Object Oriented Software Engineering OOSE

설계 패턴(Design Pattern)

- 패턴이란?
- 소프트웨어 디자인 패턴
- Singleton 패턴
- Façade 패턴
- Strategy 패턴
- Factory Method 패턴
- Adapter 패턴

Computer & Software Engineering
Lee Jong Yeol
soyeum@kumoh.ac.kr

패턴이란?

- ❑ 자주 발생되는 문제들을 해결하기 위한 솔루션
- ❑ 일정한 형태나 양식이 반복적으로 나타나는 현상에 대한 통칭
- ❑ 음악, 수학, 미술, 건축 등 다양한 분야에서 문제해결의 도구로 사용

<등차수열의 합>

1+100=101

2 +99=101

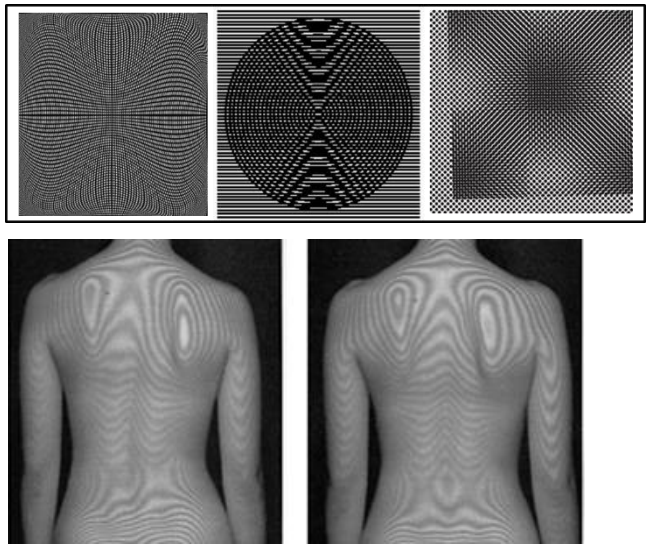
…

50+51=101

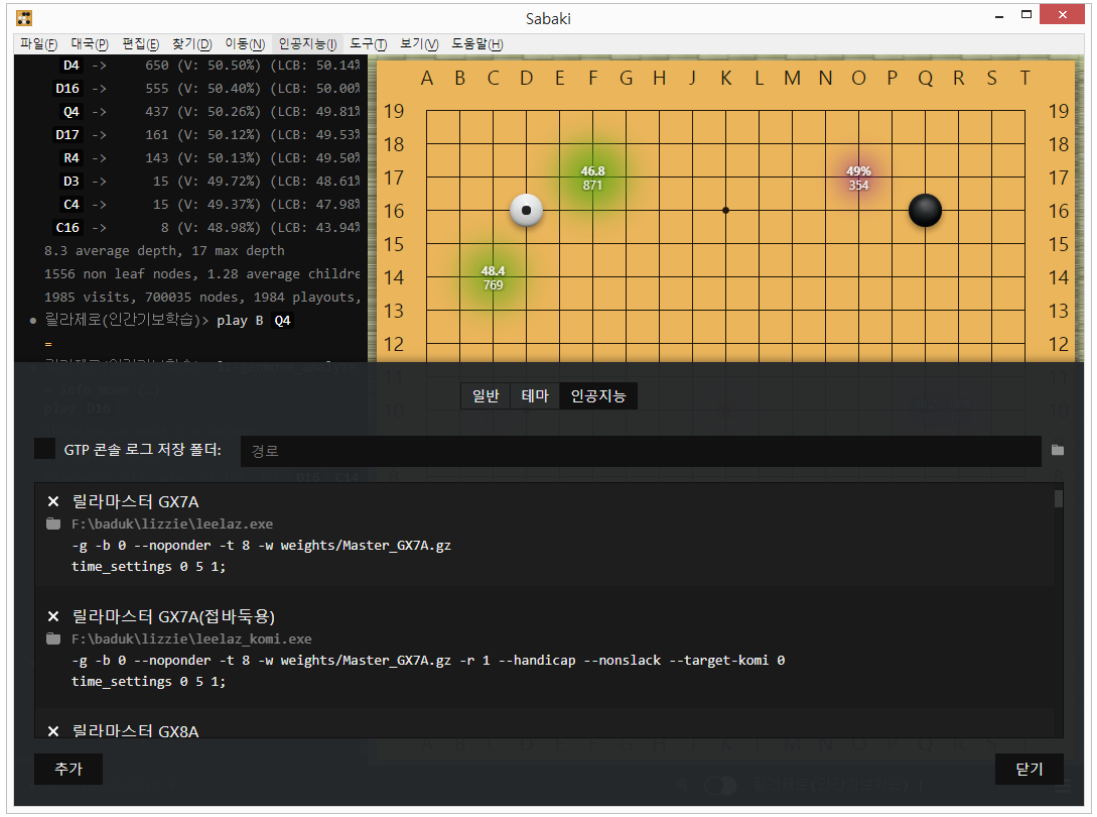
∴ 101*50 = 5050

$$S_n = \frac{n(a+l)}{2}$$

<무아레 패턴>



<인공지능 바둑 엔진>



소프트웨어 디자인 패턴

❑ 개요

- 건축물 설계에서 빈번히 발생되는 동일한 설계 내용이 있는데
  이것들을 하나의 패턴으로 인식하여 다른 건축물 설계에 재사용하면
  문제해결에 대한 고민을 반복하지 않고 무한하게 재사용 가능[Alexander]
- Alexander의 아이디어에서 영감을 얻은 Kent Beck과 Ward Cunningham 이
  1987년 OOPSLA-87에서
  'Using Pattern Languages for Object Oriented Programs'를 발표
- 'Design Patterns: Elements of Reusable Object-Oriented Software' 이란
  저서(GoF)에서 객체지향 설계를 위한 23개의 패턴 제시

  * GoF(Gang of Four)
    .소프트웨어 공학 연구자 4명을 지칭
    .Erich Gamma, John Vlissides, Richard Helm, Ralph Johnson

소프트웨어 디자인 패턴

❑ 정의

- '특정한 상황에서 일반적인 설계 문제의 해결을 위해 상호 교류하는 수정 가능한 객체와 클래스에 대한 설명이다' [GoF]
- '숙련된 객체지향 개발자 및 기타 소프트웨어 개발자는 소프트웨어 개발의 가이드라인이 되는 일반적인 원칙들과 관용적인 해결책들의 레퍼토리(repertoire)를 구축한다. 패턴(pattern)은 이러한 원칙들과 관용적 해결책들이 문제와 해결책을 기술하는 구조적인 형태로 체계화되고, 명명된 것이다' [Larman2005]
- 소프트웨어를 설계할 때 특정 상황에서 자주 사용하는 패턴을 정형화한 것
- 좋은 소프트웨어 설계를 위한 개발자들의 경험적 산물
  => 최적화된 알고리즘 코드 또는 클래스나 모듈들의 좋은 구조

소프트웨어 디자인 패턴

❑ 특징

- 경험을 통해 습득됨
- 특정한 형식을 갖고 체계적으로 작성되는 것이 일반적
- 패턴에는 각기 다른 추상화 수준이 존재하며 계속 진화
- 소프트웨어 설계 시,
  개발자들에게 추천 지침(Best practice)으로 제공되며,
  추천 지침은 개발자가 빠르고 정확하게(실수가 없도록) 설계를 할 수 있도록
  도와주어 소프트웨어 품질 향상을 기대할 수 있음

