Ship of Fools - Requirement

> 1. https://bth.instructure.com/courses/4644/assignments/38478
> 2. https://bth.instructure.com/courses/4644/pages/ship-of-fools

1. Description

Ship of Fools is a simple classic dice game. It is played with five *standard 6-faced dice* by two or more players.

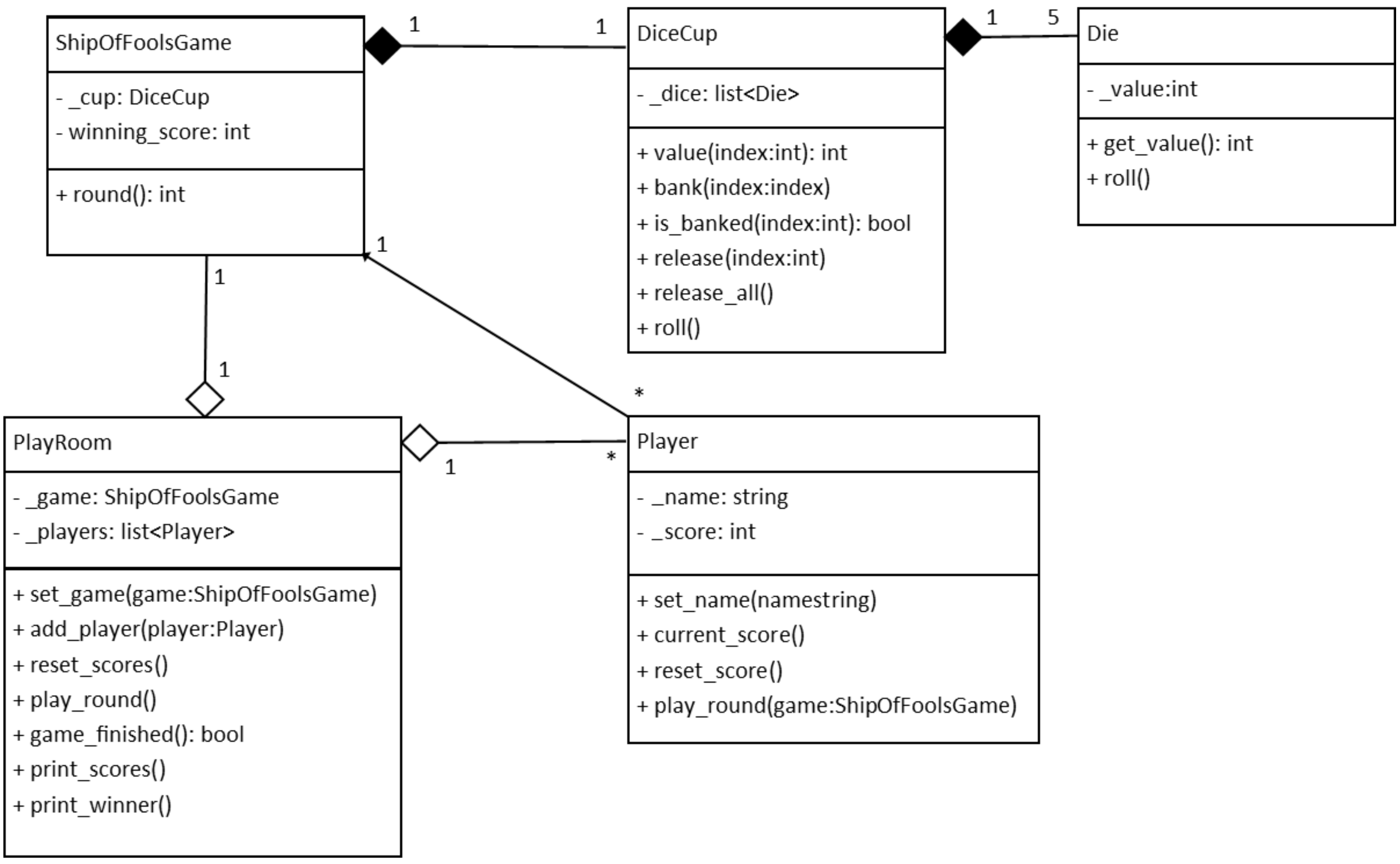
- The goal of the game is to gather a 6, a 5 and a 4 (*ship*, *captain* and *crew*) in *the mentioned order*.
- The sum of the two remaining dice (*cargo*) is preferred as high as possible. The player with the highest cargo score wins the ***round***.

