

설계패턴 실습 레포트

**과목명 설계패턴**

**담당교수 전병환 교수님**

**제출일 2023. 3. 26.**

**전공 컴퓨터.전자시스템 공학부**

**학번 201703091**

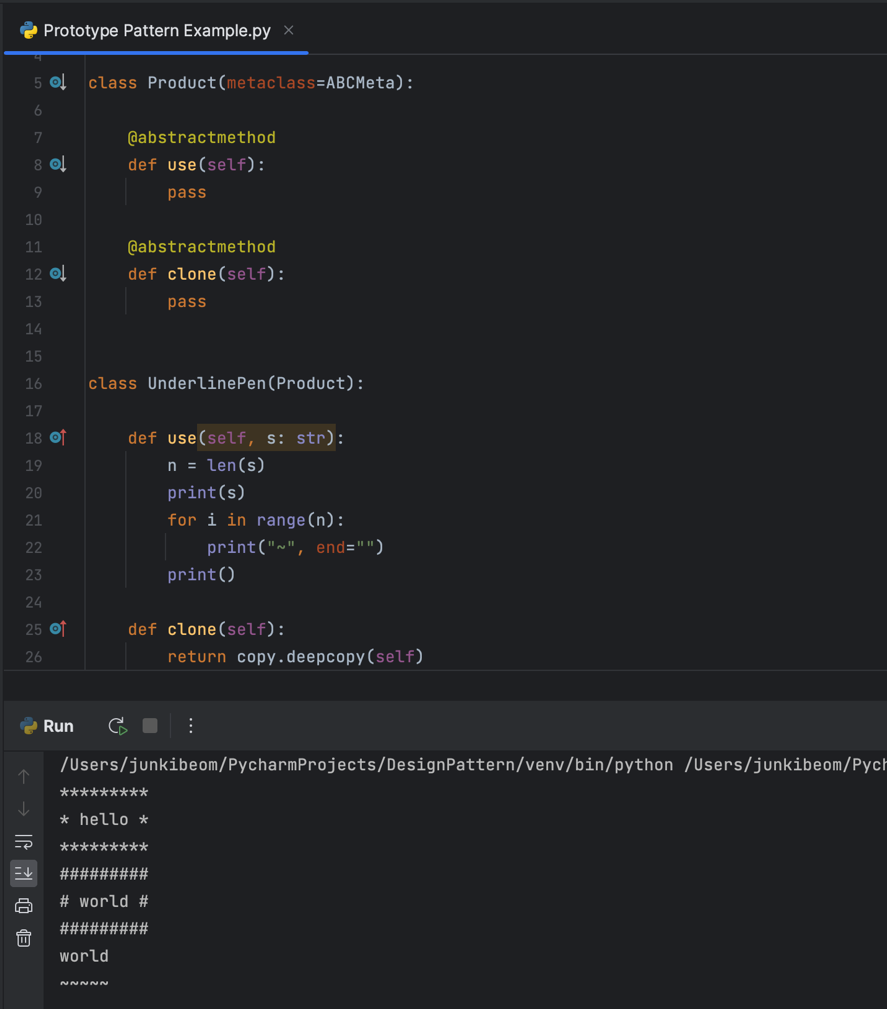
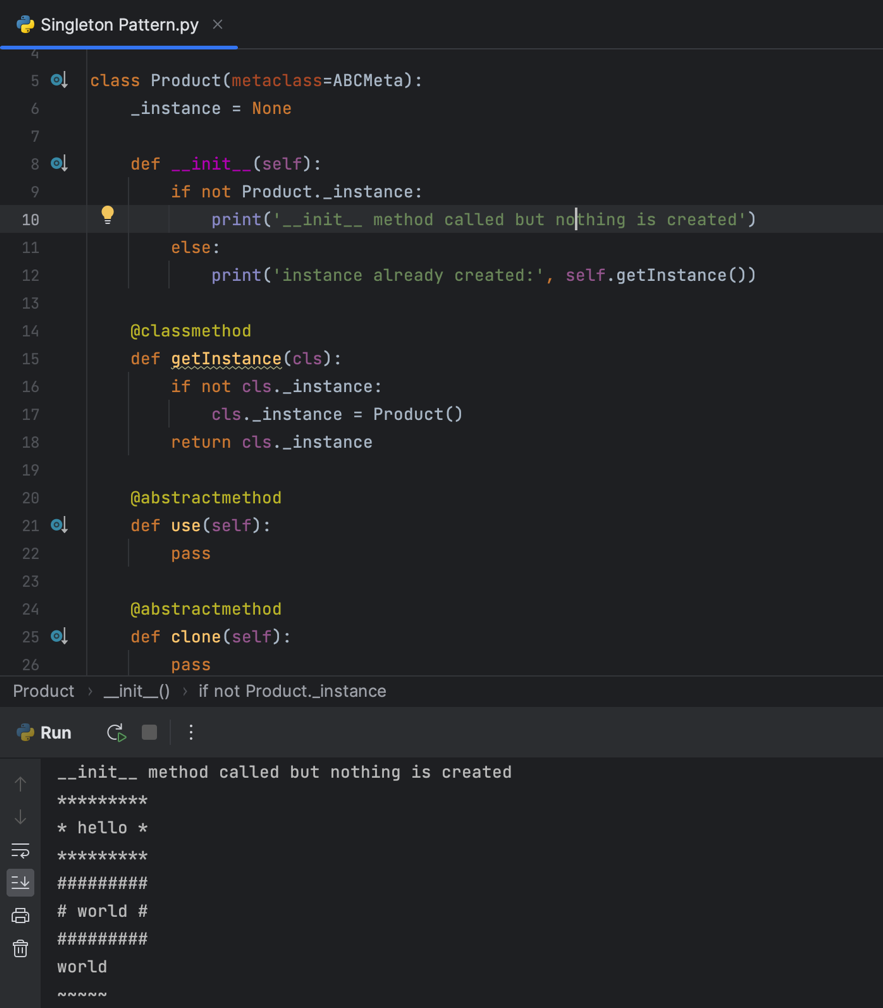
**이름 전기범**