소프트웨어 디자인 패턴

❑ 장점

- 의사소통 원할 : 디자인 패턴을 알고 있는 설계자들은 특정 문제에 대해 공통적으로 알고 있는 패턴을 이용해 해결책을 논의를 할 수 있어 보다 원활한 의사소통 가능
- 경제적 : 검증된 지식인 패턴을 사용해 높은 완성도의 디자인을 빠른 시간에 만들 수 있어 소프트웨어 개발 비용 절감으로 경제적이며 코드 수준을 한 단계 높여주고 적은 수의 클래스로 원하는 목적 달성 가능한 환경 제공
- 소프트웨어 구조 파악 용이 : 좋은 설계나 아키텍처가 패턴이라는 이름으로 명명되어 있어 개발자는 그 패턴의 이름만으로도 소프트웨어 구조를 알 수 있고, 이전의 소프트웨어 개발에서 사용한 설계나 구조를 쉽게 이해할 수 있으며, 새로운 소프트웨어에 빠르게 적용할 수 있어 소프트웨어 재사용 용이

소프트웨어 디자인 패턴

❑ GoF 패턴 분류

목적 / 범위	생성 (Creational)	구조 (Structural)	행위 (Behavioral)
클래스	Factory Method	Adapter	• Interpreter • Template Method
객체	• Abstraction • Factory • Builder • Prototype • Singleton	• Adapter • Bridge • Composite • Decorator • Façade • Proxy	• Chain of Responsibility • Command • Iterator • Mediator • Memento • Flyweight • Observer • State • Strategy • Visitor

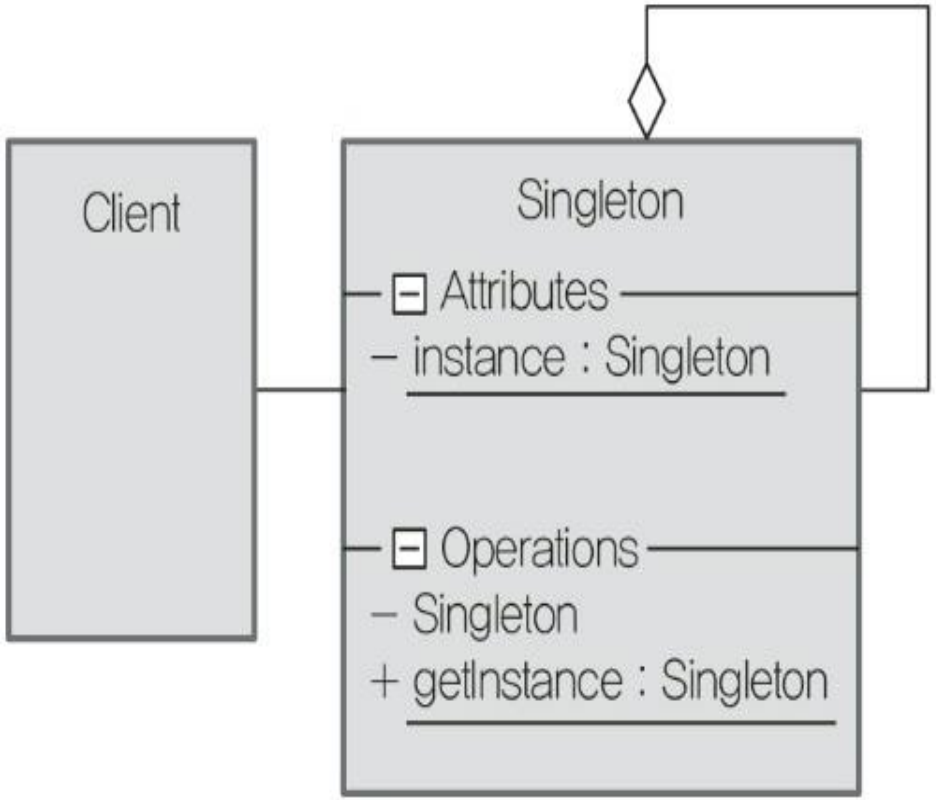
Singleton 패턴(1)

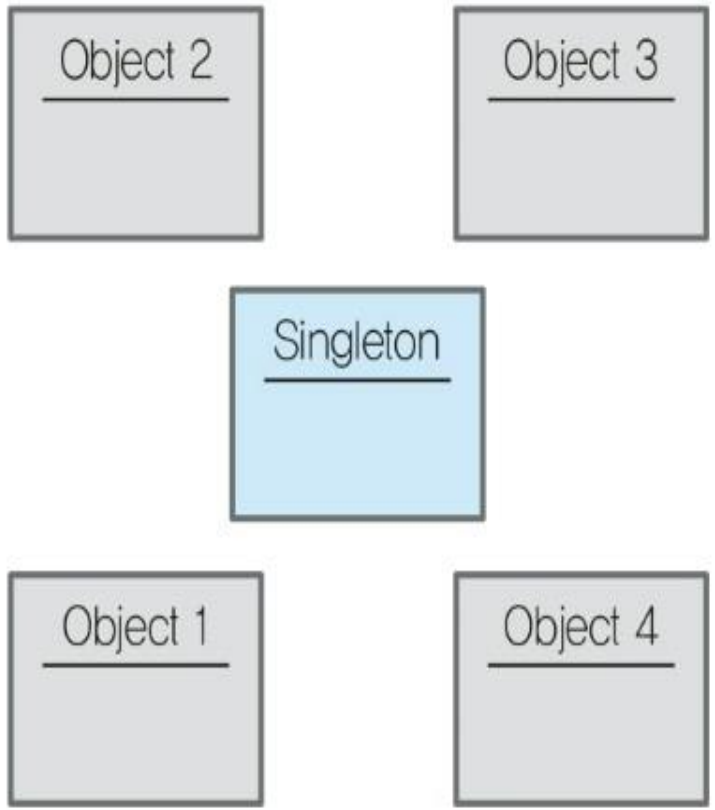
- 특정 클래스의 인스턴스가 오직 하나임을 보장하고(객체가 한 개로 제한), 이 인스턴스에 접근할 수 있는 방법 제공
- Singleton은 자기 자신의 클래스타입의 static 변수를 갖고, getInstance() 메소드에 의해 오직 한 개만 생성하여 제공 때문에 자기 자신과 연관(association)을 갖는 구조적 특징이 있음
- 인쇄관리 프로그램의 경우, 인쇄 요청은 여러 프로그램에서 동시에 올 수 있고, 한 번에 한 페이지에서부터 수백 페이지까지 다양한 요청 가능

  만일 여러 프로그램에서 서로 다른 여러 장의 문서 또는 이미지를 프린터기로 인쇄를 요청하면, 인쇄물이 섞이는 일이 발생할 수도 있을 것이므로 하나의 인쇄관리 프로그램을 통해서 프린터기로 인쇄할 문서의 페이지를 차례차례 보냄

Singleton 패턴(2)

❑ 클래스 구조





Singleton 패턴(3)

Singleton Pattern 기본 코드

```
public class Singleton {
        private static Singleton instance;
        private Singleton() {
            // 기본생성자가 private이기 때문에 외부에서 인스턴스를 생성할 수 없다.
        }
        // Singleton 객체는 getInstance() 메소드를 통해서만 객체가 생성된다.
        public static Singleton getInstatnce() {
                if(instance==null) {
                        instance = new Singleton();
                }
                return instance;
        }
}
```

Singleton 패턴 활용 (1)

■ **목표 : 인쇄관리 프로그램 개발**

■ 요구사항

프로그램은 윈도우 OS에서 동작하는 인쇄관리 프로그램이다. 문서 프로그램이나 이미지 관리 프로그램에서 인쇄를 요청을 하면 그 요청을 받아 지정된 프린트기로 인쇄 페이지를 전달하는 프로그램이다. 이때 프린트기는 프로그램으로부터 한 번에 한 페이지씩 받아 인쇄를 수행한다. 때문에 인쇄관리 프로그램은 여러 인쇄 페이지가 있을 경우 **한 번에 한 페이지씩 프린트기로 전송**해야 한다. 또한 프린트기는 여러 종류가 연결되어 있을 수 있고, 사용자가 지정한 프린트기를 통해 인쇄되어야 한다.

