

학습목표

- 소프트웨어 디자인 패턴에 대한 개념을 이해하고, 이를 설명할 수 있다.
- 소프트웨어 생성 패턴의 종류를 이해하고, 이를 적용할 수 있다.

학습내용

- 소프트웨어 디자인 패턴
- 소프트웨어 생성 패턴

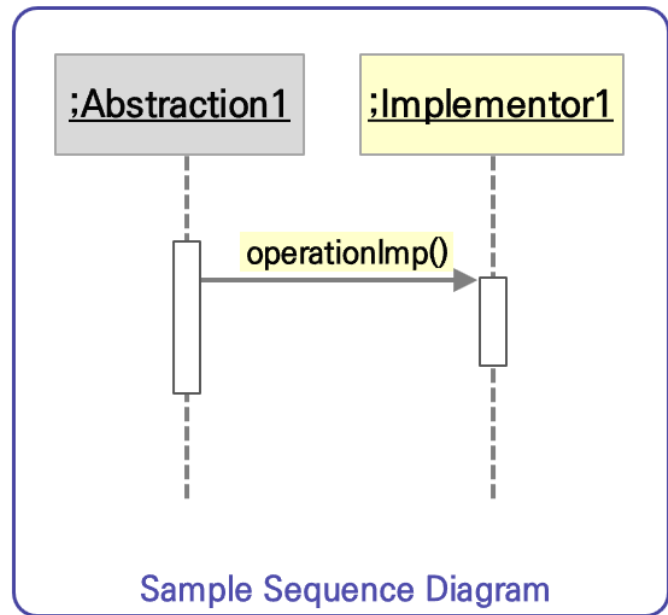
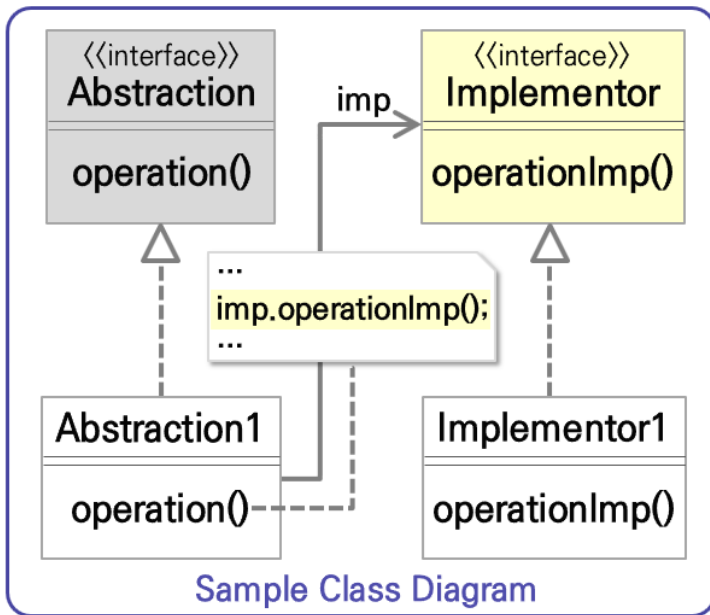
소프트웨어 디자인 패턴

1. 디자인 패턴이란?



소프트웨어 디자인 패턴이란?

소프트웨어 디자인에서 **계속 재현되는 문제를 해결하는 재사용 가능한 해결법**



〈 객체지향 모델링 〉

출처: <https://commons.wikimedia.org>

소프트웨어 디자인 패턴

1. 디자인 패턴이란?

- 객체지향 프로그래밍에서 공통으로 **디자인 문제를 찾아내고 해결하는 가이드라인**
- **특정 유형의 프로그래밍 문제를 해결**하는 방식을 제공해주는 역할
- 알고리즘과 같이 프로그램 코드로 바로 변환될 수 있는 형태는 아니지만, **특정 상황에서 구조적인 문제를 해결하는 방식을 설명**

소프트웨어 디자인 패턴

1. 디자인 패턴이란?

⚙ 디자인 문제

- 라이브러리, 패키지, 모듈, 프레임 워크 등이 없다면?

프로그래머들은
시행착오를 반복

클래스, 함수, 메소드를
직접 만들어 프로그래밍

- 설계의 오류, 개발팀 내의 커뮤니케이션의 오류와 해당 문제가 앞으로 지속해서 발생할 가능성이 있음

소프트웨어 디자인 패턴

2. 디자인 패턴 장점 및 필요성

- 개발자의 경험을 모아서 **공통적인 소프트웨어 디자인 문제를 해결하는 데 도움**이 됨

디자인 패턴 --> 교육의 도구, 프로그래밍의 필수적인 부분

디자인 문제와 그 해결책을 찾을 때
디자인 패턴이 간결한 용어 모음을 제공

소프트웨어 디자인 패턴

2. 디자인 패턴 장점 및 필요성

- 업무 논의 및 디자인 문서를 작성할 때 등 상호 간 의사결정에 용어로 쓰임
- 개발자 간의 원활한 의사소통, 소프트웨어 구조 파악 용이, 재사용을 통한 개발 시간 단축

설계 변경 요청에 대한
유연한 대처가 가능해요!



3. 디자인 패턴 단점

객체지향
설계 위주
사용

객체지향
구현 위주의
사용

초기 투자
비용의 부담

소프트웨어 디자인 패턴

4. 디자인 패턴 분류

1) 소프트웨어 생성 패턴(Creation Pattern)

객체의 생성 과정에 관여하는 패턴

싱글톤(Singleton) 패턴

- 클래스의 인스턴스가 하나임을 보장하고 접근할 수 있는 전역적인 접근점을 제공

Singleton

- uniqueInstance
- singletonData

+ static Instance()
+ SingletonOperation()
+ GenSingletonData()

Instance

return uniqueInstance;

출처: <https://commons.wikimedia.org>

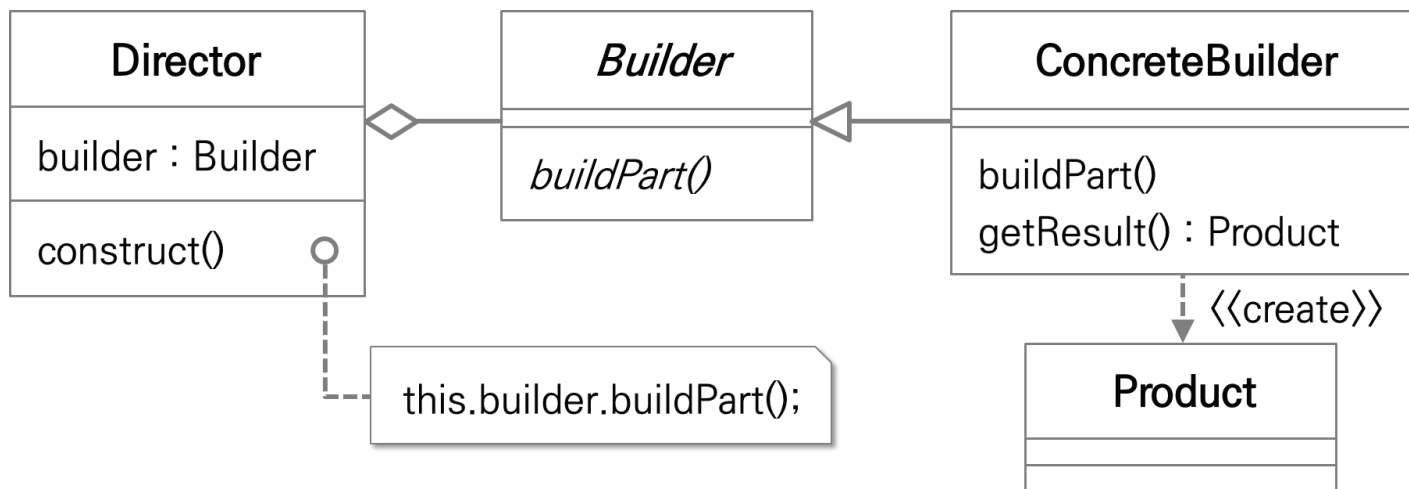
소프트웨어 디자인 패턴

4. 디자인 패턴 분류

1) 소프트웨어 생성 패턴(Creation Pattern)

