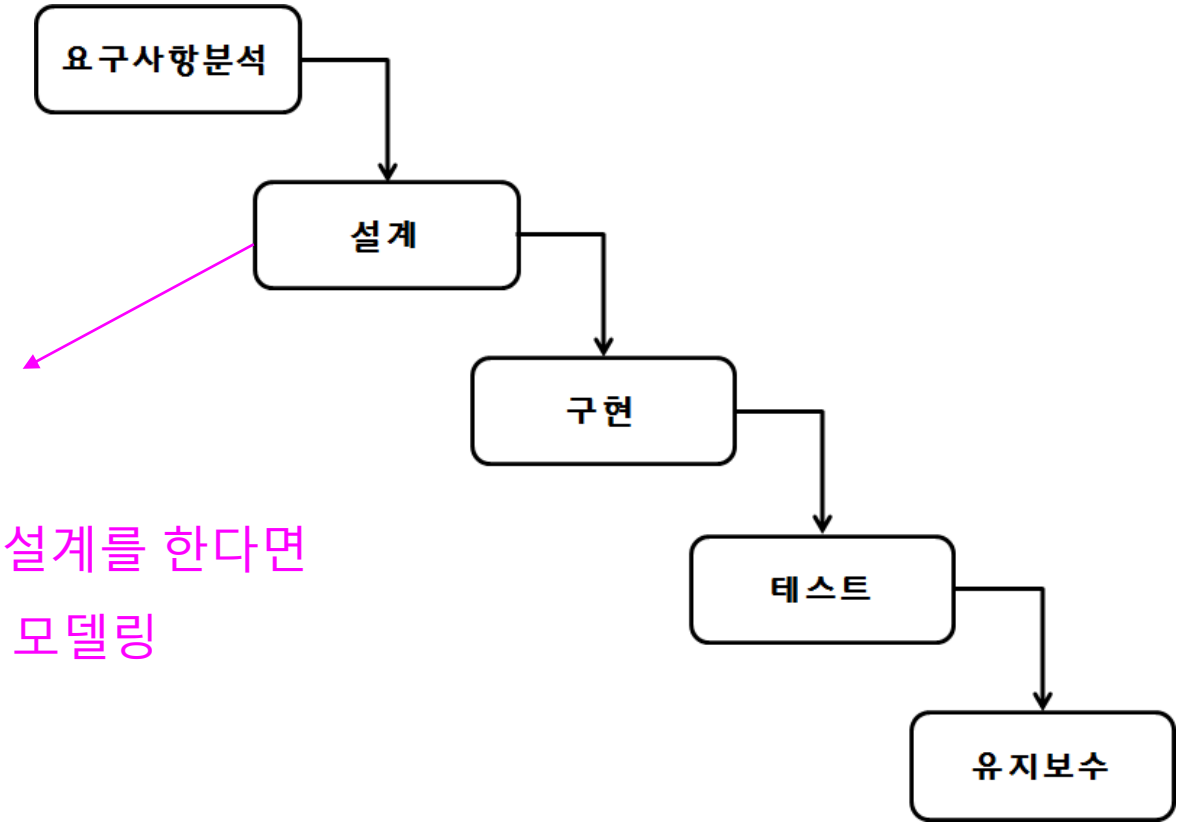
소프트웨어공학

설계 이론

소프트웨어 개발 프로세스



아키텍처 설계(상위 설계)

상세 설계(하위 설계)

객체 지향 방법론에 따라 설계를 한다면

클래스, 클래스 간의 관계 모델링

분석과 설계

- 요구사항 분석 (사용자 포함 영역)

  - 사용자의 요구가 무엇(what)인지 깊이 이해하고 파악하여 정의하는 활동(기능 요구 파악에 중점)

- 설계 (개발자 영역)

  - 위에서 정의된 요구 사항을 어떻게(how) 만족시킬 것인가에 대한 해결 방안(솔루션)을 찾고, 이를 구체적으로 정의하는 활동
  - 구조적인 관점과 함께 낮은 수준의 컴포넌트와 알고리즘에 대한 구현 문제 포함

실제 프로그래밍 언어를 이용해서 구현하는 것을 염두에 두고 기술적인 부분에 집중하는 단계

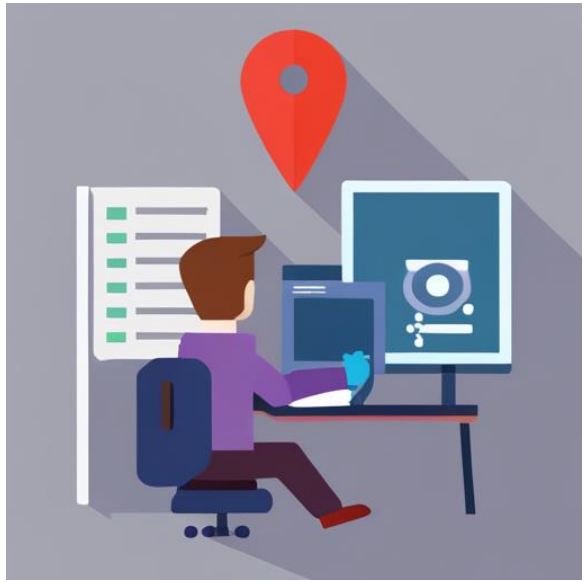
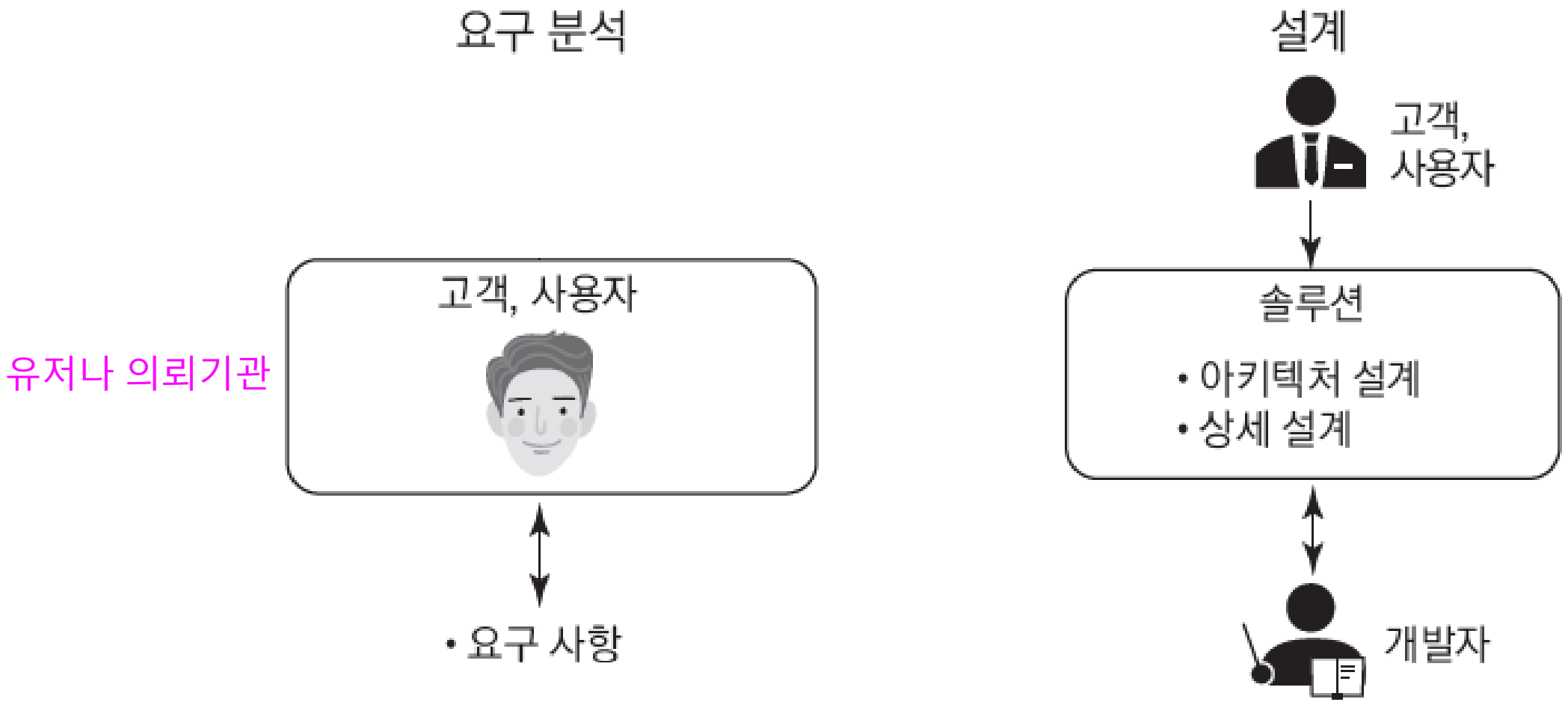


