6. Design: Architecture

Choi, Kwanghoon

Chonnam National University

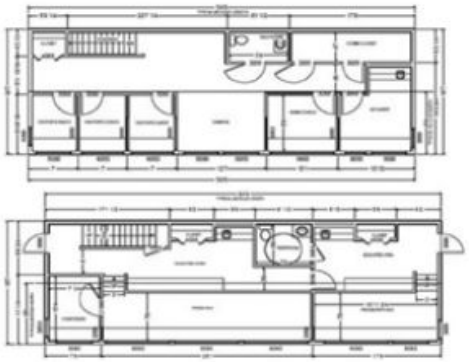
Table of Contents

Software Architecture

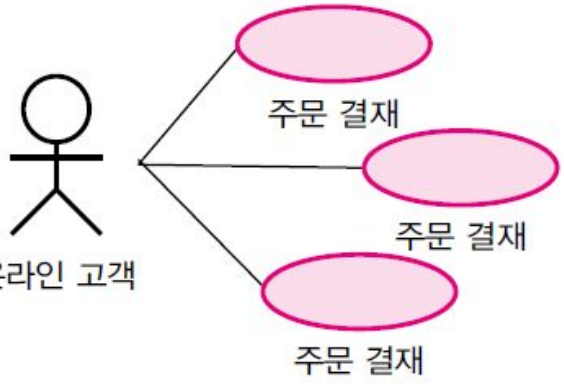
-

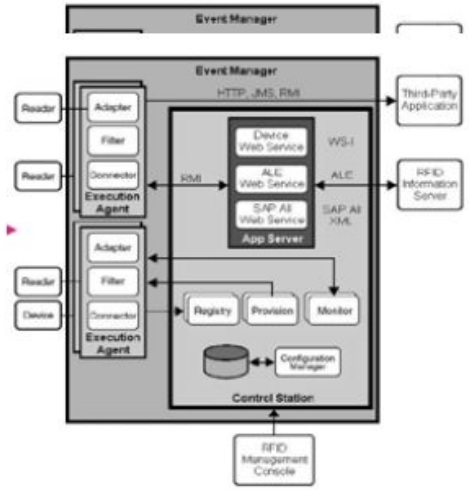
[Analysis] [Design]

건축



소프트웨어





참고) 최은만 교재

Software Architecture: Two definitions

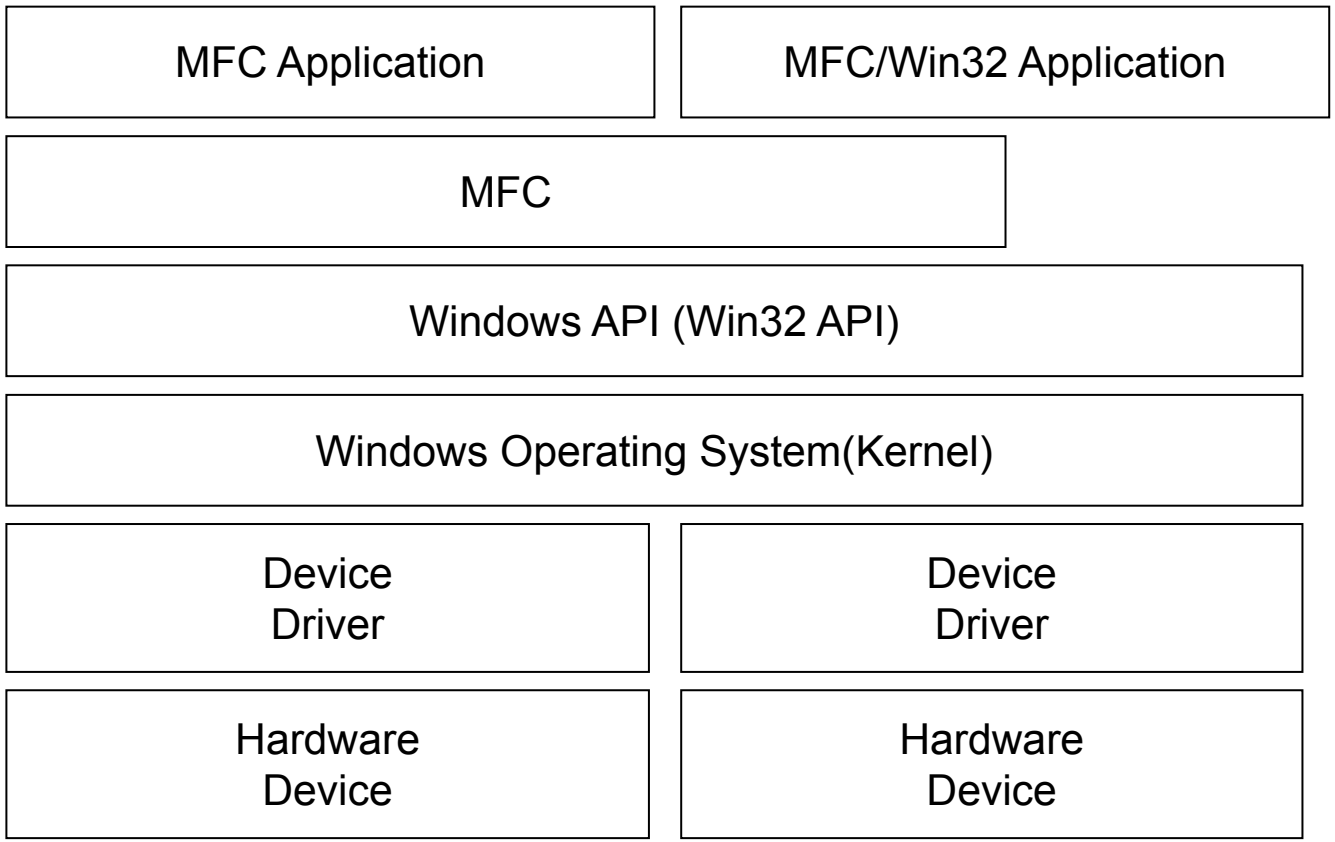
- Blueprint of a software system (설계 유형)
  - Structure
  - Behavior
  - Interaction
  - Non-functional properties

- A set of principal design decisions about the system (설계 결정의 집합)

cf. MFC(Microsoft Foundation Classes)

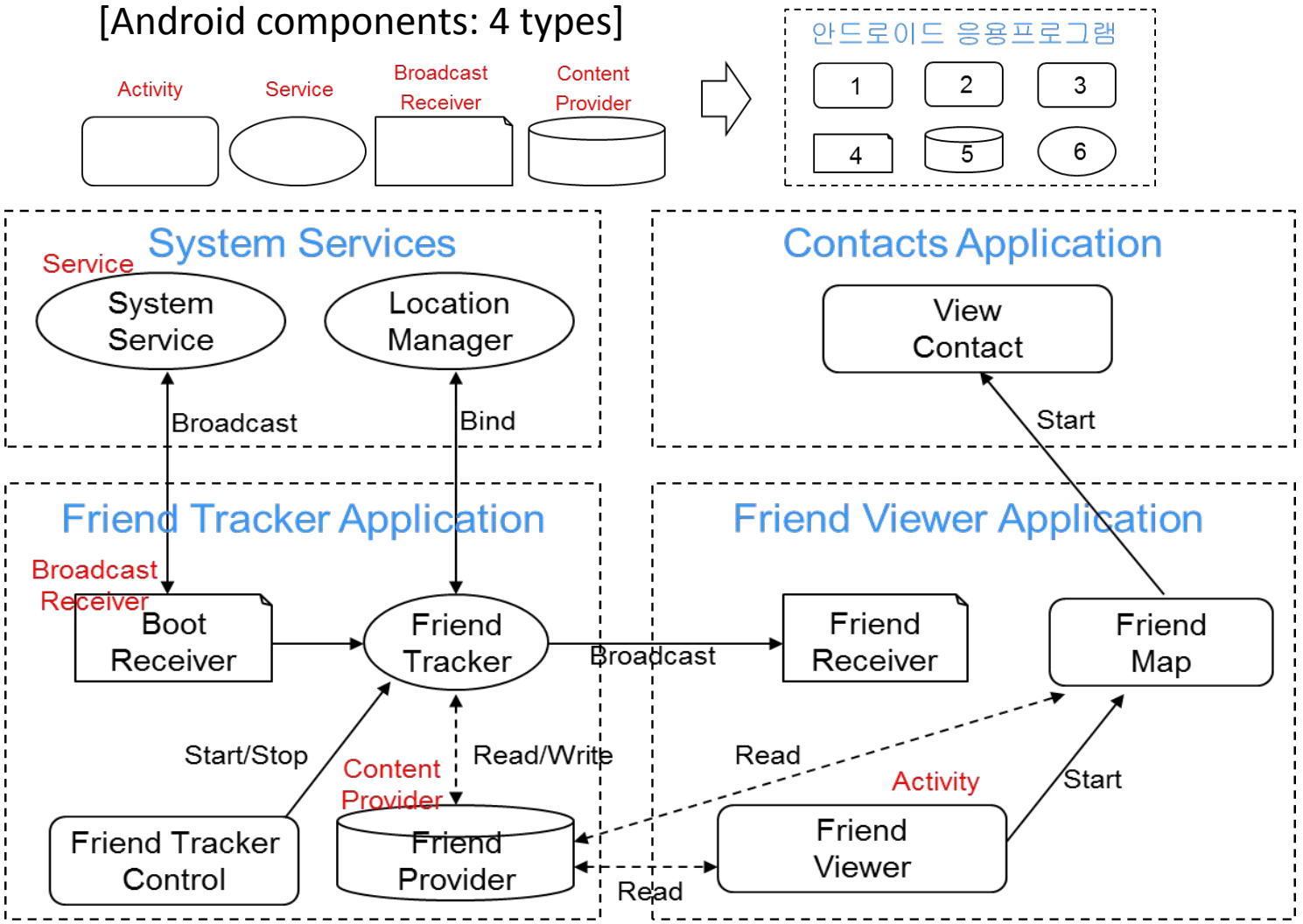
SA Example: Windows MFC

- Layered Architecture (계층 아키텍처)