빌더(Builder) 패턴

- 복합 객체의 생성과정과 표현과정을 분리시켜 동일한 생성과정에서 다양한 표현을 생성



출처: <https://commons.wikimedia.org>

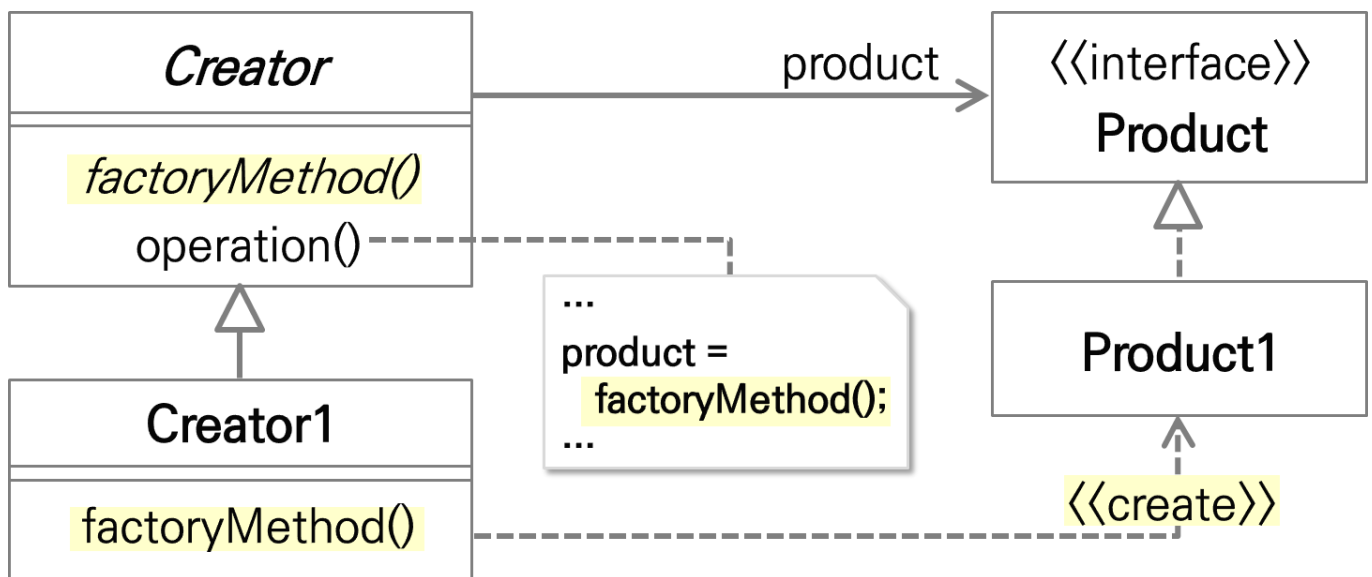
소프트웨어 디자인 패턴

4. 디자인 패턴 분류

1) 소프트웨어 생성 패턴(Creation Pattern)

팩토리 메서드(Factory Method) 패턴

- 객체를 생성하는 인터페이스를 정의하지만,
인스턴스를 만드는 클래스는 서브 클래스에서 결정



출처: <https://commons.wikimedia.org>

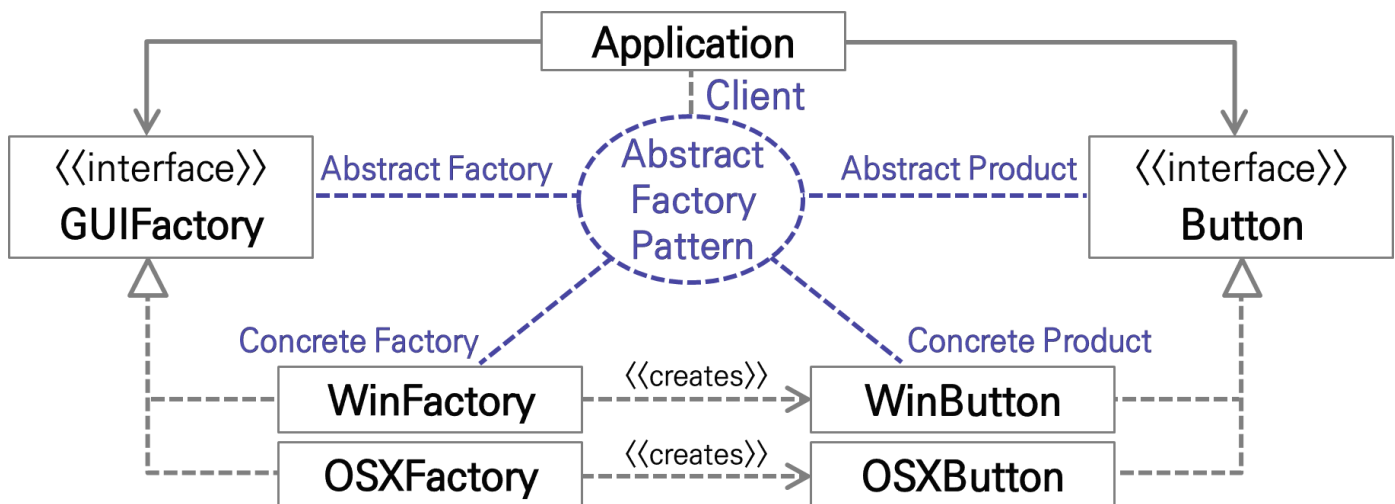
소프트웨어 디자인 패턴

4. 디자인 패턴 분류

1) 소프트웨어 생성 패턴(Creation Pattern)

추상 팩토리(Abstract Factory) 패턴

- 구체적인 클래스를 지정하지 않고 관련성이 있거나 독립적인 객체들을 생성하기 위한 인터페이스를 제공



출처: <https://commons.wikimedia.org>

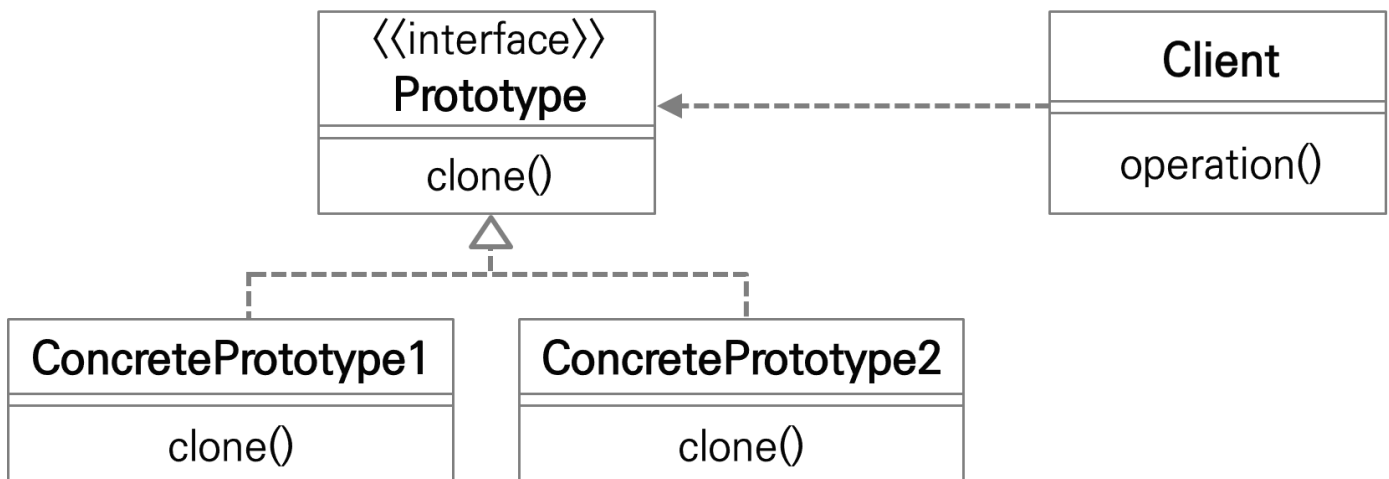
소프트웨어 디자인 패턴

4. 디자인 패턴 분류

1) 소프트웨어 생성 패턴(Creation Pattern)

원형(Prototype) 패턴

- 생성할 객체의 종류를 명시하는데 원형이 되는 예시물을 이용
- 새로운 객체를 이 원형들을 복사함으로써 생성



출처: <https://commons.wikimedia.org>

소프트웨어 디자인 패턴

4. 디자인 패턴 분류

2) 구조 패턴(Structural Pattern)

클래스나 객체의 합성에 관한 패턴

적응자(Adapter or Wrapper) 패턴

- 클래스의 인터페이스를 사용자가 기대하는 다른 인터페이스로 변환하는 패턴
- 호환성이 없는 인터페이스 때문에 함께 동작할 수 없는 클래스들이 함께 작동하도록 해주는 패턴