그림 : 뤼튼 wrtn.ai

분석과 설계

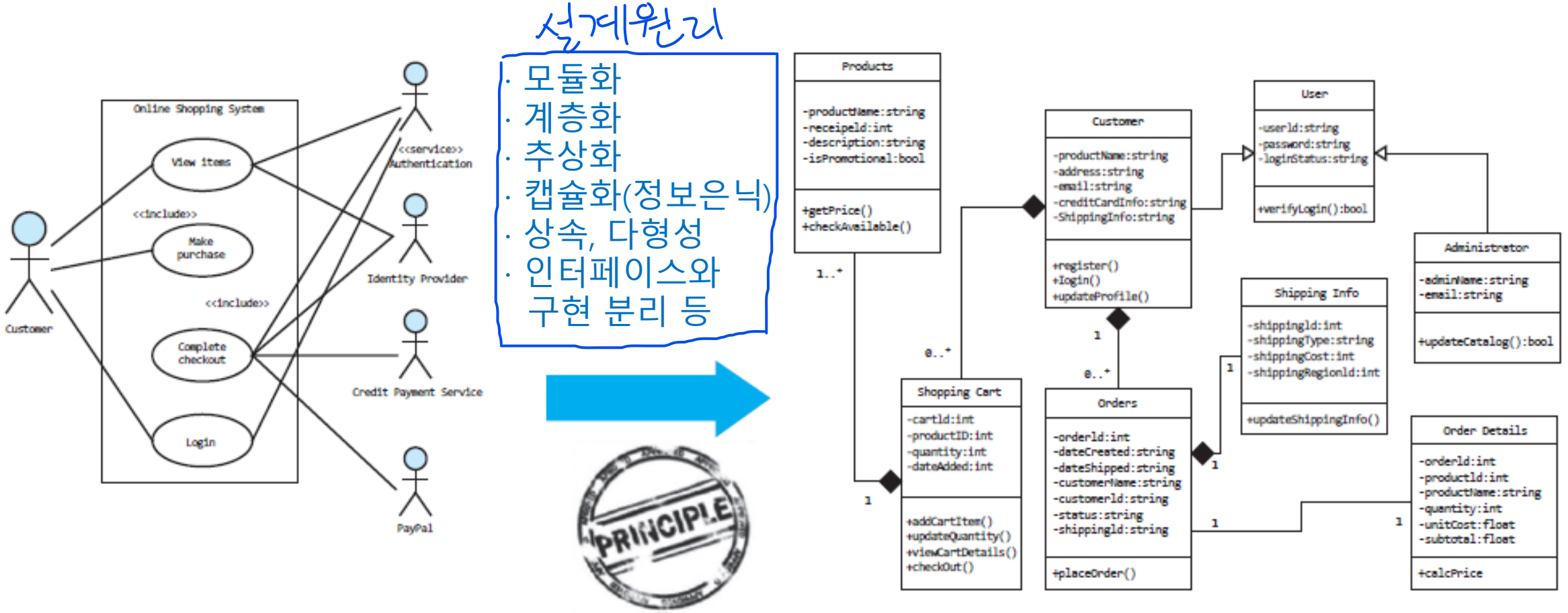
- 설계는 개발자의 창의성이 발휘되는 창조적인 과정 소프트웨어는 기존에 없던 새로운 것을 창조해 내는 것
  - 대규모 시스템의 경우, 품질 좋은 시스템을 쉽게 개발하기 위해 설계 원리와 절차를 따르는 것이 중요



분석과 설계

설계 : 로우 레벨에서의 컴포넌트 및
필요에 따라서는 특정 알고리즘에 대한 응용/구현 문제를 포함해서 작성

- 설계 원리 - 확장성을 높이고, 수정의 영향을 최소화



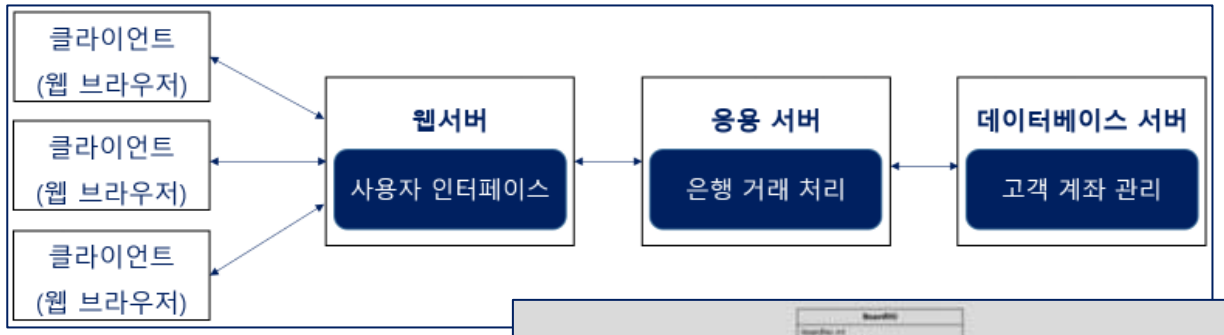
추상화 : 핵심적인 특징이나 요소 추출, 수행과정에서의 세부 단계는 고려하지 않고, 상위 수준에서의 핵심적인 큰 수행 흐름
추상 메소드, 추상 클래스

캡슐화 : 모듈의 자세한 내부정보를 감추는 것으로 정보은닉의 개념을 포함

분석과 설계

- 소프트웨어 설계는 건축 설계와 유사

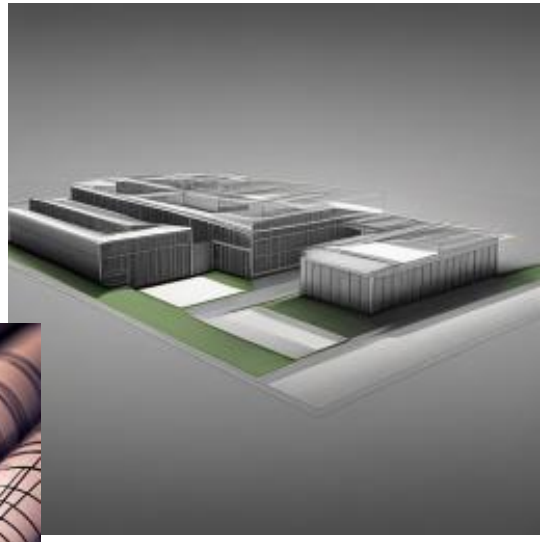
  - 사용자의 요구사항에 따라 명세서가 만들어지면 개발팀은 이 명세서를 참조하여 설계서(건축의 설계 도면)을 작성한 뒤 이를 기반으로 구현

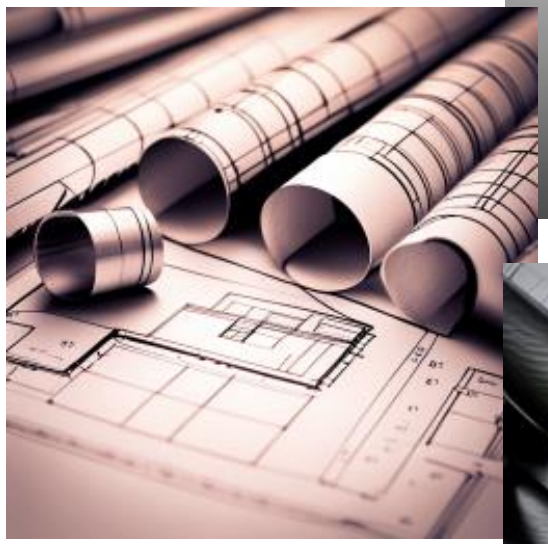


아키텍춰 설계
(상위 설계)

