# 7장. SW 아키텍처와 설계패턴

숙명여자대학교
SOOKMYUNG WOMEN'S UNIVERSITY

# Contents

# SW 설계

- 설계(Design)
  - 도메인 중심의 분석 결과물을 소프트웨어 관점의 산출물로 전환하는 과정

1. 소프트웨어 아키텍처 설계
2. 상세설계 : 소프트웨어 시스템을 구성하는 로직 설계, 즉 클래스와 메서드에 대한 설계를 비롯하여 사용자 인터페이스, 자료 구조 혹은 데이터베이스 설계, 물리적인 아키텍처 설계 등과 같은 활동을 포함

# 1. 소프트웨어 아키텍처 (1/3)

- Software architecture는 시스템의 기본적인 기술서
  - 시스템을 구성하는 컴포넌트
  - 컴포넌트들 사이의 의미 있는 연결(collaborations)
- SW 아키텍처는 비기능적 결정(non-functional decisions)을 반영하고, 기능 요구사항을 분할
- SW 아키텍처가 중요한 이유
  - 일단 시스템이 개발된 뒤에는 잘못된 구조를 바로잡기가 쉽지 않음

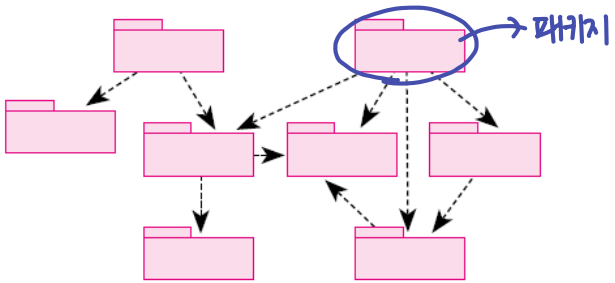
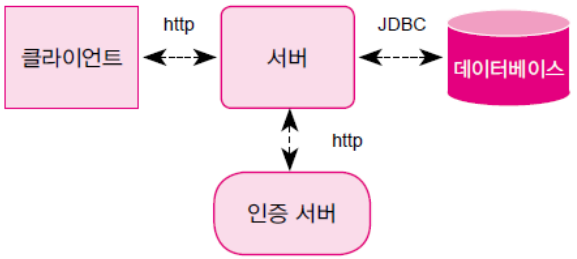
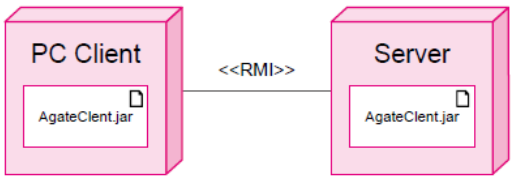
# 1. 소프트웨어 아키텍처 (2/3)

- SW 아키텍처 설계에서 고려해야 될 요구사항
  - 변경, 유지보수 용이성
  - 상용 컴포넌트의 사용
  - 시스템 성능
  - 신뢰성
  - 보안
  - 고장 인내성
  - 복구

# 1. 소프트웨어 아키텍처 (3/3)

여러 표현 방법

- SW아키텍처의 표현 = 관점(view)
  - 시스템을 특정 구성요소와 그들 사이의 관계로 표현

| 뷰 | 중심 관점 | 사례 |
|---|---|---|
| 모듈<br>패키지다이어그램 | 코드 구조<br>– 의존, 상속, 부분 관계 등 |  |
| 컴포넌트와<br>커넥터<br>컴포넌트다이어그램 | 런타임 구조<br>– 통신, 호출 관계 |  |
| 배치<br>배치다이어그램 | 소프트웨어와 환경의<br>배치 구조 |  |

# 2. SW 아키텍처 스타일

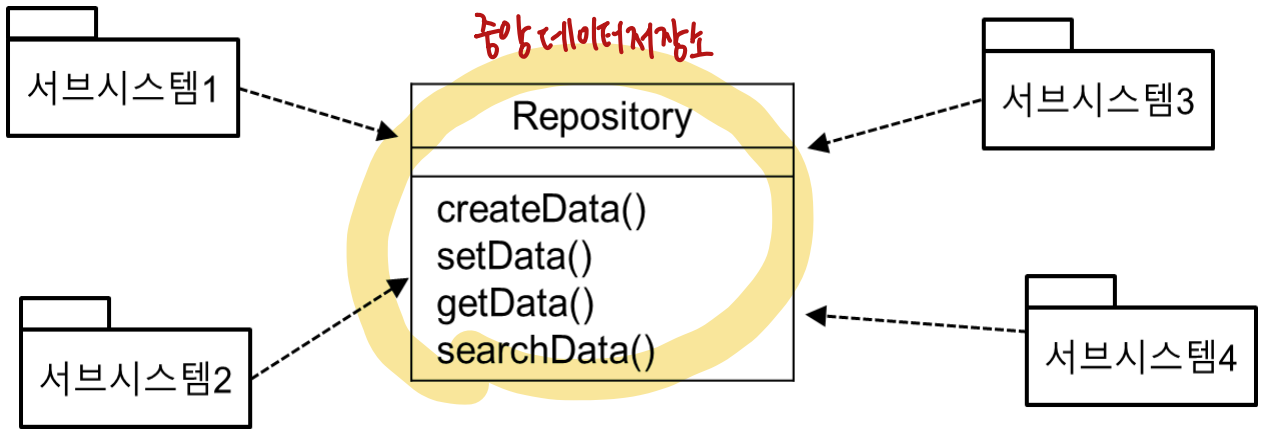
- 건축양식(고딕, 로코코, 한옥….)과 같이 일반적인 모양과 조화를 위한 스타일을 정하는 작업
  - 시스템 분할, 전체적인 제어 흐름, 오류처리 방침, 서브시스템 간의 통신 프로토콜 포함

