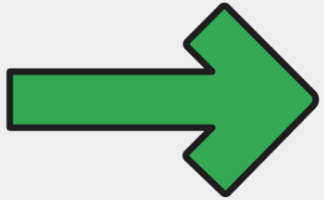
Google Developer Group
Editable University Name

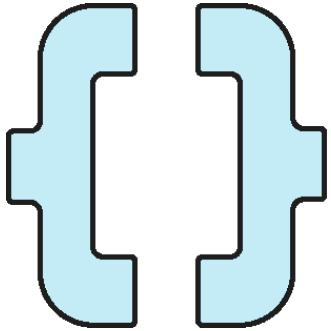
디자인 패턴

GDGoC INU Backend Part Seminar Member 조광현

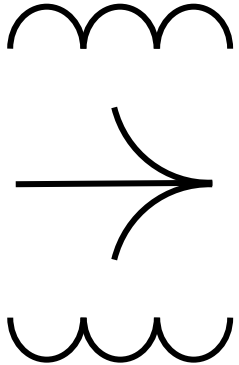


“디자인 패턴이란?”

디자인 패턴



소프트웨어 개발 방법으로 사용되는 디자인패턴(Desigin Pattern)은 과거의 소프트웨어 개발 과정에서 발견된 설계의 노하우를 축적하여 그 방법에 이름을 붙여서 이후에 재사용하기 좋은 형태로 특정 규약을 만들어서 정리한 것입니다.



디자인 패턴은 소프트웨어 설계에 있어 공통적인 문제들에 대한 표준적인 해법과 작명법을 제안하며, 알고리즘과 같이 프로그램 코드로 바로 변환될 수 있는 형태는 아니지만, 특정한 상황에서 구조적인 문제를 해결하는 방식입니다.

“  ”

“비유를 들어봅시다”

“

170

”

“

200개 이상

”

“같은 시행착오 겪을 필요 X”

객체 지향 프로그래밍 설계를 할 때
자주 발생하는 문제들을 피하기 위해
사용되는 패턴

“

디자인 패턴은 의사소통 수단의 일종으로서 여러 문제를 해결해 준다

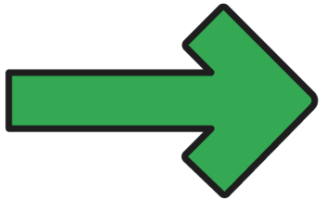
”

기능마다 별도의 클래스를 만들고.

그 기능들로 해야 할 일을 한 번에
처리해 주는 클래스를 만들자

기능마다 별도의 클래스를 만들고.

그 기능들로 해야 할 일을 한 번에
처리해 주는 클래스를 만들자



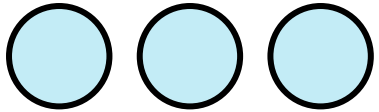
Facade 패턴을 써보자

기능마다 별

그 기능들로
처리해 주는

보자

디자인 패턴의 종류



생성 패턴

- 생성 팩토리
- 팩토리 메서드
- **빌더**
- 프로토타입
- 싱글톤

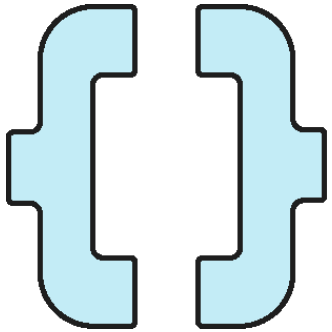
구조 패턴

- 어댑터
- 브리지
- 컴포지트
- 데코레이터
- **파사드**
- 플라이웨이트
- 프록시

행위 패턴

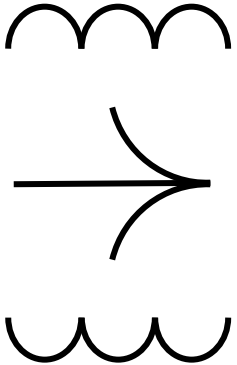
- 책임 체인
- 커맨드
- 인터프리터
- **반복자**
- 중재자
- 메멘토
- 옵저버
- 상태
- 전략
- 템플릿 메소드
- 방문자

