****

**디자인 패턴**

**패턴**

패터은 어떠한 방법론-> 구현을 해야될 어떠한 설계를 할것인가(설계기법)? GOF(Gang of FOUR)

**MVC**

M:Model 모델

* 데이터, 로직(Business Logic), Prosses

V:View 뷰

* 보여주는 양식

C:Controller 컨트롤러

* 어떤 뷰로 보여줄지.

**객체의 책임과 크기**

객체마다 자신만의 책임(responsibility)가 있다는 의미를 갖는다.

객체는 단일 책임의 원칙을 지켜야한다

OOP

상속, 은익, 다형

**Factory Pattern(팩토리 패턴)**

특정 클래스를 만들어주는 공장 즉 만들어주는 또다른 곳을 만들어서 사용해라

즉 객체를 호출할 때 또는 생성할 때 바로 불러다가 쓰지 말어라.

왜냐면. 바로 쓰게되면 클래시간 연관관계가. 강하다. (String relationship)

중간에 팩토리에 객체생성을 담당하는 책임을 가지는 놈을 만든다. 객채생성을 위임한.

1. 객체를 생성하는 클래스를 따로만들어라(객체 생성을 위임하여라)

2. loosing coupling 강한 연관관계가 맺어지지 않는다.

대표적으로 팩토리패턴을 구현해야될 것은.. 불러다 사용할 객체에따라 달라지는데.

보통 그냥 가져다쓰면 패턴을 적용 시키지 않아도 되지만.

시스템적으로 로우레벨에 관련된것들 데이터베이스 커넥션/OI 등등 이런것들은 팩토리패턴을 사용하는 것이 좋다.

**Object Oriented**

상속, 은익, 다형석

OOP의 4대 기본원칙

Object Oriented Programming Basic Principle

1. 추상화(Abstraction)

객체들의 공통적인 특징(속성, 기능)을 뽑아 이름을 붙이는 것.

2. 캡슐화(Encapsulation)

특정 객체가 독립적으로 역할을 제대로 수행하기 위해 필요한 데이터와 기능을 하나로 묶은 것. (모듈화의 의미)

정보를 객체 안에 포함시키고, 그 정보에 대한 직접 접근은 허용하지 않는 대신, 필요에 따라 확인할 수 있는 인터페이스를 외부에 공개하는 방식.

3. 상속(Inheritance)

상위 개념의 특징을 하위 개념이 물려받는 것.

4. 다형성(Polymorphism)

모듈이 갖고있는 정체성과 표현방식.

다형성을 극대화 하기 위해 추상클래스나 인터페이스를 이용한다.

* 오버로딩
* 오버라이딩
* 이기종 집합
* 다형적 변수

상속

공통적인 기능들은 상위 부모 클래스에 모아놓고

자식 클래시는 자기자신들의 특이한 기능을 따로 지정한다.

은익

속성값을 다이렉트로 호출을 하는 것을 막고.

특정한 행위를 가지고 접근한다.

다형석

오버로딩 : 메서드 인자값에 따른 메서드호출

오버라이딩 : 상위클래스의 가진 메서드를 덮어버린다.

이기종집합 : isa관계로 묶여진 것들은 상위클래스로 볼수있다 다운캐스팅

**디자인 패턴**

1. 절차지향 : 프로시저(procedure)로 프로그램을 구성하는 기법을 절차 지향(Procedural Oriented)프로그래밍

2. 객체지향 : Object Oriented 객체라고 불리우는 단위로 묶는다. 그객체들끼리 모여서 사로 상호작용하면서 만들어지는 프로그램 객체지향적인 프로그래밍을 한다.

**객체(Object)**

객체는 데이터와 **그데이터를 조작하는 프로시져(오퍼레이션,메서드,함수)**로 구성된다.

**객체가 제공하는 기능을 오퍼레이션(operation)이라고 부른다**

오퍼레이션의 사용법은 일반적으로 다음과 같이 세개로 구성된다 이세가지를 합쳐서 시그너처 라고 부른다

**오퍼레이션 사용방법 -> 시그너쳐(Signature) 구성**

1. 기능 식별 이름

2. 파라미터 및 파라미터 타입

3. 기능 실행 결과 값

**객체의 책임과 크기**

객체는 객체가 제공하는 기능으로 정의됩니다.

