**동국대학교 공과대학 컴퓨터공학전공**

**CSE 4042 객체지향 설계와 패턴, 중간시험**

### 학 번: \_\_\_\_\_201611218\_\_\_\_\_\_\_\_

### 성 명: \_\_\_\_\_김희수\_\_\_\_\_\_\_\_\_\_\_

### 점 수: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

### 문제 1. 어댑터 패턴

이미 만들어 놓은 MyString 클래스 안에는 greaterThan()과 equals() 두 가지 메소드가 있다. 레거시 시스템을 이용하여 object o1과 o2를 비교하는 compare() 인터페이스를 어댑터 패턴으로 정의하여 클라이언트 프로그램을 작성하고 싶다.

다음 물음에 답하라.

(1) Srting 클래스를 상속한 MyString 클래스가 가지고 있는 두 메소드의 프로토타입은 다음과 같다. 이를 위임을 이용한 어댑터 패턴으로 설계하여 클래스 다이어그램을 그려라(단, 클라이언트는 다음과 같은 형태로 부르고 싶다.(6 점)

<<MyString 레거시 매소드>>

boolean equals(Object o);

Boolean greaterThan(MyString s);

<<클라이언트 프로그램>>

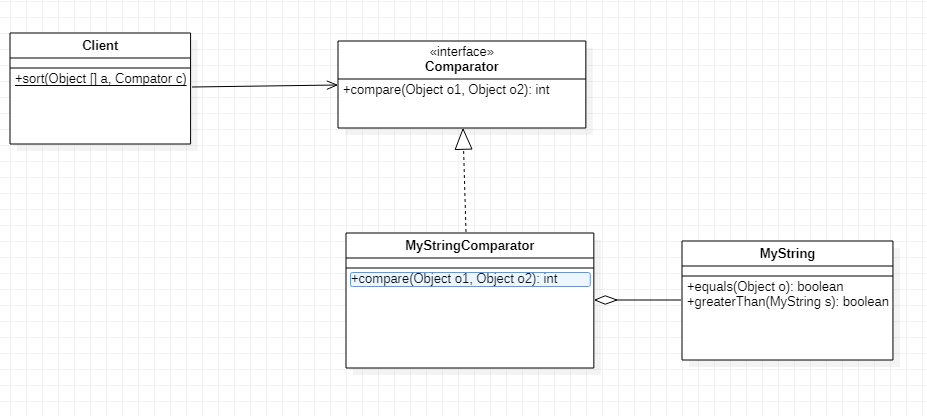
class Array {

static void sort(Object [ ] a, Comparator c);

/\* 여기에서 Comparator 인터페이스가 가진 API를 부르고 싶음. . .

c.compare(o1, o2) 와 같이 호출하고 o1이 02보다 크면 1, 같으면 0, 작으면 -1을 리턴받음. \*/

}



(2) 위에서 설계한 타겟 어댑터 인터페이스(Comparator)와 새 어댑터 클래스(MyStringComparator)를 코딩하라(7점).

public interface Comparator{

boolean compare(Object o1, Object o2);

}

public class MyStringComparator implements Comparator{

private MyString str;

public boolean compare(Object o1, Object o2){

str = o1

if (str.greaterThan((MyString)o2))

return 1;

else if (str.equals(o2))

return 0;

else

return -1;

}

}

**문제 2. 어떤 패턴**

public interface DrawAPI {

public void drawCircle(int radius, int x, int y);

}

public class RedCircle implements DrawAPI {

@Override

public void drawCircle(int radius, int x, int y) {

System.out.println("Drawing Circle[ color: red, radius: " + radius +", x: " +x+", "+ y +"]");

}

}

public class GreenCircle implements DrawAPI {

@Override

public void drawCircle(int radius, int x, int y) {

System.out.println("Drawing Circle[ color: green, radius: " + radius +", x: " +x+", "+ y +"]");

}

}

public abstract class Shape {

protected DrawAPI drawAPI;

protected Shape(DrawAPI drawAPI){

this.drawAPI = drawAPI;

}

public abstract void draw();

}

public class Circle extends Shape {

private int x, y, radius;

public Circle(int x, int y, int radius, DrawAPI drawAPI) {

super(drawAPI);

this.x = x;

this.y = y;

this.radius = radius;