생성 패턴



객체 생성에 관련된 패턴

객체의 생성과 조합을 캡슐화해 특정 객체가 생성되거나 변경되어도
프로그램 구조에 영향을 크게 받지 않도록 **유연성**을 제공

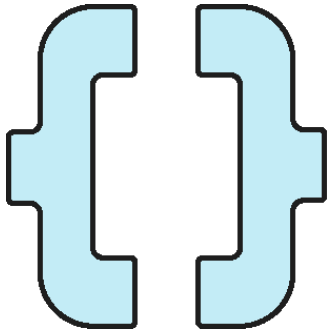


생성 패턴은 마치 공장에서 물건을 만드는 것처럼,
프로그램에서 필요한 데이터를 효율적으로 만들고 관리하는 방법

장점 : 프로그램이 더 유연하고 유지보수하기 쉬워짐

빌더 패턴

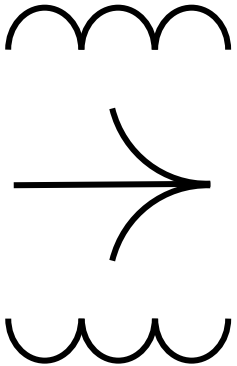
빌더 패턴



빌더라는 내부 클래스를 통해 간접적으로 생성하게 하는 패턴

복잡한 객체를 단계적으로 생성할 수 있도록 도와줌.

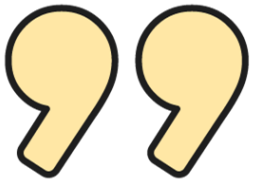
생성자의 매개변수가 많거나,
다양한 조합의 객체를 만들어야 할 때 유용하게 사용



```java
public class Person {  0개의 사용위치
    private String fullName;  3개 사용 위치
    private int yearsOld;  3개 사용 위치
    private String contactEmail;  2개 사용 위치
    private String homeAddress;  1개 사용 위치

    public Person(String fullName, int yearsOld) {  0개의 사용위치
        this.fullName = fullName;
        this.yearsOld = yearsOld;
    }
```

```java
    public Person(String fullName, int yearsOld, String contactEmail) {  0개의 사용위치
        this.fullName = fullName;
        this.yearsOld = yearsOld;
        this.contactEmail = contactEmail;
    }

    public Person(String fullName, int yearsOld, String homeAddress, String contactEmail) {  0
        this.fullName = fullName;
        this.yearsOld = yearsOld;
        this.homeAddress = homeAddress;
        this.contactEmail = contactEmail;
    }

    //....
}
```

오버로딩된 생성자가
너무 많아지는 문제가 발생



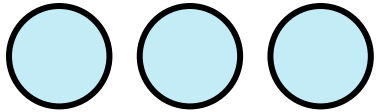
```java
public class Person {  2개 사용 위치
    private final String fullName;  1개 사용 위치
    private final int yearsOld;  1개 사용 위치
    private final String contactEmail;  1개 사용 위치
    private final String homeAddress;  1개 사용 위치

    private Person(Builder builder) {  1개 사용 위치
        this.fullName = builder.fullName;
        this.yearsOld = builder.yearsOld;
        this.contactEmail = builder.contactEmail;
        this.homeAddress = builder.homeAddress;
    }
```

```java
public static class Builder {  3개 사용 위치
    private String fullName;  2개 사용 위치
    private int yearsOld;  2개 사용 위치
    private String contactEmail;  2개 사용 위치
    private String homeAddress;  2개 사용 위치

    public Builder(String fullName, int yearsOld) {  0개의 사용위치
        this.fullName = fullName;
        this.yearsOld = yearsOld;
    }

    public Builder withEmail(String contactEmail) {  0개의 사용위치
        this.contactEmail = contactEmail;
        return this;
    }
```

```
        public Builder withAddress(String homeAddress) {  0개의 사용위치
            this.homeAddress = homeAddress;
            return this;
        }

        public Person create() {  0개의 사용위치
            return new Person( builder: this);
        }
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        // Builder를 사용하여 Person 객체 생성
        Person person = new Person.Builder( fullName: "John Doe",  yearsOld: 30)  // 필수 필드 설정
                .withEmail( contactEmail: "john.doe@example.com")  // 선택적 필드 설정
                .withAddress( homeAddress: "123 Main St, New York")  // 선택적 필드 설정
                .create();  // 최종적으로 객체 생성


    }
}
```