상세 설계
(하위 설계)

상위 설계



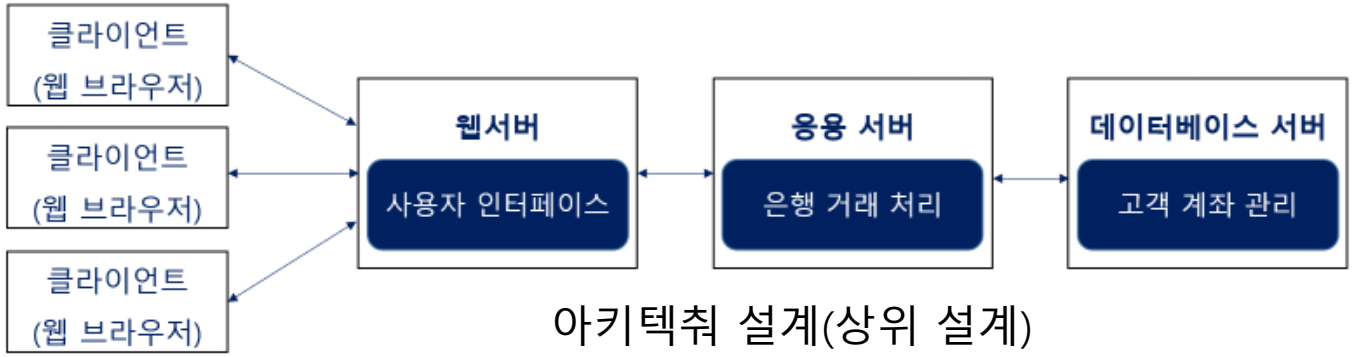


하위 설계

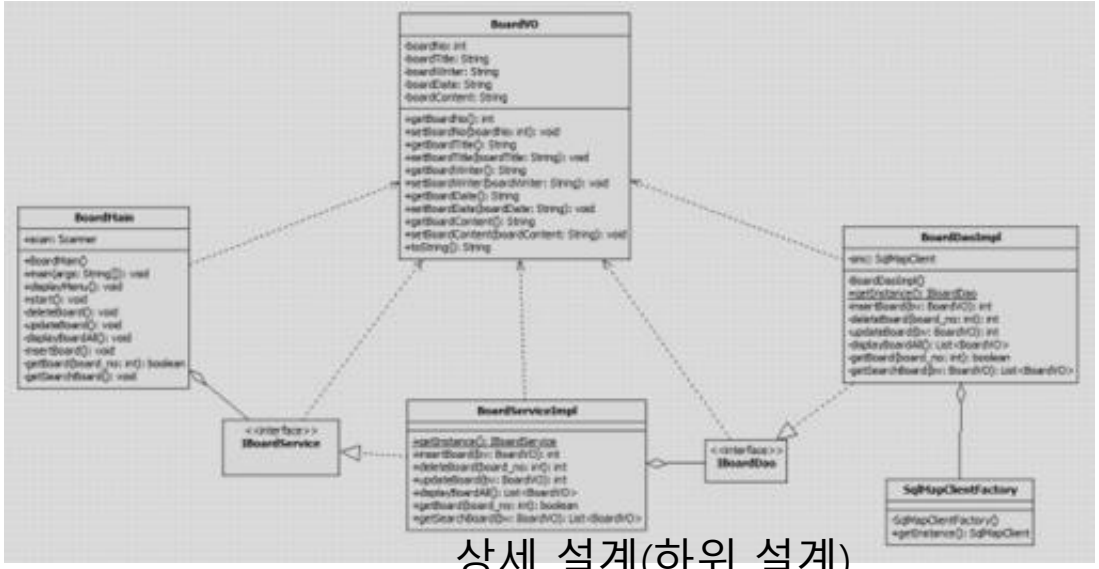
그림 : 뤼튼 wrtn.ai

분석과 설계

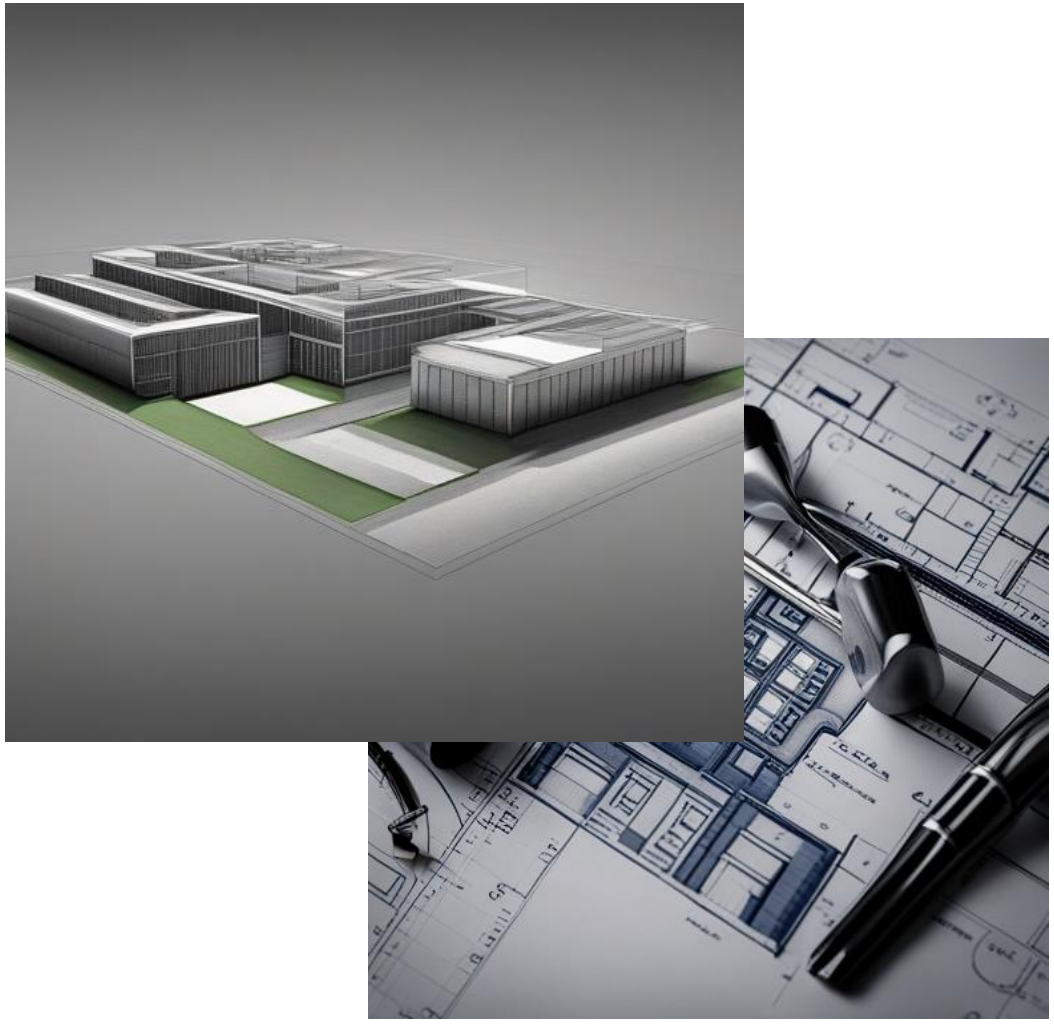
- 이때 설계서는 건축의 다양한 도면처럼 상위 설계, 하위 설계 등 여러가지 형태로 작성된 구체적인 사양서라고 할 수 있음



아키텍춰 설계(상위 설계)



상세 설계(하위 설계)



설계

- 설계

- **아키텍처 설계** : 상위 수준 설계 시스템의 기본 구조 즉, 뼈대에 대한 설계를 의미
  시스템 전체를 넓게 바라보는 단계
  . 시스템 아키텍처 설계, 소프트웨어 아키텍처 설계

