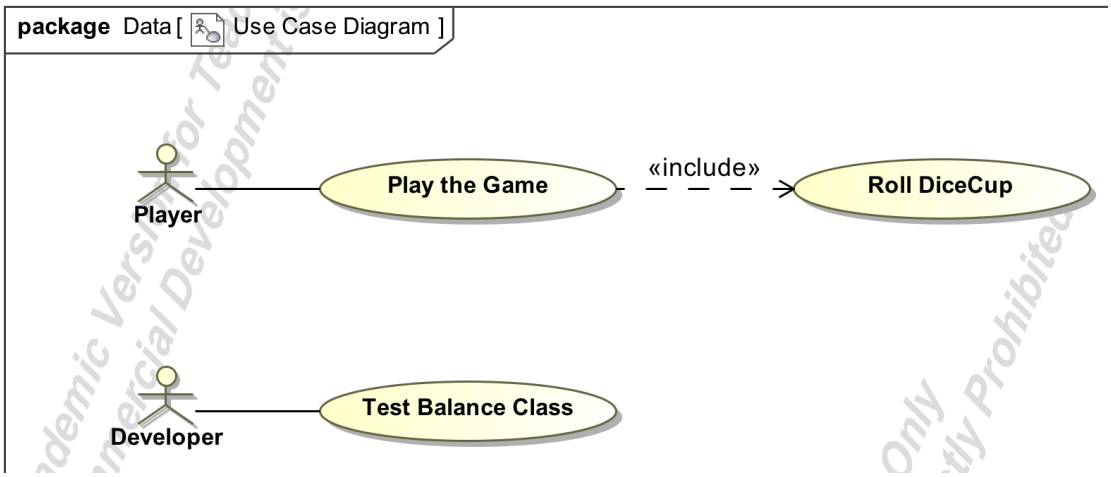
Non-functional requirements

- The client has given us a GUI from a different project and has allowed us to use it for this dice game. The GUI controls the non-functional requirements, which means the game is already designed and therefore we have not been tasked to construct any non-functional requirements.
- The program must run under Windows.
- The program should run without any noticeable delays (less than 333 ms).

It is expected that any person can play the game without a manual.

Use cases

This use case diagram describes the different actions and its relationship between the different actors.



Use Case: Play the Game
ID: 1
Brief description: The game is a two player game and consists of two dice that you will roll each turn. The dice roll determines what field the player will move her/his artifact to. Each field has their own unique condition. Each player begins with a balance of 1000 point. The game ends when one of the players reaches 3000 point.
Primary actors: Player
Secondary actors:
Preconditions: Player has access to a computer in the DTU Databar.
Main flow:

1. The players enters their names. 2. The players press the button "Roll" to play the game. 3. Include (Roll DiceCup). 4. After each roll the player will move to another field. 5. The field may change the player's points, or reward with an extra turn. 6. The turn switches after each player's roll. 7. The game ends when one of the players has won.
Post conditions: To win the game, one of the players have to: Reach 3000 points and then the game ends.
Alternative flows: None.

Use Case: Roll DiceCup
ID: 3
Brief description: The dicecup rolls two dice and sums the two values.
Primary actors: Developer & Player
Secondary actors:
Preconditions: A player has pressed the "Roll" button.
Main flow: 1. Rolls two dice. 2. Reads the values of each of them. 3. Sums the values.
Post conditions:
Alternative flows: • When a player lands on field "The Werewall", the player receives an extra dicecup roll.

Use Case: Test Balance Class
ID: 2
Brief description: The test makes it convincing that the balance can never be negative, whatever the withdraw and

deposit methods are called with. To give the best result the test will test 10 withdraws and 10 deposits from the balance.
Primary actors: Developer
Secondary actors:
Preconditions: None.
Main flow: 1. 10 deposits between 1 and 5000 will show that a balance = 0 can never reach over 3000 points. 2. 10 withdraws between 1 and 5000 will show that a balance = 3000 can never be below 0 points.
Post conditions:
Alternative flows:

System analysis

Noun/verb analysis

In the noun and verb analysis we have highlighted the noun and verbs from the customer's requirements. We have used the nouns to define the classes that we need for this program. We have then categorized the nouns under it's given class.

We have highlighted all of the verbs in the customer's requirement. We have chosen the most essential verbs to define some of our needed methods to fulfil our project.

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 pa° 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.

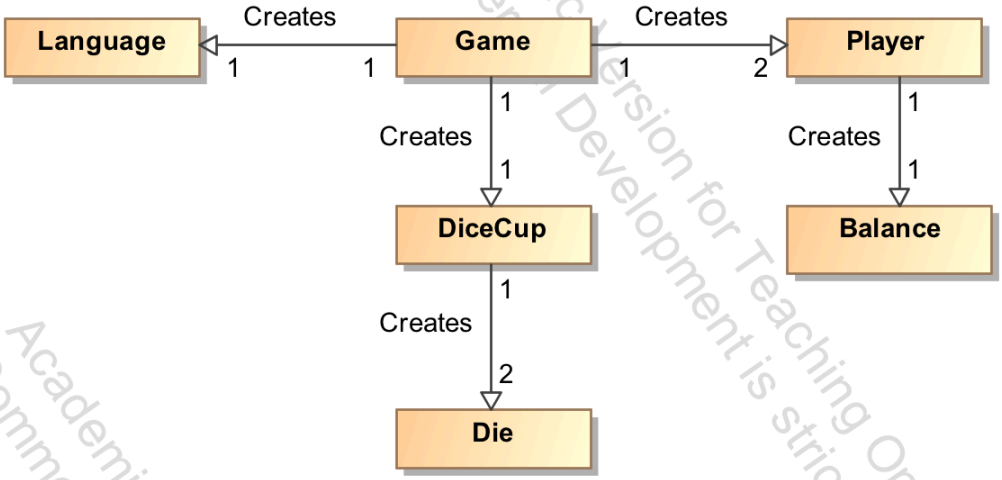
Spillet er slut når en spiller har 3000.
Spillet skal let kunne oversættes til andre sprog.
Det skal være let at skifte til andre terninger.
Vi vil gerne have at I holder jer for øje at vi gerne vil kunne bruge spilleren og hans pengebeholdning i andre spil.