- 대표적인 소프트웨어 아키텍처 스타일
  - 중앙저장소 구조
  - 클라이언트-서버구조
  - 계층구조
  - MVC 구조
  - Peer-to-Peer 구조
  - 파이프필터 구조

# 2.1 중앙저장소 구조 (1/2)

데이터중심아키텍처

- 서브시스템들이 단일 중앙 저장소의 자료를 접근하고 변경
  - 서브시스템들은 독립적이고 중앙 자료 저장소를 이용하여 상호 대화
  - 새로운 서브시스템도 저장소를 중심으로 위치함
  - 유형: 블랙보드, Repository

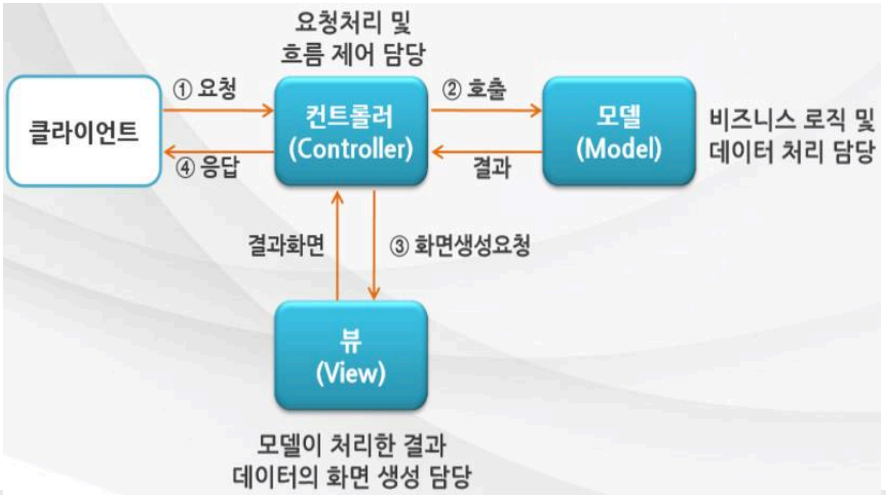


# 2.1 중앙저장소 구조 (2/2)

- 특징
  - 서브시스템 사이의 병렬처리와 통합이 용이
  - 저장소는 계속 변경 추가
  - 중앙저장소가 잘 정의되면 새로운 서비스를 서브시스템으로 쉽게 추가할 수 있음
  - 급여시스템이나 뱅킹 시스템과 같이 주로 데이터 처리업무에 적합
- 단점
  - 중앙저장소와 서브시스템의 결합이 매우 강하므로 저장소가 변경되면 모든 서브시스템이 영향을 받음
  - 단일 장애지점(Single Point Of Failure) 문제

# 2.2 MVC 구조 (1/2)

- MVC(Model-View-Controller)
  - UI로부터 비즈니스 로직과 데이터를 분리
  - 모델: 애플리케이션의 정보(데이터)를 나타냄
  - 뷰: 텍스트, 체크박스 항목 등과 같은 사용자 인터페이스 요소를 나타냄
  - 컨트롤러: 데이터와 비즈니스 로직 사이의 상호동작을 관리

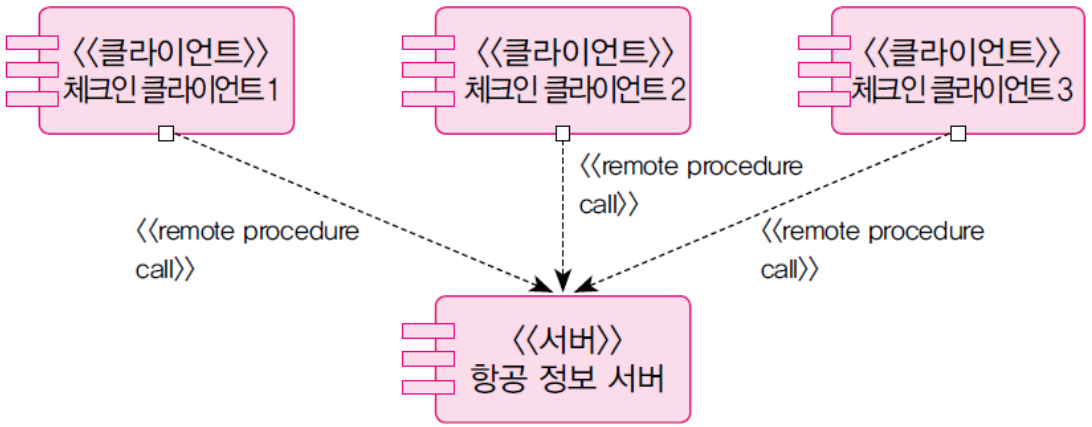


# 2.2 MVC 구조 (2/2)

- 모델 서브시스템은 시스템의 여러 뷰나 컨트롤러와 의존되지 않도록 개발
- 장점
  - 느슨한 결합, 확장성
  - 다수의 뷰를 사용하는 대화형 시스템에 적합
- 단점
  - 복잡도
  - 비효율성
  - 각 구성요소에 대한 구현을 위한 여러가지 기술에 대한 이해가 요구됨

