

# JAVA DESIGN PATTERN

# 1. 디자인 패턴

## 1. 디자인 패턴이란

- 소프트웨어 설계에서 반복적으로 발생하는 문제들에 대한 해결책을 제공하는 일종의 베스트 프랙티스입니다.
- 개발자들이 더 효율적이고 재사용 가능한 코드를 작성할 수 있도록 도와줍니다.
- 디자인 패턴은 GoF(Gang of Four)의 23가지 패턴으로, 이들은 크게 생성(Creational), 구조(Structural), 행동(Behavioral) 패턴으로 분류됩니다.

### 생성패턴 (Creational Patterns)

- 객체를 생성하는 방법에 중점을 둔 디자인 패턴
- 객체 생성에 관련된 로직을 캡슐화 하여 코드의 재사용성을 향상 시킨다
- 대표적인 패턴
  - 팩토리 패턴 (Factory Method)
    - 팩토리 클래스를 통해서 객체 생성 로직을 캡슐화하는 패턴 (인스턴스를 만드는 절차를 추상화하는 패턴; 하위 클래스에게 인스턴스 작성)
  - 추상 팩토리 (Abstract Factory)
    - 여러 개의 관련된 팩토리 메서드를 함께 사용하여 부품을 조립하듯이 객체를 생성하는 패턴
  - 빌더 (Builder)
    - 객체 생성 과정을 단계별로 분리한 패턴 (선택적 매개변수를 가지지 않고 있는 객체를 생성할 때 유용)
  - 프로토타입 (Prototype)
    - 복사해서 인스턴스를 만드는 패턴 (기존 객체를 복제하여 새로운 객체를 생성)
  - 싱글톤 (Singleton Pattern)
    - 클래스의 인스턴스가 한 개만 생성되도록 보장하는 패턴

### 구조패턴 (Structural Patterns)

- 클래스나 객체를 조합하여 더 큰 구조를 만드는 디자인 패턴
- 코드의 재사용성을 높이고 객체 간의 관계를 재정의하여 유연한 프로그램 구조를 제공합니다.
- 대표적인 패턴
  - 퍼사드 패턴 (Façade Pattern)
    - 복잡한 서브시스템에 대해 하나의 통합된 인터페이스를 제공하여 시스템 구조를 단순하게 만드는 패턴
  - 데코레이터 팩토리 (Decorator Pattern)
    - 기존 객체의 기능을 동적으로 추가하거나 확장하는 방법을 제공하는 패턴 (객체를 래핑하여 새로운 기능을 추가)
  - 프록시 (Proxy Pattern)
    - 객체의 접근을 제어하는 패턴 (프록시를 통해 객체의 생성, 소멸, 네트워크 통신 등을 관리)
  - Adapter, Bridge, Composite, Flyweight

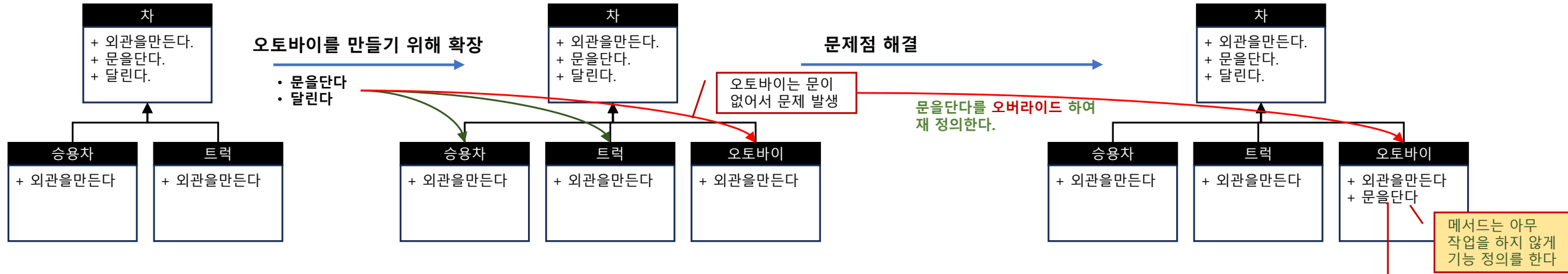
### 행동패턴 (Behavioral Patterns)

- 객체를 생성하는 방법에 중점을 둔 디자인 패턴이다
- 객체 생성에 관련된 로직을 캡슐화 하여 코드의 재사용성을 향상 시킨다
- 대표적인 패턴
  - 커맨드 패턴 (Command Pattern)
    - 인스턴스를 만드는 절차를 추상화하는 패턴
    - 하위 클래스에서 인스턴스 작성
  - 옵저버 팩토리 (Observer Pattern)
    - 객체의 상태 변경을 다른 객체들에게 전파하는 패턴 (Subject와 Observer 인터페이스를 활용하여 구현)
    - 이벤트 기반 아키텍처에 활용되며, 객체 간의 결합도를 낮추어 독립적으로 동작할 수 있다.
  - Chain of Responsibility, Interpreter, Iterator, Mediator, Memento, State, Strategy, Template Method, Visitor

# 1. 디자인 패턴

## 2. 상속보다는 구성

- 다음 클래스 다이어그램을 코드를 생각하고 어떻게 실행되는지 생각해보자

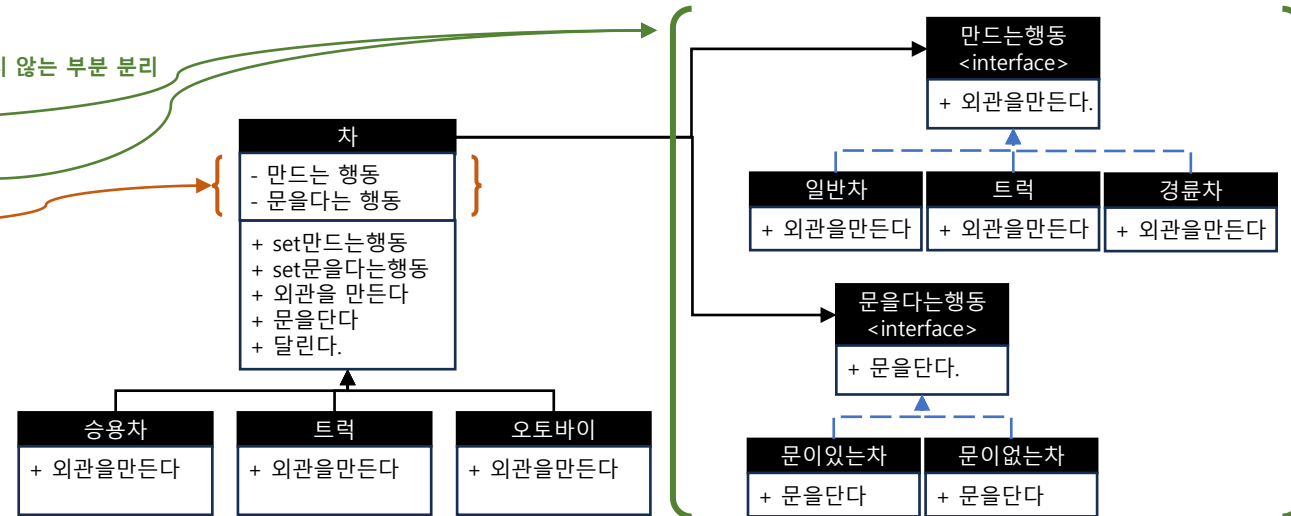


상속은 좋은 해결책이 아니다. ← 메서드에 빈 코드는 재사용 위배이다. (행위가 틀리면 모든 서브 클래스를 찾아 코드를 추가 하거나 변경해야 한다.)

디자인 원칙

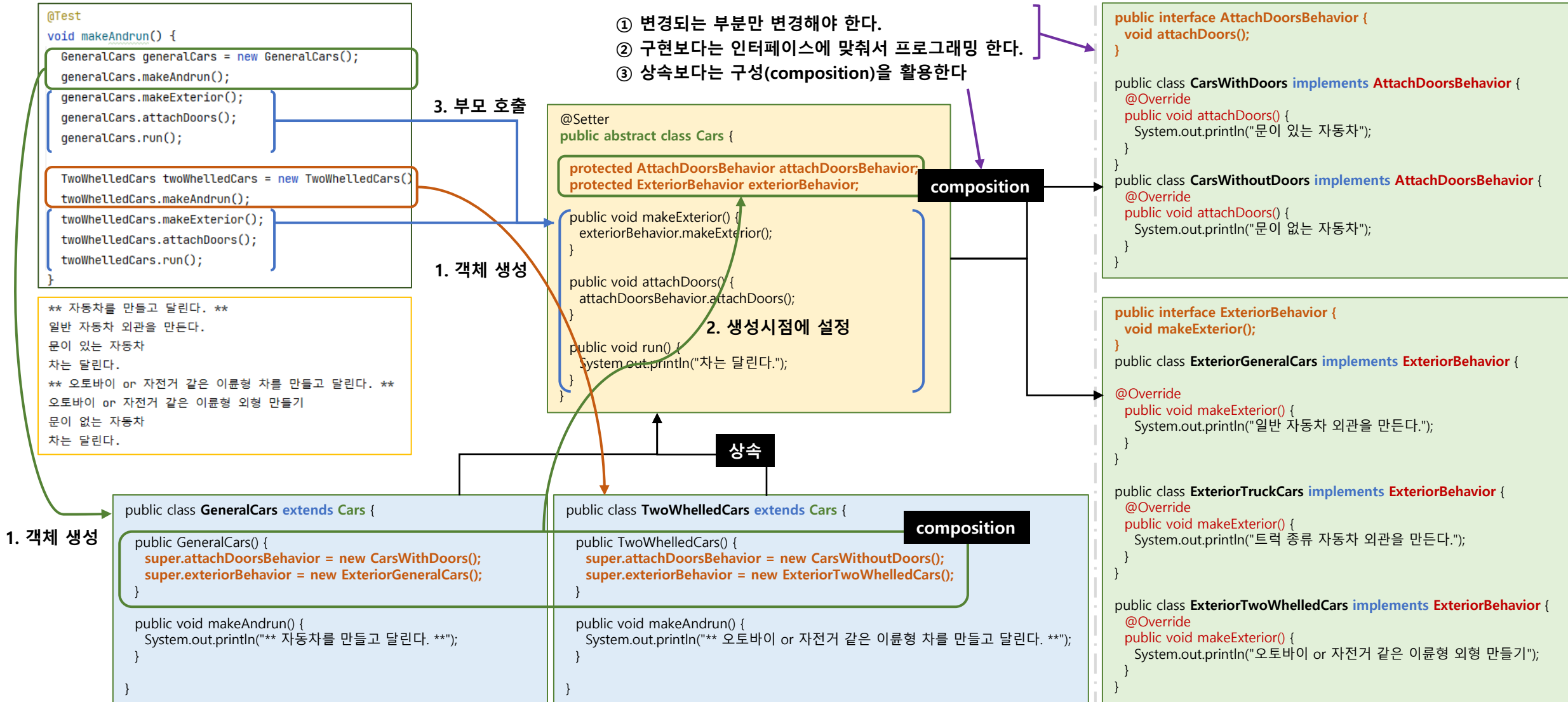
- ① 변경되는 부분만 변경해야 한다. (캡슐화)
- ② 구현보다는 인터페이스에 맞춰서 프로그래밍 한다.
- ③ 상속보다는 구성(composition)을 활용한다.