SA Example: Android Apps

- Component architecture (컴포넌트 아키텍처)



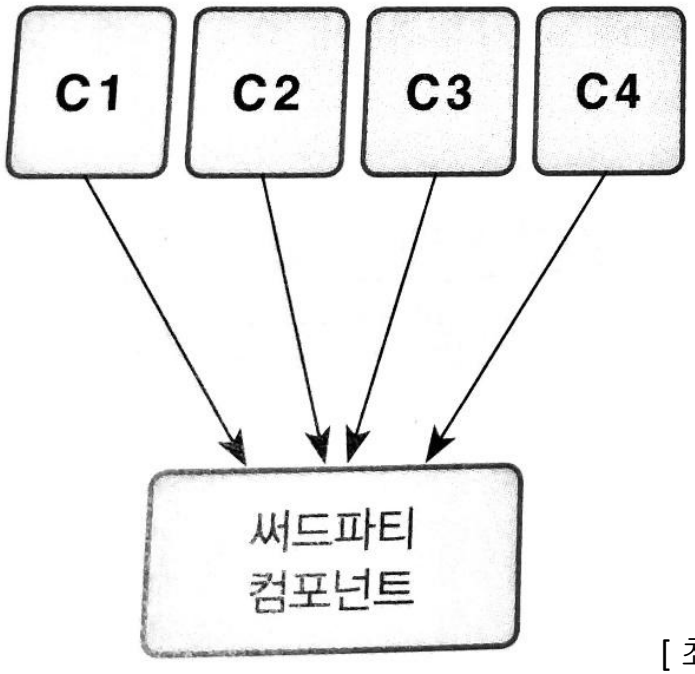
6.1 Architectural Design

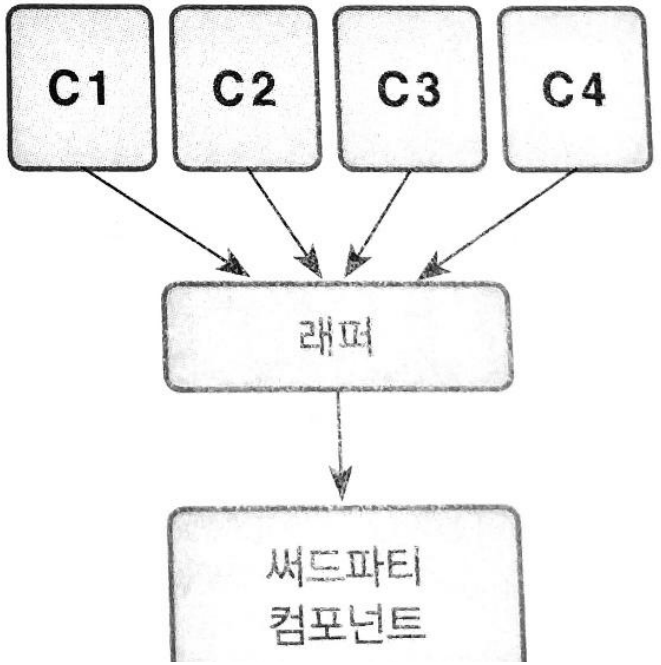
- It emphasizes the importance of components and their connectors in software system.
- It helps to control the design, development, and evolution of software system.
- It is affected from both functional and non-functional requirements.

Importance of Architectural Design

- SA design is affected from non-functional requirements

(*) Maintainability





Architecture and System Characteristics

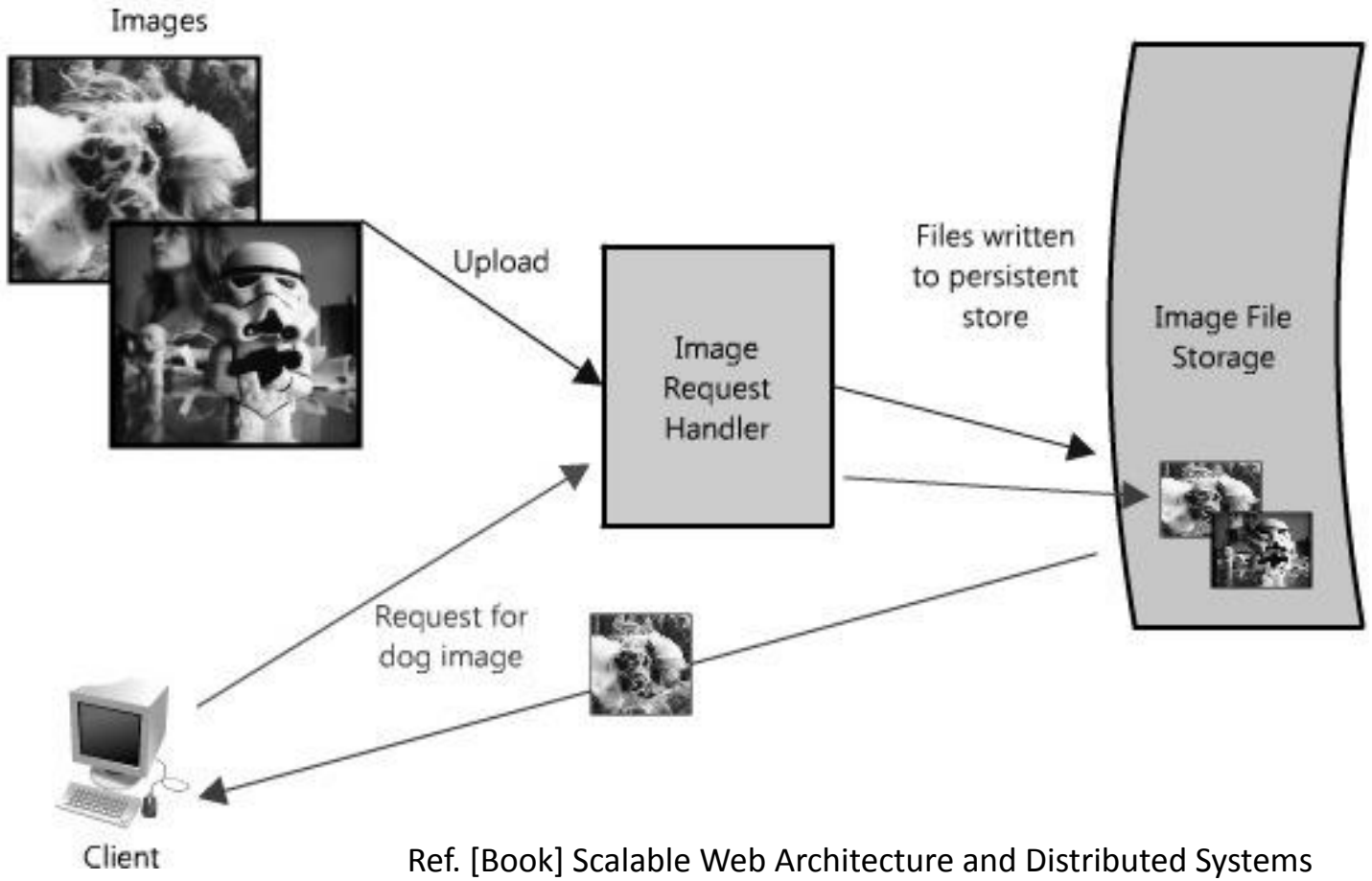
•

System Characteristics	Architectural Design Decisions (아키텍처 설계 방향)
Performance	• Localize critical operations and minimize communications.
Security	• Use a layered architecture with critical assets in the inner layers
Safety	• Localize safety-critical features in a small number of sub-systems to reduce the costs and problems of safety validation
Availability	• Include redundant components and mechanisms for fault tolerance
Maintainability	• Use fine-grain, self-contained components that may readily be changed

SA Example: Scalability & Performance

- Image Hosting Application

(1) A simplified architecture (기본 기능을 반영한 아키텍처)

