6. 아키텍처 설계

학습목표

- 아키텍처 설계 결정
- 시스템 구성
- 모듈 분해 스타일
- 제어 스타일
- 모듈 분할의 평가 기준

소프트웨어 아키텍처

- 소프트웨어의 골격이 되는 기본 구조
- 시스템을 구성하는 서브시스템들을 식별하고, 서브시스템의 제어와 통신을 위한 프레임워크를 설정하는 설계 프로세스가 아키텍처 설계
- 이러한 설계 프로세스의 산출물이 소프트웨어 아키텍처를 기술한 것

아키텍처 설계

- 시스템 설계 프로세스의 첫 번째 단계
- 요구사항 분석 프로세스와 설계 프로세스 사이의 연결을 나타냄
- 특정 명세화 활동과 동시에 진행되는 것이 보통
- 주요 시스템 컴포넌트들과 이들간의 통신을 식별하는 것을 포함

아키텍처 설계, 문서화의 장점

- Stakeholder 사이의 의사소통
  - 아키텍처는 시스템 stakeholder 간의 논의의 초점으로 이용 가능
- 시스템 분석
  - 시스템이 비기능적 요구사항을 만족할 수 있는지의 여부를 분석하는 것을 의미
- 대규모 재사용
  - 아키텍처는 시스템 간에 재사용 가능

아키텍처와 비기능적 요구사항

- 성능 (Performance)
  - 중요한 연산을 지역화시켜 통신을 최소화. 단위가 큰 컴포넌트를 사용
- 보안성 (Security)
  - 중요한 자산을 내부 계층에 두는 계층 아키텍처를 사용
- 안전성 (Safety)
  - 안전성이 중요한 부분을 작은 수의 서브시스템으로 국한시킴
- 가용성 (Availability)
  - 여분의 컴포넌트와 결함 내성을 위한 메커니즘을 포함
- 유지보수성 (Maintainability)
  - 단위가 작은 대치 가능한 컴포넌트를 사용

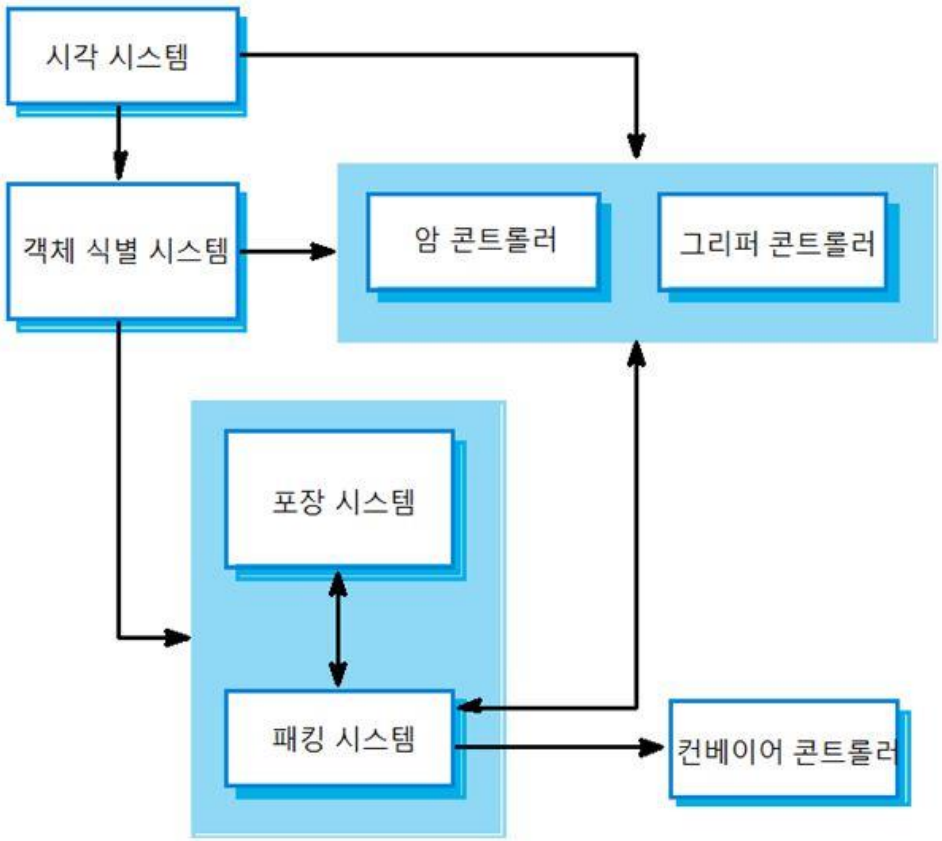
아키텍처 충돌

- 단위가 큰 컴포넌트를 사용하면 성능은 향상되지만 유지보수성이 감소
- 여분의 데이터가 도입되면 가용성은 향상되지만 보안성을 유지하기 더욱 어려워짐
- 안전성에 관련된 부분을 지역화하면 대개 통신이 증가하여 성능이 저하됨

시스템 구조화

- 시스템을 서브시스템으로 분해
- 아키텍처 설계는 대개 시스템 구조의 개관을 제시하는 블록 다이어그램으로 표현
- 서브시스템의 데이터 공유, 분산, 다른 것들과의 인터페이스를 나타내는 더욱 구체적인 모델이 또한 개발될 수 있음

택배 포장 시스템



블록 다이어그램: 구성요소 간의 관계와 시스템의 추상적 관점 이해.
독립적으로 개발될 주요 서브시스템 식별 가능.

아키텍처 설계 결정

- 아키텍처 설계는 창조적인 프로세스로 개발되는 시스템의 유형, **아키텍트**의 배경과 경험, 시스템의 특정 요구사항에 좌우됨
- 그러나 다수의 공통적인 결정이 존재함

아키텍처 설계 결정은 다음 질문의 답을 통해

- 일반적인 응용 시스템의 아키텍처?
- 시스템이 어떻게 분산?
- 적절한 아키텍처 스타일?
- 시스템을 구성하는 접근법?
- 모듈 분해 방법?
- 사용되는 제어 전략?
- 아키텍처 설계 평가 방법?
- 아키텍처 문서화 방법?

아키텍처 재사용

- 동일한 도메인의 시스템은 도메인 개념을 반영하는 유사한 아키텍처를 가짐
- 응용 시스템의 제품 라인 (product line)이 핵심 아키텍처를 중심으로 구축되고, 특정 고객의 요구사항을 만족하는 변동 부분이 있음

아키텍처 스타일

- 특정 시스템의 아키텍처 모델은 일반 아키텍처 모델이나 아키텍처 스타일과 일치
- 이러한 아키텍처 스타일을 이해하면 시스템 아키텍처를 정의하는 문제를 단순화할 수 있음
- 그러나 대부분의 대형 시스템은 이질적이므로, 단 하나의 아키텍처 스타일만을 따르는 것은 아님

아키텍처 모델의 다양한 관점

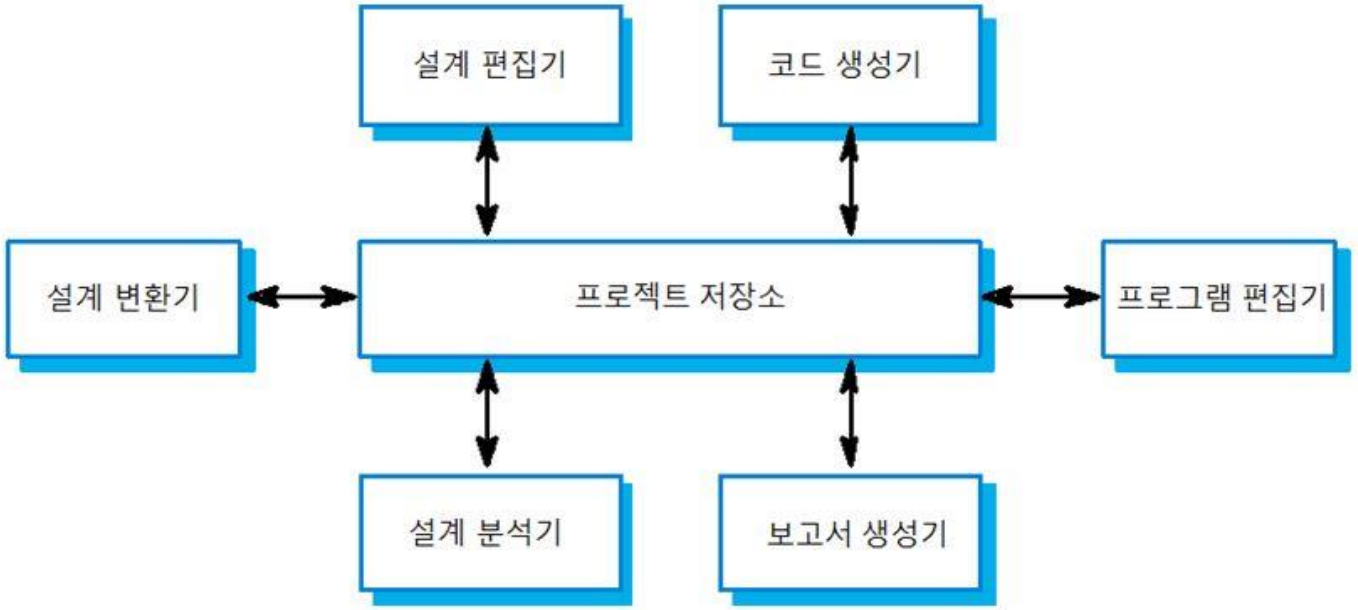
- 아키텍처 설계의 문서화에 이용
- 정적 구조 모델은 주요 시스템 컴포넌트를 나타냄
- 동적 프로세스 모델은 시스템의 프로세스 구조를 나타냄
- 인터페이스 모델은 서브시스템 인터페이스를 정의
- 데이터 흐름 모델과 같은 관계 모델은 서브시스템 간의 관계를 나타냄
- 분산 모델은 서브시스템이 컴퓨터 상에 어떻게 분산되는지를 나타냄

시스템 구성

- 시스템을 구조화하는 기본 전략을 반영
- 널리 이용되는 세 가지 모델
  - 공유 데이터 저장소 모델
  - 클라이언트 서버 모델
  - 추상 기계 혹은 계층 모델

CASE 도구 집합 아키텍처 – 저장소 모델

- 모든 공유 데이터가 중앙의 데이터베이스에 보관되어 모든 서브시스템들이 접근할 수 있다. 공유 데이터베이스에 기반을 둔 시스템 모델을 **저장소 모델**이라고 부른다.
- 각 서브시스템이 자신의 데이터베이스를 유지한다. 데이터는 서브시스템 사이에 메시지를 전달하여 상호 교환된다.



공유 저장소 모델의 장단점

❖ 장점

① 공유 저장소는 다량의 데이터를 공유하는 효과적인 방식이다.

② 데이터를 생성하는 서브시스템은 다른 시스템들이 이 데이터를 어떻게 사용하는지에 대해 알 필요가 없다.

③ 백업, 보안, 접근 제어, 오류로부터의 복구와 같은 활동들은 중앙 집중화된다.

④ 합의된 데이터 모델에 적합한 새로운 도구들을 통합하는 것은 간단하다.

