학번: \_\_\_2016112158\_\_\_\_\_\_ 이름: \_\_김희수\_\_\_



객체지향 설계와 패턴

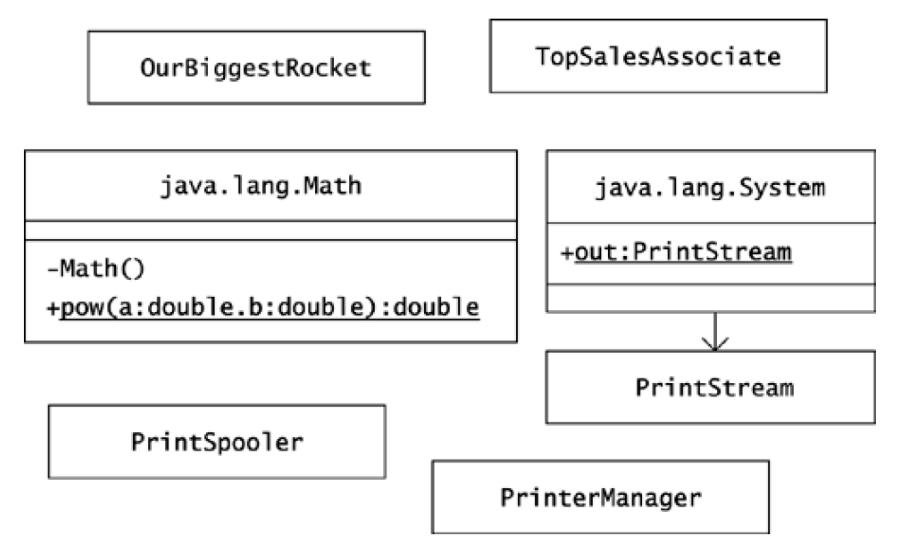
**LAB #6: 싱글톤, 옵서버, 중재자패턴**



**목적:**싱글톤,옵서버,중재자패턴

**싱글톤 패턴:**

문제 #1: 다음 클래스 중에 싱글톤을 적용하여야 할 것으로 보이는 클래스는 무엇인가? 그 이유는?



!

PrinterManager의 역할은 PrintStream을 받아와 내용을 출력한다고 하자. 즉, 여러 PrintStream이 들어오게 될 것이며 PrinterManager가 PrintSpooler를 이용해 이것들을 관리할 것이다. 하나의 프린터에 대해 그것을 관리하는 객체가 여러 개여서는 안된다. 따라서 PrinterManager는 프린터당 오직 하나만 존재해야한다. 그러므로 PrinterManager클래스에 싱글톤을 적용하는 것이 옳다

문제 #2: 제공한 SingletonExercise.java 프로그램에는 52개의 카드를 표현한 "Deck"이라는 클래스가 있다. 이 클래스를 하 나만 만들 수 있도록 싱글톤 패턴을 적용하여 변경하라.

<SingletomExercise.java>

import java.util.\*;

enum Suit {

객체지향설계와패턴 *Page 1! of 6!*동국대학교

학번: \_\_\_2016112158\_\_\_\_\_\_ 이름: \_\_김희수\_\_\_

SPADES,

HEARTS,

CLUBS,

DIAMONDS

}

class Card {

public Card(Suit s, int n) {

suit = s;

if((n < 2) || (n > 14)) {

throw new IllegalArgumentException( );

}

number = n;

}

public void print( ) {

switch(number) {

case 11:

System.out.print("Jack");

break;

case 12:

System.out.print("Queen");

break;

case 13:

System.out.print("King");

break;

case 14:

System.out.print("Ace");

break;

default:

System.out.print(number);

break;

}

System.out.print(" of ");

switch(suit) {

case SPADES:

System.out.println("spades.");

break;

case HEARTS:

System.out.println("hearts.");

break;

case CLUBS:

System.out.println("clubs.");

break;

case DIAMONDS:

System.out.println("diamonds.");

break;

}

}

private Suit suit;

private int number;

}

class Deck {

public Deck( ) {

cards = new ArrayList<Card>( );

객체지향설계와패턴 *Page 2! of 6!*동국대학교

학번: \_\_\_2016112158\_\_\_\_\_\_ 이름: \_\_김희수\_\_\_

// build the deck

Suit[] suits = {Suit.SPADES, Suit.HEARTS, Suit.CLUBS, Suit.DIAMONDS}; for(Suit suit: suits) {

for(int i = 2; i <= 14; i++) {

cards.add(new Card(suit, i));

}

}

// shuffle it!

