Design Pattern

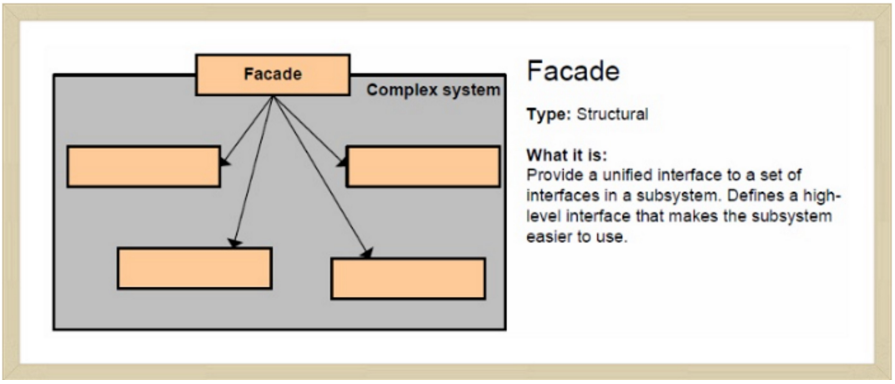
디자인패턴

Type

- 생성 : 객체 인스턴스 생성을 위한 패턴
- 구조 : 클래스 및 객체들의 구성을 통해 더 큰 구조로 만들 수 있게 해준다.
- 행동 : 클래스와 객체들이 상호작용하는 방법 및 역할을 분담하는 방법 정의

Pattern Facade

- **퍼사드는 하나의 클래스, 메소드와 관련된 일련의 기능들을 모아 퍼사드 클래스로 선언한다.**
- 퍼사드는 소프트웨어 라이브러리를 쉽게 사용할 수 있게 해준다. 퍼사드는 공통적인 작업에 대해 간편한 메소드들을 제공해준다.
- 퍼사드는 좋게 작성되지 않은 API의 집합을 하나의 좋게 작성된 API로 감싸준다.

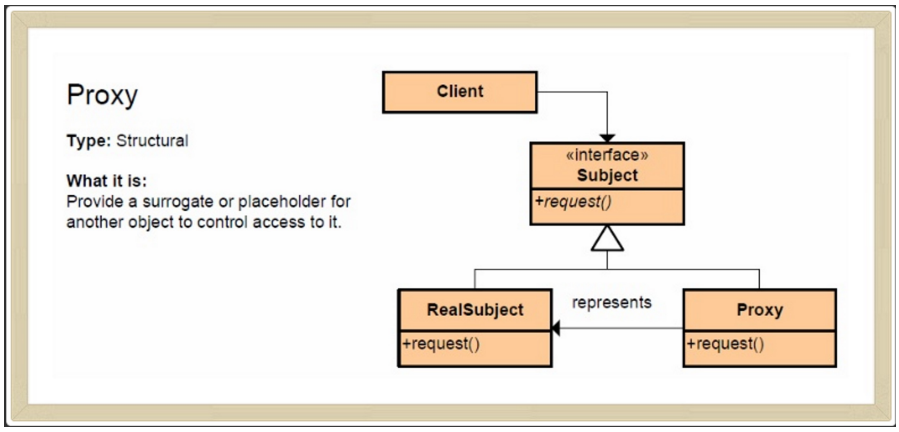


```
public class Computer {
     public void turnOn(){ System.out.println("Turn On Computer~!"); }
     public void turnOff(){ System.out.println("Turn Off Computer~!"); }
}

public class Door {
     public void open(){ System.out.println("Door Open~!"); }
     public void close(){ System.out.println("Door Close~!"); }
}

public class Light {
     public void turnOn(){ System.out.println("Turn On Light~!"); }
     public void turnOff(){ System.out.println("Turn Off Light~!"); }
}

public class FacadeClass {
     Computer computer = new Computer();
     Door door = new Door();
     Light light = new Light();

     // 수업 시작
     public void start(){
          System.out.println("수업 시작~!");
          door.open();
          light.turnOn();
          computer.turnOn();
     }

     // 수업 종료
     public void end(){
          System.out.println("수업 종료~!");
          computer.turnOff();
          light.turnOff();
          door.close();
```

```
	}
}

public class FacadeMain {
	public static void main(String[] args) {
		FacadeClass facade = new FacadeClass();

		facade.start();
		facade.end();
	}
}
```

