디자인 패턴과 프로그래밍 패러다임

1.1 디자인 패턴

- 디자인 패턴이란 ?
    - 프로그램을 설계할 때 발생했던 문제점들을 객체 간의 상호 관계 등을 이용하여 해결할 수 있도록 하나의 “규약” 형태로 만들어 놓은 것을 의미

1.1.1 싱글톤 패턴

- 싱글톤 패턴(singleton pattern)은 하나의 클래스에 오직 하나의 인스턴스만 가지는 패턴
- 보통 DB 연결 모듈에 많이 사용
- 하나의 인스턴스를 만들고 해당 인스턴스를 다른 모듈들이 공유하며 사용하기 때문에 인스턴스를 생성할 때 드는 비용이 줄어드는 장점이 있다. 하지만 의존성이 높아 진다는 단점이 있다.

자바스립트의 싱글톤 패턴

- js에서는 리터럴 {} 또는 new Object로 객체를 생성하게 되면 다른 어떤 객체와도 같지 않기 때문에 이 자체만으로 싱글톤 패턴을 구현할 수 있다.

```
const obj = {
a : 27
}

const obj2 = {
a : 27
}
obj != obj2
```

```
class Singleton {
    constructor() {
        if (!Singleton.instance) {
            Singleton.instance = this
        }
```

```
        return Singleton.instance
    }
    getInstance() {
        return this
    }
}
const a = new Singleton()
const b = new Singleton()
console.log(a === b) // true
```

- a와 b는 하나의 인스턴스를 가짐

데이터베이스 연결 모듈

```
const URL = 'mongodb://localhost:27017/kundolapp'
const createConnection = url => ({"url" : url})
class DB {
    constructor(url) {
        if (!DB.instance) {
            DB.instance = createConnection(url)
        }
        return DB.instance
    }
    connect() {
        return this.instance
    }
}
const a = new DB(URL)
const b = new DB(URL)
console.log(a === b) // true
```

- DB.instance라는 하나의 인스턴스를 기반으로 a,b를 생성
- 이를 통해 db연결에 관한 인스턴스 생성 비용을 아낄 수 있음.

자바에서의 싱글톤 패턴

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

```
public class HelloWorld{
     public static void main(String []args){
        Singleton a = Singleton.getInstance();
        Singleton b = Singleton.getInstance();
        System.out.println(a.hashCode());
        System.out.println(b.hashCode());
        if (a == b){
         System.out.println(true);
        }
     }
}

1. 클래스안에 클래스(Holder), static이며 중첩된 클래스인 singleInstanceHolder를
기반으로 객체를 선언했기 때문에 한 번만 로드되므로 싱글톤 클래스의 인스턴스는 애플리케이션 당 하나만 존재하며
클래스가 두 번 로드되지 않기 때문에 두 스레드가 동일한 JVM에서 2개의 인스턴스를 생성할 수 없습니다.
그렇기 때문에 동기화, 즉 synchronized를 신경쓰지 않아도 됩니다.

2. final 키워드를 통해서 read only 즉, 다시 값이 할당되지 않도록 했습니다.

3. 중첩클래스 Holder로 만들었기 때문에 싱글톤 클래스가 로드될 때 클래스가 메모리에 로드되지 않고
어떠한 모듈에서 getInstance()메서드가 호출할 때 싱글톤 객체를 최초로 생성 및 리턴하게 됩니다.
```

