

Ch.1 정리

1. 디자인 패턴

- 프로그램을 설계할 때 발생했던 문제점들을 객체 간의 상호 관계 등을 이용하여 해결 할 수 있도록 하나의 '규약' 형태로 만들어 놓은 것을 의미한다. → 소프트웨어 개발에서 자주 발생하는 문제들을 해결하기 위해 만들어진 모범 사례

2. 디자인 패턴의 종류

- 2.1 - 싱글톤 패턴(singleton pattern)
- 2.2 - 팩토리 패턴(factory pattern)
- 2.3 - 전략 패턴(strategy pattern)
- 2.4 - 옵저버 패턴(observer pattern)
- 2.5 - 프록시 패턴(proxy pattern)
- 2.6 - MVC 패턴(model,view,controller pattern)
- 2.7 - MVP 패턴(model,view,presenterpattern)
- 2.8 - MVVM 패턴(model,view,view model pattern)

2.1 싱글톤 패턴(Singleton Pattern)

- 하나의 클래스에 오직 하나의 인스턴스만 가지는 패턴

사용 예시) 데이터베이스 연결 모듈

장점 : 인스턴스를 생성할 때 드는 비용이 줄어듦

단점 : 의존성이 높아짐

Test Code

```
Execute | Beautify | Share | Source Code | Help
1 class Singleton {
2     private static class singleInstanceHolder {
3         private static final Singleton INSTANCE = new Singleton();
4     }
5     public static Singleton getInstance() {
6         return singleInstanceHolder.INSTANCE;
7     }
8 }
9
10 public class HelloWorld{
11
12     public static void main(String []args){
13         Singleton a = Singleton.getInstance();
14         Singleton b = Singleton.getInstance();
15         System.out.println(a.hashCode());
16         System.out.println(b.hashCode());
17         if (a == b) {
18             System.out.println(true);
19         }
20     }
21 }
```

```
Terminal
2868468723
2868468723
true
```

→ a와 b는 하나의 인스턴스를 가진다.

2.2 팩토리 패턴(Factory Pattern)

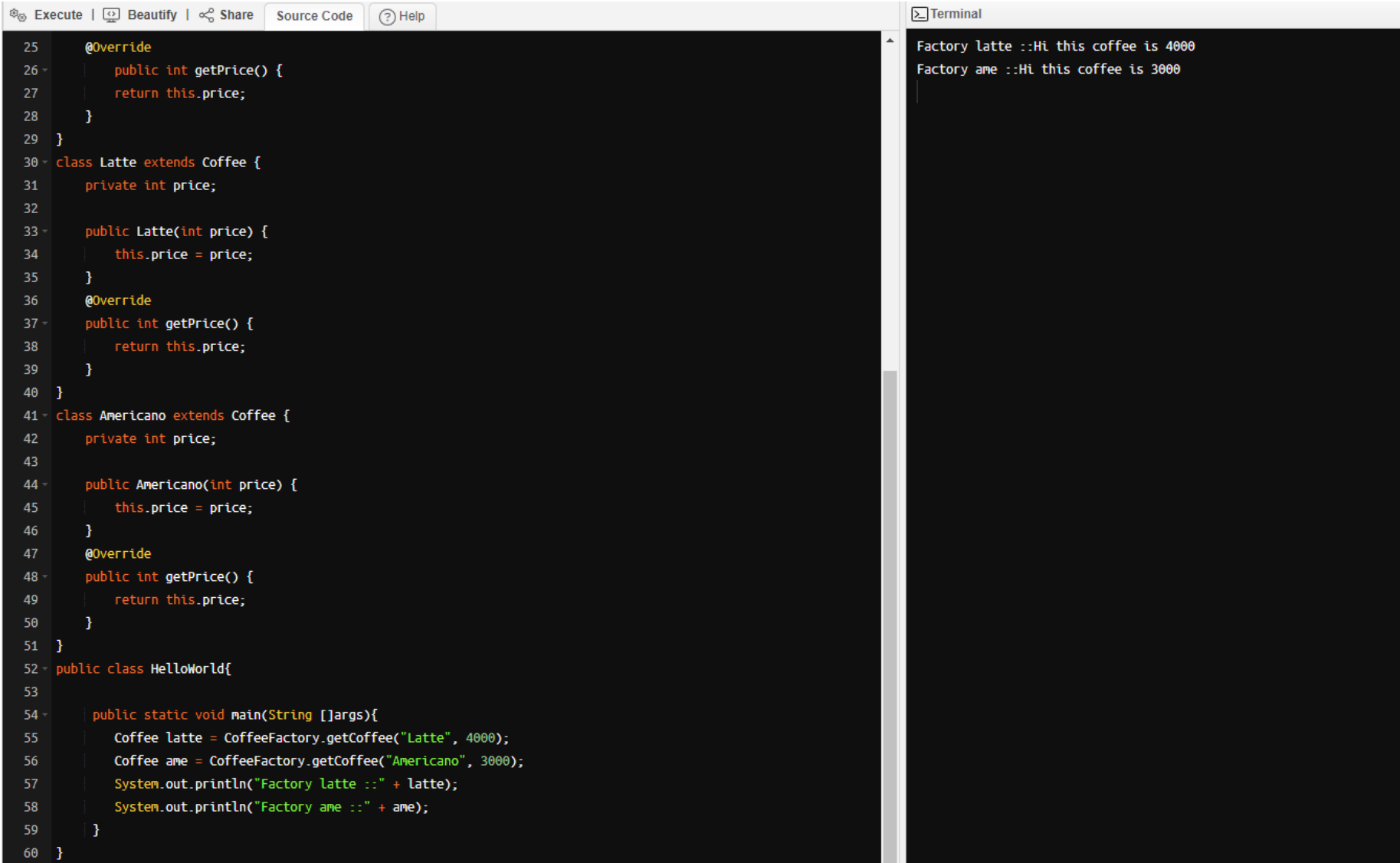
- 객체를 사용하는 코드에서 객체 생성 부분을 떼어내 추상화한 패턴이자 상속 관계에 있는 두 클래스에서 상위 클래스가 중요한 뼈대를 결정하고, 하위 클래스에서 객체 생성에 관한 구체적인 내용을 결정하는 패턴