❖ 단점

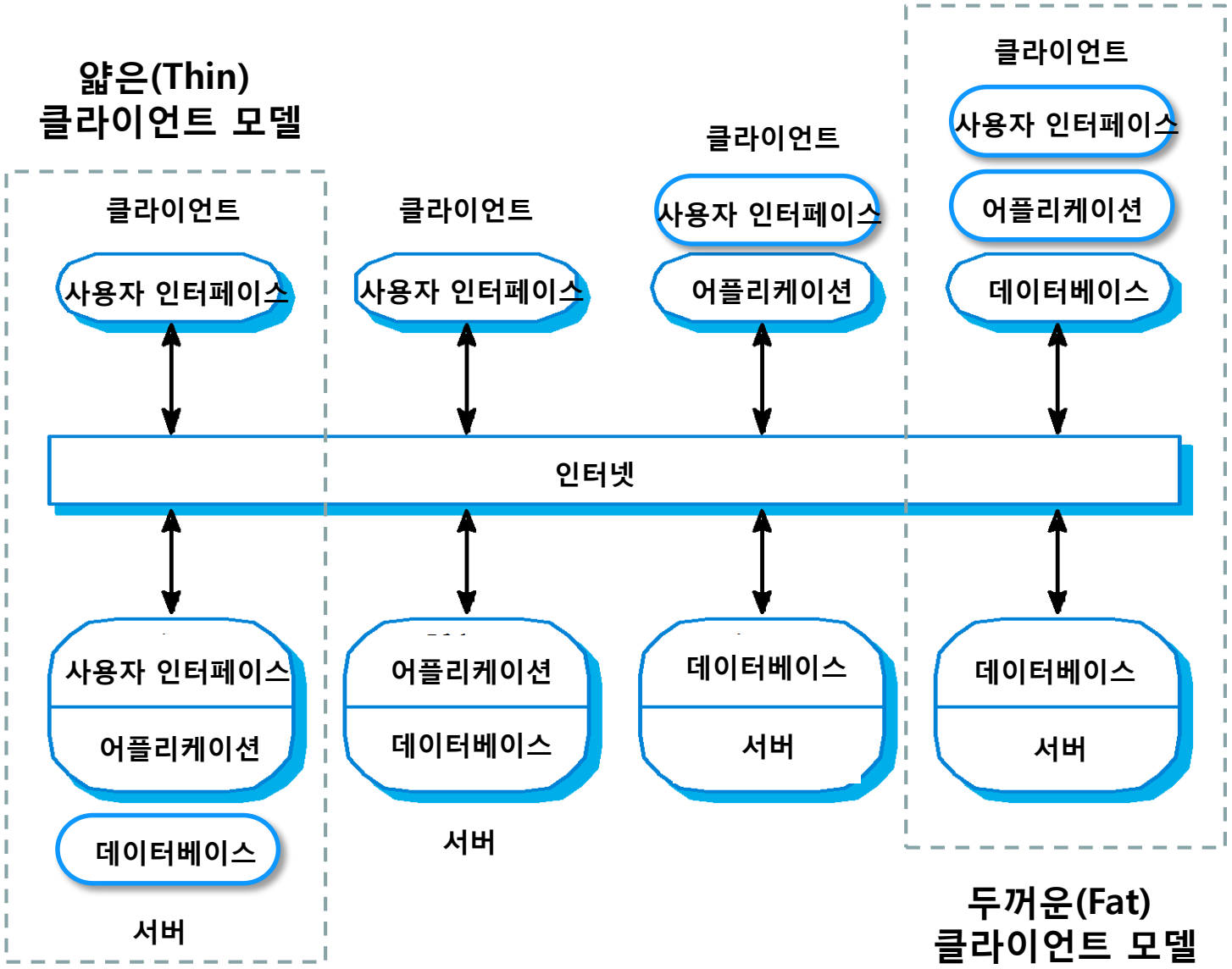
① 서브시스템들은 저장소 데이터 모델에 대해 합의를 해야 한다.

② 합의된 데이터 모델에 맞추어 다량의 정보가 생성됨에 따라서 진화가 어려울 수도 있다.

③ 상이한 서브시스템들은 보안, 복구, 백업 정책에 관해 서로 다른 요구사항을 가질 수도 있다.

④ 저장소를 다수의 컴퓨터로 분산시키는 것은 어려울 수 있다.

클라이언트 서버 모델


얇은(Thin)
클라이언트 모델
클라이언트
사용자 인터페이스
클라이언트
사용자 인터페이스
클라이언트
사용자 인터페이스
어플리케이션
클라이언트
사용자 인터페이스
어플리케이션
데이터베이스
인터넷
사용자 인터페이스
어플리케이션
데이터베이스
어플리케이션
데이터베이스
데이터베이스
서버
데이터베이스
서버
서버
두꺼운(Fat)
클라이언트 모델

클라이언트 서버 모델의 장단점

- 장점

① 분산 아키텍처. 많은 분산 프로세서를 가지는 네트워크 시스템을 효과적으로 이용할 수 있다. 새로운 서버를 추가하여 시스템의 나머지와 통합하거나 시스템의 다른 부분에 영향을 주지 않고 서버를 투명하게 업그레이드하는 것이 용이하다.

- 단점

① 새로운 서버의 통합을 통한 위해 기존의 클라이언트와 서버에 대한 변경이 요구될 수 있다. 서버 간에 공유하는 데이터 모델이 존재하지 않아 서브시스템들은 상이한 방식으로 자신들의 데이터를 구성할 수도 있다. 이것은 특정한 데이터 모델이 각 서버별로 설정되어 성능을 최적화할 수도 있다는 것을 의미한다. 물론 만일 XML 기반의 데이터 표현이 이용되면, 하나의 스키마를 다른 스키마로 변환하는 것은 비교적 쉬울 수 있다. 그러나 XML은 데이터를 표현하는 비효율적인 방식이므로, 만일 XML이 이용되면 성능 문제가 야기될 수 있다.

Thin & fat 클라이언트

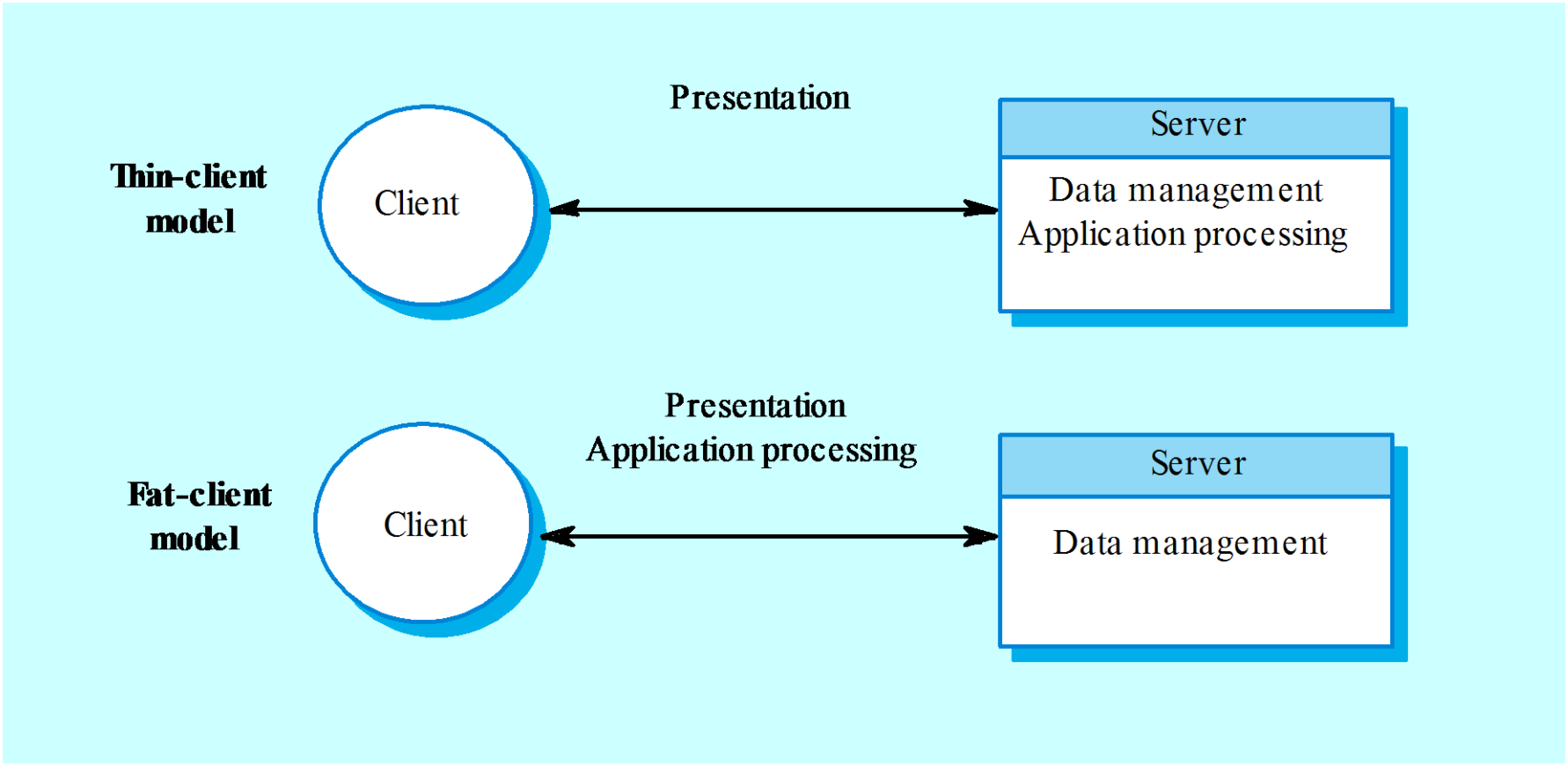
❖ Thin-client model (얇은 클라이언트 모델)

- 모든 애플리케이션의 처리와 데이터 관리가 서버에서 수행됨. 클라이언트는 단순히 표현(presentation) 소프트웨어의 실행에 책임을 짐

❖ Fat-client model (두꺼운 클라이언트 모델)

