Report

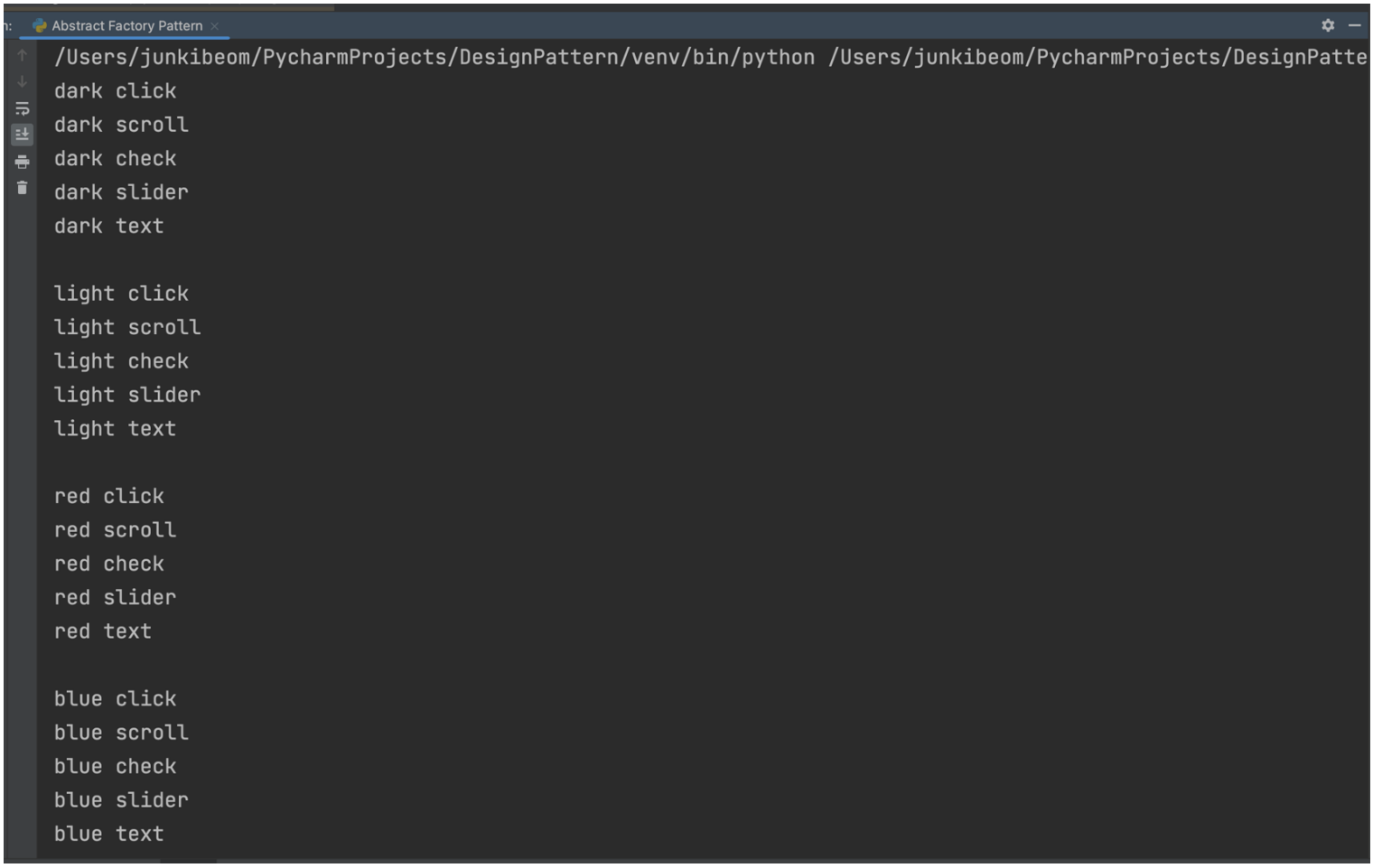
설계패턴 실습 레포트

과목명 설계패턴
담당교수 전병환 교수님
제출일 2023. 3. 15.
전공 컴퓨터.전자시스템 공학부
학번 201703091
이름 전기범

한국외국어대학교
HANKUK UNIVERSITY OF FOREIGN STUDIES

실습 1)
GUI tool 을 개발하는 과정에서 버튼, 스크롤, 체크박스, 슬라이더, 텍스트박스 이렇게 5 개 객체를 Abstract Factory Pattern 으로 생성하려고 한다.
이때, 다양한 테마를 제공하기 위해서 다크모드, 라이트모드, 레드모드, 블루모드로 생성할 수 있도록 만들고자 한다.
각 테마별 모드 별로 객체들을 얻고 이들이 가지고 있는 함수들을 실행하는 코드를 작성해 보시오. (예를 들어 다크모드에서 click 의 경우 print("dark click") 과 같은 식으로 간단하게 구현)