Singleton 패턴 활용 (2)

Singleton 패턴 활용 (4)

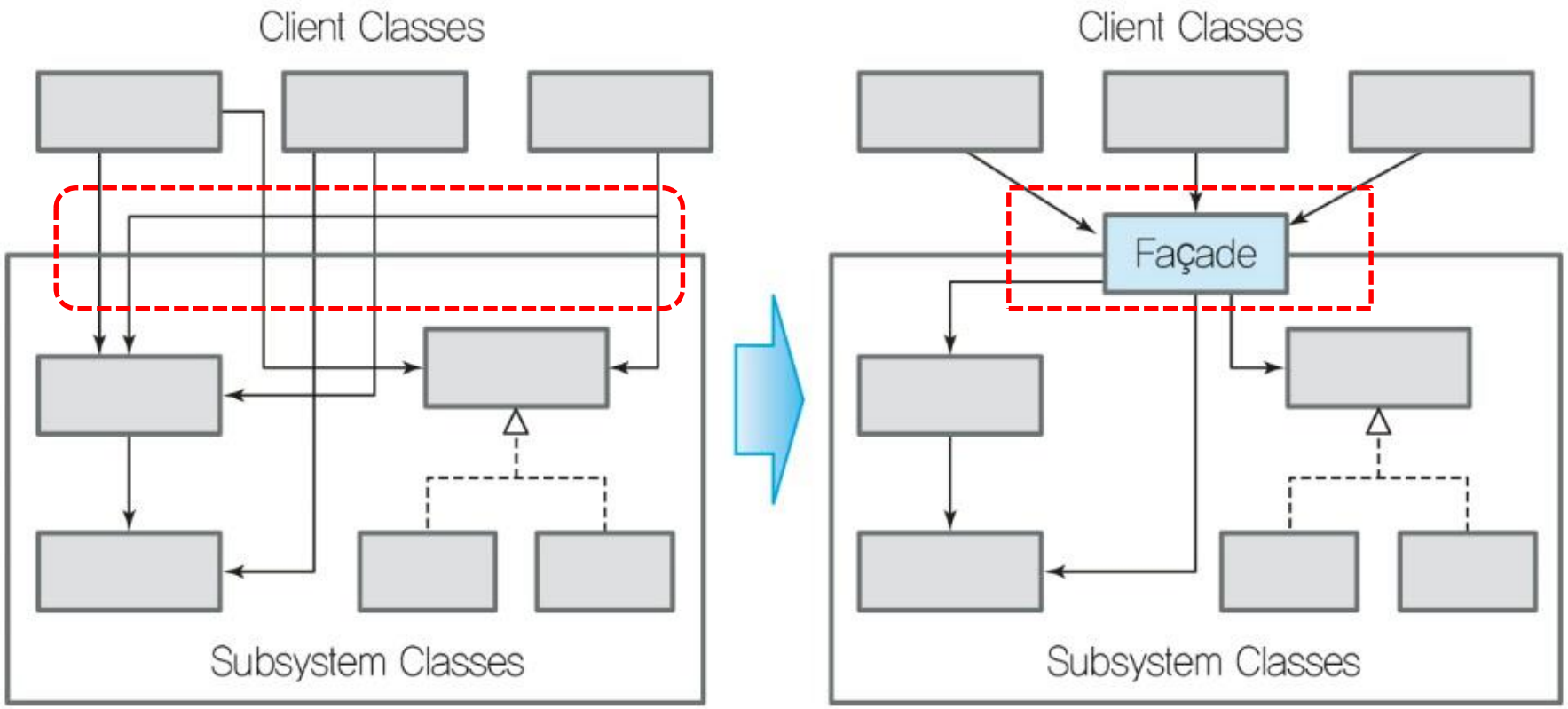
```
public class PrinterManager {

        private static PrinterManager instance;

        private PrinterManager () {

         }

        public static PrinterManager getInstatnce() {
                if(instance==null) {
                        instance = new PrinterManager ();
                }
                return instance;
        }
}
```

Façade 패턴(1)

- ❑ Façade 는 건물의 출입구로 이용되는 정면 외벽 부분을 가리키며
  정문의 현관을 포함하고 큰 건물의 경우에는 '안내소'가 위치함
- ❑ 개발자가 사용해야 하는 서브시스템의 가장 앞쪽에 위치하고 있으면서
  하위시스템에 있는 객체들을 사용할 수 있도록 하는 역할 수행
- ❑ 시스템간 복잡한 연관관계가 있을 경우,
  두 시스템 사이에 위치하여 복잡성을 줄이고 서브시스템을 구조화하여
  서브시스템으로의 접근을 하나의 Façade 객체로 제공하는 패턴

Façade 패턴(2)

❑ 패턴 적용 전/후



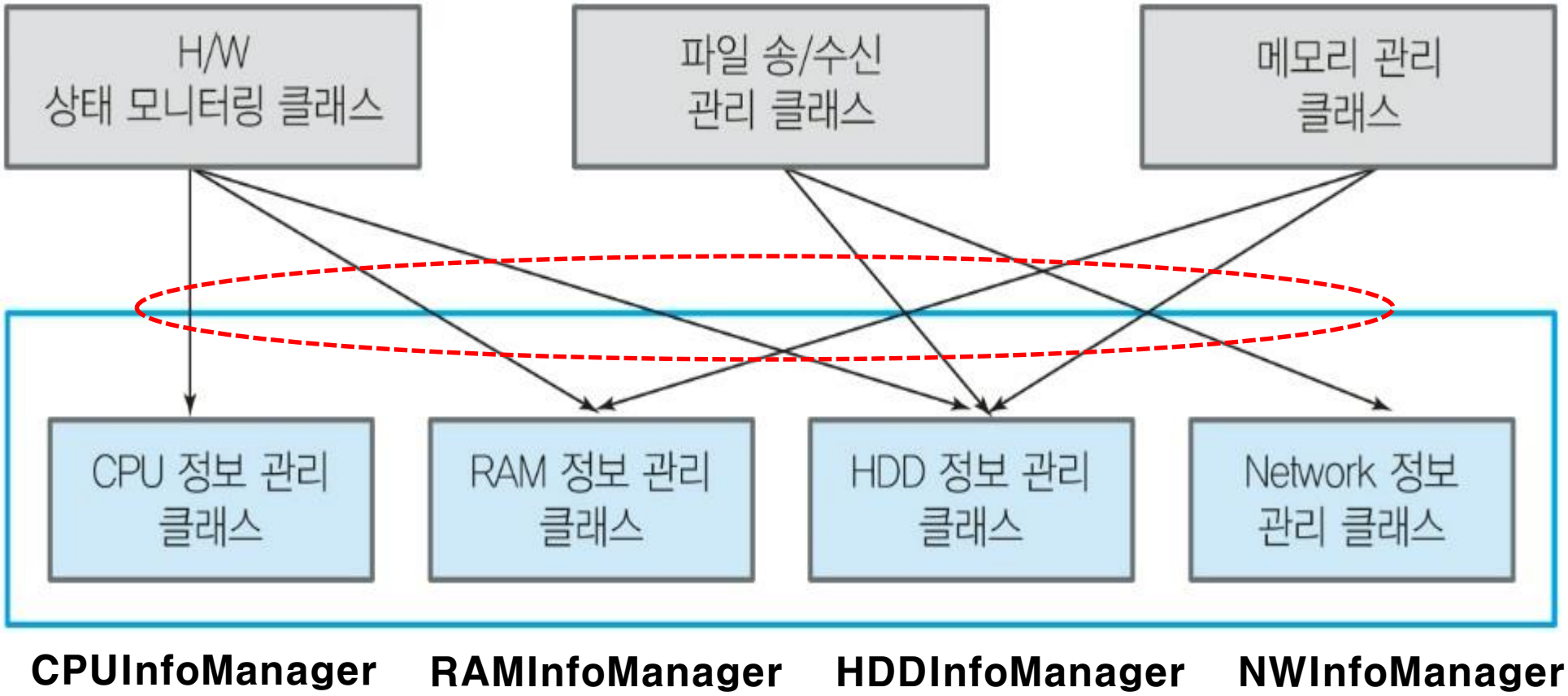
Façade 패턴(3)

