😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

😀아키텍처 설계

📖 p166. 아키텍처 VS 컴포넌트

요구공학 사이에 중요한 연결

- 아키텍처 설계는 주요 구조 컴포넌트들과 그들 간의 관계를 식별
- 요구공학 프로세스와 아키텍처 설계 사이에는 중첩되는 부분이 상당히 많다.

/* 애자일 프로세스에서는 일반적으로 애자일 개발 프로세스의 초기 단계가 전체 시스템 아키텍처 설계에 초점을 맞추어야 한다는 것을 받아들인다. */

컴포넌트 리팩토링	**아키텍처 리팩토링**
• 변화에 대응하여 **컴포넌트를 리팩토링**하는 것은 **비교적 쉽다**.	• 아키텍처의 점진적인 개발은 보통 성공적이지 않다. • 변화에 대응하여 **시스템 아키텍처를 리팩토링**하는 것은 대부분의 시스템 컴포넌트들을 아키텍처 변화에 따라 수정해야 할 수 있기 때문에 **비용이 많이 든다.**

📖 p167. 아키텍처 2가지 분류

작은수준 아키텍처	**개별 프로그램의 아키텍처**와 관련이 있다. 이 수준에서는 개별 프로그램을 컴포넌트들로 분해하는 방법과 관련되어 있다.
큰 수준 아키텍처	**다른 시스템들과 프로그램 컴포넌트를 포함**하는 복잡한 기업 시스템과 연관되어 있다. 이와 같은 기업 시스템은 서로 다른 회사에서 소유하고 관리하는 컴퓨터들에 분산되어 있을 수도 있다. (이부분 안중요)

소프트웨어 아키텍처는 **시스템의 성능, 견고성, 분산성, 유지보수성**에 영향을 주므로 중요하다.

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

📜 소프트웨어 아키텍처를 명시적으로 설계하고 문서화하는 것의 세 가지 장점

1. **이해당사자 간의 의사소통**
    - 상위 수준의 시스템 표현 ->
      다양한 범위의 이해당사자들간 논의의 중심으로 사용 가능

2. **시스템 분석**
    - 시스템 개발의 초기 단계에 시스템 아키텍처를 명시적으로 만드는 것은 어느 정도 분석이 필요하다.

3. **대규모 재사용**
    - 시스템 아키텍처는 비슷한 요구사항을 가진 시스템과 종종 동일하다.

📖 p168. 블록 다이어그램

블록 다이어그램

- 특징
    - 소프트웨어 설계 프로세스에 관여하는 사람들 사이의 의사소통을 지원한다.
    - 개발 프로세스에 관여하는 서로 다른 분야의 사람들이 시스템 구조를 쉽게 이해할 수 있는 상위 수준의 그림으로 보여준다.
        - = 직관적이다.

다이어그램의 **각 상자**는 컴포넌트를 나타낸다.
화살표는 데이터 또는 제어신호가 컴포넌트에서 다른 컴포넌트에 화살표 방향으로 전달되는 것을 의미

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

😎아키텍처 설계 결정

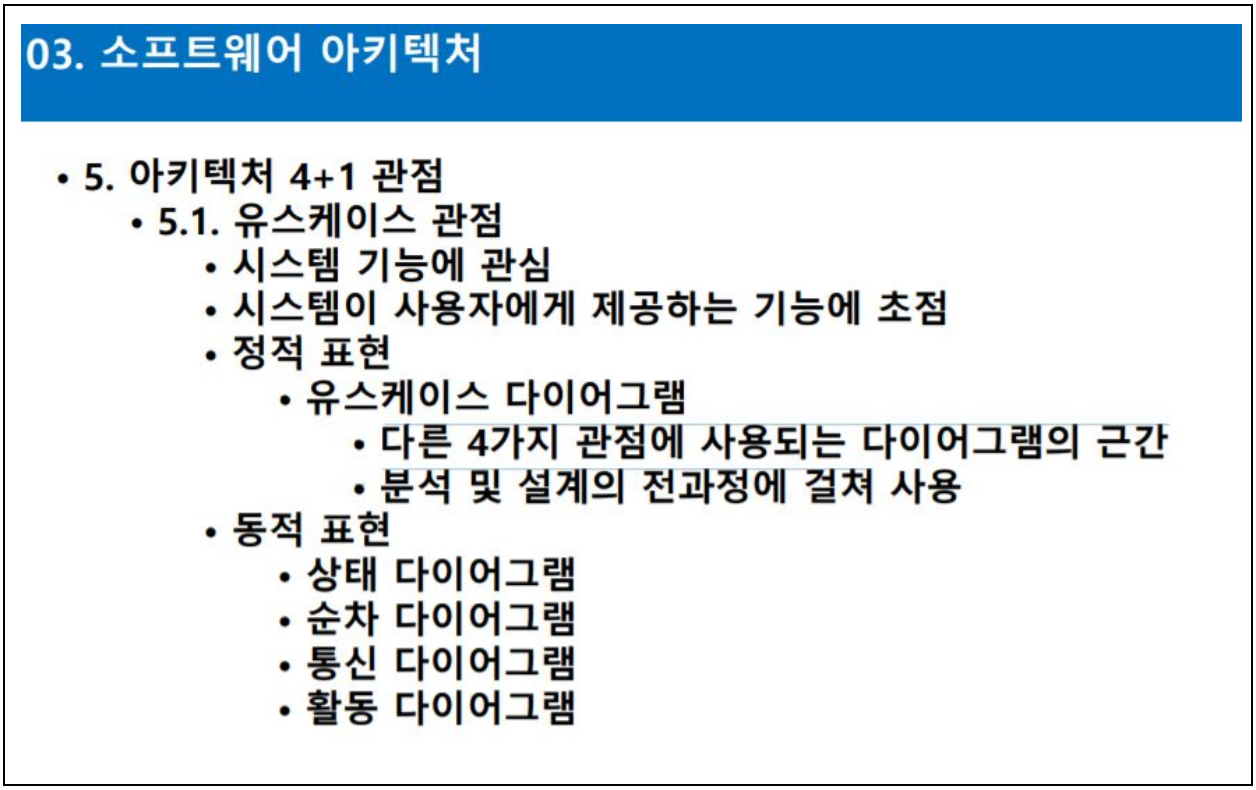
📖 p169. 아키텍처 설계란?

아키텍처 설계

- 시스템의 기능적, 비기능적 요구사항을 만족시키기 위한 시스템 구조를 설계하는 창의적인 프로세스이다.
  /* 설계 프로세스에 정해진 방법은 없다. */
- 일련의 활동이라기 보단 연속적인 결정이다.

시스템 아키텍트

- 아키텍처를 설계하는 사람



😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

📖 p170. 비기능적 특성과 SW아키텍처

소프트웨어 시스템의 아키텍처는 **특정 아키텍처 패턴 또는 스타일**에 기반을 둘 수 있다.

- 시스템 아키텍처 패턴 = 시스템 아키텍처 스타일

비기능적 특성과 소프트웨어 아키텍처 간에는 밀접한 관계

- 아키텍처 스타일과 구조의 선택은 시스템의 **비기능적 요구사항에 좌우된다.**

> 📌 **외우는 팁! : 보성 안가유~** (**보**안 **성**능 **안**전 **가**용 **유**지보수)

1. **성능**
   - **성능**이 중요한 요구사항
     - **분산된 것보다 같은 컴퓨터에** 배치된 소수의 컴포넌트 안에 중요한 작업이 모이도록 아키텍처가 설계되어야 한다.
2. **보안성**
   - **보안**이 중요한 요구사항
     - **아키텍처에 계층 구조가 사용**
     - **가장 중요한 자산이 가장 안쪽** 계층에서 보호되고 **높은 수준의 보안 검증**이 이계층에 적용
3. **안전성**
   - **안전**이 중요한 요구사항 (Mission Critical한것)
     - 단일 컴포넌트나 소수의 컴포넌트에 같이 배치되도록 아키텍처가 설계
4. **가용성**
   - **가용성**이 중요한 요구사항
     - 중복 컴포넌트를 포함하여 시스템의 중단 없이 컴포넌트를 교체하고 갱신하는 것이 가능하게 아키텍처 설계
     - Ex) 결함내성 시스템 아키텍처
5. **유지보수성**
   - **유지보수성**이 중요한 요구사항
     - 변경이 용이한 세분화되고 독립적인 컴포넌트를 사용되도록 시스템 아키텍처가 설계되어야 한다.
     - **[데이터 생산자는 소비자와 분리]**되어야 하며
       [데이터 구조를 공유하는 것은 피해야 한다.]

아키텍처들 사이에서는 **잠재적인 충돌**이 있다.

- 성능과 유지보수성 둘 다 중요한 시스템 요구사항이면 어느 정도 **타협점(trade off)**을 찾아야 한다.
  💡 **서로 다른 아키텍처 패턴이나 스타일을 사용함**으로써 이것을 할 수 있다.

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

😎아키텍처 뷰

📖 p172. 아키텍처 뷰 종류

네가지 기본적인 아키텍처 뷰

> 📌 **외우는 팁! : 프로세스 개논물~** (**프**로세스 **개**발(개념) **논**리 **물**리)

1. **논리적 뷰**
   - **객체** 또는 **객체 클래스**로 시스템의 핵심 추상화를 보여줌
   - 시스템 요구사항을 논리적 뷰의 개체들에 연관 짓는 것이 가능
2. **프로세스 뷰**
   - 런타임에 시스템이 어떻게 상호 작용하는 프로세스들로 구성되는지 보여줌
   - **성능**이나 **가용성**과 같은 **비기능적 시스템 특성들을 판단하는데 유용**
3. **개발 뷰**
   - 소프트웨어가 개발을 위해 어떻게 분해되는지를 보여줌
   - 개발자에게 유용
4. **물리적 뷰**
   - 시스템 하드웨어와 소프트웨어 컴포넌트들이 시스템의 프로세서에 어떻게 분산되는지를 보여줌
   - 시스템 배치를 계획하는 시스템 엔지니어에게 유용

\+ **개념적 뷰**

- 시스템 아키텍처를 이해당사자들에게 설명하고 아키텍처 상의 의사결정을 알리는데 사용.

[💡변별력] 아키텍처 기술언어 : UML, ADL

- 5. 아키텍처 4+1 관점
  - 5.2. 논리적 관점
    - 시스템의 내부에 관심
    - 시스템의 기능을 제공하기 위한 클래스/컴포넌트의 종류/관계에 초점
    - 정적 표현
      - 클래스 다이어그램
      - 객체 다이어그램
    - 동적 표현
      - 상태 다이어그램
      - 순차/통신 다이어그램
      - 활동 다이어그램

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

- 5. 아키텍처 4+1 관점
  - 5.3. 구현 관점
    - 소프트웨어 서브시스템의 모듈의 구조화에 관심
      - 모듈: 원시코드, 데이터 파일, 컴포넌트, 실행 파일 등
    - 정적 표현
      - 컴포넌트 다이어그램
    - 동적 표현
      - 상태 다이어그램
      - 순차 다이어그램
      - 통신 다이어그램
      - 활동 다이어그램

- 5. 아키텍처 4+1 관점
  - 5.4. 프로세스 관점
    - 실제 구동 환경에 관심
    - 시스템 내부의 구조에 관심
      - 클래스 간의 관계, 클래스의 동작, 클래스 간 상호작용
    - 개발자와 시스템 통합자를 위함.
    - 동적 표현
      - 상태 다이어그램
      - 순차 다이어그램
      - 협동 다이어그램
      - 활동 다이어그램
    - 시스템 구성 표현
      - 컴포넌트 다이어그램
      - 배치 다이어그램

- 5. 아키텍처 4+1 관점
  - 5.5. 배치 관점
    - 시스템을 구성하는 처리 장치 간의 물리적인 배치에 관심
    - 서브시스템들의 연관성과 실행을 노드 간의 관계로 표현
    - 정적 표현
      - 배치 다이어그램
    - 동적 표현
      - 상태 다이어그램
      - 순차 다이어그램
      - 통신 다이어그램
      - 활동 다이어그램

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

😎아키텍처 패턴

📖 p173. 아키텍처 패턴이란?

아키텍처 패턴

- 소프트웨어 시스템에 대한 지식을 표현하고 공유하고 **재사용하는 방법**

MVC 패턴	**웹 기반 시스템에서 상호작용 관리의 기반** 대부분 언어 프레임워크에서 지원된다. 시스템 데이터로부터 표현과 상호 작용을 분리시킨다.

