University of Zurich UZH

Software Construction

Assignment 2

Exercise 1

Group 50

Ece Asirim, 17-207-465
Ebubekir Celik, 21-752-431
Roberto Honegger, 16-715-419
Metehan Yildirim, 21-752-506

1. Following the Responsibility Driven Design, start from the game's requirements and rules and derive classes, responsibilities, and collaborations (use CRC cards). Describe each step you make and store the final cards in your answer

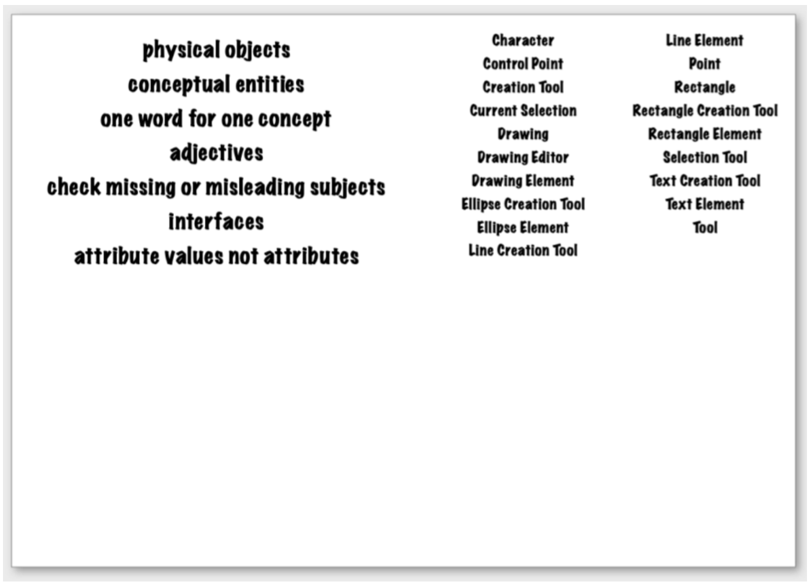
Corresponding with Responsibility Driven Design technique we initially started decomposing the battleship system according to the objects the system is supposed to manipulate. The initial exploration included finding all the relevant classes in our system, determining the responsibilities of each class and how the objects collaborate with each other to fulfill their responsibilities.

Requirement Specification
Firstly, we answered the requirements specification like what are the goals of the system being designed, its expected inputs and the desired response. In our case we found that the goal is to implement a Tutto game with two to four players within the terminal, which receives the input from the user, to determine what he or she wants to do within their turn.

Finding Classes
After answering the initial questions, we moved on by finding the nouns within Appendix A & B. Those nouns were potential classes we could use in our implementation of the Tutto game. We highlighted the nouns with a different color and wrote them down as a list to have an overview of the candidate classes. After that step we moved on to refining a list of candidate classes, which can be seen below:

player	user
computer	card
strategic decision	type
turn	terminal
game	dice
program	rules
points	turn
current score	scores of all the players
characters	alphabetical order

As mentioned in the lecture, we used the guidelines and categorized all of the nouns by determining whether they were physical objects, conceptual entities or even an adjective.



physical objects
conceptual entities
one word for one concept
adjectives
check missing or misleading subjects
interfaces
attribute values not attributes

Character
Control Point
Creation Tool
Current Selection
Drawing
Drawing Editor
Drawing Element
Ellipse Creation Tool
Ellipse Element
Line Creation Tool

Line Element
Point
Rectangle
Rectangle Creation Tool
Rectangle Element
Selection Tool
Text Creation Tool
Text Element
Tool

For that purpose we coloured the list from before with a specific color code. We used the color blue for physical objects, orange for conceptual entities, purple for interfaces, green for “Attribute Values and not Attributes” and we came up with the following edited list:

player
computer
strategic decision
turn
game
program
points
current score
characters

user
card
type
terminal
dice
rules
scores of all the players
alphabetical order

Class Selection Rational

As stated in the class, we used the class selection rationale to further elaborate on our list of nouns and to make our design for our implementation more comprehensible. We tried to use only one word for the same concept, be wary of adjectives and sentences with missing or misleading subjects. We also tried to model our categories, interfaces to the system and values of attributes instead of attributes themselves.