사용 예시) 데이터베이스 연결 객체 생성, GUI 컴포넌트 생성, 문서 생성 시스템

장점 : 유연성 및 유지 보수성 증가

단점 : 복잡성 증가, 추가 추상화 계층 요구

Test Code



→ CoffeeFactory라는 상위 클래스가 중요한 뼈대를 결정하고 하위 클래스인 LatteFactory가 구체적인 내용을 결정한다.

2.3 전략 패턴(Strategy Pattern)

- 객체의 행위를 바꾸고 싶은 경우 '직접' 수정하지 않고 전략이라고 부르는 '캡슐화한 알고리즘'을 컨텍스트 안에서 바꿔주면서 상호 교체가 가능하게 만드는 패턴

사용 예시) 정렬 알고리즘, 데이터 압축, 지불 방식 선택

장점 : 유연성 증가, 코드 중복 감소, 테스트 용이, 클래스 응집도 향상

단점 : 클래스 수 증가, 코드 복잡성 증가, 전략 간의 의존성 문제

Test Code

```
67 }
68
69 public void removeItem(Item item) {
70     this.items.remove(item);
71 }
72
73 public int calculateTotal() {
74     int sum = 0;
75     for (Item item : items) {
76         sum += item.getPrice();
77     }
78     return sum;
79 }
80
81 public void pay(PaymentStrategy paymentMethod) {
82     int amount = calculateTotal();
83     paymentMethod.pay(amount);
84 }
85 }
86
87 public class HelloWorld {
88     public static void main(String[] args) {
89         ShoppingCart cart = new ShoppingCart();
90
91         Item A = new Item("kundo1A", 100);
92         Item B = new Item("kundo1B", 300);
93
94         cart.addItem(A);
95         cart.addItem(B);
96
97         // pay by LUNACard
98         cart.pay(new LUNACardStrategy("kundol@example.com", "pukubababo"));
99
100        // pay by KAKAOCARD
101        cart.pay(new KAKAOCARDStrategy("Ju hongchul", "123456789", "123", "12/01"));
102    }
103 }
```

```
400 paid using LUNACard.
400 paid using KAKAOCARD.
```

→ 카카오텔드는 `String nm, String ccNum, String cvv, String expiryDate` 를 입력받고 루나카드는 `String email, String pwd` 을 입력 받는다.