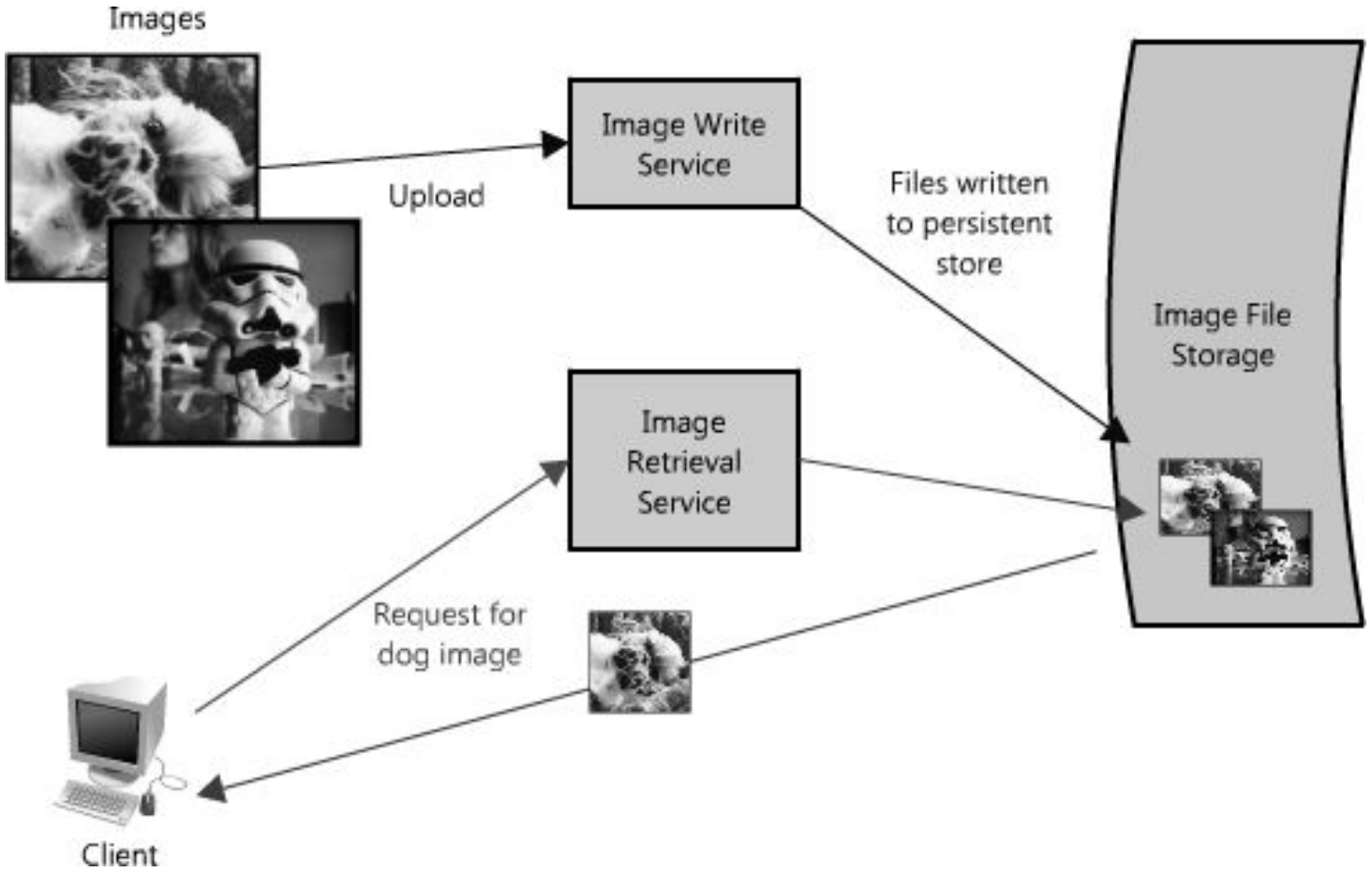


Ref. [Book] Scalable Web Architecture and Distributed Systems

SA Example: Scalability & Performance

- Image Hosting Application

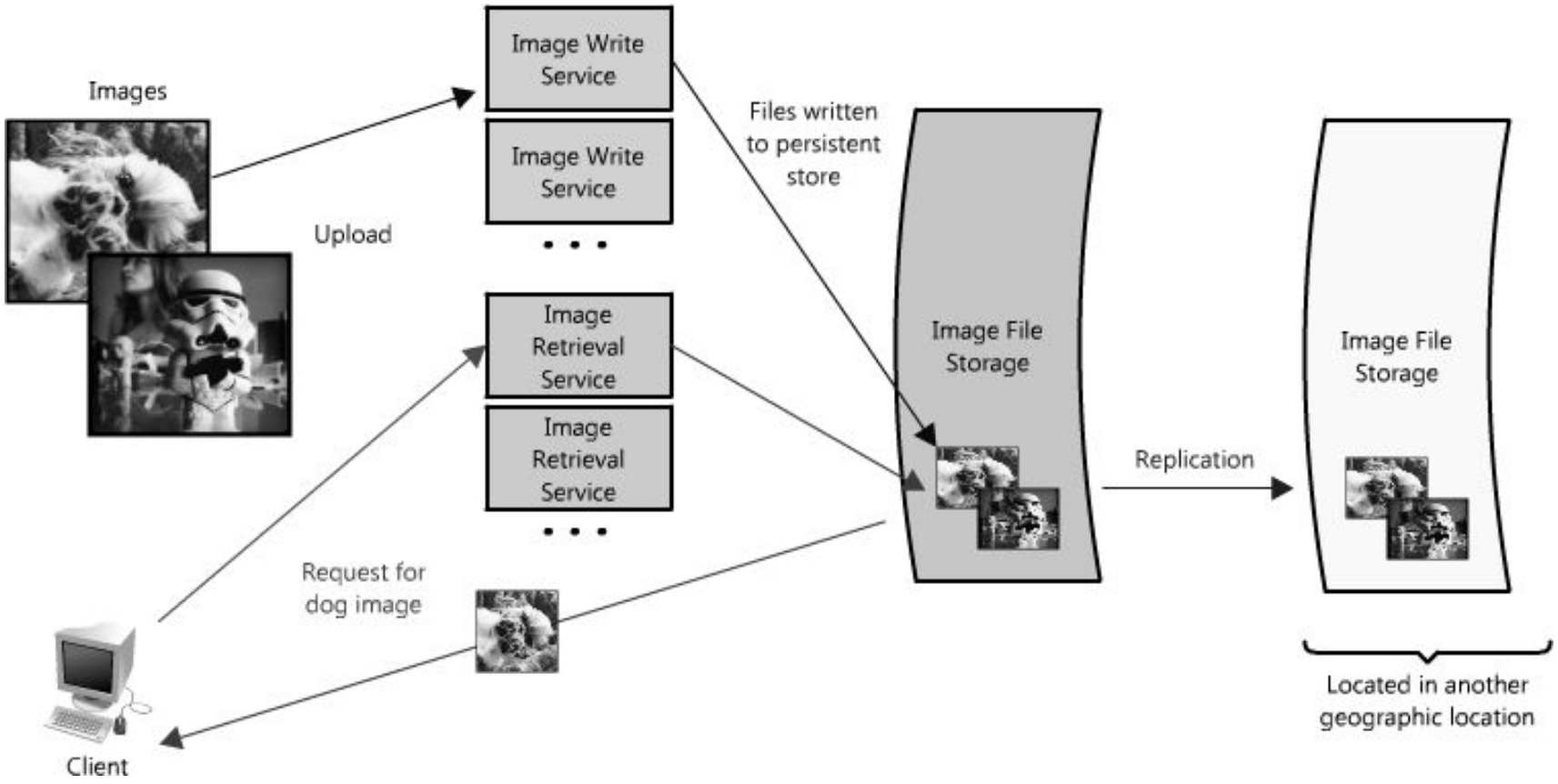
(2) Splitting out reads and writes (다운로드 > 업로드)



SA Example: Scalability & Performance

- Image Hosting Application

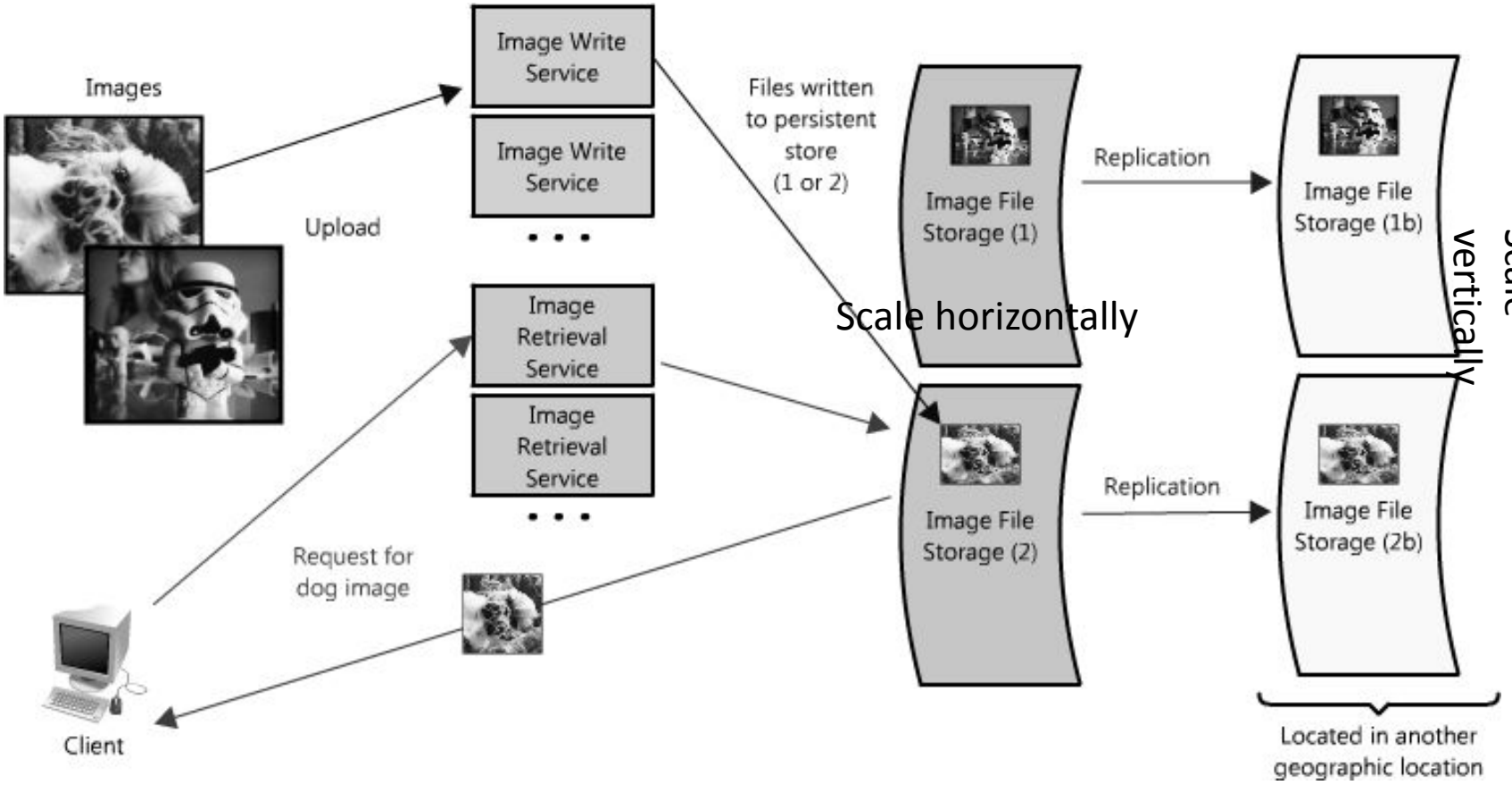
(3) Redundancy (서로 다른 지역에 동일한 데이터를 복사)



