

디자인 패턴

프로그래밍 실무에서 발생한 문제 → 객체 상호간 관계 등을 이용해 해결 한 수단으로는 하나의 규칙!

④ 라이브러리 or 프레임워크 만드는데 기본

Ex) paupart.js (진짜 패턴)

⑤ 디자인 패턴 가능

디자인 패턴 종류

- 생성 패턴: 객체 생성방법이 들어간 패턴 / 설계도면, 팩토리, 브레인 $\text{class} \rightarrow \text{object}$
- 구조 패턴: 객체, 클래스로 구조를 만들 때 유연하고 확장性 있는 방법들을 만든 패턴 / 프록시, 기다리기
- 행동 패턴: 객체나 클래스 간의 일관성을, 행동을 얻기 위한 패턴 / 이터레이터, 음성화, 전략

ex: flux, MVC, MVP, MVVM

라이브러리, 프레임워크

- 라이브러리: 꼭 필요한 수 있는 특정 기능을 모듈화 (fine)
- 프레임워크: 꼭 필요한 수 있는 특정 기능을 모듈화 (strict)

싱글톤 패턴

하나의 클래스에서 1개의 인스턴스만 가지는 패턴

DB 연결 초기화 사용 (비동기적)

- (+) 인스턴스 생성 대신 I/O 배운드 감소
- (-) 1번 인스턴스만 사용 → 의존성↑, TDD 불편

* 싱글톤 패턴 X

* 싱글톤 패턴 O

```
class Singleton {
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }
}

const a = new Rectangle(1, 2);
const b = new Rectangle(1, 2);
console.log(a === b) // false
```

* DB 연결 패턴

```
class Singleton {
    constructor(url, options) {
        if (Singleton.instance) {
            return Singleton.instance;
        }
        getinstance() {
            const url = 'mongodb://localhost:28017/test';
            const createConnection = url => {
                class Connection {
                    constructor(url) {
                        if (url === '') {
                            return null;
                        }
                    }
                }
                const a = new Connection();
                const b = new Connection();
                console.log(a === b) // true
            }
            return this.instance;
        }
    }
}
```

Rect
A → B → C

* Java 싱글톤

* MongoDB 연결 패턴

```
class Singleton {
    private static final class SingletonHolder {
        private static final Singleton INSTANCE = new Singleton();
    }
    public static Singleton getinstance() {
        return SingletonHolder.INSTANCE;
    }
}

public class Main {
    public static void main(String[] args) {
        MongoClient mongoClient = MongoClients.create("mongodb://localhost:28017/test");
        System.out.println("MongoDB 연결 성공!");
        mongoClient.close();
    }
}
```

* MongoDB 연결 패턴 (의존성↑)

```
class Singleton {
    private static final class SingletonHolder {
        private static final Singleton INSTANCE = new Singleton();
    }
    public static Singleton getinstance() {
        return SingletonHolder.INSTANCE;
    }
}

public static void main(String[] args) {
    MongoClient mongoClient = MongoClients.create("mongodb://localhost:28017/test");
    System.out.println("MongoDB 연결 성공!");
    mongoClient.close();
}
```

○ → ○ → A
MongoDB 연결 패턴
A → B → C
MongoDB 연결 패턴
B → A → D
TDD 가능!

싱글톤 구현 1가지

단순 예제로: 싱글톤 패턴 생성 예제 확인

○ 가짜 예제 X × 새로운 예제

원형 설계법 → 미리 설정
Java (미스터리)
설계도면에 템플릿을 가져오기 (sleep 때문에 가능)

○ Synchronized: 단순화한 전파자 lock (다른 스레드 접근)

```
public class Singleton {
    private static Singleton instance;
    private Singleton() {
    }
    public static synchronized Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

생략부

getinstance → ○ → ○

• 정적멤버, 정적법: 미리 인스턴스 생성 (static), 클래스로 동적생성, 대체 인스턴스

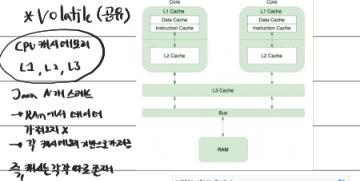
(-) 자원낭비 (설계도면과 맞지 않음)