Physical Objects:

- player
- game
- user
- card

Conceptual Entities:

- game
- rules
- points
- score of all the players

One word for one concept:

- strategic decision → strategy
- user → player
- current score → score
- program → game

Adjectives:

Check missing or misleading subjects:

Interfaces:

- Terminal

Attribute Values and not Attributes:

- type
- alphabetical order
- points

Our preliminary analysis yielded the following candidates, but we also had in mind that our list might change over the design process.

player	game	strategy	card	cardDeck
score	dice	diceSet	turn	

CRC Sessions

Class: player	
Superclass(es): -	
Subclasses: -	
Stores the name of the player as an attribute	game
Stores the points of a player as an attribute	points

Class: game	
Superclass(es): -	
Subclasses: -	
Implying the gameplay	turn
Implying the user interface	player

Class: strategy	
Superclass(es): -	
Subclasses: -	
Storing all the rules of the game	Dice, card
Storing the different points for the dice combinations	player

Class: Card	
Superclass(es): -	
Subclasses: CardDeck	
Stores the cardtype	turn
Stores the effect it has on the points of a player	player

Class: CardDeck	
Superclass(es): Card	
Subclasses: -	
Stores all the cards within the game	Card
Shuffles itself	Card
Draw a Card	Player

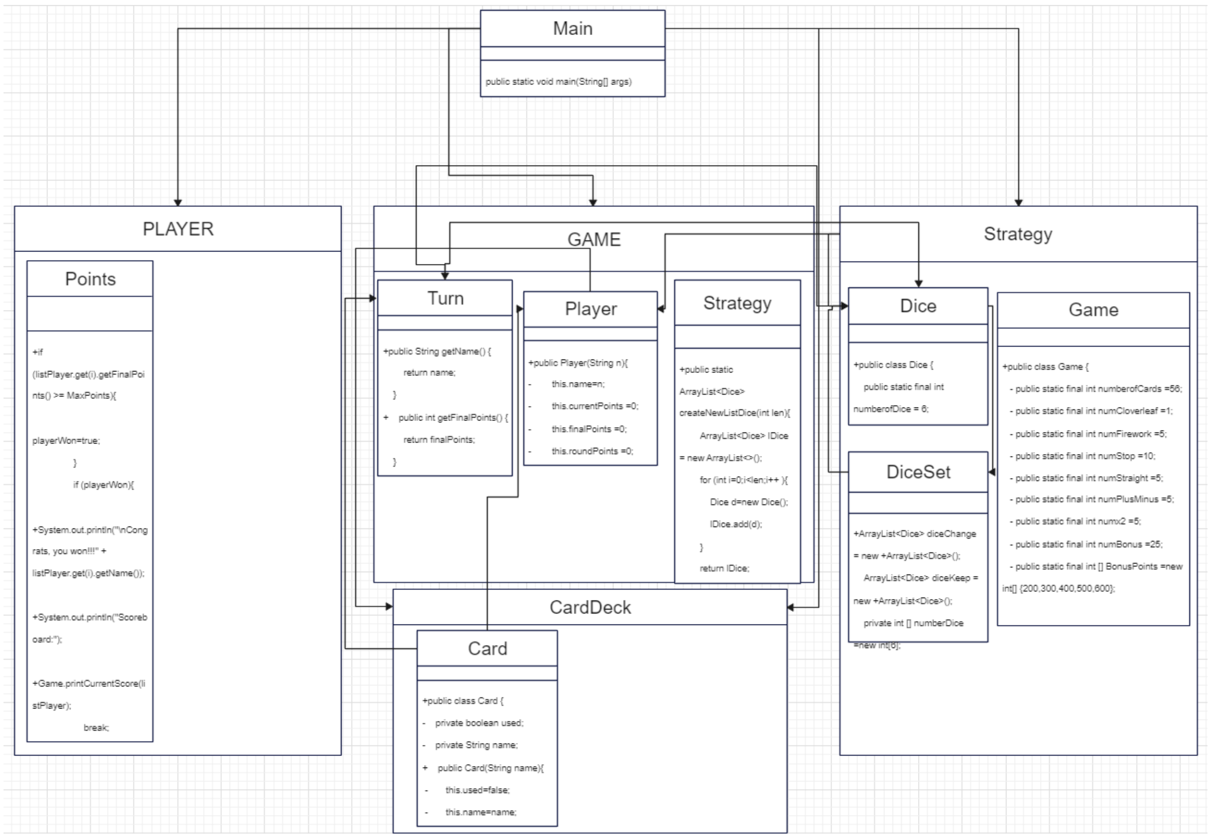
Class: Dice	
Superclass(es): -	
Subclasses: -	
Rolling the dice and Store the value in DiceSet	DiceSet
When it’s the opponent’s turn, the dices are rolled again	turn
Forming user’s strategy based on dice combinations	strategy