# 2.3 클라이언트/서버 구조 (1/2)

- 클라이언트 서버(Client/Server)
  - 서버는 여러 클라이언트에게 서비스 제공
  - 요청과 결과를 받기 위하여 동기화 되는 일을 제외하고는 모두 독립적
  - 대부분의 웹 기반 애플리케이션과 파일 서버, 전송 프로토콜 포함

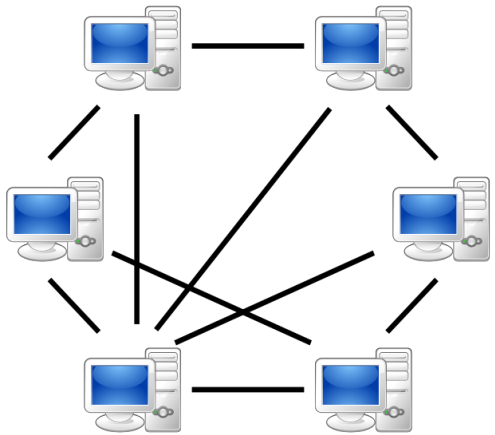


# 2.3 클라이언트/서버 구조 (2/2)

- 장점
  - 데이터 집중화 : 모든 데이터는 서버에서 집중 관리하므로 구성과 관리를 단순화
  - 보안 : 모니터링과 인증을 통한 접근제어
- 단점
  - 병목현상 : 서버의 부하로 인한 네트워크 속도 저하
  - 비용 : 설치 및 관리 비용
  - 비강인성 : 서버의 고장으로 인한 서비스 불가

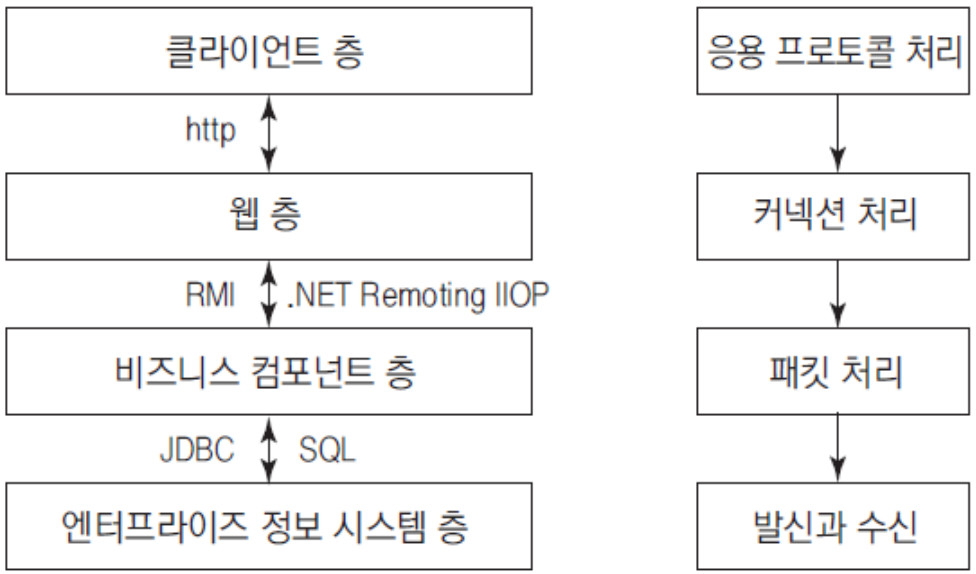
# 2.4 Peer-to-Peer

- 클라이언트 서버 유형을 일반화 시켜 놓은 것
  - 각 서브시스템이 클라이언트 또는 서버로 동작할 수 있음
  - 각 서브시스템이 서비스를 요청/제공할 수 있음
- 클라이언트 서버 시스템보다 설계가 어려움
  - 데드록(deadlock)의 가능성과 제어구조의 복잡성 때문



# 2.5 계층(layered) 구조

- SW 기능을 상호작용하는 여러 수직층으로 분할
  - 인접한 층 사이에서만 메시지 교환
  - 장점 : 추상화, 캡슐화, 재사용이 용이
  - 단점 : 이웃 계층과의 통신이 매우 제한적



