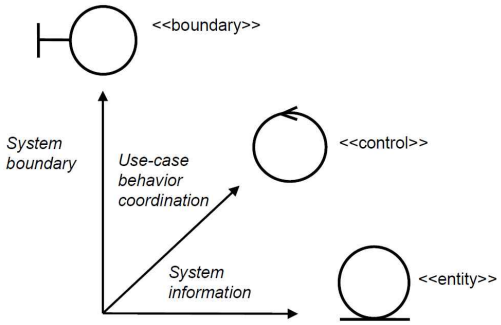
Static

멤버 변수 중 모든 객체에 공통적으로 사용해야 하는 것에 Static 키워드를 붙인다.
Static variable 은 메모리 할당을 한 번만 해서 메모리 사용에 효율적이다.
클래스가 메모리에 할당 될 때 생성되기 때문에 객체를 생성하지 않고도 사용이 가능하다.
static method는 클래스를 통해 호출한다. [객체명.메소드]가 아니라 [클래스명.Static메소드]
instance variable이나 instance method를 사용하지 않는 경우 static 키워드를 사용한다.

MVC Model (ECD 패턴과 매핑)

Model - Entity : System information. 저장영역
View - Boundary : System boundary. 사람과 소통
Controller - Control : Use-case behavior coordination. 제어



Generic 〈 〉

ArrayList 와 같은 자료구조(Collection)에서 구성원의 타입을 제한한다.
장점 : 구성원의 타입을 정의함으로써 의도하지 않은 타입의 객체 저장을 막아준다. -〉 타입안정성

Set (hashSet)

중복을 허용하지 않으며, 중복된 구성원이 들어오면 무시한다. No Duplicate.
하지만, int 나 String 타입은 중복된 값을 Set이 구분 할 수 있지만, 사용자 클래스는 Set이 구분을 하지 못하므로, Override해줘야 하는데, public boolean equals(Object)로 무엇이 중복되면 같은 값인지 알려주고,
public int hashCode()로 return 값을 주어 객체 코드를 주어 같은 객체로 인식할 수 있게 한다.

```
public class Music
{
    private String title;
    private String singer;

    @Override
    public boolean equals(Object m){
        Music music = (Music) m;
        if(this.getTitle().equals(music.getTitle())
                && this.getSinger().equals(music.getSinger()))
        {
            return true;
        }
        else
            return false;
    }

    @Override
    public int hashCode()
    {
        return this.getTitle().hashCode()
                + this.getSinger().hashCode();
    }
```

public int compareTo(클래스명 파라미터)

list를 비교해서 오름차순 정렬(ascending sort)해준다. Comparable〈 〉을 implements해야 한다.

```
public class Music implements Comparable<Music>
{
    private String title;
    private String singer;
    private int favoriteStar;

    @Override
    public int compareTo(Music m) {
        return this.getFavoriteStar() - m.getFavoriteStar();
    }
```

return 값이 (-)이면 앞으로 sort, (+)이면 뒤로 sort.

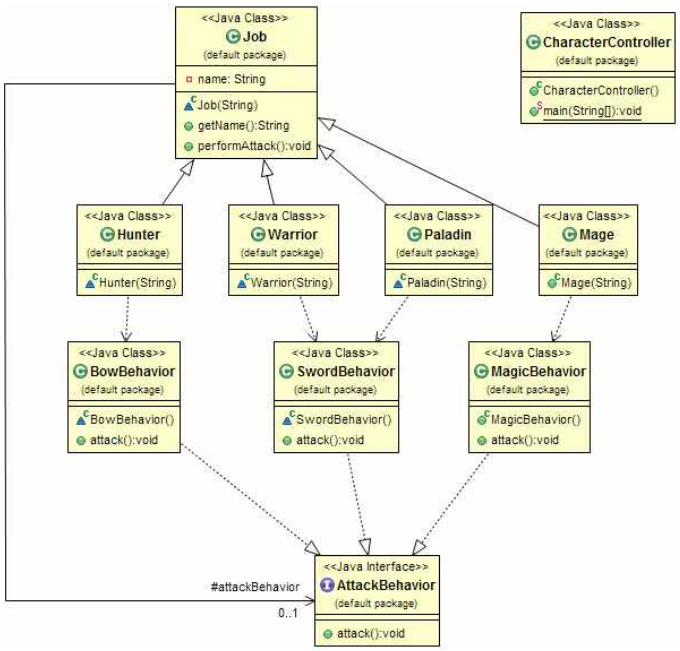
HashMap〈Key(ID), 값(Value)〉

key라는 고유한 값을 부여한 것으로 key로 검색하면 바로 찾을 수 있게!

```
HashMap<String, Music> songMap =
        new HashMap<String, Music>();
songMap.put("빅뱅 하루하루", new Music("하루하루","빅뱅"));
songMap.put("GOD 거짓말", new Music("거짓말","GOD"));

songMap.get("GOD 거짓말");
```