- 7.4. MVC 모델(Model, View, Controller Model)
  - 중앙 데이터 구조
  - 여러 개의 뷰 서브시스템을 필요로 하는 상호작용 시스템에 적합
  - 구성
    - 모델(Model) 서브시스템
      - 모든 데이터 상태와 로직을 처리하고, 호출에만 응답
    - 뷰(View) 서브시스템
      - 모델 서브시스템이 제공한 데이터를 사용자에게 보여줌
    - 제어(Controller) 서브시스템
      - 뷰와 모델 사이에 전달자 역할

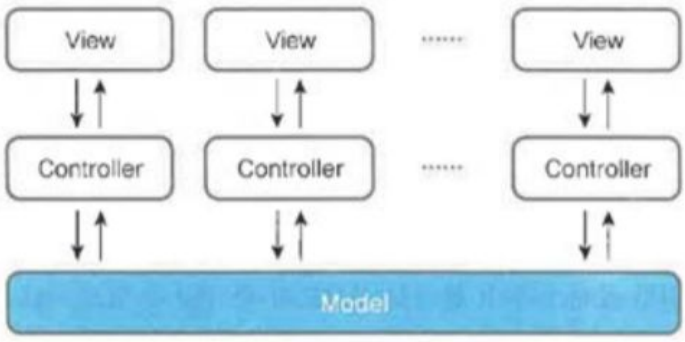


그림 5-26 MVC 모델

참고 PDF http://kocw.xcache.kinxcdn.com/KOCW/document/2018/wonkwang/leesangwon0405/5.pdf

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

패턴 종류들	• Strategy • Decorator • Factory • Observer • Chain of responsibility • Singleton • Flyweight • Adapter • Façade • Template method • Builder • Iterator • Composite • Command • Mediator • State • Proxy • Abstract factory • Prototype • Bridge • Interpreter • Memento • Visitor

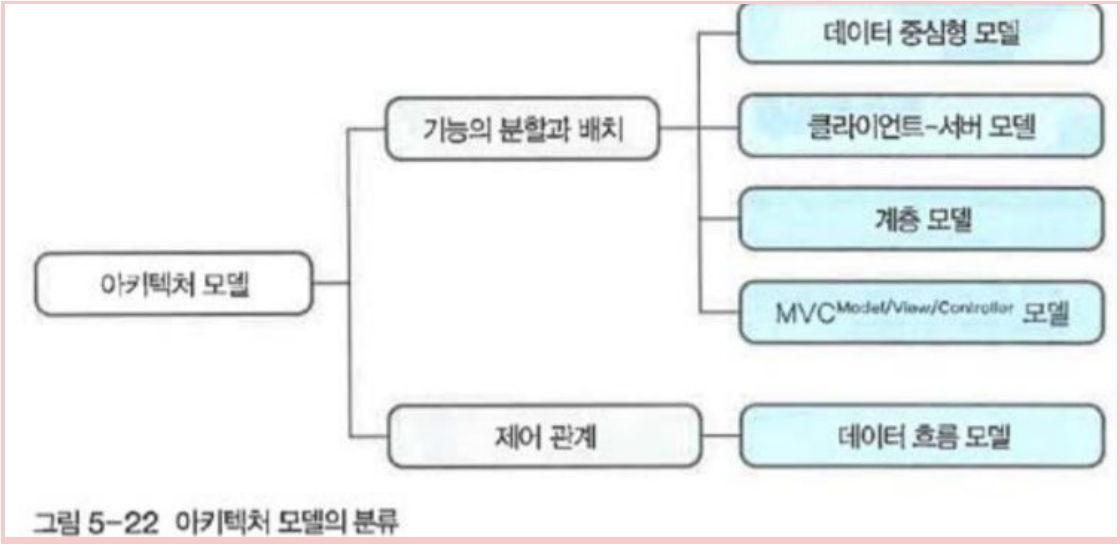


그림 5-22 아키텍처 모델의 분류

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

03. 소프트웨어 아키텍처

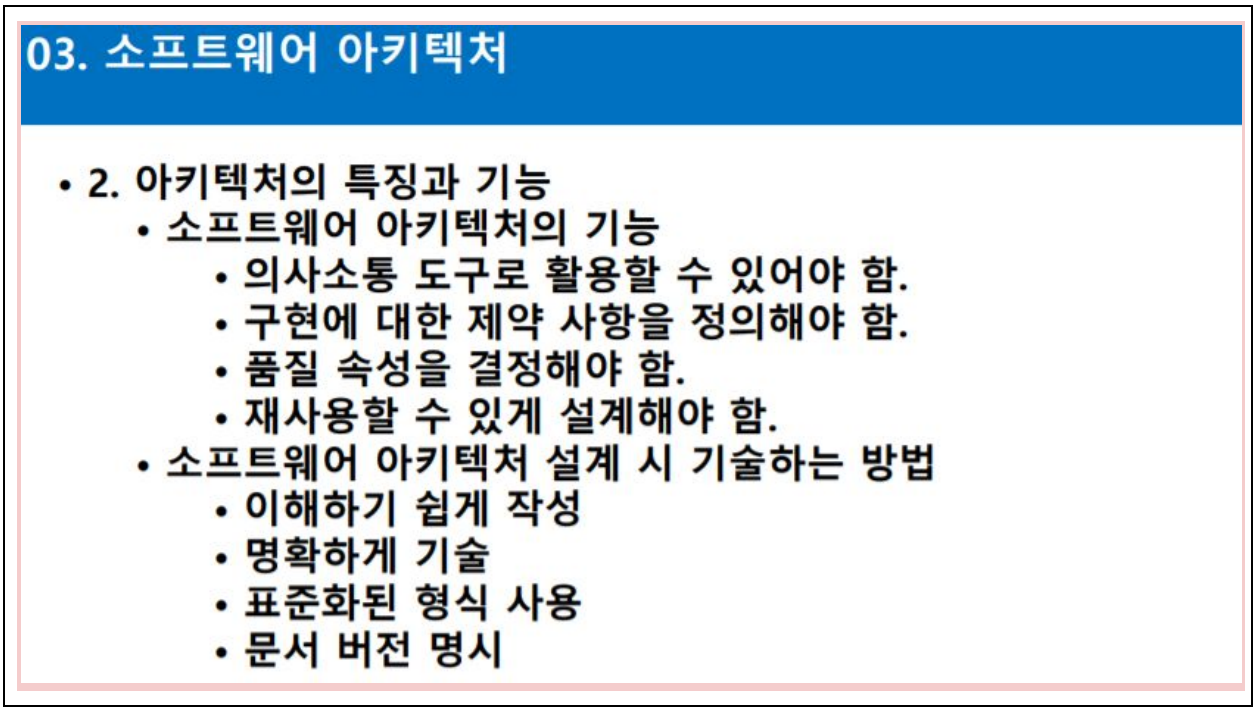
- 2. 아키텍처의 특징과 기능
  - 소프트웨어 아키텍처의 정의
    - 외부에서 인식 가능한 특성이 담긴 소프트웨어의 골격이 되는 기본 구조
    - 개발할 소프트웨어의 구조, 구성요소, 구성 요소의 속성, 구성 요소 간의 관계와 상호작용을 판단하고 결정하는 것
    - 다루는 내용
      - 구성 요소
      - 구성 요소들 사이의 관계
      - 구성 요소들이 외부에 드러내는 속성
      - 구성 요소들과 주변 환경 사이의 관계
      - 구성 요소들이 제공하는 인터페이스
      - 구성 요소들의 협력 및 조립 방법

04. 디자인 패턴

- 1. 디자인 패턴의 이해
  - 디자인 패턴의 정의
    - 자주 사용하는 설계 형태를 정형화 하여 유형별로 만든 설계 템플릿
  - 장점
    - 효율성과 재사용성 제고
    - 개발자(설계자) 간의 원활한 의사소통
    - 소프트웨어 구조 파악 용이
    - 재사용을 통한 개발시간 단축
    - 설계 변경 요청에 대한 유연한 대처
  - 단점
    - 객체지향 설계/구현 위주
    - 초기 투자 비용 부담

그림 5-28 디자인 패턴의 예

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션



😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

—— 📜 계층 아키텍처 ——

📖 p176.

분리와 독립의 개념은 **변경의 지역화를 가능**하게 하므로 아키텍처 설계의 기본이다.

- **MVC 패턴**은 시스템의 요소들을 분리하여 독립적으로 변경될 수 있도록 하였다.
- **시스템 기능은 분리된 계층들로 구성**되고 각 계층은 바로 아래 계층에서 제공하는 기능과 서비스에만 의존한다.

계층적인 접근은 시스템의 **점증적 개발을 지원한다**.

- 변경하지 않고 확장된 기능을 가지는 새로운 계층으로 기존의 계층을 대체가능.

계층 아키텍처	
설명	• **시스템을 각 계층으로 표현** • 각 계층은 상위 계층에 서비스를 제공 • 가장 아래계층은 시스템 전체에서 사용되는 핵심 서비스를 나타냄
언제사용	• 새로운 기능을 기존 시스템 위에 구축할 때 • 개발을 여러 팀으로 나누어 하고 각 팀이 기능 계층에 대한 책임이 있을 때 • 다중수준 보안이 필요할 때
장점	**인터페이스가 유지된다면 전체계층을 대체하는것이 가능하다.** 시스템의 확실성을 향상시키기 위해 중복된 기능이 각 계층에 제공될 수 있다.
단점	**현실적으로 계층들을 명확하게 분리하는 것이 종종 어렵다.** 상위 수준 계층이 바로 아래 계층을 통하지 않고 하위 수준계층과 직접 상호작용해야 할 수 있다.

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

- 7.3. 계층 모델(Layering Model)
  - 기능을 몇 개의 계층으로 나누어 배치
  - 상위 계층은 클라이언트, 하위 계층은 서버 역할
  - 구성
    - 프레젠테이션(사용자 인터페이스) 계층
    - 비즈니스 로직(애플리케이션) 계층
    - 데이터(데이터베이스) 계층

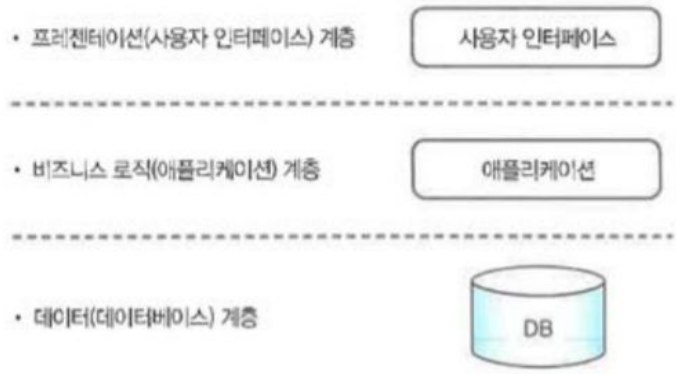


그림 5-25 3계층 모델

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

📜 4계층 아키텍처

4	사용자 인터페이스
3	사용자 인터페이스 관리 인증과 권한
2	핵심 비즈니스 로직/애플리케이션 기능 시스템 공통기능 (Application Tier)
1	시스템 지원(OS, 데이터베이스등) (Data Tier)

📜 3계층 아키텍처

3	프레젠테이션 로직 (Client Tier)
2	비즈니스 로직 (Application Tier)
1	데이터베이스 로직 (Data Tier)

📜 2계층 아키텍처

2	<table><tr><td>Application(응용프로그램)</td></tr><tr><td>프레젠테이션, 비즈니스 로직</td></tr></table> (Client Tier)
1	데이터베이스 로직 (Data Tier)

📜 1계층 아키텍처

1	<table><tr><td>Application(응용프로그램)</td></tr><tr><td>프레젠테이션 로직, 비즈니스 로직, 데이터베이스 로직</td></tr></table> (Client Tier)

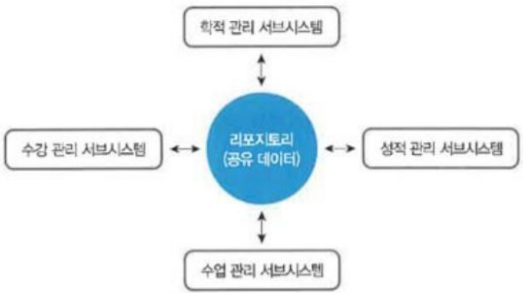
😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

—— 📜 저장소 아키텍처 ——

📖 p178.

대량의 데이터를 사용하는 시스템들의 대부분을 공유 데이터베이스 또는 저장소 중심으로 구성한다.

- **한 컴포넌트에서 생성된 데이터가 다른 컴포넌트에서 사용되는 응용분야에 적합하다.**

저장소 아키텍처	
설명	• **시스템의 모든 데이터는 모든 시스템 컴포넌트들이 접근할 수 있는 중앙 저장소에서 관리된다.** • 컴포넌트들은 직접 상호작용하지 않고 저장소를 통해서만 상호작용한다. ○ **한놈이 저장소를 만들고 나머지는 공유만...**
언제사용	장기간 저장되어야 하는 대량의 정보를 생성하는 시스템을 가지고 있을 때 사용.
장점	**컴포넌트들이 독립적이고 다른 컴포넌트들의 존재를 알 필요가 없다.** **한 컴포넌트에 의한 변경이 모든 컴포넌트로 전파된다.** **모든 데이터는 한 장소에 일관성 있게 관리된다.**
단점	**저장소의 문제는 전체 시스템에 영향을 끼친다.** 저장소를 여러 컴퓨터에 분산시키는 것은 어려울 수 있다. 저장소 패턴은 시스템의 정적 구조와 연관되어 있고 런타임 구성을 보여주지 못한다.
• 7.1. 데이터 중심형 모델 • 리포지토리 모델(Repository Model) • 주요 데이터가 리포지토리(Repository)에서 중앙 관리됨. • 장점 • 데이터가 리포지토리에 모여 있어서 데이터의 일관성있는 관리 가능 • 새로운 서브시스템을 추가하기 용이 • 단점 • 리포지토리의 병목현상(리포지토리 변경이 서브시스템에 큰 영향)  	

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

—— 📜 클라이언트-서버 아키텍처 ——

📖 p179.