- **상세 설계 :** 하위 수준 설계 (컴포넌트 설계, 모듈 설계, 절차 설계 등으로도 불리움)
  . 소프트웨어 아키텍처의 구성 요소인 모듈(컴포넌트)에 대한 기능 및 상세 절차(알고리즘) 등

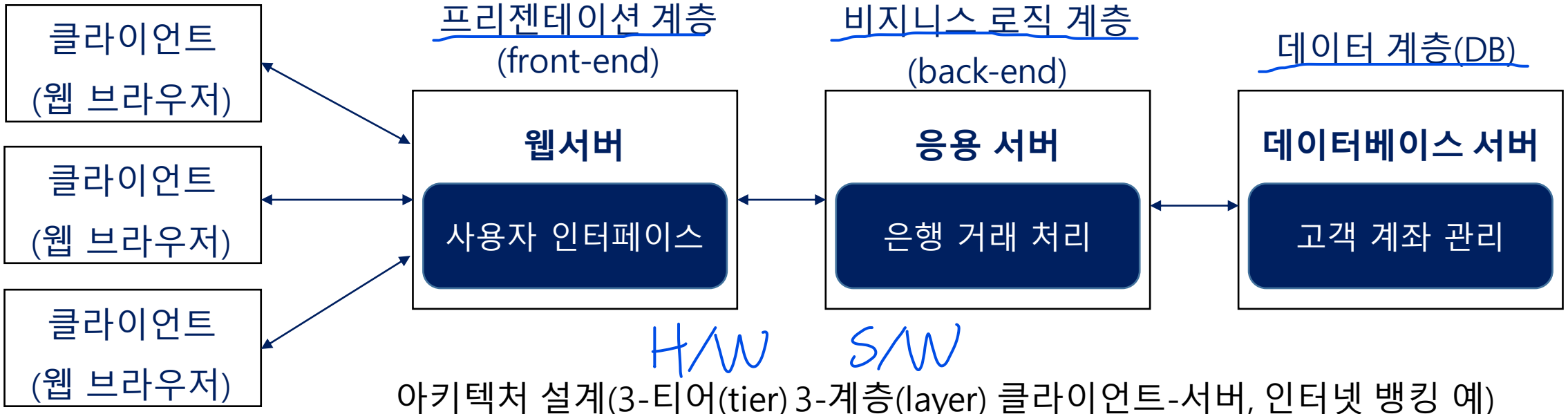
- **데이터 설계** (데이터베이스 설계)
  . 소프트웨어를 구현하기 위해 필요한 자료구조와 데이터베이스 구조를 정의하는 작업

- **인터페이스 설계**
  . 시스템 인터페이스(SI, System Interface) 설계, 사용자 인터페이스(UI, User Interface) 설계

아키텍처 설계

- **아키텍처 설계** : 상위 수준 설계 (시스템의 전체적인 구조, 큰 그림)

  - 상위수준에서 소프트웨어를 설계하는 기본 틀을 제공



아키텍처 설계(3-티어(tier) 3-계층(layer) 클라이언트-서버, 인터넷 뱅킹 예)

<웹서비스 기반 시스템의 클라이언트-서버 구조>

3-티어 아키텍처 : 시스템의 복잡성을 줄이고, 환경변화에 유용하게 대응할 수 있으며
효율적인 서버관리와 유지보수 등에 장점 (보안성, 성능 향상, 확장성)

계층

-프레젠테이션 계층

사용자 인터페이스를 지원하며, 사용자와 직접 만나는 화면을 만들어주는 부분

클라이언트 계층 또는 front-end라고도 함

웹서버에 해당하고, 주로 HTML CSS JavaScript, React.js 등으로 개발을 수행

-비즈니스 로직 계층

실제 사용자의 기능 요구사항 및 특정 응용 알고리즘을 구현하는 곳으로 어플리케이션 계층, 미들웨어 또는 back-end라고도 함. PHP 자바, 스프링 부트 등으로 개발을 수행

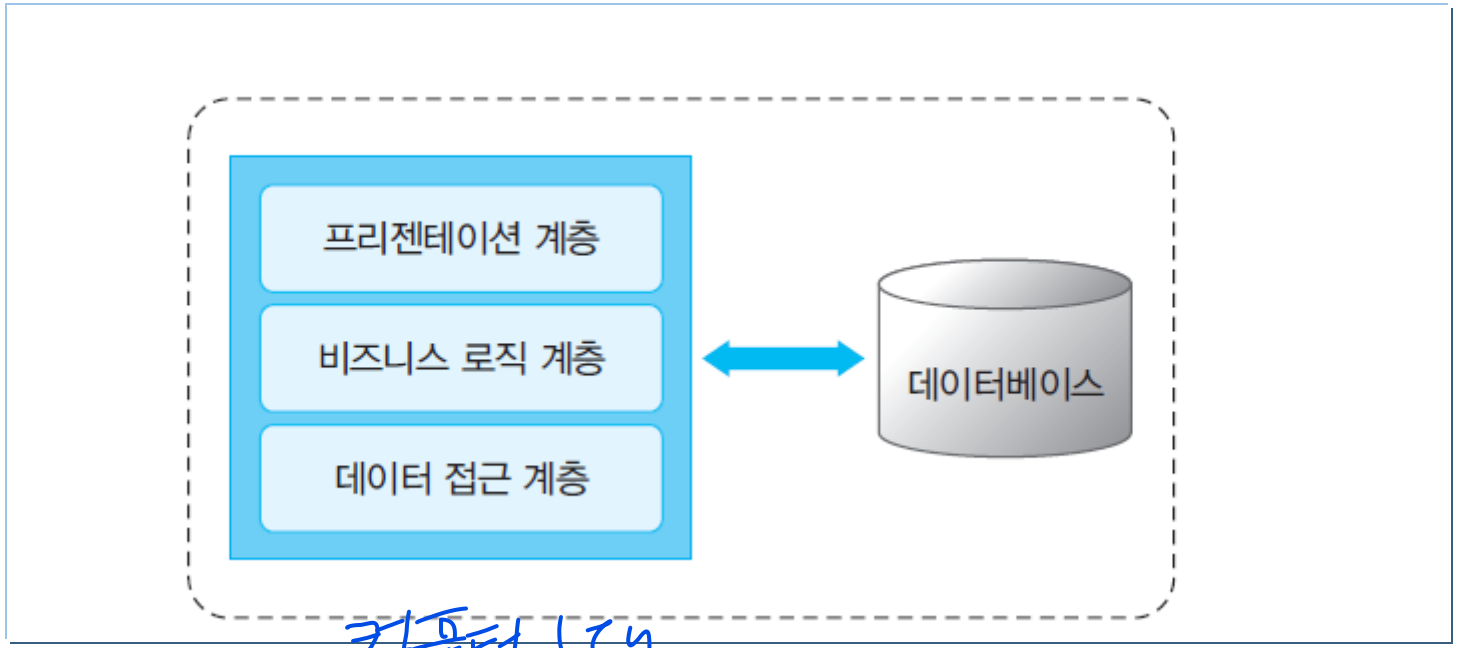
즉, 어플리케이션 로직을 구현하는 응용 서버, 추가적으로 어떤 형태의 데이터가 필요하고 반환될 것인지를 결정하는 데이터 엑세스 로직을 구현

-데이터 계층

데이터베이스 서버, 데이터베이스(DB)에 접근해 데이터 처리와 관리를 하며 필요한 데이터를 제공(DBMS)