❑ Façade 패턴이 필요한 상황

- 복잡한 서브시스템에 대한 단순한 인터페이스 제공이 필요할 때
- 클라이언트와 구현 클래스 간 너무 많은 의존성이 존재하여 클라이언트와 다른 서브시스템 간 결합도를 줄일 필요가 있을 때
- 빌딩 블록(Building Block) 아키텍처, 컴포넌트 기반 개발 (Component Based Development), Service Oriented Architecture 등의 경우와 같이 서로의 내부구조를 감추고 블랙박스로 이해해야 할 때
- 서브시스템들이 계층 구조를 이루고 있어서
  각 서브시스템의 계층별 접근점(Façade 객체)을 제공하려 할 때

Façade 패턴(4)

❑ 높은 결합도 구조



Façade 패턴(6)

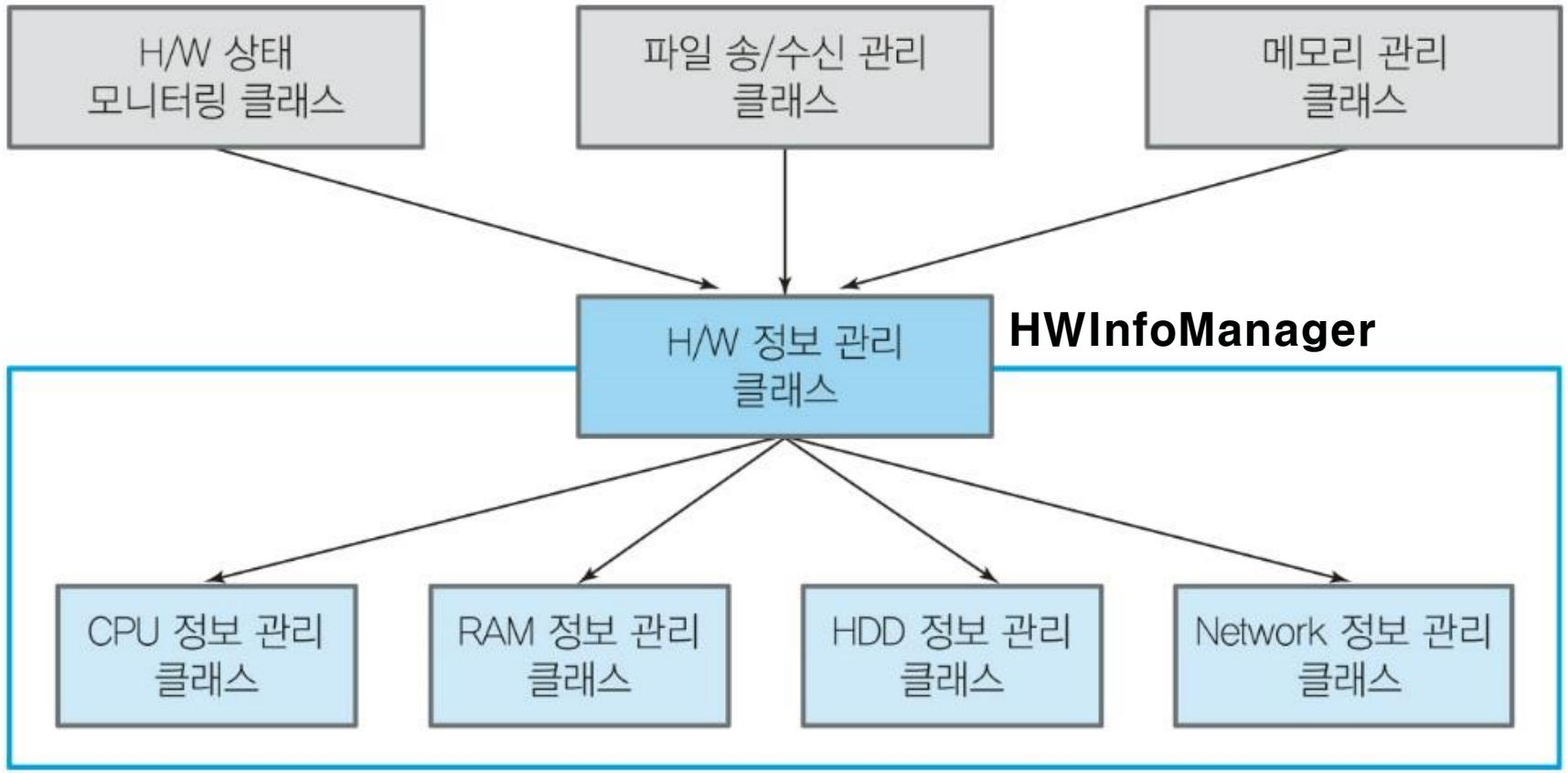
1.Façade를 사용하지 않은 구조의 코드

```
// H/W 상태 모니터링 클래스내 H/W상태 정보를 그래프로 표출하는 메소드
public void displayHwStateGraph() {
        // 현재 HDD 정보는 HDDInfoManager를 이용
        HDDInfo hddInfo = HDDInfoManager.getInstance().getHDDInfo();
        // 현재 CPU 정보는 CPUInfoManager를 이용
        CPUInfo cpuInfo = CPUInfoManager.getInstance().getCPUInfo();
        // 현재 RAM 정보는 RAMInfoManager를 이용
        RAMInfo ramInfo = RAMInfoManager.getInstance().getRamInfo();

        //HDD정보를 그래피컬하게 전달
        DrawHWInfo("hdd",hddInfo);
        DrawHWInfo("cpu",cpuInfo);
        DrawHWInfo("ram",ramInfo);
}
```

[코드설명]
위의 "RAMInfoManager.*getInstance*().getRamInfo()" API를 만든 개발자가
getRamInfo() 메소드를 이용해 지난 시간 Ram의 상태 정보를 알 수 있도록 파라미터를 추가하여
재개발 해야하는 일이 발생 했다고 가정하면,
-변경 전 : getRamInfo()
-변경 후 : getRamInfo(int _pastime)
이럴 경우 getRamInfo()를 사용한 개발자들은 해당 코드를 모두 getRamInfo(0);으로 변경 필요