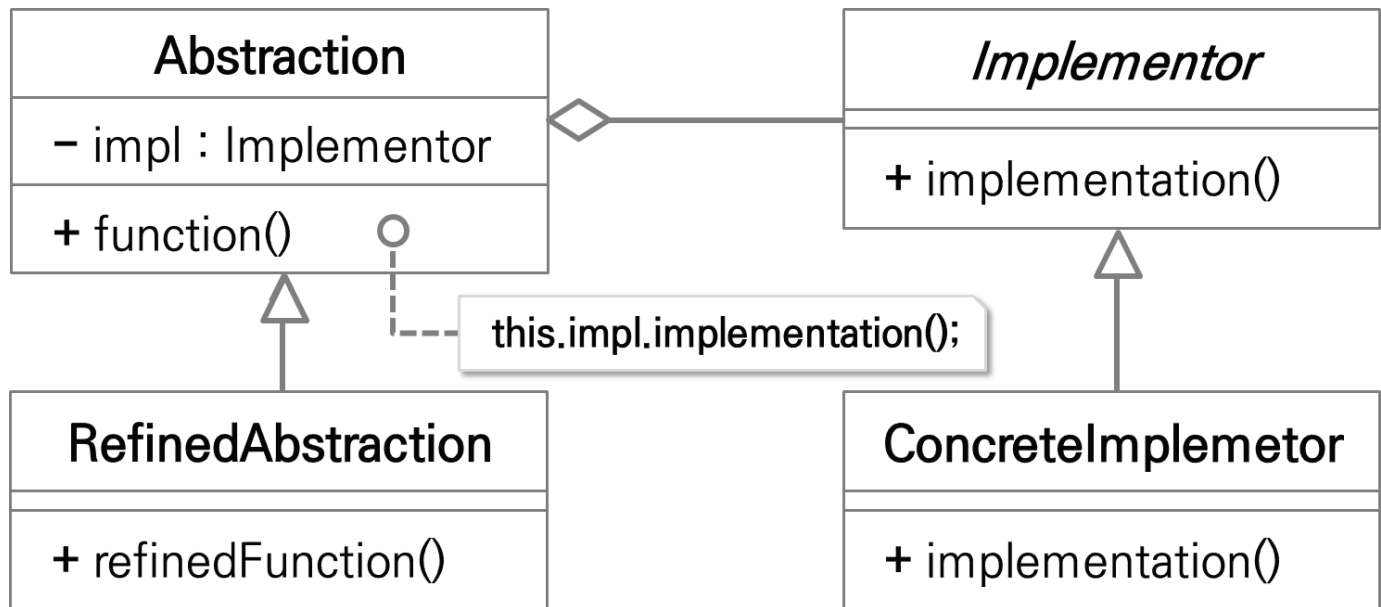
소프트웨어 디자인 패턴

4. 디자인 패턴 분류

2) 구조 패턴(Structural Pattern)

브리지(Bridge) 패턴

- 구현부에 추상층을 분리하여 각자 변형할 수 있는 패턴



출처: <https://commons.wikimedia.org>

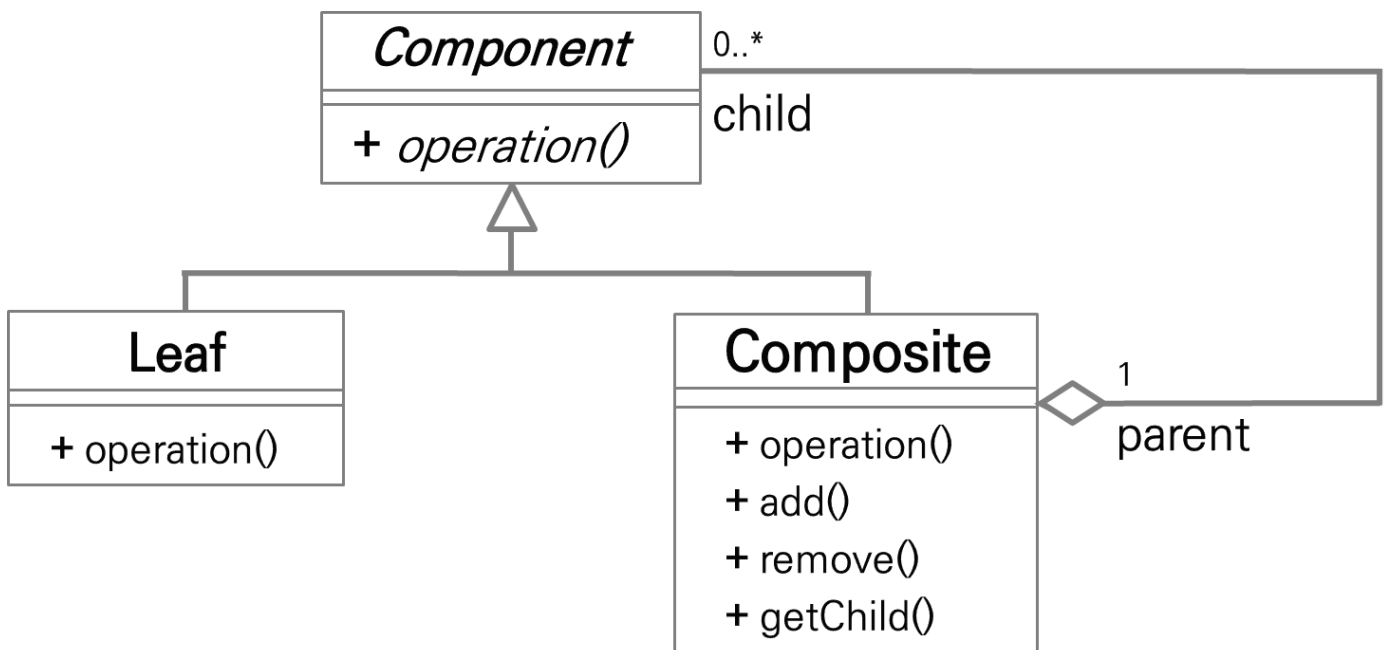
소프트웨어 디자인 패턴

4. 디자인 패턴 분류

2) 구조 패턴(Structural Pattern)

컴포지트(Composite) 패턴

- 객체들의 관계를 트리 구조로 부분-전체 계층을 표현하는 패턴, 사용자가 단일/ 복합 객체 모두 동일하게 다룸



출처: <https://ko.wikipedia.org>

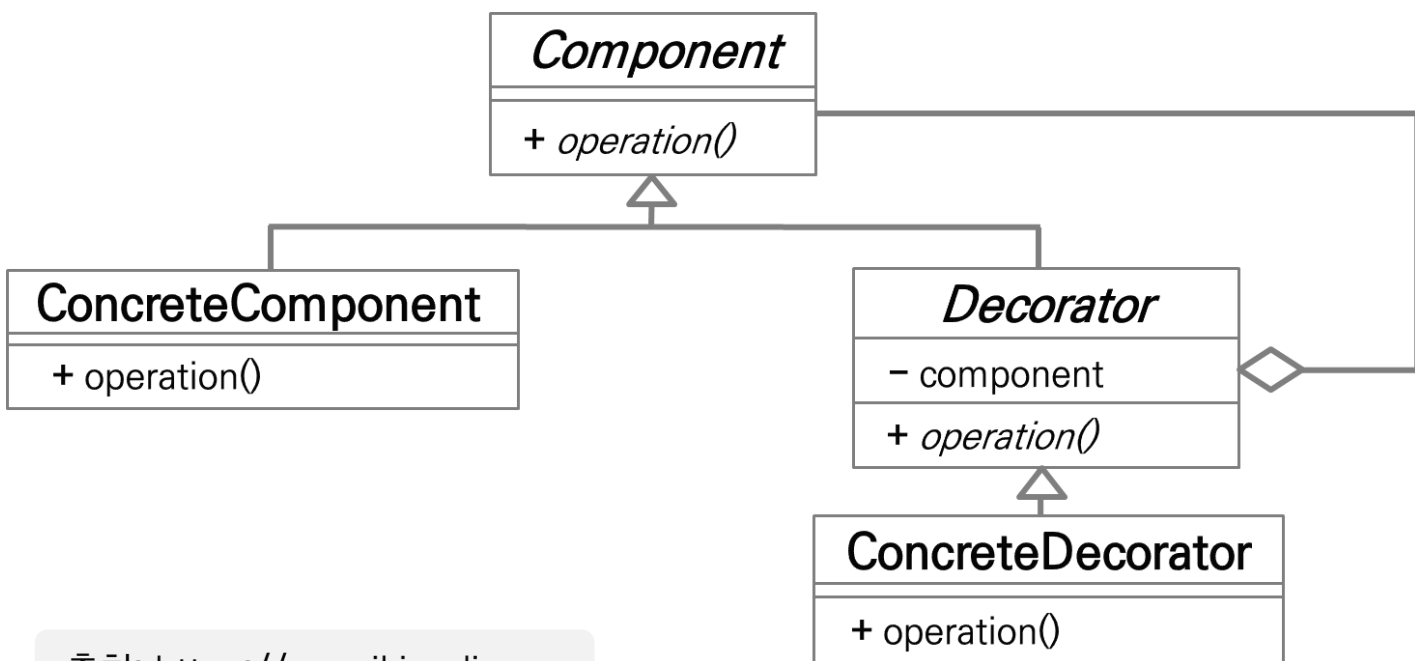
소프트웨어 디자인 패턴

4. 디자인 패턴 분류

2) 구조 패턴(Structural Pattern)