변경되는 부분과 되지 않는 부분 분리



# 1. 디자인 패턴

## 2-1. 상속보다는 구성 예제



# 2. 객체 생성

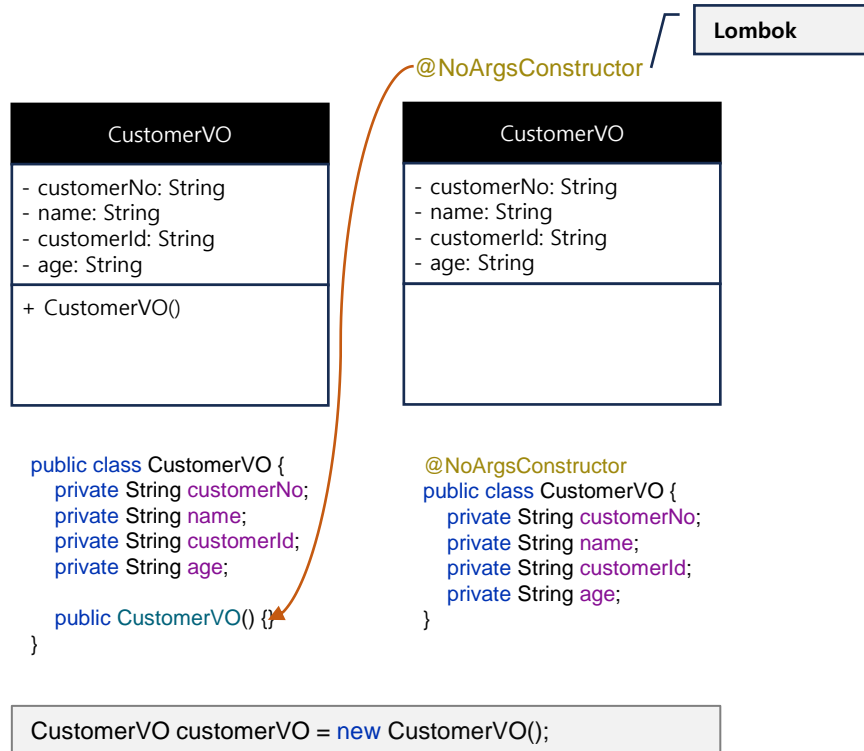
## 1-1. 기본 생성자

- 일반적으로 객체를 생성하여 인스턴스를 얻는 방법은 **public new** 연산자를 사용하며 다음과 같은 라이프 사이클을 가지고 있습니다.
  - 메모리(Heap 영역)에 데이터를 저장할 공간을 할당한다.
  - 생성자를 호출하여 객체를 생성하고 참조값(reference value)을 반환하여 변수(객체)에 저장한다.
  - 프로그램 코드에서는 new 연산자를 통해서 참조값을 저장한 객체로만 접근이 하여 사용된다.
  - 사용하지 않게 되면 JVM의 가비지 컬렉터 Garbage Collector에 의해서 메모리에서 해제된다
- 객체를 생성할 때 마다 메모리를 할당하고 초기화 하는 과정에서 객체 생성 비용이 발생하고(시간 소요) 생성된 객체는 가비지 컬렉터 Garbage Collector에 의해 메모리에서 해제되는 과정이 반복되어 성능 저하가 될 수 있으며 개발자가 메모리를 할당하고 해제하는 과정에서 실수가 발생하여 메모리 누수가 발행할 수 있다. 다음은 new 연산자를 사용해서 객체를 생성하는 단점으로 다음과 같은 방법이 있습니다. (new 연산자를 사용하여 객체를 생성하는 것이 나쁜 방법은 아니다.)
  - 메모리 누수 : 객체를 생성하고 더 이상 사용하지 않을 때는 null로 초기화 하거나 객체 참조 변수를 null로 초기화 하거나 객체 참조 변수를 지역 변수로 선언 사용하여 메모리 누수를 방지한다. [ 참조 : 1.4 메모리 누수 ]
  - 성능 저하 : 객체 풀링(object pooling)기법을 사용하여 메모리에 미리 객체를 할당 하여 풀(pool)에 넣어 두고 필요한 점에 꺼내서 사용하고 반환 하는 방식으로 메모리를 희생 하여 성능 저하를 막는 방법으로 DB 커넥션 풀링이 객체 풀링 기법의 일종이다..
  - 객체 생성 비용 : 객체를 미리 생성 하고 재사용하는 싱글톤 패턴(singleton pattern)등 디자인 패턴을 사용하여 객체 생성 비용을 감소한다.
- 다음과 같은 방법으로 객체를 생성하는 할 수 있습니다.
  - 기본 생성자
  - 점층적 생산자 패턴(Telescoping Constructor Pattern)
  - 주/부 생성자
  - 정적 팩토리 메서드 (static factory method)
  - 자바빈즈 패턴 (JavaBeans pattern)
  - GoF(Gang of Four)의 디자인 패턴(Design Pattern) 중 생성 패턴 Creational Patterns

# 2. 객체 생성

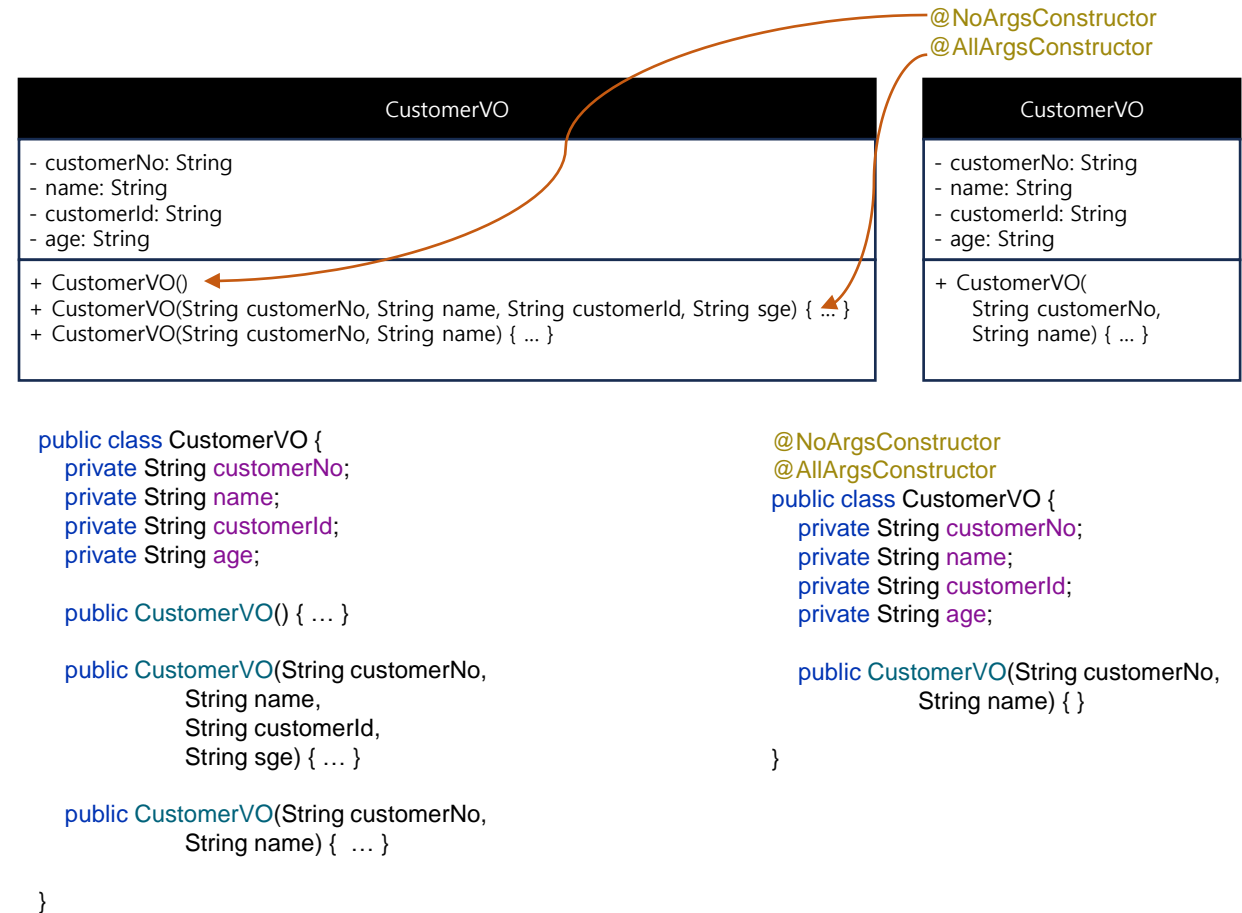
## 1-1. 기본 생성자

- 가장 일반적인 방법으로 클래스명과 동일한 이름의 메서드로 파라미터가 없는 메서드를 기본 생성자라 합니다.



- 문제점** : 멤버변수의 초기값이 참조형 변수이면 널null로 되어 NullPointerException을 유발할 수 있는 잠재적 오류

- 멤버변수의 초기값 설정을 위해 기본 생성자를 오버로딩(Overloading)한 메서드를 만들수 있습니다.



## 2. 객체 생성

### 1-2. 매개변수 2개 생성자 - 점층적 생성자 패턴(Telescoping Constructor Pattern)

- 통한 null이 발생할 수 있는 요소나 초기값 설정 후 변경 되지 않는 멤버변수의 초기값 설정을 위해 매개변수의 개수에 따른 기본생성자를 만들어서 점층적으로 생성자를 호출하게 하여 객체 생성을 하는 패턴

```
// 고객명, 고객식별번호를 받는 기본 생성자
public CustomerVO(String name, String customerId ) {
    this("", name, customerId, calAge(customerId));
}
```

Null 값 초기화를 없애기 위해 기본 생성자를 호출 하는 방식

```
// 매개변수 전체를 받는 기본 생성자
public CustomerVO(String customerNo, String name, String customerId, String age) {
    this.customerNo = customerNo;
    this.name = name;
    this.customerId = customerId;
    this.age = age;
}
```

객체가 생성 되기전에 메서드를 호출할 수 없기 때문에 Static으로 선언함

```
// 나이 자동 계산 메서드
private static String calAge(String customerId) {
    return
String.valueOf(Integer.parseInt(LocalDate.now().format(DateTimeFormatter.ofPattern("yyyy"))
    - Integer.parseInt(customerId.substring(0,4))));
}
```

```
@AllArgsConstructor
@ToString
public class CustomerVO {
    private String customerNo;
    private String name;
    private String customerId;
    private String age;
}
```

```
@AllArgsConstructor
@Setter
@Getter
public class YoungCustomerVO {
    private String customerNo;
    private String name;
    private String customerId;
    private String age;
}
```

```
@Test
void creationalClass() {
    YoungCustomerVO youngVO = new YoungCustomerVO(
        "C0001",
        "홍길동",
        "CUST_001",
        "20"
    );

    CustomerVO customerVO = new CustomerVO(
        youngVO.getCustomerNo(),
        youngVO.getName(),
        youngVO.getCustomerId(),
        youngVO.getAge()
    );

    System.out.println(customerVO.toString());
}
```

특히 문제는 없지만  
- getter 메서드가 노출 되는 단점이 있음

주/부 생성자로 극복  
할 수 있음

# 2. 객체 생성

## 2. 주/부 생성자

- 주생성자(primary constructor)와 부생성자(secondary constructor)를 만들어서 부생성자에서 주생성자를 호출하는 패턴

```
@AllArgsConstructor
@Setter
@Getter
public class YoungCustomerVO {
    private String customerNo;
    private String name;
    private String customerId;
    private String age;
}
```

```
@Test
void creationalClass() {
    YoungCustomerVO youngVO = new YoungCustomerVO(
        "C0001",
        "홍길동",
        "CUST_001",
        "20"
    );

    CustomerVO customerVO = new CustomerVO(
        youngVO.getCustomerNo(),
        youngVO.getName(),
        youngVO.getCustomerId(),
        youngVO.getAge()
    );

    System.out.println(customerVO.toString());
}
```

```
@AllArgsConstructor
@ToString
public class CustomerVO {
    private String customerNo;
    private String name;
    private String customerId;
    private String age;
}
```

```
@AllArgsConstructor
@ToString
public class CustomerVO {
    private String customerNo;
    private String name;
    private String customerId;
    private String age;

    public CustomerVO(YoungCustomerVO youngCustomerVO) {
        this(youngCustomerVO.getCustomerNo(),
            youngCustomerVO.getName(),
            youngCustomerVO.getCustomerId(),
            youngCustomerVO.getAge());
    }
}
```

```
@Test
void creationalClass() {
    YoungCustomerVO youngVO = new YoungCustomerVO(
        "C0001",
        "홍길동",
        "CUST_001",
        "20"
    );

    CustomerVO customerVO = new CustomerVO(youngVO);

    System.out.println(customerVO.toString());
}
```

단일책임 원칙 준수

부 생성자의 파라미터로 객체를 받아서  
부 생성자에서 this를 사용해서 주 생성  
자를 호출 - **getter 메서드를 감춘다.**



## 2. 객체 생성

### 3. 정적 팩토리 메서드 (static factory method)

- 객체 생성하는 역할을 정적 메서드로 분리하여 new 연산자가 아닌 의미 있는 이름 부여하여 목적에 맞게 객체를 생성합니다.
- 객체를 상속받았다면 하위 객체를 반환할 수 있으며 객체 생성을 캡슐화 할 수 있습니다.

정적 팩토리 메서드는 단순히 new 연산자를 사용하는 것 보다 객체 지형 프로그램을 통해서 가독성이 좋은 코드를 만들고 책임 생성 책임을 분리할 수 있는 정적 팩토리 클래스로 확장할 수 있으며 Entity와 DTO 변환, DTO와 DTO변환 시 내부 구현을 감추어 캡슐화를 할 수 있습니다

```
@AllArgsConstructor
@ToString
public class CustomerVO {
    private String customerNo;
    private String name;
    private String customerId;
    private String age;

    // 부생성자
    public CustomerVO(YoungCustomerVO youngCustomerVO) {
        this(youngCustomerVO.getCustomerNo(),
            youngCustomerVO.getName(),
            youngCustomerVO.getCustomerId(),
            youngCustomerVO.getAge());
    }

    // 정적 팩토리 메서드 (static factory method)
    public static CustomerVO newYoungCustomerVO(YoungCustomerVO youngCustomerVO) {
        return new CustomerVO(youngCustomerVO.getCustomerNo(),
            youngCustomerVO.getName(),
            youngCustomerVO.getCustomerId(),
            youngCustomerVO.getAge());
    }
}
```

```
@Test
void creationalClass() {
    YoungCustomerVO youngVO = new YoungCustomerVO(
        "C0001",
        "홍길동",
        "CUST_001",
        "20"
    );

    CustomerVO customerVO = CustomerVO.newYoungCustomerVO(youngVO);

    System.out.println(customerVO.toString());
}
```

# 2. 객체 생성

## 4. 자바빈즈 패턴

- 매개변수 없는 객체를 생성자를 만들고 세터setter메서드를 호출하여 멤버 변수에 값을 설정하는 방법
- 객체 불변성을 유지하기 위해서는 수동으로 freezing을 해야 한다.

```
public class CustomerVO {
    private String customerNo;
    private String name;
    private String customerId;
    private String age;

    // 기본 생성자
    public CustomerVO(){}

    public String getCustomerNo() {
        return customerNo;
    }

    public void setCustomerNo(String customerNo) {
        this.customerNo = customerNo;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
    ....
}
```

```
@Test
@DisplayName("JavaBeans 패턴 - 생성 테스트")
void createCustomer() {
    CustomerVO customerVO = new CustomerVO();
    customerVO.setCustomerNo("C0001");
    customerVO.setName("홍길동");
    customerVO.setCustomerId("20020523140000");
    customerVO.setAge("21");

    System.out.println(customerVO.toString());

    // 객체 불변성 깨짐
    customerVO.setName("김나리");

    System.out.println(customerVO.toString());
}
```