SA Example: Scalability & Performance

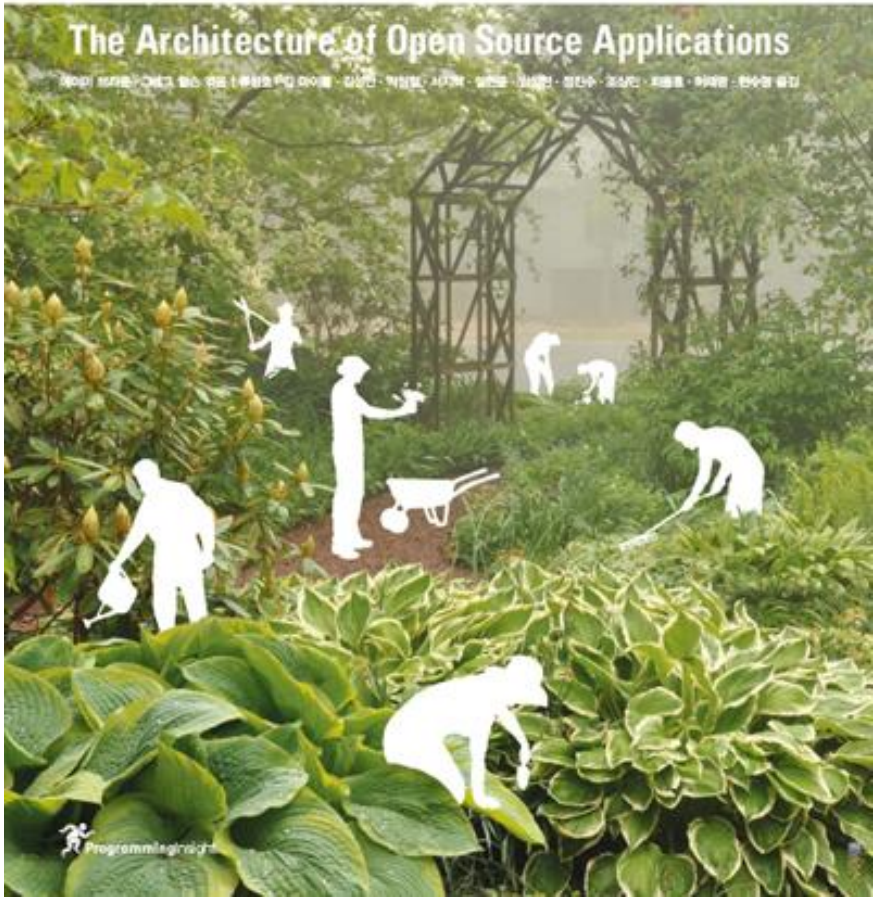
- Image Hosting Application

(4) Partitioning (하나의 사이트에 포함되기 어려운 빅데이터)



SA Examples

- 25개 애플리케이션으로 배우는
  오픈 소스 소프트웨어 아키텍처



http://aosabook.org

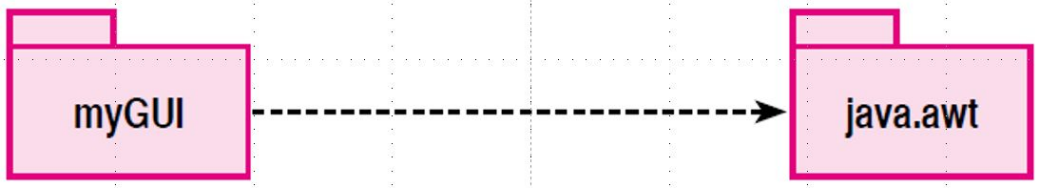
Scalable Web Architecture and Distributed Systems

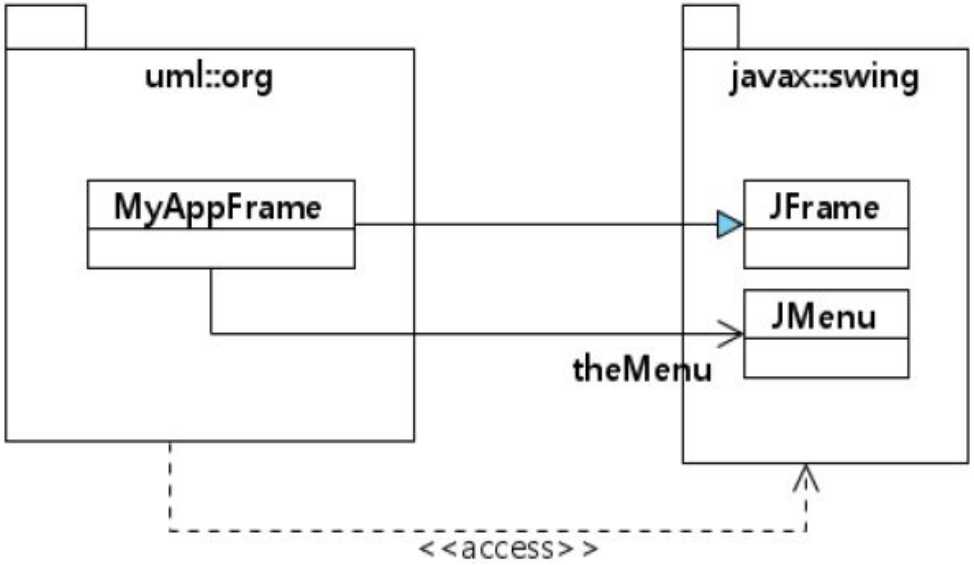
http://aosabook.org/en/distsys.html

6.2 Architectural Representations

- (1) Package Diagram in UML

[Notation]





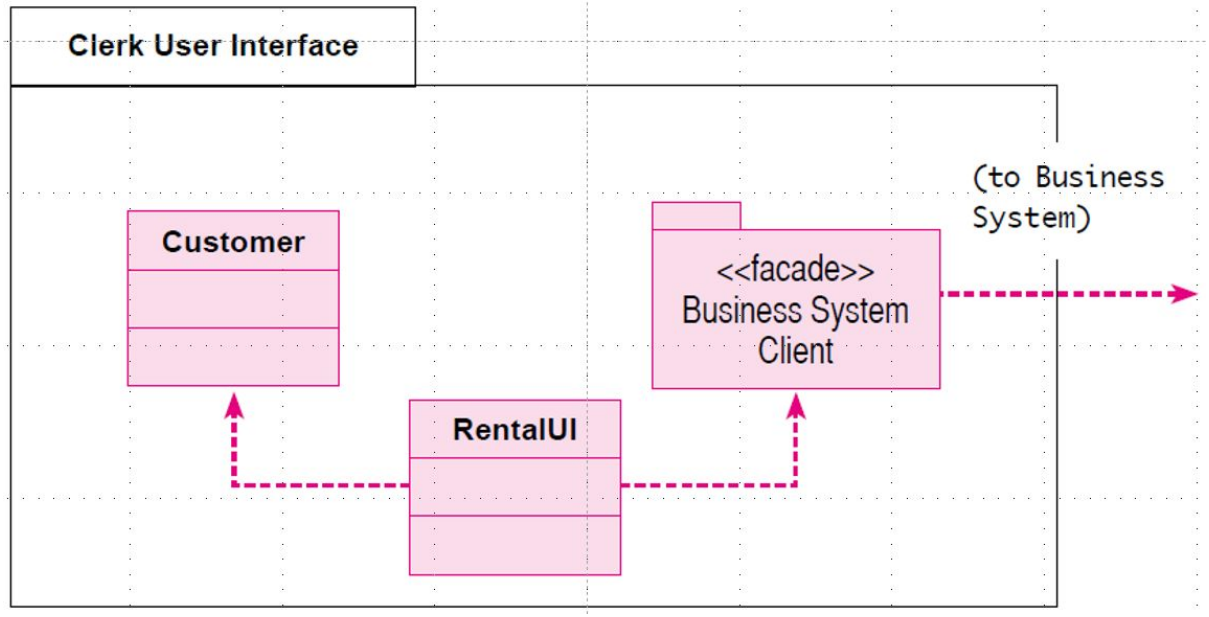

```
using namespace javax::swing ;
namespace uml {
  namespace org {
    class MyAppFrame :
      public JFrame {

      private: JMenu theMenu ;
        ...
    } ;
```