다시말하면 객체마다 자신만의 책임(responsibility)이 있다는것입니다.

객체지향적 프로그래밍하다보면 어려운 것이 객체마다 기능을 할당하는 과정이 아닐까 싶습니다.

**단일책임원칙(Single Responsibility Principle:SRP)**

객체는 자기자신만의 책임이 있는데.. 그책임은 단일이어야합니다.

객체는 단 한 개의 책임만을 가져야 한다.

**의존(dependency)**

한 객체가 다른 객체를 생성하거나 다른 객체의 메서드를 호출할 때 이를 그 객체의 의존(dependency) 한다고 한다.

의존은 영향은 꼬리에 꼬리를 문 것처럼 전파되는 특징이 있다.

C클래스가 B클래스에 의존하고 B클래스가 A클래스에 의존한다고 하면 C가 A클래시까지 영향을 줄수 수있일수 있다.

C

B

A

또한 의존의 이런 특징 때문에 의존이 순화해서 발생할 경우 다른 방법이 없는지 고민해야한다.

A클래스를 변경하면 C클래스의 변화는 다시 A클래스에 영향을 줄수 있다..이는 결국 A클래스 변경한 여파가 다시 A클래스의 변화를 준다.. **즉 순환의존의 이런 특징 때문에 .. 의존 역전 원칙 (DIP)가 있다.**

**캡슐화(encapsulation)**

객체 지향의 장점은 한곳의 구현 변경이 다른곳에 변경을 가하지 않도록 해준다는데 있다.

데이터 중심적인 프로그래밍 하지 말고

기능 중심적 프로그래밍을 하면 된다 (메서드로 빼라)

기능 구현을 캡슐화하면 내부 구현이 변경되더라도, 기능을 사용하는 곳의 영향을 최소화할 수 있다.

**캡슐화를 위한 두개의 규칙**

**1. Tell, Don’t Ask**

**- 데이터를 물어보지 않고 기능을 실행해달라고 하는 규칙**

**2. 데미테르법칙(Law of Demeter)**

**- 메서드에서 생성한 객체의 메서드만 호출**

**- 파라미터로 받은 객체의 메서드만 호출**

**- 필드로 참조하는 객체의 메서드만 호출**

**객체지향 설계 과정**

1. 제공해야 할 기능을 찾고 또는 세분화하고 그기능을 알맞은 객체에 할당한다.

- 기능을 구현하는데 필요한 데이터를 객체에 추가 한다 객체에 데이터를 먼저 추가하고 그데이터를

이용하는 기능을 넣을수 있다.

- 기능은 최대한 캡슐화해서 구현한다.

2. 객체 간에 어떻게 메시지를 주고 받을 지 결정한다.

3. 과정 1과 과정2를 개발하는동안 반복한다.

**다형성과 추상타입**

다형성이란 한객체가 여러 타입을 가질수 있다는 것을 뜻한다. (IsA인관계)

**추상화(abstraction)**

데이터나 프로세스 등을 의미가 비슷한 개념이나 표현으로 정의하는 과정

변화가있는것들을 추상화하여라 추상화하여라 행위로 바꿔라.

**재사용: 상속보단 조립**

**상속에의한 단점**

1. 부모클래스의 변경에 의해 자식 클래스가 변경될수가있다.

클래스를 상속받는다는 것은 그 클래스에 의존한다는 뜻이다. 따라서 의존하는 클래스의 코드가 변경되면 영향을 받을수 있는것이다. 부모클래스가 변경여파가 자식클래스들에게 (하위클래스들에계 계층도처럼) 전파되는것이다

2. 유사한 기능을 확장하는 과정에서 클래스의 개수가 불필요하게 증가할수 있다.

3. 상속오용 : ISA 관계가 아니라 HasA 관계과되면 안된다. 사용자가 나도모르게 상위클래스 즉 부모클래스의 오퍼레이션을 가져다 사용하므로써 현재 사용객체의 영향을 줄수가 있다.