Class: DiceSet	
Superclass(es): dice	
Subclasses: -	
Storing the values of each each dice rolled for a set of dice	Dice

Class: Turn	
Superclass(es): -	
Subclasses: -	
When it’s the opponent’s turn, the cards are dealt again	Card
Stores the dices rolled for each round	Dice

2. Following the Responsibility Driven Design, describe the main classes you designed to be your project in terms of responsibilities and collaborations; also draw their class diagram.

Initial Class Diagram:



```
public static void main(String[] args)
```

```
+if
(listPlayer.get(i).getFinalPoi
nts() >= MaxPoints){

playerWon=true;
            }
            if (playerWon){

+System.out.println("\nCong
rats, you won!!!" +
listPlayer.get(i).getName());

+System.out.println("Scoreb
oard:");

+Game.printCurrentScore(li
stPlayer);
                break;
```

```
+public String getName() {
     return name;
  }
+   public int getFinalPoints() {
     return finalPoints;
  }
```

```
+public Player(String n){
-       this.name=n;
-       this.currentPoints =0;
-       this.finalPoints =0;
-       this.roundPoints =0;
```

```
+public static
ArrayList<Dice>
createNewListDice(int len){
     ArrayList<Dice> lDice
= new ArrayList<>();
     for (int i=0;i<len;i++ ){
        Dice d=new Dice();
        lDice.add(d);
     }
     return lDice;
```

```
+public class Card {
-   private boolean used;
-   private String name;
+   public Card(String name){
-      this.used=false;
-      this.name=name;
```

```
+public class Dice {
   public static final int
numberofDice = 6;
```

```
+ArrayList<Dice> diceChange
= new +ArrayList<Dice>();
   ArrayList<Dice> diceKeep =
new +ArrayList<Dice>();
   private int [] numberDice
=new int[6];
```

```
+public class Game {
  - public static final int numberofCards =56;
  - public static final int numCloverleaf =1;
  - public static final int numFirework =5;
  - public static final int numStop =10;
  - public static final int numStraight =5;
  - public static final int numPlusMinus =5;
  - public static final int numx2 =5;
  - public static final int numBonus =25;
  - public static final int [] BonusPoints =new
int[] {200,300,400,500,600};
```

After conducting the CRC Session we noticed that our initial design was quite complicated to play and we decided to rethink the structure of the classes, the responsibilities and the collaborators.

Identifying Responsibilities

Each class should exist because of an implied responsibility. To identify the responsibilities of each class, we highlighted the verbs in the appendix with a blue color. By analyzing those verbs, we can gather more information about the interconnection of the classes. By assigning a responsibility to the information expert, which is a class that has the information necessary to fulfill the responsibility, we can determine the classes which might act as collaborators in our system. This is a basic guiding principle of object design. It expresses the common "intuition" that objects do things related to the information they have.

Each class is interconnected and exchanges information with each other. An error in any class will affect all or part of the code and cause the code to run incorrectly. In our case the responsibilities of our classes look as follows:

Main: The main class also stores the game logic and initiates the whole game.

GameMaterial (Package):

Card: This class stores the information of the name of the card and whether it has already been used or not.
Dice: This class allows one to roll the dices and stores the information of how many dices are used within the game.

CardType (Package within GameMaterial):

Bonus: implies the rules for the Bonus card, while extending the class card. If a player achieves a TUtto, he gets the bonus points written on the card.
Cloverleaf: This class implies the rules for the Cloverleaf Card. The player needs to try to accomplish a Tutto. In case of a Tutto the player wins the game immediately, if not he doesn't score any points at all during his or her turn.
Fireworks: This class implies the rule for the Firework card, while extending the Card class. The turn ends when the player rolls a Null.
PlusMinus: Extending the class Card and implying the rules for this cardType. The player gets 1000 points if s/he achieves to play a Tutto and the leading player gets 1000 deducted. If the Tutto is not reached, the player doesn't score any points at all.
Stop: This class extends the class card and immediately stops the turn of the player without rolling any dice at all.
Straight: Extends the class Card and asks the player to achieve a Straight (1,2,3,4,5,6). Each time at least one dice needs to be kept. If the player manages to do so he or she gets 2000 points, if not null points.
x2: If a Tutto is achieved, all the points rolled so far get doubled. This class implies the rule for this card and extends the class Card.