Collections.shuffle(cards, new Random( ));

}

public void print( ) {

for(Card card: cards) {

card.print( );

}

}

private List<Card> cards;

}

public class SingletonExercise {

public static void main(String args[]) {

Deck deck = new Deck( );

deck.print( );

}

}

객체지향설계와패턴 *Page 3! of 6!*동국대학교

학번: \_\_\_2016112158\_\_\_\_\_\_ 이름: \_\_김희수\_\_\_

**옵서버 패턴**

문제 #3: 다음의 Dogbot 클래스를 Java의 Observer 인터페이스 타입을 이용하여Observerable 클래스로 만들어라.

class DogBot

{

protected int hungry,tired;

public DogBot(int h,int t)

{

hungry=h;

tired=t;

}

public boolean eat()

{

if(hungry>6)

{

hungry-=3;

return true;

}

else

return false;

}

public void rest()

{

hungry++;

tired-=2;

}

public void play()

{

hungry+=2;

tired+=3;

}

public String noise()

{

if(hungry>8&&tired<11)

return "whine ";

else if(tired>7&&tired>hungry)

return "snore ";

else

return "woof ";

}

}

구체적인 방법은 다음과 같다.

1. Dogbot 클래스를 Java의 Observer 인터페이스의 서브 클래스로 만든다.
2. 원래의 Dogbot 클래스의 메소드를 선언하는 Dogbot인터페이스를 만든다.
3. 원래의 Dogbot 클래스를 변형한 PlainDogbot 클래스를 만든다. 즉 Dogbot인터페이스를 구현한 코드.
4. Dogbot의 어떤 액션이 일어날 때 이름과 액션을 인쇄하는 DogWatcher.java를 만든다.
5. 다음에 있는 test.java 코드로 위 프로그램이 올바로 구현되었는지 테스트하라.

객체지향설계와패턴 *Page 4! of 6!*동국대학교

학번: \_\_\_2016112158\_\_\_\_\_\_ 이름: \_\_김희수\_\_\_

<test.java>

class Test1

{

public static void main(String[] args)

{

DogBot patch = new PlainDogBot(4,2,"Patch");

DogBot rover = new PlainDogBot(9,6,"Rover");

ObservableDogBot observedPatch = new ObservableDogBot(patch);

ObservableDogBot observedRover = new ObservableDogBot(rover);

DogWatcher jim = new DogWatcher("Jim");

DogWatcher fred = new DogWatcher("Fred");

System.out.println("\nStarting Observation Test 1 with:\n");

System.out.println(observedPatch);

System.out.println(observedRover);

System.out.println(jim);

System.out.println(fred);

System.out.println("\nNo-one is observing Rover or Patch");

System.out.println(observedRover+" has "+observedRover.countObservers()+" observers");

System.out.println(observedPatch+" has "+observedPatch.countObservers()+" observers\n");

System.out.println("Rover rests");

observedRover.rest();

System.out.println("Patch plays");

observedPatch.play();

System.out.println("\nJim observes Rover and Patch\n");

observedRover.addObserver(jim);

observedPatch.addObserver(jim);

System.out.println("Patch plays");

observedPatch.play();

System.out.println("Rover plays");

observedRover.play();

System.out.println("Rover makes noise: "+observedRover.noise());

System.out.println("Patch makes noise: "+observedPatch.noise());

System.out.println("\nFred observes Patch");

observedPatch.addObserver(fred);

System.out.println(observedRover+" has "+observedRover.countObservers()+" observers");

System.out.println(observedPatch+" has "+observedPatch.countObservers()+" observers\n");

System.out.println("Patch plays");

observedPatch.play();

System.out.println("Rover plays");

observedRover.play();

System.out.println("Patch looks at food");

if(observedPatch.eat())

System.out.println(" and Patch eats");

else

System.out.println(" and Patch decides not to eat"); System.out.println("Rover makes noise: "+observedRover.noise()); System.out.println("Patch makes noise: "+observedPatch.noise()); System.out.println("\nFred stops observing Patch\n"); observedPatch.deleteObserver(fred); System.out.println("Patch rests");

observedPatch.rest();

객체지향설계와패턴 *Page 5! of 6!*동국대학교

학번: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 이름: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

System.out.println("\nFred starts observing Rover\n");

observedRover.addObserver(fred);

System.out.println("Patch rests");

observedPatch.rest();

System.out.println("Rover rests");

observedRover.rest();

System.out.println("Patch looks at food");

if(observedPatch.eat())

System.out.println(" and Patch eats");

else

System.out.println(" and Patch decides not to eat");

System.out.println("Patch looks at food again");

if(observedPatch.eat())

System.out.println(" and Patch eats");

else

System.out.println(" and Patch decides not to eat"); System.out.println("Rover makes noise: "+observedRover.noise()); System.out.println("Patch makes noise: "+observedPatch.noise()); System.out.println("Rover looks at food"); if(observedRover.eat())

System.out.println("Rover eats");

else

System.out.println("Rover decides not to eat");

System.out.println("\nJim stops observing Rover\n");

observedRover.deleteObserver(jim);

System.out.println("Rover looks at food again");

if(observedRover.eat())

System.out.println(" and Rover eats");

else

System.out.println(" and Rover decides not to eat");

System.out.println("Rover rests");