빌더 패턴 장점



가독성이 좋아짐

어떤 값이 설정되는지 명확하게 보인다

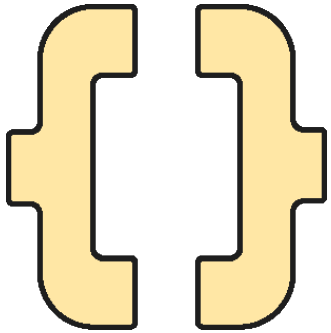
유연성 증가

필요한 값만 선택적으로 설정 가능

불변 객체 생성 가능

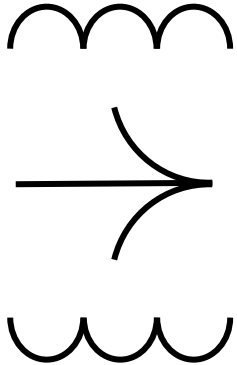
필드를 final로 선언하여 변경 불가능한 객체를 만들 수 있음

구조 패턴



클래스나 객체를 조합해 더 큰 구조를 만드는 패턴

서로 다른 인터페이스를 지닌 2개의 객체를 묶어 단일 인터페이스를 제공하거나 서로 다른 객체들을 묶어 **새로운 기능을 제공하는 패턴**

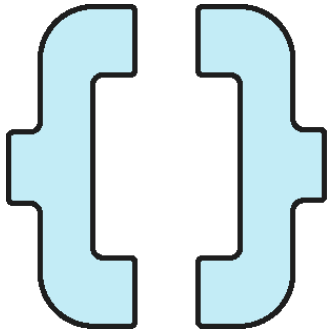


구조 패턴은 마치 레고 블록을 조립해서 멋진 건축물을 만드는 것처럼, 프로그램 내의 여러 객체들을 효율적으로 연결하고 관리하는 방법이다

장점 : 다양한 구조를 통해 클래스와 객체의 관계를 강화

“Fasade 패턴”

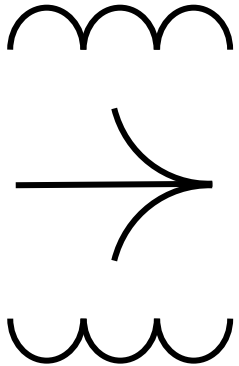
파사드 패턴



내부의 복잡한 처리들을 대신 수행해서 결과만 넘겨주는 객체

은행의 창구에서 근무하는 은행원 같은 역할이라 생각하면 쉽다

파사드는 건물의 출입구가 있는 정면을 가리키는 단어



시스템의 복잡성을 감추고, 사용자(Client)가 시스템에 접근할 수 있는 인터페이스(Interface)를 사용자(Client)에게 제공

```java
// 서브시스템 클래스들
class SubsystemA {  2개 사용 위치
    public String operationA() {  1개 사용 위치
        return "SubsystemA: Operation A";
    }
}

class SubsystemB {  2개 사용 위치
    public String operationB() {  1개 사용 위치
        return "SubsystemB: Operation B";
    }
}

class SubsystemC {  2개 사용 위치
    public String operationC() {  1개 사용 위치
        return "SubsystemC: Operation C";
    }
}
```

```java
// 파사드 클래스
class Facade {  2개 사용 위치
    private SubsystemA subsystemA;  2개 사용 위치
    private SubsystemB subsystemB;  2개 사용 위치
    private SubsystemC subsystemC;  2개 사용 위치

    public Facade() {  1개 사용 위치
        this.subsystemA = new SubsystemA();
        this.subsystemB = new SubsystemB();
        this.subsystemC = new SubsystemC();
    }

    public void performOperations() {  1개 사용 위치
        System.out.println(subsystemA.operationA());
        System.out.println(subsystemB.operationB());
        System.out.println(subsystemC.operationC());
    }
}
```