Gameplay (Package):

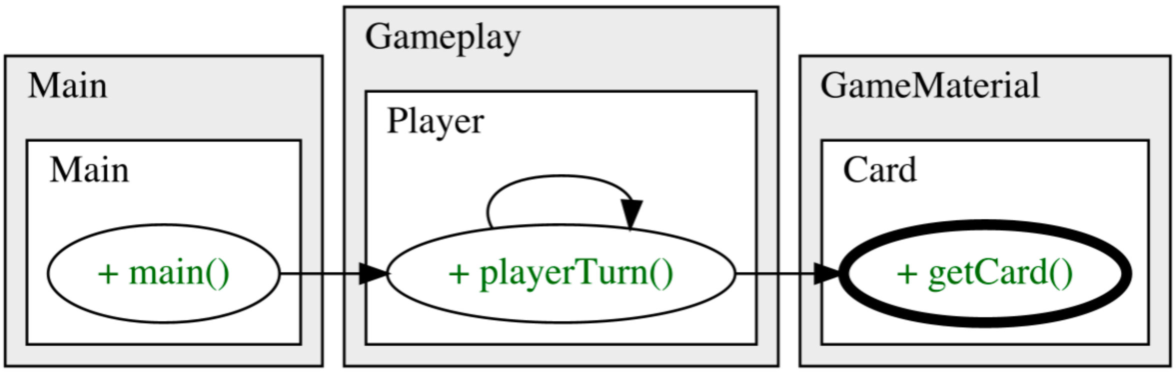
Game: This class stores the gameplay of the game and hereby the whole game logic.
Player: This class stores the information of a player and how many points he or she can roll within his or her turn. Depending on the card drawn before rolling a dice.

Finding Collaborations

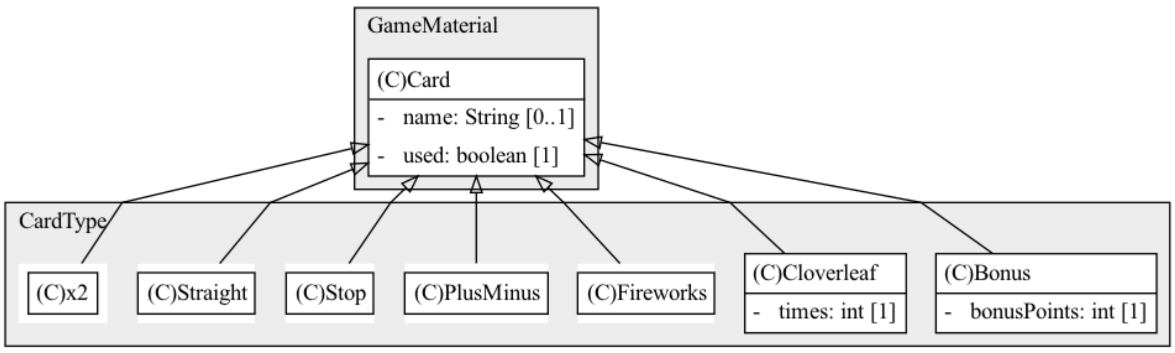
Since not every class can fulfill their responsibility by itself, it is necessary to have certain collaborators, which provide the needed information to the class in order to make the code work properly.

In order to find those collaborations, it is necessary to determine what the classes know in specific and from where they source their information or results. Classes that do not interact with each other should be discarded. In the following paragraphs we shed light on the information each class contains and how they could be implemented in our battleship game.