2.4 옵저버 패턴(Observer Pattern)

- 주체가 어떤 객체(subject)의 상태 변화를 관찰하다가 상태 변화가 있을 때마다 메서드 등을 통해 옵저버 목록에 있는 옵저버들에게 변화를 알려주는 디자인 패턴

사용 예시) 이벤트 기반 시스템

장점 : 느슨한 결합, 동적 관계 설정, 확장성, 실시간 업데이트

단점 : 복잡성 증가, 예측 불가능한 갱신 순서, 성능 문제, 메모리 누수 가능성 증가

Test Code

```
Execute | Beautify | Share | Source Code | Help

44 - public void postMessage(String msg) {
45     System.out.println("Message sent to Topic: " + msg);
46     this.message = msg;
47     notifyObservers();
48 }
49 }
50
51 - class TopicSubscriber implements Observer {
52     private String name;
53     private Subject topic;
54
55 - public TopicSubscriber(String name, Subject topic) {
56     this.name = name;
57     this.topic = topic;
58 }
59
60 @Override
61 - public void update() {
62     String msg = (String) topic.getUpdate(this);
63     System.out.println(name + ":: got message >> " + msg);
64 }
65 }
66
67 - public class HelloWorld {
68 - public static void main(String[] args) {
69     Topic topic = new Topic();
70     Observer a = new TopicSubscriber("a", topic);
71     Observer b = new TopicSubscriber("b", topic);
72     Observer c = new TopicSubscriber("c", topic);
73     topic.register(a);
74     topic.register(b);
75     topic.register(c);
76
77     topic.postMessage("anumu is op champion!!");
78 }
79 }
```

```
Terminal
Message sent to Topic: anumu is op champion!!
a:: got message >> anumu is op champion!!
b:: got message >> anumu is op champion!!
c:: got message >> anumu is op champion!!
```

→ `Observer a = new TopicSubscriber("a", topic);` 으로 옵저버를 선언할 때 해당 이름과 어떠한 토픽의 옵저버가 될 것인지를 정했다.

2.5 프록시 패턴(Proxy Pattern)

- 대상 객체(subject)에 접근하기 전 그 접근에 대한 흐름을 가로채 대상 객체 앞단의 인터페이스 역할을 하는 디자인 패턴

사용 예시) 객체의 속성, 변환 등을 보완하며 보안, 데이터 검증, 캐싱, 로깅 및 프록시 서버

장점 : 접근 제어, 성능 최적화, 로깅 및 감시, 원격 프록시, 인터페이스 일관성 유지

단점 : 추가적인 레이어, 성능 저하, 디버깅 어려움, 복잡성 증가

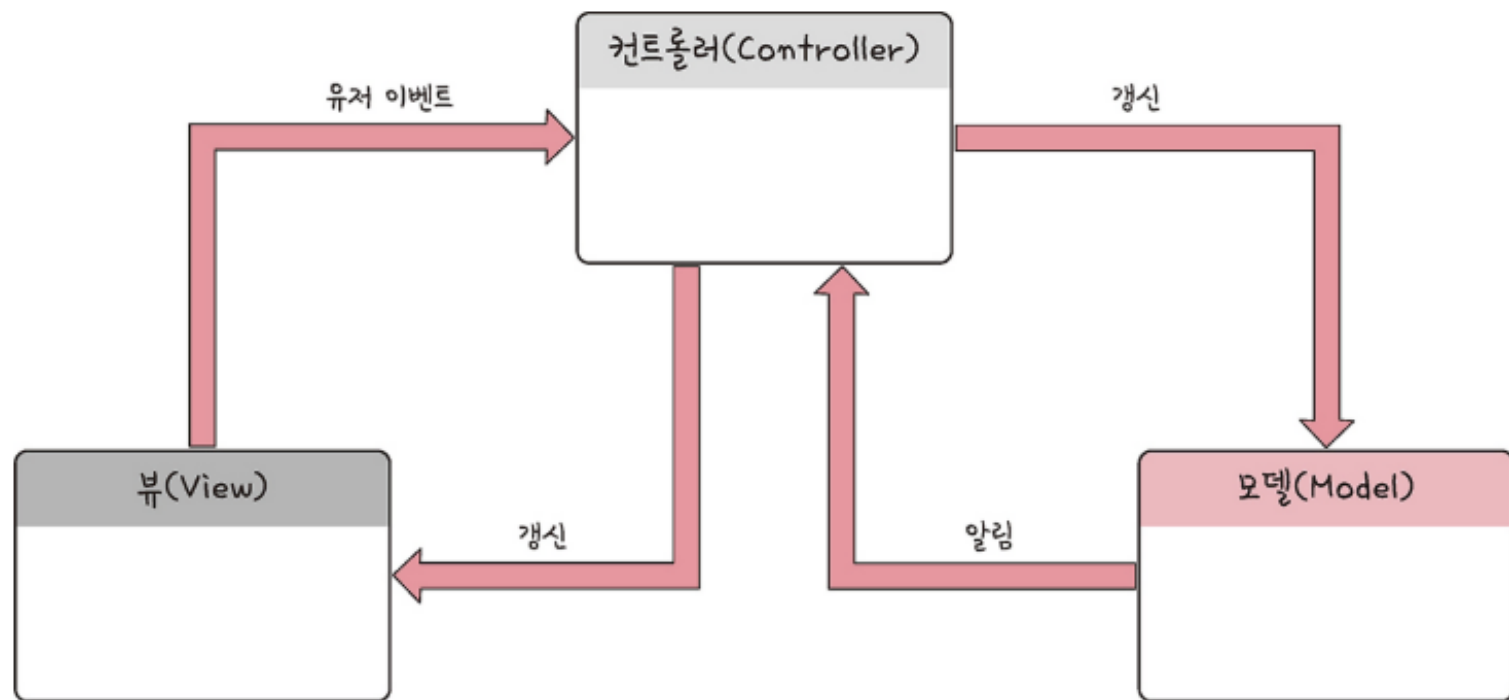
2.6 MVC 패턴(Model,View,Controller Pattern)