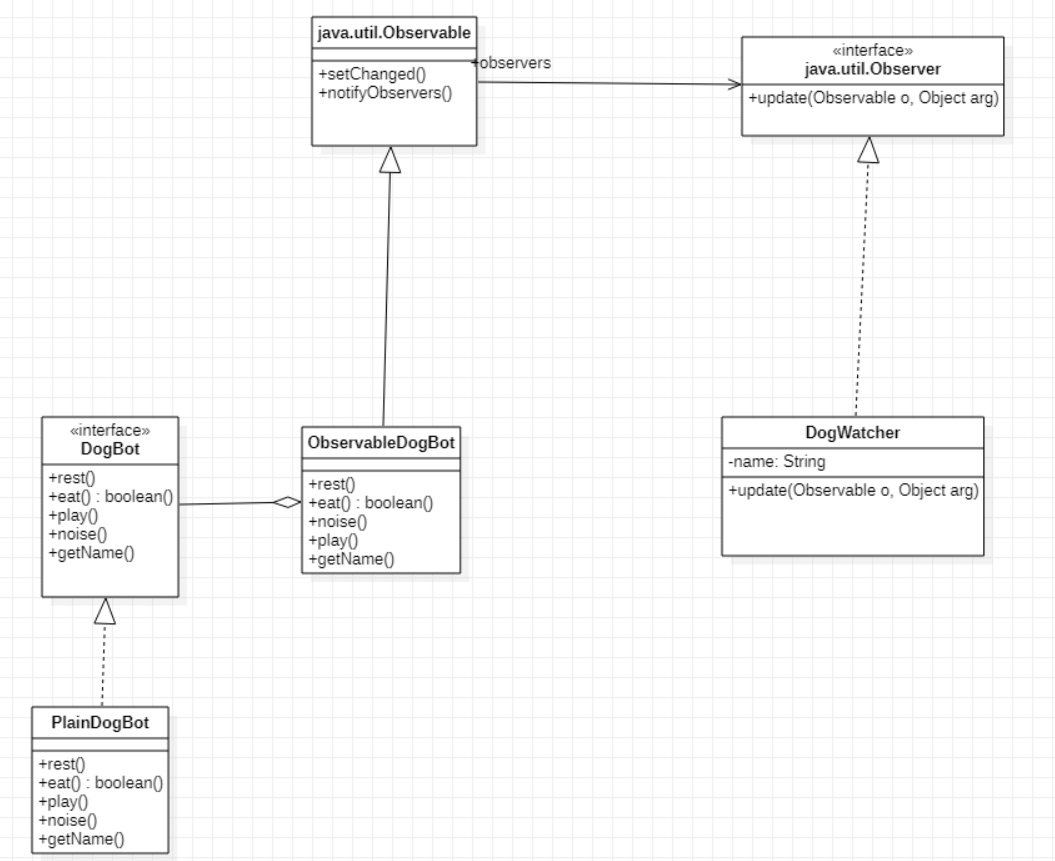
observedRover.rest();

System.out.println("\nObservation Test 1 ends\n");

}

}

UML을 그려보면 다음과 같다

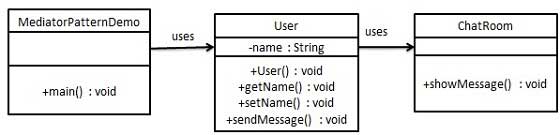


**중재자 패턴**

문제 #4: 중재자 패턴은 여러 개체 또는 클래스 간의 통신 복잡성을 줄이는 데 사용된다. 이 패턴은 일반적으로 서로 다른 클래스 간의 모든 통신을 처리하고 느슨한 결합으로 코드의 유지보수가 쉽도록 지원하는 중개자 클래스를 제공한다.

여러 사용자가 대화방에 메시지를 보낼 수 있는 대화방의 예를 통해 중재자 패턴을 공부해보자. 모든 사용자에게 메시지를 표시하는 것은 대화방의 책임이다. 따라서 ChatRoom과 User 클래스를 만들었다. 사용자 객체는 ChatRoom() 메소드를 사용하여 메시지를 공유한다.

중재자 패턴을 적용하여 아래와 같은 메시지를 출력할 수 있는 프로그램을 작성하라.



[출력]

Thu Jan 31 16:05:46 KST 2021 [Park Seri] : Hi! Park Seri!

Thu Jan 31 16:05:46 KST 2021 [Gildong] : Hello! Gildong!

**제출 방법**

1. 문제 1은 word 파일에 답을 적어 파일을 제출한다.
2. 문제 2, 3는 주어진 파일에 메소드 body 완성 후 실행하고 문제 3은주어진 클라이언트를 사용하라.
3. 원시코드의 헤더 커멘트에 반드시 자신의 이름을 표시하고 간단히 설명하여야 한다. e-Class에 모든 원시코드를 업로드 한다. 파일 이름은 학번과 이름을 넣을 것.

객체지향설계와패턴 *Page 6! of 6!*동국대학교