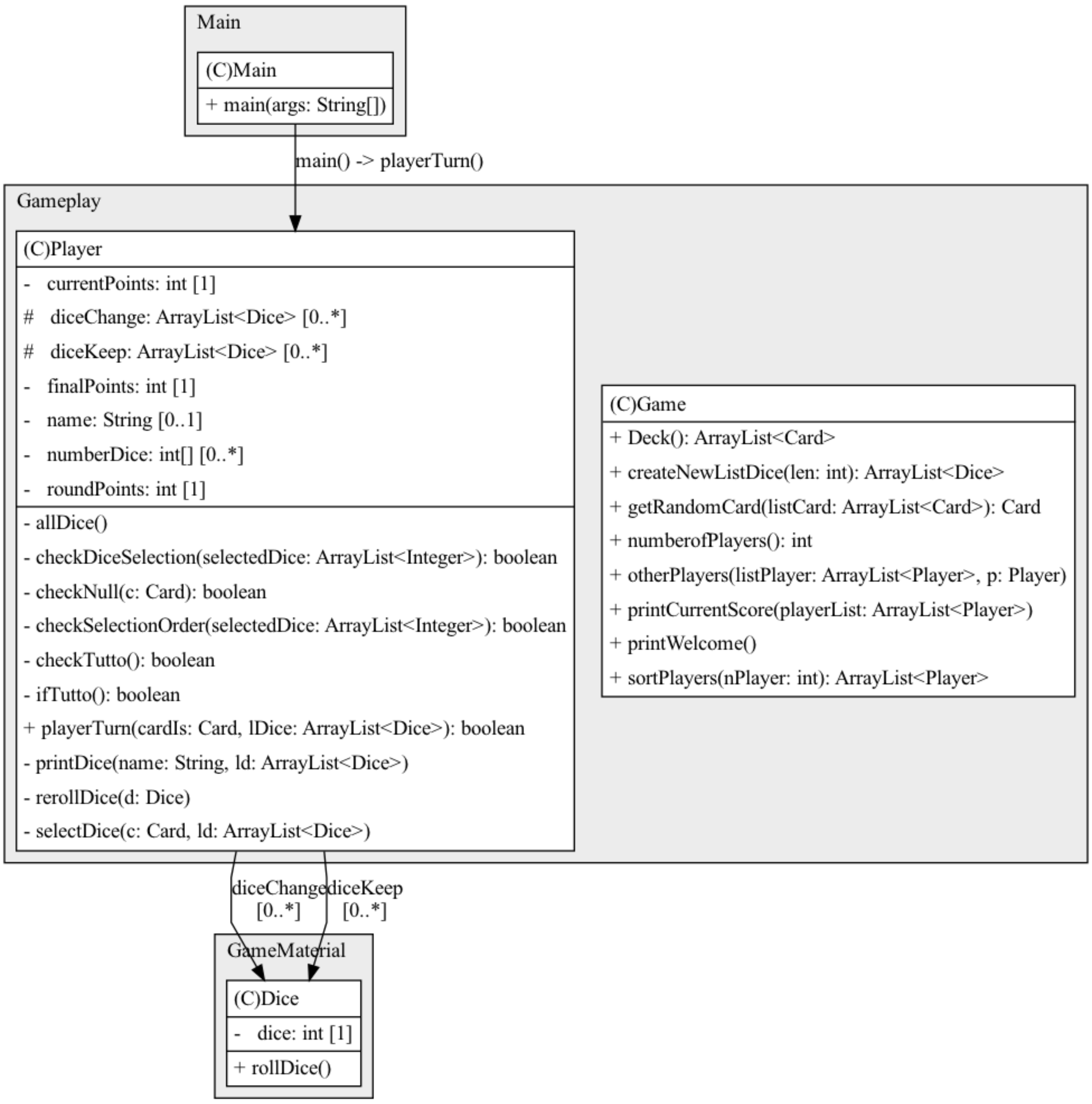
In our case all of the cardTypes cards get instantiated as soon as a card is drawn within the main class, which stores the game logic. Depending on what card is drawn, the different rules of the cards are applied. The getCard method within the Class card always gets called if a new turn starts from the player within the Player class. This turn is called from the main class.



Structuring Inheritance Hierarchies



As we can see in the upper figure, each class within the CardType package extends and inherits the fields from the Card Class within the GameMaterial package according to the abstract factory design pattern. It is one of the well-known Gang of Four creational design patterns and is used to make families of items that are made to function as a unit. You don't define the actual classes of the objects to construct when using the abstract factory design; instead, you give an interface to create families of related or dependent items. From the perspective of the client, this implies that a client is able to establish a family of related objects without being aware of the object definitions or the names of their specific classes.