Blue: Nouns
Red: Verbs

Nouns	Classes
Personer, Spillerne, Spiller	Player
Sprog	Language
Pengebeholdning, Beholdning	Balance
Terninger, (andre) Terninger	Die
Terninger, Numrene	Dicecup
Terningespil, Spil, Effekt, Spillet, Felt	Game
Verbs	**Methods**
Slå	newRoll(), getEqual(), GetSum(), getDie1(), getDie2()
Udskrives	GUI.Showmessage(), getField1-11()
Har	player.getBalance()

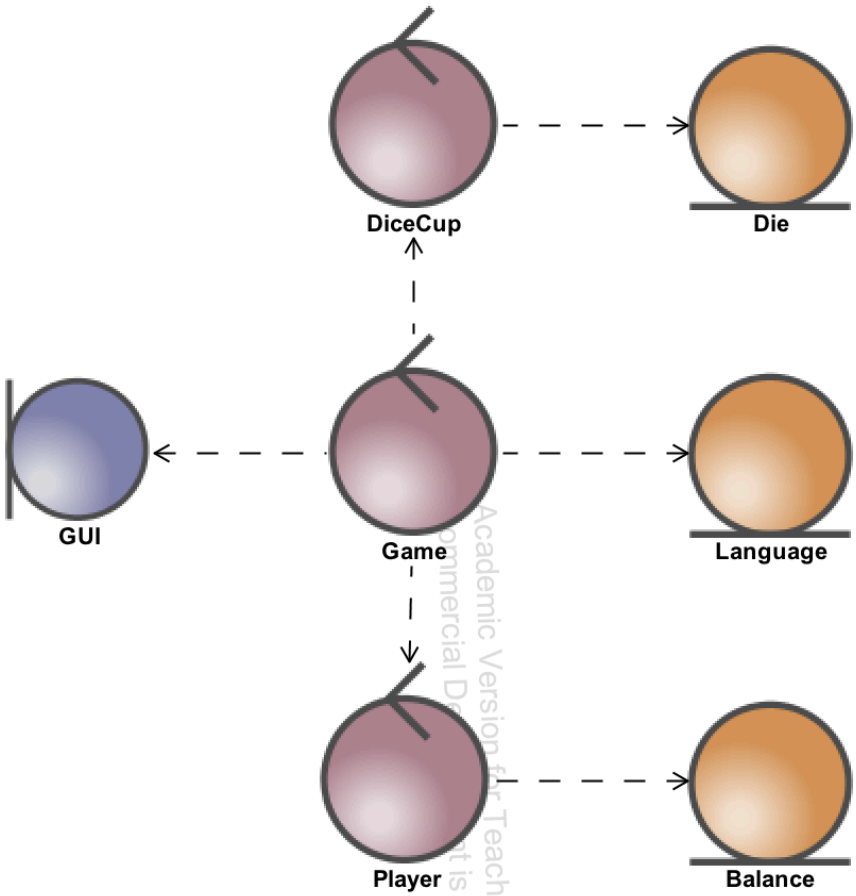
Domain model

The Domain model describes the connection between the different classes and how they cooperate. This model is very meaningful for the project, since it provide us with knowledge how the different classes relate to each other.