**조립을 이용한 재사용**

위처럼 객체상속으로 모든 기능을 구현하려면

예로 파일을 읽어와 암호화하는 기능을 위해 파일읽은 기능을 상속하여 암호화하는부분을 가져다쓰는거와다르게

조립을 이용하면 파일불러오는 객체 불러와서 조립하고 그뒤에 암호화하는부분객체를 가져다가 조립하여

사용하면 훨신 좋다는것이다. (불필요한 클래스도 안늘고)

**위임(delegation)**

내가할일을 다른 객체에게 넘긴다는 의미를 담고있으며 보통 조립방식을 이용해서 위임을 구현한다.

객체안에서 또다른 객체를 사용하여 내가 원하는 기능을 사용한다.

위임을 상ㅇ하면 내가 바로 실행할 수 있는걸 다른객체에게 한번더 요처앟게된다. 이과정에서 메서드 호출이 추가 되기 때문에 실행 시간은 다소 증가한다. 연산속도가 매우 중요한 시스템에서는 많은 위엄 코드가 성능에 문제를 일으킬수 있지만 대부분 경우에는 위임으로 인한 발생하는 성능 저하보다 위임을 통해 얻는 유연함/재사용의 장점이 더크다.

**상속은 언제 사용하나?**

명확한 IS-A 관계가 성립되어야 한다

HS-A관계일때는 위임방식을 권한다.

**설계원칙/DI와 서비스 로케이터**

**설계원칙:SOLID**

1. 단일 책임 원칙(Single responsibility principle:SRP)

2. 개방-폐쇄 원칙(Open-closed principle:OCP)

3. 리스코프 치환 원칙(Liskov substitution principle:LSP)

4. 인터페이스 분리 원칙(Interface sefreaation principle:ISP)

5. 의존 역전 원칙(Dependency inversion principle:DIP)

**1. 단일 책임 원칙(Single responsibility principle:SRP)**

객체를 객체로 존재하는 이유가 책임인데, 단일책임 원칙은 이책임과 관련된 원칙이다.

클래스가 여러 책임을 갖게되면 그 클래스는 각 책임마다 변경되는 이유가 발생하기 때문에 클래스가 한 개의 이유로만 변경되려면 클래스는 한 개의 책임만을 가져야 한다 이러한 이유로 이원칙은 다른 말로 “클래스를 변경하는 이유는 단 한 개여야 한다.” 고도 표현한다.

만약에 하나의 객체안에

데이터를 html으로 불러와 -> 파싱후 -> 화면에 보여주는 클래스에 다있다면

데이터불러오는기능변경시 html에서 소캣통신으로 바뀐다면

책임이 데이터 불러오는 부분도 있기 때문에 변경이 이루어져야한다.

그렇기 때문에 클래스는 하나의 책임만 가지고있어야한다.

**2. 개방-폐쇄 원칙(Open-closed principle:OCP)**

확장에는 열려 있어야 하고 변경에는 닫혀 있어야 한다.

기능을 변경하거나 확장할 수 있으면서

그 기능을 사용하는 코드는 수정하지 않는다.?

FlowController

<<Interface>>

ByteSource

SocketByteSource

FileByteSource

byte를 읽어오는 기능을 추가해야할 경우 ByteSource인터페이스를 상속받은 MemoryByteSource클래스를 구현함으로써 기능 추가가 가능하다 그리고 새로운 기능이 추가 되었지만 이새로운 기능을 사용할 FlowController클래스의 코드는 변경되지 않는다.

이를 개방 폐쇄원칙은(사용되는 기능의) 확장에는 열려있고(기능을 사용하는 코드의) 변경에는 닫혀 있다고 한다.

**3. 리스코프 치환 원칙(Liskov substitution principle:LSP)**

앞서 개방 폐쇄 원칙은 추상화와 다형성(상속)을 이용해서 구현하였는데 리스코프 치환 원칙은 개방 폐쇄원칙을 받쳐 주는 다형성에 관현 원칙을 제공한다 리스코프 치환 원칙은.

\* 상위(부모) 타입의 객체를 하위(자식) 타입 객체로 치환해도 상위타입사용하는 프로그램은 정상작동되야한다.