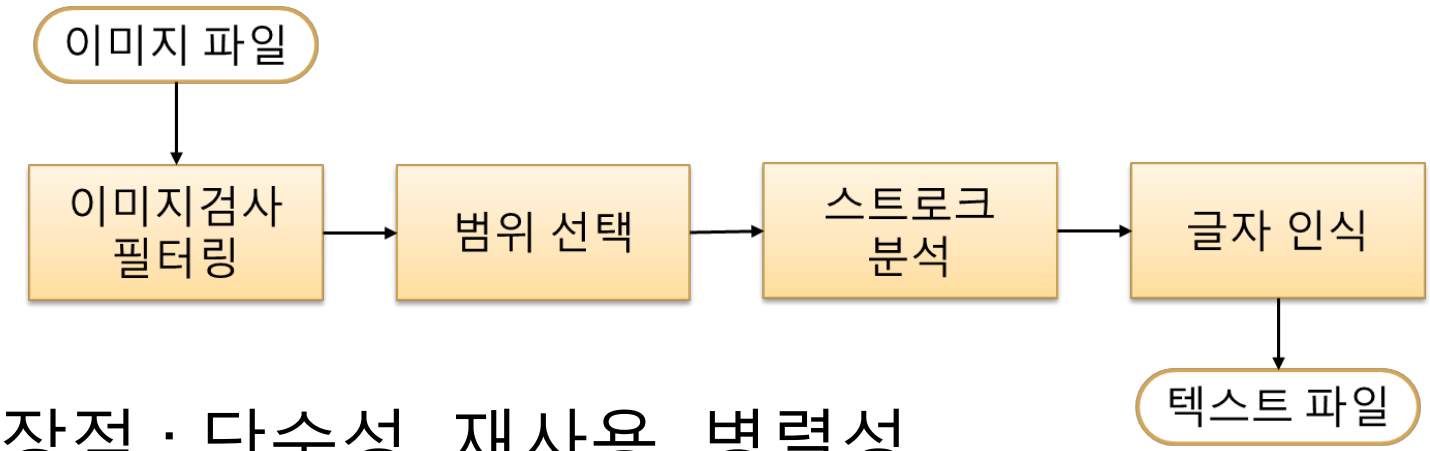

(a) 애플리케이션 서버 아키텍처 (b) 통신 시스템

# 2.6 파이프필터 구조 (1/2)

- 서브시스템이 입력 데이터를 받아 처리하고 결과를 다른 시스템에 보내는 작업이 반복
- 필터 사이에 데이터를 이동시키면서 단계적으로 처리
  - 서브시스템 = 필터
  - 서브시스템 사이의 관계 = 파이프
  - 서브시스템이 입력 데이터를 받아 처리하고 결과를 다른 시스템에 보내는 작업이 반복
  - 사용자 개입없이 데이터 스트림 변형시키기에 적합

# 2.6 파이프필터 구조 (2/2)

- 복잡한 상호작용이 필요한 정보관리시스템이나 인터랙티브 시스템에는 적합하지 않음




- 장점 : 단순성, 재사용, 병렬성
- 단점 :
  - 자원의 낭비 : 데이터가 여러곳에 저장되고, 각 필터는 입/출력 형식으로 변환해야 함

# 3. 설계 패턴 개요 (1/5)

- 패턴(Pattern) :실세계에 존재하는 사물이나 개념에서 반복적으로 발생하는 형상을 추상화하여 정의한 것
  - 설계 패턴, 아키텍처 패턴, 분석 패턴, 구현 패턴, 사용자 인터페이스 패턴, 작업 패턴 등
- 패턴 사용의 장점
  - 생산성 증가, 전문가 경험의 전달 및 학습 효과
  - 솔루션에 대한 불필요한 논쟁 제거, 결과물의 품질 향상

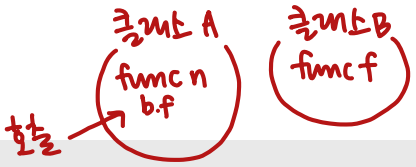
# 3. 설계 패턴 개요 (2/5)

- 디자인 패턴(design pattern)이란?
  - 소프트웨어 설계에서 자주 발생하는 문제에 대한 일반적이고 반복적인 해결책을 말함
  - 많은 개발자들이 경험상 체득한 설계 지식을 검증하고 이를 추상화하여 일반화한 템플릿

# 3. 설계 패턴 개요 (3/5)

- 디자인 패턴을 사용하는 이유
  - 쉽게 재사용 가능 ⇒ 검증된 것을 재사용함으로써 생산성을 높일 수 있음
  - 설계 작업이 쉬워짐
  - 설계 관련 지식이 정리됨
  - 객체지향 설계 원리를 잘 따르게 됨
  - 디자인을 논의하기 위한 의사소통이 쉬워짐

- 디자인 패턴의 원리 = 위임과 상속



# 3. 설계 패턴 개요 (4/5)

- 설계패턴을 정의하기 위한 형식

| 구분 | 내용 |
|---|---|
| 패턴 명 | 패턴 자체의 내용을 효과적으로 전달할 수 있는 이름 |
| 유형 | 여러 개의 패턴을 체계적으로 분류<br>• 생성(Creational): 객체들의 생성과 관련된 패턴<br>• 구조(Structure): 클래스, 객체의 정적인 구조와 관련된 패턴<br>• 행위(Behavioral): 클래스와 객체의 반응과 책임 할당 |
| 목적 | 이 패턴이 무엇을 하며 어떤 의도로 작성되었는지 무엇을 해결하는지를 설명하여 기술 |
| 별칭 | 위의 공식적인 이름 외에 잘 알려진 다른 명칭 |
| 구조 | 패턴 안에서 문제를 해결하기 위해 사용되는 클래스와 객체의 구조를 UML 다이어그램으로 표현 |
| 구성 요소 | 구조 항목에 포함된 각종 클래스, 객체의 의미와 그 책임을 설명 |
| 협력 과정 | 각 클래스와 객체가 자신에게 맡겨진 책임을 수행하기 위해 서로 메시지를 주고받는 과정을 묘사 |
| 구현 | 패턴을 구현할 때 고려 사항과 힌트, 함정, 방법, 프로그래밍 언어별 주의할 점 등을 기술함 |
| 샘플 코드 | 특정 언어로 패턴을 구현한 예제, 실제로 사용되는 시스템에서 발견되는 패턴의 예제 |
| 솔루션 | 패턴이 목적을 달성하기 위해 어떤 면을 해결하는지 설명하고 패턴을 적용할 때 발생할 수 있는 문제점과 패턴 적용 시 효과 등을 기술 |
| 관련 패턴 | 본 패턴과 유사하거나 밀접하게 관련된 다른 패턴 |

