

**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

computer

strategic decision

turn

game

program

points

current score

characters

user

card

type

terminal

dice

rules

turn

scores of all the players

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.

Table

Description automatically generated with medium confidence

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:

Diagram, schematic

Description automatically generated

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.

Diagram

Description automatically generated

**Structuring Inheritance Hierarchies**

Diagram

Description automatically generated

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.

Table

Description automatically generated

**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.

Text

Description automatically generated with medium confidence