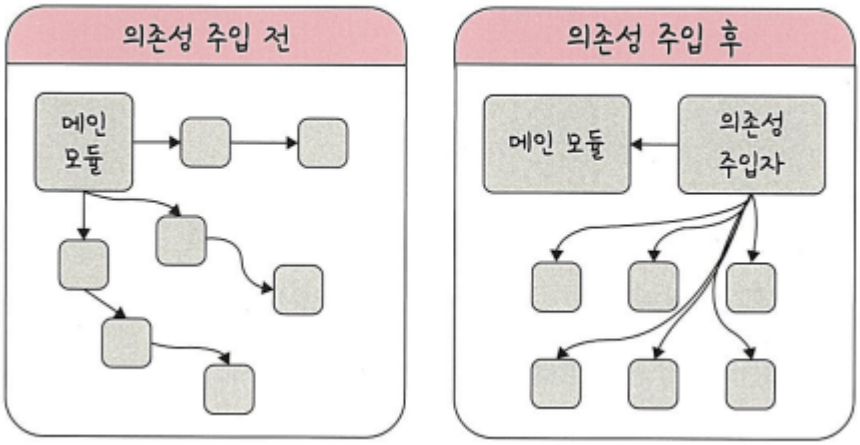
싱글톤 패턴의 단점

- 싱글톤 패턴은 TDD(Test Driven Development)를 할 때 걸림돌이 됨
- TDD를 할 때 단위 테스트를 주로 하는데, 단위 테스트는 테스트가 서로 독립적이여 하며, 테스트를 어떤 순서로든 실행할 수 있어야 한다.
- 하지만 싱글톤 패턴은 미리 생성된 하나의 인스턴스를 기반으로 구현하는 패턴이므로 각 테스트마다 독립적인 인스턴스를 만들기가 어려움 !

의존성 주입

- 싱글톤 패턴은 사용하기가 쉽고 굉장히 실용적이지만 모듈간의 결합을 강하게 만들 수 있다는 단점이 있다.
- 이때 의존성 주입(DI, Dependency Injection)을 통해 모듈간의 결합을 조금 더 느슨하게 만들어 해결할 수 있다.
- A가 B에 의존성이 있다는것은 B의 변경사항에 대해 A또한 변해야 된다는 것을 의미.

▼ 그림 1-6 의존성 주입



- 메인 모듈이 직접 다른 하위 모듈에 대한 의존성을 주기보다는 중간에 의존성 주입자(dependency injector)가 이 부분을 가로채 메인 모듈이 간접적으로 의존성을 주입하는 방식이다.
- 이를 통해 메인 모듈은 하위 모듈에 대한 의존성이 떨어지게 된다. → decoupling

의존성 주입의 장점

- 모듈들을 쉽게 교체할 수 있는 구조가 되어 테스팅하기 쉽고 마이그레이션 하기도 수월
- 구현할 때 추상화 레이어를 넣고 이를 기반으로 구현체를 넣어 주기 때문에 애플리케이션 의존성 방향이 일관된고, 애플렉이션을 쉽게 추론할 수 있으며 모듈 간의 관계들이 조금 더 명확해진다.

의존성 주입의 단점

- 모듈들이 더욱더 분리되므로 클래스 수가 늘어나 복잡성이 증가될 수 있으며 약간의 런타임 패널티가 생김

의존성 주입의 원칙

- 의존성 주입은 “상위 모듈은 하위 모듈에서 어떠한 것도 가져오지 않아야 한다”
- 둘 다 추상화에 의존해야 하며, 이때 추상화는 세부 사항에 의존하지 말아야 한다.

1.1.2 팩토리 패턴

- 팩토리 패턴은 객체를 사용하는 코드에서 객체 생성 부분을 떼어내 추상화한 패턴이자 상속 관계에 있는 두 클래스에서 상위 클래스가 중요한 뼈대를 결정하고, 하위 클래스에서 객체 생성에 관한 구체적인 내용을 결정하는 패턴
- 상위 클래스와 하위 클래스가 분리되기 때문에 느슨한 결합을 ㄱㅏ짐
- 상위 클래스에서는 인스턴스 생성 방식에 대해 전혀 알 필요가 없기 때문에 더 많은 유연성을 갖게 된다.
- 객체 생성 로직이 따로 떼어져 있기 때문에 코드를 리팩터링 하더라도 한 곳만 고칠 수 있게 되니 유지 보수성이 증가

js 팩토리 패턴

- js에서는 팩토리 패턴을 구현한다면 간단하게 new Object()로 구현 가능

```
const num = new Object(42)
const str = new Object('abc')
num.constrctor.name; // Number
```