/* 제공-요청, 분리와 독립 */

클라이언트-서버 패턴

- 분산 시스템을 위하여 공통적으로 사용되는 런타임 구성

이 모델의 주요 컴포넌트

1. 서비스를 다른 컴포넌트에 제공하는 **서버들의 집합**
2. 서버에 의해 제공되는 서비스를 요청하는 **클라이언트들의 집합**
3. 클라이언트가 서비스에 접근할 수 있게 해 주는 **네트워크**

클라이언트-서버 아키텍처는 보통 분산 시스템 아키텍처로 여겨지지만, 분리된 서버에서 실행되는 독립적인 서비스들의 논리적 모델이 단일 컴퓨터 상에 구현될 수 있다.

클라이언트-서버 아키텍처	
설명	• **클라이언트-서버 아키텍처에서는 시스템이 각 서비스가 독립적인 서버에 의해 제공되는 서비스들의 집합으로 표현된다.**
언제사용	공유 데이터베이스에 있는 데이터를 여러 지역에서 접근해야 할 때 사용한다.
장점	서버가 네트워크 상에 분산될 수 있다는 것이다. 일반적인 기능은 모든 클라이언트가 사용할 수 있으며 모든 서비스에 의해 구현될 필요가 없다.
단점	각 서비스가 단일 장애점이므로 서비스 거부 공격(Dos)에 민감하다. 성능이 시스템뿐만 아니라 네트워크에도 영향을 받기 때문에 성능을 예측하기 어려울 수 있다.

📌 **클라이언트-서버 모델의 가장 중요한 장점은 분산 아키텍처라는 것이다.**

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

- **7.2. 클라이언트-서버 모델(Client-Server Model)**
  - **데이터와 처리 기능을 클라이언트와 서버에 분할하여 사용**
  - **분산 아키텍처 유형에 적합**
  - **구성**
    - **서버: 서비스 제공**
    - **클라이언트: 서비스 요청**
    - **서비스**

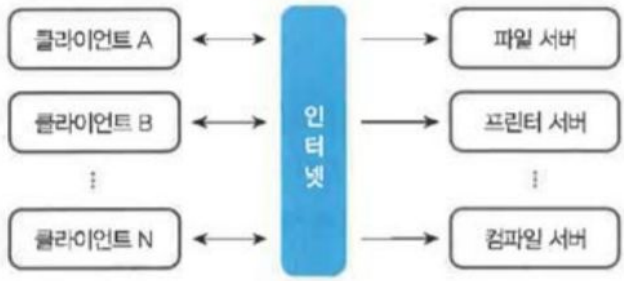


그림 5-24 클라이언트-서버 모델

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

—— 📜 파이프 필터 아키텍처 ——

📖 p181.

/* 데이터가 하나의 변환에서 다른 변환으로 순차적 진행 */

기능적 변환들이 입력과 출력을 처리하는 시스템의 런타임 구성 모델이다.

데이터가 하나의 변환에서 다른 변환으로 흘러가고 순서에 따라 가면서 변형된다.

- 입력 데이터는 출력으로 바뀔 때까지 변환들을 통해 흘러간다.
- **변환은 순차적으로 또는 병렬적으로 실행될 수 있다.**

사용자 상호작용이 제한되어 있는 일괄처리 시스템과 임베디드 시스템에 가장 적합하다.

- 단순한 텍스트 입출력 O
- 그래픽 사용자 인터페이스 X
- 대화식 시스템 X

파이프 아키텍처	
설명	**시스템에서 데이터 처리는 각 처리 컴포넌트가 분리되어 있으며 한가지 종류의 변환을 수행하도록 구성되어 있다.** 데이터는 처리를 위해 컴포넌트에서 다른 컴포넌트로 흐른다(파이프)
언제사용	입력으로부터 출력을 생성하기 위하여 **개별적인 단계를 거치는 데이터 처리 애플리케이션(배치 및 트랜잭션기반)**에서 보통 사용.
장점	• 이해하기 쉽고 **변환의 재사용을 지원**한다. • 워크플로 유형은 많은 비즈니스 프로세스의 구조와 일치한다. • 변환을 추가하여 기능을 변경하는 것이 명료하다. • **순차적 또는 병행 시스템으로 구현할 수 있다**.
단점	데이터를 주고 받는 변환 간에 데이터 전송 형식이 일치해야 한다. 각 변환은 입력을 분석해야 하고 약속된 형식에 맞추어 출력을 생성해야 한다. 이는 **시스템 부담을 증가**시키며 **호환되지 않는 데이터 구조**를 사용하는 아키텍처 컴포넌트를 **재사용할 수 없음**을 의미한다.

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

- 7. 아키텍처 모델
  - 7.5. 데이터 흐름 모델
    - 파이프 필터(Pipe & Filter) 구조
    - 사용자의 개입 없이 데이터의 흐름이 전환되는 경우에 사용
    - 구성
      - 필터(Filter)
        - 데이터 스트림을 입력 받아 처리하는 서브시스템
      - 파이프(Pipe)
        - 필터에서 처리된 데이터 스트림을 다른 필터에 전달

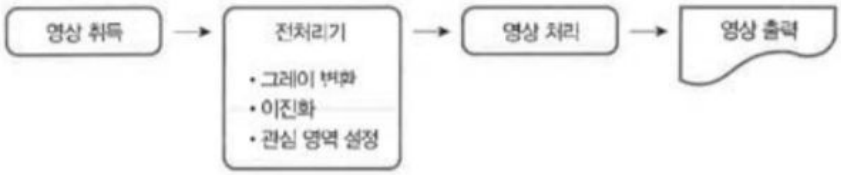


그림 5-27 데이터 흐름 모델의 예 : 이미지 프로세싱 과정

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

—— 📜 Mediator 아키텍처 ——

Mediator 패턴 – 개요

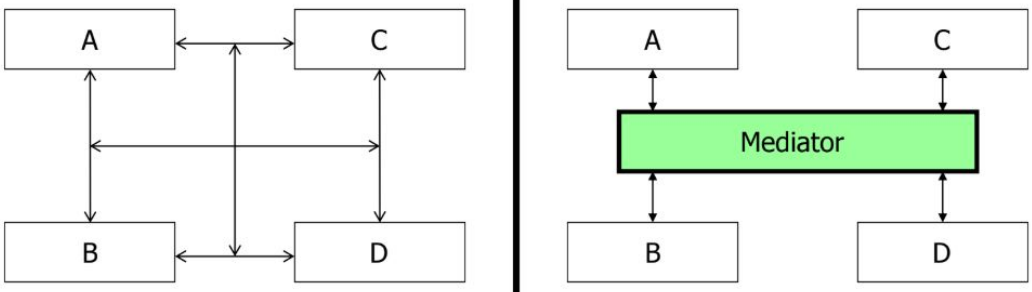
· 중재자를 사용하라.

me·di·a·tor

명사 : 영영사전

명사

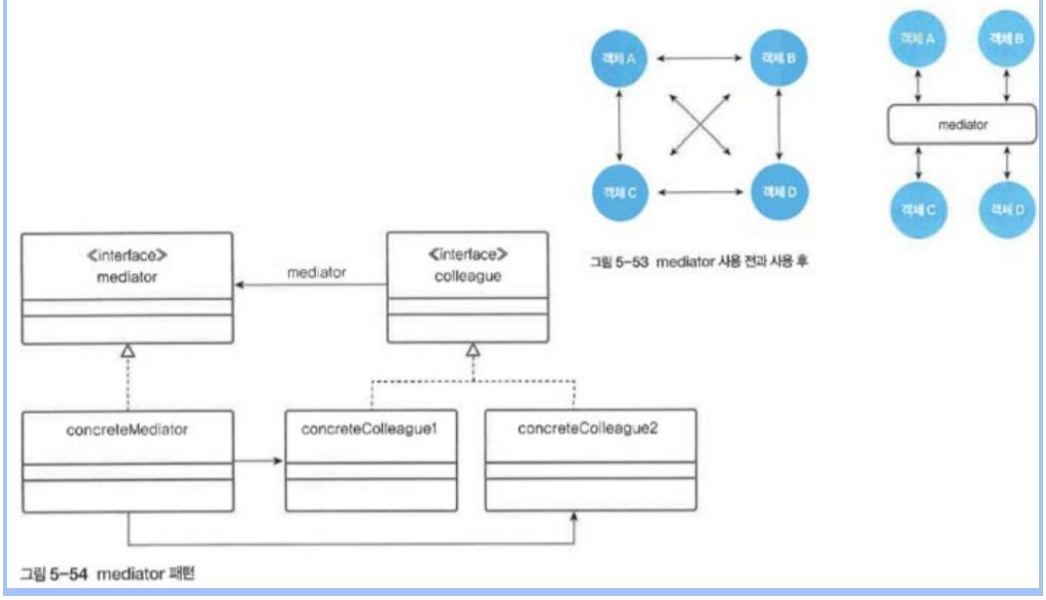
1. 중재인, 조정자, 매개자; [the M~] 하느님과 사람 사이의 중개자 ((그리스도))



중복된 코드가 많음.
페이지 삽입/삭제가 어려움.

중재자를 이용
유지보수가 용이

• 22. mediator 패턴



객체간의 복잡한 상호 작용 시 중재자를 사용하라.

중재자는 **객체 간의 느슨한 결합을 허용**한다.

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

```
public class Mediator
{
  Welcome welcome;
  Shop shop;
  Purchase purchase;
  Exit exit;
  Login login;//로그인추가

  public Mediator()
  {
    welcome = new Welcome(this);
    shop = new Shop(this);
    purchase = new Purchase(this);
    exit = new Exit(this);
    login = new Login(this);//로그인추가
  }

  public void handle(String state)
  {
    if(state.equals("welcome.login")){
      login.go();
    } else if(state.equals("shop.purchase")){
      purchase.go();
    }else if(state.equals("login.shop")){//로그인 추가
      shop.go();
    }else if(state.equals("login.exit")){//로그인추가
      exit.go();
    }else if(state.equals("welcome.exit")){
      exit.go();
    } else if(state.equals("shop.exit")){
      exit.go();
    } else if(state.equals("purchase.exit")){
      exit.go();
    } else if(state.equals("exit.shop")){
        shop.go();
        // Exit에서 원한다면 Shop으로 간다.
    }
  }

  public Welcome getWelcome()
  {
    return welcome;
```

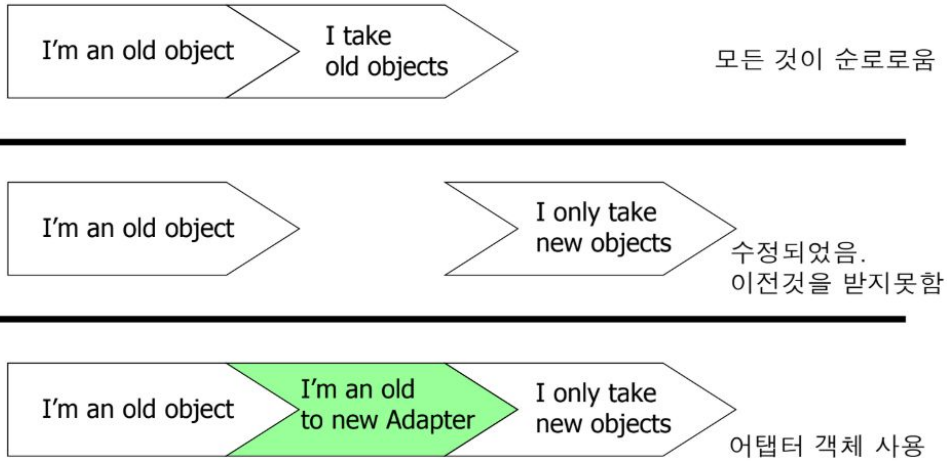
😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

—— 📜 Adaptor 패턴 아키텍처 ——

Adapter 패턴 – 개요

KGU 경기대학교

· 변환해서 사용하라.



변환해서 사용하라.