Façade 패턴(5)

❑ 낮은 결합도 구조



Façade 패턴(7)

2.Façade가 사용된 구조의 코드

```
// H/W 상태 모니터링 클래스내 H/W상태 정보를 그래프로 표출하는 메소드
public void displayHwStateGraph() {
        // H/W정보는 Façade패턴을 적용하여 만들어진 HWInfoManager를 이용
        HDDInfo hddInfo = HWInfoManager.getInstance().getHDDInfo();
        CPUInfo cpuInfo = HWInfoManager.getInstance().getCPUInfo();
        RAMInfo ramInfo = HWInfoManager.getInstance().getRamInfo();

        //HDD정보 그리픽컬하게 전달
        DrawHWInfo("hdd",hddInfo);
        DrawHWInfo("cpu",cpuInfo);
        DrawHWInfo("ram",ramInfo);
}
```

: Façade를 적용하면 API 사용자는 HDD정보, CPU정보, RAM정보를 HWInfoManager를 통해서 알 수 있기 때문에 하드웨어 유형에 따라 제공 클래스를 찾아야 하는 번거로움을 해결할 수 있다.

Façade 패턴(8)

3.Façade 클래스

```
// Façade패턴을 적용하여 만들어진 'H/W정보 관리 클래스'
public class HWInfoManager {
        private static HWInfoManager instance;
        public static HWInfoManager getInstance() {
                if(instance==null) {
                        instance = new HWInfoManager();
                }
                return instance;
        }
        // CPU정보 반환
        public CPUInfo getCPUInfo() {
                return CPUInfoManager.getInstance().getCPUInfo();
        }
        // HDD정보 반환
        public HDDInfo getHDDInfo() {
                return  HDDInfoManager.getInstance().getHDDInfo();
        }
        // RAM정보 반환
        public RAMInfo getRamInfo() {
                return  RAMInfoManager.getInstance().getRamInfo();
        }
}
```

Strategy 패턴(1)

- 다양한 알고리즘이 존재할 때 이들 각각을 하나의 클래스로 캡슐화하여 필요할 때 알고리즘을 대체 가능하도록 클래스 설계
- 클라이언트에 영향을 주지 않고 다양한 알고리즘으로 변형할 수 있어 알고리즘을 바꾸더라도 클라이언트는 변경 불필요
- GoF의 분류적인 관점에서는 알고리즘을 담당하는 각각의 클래스를 만들어 책임을 분산하기 위한 목적으로 만든 행위 패턴이고,
  각각의 알고리즘을 필요한 시점(런타임 시)에서 동적으로 변경하여 사용할 수 있기 때문에 범위적인 측면에서는 객체 패턴

Strategy 패턴(2)

❑ Strategy 패턴이 유용하게 사용되는 경우

- 행위들이 조금씩 다를 뿐 개념적으로 관련된 많은 클래스들이 존재하는 경우, 각각의 서로 다른 행위 별로 클래스를 작성해서 알고리즘을 선택해야할 때
- 저장 공간과 처리 속도 간의 절충에 따라 서로 다른 알고리즘 사용할 때
- 많은 행위를 정의하기 위해 클래스 안에 복잡한 다중 조건문을 사용해야 하는 경우
- 어떤 알고리즘이 클라이언트가 알아서는 안될 데이터를 사용하거나, 알고리즘에 종속된 복잡한 자료구조를 사용할 때

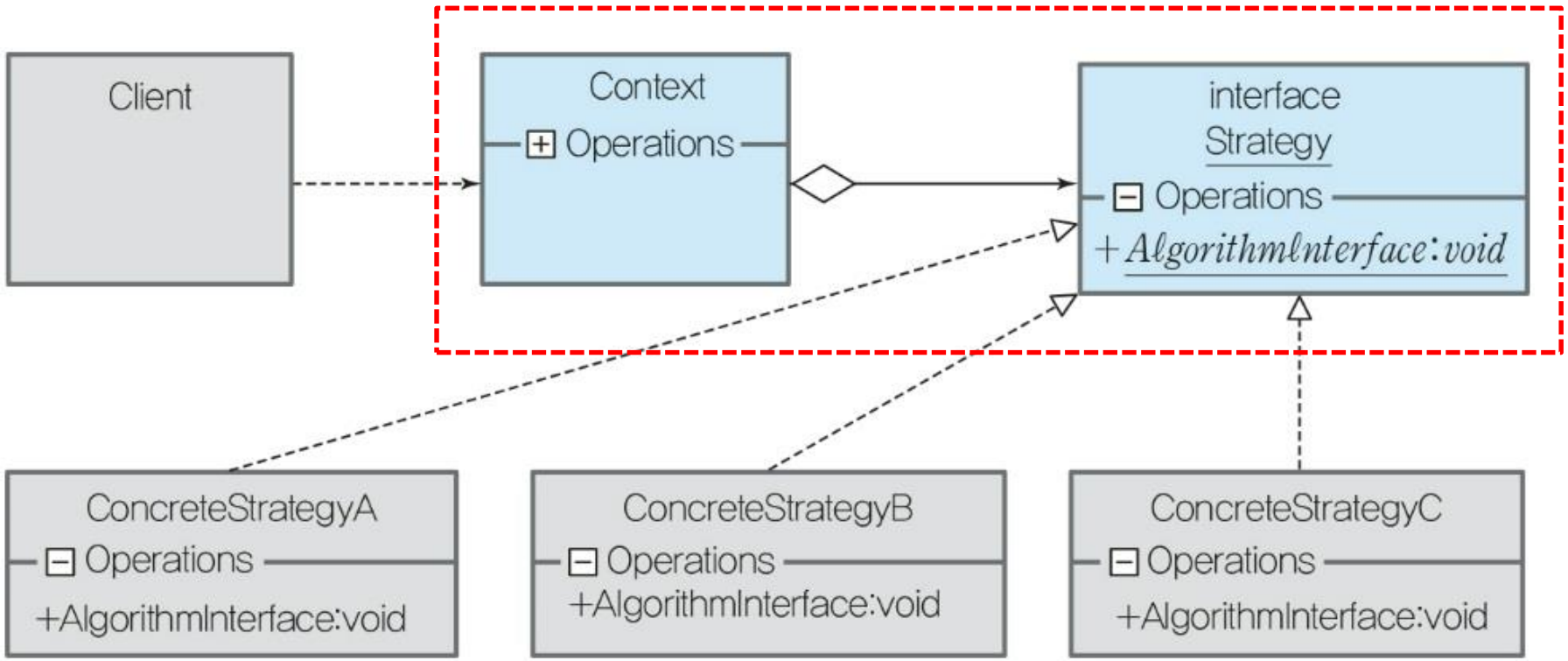
Strategy 패턴(3)

❑ 클래스 구성

- **Strategy**
  : 알고리즘의 접근을 허용하는 인터페이스
- **ConcreteStrategy**
  : 실제 알고리즘이 구현된 클래스 (여러 개 존재)
- **Context**
  : Strategy 인터페이스를 통해 알고리즘을 사용

Strategy 패턴(4)

❑ 클래스 구조



Strategy 패턴(5)