MySQL, 오라클, MongoDB 등을 많이 사용

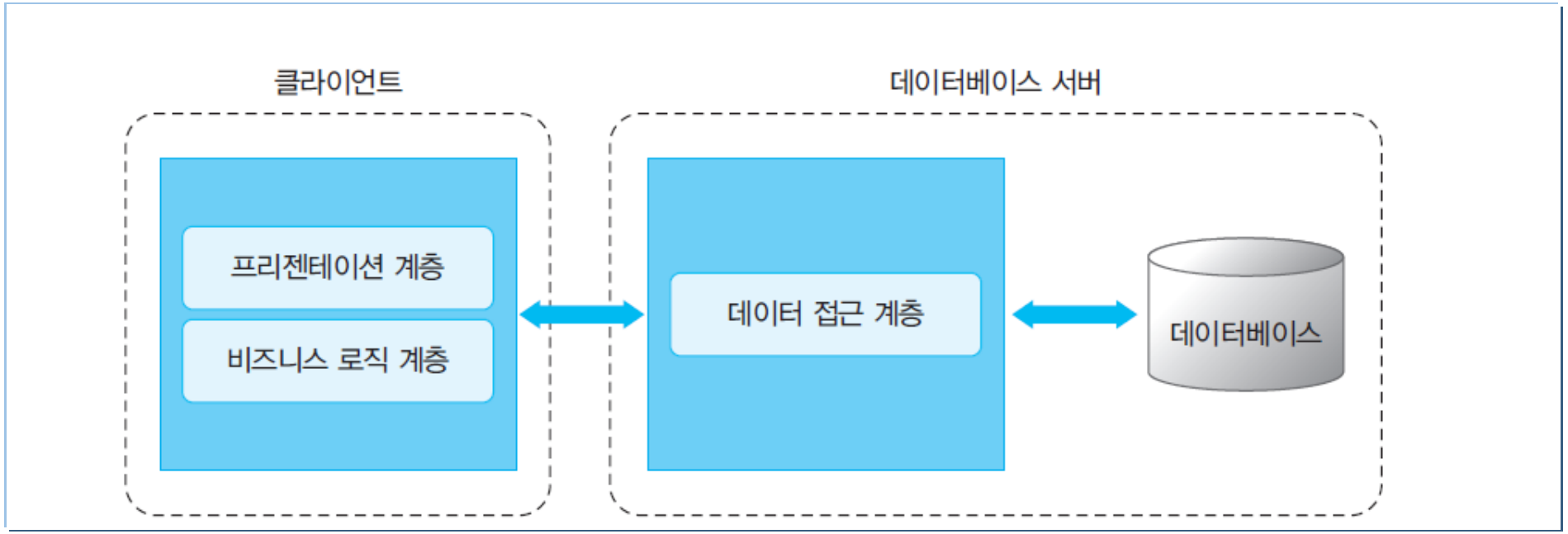
아키텍처 설계



아키텍춰 설계(1-티어(tier) 3-계층(layer) 클라이언트-서버 구조)

1-tier 아키텍처는 작은 비용으로 매우 쉽게 구성할 수 있는 장점이 있지만 확장성, 이식성 측면에서 적합하지 않는 구조일 뿐만 아니라 데이터베이스의 내용을 여러 사람이 공유할 수 없다.

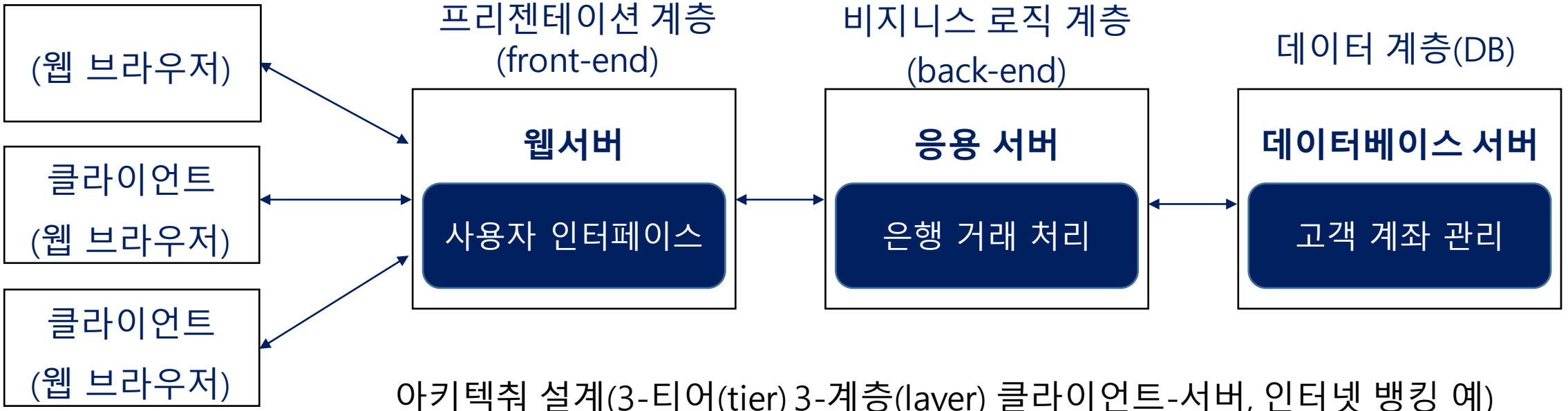
아키텍처 설계



아키텍춰 설계(2-티어(tier) 3-계층(layer) 클라이언트-서버 구조)

데이터베이스를 분리하여 여러 사람과 공유하여 사용할 수 있으며 데이터베이스의 변경이 용이하다. 그러나 클라이언트의 수가 늘어날수록 서버에 부하가 집중되어 전체 애플리케이션 성능이 떨어진다. 또한 비즈니스 로직이 변경되면 이에 따라 모든 클라이언트들을 변경해주어야 한다