# 3. 설계 패턴 개요 (5/5)

gang of four

- 디자인 패턴의 분류 : GoF[1]의 23개 패턴

| 영역 | | 목적 | | |
|---|---|---|---|---|
| | | 생성 패턴 (Creational Pattern) | 구조 패턴 (Structural Pattern) | 행위 패턴 (Behavior Pattern) |
| 의미 | | • 객체의 생성 방식을 결정<br>• 클래스의 정의, 객체 생성 방식의 구조화, 캡슐화 | • 객체를 조직화하는 일반적인 방법 제시<br>• 별도의 클래스 라이브러리를 통합하는 데 유용하며 런타임 시에 객체가 구성된 구조를 변경 가능<br>• 객체 구성에 유동성과 확장성의 추가가 가능 | • 객체의 행위를 조직화 관리, 연합하고 객체나 클래스의 연동에 대한 유형을 제시하고자 할 때 사용되며 런타임 시 복잡한 제어 흐름을 결정짓는 데 사용 |
| 범위 | 클래스 | Factory Method | Adapter(Method) | Interpreter/Template |
| | 객체 | Abstract Factory<br>Builder<br>Prototype<br>Singleton | Adapter<br>Bridge<br>Composite<br>Decorator<br>Facade<br>Proxy<br>Flyweight | Command<br>Iterator, Mediator<br>Memento, Observer<br>State, Strategy<br>Visitor,<br>Chain of Responsibility |

1) *Design Patterns: Elements of Reusable Object-Oriented Software* (1994), E. Gamma, R. Helm, R. Johnson, J. Vlissides.

# 3.1 생성 패턴

- 생성 패턴
  - 클래스의 인스턴스가 어떻게 생성되는가를 추상화하여 나타낸 것
  - 객체가 생성, 합성, 표현되는 방식에 관계없이 시스템을 독립적으로 생성할 수 있도록 지원

| 생성 패턴 | 적용 측면 |
|---|---|
| Abstract Factory | 제품 군의 유사 객체들을 생성하는 방법 제공 |
| Builder | 유사한 다수 객체들을 생성하여 합성하는 방법 제공 |
| Factory Method | 객체 생성 시점을 하위 클래스가 결정할 수 있도록 제공 |
| Prototype | 생성 객체를 복사 및 변경하여 객체를 생성하는 방법 제공 |
| Singleton | 오직 하나의 객체만 생성 |

# 3.1 생성 패턴 – 싱글톤 (1/2)

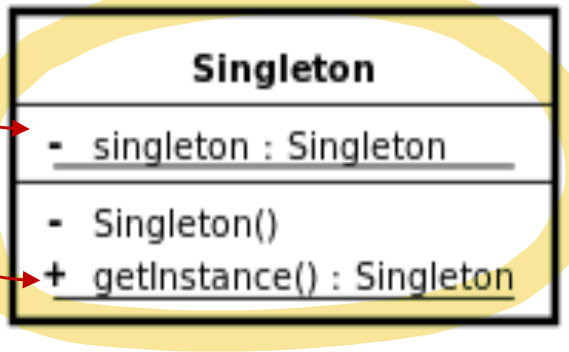
- 싱글톤(Singleton) 패턴
  - 한 클래스의 인스턴스가 단지 하나만 생성될 수 있도록 보장하기 위해 사용
- 문제 및 동기
  - 하나뿐인 장치를 다수가 공유하고자 할 때, 해당 클래스의 인스턴스를 다수가 요청할 때마다 생성하면 다수의 장치가 존재하는 것으로 잘못 나타나는 문제가 발생함
  - 요청할 때마다 생성되는 장치 객체의 동일한 주소 값을 반환 ⇒ 하나의 객체만 사용할 수 있게 함