- 숫자를 전달하거나 문자열을 전달함에 따라 다른 타입의 객체를 생성하는 것을 볼 수 있다.
- 즉, 전달받은 값에 따라 다른 객체를 생성하며 인스터스의 타입 등을 정한다.

```
class CoffeeFactory {
    static createCoffee(type) {
        const factory = factoryList[type]
        return factory.createCoffee()
    }
}
class Latte {
    constructor() {
        this.name = "latte"
    }
}
class Espresso {
    constructor() {
        this.name = "Espresso"
    }
}

class LatteFactory extends CoffeeFactory{
    static createCoffee() {
        return new Latte()
```

```
    }
}
class EspressoFactory extends CoffeeFactory{
    static createCoffee() {
        return new Espresso()
    }
}
const factoryList = { LatteFactory, EspressoFactory }


const main = () => {
    // 라떼 커피를 주문한다.
    const coffee = CoffeeFactory.createCoffee("LatteFactory")
    // 커피 이름을 부른다.
    console.log(coffee.name) // latte
}
main()
```

- CoffeFactory라는 상위 클래스가 중요한 뼈대를 결정하고 하위 클래스인 LatteFactory가 구체적인 내용을 결정하고 있다.
- 참고로 이는 의존성 주입이라고도 볼 수 있으며 ConffeFactory에서 LatteFactory의 인스턴스를 생성하는 것이 아닌 LatteFactory에 생성한 인스턴스를ConffeFactory에 주입하고 있기 때문.
- ConffeFactory를 보면 static으로 createCoffe() 정적 메소드를 정의 → 정적 메소드를 쓰면 클래스ㅜ의 인스턴스 없이 호출이 가능하며 메모리를 절약할 수 있고, 개별 인스턴스에 묶이지 않으며 클래스 내의 함수를 정의할 수 있는 장점이 있다.

자바의 팩토리 패턴

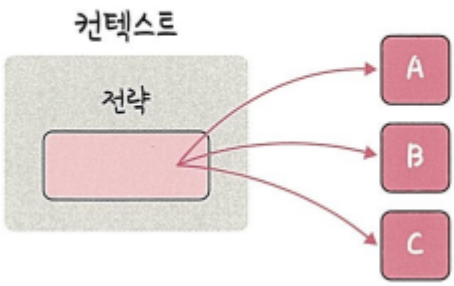
```
abstract class Coffee {
    public abstract int getPrice();

    @Override
    public String toString(){
        return "Hi this coffee is "+ this.getPrice();
    }
}

class CoffeeFactory {
    public static Coffee getCoffee(String type, int price){
        if("Latte".equalsIgnoreCase(type)) return new Latte(price);
        else if("Americano".equalsIgnoreCase(type)) return new Americano(price);
        else{
            return new DefaultCoffee();
        }
    }
}
```

```
class DefaultCoffee extends Coffee {
    private int price;

    public DefaultCoffee() {
        this.price = -1;
    }

    @Override
    public int getPrice() {
        return this.price;
    }
}
class Latte extends Coffee {
    private int price;

    public Latte(int price){
        this.price=price;
    }
    @Override
    public int getPrice() {
        return this.price;
    }
}
class Americano extends Coffee {
    private int price;

    public Americano(int price){
        this.price=price;
    }
    @Override
    public int getPrice() {
        return this.price;
    }
}
public class HelloWorld{
     public static void main(String []args){
        Coffee latte = CoffeeFactory.getCoffee("Latte", 4000);
        Coffee ame = CoffeeFactory.getCoffee("Americano",3000);
        System.out.println("Factory latte ::"+latte);
        System.out.println("Factory ame ::"+ame);
     }
}
```

1.1.3 전략 패턴

- 전략 패턴(strategy pattern)은 정책 패턴(policy pattern)이라고도 하며, 객체의 행위를 바꾸고 싶은 경우 ‘직접’ 수정하지 않고 전략이라고 부르는 ‘캡슐화한 알고리즘’을 컨텍스트안에서 바꿔주면서 상호 교체가 가능하게 만드는 패턴