- MVC 패턴은 모델(Model), 뷰(View), 컨트롤러(Controller)로 이루어진 디자인 패턴

사용 예시) React.JS

장점 : 재사용성과 확장성이 용이

단점 : 애플리케이션이 복잡해질수록 모델과 뷰의 관계가 복잡해짐



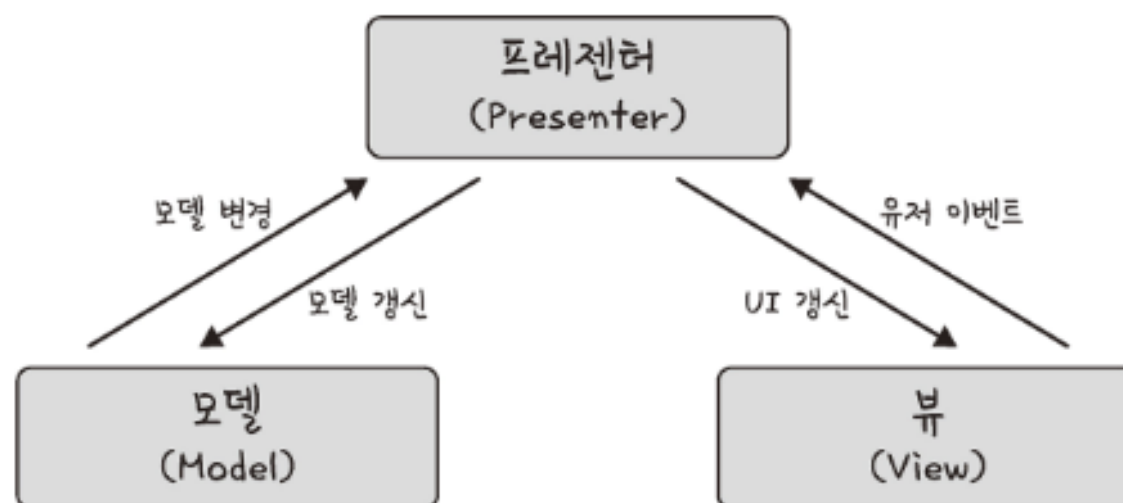
2.7 MVP 패턴(Model, View, PresenterPattern)

- MVP 패턴은 MVC 패턴으로부터 파생되었으며 MVC에서 C에 해당하는 컨트롤러가 프레젠테터(presenter)로 교체된 패턴 → MVC 패턴 보다 더 강한 결합을 지닌 디자인 패턴