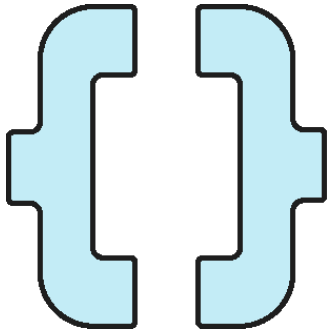
```java
// 클라이언트 코드
public class Main {  0개의 사용위치  신규 *
    public void main(String[] args) {  신규 *
        Facade facade = new Facade();
        facade.performOperations();
    }
}
```

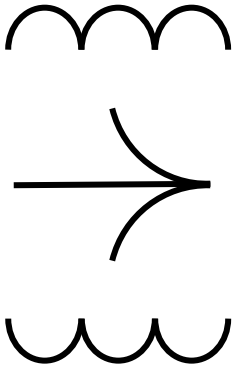
파사드 패턴 장점

복잡성 감소
클라이언트는 서브시스템의 복잡성을 알 필요 없이 간단한 인터페이스만 사용하면 됩니다.

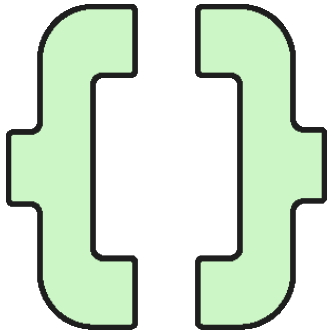
결합도 감소
클라이언트와 서브시스템 간의 결합도를 낮춰 유지보수가 용이해집니다.

유연성 증가
서브시스템의 내부 구현을 변경하더라도 파사드 인터페이스는 유지할 수 있습니다.



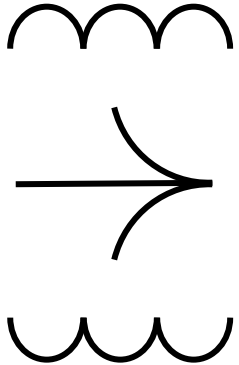


행위 패턴



객체나 클래스 사이의 알고리즘이나 책임 분배에 관련된 패턴

한 객체가 혼자 수행할 수 없는 작업을 여러개의 객체로 어떻게 분배하는지,
또 그렇게 하면서도 객체 사이의 결합도를 최소화하는 것에 중점을 두는 방식

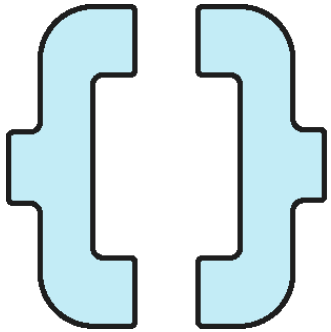


행위 패턴은 마치 팀원이 각자의 역할과 소통 방식을 정리해서 협력하는 것처럼,
프로그램 내 객체들이 효율적으로 협력하고 책임을 나눌 수 있도록 돕는 설계 방식

장점 : 복잡한 흐름 제어를 간소화하고, 객체 간의 상호작용을 더 효과적으로 만듦.

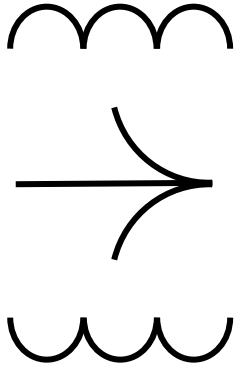
Iterator(반복자) 패턴

반복자(iterator) 패턴



일련의 데이터 집합에 대하여 순차적인 접근(순회)을 지원하는 패턴

이 패턴은 컬렉션(리스트, 배열 등)의 내부 구조를 노출하지 않고도
요소들을 순차적으로 접근할 수 있도록 도와줌



```java
// Iterator 인터페이스
interface Iterator<T> {  2개 사용 위치
    boolean hasNext();  1개 사용 위치
    T next();
}
```