데코레이터(Decorator) 패턴

- 주어진 상황 및 용도에 따라 어떤 객체에 덧붙이는 패턴
- 기능확장이 필요할 때 대신 쓸 수 있는 대안



출처: <https://en.wikipedia.org>

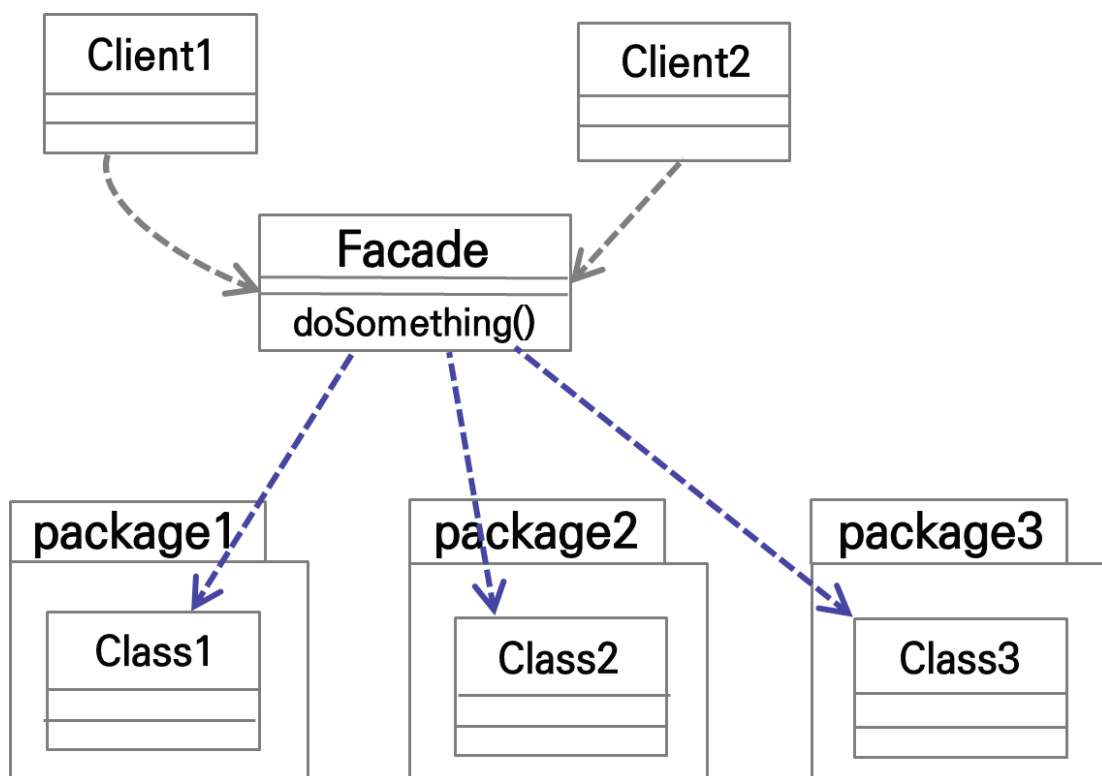
소프트웨어 디자인 패턴

4. 디자인 패턴 분류

2) 구조 패턴(Structural Pattern)

퍼사드(Facade) 패턴

- 서브 시스템에 있는 인터페이스 집합에 하나의 인터페이스를 제공
- 서브 시스템을 좀 더 쉽게 사용하기 위해 고수준의 인터페이스를 정의



출처: <https://en.wikipedia.org>

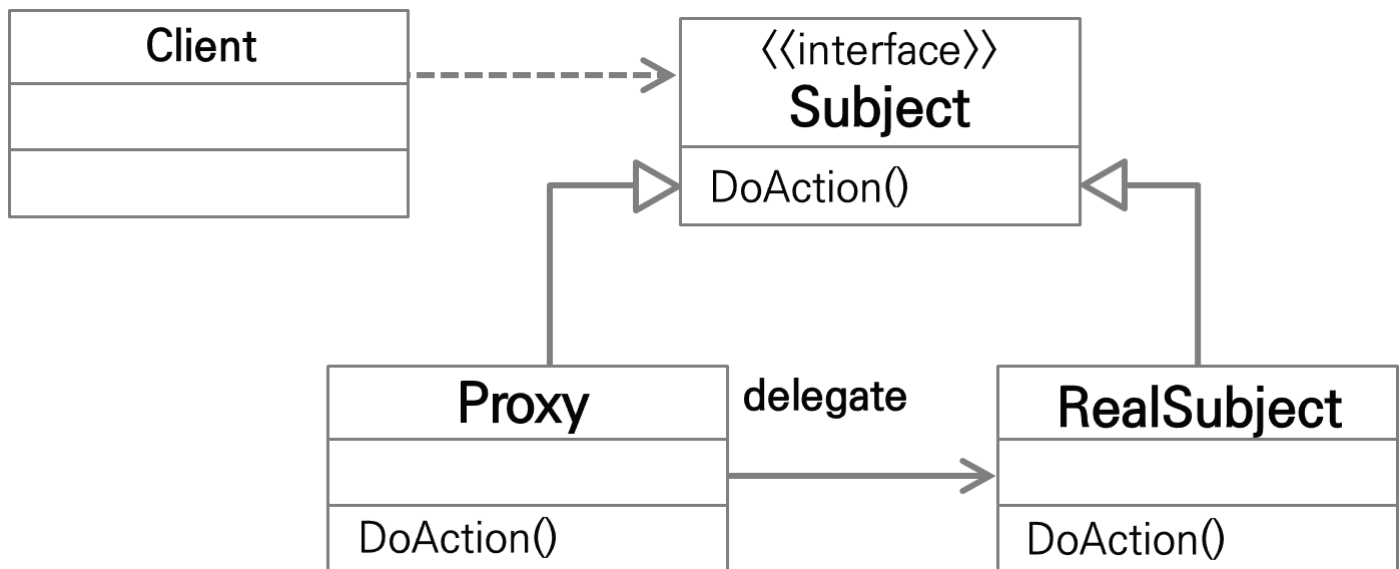
소프트웨어 디자인 패턴

4. 디자인 패턴 분류

2) 구조 패턴(Structural Pattern)