실행 결과)



해결방안)
Abstract Factory Pattern 에서 새로운 기능 및 테마를 구현 하는 것은 간단했습니다.
기존에 있던 기능을 위한 interface 를 상속 받는 새로운 테마에 해당하는 클래스들을 작성해줬습니다.
기존에 없었던 기능인 Slider 와 TextBox 기능도 interface 선언 후, 각 테마에 맞게 상속 및 작성을 진행했고,

테마 별 기능을 만드는 Factory 클래스를 작성하였습니다.
이후 Main 문에서 테마 별 Factory()를 호출하여 인스턴스 생성, 메서드 호출로 기능을 수행하도록 하였고
위 사진과 같은 실행결과가 나오게 되었습니다.

소스코드)

```
# Button abstract(interface)
class Button:
    def click(self):
        pass

class DarkButton(Button):
    def click(self):
        print("dark click")

class LightButton(Button):
    def click(self):
        print("light click")

class RedButton(Button):
    def click(self):
        print("red click")

class BlueButton(Button):
    def click(self):
        print("blue click")


# Scrollbar abstract(interface)
class ScrollBar:
    def scroll(self):
        pass

class DarkScrollBar(ScrollBar):
    def scroll(self):
        print("dark scroll")

class LightScrollBar(ScrollBar):
    def scroll(self):
        print("light scroll")

class RedScrollBar(ScrollBar):
    def scroll(self):
        print("red scroll")

class BlueScrollBar(ScrollBar):
    def scroll(self):
        print("blue scroll")


# Checkbox abstract(interface)
class CheckBox:
    def check(self):
        pass
```

```
class DarkCheckBox(CheckBox):
    def check(self):
        print("dark check")

class LightCheckBox(CheckBox):
    def check(self):
        print("light check")

class RedCheckBox(CheckBox):
    def check(self):
        print("red check")

class BlueCheckBox(CheckBox):
    def check(self):
        print("blue check")


# Slider abstract(interface)
class Slider:
    def slide(self):
        pass
class DarkSlider(Slider):
    def slide(self):
        print("dark slider")

class LightSlider(Slider):
    def slide(self):
        print("light slider")

class RedSlider(Slider):
    def slide(self):
        print("red slider")

class BlueSlider(Slider):
    def slide(self):
        print("blue slider")


# TextBox abstract(interface)
class TextBox:
    def text(self):
        pass

class DarkTextBox(TextBox):
    def text(self):
        print("dark text")

class LightTextBox(TextBox):
    def text(self):
        print("light text")

class RedTextBox(TextBox):
    def text(self):
        print("red text")

class BlueTextBox(TextBox):
    def text(self):
```

```
        print("blue text")


# UIFactory Abstract(interface)
class UIFactory:
    def getButton(self):
        pass

    def getScrollBar(self):
        pass

    def getCheckBox(self):
        pass

    def getSlider(self):
        pass

    def getTextBox(self):
        pass

class DarkFactory(UIFactory):
    def getButton(self):
        return DarkButton()

    def getScrollBar(self):
        return DarkScrollBar()

    def getCheckBox(self):
        return DarkCheckBox()

    def getSlider(self):
        return DarkSlider()

    def getTextBox(self):
        return DarkTextBox()

class LightFactory(UIFactory):
    def getButton(self):
        return LightButton()

    def getScrollBar(self):
        return LightScrollBar()

    def getCheckBox(self):
        return LightCheckBox()

    def getSlider(self):
        return LightSlider()

    def getTextBox(self):
        return LightTextBox()

class RedFactory(UIFactory):
    def getButton(self):
        return RedButton()

    def getScrollBar(self):
        return RedScrollBar()
```

```
    def getCheckBox(self):
        return RedCheckBox()

    def getSlider(self):
        return RedSlider()

    def getTextBox(self):
        return RedTextBox()

class BlueFactory(UIFactory):
    def getButton(self):
        return BlueButton()

    def getScrollBar(self):
        return BlueScrollBar()

    def getCheckBox(self):
        return BlueCheckBox()

    def getSlider(self):
        return BlueSlider()

    def getTextBox(self):
        return BlueTextBox()

# darkTheme UI
dark_factory = DarkFactory()
dark_btn = dark_factory.getButton()
dark_scrollBar = dark_factory.getScrollBar()
dark_checkBox = dark_factory.getCheckBox()
dark_slider = dark_factory.getSlider()
dark_textBox = dark_factory.getTextBox()

dark_btn.click()
dark_scrollBar.scroll()
dark_checkBox.check()
dark_slider.slide()
dark_textBox.text()

print() # \n
# lightTheme UI
light_factory = LightFactory()
light_btn = light_factory.getButton()
light_scrollBar = light_factory.getScrollBar()
light_checkBox = light_factory.getCheckBox()
light_slider = light_factory.getSlider()
light_textBox = light_factory.getTextBox()

light_btn.click()
light_scrollBar.scroll()
light_checkBox.check()
light_slider.slide()
light_textBox.text()

print() # \n
# redTheme UI
red_factory = RedFactory()
```

```
red_btn = red_factory.getButton()
red_scrollBar = red_factory.getScrollBar()
red_checkBox = red_factory.getCheckBox()
red_slider = red_factory.getSlider()
red_textBox = red_factory.getTextBox()

red_btn.click()
red_scrollBar.scroll()
red_checkBox.check()
red_slider.slide()
red_textBox.text()

print() # \n
# blueTheme UI
blue_factory = BlueFactory()
blue_btn = blue_factory.getButton()
blue_scrollBar = blue_factory.getScrollBar()
blue_checkBox = blue_factory.getCheckBox()
blue_slider = blue_factory.getSlider()
blue_textBox = blue_factory.getTextBox()

blue_btn.click()
blue_scrollBar.scroll()
blue_checkBox.check()
blue_slider.slide()
blue_textBox.text()
```