# 3.1 생성 패턴 – 싱글톤 (2/2)

- 적용 대상
  - 정확히 하나의 인스턴스만 생성되어야 하는 경우
  - 예: DB 커넥션을 위한 인터페이스
- 패턴구조
  - 클래스 자체를 정적변수로, 생성자는 private으로 선언
  - 유일한 객체를 접근하는 정적 메소드를 둠

static 속성으로 정의

static 메소드로 정의




11/15 코드사진참고

11/13

# 3.2 구조 패턴

- 클래스의 기본 구조를 확장하여 더 큰 구조를 제공할 수 있도록 클래스와 객체를 어떻게 구성하는지를 표현
  - 새로운 기능을 제공하기 위하여 어떻게 기존의 객체를 합성할 것인가를 설명함

| 구조 패턴 | 적용 측면 |
|---|---|
| Adapter | 객체의 인터페이스를 융통성 있게 구성하는 방법 제공 |
| Bridge | 객체의 인터페이스와 구현을 분리하는 방법 제공 |
| Composite | 객체의 구조와 서로 다른 객체를 합성하는 방법 제공 |
| Decorator | 하위 클래스 생성 없이 객체의 기능을 확장하는 방법 제공 |
| Facade | 서브시스템에 대한 통합 인터페이스를 구성하는 방법 제공 |
| Flyweight | 객체의 저장소 접근 및 활용 방법 제공 |
| Proxy | 객체의 접근 방법 및 위치에 대한 구성 방법 제공 |

# 3.2 구조 패턴 – 어댑터 (1/3)

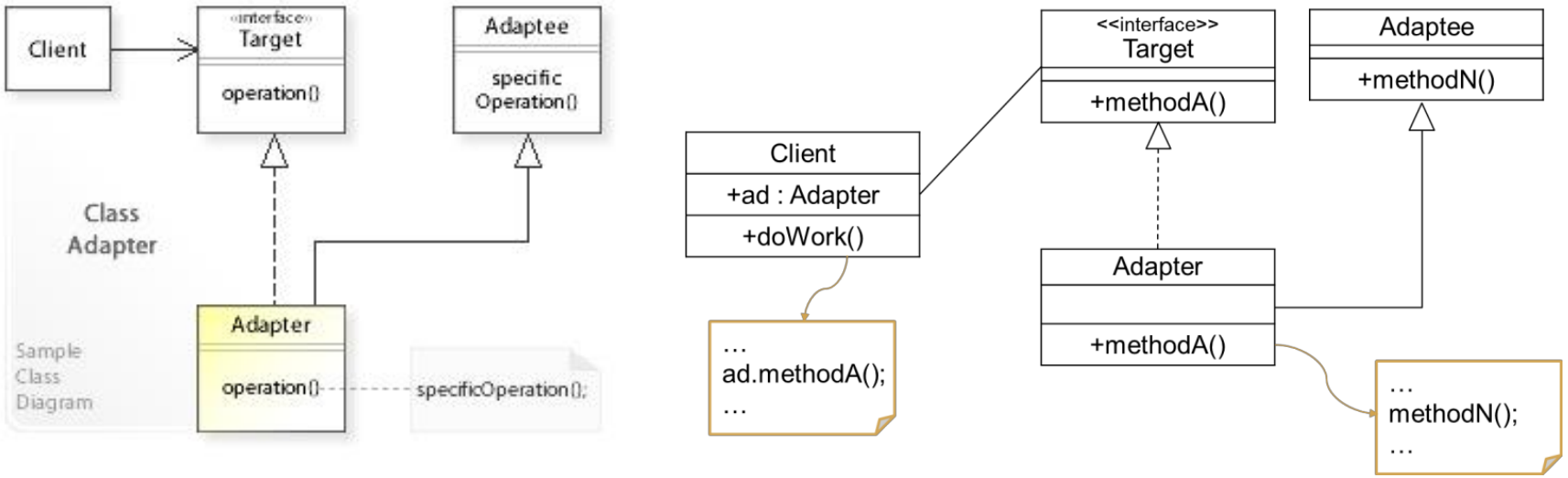
- 어댑터(Adapter) 패턴
  - 클래스의 인터페이스를 클라이언트가 기대하는 다른 인터페이스로 변환하는 방법 제공
  - 호환되지 않는 인터페이스로 인해 사용할 수 없었던 클래스에 어댑터를 사용하면 함께 작동할 수 있도록 지원함
- 문제 및 동기
  - 응용 프로그램이 요구하는 인터페이스와 Toolkit에 정의된 인터페이스가 일치하지 않아 Toolkit에서 제공하는 클래스를 재사용하기 어려운 상황이 발생함

# 3.2 구조 패턴 – 어댑터 (2/3)

- 적용 대상
  - 기존 클래스를 사용해야 하나 인터페이스가 수정되어야 하는 경우
  - 이미 만들어진 클래스를 재사용하고자 하나 이 재사용 가능한 라이브러리를 수정할 수 없는 경우
- 어댑터 클래스
  - 사용하고자 하는 서비스에 연관되어 있고, 클라이언트가 원하는 인터페이스를 구현한 클래스

# 3.2 구조 패턴 – 어댑터 (3/3)

- 패턴 구조
  - 클래스 Target과 클래스 Adaptee를 다중 상속받아 클래스 Adapter를 정의
  - 클래스 Target을 상속받은 Adapter가 Adaptee 클래스의 인터페이스를 제공