Strategy Pattern 예제

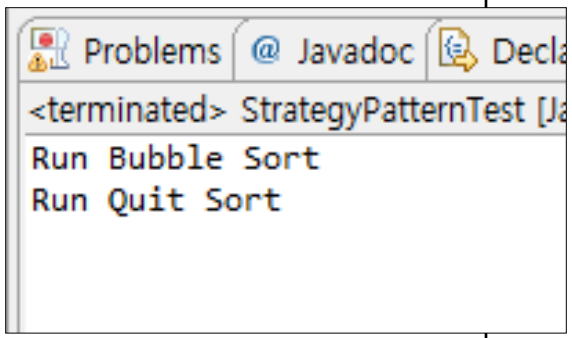
```java
// #1 SortStrategy.java - Strategy 인터페이스
package StrategyPattern;
import java.util.ArrayList;
public interface SortStrategy {
    public void sort(ArrayList<Integer> dataList);
}

// #2 BubbleSort.java - Strategy 구상 클래스
package StrategyPattern;
import java.util.ArrayList;
public class BubbleSort implements SortStrategy {
    @Override
    public void sort(ArrayList<Integer> dataList) {
        System.out.println("Run Bubble Sort");
    }
}
```

```java
// #3 QuickSort.java - Strategy 구상 클래스
package StrategyPattern;
import java.util.ArrayList;
public class QuickSort implements SortStrategy {
    @Override
    public void sort(ArrayList<Integer> dataList) {
        System.out.println("Run Quick Sort");
    }
}

// #4 Context.java - 알고리즘을 사용하는 클래스
package StrategyPattern;
import java.util.ArrayList;
public class Context {
    private SortStrategy srtStrategy;
    public void setContext(SortStrategy srtStrategy) {
        this.srtStrategy = srtStrategy;
    }
    public void sortExcute(ArrayList<Integer> arrayList)  {
        this.srtStrategy.sort(arrayList);
    }
}
```

Strategy 패턴(6)

Strategy Pattern 예제 (계속)

```
// #5 StrategyPatternTest.java - Strategy 패턴 사용 클라이언트 프로그램
package StrategyPattern;
import java.util.ArrayList;
import java.util.Random;
public class StrategyPatternTest {
        public static void makeNum(ArrayList<Integer> arrayList) {
                Random random = new Random();
                for(int i=0; i< 10; i++) {
                        arrayList.add(random.nextInt(1000));
                }
        }
        public static void main(String[] args) {
                ArrayList<Integer>  arrayList= new ArrayList<Integer>();
                StrategyPatternTest.makeNum(arrayList);
                Context context = new Context();

                context.setContext(new BubbleSort());
                context.sortExcute(arrayList) ;

                context.setContext(new QuickSort());
                context.sortExcute(arrayList);

        }
}
```



Factory Method 패턴(1)

- 객체를 생성하기 위해 일정한 절차가 필요하거나
  객체를 생성하는 시점이 불명확할 경우 객체를 생성하는 메소드 이용
- 객체를 생성하기 위한 인터페이스를 정의하지만,
  어떤 클래스의 인스턴스를 생성할지에 대한 결정은 하위 클래스에서
  이루어지도록 인스턴스 생성 책임 미룸
- 이 패턴은 기반 클래스가 모든(혹은 대부분의) 일을 하지만
  정확히 어떤 객체를 갖고 작업할지에 대해서는 런타임 시로 미룰 때 유용

Factory Method 패턴(2)

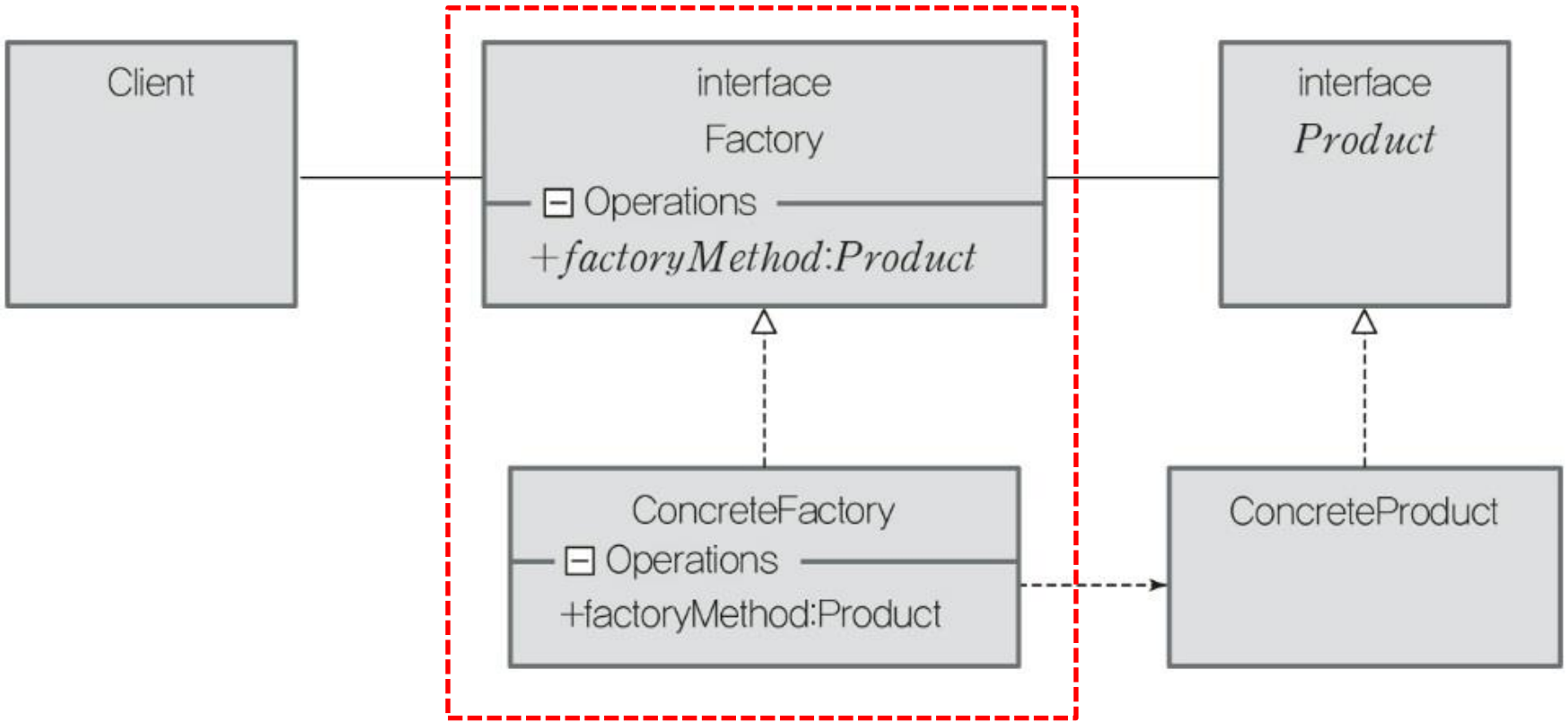
❑ 클래스 구성

- Creator(Factory)
  - 실제 타입이 알려지지 않은 객체를 생성할 수 있는 메서드로 정의하여 이를 추상 메서드 호출을 통해 수행한다.
- Concrete Creator(ConcreteFactory)
  - Creator의 객체 생성 추상 메서드를 오버라이드하여 Concrete Product를 생성하는 파생 클래스
- Product
  - Factory 메서드가 생성하는 객체의 인터페이스를 정의
  - Creator는 이 인터페이스를 통해 Concrete Product에 접근
- Concrete Product
  - Creator(기반 클래스) 메서드에 의해 사용되는 객체
  - Product 인터페이스를 구현

Factory Method 패턴(3)

❑ 클래스 구조

▪ 두 개의 인터페이스와 두 개의 하위 클래스로 구성



Adapter 패턴(1)

