디자인 패턴

한국기술교육대학교
온라인평생교육원

학습내용

- 디자인 패턴의 개요
- 디자인 패턴의 유형 (1) 생성 패턴
- 디자인 패턴의 유형 (2) 구조 패턴
- 디자인 패턴의 유형 (3) 행위 패턴

학습목표

- 디자인 패턴의 개념, 요소, 원칙을 설명할 수 있다.
- 생성 패턴을 활용하여 공통 모듈 설계를 할 수 있다.
- 구조 패턴을 활용하여 공통 모듈 설계를 할 수 있다.
- 행위 패턴을 활용하여 공통 모듈 설계를 할 수 있다.

디자인 패턴의 개요

디자인 패턴의 개념

- 소프트웨어 설계에 있어 공통된 문제들에 대한 표준적인 해법
- 반복적으로 나타나는 문제들을 해결해 온 전문가들의 경험을 모아서 정리한 일관된 솔루션
- 프로그래머들이 유용하다고 생각되는 객체들의 일반적인 상호 작용 방법들을 모은 목록

디자인 패턴의 개요

디자인 패턴의 4요소

패턴 이름 (Pattern Name)	문제 (Problem)
해법 (Solution)	결과 (Consequence)

패턴 이름(Pattern Name)

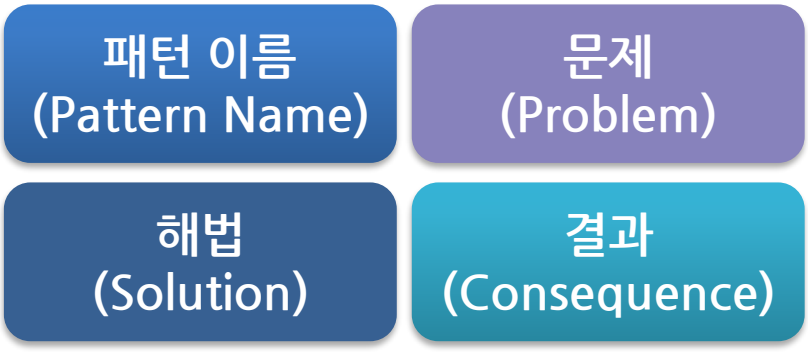
- 한두 단어로 설계 문제와 해법 서술
- 설계의 의도 표현
- 설계에 대한 생각을 쉽게 하고, 개발자들 간의 의사소통이 원활해 짐

문제(Problem)

- 언제 패턴을 사용하는가를 서술
- 해결할 문제와 그 배경을 설명
- '어떤 알고리즘을 객체로 만들까' 같은 문제 설명

디자인 패턴의 개요

디자인 패턴의 4요소



해법(Solution)

- 설계를 구성하는 요소들과 그 요소들 간의 관계, 책임, 협력관계 서술
- 구체적이지 않은 추상적인 설명

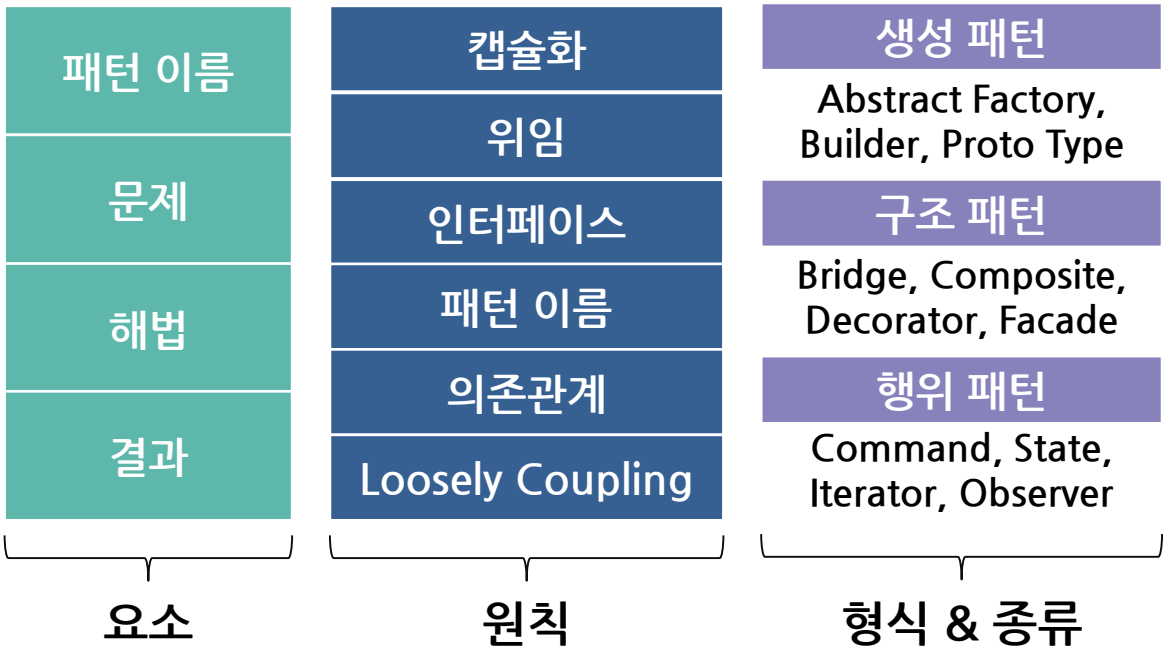
결과(Consequence)

- 디자인 패턴을 적용해서 얻는 결과와 장단점 서술
- 시스템의 유연성, 확장성, 이식성에 영향을 줌
- 나중에 패턴들을 이해하거나 평가하는 데 도움을 줌

디자인 패턴의 개요

디자인 패턴의 원칙

원칙	내용
캡슐화	• 바뀌는 부분은 캡슐화
위임	• 상속보다는 위임을 활용
인터페이스	• 구현이 아닌 인터페이스에 맞춰서 프로그래밍
Loosely Coupling	• 서로 상호작용을 하는 객체 사이에서는 가능하면 느슨하게 결합하는 디자인 사용
개방 & 폐쇄	• 클래스 확장에 대해서는 OPEN, 변경에 대해서는 CLOSE
의존관계	• 추상화된 클래스에 의존하고 구현 클래스 의존은 배제



디자인 패턴의 개요

디자인 패턴의 종류

생성 패턴

구조 패턴

행위 패턴

디자인 패턴의 개요

디자인 패턴의 종류

생성 패턴

의미		• 객체 생성 방식 결정 • 클래스의 정의, 객체생성방식의 구조화, 캡슐화
범위	클래스	• Factory method : 인스턴스화 될 객체의 서브클래스
	객체	• Abstract Factory : 제품객체군 • Builder : 복합 객체 생성 • Proto Type : 인스턴스화 될 객체 클래스 • Singleton : 인스턴스가 하나일 때

디자인 패턴의 개요

디자인 패턴의 종류

구조 패턴

의미		• 객체를 조직화 하는 일반적인 방법 제시 • 별도의 클래스를 통합하는데 유용하며 런타임 시에 객체가 구성된 구조를 변경 가능 • 객체 구성에 유동성, 확장성 추가 가능
범위	클래스	• Adaptor : 객체 인터페이스
	객체	• Bridge : 객체 구현 • Composite : 객체의 합성과 구조 • Decorator : 서브클래싱 없이 객체의 책임성 • Façade : 서브시스템에 대한 인터페이스 • Flyweight : 객체 저장 비용 • Proxy : 객체 접근방법

디자인 패턴의 개요

디자인 패턴의 종류

행위 패턴

의미		• 객체의 행위를 조직화 관리, 연합 • 객체나 클래스의 연동에 대한 유형을 제시하고자 할 때 사용 • 런타임 시 복잡한 제어 흐름을 결정짓는 데 사용
범위	클래스	• Interpreter : 언어의 문법과 해석방법 • Template Method : 알고리즘 단계
	객체	• Chain of Responsibility : 요청처리객체 • Command : 요청처리 시점과 방법 • Iterator : 집합객체 요소 접근 / 순회방법 • Mediator : 객체 상호작용 • Memento : 객체 정보 외부저장 • Observer : 종속 객체 상태 변경 • State : 객체상태 • Strategy : 알고리즘 • Visitor : 클래스 변경 없이 객체에 적용할 수 있는 오퍼레이션

디자인 패턴의 개요

디자인 패턴, 아키텍처, 프레임워크 비교

	디자인패턴	아키텍처	프레임워크
목적	• 실제 구현 과정에서 해결방안으로 제시	• 전체 시스템의 구조나 설계 모형 재사용	• 시스템 구현효율성 향상과 개발편의성 제공
범위	• 모든 구현 과정 중 일관된 문제에 적용	• 모든 종류의 시스템에 적용 가능	• 하나의 시스템을 구축하기 위한 틀
효과	• 유지보수의 용의성, 이해하기 쉬운 코드로 단순화 가능 • 문제에 대한 패턴을 인식하여, 일치하는 디자인 패턴을 찾아 프로젝트에 적용 • 어플리케이션의 확장성 향상, 비즈니스 요구사항 변화에 적시 대응 • 어떤 문제점에 대한 해결안을 찾는 시간과 개발 기간 단축	• 구체적인 코드와 함께 요구변경이나 다른 제약 조건 등에 대한 대응 방안이 다름 • 구현 경험 이해로 바람직한 아키텍처 설계 가능 • 구조도로서의 이해와 실제 설계작업 중 각 구현상 특징별 구체적인 코딩방안까지 설계할 수 있어야 함	• Class Library로 프로젝트의 일부를 갖고 있음 • 정해진 설계 방식에 따라 실제 코드가 일부 작성, 프레임워크 클래스를 서브 클래싱 해서 유저 코드 작성 • 프레임워크 코드가 유저코드 호출 방식 • 실행 흐름을 프레임워크가 제어 • 객체의 연동은 구조 프레임워크가 정의