반복자가 가져야 할 기본 동작을 정의하는 인터페이스

hasNext(): 다음 요소가 있는지 확인합니다.
next(): 현재 요소를 반환하고 다음 요소로 이동합니다.

```java
// Concrete Iterator 클래스
class ArrayIterator<T> implements Iterator<T> {  1개 사용 위치
    private T[] array;  3개 사용 위치
    private int index = 0;  2개 사용 위치

    public ArrayIterator(T[] array) {  1개 사용 위치
        this.array = array;
    }

    @Override  1개 사용 위치
    public boolean hasNext() {
        return index < array.length;
    }

    @Override
    public T next() {
        return array[index++];
    }
}
```

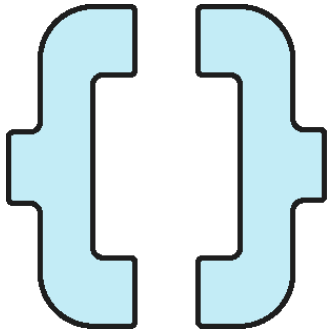
Iterator 인터페이스를 구현한 반복자의 실제 동작을 정의

클라이언트 코드에서는 ArrayIterator를 사용하여 배열의 요소를 순차적으로 접근

```
// Client 코드
public class Main {
    public static void main(String[] args) {
        String[] fruits = {"apple", "banana", "orange"};

        // 반복자 생성
        Iterator<String> iterator = new ArrayIterator<>(fruits);

        // 반복자를 사용하여 배열 순회
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
    }
}
```

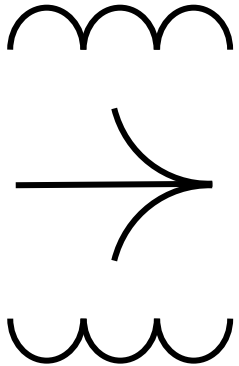
클라이언트 코드에서는 ArrayIterator를 사용하여 배열의 요소를 순차적으로 접근

어디서 많이 봤는데



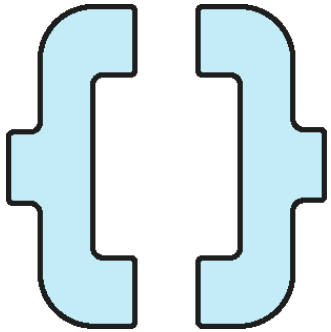
1. Java Collections Framework

자바에서 반복자 패턴은 Java Collection Framework에서 광범위하게 사용된다. 특히, java.util.Iterator와 java.util.Enumeration 인터페이스는 이 패턴의 대표적인 구현



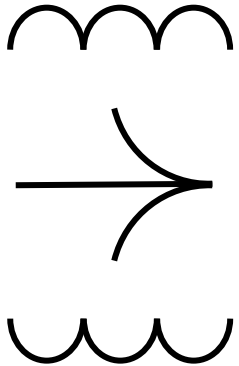
ArrayList, HashSet, LinkedList 등 대부분의 컬렉션 클래스는 iterator() 메서드를 제공하며, 이를 통해 컬렉션 요소를 순회할 수 있다.

반복자 패턴 장점



1.코드의 단순화 및 가독성 향상:
반복 로직이 캡슐화되어 클라이언트 코드가 간결해지고 가독성이 높아집니다.
클라이언트는 복잡한 탐색 로직을 신경 쓰지 않아도 됩니다

2.컬렉션과 탐색 로직의 분리:
컬렉션의 내부 구조와 탐색 알고리즘을 분리하여 유지보수성과 확장성을 높입니다.
새로운 탐색 방식이나 컬렉션 구조를 추가해도 기존 클라이언트 코드는 영향을 받지 않습니다



3.동시 탐색 지원:
여러 반복자가 동일한 컬렉션을 독립적으로 탐색할 수 있습니다. 각 반복자가 자신의 상태를 관리하므로 병렬 처리가 가능합니다

Google Developer Group
Editable University Name

감사합니다

GDGoC INU Backend Part Seminar Member 조광현