Exercise_02

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

November 2022

Description of implementation decisions

- **GameUtils**:
  As we were developing the classes, we realized that there are some global methods used by all classes that do not belong to a specific class per se. Those are convenience methods like generateCoordinatesFromStartEnd which take two Coordinate objects as input and produce a list of all the coordinates contained between those two Coordinate objects. It could be argued that those functionalities should belong to the respective objects using them. For example, the input validation should belong to GameMaster, who is responsible for adhering to the rules of the game. Or it could belong to the Player itself, making the Player responsible for checking herself or her opponent. In our opinion, the design looks less cluttered with a GameUtils class, as it serves as an addition to the GameMaster having all the methods the GameMaster needs to make sure the rules are adhered. Furthermore, the GameUtils class can contain global final rules of the game like the gamesize and parameters for the PlayerComputer random number generation.

- **Coordinate**:
  We decided to not fall for the PRIMITIVE OBSESSION antipattern and give the coordinates of the grid an own representation. This facilitates the usage among different classes. This encapsulation makes it possible to change the internal representation, as long as the basic functions getRow and getCol are supplied. It makes it possible to implement different methods of comparing Coordinate objects for example. Towards the end of the assignment, we realized that this class could benefit from the FLYWEIGHT pattern

- **User Input**:
  We decided to use input validation within the functions which accept human user input (callShot and placeFleet) in favor of contract (for example do not allow invalid Coordinate to be created in the constructor). The reason is that we don't know yet how to properly implement an error handling. So given the user inputs a wrong Coordinate (invalid, wrong format), the error would be thrown but not handled and the game ends. With input validation we tried to catch all possible errors a user could commit to make sure the subsequent objects receive valid Coordinate objects / list of Coordinate objects. Subsequent objects rely on that validation and assume every input from the user they receive was properly checked.

- **Player**:
  We decided against a Player interface in favor of an abstract class Player. Initially, we implemented an interface Player, but realized that the two implementations PlayerHuman and PlayerComputer only differ in the methods callShot and placeFleet, but have identical methods for receiveShot, getBoatTypeString, isHuman and hasLost. So, an abstract class with both abstract and non-abstract methods seemed like a sensible way to go. For further improvements, the PlayerComputer class would surely benefit from a STRATEGY pattern implementing different playing and placing strategies.

- **Fleet & FleetSpec**:
  The Fleet makes use of the ITERATOR pattern, as it must be able to provide an iterable collection of Boat objects, without revealing the concrete implementation. It serves as an intermediary between Player and Boat and delegates most of the heavy lifting to the Boat class. We use the FleetSpec enum type to make debugging easier by enabling different amounts of Boats. Strictly speaking, it's not necessary and different constructors in Fleet could serve the same purpose.

- **Boat & BoatType**:
  BoatType is an enum type which specifies which boats are available in the game and directly defines each BoatType in terms of length and nomenclature. The Boat keeps track of its responsibilities and communicates directly with the Fleet class.

- **Ocean Grid:**
  We decided to leave the "X"-marks even when a ship sinks in the ocean grid instead of changing the notation to the first letter of the boat to make it clear which blocks have been shot at.