▼ 그림 1-8 전략 패턴



- 결제 방식의 전략만 바꿔서 두 가지 방식으로 결제하는 전략 패턴 코드

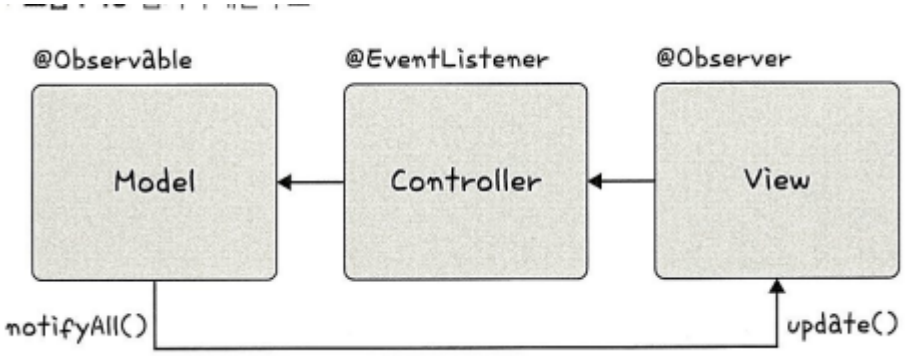
```
import java.text.DecimalFormat;
import java.util.ArrayList;
import java.util.List;
interface PaymentStrategy {
    public void pay(int amount);
}

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 addItem(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", "pukubababo"));
```

```
        // pay by KAKAOBank
        cart.pay(new KAKAOCardStrategy("Ju hongchul", "123456789", "123", "12/01"));
    }
}
```

1.1.4 옵저버 패턴

- 옵저버 패턴(observer pattern)은 주체가 어떤 객체의 상태 변화를 관찰하다가 상태 변화가 있을 때마다 메서드 등을 통해 옵저버 목록에 있는 옵저버들에게 변화를 알려주는 디자인 패턴
- 주체란 객체의 상태 변화를 보고 있는 관찰자이며, 옵저버들이란 이 객체의 상태 변화에 따라 전달되는 메서드 등을 기반으로 '추가 변화 사항'이 생기는 개체들을 의미
- 주체와 객체를 따로 두지 않고 상태가 변경되는 객체를 기반으로 구축하기도 한다.



- 옵저버 패턴은 주로 이벤트 기반 시스템에 사용하며 MVC 패턴에도 사용 된다.
- 예를 들어 주체라고 볼 수 있는 모델에서 변경사항이 생겨 update() 메서드로 옵저버인 뷰에 알려주고 이를 기반으로 컨트롤러가 작동

자바에서의 옵저버 패턴

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

```
interface Subject {
    public void register(Observer obj);
    public void unregister(Observer obj);
    public void notifyObservers();
    public Object getUpdate(Observer obj);
}

interface Observer {
    public void update();
}

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

    public Topic() {
        this.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(Observer obj) {
        return this.message;
    }

    public void postMessage(String msg) {
        System.out.println("Message sended 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(this);
        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("amumu is op champion!!");
    }
}
```

- topic을 기반으로 옵저버 패턴을 구현
- topic은 주체이자 객체
- class Topic implements Subject를 통해 Subject interface를 구현했고 Observer a = new TopicSubscriber(”a”, topic);으로 옵저버를 선언할 때 해당 이름과 어떠한 토픽의 옵저버가 될것인지를 정함

자바 상속과 구현의 차이

상속은 일반 클래스, abstract 클래스를 기반으로 구현하며, 구현은 인터페이스를 기반으로 구현

- 상속
  - 상속(extends)은 자식 클래스가 부모 클래스의 메서드 등을 상속 받아 사용하며 자식 클래스에서 추가 및 확장을 할 수 있는 것을 말한다.
  - 이로 인해 재사용성, 중복성의 최소화가 이루어진다.
- 구현
  - 구현(implements)는 부모 인터페이스(interface)를 자식 클래스에서 재정의하여 구현하는 것을 마하며, 상속과는 달리 반드시 부모 클래스의 메서드를 재정의하여 구현해야 한다.

js에서 옵저버 패턴

- js에서 옵저버 패턴은 프록시 객체르 통해 구현

프록시 객체

- 프록시(proxy)객체는 어떠한 대상의 기본적인 동작(속성 접근, 할당, 순회, 열거, 함수 호출)의 작업을 가로챌 수 있는 개체를 뜻하며, js에서 프록시 객체는 두 개의 매개변수를 가진다.
  - target : 프록시할 대상
  - handler : 프록시 객체의 target 동작을 가로채서 정의할 동작들이 정해져 있는 함수