위를 다시말해

자식클래스를 부모클래스로 다형성에의해 치환된후

부모의 오퍼레이션을 사용하여도 프로그램은 정상적으로 작동이되어야한다.

예를들어 직사각형 정사각형 문제가 있다고 치자.

사각형은 부모클래스

정사각형은 자식클래스

나중에..이건좀더 이해되고나서.. 머리로는 이해되는데 글로는 잘안써진다.

**4. 인터페이스 분리 원칙(Interface sefreaation principle:ISP)**

인터페이스는 그 인터페이스를 사용하는 클라이언트를 기준으로 분리된다.

사용에 맞게 인터페이스를 가지고 만들어서 사용해야지

하나의 인터페이스로 뭐든 것을 다 덮어버릴려면 그 인터페이스가 변경됐을 때 뭐든

클래스들이 재컴파일 되어야 한다는 문제가 발생된다.

이것은 클래스의 **단일책임원칙과 도 연결이된다**

**5. 의존 역전 원칙(Dependency inversion principle:DIP)**

\* 고수준 모듈은 저수준 모듈의 구현에 의존해서는 안 된다, 저수준 모듈이 고수준 모듈에서 정의한 추상 타입에 의존해야한다?

|  |  |
| --- | --- |
| 고수준 모듈 | 저수준모듈 |
| 바이트 데이터를 읽어와  암호화하고 결과 바이트 데이터를 쓴다 | 1. 파일에서 바이트 데이터를 읽어온다.  2. AES 알고리즘으로 암호화환다  3. 파일에 바이트 데이터를 쓴다. |

고수준 모듈은 상대적으로 큰틀(즉, 상위수준)에서 프로그램을 다룬다면, 저수준 모둘은 각 개발 요서 (즉, 상세) 가 어떻게 구현될지에 대해서 다룬다 프로젝트 초기에 요구사항이 어느 정도 안정화 되면 이후부터 큰 틀에서 프로그램이 변경되기 보다는 상세 수준에서의 변경이 발생할 가능성이 높아진다.

**6. DI(Dependency Injection) 와 서비스 로케이터**

**..다음이시간에**

**주요 디자인패턴**

**전력적 패턴**

Calculator

<<interface>>

DiscountStrategy

FirstGuestDisconuntStrategy

NonGuestDisconuntStrategy

Calculator에서는 DiscontStrategy를 사용하지만

전략적으로 First 또는 Non 객체가 들어오기 때문에

계방 폐쇄원칙을 따른다.

|  |
| --- |
| **package** designpattern;  **interface** Connection{  **void** say();  }  **class** ByteConnection **implements** Connection{  @Override  **public** **void** say() {  System.*out*.println("ByteSay");  }  }  **class** HtmlConnection **implements** Connection{  @Override  **public** **void** say() {  System.*out*.println("HtmlSay");  }  }  **public** **class** StrategyPattern {  **public** **static** **void** main(String[] args) **throws** InterruptedException {  Connection c = **null**;  c = *getConnection*();  c.say();  Thread.*sleep*(1);  c = *getConnection*();  c.say();  Thread.*sleep*(1);  c = *getConnection*();  c.say();  Thread.*sleep*(1);  c = *getConnection*();  c.say();  }  **public** **static** Connection getConnection(){ //팩토리클래스를 따로두어 처리하면 팩토리 패턴이 먹힌다.  **long** s = System.*currentTimeMillis*();  **if**(s%2==0){  **return** **new** ByteConnection();  }**else**{  **return** **new** HtmlConnection();  }  }  } |
| 극단적인 단편적인 예코딩이다. |

**템플릿 메서드(Template Method)**

프로그램을 구현하다 보면 완전히 동일한 절차를 가진 코드를 작성하게 될 때가 있다 심지어 이코드들은 절차 중 일부 과정의 구현만 다를뿐 나머지의 구현은 똑같을 때도 있다

1. 상위 클래스는 실행 과정을 구현한 메서드**(템플릿메서드)**를 제공한다. 별도의 추상클래스로 뺀다

**상위클래스 흐름제어**

템플릿 메서드 패턴의 특징은 하위 클래스가 아닌 상위 클래스에서 흐름 제어를 한다는 것이다.