디자인 패턴의 유형 (1) 생성 패턴

생성 패턴의 종류

- Abstract Factory 패턴
- Builder 패턴
- Singleton 패턴

디자인 패턴의 유형 (1) 생성 패턴

Abstract Factory 패턴

정의

- 구체적인 클래스를 지정하지 않고 관련성을 갖는 객체들의 집합을 생성하거나 서로 독립적인 객체들의 집합을 생성할 수 있는 인터페이스를 제공하는 패턴

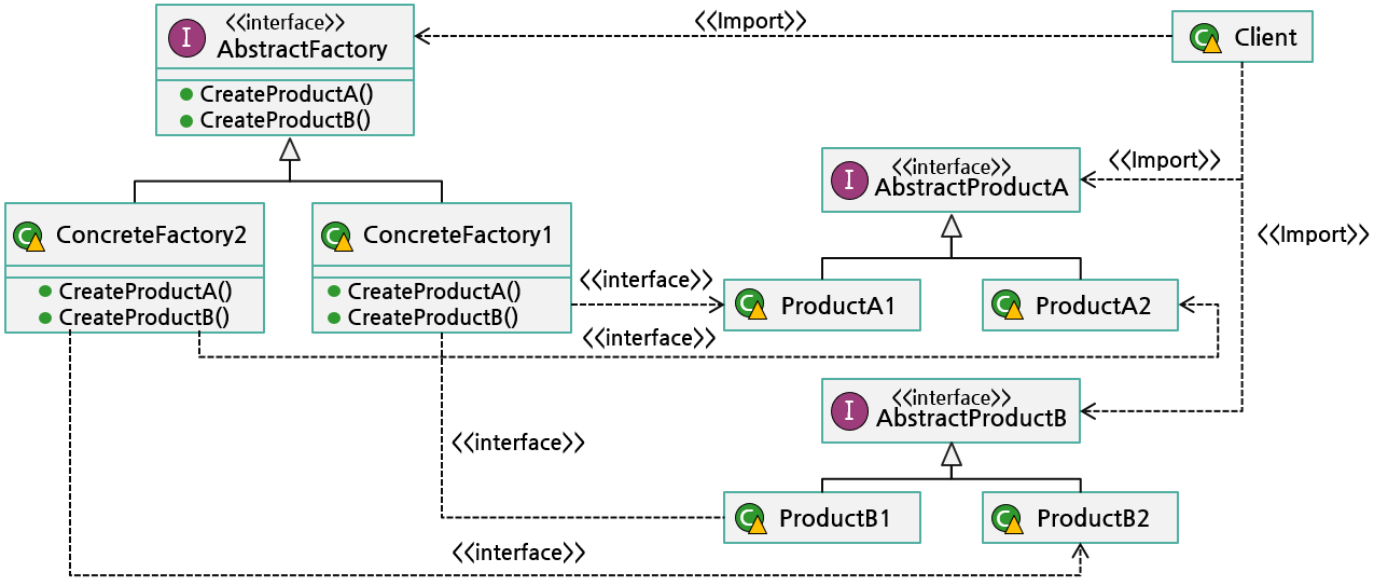
객체

객체명	설명	예시 (자동차)
Abstract Factory	▪ 개념적 제품에 대한 객체를 생성하는 연산으로 인터페이스를 정의	회사
Concrete Factory	▪ 구체적인 제품에 대한 객체를 생성하는 연산을 구현	자동차 생산공장
Abstract Product	▪ 개념적 제품 객체에 대한 인터페이스를 정의	트럭, 승용차
Concrete Product	▪ 구체적으로 팩토리가 생성할 객체를 정의하고, Abstract Product가 정의하는 인터페이스를 구현	SUV, 세단
Client	▪ Abstract Factory와 Abstract Product 클래스에 선언된 인터페이스를 사용	운전자

디자인 패턴의 유형 (1) 생성 패턴

Abstract Factory 패턴

객체



디자인 패턴의 유형 (1) 생성 패턴

Abstract Factory 패턴

사용

클라이언트가 객체 생성에 독립적으로 설계

객체 생성을 클라이언트가 직접 하는 것이 아니라,
간접적으로 수행함으로써 클라이언트가
객체의 생성이나 구성 또는 표현 방식에
독립적이도록 할 때

제품군 선택을 용이하게 디자인

여러 제품군 중 사용할 제품군을
쉽게 선택할 수 있도록 만들고 싶을 때

디자인 패턴의 유형 (1) 생성 패턴

Abstract Factory 패턴

사용

여러 제품군에 속한 객체 생성 디자인

서로 관련된 제품들이 하나의 제품군을 형성하고,
이런 제품군이 여러 개 존재하는 상황에서
생성되는 제품 객체가 항상 같은 제품군에
속하는 것을 확실히 보장받고 싶을 때

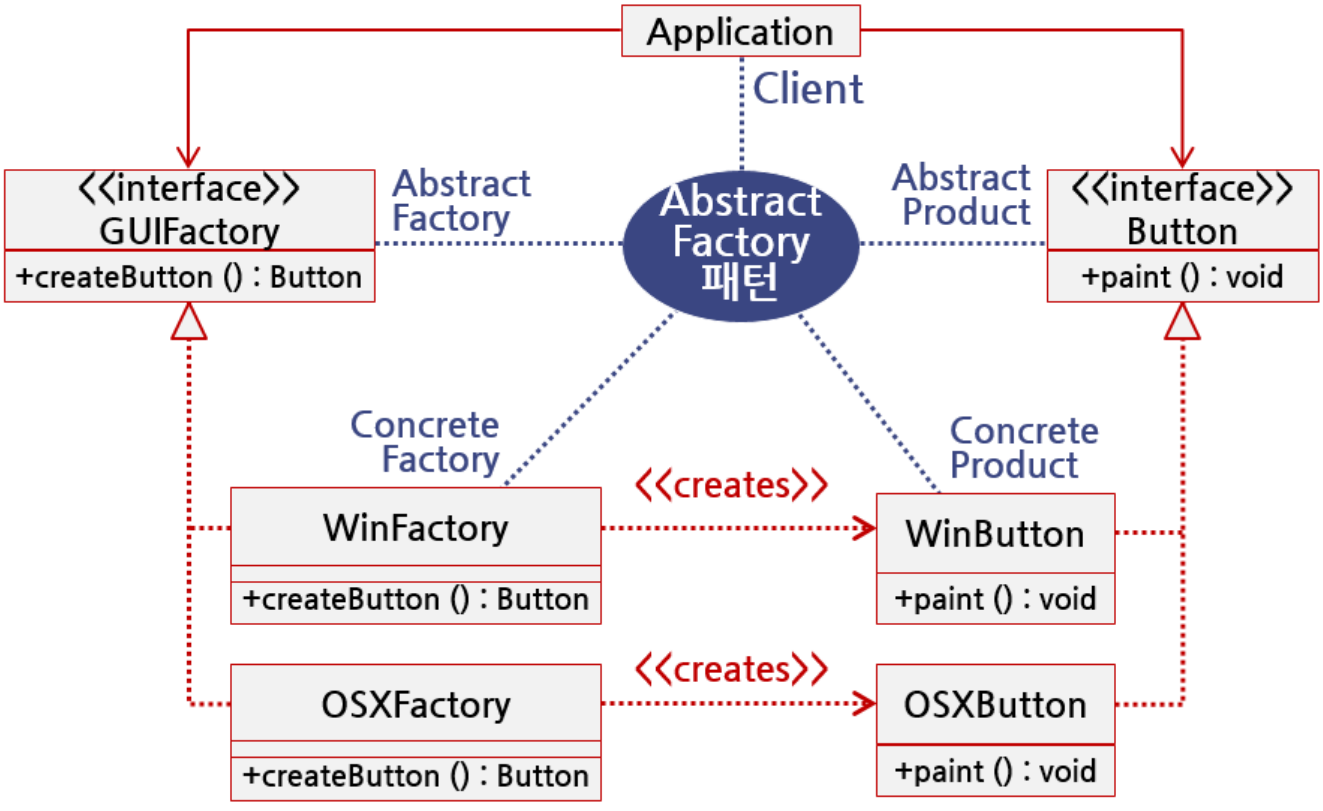
인터페이스만 드러난 객체 구현

제품들에 대한 클래스 라이브러리를
만들어야 하는데, 그 인터페이스만 드러내고
구현은 숨기고 싶을 때

디자인 패턴의 유형 (1) 생성 패턴

Abstract Factory 패턴

사용



디자인 패턴의 유형 (1) 생성 패턴

Builder 패턴

정의

- 복잡한 객체를 생성하는 방법과 표현하는 방법을 정의하는 클래스를 별도로 분리해 서로 다른 표현이라도 이를 생성할 수 있는 동일한 절차를 제공
- 제품을 여러 단계로 나눠서 만들 수 있도록 제품 생산 단계를 캡슐화

디자인 패턴의 유형 (1) 생성 패턴

Builder 패턴

구분

Builder (Abstract Builder)

Concrete Builder

Director

Product

Builder (Abstract Builder)

- 인스턴스를 생성하기 위한 인터페이스(API)를 결정
- Builder 클래스에는 인스턴스의 각 부분을 만들기 위한 메소드가 준비

Concrete Builder

- Builder 클래스의 인터페이스(API)를 구현
- 인스턴스의 각 부분을 만드는 부분, 조립하는 부분, 최종적으로 인스턴스를 생산하는 부분을 모두 구현함

디자인 패턴의 유형 (1) 생성 패턴

Builder 패턴

구분

Builder (Abstract Builder)