프록시(Proxy) 패턴

- 어떤 다른 객체로 접근하는 것을 통제하기 위해 그 객체의 매니저 혹은 자리 채움자를 제공



출처: <https://ko.wikipedia.org>

소프트웨어 디자인 패턴

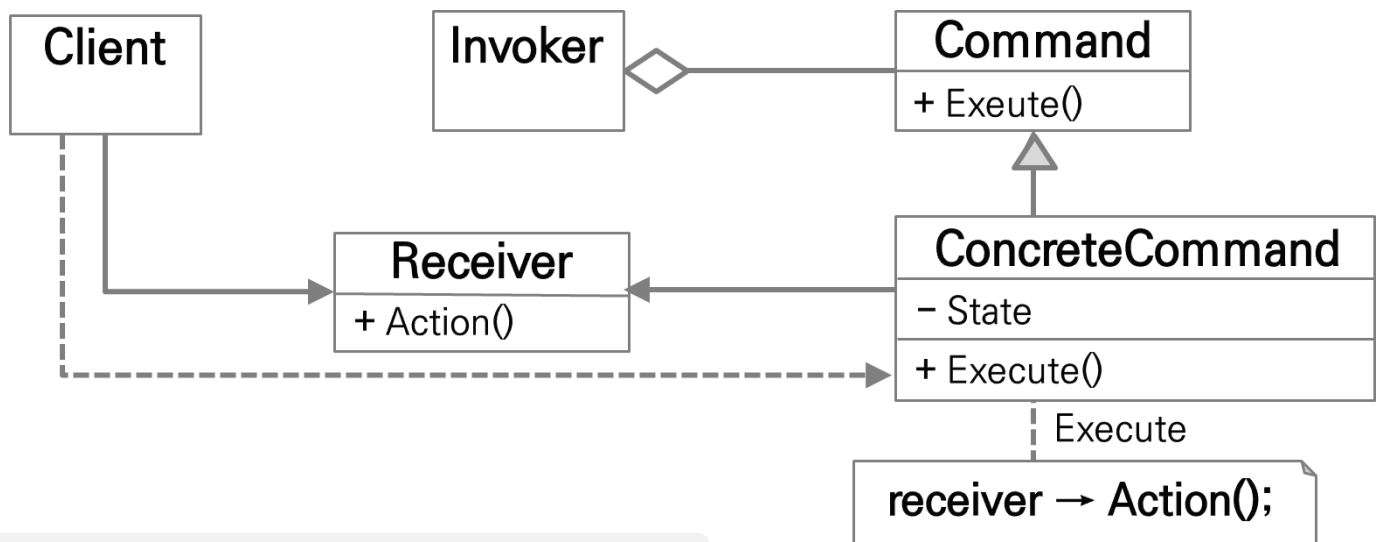
4. 디자인 패턴 분류

3) 행동 패턴

클래스나 객체들이 상호작용하는 방법과
책임을 분산하는 방법

커맨드(Command) 패턴

- 요청을 객체로 캡슐화하여 서로 다른 사용자의 매개 변수화, 요청 저장 혹은 로깅, 연산의 취소를 지원하게 만듦



출처: <https://commons.wikimedia.org>

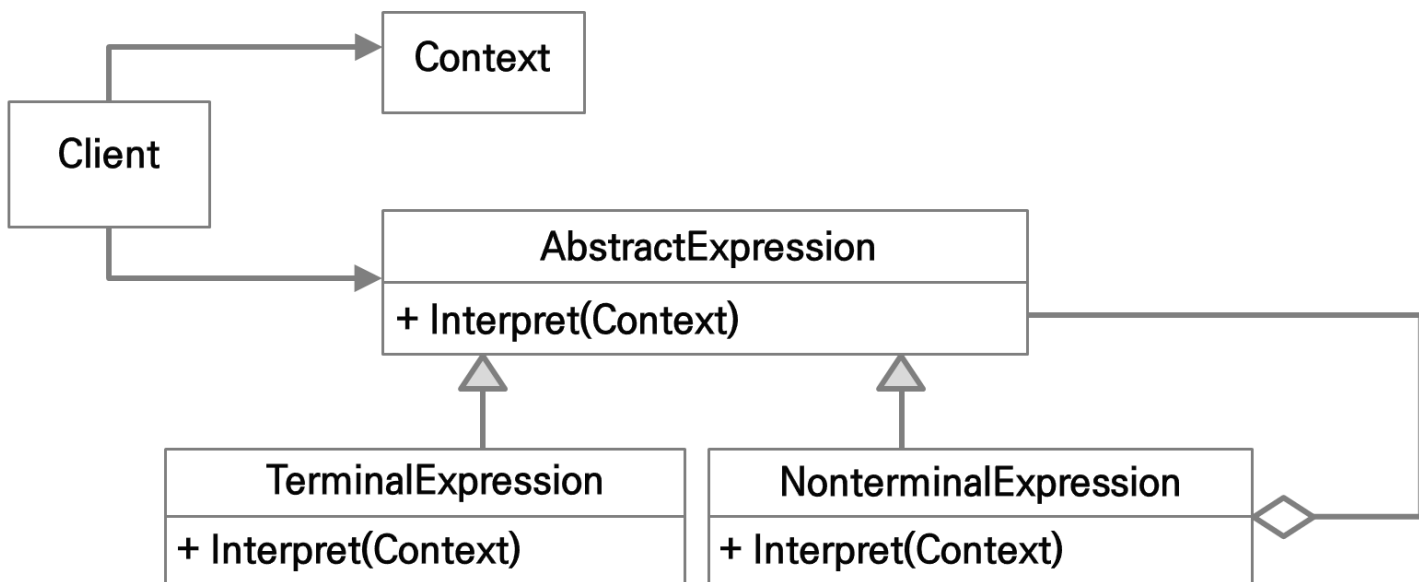
소프트웨어 디자인 패턴

4. 디자인 패턴 분류

3) 행동 패턴

인터프리터(Interpreter)패턴

- 주어진 언어에 대해서 문법을 위한 표현수단을 정의
- 해당 언어로 된 문장을 해석하는 해석기를 사용하는 패턴



출처: <https://pt.wikipedia.org>

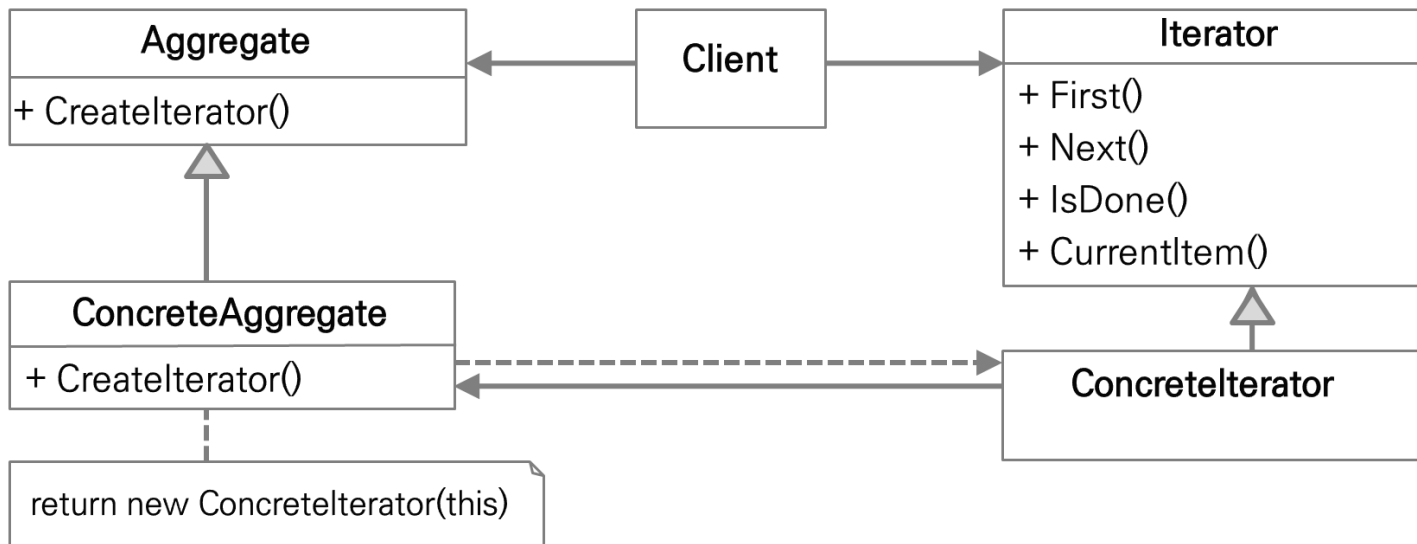
소프트웨어 디자인 패턴

4. 디자인 패턴 분류

3) 행동 패턴

이터레이터(Iterator) 패턴

- 내부 표현 부를 노출하지 않고, 어떤 객체 집합의 원소들을 순차적으로 접근할 수 있는 방법 제공



출처: <https://commons.wikimedia.org>

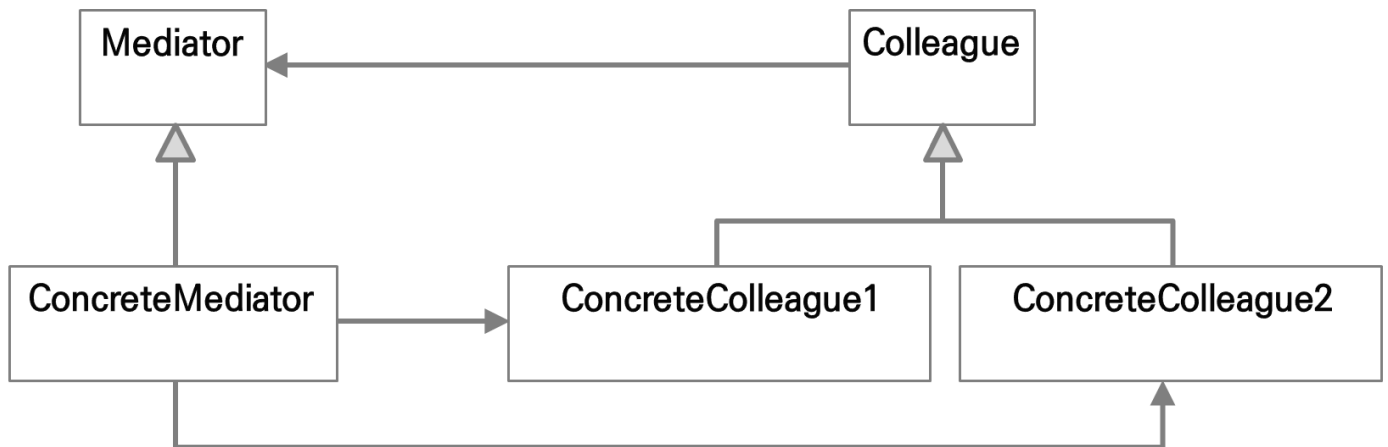
소프트웨어 디자인 패턴

4. 디자인 패턴 분류