❑ 배경

- 소프트웨어 개발할 때는 관련 라이브러리에서 제공하는 API를 대부분 사용
- 프로젝트의 기간이 많지 않을 때 관련 라이브러리의 존재는
  가뭄의 단비와 같이 개발자에게 기쁜 일
- 하지만 사용해야 하는 라이브러리에서 제공하는 API가 내가 개발해서
  사용해야 할 모듈의 인터페이스가 다르다면 대략 난감
- 라이브러리 제공자에게 도움을 요청해 바꿀 수 있다면 쉽게 해결 되지만
  대부분 그럴 가능성은 희박함
- 적은 비용으로 기존의 라이브러리를 사용할 수 있는 방법이 필요함

Adapter 패턴(2)

❑ 목적

- 클래스의 인터페이스를 클라이언트가 기대하는 다른 인터페이스로 변환
- Adapter 패턴은 호환성이 없는 인터페이스이기 때문에
  같이 사용할 수 없는 클래스를 개조하여 함께 작동하도록 해줌

❑ 구조

- 상속을 활용한 Adapter 패턴
  : Adapter 클래스는 Adaptee를 상속 또는 구현하여 클라이언트가 사용할
  도메인에 종속적인 메서드를 제공하는 구조
- 객체 합성을 이용한 Adapter 패턴
  : Adapter 클래스는 Adaptee 클래스 타입의 멤버 변수를 선언하여
  클라이언트가 사용할 도메인에 종속적인 메서드를 제공하는 구조

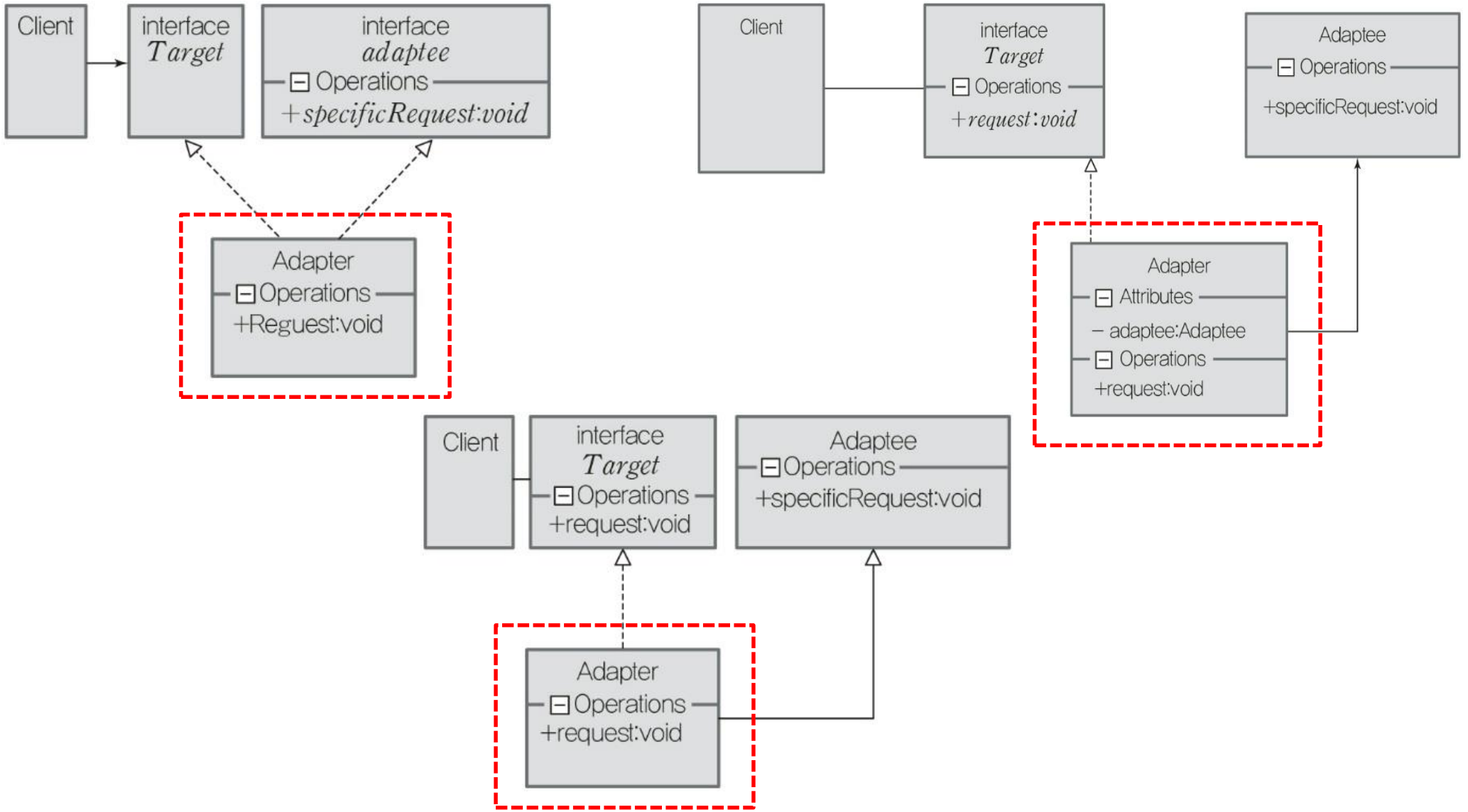
Adapter 패턴(3)

❑ 클래스 구성

- **Adaptee**
  : Client가 원하는 인터페이스를 지원하지 않는 객체
- **Target**
  : Client가 Adaptee로 부터 지원을 바라는 오퍼레이션을 가진 인터페이스
- **Adapter**
  : Adaptee가 Target 인터페이스를 지원하는 것처럼 보이게 해주는 클래스로써 상속이나 위임을 사용하여 구현된다.