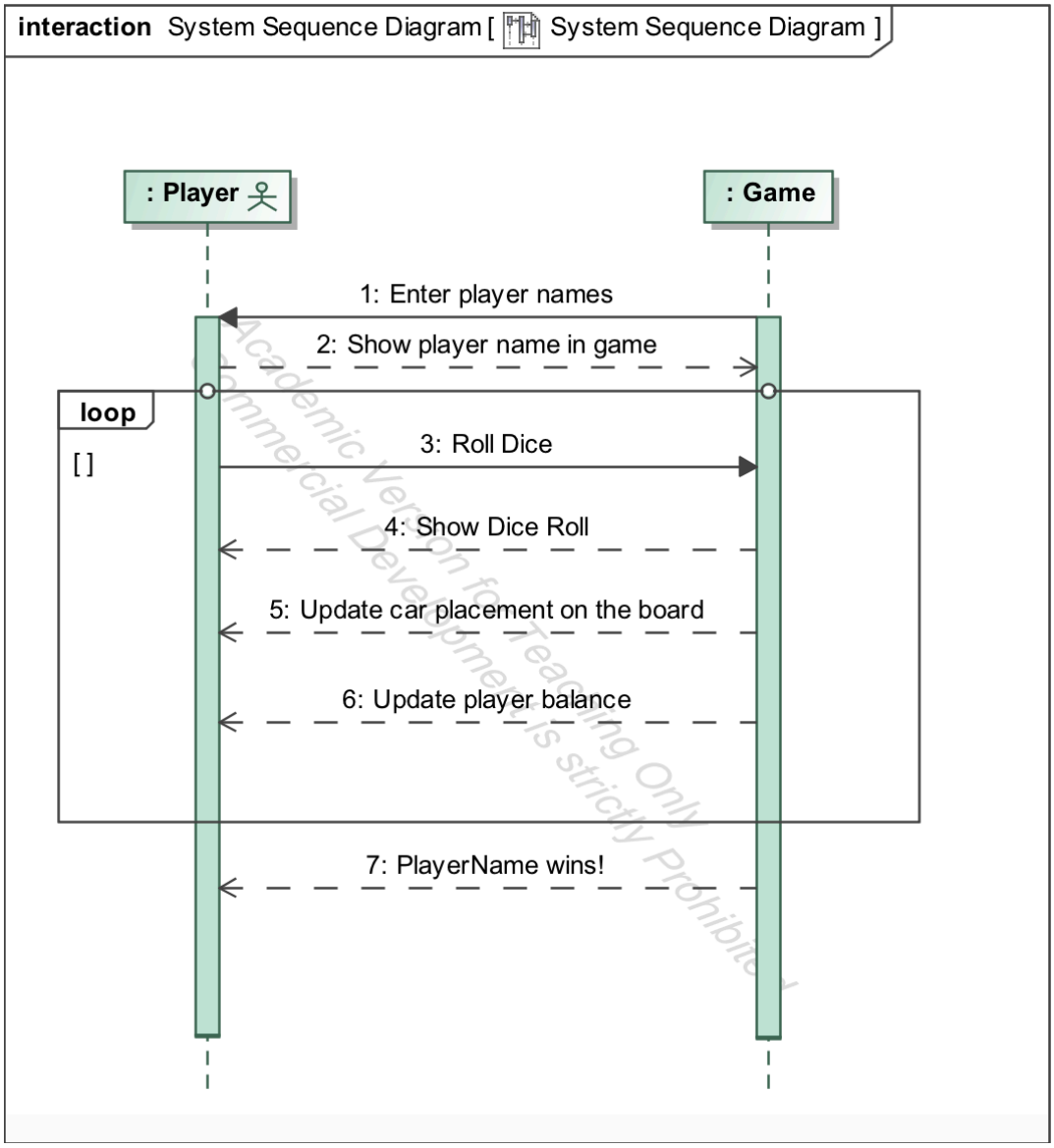
BCE model

The different program classes can be categorized in 3 different type of classes. The different types are called the Boundary, Control, and Entity class.
Boundary classes are those classes which communicate with the system's surroundings.
Control classes share and gather the responsibility between other objects and determines the sequence in different actions.
Entity classes is the information which is used by the Boundary classes and shared with the control classes. These classes are the memory where the data is stored.



System sequence-diagram (SSD)

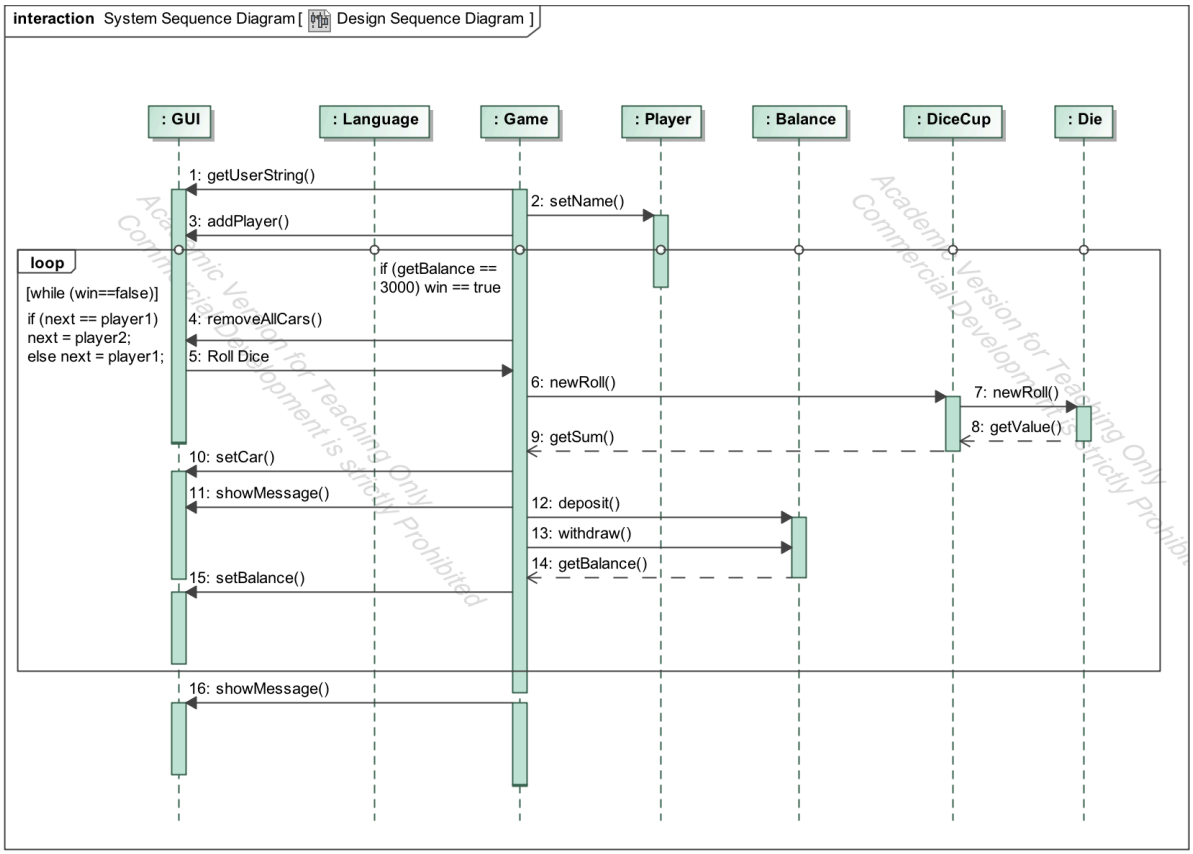
The sequence diagram shows, particular scenario of a use cases, the events that external actors generate, their order, and possible inter-system events.



System design

Design sequence-diagram (DSD)

A system sequence diagram is a sequence diagram that shows, for a particular scenario of a use case, the events that the different classes generate, their order, and possible inter-system events.

