Exercise_02

Ramona Walker, Dominik Johann Arnold, Mark Woolley, Otto Buck

November 2022

Description of implementation decisions

- **Deck**:
  We first wanted to use a deck together with a discard pile, taking a card out of the deck each time it is drawn as one would do it with a physical deck. However we realized, that we have less space for errors if we only iterate through the deck, without actually discarding the cards. Like this we can be sure that a deck will always contain the same cards and only the order of the cards can change.

  We could copy some ideas of the deck used in the book, like giving the card enum types. However we noticed, that these decks cannot be made exactely the same way: For example the cards in our deck are not unique. Therefore it is not possible to make use of the Flyweight design pattern here.

- **Ruleset**:
  We decided to make an abstract class Ruleset for all the different rules for the cards. The reason is, that most card rules have similar properties: they often give a bonus when Tutto is reached, multiple cards prevent you from ending your move or drawing new cards. Also we thought that it is not necessary to define the rules directly on *Card*, to provide better encapsulation.

- **DefaultParser**:
  This time we decided to implement an input parser for our program. This way we can assume that all the input we receive is correct which allowed us to use proper preconditions. We do not again have to validate our input but could use design by contract instead.

- **Die**
  For the values of the dice we decided to make an enum type for the same reasons Martin P. Robillard used an enum for the cardrank. With the enum (instead of integer) we don't have a problem with integers higher then expected or similar and we don't need to additionally validate the value of the dice.

- **DiceSet**
  We used the Flyweight pattern for DiceSet since we realized, that every player can play with the same set and we only need (and want) to have one in the game.

- **DiceCombo**
  Since the number of possible DiceCombos a player can take out is very restricted (12) we decided to make an enum type and let the player chose from a list which combo he wants to take out instead of letting him type it in freely. This reduces the valid console inputs of the player to a minimum and thereby make it easier for the input parser.

- **DeckSpec**
  We used the Builder design pattern (not from the book) in DeckSpec. This was especially handy when it came to writing tests and debugging, since we could quickly build a deck with only the cards desired in it.

- **general implementation decisions:**
  There were a few things that were not quite clear to us in the rules given by Abacus-Spiele:

  - In the rules of Cloverleaf it doesn't say, that you don't have to draw a new card, when you achieve Tutto (as it is said in Fireworks). But we thought that it makes more sense to implement it that way.
  - When Tutto is achieved with the plus/minus-card, we decided that you cannot drop below zero points.
  - With the x2 card, we decided to only double the points rolled while this card was open.

We realized that we flipped the order of rolling the dice and asking weather one wants to continue. This doesn't change the chances of the players however, since when you roll a null you have lost and you cannot end your turn (without losing the points). If you want to stop before the roll you can say so before you took all the combinations out, and all the valid combinations will be taken out automatically.

- You can play the game normally by going in *Main*, setting playDebug = false and uncommenting the line:
  DeckSpec ds = new DeckSpec.DeckSpecBuilder().setDefault().build();

  If you keep playDebug = true, you will always roll ones, making it easier to test cards like cloverleaf. You can build a costumized deck by inserting:
  $DeckSpecds = newDeckSpec.DeckSpecBuilder().set < whatever\_you\_want\_in\_it(amount) > .build();$