사용 예시) 뷰와 비즈니스 로직의 분리가 중요한 경우에 유용하게 사용

장점 : 명확한 역할 분리, 테스트 용이, 유연성, 재사용성

단점 : 초기 설정 복잡성, 추가적인 코드 작성, 상호 참조 문제



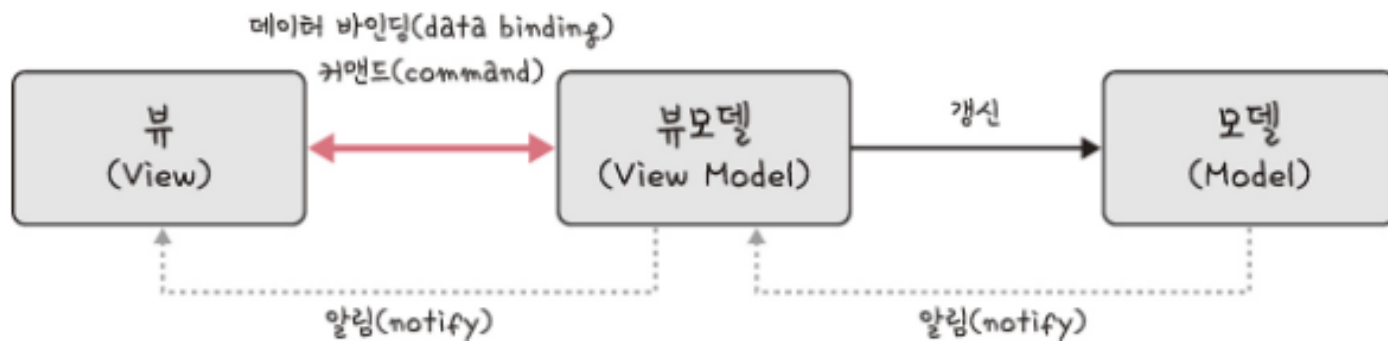
2.8 MVVM 패턴(Model, View, View ModelPattern)

- MVVM 패턴은 MVC의 C에 해당하는 컨트롤러가 뷰모델(view model)로 바뀐 패턴 → MVVM 패턴은 MVC 패턴과는 다르게 커맨드와 데이터 바인딩을 가진다.

사용 예시) Vue.JS

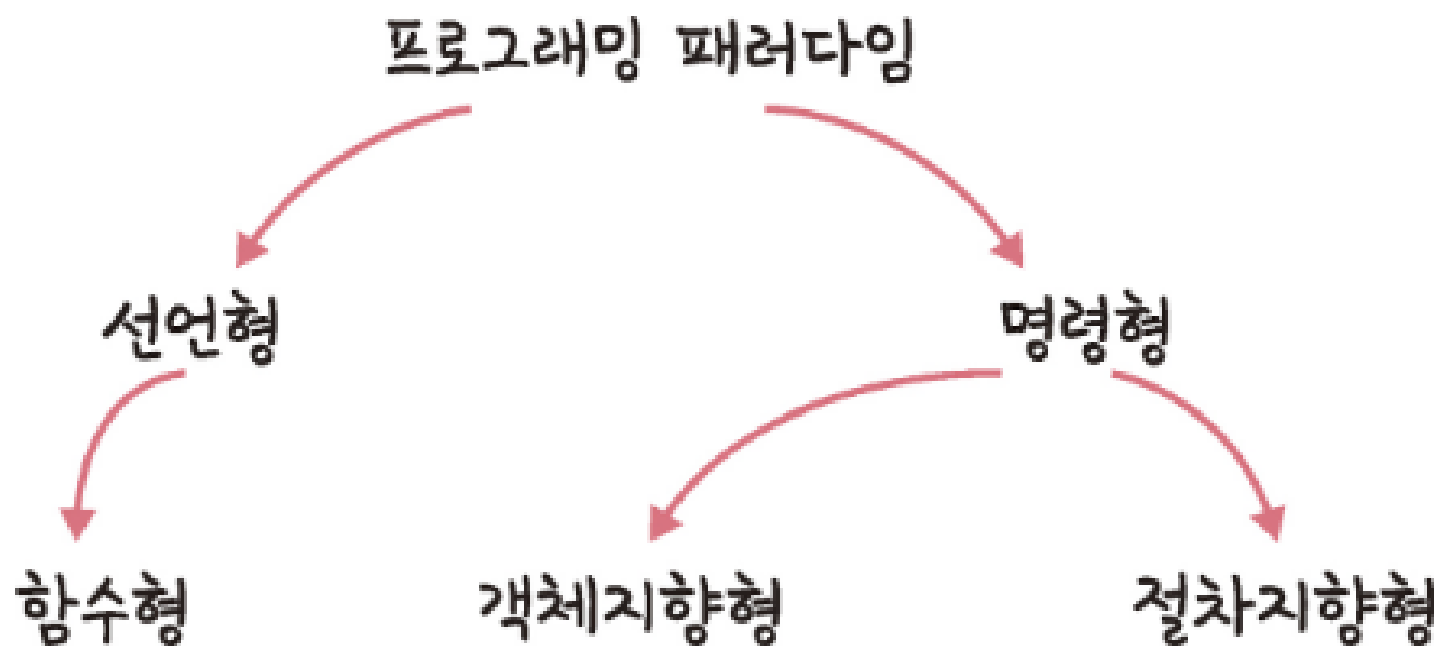
장점 : 뷰와 뷰모델 사이의 양방향 데이터 바인딩을 지원하며 UI를 별도의 코드 수정 없이 재사용할 수 있고 단위 테스트하기 쉬움