}

public void draw() {

drawAPI.drawCircle(radius,x,y);

}

}

public class PatternDemo {

public static void main(String[] args) {

Shape redCircle = new Circle(100,100, 10, new RedCircle());

Shape greenCircle = new Circle(100,100, 10, new GreenCircle());

redCircle.draw();

greenCircle.draw();

}

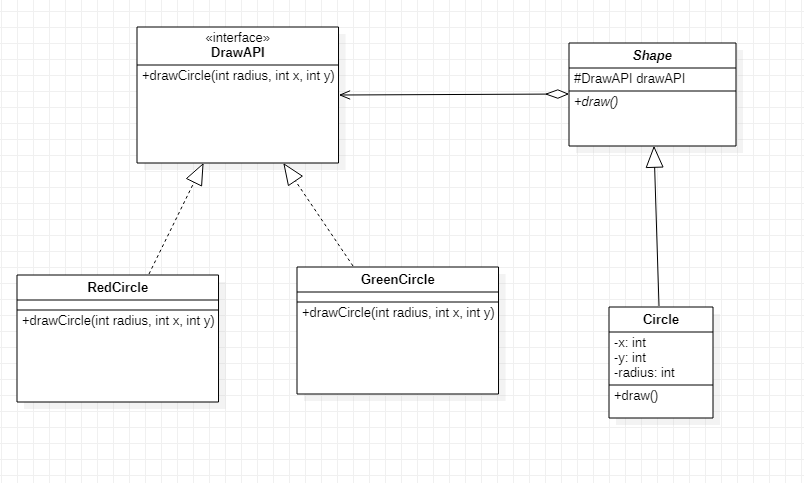
}

(1) 위 프로그램의 실행 결과를 적어라(2점).

Drawig Circle[color: red, radius: 10 , x : 100 , y : 100]

Drawig Circle[color: green, radius: 10 , x : 100 , y : 100]

(2) 위 프로그램의 내용과 일치하는 클래스 다이어그램을 그려라(6점).



(3) 이 프로그램에 사용된 패턴은 무엇인가? 또한 그렇게 판단한 이유를 설명하라(4점).

브릿지 패턴을 사용되었다. 도형을 나타내는 개념적인 계층(Shape)과 그림을 그려주는 구체적인 계층(DrawAPI)을 분리하였기 때문이다.

**문제 3. 어떤 패턴**

성적을 출력하는 기능, 예를 들면 입력된 성적 값을 출력하는 프로그램을 작성하고 있다. 입력된 점수를 저장하는 ScoreRecord 클래스와 점수를 목록의 형태로 출력하는 DataView 클래스를 다음과 같이 설계하였다.



점수의 추가, 즉 ScoreRecord 클래스의 addScore 메소드가 호출되면 ScoreRecord 클래스는 자신의 필드인 scores 객체에 점수를 추가한다. 그리고 DataSheetView 클래스의 update 메소드를 호출함으로써 성적을 출력하도록 요청한다. 또한 DataSheetView 클래스는 ScoreRecord 클래스의 getScoreRecord 메소드를 호출하여 출력할 점수를 구한다. 점수를 출력할 때는 viewCount 만큼만 출력한다.

