**Exercise 2: A Terminal-based Tutto Game - Implementation**

For the implementation of the Tutto game we followed a similar approach as in the last exercise. We created as many classes as necessary and as few as possible in order to avoid God classes but still keep a manageable number of classes as well as a slim design. The Classes are grouped in four main categories called “folders” for the sake of simplicity. These are “card”, “dice”, “game” and “player”. Each containing multiple classes and an interface in the folder card. This answer sheet is structured the same way as the folders (in bold letters) are structured to be more understandable.

Any Methods mentioned in this answer sheet will be using a simplified presentation without it´s parameters. For example a method add(int i, int j), will simply be represented by add().

**Folder Card**

The folder card contains all the different classes regarding the cards of the Tutto game. The most fundamental class is the card class. Objects of the class card have two attributes. A CardType and an ID, both of which are set to private and final to be in accordance with the Design Concept of encapsulation and the limitation of accessibility. Additionally, all the cards are created as flyweight objects to make sure that they are unique and are stored in a List named CARDS, which again is private and final. The cardTypes are stored as an enum Type with their respective quantity. For example, PlusMinus has the assigned quantity 5.

For each of the cards, that have a different effect on the game and change the rules, there exists an own class. However, to make the design more efficient we created an interface name CardMode, which is implemented by each of the card classes and holds methods for the classes of cards. Before diving deeper into the “specialised” card classes, there are two more classes in the card “folder” that need to be introduced. The class Deck models a deck of the 56 Tutto cards. Objects of the Type deck have two private attributes, a CardStack and a count. The implemented methods are shuffle, draw and push (pushes a card on top of the deck). Each of these methods are public in order to ease the access for client code. The Class CardStack is responsible for the creation of card Stack objects and the necessary methods. Once again to be in Line with the Design Pattern of encapsulation and to limit the mutability, the attributes are set to private.

Before covering the individual “speciality” Card Classes we would like to introduce the method play() in the interface Cardmode. Since said interface is implemented in each of the “speciality” Card classes each of the “speciality” Card Classes inherits said method. Each of the “speciality” Card Classes overrides said method in accordance with the composite Design Pattern to suite their respective needs and to adapt the game rules depending on the effect the drawn card has on the game. For example, the Class “stop” overrides the play method to set all points to zero. The other methods of the interface are being overwritten equally.

In total the folder card contains 7 “speciality” Card Classes. Namely: “Bonus”, “Cloverleaf”, “Fireworks”, “PlusMinus, “Stop”, “Straight” and “X2”.

As previously explained, any of these Classes overrides and modifies the methods of the Interface according to their impact onto the gameplay. Cards that make a Tutto impossible have always a return value of false for the isTutto() method.

The Attributes of the Speciality Card Classes are kept, except for the isTutto Boolean, private and when necessary, final. This is once again to be in accordance with the learnt design patterns.

We will provide a detail view of the aforementioned Classes and briefly explain, what they exactly do. We will however not individually cover each method for the sake of briefness.

**Class Bonus**

The Class “Bonus” Overrides the Play() method so that after each turn the “bonus” value is added to the points achieved by the player. Since a Tutto is possible while a bonus card is drawn, the variable isTutto is mutable and returned by the boolean method isTutto().

**Class Cloverleaf**

The Cloverleaf Class does not have a public isTutto Boolean but a private aTutto attribute of an integer value. The Class Cloverleaf modifies the play() function so that as soon as the aTutto “counter” reaches 2 it sets for a DiceComp (will be covered later) Object the cloverleaf attribute to true by calling the setCloverleaf() function. This leads to an instant win by the Player

**Class Fireworks**

The Class fireworks overwrites the play() function so that if a Tutto is achieved, all dice are cleared and that player restarts with 6 dice and no new card.

The isTutto() function is overwritten to always return a the value false.

**Class PlusMinus**

The PlusMinus Class overrides the methods of the interface similarly to the previous examples.

**Class Stop**

The Stop card overrides the play() method to set the points to zero. The default return value for the isTutto() method is set to false since a Tutto is not achievable if a Stop Card is dawn.

**Class Straight**

The Straight Class differs from the other classes since it has an additional Method named printStraight() which prints the values of the dice in the StraightList. The isTutto attribute is set to true if a straight is achieved and the points are added. This is once again done by overriding the play() method and using functions from the DiceComp class.

**Class X2**

The X2 Class overrides the methods of the interface similarly to the previous examples. It doubles the amount of points if a Tutto is achieved. Therefore the isTutto() method is not set to false by default but is returning the isTutto attribute.

Throughout the entire Implementation of the Folder Card methods use Input validation to make sure that client code is up to standard. For example in the Class Cardstack the method push checks the input with following statement “assert pCard != null && !aCards.contains(pCard);”. Additionally, to input validation the Pattern Design by Contract is being used in various Classes.

**Folder Dice**

The next folder “Dice” contain three classes. Said Classes are “Dice”, “DiceComp” and “Drilling”.

**Class Dice**

Objects of the Class dice have a private attribute aNumer, where it´s value is stored. Additionally, there is a private static and final DICES variable, which is of the Class DiceComp, where all 6 dices are being added to.

In Order to be compliant with the Encapsulation Design Pattern and the restrictions of access from client code, there are multiple methods which for example return the value of a Dice, return the DiceComp DICES or the roll() method, which assigns to the aNumber attribute of a dice a random value between 1 and 6. For testing purposes a setANumber method, which allows to assign a specific value to the a Number attribute of a Dice has been implemented.