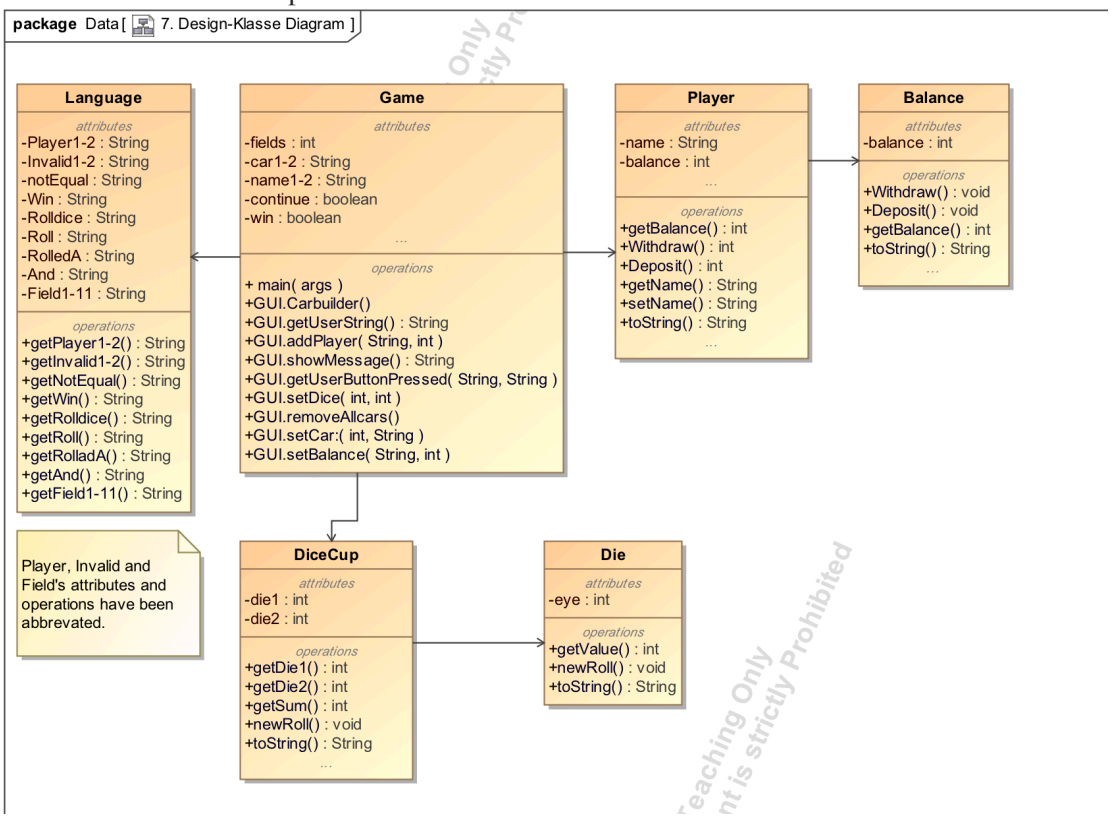


Design class-diagram with named relations

The classes in a class diagram represent both the main objects, interactions in the application and the classes to be programmed.

In the diagram, classes are represented with boxes which contain three parts:

- The top part contains the name of the class.
- The middle part contains the attributes of the class.
- The bottom part contains the methods the class can execute.



Test

Balance Test

(See appendix 2 for source code.)

The costumer requested that we made a test to make sure that the balance complies with the balance requirement, which is; the balance may never be negative.

We created a new class called "TestNegativeBalance". To show that the balance never could be negative we chose to show it by making 10 different withdraws and deposits.

By looking at the 20 different results the balance would not show a negative result.

Withdrawal test: 1 -
Balance reset to 0!
Trying to deposit: 2021
Balance is now: 2021

Withdrawal test: 2 -
Balance reset to 0!
Trying to deposit: 4602
Balance is now: 3000

Withdrawal test: 3 -
Balance reset to 0!
Trying to deposit: 3837
Balance is now: 3000

Withdrawal test: 4 -
Balance reset to 0!
Trying to deposit: 3424
Balance is now: 3000

Withdrawal test: 5 -
Balance reset to 0!
Trying to deposit: 2570
Balance is now: 2570

Withdrawal test: 6 -
Balance reset to 0!
Trying to deposit: 3628
Balance is now: 3000

Withdrawal test: 7 -
Balance reset to 0!
Trying to deposit: 2366
Balance is now: 2366

Withdrawal test: 8 -
Balance reset to 0!
Trying to deposit: 2050
Balance is now: 2050

Withdrawal test: 9 -
Balance reset to 0!
Trying to deposit: 4301
Balance is now: 3000

Withdrawal test: 10 -
Balance reset to 0!
Trying to deposit: 1060
Balance is now: 1060

Deposit test: 1 –
Balance reset to 3000!
Trying to withdraw: 2810
Balance is now: 190

Deposit test: 2 –
Balance reset to 3000!
Trying to withdraw: 1069
Balance is now: 1931

Deposit test: 3 –
Balance reset to 3000!
Trying to withdraw: 2515
Balance is now: 485

Deposit test: 4 –
Balance reset to 3000!
Trying to withdraw: 203
Balance is now: 2797

Deposit test: 5 –
Balance reset to 3000!
Trying to withdraw: 138
Balance is now: 2862

Deposit test: 6 –
Balance reset to 3000!
Trying to withdraw: 2627
Balance is now: 373

Deposit test: 7 –
Balance reset to 3000!
Trying to withdraw: 1726
Balance is now: 1274

Deposit test: 8 - Balance reset to 3000!
Trying to withdraw: 3014
Balance is now: 0

Deposit test: 9 –
Balance reset to 3000!
Trying to withdraw: 2647
Balance is now: 353

Deposit test: 10 –
Balance reset to 3000!
Trying to withdraw: 65
Balance is now: 2935

Responsetime Test

The customer requested that the program could run without remarkable delays (333.0 ms.). We ran the test 10 times on the DTU data bar, and found an average response time.

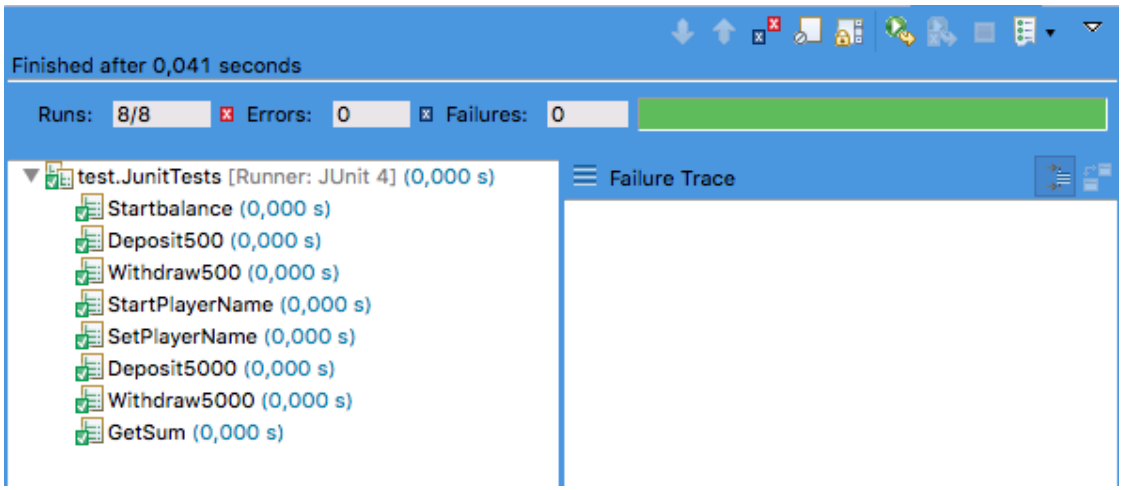
Test	Response Time
1	1,0877 ms.
2	1,1963 ms.
3	1,0104 ms.
4	1,5473 ms.
5	0,9538 ms.
6	1,3822 ms.
7	1,25 ms.
8	1,7083 ms.
9	1,4367 ms.
10	1,0755 ms.
Average Response Time	***1,2648 ms.***

Junit Test

JUnit is a regression testing framework which we used to implement unit testing in Java and increase the quality of code. We wanted to secure that the different methods worked correctly.