Concrete Builder

Director

Product

Director

- Builder 인터페이스(API)를 사용해 원하는 부품을 조립 후 인스턴스를 생산

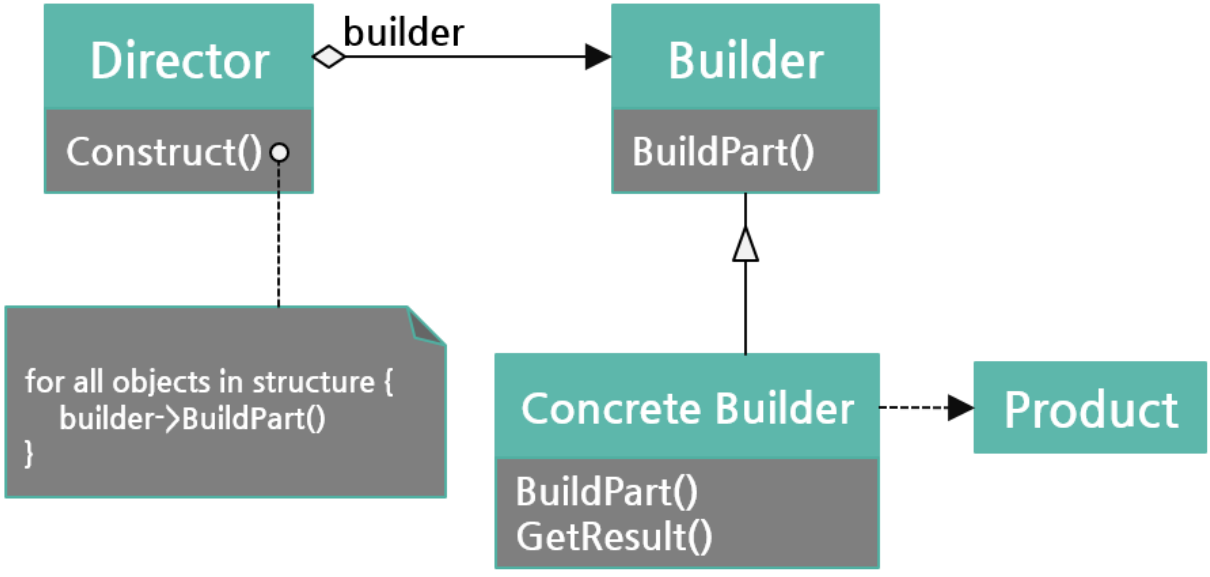
Product

- Builder에 의해 최종적으로 생산된 제품으로 Director가 사용

디자인 패턴의 유형 (1) 생성 패턴

Builder 패턴

구분

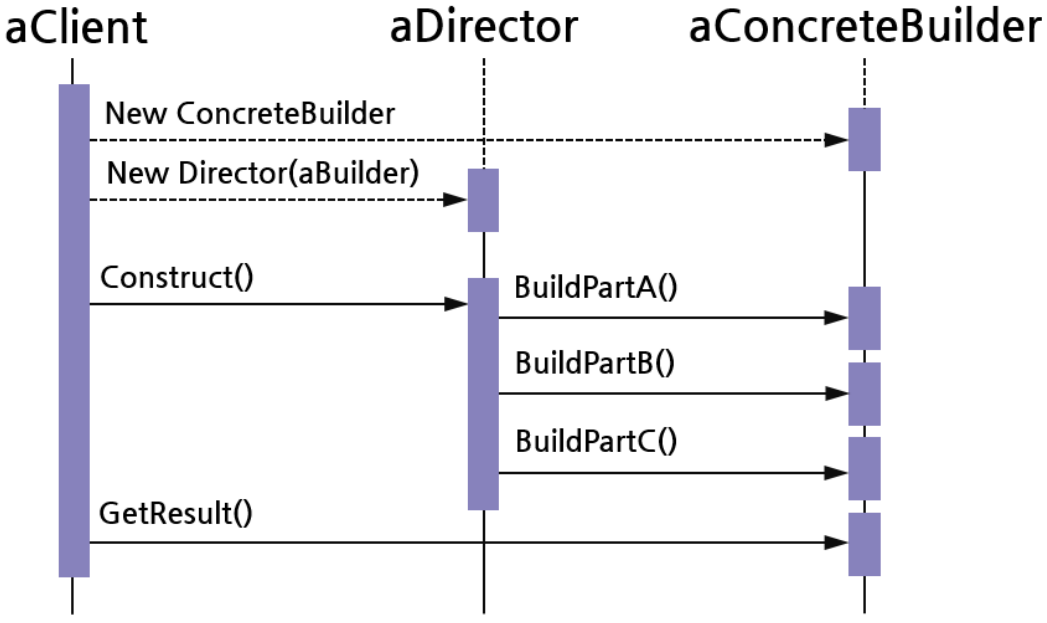


디자인 패턴의 유형 (1) 생성 패턴

Builder 패턴

사용

- 복합 객체의 생성 알고리즘과 이를 합성하는 요소 객체들 간의 조립 방법이 서로 독립적일 때 사용




```
class Builder {
    private final int DEFAULT_VALUE = -1;
    private int a = DEFAULT_VALUE;
    private int b = DEFAULT_VALUE;
    private int c = DEFAULT_VALUE;
    public Builder a(int v) {
        this.a = v;
        return this;
    }
    public Builder b(int v) {
        this.b = v;
        return this;
    }
    public Builder c(int v) {
        this.c = v;
        return this;
    }
    public Foo build() {
        return new Foo(a, b, c);
    }
}
```

```
class Foo {
    private int a;
    private int b;
    private int c;
    public Foo(int a, int b, int c) {
        this.a = a;
        this.b = b;
        this.c = c;
    }
    public String toString() {
        return "a=" + a + ", b=" + b + ", c=" + c;
    }
}

public class BuilderApp {
    public static void main(String[] args) {
        Foo f = new Builder().a(1).c(2).build();
        System.out.println(f.toString());
    }
}
```

디자인 패턴의 유형 (1) 생성 패턴

Singleton 패턴

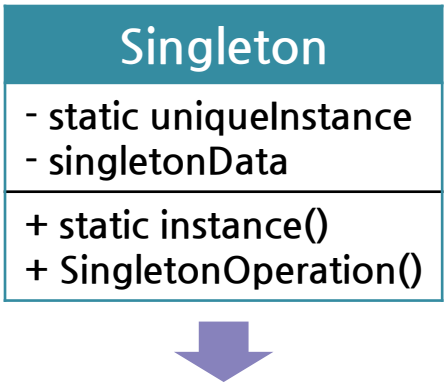
정의

- 지정한 클래스의 인스턴스가 반드시 1개만 존재하도록 하는 패턴
- Singleton의 인스턴스는 getInstance()메소드를 통해서만 생성할 수 있도록 하여, 하나의 인스턴스만을 생성하도록 제어함

디자인 패턴의 유형 (1) 생성 패턴

Singleton 패턴

정의



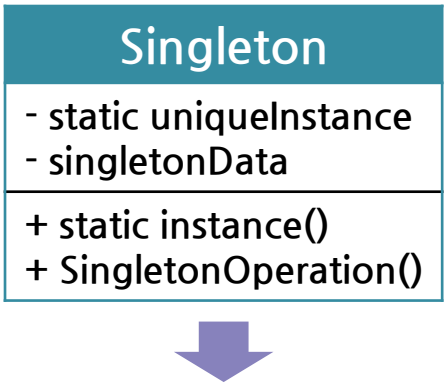
필요 시 인스턴스를 생성(생성 속도가 문제가 되지 않을 경우 사용)

```
public class Singleton {
private static Singleton uniqueInstance ;
// 생성자 private
private Singleton ()   { }
public static synchronized Singleton getInstance() {
                // 필요한 상황이 닥치기 전에는 생성하지 않는 Lazy
Instantiation
                if (uniqueInstance == null) {
                                uniqueInstance = new Singleton () ;
                }
                return uniqueInstance ;
}
// 기타 메소드
}
```

디자인 패턴의 유형 (1) 생성 패턴

Singleton 패턴

정의



인스턴스를 실행 중에 수시로 만들고 관리하기 불편할 경우
처음부터 인스턴스를 생성

```
public class Singleton {
private static SingletonuniqueInstance  = new Singleton() ;
private Singleton ()   { }
public static Singleton getInstance() {
                return uniqueInstance ;
}
}
```

디자인 패턴의 유형 (1) 생성 패턴

Singleton 패턴

사용

- 클래스의 인스턴스가 오직 하나이어야 함을 보장하고, 잘 정의된 접근 방식에 의해 모든 클라이언트가 접근할 수 있도록 해야 할 때
- 시스템에서 유일한 인스턴스를 가져야 하는 공통 모듈 설계에 활용

디자인 패턴의 유형 (2) 구조 패턴

구조 패턴의 종류

- Adaptor 패턴
- Façade 패턴

디자인 패턴의 유형 (2) 구조 패턴

Adaptor 패턴

정의

- 클래스의 재사용성을 높이기 위해 요구되는 특정 기능으로 변환·적용하여 클래스 간 호환성을 확보하는 패턴
- Wrapper 패턴이라고도 함
- 이미 제공되어 있는 것을 그대로 사용할 수 없는 경우, 이미 제공되어 있는 것과 필요한 것 사이의 간격을 메우는 패턴

디자인 패턴의 유형 (2) 구조 패턴

Adaptor 패턴

종류

- 상속(Inheritance)을 이용한 Adaptor 패턴
- 위임(Delegation)을 이용한 Adaptor 패턴

디자인 패턴의 유형 (2) 구조 패턴

Adaptor 패턴

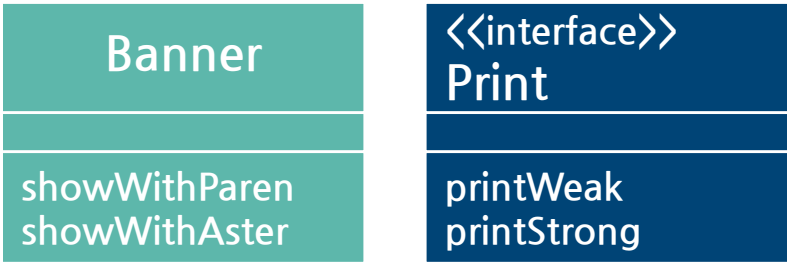
사례

Banner ➡ 이미 제공되어 있는 것

- showWithParen → 문자열 앞뒤에 괄호를 쳐서 표시하는 메소드
- showWithAster → 문자열 앞뒤에 '*'를 붙여서 표시하는 메소드