# 3.3 행위 패턴

- 객체에 주어지는 기능 및 책임과 관련 있으며, 객체 간의 상호작용을 표현
  - 행위 패턴은 상속을 통해서 클래스에 존재하는 동작을 전달(배포)

| 구조 패턴 | 적용 측면 |
|---|---|
| Chain of Responsibility | 서비스 요청을 충족하는 객체를 찾는 방법 제공 |
| Command | 언제 어떻게 서비스를 제공할 것인가에 대한 방법 제공 |
| Interpreter | 언어 문법과 해석에 대한 방법 제공 |
| Iterator | 집합 요소들의 접근에 대한 방법 제공 |
| Mediator | 객체 간 상호작용을 단순화하는 빙법 제공 |
| Memento | 객체 외부에 저장되는 정보를 다루는 방법 제공 |
| Observer | 의존 객체의 행동을 관찰하는 수단 제공 |
| State | 객체 상태를 표현하고 상태에 따라 동작하는 방법 제공 |
| Strategy | 다수의 구현 알고리즘의 선택을 지원하는 방법 제공 |
| Template Method | 한 알고리즘의 처리 단계를 재구성하는 방법 제공 |
| Visitor | 클래스 변경 없이 연산을 적용하는 방법 제공 |

# 3.3.1 옵저버 패턴 (1/3)

- 옵저버(Observer)
  - 객체 간에 일대다 의존성 관계를 부여하고, 한 객체의 상태가 변경되었을 때 다른 객체들에게 자동으로 알려주어 필요한 변경을 수행하고자 할 때 사용하는 패턴
- 문제 및 동기
  - 어떤 객체에 변화가 일어났을 때 이를 감지하여 다른 객체에 통보해주는 방법
  - 예) 엑셀 시트의 데이터 값을 변경하는 경우 그래프 객체의 내용이 자동 변경되는 경우

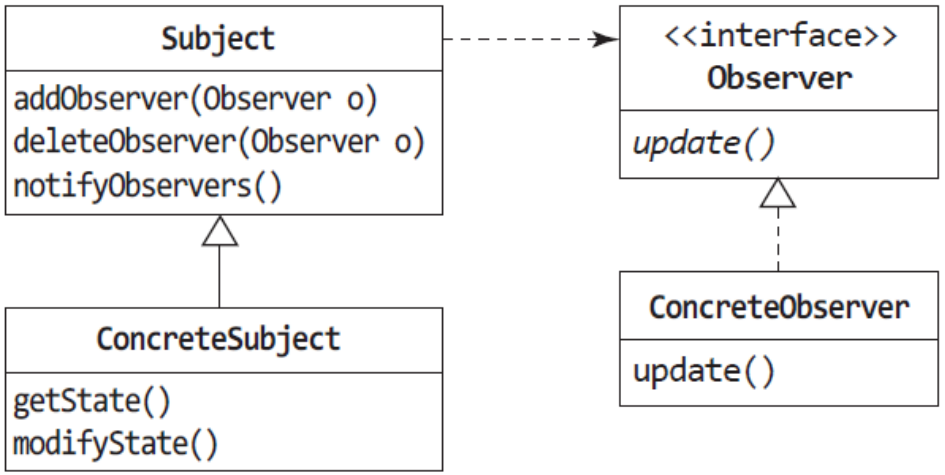
# 3.3.1 옵저버 패턴 (2/3)

- 적용 대상
  - 한 객체의 변경이 발생하면 다른 객체가 동시에 변경되어야 하는 경우

- 발행-구독(publish-subscribe) 패턴
  - 데이터를 보관하고 있는 Subject가 그 데이터를 이용하는 옵서버와 통신하면서 느슨하게 결합

# 3.3.1 옵저버 패턴 (3/3)

- 패턴 구조
  - Subject 클래스 : 옵저버 목록을 유지, 변경을 고지
  - Observer 클래스 : 변경을 통지 받고 접근을 요청
  - ConcreteSubject 클래스 : 객체가 변경된 것을 옵저버에게 통보하며, ConcreteObserver는 갱신 인터페이스를 구현



# 3.3.2 비지터 패턴 (1/3)

- 비지터(Visitor) 패턴
  - 객체의 구성 요소에 대응하여 수행되어야 하는 연산을 정의함으로써, 기존 클래스에 대한 변경 없이 새로운 연산을 쉽게 추가할 수 있도록 지원하는 패턴
- 데이터 구조와 처리를 분리
  - 데이터 구조 안을 돌아다니는 주체인 방문자를 나타내는 클래스를 준비해서 처리를 맡김
  - 새로운 처리를 추가하고 싶을 땐 새로운 방문자를 만들고 데이터 구조는 방문자를 받아들이면 됨

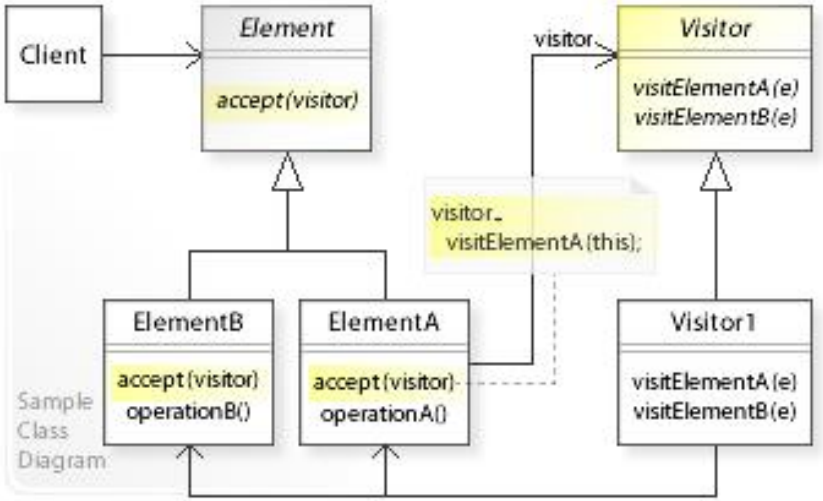
# 3.3.2 비지터 패턴 (2/3)