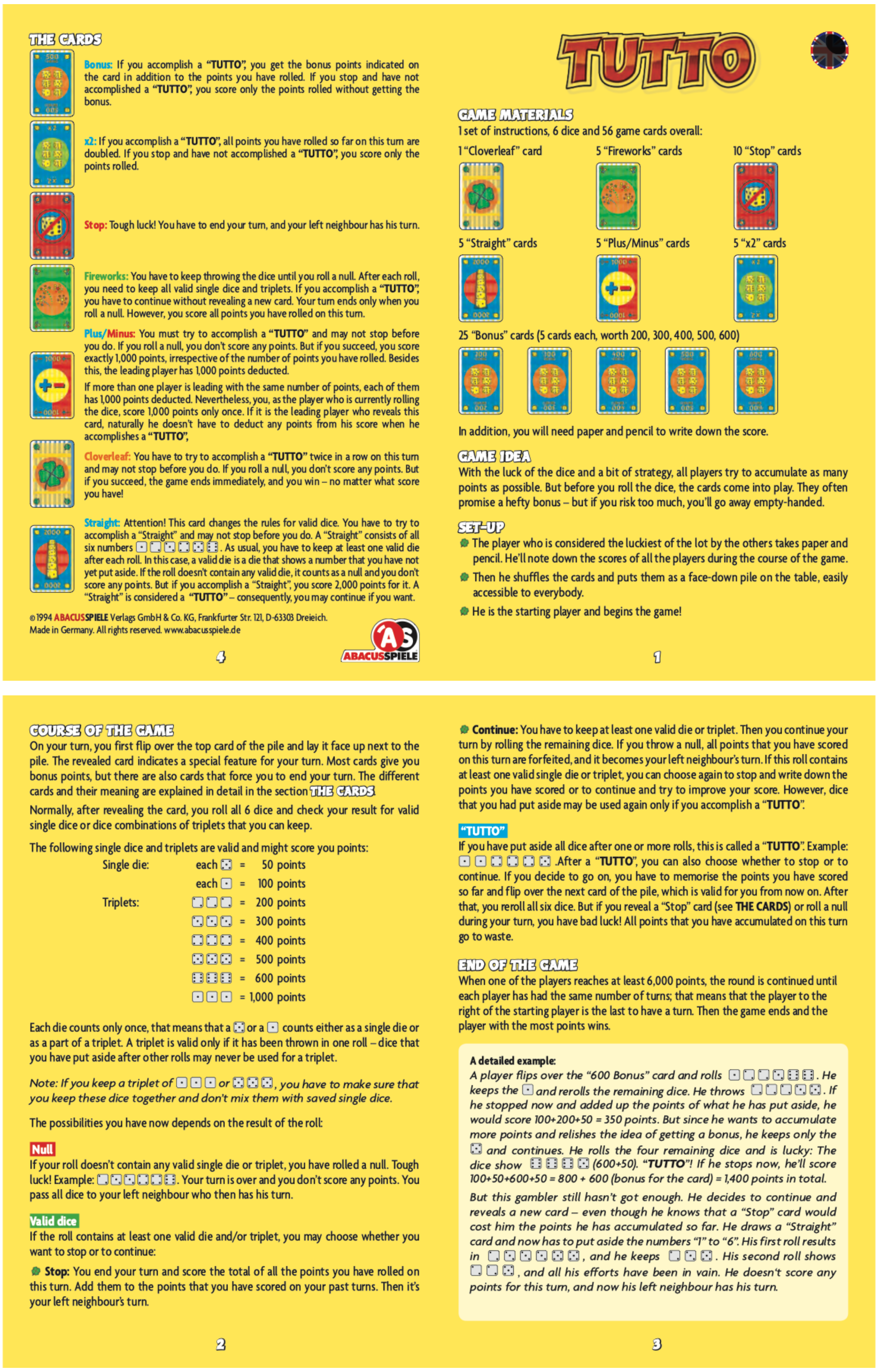
Main
(C)Main
+ main(args: String[])
main() -> playerTurn()
Gameplay
(C)Player
- currentPoints: int [1]
# diceChange: ArrayList<Dice> [0..*]
# diceKeep: ArrayList<Dice> [0..*]
- finalPoints: int [1]
- name: String [0..1]
- numberDice: int[] [0..*]
- roundPoints: int [1]
- allDice()
- checkDiceSelection(selectedDice: ArrayList<Integer>): boolean
- checkNull(c: Card): boolean
- checkSelectionOrder(selectedDice: ArrayList<Integer>): boolean
- checkTutto(): boolean
- ifTutto(): boolean
+ playerTurn(cardIs: Card, lDice: ArrayList<Dice>): boolean
- printDice(name: String, ld: ArrayList<Dice>)
- rerollDice(d: Dice)
- selectDice(c: Card, ld: ArrayList<Dice>)
(C)Game
+ Deck(): ArrayList<Card>
+ createNewListDice(len: int): ArrayList<Dice>
+ getRandomCard(listCard: ArrayList<Card>): Card
+ numberofPlayers(): int
+ otherPlayers(listPlayer: ArrayList<Player>, p: Player)
+ printCurrentScore(playerList: ArrayList<Player>)
+ printWelcome()
+ sortPlayers(nPlayer: int): ArrayList<Player>
diceChange [0..*]
diceKeep [0..*]
GameMaterial
(C)Dice
- dice: int [1]
+ rollDice()