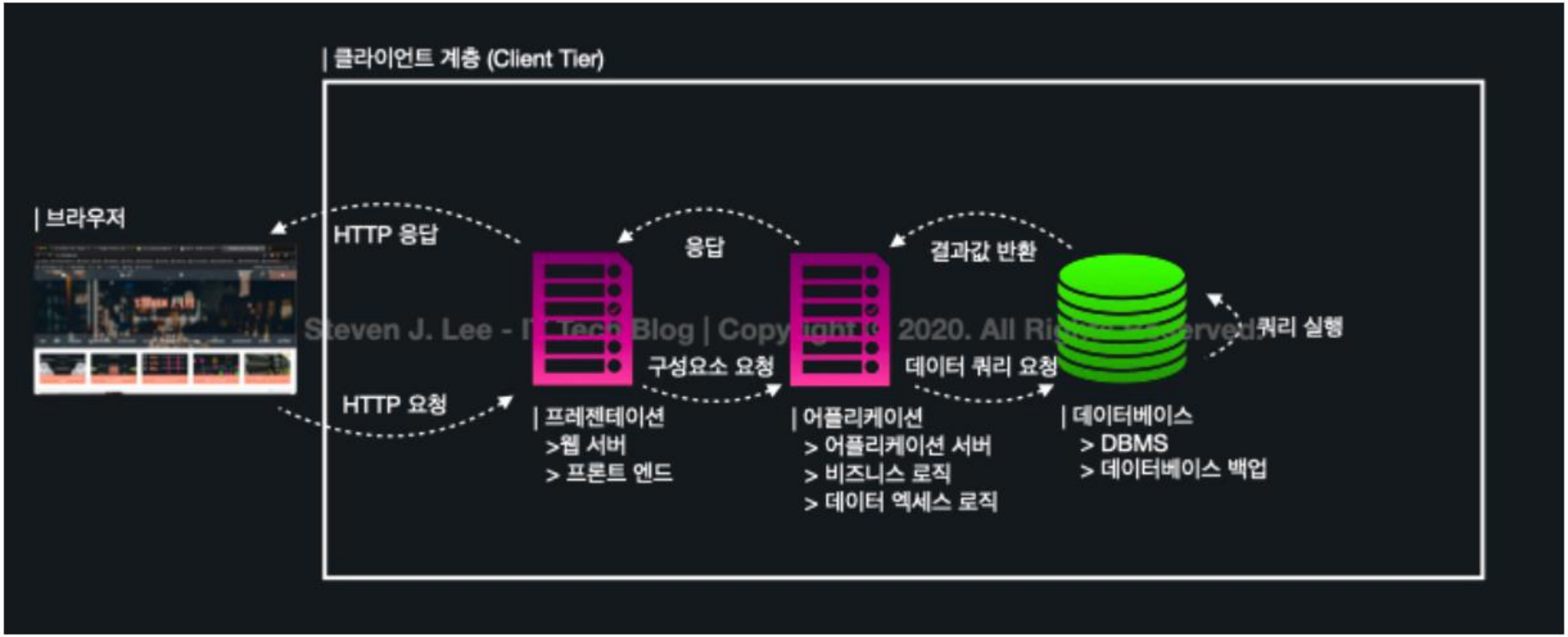
아키텍처 설계



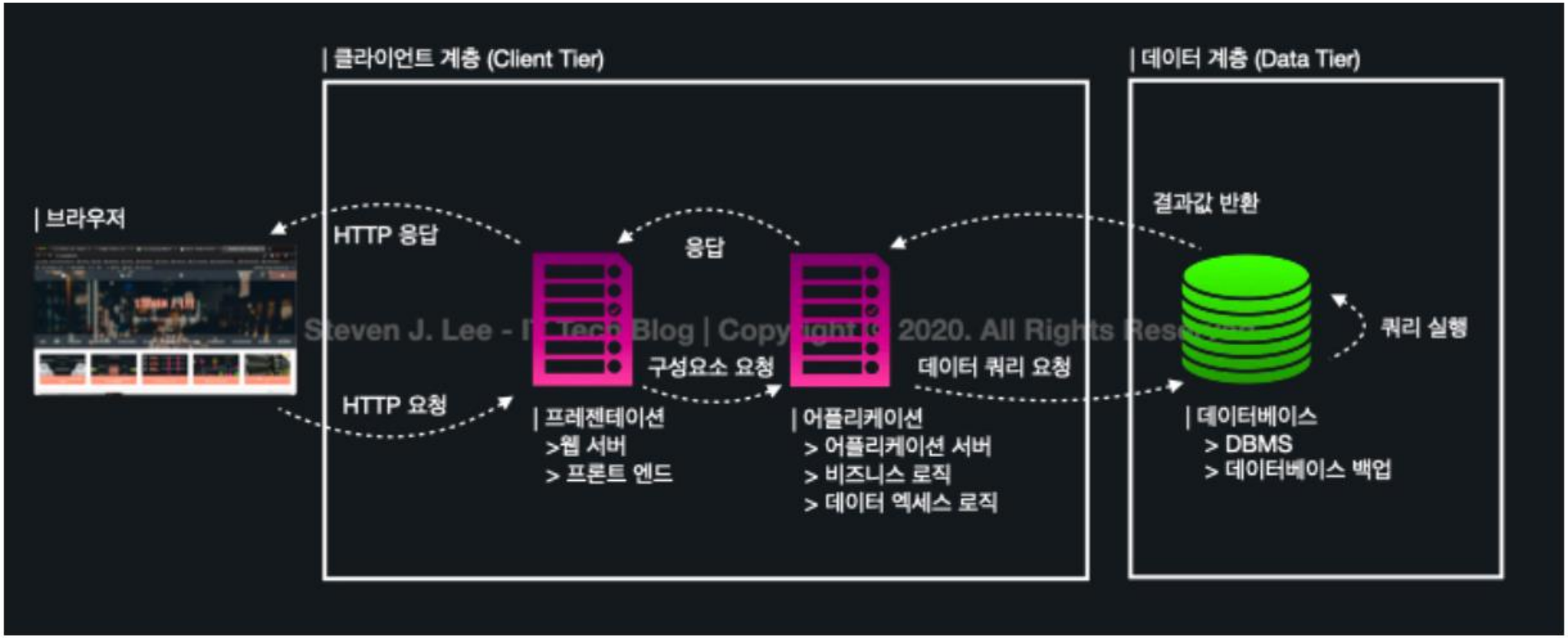
아키텍춰 설계(3-티어(tier) 3-계층(layer) 클라이언트-서버, 인터넷 뱅킹 예)

구성하는 작업이 복잡하고 비용이 많이 소모되지만 보안성(security), 성능(performance), 확장성(scalability) 등 여러 품질 속성 면에서 1-tier 아키텍처나 2-tier 아키텍처에 비해 장점을 지니고 있다.