Strategy Pattern

유지보수에 중점. 자주 변경되는 부분을 찾아내서 각각을 캡슐화 한다.
바뀌지 않는 부분에 영향을 주지 않고 바뀌는 부분의 변경이 용이하다.




```java
public class CharacterController {

    public static void main(String[] args) {
        Mage magicion = new Mage("마법사");
        Warrior warrior = new Warrior("전사");
        Hunter hunter = new Hunter("헌터");
        Paladin paladin = new Paladin("팔라딘");

        System.out.println(magicion.getName());
        magicion.performAttack();
        System.out.println("------------");

        System.out.println(warrior.getName());
        warrior.performAttack();
        System.out.println("------------");

        System.out.println(hunter.getName());
        hunter.performAttack();
        System.out.println("------------");

        System.out.println(paladin.getName());
        paladin.performAttack();
        System.out.println("------------");

    }

}
```

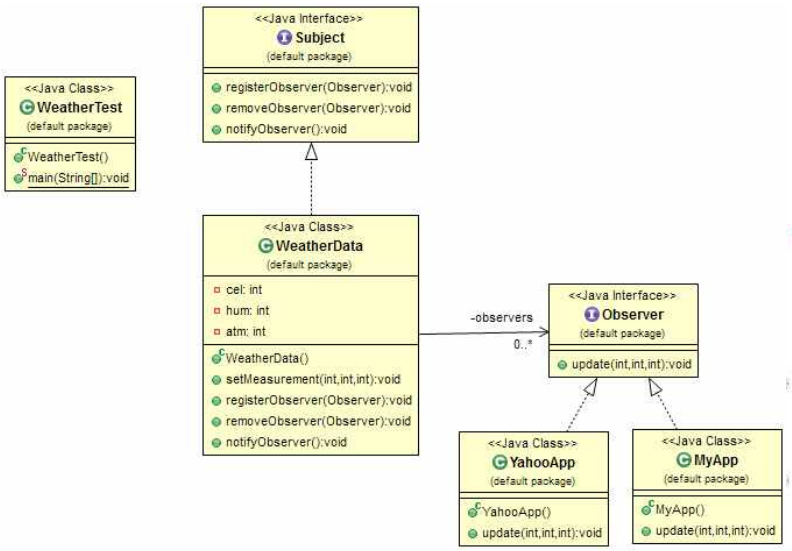
```java
class Job {
    private String name;
    protected AttackBehavior attackBehavior;

    Job(String characterName){
        this.name = characterName;
    }

    public String getName(){
        return this.name;
    }

    public void performAttack(){
        this.attackBehavior.attack();
    }

}
```

```java
public class Mage extends Job{
    public Mage(String name){
        super(name);
        this.attackBehavior = new MagicBehavior();
    }
}
```

```java
public class MagicBehavior implements AttackBehavior{
    public void attack(){
        System.out.println("Magic Attack!");
    }
}
```

Observer Pattern

1:N 의존성을 정의한다. 한 객체의 상태가 바뀌면 그 객체를 Listen하고 있는 객체들에게 자동으로 내용 갱신.
Data를 다루고 있는 클래스가 Listen하고 있는 클래스의 영향을 받지 않는다. 재컴파일이 필요없다.