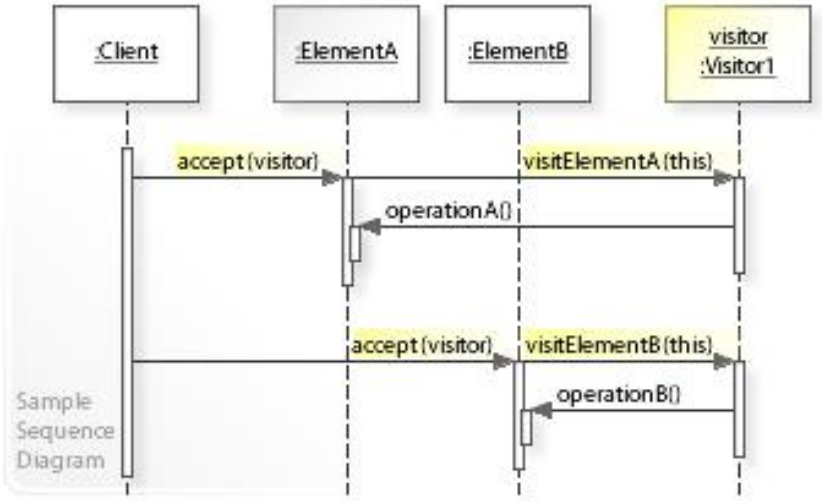
- 적용 대상
  - 기존 객체에 대한 변경 없이 새로운 행위를 추가하거나 다양한 작업을 수행해야 하는 경우

- 개방폐쇄원칙(OCP, Open-Closed Principle)을 적용하는 방법의 하나
  - 소프트웨어 개체(클래스, 모듈, 함수 등등)는 확장에 대해 열려 있어야 하고, 수정에 대해서는 닫혀 있어야 한다는 프로그래밍 원칙

# 3.3.2 비지터 패턴 (3/3)

- 패턴 구조
  - 클래스 Visitor는 객체의 구성 요소에 대응하는 각 클래스의 Visit 연산을 선언 → visit(ele A), visit(ele B), … 새로운 구조는 visit만 추가
  - 클래스 Element는 인자로 Visitor 객체를 갖는 Accept 연산을 정의



# 4. 패턴이 제공하는 장점

- 설계 결과물의 품질을 향상 시킴으로써, 개발된 SW 시스템의 확장성과 유지보수성이 높아짐
- 특정 요구사항과 도메인 클래스에 부합되는 패턴을 쉽게 찾을 수 있다면, 설계 활동의 시간과 노력이 감소됨
- 설계 패턴을 사용하면 개발자 간의 정보 교환 및 의사소통을 위한 공통의 용어를 확보할 수 있음
- 설계 패턴을 충분히 잘 이해하고 있다면, 모델에 대한 리팩토링(refactoring) 대상 찾기가 쉬워짐

↑ 재캡화

# 5. 설계 문서화 (1/2)

- 아키텍처 문서를 작성하는 이유
  - 설계자나 설계 팀이 더 좋은 설계 결정을 내리는데 도움을 준다.
  - 다른 사람과 설계에 대하여 이야기를 할 수 있다.
- 설계문서는 다음 세가지 그룹이 의사 교환하는데 사용
  - 설계를 구현할 사람 즉 프로그래머
  - 미래에 설계를 변경할 엔지니어
  - 설계된 시스템과 인터페이스 할 다른 시스템 또는 서브시스템을 개발하는 엔지니어

# 5. 설계 문서화 (2/2)

1. 개요 기초적설계
   - 1.1 시스템의 목표
   - 1.2 하드웨어와 소프트웨어
   - 1.3 소프트웨어의 주요기능
   - 1.4 설계상 제약사항
   - 1.5 참조된 개발 문서
2. 시스템구조
   - 2.1 시스템구조 개요
   - 2.2 시스템구조도
   - 2.3 자료사전
3. 모듈설계 (각 모듈에 대한)
   - 3.1 모듈 이름
   - 3.2 알고리즘
   - 3.3 인터페이스
   - 3.4 오류 메시지
   - 3.5 사용하는 파일
   - 3.6 호출하는 모듈
   - 3.7 기능 설명
4. 파일 구조 또는 데이터베이스 설계
   - 4.1 외부파일(데이터베이스)의 논리적 구조
   - 4.2 공유자료
   - 4.3 파일접근방식(데이터베이스 관리 체제)
5. 요구분석 참조 표
6. 제약 사항
7. 참고 사항

참고 문헌

부록

요구분석 참조 표

| 번호 | 기능/성능 요구항목 | 문서면 | 모듈이름 |
|---|---|---|---|
| 1 | 갱신 정보 추출 | pp.12 – 14 | EXTRACT_UPDATE |
| 2 | 만료일 변경 | pp.27 | CHANGE_EXPIRE_DATE<br>RESTORE_RECORD |
| … | … | … | … |