<<interface>> Print ➡ 필요로 하는 것

- printWeak → 문자열을 약하게 표시(괄호를 붙임)
- printStrong → 문자열을 강하게 표시('*' 붙임)



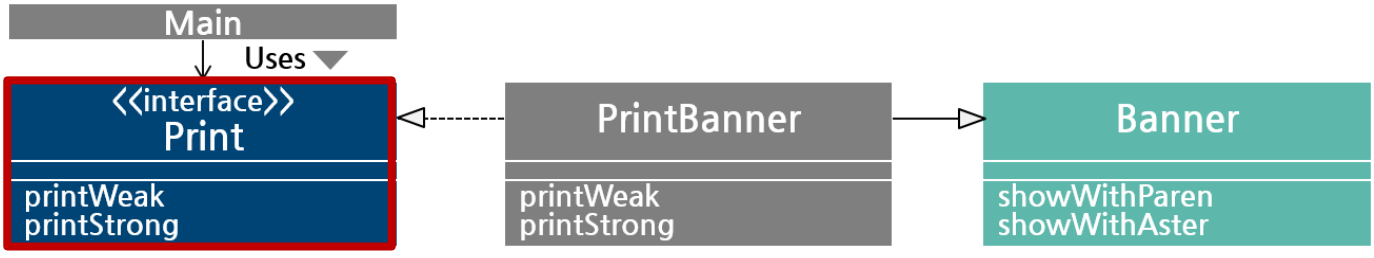
목표

- Banner 클래스라는 기존의 클래스를 이용해서, Print 인터페이스를 충족시키는(구현하는) 클래스 만들기

디자인 패턴의 유형 (2) 구조 패턴

Adaptor 패턴

상속을 이용한 것

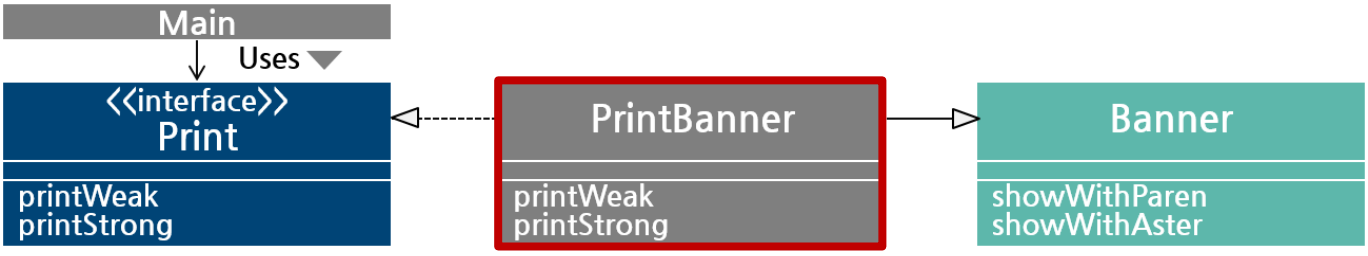



```
public interface Print {
    public abstract void printWeak();
    public abstract void printStrong();
}
```

디자인 패턴의 유형 (2) 구조 패턴

Adaptor 패턴

상속을 이용한 것

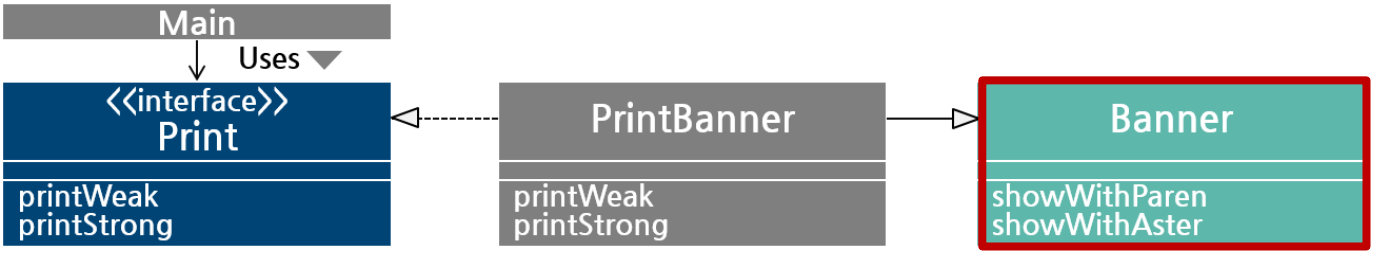



```
public class PrintBanner extends Banner implements Print {
    public PrintBanner(String string) {
        super(string);
    }
    public void printWeak() {
        showWithParen();
    }
    public void printStrong() {
        showWithAster();
    }
}
```

디자인 패턴의 유형 (2) 구조 패턴

Adaptor 패턴

상속을 이용한 것

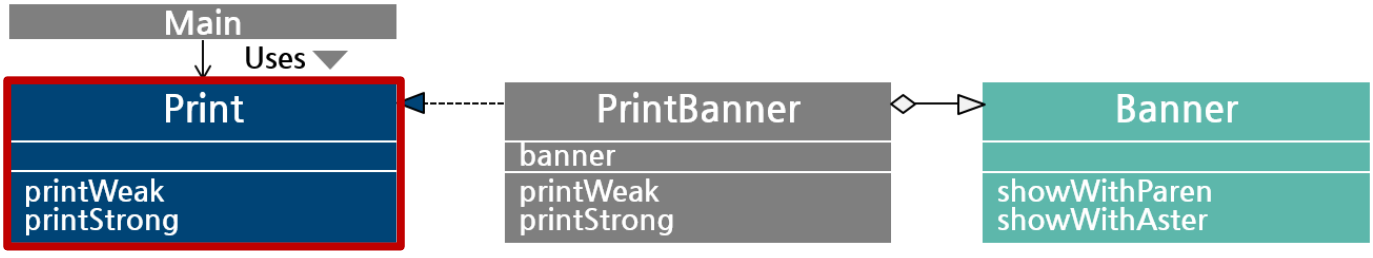



```
public class Banner {
    private String string;
    public Banner(String string) {
        this.string = string;
    }
    public void showWithParen() {
        System.out.println("(" + string + ")");
    }
    public void showWithAster() {
        System.out.println("*" + string + "*");
    }
}
```

디자인 패턴의 유형 (2) 구조 패턴

Adaptor 패턴

위임을 이용한 것

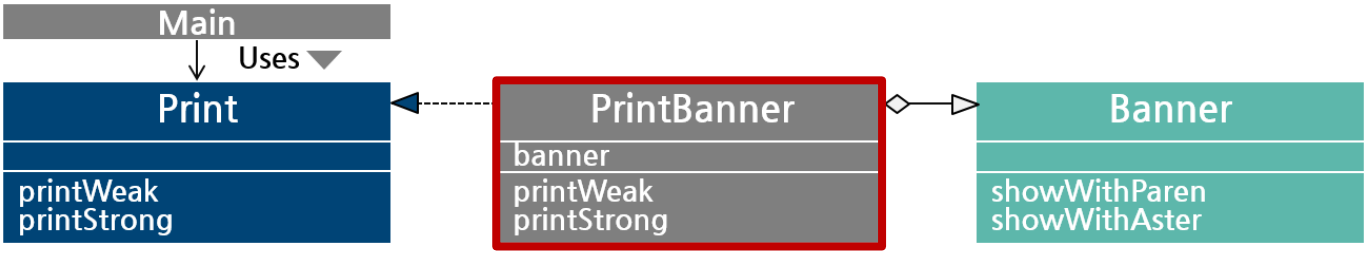



```
public interface Print {
    public abstract void printWeak();
    public abstract void printStrong();
}
```

디자인 패턴의 유형 (2) 구조 패턴

Adaptor 패턴

위임을 이용한 것

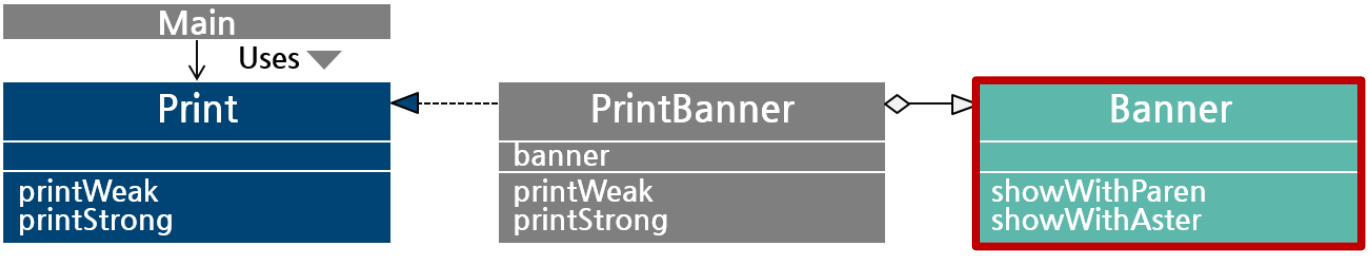



```
public class PrintBanner extends Print {
    private Banner banner;
    public PrintBanner(String string) {
        this.banner = new Banner(string);
    }
    public void printWeak() {
        banner.showWithParen();
    }
    public void printStrong() {
        banner.showWithAster();
    }
}
```