단점 : 복잡성 증가, 디버깅 어려움, 메모리 누수, 테스트 어려움, 플랫폼 종속성



3. 프로그래밍 패러다임(Programming Paradigm)

- 프로그래밍 패러다임(programming paradigm)은 프로그래머에게 프로그래밍의 관점을 갖게 해주는 역할을 하는 개발 방법론입니다.



3.1 선언형과 함수형 프로그래밍

- '무엇을' 풀어내는가에 집중하는 패러다임

3.2 객체지향 프로그래밍

- 객체들의 집합으로 프로그램의 상호 작용을 표현하며 데이터를 객체로 취급하여 객체 내부에 선언된 메서드를 활용하는 방식

Test Code

```
Execute | Beautify | Share | Source Code | Help | Terminal

1 import java.util.List;
2
3 class ListExample {
4     private List<Integer> list;
5     private int mx;
6
7     public ListExample(List<Integer> list) {
8         this.list = list;
9         this.mx = list.stream().reduce(0, (max, num) -> num > max ? num : max);
10    }
11
12    public int getMax() {
13        return this.mx;
14    }
15 }
16
17 public class HelloWorld {
18     public static void main(String[] args) {
19         List<Integer> ret = List.of(1, 2, 3, 4, 5, 11, 12);
20         ListExample a = new ListExample(ret);
21         System.out.println(a.getMax());
22     }
23 }
```

→List라는 클래스를 만들고 a라는 객체를 만들 때 최댓값을 추출해내는 메서드

3.2 객체지향 프로그래밍의 특징

- 추상화
- 캡슐화
- 상속성
- 다형성
- 설계원칙

* 추상화

복잡한 문제의 본질을 이해하기 위해 세부 사항은 배제하고 중요한 부분을 중심으로 간략화하는 기법

* 캡슐화

속성과 메소드를 하나로 묶어서 객체로 구성, 프로그램 변경에 대한 오류의 파급효과가 적다 → 재사용 용이

* 상속성

상위 클래스의 메소드와 속성을 하위 클래스가 물려받는 것

* 다형성

한 메시지가 객체에 따라 다른 방법으로 응답할 수 있는 것

장점 : 유연하고 변경이 용이하게 만들기 때문에 대규모 소프트웨어 개발에 많이 사용된다.

소프트웨어 개발과 보수를 간편하게 한다.

상속을 통한 재사용과 시스템의 확장 용이하다.

사용자와 개발자 사이의 이해를 쉽게 해준다.

단점 : 프로그래밍 구현을 지원해 주는 정형화된 분석 및 설계 방법이 부족하다.

* 설계원칙(SOLID원칙)

속성과 메소드를 하나로 묶어서 객체로 구성, 프로그램 변경에 대한 오류의 파급효과가 적다 → 재사용 용이

단일 책임 원칙(S)

단일 책임 원칙(SRP, Single Responsibility Principle)은 모든 클래스는 각각 하나의 책임만 가져야 하는 원칙

개방-폐쇄 원칙(O)

개방-폐쇄 원칙(OCF, Open Closed Principle)은 유지 보수 사항이 생긴다면 코드를 쉽게 확장할 수 있도록 하고 수정할 때는 닫혀 있어야 하는 원칙

리스코프 치환 원칙(L)

리스코프 치환 원칙(LSP, Liskov Substitution Principle)은 프로그램의 객체는 프로그램의 정확성을 깨뜨리지 않으면서 하위 타입의 인스턴스로 바꿀 수 있어야 하는 원칙