3) 행동 패턴

메디에이터(Mediator) 패턴

- 한 집합에 속해있는 객체들의 상호작용을 캡슐화하는 객체를 정의



출처: <https://ko.m.wikipedia.org>

소프트웨어 디자인 패턴

4. 디자인 패턴 분류

3) 행동 패턴

옵저버(Observer) 패턴

- 객체들 사이에 1:N의 의존관계를 정의
- 어떤 객체의 상태가 변할 때, 의존관계에 있는 모든 객체가 통지 받고 자동으로 갱신됨

출처: <https://commons.wikimedia.org>

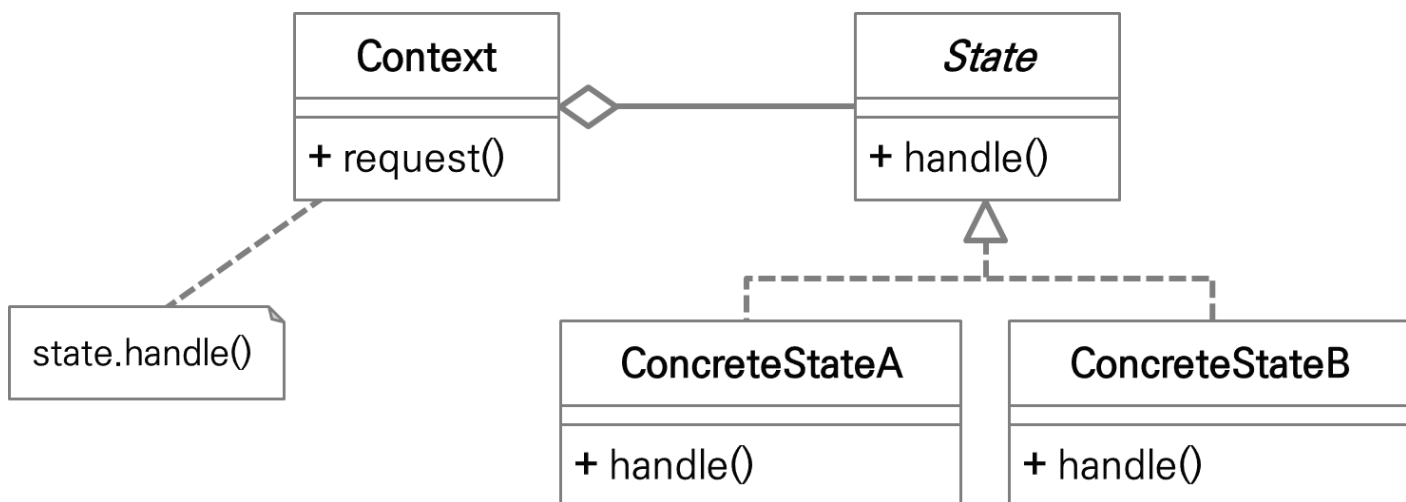
소프트웨어 디자인 패턴

4. 디자인 패턴 분류

3) 행동 패턴

상태(State) 패턴

- 객체의 내부 상태가 변경될 때 행동을 변경하도록 허락
- 객체는 자신의 클래스가 변경되는 것처럼 보이게 됨



출처: <https://ko.m.wikipedia.org>

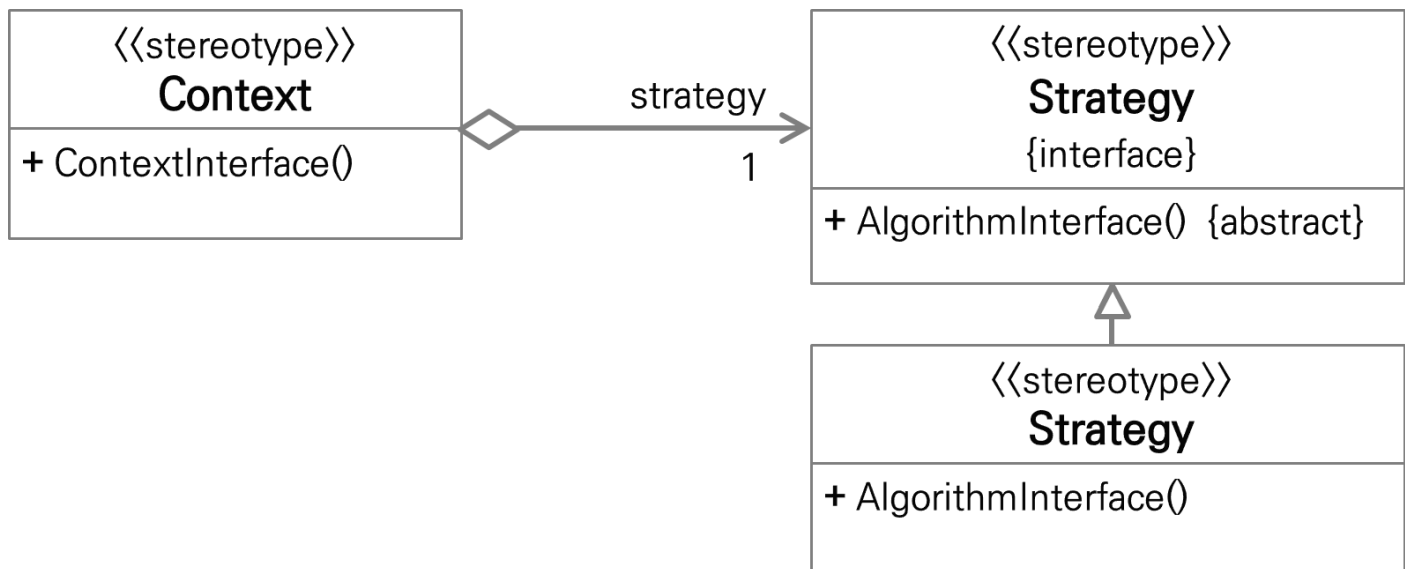
소프트웨어 디자인 패턴

4. 디자인 패턴 분류

3) 행동 패턴

스트레이티지(Strategy) 패턴

- 동일 계열의 알고리즘을 정의
- 각각 캡슐화하며, 이들을 상호교환 가능하도록 함



출처: <https://commons.wikimedia.org>

소프트웨어 디자인 패턴

4. 디자인 패턴 분류

3) 행동 패턴

템플릿(Template) 패턴

- 객체의 연산에서 알고리즘의 뼈대만 정의, 나머지는 서브 클래스에서 이뤄지게 함
- 알고리즘의 구조는 변경하지 않고, 알고리즘의 각 단계를 서브 클래스에서 재정의하게 함