일반적인 경우 하위 타입이 상위 타입의 기능을 재사용할지 여부를 결정하기 때문에 흐름제어는 하위 타입이 하게된다.

템플릿 메서드 패턴에서는 상위 타입의 템플릿메서드가 모든 실행 흐름을 제어하고 하위 타입의 메서드는 템플릿 메서드에서 호출되는 구조를 가진다

다시말해 훅패턴은 아니지만

사용자가 템플릿메서드를 호출하고 호출한 템플릿 메소드에서 추상메서드로 해놓은

사용자들이 구현해놓은 메서드를 호출하여 처리한다.

이렇게 상위 클래스에서 실행 시점이 제어되고 기본 구현을 제공하면서 하위 클래스에서 알맞게 확장할수 있는 메서드를 훅(hook)메서드라고 부른다.,

|  |
| --- |
| **package** designpattern;  **abstract** **class** Logic{  **void** login(String id, String pwd) **throws** Exception{  **if**("VISUAL".equals(id) && "VISUAL".equals(pwd)){  start(id);  }**else**{  **throw** **new** Exception("로그인실패");  }  }  **abstract** **void** start(String id);//훅메서드  }  **class** ByteLogic **extends** Logic{  @Override  **void** start(String id) {  System.*out*.println("ByteLogic Login Welcom "+id);  }  }  **class** HtmlLogic **extends** Logic{  @Override  **void** start(String id) {  System.*out*.println("HtmlLogic Login Welcom "+id);  }  }  **public** **class** TemplateMethodPattern {  **public** **static** **void** main(String[] args) **throws** Exception {  **new** ByteLogic().login("VISUAL", "VISUAL");  **new** HtmlLogic().login("VISUAL", "VISUAL");  **new** HtmlLogic().login("VISUAL", "VISUALGSADES");  }      } |

**상태패턴(State Pattern)**

클래스내부에 상태를 가진 객체를 두고

행위에따라 클래스내부에있는 상태프로퍼티 값을 변경하는 것.

|  |
| --- |
| **package** designpattern;  **abstract** **class** State {  StatePattern caller = **null**;  **public** State(StatePattern caller) {  **this**.caller = caller;  }  **abstract** **public** **void** nextLevel();  }  **class** DefaultState **extends** State{ //point 0~10  **public** DefaultState(StatePattern caller) {  **super**(caller);  }  @Override  **public** **void** nextLevel() {  caller.setState(**new** Level1State(caller));  }  **public** String toString() {  **return** "DEFAULT";  }  }  **class** Level1State **extends** State{ //point 11~40  **public** Level1State(StatePattern caller) {  **super**(caller);  }  @Override  **public** **void** nextLevel() {  caller.setState(**new** Level2State(caller));  }  **public** String toString() {  **return** "LEVEL1";  }  }  **class** Level2State **extends** State{//point 31~99  **public** Level2State(StatePattern caller) {  **super**(caller);  }  @Override  **public** **void** nextLevel() {  caller.setState(**new** Level3State(caller));  }  **public** String toString() {  **return** "LEVEL2";  }  }  **class** Level3State **extends** State{//point 100~  **public** Level3State(StatePattern caller) {  **super**(caller);  }  @Override  **public** **void** nextLevel() {  caller.setState(**new** Level3State(caller));  }  **public** String toString() {  **return** "LEVEL3";  }  }  **public** **class** StatePattern {  State state = **null**;  **public** StatePattern() {  state = **new** DefaultState(**this**);  }    **public** **void** run(){  state.nextLevel();  }  **public** **void** setState(State state){  **this**.state = state;  }    **public** **void** printState(){  System.*out*.println(state);  }    **public** **static** **void** main(String[] args) {  StatePattern s = **new** StatePattern();  s.printState();  s.run();  s.printState();  s.run();  s.printState();  s.run();  s.printState();  s.run();  s.printState();  }  } |

**데코레이터(Decorator)**

상속은 기능을 확장하는 방법을 제공한다

파일출력 클래스가 있다고 치자 하지만 zip으로된 파일로 출력이 라던지 하려면 파일출력 클래스를 상속받아서 처리해야될것이다.. 그렇게하면 클래스의 개수가 많아지고. 복잡해진다.