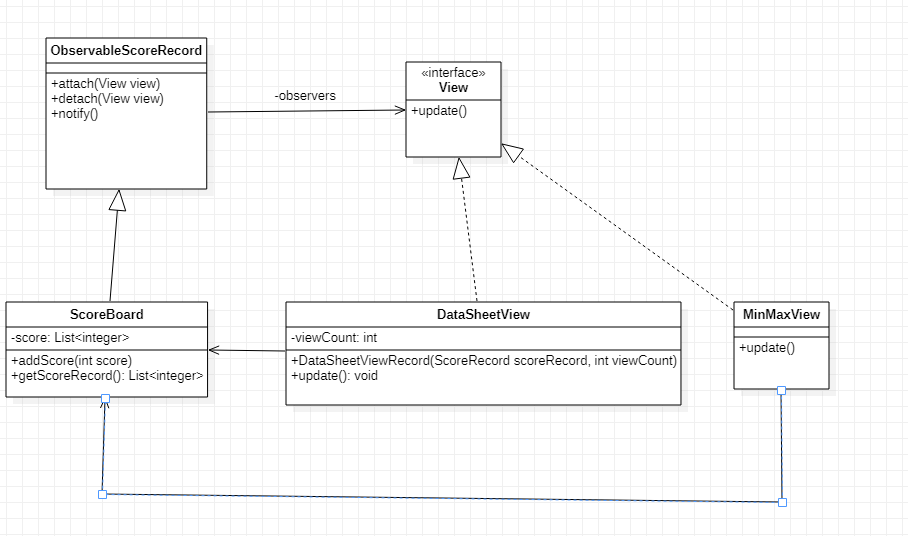
(1) 위 설계에서 성적을 다른 형태, 예를 들면 성적을 목록으로 출력하지 않고 최소/최대 값만 출력하려면 어떻게 하여야 하나? 가장 효과적인 방법을 선택하여 자세히 설명하라(5점).

MinMaxView클래스를 정의한다. MinMaxView클래스에선 update() 메소드를 호출하여 ScoreBoard에서 성적을 요청한다. 받아온 성적에서 최소값을 구해 두개만 출력한다

(2) 성적을 출력하는 형태는 다양하게 하기 위하여 위 설계에 가장 알맞은 패턴은 무엇인가? 그 이유는?(5점)

옵저버 패턴. ScoreBoard의 변경이 자동으로 DataSheetView나 MinMaxView에 통지되어야 하기 때문. ScoreBoard – View 관계가 1대 다 관계가 되고 1대다 관계를 처리할땐 옵저버 패턴이 가장 적절하다.

(3) (2)에서 답한 패턴을 적용하여 성적을 출력하는 시스템의 설계를 클래스 다이어그램으로 그려라(5점).



**문제 4. 디자인 패턴 선택**

(각 1점)

1) 전체와 부분 관계의 구조를 동일하게 취급하게 하려고 할 때 \_\_\_\_\_\_\_컴포지트 패턴\_\_\_\_\_\_\_\_\_\_\_

2) 인스턴스 한 개를 유지하면서 전역적인 접근을 허용하려고 할 때 \_\_\_\_\_\_\_\_\_싱글톤 패턴\_\_\_\_\_\_\_\_\_\_\_\_

3) 어떤 객체와 그 객체의 상태의 변화를 통지 받고 싶은 객체들 사이에 1대다 (1-to-many) 의 관계를 정의하려 할 때

\_\_\_\_\_옵저버 패턴\_\_\_\_\_\_\_\_\_\_\_\_\_\_

4) 복잡한 패키지 안에 손쉬운 API를 제공하려 할 때 퍼싸드 패턴\_\_\_\_\_\_\_\_\_\_\_\_

5) 타겟 객체와 handshaking할 수 있도록 인터페이스를 제공하되 타겟 객체를 손대지 않고 새로운 인터페이스를 정의하려 할 때

\_\_\_\_어댑터 패턴\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

6) 객체들 사이의 다대다(many-to-many) 인터랙션 관계를 캡슐화 하려 할 때 \_\_\_\_\_중재자 패턴\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

7) 구체적인 클래스의 기능을 인터페이스를 구현한 클래스와 독립적으로 만들려고 한다. 그리하여 두 유형의 클래스는 서로 영향을 주지 않고 구조적으로 변경할 수 있게 할 때. \_\_브릿지 패턴\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

8) 호환되지 않는 두 인터페이스 사이의 다리 역할을 하려 할 때. 즉 두 개의 독립적인 인터페이스의 기능을 결합하려고 사용하는 구조적 패턴은 \_\_\_\_\_\_\_\_어댑터 패턴\_\_\_\_\_\_\_\_\_\_\_\_\_

9) 여러 개체 또는 클래스 간의 커뮤니케이션 복잡도를 줄이려 할 때 \_\_\_\_\_\_\_\_중재자 패턴\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

10) 하나의 객체가 수정된 경우 종속 객체에 자동으로 통지되는 것과 같이 객체 간의 일대 다 관계를 가질 때

\_\_\_\_옵저버 패턴\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_