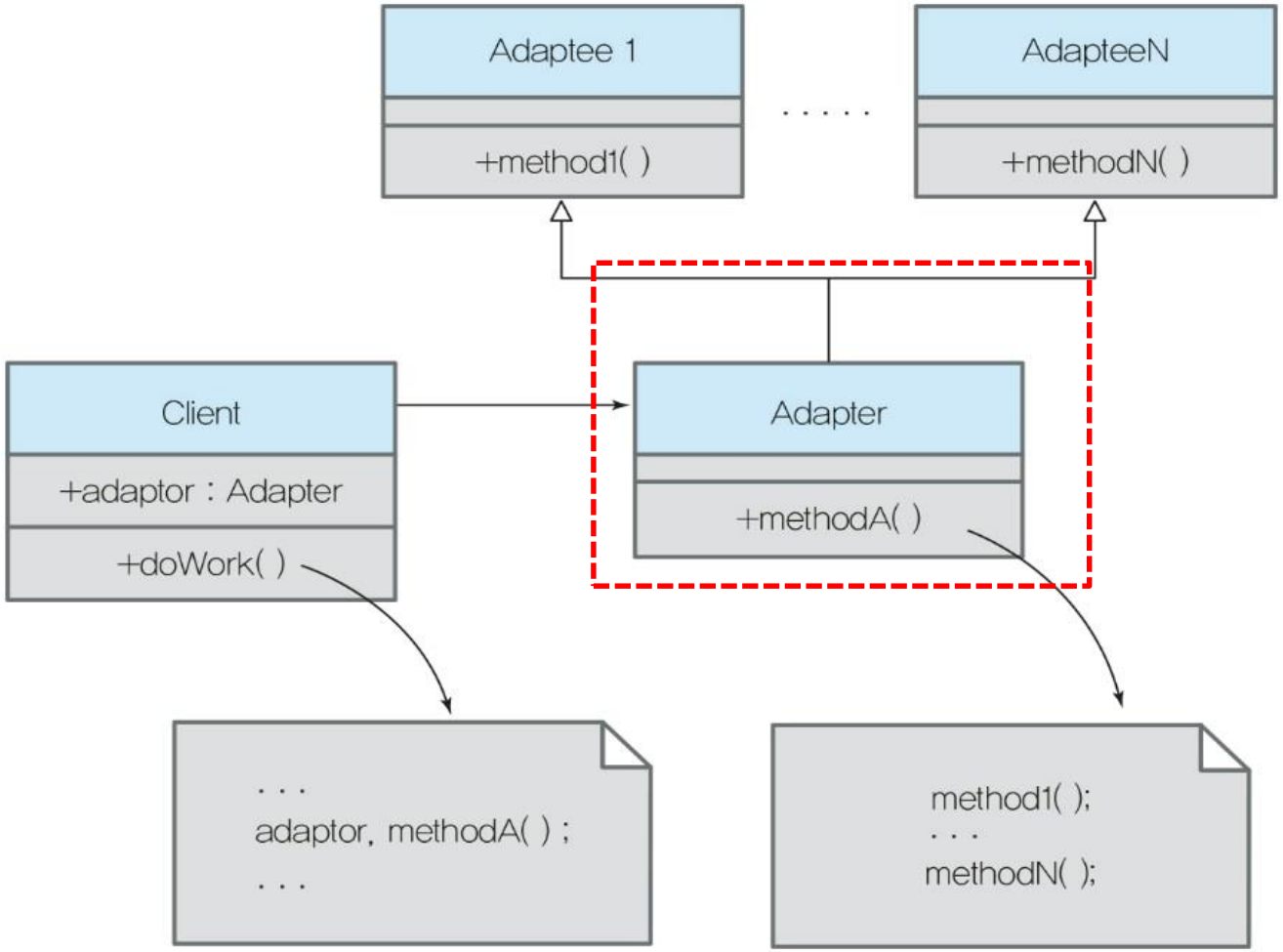
Adapter 패턴(4)

❑ 클래스 구조(1)



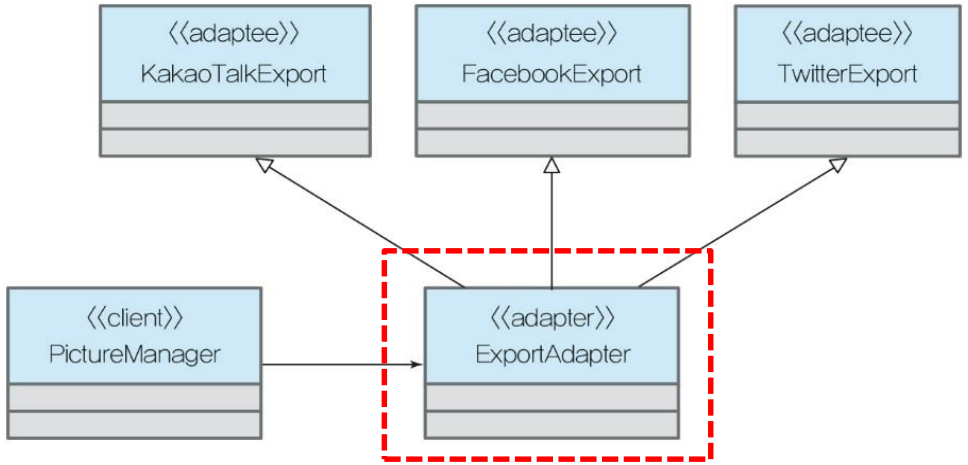
Adapter 패턴(5)

❑ 클래스 구조(2)

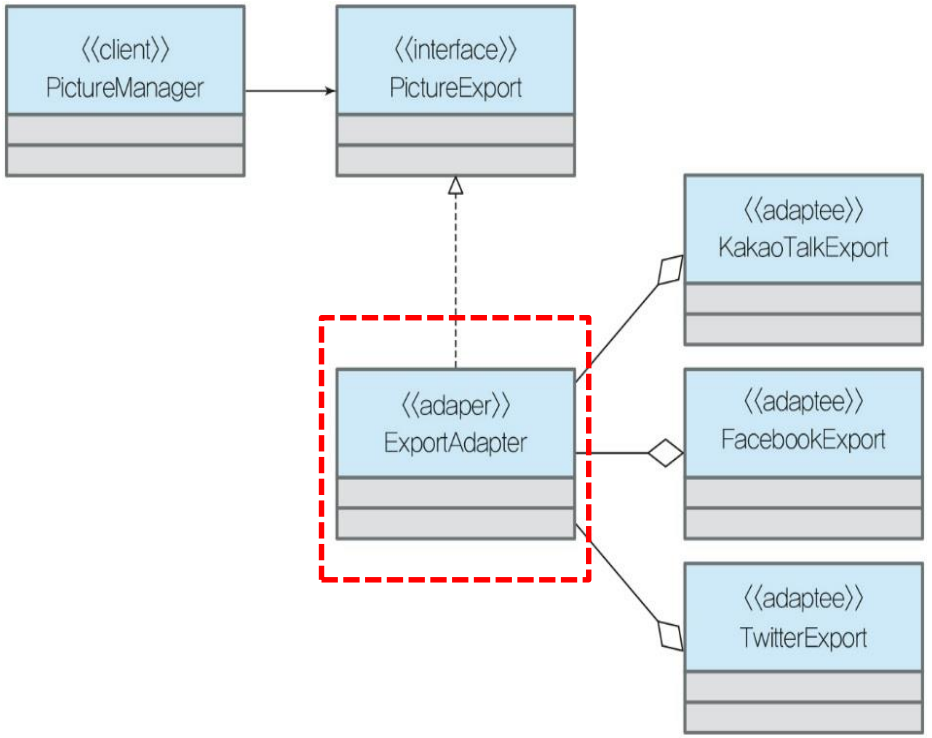


Adapter 패턴(7)

❑ 사진 내보내기 기능을 위한 Adapter 패턴 구조



❑ 다중 상속을 배제한 Adapter 패턴 구조



Adapter 패턴(8)

❑ Adapter 패턴이 유용하게 사용되는 경우

- 기존의 클래스를 사용해야 하나 인터페이스가 수정되어야 하는 경우
- 아직 예측하지 못한 클래스나 실제 관련되지 않는 클래스들이
  기존의 클래스를 재사용 하고자 하지만,
  이미 정의된 재사용 가능한 클래스가 지금 요청하는 인터페이스를
  꼭 정의 하고 있지 않는 경우
  => 즉, 이미 만들어진 것을 재사용하고자 하나
  재사용 가능한 라이브러리를 수정할 수 없는 경우

패턴의 남용(濫用)

- 過猶不及(과유불급)
- 항상 고려해야 하며 패턴을 배웠다고 해서 패턴을 사용하지 말아야 할 곳에 까지 패턴을 사용하게 되면 필요 이상으로 설계가 복잡해짐
- 무분별한 패턴 사용으로 인해 유지보수가 어려운 소프트웨어가 만들어짐
- 패턴을 남용하지 않기 위해서는 패턴에 대해 정확히 이해하고, 소프트웨어의 기능이나 규모에 따라 패턴을 유연하게 사용할 수 있는 설계 능력을 갖추어야 함