객체 불변성을  
유지하기 위해  
서는 수동으로  
freezing

```
@Test
@DisplayName("JavaBeans 패턴 - 생성 테스트- 객체불변성 유지")
void createCustomer_freezing() {
    CustomerVO customerVO = new CustomerVO();
    customerVO.setCustomerNo("C0001");
    customerVO.setName("홍길동");
    customerVO.setCustomerId("20020523140000");
    customerVO.setAge("21");

    customerVO.freeze();

    System.out.println(customerVO.toString());

    // 객체 불변성 깨짐
    customerVO.setName("김나리");

    System.out.println(customerVO.toString());
}
```

```
public class CustomerVO {
    private String customerNo;
    private String name;

    // 객체 불변성 유지를 위해 freezing 체크 변수
    private boolean freeze = false;

    // 기본 생성자
    public CustomerVO(){}

    public String getCustomerNo() { return customerNo; }

    public void setCustomerNo(String customerNo) {
        this.customerNo = customerNo;
    }

    public String getName() { return name; }

    public void setName(String name) {
        if (isFreeze()) throw new AssertionError("[객체 불변성] 객체 값을 변경 할 수 없습니다..");
        this.name = name;
    }

    // 객체 불변성 유지를 위한 freezing 코드
    public boolean isFreeze() { return freeze; }

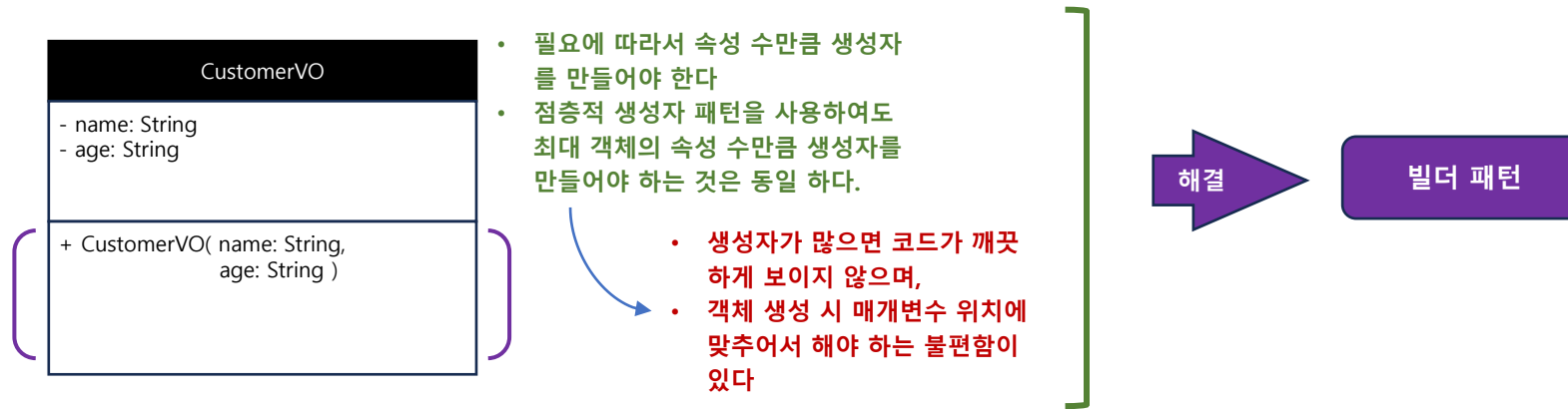
    public void freeze() { this.freeze = true; }
}
```

java.lang.AssertionError: [객체 불변성] 객체의 값을 변경 할 수 없습니다.

# 3. 생성 패턴 (Creational Patterns)

## 3. 빌더(Builder)

- 복잡한 객체를 단계별로 생성할 수 있는 생성 디자인 패턴으로 객체 구성 또는 생성코드를 자체 클래스에서 추출하여 빌더라는 별도의 객체에서 생성하도록 하는 것입니다.



```
@AllArgsConstructor
@ToString
public class CustomerVO {
    private String name;
    private String age;
}
```

점층적 생성자

```
public class CustomerBuilder {
    private String name;
    private String age;

    public CustomerBuilder name(String name) {
        this.name = name;
        return this;
    }

    public CustomerBuilder age(String age) {
        this.age = age;
        return this;
    }

    public CustomerVO build() {
        return new CustomerVO (
            this.name,
            this.age );
    }
}
```

```
void createBuilder() {
    CustomerVO customerVO = new
    CustomerBuilder()
        .age("29")
        .name("홍길동")
        .build();
    System.out.println(customerVO.toString());
}
```

→ CustomerVO(name=홍길동, age=29)

### 장점

- 유연성과 확장성**
  - 객체 생성 과정을 단순화하면서도 유연성 제공
  - 다양한 속성을 가진 객체 생성, 속성 추가 시 빌더 클래스만 수정
- 가독성**
  - 체인 형태로 메서드 호출로 코드가 간결하고 가독성이 좋다.
- 객체 생성 순서 제어**
  - 객체 생성 단계에서 명시적으로 필수 속성을 먼저 설정하고 선택적 속성을 나중에 설정할 수 있다.

### 단점

- 복잡성**
  - 빌더 클래스를 정의 하고 관리해야 한다.
- 오버헤드**
  - 객체 생성 과정을 분리하므로 약간의 오버헤드가 발생할 수 있다.

# 3. 생성 패턴 (Creational Patterns)

## 3-1. 정적 빌더

- 빌더 객체를 다른 파일로 생성하지 않고 클래스 내부에 클래스를 만드는 이너 클래스를 만들어서 하는 방법.

```
@ToString
public class CustomerSaticVO {
    private String name;
    private String age;

    public CustomerSaticVO(Builder builder) {
        this.name = builder.name;
        this.age = builder.age;
    }
}

public static class Builder {
    private String name;
    private String age;

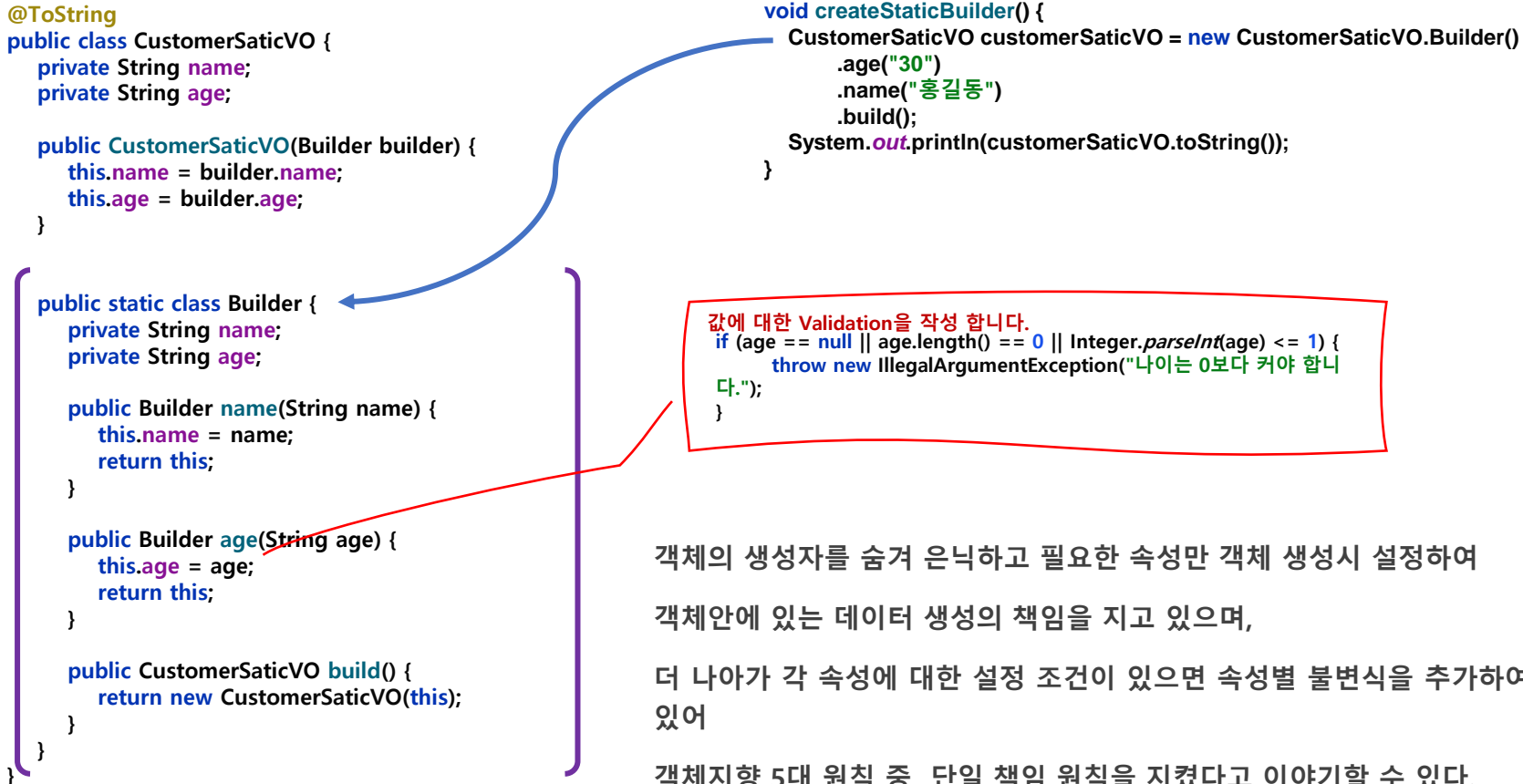
    public Builder name(String name) {
        this.name = name;
        return this;
    }

    public Builder age(String age) {
        this.age = age;
        return this;
    }

    public CustomerSaticVO build() {
        return new CustomerSaticVO(this);
    }
}

void createStaticBuilder() {
    CustomerSaticVO customerSaticVO = new CustomerSaticVO.Builder()
        .age("30")
        .name("홍길동")
        .build();
    System.out.println(customerSaticVO.toString());
}

값에 대한 Validation을 작성 합니다.
if (age == null || age.length() == 0 || Integer.parseInt(age) <= 1) {
    throw new IllegalArgumentException("나이는 0보다 커야 합니다.");
}
```



객체의 생성자를 숨겨 은닉하고 필요한 속성만 객체 생성시 설정하여

객체안에 있는 데이터 생성의 책임을 지고 있으며,

더 나아가 각 속성에 대한 설정 조건이 있으면 속성별 불변식을 추가하여 유효성 검사를 할 수 있어

객체지향 5대 원칙 중 단일 책임 원칙을 지켰다고 이야기할 수 있다.

# 3. 생성 패턴 (Creational Patterns)

## 3-2. Lombok Builder

- 롬복(lombok)의 @Builder를 하면 빌더 객체의 생성은 롬복에서 자동으로 생성해 주는 것을 사용하면 소스가 간단하며 개발 생산성을 높일 수 있습니다.

```
@Builder
public class LombokBuilderVO {
    private String name;
    private String age;
}
```

```
@Test
void createLombokBuilderVO() {
    LombokBuilderVO lombokBuilderVO
    = LombokBuilderVO.builder()
    .age("30")
    .name("홍길동")
    .build();
}
```

```
public class LombokBuilderVO {
    private String name;
    private String age;
    LombokBuilderVO(final String name, final String age) {
        this.name = name;
        this.age = age;
    }

    public static LombokBuilderVOBuilder builder() {
        return new LombokBuilderVOBuilder();
    }

    public static class LombokBuilderVOBuilder {
        private String name;
        private String age;

        LombokBuilderVOBuilder() {
        }

        public LombokBuilderVOBuilder name(final String name) {
            this.name = name;
            return this;
        }

        public LombokBuilderVOBuilder age(final String age) {
            this.age = age;
            return this;
        }

        public LombokBuilderVO build() {
            return new LombokBuilderVO(this.name, this.age);
        }

        public String toString() {
            return "LombokBuilderVO.LombokBuilderVOBuilder(name="
            + this.name + ", age=" + this.age + ")";
        }
    }
}
```

```
public class LombokCreateBuilderVO {
    private String name;
    private String age;

    @Builder
    public LombokCreateBuilderVO(
        String name,
        String age) {
        this.name = name;
        this.age = age;
    }
}
```

```
@Test
void createLombokCreateBuilderVO() {
    LombokCreateBuilderVO
    LombokCreateBuilderVO =
    LombokCreateBuilderVO.builder()
    .age("30")
    .name("홍길동")
    .build();
}
```

```
public class LombokCreateBuilderVO {
    private String name;
    private String age;

    public LombokCreateBuilderVO(String name, String age) {
        this.name = name;
        this.age = age;
    }

    public static LombokCreateBuilderVOBuilder builder() {
        return new LombokCreateBuilderVOBuilder();
    }

    public static class LombokCreateBuilderVOBuilder {
        private String name;
        private String age;

        LombokCreateBuilderVOBuilder() {
        }

        public LombokCreateBuilderVOBuilder name(final String
        name) {
            this.name = name;
            return this;
        }

        public LombokCreateBuilderVOBuilder age(final String age) {
            this.age = age;
            return this;
        }

        public LombokCreateBuilderVO build() {
            return new LombokCreateBuilderVO(this.name, this.age);
        }

        public String toString() {
            return
            "LombokCreateBuilderVO.LombokCreateBuilderVOBuilder(name="
            + this.name + ", age=" + this.age + ")";
        }
    }
}
```

# 3. 생성 패턴 (Creational Patterns)

## 3-3. GoF – 빌더 패턴

- 복잡한 객체를 생성하는 방법과 표현하는 방법을 정의하는 클래스를 분리하여, 서로 다른 표현이라도 이를 생성할 수 있는 동일한 절차를 제공하는 패턴
- 즉, 객체를 생성하는 패턴으로 객체 생성을 담당하는 빌더와 조립 과정을 담당하는 디렉터(Director)로 나뉘어져 있습니다.

예) Wave, Tving에 가입 정보와 상품 정보를 넘겨주어야 하는데 웨이브는 Json 포맷, Tving은 xml로 전달해야 한다.