6.2 Architectural Representations

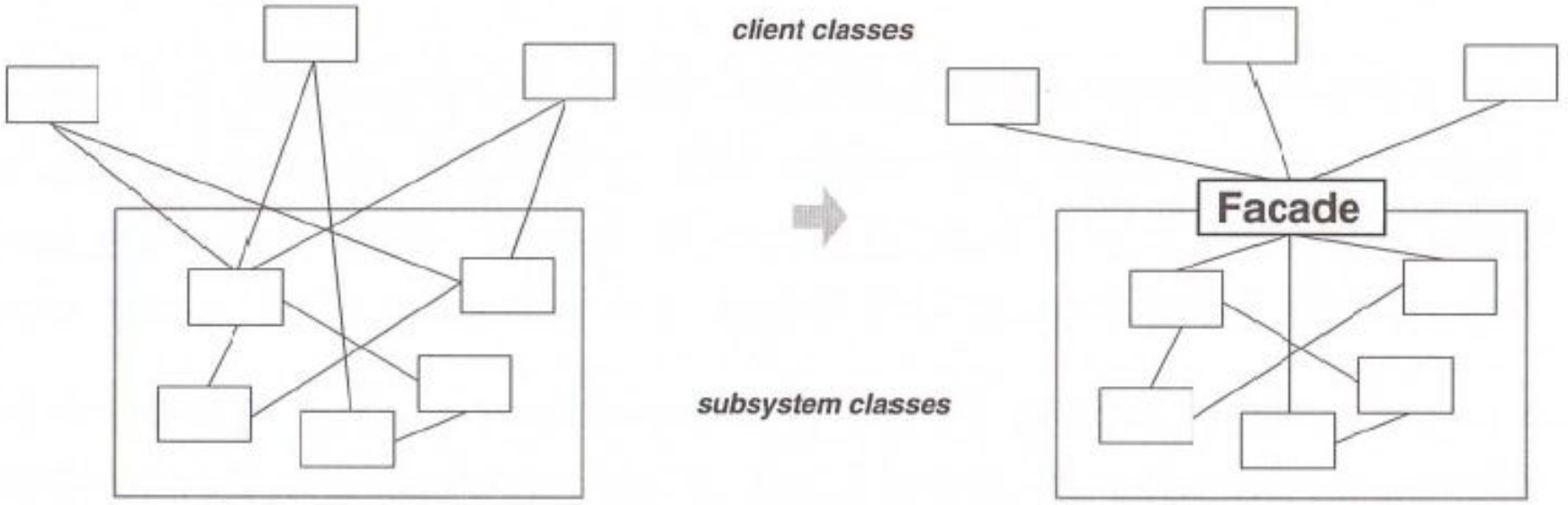
- Hierarchical Decomposition(계층적 분할) by Package Diagram

[Example]



6.2 Architectural Representations

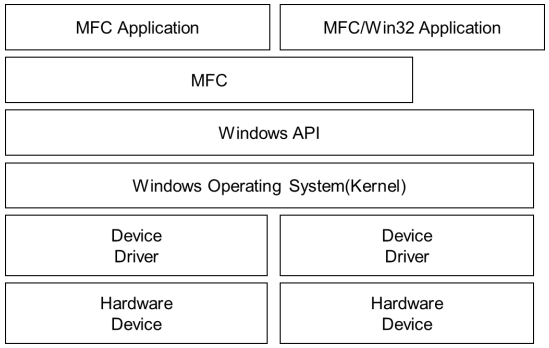
- Façade design pattern
  - provides a simplified interface to a larger body of modules and/or components in order to form a subsystem

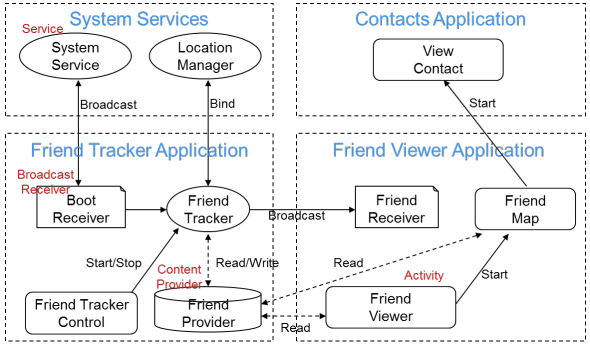


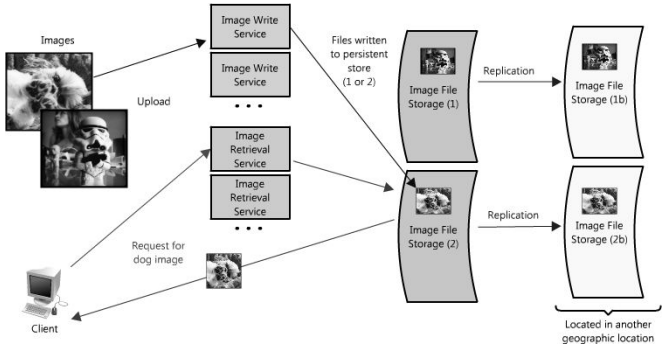
참고) 최은만 교재

6.2 Architectural Representations

- (2) Simple, informal block diagrams showing entities and relationships
  - Very useful, though being criticized because they lack semantics





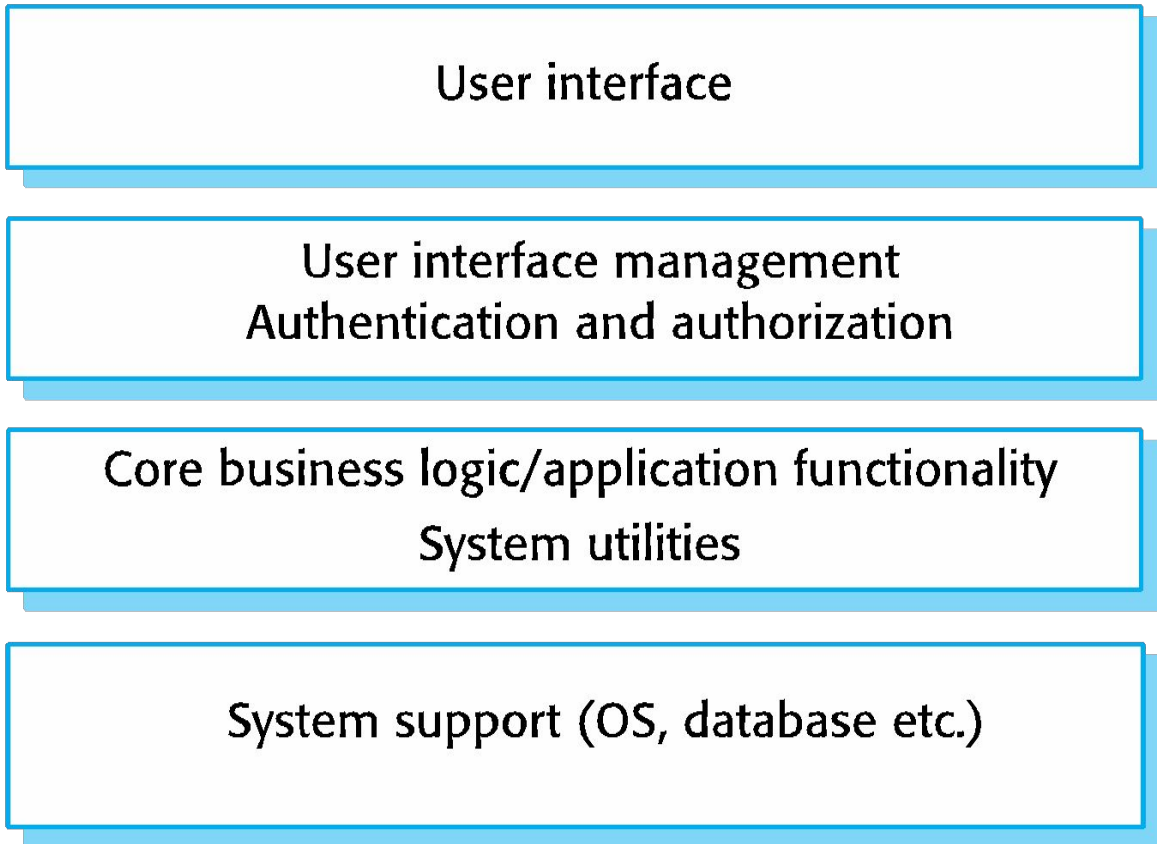


6.3 Architectural Patterns

- Layered architecture
- Client-server architecture
- Model-View-Controller (MVC)
- Event-driven control
- Repository architecture
- Pipe and filter architecture

Layered Architecture

- A generic layered structure



cf. 최은만 교재, Ian Sommerville, Software Engineering

Layered Architecture (cont.)