```
public class Singleton {
    private static final Singleton INSTANCE = new Singleton();
    private Singleton() {
    }
    public static Singleton getInstance() {
        return INSTANCE;
    }
}
```

○ 설계망비파: LazyHolder (증집 클래스): singletonInstanceHolder 내부로 네스팅 → getinstance() 호출 시 딴번 생성 (체이닝)

```
class Singleton {
    private static class SingletonHolder {
        private static final Singleton INSTANCE = new Singleton();
    }
    public static Singleton getInstance() {
        return SingletonHolder.INSTANCE;
    }
}
```

○ 이중 척연 패턴 (DCL): 상단 패턴 참조 전, 객체 생성 전



```
public class Test {
    volatile boolean flag = true;
    public void test() {
        new Thread() {
            int cmt = 0;
            while (true) {
                if (cmt++ > 100000000) {
                    System.out.println("Thread 1 finished");
                    break;
                }
            }
        }.start();
        new Thread() {
            int cmt = 0;
            while (true) {
                if (cmt++ > 100000000) {
                    System.out.println("MainMemory flag = " + flag);
                    break;
                }
            }
        }.start();
    }
}
```

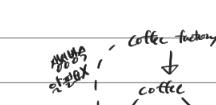
○ Enum: thread safe 뷰어 (체이닝)

```
public enum SingletonEnum {
    INSTANCE;
    public void oortCloud() {
    }
}
```

제작자 패턴

각 객체 생성 부분을 추상화한 패턴
상기 class (위아래), 하위 class (각각 생성 구체화된 대상)

(+) 리팩토링 → 편집 단위끼리 → 유연성↑
(각각 생성 단위로 따로 존재)



public class Main {
 public static void main(String[] args) {
 Coffee coffee = CoffeeFactory.createCoffee(CoffeeType.LATTE);
 System.out.println(coffee.getName()); // Latte
 }
}

○ 이터레이터 패턴: 각각의 사용에 collections의 형식을 활용하는 패턴
(각각의 구조에 속한 값들이 interface로 처리 가능)

○ 이터레이터 프로토콜: iterable 프로토콜 처리, 처리

* Container = collections (동일한 형태의集合)

이터레이터 패턴: 접근 자체와 같은 패턴

```
public - 차원, 차원화된 접근 가능
protected - 차원 클래스 O, 차원화된 클래스
private - 차원 자체 X
접근 자체와 같은 패턴
```

○ Container = collections (동일한 형태의集合)

```
const mp = new Map();
mp.set('a', 1);
mp.set('b', 2);
mp.set('ccc', 3);
const st = new Set();
st.add(1);
st.add(2);
st.add(3);
const a = [];
for(let i = 0; i < 10; i++) a.push(i);
for(let aa of a) console.log(aa);
for(let a of mp) console.log(a);
for(let a of st) console.log(a);
```

전략 패턴 : 객체의 행위를 나누고 싶은 경우 '직접' 수정 X, '캡슐화된 알고리즘'을 컨테이너에서 바꾸기

상속과 함께 가능하게 만드는 패턴 / `passport.js`

```

import java.text.DecimalFormat;
import java.util.ArrayList;
import java.util.List;
interface PaymentStrategy {
    public void pay(int amount);
}

class KAKAOCardStrategy implements PaymentStrategy {
    private String name;
    private String cardNumber;
    private String cvv;
    private String dateOfExpiry;

    public KAKAOCardStrategy(String nm, String ccNum, String cvv,
    String expiryDate){
        this.name=nm;
        this.cardNumber=ccNum;
        this.cvv=cvv;
        this.dateOfExpiry=expiryDate;
    }

    @Override
    public void pay(int amount) {
        System.out.println(amount + " paid using KAKAOCard.");
    }
}

class LUNACardStrategy implements PaymentStrategy {
    private String emailId;
    private String password;

    public LUNACardStrategy(String email, String pwd){
        this.emailId=email;
        this.password=pwd;
    }