디자인 패턴의 유형 (2) 구조 패턴

Adaptor 패턴

위임을 이용한 것



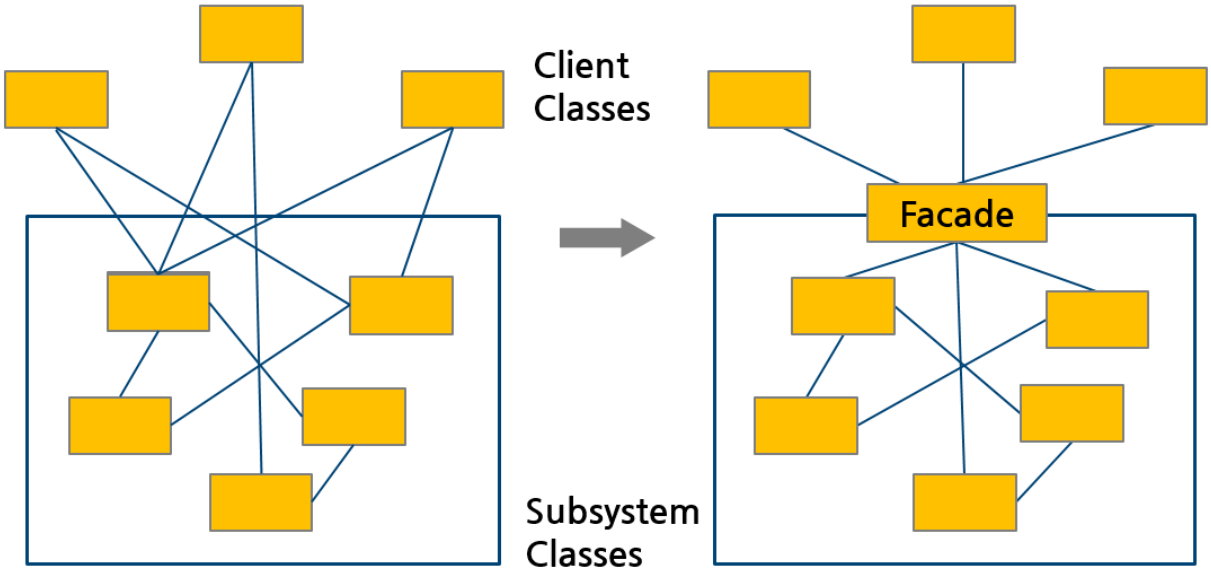

```
public class Banner {
    private String string;
    public Banner(String string) {
        this.string = string;
    }
    public void showWithParen() {
        System.out.println("(" + string + ")");
    }
    public void showWithAster() {
        System.out.println("*" + string + "*");
    }
}
```

디자인 패턴의 유형 (2) 구조 패턴

Adaptor 패턴

정의

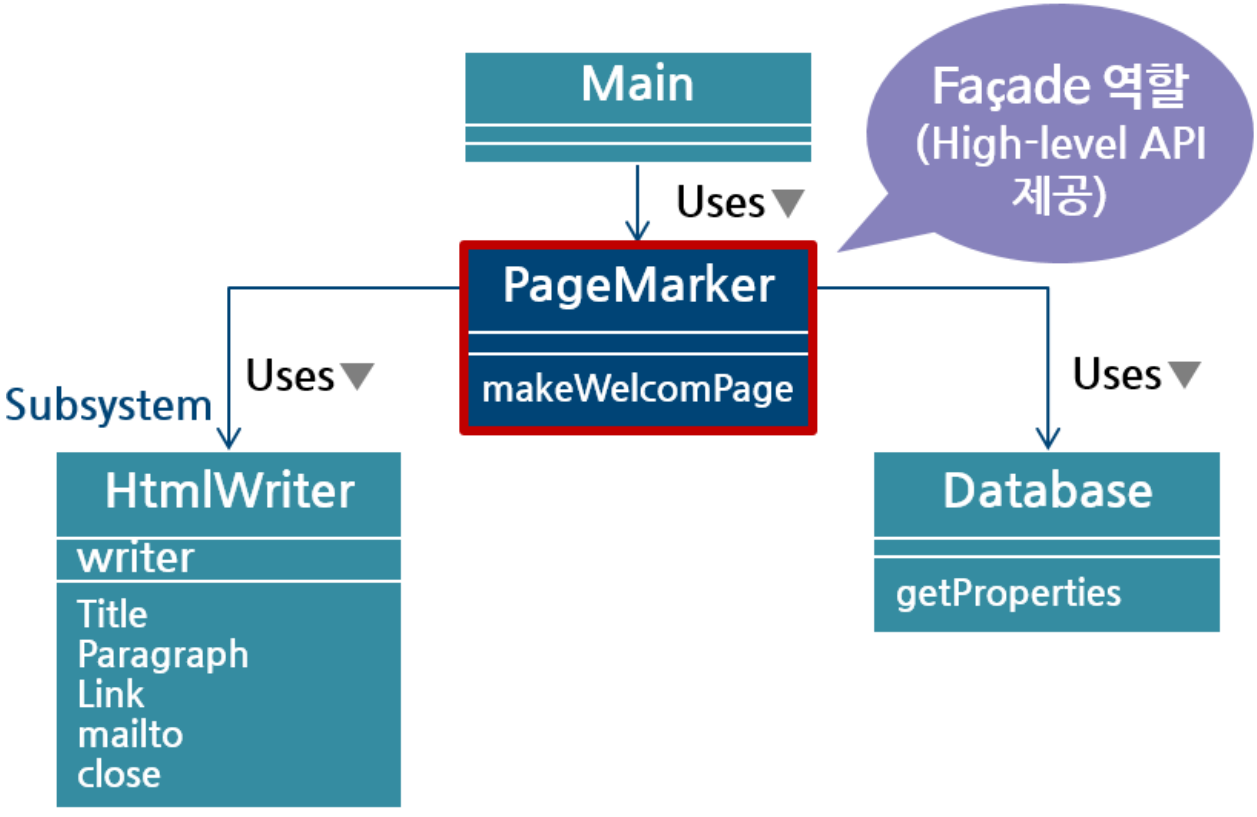
- 복잡한 내부 구조를 보이지 않으면서 외부와의 일관된 인터페이스를 제공하는 패턴



디자인 패턴의 유형 (2) 구조 패턴

Adaptor 패턴

클래스 다이어그램(사례)



디자인 패턴의 유형 (3) 행위 패턴

행위 패턴의 종류

- Chain of Responsibility
- Command 패턴

디자인 패턴의 유형 (3) 행위 패턴

Chain of Responsibility(책임 패턴)

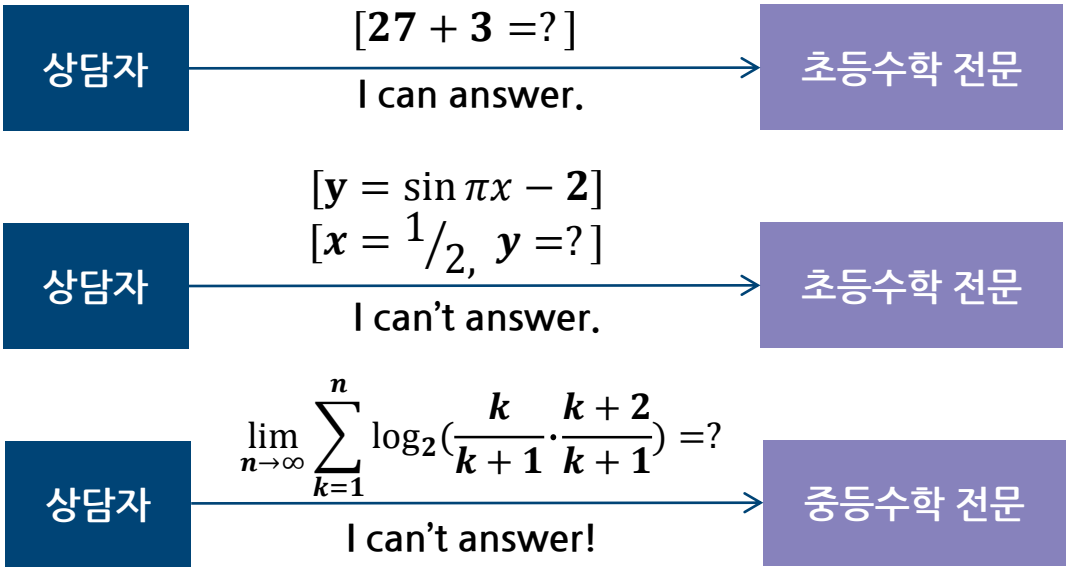
정의

- 요청을 처리할 수 있는 기회를 하나 이상의 객체에 부여함으로써 객체 사이의 결합도를 없애는 패턴
- 요청을 해결할 객체를 만날 때까지 객체 고리를 따라서 요청을 전달하는 패턴

디자인 패턴의 유형 (3) 행위 패턴

Chain of Responsibility(책임 패턴)