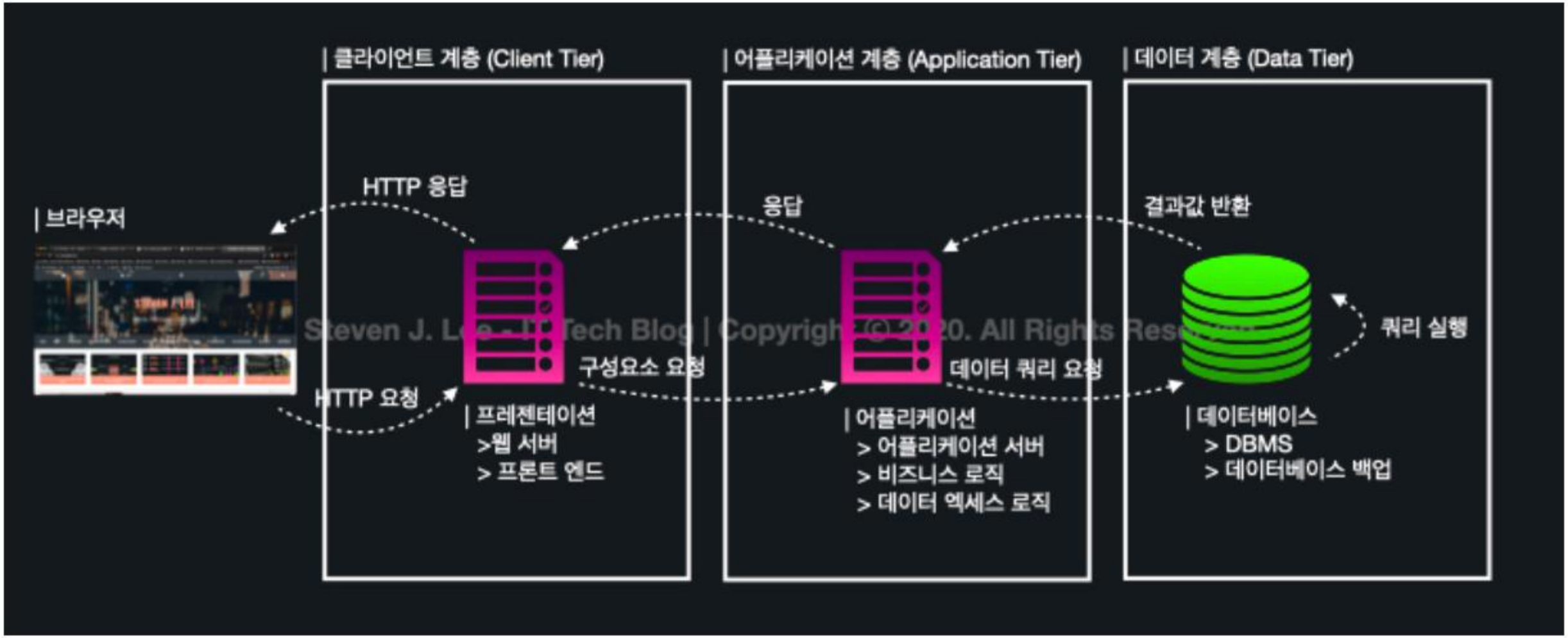
1. 1계층 구조



2. 2계층 구조



3. 3계층 구조

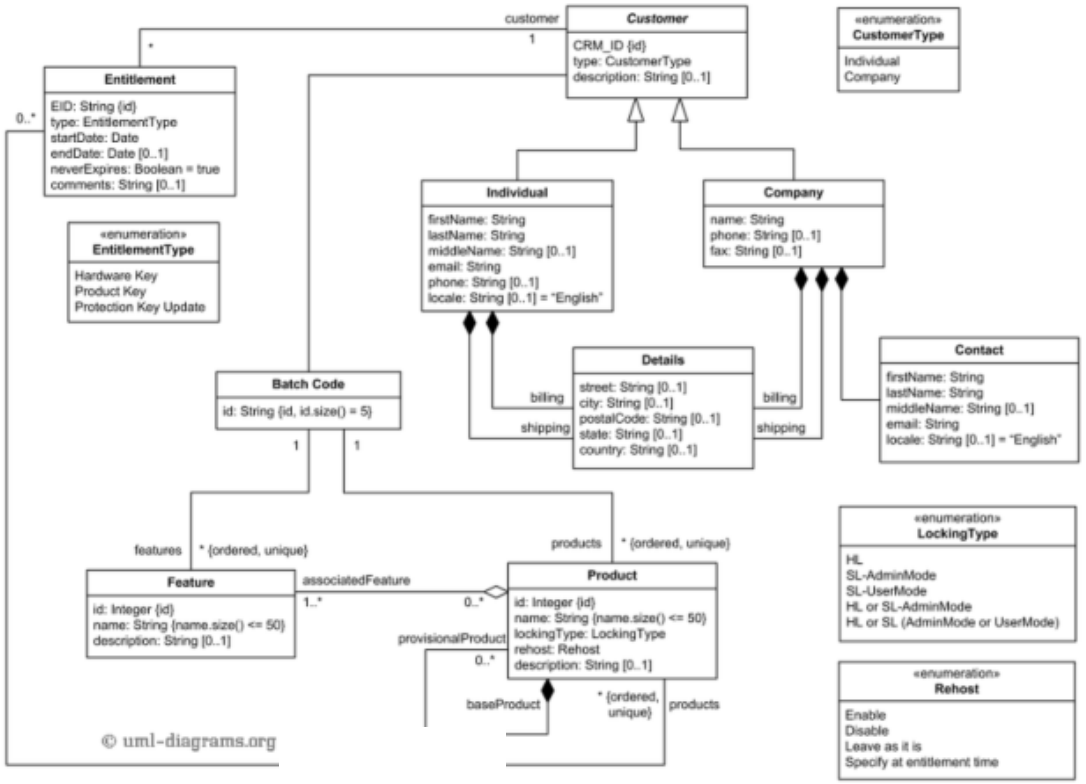

| 클라이언트 계층 (Client Tier)
| 어플리케이션 계층 (Application Tier)
| 데이터 계층 (Data Tier)
| 브라우저
HTTP 응답
응답
결과값 반환
쿼리 실행
구성요소 요청
데이터 쿼리 요청
HTTP 요청
| 프레젠테이션
>웹 서버
> 프론트 엔드
| 어플리케이션
> 어플리케이션 서버
> 비즈니스 로직
> 데이터 엑세스 로직
| 데이터베이스
> DBMS
> 데이터베이스 백업

상세 설계

- **상세 설계** : 하위 수준 설계 (상세 내용(모듈 등)을 정의)

  ☞ 컴포넌트 설계, 모듈 설계, 절차 설계 등으로도 불리움

-소프트웨어 아키텍처의 구성 요소인 모듈(컴포넌트)에 대한 기능 및 상세 절차(알고리즘) 등



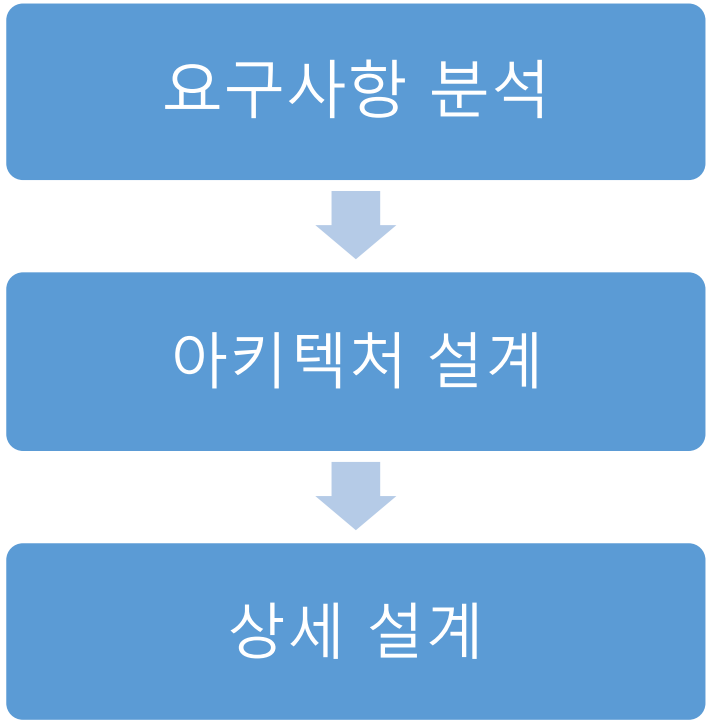
요구사항 분석

아키텍처 설계

상세 설계

모듈화(Modularity)

- 모듈화(Modularity)
  - 소프트웨어 공학의 원리 중 가장 근간이 되는 개념
  - 작은 단위로 나누는 것, 규모가 큰 것을 여러 개로 나눈 조각
  - 소프트웨어 구조를 이루는 기본 단위
  - 구체적으로 하나 또는 몇 개의 논리적인 기능을 수행하기 위한 명령어 묶음, 완전한 독립 프로그램



- 모듈화의 장점
  - 모듈화된 소프트웨어는 이해하기가 쉽고, 개발도 쉽다.
  - 팀 단위 개발 작업이 쉬워진다.
  - 변경에 의한 수정 사항 반영이 쉬워진다.
  - 재사용 가능성이 높아진다.
  - 요구사항 분석부터 설계 및 구현까지 일관성있고, 체계적인 개발 구조를 제공해 줄 수 있다.

모듈화(Modularity)

- 다양한 크기의 집합에 대해 모듈이라고 부를 수 있음
  - 용도가 비슷한 것끼리 묶어놓은 라이브러리 함수, 그래픽 함수
  - 추상화된 자료, 클래스(객체), 메서드 또는 그 묶음

모듈 간의 의존성 관계를 결합력이라고 함
결합력이 작을수록, 즉 다른 모듈에 의존하는 것이 작을수록,
즉 다른 모듈이 필요 없이 독립적으로 동작이 가능할수록
좋은 설계, 모듈 간의 상호 교류가 많고, 의존이 깊을수록 결합도가 높아져서 안 좋음

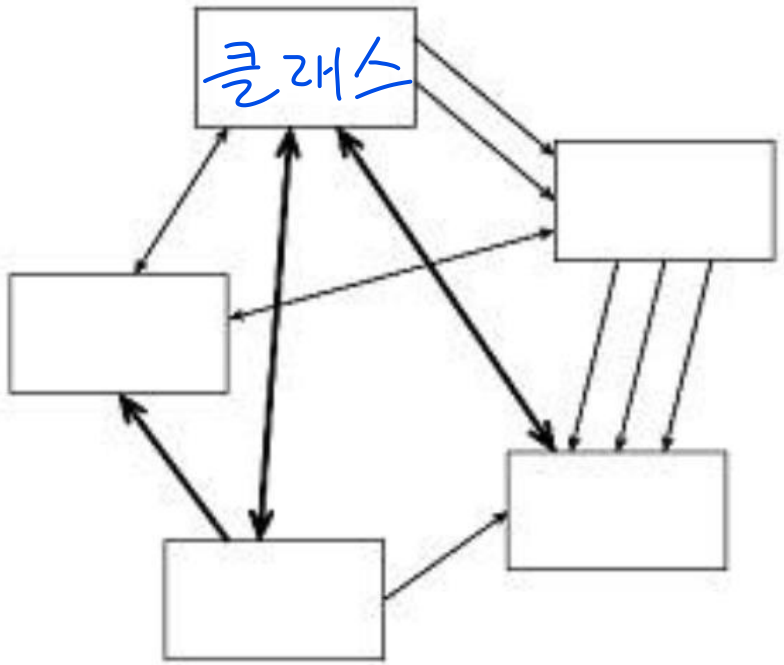
- 어느 정도의 크기로 나누어야 할 것인가?
  - 무조건 작게 나눈다고 좋은 것은 아님, 모듈의 크기가 작아지면 그만큼 개수가 많아지고, 그러면 모듈 간의 통신 횟수가 많아져 복잡해짐
  - 따라서 모듈이 크기에 정답은 없으며, 문제의 유형이나 특성을 고려해 결정해야 한다.