Our JUnit test was successful and showed a low response time for our methods.

Picture of JUnit result:



GRASP Principles

GRASP = General Responsibility Assignment Software Patterns

We have made our project in relation with 5 principles of GRASP.

1. Creator
2. Information Expert
3. Low Coupling
4. Controller
5. High Cohesion

1. Creator:

To comply with the Creator principle we have a close connection between following classes:

- DiceCup -> Die: DiceCup class closely uses the Die class.
- Game -> Player: The Player class records information from the Game class.
- Player -> Balance: The Balance class records information from the Player class.
- Game -> DiceCup: Instances of Game class is closely used in the DiceCup.
- Game -> Language: Instances of Game class records instances of Languages.

2. Information Expert

We have assigned the responsibilities of our information to corresponding classes. All information regarding the player lies in the “Player class” (or Balance Subclass). And again for the “Die” and “DiceCup “classes, which are responsible for the Dice, rolls.

3. Low Coupling

We have met the low coupling measure for both our “Player class” and “DiceCup class”, which have low dependency on other classes and are easily reused in our next CDIO project.

4. Controller

Our Game class is used as our Controller, since it deals and delegates the work to the other objects.

5. High Cohesion

To obtain an effective and simple system, we made classes that have a clear and simple purpose, which relates to another class such as “DiceCup” and “Die” class. We also saw this with “Player” and “Balance” class.

Configuration

Requirements and setup guide for the the program:

1. Java Development Kit version 1.8
   http://www.oracle.com/technetwork/java/javase/downloads/index.html
2. Eclipse Installed on your computer. Preferably Mars 4.5 for java EE developers: https://www.eclipse.org/downloads/
3. Open Eclipse once you have installed it.
4. Select the default workspace
5. Go to file -> Open Import -> Choose General -> Choose Existing Projects into Workspace.
6. Make sure that you have unzipped the file “22_CDIO2.zip”
7. Browse and select the unzipped folder -> press OK -> and select finish.
8. Now the project is imported to Eclipse and you are now able to run the game.
9. Double click on the project in the Project Explorer.
10. Now the project is open, and you can press run.

Conclusion

Due to preliminary work such as CDIO1, use cases, BCE models, GUI and sequence diagrams we have managed to code a board game based on the requirements received from the costumer.

The game was constructed by first creating all the models and diagrams and after we were able to start building the program. We took advantages of the ”Die” and ”DiceCup” classes, which we had already created in the CDIO1 assignment, and used them as the grounding for the new board game. Later we had to create new classes and soon we had developed the game. One importing thing to be done was testings of the program to make sure that it worked correctly. We made several tests during the actual game programming. We have learned, by working on the CDIO-projects, that the more tests you make the better the program will turn out. In this assignment we also had to make the game easy to translate into other languages. Therefor we made a Language class which contains all wordings that are defined as “String”.

On this assignment we spend more time on following the assignment instructions, which made a better structure during the designing, implementation, testing and the documentation.