인터페이스 호환성 이점

• **9. adaptor 패턴**

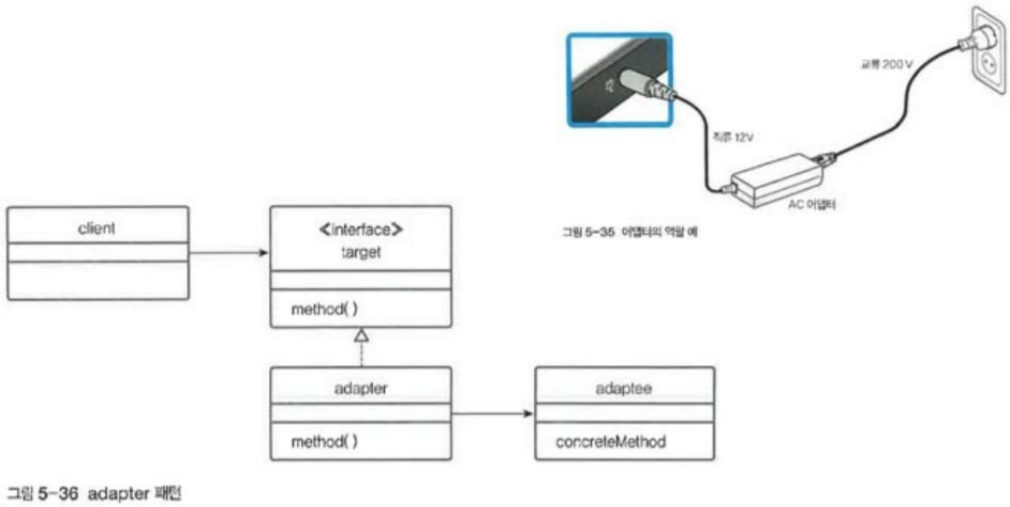


그림 5-36 adapter 패턴

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

```
public class AceToAcmeAdapter implements AcmeInterface
{
  AceClass aceObject;
  String firstName;
  String lastName;

  public AceToAcmeAdapter(AceClass a)
  {
    aceObject = a;
    firstName = aceObject.getName().split(" ")[0];
    lastName = aceObject.getName().split(" ")[1];
  }

  public void setFirstName(String f)
  {
    firstName = f;
  }

  public void setLastName(String l)
  {
    lastName = l;
  }

  public String getFirstName()
  {
    return firstName;
  }

  public String getLastName()
  {
    return lastName;
  }
}
```

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

—— 📜 Facade 패턴 아키텍처 ——

Façade 패턴 – 개요

- 사용하기 쉬운 인터페이스를 제공하라.

fa·cade

미국·영국 [fəˈsɑːd] 영국식

출판사별 옥스퍼드 동아출판 YBM 교

명사

명사

1. (건물의) 정면[앞면]
2. [주로 단수로] (실제와는 다른) 표면, 허울

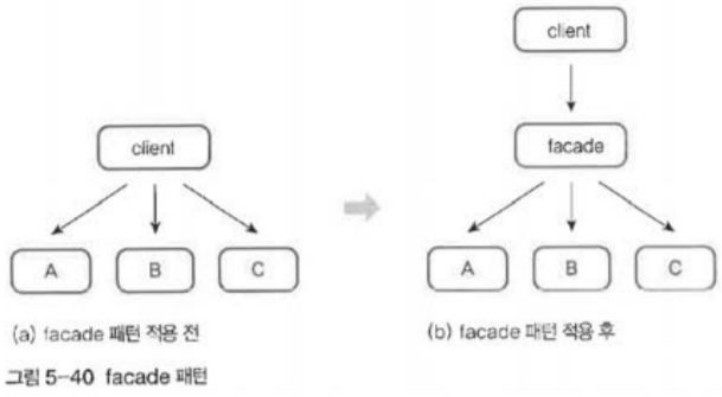
I've got a difficult interface	I am the Facade	I only need to deal with a simple interface

Adapter and Façade work in much the same way,
but they have different purposes.

The Adapter pattern adapts code to work with other code.
But the Façade pattern gives you a wrapper
that makes the original code easier to deal with.

사용하기 쉬운 인터페이스를 제공하라.
사용성(usability) 증가

- **12. facade 패턴**



(a) facade 패턴 적용 전 (b) facade 패턴 적용 후

그림 5-40 facade 패턴

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

```
public class TestFacade
{
  public static void main(String args[])
  {
    TestFacade t = new TestFacade();
  }

  public TestFacade()
  {

    SimpleProductFacade simpleProductFacade =
      new SimpleProductFacade();

    simpleProductFacade.setName("printer");

    System.out.println("The product is a " +
      simpleProductFacade.getName());
  }
}
```

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

```
public String getName()
{
  return new String(nameChars);
}
```

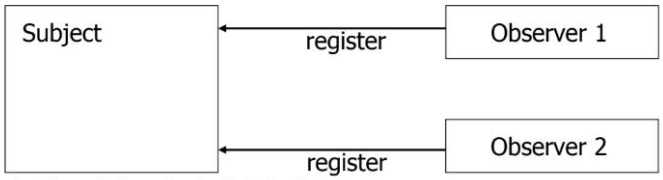
😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

—— 📜 Observer 패턴 아키텍처 ——

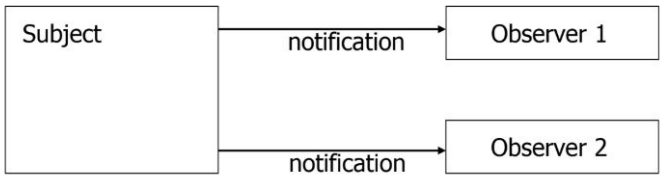
Observer 패턴 – 개요

KGU 경기대학

· 원하는 이벤트 발생을, 자동으로 통지 받아라.



원하는 이벤트가 발생했을때
자신을 불러달라고 등록



해당 이벤트가 발생했음을 통지

원하는 이벤트 발생을, 자동으로 통지 받아라.
Loose coupling 이점

• **16. observer 패턴**

그림 5-45 observer 패턴의 비유

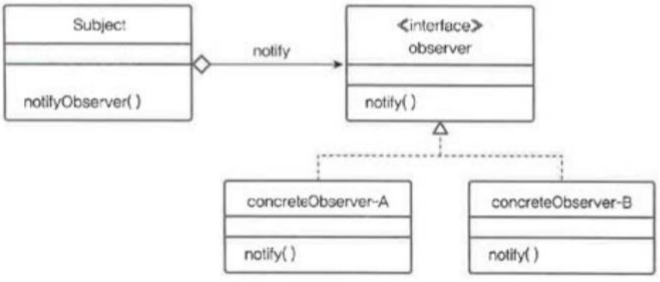


그림 5-46 observer 패턴

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

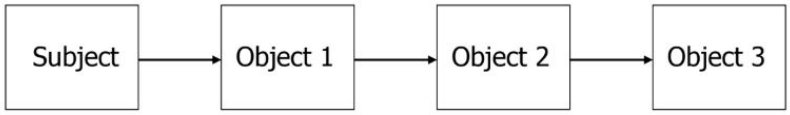
```
public class TestObservable
{
  public static void main(String args[])
  {
    Database database = new Database();

    Archiver archiver = new Archiver();
    Client client = new Client();
    Boss boss = new Boss();

    database.addObserver(archiver);
    database.addObserver(client);
    database.addObserver(boss);

    database.editRecord("delete", "record 1");
  }
}
```

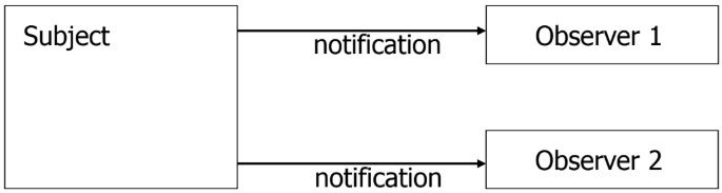
😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

—— 📜 Chain of Responsibility 패턴 아키텍처 ——

Chain of Responsibility 패턴 – 개요

· 책임에 따라 일렬로 나열하라.





• **20. chaine of responsibility 패턴**

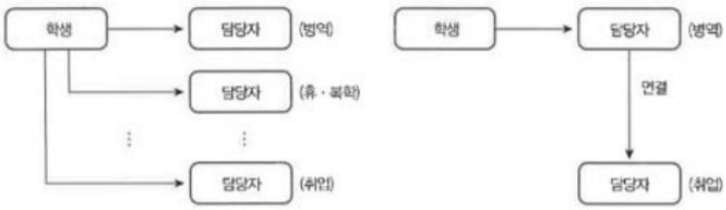


그림 5-50 학생 전화 상담 예

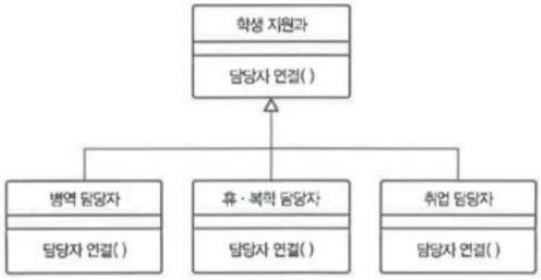


그림 5-51 chaine of responsibility 패턴

책임에 따라 일렬로 나열하라.
권한에 따른 역할 구분 이점

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

```java
public class TestHelp
{
  public static void main(String args[])
  {
    final int FRONT_END_HELP = 1;
    final int INTERMEDIATE_LAYER_HELP = 2;
    final int GENERAL_HELP = 3;

    Application app = new Application();

    IntermediateLayer intermediateLayer = new IntermediateLayer(app);

    FrontEnd frontEnd = new FrontEnd(intermediateLayer);

    frontEnd.getHelp(3);
  }
}
```

```java
public class IntermediateLayer implements HelpInterface
{
  final int FRONT_END_HELP = 1;
  final int INTERMEDIATE_LAYER_HELP = 2;
  HelpInterface successor;

  public IntermediateLayer(HelpInterface s)
  {
    successor = s;
  }

  public void getHelp(int helpConstant)
  {
    if(helpConstant != INTERMEDIATE_LAYER_HELP){
      successor.getHelp(helpConstant);
    } else {
      System.out.println("This is the intermediate layer. Nice, eh?");
    }
  }
}
```

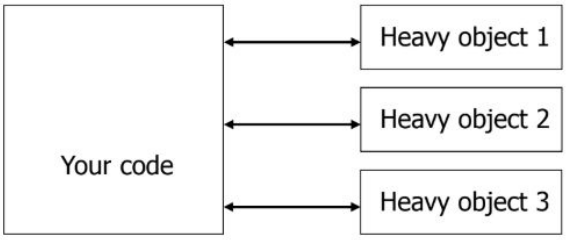
😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

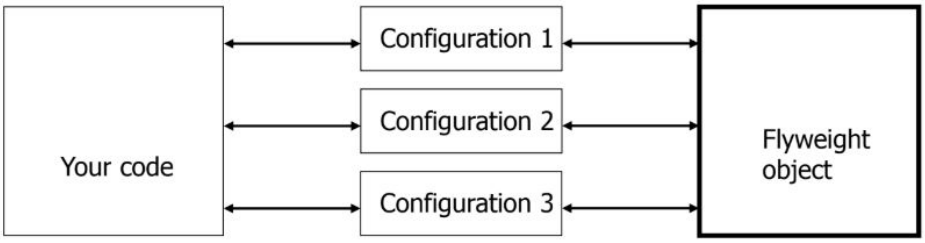
—— 📜 Flyweight 패턴 아키텍처 ——

Flyweight 패턴 – 개요

KGU 경기대학교

· 자원 절감을 위해, 객체가 상황에 따라 다르게 보이도록 해라.