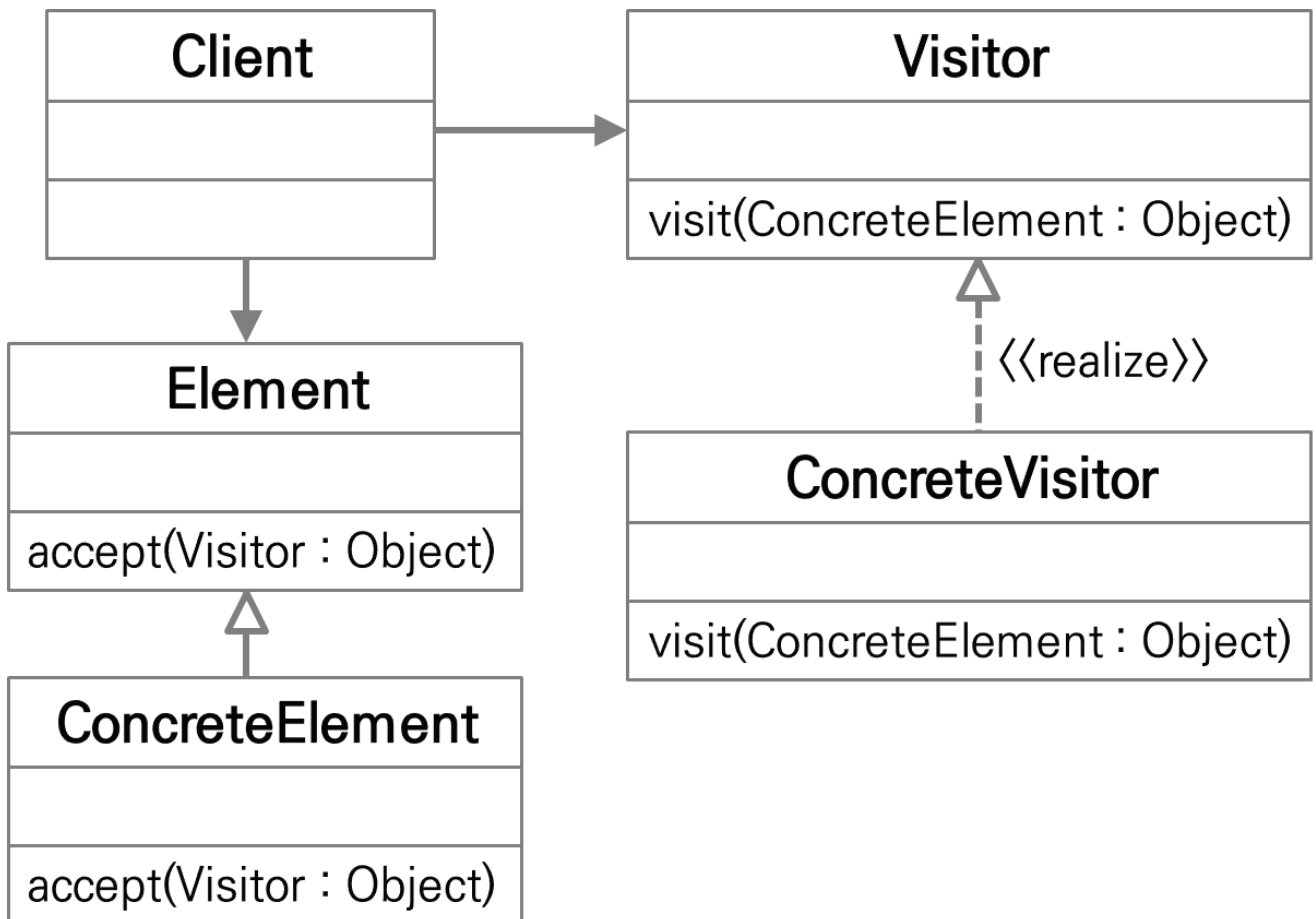
소프트웨어 디자인 패턴

4. 디자인 패턴 분류

3) 행동 패턴

비지터(Visitor) 패턴

- 객체구조를 이루는 원소에 대해 수행할 연산을 표현

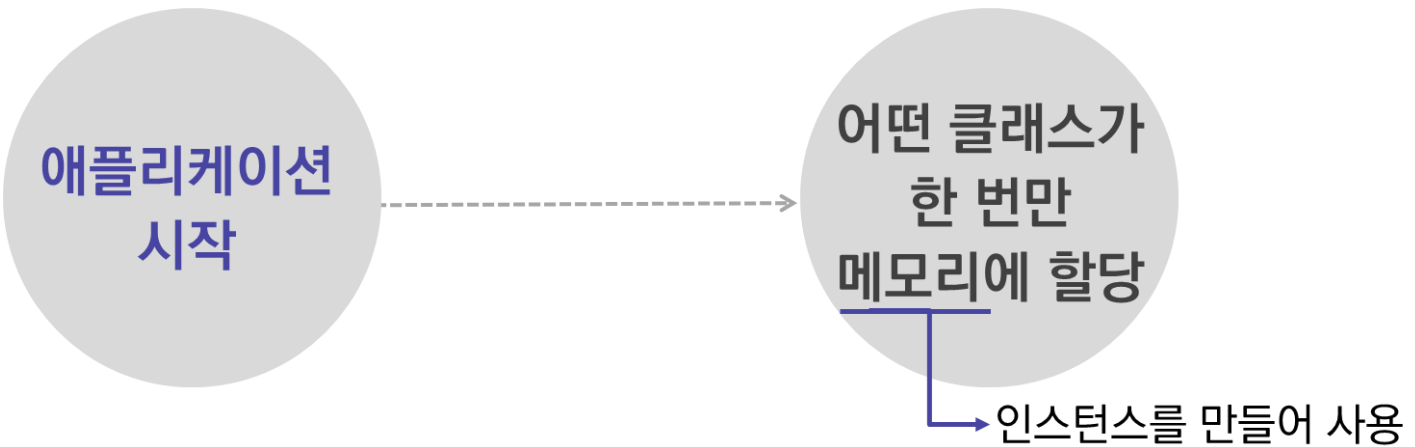


출처: <https://commons.wikimedia.org>

소프트웨어 생성 패턴

1. 싱글톤(Singleton) 패턴

1) 정의 및 개념



- 생성자를 여러 차례 호출되더라도 객체는 하나만 생성되고 최초 생성된 인스턴스만 반환

동일 인스턴스를 재사용 하기 위한 패턴

Database Connection Pool처럼

객체를 여러 개 생성해서 사용하는 상황에서 많이 사용

소프트웨어 생성 패턴

1. 싱글톤(Singleton) 패턴

2) 선언

선언 1

기본적인 선언 : 단일 스레드 환경에서 사용하면 문제 없음



문제점

- 멀티스레딩 환경에서 동시에 접근할 경우 인스턴스가 두 개 생성될 가능성



해결책

- getInstance() 메소드 동기화

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

소프트웨어 생성 패턴

1. 싱글톤(Singleton) 패턴

2) 선언

선언 2

synchronized 키워드를 이용한 멀티스레드 환경 선언



문제점

- synchronized 키워드를 사용한 동기화 → 실행 시간이 느려짐
- 속도가 느려지는 것에 문제가 없다면 사용 가능



해결책

- DCL(Double Checking Locking) 사용을 통해 동기화 영역 축소

```
public class Singleton {
    private static Singleton SingletonInstance;

    private Singleton(){
    }

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

소프트웨어 생성 패턴

1. 싱글톤(Singleton) 패턴

2) 선언

선언 3

DCL을 사용한 선언



문제점

- 멀티코어 환경에서 실행되는 애플리케이션에서 하나의 CPU를 제외하고, 다른 CPU가 lock이 걸리게 됨
- 멀티 코어 환경에서 사용하기 부적합

```
public class Singleton {
    private static Singleton SingletonInstance;

    private Singleton(){
    }

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

소프트웨어 생성 패턴

1. 싱글톤(Singleton) 패턴

2) 선언

선언 4

Eager initialization 방법으로 선언



문제점

- 프로그램이 실행된 후, 처음부터 끝까지 객체가 메모리에 존재
- 인스턴스 사용 X : 계속 존재

Singleton 클래스를 로딩하면서
객체 생성 → 객체 반환

```
public class Singleton {
    private static volatile Singleton SingletonInstance = new Singleton();

    private Singleton(){
    }

    public static Singleton getInstance(){
        return SingletonInstance;
    }
}
```


소프트웨어 생성 패턴

1. 싱글톤(Singleton) 패턴

2) 선언

선언 4

Eager initialization 방법으로 선언



volatile란?

```
public class Singleton {  
    private static volatile Singleton SingletonInstance = new Singleton();  
    private Singleton(){  
    }  
    public static Singleton getInstance(){  
        return SingletonInstance;  
    }  
}
```

- 스레딩 환경에서 동기화 키워드로 컴파일러가 특정 변수에 대한 옵티마이저가 캐싱을 적용하지 못하도록 하는 키워드
- 모든 스레드에 대해 항상 최신의 값을 유지할 수 있게 함

소프트웨어 생성 패턴

1. 싱글톤(Singleton) 패턴

2) 선언

선언 5

중첩 클래스를 이용한 Holder 선언(lazy Initialization 기법)

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

메모리 점유율 면에서 유리, **성능 문제도 없음**

소프트웨어 생성 패턴

1. 싱글톤(Singleton) 패턴

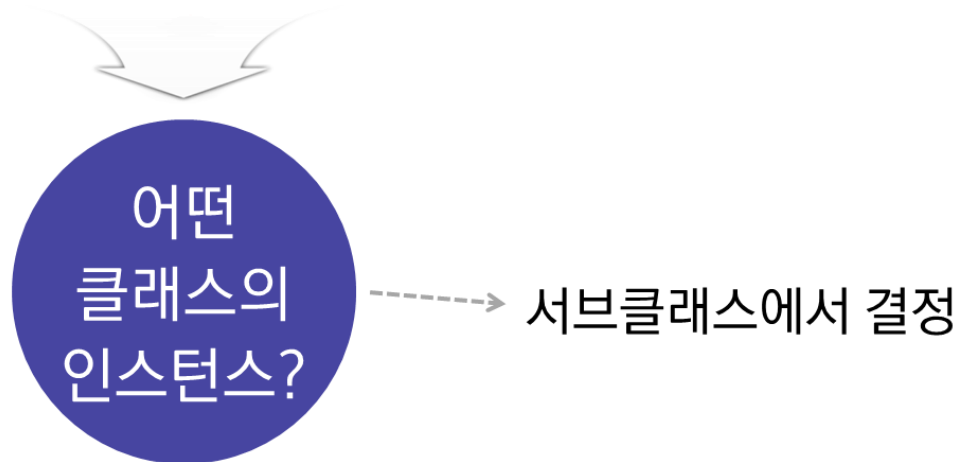
3) 사용법

```
Singleton singletonObj = Sigleton.getInstance()
```

2. 팩토리 메소드(Factory Method) 패턴

1) 정의 및 개념

객체를 생성하기 위한 인터페이스 정의



팩토리 메소드 패턴 이용

→ 서브 클래스에서 클래스 인스턴스 제작

소프트웨어 생성 패턴

2. 팩토리 메소드(Factory Method) 패턴

1) 정의 및 개념

- new 키워드를 호출하는 부분을 서브 클래스에서 하게 됨
- 객체를 만들어 내는 공장을 만드는 패턴
- 팩토리 메소드 패턴은 클래스 간의 결합도를 낮춤

↳ 클래스에 변경 이슈가 생겼을 경우
다른 클래스에 영향을 주는 정도



추상팩토리 패턴이란?

- 인터페이스를 이용하여 서로 연관되거나 의존하는 객체를 구상 클래스로 지정하지 않고 생성

소프트웨어 생성 패턴

2. 팩토리 메소드 (Factory Method) 패턴

2) 선언

[Super Class]

추상 클래스로 Super Class 선언

```
public abstract class Animal {
    public abstract String getName();
    public abstract String getType();

    @Override
    public String toString(){
        return "Animal Type : "+getType()+"\n"+"Animal Name : " + getName()+"\n";
    }
}
```

[Sub Class]

Super Class 를 상속 받는 Sub Class 선언

```
// Sub Class1
public class Dog extends Animal {
    private String type;
    private String name;

    public Dog(String type, String name) {
        this.type = type;
        this.name = name;
    }

    @Override
    public String getType() {
        return this.type;
    }

    @Override
    public String getName() {
        return this.name;
    }
}
```

```
// Sub Class2
public class Cat extends Animal {
    private String type;
    private String name;

    public Cat(String type, String name){
        this.type = type;
        this.name = name;
    }

    @Override
    public String getType() {
        return this.type;
    }

    @Override
    public String getName(){
        return this.name;
    }
}
```

소프트웨어 생성 패턴

2. 팩토리 메소드(Factory Method) 패턴

2) 선언

[Sub Class]

클래스를 선언하는 팩토리 클래스, 클래스 타입으로 해당 클래스를 생성

```
// Factory Class
public class AnimalFactory {
    public static Animal getAnimal(String type){
        if("Dog".equals(type)){
            return new Dog("Dog","Mung");
        }else if("Cat".equals(type)){
            return new Cat("Cat","Navi");
        }else{
            return null;
        }
    }
}
```