실습 1)

1. 강의에서 마지막 Example 코드를 타이핑하여 결과를 확인하시오.

2. Product Class에 Singleton Pattern을 적용한 후 결과를 확인하시오.

3. 2번의 결과에 대해 왜 그러한 결과가 나오는지 논하시오.

실행 결과)

해결방안)

강의자료에 있는 내용을 참고하여 \_\_init\_\_, getInstance 메서드를 Product 메서드에 추가하여 하나의 인스턴스를 사용하는 Singleton 구조로 만들었습니다.

Product Class를 Singleton으로 변경하면 하나의 인스턴스를 공유하여 출력이 달라질 거라 기대를 하였는데, 기존 Prototype과 출력 결과가 같았습니다.

왜 그런 결과가 나왔는지 생각을 해보았는데, MessageBox 와 UnderlinePen 클래스는 여전히 clone을 이용해서 인스턴스를 복제하기 때문에 새로운 인스턴스가 생기므로 결과에 영향을 미치지 않는다는 것을 확인했습니다.

소스코드)

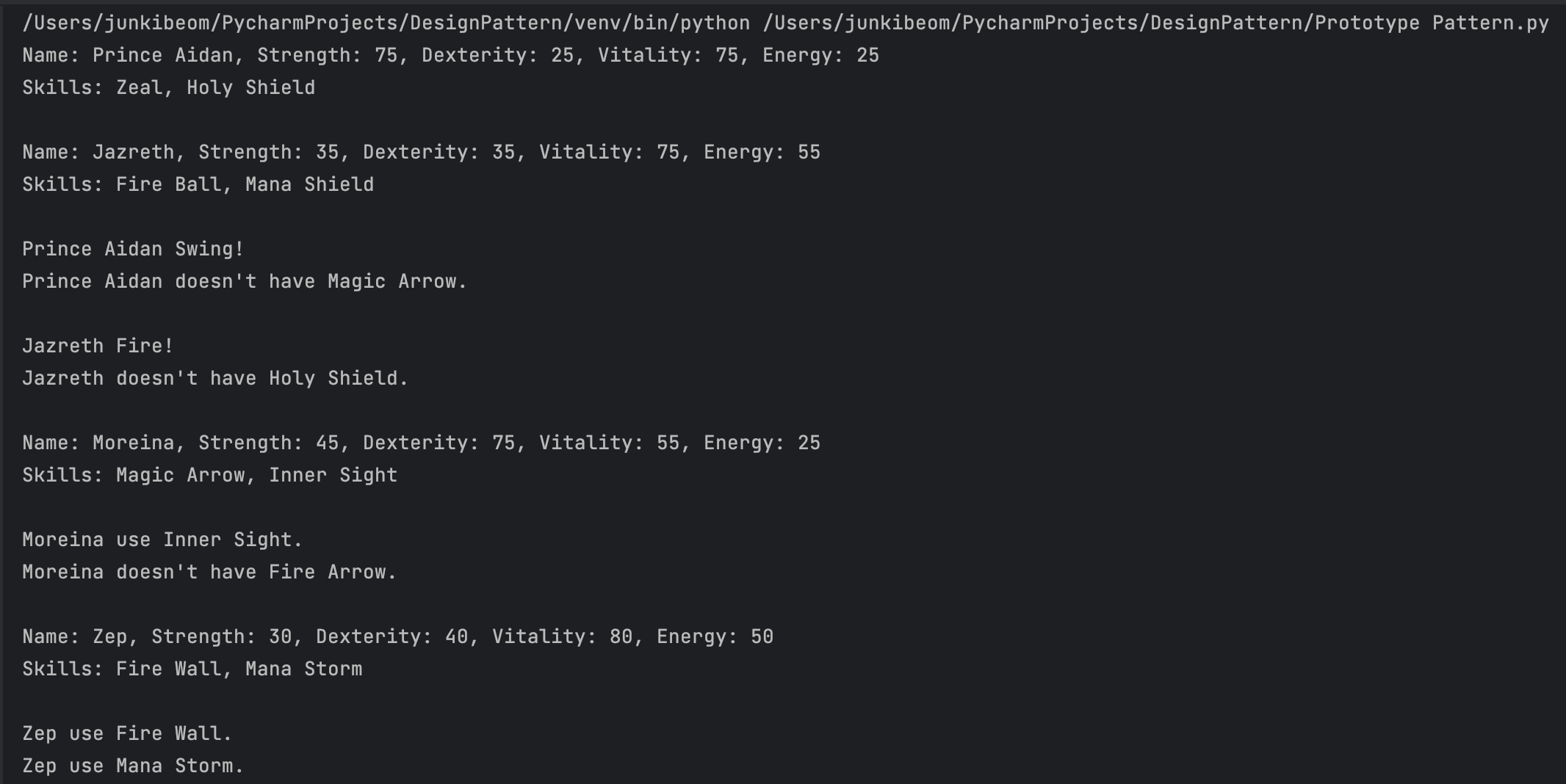
import copy  
from abc import \*  
  
  
class Product(metaclass=ABCMeta):  
  
 @abstractmethod  
 def use(self):  
 pass  
  
 @abstractmethod  
 def clone(self):  
 pass  
  
  
class UnderlinePen(Product):  
  
 def use(self, s: str):  
 n = len(s)  
 print(s)  
 for i in range(n):  
 print("~", end="")  
 print()  
  
 def clone(self):  
 return copy.deepcopy(self)  
  
  
class MessageBox(Product):  
  
 def \_\_init\_\_(self, deco: str):  
 self.deco = deco  
  
 def use(self, s: str):  
 n = len(s) + 4  
  
 for i in range(n):  
 print(self.deco, end="")  
 print()  
 print(self.deco, s, self.deco)  
 for i in range(n):  
 print(self.deco, end="")  
 print()  
  