    @Override
    public void pay(int amount) {
        System.out.println(amount + " paid using LUNACard.");
    }
}

class Item {
    private String name;
    private int price;
    public Item(String name, int cost){
        this.name=name;
        this.price=cost;
    }

    public String getName() {
        return name;
    }

    public int getPrice() {
        return price;
    }
}

class ShoppingCart {
    List<Item> items;

    public ShoppingCart(){
        this.items=new ArrayList<Item>();
    }

    public void addNewItem(Item item){
        this.items.add(item);
    }

    public void removeItem(Item item){
        this.items.remove(item);
    }

    public int calculateTotal(){
        int sum = 0;
        for(Item item : items){
            sum += item.getPrice();
        }
        return sum;
    }

    public void pay(PaymentStrategy paymentMethod){
        int amount = calculateTotal();
        paymentMethod.pay(amount);
    }
}

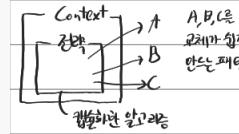
public class HelloWorld{
    public static void main(String []args){
        ShoppingCart cart = new ShoppingCart();

        Item A = new Item("kundola",100);
        Item B = new Item("kundolB",300);

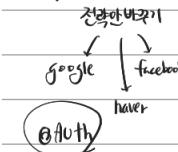
        cart.addItem(A);
        cart.addItem(B);

        // pay by LUNACard
        cart.pay(new LUNACardStrategy("kundol@example.com",
        "pukubaboo"));
        // pay by KAKAOBank
        cart.pay(new KAKAOCardStrategy("Ju hongchul", "123456789",
        "123", "12/01"));
    }
}

```

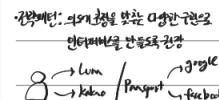


`passport.js`: 인증을 관리하는 패턴



인증서버 vs 결제서버

(인증은 쉽게 교체하거나 위한 대안)



그림자금: 단지 일시적 편의 & 보안 위험



중재자 패턴 : 주체가 다른 객체의 상대방과 같은 때마다 서비스들을 통해

중재자 패턴은 중재자를에게 변화를 알려주는 패턴 / Twitter, MVC패턴, Thread

- 주체: 객체 보유하는 관찰자 / 주체, 관찰 구분없이 상대방 변경되는 객체를 기반으로 구현하기도 함

중재자는 객체 속의 변화에 따라 '우선' 변화사항이 생기는 객체들을 미리

`MVC` 패턴 `Model` `View` `Controller`

```

import java.util.ArrayList;
import java.util.List;

interface Subject {
    public void register(Observer obj);
    public void unregister(Observer obj);
    public Object getupdate();
}

interface Observer {
    public void update();
}

class Topic implements Subject {
    private List<Observer> observers;
    private String message;

    public Topic() {
        observers = new ArrayList<>();
        this.message = "";
    }

    @Override
    public void register(Observer obj) {
        if (!observers.contains(obj)) observers.add(obj);
    }

    @Override
    public void unregister(Observer obj) {
        observers.remove(obj);
    }

    @Override
    public void notifyObservers() {
        this.observers.forEach(Observer::update);
    }

    @Override
    public Object getupdate() {
        return this.message;
    }

    public void postMessage(String msg) {
        System.out.println("Message vended to Topic: " + msg);
        this.message = msg;
        notifyObservers();
    }
}

class TopicSubscriber implements Observer {
    private String name;
    private Subject topic;

    public TopicSubscriber(String name, Subject topic) {
        this.name = name;
        this.topic = topic;
    }

    @Override
    public void update() {
        String msg = (String) topic.getUpdate();
        System.out.println(name + ": got message > " + msg);
    }
}

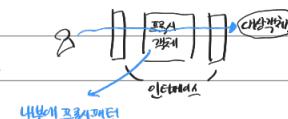
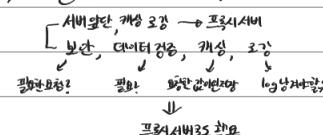
public class HelloWorld {
    public static void main(String[] args) {
        Topic topic = new Topic();
        Observer a = new TopicSubscriber("a", topic);
        Observer b = new TopicSubscriber("b", topic);
        Observer c = new TopicSubscriber("c", topic);
        topic.register(a);
        topic.register(b);
        topic.register(c);
        topic.postMessage("amnu is op champion!!!");
    }
}

```

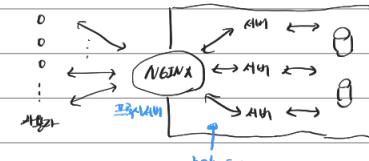
ex)

A가 follow → A가 이를 → 전송됨

프록시 패턴 : 대상 객체에 접근 전, 혹은 가로채 객체의 일부로써 역할하는 패턴



* 프록시서비 : Server, Client 사이에 자리를 통해 다른 네트워크에 접속할 수 있게 하는 시스템



* 프록시서비 개념
개인간 대화의 확장
(= 트래픽 ↓ = 용량↓)

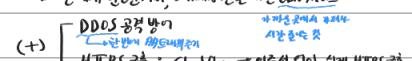
```