- 서버는 오직 데이터 관리만을 책임짐. 클라이언트 상의 소프트웨어는 애플리케이션 논리와 사용자와의 상호작용을 구현함

Thin & fat 클라이언트



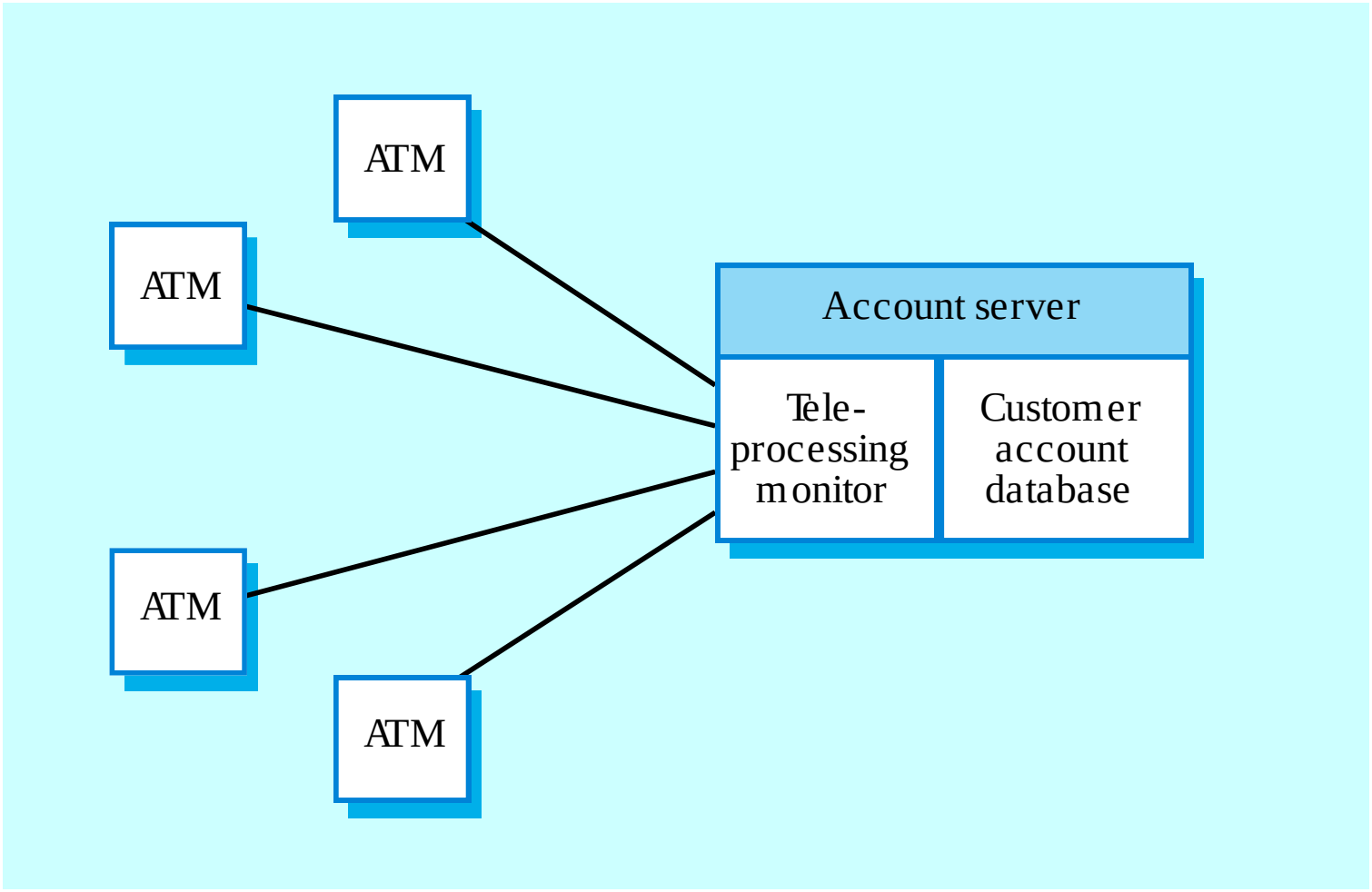
Thin client model

- 레거시 시스템이 클라이언트 서버 아키텍처로 변환될 때 사용됨
  - 레거시 시스템 자체는 서버로 동작하고 그래픽 인터페이스는 클라이언트에서 구현됨.
- 주요 단점은 서버와 네트워크 모두에 커다란 처리 부하가 가해지는 것임

Fat client model

- 많은 처리가 클라이언트에 위임되어 애플리케이션이 논리적으로 수행됨
- 클라이언트 시스템의 기능이 사전에 알려진 새로운 C/S 시스템에 적합
- 관리에 관해서는 thin client model보다 더 복잡. 애플리케이션의 새로운 버전은 모든 클라이언트에 설치되어야 함

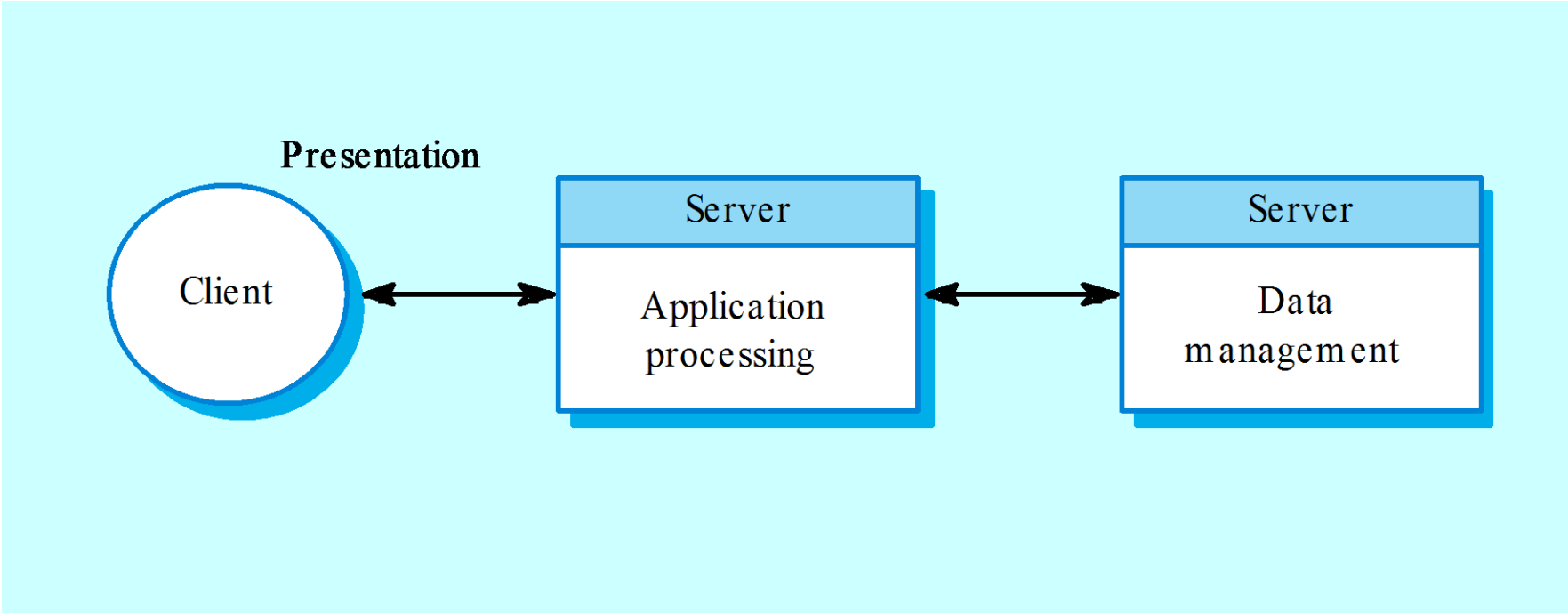
클라이언트-서버 ATM 시스템



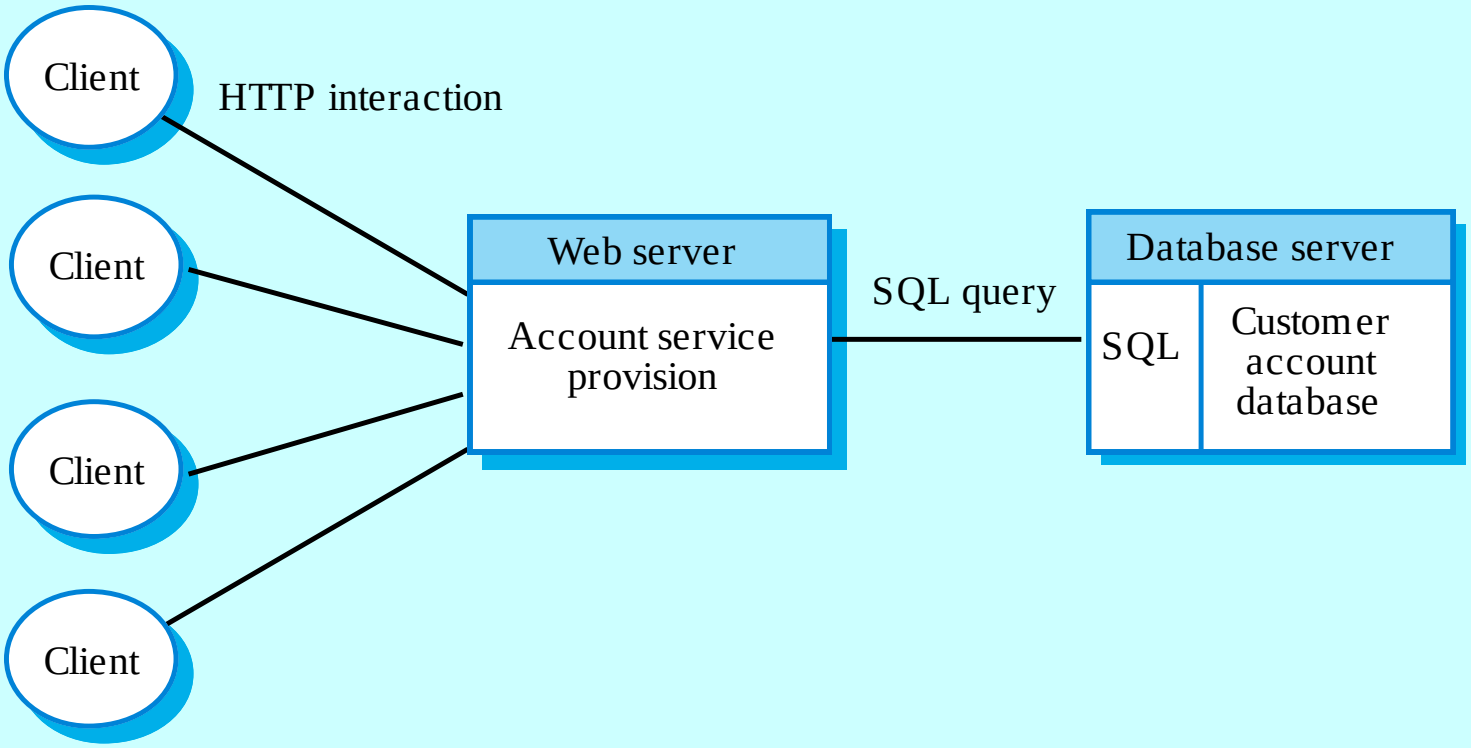
3단(Three-tier) 아키텍처

- 3단 아키텍처에서 각 애플리케이션 아키텍처 계층은 별도의 프로세서에서 실행됨
- thin-client 방법보다는 성능이 우수하고 fat-client 방법보다는 관리가 단순함
- 더욱 확장 가능한 아키텍처 – 요구가 증가하면, 별도의 서버들을 추가할 수 있음