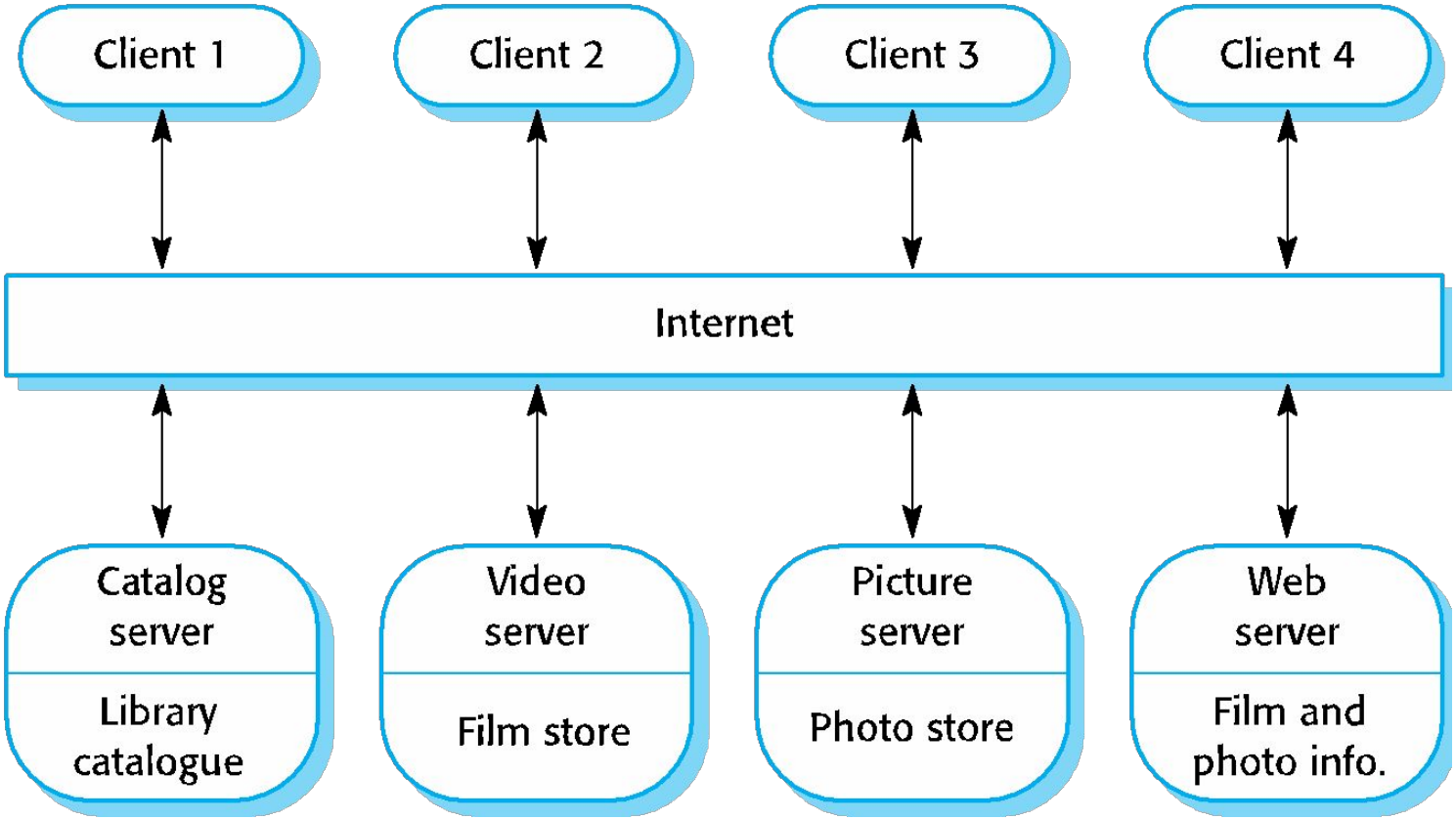
- Used to model the interfacing of sub-systems
  - Organize the system into a set of layers each of which provide a set of services

Name	Layered architecture
Description	• Organizes the system into layers with related functionality associated with each layer. (시스템을 계층적으로 구성하고 각 계층에 연관된 기능을 배치) • A layer provides services to the layer above it so the lowest-level layers represent core services that are likely to be used throughout the system. (각 계층은 그 위 계층에 서비스를 제공. 가장 아래 계층은 핵심 서비스를 표현)
Example	Windows software, Linux software, Android platform
When used	• Used when building new facilities on top of existing systems (이미 있는 시스템 위에 새로운 기능을 쌓을 때); • When the development is spread across several teams with each team responsibility for a layer of functionality (여러 팀이 각 기능 계층을 하나씩 맡아 개발할 때); • When there is a requirement for multi-level security (보안성을 여러 겹으로 강화하는 요구사항이 있을 때).

Layered Architecture (cont.)

Name	Layered architecture
Advantages	• Allows replacement of entire layers so long as the interface is maintained (인터페이스를 동일하게 유지하면 하나의 계층을 새로운 구현으로 대체할 수 있다). • Redundant facilities (e.g., authentication) can be provided in each layer to increase the dependability of the system (각 계층에 기능(예: 인증)을 중복 추가하여 신뢰성을 높일 수 있다).
Disadvantages	• In practice, providing a clean separation between layers is often difficult and a high-level layer may have to interact directly with lower-level layers rather than through the layer immediately below it (계층적으로 명백하게 구분하는 작업이 보통 어렵다. 위 계층이 바로 아래 계층을 이용하기보다 더 아래 계층을 사용할 수도 있다). • Performance can be a problem because of multiple levels of interpretation of a service request as it is processed at each layer (여러 계층을 통해서 하나의 서비스를 제공하므로 성능 문제가 발생할 수 있다).

Client-Server Architecture



cf. 최은만 교재, Ian Sommerville, Software Engineering

Client-Server Architecture (cont.)

- A set of servers providing services and a set of clients using them

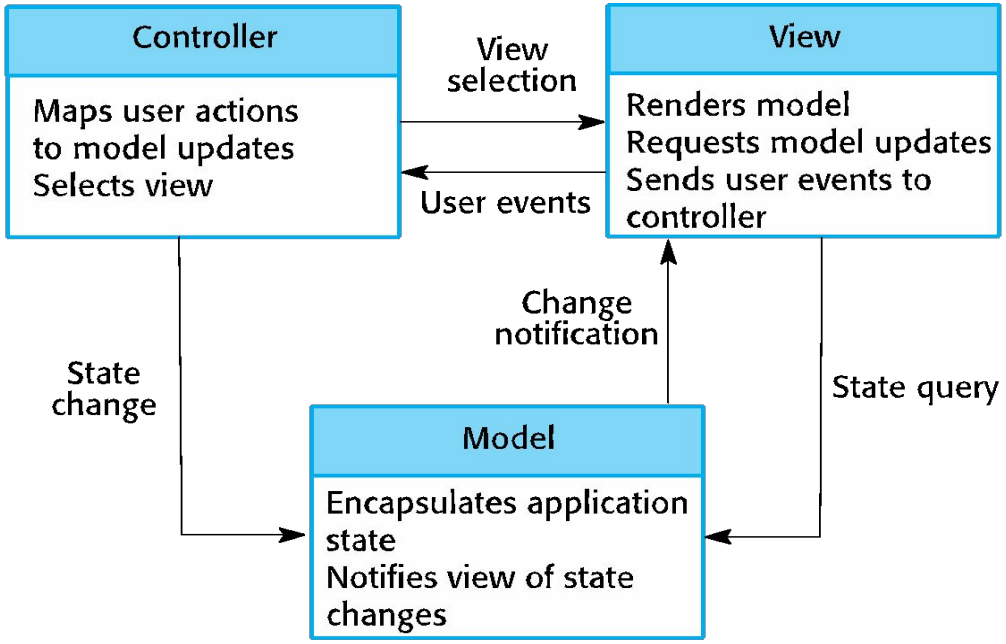
Name	Client-server
Description	• In a client–server architecture, the functionality of the system is organized into services, with each service delivered from a separate server (클라이언트-서버 아키텍처에서 시스템의 기능을 서비스로 구성하고 서버를 통하여 각 서비스를 제공). • Clients are users of these services and access servers to make use of them (클라이언트는 이 서비스 사용자이고 서버에 접속하여 이용).
Example	A film and video/DVD library organized as a client–server system. Web.
When used	• Used when data in a shared database has to be accessed from a range of locations (여러 지역에서 공유 데이터베이스를 사용할 때). • Because servers can be replicated, may also be used when the load on a system is variable (시스템 부하가 가변적이고 복제 서버를 둘 수 있을 때).

Client-Server Architecture (cont.)

Name	Client-server
Advantages	• The principal advantage of this model is that servers can be distributed across a network (네트워크로 서버들을 분산 배치할 수 있다는 구조가 이 아키텍처 모델의 주요 장점). • General functionality (e.g., a printing service) can be available to all clients (모든 사용자가 서비스를 받을 수 있다).
Disadvantages	• Each service is a single point of failure so susceptible to denial of service attacks or server failure (각 서비스는 서비스 거부 공격(DOS)이나 서버 다운에 취약). • Performance may be unpredictable because it depends on the network as well as the system (성능은 컴퓨터 시스템 뿐만 아니라 네트워크에도 의존). • May be management problems if servers are owned by different organizations(여러 다른 기관이 서버를 소유하는 경우 관리 문제 발생 가능).