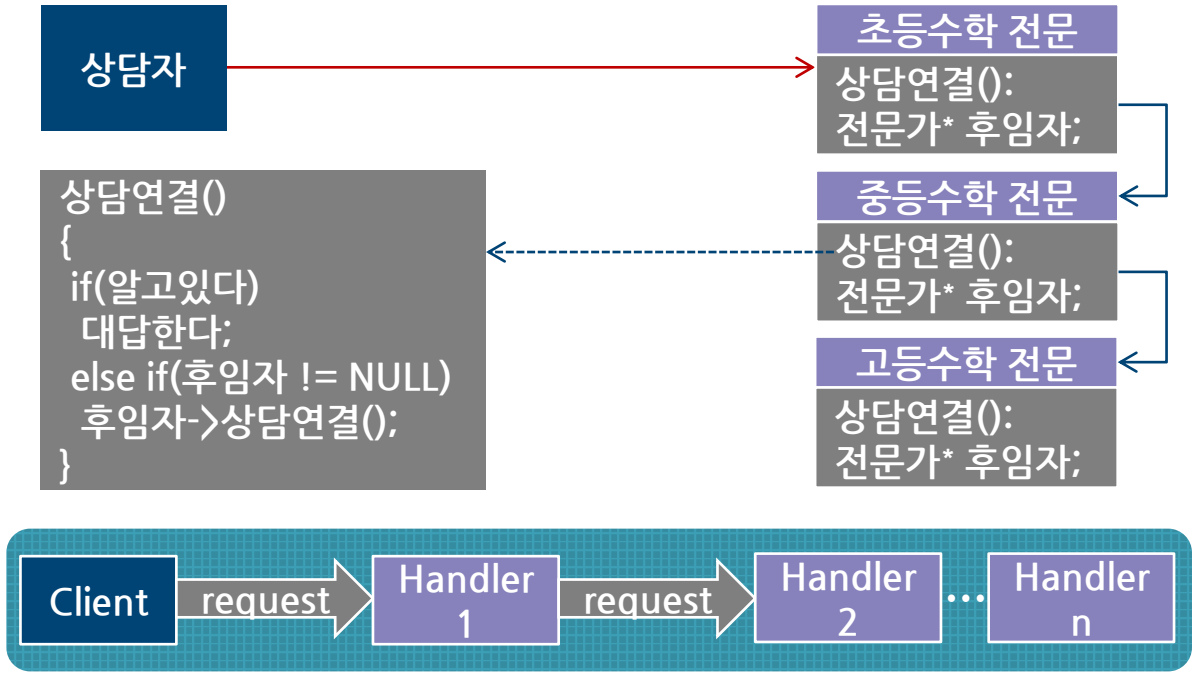
사례 - 책임 패턴 사용 전



디자인 패턴의 유형 (3) 행위 패턴

Chain of Responsibility(책임 패턴)

사례 - 책임 패턴 사용 후

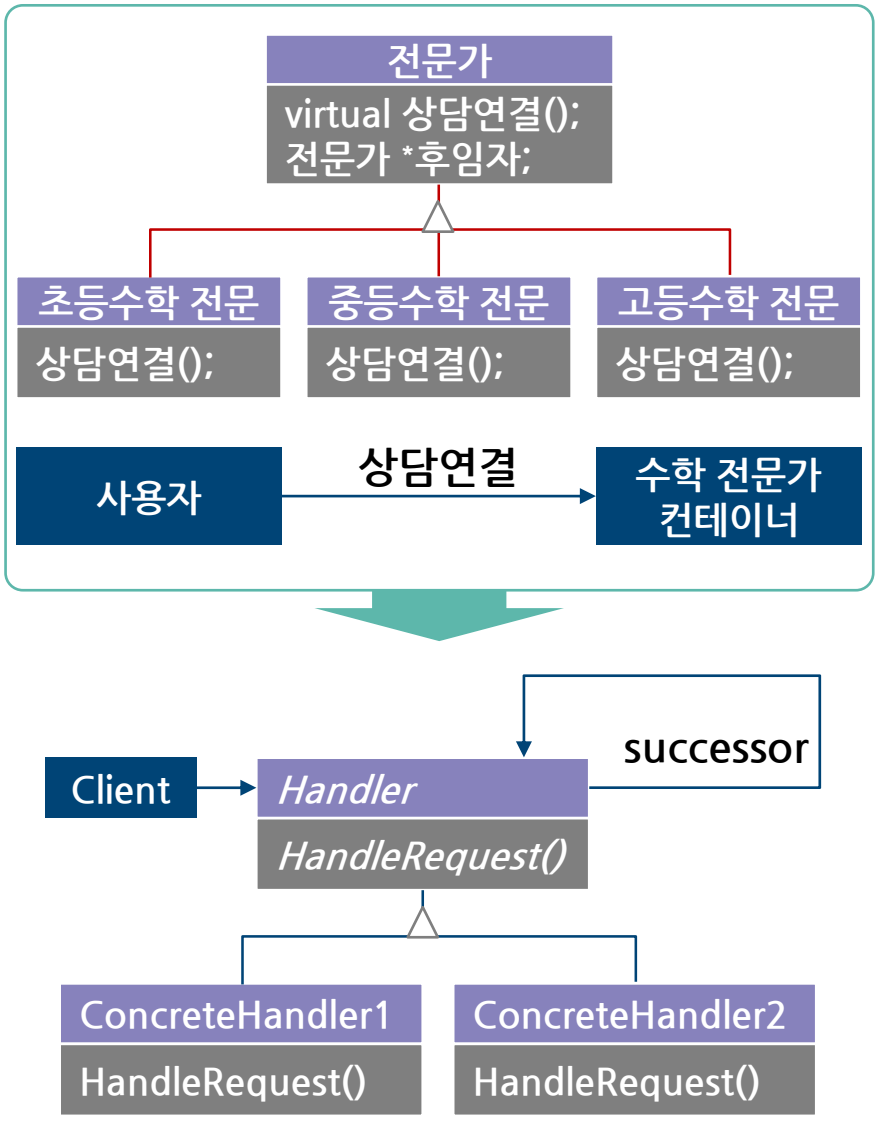


디자인 패턴의 유형 (3) 행위 패턴

Chain of Responsibility(책임 패턴)

사용

- 하나 이상 객체의 요청을 처리해야 하는 경우, 핸들러가 누가 선행자인지 모를 때 사용
- 유기적으로 연결된 여러 객체들 중에 하나의 객체에 요청을 전달할 때 사용

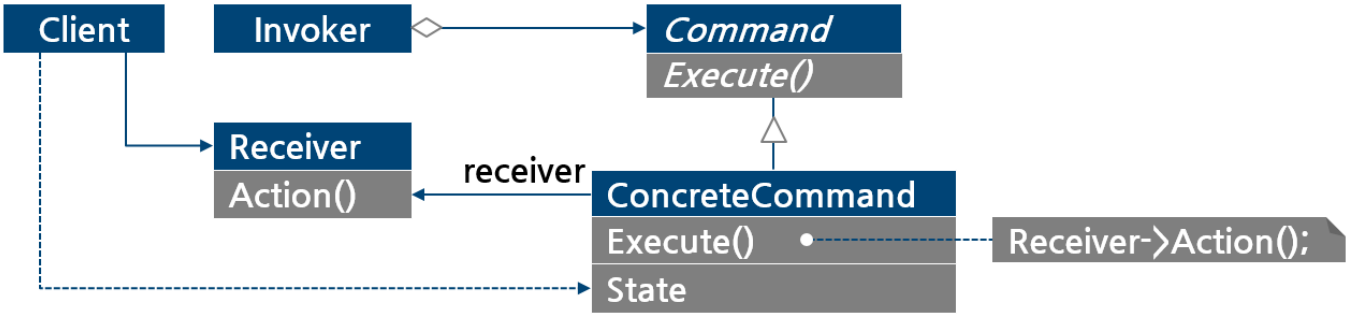


디자인 패턴의 유형 (3) 행위 패턴

Command 패턴

정의

- 동작이나 트랜잭션에 대한 요청을 객체를 통해 캡슐화 함으로써 해당 요청을 저장하거나 로그에 기록함
- 실행 취소(Undo), 재실행(Redo) 등의 기능을 제공하고자하는 설계 패턴
- 요청을 파라미터화 하여 Log, Undo, Redo 기능을 갖도록 정의



디자인 패턴의 유형 (3) 행위 패턴

Command 패턴

사례

간단한 그림 그리기 프로그램

마우스를 Drag 하면 빨간 점이 연결되어 그림이 그려진다.

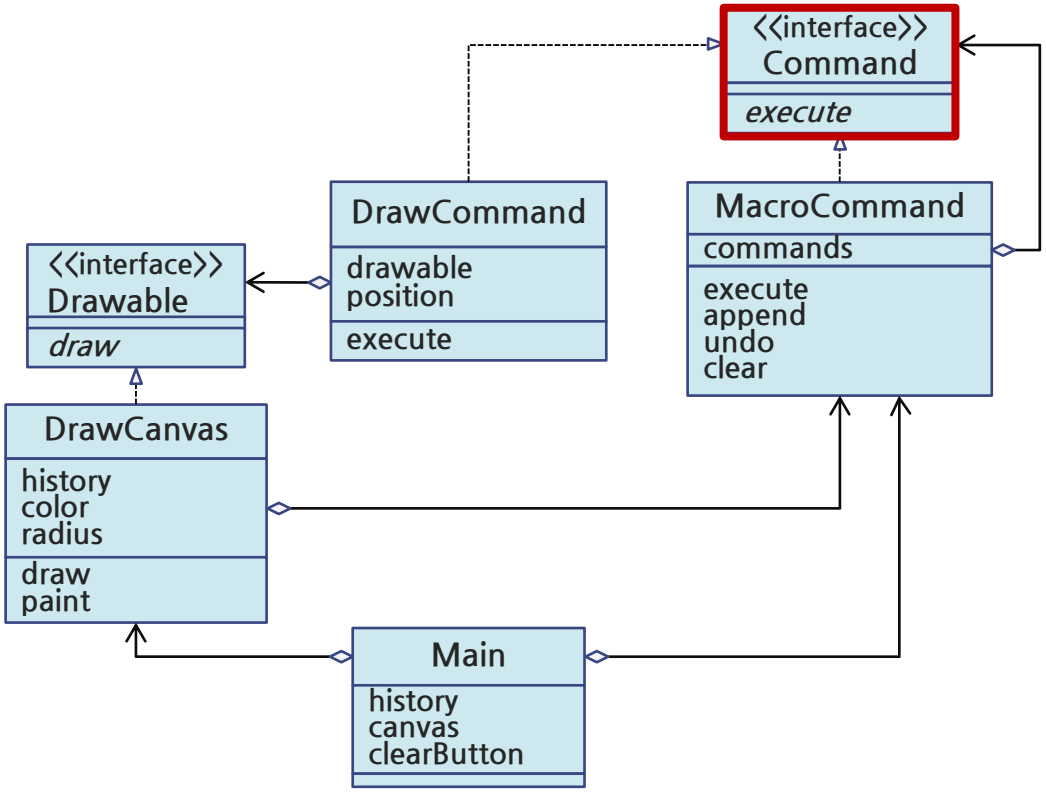
'Clear' 버튼을 누르면 점이 모두 지워진다.