**Class DiceComp**

The Class DiecComp is one of the main Components of the Tutto game and is responsible for “checking” all moves of the player and therefore more complex. It is also responsible for the “manipulation” of Dice Values by rolling them and the fundamental steps in the game.

To limit the accessibility of the attributes of a DiceComp Object all attributes have been set to private as well as mostly to final. These Design methods from Chapter 4 on top of the Designpattern of Encapsulation leads to robust code.

Objects of the Class DiceComp have several Attributes, visible below.

private final List<Dice> aDices;  
private final List<Dice> selectedDices;  
private final List<List<Dice>> rolledDices;  
private final List<Dice> tempDices;  
private final List<Dice> aDicesWithPoints;  
private final List<Dice> straightList = Arrays.*asList*(new Dice[6]);  
private final List<Drilling> drillingList;  
private boolean plusMinusTutto = false;  
private boolean cloverleaf = false;  
private int points;

While the Constructor and some Methods are public other methods have ben set to private if there is no need for client code to access them. This step was taken to “protect” the attributes from mutability.

For the Class DiceComp we will be covering each method with a short explanation of what it´s purpose is since these methods are the fundamentals for the implementation of the Tutto game.

* The public method **roll()** rolls all the Dices in the List aDices.
* The private method **getResults()** loops over the drillingList and adds the points in addition to the rolled Dices of value 1 and 5.
* The Private Method **testPoints()** tests the Dice values checks if with any points are possible with the rolled dices.
* The Private Method **addDicesWithPoints()** adds all drillings and dices of Value 1 and 5 to the aDicesWithPoints List.
* The methods **getPoints()** and **add()** are self explanatory.
* The public Method **Split()** checks if any Dices can be chosen. It does so by checking which Cardmode is set and adapting the choosing rules according to each Cardmode. If a split is possible, it returns the value true, if not false.
* The public Method **isNull()** checks if the Dicecomposition is a Null by calling the **testPoints()** method and negating the result as return value.
* The public boolean method **isTutto()** checks if a Tutto has been achieved.
* The private boolean **hasNoPoints()** method checks if the List of Dice contain any Dices which merit points
* The public method **drillingNotValid()** checks if Triplets are not achievable with the rolled Dices
* The public method **getLength()** returns the size of aDices
* The public method **getaDices()** returns the List of Dices aDices
* The public method **isStraight()** checks if a straight is achievable, returns false if it isn´t and true if it is.
* The public method **isNoStraight()** returns true if a straight is not possible anymore with the DiceComp is self-explanatory
* The public method **getStraightList()** returns the Attribute straightList
* The public method **clear()**
* The public method **setPointsToZero()** is self-explanatory
* The public method **addBonusPoints()** is self-explanatory
* The public method **setPlusMinusTutto()** sets the booleanvalue of plusMinusTutto to true
* The public method **setPlusMinusTuttoBack()** sets the booleanvalue of plusMinusTutto to false
* The public method **plusMinusTutto()** returns the boolean value of plusMinusTutto
* The public method **setCloverleaf()** sets the booleanvalue of cloverleaf to true
* The public method **isCloverleaf()** returns the boolean value of cloverleaf

Throughout the Class certain methods use Design by Contract to make sure that client code is up to standard. All Methods are commented to indicate their use and purpose to make the code more understandable and collaboration between team members easier.

**Class Drilling**

The Class Drilling is responsible for handling Triplets. Each Object of the Class has two private and final attributes. A List of Dices and an integer value that represents the points the triplets are worth. The Constructor holds a Integerlist of the values the Triplets represent and automatically assigns the value during the Creation of a Drilling Object. The Attributes can only be accessed by client code through the public methods getPoints(), which returns the value of points and the method getaDices() which returns the List of Dices aDices. To make sure that a Drilling Object is only created if a Drilling ha indeed ben rolled, the Design by Contract Pattern has been used so that client Code is responsible for the validity of the input.

**Folder Game**

**Class Game**

The Class Game is the main Method and starts the game by creating a new Object of the Class Gamemodel called Tutto and then calling the function start() on said model.

**Class GameModel**

The Class GameModel is next to the DiceComp the most extensive class. It functions as an intermediary between the Input of the Players and the DiceComp “running” the game in the background. The Class GameModel is responsible for the course of the game, the handling of the Players, the alteration of turns between the Players the drawing of Cards and pushing them back on the stack. Objects of the Class Gamemodel have 6 private and except for the playerList final attributes. This was once again done, to restrict access of client code on an Objects attributes and to follow the Design Pattern of Encapsulation. The Method Tutto() which “runs” the game is set to private as well.

**Class Input**

The Class Input is responsible for handling all the Input by the players. Objects of the Class Input have one final and private Attribute.

One of the main Responsibilities next to handling the input is the in Chapter 2.8 covered Input validation. The Input by the Human players is checked thoroughly by various statements in order to validate the conformity of said input, so that it can be handled by the GameModel Class.

**Class InputAsker**

The Class InputAsker has one private and final Attribute aScanner and 2 public Methods.

**Folder Player**

**Class Player**

The Class Player is responsible for the creation of Player objects. Each Player Object has a private score and a private and final name. Methods that can be applied on a Player Object are getScore(), setScore() and getName() which return the asked value/string or set a score for the player. Additionally, the Class holds two methods to either compare two player names or two player scores.