Model-View-Controller (MVC)

- Idea: Separate presentation and interaction from system data

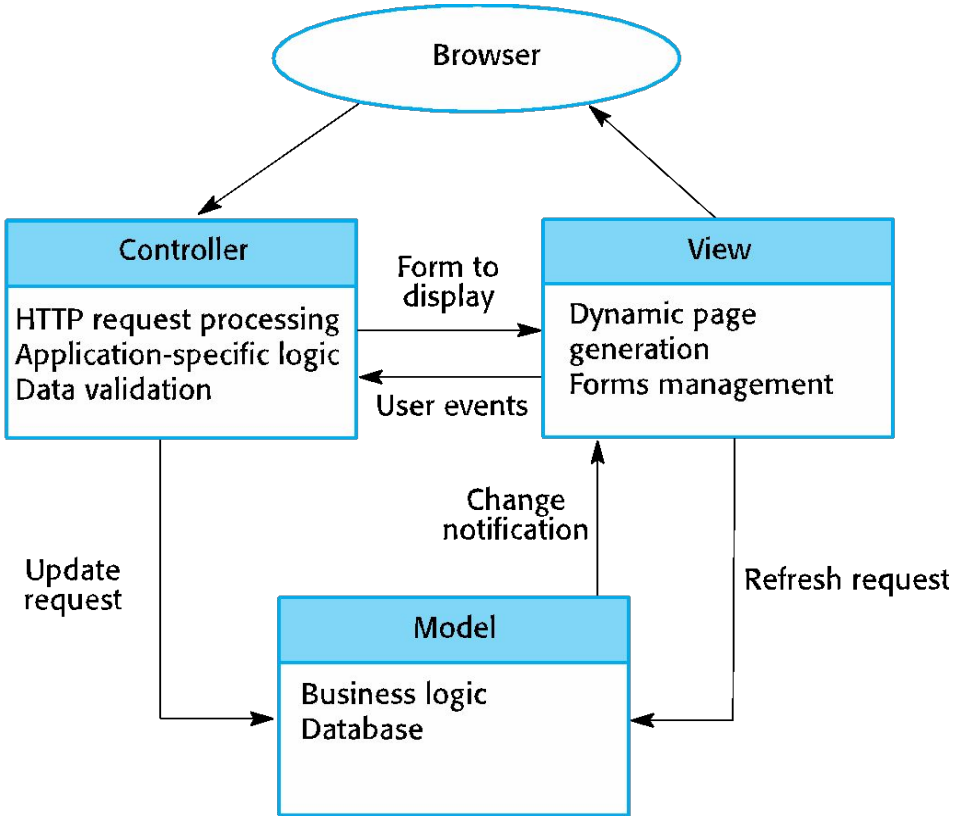


[General roles of MVC Elements]

cf. 최은만 교재, Ian Sommerville, Software Engineering

Model-View-Controller (MVC) (cont.)

- Example: Web application architecture using the MVC pattern



[MVC-based Web Application Architecture]

Model-View-Controller (MVC) (cont.)

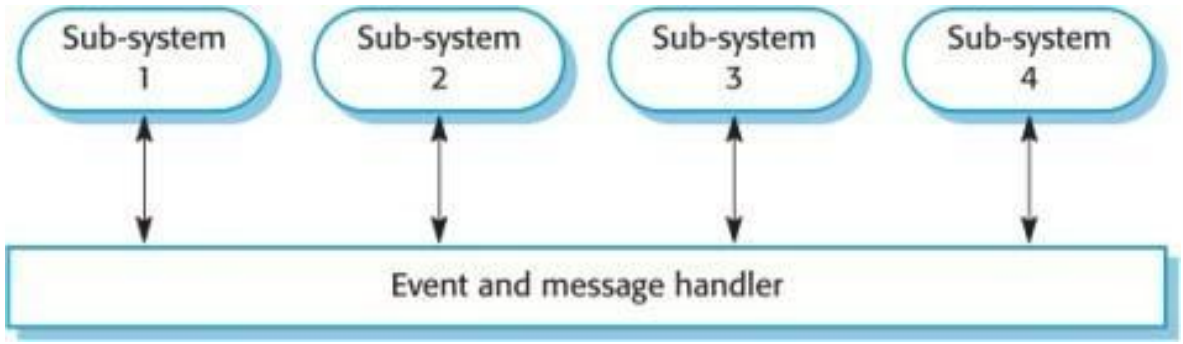
Name	MVC (Model-View-Controller)
Description	• Separates presentation and interaction from the system data (데이터를 보여주는 것과 처리하는 것을 분리). • The system is structured into three logical components that interact with each other (다음 세가지 논리적 요소로 시스템을 구성). - The Model component manages the system data and associated operations on that data (모델: 데이터 처리 담당) - The View component defines and manages how the data is presented to the user. (뷰: 데이터 보여주기. 예를 들어 동일한 데이터를 표나 그래프로 보여주기) - The Controller component manages user interaction (e.g., key presses, mouse clicks, etc.) and passes these interactions to the View and the Model. (컨트롤러: 모델과 뷰를 조합하여 : 사용자 입력을 처리)
Example	• Mobile/Web/Desktop GUI applications
When used	• Used when there are multiple ways to view and interact with data (여러 방법으로 데이터를 보여주고 다룰 때). Also used when the future requirements for interaction and presentation of data are unknown (데이터를 보여주고 다루는 방법에 대한 요구사항이 명확하지 않을 때).

Model-View-Controller (MVC) (cont.)

Name	MVC (Model-View-Controller)
Advantages	• Allows the data to change independently of its representation and vice versa (데이터 처리와 데이터 보여주는 방법을 각자 쉽게 변경 가능). • Supports presentation of the same data in different ways with changes made in one representation shown in all of them (예를 들어, 표, 그래프, 텍스트 등 여러 방법으로 동일한 데이터를 보여줄 때 한 가지 보기 방법에서 데이터를 변경하면 나머지 보기 방법들에서도 변경된 데이터를 쉽게 보여줄 수 있음).
Disadvantages	• Can involve additional code and code complexity when the data model and interactions are simple (데이터 모델과 데이터를 사용하는 방법이 간단할 때 MVC를 적용하면 코드를 더 작성하게 되고 복잡한 코드가 될 수 있음).

Event-driven Control/Arch.

- Broadcast models/Publish-subscribe models



cf. 최은만 교재, Ian Sommerville, Software Engineering

Event-driven Control (cont.)

Name	Event-driven control
Description	● In centralized control models, control decisions are usually determined by the values of some system state variables (중앙 제어 모델은 시스템 상태 변수 값에 따라 제어). ● By contrast, event-driven control models are driven by externally generated events (이벤트 기동 모델은 외부 이벤트에 따라 제어). ● Broadcast models (방송 모델): an event is, in principle, broadcast to all components (동일한 이벤트가 모든 요소에 전달). Any component that has been programmed to handle that event can respond to it (이 이벤트 처리를 등록한 요소를 실행).
Example	● IoT applications, Distributed system communications

Event-driven Control (cont.)

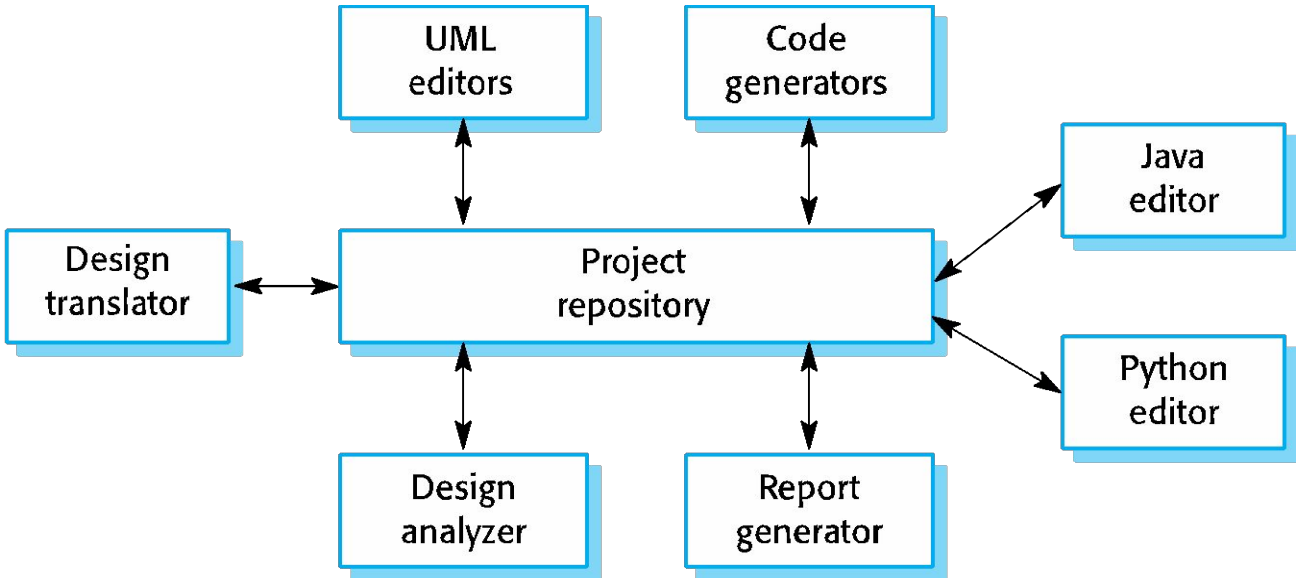
Name	Event-driven control
When used	• Broadcast models are effective in integrating components distributed across different computers on a network (방송 모델을 통해 네트워크 상에 서로 다른 컴퓨터에 분산된 컴포넌트들을 통합하는데 효과적).

Advantages	• The advantage of this broadcast approach is that evolution is relative simple (시스템 변경이 상대적으로 쉬운 장점).
	  • A new component to handle particular classes of events can be integrated by registering its events with the event handler.
	  • Any component can activate any other component without knowing its name or location.
	  • It is easy to be implemented on distributed systems.

Disadvantages	• The disadvantage of this broadcast model is that components don't know if or when events will be handled (방송 모델의 컴포넌트에서 이벤트가 처리될지 언제 처리될지 모르는 단점).
	  • When a component generates an event it does not know which other components have registered an interest in that event.
	  • It is possible for different components to register for the same events, which may cause conflicts in handling the event.