프록시 객체를 이용한 옵저버 패턴

```
function createReactiveObject(target, callback) {
    const proxy = new Proxy(target, {
        set(obj, prop, value){
            if(value !== obj[prop]){
                const prev = obj[prop]
                obj[prop] = value
                callback(`${prop}가 [${prev}] >> [${value}] 로 변경되었습니다`)
            }
            return true
        }
    })
    return proxy
}
const a = {
    "형규" : "솔로"
}
const b = createReactiveObject(a, console.log)
b.형규 = "솔로"
b.형규 = "커플"
// 형규가 [솔로] >> [커플] 로 변경되었습니다
```

- 프록시 객체의 get() 함수는 속성과 함수에 대한 접근을 가로채며, has() 함수는 in 연산자의 사용을 가로챈다 . set() 함수는 속성에 대한 접근을 가로챈다.
- set(). 함수를 통해 속성에 대한 접근을 가로채서 형규라는 속성이 솔레어서 커플로 되는 것을 감시할 수 있다.

Vue.js 3.0의 옵저버 패턴

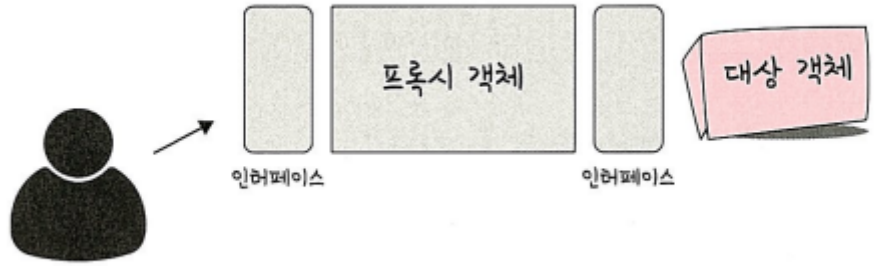
- ref나 reactive로 정의하면 해당 값이 변경되었을 때 자동으로 DOM에 있는 값이 변경되는데, 이는 앞서 설명한 프록시 객체를 이용한 옵저버 패턴을 이용하여 구현
- DOM(Document Object Model)
    - 문서 객체 모델을 말하며, 웹 브라우저상의 화면을 이루고 있는 요소를 지칭

```
function createReactiveObject(
    target: Target,
    isReadonly: boolean,
    baseHandlers: ProxyHandler < any > ,
    collectionHandlers: ProxyHandler < any > ,
    proxyMap: WeakMap < Target, any >
) {
    if (!isObject(target)) {
        if (__DEV__) {
            console.warn(`value cannot be made reactive: ${String(target)}`)
        }
        return target
    }
    // target is already a Proxy, return it.
    // exception: calling readonly() on a reactive object
    if (
        target[ReactiveFlags.RAW] &&
        !(isReadonly && target[ReactiveFlags.IS_REACTIVE])
    ) {
        return target
    }
    // target already has corresponding Proxy
    const existingProxy = proxyMap.get(target)
    if (existingProxy) {
        return existingProxy
    }
    // only a whitelist of value types can be observed.
    const targetType = getTargetType(target)
    if (targetType === TargetType.INVALID) {
        return target
    }
    const proxy = new Proxy(
        target,
        targetType === TargetType.COLLECTION ? collectionHandlers :
        baseHandlers
    )
    proxyMap.set(target, proxy)
    return proxy
}
```

1.1.5 프록시 패턴과 프록시 서버

- 프록시 패턴
  - 프록시 패턴은 대상 객체에 접근하기 전 그 접근에 대한 흐름을 가로채 대상 객체 앞단의 인터페이스 역할을 하는 디자인 패턴