function createReactiveObject(target, callback) {
    const proxy = new Proxy(target, {
        set(target, prop, value) {
            if(prop === obs[prop]){
                const prev = obs[prop];
                obs[prop] = value;
                obseverC(!{prev} >> !{value});
            }
        }
    });
    return proxy;
}
const a = createReactiveObject();
a.value = "Hello";
a.value

```

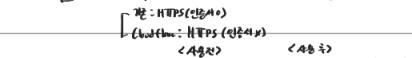
* 프록시서비 쓰는 Chatflow

→ 간접화 분산된 서버, contents 전달 빠른 CDN이나



* 프록시서비 개념
개인간 대화의 확장
(= 트래픽 ↓ = 용량↓)

(+) HTTPS 공격 방지 → 인증서 확인 후 https로



* 프록시서비 개념
개인간 대화의 확장
(= 트래픽 ↓ = 용량↓)

<서버> → <서버> → <서버>

DDoS 공격 방지 (개인간 대화로 통증 + 계정 분할)

* 프록시서비 개념
개인간 대화의 확장
(= 트래픽 ↓ = 용량↓)

→ https://www.aaa.com : 1234/test

→ https://www.aaa.com : 1234/test

CORS가 FE의 프록시서비 : 서버는 보호자에게서 리소스로 되어 다른 서버를 통해 요청이 왔을 때

(HTTP에서 자체 막기 때문)

<Proxy>

FEAM → /api → User API

/api → CORS (`!{200, 0, 1, true}`)

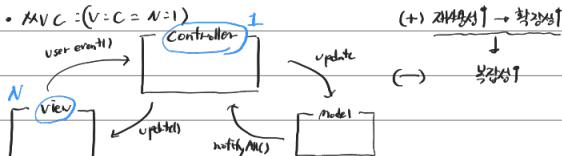
→ User API

FEAM → /api → User API

/api → CORS (`!{200, 0, 1, true}`)

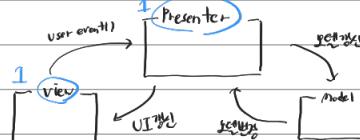
→ User API

MVC, MVP, MVVM 패턴

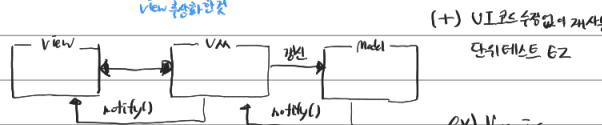


Ex) Spring WEB MVC (framework)

- MVP : $(V:P = 1:1)$ (더 강한 결합)



- MVVM : $C + Vn (view model) / Vn = V = 1:N$



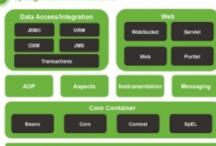
Ex) Vue.js

data binding : 표현보관 데이터 = 표현기기에서 데이터에 대처

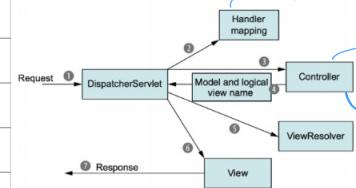
	MVC	MVP	MVVM
관계	$C:V = 1:N$	$P:V = 1:1$	$V:V = 1:N$
창조	$V \in C$ 창조X	$V \in P$ 창조O	$V \in Vn$ 창조O

Spring Web MVC

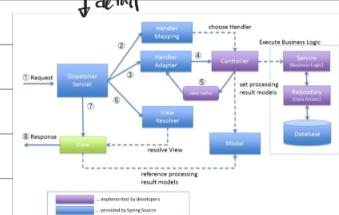
Spring Framework Runtime



* Class Implementation



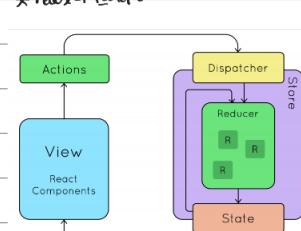
detail



flux 패턴 : MVC 복잡한 문제 해결 패턴 ('한방향')



* Flux 패턴 (flux.js)