데코레이터는 위임방식으로 이러한복잡도를 계층형으로 줄인다.

|  |
| --- |
| **package** designpattern;  //데코레이터패턴  **interface** FileOut\_I{  **public** **void** write(**byte**[] data);  }  **class** FileOut **implements** FileOut\_I{  @Override  **public** **void** write(**byte**[] data) {  System.*out*.println("Last : "+**new** String(data));  }  }  **abstract** **class** Decorator **implements** FileOut\_I{  **private** FileOut\_I delegate;  **public** Decorator(FileOut\_I delegate){  **this**.delegate = delegate;  }    **protected** **void** doDelegate(**byte**[] data){  delegate.write(data);  }  }  **class** Encryption **extends** Decorator{  **public** Encryption(FileOut\_I delegate) {  **super**(delegate);  }  @Override  **public** **void** write(**byte**[] data) {  String s = **new** String(data)+"\_Encryption";  doDelegate(s.getBytes());  }  }  **class** ZipOut **extends** Decorator{  **public** ZipOut(FileOut\_I delegate) {  **super**(delegate);  }  @Override  **public** **void** write(**byte**[] data) {  String s = **new** String(data)+"\_Zip";  doDelegate(s.getBytes());  }  }  **class** Mp3Out **extends** Decorator{  **public** Mp3Out(FileOut\_I delegate) {  **super**(delegate);  }  @Override  **public** **void** write(**byte**[] data) {  String s = **new** String(data)+"\_Mp3";  doDelegate(s.getBytes());  }  }  **public** **class** DecoratorPattern {  **public** **static** **void** main(String[] args) {  FileOut\_I out = **new** Encryption(**new** ZipOut(**new** Mp3Out(**new** FileOut())));  out.write("Show".getBytes());    out = **new** Mp3Out(**new** Encryption(**new** ZipOut(**new** FileOut())));  out.write("Show".getBytes());  }  } |
| **상속으로 처리를하면 금방금방 만들긴하겠지만 업무에따라서 클래스가 늘어난다는 문제점이 발생한다**  **이에 따라서.**  **데코레이터 패턴을 사용하면 각 확장 기능들의 구현이 별도의 클래스로 분리되기 때문에**  **각확장 기능 및 원래 기능을 서로 영향 없이 변경할 수 있도록 만들어준다.** |

**프록시(proxy)패턴**

말그대로 프록시패턴을 가지고 처리하는 패턴방식이다.

다시말해 사용하고자하는 객체의 똑 같은 인터페이를 impl 하여 만든다음

안쪽에 결론적으로 사용할 객체의 호출부를. 상황에 맞게 호출하게 만드는 프록시 역할을 하는 패턴이다.

특정 객체를 통하여 경유하여 조건을 처리하여 특정 메서드를 호출하게처리한다.

|  |
| --- |
| **package** designpattern;  **interface** Image\_I{  **public** **void** draw();  }  **class** Image **implements** Image\_I{  @Override  **public** **void** draw() {  System.*out*.println("그림을 그렸습니다.");  }  }  **class** ImageView **implements** Image\_I{  @Override  **public** **void** draw() {  **if**(ProxyPattern.*viewCnt*%2==0)  System.*out*.println("인터페이스를 구현하여 프록시객체를만들었습니다. 상황에 맞게 그림을 그렸습니다.");  }  }  **class** ImageViewObj **extends** Image{  @Override  **public** **void** draw() {  **if**(ProxyPattern.*viewCnt*%2==0)  System.*out*.println("상속하여 프록시객체를만들었습니다 상황에 맞게 그림을 그렸습니다.");  }  }  **public** **class** ProxyPattern {  **public** **static** **int** *viewCnt*=0;  **public** **static** **void** main(String[] args) {  Image\_I img=**null**;  **for**(**int** i = 0 ; i < 10 ; i++){  img = **new** Image();  img.draw();  *viewCnt*++;  }    **for**(**int** i = 0 ; i < 10 ; i++){  img = **new** ImageView();  img.draw();  *viewCnt*++;  }    **for**(**int** i = 0 ; i < 10 ; i++){  img = **new** ImageViewObj();  img.draw();  *viewCnt*++;  }  }    } |

**어뎁터(Adapter)패턴**

서로 다른 인터페이스를 가진 객체들끼리 이어서 처리 해줘야할경우

어뎁터 패턴을 사용한다.