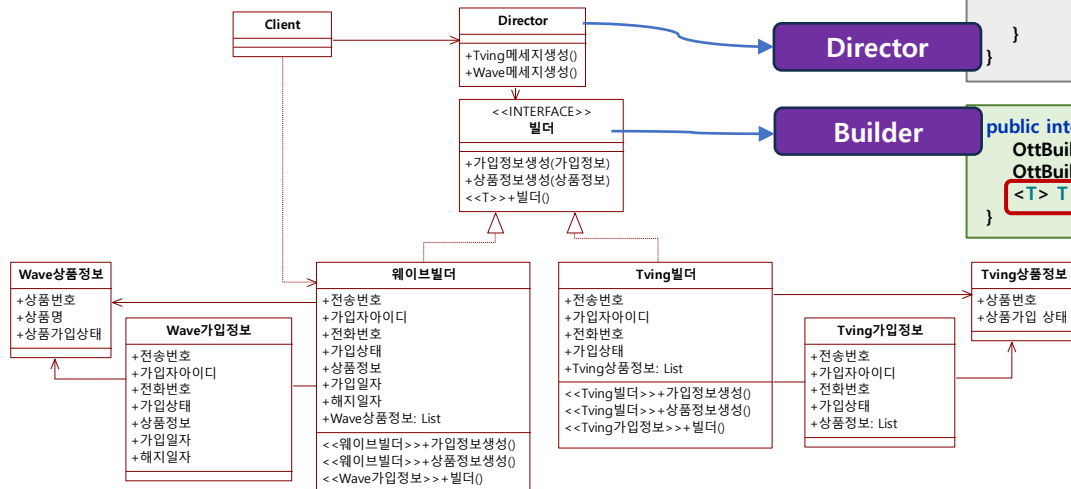
- 외부로 노출된 서비스로 TvingBuilder, WaveBuilder를 이용해서 연동 업체별 연동 객체를 생성
- TvingBuilder 객체를 new로 선언하고 build() 메소드를 별도로 호출한 것은 빌더 패턴을 이용해서 지연 생성을 할 수 있기 때문이다.

```
public class Director {  
  
    public String makeTvingVO(SubscriberVO subscriberVO,  
        List<ProductVO> productVOs) {  
        TvingBuilder tvingBuilder = new TvingBuilder();  
  
        tvingBuilder.subscriber(subscriberVO)  
            .product(productVOs);  
  
        TvingVO tvingVO = tvingBuilder.build();  
        return tvingVO.toString();  
    }  
  
    public String makeWaveVO(SubscriberVO subscriberVO,  
        List<ProductVO> productVOs) {  
        WaveBuilder waveBuilder = new WaveBuilder();  
  
        ....  
  
        WaveVO waveVO = waveBuilder.build();  
        return new Gson().toJson(waveVO);  
    }  
}
```

```
public class TvingBuilder implements OttBuilder {  
  
    private String transectionId;  
    private String subscriberId;  
    private String phoneNumber;  
    private String status;  
    List<TvingProdVO> waveProdVOs;  
  
    public TvingBuilder() {  
        this.transectionId = "TVING_" + UUID.randomUUID().toString();  
    }  
  
    @Override  
    public OttBuilder subscriber(SubscriberVO subscriberVO) {  
        this.subscriberId = subscriberVO.getSubscriberId();  
        this.phoneNumber = subscriberVO.getPhoneNumber();  
        this.status = subscriberVO.getStatus();  
        return this;  
    }  
  
    @Override  
    public OttBuilder product(List<ProductVO> productVOs) {  
        this.waveProdVOs = new LinkedList<>();  
        for( ProductVO productVO : productVOs) {  
            this.waveProdVOs.add(new TvingProdVO(productVO.getProductId(),  
                productVO.getProductStatus()));  
        }  
  
        return this;  
    }  
  
    @Override  
    public TvingVO build () {  
        return new TvingVO(this.transectionId,  
            this.subscriberId,  
            this.phoneNumber,  
            this.status,  
            this.waveProdVOs);  
    }  
}
```

```
public interface OttBuilder {  
    OttBuilder subscriber(SubscriberVO subscriberVO);  
    OttBuilder product(List<ProductVO> productVOs);  
    <T> T build();  
}
```

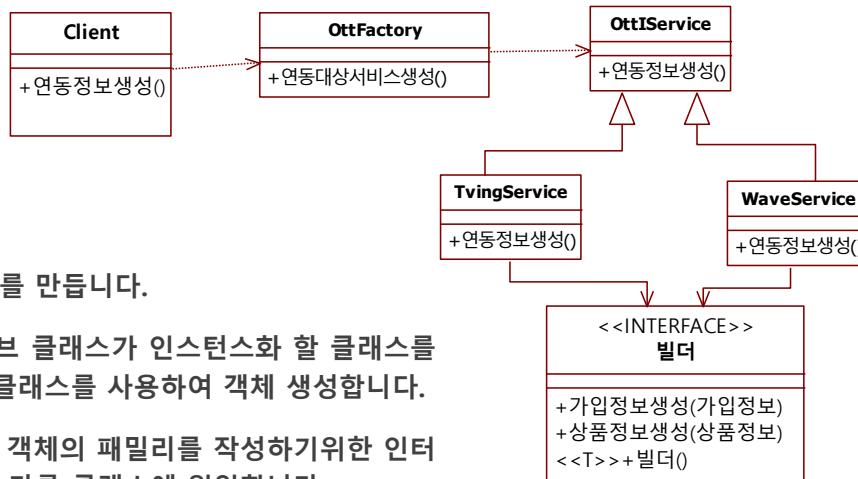
Tving, Wave 등 생성 객체를 다르게 생성하여 받아야 해서 T Type으로 리턴형으로 선언한다



# 3. 생성 패턴 (Creational Patterns)

## 1. 팩토리 메소드(Factory Method)

- 객체 생성을 위한 인터페이스를 정의하고 서브클래스가 인스턴스화 할 클래스를 결정 합니다.
- 빌더 패턴의 Director 클래스처럼 인스턴스화 로직을 클라이언트에 노출하지 않기 위해 별도의 팩토리 클래스를 만들어 클라이언트는 팩토리 클래스를 통해 객체 생성 합니다.
- 생성할 객체의 공통적인 기능을 이름을 지어 추상화하고 서브 클래스에서 인스턴스화를 결정하게 하고 기능하고 다양성을 통해 공통적인 기능을 구현하는 것이 팩토리 메서드 패턴 입니다.
- 변경 사항이 발생하면 해당 클래스만 최소로 변경하거나 확장하여 기능 개발을 할 수 있게 됩니다. 디자인 원칙 중 책임 원칙과 개방-폐쇄 원칙을 준수합니다.
- 팩토리 메서드는 객체 생성을 위임하는 클래스를 의미합니다.



- ① **팩토리** : 인스턴스화 로직을 클라이언트에 노출시키지 않고 객체를 만듭니다.
- ② **팩토리 메소드** : 객체 생성을 위한 인터페이스를 정의하지만 서브 클래스가 인스턴스화 할 클래스를 결정하도록 합니다. 즉 상속을 사용하고 파생 클래스 또는 서브 클래스를 사용하여 객체 생성합니다.
- ③ **추상 팩토리** : 구체적인 클래스를 지정하지 않고 관련 또는 종속 객체의 패밀리를 작성하기 위한 인터페이스를 제공합니다. 즉 컴포지션을 사용하여 객체 생성 책임을 다른 클래스에 위임합니다.

### 장점

- **코드의 결합도 낮춤**
  - Client는 인터페이스와 동작하므로 다양한 구현 클래스에 정의한 코드로 실행한다.
- **확장성 높음**
  - 객체의 자료형이 하위 클래스에 의해 결정되므로, 새로운 클래스를 추가하기 쉽고 OCP 원칙을 준수한다.
- **동일한 형태의 프로그램**
  - 팩토리 메서드는 객체 생성을 캡슐화 하므로, 객체 생성 방식이 달라도 동일한 메서드를 호출할 수 있습니다.
- **확장성 있는 프로젝트 구성**
  - 팩토리 메서드는 다른 패턴과 함께 사용할 수 있으므로, 유연하고 확장성 있는 프로젝트를 구성할 수 있습니다.

### 단점

- **많은 클래스 생성 가능성**
  - 객체가 늘어날 때마다 하위 클래스를 재정의해야 하므로, 불필요하게 많은 클래스를 생성할 수 있다.
- **오버헤드**
  - 팩토리 메서드는 객체 생성을 위임하는 클래스를 의미합니다.

# 3. 생성 패턴 (Creational Patterns)

## 1-1. 팩토리 메소드(Factory Method) 예제

### 빌더 패턴의 Client 소스

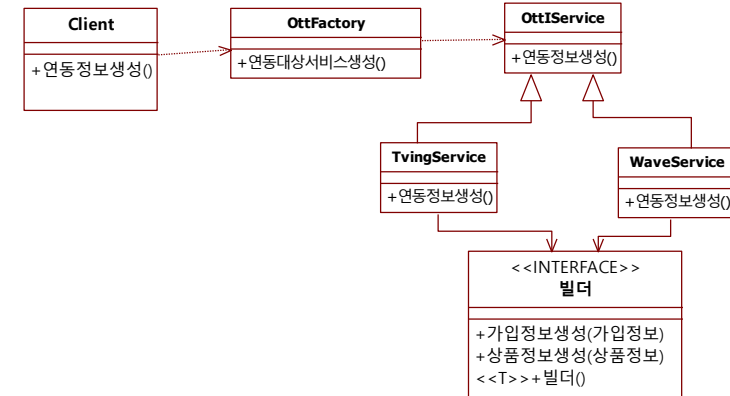
```
public class OttService {
    private final Director director;
    private final List<String> OTT_PROD_TYPE_LIST = List.of("WAVE", "TVING");
    public OttService(Director director) {
        this.director = director;
    }

    public String getTransData(SubscriberVO subscriberVO,
                               List<ProductVO> productVOs) {
        String transData = "";
        String prodType = productVOs.stream()
            .filter(productVO -> OTT_PROD_TYPE_LIST.stream().anyMatch(type -> type.equals(productVO.getProductType())))
            .map(ProductVO::getProductType)
            .findFirst()
            .orElse("");
        if ("WAVE".equals(prodType)) {
            transData = ottWaveMakeMsg(subscriberVO, productVOs);
        } else if ("TVING".equals(prodType)) {
            transData = ottTvingMakeMsg(subscriberVO, productVOs);
        }
        return transData;
    }

    public String ottTvingMakeMsg(SubscriberVO subscriberVO, List<ProductVO> productVOs) {
        return director.makeTvingVO(subscriberVO, productVOs);
    }

    public String ottWaveMakeMsg(SubscriberVO subscriberVO, List<ProductVO> productVOs) {
        return director.makeWaveVO(subscriberVO, productVOs);
    }
}
```

Factory 대상

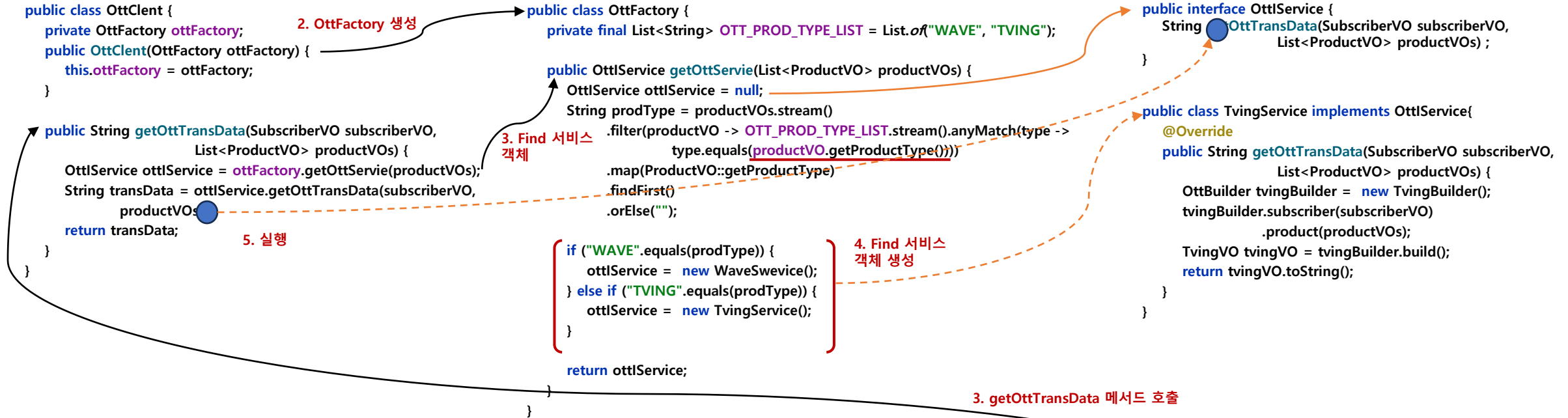


- ① **Client** : OttFactory로 부터 연동 대상이 되는 인스턴스를 받고 연동 정보를 생성합니다.
- ② **OttFactory** : 연동 대상이 되는 서비스를 생성합니다..
- ③ **OttService** : 추상 클래스로 연동 정보생성을 위한 메소드를 정의합니다
- ④ **XXXXService** : OttService의 구현체로 빌더 패턴의 Director클래스의 객체 생성을 각각의 클래스에서 담당하게 합니다.



