Lecture 7: Ship Of FoolsGame (I)

1. TDD（测试驱动开发）/DDD（领域驱动设计）

1.1 从最主要的业务逻辑开始我们的游戏设计

1.1.1 主要代码

```
class ShipOfFoolsGame:
    def __init__(self):
        self._cup = DiceCup()
        self._scores = []

    def play(self):
        for round in range(7):
            self._cup.roll()
            # 如果没有骰子可以投掷了怎么办?

            if not self.has_ship and self._cup.has_ship:
                self._cup.bank_ship()
                self._scores.append(self.ship)

            if self.has_ship:
                if not self.has_captain and self._cup.has_captain:
                    self._cup.bank_captain()
                    self._scores.append(self.captain)

                if self.has_captain and self._cup.has_crew:
                    self._cup.bank_crew()
                    self._scores.append(self.crew)

            if self.has_ship and self.has_captain and self.has_crew:
                print(f'Now please select cargo(s) from {self._cup.scores}.')
                cargo = input('Your selection is: ').strip()
                # 这里需要考虑:
                #  1) 如果用户不想选择任何一个 cargo，怎么办?
                #  2) 如果用户选择的 cargo 不在可选列表中，怎么办?

                self._cup.bank_cargo(cargo)
                self._scores.append(cargo)

```

```
34          print(f'Your final score in this round is {self.scores[-2:]} =', self
```

1.1.2 业务逻辑中的主要问题

1. 存在冗余的逻辑判断：角色是按顺序选取的，如果当前角色被选取了，则之前的所有角色都被选取了；

```
if not self.has_ship and \
   self._cup.has_ship:
     self._cup.bank_ship()
     self._scores.append(self.sh

if self.has_ship:
    if not self.has_captain and
         self._cup.has_captain:
           self._cup.bank_captai
           self._scores.append(s

    if self.has_captain and \
         not self.has_crew and \
         self._cup.has_crew:
           self._cup.bank_crew()
           self._scores.append(s

if self.has_ship and \
     self.has_captain and \
     self.has_crew:
     ...
```

```
if not self.has_ship and \
   self._cup.has_ship:
     self._cup.bank_ship()
     self._scores.append(self.sh

if self.has_ship:
if not self.has_captain and \
     self._cup.has_captain:
       self._cup.bank_captain()
       self._scores.append(self.

if self.has_captain and \
     not self.has_crew and \
     self._cup.has_crew:
       self._cup.bank_crew()
       self._scores.append(self.

if self.has_ship and \
     self.has_captain and \
     self.has_crew:
     ...
```

2. 大量重复代码，不利于扩展：如果出现多个角色需要选择怎么办？

```
if not self.has_ship and \
   self._cup.has_ship:
     self._cup.bank_ship()
     self._scores.append(self.sh

if not self.has_captain and \
     self._cup.has_captain:
       self._cup.bank_captain()
       self._scores.append(self.

if not self.has_crew and \
     self._cup.has_crew:
```

1. 使用list，存储所有的角色；
2. 使用指针，始终指向当前未被选中的角色。

```
13          self._cup.bank_crew()
14          self._scores.append(self.
15
16  if self.has_crew:
17      ...
```

3. `self` 和 `self._cup` 都需要判断：`has_ship`、`has_captain`、`has_crew`，这些共同的属性/动作，应该被抽象出来？