• **13. flyweight 패턴**

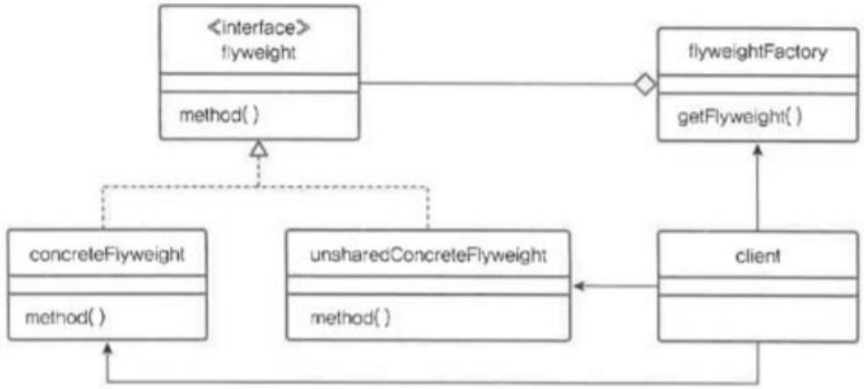


그림 5-41 flyweight 패턴

자원 절감을 위해, 객체가 상황에 따라 다르게 보이도록 해라.
공용 객체를 사용하여서 자원 절감 이점

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

```
public class TestFlyweight
{
  public static void main(String args[])
  {
    String names[] = {"Ralph", "Alice", "Sam"};
    int ids[] = {1001, 1002, 1003};
    int scores[] = {45, 55, 65};

    double total = 0;
    for (int loopIndex = 0; loopIndex < scores.length; loopIndex++){
      total += scores[loopIndex];
    }

    double averageScore = total / scores.length;

    Student student = new Student(averageScore);

    for (int loopIndex = 0; loopIndex < scores.length; loopIndex++){
      student.setName(names[loopIndex]);
      student.setId(ids[loopIndex]);
      student.setScore(scores[loopIndex]);

      System.out.println("Name: " + student.getName());
      System.out.println("Standing: " +
        Math.round(student.getStanding()));
      System.out.println("");
    }
  }
}
```

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

```
public class Student
{
  String name;
  int id;
  int score;
  double averageScore;

  public Student(double a)
  {
    averageScore = a;
  }

  public void setName(String n)
  {
      name = n;
  }

  public void setId(int i)
  {
      id = i;
  }

  public void setScore(int s)
  {
      score = s;
  }

  public String getName()
  {
      return name;
  }

  public int getID()
  {
      return id;
  }

  public int getScore()
  {
      return score;
  }

public double getStanding()
{
    return (((double) score) / averageScore - 1.0) * 100.0;
}
```

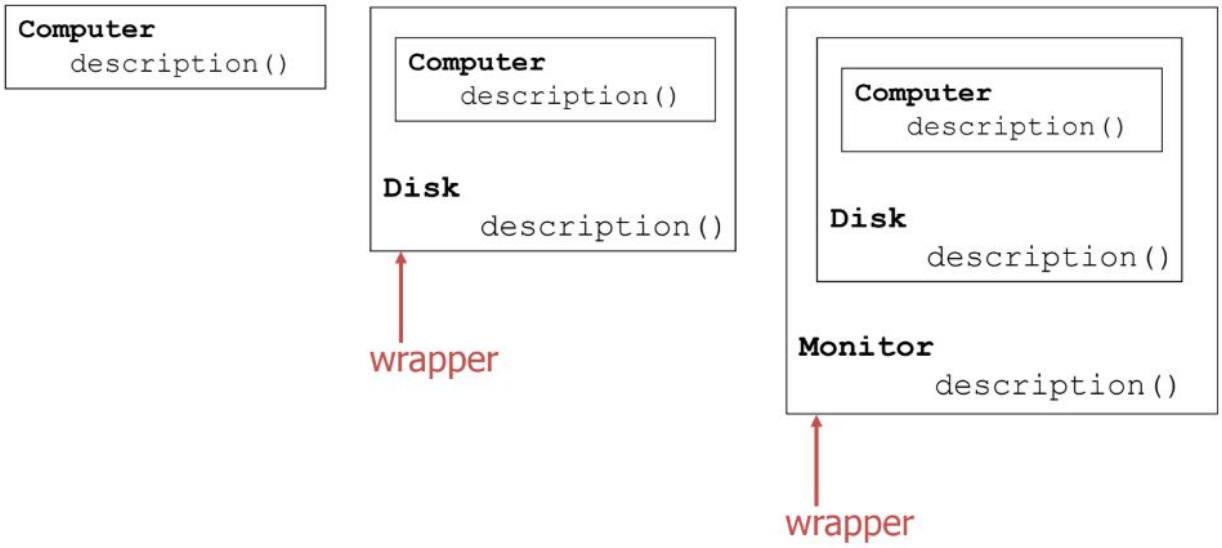
😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

—— 📜 Decorator 패턴 아키텍처 ——

Decorator 패턴 – 개요

KGU 경기대학교

- 객체의 기능을 동적으로 확장시켜라.



객체의 기능을 동적으로 확장시켜라.
유연한 확장의 이점

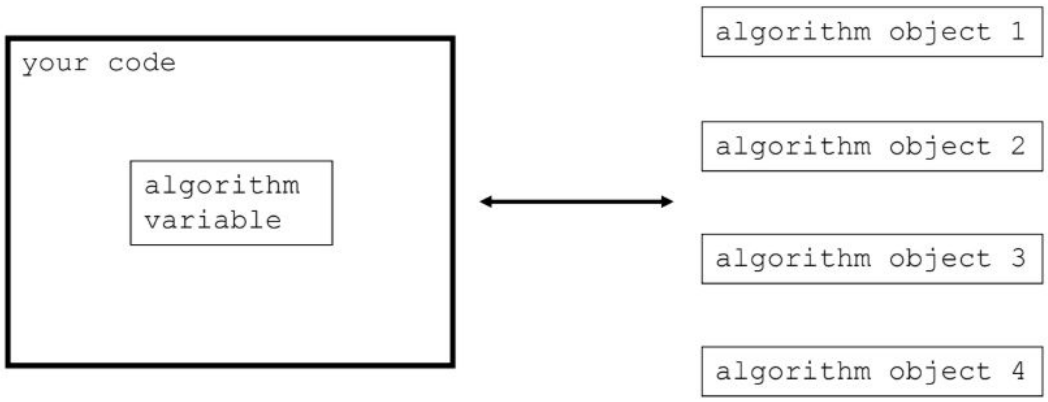
😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

—— 📜 Strategy 패턴 아키텍처 ——

Strategy 패턴 - 개요

KGU

· 다양한 알고리즘을 동적으로 변경하면서 사용하라.



• **17. strategy 패턴**

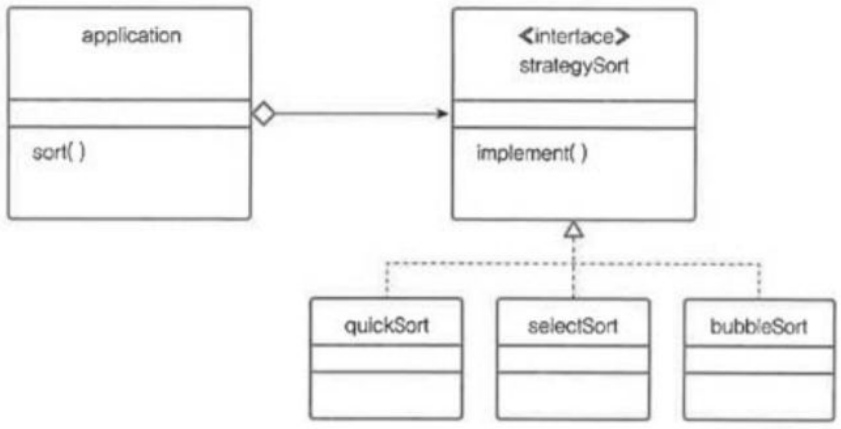


그림 5-47 strategy 패턴

다양한 알고리즘을 동적으로 변경하면서 사용하라.
다양한 알고리즘 활용

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

—— 📜 Template Method패턴 아키텍처 ——

Template Method 패턴 – 개요

KGU

· 공통된 메소드를 상속하여 재사용하라.

Automative robot	Cookie robot

start();
getParts();
assemble();
test();
stop();
	start();
	getParts();
	assemble();
	test();
	stop();

• **18. template method 패턴**

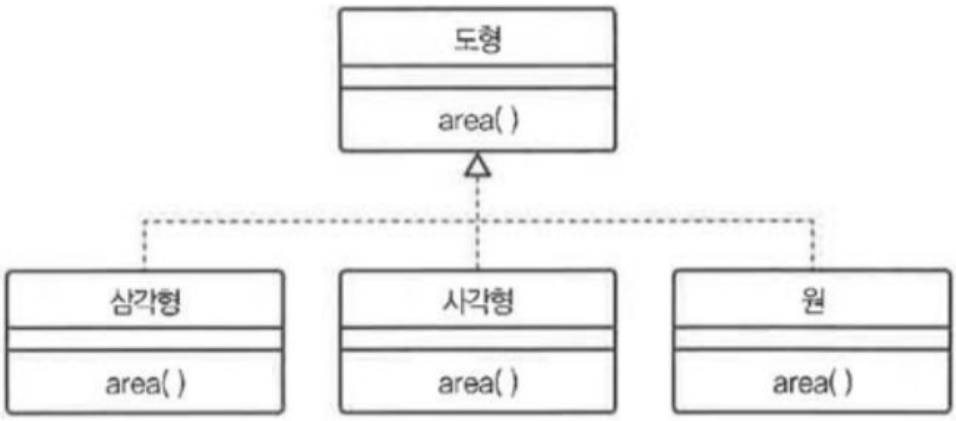


그림 5-48 template method 패턴

공통된 메소드를 상속하여 재사용하라.
상황에 맞게 수정할 다단계 알고리즘을 갖는 경우 유용

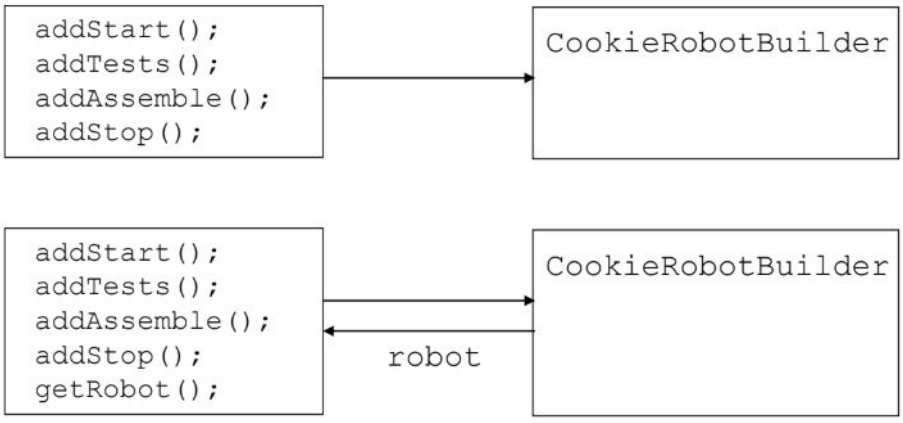
😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

—— 📜 Builder패턴 아키텍처 ——

Builder 패턴 – 개요

KGU

· 메소드 실행 순서를 각자 정해서 사용해라.



• 25. interpreter 패턴

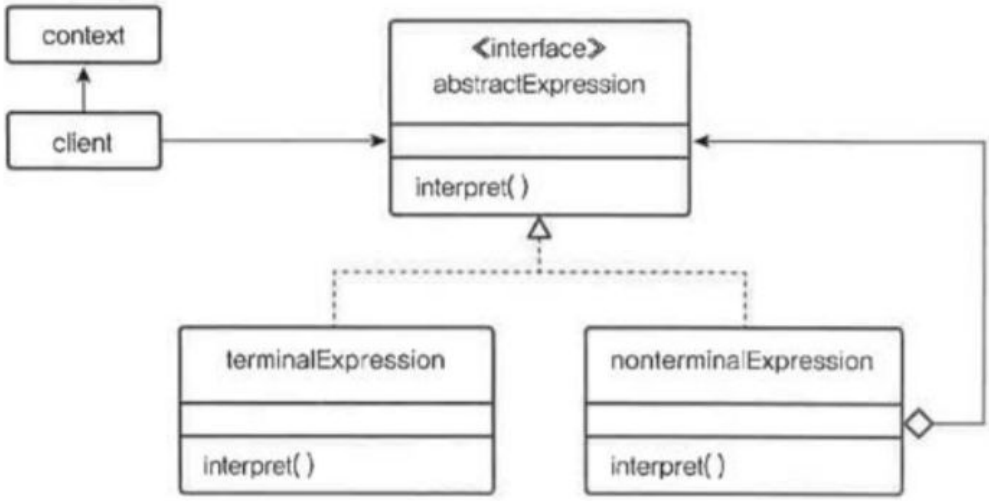


그림 5-57 interpreter 패턴

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

메소드 실행 순서를 각자 정해서 사용해라.

Template Method 패턴의 경우 고객은 순서를 그대로 따라야 하지만, Builder 에서는 순서를 고객이 정할 수 있다.

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

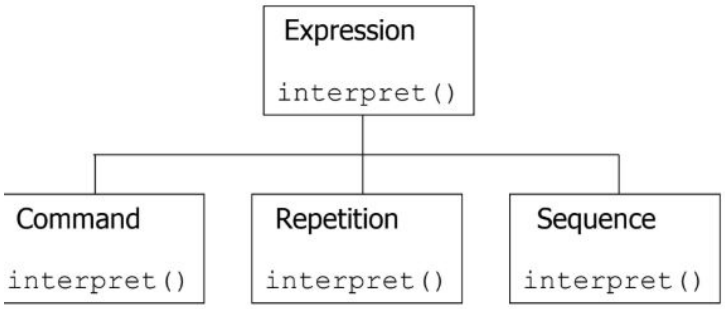
—— 📜 Interpreter패턴 아키텍처 ——

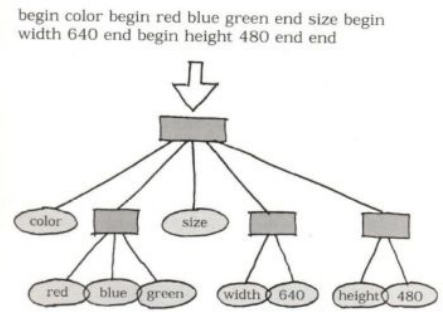
Interpreter 패턴 – 개요

KGU 경기대학

- 프로그램을 수정하기 보다는 미니 언어를 사용하라.