Repository Architecture

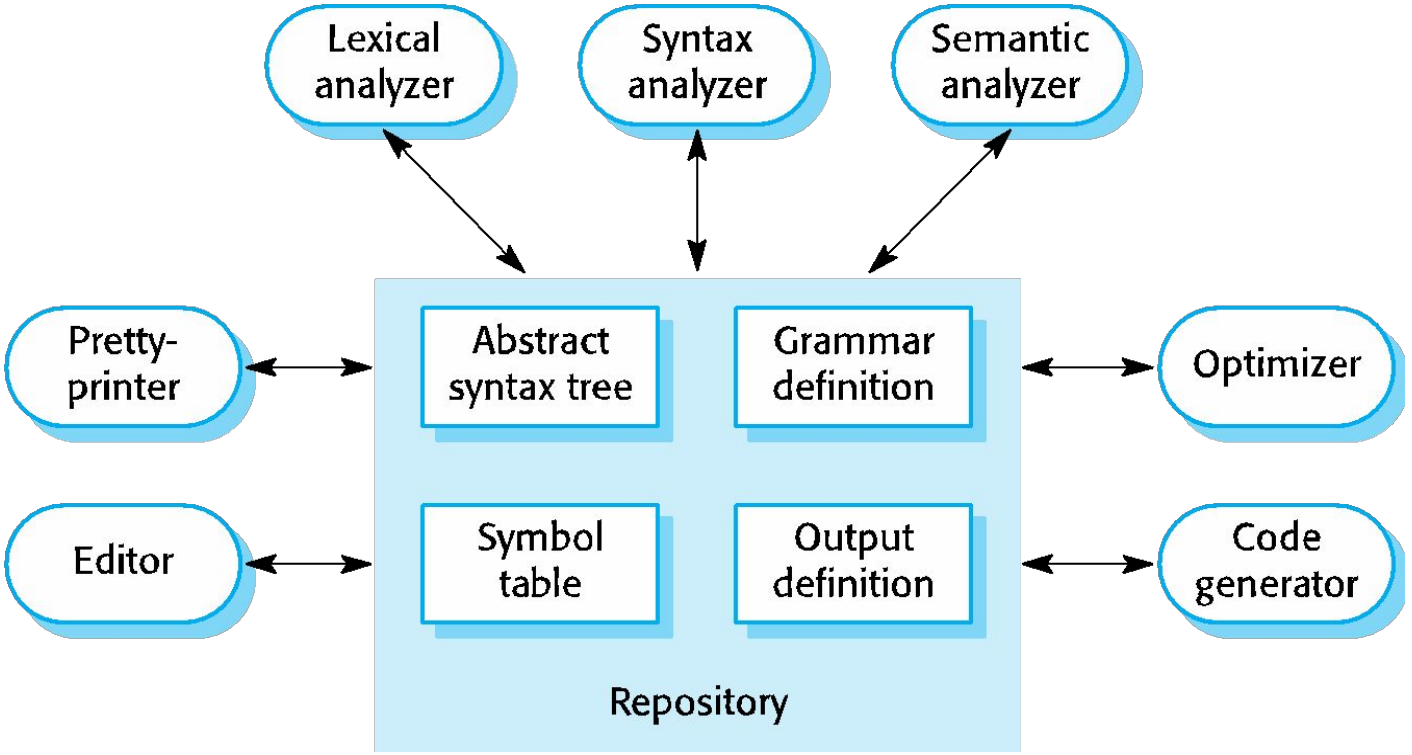
- For sub-systems exchanging data in two ways:
  - Shared data are held in a central database or repository accessible by all the sub-systems
  - Each sub-system maintains its own database and passes data explicitly to other sub systems



cf. 최은만 교재, Ian Sommerville, Software Engineering

Repository Architecture (cont.)

- Example: A repository architecture for a language processing system such as IDE



cf. 최은만 교재, Ian Sommerville, Software Engineering

Repository Architecture (cont.)

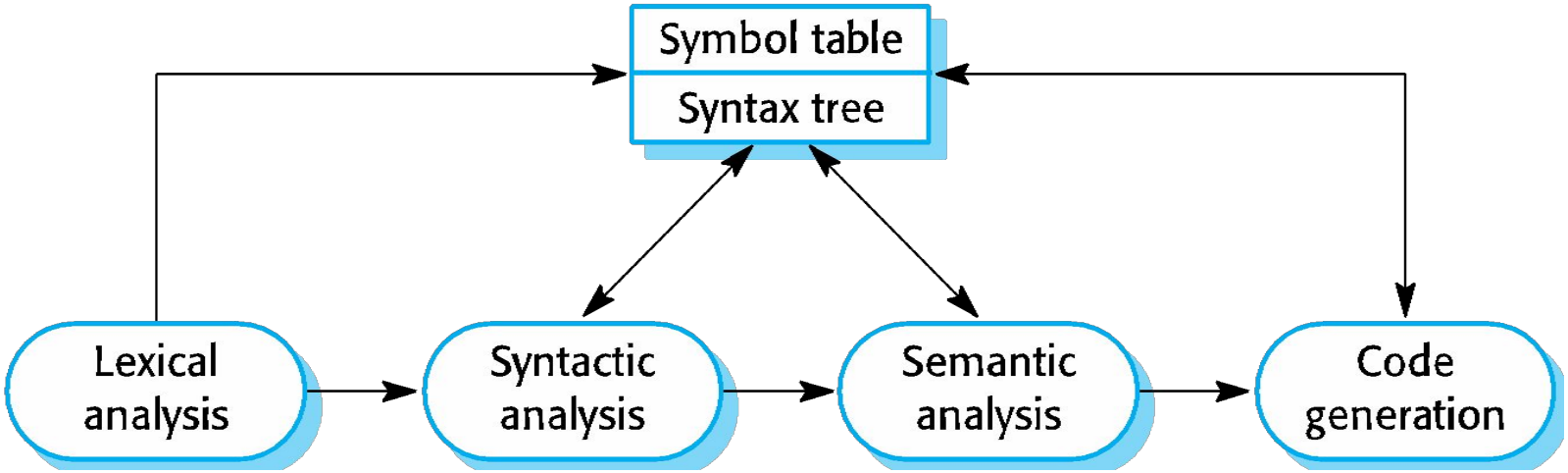
Name	Repository
Description	• All data in a system is managed in a central repository that is accessible to all system components (시스템의 모든 데이터를 중앙 저장소에서 관리하고 모든 컴포넌트들을 접근 가능하도록 한다). • Components do not interact directly, only through the repository (컴포넌트들은 서로 직접 데이터를 교환하지 않고 이 저장소를 통한다).
Example	• IDE where the (plug-in) components use a repository of system design information. • Each software tool generates information, and it is available for use by other tools.
When used	• You should use this pattern when you have a system in which large volumes of information are generated that has to be stored for a long time (큰 용량의 정보를 만들어 오래 저장하는 유형의 시스템). • You may also use it in data-driven systems where the inclusion of data in the repository triggers an action or tool (새로운 데이터를 중앙 저장소에 추가하면 어떤 액션이나 도구를 실행하는 데이터 구동 시스템).

Repository Architecture (cont.)

Name	Repository
Advantages	• Components can be independent—they do not need to know of the existence of other components (컴포넌트들은 서로 독립적). • Changes made by one component can be propagated to all components (어느 컴포넌트에서 중앙 저장소를 변경하면 모든 컴포넌트들도 이 변경된 저장소를 갖게 된다). • All data can be managed consistently (e.g., backups done at the same time) as it is all in one place (중앙 저장소의 데이터를 일관성 있게 관리할 수 있다).
Disadvantages	• The repository is a single point of failure so problems in the repository affect the whole system (중앙 저장소에 문제가 있으면 전체 시스템에 영향을 준다). • May be inefficiencies in organizing all communication through the repository (중앙 저장소를 통해서만 데이터를 주고 받을 수 있는 비효율적 구성). • Distributing the repository across several computers may be difficult (여러 컴퓨터들에 동일한 중앙 저장소를 분산시키기가 어렵다).

Pipe and Filter Architecture

- Transformative architecture
  - suitable for a batch sequential model



cf. 최은만 교재, Ian Sommerville, Software Engineering

Pipe and Filter Architecture (cont.)

Name	**Pipe and filter**
Description	• The processing of the data in a system is organized so that each processing component (filter) is discrete and carries out one type of data transformation (각 데이터 처리 컴포넌트(필터)는 명확히 구분되어 있고 한가지 유형의 데이터 변환을 담당한다). • The data flows (as in a pipe) from one component to another for processing (한 컴포넌트가 출력한 데이터를 다른 컴포넌트로 입력하는 형태로 처리).
Example	• A compiler architecture.
When used	• Commonly used in data processing applications (both batch- and transaction-based) where inputs are processed in separate stages to generate related outputs (배치 또는 트랜잭션 방식의 데이터 처리 응용 프로그램에서 흔히 사용. 사용자 입력은 여러 단계를 통해 처리되어 최종 출력을 낸다).

cf. 교재 6.4.3 트랜잭션 처리 스타일

Pipe and Filter Architecture (cont.)

Name	Pipe and filter
Advantages	• Easy to understand and supports transformation reuse (이해하기 쉽고, 변환 필터 재사용이 쉬움). • Workflow style matches the structure of many business processes (비즈니스 처리 구조에 적합). • Evolution by adding transformations is straightforward (나중에 변환을 추가하기 쉬움). • Can be implemented as either a sequential or concurrent system (순차 또는 동시 시스템으로 구현 가능).
Disadvantages	• The format for data transfer has to be agreed upon between communicating transformations (서로 인접한 변환들 간의 데이터 형식을 미리 약속해야 함). • Each transformation must parse its input and unparse its output to the agreed form (각 변환 선행/후행 작업이 필요. 변환 전에 입력을 자료구조로 바꾸고 처리한 다음 이 자료 구조를 출력). This increases system overhead and may mean that it is impossible to reuse functional transformations that use incompatible data structures (이 구조로 인해 시스템 부하가 늘어남. 호환되지 않는 자료 구조를 사용하는 변환을 재사용할 수 없음).