# 3. 생성 패턴 (Creational Patterns)

## 1-1. 팩토리 메소드(Factory Method) 예제



```
class OttClientTest {
    private OttClient ottClient;
    private SubscriberVO subscriberVO = new SubscriberVO();
    private List<ProductVO> productVOs = new LinkedList<>();
```

```
@BeforeEach
void setUp() {
    1. OttClient 생성
    ottClient = new OttClient(new OttFactory());
    subscriberVO.setSubscriberId("SUB_00001");
    subscriberVO.setPhoneNumber("010092349034");
    subscriberVO.setStatus("A");
    subscriberVO.setStartDate("20230409123020");
    subscriberVO.setEndDate("99991231235959");

    productVOs = List.of(
        new ProductVO("PROD_00001", "영상", "A", "UFLIX"),
        new ProductVO("PROD_00002", "영상_002", "A", "WAVE")
    );
}
```

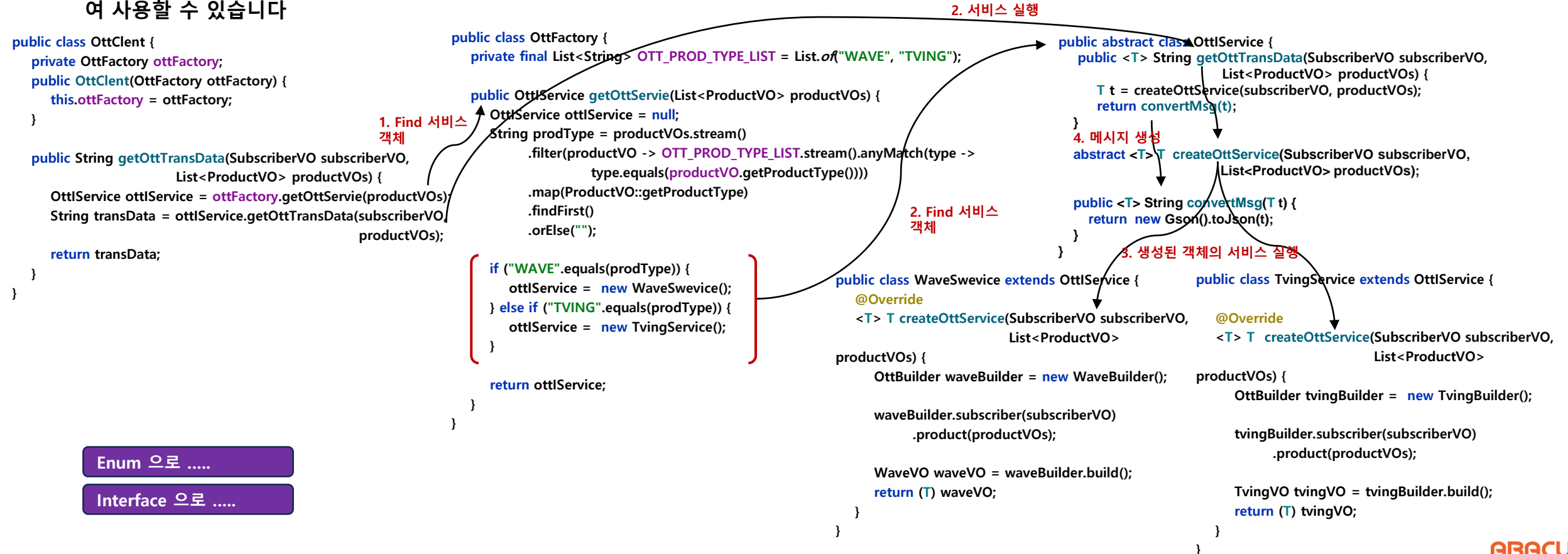
```
@Test
void getOttTransData() {
    String transData = ottClient.getOttTransData(subscriberVO,
        productVOs);
    System.out.println(transData);
}
```

```
{
  "transactionId": "WAVE_cd8d9d62-2d2d-416c-92eb-7777646236d0",
  "subscriberId": "SUB_00001",
  "phoneNumber": "010092349034",
  "status": "A",
  "startDate": "20230409123020",
  "endDate": "99991231235959",
  "waveProdVOList": [
    {
      "productId": "PROD_00001",
      "productName": "영상",
      "productStatus": "A"
    },
    {
      "productId": "PROD_00002",
      "productName": "영상_002",
      "productStatus": "A"
    }
  ]
}
```

# 3. 생성 패턴 (Creational Patterns)

## 2. 추상 팩토리 (Abstract Factory)

- 구체적인 클래스를 지정하지 않고 관련 또는 종속 객체의 패밀리를 작성하기 위한 추상 클래스 제공
- 클라이언트는 추상 인터페이스만 사용하여 어떤 객체가 생성되는지 전혀 알 수 없습니다.
- Abstract, Interface로 추상 클래스를 만들고 공통적인 기능을 명명하여 서브 클래스에서 기능을 구현하도록 합니다.
- Abstract로 추상 클래스를 사용하는 경우는 공통적인 업무 로직이 있는 경우 사용됩니다. 즉 추상 클래스에서 기능 구현하고 서브 클래스에 있는 메서드를 호출하여 사용할 수 있습니다



# 3. 생성 패턴 (Creational Patterns)

## 2. 추상 팩토리 (Abstract Factory)

```
try {
    NucmService nucmService = nucmServiceFactory.getNucmService(getEventCd(jsonObject));
    T messageDto = getMessageObject(nucmService, nuomResultDTO.getEventDtlCd(), jsonObject);
    nuomResultDTO = nucmService.runProcess(messageDto, nuomResultDTO.getEventDtlCd());
} catch (RuntimeException ex) {
    nuomResultDTO.setMessageCode(ex.getMessage());
    nuomResultDTO.setOrderStatus("F");
}

@Service
@RequiredArgsConstructor
public class NucmServiceFactory {

    private final List<NucmService> nucmServiceList;

    public NucmService getNucmService(String eventCd) {
        return nucmServiceList.stream()
            .filter(nucmService -> nucmService.getEventCd().equalsIgnoreCase(eventCd))
            .findFirst()
            .orElseThrow(() -> new RuntimeException(String.format(" Invlaid Vehicle Name - %s", eventCd)));
    }
}

public class NuomResultService {
    public NuomResultDTO setNucmResultDTO(String orderId,
        String eventCd,
        String eventDtlCd ) {
        return NuomResultDTO.builder()
            .orderId(orderId)
            .eventCd(eventCd)
            .eventDtlCd(eventDtlCd)
            .build();
    }
}

public interface NucmService {
    String getEventCd();
    Class getClassName(String eventDtlCd);

    <T> NuomResultDTO runProcess(T meageObject, String eventDtlCd);
}
```

```
@Slf4j
@RequiredArgsConstructor
@Service
public class NucmNacService extends NucmResultService implements NucmService {

    @Override
    public String getEventCd() {
        return EventCdDtlCdConst.EVENT_NAC_CD;
    }

    @Override
    public Class getClassName(String eventDtlCd) {
        return MessageNacDTO.class;
    }

    @Override
    public <T> NuomResultDTO runProcess(T meageObject, String eventDtlCd) {

        log.info("Run NAC ..... ");
        MessageNacDTO messageNacDTO = (MessageNacDTO) meageObject;
        log.info("messageNacDTO :: " + messageNacDTO.toString());

        NuomResultDTO nuomResultDTO = super.setNucmResultDTO(messageNacDTO.getOrderId(),
            messageNacDTO.getEventCd(),
            messageNacDTO.getEventDtlCd());

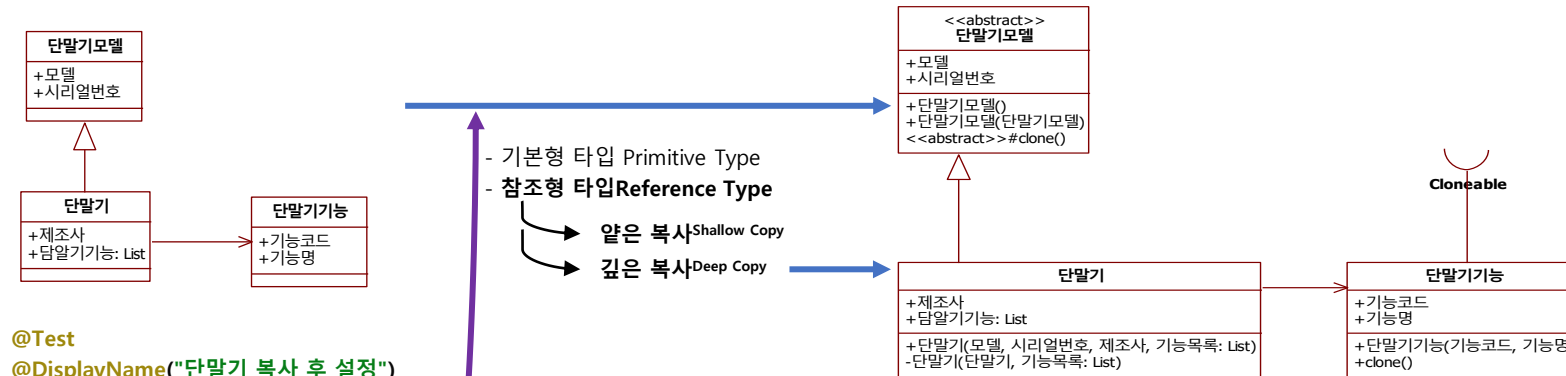
        nuomResultDTO.setOrderStatus("C");
        nuomResultDTO.setMessageCode("200");
        return nuomResultDTO;
    }
}
```

[Spring-Boot/springpattern/src/main/java/com/hyomee/sp at master · hyomee/Spring-Boot \(github.com\)](https://github.com/hyomee/Spring-Boot)

# 3. 생성 패턴 (Creational Patterns)

## 4. 프로토타입(Prototype)

- 복잡한 계층구조를 가지고 있는 객체의 인스턴스를 새로 만들어야 할 때 사용할 수 있는 패턴으로 clone 메서드를 재 정의하여 사용하는 패턴
- getter/setter 를 이용해서 작업할 때 객체의 멤버가 적고 객체의 연관 관계가 복잡하지 않으면 쉽게 코드를 작성할 수 있지만 멤버 변수가 많고 복잡한 관계를 가지고 있는 객체이면 코드 작성에 시간이 걸리고 소스 코드가 복잡 해져 읽기 어려운 코드가 만들어지는 문제를 해결하기 위한 자바의 clone()을 사용해서 복사하여 객체를 생성하는 방법을 프로타입 패턴



```
@Test
@DisplayName("단말기 복사 후 설정")
void createDeviceVO_TEST() {
    List<FunctionVO> functionVOs = List.of(
        new FunctionVO("F001", "통화기능"),
        new FunctionVO("F002", "영상기능")
    );
    DeviceModelVO deviceModelVO = new DeviceModelVO("S23", "S12345");
    DeviceVO deviceVO = new DeviceVO(deviceModelVO, "삼성", functionVOs);
    DeviceVO deviceVO01 = deviceVO;
    deviceVO01.setModel("S45");
    deviceVO01.setSerialNo("S45_00001");
    System.out.println("DeviceVO : " + deviceVO.toString());
    System.out.println("DeviceVO_01 : " + deviceVO01.toString());
}
```

```
DeviceVO : DeviceVO(model='S45', serialNo='S45_00001', company='삼성', functionVOs=[FunctionVO(functionId='F001', functionName='통화기능'), FunctionVO(functionId='F002', functionName='영상기능')])

DeviceVO_01 : DeviceVO(model='S45', serialNo='S45_00001', company='삼성', functionVOs=[FunctionVO(functionId='F001', functionName='통화기능'), FunctionVO(functionId='F002', functionName='영상기능')])
```

기존에 구현되어 있는 클래스를 통해 새로운 클래스를 만드는 것

### 장점

- 서브클래싱 감소
  - 프로토타입 패턴은 새로운 객체를 생성할 때 서브클래싱을 피할 수 있습니다..
- 클라이언트의 복잡성 감소
  - 클라이언트가 새로운 인스턴스를 생성하는 복잡한 로직을 숨길 수 있습니다.
- 런타임에 객체 추가 및 제거 가능
  - 프로토타입 패턴은 런타임에 객체를 추가하거나 제거할 수 있습니다.

### 단점

- 복잡한 객체 복제 어려움
  - 복제 대상 객체가 복잡한 내부 구조를 가지고 있다면 완벽한 복사본을 만드는 것이 어려울 수 있습니다.
- 내부 의존성을 가진 객체 복제 어려움
  - 객체가 다른 객체에 의존하고 있을 때 복제하기 어려울 수 있습니다.
- 외부 리소스 참조를 가진 객체 복제 어려움
  - 외부 리소스를 참조하는 객체를 복제하는 것도 어려울 수 있습니다.

# 3. 생성 패턴 (Creational Patterns)

## 4-1. 프로토타입(Prototype) 예제

```
public abstract class DeviceModelVO implements Cloneable {
    private String model;
    private String serialNo;

    public DeviceModelVO() {}

    protected DeviceModelVO(DeviceModelVO deviceModelVO) {
        if (deviceModelVO != null) {
            this.model = deviceModelVO.getModel();
            this.serialNo = deviceModelVO.getSerialNo();
        }
    }

    @Override
    protected abstract DeviceModelVO clone();

    protected void setModel(String model) {
        this.model = model;
    }

    protected void setSerialNo(String serialNo) {
        this.serialNo = serialNo;
    } // Setter 생략
}
```

```
public class DeviceVO extends DeviceModelVO {
    private String company;
    private List<FunctionVO> functionVOs;

    public DeviceVO( String model,
                    String serialNo,
                    String company,
                    List<FunctionVO> functionVO) {
        super();
        super.setModel(model);
        super.setSerialNo(serialNo);
        this.company = company;
        this.functionVOs = functionVO;
    }

    private DeviceVO( DeviceVO deviceVO,
                    List<FunctionVO> functionVOs) {
        super(deviceVO);
        if (deviceVO != null) {
            this.company = deviceVO.company;
            this.functionVOs = functionVOs;
        }
    }

    @Override
    public DeviceModelVO clone() {
        List<FunctionVO> functionVOs = new LinkedList<>();
        for (FunctionVO functionVO : this.functionVOs) {
            functionVOs.add(functionVO.clone());
        }
        return new DeviceVO(this, functionVOs);
    } // Setter/Getter 생략
}
```

```
@Test
@DisplayName("프로토타입 패턴으로 단말기 객체 생성")
void createDeviceVO_TEST_01() {
    List<FunctionVO> functionVOs = List.of(
        new FunctionVO("F001", "통화기능"),
        new FunctionVO("F002", "영상기능")
    );

    DeviceVO deviceVO =
        new DeviceVO("S32", "S32_00001", "삼성", functionVOs);

    DeviceVO deviceVO_01 = (DeviceVO) deviceVO.clone();

    deviceVO_01.setModel("S45");
    deviceVO_01.setSerialNo("S45_00001");
    deviceVO_01.getFunctionVOs()
        .add(new FunctionVO("F003", "MMS"));

    DeviceVO deviceVO_02 = (DeviceVO) deviceVO.clone();

    deviceVO_02.setModel("A01");
    deviceVO_02.setSerialNo("A01_00001");
    deviceVO_02.setCompany("애플");

    System.out.println("DeviceVO : " + deviceVO.toString());
    System.out.println("deviceVO_01 : " + deviceVO_01.toString());
    System.out.println("deviceVO_02 : " + deviceVO_02.toString());
}
```