```

const initialState = {
  visibilityFilter: 'SHOW_ALL',
  todos: []
}

function appReducer(state = initialState, action) {
  switch(action.type) {
    case 'SET_VISIBILITY_FILTER':
      return Object.assign({}, state, { visibilityFilter: action.filter })
    case 'ADD_TODO':
      return Object.assign([], state, [
        ...state,
        {
          id: action.id,
          text: action.text,
          completed: false
        }
      ])
  }
}

```

이란수입과 표준화된 패턴

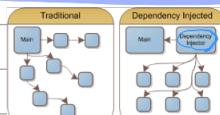
- 개인이 다른 프로젝트 간移植으로(의존성 분리) 코드를重용하는 것

- 노드.js (보통 개발 환경)

- DI : 풍선을 끌어올려거나 미리 X → 2개로 쪼개어야 한다.

총괄하는 세부구조가 서로 X → 세부구조는 총괄에 대처 불가능하다.

* 의존 관계되어 A → B, B가 변화면 A도跟着



```

import java.util.*;
public void main() {
    System.out.println("전체");
}
class A {
    public void a() {
        System.out.println("A");
    }
}
class B {
    public void b() {
        new A().a();
    }
}
public class main{
    public static void main(String args[]) {
        new B().b();
    }
}

```

* DIP X

```

import java.util.*;
class BackendDeveloper {
    public void writeJava() {
        System.out.println("자바가 좋아요 안티내시브");
    }
}

class FrontEndDeveloper {
    public void writeJavaScript() {
        System.out.println("자바스크립트가 좋아요 안티내시브");
    }
}

class Project {
    private final BackendDeveloper backendDeveloper;
    private final FrontEndDeveloper frontEndDeveloper;

    public Project(BackendDeveloper backendDeveloper, FrontEndDeveloper frontEndDeveloper) {
        this.backendDeveloper = backendDeveloper;
        this.frontEndDeveloper = frontEndDeveloper;
    }

    public void implement() {
        backendDeveloper.writeJava();
        frontEndDeveloper.writeJavaScript();
    }

    public static void main(String args[]) {
        Project project = new Project(new BackendDeveloper(), new FrontEndDeveloper());
        project.implement();
    }
}

```

(+) 재사용성, 표준화된 패턴 가능

단점 : 미리작성된 EZ → 다른 패턴 생성(DI는 아니지만)

내부 구조에 의존성이 있어서 원래의 패턴이 고착화됨

(-) 확장↑ = 복잡성

코드↑ = 비용↑

종합적인 (현대적, 경제적) → 경제적 기반 확장 가능 (경쟁력 확보)

* DIP O

```

interface Developer {
    void develop();
}

class BackendDeveloper implements Developer {
    @Override
    public void develop() {
        writeJava();
    }

    public void writeJava() {
        System.out.println("자바가 좋아요 내시브");
    }
}

class FrontendDeveloper implements Developer {
    @Override
    public void develop() {
        writeJavaScript();
    }

    public void writeJavaScript() {
        System.out.println("자바스크립트가 좋아요 내시브");
    }
}

class Project {
    private final List<Developer> developers;
    public Project(List<Developer> developers) {
        this.developers = developers;
    }

    public void implement() {
        developers.forEach(Developer::develop);
    }

    public static void main(String args[]) {
        List<Developer> dev = new ArrayList<>();
        dev.add(new BackendDeveloper());
        dev.add(new FrontendDeveloper());
        Project project = new Project(dev);
        project.implement();
    }
}

```

Static 사용 단장

정리 : static 변수 → 디클래스팅 가능 → AOT 가능

단점 : 비효율 (필터링 어려움)

정리 : static 단점 x 전부 0 → 디클래스팅 단점 0 (단점)

기획서 ① 어떤 종류의 상세한 험경 '법률화' ② 컨테스트 사용 (같은 환경에서 계속 적용되는 경우 컨테스트 사용)

ex) 배송지를 Context, 이를 contextual information

HTTP 헤더 Context, HTTP 헤더 contextual information
트래킹 Context, Session ID contextual information