인터페이스 분리 원칙(I)

인터페이스 분리 원칙(ISP, Interface Segregation Principle)은 하나의 일반적인 인터페이스보다 구체적인 여러 개의 인터페이스를 만들어야 하는 원칙

의존 역전 원칙(D)

의존 역전 원칙(DIP, Dependency Inversion Principle)은 자신보다 변하기 쉬운 것에 의존하던 것을 추상화된 인터페이스나 상위 클래스를 두어 변하기 쉬운 것의 변화에 영향받지 않게 하는 원칙

3.3 절차형 프로그래밍

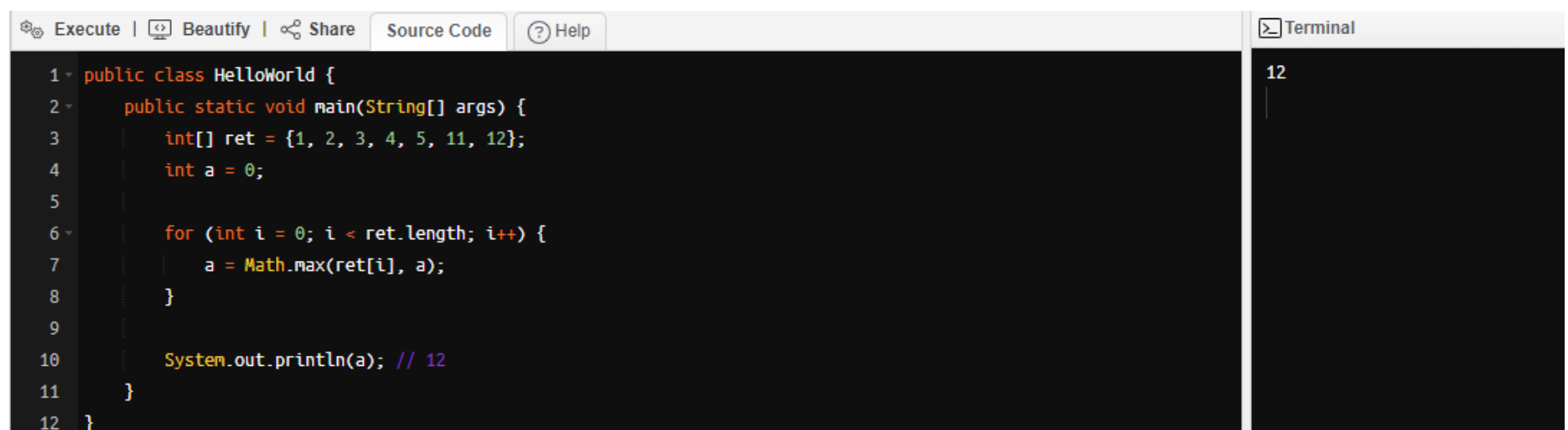
- 일련의 처리 절차를 정해진 문법에 따라 순서대로 기술해나가는 과정
- 절차형 프로그래밍은 로직이 수행되어야 할 연속적인 계산 과정으로 이루어져 있다.

사용 예시) 대기 과학 관련 연산 작업, 머신 러닝의 배치 작업

장점 : 코드의 가독성이 좋으며 실행 속도가 빠름

단점 : 모듈화하기가 어렵고 유지 보수성이 떨어짐

Test Code



```
1 public class HelloWorld {
2     public static void main(String[] args) {
3         int[] ret = {1, 2, 3, 4, 5, 11, 12};
4         int a = 0;
5
6         for (int i = 0; i < ret.length; i++) {
7             a = Math.max(ret[i], a);
8         }
9
10        System.out.println(a); // 12
11    }
12 }
```

→ 반복문을 통해 최대 값을 찾음

4. 패러다임의 혼합

- 비즈니스 로직이나 서비스의 특징을 고려해서 패러다임을 정하는 것이 좋다.
- 하나의 패러다임을 기반으로 통일하여 서비스를 구축하는 것도 좋지만 여러 패러다임을 조합하여 상황과 맥락에 따라 패러다임 간의 장점만 취해 개발하는 것이 더 좋다.

예시) 백엔드에 머신 러닝 파이프라인과 거래 관련 로직이 있다면 머신 러닝 파이프라인은 절차지향형 패러다임, 거래 관련 로직은 함수형 프로그래밍을 혼합하여 사용하는 것