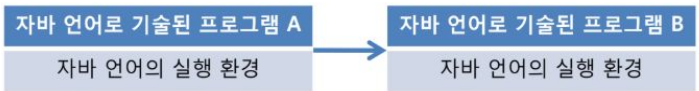
```
expression ::= <command> | <repetition> | <sequence>
```





자바 언어 vs 미니 언어

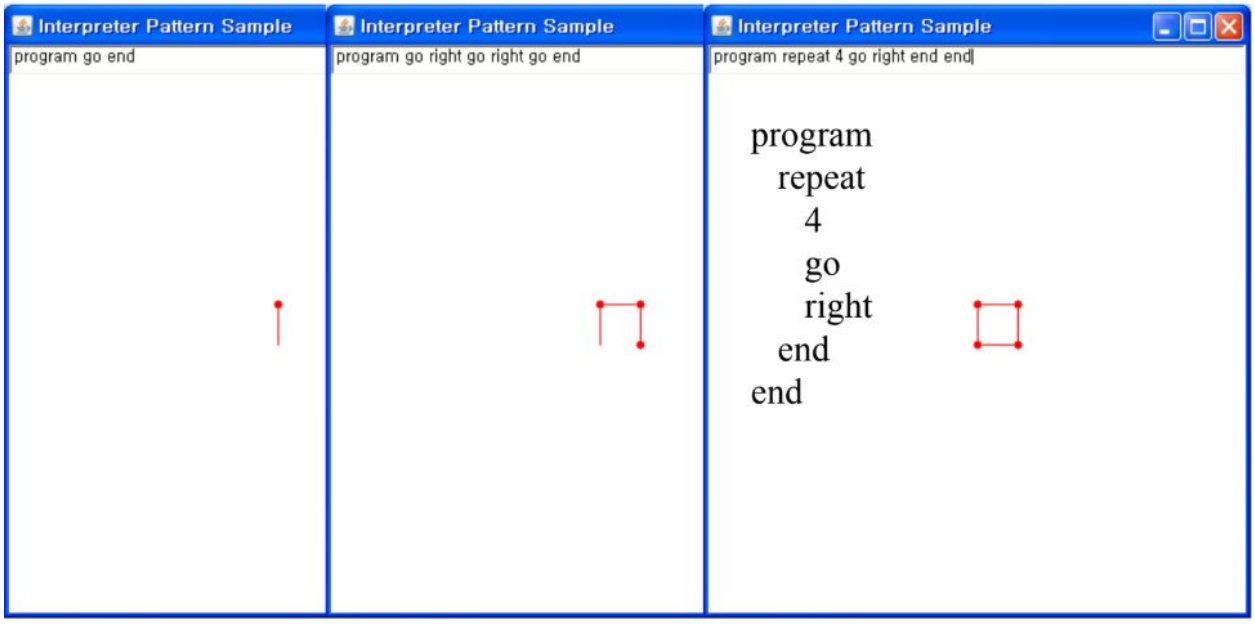
- 미니 언어
  - 특정 문제를 기술
  - 인터프리터 요구됨 → 미니 언어를 번역, 실행
  - 변화 발생된 경우 → 자바보다는 미니 프로그램 수정
- 문제에 변화가 생긴 경우



😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

예제

KGU 경기대학교



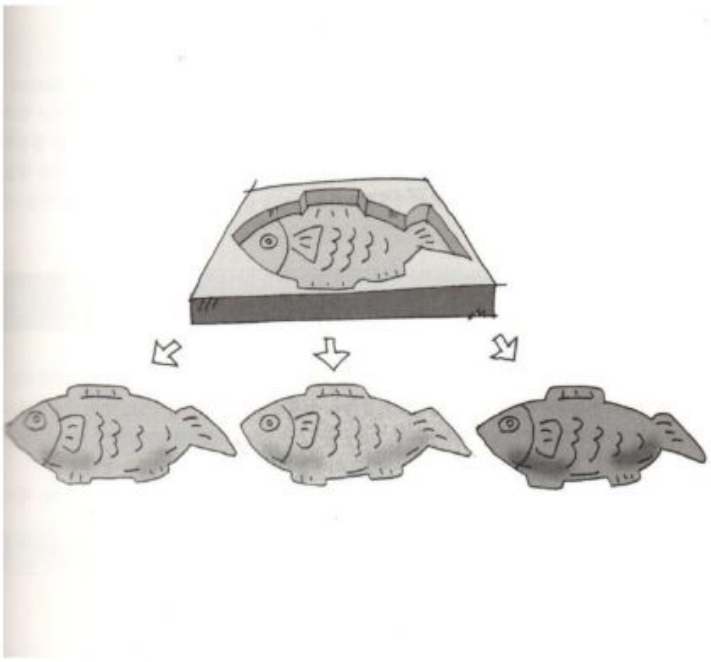
프로그램을 수정하기 보다는 미니 언어를 사용하라.
특정 문제 분야에 유연성을 높임

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

—— 📜 Factory패턴 아키텍처 ——

Factory 패턴 – 개요

· 객체 생성 공장을 운영하라.



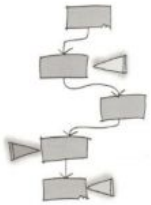
객체 생성 공장을 운영하라.
- 팩토리는 다른 객체들을 생성해주는 객체이다.

객체 생성을 한 곳에서 관리

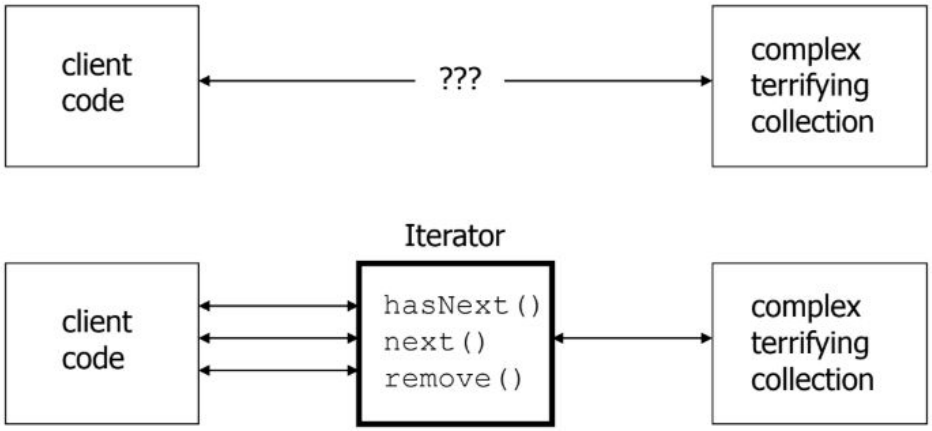
😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

—— ❗ 📜 Iterator패턴 아키텍처 ——

Iterator 패턴 – 개요



· 방문자를 통해서 콜렉션의 각 요소에 접근하라.



요약
콜렉션 • 데이터의 집합 방문자를 통해서 콜렉션의 각 요소에 접근하라. 콜렉션의 표현에 **독립적**
Iterator 라는 인터페이스를 구현(implement)하여 사용 • Next • hasNext • Remove

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

- **15. iterator 패턴**

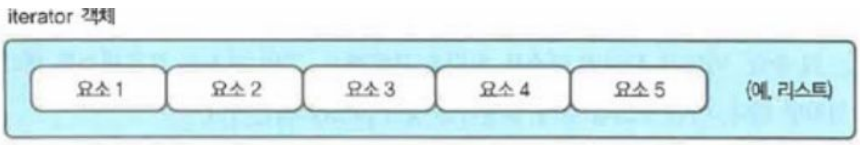


그림 5-43 iterator 객체의 예

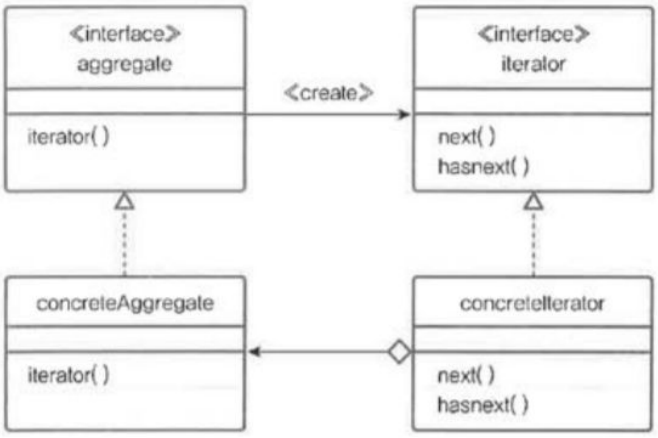


그림 5-44 iterator 패턴

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

TestDivision

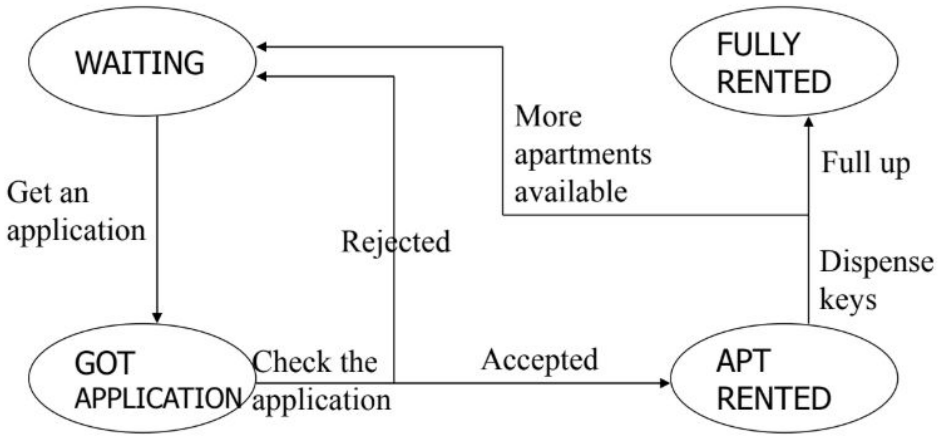
```
public class TestDivision
{
  Division division[];
  DivisionIterator iterator;

  public static void main(String args[])
  {
    TestDivision d = new TestDivision();
  }

  public TestDivision()
  {
    division = new Division[3];
    division[0] = new Division("Sales");
    division[1] = new Division("R&D");
    division[2] = new Division("Marketing");

    division[0].add("Ted");
    division[0].add("Bob");
    division[0].add("Carol");
    division[0].add("Alice");

    division[1].add("Ted");
    division[1].add("Bob");
    division[1].add("Carol");

    division[2].add("Ted");
    division[2].add("Bob");
    division[2].add("Carol");

    iterator = division[0].iterator();

    while (iterator.hasNext()){
      VP vp = iterator.next();
      vp.print();
    }
    System.out.println("");
    iterator = division[1].iterator();

    while (iterator.hasNext()){
      VP vp = iterator.next();
      vp.print();
    }
    System.out.println("");
    iterator = division[2].iterator();

    while (iterator.hasNext()){
      VP vp = iterator.next();
      vp.print();
    }
  }
}
```

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

—— 📜 State패턴 아키텍처 ——

State 패턴 – 개요

· 상태에 따라 행위가 달라지면, 상태를 객체로 취급하라.