Pattern Proxy

- 대리자로써 일을 맡기면 그 일을 처리하고, 완료되면 그 결과를 알려주는 패턴
- **실제 호출하고자 하는 객체의 상태를 유지한채 Proxy 객체가 부가적인 일을 해준다.**



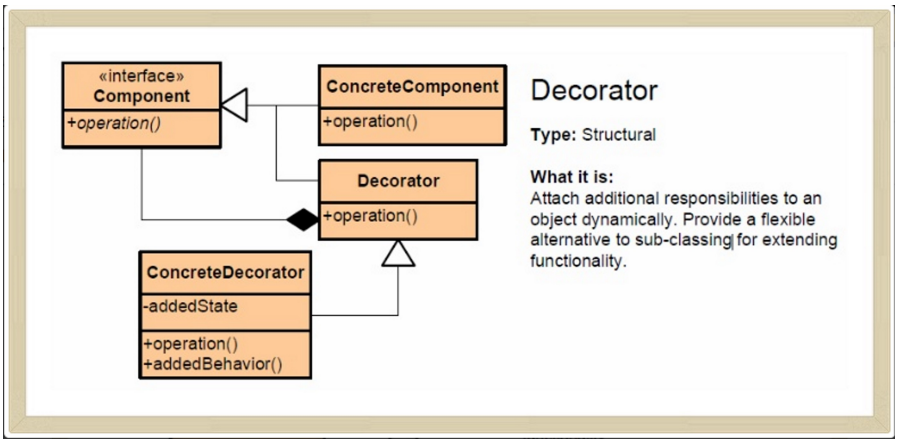
- Proxy : MVC → Controller
- RealSubject : MVC → View
- Proxy가 RealSubject 객체를 가지고 있다.

```
interface ProxyInterface { public String read(); }

public class Board implements ProxyInterface{
	@Override
	public String read(){ return "글 내용"; }
}

public class ProxyBoard implements ProxyInterface{
	Board board;

	@Override
	public String read() {
		System.out.println("proxy : 조회수 1증가");
		board = new Board();
		return board.read();
	}
}

public class ProxyMain {
	public static void main(String[] args) {
		// 실제 결과값만 던져주는 객체
		ProxyInterface board = new Board();
		System.out.println(board.read());
```

```
        // 실제 결과 값에 부가적인 일을 해주는 Proxy 객체
        ProxyInterface proxyBoard = new ProxyBoard();
        System.out.println(proxyBoard.read());
    }
}
```

Pattern Decorate

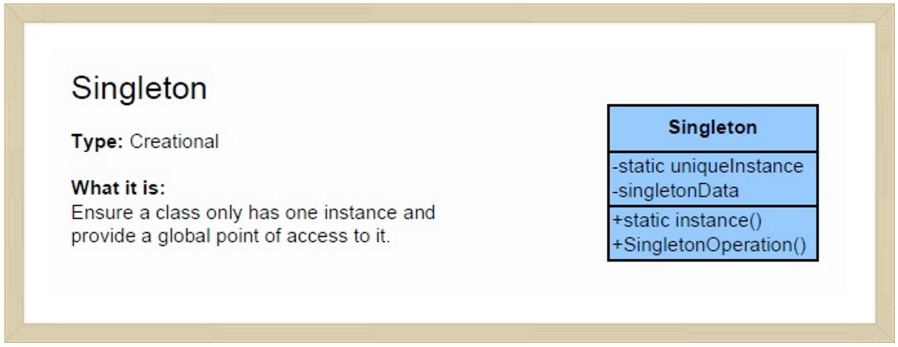
- **실제 호출하고자 하는 객체의 결과값에 Decorator 객체의 부가적인 일이 결과값에 영향을 미친다.**



```
interface DecorateInterface { public String read(); }

public class Board implements DecorateInterface {
    @Override
    public String read() { return "글 내용"; }
}

public class DecorateBoard implements DecorateInterface{
    Board board;

    @Override
    public String read() {
        board = new Board();
        return board.read() + " : content";
    }
}

public class DecoMain {
    public static void main(String[] args) {
        DecorateInterface board = new Board();
        System.out.println(board.read());

        DecorateInterface decoBoard = new DecorateBoard();
        System.out.println(decoBoard.read());
    }
}
```