이렇게 한다면 기존 흐름 소스를 수정하지 않아도 된다.

|  |
| --- |
| **package** designpattern;  //오리지널  **interface** Print\_I {  **public** **void** print(String msg);  }  **class** DefaultPrint **implements** Print\_I {  @Override  **public** **void** print(String msg) {  System.*out*.println("DefaultPrint\_"+msg);  }  }  // 전혀 새로운 인터페이스  **interface** QuickPrint\_I {  **public** **void** print(**byte**[] msg);  }  **class** QuickPrint **implements** QuickPrint\_I {  @Override  **public** **void** print(**byte**[] msg) {  System.*out*.println("QuickPrint\_"+**new** String(msg));  }  }  //어뎁터 처리  **class** AdapterPrint **implements** Print\_I{  QuickPrint qp =**null**;  **public** AdapterPrint() {  qp=**new** QuickPrint();  }  @Override  **public** **void** print(String msg) {  System.*out*.println("어뎁터처리 들어감");  qp.print( msg.getBytes());  }  }  **public** **class** AdapterPattern {  **public** **static** **void** main(String[] args) {  Print\_I print = *getPrintObj*();  print.print("안녕하세요");  }    **public** **static** Print\_I getPrintObj(){  **return** **new** AdapterPrint();  // return new DefaultPrint();  }  } |

**옵저버(Observer)패턴**

객체의 책임을 전략전패턴으로 처리후.처리한다.

매니저 같은..

옵저버는 옵저버 객체들을 관리하고 그것을 통보하는.. 말그대로 옵저버 같은 역할을 한다.

고려사항

1. 주제 객체의 통지 기능 실행 주체

2. 옵저버 인터페이스 분리

3. 통지 시점에서의 주제 객체 상태

4. 옵저버 객체의 실행 제약조건

|  |
| --- |
| **package** designpattern;  **import** java.util.ArrayList;  **interface** ObserVerObject\_I{  **public** **void** run(String state);  }  **class** MailObserver **implements** ObserVerObject\_I{  @Override  **public** **void** run(String state) {  System.*out*.println("MAIL Observer");  }    }  **class** SMSObserver **implements** ObserVerObject\_I{  @Override  **public** **void** run(String state) {  System.*out*.println("SMS Observer");  }    }  **class** Observer{  ArrayList<ObserVerObject\_I> list = **new** ArrayList<ObserVerObject\_I>();    **public** **void** add(ObserVerObject\_I o){  list.add(o);  }    **public** **void** remove(ObserVerObject\_I o){  list.remove(o);  }      **public** **void** changeState(String state){  **for** (**int** i = 0; i < list.size(); i++) {  list.get(i).run(state);  }  }  }  **public** **class** ObserverPettern {  **public** **static** **void** main(String[] args) {  ObserVerObject\_I mail = **new** MailObserver();  ObserVerObject\_I sms = **new** SMSObserver();    Observer ob = **new** Observer();  ob.add(mail);  ob.add(sms);  ob.changeState("gg");  }  } |

**파사드(Facade)패턴**

UseObject

EMPDAO

DBDAO

QDAO

ManObject

위처럼 서로다른 객체에서 DAO를 바로바로 가져다 쓰면 똑 같은 호출 부가 중복되어 처리되느 오류가 생길수도 있다. 가운데에서 처리해주는 것 파사드를 두면좋다.

UseObject

EMPDAO

DBDAO

QDAO

ManObject

FacadeObj

파사드 패턴은 클래스와 비교해보면 파사드는 마치 서브 시스템의 상세함을 감춰 주는 인터페이스와 유사하다. 파사드를 통해서 서비 시스템의 상세한 구현을 캡슐화하고 이를 통해 상세한 구현이 변경되더라도 파사드를 사용하는 코드에 주는 영향을 줄일수 있게된다.