Example:

- In the first round turn, if a player ***rolls*** **6 4 3 3 1** *(note we five dice at the beginning)*, the player can ***bank*** the 6 (ship), but the rest needs to be ***re-rolled*** since there is no 5.
    - In the second round turn, if the player rolls 6 5 4 4 (four dice, since *the 6 from last turn is banked*), the player can bank the 5 (captain) and the 4 (crew). The player has three choices for the remaining 6 and 4. The player can bank both and *score* 10 points, or re-roll one or two of the dice and hope for a higher score.
    - In the second round turn, if the player instead rolled 4 4 3 1, all dice needs to be re-rolled since there is no 5.

2. Model

Below is a suggestion how the game can be modeled by a set of classes:



The division of responsibility between the different classes is as follows.

- `Die` **:** Responsible for handling randomly generated integer values between 1 and 6.
- `DiceCup` : Handles five objects (dice) of class `Die` . Has the ability to bank and release dice individually. Can also roll dice that are not banked.
- `ShipOfFoolsGame` : Responsible for ***the game logic*** and has the ability to play a round of the game resulting in a score. Also has a *property* that tells *what accumulated score results in a winning state*, for example 21.
- `Player` : Responsible for the score of the individual player. Has the ability, given a game logic, play a round of a game. The gained score is accumulated in the *attribute* `score` .
- `PlayRoom` : Responsible for handling a number of players and a game. Every round the room lets each player play, and afterwards check if any player has reached the winning score.

3. Implementation

The algorithm for the *game logic* can be quite simple (pseudo code):

```
has_ship = False
has_captain = False
has_crew = False

# This will be the sum of the remaining dice, i.e., the score.
```

```
crew = 0

# Repeat three times
for round in range(3):
    roll unbanked dice
    if not has_ship and a dice is 6:
        bank it
        has_ship = True
    if has_ship and not has_captain and a dice is 5:
        # A ship but not a captain is banked
        bank it
        has_captain = True
    if has_captain and not has_crew and a dice is 4:
        # A ship and captain but not a crew is banked
        bank it
        has_crew = True

    if has_ship and has_captain and has_crew:
        # Now we got all needed dice, and can bank the ones we like to save.
        bank all unbanked dice > 3

# If we have a ship, captain and crew (sum 15),
# calculate the sum of the two remaining.
if has_ship and has_captain and has_crew:
    crew = sum of dice - 15
```

3.1 Main script

This can be the main script of the program.

```
if __name__ == "__main__":
    room = PlayRoom()
    room.set_game(ShipOfFoolsGame())
    room.add_player(Player("Ling"))
    room.add_player(Player("Chang"))
    room.reset_scores()
    while not room.game_finished():
        room.play_round()
        room.print_scores()
    room.print_winner()
```

3.2 Hints

Many methods involves to iterate over a list that is an attribute of the class. There are some general algorithms that we often want to implement when we program. Note that you can reach elements in two different ways. *Either by getting a reference to one object after another, or by index*. We start with two examples where we use the reference and then one where we use the index (which is needed in this case).

Search (Pseudo code)

```
# To set a value given through a parameter or a fixed value
value_to_find = <something>
# Set value to search for
found = False
# For each value of the attribute (presumed a list)
for obj in self.attribute_name:
    # Asks the object for the value and compare
    if obj.get_value() == value_to_find
         # Update the value
         found = True
```

Accumulate (for example adding, or concatenating e.g., a string)

```
# Set initial value
sum_of_values = 0
# For each value of the attribute (presumed a list)
for obj in self.attribute_name:
    # Asks the object for the value and add.
    sum_of_values += obj.get_value()
```

Example

If we want to check if a not banked dice has the value 4, and if so change the value of a helper variable (*has_cerew*).

```
# Set initial value
has_crew = False
# For each value of the attribute (presumed a list)
for i in range(len(self._dice)):
    if dice[i].get_value() == 4:
        dice[i].bank()
        has_crew = True
```