# 3. 생성 패턴 (Creational Patterns)

## 5. 싱글톤(Singleton)

- 전체 시스템에 대해 하나의 인스턴스 만 허용하는 클래스의 인스턴스를 제한하는 소프트웨어 디자인 패턴.
- 클래스도 전역 변수처럼 하나의 인스턴스를 만들어서 사용하게 하는 패턴이 싱글톤 패턴으로 데이터베이스 연결 모듈, 디스크 I/O 모듈, 스레드 풀 등에 적용할 수 있으며 국내에서 많이 사용하는 스프링에서 사용되고 있다.
- 주의사항 :
  - 멀티 스레드 환경에서 싱글톤으로 구현된 객체의 멤버 변수의 값을 변경하면 값 변경에 따른 예상하지 못한 오류를 유발할 수 있다
  - 또한 클래스로더를 어떻게 구성 하느냐에 따라 자바환경에서 하나의 인스턴스를 보장할 수 없으며 mock 오브젝트를 만들어서 테스트하기 힘들어서 많은 개발자가 안티 패턴으로 간주한다

Singleton
-Instance: Singleton
-Singleton() +getInstance(): Singleton

객체 생성 전에 해당 객체에 접근 해야함

```
public class Singleton {  
    private static Singleton instance;  
    private String value;  
  
    private Singleton(String value) {  
        System.out.println(value  
            + " 값으로 Singleton 객체가 생성 되었습니다");  
        this.value = value;  
    }  
  
    public static Singleton getInstance(String value) {  
        if (instance == null) {  
            instance = new Singleton(value);  
        }  
        return instance;  
    }  
  
    public String getValue() {  
        return this.value;  
    }  
}
```

@Test  
void getInstance() {  
 Singleton singleton01 = Singleton.getInstance("Init\_01");  
 Singleton singleton02 = Singleton.getInstance("Init\_02");  
 System.out.println("singleton01 :: " + singleton01.getValue());  
 System.out.println("singleton02 :: " + singleton02.getValue());  
}

Init\_01 값으로 Singleton 객체가 생성 되었습니다  
singleton01 :: Init\_01  
singleton02 :: Init\_01

Init\_02 로 생성 했지만  
Init\_01 로 만들어진 객체를 사용  
하므로 "Init\_01"로 표시됨

### 장점

- 리소스 절약
  - 인스턴스가 하나뿐이므로 메모리와 시스템 자원을 효율적으로 사용할 수 있다.
- 데이터 공유
  - 인스턴스가 하나뿐이므로 데이터를 쉽게 공유할 수 있다.
- 전역객체
  - 인스턴스가 어디서든 접근 가능하므로 전역 객체로 사용할 수 있다.
- 상태 유지
  - 인스턴스가 하나뿐이므로 객체의 상태를 유지할 수 있다.
- 확장성
  - 인스턴스가 하나뿐이므로 클래스를 확장하기 쉽다

### 단점

- 테스트 어려움
  - 인스턴스가 전역 상태를 유지하므로 테스트를 어렵게 만든다..
- 멀티 인스턴스 문제
  - 인스턴스가 전역 상태를 유지하므로 멀티스레드 환경에서 동기화 문제를 신경써야 한다.
- 객체 의존성 높음
  - 인스턴스가 전역 상태를 유지하므로 다른 객체들이 이를 의존하는 경우 객체 간의 결합도가 높아질 수 있다..



# 3. 생성 패턴 (Creational Patterns)

## 5-1. 멀티 스레드 싱글톤

- 싱글톤으로 만들어진 객체는 어플리케이션에서 하나만 생성이 되어야 하는데 멀티 스레드에서는 하나의 객체가 생성되는 것을 보장하고 있지 않다.

Singleton
-Instance: Singleton
-Singleton() +getInstance(): Singleton

```
public class Singleton {  
    private static Singleton instance;  
    private String value;
```

```
    private Singleton(String value) {  
        System.out.println(value  
            + " 값으로 Singleton 객체가 생성 되었습니다");  
        this.value = value;  
    }
```

```
    public static Singleton getInstance(String value) {  
        if (instance == null) {  
            instance = new Singleton(value);  
        }  
        return instance;  
    }
```

```
    public String getValue() {  
        return this.value;  
    }  
}
```

@Test

```
void multiGetInstance() {  
    Runnable singleton_thread_01 = ()-> {  
        Singleton singleton01 = Singleton.getInstance("Init_01");  
        System.out.println("singleton01 :: " + singleton01.getValue());  
    };  
}
```

```
Runnable singleton_thread_02 = ()-> {  
    Singleton singleton02 = Singleton.getInstance("Init_02");  
    System.out.println("singleton02 :: " + singleton02.getValue());  
};
```

```
read thread01 = new Thread(singleton_thread_01);  
read thread02 = new Thread(singleton_thread_02);  
read01.start();  
read02.start();
```

싱글톤 패턴은 Application에서 하나만 생성 되어야 함  
singleton\_thread\_01 이 실행하여 객체 생성하기 전에  
singleton\_thread\_02 이 실행되어 두개의 객체가 생긴다

```
Init_01 값으로 Singleton 객체가 생성 되었습니다  
Init_02 값으로 Singleton 객체가 생성 되었습니다  
singleton02 :: Init_02  
singleton01 :: Init_01
```

```
Init_01값으로 Singleton 객체가 생성 되었습니다  
singleton01 :: Init_01  
singleton02 :: Init_01
```

```
public class Singleton {  
    private static Singleton instance;  
    private String value;  
  
    private Singleton(String value) {  
        System.out.println(value  
            + " 값으로 Singleton 객체가 생성 되었습니다");  
        this.value = value;  
    }  
  
    public static Singleton getInstance(String value) {  
        Singleton singleton = instance;  
  
        if (singleton != null) {  
            return singleton;  
        }  
  
        synchronized(Singleton.class) {  
            if (instance == null) {  
                instance = new Singleton(value);  
            }  
        }  
        return instance;  
    }  
  
    public String getValue() {  
        return this.value;  
    }  
}
```

객체 생성시 동기화-synchronized 를  
하여 객체가 생성될 때 까지 락Lock처리

DCL(Double-Checked  
Locking) 방법으로  
수정

성능 저하 발생

싱글톤 패턴은 모든 메소드와 변수를 static로 선언하면 동일한 결과를 얻을 수 있으나 주의가 필요하고 클래스 로더 2개 이상 이라면 직접 지정하여 싱글톤 객체가 하나만 되도록 하여야 하고, 리플렉션, 직렬화, 역직렬화를 사용할 때는 문제의 요소가 있어서 주의가 필요하며, 강한 결합으로 객체 지향 원칙의 개방 폐쇄 원칙에 위배된다. 여러 단점이 있지만 언제나 선택은 개발자의 몫이다.

인스턴스가 생성되어 있으면 인스턴스화 작업을 진행하지 않고 만들어진 객체를 돌려 주고 인스턴스가 생성되어 있지 않으며 동기화 작업을 통해 인스턴스 생성

# 3. 생성 패턴 (Creational Patterns)

## 5-2. 싱글톤 정리

**1. 즉시 초기화:** 인스턴스는 클래스가 로드된 후에 초기화

```
public class InstantlyInitialization {  
    private static InstantlyInitialization instance  
        = new InstantlyInitialization();  
  
    private InstantlyInitialization() {  
    }  
  
    public static InstantlyInitialization getInstance() {  
        return instance;  
    }  
}
```

**2. 정적 블록 초기화:** 인스턴스가 실제로 사용되기 전에 생성  
- 인스턴스를 초기화하면 초기화 중에 예외가 발생하는 경우 처리할 수 있다.

```
public class StaticBlockInitialization {  
    private static StaticBlockInitialization instance;  
  
    private StaticBlockInitialization() {  
        System.out.println("Object initialized by static block");  
    }  
  
    static {  
        try {  
            instance = new StaticBlockInitialization();  
        } catch (Exception e) {  
            throw new RuntimeException("Exception occurred");  
        }  
    }  
  
    public static StaticBlockInitialization getInstance() {  
        return instance;  
    }  
}
```

효율적 메모리는 아님

**3. 초기화 지연:** 인스턴스는 클래스가 로드된 후에 초기화

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

다중 스레드 환경을 지원하기 위해 지연 초기화를 항상

→ **Thread Safe Singleton**

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

성능 향상 방법

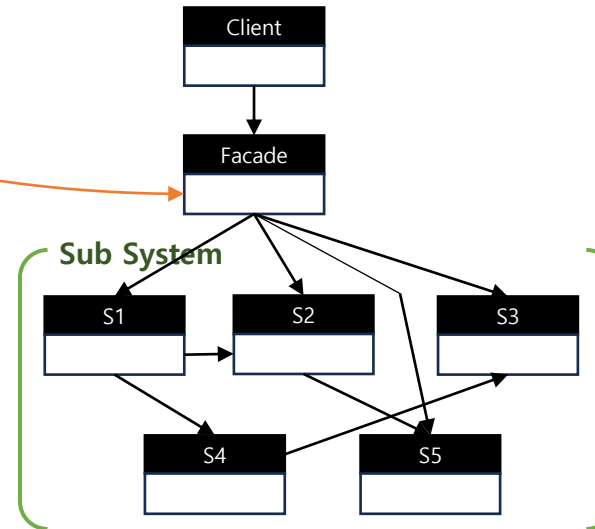
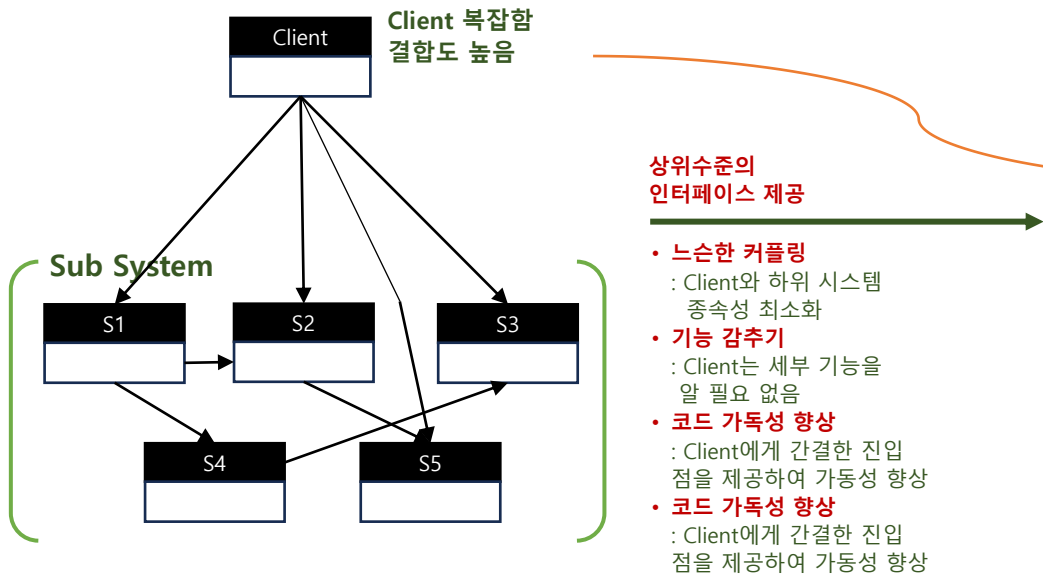
```
public static SynchronizedSingleton getInstance() {  
    if (null == instance) {  
        synchronized (SynchronizedSingleton.class) {  
            if (null == instance) {  
                instance = new SynchronizedSingleton();  
            }  
        }  
    }  
    return instance;  
}
```



# 3. 구조패턴 (Structural Patterns)

## 1. 퍼사드 패턴(Façade Pattern)

- 복잡한 시스템의 인터페이스를 간단화 시키는 구조적 디자인 패턴
- 서브시스템의 기능을 단순화된 상위 수준의 인터페이스를 제공하여 서브 시스템의 종속성을 줄여 시스템의 복잡성을 감출 수 있습니다.
- 복잡성을 관리 및 캡슐화하고, 유지 관리성을 개선하고, 느슨한 결합을 촉진하고, 크고 복잡한 소프트웨어 시스템에서 코드의 전반적인 가독성을 향상시키는 데 사용됩니다



모듈화 되고 유지 관리 가능하며 확장 가능한 소프트웨어 시스템을 구축

### 장점

- 단순화된 인터페이스
- 구현 세부 정보 캡슐화
- 느슨한 결합 촉진
- 더 쉬운 유지 관리
- 향상된 코드 가독성 및 여러 하위 시스템에 대한 통합 API