사용자가 마우스를 끌 때마다, '이 위치에 점을 그려라'라는 명령이 DrawCommand 클래스로 전송된다.

디자인 패턴의 유형 (3) 행위 패턴

Command 패턴

구분



Command

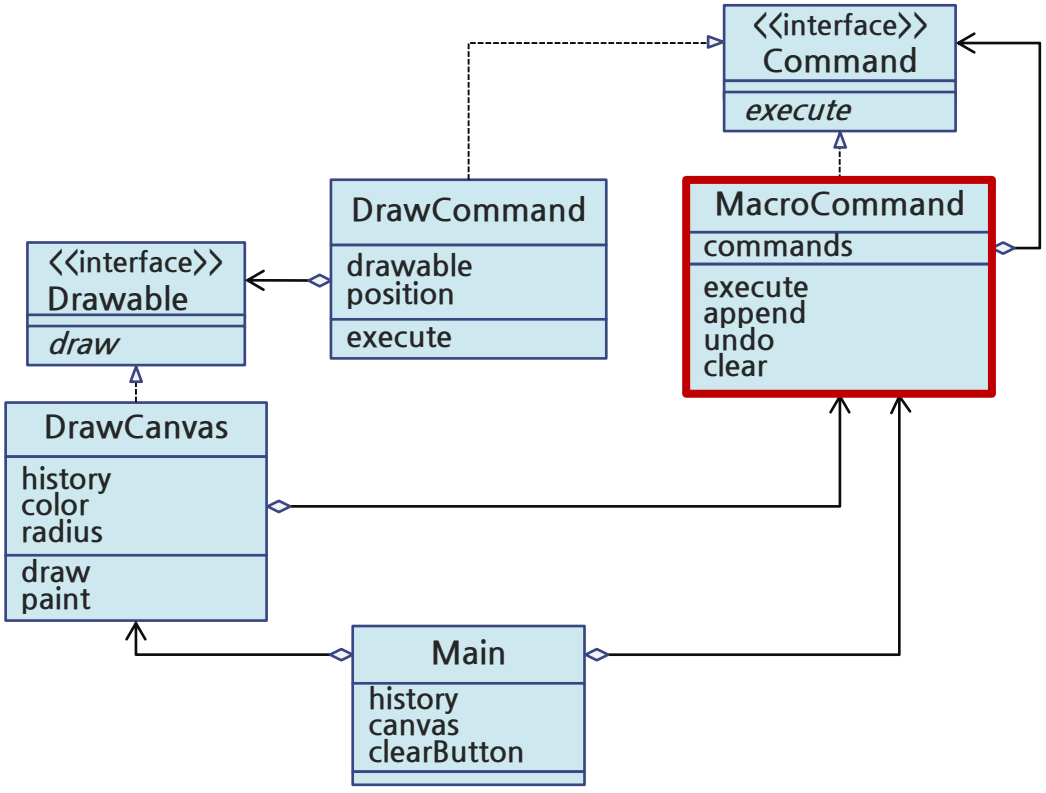
'명령'을 표현하는 인터페이스

- execute() :
  무언가를 실행하는 메소드

디자인 패턴의 유형 (3) 행위 패턴

Command 패턴

구분



MacroCommand

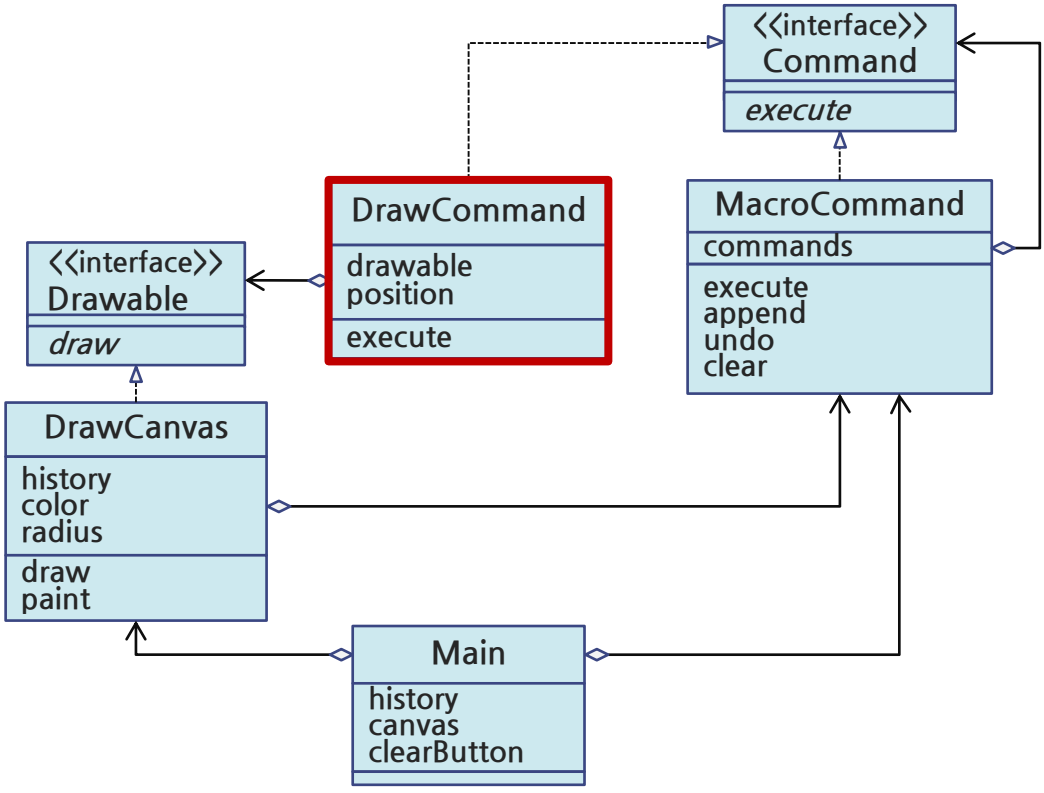
'여러 개의 명령을 모은 명령'을 나타내는 클래스

- commands : java.utilStack형으로 다수의 command를 모아둠
- execute() : 자신이 가지고 있는 모든 명령의 execute()을 호출
- append() : MacroCommand 클래스에 새로운 Command를 추가
- undo() : commands의 마지막 명령을 삭제하는 메소드
- clear() : commands의 모든 명령을 삭제하는 메소드