```

const ThemeContext = React.createContext('light');

class App extends React.Component {
  render() {
    return (
      <ThemeContext.Provider value="dark">
        <Toolbar />
      </ThemeContext.Provider>
    );
  }
}

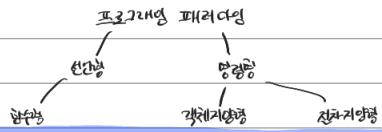
function Toolbar() {
  return (
    <div>
      <ThemedButton />
    </div>
  );
}

class ThemedButton extends React.Component {
  static contextType = ThemeContext;
  render() {
    return <button theme={this.context}>{this.props.children}</button>;
  }
}

```

II 3.22 챕터 패리다임

프로그래밍의 관점은 크게 해주는 개발 방법론



선언과 명령형 프로그래밍 : '무엇을' 풀어내는 개체 같은

```
const ret = [1, 2, 3, 4, 5, 11, 12];
ret.reduce((max, num) => num > max ? num : max, 0);
console.log(ret) // 12
```

선언형 프로그래밍 ← '무엇인가' 블록처럼 쓰거나 모색구현 ④ 고차함수'를 통해 재사용↑

JavaScript (함수체) → 객체지향(x) 함수형(o)

```
const pure = (a, b) => {
  return a + b
}
```

a, b가 어떤 타입을 받든
다른 타입과 혼용이 안된다.

고차함수

함수 자체를 다른 곳에 사용

※ 함수체 (고차함수포함)

변수, 변수에 할당 가능

함수 (함수()) 가능

함수() → return 함수()

객체지향 PG

• 객체들의 집합, PG는 상호작용을 표현하는데 대비되는 객체를 적자로 제공 → 객체 내부가 선언된 대소드를 활용

(→ 실제로 많은 시간 소요, 처리속도↓)

• 추상화 : 복잡한 시스템에서 core 개념 자체를 간추려내는 것(타이머)

○ OOP (16) 추상화, 반복, 초기화

• 추상화 : 핵심의 속성을 대신 기반으로 초기화하기

• 상속성 : 하위(class)가 상위 class의 특성을 물려받아 재사용 / 추가 / 확장하는 것

• 고조상화 : 상위클래스에서 재사용한 부분은 하위클래스에서 재정의 (+)

• 다형성 : 1개의 class or method 가 다양한 방법으로 활용되는 것 → 오버로딩 / 오버라이딩

• 오버로딩 : 같은 이름을 가진 메소드 여러개는 것

→ Ext(string) Ext(string n, string l) ...

• 오버라이딩 : 상위클래스로부터 상속받은 메소드를 하위클래스가 재정의 → 동작다양성

```
class Animal {
  public void bark()
}

class Dog extends Animal {
  @Override
  public void bark()
}
```

설계 원칙 (SOULD)

SRP - 모든 class는 각각 고유한 책임을 갖는다.

OCP - 유지보수 시 고르게 바뀐다, 동일한 바뀐다 → 기본적으로는 광범위하고, 확장은 어렵다

LSP - 부모클래스에서 가져온게 끝나면 그 이상은 자식클래스에서 가져온다

ISP - 자신의 인터페이스만 갖고 있는 인터페이스 만들기

DIP - 계층마다 필요한 것만 가져온다 → 확장은 어렵지만, 인터페이스를 주는

방법은 서로 다른 방법에 영향을 주지 않으며, 상세한 구현은 하위층 변화에 미안관으로 드릴까봐.

선언형 PG (제네로인 디자인)

↑ 가수성 → 계산이 매우 간단하게 사용
설명하기

```
const ret = [1, 2, 3, 4, 5, 11, 12];
let a = 0;
for (let i=0; i<ret.length; i++) {
  a = Math.max(ret[i], a)
}
console.log(a)
```