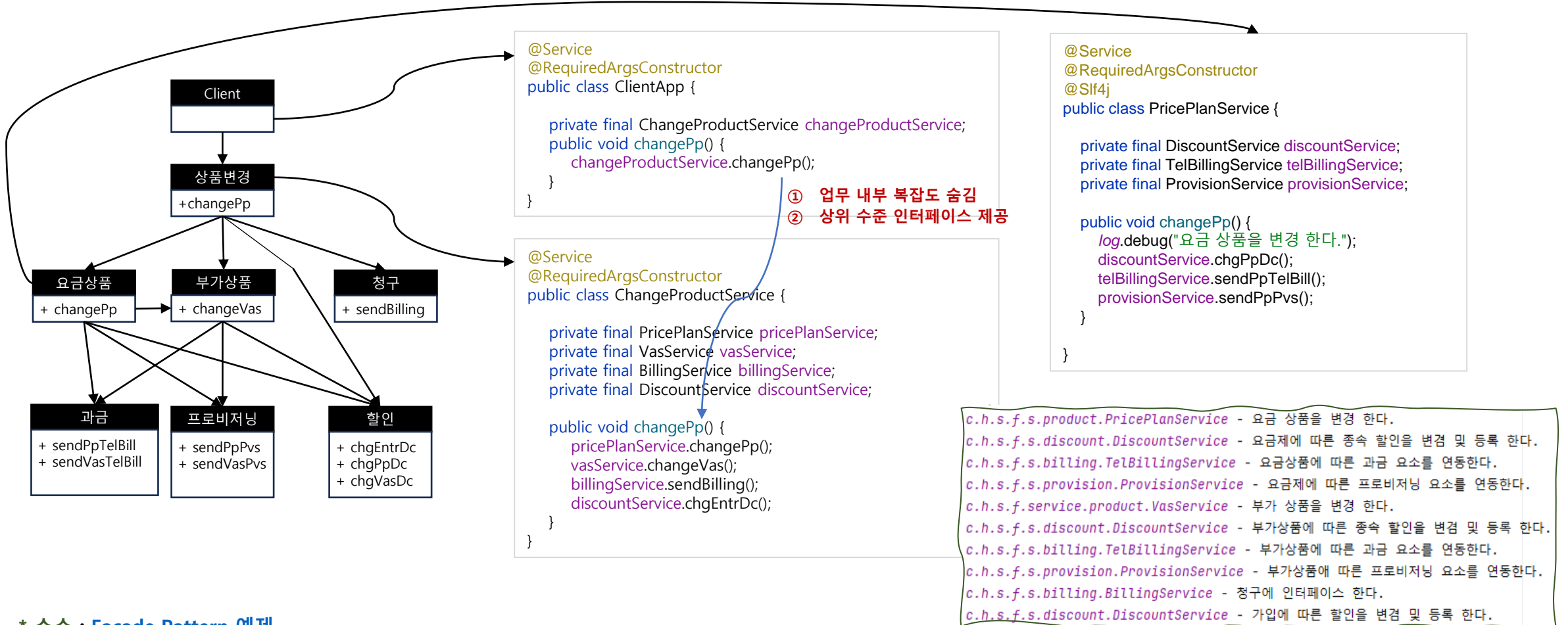
### 단점

- 캡슐화 위반
  - Façade Class는 Sub System 내부 구현을 알아야 해서 내부 사항을 노출 시킨다
- 유연성 감소
  - 특정 서브 시스템에 종속성 발생하여 다른 서브 시스템과 결합도 발생
  - 새로운 기능 추가 및 변경 할 때 퍼사드 클래스 수정
- 단일 책임 원칙 위배
  - 퍼사드 클래스는 여러 서브 시스템과 연결 되어 있어 단일 책임 원칙을 위반할 수 있다. (역할 불분명)
- 복잡한 서브 시스템 처리 어려움
  - 단순한 인터페이스 제공으로 Client의 복잡도는 감소 하지만 내부의 복잡한 서브 시스템을 다루기 어려울 수 있음

# 3. 구조패턴 (Structural Patterns)

## 1-1. 퍼사드 패턴(Façade Pattern) 예제

- 통신사에 가입한 가입자가 요금제변경을 하면 요금 상품에 따라 부가 상품이 변경되고 요금 상품에 따라서 할인율이 변경되고 과금, 청구, 프로비저닝 요소를 변경되어야 합니다.

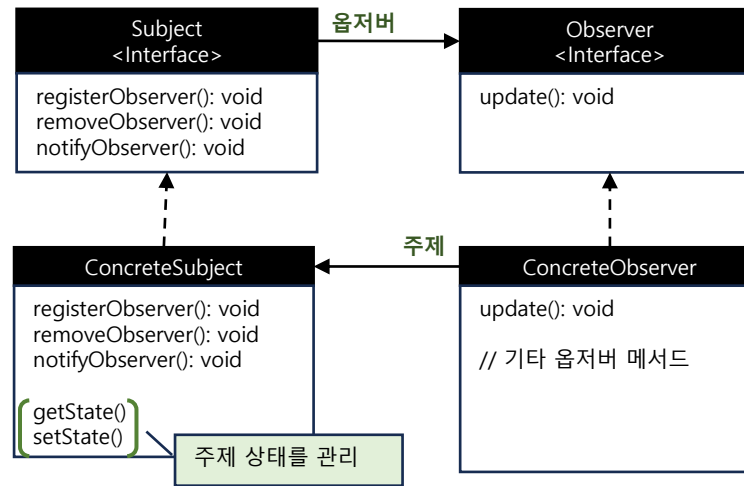
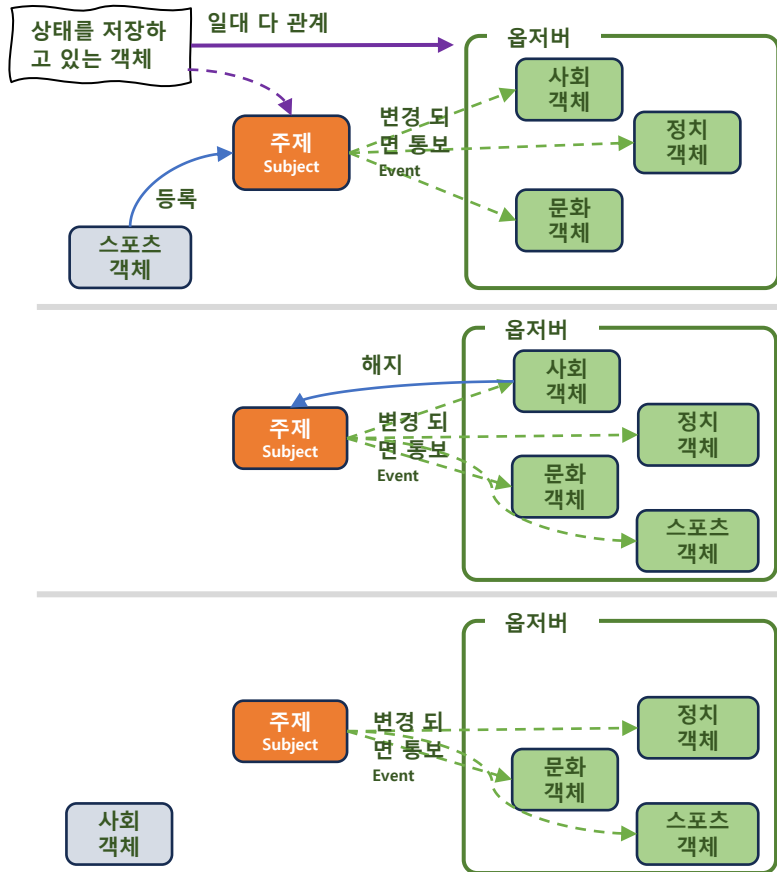


\* 소스 : [Façade Pattern 예제](#)

# 3. 행동패턴 (Behavioral Patterns)

## 1. 옵저버 팩토리 (Observer Pattern)

- 객체 간의 일대다 종속성이므로 한 객체의 상태가 변경되면 모든 종속 객체에 알림이 전송되는 게시자-구독자 모델.
- 다른 객체가 감시하고 있는 객체를 Subject/Publisher라고 하고, 주체의 상태를 감시하고 있는 Object를 Observer/Subscriber라고 합니다.
- 이 패턴은 주로 분산 이벤트 핸들링 시스템을 구현하는 데 사용되고, 객체의 상태 변화를 감지하고 이에 따라 다른 동작을 수행해야 할 때 유용합니다.



- **Subject** : 객체에서 옵저버로 등록/삭제/통보하고 싶을 때 이 인터페이스를 사용해야 합니다.
- **ConcreteSubject** : Subject 인터페이스의 구현체로 등록/삭제 및 상태가 변경 될 때 마다 Observer에 통보하는 기능을 구현해야 합니다.
- **Observer** : 주제(Subject)의 상태가 변경될 때 호출되는 update 메서드로 옵저버 대상은 반드시 Observer 인터페이스를 구현해야 합니다.
- **ConcreteObserver** : Observer 인터페이스의 구현체로 옵저버는 특정 주제에 등록되어야 통보를 받을 수 있습니다.

### 장점

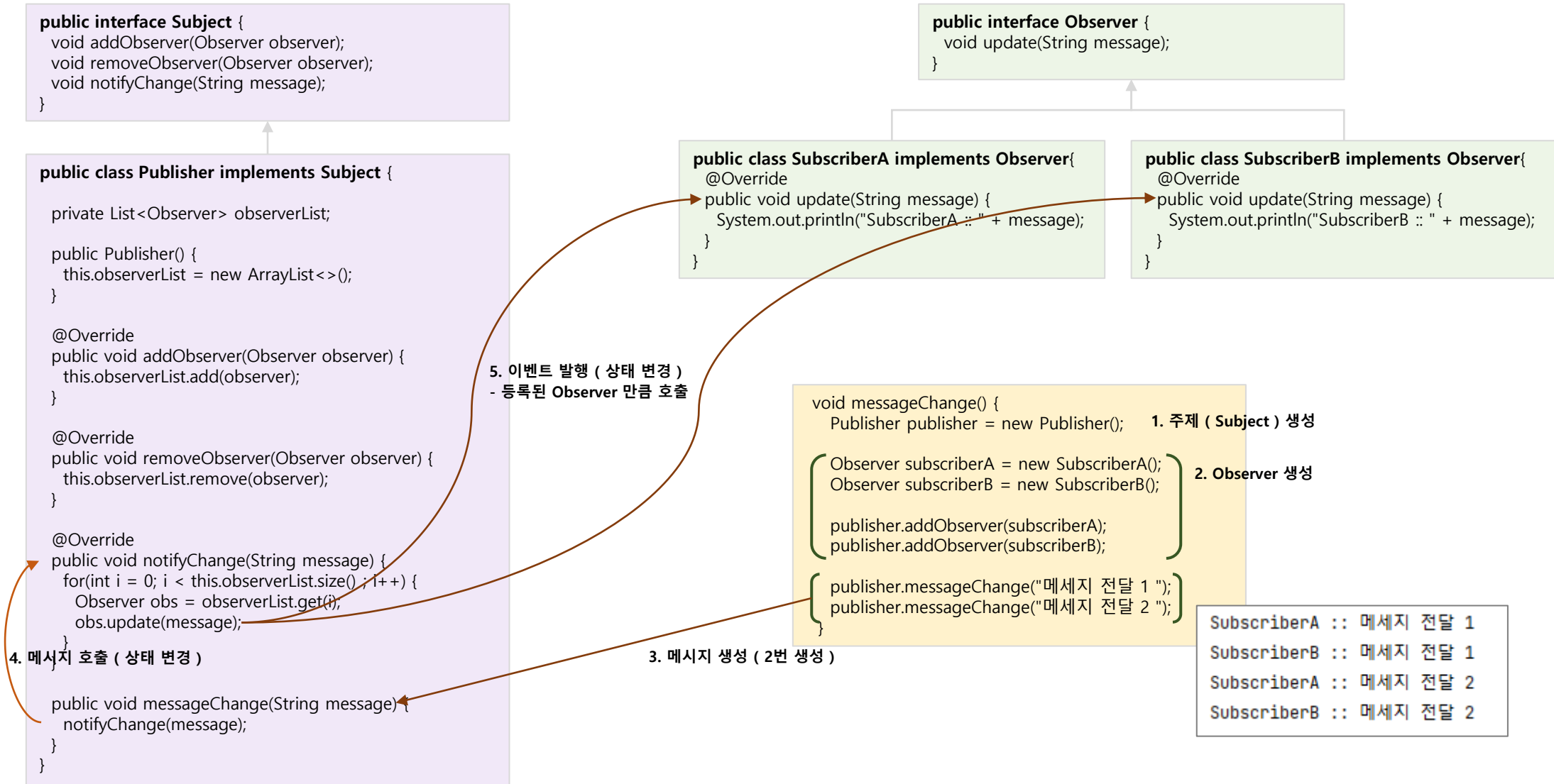
- **느슨한 결합 ( Loose Coupling )**
  - 주제(Subject)와 옵저버(Observer)는 서로 독립적으로 존재하며, 상호작용할 때 직접적인 의존성이 없어 유연성이 높아진다.
- **확장성 ( Scalability )**
  - 새로운 옵저버를 추가하거나 기존 옵저버를 제거하는 것이 간단하여 시스템을 확장하거나 변경하기 용이합니다
- **이벤트 핸들링**
  - 상태 변화가 발생할 때마다 옵저버에게 알림을 보내므로, 이벤트 기반 시스템에서 유용합니다

### 단점

- **성능 저하**
  - 옵저버 패턴은 많은 옵저버가 등록되어 있을 때 각 옵저버에게 통지하는 데 시간이 소요될 수 있습니다.
- **순서 문제**
  - 어떤 옵저버가 먼저 실행되어야 하는지 정확히 결정해야 하므로 옵저버들 사이의 호출 순서를 관리해야 합니다.