실습 2)
주어진 클래스
- Actor 클래스
- Hero 클래스
- Monster 클래스
어떤 게임을 개발하는 과정에서 Builder Pattern 을 이용하여 주인공과 몬스터를 쉽게 생성하고자 한다.
1. 주인공과 몬스터 각각에 대한 생성을 담당하는 구체 Builder 클래스들을 구현해보자.
- Builder 클래스에서는 공통 argument 들을 셋업하며, 하위 Builder 클래스들은 각각 Hero 와 Monster 에 특화된 argument 셋업을 할 것
2. 캐릭터들의 위치 (x, y)의 각각 요소들이 0 보다 작은 값으로 들어오면 0 으로, 100 보다 큰 값으로 들어오면 100 으로 자동 제어되어 설정 후 생성한다.
3. 주인공의 체력, 민첩성, 힘은 외부인자로 주어진 값으로 설정할 수 있다.
4. 몬스터의 체력, 민첩성, 힘은 50 에서 100 사이의 임의의 값으로 설정된다.
5. 몬스터의 위치(x, y)는 각각 0~100 사이의 임의의 값으로 설정된다.
6. Builder 에서는 return self 방법을 이용한다.

실행결과)

```
/Users/junkibeom/PycharmProjects/DesignPattern/venv/bin/python /Users/junkibeom/PycharmProjects/DesignPattern/venv/Builder Patt
새로운 아마존 이(가) 태어났습니다.
착용 중인 아이템은 그랜드 메이트런 보우, 와이어 플리스입니다.
주인공1의 위치 (0, 100)
주인공1의 상태 (80, 400, 108) (None, None, None, '그랜드 메이트런 보우', '와이어 플리스', '아마존')

새로운 원소술사 이(가) 태어났습니다.
착용 중인 아이템은 엘드리치 오브, 웜하이드입니다.
주인공2의 위치 (95, 10)
주인공2의 상태 (420, 50, 156) ('연쇄 번개', 80, 302445, '엘드리치 오브', '웜하이드', '원소술사')
연쇄 번개기술을 사용했습니다.
주인공2의 상태 (421, 51, 157) ('연쇄 번개', 81, 302445, '엘드리치 오브', '웜하이드', '원소술사')

불길의 강 지역에서 몬스터 출몰
몬스터1의 위치 (39, 98)
몬스터1의 상태 (51, 90, 69) ('불길의 강', None, None)

평원 외곽 지역에서 몬스터 출몰
몬스터2의 위치 (38, 7)
몬스터2의 상태 (55, 66, 77) ('평원 외곽', '파랑', 4.5)

매장지 지역에서 몬스터 출몰
몬스터3의 위치 (70, 96)
몬스터3의 상태 (89, 83, 72) ('매장지', None, 7.8)
```