Pattern Singleton

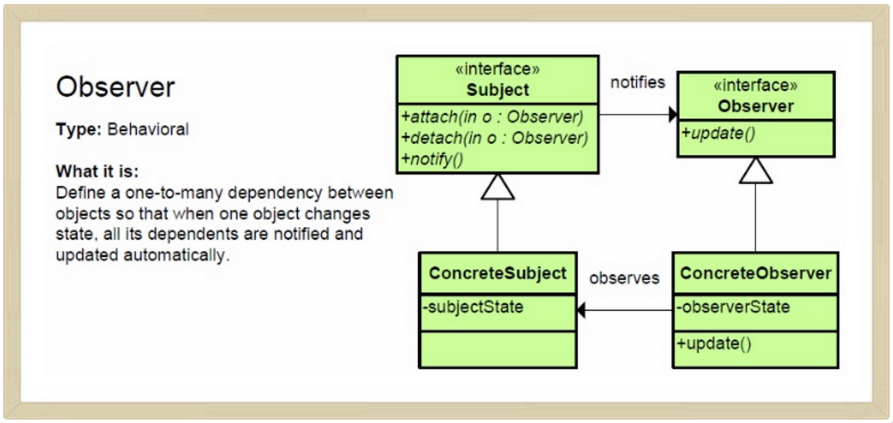
- **무분별한 객체생성을 방지하고, 하나의 객체만 생성하여 이용하는 패턴**



```
public class Singleton {
	// 외부에서 Instance에 접근하지 못하도록 private 처리
	private static Singleton instance = null;
	private int num = 0;

	// 외부에서 Instance를 생성하지 못하도록 생성자 private 처리
	private Singleton(){ }

	public static Singleton getInstance(){

		// instance가 null이면 새로 생성
		// instance가 기존에 있었으면 기존 instance 반환

		if(instance == null){
			instance = new Singleton();
		}

		return instance;
	}

	public void Show(int n){
		num += n;
		System.out.println(num);
	}
}

public class SingletonMain {
	public static void main(String[] args) {
		Singleton single1 = Singleton.getInstance();
		Singleton single2 = Singleton.getInstance();
		Singleton single3 = Singleton.getInstance();

		single1.Show(1);
		single2.Show(2);
		single3.Show(3);
	}
}
```

Pattern Observer

- **한 객체의 상태가 바뀌면 그 객체에 의존하는 다른 객체 모두에게 일방적으로 통지가 가고 자동으로 내용이 갱신되는 패턴. 1 : N(다)의 의존성으로 정의한다.**



```
// 이 인터페이스를 구현하는 클래스는 감시역활을 하게된다는 뜻의 Marker 역활의 인터페이스
public interface Observer {
     // 옵저빙을 할 타겟이 필요

     // 타겟의 변화 감지
     public void Update(int msg);
}

// 감시를 당할 Subject 객체
class Subject {
     // 나를 감시할 Observer 인터페이스들
     ArrayList<Observer> observers = new ArrayList<>();

     int number = 0;

     // 나를 감시할 Observer 추가
     public void attach(Oberserver observer){
          Observers.add(observer);
     }

     // 나를 감시하는 Observer 제거
     public void detach(){
          int idx = observers.indexOf(observer);
          observers.remove(idx);
     }

     // 실행태스크에 값을 입력하고— 변경사항이 있음을 알리는 메서드를 호출한다,
     public void setNumber(int n) {
          number = n;
          notifyChanged();
     }

     // 실행태스크에 변경사항이 있으면 전체 옵저버에 알려준다.
     private void notifyChanged(){
          for(Observer o : observers)
               o.Update(number);
     }
}

// 이 클래스가 마킹역활의 Observer Interface 구현하므로 실제적인 감시역활을 하게된다.
public Listener implement Observer {
     public Listener(Targer target){
          Target.attach(this);
     }
```

```
	// 타겟의 변화감지
	@Override
	public void Update(int msg) {
		System.out.println("Binary : "+Integer.toBinaryString(msg));
	}
}

public class ObserverMain {

	public static void main(String[] args) {

		TargetJob target = new TargetJob();
		new Listener(target);
		//Targer.attach(new Listener());

		Scanner scan = new Scanner(System.in);

		while(true){
			System.out.println("숫자를 입력하세요");
			target.setNumber(scan.nextInt());

		}
	}
}
```