▼ 그림 1-15 프록시 패턴



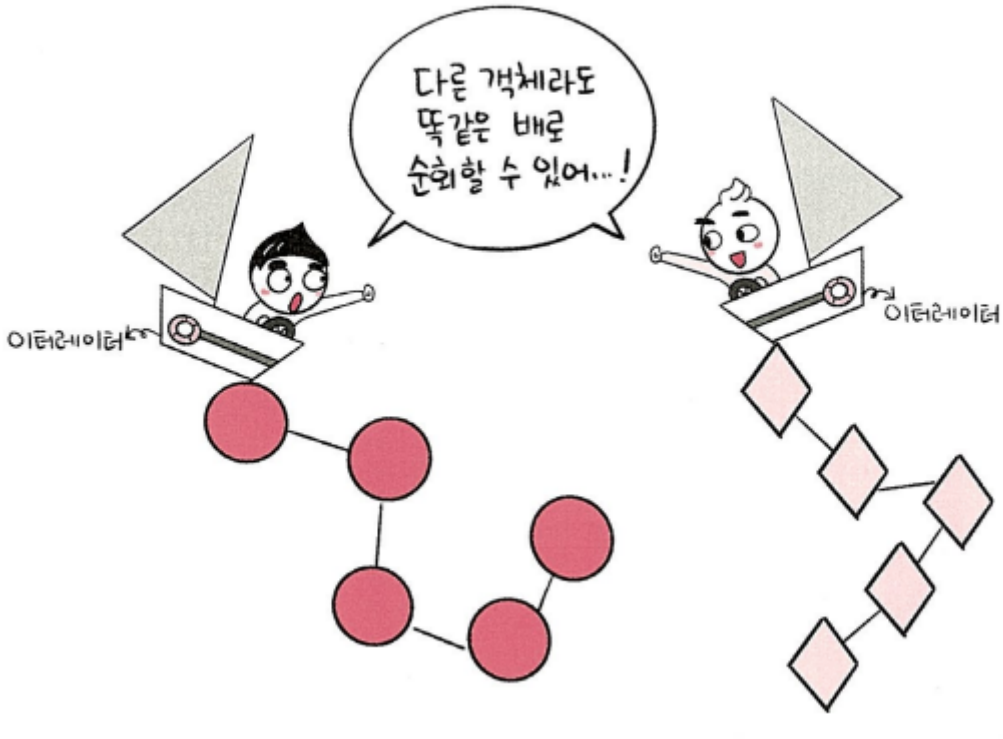
  - 이를 통해 객체의 속성, 변환 등을 보완하며 보안, 데이터 검증, 캐싱, 로깅에 사용
  - 이는 앞서 설명한 프록시 객체로 쓰이기도 하지만 프록시 서버로도 활용

- 프록시 서버
  - 프록시 서버는 서버와 클라이언트 사이에서 클라이언트가 자신을 통해서 다른 네트워크 서비스에 간접적으로 접속할 수 있게 해주는 컴퓨터 시스템이나 응용 프로그램을 가리킨다.

1.1.6 이터레이터 패턴

- 이터레이터 패턴(iterator pattern)은 이터레이터를 사용하여 컬렉션의 요소들에 접근하는 디자인 패턴

▼ 그림 1-22 이터레이터 패턴



- 순회 할 수 있는 여러가지 자료형의 구조와는 상관없이 이터레이터라는 하나의 인터페이스로 순회 가능

자바스크립트에서의 이터레이터 패턴

```
const mp = new Map()
mp.set('a', 1)
mp.set('b', 2)
mp.set('cccc', 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)
/*
a, b, c
[ 'a', 1 ]
[ 'b', 2 ]
[ 'c', 3 ]
1
2
```

```
3
*/
```

- 다른 자료구조인 set과 map임에도 똑같은 for a of b 라는 이터레이터 프로토콜을 통해 순회하는 것을 볼 수 있음

1.1.7 노출모듈 패턴

- 노출 모듈 패턴은 즉시 실행함수를 통해 private , public 과 같은 접근 제어자를 만드는 패턴이다.
- js는 private이나 public같은 접근 제어자가 존재하지 않고 전역 범위에서 스크립트가 실행된다.
- 그렇기에 노출모듈 패턴을 통해 private 와 public 접근 제어자를 구현하기도 한다.