3단 C/S 아키텍처


Presentation
Client
Server
Application processing
Server
Data management

인터넷 뱅킹 시스템

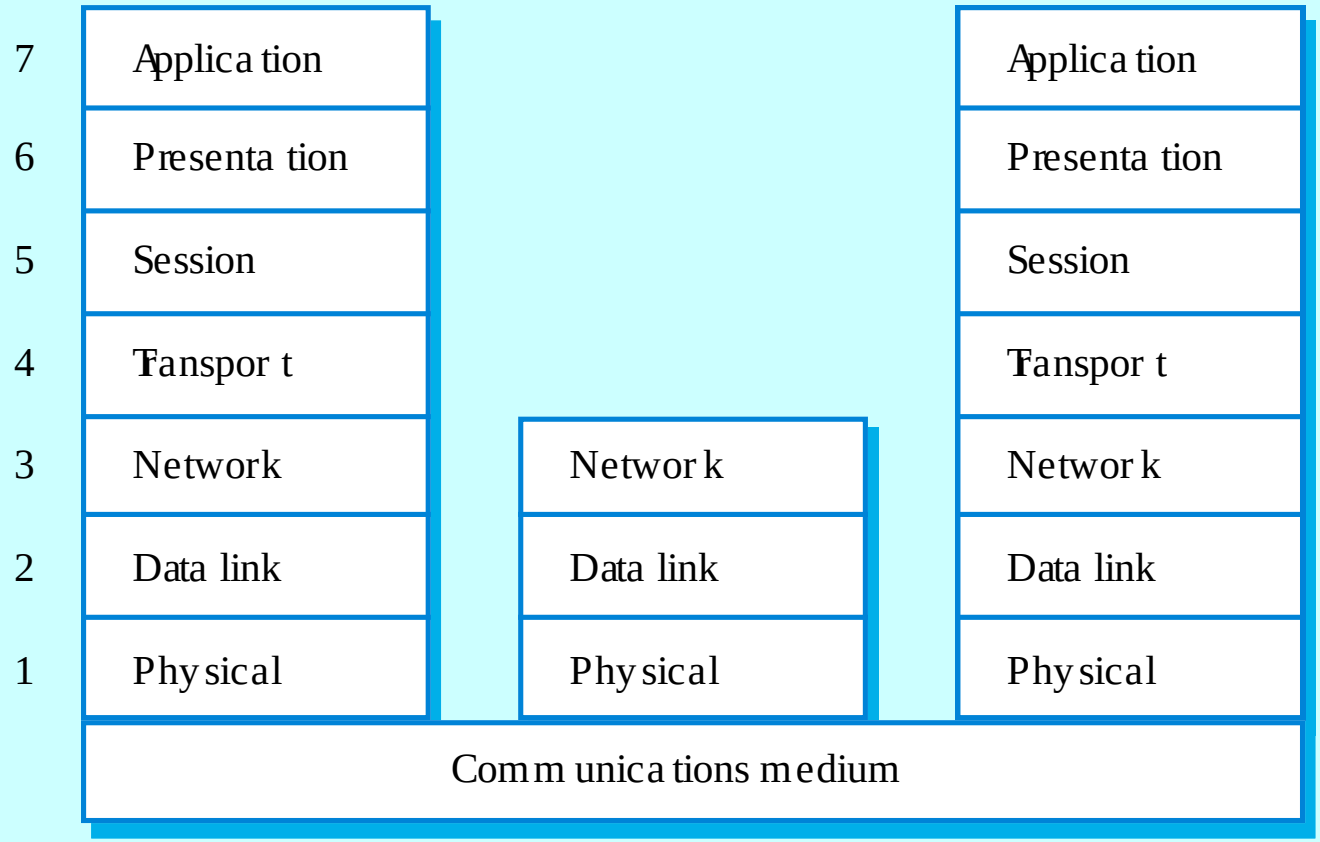

Client
HTTP interaction
Client
Client
Client
Web server
Account service provision
SQL query
Database server
SQL
Customer account database

C/S 아키텍처의 이용

아키텍처	응용 시스템
얇은 클라이언트를 가진 2 단 C/S 아키텍처	별도의 응용처리 및 데이터 관리 기능이 비실용적인 레거시 시스템. 데이터 관리 기능이 거의 혹은 전혀 없는 컴파일러와 같은 계산이 집중되는 응용 시스템. 응용처리가 거의 혹은 전혀 없는(브라우징과 질의 기능) 데이터 중심 응용 시스템
두꺼운 클라이언트를 가진 2 단 C/S 아키텍처	응용처리가 클라이언트상의 기성품 소프트웨어(예: MS 엑셀에 의해 제공되는 응용 시스템). 데이터의 집중적인 계산이 요구되는(예: 데이터 가시화) 응용 시스템. 잘 설정된 시스템 관리 환경에서 사용되는 안정적인 최종 사용자 기능을 갖는 응용 시스템.
3 단 혹은 다단 C/S 아키텍처	수백 혹은 수천 개의 클라이언트를 갖는 대규모 응용 시스템. 데이터와 응용처리가 모두 가변적인 응용 시스템. 여러 소스로부터 나온 데이터가 통합된 응용 시스템

가상 머신 시스템 – 계층 모델

OSI 참조 모델 – 계층 모델



계층 모델의 장단점

❖ 장점

① 시스템의 점증적인 개발을 지원한다. 한 계층이 개발됨에 따라, 그 계층에 의해 제공되는 서비스들의 일부를 사용자가 이용할 수 있다.

② 변경 가능하며 이식 가능하다. 계층의 인터페이스가 변경되지 않는 한, 동등한 다른 계층에 의해 대체될 수 있다.

❖ 단점

① 시스템을 구조화하는 것이 어려울 수 있다. 최상위 수준의 사용자가 요구하는 서비스는 그 아래의 여러 계층이 제공하는 서비스에 접근하기 위해 인접 계층을 '뚫고 지나가야' 할 수도 있다. 이런 방식은 시스템의 외부 계층이 바로 그 이전의 계층에 종속되지 않음에 따라 계층 모델을 망가뜨리게 된다.

②여러 수준의 명령어 해석이 필요하기 때문에 성능이 문제가 될 수 있다. 최상위 계층으로부터의 서비스 요청은 처리되기 전에 상이한 계층에서 여러 번 해석되어야 한다.

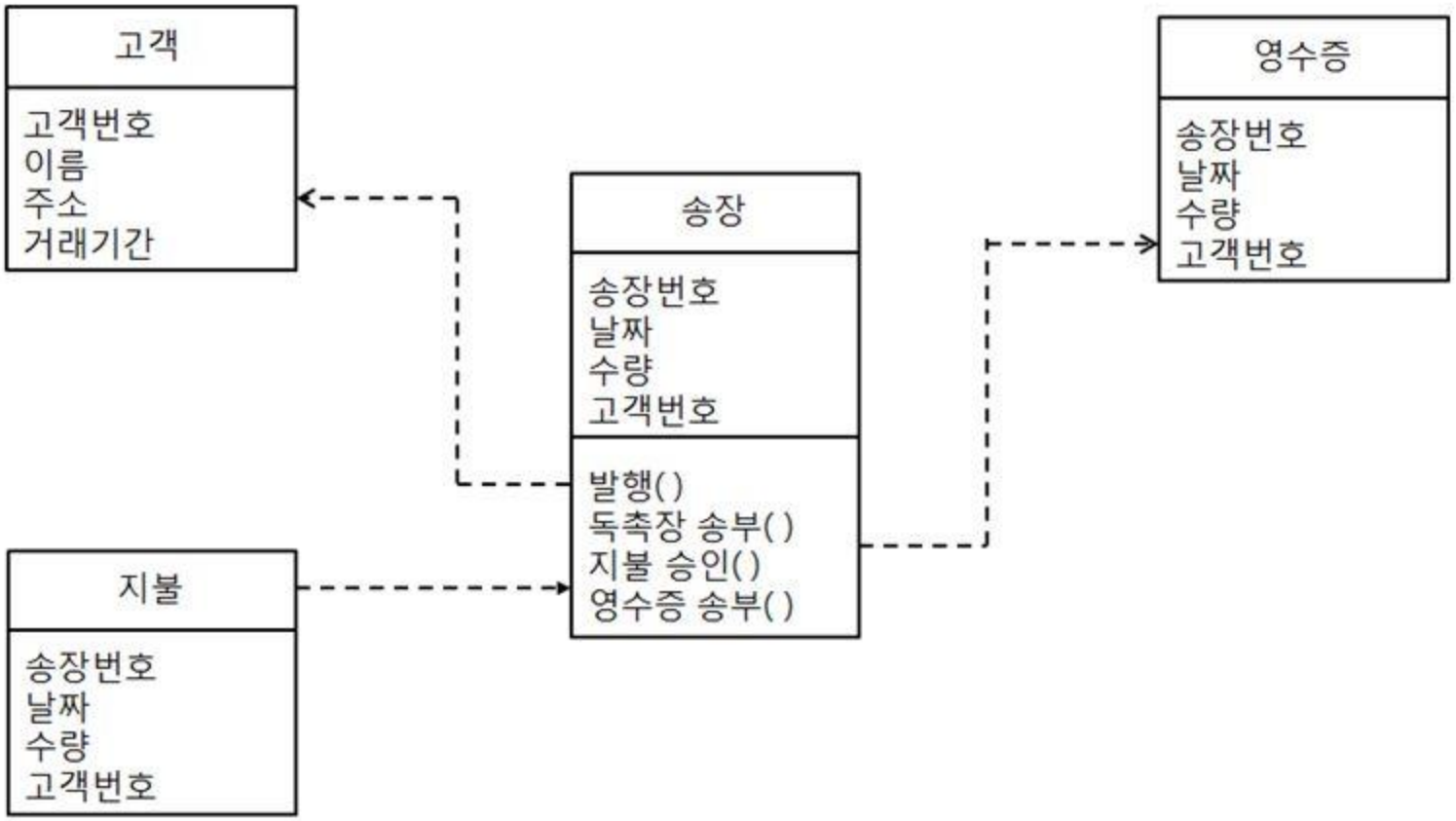
모듈 분해 스타일

- 서브시스템을 모듈로 분해하는 스타일
- 시스템 구조와 모듈 분해 사이에 엄격한 구분이 없음

서브시스템과 모듈

- 서브시스템 자신의 오퍼레이션은 다른 서브시스템에 의해 제공되는 서비스에 독립적임
- 모듈은 다른 컴포넌트에 서비스를 제공하는 컴포넌트이지만, 보통 별개의 시스템으로 간주되지 않음

송장 처리 시스템 – 객체 중심 분해



객체지향 모델의 장단점

❖ 장점

① 객체들의 결합도는 비교적 낮기 때문에 특정 객체를 수정할 때 변경의 파급 효과는 최소화된다.

② 객체는 대개 실세계 개체의 자연스러운 표현이므로 시스템의 구조는 쉽게 이해 가능하다.

③ 실세계 개체들이 상이한 시스템에서도 반복해서 나타날 수 있으므로 그 개체에 해당하는 객체는 다른 시스템에서도 재사용될 수 있다.