3) 사용법

```
// Main test
public class Test {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Animal a = AnimalFactory.getAnimal("Dog");
        Animal b = AnimalFactory.getAnimal("Cat");

        System.out.println(a.toString());
        System.out.println(b.toString());
    }
}
```

[결과]


Animal Type : Dog
Animal Name : Mung

Animal Type : Cat
Animal Name : Navi

소프트웨어 생성 패턴

3. 빌더 (Builder Method) 패턴

1) 정의 및 개념



복잡한
객체를
생성하는
방법

표현하는
방법을
정의하는
클래스

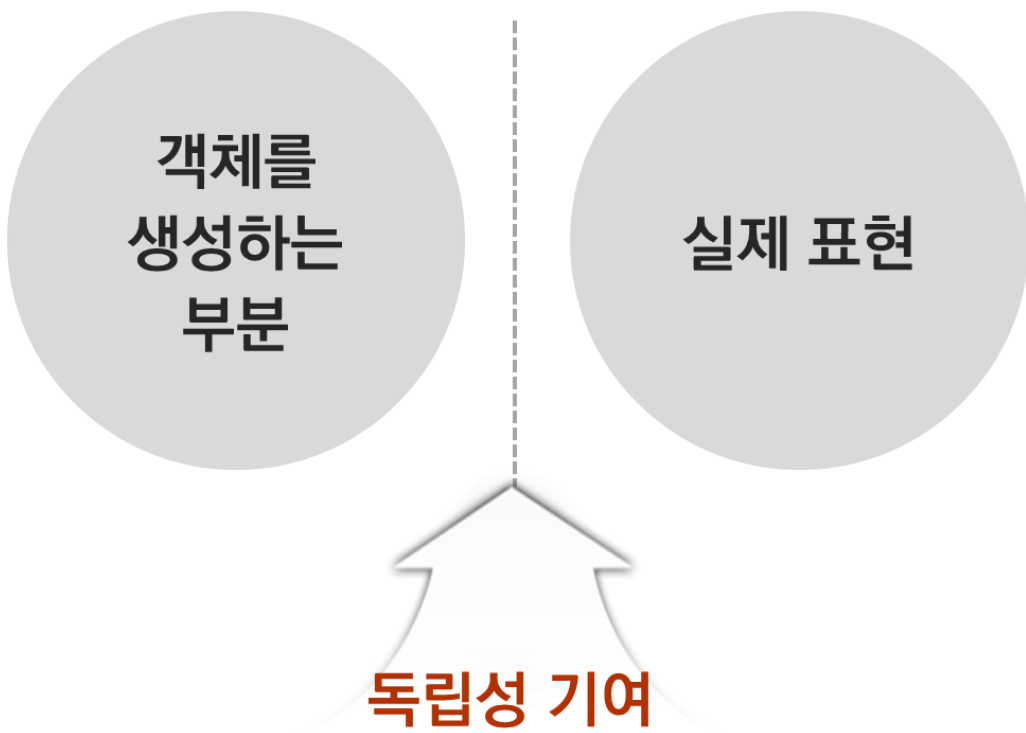
서로 다른 표현이라도
이를 생성할 수 있는 **동일한 구축 공정을 제공**

소프트웨어 생성 패턴

3. 빌더 (Builder Method) 패턴

1) 정의 및 개념

서로 다른 표현방식을 가지는 객체를 동일한 방식으로
생성하고 싶을 경우에 사용



소프트웨어 생성 패턴

3. 빌더(Builder Method) 패턴

1) 정의 및 개념

- 객체를 생성할 때 부분 생성 후 최종 결과를 얻어가는 방식으로 객체 생성과정을 상세히 볼 수 있음
- 객체 생성 추가: 쉬움 /
새로운 객체 구성 추가: 어려움
- 생성되는 객체의 구성을 명확히 하여 추가하거나 수정해야 하는 부분이 없어야 함

소프트웨어 생성 패턴

3. 빌더 (Builder Method) 패턴

2) 선언

빌더 클래스 선언

1/3

- class 안에 중첩 static class 생성
- 바깥쪽 class의 argument들을 안쪽 static class (builder class)로 옮김
- builder class의 생성자를 public static으로 선언
- 필요한 파라미터 요청

```
public class Rect {
    private final int width;
    private final int height;
    private final int margin;
    private final int padding;

    public static class Builder {
        private final int width;
        private final int height;
        private int margin;
        private int padding;

        public Builder(int width, int height){
            this.width = width;
            this.height = height;
        }

        public Builder margin(int margin){
            this.margin = margin;
            return this;
        }
        public Builder padding(int padding){
            this.padding = padding;
            return this;
        }
        public Rect build(){
            return new Rect(this);
        }
    }

    private Rect(Builder builder){
        width = builder.width;
        height = builder.height;
        margin = builder.margin;
        padding = builder.padding;
    }
}
```

소프트웨어 생성 패턴

3. 빌더 (Builder Method) 패턴

2) 선언

빌더 클래스 선언

2/3

- builder class에는 선택적 파라미터에 대한 setter method 포함
- 선택적 인자를 설정한 후, 같은 builder object를 리턴

```
public class Rect {  
    private final int width;  
    private final int height;  
    private final int margin;  
    private final int padding;  
  
    public static class Builder {  
        private final int width;  
        private final int height;  
        private int margin;  
        private int padding;  
  
        public Builder(int width, int height){  
            this.width = width;  
            this.height = height;  
        }  
  
        public Builder margin(int margin){  
            this.margin = margin;  
            return this;  
        }  
        public Builder padding(int padding){  
            this.padding = padding;  
            return this;  
        }  
        public Rect build(){  
            return new Rect(this);  
        }  
    }  
  
    private Rect(Builder builder){  
        width = builder.width;  
        height = builder.height;  
        margin = builder.margin;  
        padding = builder.padding;  
    }  
}
```

소프트웨어 생성 패턴

3. 빌더 (Builder Method) 패턴

2) 선언

빌더 클래스 선언

3/3

- 클라이언트 프로그램이 요청하는 object를 받을 수 있도록 build method 제작
- build method에서는 바깥쪽 class의 생성자가 builder 클래스의 인자를 받을 수 있도록 제공

```
public class Rect {
    private final int width;
    private final int height;
    private final int margin;
    private final int padding;

    public static class Builder {
        private final int width;
        private final int height;
        private int margin;
        private int padding;

        public Builder(int width, int height){
            this.width = width;
            this.height = height;
        }

        public Builder margin(int margin){
            this.margin = margin;
            return this;
        }
        public Builder padding(int padding){
            this.padding = padding;
            return this;
        }
        public Rect build(){
            return new Rect(this);
        }
    }

    private Rect(Builder builder){
        width = builder.width;
        height = builder.height;
        margin = builder.margin;
        padding = builder.padding;
    }
}
```

소프트웨어 생성 패턴

3. 빌더 (Builder Method) 패턴

3) 사용법

```
public class BuilderPattern {  
    public static void main(String []args){  
        Rect rect = new Rect.Builder(200,100)  
            .margin(5)  
            .padding(10)  
            .build();  
        System.out.println(rect.toString());  
    }  
}
```

[결과]

width : 200
height : 100
margine : 5
padding : 10

핵심정리

1. 소프트웨어 디자인 패턴

- 소프트웨어 디자인에서 특정 문맥, 상황에서 공통으로 발생하는 문제에 대해 재사용 가능한 해결책임
- 유지 보수하기 쉬운 객체지향 시스템을 만들 수 있음
- 알고리즘과 같이 프로그램 코드로 바로 변환될 수 있는 형태는 아니지만, 특정 상황에서 구조적인 문제를 해결하는 방식을 설명함
- 업무 논의 및 디자인 문서를 작성할 때 상호 간에 의사 결정 용어로 쓰일 수 있음
- 디자인 패턴은 목적에 따라서 소프트웨어 생성 패턴, 구조 패턴, 행위 패턴 3가지로 분류됨
- 소프트웨어 생성 패턴에는 싱글톤, 빌더, 팩토리 메서드, 추상 팩토리, 원형 패턴이 있음
- 구조 패턴에는 적응자, 브리지, 컴포지트, 데코레이터, 퍼사드, 프록시 패턴이 있음
- 행위 패턴에는 커맨드, 인터프리터, 이터레이터, 메디에이터, 옵저버, 상태, 스트레이트지, 템플릿, 비지터 패턴이 있음

핵심정리

2. 소프트웨어 생성 패턴

- 싱글톤 패턴은 애플리케이션이 시작될 때 어떤 클래스가 한 번만 메모리에 할당하고 그 메모리에 인스턴스를 만들어서 사용하며, 최초 생성된 인스턴스를 재사용하여 사용하기 위한 패턴 임
- 팩토리 메소드 패턴은 객체를 생성하기 위한 인터페이스를 정의하는데 어떤 클래스의 인스턴스를 만들지는 서브 클래스에서 결정하는 패턴 임
- 빌더 패턴은 복잡한 객체를 생성하는 방법과 표현하는 방법을 정의하는 클래스를 별도로 분리하여 서로 다른 표현이라도 이를 생성할 수 있는 동일한 구축 공정을 제공할 수 있도록 함