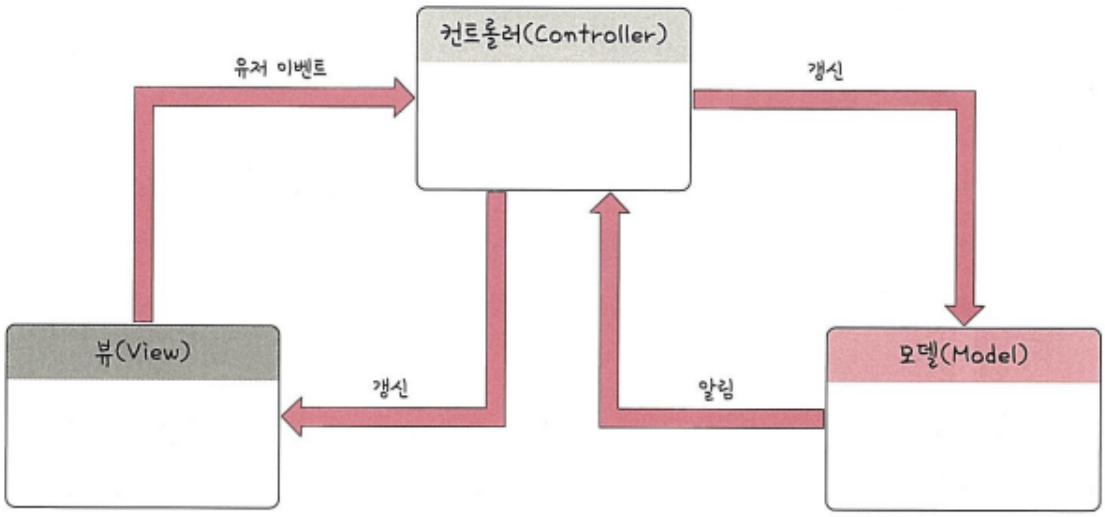
```
const pukuba = (() => {
    const a = 1
    const b = () => 2
    const public = {
        c : 2,
        d : () => 3
    }
    return public
})()
console.log(pukuba)
console.log(pukuba.a)
// { c: 2, d: [Function: d] }
// undefined
```

- a와 b는 다른 모듈에서 사용할 수 있는 변수나 함수인 private 범위를 가진다.
- 다른 모듈에서 접근할 수 없고 c와 d는 다른 모듈에서 사용할 수 있는 변수나 함수인 public범위를 가진다.
- 이 원리를 기반으로 만든 js 모듈 방식으로는 CJS(CommonJS)모듈 방식이 있다.

1.1.8 MVC 패턴

- MVC 패턴은 모델, 뷰, 컨트롤러로 이루어진 디자인 패턴이다.

▼그림 1-23 MVC 패턴



- 애플리케이션의 구성 요소를 세 가지 역할로 구분하여 개발 프로세스에서 각각의 구성 요소에만 집중해서 개발할 수 있다.
- 재상용성과 확장성이 용이하다는 장점이 있고, 애플리케이션이 복잡해질수록 모델과 뷰의 관계가 복잡해지는 단점이 있다.

모델

- 모델은 애플리케이션의 db, 상수, 변수 등을 뜻한다.
- 예를 들어 사각형 모양의 박스 안에 글자가 들어 있다면 그 사각형 모양의 박스 위치 정보, 글자 내용, 글자 위치, 글자 포맷등에 관한 정보를 모두 가지고 있어야 한다.
- 뷰에서 데이터를 생성하거나 수정하면 컨트롤러를 통해 모델을 생성하거나 갱신한다.

뷰

- 뷰는 inputbox, checkbox, textarea등 사용자 인터페이스 요소를 나타낸다. 즉, 모델을 기반으로 사용자가 볼 수 있는 화면을 뜻한다.
- 모델이 가지고 있는 정보를 따로 저장하지 않아야하며 단순히 사각형 모양 등 화면에 표시하는 정보만 가지고 있어야 한다.
- 또한 변경이 일어나면 컨트롤러에 이를 전달해야 한다.

컨트롤러

- 컨트롤러는 하나 이상의 모델과 하나 이상의 뷰를 잇는 다리 역할을 하며 이벤트 등 메인 로직을 담당한다.
- 또한 모델과 뷰의 생명주기도 관리하며, 모델이나 뷰의 변경 통지를 받으면 이를 해석하여 각가의 구성 요소에 해당 내용에 대해 알려준다.