디자인 패턴의 유형 (3) 행위 패턴

Command 패턴

구분



DrawCommand

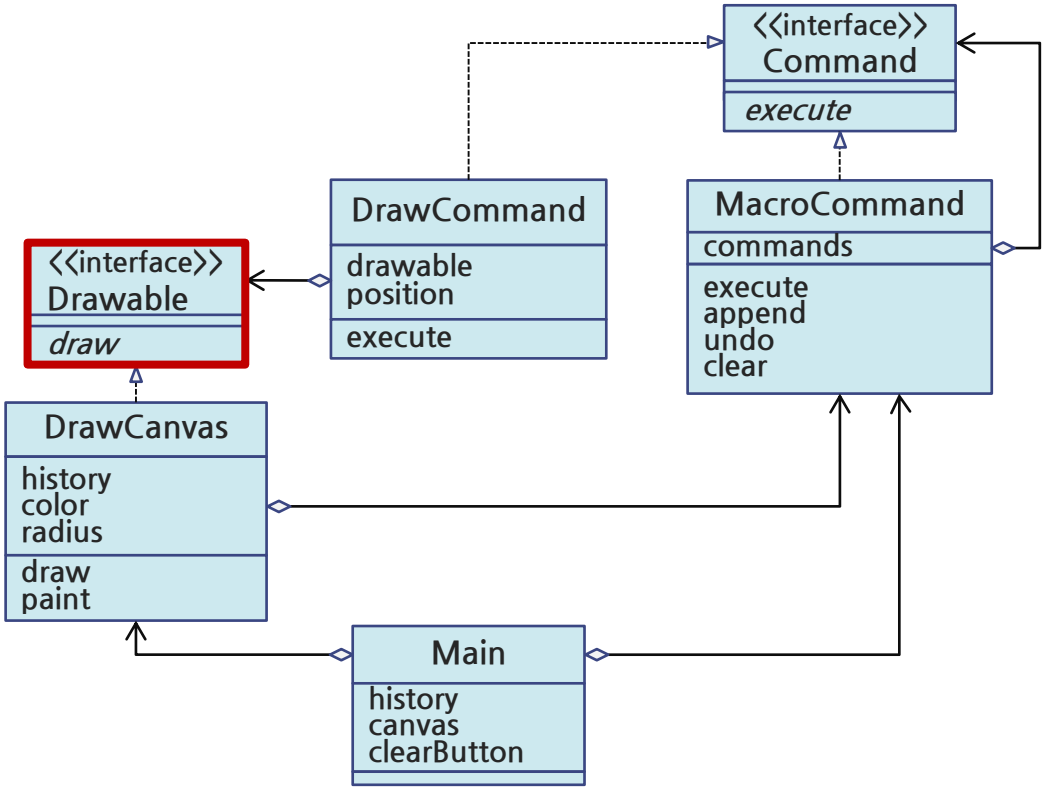
'그림 그리기 명령'을 표현한 클래스

- drawable :
  그림 그리기를 실행할 대상(객체)를 저장함
- position :
  그림 그리기를 행할 위치를 나타냄

디자인 패턴의 유형 (3) 행위 패턴

Command 패턴

구분



Drawable

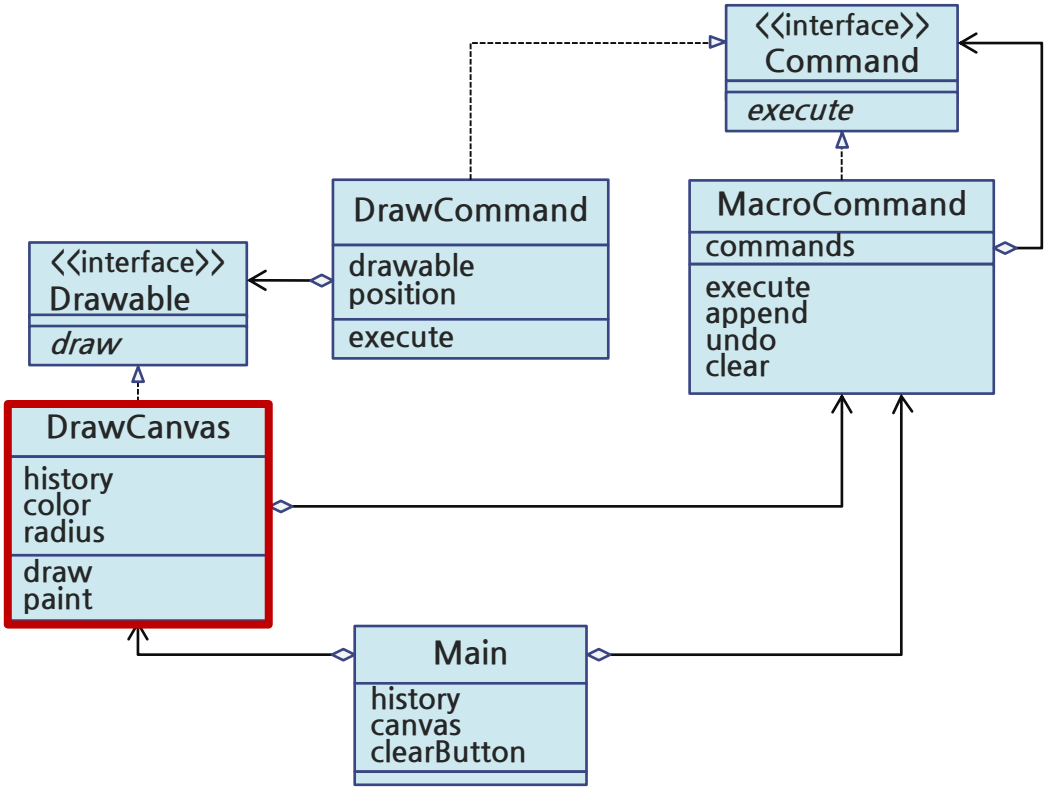
'그리기 대상'을 표현한 인터페이스

- draw(int x, int y) :
  그림 그릴 좌표(x, y)를 나타냄

디자인 패턴의 유형 (3) 행위 패턴

Command 패턴

구분



DrawCanvas

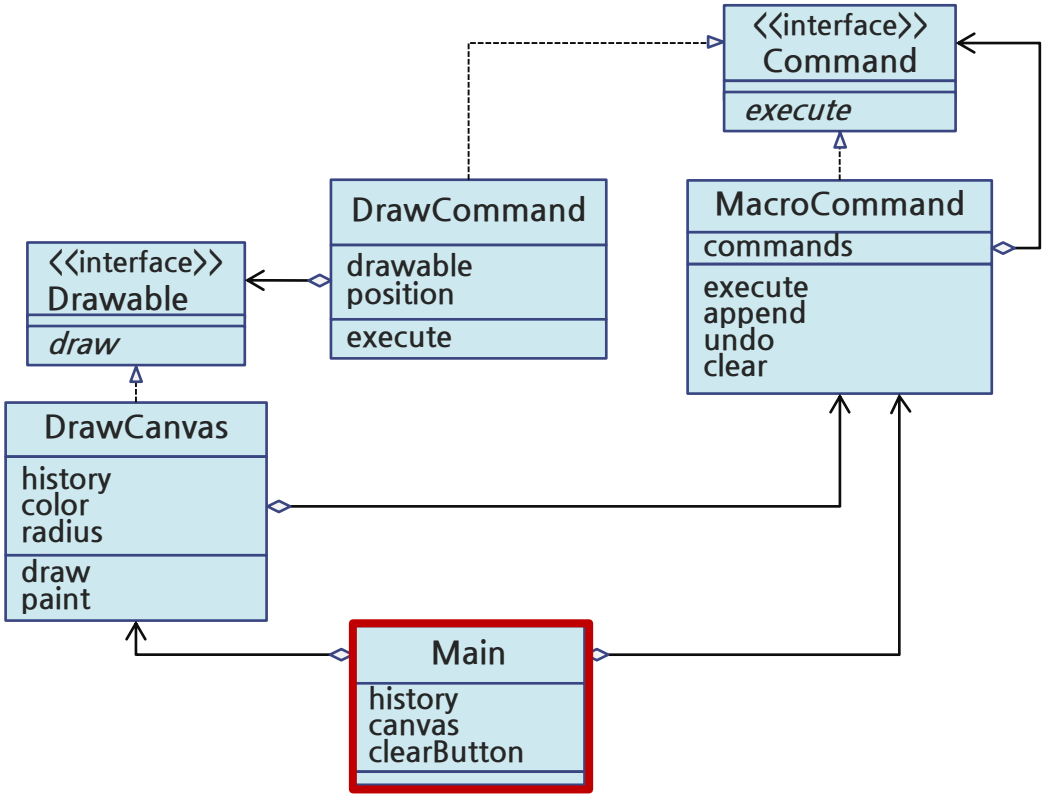
'그리기 대상'을 구현한 클래스

- history :
  지금까지 실행한 그림 그리기 명령어들의 집합 보유
- paint() :
  DrawCanvas를 다시 그릴 필요가 생겼을 때 java.awt 프레임워크로부터 자동 호출되는 메소드

디자인 패턴의 유형 (3) 행위 패턴

Command 패턴

구분



Main

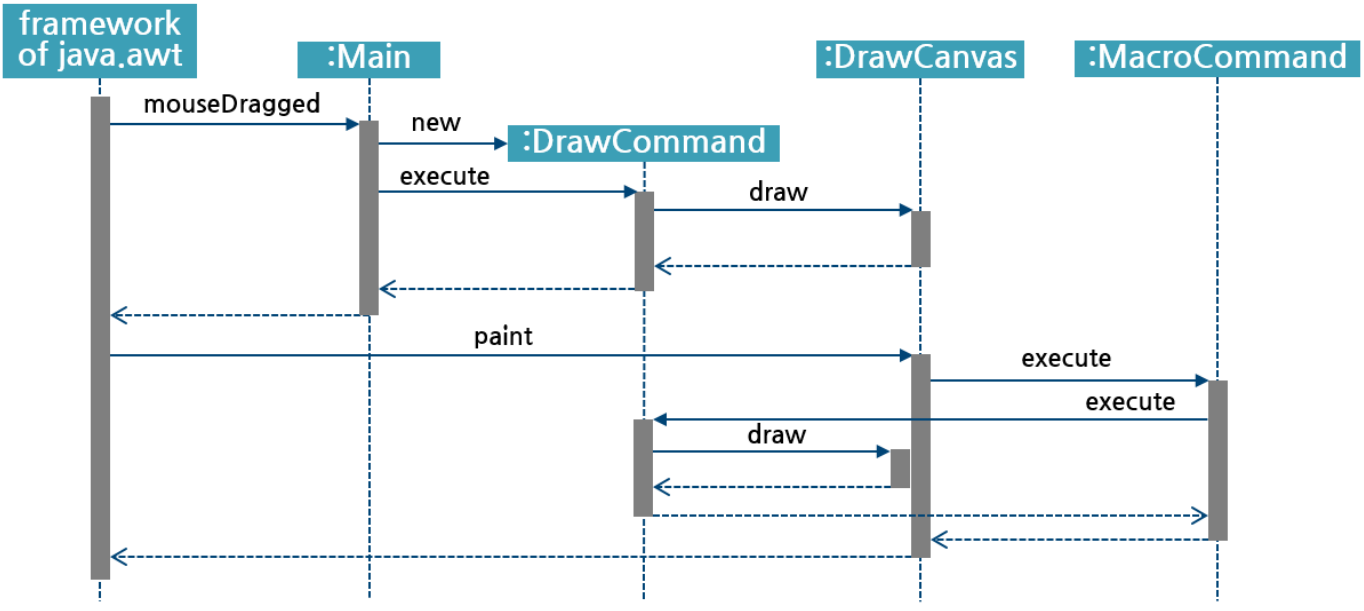
동작 테스트용 클래스

- canvas :
  그림 그리는 영역을 나타냄
- clearButtion :
  javax.swing.Jbutton 클래스

디자인 패턴의 유형 (3) 행위 패턴

Command 패턴

명령의 실행



학습정리

1. 디자인 패턴의 개요

- 디자인 패턴은 소프트웨어 설계에 있어 공통된 문제들에 대한 표준적인 해법임
- 디자인 패턴의 4요소 : 패턴 이름, 문제, 해법, 결과
- 디자인 패턴에는 캡슐화, 위임, 인터페이스, Loosely Coupling, 개방 & 폐쇄, 의존관계 원칙이 있음

2. 디자인 패턴의 유형

- 디자인 패턴의 유형 : 생성, 구조, 행위 패턴
- 생성 패턴에는 Abstract Factory, Builder, Singleton 패턴이 있음
- 구조 패턴에는 Adpator, Façade 패턴이 있음
- 행위 패턴에는 Chain of Responsibility, Command 패턴이 있음