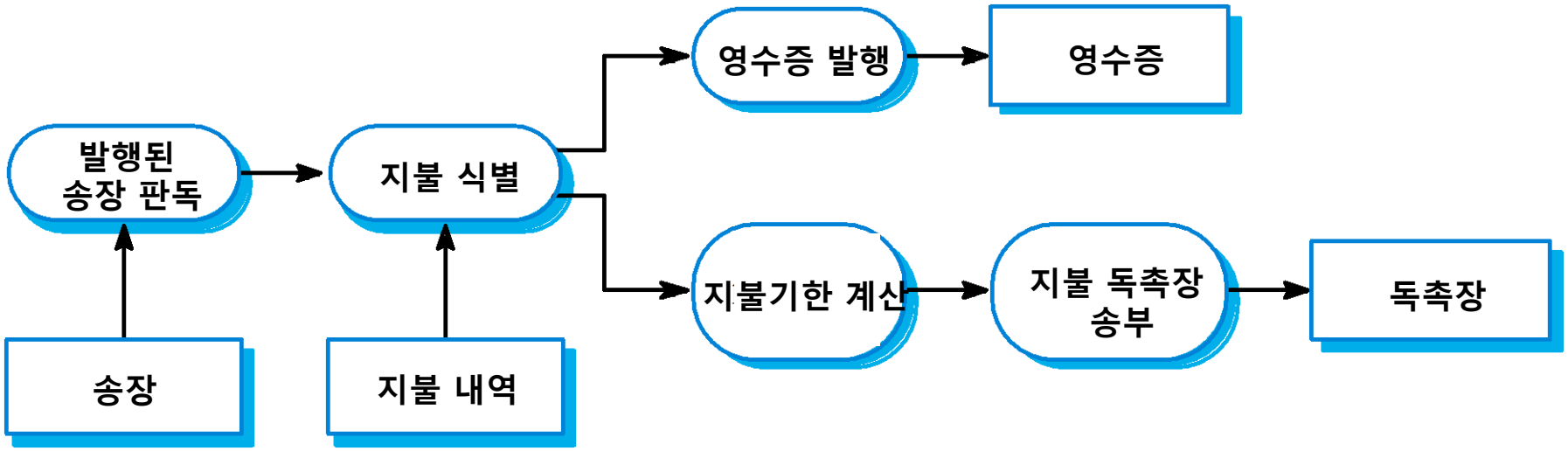
❖ 단점

① 서비스를 사용하기 위해 객체는 다른 객체의 이름과 연산을 명시적으로 참조해야 한다.

② 시스템을 변경하기 위해 인터페이스를 변경해야 한다면, 해당 인터페이스와 연관된 객체의 모든 사용자를 대상으로 변경의 효과를 평가해야 한다.

③ 일반적으로 객체는 소규모의 개체와는 명백하게 대응되는 반면에, 복잡한 개체를 객체로 표현하는 것은 어려운 경우가 많다.

송장 처리 시스템 – 기능 중심 분해



기능 중심 분해 모델의 장단점

❖ 장점

① 변환의 재사용을 지원한다.

② 많은 사람들이 자신들의 작업을 입력과 출력 처리 관점에서 생각한다는 점에서 직관적이다.

③ 새로운 변환을 첨가하여 시스템을 진화시키는 것이 쉽다.

④ 병행 시스템이나 순차 시스템으로 구현하는 것이 간단하다.

❖ 단점

① 모든 변환에 적용될 수 있는 공통적인 데이터 이동 양식이 필요하다.

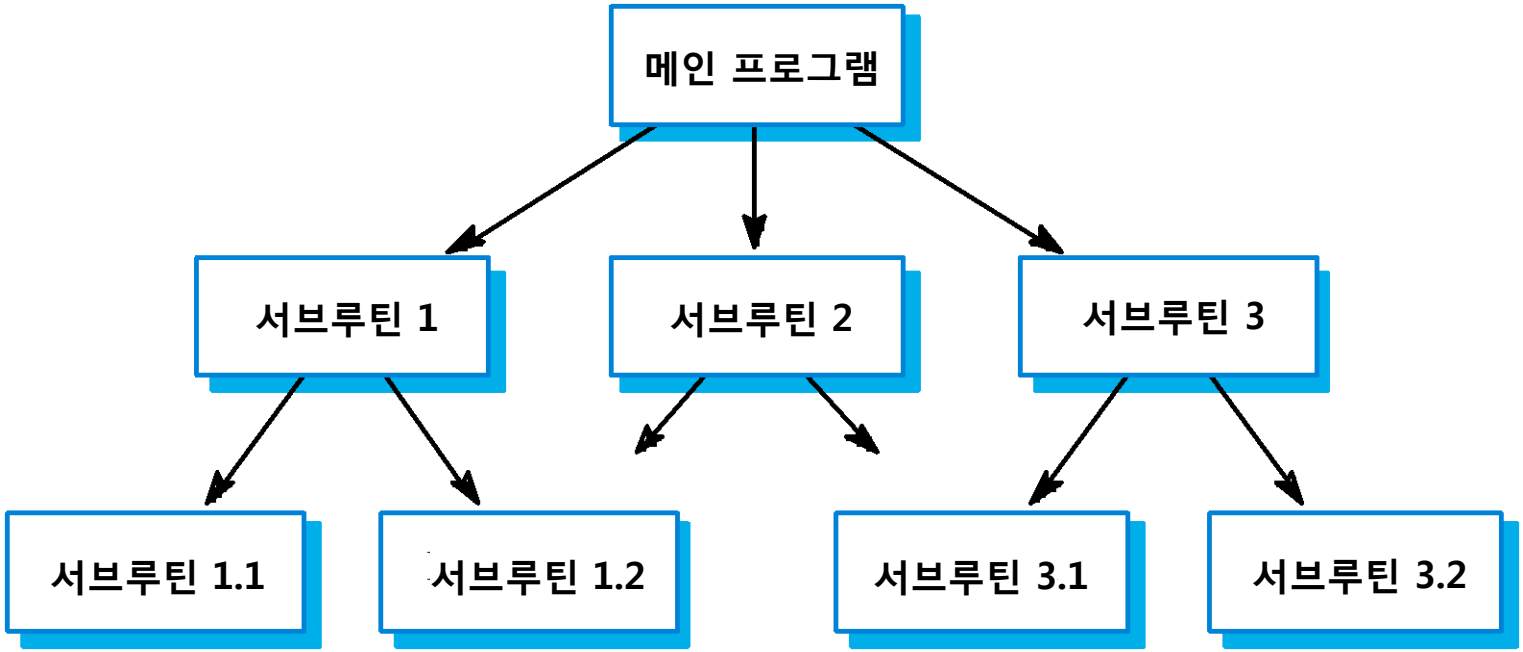
② 대화식 시스템은 처리될 데이터의 스트림이 필요하기 때문에 파이프라인 모델을 사용하여 작성하기 어렵다.

제어 스타일

- 서브시스템 사이의 제어 흐름에 관련됨. 시스템 분해 모델과는 상이함
- 중앙 집중식 제어
  - 하나의 서브시스템이 제어에 관한 전체적인 책임을 지고 다른 서브시스템을 시작 및 종료시킴
- 이벤트 기반 제어
  - 각 서브시스템은 다른 서브시스템이나 시스템 환경으로부터 외부에서 생성되는 이벤트에 반응할 수 있음

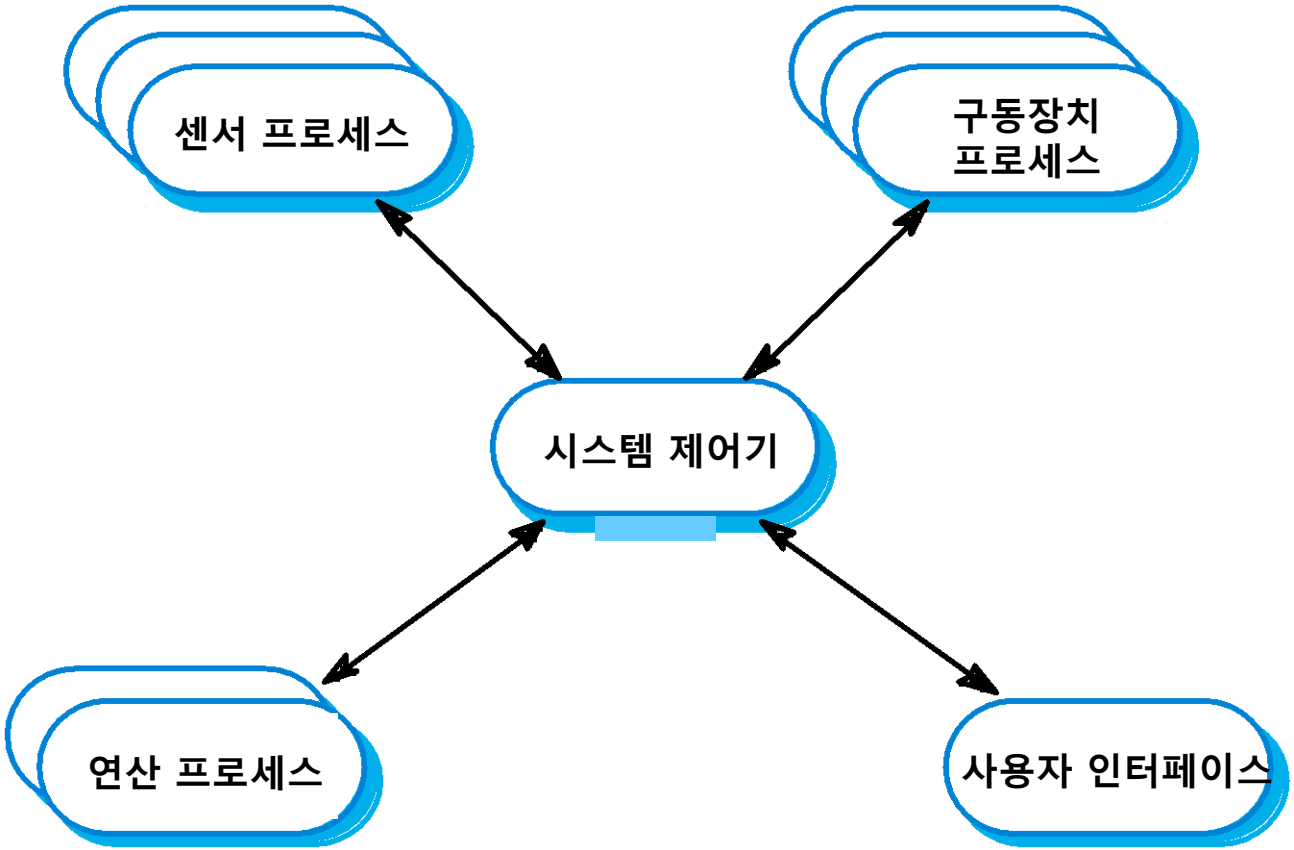
중앙 집중식 제어 - 호출 반환 모델

제어가 서브루틴 계층의 맨 위에서 시작하여 서브루틴 호출을 통해서 트리의 하위 수준으로 전달되는, 하향식 서브루틴 모델이다. 이 서브루틴 모델은 순차적인 시스템에만 적용될 수 있다.