Appendix

Source code

Game

```
/*
 * Game
 *
 * Version info
 *
 * Copyright notice */

package game;

import java.awt.Color;
import java.util.Set;

import desktop_codebehind.Car;
import desktop_codebehind.Car.Builder;
import desktop_resources.GUI;

public class Game {

	public static void main(String[] args) {


		// Creates new entities of our subclasses.
		DiceCup dicecup = new DiceCup();
		Player player1 = new Player();
		Player player2 = new Player();
		Language language = new Language();

		// Initialize fields variable for the game board.
		int fields = 0;

		// Creates new entity for the player cars (pieces).
		Car car1 = new Car.Builder() // chaining
				.primaryColor(Color.ORANGE)
				.secondaryColor(Color.BLACK)
				.typeRacecar()
				.patternDotted()
				.build();


		Car car2 = new Car.Builder() // chaining
					.primaryColor(Color.CYAN)
					.secondaryColor(Color.BLACK)
					.typeRacecar()
```

```
                                .patternDotted()
                                .build();

                // New variables for player names, and input thru the GUI.
                String name1 = GUI.getUserString(language.getPlayer1());
                String name2 = GUI.getUserString(language.getPlayer2());

                // Boolean variable for while loop.
                boolean Continue = false;

                // Checks if the playernames live up to the requirements.
                while (Continue == false) {
                        // Player name must be between 1 and 15 characters and the
first character cannot be space.
                        if (name1.length() < 1 || name1.length() > 15 ||
name1.indexOf(" ") == 0)
                                name1 = GUI.getUserString(language.getInvalid1());


                        else if (name2.length() < 1 || name2.length() > 15 ||
name2.indexOf(" ") == 0)
                                name2 = GUI.getUserString(language.getInvalid2());
                        // Player names must not be identical. (Ignores case)
                        else if (name1.equalsIgnoreCase(name2))
                                name2 = GUI.getUserString(language.getnotEqual());
                        else
                                Continue = true;
                }
                // set player names from GUI input.
                player1.setName(name1);
                player2.setName(name2);

                // add players to the GUI.
                GUI.addPlayer(player1.getName(), player1.getBalance(), car1 );
                GUI.addPlayer(player2.getName(), player2.getBalance(), car2 );

                // declare next turn to player1. (First turn).
                Player next = player1;

                // while loop that runs until a winner is found. (win == true).
                boolean win = false;
                while (win == false)
                {
                        // The end! Congratulates the winner!
                        if (player1.getBalance() == 3000) {
                                GUI.showMessage(player1.getName() +
language.getWin());
                                win = true;
                                break;
```

```
                        }
                        else if (player2.getBalance() == 3000) {
                                GUI.showMessage(player2.getName() +
language.getWin());
                                win = true;
                                break;
                        }

                        // Game begins! GUI method displays 'Roll' button.
                        GUI.getUserButtonPressed(language.getrollDice(),
language.getRoll());

                        // Calls the method for a new roll, and displays it in the GUI
                        dicecup.newRoll();
                        GUI.setDice(dicecup.getDie1(), dicecup.getDie2());

                        // If player1 just rolled, sets next turn to player2.
                        if(next == player1) {
                                next = player2;
                                // Remove all cars belonging to player1 from the board.
                                GUI.removeAllCars(player1.getName());
                                fields = dicecup.getSum();
                                // Sets the car on the board according to the diceroll
                                GUI.setCar(fields - 1, player1.getName());
                                GUI.showMessage(player1.getName() +
language.getRolleda() + dicecup.getDie1() + language.getAnd() + dicecup.getDie2());
                                // Switch with all game fields and the corresponding
actions.
                                switch (fields) {
                                case 2:  player1.deposit(250);
                                GUI.showMessage(language.getField1());
                                break;
                                case 3:  player1.withdraw(200);
                                GUI.showMessage(language.getField2());
                                break;
                                case 4:  player1.withdraw(100);
                                GUI.showMessage(language.getField3());
                                break;
                                case 5:  player1.withdraw(20);
                                GUI.showMessage(language.getField4());
                                break;
                                case 6:  player1.deposit(180);
                                GUI.showMessage(language.getField5());
                                break;
                                case 7:
                                GUI.showMessage(language.getField6());
                                break;
                                case 8:  player1.withdraw(70);
                                GUI.showMessage(language.getField7());
```

```
                        break;
                        case 9:  player1.withdraw(60);
                        GUI.showMessage(language.getField8());
                        break;
                        case 10: player1.withdraw(80);
                        GUI.showMessage(language.getField9());
                        next = player1;
                        break;
                        case 11: player1.withdraw(90);
                        GUI.showMessage(language.getField10());
                        break;
                        case 12: player1.deposit(650);
                        GUI.showMessage(language.getField11());
                        break;
                        }
                        // Updates balance in the GUI.
                        GUI.setBalance(player1.getName(),
player1.getBalance());

                }
                else {
                        // Repeat for player 2
                        next = player1;
                        GUI.removeAllCars(player2.getName());
                        fields = dicecup.getSum();
                        GUI.setCar(fields - 1, player2.getName());
                        GUI.showMessage(player2.getName() +
language.getRolleda() + dicecup.getDie1() + language.getAnd() + dicecup.getDie2());

                        switch (fields) {
                        case 2:  player2.deposit(250);
                        GUI.showMessage(language.getField1());
                        break;
                        case 3:  player2.withdraw(200);
                        GUI.showMessage(language.getField2());
                        break;
                        case 4:  player2.withdraw(100);
                        GUI.showMessage(language.getField3());
                        break;
                        case 5:  player2.withdraw(20);
                        GUI.showMessage(language.getField4());
                        break;
                        case 6:  player2.deposit(180);
                        GUI.showMessage(language.getField5());
                        break;
                        case 7:
                        GUI.showMessage(language.getField6());
                        break;
                        case 8:  player2.withdraw(70);
```

```
                                GUI.showMessage(language.getField7());
                                break;
                                case 9:  player2.withdraw(60);
                                GUI.showMessage(language.getField8());
                                break;
                                case 10: player2.withdraw(80);
                                GUI.showMessage(language.getField9());
                                next = player2;
                                break;
                                case 11: player2.withdraw(90);
                                GUI.showMessage(language.getField10());
                                break;
                                case 12: player2.deposit(650);
                                GUI.showMessage(language.getField11());
                                break;
                                }
                                GUI.setBalance(player2.getName(),
player2.getBalance());


                        }

                }

        }
}
```

Player

```
/*
* Player
*
* Version info
*
* Copyright notice */

package game;

public class Player {
	private String name;
	private Balance balance = new Balance();

	public int getBalance() {
		return balance.getBalance();
	}
	// Method to withdraw from the balance
	public int withdraw(int i) {
		balance.withdraw(i);
		return balance.getBalance();
	}
	// Method to deposit from the balance
	public int deposit(int i) {
		balance.deposit(i);
		return balance.getBalance();
	}
	// method to return player name.
	public String getName()	{
		return name;
	}
	// method to set player name.
	public void setName(String name) {
		this.name = name;
	}
	@Override
	public String toString() {
		return "################################\nPlayer [name=" + name
+ ", balance=" + balance + "]\n################################";
		//return "Player [name=" + name + ", balance=" + balance + "]";
	}

}
```

Balance

```
/*
* Balance
*
```

```
* Version info
*
* Copyright notice */

package game;

public class Balance {

	// Balance variable, starting at 1000.
	private int balance = 1000;

	// Method to withdraw from the balance
	public void withdraw(int balance) {
		this.balance = this.balance - balance;
	}
	// Method to deposit to the balance
	public void deposit(int balance) {
		this.balance = this.balance + balance;
	}
	// Method to get the current balance, and corrects it to 0 or 3000 if less or
more.
	public int getBalance()		{
		if (balance>3000)
			balance = 3000;
		else if (balance<0)
			balance = 0;
		return balance;
	}
	// toString method to keep track of variables in the class - For troubleshooting.
	@Override
	public String toString() {
		return "Balance [balance=" + balance + "]";
	}
}
```

DiceCup

```
/*
* DiceCup
*
* Version info
*
* Copyright notice */

package game;

public class DiceCup {
	private Die die1 = new Die();
	private Die die2 = new Die();

	//Method to return the value of die1
	public int getDie1() {
		return die1.getValue();
	}
	//Method to return the value of die2
	public int getDie2() {
		return die2.getValue();
	}
	//Method to return the sum of die1 and die2
	public int getSum() {
		return die1.getValue() + die2.getValue();
	}
	//Method to roll both dices.
	public void newRoll() {
		die1.newRoll();
		die2.newRoll();
	}
	@Override
	public String toString() {
		return "DiceCup [" + (die1 != null ? "die1=" + die1 + ", " : "") + (die2
!= null ? "die2=" + die2 : "") + "]";
	}
}
```

Die

```
/*
* Die
*
* Version info
*
* Copyright notice */

package game;

public class Die {

	private int eye; // die eyes

	// method to return eye value
	public int getValue() {
	return eye;

	}
	// method to roll the die (generate random value between 1 - 6)
	public void newRoll() {
		eye = (int) Math.ceil(Math.random()*6);
	}
	@Override
	public String toString() {
		return "Die [eye=" + eye + "]";
	}
}
```