1.2 设计游戏的次要逻辑：投掷骰子、投掷杯子中的所有骰子

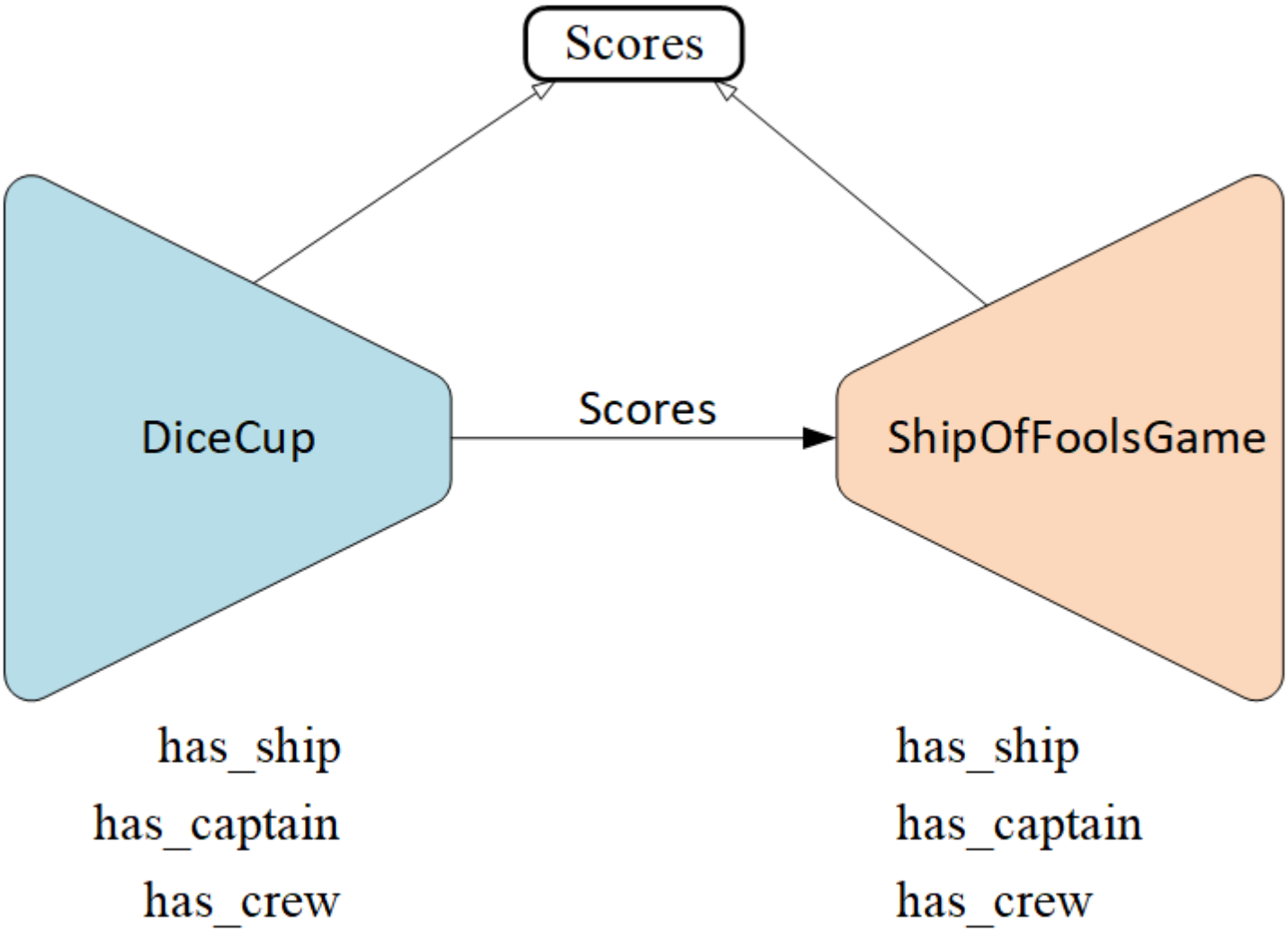
```python
class Die:
    def __init__(self):
        self.value = 0
        self.banked = False
        # 骰子有六个面，分值依次为从 1 到 6
        self._candidates = list(range(1, 7))

    def bank(self):
        self._banked = True

    def roll(self):
        self.value = random.choice(self._candidates)
        return self.value
```

```python
class DiceCup(Scores):
    def __init__(self):
        super().__init__()
        dice_num = 5
        self.all_dice = [Die() for _ in range(dice_num)]
        self._dice = []

    def roll(self):
        """
          投掷杯子中的筛子，需要同时记录：投掷了哪些骰子 和 这些骰子的得分，为什么？
        """
        self._dice = [_ for _ in self.all_dice if not _.banked]
        # 如果 self._dice 为空怎么办？考虑抛出异常？
        self._scores = [_.roll() for _ in self._dice]
        print('\nScores in this roll are', self._scores)

    def _bank(self, role, value):
```

```
18          """
19            从这里的实现可以看出，bank = remove
20            即：bank 一个骰子，实际上是要删除该骰子，而不是真的要保存它
21          """
22          # value = getattr(self, role)
23          print(f'\tBank the {role} ...')
24          idx = self._scores.index(value)
25          self._dice[idx].bank()
26          self._scores.pop(idx)
27          self._dice.pop(idx)
28
29      def bank_ship(self):
30          # 这样的调用是否非常啰嗦?
31          self._bank('ship', self.ship)
32
33      def bank_captain(self):
34          pass
35
36      def bank_crew(self):
37          pass
38
39      def bank_cargo(self, cargo):
40          pass
```

2. 主类分析：`DiceCup` vs. `ShipOfFoolsGame`

2.1 抽象`DiceCup`和`ShipOfFoolsGame`之间的关系



2.2 建立 `Scores` 实体(Entity)类

a. `getter` vs. `@property`：在Python中，不使用 `getter`，如：`get_ship`、`get_scores`，而使用 `@property`；

b. `@property` 的作用：

i. 保护私有变量，不被外部函数随意更改；

ii. 有些类成员的value，是需要依赖于其它类成员或函数进行动态计算，如要统计 @property: Python中的量子力学

iii. **便于覆盖：**在子类继承父类时，能够方便地实现子类对父类的某个属性的计算方法的覆盖；

iv. **懒加载：**可以在需要的时候再计算。

```
class Scores:
    def __init__(self):
        self._scores = []
        self.ship = 6
        self.captain = 5
        self.crew = 4

```

```
    @property
    def ship(self):
        return 6

    @property
    def captain(self):
        return 5

    @property
    def crew(self):
        return 4

    @property
    def scores(self):
        return self._scores

    @property
    def has_ship(self):
        return self.ship in self._scores

    @property
    def has_captain(self):
        pass

    @property
    def has_crew(self):
        pass
```

2.3 建立两个子类

```
class DiceCup(Scores):
    def __init__(self):
        super().__init__()
        dice_num = 5
        self.all_dice = [Die() for _ in range(dice_num)]
        self._dice = []

    def roll(self):
        pass

    def _bank(self, role, value):
        pass

    def bank_ship(self):
        pass
```

```
16
17      def bank_captain(self):
18          pass
19
20      def bank_crew(self):
21          pass
22
23      def bank_cargo(self, cargo):
24          pass
```

```
1  class ShipOfFoolsGame(Scores):
2      def __init__(self):
3          super().__init__()
4          self._cup = DiceCup()
5
6      def play(self):
7          pass
```