 def clone(self):  
 return copy.deepcopy(self)  
  
  
class Manager:  
 def \_\_init\_\_(self):  
 self.showcase = {"a": 1}  
  
 def register(self, name: str, proto: Product):  
 self.showcase[name] = proto  
  
 def create(self, protoName):  
 p = self.showcase[protoName]  
 return p.clone()  
  
  
manager = Manager()  
  
m1 = MessageBox("\*")  
m2 = MessageBox("#")  
p1 = UnderlinePen()  
  
manager.register("msg\*", m1)  
manager.register("msg#", m2)  
manager.register("pen", p1)  
msg1 = manager.create("msg\*")  
msg2 = manager.create("msg#")  
pen = manager.create("pen")  
  
word = "hello"  
msg1.use(word)  
word = "world"  
msg2.use(word)  
pen.use(word)

실습 2)

4. 자유롭게 상황을 설정 후 Prototype Pattern을 적용한 클래스를 설계해보고 인스턴스를 생성하는 코드를 작성하시오.

   ex) 게임프로그램에서 캐릭터를 다루는 클래스를 구현하고자 할때,  prototype 패턴을 적용해보겠다. 등

5. 4번에 대해 왜 그러한 상황을 설정하였고, Prototype Pattern이 적절한가에 대해 남에게 설명한다는 생각으로 정리하시오.

실행 결과)

해결방안)

교수님께서 게임 프로그램에서 캐릭터를 다루는 클래스라고 언급하신 부분도 있고, 생성패턴 대부분이 게임 캐릭터 생성을 예시로 하기엔 좋아 보인다 생각하여 이렇게 작성하게 되었습니다.

게임 캐릭터의 경우 기본적인 틀(힘, 민첩, 체력 등의 스탯)은 모두 동일하게 갖고 직업별로 나뉘며,

플레이어에 따라 똑같은 직업의 캐릭터가 여러 번 생성될 수 있기 때문에 Prototype Pattern이 적절하다 생각하게 되었습니다.

Prototype 형식으로 만들기 위해 강의자료에 있는 Example 참고를 하여 진행을 하였습니다.

Character 클래스에 기본적으로 필요한 변수들을 선언했고, 필요한 추상 함수들 선언을 했습니다.

이후 Warrior, Rogue, Sorcerer 클래스에서 상속받은 클래스들을 작성했습니다.

CharacterManager 클래스에서 캐릭터를 생성마다 deepcopy하여 clone하도록 구현하였습니다.

소스코드)

import copy  
from abc import \*  
  
  
class Character(metaclass = ABCMeta):  
 def \_\_init\_\_(self, name, strength, dexterity, vitality, energy, skills):  
 self.name = name  
 self.strength = strength  
 self.dexterity = dexterity  
 self.vitality = vitality  
 self.energy = energy  
 self.skills = []+skills  
  
 @abstractmethod  
 def use\_skill(self):  
 pass  
  
 @abstractmethod  
 def clone(self):  
 pass  
  
  
class Warrior(Character):  
 def \_\_init\_\_(self, name, strength, dexterity, vitality, energy, skills):  
 super().\_\_init\_\_(name, strength, dexterity, vitality, energy, skills)  
  
 def use\_skill(self, skill):  
 if skill == "Zeal":  
 print(f"{self.name} Swing!")  
 elif skill in self.skills:  
 print(f"{self.name} use {skill}.")  
 else:  
 print(f"{self.name} doesn't have {skill}.")  
  
 def clone(self):  
 return copy.deepcopy(self)  
  
  
class Rogue(Character):  
 def \_\_init\_\_(self, name, strength, dexterity, vitality, energy, skills):  
 super().\_\_init\_\_(name, strength, dexterity, vitality, energy, skills)  
  
 def use\_skill(self, skill):  
 if skill == "Magic Arrow":  
 print(f"{self.name} Magic Arrow!")  
 elif skill in self.skills:  
 print(f"{self.name} use {skill}.")  
 else:  
 print(f"{self.name} doesn't have {skill}.")  
  