Language

```
package game;

public class Language {

	private String player1 = "Please type player 1's name: ";
	private String player2 = "Please type player 2's name: ";
	private String invalid1 = "Invalid name! Please type player 1's name: ";
	private String invalid2 = "Invalid name! Please type player 2's name: ";
	private String notEqual = "You cannot take another players name! Please
choose a new one: ";
	private String win = "Wins! ";
	private String rolldice = "Press to roll the dice";
	private String roll = " Roll" ;
	private String rolleda = " rolled : " ;
	private String and = " and " ;
	private String field1 = "You have build a tower and sold it for +250";
	private String field2 = "Your mom fell out of the bed and a caused a crater -
200";
	private String field3 = "You have to  pay to enter the Palace Gates -100";
	private String field4 = "You reached the Cold Desert and have to pay for water
to survive - 20";
	private String field5 = "You have charged refugees to pay you for entering
your Walled City +180";
	private String field6 = "Nothing happens here... neither does anything happen
to your Balance";
	private String field7 = "It's completely dark in the Black Cave you buy a
flashlight -70";
	private String field8 = "You have to pay for an overnight stay in the Mountain
Huts -60";
	private String field9 = "You kill a werewolf. Once you reach the Werewolf
Wall you have to pay for your kill. But you are also congratulated on your kill
therefore -80  and an extra turn";
	private String field10 = "You fall into The Pit and you have to pay your savior
-90";
	private String field11 = "You have reached the Goldmine. You stack your
pocket with +650";

	public String getPlayer1()	{
		return player1;
	}
	public String getPlayer2()	{
		return player2;
	}
	public String getInvalid1()	{
		return invalid1;
	}
	public String getInvalid2()	{
```

```
        return invalid2;
    }
    public String getnotEqual()    {
        return notEqual;
    }
    public String getWin()         {
        return win;
    }
    public String getrollDice()    {
        return rolldice;
    }
    public String getRoll()        {
        return roll;
    }
    public String getRolleda()     {
        return rolleda;
    }
    public String getAnd()         {
        return and;
    }
    public String getField1()      {
        return field1;
    }
    public String getField2()      {
        return field2;
    }
    public String getField3()      {
        return field3;
    }
    public String getField4()      {
        return field4;
    }
    public String getField5()      {
        return field5;
    }
    public String getField6()      {
        return field6;
    }
    public String getField7()      {
        return field7;
    }
    public String getField8()      {
        return field8;
    }
    public String getField9()      {
        return field9;
    }
    public String getField10()     {
        return field10;
```

```
        }
        public String getField11()      {
                return field11;
        }

}
```

Junit

```
package test;

import static org.junit.Assert.*;
import game.*;

import org.junit.Test;

public class JunitTests {

	// Tests start balance to be 1000
	@Test
	public void Startbalance() {
		Balance balance = new Balance();
		int result = balance.getBalance();
		assertEquals(1000, result);

	// Tests withdraw method
	}
	@Test
	public void Withdraw500() {
		Balance balance = new Balance();
		balance.withdraw(500);
		int result = balance.getBalance();
		assertEquals(500, result);
	}

	// Tests withdraw method can't return less than 0
	@Test
	public void Withdraw5000() {
		Balance balance = new Balance();
		balance.withdraw(5000);
		int result = balance.getBalance();
		assertEquals(0, result);
	}

	// Tests deposit method
	@Test
	public void Deposit500() {
		Balance balance = new Balance();
		balance.deposit(500);
		int result = balance.getBalance();
		assertEquals(1500, result);
	}

	// Tests deposit method can't return more than 3000
	@Test
	public void Deposit5000() {
```

```
            Balance balance = new Balance();
            balance.deposit(5000);
            int result = balance.getBalance();
            assertEquals(3000, result);
        }

        // Tests getName() method
        @Test
        public void StartPlayerName() {
            Player player = new Player();
            String result = player.getName();
            assertEquals(null, result);
        }

        // tests setName() method
        @Test
        public void SetPlayerName() {
            Player player = new Player();
            player.setName("Rasmus");
            String result = player.getName();
            assertEquals("Rasmus", result);
        }

        // Tests newroll() and getters in DiceCup.
        @Test
        public void GetSum() {
            DiceCup dicecup = new DiceCup();
            dicecup.newRoll();
            int result = dicecup.getDie1() + dicecup.getDie2();
            assertEquals(dicecup.getSum(), result);
        }

}
```

TestNegativeBalance

```
package test;

import game.Balance;

public class TestNegativeBalance {

	public static void main(String[] args) {
		Balance Balance = new Balance();


		// Deposit test 10 times with random numbers
		for (int i=0; i < 10; i++)
		{
			int dep = (int)Math.ceil(Math.random()*5000);
			Balance.withdraw(3000);
			Balance.getBalance();
			System.out.println("Withdrawal test: " + (i + 1) + " - Balance
reset to 0!");
			System.out.println("Trying to deposit: " + dep);
			Balance.deposit(dep);
			System.out.println("Balance is now: " + Balance.getBalance());
		}

		// Withdraw test 10 times with random numbers
		for (int i=0; i < 10; i++)
		{
			int wit = (int)Math.ceil(Math.random()*5000);
			Balance.deposit(3000);
			Balance.getBalance();
			System.out.println("Deposit test: " + (i + 1) + " - Balance reset
to 3000!");
			System.out.println("Trying to withdraw: " + wit);
			Balance.withdraw(wit);
			System.out.println("Balance is now: " + Balance.getBalance());
		}
	}

}
```