**추상 팩토리(Abstract Factory)패턴**

추상클래스에서

팩토리메서드를 static으로 잡아두

그 현재 추상클래스를 상속받은 클래스들을 상태값에 따라서 다르게 리턴해준다.

|  |
| --- |
| **package** designpattern;  **abstract** **class** Man{  **public** **static** Man getMan(**int** age){ //추상클래스에서 팩토리를 정의한다.  **if**(age>50){  **return** **new** OldMan();  }**else**{  **return** **new** YongMan();  }  }    **abstract** **public** **void** play();  }  **class** YongMan **extends** Man{  @Override  **public** **void** play() {  System.*out*.println("YongMan Play");  }  }  **class** OldMan **extends** Man{  @Override  **public** **void** play() {  System.*out*.println("Old Play");  }  }  **public** **class** AbstractFactory {  **public** **static** **void** main(String[] args) {  Man man = Man.*getMan*(40);  man.play();  man = Man.*getMan*(100);  man.play();  }  } |

**컴포지트(Abstract Factory)패턴**

옵저버패턴달리. 관리하는 객체들의 인터페이스와 관리하는 객체의 인터페이스를 동일시 처리하는 방법

1. 컴포넌트그룹을 관리한다.

2. 컴포지트에 기능 실행을 요청하면 컴포지트는 포함하고 있는 컴포넌트들에게 기능 실행 요청을 위임한다.

클라이언트가 컴포지트와 컴포넌트를 구분하지 않고 컴포넌트 인터페이스로만으로 프로그래밍 할수 있게 돕는점이다.

|  |
| --- |
| **package** designpattern;  **import** java.util.ArrayList;  **interface** ComObject\_I{  **public** **void** run(String state);  }  **class** MailCom **implements** ComObject\_I{  @Override  **public** **void** run(String state) {  System.*out*.println("MAIL ComPosite");  }    }  **class** SMSCom **implements** ComObject\_I{  @Override  **public** **void** run(String state) {  System.*out*.println("SMS ComPosite");  }    }  **class** ComGroup **implements** ComObject\_I{  ArrayList<ComObject\_I> list = **new** ArrayList<ComObject\_I>();    **public** **void** add(ComObject\_I o){  list.add(o);  }    **public** **void** remove(ComObject\_I o){  list.remove(o);  }      @Override  **public** **void** run(String state) {  **for** (**int** i = 0; i < list.size(); i++) {  list.get(i).run(state);  }  }  }  **public** **class** ComPositePettern {  **public** **static** **void** main(String[] args) {  ComObject\_I mail = **new** MailCom();  ComObject\_I sms = **new** SMSCom();    ComGroup ob = **new** ComGroup();  ob.add(mail);  ob.add(sms);  ob.run("gg");  }  } |

**널객체(Null)패턴**

매번 if(a!=null) 이런식으로 널체크를 했을것이다..

이런 방법은 누락되면 널포인트입셉션을 낼가능성이 아주 많은 코드이다.

|  |
| --- |
| **package** designpattern;  **abstract** **class** Printer{  **abstract** **public** **void** print(String msg);  }  **class** InkPrinter **extends** Printer{  @Override  **public** **void** print(String msg) {  System.*out*.println("Ink PRINT : "+msg);  }  }  **class** DotPrinter **extends** Printer{  @Override  **public** **void** print(String msg) {  System.*out*.println("Dot PRINT : "+msg);    }  }  **class** NullPrint **extends** Printer{  @Override  **public** **void** print(String msg) {  System.*out*.println("Null PRINT : "+msg);  }    }  **public** **class** NullPattern {  **public** **static** **void** main(String[] args) {  *getPrint*(100).print("100");  *getPrint*(200).print("200");  *getPrint*(500).print("500");  }  **static** **public** Printer getPrint(**int** money) {  **if** (money == 100) {  **return** **new** DotPrinter();  } **else** **if** (money == 200) {  **return** **new** InkPrinter();  }  **return** **new** NullPrint(); // 아무것도 걸리지 않으면 널 인객체를 반환한다 이렇게하여 널포인트를 막는다.  }  } |