 def clone(self):  
 return copy.deepcopy(self)  
  
  
class Sorcerer(Character):  
 def \_\_init\_\_(self, name, strength, dexterity, vitality, energy, skills):  
 super().\_\_init\_\_(name, strength, dexterity, vitality, energy, skills)  
  
 def use\_skill(self, skill):  
 if skill == "Fire Ball":  
 print(f"{self.name} Fire!")  
 elif skill in self.skills:  
 print(f"{self.name} use {skill}.")  
 else:  
 print(f"{self.name} doesn't have {skill}.")  
  
 def clone(self):  
 return copy.deepcopy(self)  
  
  
class CharacterManager:  
 def \_\_init\_\_(self):  
 self.characters = {}  
  
 def add\_character(self, name: str, proto: Character):  
 self.characters[name] = proto  
  
 def get\_character(self, protoName):  
 print(f"Name: {protoName.name}, Strength: {protoName.strength}, Dexterity: {protoName.dexterity},"  
 f" Vitality: {protoName.vitality}, Energy: {protoName.energy}")  
 print("Skills:", ", ".join(protoName.skills))  
 print()  
  
 def create\_character(self, protoName):  
 char = self.characters[protoName]  
 return char.clone()  
  
  
# name, strength, dexterity, vitality, energy, skills  
manager = CharacterManager()  
warrior1 = Warrior("Prince Aidan", 75, 25, 75, 25, ["Zeal", "Holy Shield"])  
sorcerer1 = Sorcerer("Jazreth", 35, 35, 75, 55, ["Fire Ball", "Mana Shield"])  
rouge1 = Rogue("Moreina", 45, 75, 55, 25, ["Magic Arrow", "Inner Sight"])  
  
manager.add\_character("Warrior", warrior1)  
manager.add\_character("Sorcerer", sorcerer1)  
manager.add\_character("Rogue", rouge1)  
  
warrior1 = manager.create\_character("Warrior")  
sorcerer1 = manager.create\_character("Sorcerer")  
  
manager.get\_character(warrior1)  
manager.get\_character(sorcerer1)  
  
warrior1.use\_skill("Zeal")  
warrior1.use\_skill("Magic Arrow")  
print()  
  
sorcerer1.use\_skill("Fire Ball")  
sorcerer1.use\_skill("Holy Shield")  
print()  
  
rouge1 = manager.create\_character("Rogue")  
manager.get\_character(rouge1)  
  
rouge1.use\_skill("Inner Sight")  
rouge1.use\_skill("Fire Arrow")  
print()  
  
sorcerer2 = Sorcerer("Zep", 30, 40, 80, 50, ["Fire Wall", "Mana Storm"])  
manager.add\_character("Sorcerer", sorcerer2)  
sorcerer2 = manager.create\_character("Sorcerer")  
manager.get\_character(sorcerer2)  
sorcerer2.use\_skill("Fire Wall")  
sorcerer2.use\_skill("Mana Storm")

느낀점)

실습1의 Singleton의 경우 하나의 인스턴스를 공유하여 사용하기 위해 쓰는거라 Clone을 그대로 두면 의미가 없다는걸 배웠습니다. 하나의 객체를 참조하게 만들어야 그게 진정한 싱글톤 구조이며 이걸 디테일하게 사용하려면 까다로워 보였습니다.

실습2의 경우엔 지난주에 진행했던 Builder패턴와 비슷한 느낌을 받았습니다.

아무래도 두 패턴 모두 같은 생성패턴의 범주에 있어서 그렇게 느낀거 같습니다.

Prototype Pattern은 이미 존재하는 객체를 복제하여 새로운 객체를 생성하므로 객체 생성에 드는 비용을 절감을 하는 특징이 있고,

Builder Pattern은 복잡한 객체를 생성하기 위해 객체의 생성 과정을 분리하여 단계적으로 실행한다는 특징이 있어서 두 패턴의 특징과 차이를 명확하게 이해하고 사용하는 것이 좋다는 생각이 들었습니다.