TestResponsetime

```
package test;
import java.awt.Color;
import java.util.Set;
import desktop_codebehind.Car;
import desktop_codebehind.Car.Builder;
import desktop_resources.GUI;
import game.*;

/* Description:
* This tests starts when the player press OK,
* and runs thru an entire game (without user input)
* and prints the average responsetime.
*/
public class testresponsetime {

	public static void main(String[] args) {
		GUI.showMessage("Start test!");
		double lStartTime = System.currentTimeMillis();
		double counter = 0;
		double difference = 0;
		// Import of 1 dicecup class and 2 player classes.
		DiceCup dicecup = new DiceCup();
		Player player1 = new Player();
		Player player2 = new Player();
		Language language = new Language();
		int fields = 0;


		Car car1 = new Car.Builder() // chaining
			.primaryColor(Color.ORANGE)
			.secondaryColor(Color.BLACK)
			.typeRacecar()
			.patternDotted()
			.build();


		Car car2 = new Car.Builder() // chaining
				.primaryColor(Color.CYAN)
				.secondaryColor(Color.BLACK)
				.typeRacecar()
				.patternDotted()
				.build();


		String name1 = "Player 1";
		String name2 = "Player 2";
		boolean Continue = false;
```

```
		while (Continue == false) {
			if (name1.length() < 1 || name1.length() > 15 ||
name1.indexOf(" ") == 0)
				name1 = GUI.getUserString(language.getInvalid1());

			else if (name2.length() < 1 || name2.length() > 15 ||
name2.indexOf(" ") == 0)
				name2 = GUI.getUserString(language.getInvalid2());
			else if (name1.equals(name2))
				name2 = GUI.getUserString(language.getnotEqual());
			else
				Continue = true;
		}
		// set player names from GUI input.
		player1.setName(name1);
		player2.setName(name2);

		// add players to the GUI.
		GUI.addPlayer(player1.getName(), player1.getBalance(), car1 );
		GUI.addPlayer(player2.getName(), player2.getBalance(), car2 );

		// declare next turn to player1. (First turn).
		Player next = player1;

		// while loop that runs until a winner is found. (win = true).
		boolean win = false;
		while (win == false)
		{
			// The end! Congratulates the winner!
			counter++;
			if (player1.getBalance() == 3000) {
				//GUI.showMessage(player1.getName() +
language.getWin());
				win = true;
				break;

			}
			else if (player2.getBalance() == 3000) {
				//GUI.showMessage(player2.getName() +
language.getWin());
				win = true;
				break;
			}

			// Game begins! GUI method displays 'Roll' button.
			//GUI.getUserButtonPressed(language.getrollDice(),
language.getRoll());
```

```
                    // Calls the method for a new roll, and displays it in the GUI
                    dicecup.newRoll();
                    GUI.setDice(dicecup.getDie1(), dicecup.getDie2());

                    // If player1 just rolled, sets next turn to player2.
                    if(next == player1) {
                            next = player2;
                            GUI.removeAllCars(player1.getName());
                            fields = dicecup.getSum();
                            GUI.setCar(fields - 1, player1.getName());
                            //GUI.showMessage(player1.getName() +
language.getRolleda() + dicecup.getDie1() + language.getAnd() + dicecup.getDie2());

                            switch (fields) {
                            case 2:  player1.deposit(250);
//                          GUI.showMessage(language.getField1());
                            break;
                            case 3:  player1.withdraw(200);
//                          GUI.showMessage(language.getField2());
                            break;
                            case 4:  player1.withdraw(100);
//                          GUI.showMessage(language.getField3());
                            break;
                            case 5:  player1.withdraw(20);
//                          GUI.showMessage(language.getField4());
                            break;
                            case 6:  player1.deposit(180);
//                          GUI.showMessage(language.getField5());
                            break;
                            case 7:
//                          GUI.showMessage(language.getField6());
                            break;
                            case 8:  player1.withdraw(70);
//                          GUI.showMessage(language.getField7());
                            break;
                            case 9:  player1.withdraw(60);
//                          GUI.showMessage(language.getField8());
                            break;
                            case 10: player1.withdraw(80);
//                          GUI.showMessage(language.getField9());
                            next = player1;
                            break;
                            case 11: player1.withdraw(90);
//                          GUI.showMessage(language.getField10());
                            break;
                            case 12: player1.deposit(650);
//                          GUI.showMessage(language.getField11());
                            break;
                            }
```

```
                        GUI.setBalance(player1.getName(),
player1.getBalance());
                        // Prints out player1's roll

                }
                else {
                        next = player1;
                        GUI.removeAllCars(player2.getName());
                        fields = dicecup.getSum();
                        GUI.setCar(fields - 1, player2.getName());
//                      GUI.showMessage(player2.getName() +
language.getRolleda() + dicecup.getDie1() + language.getAnd() + dicecup.getDie2());

                        switch (fields) {
                        case 2:  player2.deposit(250);
//                      GUI.showMessage(language.getField1());
                        break;
                        case 3:  player2.withdraw(200);
//                      GUI.showMessage(language.getField2());
                        break;
                        case 4:  player2.withdraw(100);
//                      GUI.showMessage(language.getField3());
                        break;
                        case 5:  player2.withdraw(20);
//                      GUI.showMessage(language.getField4());
                        break;
                        case 6:  player2.deposit(180);
//                      GUI.showMessage(language.getField5());
                        break;
                        case 7:
//                      GUI.showMessage(language.getField6());
                        break;
                        case 8:  player2.withdraw(70);
//                      GUI.showMessage(language.getField7());
                        break;
                        case 9:  player2.withdraw(60);
//                      GUI.showMessage(language.getField8());
                        break;
                        case 10: player2.withdraw(80);
//                      GUI.showMessage(language.getField9());
                        next = player2;
                        break;
                        case 11: player2.withdraw(90);
//                      GUI.showMessage(language.getField10());
                        break;
                        case 12: player2.deposit(650);
//                      GUI.showMessage(language.getField11());
                        break;
                        }
```

```
                                GUI.setBalance(player2.getName(),
player2.getBalance());


                        }
                        double lEndTime = System.currentTimeMillis();
                        difference = lEndTime - lStartTime;
                        System.out.println("Elapsed milliseconds : " + difference);

                }
                System.out.println("Antal kast: " + counter);
                System.out.println(difference/counter);
        }
}
```

Word- and symbol explanation, abbreviation

Abbreviation explanation:

CDIO = Conceive, Design, Implement and Operate

GUI = Graphical user interface

UML = Unified Modeling Language

BCE = Boundary Control Entity

SSD = System Sequence Diagram

DSD = Design System Diagram

DCD = Design Class Diagram

GRASP = General Responsibility Assignment Software Patterns/Principles

Literature and source index

- Jim Arlow, Ila Neustadt; ”*UML 2 and the Unified Process, Practical Object-Oriented Analysis and Design*.”