해결방안)
공통된 argument 가 있는 ActorBuilder 를 먼저 작성 후, ActorBuilder 를 상속받는 HeroBuilder 와 MonsterBuilder 를 구현했습니다. 각 Builder 에서는 해당 클래스에서 특화된 arguments 들을 setup 하였고,
캐릭터들의 위치가 0~100 사이로 들어갈 수 있도록 Python 의 내장함수인 max 와 min 을 이용하여 범위를 벗어나면 자동 제어 되도록 하였습니다. max(0, min(value, 100))
주인공의 체력, 민첩성, 힘은 외부(Client)에서 따로 설정이 가능하며
몬스터의 체력, 민첩성, 힘, 위치는 Random.radint(a, b)함수를 이용하여 임의 값을 가지도록 하였습니다.
몬스터의 경우 Director(preset)를 이용해 생성도 가능하여 해당 방법으로도 구현을 해보았습니다.

주인공과 몬스터의 위치 정보를 확인하기 위해 Actor 클래스에 getPosition() 메서드를 추가하여 위치정보 리턴,
주인공의 스킬, 레벨, 돈, 무기, 방어구, 직업 등을 확인하기 위해 Hero 클래스에 getHero()라는 메서드를 만들어 해당 정보를 리턴,
몬스터의 거주지, 색상, 아이템 드롭 확률, 몬스터를 생성시 랜덤으로 변화하는 능력치 및 위치를 확인하기 위해 Monster 클래스에 getMonster()라는 메서드를 만들어 해당 정보를 리턴하게 하였습니다.

Client 부분에 클래스와 Builder 들이 잘 작성이 되었는지 확인하기 위하여 실제 게임 디아블로 2 에 있는 정보를 집어 넣어 테스트를 진행해보았고 위 사진과 같은 실행결과가 나오게 되었습니다