• **23. state 패턴**

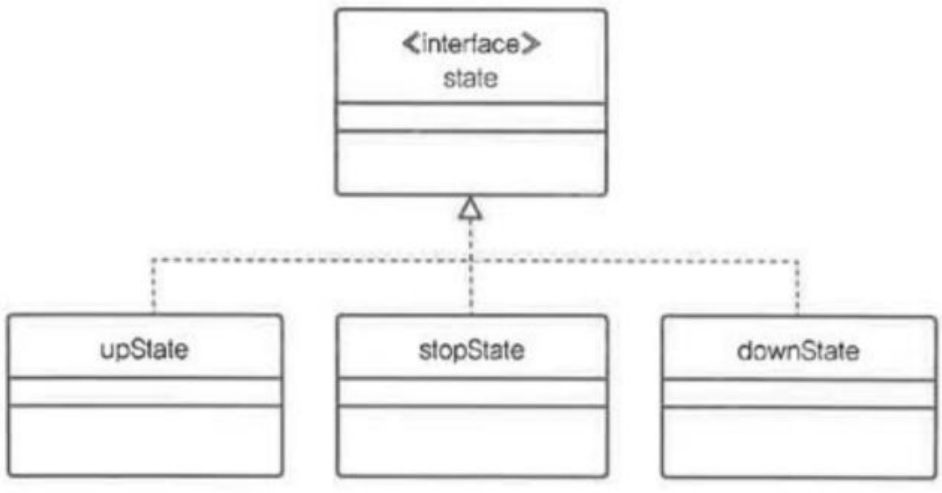


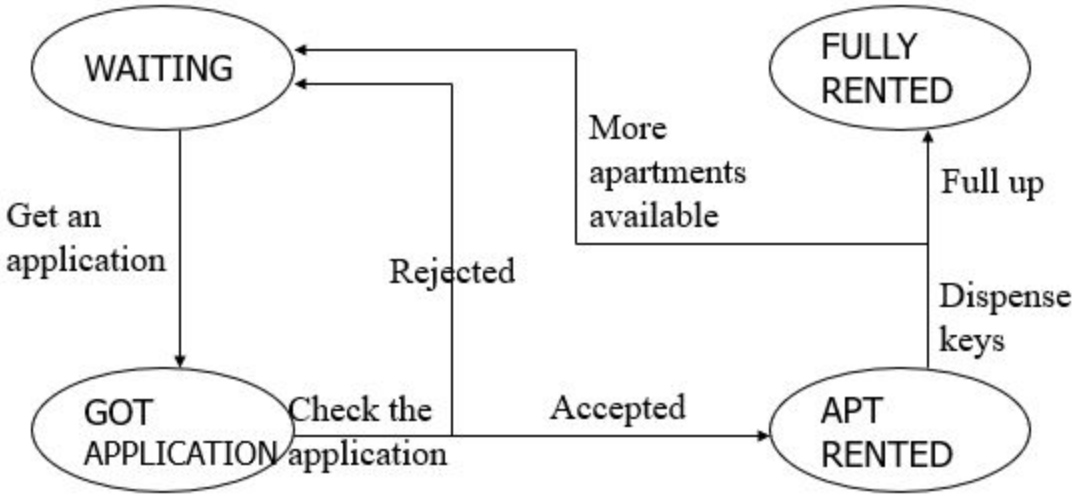
그림 5-55 state 패턴

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

요약
상태에 따라 행위가 달라지면, 상태를 **객체로 취급**하라. **상태**와 **행위**가 **합쳐짐**

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

기숙사의 방을 정해주는 프로그램 (Ver.1.0.1)



대기상태 → 접수상태 → (옳은지 확인) → Accept(옳지 않으면 Reject) → 방대여상태(빈방이 있으면) → 대기상태 (빈방이 없으면 완료상태)
Operation 마다 Switch문을 이용, 각 상태는 숫자의 형태로 고정값으로 지정
영환띠

기숙사의 방을 정해주는 프로그램 (Ver.1.2.1)

State 패턴을 적용함

기숙사의 방을 정해주는 프로그램 (Ver.2.1.1)

기숙사에 영어를 는잘하 친구들을 받고 싶다.
영어가 유창성 여부판단 → 토익점수 750점 이상이면 Accept 아니면 Reject

❗ 변경된 요구사항이 적용된 2.1.1버전의 프로그램을 1.0.1버전과 1.2.1버전으로 각각 구현

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

—— 📜 Proxy패턴 아키텍처 ——

Proxy 패턴 – 개요

KGU 경기

· 다른 객체를 위한 대리인을 사용하라.

지역 객체를 다루는 지역 프로그램
다른 컴퓨터에 있는 원격 객체도 처리하기를 희망

프락시 사용. 다른 객체를 위한 대리인

다른 객체를 위한 대리인을 사용하라.
투명성(객체의 거주 위치를 몰라도 됨) 높임

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

—— 📜 Singleton패턴 아키텍처 ——

Singleton 패턴 – 개요

KGU 경기대학교

· 기독교에서의 하나님

유일신 여호와 하나님만을 섬겨라

1 너는 나 외에는 다른 신들을 네게 두지 말라.

```
public class God {
   ...
   public God (...) {
      ...
   }
   ...
}
```

다양한 God 객체를 만들 수 있음.

어떻게 하면 객체를 유일하게 하나만 만들 수 있을까 ?

40

싱글톤(Singleton) 패턴 ?

싱글톤(Singleton) 이란 **하나의 해당 클래스에서 단 하나의 인스턴스만 만들도록 보장하는 방법**을 싱글톤 패턴이라고 한다. 싱글톤 패턴은 하나의 인스턴스만을 재사용하게 된다. 따라서 객체를 여러 번 생성할 필요가 없는 경우에 싱글톤을 사용하면 **불필요한 자원 낭비나 오버헤드 등을 막을 수 있다**. 싱글톤은 자바와 데이터베이스를 연동(DAO 클래스) 하여 사용할 때 자주 사용된다.

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

위키백과
소프트웨어 디자인 패턴에서 **싱글턴 패턴**(Singleton pattern)을 따르는 클래스는, 생성자가 여러 차례 호출되더라도 **실제로 생성되는 객체는 하나**이고 최초 생성 이후에 호출된 생성자는 최초의 생성자가 생성한 객체를 리턴한다. 이와 같은 디자인 유형을 **싱글턴 패턴**이라고 한다. 주로 공통된 객체를 여러개 생성해서 사용하는 DBCP(DataBase Connection Pool)와 같은 상황에서 많이 사용된다.

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

—— 📜 2oo3 architecture——

2 out of 3로 3개중 2개만 만족하면 시스템은 정상이라고 판단하는 구조

항공기 관련 소프트웨어가 이런 시스템을 사용한다.

교수님이 주신 veto코드를 2 out of 3코드로 변환해보자

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

😎왜 설계 패턴의 인가?

- 개발자를 힘들게 하는 **"변심"**
- 나중에 소프트웨어를 수정, 확장할 때 비용이 더 들수 있다.
- design patterns make your code closed for modification, but open
- for extension

😎설계 패턴의 이점

- 1. 디자인 패턴의 이해
  - 디자인 패턴의 정의
    - 자주 사용하는 설계 형태를 정형화 하여 유형별로 만든 설계 템플릿
  - 장점
    - 효율성과 재사용성 제고
    - 개발자(설계자) 간의 원활한 의사소통
    - 소프트웨어 구조 파악 용이
    - 재사용을 통한 개발시간 단축
    - 설계 변경 요청에 대한 유연한 대처
  - 단점
    - 객체지향 설계/구현 위주
    - 초기 투자 비용 부담

- **재사용성 증가**
- **확장 가능성 증가**
- **유지보수 용이성 증가**

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

😎애플리케이션 아키텍처

📜 트랜잭션 처리 시스템

데이터베이스에 있는 정보에 대한 사용자 요청이나 데이터베이스 갱신을 위한 요청을 처리하기 위하여 설계되었다.

- 기술적으로 데이터베이스 트랜잭션 = 연속된 작업의 일부.
- 한 트랜잭션 내의 모든 작업들은 데이터베이스 변경이 영구적으로 이루어지기 전에 완료되어야 한다.

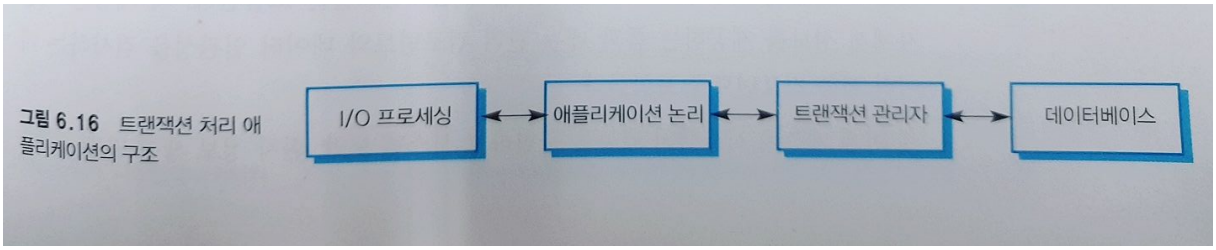


그림 6.16 트랜잭션 처리 애플리케이션의 구조

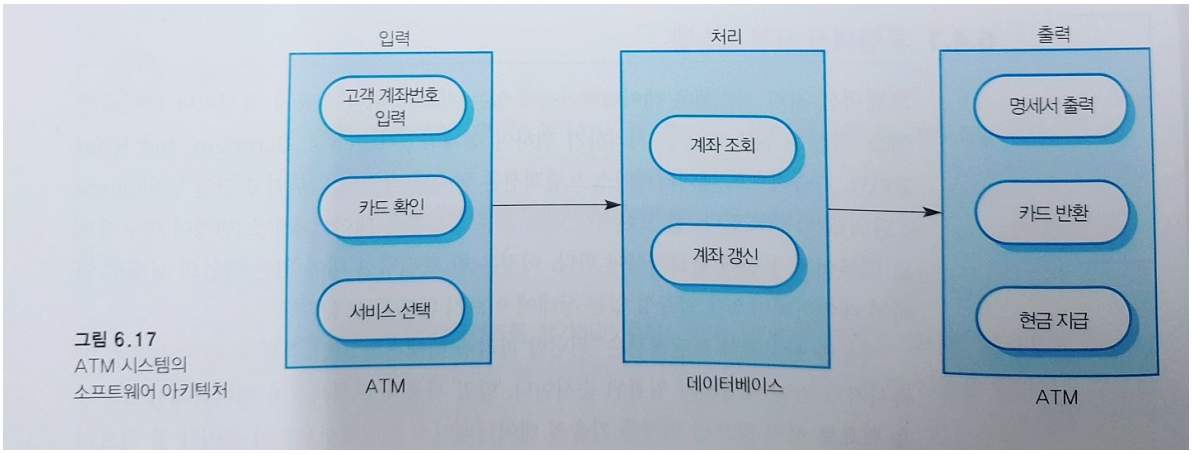


그림 6.17 ATM 시스템의 소프트웨어 아키텍처

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

📜 정보 시스템

[안중요] 공유 데이터베이스와 상호 작용에 관련된 모든 시스템은 트랜잭션 기반 정보 시스템으로 여겨질 수 있다.

정보시스템은 거의 항상 사용자 인터페이스가 웹 브라우저로 구현된 웹 기반 시스템이다.

📜 언어 처리 시스템

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

😀 소프트웨어의 진화

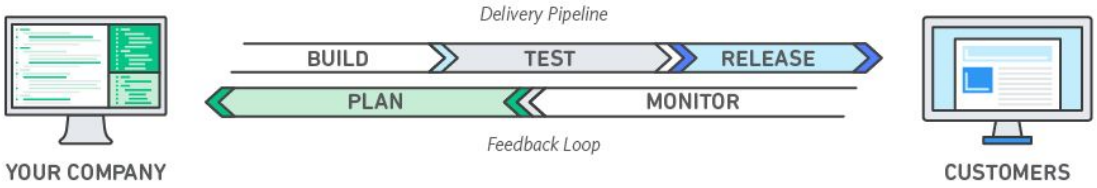
SoftWare는 2번의 생을 가진다.

- 1생 (개발동안의 삶)
  - 계약~출시 까지
  - 삶이 짧다
- 2생 (이후 서비스기간)
  - 출시~폐기처분 까지
  - 삶이 길다.

데브옵스란?

DevOps 모델 정의

데브옵스는 애플리케이션과 서비스를 빠른 속도로 제공할 수 있도록 조직의 역량을 향상시키는 문화 철학, 방식 및 도구의 조합입니다. 기존의 소프트웨어 개발 및 인프라 관리 프로세스를 사용하는 조직보다 제품을 더 빠르게 혁신하고 개선할 수 있습니다. 이러한 빠른 속도를 통해 조직은 고객을 더 잘 지원하고 시장에서 좀 더 효과적으로 경쟁할 수 있습니다.



DevOps 작동 방식

DevOps 모델에서는 개발팀과 운영팀이 더 이상 "사일로"에 묶여 있지 않습니다. 때로는 이 두 팀이 단일팀으로 병합되어 엔지니어가 개발에서 테스트, 배포, 운영에 이르기까지 전체 애플리케이션 수명주기에 걸쳐 작업하고 단일 기능에 한정되지 않은 광범위한 기술을 개발합니다.

일부 DevOps 모델에서 품질 보증 팀과 보안 팀 또한 애플리케이션 수명주기에 걸쳐 개발 및 운영과 좀 더 긴밀하게 통합됩니다. DevOps 팀 전체가 보안을 중점으로 두는 경우 때때로 DevSecOps라고 불립니다.

이러한 팀에서는 데브옵스 방식을 사용하여 속도가 느리고 수동으로 수행되던 프로세스를 자동화합니다. 또한, 애플리케이션을 안정적으로 빠르게 운영하고 개선하는 데 도움이 되는 기술 스택과 도구를 사용합니다. 이러한 도구 덕분에 엔지니어는 이전 같았으면 다른 팀의 도움이 필요했을 코드 배포 또는 인프라 프로비저닝과 같이 작업을 독립적으로 수행할 수 있으며, 따라서 팀의 작업 속도가 더욱 빨라집니다.