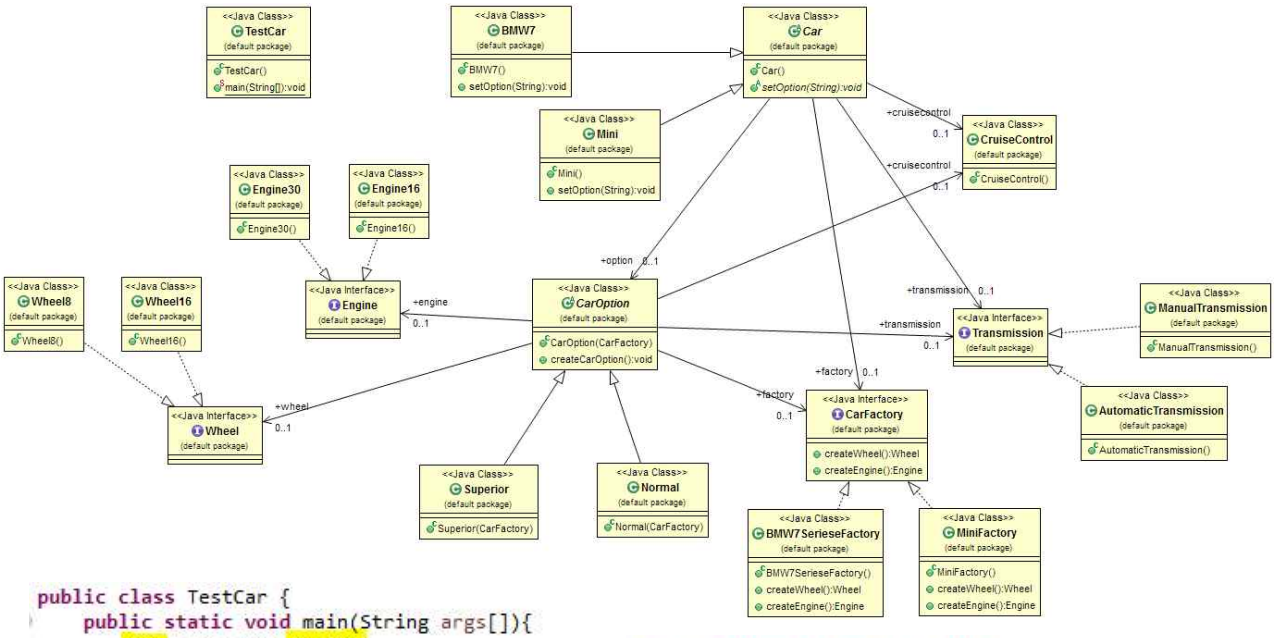

```java
public class WeatherTest {

    public static void main(String[] args) {
        WeatherData weatherData = new WeatherData();

        YahooApp yahooApp = new YahooApp();
        MyApp myApp = new MyApp();
        weatherData.registerObserver(yahooApp);
        weatherData.registerObserver(myApp);

        weatherData.setMeasurement(25, 60, 30);
        weatherData.notifyObserver();
        weatherData.setMeasurement(27, 55, 32);
        weatherData.notifyObserver();

    }
}
```

```java
public class YahooApp implements Observer{
    @Override
    public void update(int t, int h, int p) {
        System.out.println("YahooApp : " + t + "ㅌ/" + h + "%");
    }
```

```java
public class WeatherData implements Subject{
    private ArrayList<Observer> observers;
    private int cel;
    private int hum;
    private int atm;

    public WeatherData(){
        observers = new ArrayList<Observer>();
    }

    public void setMeasurement(int cel, int hum, int atm){
        this.cel = cel;
        this.hum = hum;
        this.atm = atm;
    }

    @Override
    public void registerObserver(Observer o) {
        observers.add(o);
    }

    @Override
    public void removeObserver(Observer o) {
        observers.remove(o);

    }

    @Override
    public void notifyObserver() {
        for(int i=0; i<observers.size(); i++){
            Observer o = observers.get(i);
            o.update(this.cel, this.hum, this.atm);
    }
```

Factory Pattern

객체생성(인터페이스) - 객체 내용 서브 클래스. 객체를 생성하기 위한 인터페이스를 정의하면,
인터페이스에 따라 어떤 객체가 생성되는지 서브 클래스에서 정의한다.
Factory가 없으면 모든 종류마다 구성을 직접 해야 하기 때문에 오류 가능성이 높아진다.




```
public class TestCar {
    public static void main(String args[]){
        Car car = new BMW7();
        car.setOption("superior");
    }
}
```

```
public abstract class Car {
    public CarOption option;
    public CarFactory factory;
    public CruiseControl cruisecontrol;
    public Transmission transmission;

    public Car(){

    }
    public abstract void setOption(String type);
}
```

```
public abstract class CarOption {
    public CarFactory factory;
    public Transmission transmission;
    public CruiseControl cruisecontrol;
    public Wheel wheel;
    public Engine engine;

    public CarOption(CarFactory factory){
        this.factory = factory;
        createCarOption();
    }


    public void createCarOption(){
        this.engine = factory.createEngine();
        this.wheel = factory.createWheel();
    }
}
```

```
public class BMW7 extends Car{

    @Override
    public void setOption(String type) {
        if(type.equals("superior")){
            this.option = new Superior(new BMW7SerieseFactory());
        }
        else if(type.equals("normal")){
            this.option = new Normal(new BMW7SerieseFactory());
        }
    }
```

```
3  public class BMW7SerieseFactory implements CarFactory{
4
5      @Override
6      public Wheel createWheel() {
7          System.out.println("10인치 휠 장착");
8          Wheel w = new Wheel16();
9          return w;
10     }
11
12     @Override
13     public Engine createEngine() {
14         System.out.println("3.0 기본 엔진 장착");
15         Engine e = new Engine30();
16         return e;
17     }
```