소스코드)

```
import random

# Product
class Actor:
    def __init__(self, x, y, vital, agi, strength):
        self.x = x
        self.y = y

        self.vitality = vital
        self.agility = agi
        self.strength = strength

    def getPosition(self):
        return self.x, self.y

    def getInfo(self):
        return self.vitality, self.agility, self.strength

class Hero(Actor):
    def __init__(self, x, y, vital, agi, strength, skill, level,
money, weapon, armour, job):
        super().__init__(x, y, vital, agi, strength)

        self.skill = skill
```

```
        self.level = level
        self.money = money
        self.weapon = weapon
        self.armour = armour
        self.job = job

    def printState(self):
        print("새로운 " + self.job + " 이(가) 태어났습니다.")
        print("착용 중인 아이템은 " + self.weapon + ", " + self.armour +
"입니다.")

    def action(self):
        print(self.skill + "기술을 사용했습니다.")

    def levelUp(self):
        self.level += 1
        self.vitality += 1
        self.agility += 1
        self.strength += 1

    def getHero(self):
        return self.skill, self.level, self.money, self.weapon,
self.armour, self.job

class Monster(Actor):
    def __init__(self, x, y, vital, agi, strength, residence, color,
itemDropRate):
        super().__init__(x, y, vital, agi, strength)

        self.residence = residence
        self.color = color
        self.itemDropRate = itemDropRate

    def printState(self):
        print(self.residence + " 지역에서 몬스터 출몰")

    def getMonster(self):
        return self.residence, self.color, self.itemDropRate

# Builder
class ActorBuilder:
    def __init__(self):
        self.x = None
        self.y = None

        self.vitality = None
        self.agility = None
        self.strength = None

    # Setter
    def setX(self, x):
        self.x = max(0, min(x, 100))
        return self

    def setY(self, y):
```

```
        self.y = max(0, min(y, 100))
        return self

    def setVitality(self, vital):
        self.vitality = vital
        return self

    def setAgility(self, agi):
        self.agility = agi
        return self

    def setStrength(self, strength):
        self.strength = strength
        return self

# Concrete Builder
class HeroBuilder(ActorBuilder):
    def __init__(self):
        super().__init__()
        self.skill = None
        self.level = None
        self.money = None
        self.weapon = None
        self.armour = None
        self.job = None

    # Setter
    def setSkill(self, skill):
        self.skill = skill
        return self

    def setLevel(self, level):
        self.level = level
        return self

    def setMoney(self, money):
        self.money = money
        return self

    def setWeapon(self, weapon):
        self.weapon = weapon
        return self

    def setArmour(self, armour):
        self.armour = armour
        return self

    def setJob(self, job):
        self.job = job
        return self

    def heroBuild(self):
        hero = Hero(self.x, self.y, self.vitality, self.agility,
self.strength,
                    self.skill, self.level, self.money, self.weapon,
self.armour, self.job)
        return hero
```

```
class MonsterBuilder(ActorBuilder):
    def __init__(self):
        super().__init__()
        self.residence = None
        self.color = None
        self.itemDropRate = None

    # Setter
    def setResidence(self, residence):
        self.residence = residence
        return self

    def setColor(self, color):
        self.color = color
        return self

    def setItemDropRate(self, itemDropRate):
        self.itemDropRate = itemDropRate
        return self

    def monsterBuilder(self):
        self.x = random.randint(0, 100)
        self.y = random.randint(0, 100)
        self.vitality = random.randint(50, 100)
        self.agility = random.randint(50, 100)
        self.strength = random.randint(50, 100)

        monster = Monster(self.x, self.y, self.vitality, self.agility,
self.strength,
                          self.residence, self.color,
self.itemDropRate)
        return monster

# Director(preset)
class Director:
    def monster(builder:MonsterBuilder):
        builder.setX(random.randint(0, 100))
        builder.setY(random.randint(0, 100))
        builder.setVitality(random.randint(50, 100))
        builder.setAgility(random.randint(50, 100))
        builder.setStrength(random.randint(50, 100))

# Client
hero1 = HeroBuilder().setX(-
49).setY(140).setVitality(80).setAgility(400).setStrength(108).\
    setJob("아마존").setWeapon("그랜드 메이트런 보우").setArmour("와이어
플리스").heroBuild()
hero1.printState()
print("주인공 1 의 위치", hero1.getPosition())
print("주인공 1 의 상태", hero1.getInfo(), hero1.getHero(),"\n")

hero2 =
HeroBuilder().setX(95).setY(10).setVitality(420).setAgility(50).setStr
ength(156)\
    .setLevel(80).setSkill("연쇄
```

```
번개").setMoney(302445).setJob("원소술사").setWeapon("엘드리치
오브").setArmour("웜하이드")\
    .heroBuild()
hero2.printState()
print("주인공 2 의 위치", hero2.getPosition())
print("주인공 2 의 상태", hero2.getInfo(), hero2.getHero())
hero2.action()
hero2.levelUp()
print("주인공 2 의 상태", hero2.getInfo(), hero2.getHero(),"\n")

monster1 = MonsterBuilder().setResidence("불길의 강").monsterBuilder()
monster1.printState()
print("몬스터 1 의 위치", monster1.getPosition())
print("몬스터 1 의 상태", monster1.getInfo(), monster1.getMonster(),"\n")

monster2 = MonsterBuilder().setResidence("평원
외곽").setColor("파랑").setItemDropRate(4.5).monsterBuilder()
monster2.printState()
print("몬스터 2 의 위치", monster2.getPosition())
print("몬스터 2 의 상태", monster2.getInfo(), monster2.getMonster(),"\n")

# Client (using Director(Preset))
preset1 = MonsterBuilder()
Director.monster(preset1)
monster3 =
preset1.setResidence("매장지").setItemDropRate(7.8).monsterBuilder()
monster3.printState()
print("몬스터 3 의 위치", monster3.getPosition())
print("몬스터 3 의 상태", monster3.getInfo(), monster3.getMonster(),"\n")
```

느낀점)

Abstract Factory Pattern 실습의 경우 새로운 기능을 추가 하는 것이 어렵지 않았습니다.
쉽게 생각하면 Copy&Paste, Modify 의 연속으로
다형성의 예시를 잘 보여주는 케이스라고 생각이 되었으나, 새로운 기능이 추가되면 기존 클래스를 수정하거나, 새로운 인터페이스 및 클래스를 지속적으로 만들고 관리해야 한다는 단점이 있었습니다.
반면 Builder Pattern 의 경우 구현에 있어 조금 고민이 되었으나, 사용성에 있어서는 훨씬 좋다는 생각이 들었습니다. 복잡한 생성과정 없이 Builder 하나로 일관되게 생성이 가능하다는 것이 큰 장점으로 느껴집니다.