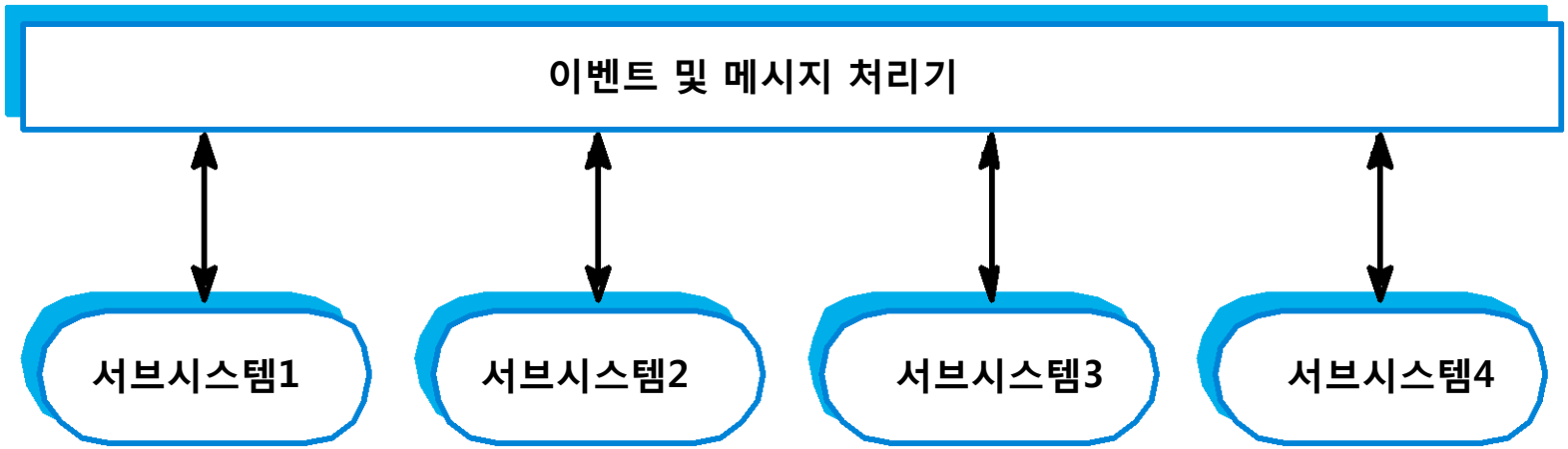
중앙 집중식 제어 - 실시간 시스템 제어

이 모델은 병렬 시스템에 적용될 수 있다. 한 시스템 구성요소가 시스템 관리자로 지정되어, 다른 시스템 프로세서를 시작 혹은 중단시키거나 조정할 수 있다. 한 프로세스는 다른 프로세스와 병렬로 실행될 수 있는 서브시스템이나 모듈이다.



이벤트 기반 시스템 – 브로드캐스트 모델

중앙 집중식 제어 모델에서 제어의 결정은 몇몇 시스템 상태 변수의 변화에 따라 정해진다. 이에 반해 이벤트 기반 제어 모델에서는 외부에서 생성된 이벤트에 의해 결정된다.



이벤트는 모든 서브시스템에 전파된다. 그 이벤트를 처리하도록 프로그램이 된 어떤 서브시스템이 그 이벤트에 응답한다.

브로드캐스트 모델의 장단점

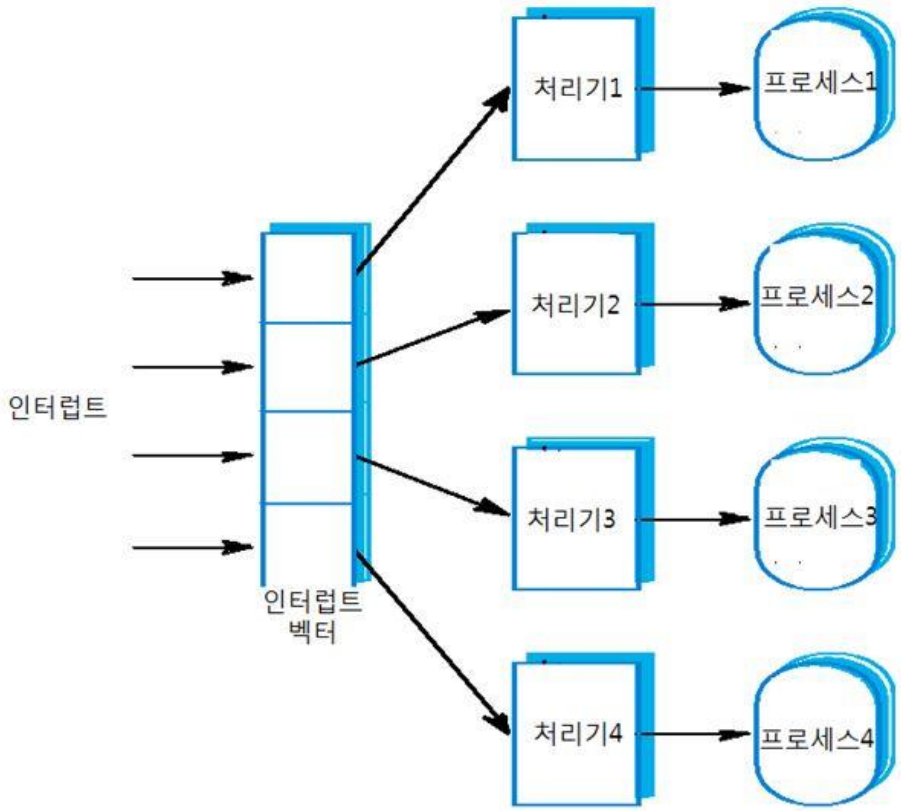
- 장점

  ① 진화가 비교적 단순하다. 특정 부류의 이벤트를 처리하는 새로운 서브시스템은 이벤트 처리기에 이벤트를 등록함으로써 통합될 수 있다. 모든 서브시스템은 다른 서브시스템의 이름이나 위치를 모르고서도 실행시킬 수 있다. 서브시스템들은 분산 환경에서 구현될 수 있다. 이러한 분산은 다른 서브시스템에 대해 투명성을 지니고 있다.

- 단점

  ① 서브시스템이 이벤트가 처리될지의 여부, 혹은 언제 처리될지를 알지 못한다. 한 서브시스템이 이벤트를 생성할 때, 다른 서브시스템은 그 이벤트를 등록했는지 알지 못한다. 상이한 서브시스템들이 동일한 이벤트를 등록하는 것이 가능하다. 이것은 이벤트 처리 결과를 이용할 때 모순을 유발할 수도 있다.

이벤트 기반 시스템 – 인터럽트 기반 모델



이 모델은 실시간 시스템에서만 사용되는데, 여기에서는 외부의 인터럽트가 인터럽트 처리기에 의해 감지된다. 그 다음 인터럽트 처리를 위한 다른 구성요소로 전달된다.

인터럽트 기반 모델의 장단점

❖ 장점

① 구현할 이벤트에 대해 매우 빠른 응답을 허용한다.

❖ 단점

① 프로그래밍 하기가 복잡하고 검증이 어렵다. 시스템 테스트 동안, 인터럽트 타이밍의 패턴을 되풀이하는 것은 불가능할 수 있다. 만일 하드웨어 문제로 인터럽트의 수가 제한이 되면, 이 모델을 사용하여 개발된 시스템을 변경하는 것은 어려울 수 있다. 일단 이러한 제한에 도달하면, 어떠한 유형의 이벤트도 처리될 수 없다. 이러한 제한은 몇 가지 유형의 이벤트를 하나의 단일 인터럽트로 대응시켜서 이벤트가 발생했을 때 처리기가 처리하도록 함으로써 해결할 수 있다. 그러나 만일 개개의 인터럽트에 대해 매우 빠른 응답이 필요한 경우, 인터럽트의 대응은 비실용적일 수 있다.

모듈 분할의 평가 기준

- 어떤 모듈이 한 가지 목적을 위한 기능만을 제공하고 있고, 다른 모듈과의 상호작용이 적은 경우, 해당 모듈은 "**기능적 독립성이 높다**". 기능적 독립성이 높은 모듈들로 구성된 소프트웨어는 각 모듈들의 기능이 명확하게 분리되어 있고 인터페이스도 분명하게 되어 있으므로 개발이 용이하다. 또한 기능적 독립성이 높은 모듈은 설계의 변경과 프로그램의 수정에 의한 부작용이나 파급효과가 국소화되므로 테스트와 유지보수가 수월하다.

캡슐화

- Encapsulation
  - 서로 관련성이 많은 데이터들과 이와 관련된 함수들을 한 묶음으로 처리하는 것으로서, 캡슐화된 모듈의 세부 내용이 외부에 은폐되어 변경이 발생해도 오류의 파급효과를 최소화시키는 것이다. 캡슐을 구성하는 내부의 입자들의 구체적인 내용은 몰라도 캡슐 자체의 이름과 기능으로 사용하는 예를 따라서 캡슐로 둘러싼다는 의미에서 캡슐화라고 부른다.

정보 은닉

❖ Information Hiding

- 각 모듈들은 다른 모듈에게 인터페이스를 통해 서비스를 제공한다. 인터페이스는 모듈의 외부 명세이며, 해당 모듈의 서비스를 이용하기 위해 알고 있어야 하는 정보이다. David L. Parnas가 제안한 "**정보 은닉**"의 개념은 인터페이스와 구현을 명확하게 분리하여 모듈의 구현에 대한 상세한 설계 정보는 다른 모듈로부터 은폐되어 참조할 수 없도록 해야 한다는 것이다.
- **캡슐화**는 정보 은닉을 통해 실현되었으며, 모듈의 프로시져와 모듈 내부의 데이터에 대한 조작에 관한 제약 조건을 규정한 것이다. 모듈 내부에 정의된 변수와 하위 모듈은 외부로부터 은폐되어 직접 액세스할 수 없다. 다른 모듈로부터 액세스하려면 인터페이스로서 정의된 프로시져와 함수를 통해야 한다.

❖ 장점

① 모듈의 구현 내용을 살펴보지 않아도 모듈의 인터페이스를 이해하고 있으면 모듈을 이용할 수 있다.

② 모듈의 구현을 변경하더라도 인터페이스를 변경하지 않는다면, 해당 모듈을 호출하고 있는 모듈은 아무런 영향을 받지 않고 그대로 사용할 수 있다. 이것은 프로그램의 변경이 수월하게 되며 유지보수성을 향상시킬 수 있다.

응집도와 결합도

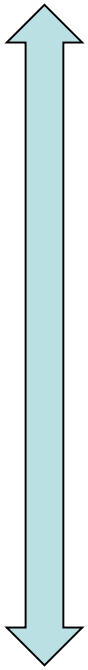
- 정보 은닉을 위해서는 모듈의 응집도를 높게 하고, 결합도는 낮게 하는 것이 좋다. 소프트웨어 설계의 관점에서도 이러한 원칙을 준수하는 것이 바람직하다.

응집도(Cohesion)

❖ 한 모듈 내부에 있는 구성 요소들 사이의 기능적 관련성을 평가하는 기준이다. 한 모듈 내부의 구성 요소 사이의 기능적인 관련 정도 및 모듈 내부의 응집도를 최대화하는 방향을 설계하는 것이 바람직하다.

Bad

약



강

Good

① **우연적 응집도(Coincidental cohesion):** 명확한 이유도 없이 구성 요소들을 모아서 독립적인 기능을 구성한 모듈. 예를 들어, 우연히 같은 문자로 시작되는 함수들을 모아서 작성한 모듈이나 에디터를 사용해서 기계적으로 동일한 크기로 분할해서 작성한 모듈 등이 여기에 해당한다.