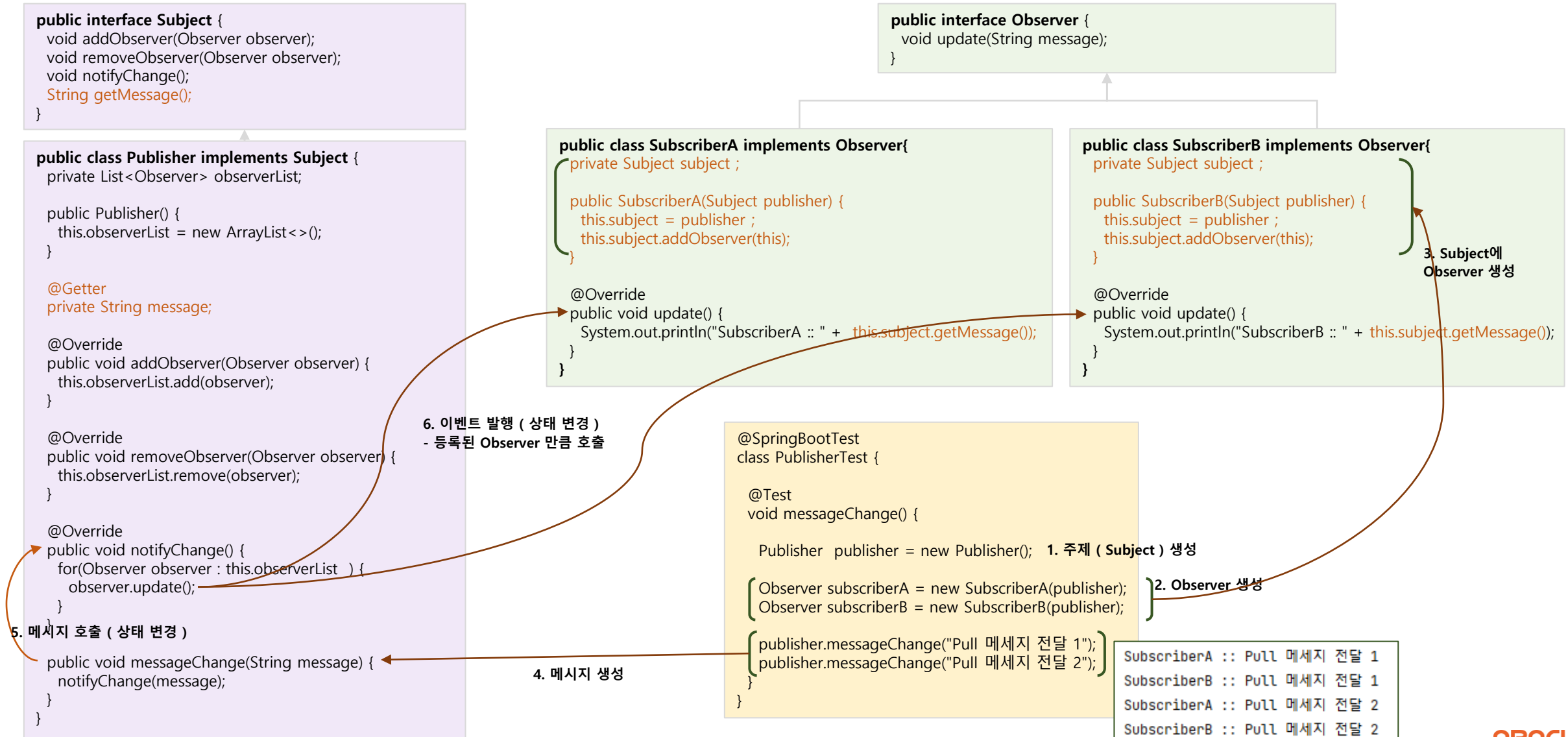
# 3. 행동패턴 (Behavioral Patterns)

## 1-2. 옵저버 패턴 – Push 방식



# 3. 행동패턴 (Behavioral Patterns)

## 1-3. 옵저버 패턴 – Pull 방식



# 3. 행동패턴 (Behavioral Patterns)

## 1-4. 자바빈 PropertyChangedEvents, PropertyChangedListener 사용

- **PropertyChangedListener**은 객체 간의 상태 변화를 감지하고 이를 처리하는데 유용한 패턴
- **PropertyChangedEvents**는 자바에서 **\*\*빈(Beans)\*\***이 **바운드(Bound)** 또는 **제약(Constrained)** 속성을 변경할 때 전달되는 이벤트
  - `getPropertyName()`: 변경된 속성의 프로그래밍적 이름을 가져옵니다. 여러 속성이 변경되었다면 이름은 null일 수 있습니다.
  - `getOldValue()`: 속성의 이전 값을 가져옵니다. 여러 속성이 변경되었다면 값은 null일 수 있습니다.
  - `getNewValue()`: 속성의 새 값을 가져옵니다. 여러 속성이 변경되었다면 값은 null일 수 있습니다.
- 이벤트 소스는 **PropertyChangedListener** 및 **VetoableChangeListener** 메서드에 인자로 **PropertyChangedEvent** 객체를 보냅니다

```
public class MyBean {
    private String myProperty;
    private PropertyChangedSupport propertyChangeSupport =
        new PropertyChangedSupport(this);

    public void setMyProperty(String newValue) {
        String oldValue = myProperty;
        myProperty = newValue;
        propertyChangeSupport.firePropertyChange("myProperty", oldValue, newValue);
    }

    public String getMyProperty() {
        return myProperty;
    }

    public void addPropertyChangedListener(PropertyChangedListener listener) {
        propertyChangeSupport.addPropertyChangedListener(listener);
    }

    public void removePropertyChangedListener(PropertyChangedListener listener) {
        propertyChangeSupport.removePropertyChangedListener(listener);
    }
}
```

5. 이벤트 발행

```
public class MyPropertyChangedListener implements PropertyChangedListener {

    @Override
    public void propertyChange(PropertyChangedEvent evt) {
        String propertyName = evt.getPropertyName();
        Object oldValue = evt.getOldValue();
        Object newValue = evt.getNewValue();

        System.out.println("Property changed: " + propertyName);
        System.out.println("Old value: " + oldValue);
        System.out.println("New value: " + newValue);
    }
}
```

```
Property changed: myProperty
Old value: null
New value: JAVA API Level : 옵저버 : Bean의 속성 초기값
```

```
Property changed: myProperty
Old value: JAVA API Level : 옵저버 : Bean의 속성 초기값
New value: JAVA API Level : 옵저버 : Bean의 속성 변경
```

```
class MyBeanTest {

    @Test
    void addPropertyChangedListener() {
        MyBean myBean = new MyBean();
        MyPropertyChangedListener listener = new MyPropertyChangedListener();
        myBean.addPropertyChangedListener(listener);
        myBean.setMyProperty("JAVA API Level : 옵저버 : Bean의 속성 초기값");
        myBean.setMyProperty("JAVA API Level : 옵저버 : Bean의 속성 변경");
    }
}
```

1. Bean 생성  
2. Listener 생성

4. 값 변경

3. Bean에 리스너 등록

### 장점

- **이벤트 기반 모델**
  - 속성 변경 이벤트를 기반으로 동작으로 체 간의 상호작용을 이벤트 기반으로 설계.
- **느슨한 결합**
  - 리스너(Listener) 간의 결합도가 낮아집니다. 빈이나 리스너를 변경하더라도 다른 부분에 영향을 미치지 않습니다
- **확장성**
  - 새로운 리스너를 추가하거나 기존 리스너를 제거할 수 있습니다

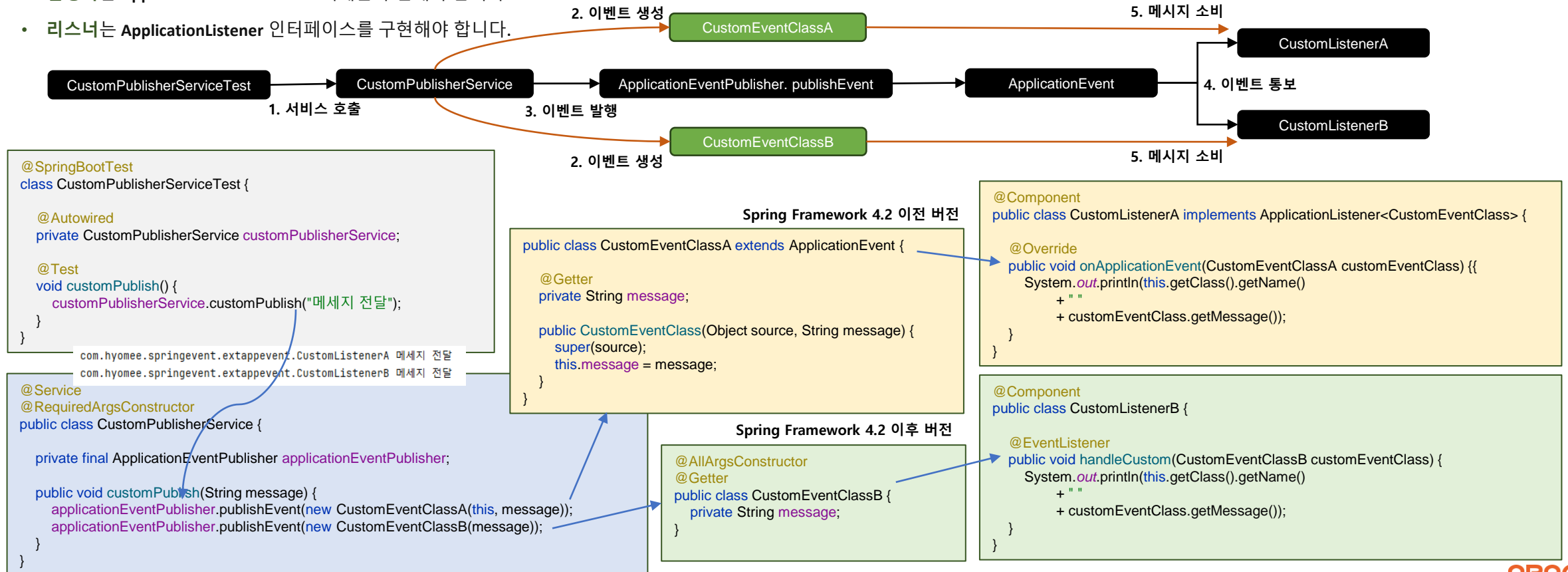
### 단점

- **명시적 등록**
  - 빈 클래스에서 리스너를 명시적으로 등록해야 하므로 코드의 가독성이 떨어뜨릴 수 있습니다.
- **메모리 사용**
  - 리스너 객체가 메모리를 차지하므로 많은 리스너를 등록할 경우 메모리 사용량이 증가할 수 있습니다
- **복잡성**
  - 여러 속성에 대한 리스너를 관리해야 하므로 복잡성이 증가할 수 있습니다.

# 3. 행동패턴 (Behavioral Patterns)

## 1-5. Spring Event

- 서비스 간의 강한 의존성을 줄이기 위해서 내부에서 데이터를 전달하는 방법 중 하나로 EDA(Event-driven architecture)는 애플리케이션의 분리와 유연성을 촉진하는 아키텍처 패턴입니다.
- 이벤트 발행은 `ApplicationContext`가 제공하는 기능 중 하나입니다. 이를 위해 몇 가지 간단한 지침이 다음과 같습니다.
  - 이벤트 클래스는 Spring Framework 4.2 이전 버전을 사용하는 경우에는 `ApplicationEvent`를 확장해야 하고 4.2 버전 이후에는 이벤트 클래스가 더 이상 `ApplicationEvent` 클래스를 확장할 필요가 없습니다.
  - 발행자는 `ApplicationEventPublisher` 객체를 주입해야 합니다.
  - 리스너는 `ApplicationListener` 인터페이스를 구현해야 합니다.





# 3. 행동패턴 (Behavioral Patterns)

## 1-6. Spring Event - 비동기

- `@EnableAsync` 어노테이션을 통해 비동기를 사용하겠다고 선언하고, 비동기로 동작하고자 하는 메서드에 `@Async` 어노테이션을 설정하면 됩니다.

```
@EnableAsync
@SpringBootApplication
public class PatternApplication {
    public static void main(String... args) {
        SpringApplication.run(PatternApplication.class, args);
    }
}
```

```
@RequiredArgsConstructor
@RestController
@Slf4j
public class EventController {

    private final CustomPublisherService customPublisherService;

    @GetMapping("/event/{message}")
    private void sendMessage(@PathVariable String message) {
        customPublisherService.customPublish(message);
        log.info("메세지 전송됩니다.");
    }
}
```

```
@Service
@RequiredArgsConstructor
public class CustomPublisherService {

    private final ApplicationEventPublisher applicationEventPublisher;

    public void customPublish(String message) {
        applicationEventPublisher.publishEvent(new CustomEventClassA(this, message));
        applicationEventPublisher.publishEvent(new CustomEventClassB(message));
    }
}
```

```
public class CustomEventClassA extends ApplicationEvent {

    @Getter
    private String message;

    public CustomEventClass(Object source, String message) {
        super(source);
        this.message = message;
    }
}
```

```
@AllArgsConstructor
@Getter
public class CustomEventClassB {
    private String message;
}
```

```
@Component
public class CustomListenerA implements ApplicationListener<CustomEventClassA> {

    @Async
    @Override
    public void onApplicationEvent(CustomEventClassA customEventClassA) {
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        System.out.println(this.getClass().getName()
            + " "
            + customEventClassA.getMessage());
    }
}
```

```
@Component
public class CustomListenerB {

    @Async
    @EventListener
    public void handleCustom(CustomEventClassB customEventClassB)
        throws InterruptedException {
        Thread.sleep(2000);
        System.out.println(this.getClass().getName()
            + " "
            + customEventClassB.getMessage());
    }
}
```

```
2024-02-28T22:48:14.714+09:00 INFO 26712 --- [nio-8080-exec-1] c.h.s.extappevent.EventController : 메세지 전송됩니다.
com.hyomee.springevent.extappevent.CustomListenerB 메세지
com.hyomee.springevent.extappevent.CustomListenerA 메세지
```



# 3. 행동패턴 (Behavioral Patterns)

## 1-6. Spring Event - @TransactionalEventListener

- Spring 4.2부터 프레임워크는 @EventListener의 확장인 새로운 @TransactionalEventListener 주석을 제공하여 이벤트 리스너를 트랜잭션 단계에 바인딩할 수 있습니다.
- 다음 트랜잭션 단계에 바인딩할 수 있습니다.
  - AFTER\_COMMIT(기본값)은 트랜잭션이 성공적으로 완료된 경우 이벤트를 발생시키는 데 사용됩니다.
  - AFTER\_ROLLBACK – 트랜잭션이 롤백 된 경우
  - AFTER\_COMPLETION – 트랜잭션이 완료된 경우(AFTER\_COMMIT 및 AFTER\_ROLLBACK)
  - BEFORE\_COMMIT는 트랜잭션 커밋 직전에 이벤트를 발생시키는 데 사용됩니다.

```
@TransactionalEventListener(phase = TransactionPhase.BEFORE_COMMIT)
public void handleCustom(CustomSpringEvent event) {
    System.out.println("Handling event inside a transaction BEFORE COMMIT.");
}
```

- 이 수신기는 이벤트 생산자가 실행 중이고 커밋 되려는 트랜잭션이 있는 경우에만 호출됩니다.
- 그리고 실행 중인 트랜잭션이 없는 경우 fallbackExecution 속성을 true로 설정하여 이를 재정의하지 않는 한 이벤트가 전혀 전송되지 않습니다.

[참고 : Spring-Event](#)

**THANKS**  
**ABACUS**

[www.iabacus.co.kr](http://www.iabacus.co.kr)

Tel. 82-2-2109-5400

Fax. 82-2-6442-5409