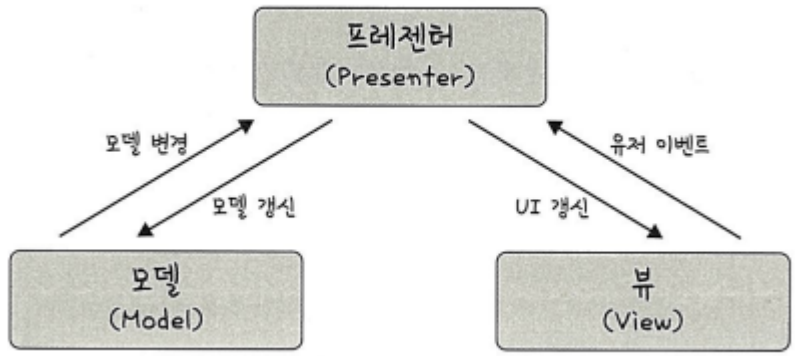
리액트

- mvc패턴을 이용한 대표적인 라이브러리
- 리액트는 유저 인터페이스를 구축하기 위한 라이브러리
- 가상 'DOM'을 통해 실제 DOM을 조작하는 것을 추상화해서 성능을 높였다.
- 대표적인 특성으로는 불변성(immutable)이 있다.
- 예를 들어 state는 setStage 를 통해서만 수정이 가능하고, props를 기반으로 해서 만들어지는 컴포넌트인 pureComponent가 있다.
- 단반향 바인딩이 적용되어 있고, 자유도가 높고, 메타가 운영하고 있다.

1.1.9 MVP패턴

- MVP패턴은 MVC패턴으로부터 파생되었으며 C에 해당하는 컨트롤러가 프레젠터로 교체된 패턴

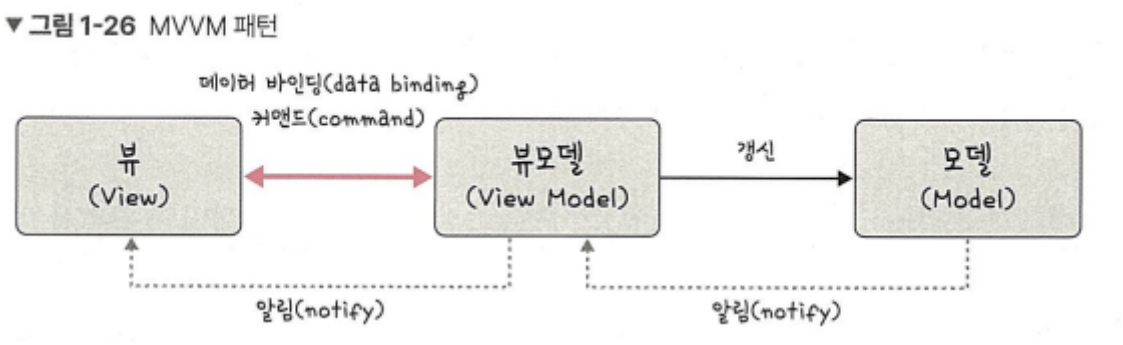
▼그림 1-25 MVP 패턴



- 뷰와 프레젠터는 일대일 관계이기 때문에 MVC패턴보다 더 강한 결합을 지닌 디자인 패턴이다.

1.1.10 MVVM 패턴

- MVVM패턴은 MVC의 C에 해당하는 컨트롤러가 뷰모델(view model)로 바뀐 패턴



- 뷰 모델은 뷰를 더 추상화한 계층이며, MVVM패턴은 MVC 패턴과는 다르게 커맨드와 데이터 바인딩을 가지는 것이 특징
- 뷰와 뷰모델 사이의 양방향 데이터 바인딩을 지원하며 UI를 별도의 코드 수정 없이 재사용할 수 있고 단위 테스팅 하기 쉽다는 장점이 잇다.

MVVM 패턴의 예 : VUE

- 대표적인 프레임워크로는 뷰가 있다.
- Vue.js는 반응형이 특징인 프런트엔드 프레임워크이다. 예를 들어 watch와 computed 등으로 쉽게 반응형적인 값들을 구축할 수 있다.
- 함수를 사용하지 않고 값 대입만으로도 변수가 변경되며 양반향 바인딩, html을 토대로 컴포넌트를 구축할 수 있다는 점이 특징이다. 재사용 가능한 컴포넌트 기반으로 UI를 구축할 수 있다.

- 데이터 바인딩
    - 화면에 보이는 데이터와 웹 브라우저의 메모리 데이터를 일치시키는 기법으로, 뷰 모델을 변경하면 뷰가 변경된다.