② **논리적 응집도(Logical cohesion):** 유사한 성격을 같거나 특정 유형으로 분류되는 구성 요소들로 모듈을 형성하여, 겉모습으로는 하나의 기능을 가지지만 실제로는 다양한 기능을 포함하는 모듈. 예를 들어, 정수의 덧셈과 행렬의 덧셈을 결합한 덧셈 모듈, 단말기 출력 기능과 파일 출력 기능을 결합한 출력 모듈 등이 있다.

③ **시간적 응집도(Temporal cohesion):** 기능적으로는 유사성이 없으나 실행하는 시간이 서로 가깝게 연계되어 있어서 한 군데 모아서 만든 모듈. 즉, 특정 시간에 처리되는 몇 개의 기능을 모아서 하나의 모듈로 작성한 것이다. 예를 들어, 초기값 설정 모듈 등이 여기에 해당한다.

④ **절차적 응집도(Procedural cohesion):** 순서적으로 선후관계에 있는 기능들을 한 군데 모아서 만든 모듈. 반복문이나 선택문에 의해 한 덩어리가 된 기능들을 모아서 만든 모듈 등이 여기에 해당한다.

⑤ **통신적 응집도(Communicational cohesion):** 동일한 데이터에 대해 액세스하는 여러 개의 기능들을 한 군데 모아서 만든 모듈. 명단 데이터에 대한 등록, 변경, 삭제, 검색 등의 처리를 모아서 만든 모듈 등이 여기에 해당한다.

⑥ **순차적 응집도(Sequential cohesion):** 어떤 기능의 출력이 다른 기능의 입력으로 되어 있고, 다시 그 기능의 출력이 또 다른 기능의 입력으로 되는 관계에 있는 모듈 등이 여기에 해당한다.

⑦ **기능적 응집도(Functional cohesion):** 모듈 내부의 모든 구성 요소들이 단일 기능을 수행하기 위해 반드시 필요한 경우가 여기에 해당한다.

결합도(coupling)

Chang and Eng - The Original Siamese Twins

Chang and Eng - The Original Siamese Twins

Chang Eng were conjoined twins born in Siam (now Thailand) to a Chinese family. It is because of their fame that the phrase 'Siamese twins' has sometimes been used to describe conjoined twins.

When the boys were 17, they joined the Scottish merchant Robert Hunter, who managed a tour in which they displayed themselves, and were examined by doctors across the United States and Europe. In 1832 they realised that their new manager, Captain Coffin, was taking most of their profits, and made a separate arrangement with the circus owner Phineas T Barnum, with whom they toured until 1839.

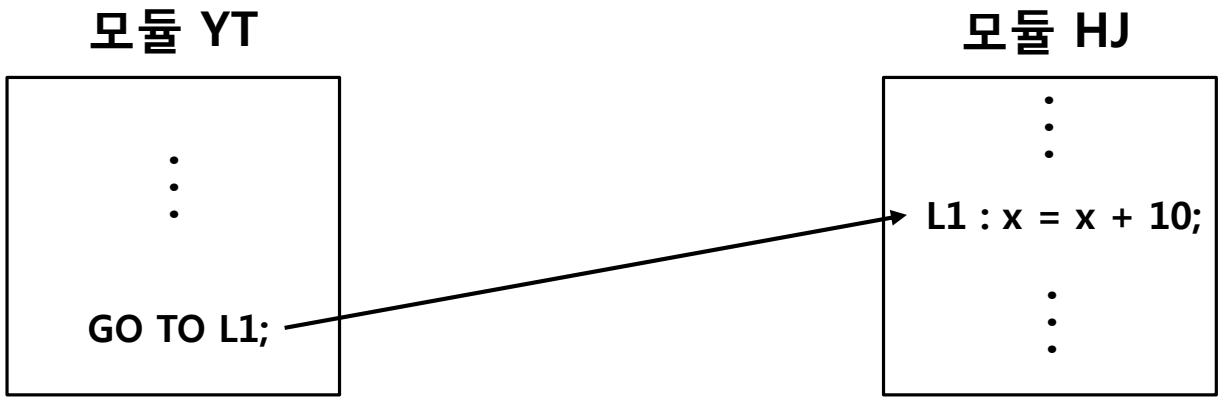
In that same year they became American citizens and settled in North Carolina. They took the name Bunker, and in 1843 married two sisters, Adelaide and Sarah Ann Yates. During their marriages, Eng had 11 children and Chang 10. They ran two households, and became wealthy plantation owners, known for their brutality towards their enslaved workers.

After the Civil War, in which sons of both Chang and Eng served in the Confederate army, the Bunkers lost much of their fortune. They returned to displaying themselves in order to support their families. In 1870 Chang became paralysed from a stroke, which required Eng to support him physically. In January 1874 Chang Bunker died after a severe case of bronchitis, possibly from a cerebral clot. Eng died shortly thereafter. Many of their numerous descendants continue to live in the region.

Bad 강 

내용 결합도(Content coupling)

- 어떤 모듈이 인터페이스를 통하지 않고 다른 모듈이 관리하는 데이터 또는 제어 구조를 직접 이용하여 데이터를 직접 참조, 변경하는 경우의 결합도. 그림과 같이 GO TO 문 등에 의해 다른 모듈의 내부로 제어가 직접 전달되는 경우가 내용 결합도에 해당한다.

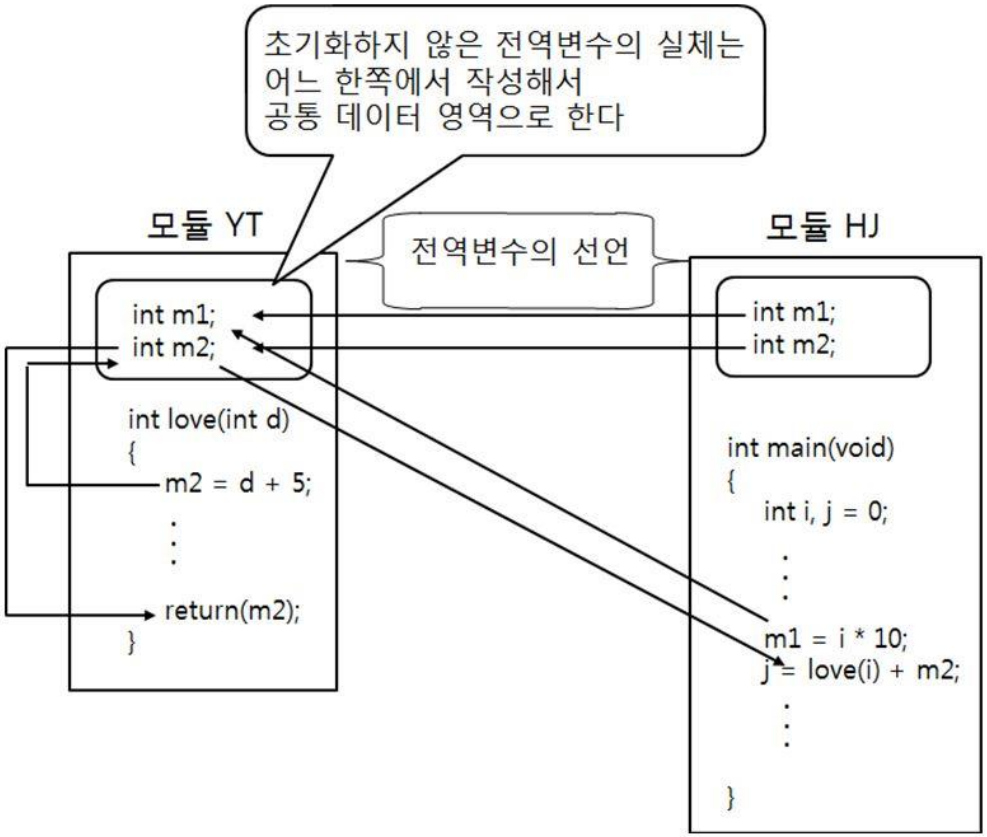


공통 결합도(Common coupling) (1/2)

- 공통 데이터 영역에 정의한 데이터를 통해서 정보 교환을 하는 수행하는 모듈들 사이의 결합. 공통 데이터 영역의 데이터 구조 등을 변경한 경우에 이를 참조하고 있는 모든 모듈이 영향을 받는다. 또한 모듈 내부의 데이터를 참조하는 명령과 공통 영역의 데이터를 참조하는 명령은 구별되지 않으므로, 어떤 모듈이 공통 데이터 영역의 데이터를 참조하고 있는지 명시적으로 기술되지 않는다. 따라서 프로그램의 판독을 곤란하게 하고, 공통 데이터 영역을 변경했을 경우에 영향을 받는 모듈을 간과하기 쉽다는 문제점이 있다.

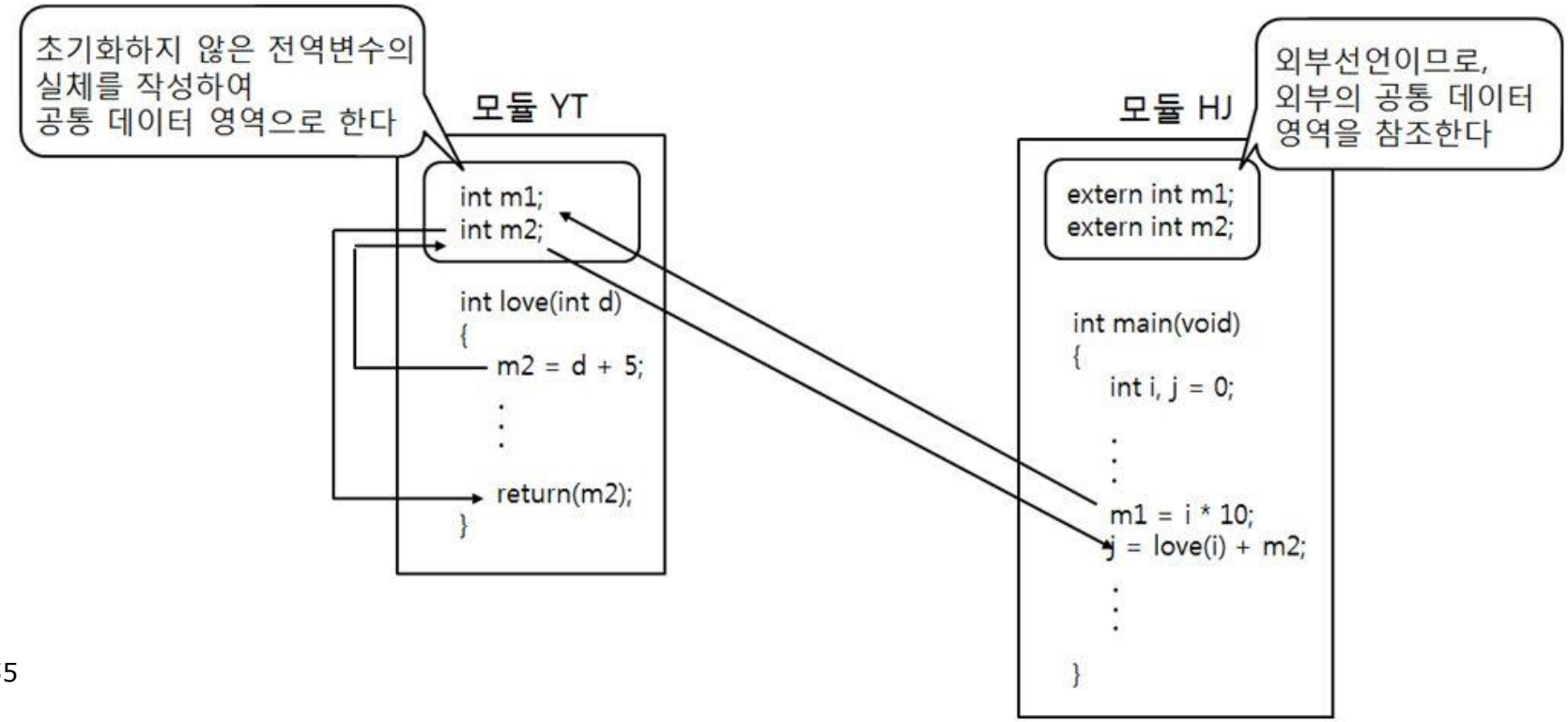
공통 결합도(Common coupling) (2/2)

- C 프로그램에서 동일 이름의 전역 변수를 두 개의 모듈에서 초기화하지 않고 선언한 경우 혹은 어느 한쪽 모듈에서만 초기화하여 선언한 경우에는 전역 변수의 실체는 한 개만 생성되어 두 모듈에서 공유하는 경우가 공통 결합도에 해당한다.