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

배경

- 진화 = 유지보수
- SW 장수시대
- SW 무병 장수시대
- 변경이 필수적
- 데이터
    - Lientz 60~90% 유지보수비용
    - Jones 75% 직원이 유지보수와 관련된 업무에 종사
- Greenfield 소프트웨어 개발
    - SW가 독립적인 환경에서 처음부터 개발됨
- Brownfield 소프트웨어 개발 📖 (P.260 하단)
    - sw가 다른 sw 의존적인 환경에서 개발되고 관리됨
    - 유지보수를 위해서 변경의 영향을 이해하고 분석해야 함

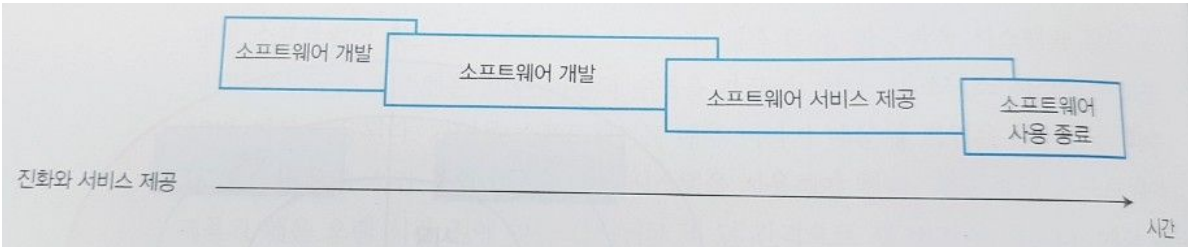
진화모델

- 나선형 프로세스
    - 릴리스(새로운 버전) 간격이 매우 줄어듬
    - 개발팀과 유지보수팀이 같은 경우

- 개발팀과 유지보수팀이 다른 경우 → 외주를 줌

진화와 서비스 제공

- 진화 : SW 아키텍쳐와 기능에 상당한 변경이 가해짐
- 서비스 제공 : 상대적으로 작지만 필수적인 변경만 가해짐



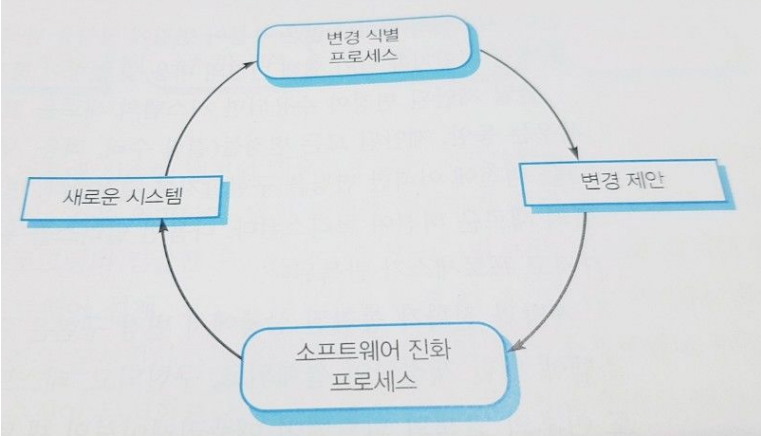
SW 개발 →	SW 진화 →	SW 서비스 →	SW 은퇴(사용종료)

- 사용종료(retirement)를 **폐기처분(decommissioning)**이라고도 부름 , 이때 폐기처분 프로세스는?

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

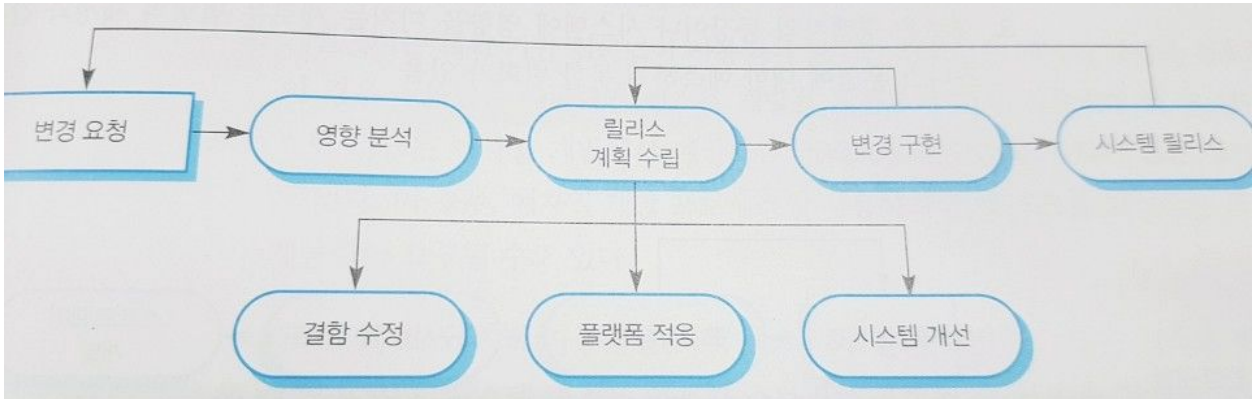
변경식별 프로세스

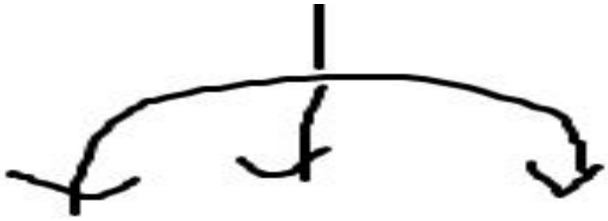
- 변경 제안이 시스템 진화를 주도함



	변화 식별 프로세스	
새로운 시스템		변화 제안
	📌 SW 진화 프로세스 (중요)	

- 영향 분석(impact analysis)
  - 변경의 영향 및 비용을 평가



변화 요구 Change requests	**영향분석** Impact analysis	**Release 계획** Release planning	**Change** implementation	**System release**
				

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

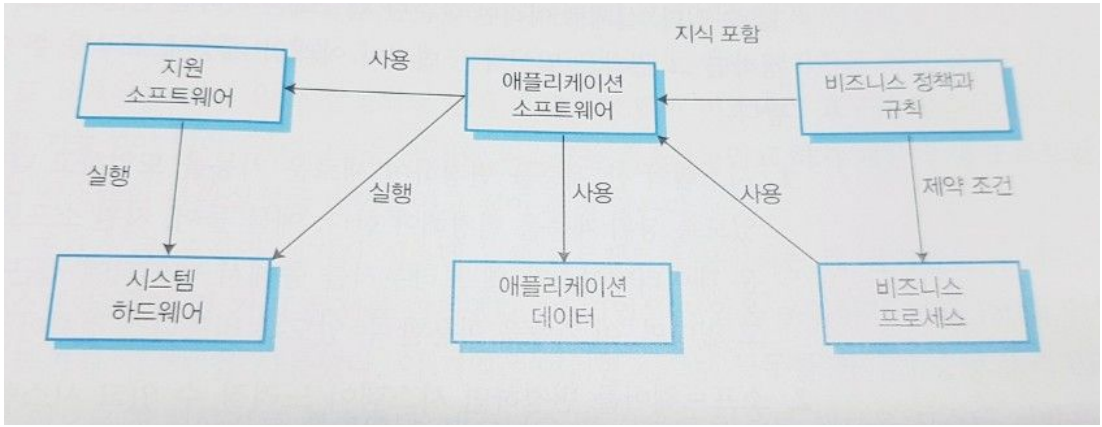
	버그 고치기 Falut repair	Platform adaptaion	System enhancement	

변경구현

- 일반적인 경우 : 요구된 변경을 반영하도록 요구 사항 명세서도 업데이트 해야함
- 긴급한 경우 :
- 긴급한 변경: 프로그램 수정이 우선 순위가 높은
    1) 심각한 결함 감지
    2) 운영 환경의 변화
    3) 경쟁자 등장

레거시 시스템

- 구형 소프트웨어 시스템
- Legacy~ 구관이 명관이다.



- 데이터
    - 2천억 라인 이상의 코볼코드 사용됨
    - 1960~1990년 대 개발됨
    - 코볼 프로그래머 희귀
    - 인터넷이 활성화되기 전에 개발되어 보안 취약점을 가짐

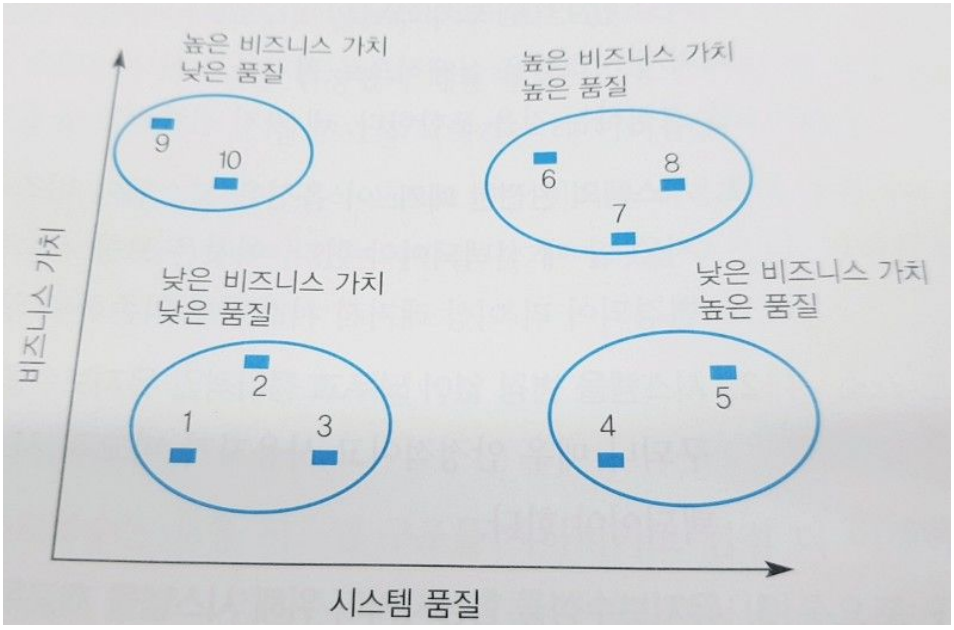
- 레거시 시스템 vs 새로운 시스템

- 새로운 시스템으로 교체하는 것이 어렵고 위험한 이유
  269페이지
- 레거시 시스템을 변경하기 힘든 이유?

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

레거시 시스템 관리

- 전략
  - 완전한 폐기(1, 2, 3)
  - 변경없이 정기적인 유지보수(6, 7, 8)
  - 시스템을 재공학 (9, 10)
  - 전체 또는 일부분을 새로운 시스템으로 대체 (4, 5)



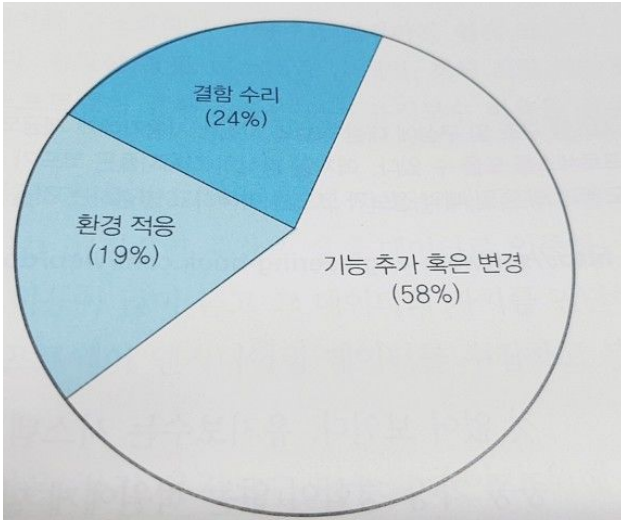
소프트웨어 유지보수

- 변경의 영향
  - 코드변경 < 설계변경 < 명세서 변경

- 소프트웨어 유지보수 유형 p.276
  - 결함수리 : 버그(snowball effect), 취약점을 고치기
  - 환경적응 : 앗
  - 기능추가
- 다른용어로
  - Corrective maintenance
  - Adaptive maintenance
  - Perfective maintenance

😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

- 새로운 기능 추가에 드는 비용이 비싼 이유
    - 유지보수되는 프로그램을 이해해야함
    - 책 277
    - 프로그램 유지보수 업무는 인기가 없음
    - 자꾸 고칠수록 구조가 저하되고, 변경하기 점점 어려워짐



😀: 제목1 😎: 제목2 📜: 목차1 📣: 알림 📌: 쪽집게 📖: 책내용 📘: 참고 📒: 참고 💡: 솔루션

리팩토링

- Refectoring

- 프로그램의 품질 저하를 느리게 하는 부분으로 개선하는 활동
    - 구조개선
    - 복잡성 줄임
    - 이해하기 쉽도록 수정

- 원칙
    - 기능을 추가하지 말고 프로그램 개선에 집중

- 재공학
    - 유지보수 비용이 증가할때 시행

- 리팩토링
    - 개발 및 진화 과정에서 유지보수 비용과 어려움을 증가시키는 구조 및 코드의 품질 저하를 방지
    - 유명인 :
        - 마틴 파울러 (“프로그램의 가치를 높이는 코드 정리기술” 저자)
        - Bad Code Smell
            - 코드에도 악취가 있다..
            - 악취의 예
                - W중복코드
                - 길이가 긴 메소드
                - 스위치(case 문
                - 데이터 군집
                - 추측에 근거한 일반성

- 코드 리펙토링을 넘어서 **“설계 리펙토링”**
    - 관련된 설계 패턴을 식별하고,
      설계패턴을 구현한 코드를 기존코드로 대체함
      EX) 이클립스에서 코드 리펙토링을 돕는 기능들이 있다.