Appendix A:

In this implementation of the game, two up to four players can play the game. The computer takes care of the logistics of the game (turns, dice rolling, keeping the score, etc.), but the players have to take their strategic decision at each turn.
When the game starts, the program asks how many players want to play; afterwards it asks for their names. The computer also asks the points that are necessary to win the game (e.g., 6’000). The players then take turns following an alphabetical order based on their names.
At the start of each turn, the user can: Roll the dice or Display the current scores, by inputing either one of the underlined characters. Once the user selects R, a card is automatically drawn and its type shown on the terminal. Subsequently, the computer rolls the dice, shows the results to the user, and, based on the rules, either inform the user that their turn is finished or asks them whether they want to Roll the dice or End their turn. At the end of each turn, the current scores are updated. At the end of every turn, the game checks whether a user reaches the points necessary to win the game.
In this case, this user is declared as the winner and the scores of all the other players are also shown; the game ends. Otherwise, the turn passes to the next player in alphabetical order.

Appendix B:

Please refer to the official rules by ABACUSSPIELE, which are available here and are reported in the following page for your convenience. All the rights are with ABACUSSPIELE.



TUTTO

GAME MATERIALS

1 set of instructions, 6 dice and 56 game cards overall:

1 "Cloverleaf" card

5 "Fireworks" cards

10 "Stop" cards

5 "Straight" cards

5 "Plus/Minus" cards

5 "x2" cards

25 "Bonus" cards (5 cards each, worth 200, 300, 400, 500, 600)

In addition, you will need paper and pencil to write down the score.

GAME IDEA

With the luck of the dice and a bit of strategy, all players try to accumulate as many points as possible. But before you roll the dice, the cards come into play. They often promise a hefty bonus – but if you risk too much, you'll go away empty-handed.

SET-UP

- The player who is considered the luckiest of the lot by the others takes paper and pencil. He'll note down the scores of all the players during the course of the game.
- Then he shuffles the cards and puts them as a face-down pile on the table, easily accessible to everybody.
- He is the starting player and begins the game!

COURSE OF THE GAME

On your turn, you first flip over the top card of the pile and lay it face up next to the pile. The revealed card indicates a special feature for your turn. Most cards give you bonus points, but there are also cards that force you to end your turn. The different cards and their meaning are explained in detail in the section THE CARDS.

Normally, after revealing the card, you roll all 6 dice and check your result for valid single dice or dice combinations of triplets that you can keep.

The following single dice and triplets are valid and might score you points:

Single die:	each ⚄	=	50 points
	each ⚀	=	100 points
Triplets:	⚁⚁⚁	=	200 points
	⚂⚂⚂	=	300 points
	⚃⚃⚃	=	400 points
	⚄⚄⚄	=	500 points
	⚅⚅⚅	=	600 points
	⚀⚀⚀	=	1,000 points

Each die counts only once, that means that a ⚄ or a ⚀ counts either as a single die or as a part of a triplet. A triplet is valid only if it has been thrown in one roll – dice that you have put aside after other rolls may never be used for a triplet.

Note: If you keep a triplet of ⚀⚀⚀ or ⚄⚄⚄, you have to make sure that you keep these dice together and don't mix them with saved single dice.

The possibilities you have now depends on the result of the roll:

Null

If your roll doesn't contain any valid single die or triplet, you have rolled a null. Tough luck! Example: ⚁⚂⚂⚃⚃⚅. Your turn is over and you don't score any points. You pass all dice to your left neighbour who then has his turn.

Valid dice

If the roll contains at least one valid die and/or triplet, you may choose whether you want to stop or to continue:

- **Stop:** You end your turn and score the total of all the points you have rolled on this turn. Add them to the points that you have scored on your past turns. Then it's your left neighbour's turn.