외부 결합도(External coupling)

- 외부 변수로 선언한 데이터를 공유하는 모듈들 사이의 결합도. 공통 결합도와 유사하지만, 공유하는 데이터를 외부 선언하므로 공통 결합도에서 지적되었던 문제점은 발생하지 않는다. 필요한 데이터만을 외부 선언하므로 불필요한 데이터까지 공유하는 것이 적어지게 되어 결합도가 낮아진다.

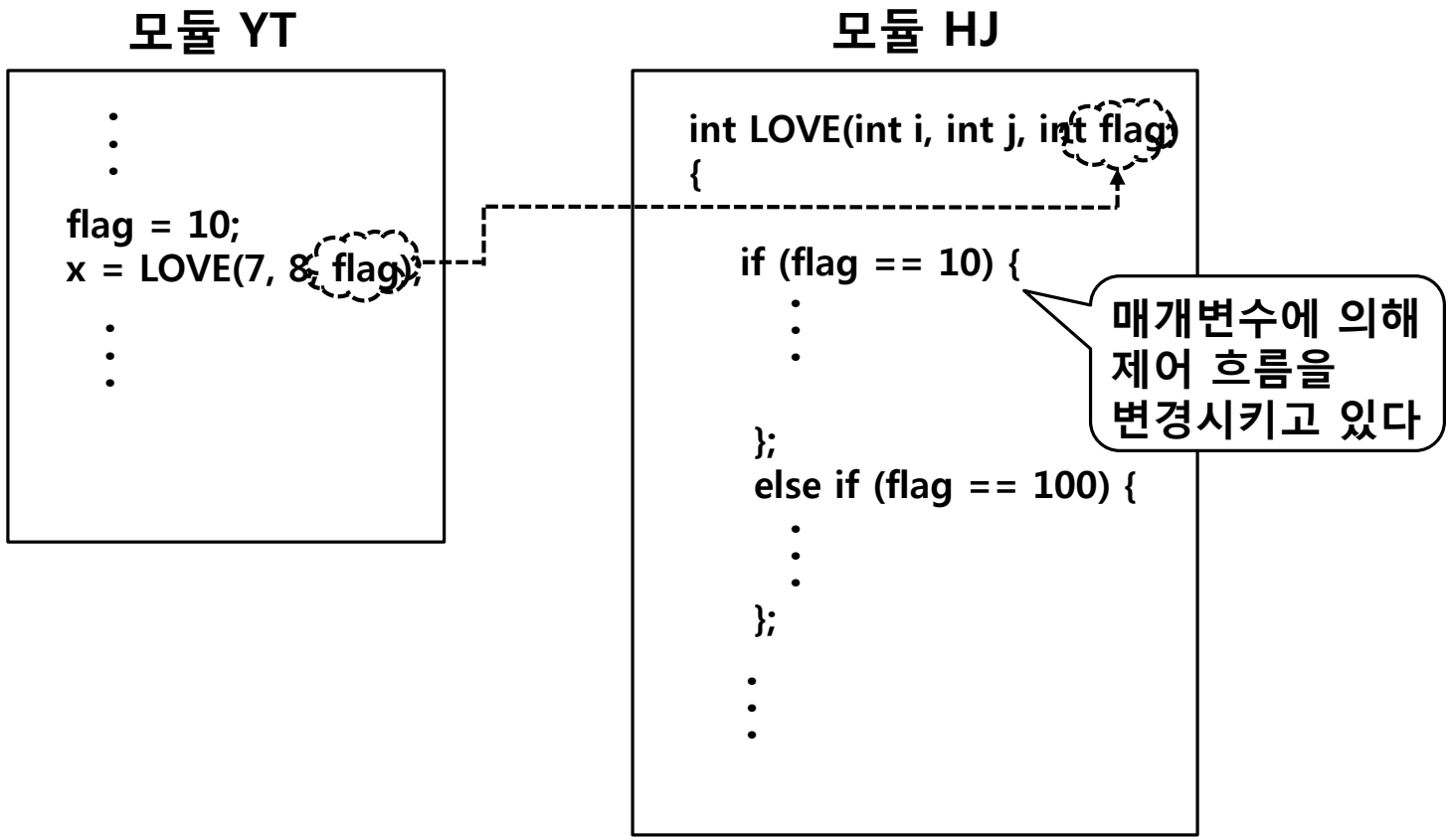


제어 결합도(Control coupling) (1/2)

- 어떤 모듈이 다른 모듈을 호출하는 경우, 호출되는 모듈의 제어를 지시하는 데이터를 매개변수로 전달하는 모듈들 사이의 결합도.

- 다음 슬라이드의 그림에서 모듈 YT는 모듈 HJ의 함수 LOVE를 호출하고 있다. 이때 함수 LOVE에서는 세 번째 매개변수 flag의 값을 분기 조건으로 사용하여 제어 흐름을 변경한다. 따라서 모듈 YT에서는 함수 LOVE의 내부에서의 제어 방식을 고려하여 매개변수 flag의 값을 정해야 한다. 이와 같이 호출하는 모듈은 호출되는 모듈의 내부 논리를 파악하고 있어야 하고, 어떻게 제어하면 좋을지 이해하고 있을 필요가 있다.

제어 결합도(Control coupling) (1/2)

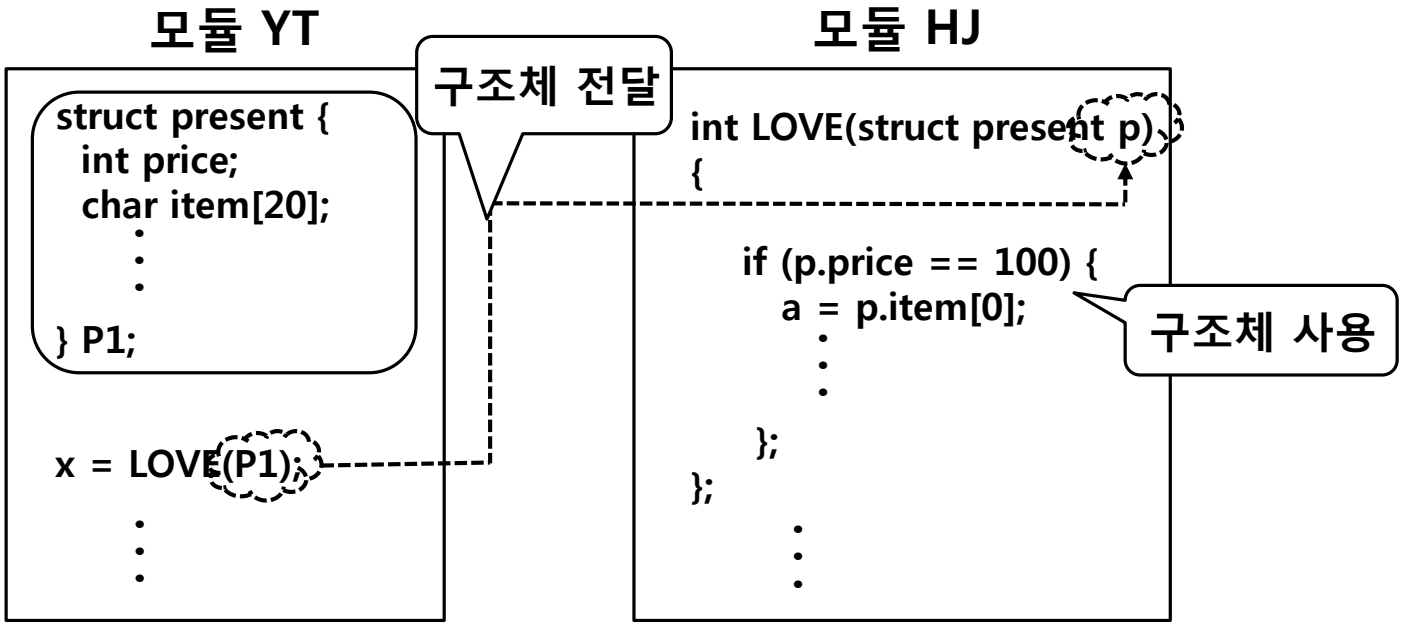


스탬프 결합도(Stamp coupling) (1/2)

- 모듈들 간에 공유 데이터 영역에 없는 데이터로, 배열이나 레코드(구조체) 등의 데이터 구조를 전달하고 받는 모듈들 사이의 결합도.

- 다음 슬라이드 그림의 프로그램에서 모듈 YT가 모듈 HJ의 함수 LOVE를 호출할 경우 present 타입의 구조체를 매개변수로서 전달하고 있다. 이때 함수 LOVE는 구조체의 일부만을 사용하고 있지만, 함수 LOVE를 작성하기 위해서는 present 타입 구조체의 정의를 모두 알고 있어야 한다.

스탬프 결합도(Stamp coupling) (2/2)



약 Good

데이터 결합도(Data coupling) (1/2)

- 모듈들 사이에 단일 변수를 매개변수로 전달하는 경우 혹은 데이터의 필요한 부분만을 매개변수로 전달하는 방법으로만 정보 교환을 하는 모듈들 사이의 결합도. 만일 단일 변수를 매개변수로 전달하는 경우는 데이터 결합도가 되고, 결합도는 가장 낮다. 또는 호출된 모듈이 구조체 데이터의 일부 요소만을 사용하는 경우도 데이터 결합도에 해당한다.
- 다음 슬라이드의 그림은 이전 슬라이드의 그림 대신 함수 LOVE에서 사용하는 정수값과 문자만을 전달하도록 변경한 프로그램이다. 이와 같이 호출된 모듈이 구조체 데이터의 일부 요소만을 사용하는 경우에 그 요소를 기본형 매개변수로 하여 전달하도록 하면, 호출되는 모듈은 호출하는 모듈의 데이터구조에 관한 전체적인 지식을 필요로 하지 않기 때문에 결합도가 낮아진다.

데이터 결합도(Data coupling) (2/2)