- **Continue:** You have to keep at least one valid die or triplet. Then you continue your turn by rolling the remaining dice. If you throw a null, all points that you have scored on this turn are forfeited, and it becomes your left neighbour's turn. If this roll contains at least one valid single die or triplet, you can choose again to stop and write down the points you have scored or to continue and try to improve your score. However, dice that you had put aside may be used again only if you accomplish a "**TUTTO**".

"TUTTO"

If you have put aside all dice after one or more rolls, this is called a "**TUTTO**". Example: ⚀ ⚀ ⚃ ⚃ ⚃ ⚄ .After a "**TUTTO**", you can also choose whether to stop or to continue. If you decide to go on, you have to memorise the points you have scored so far and flip over the next card of the pile, which is valid for you from now on. After that, you reroll all six dice. But if you reveal a "Stop" card (see **THE CARDS**) or roll a null during your turn, you have bad luck! All points that you have accumulated on this turn go to waste.

END OF THE GAME

When one of the players reaches at least 6,000 points, the round is continued until each player has had the same number of turns; that means that the player to the right of the starting player is the last to have a turn. Then the game ends and the player with the most points wins.

A detailed example:
*A player flips over the "600 Bonus" card and rolls ⚀⚁⚁⚂⚅⚅. He keeps the ⚀ and rerolls the remaining dice. He throws ⚁⚁⚁⚂⚄. If he stopped now and added up the points of what he has put aside, he would score 100+200+50 = 350 points. But since he wants to accumulate more points and relishes the idea of getting a bonus, he keeps only the ⚄ and continues. He rolls the four remaining dice and is lucky: The dice show ⚅⚅⚅⚄ (600+50). "**TUTTO**"! If he stops now, he'll score 100+50+600+50 = 800 + 600 (bonus for the card) = 1,400 points in total.*

But this gambler still hasn't got enough. He decides to continue and reveals a new card – even though he knows that a "Stop" card would cost him the points he has accumulated so far. He draws a "Straight" card and now has to put aside the numbers "1" to "6". His first roll results in ⚁⚂⚂⚂⚄⚄, and he keeps ⚁⚂⚄. His second roll shows ⚁⚁⚄, and all his efforts have been in vain. He doesn't score any points for this turn, and now his left neighbour has his turn.

THE CARDS

Bonus: If you accomplish a "**TUTTO**", you get the bonus points indicated on the card in addition to the points you have rolled. If you stop and have not accomplished a "**TUTTO**", you score only the points rolled without getting the bonus.

x2: If you accomplish a "**TUTTO**", all points you have rolled so far on this turn are doubled. If you stop and have not accomplished a "**TUTTO**", you score only the points rolled.

Stop: Tough luck! You have to end your turn, and your left neighbour has his turn.

Fireworks: You have to keep throwing the dice until you roll a null. After each roll, you need to keep all valid single dice and triplets. If you accomplish a "**TUTTO**", you have to continue without revealing a new card. Your turn ends only when you roll a null. However, you score all points you have rolled on this turn.

Plus/Minus: You must try to accomplish a "**TUTTO**" and may not stop before you do. If you roll a null, you don't score any points. But if you succeed, you score exactly 1,000 points, irrespective of the number of points you have rolled. Besides this, the leading player has 1,000 points deducted.

If more than one player is leading with the same number of points, each of them has 1,000 points deducted. Nevertheless, you, as the player who is currently rolling the dice, score 1,000 points only once. If it is the leading player who reveals this card, naturally he doesn't have to deduct any points from his score when he accomplishes a "**TUTTO**",

Cloverleaf: You have to try to accomplish a "**TUTTO**" twice in a row on this turn and may not stop before you do. If you roll a null, you don't score any points. But if you succeed, the game ends immediately, and you win – no matter what score you have!

Straight: Attention! This card changes the rules for valid dice. You have to try to accomplish a "Straight" and may not stop before you do. A "Straight" consists of all six numbers ⚀⚁⚂⚃⚄⚅. As usual, you have to keep at least one valid die after each roll. In this case, a valid die is a die that shows a number that you have not yet put aside. If the roll doesn't contain any valid die, it counts as a null and you don't score any points. But if you accomplish a "Straight", you score 2,000 points for it. A "Straight" is considered a "**TUTTO**" – consequently, you may continue if you want.