응집도가 낮을수록 서로 관련성이 적은 요소들이 모여있다
즉, 모듈 하나가 단일 기능으로 구성되지 않고,
쓸데없이 이것저것 섞여있으면 응집력이 약함
서로 얼마나 어울리는 요소들만 들어있는지가 응집도

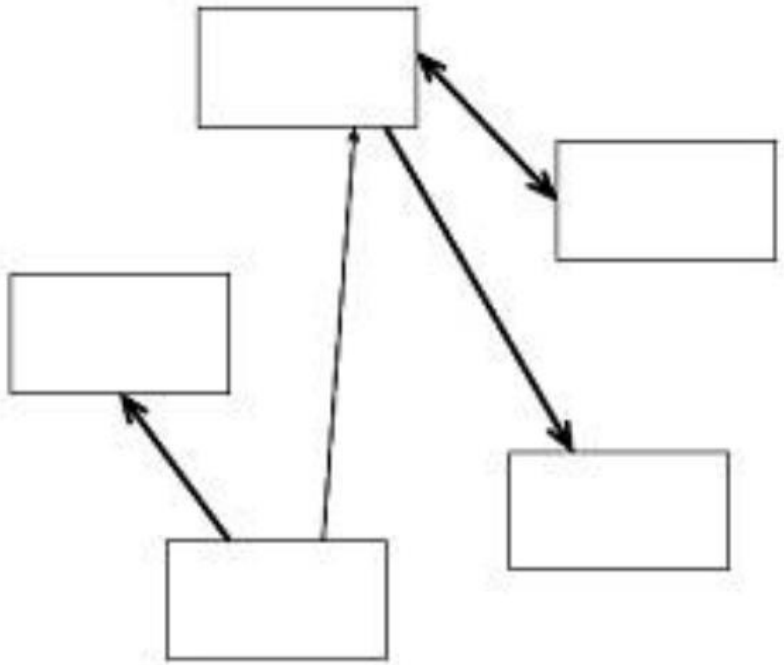
- 다음과 같은 원칙을 지키면 보통 좋은 설계라고 함
  - 모듈 간의 결합(coupling)은 느슨하게(최소), 모듈 내의 구성요소 간의 응집(cohesion)은 강하게(최대)

모듈화(Modularity)

- 결합력 예



강한 결합력(안좋음)



느슨한 결합력(보다 좋음)

모듈화(Modularity)

- 응집도 예 (예가 조금 억지스럽긴 하나 쉬운 이해를 위해 넣음)

  - 원의 면적과 둘레를 계산하는 클래스

```
class Circle {
    double pi = 3.14;
    double circleA(int radius) {
        return pi * radius * radius;
    }
    double circleC(int radius) {
        return 2 * pi * radius;
    }
}
```

높은 응집도

```
class Circle {
    double pi = 3.14;
    double e = 2.7182;
    double circleA(int radius) {
        return pi * radius * radius;
    }
    double circleC(int radius) {
        return 2 * pi * radius;
    }
}
```

관련성 X

낮은 응집도

컴포넌트/모듈/서브시스템

- 컴포넌트/모듈/서브시스템
  - 같은 의미로 사용되는 경우가 많으나 컴포넌트는 독립적으로 취급될 수 있는 단위로 프로그래밍 언어 측면에서 모듈, 시스템 측면에서 서브시스템이라고도 부름
  - 클래스 또는 클래스의 모임, 즉 패키지 (비객체지향 언어의 라이브러리와 유사)
  - 소프트웨어 개발의 효율을 높이기 위하여 '재사용 가능한 기능 단위'로 분리하여 제공
    - 객체(클래스)는 시스템을 구성하는 최소 단위, 이것을 모아서 재사용 가능한 '기능 단위'로 정리하여 구성한 것이 컴포넌트(모듈)
    - 따라서 한 개의 객체가 컴포넌트(모듈)를 구성할수도 있고, 수백 개의 객체가 컴포넌트를 구성할 수도 있음 (다 그런 것은 아니지만) 보통 10개 미안의 객체가 한 개의 컴포넌트(모듈)를 구축하는 것이 일반적인 형태
  - 시스템의 복잡도를 줄일 수 있음, 복잡한 시스템을 독립적인 서브 시스템으로 분할하면 수정 및 유지보수가 쉽고, 재사용성도 향상

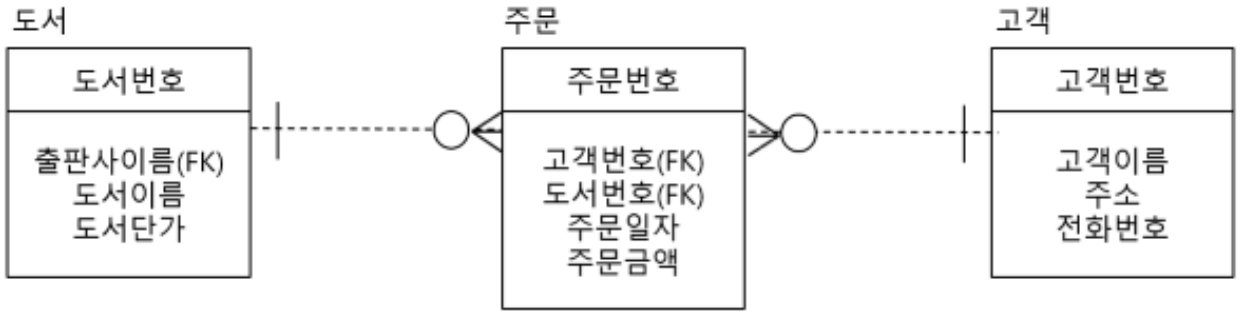
데이터 설계

- **데이터 설계**

  - 소프트웨어를 구현하기 위해 필요한 자료구조와 데이터베이스 구조를 정의하는 작업

    예) 데이터베이스 설계(데이터 모델링)

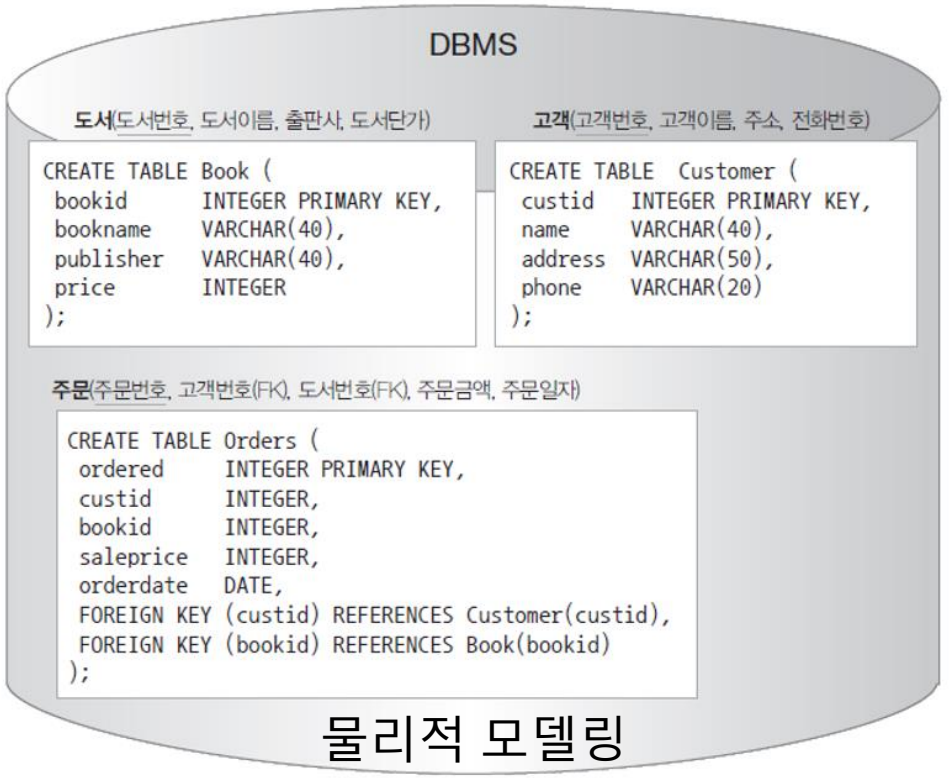
※ ER(Entity Relationship) 다이어그램을 이용한 개념적 모델링, 논리적 모델링, 물리적 모델링 등)



개념적 모델링

도서 (도서번호, 도서이름, 출판사이름, 도서단가)
고객 (고객번호, 고객이름, 주소, 전화번호)
주문 (주문번호, 고객번호(FK), 도서번호(FK), 주문일자, 주문금액)

논리적 모델링



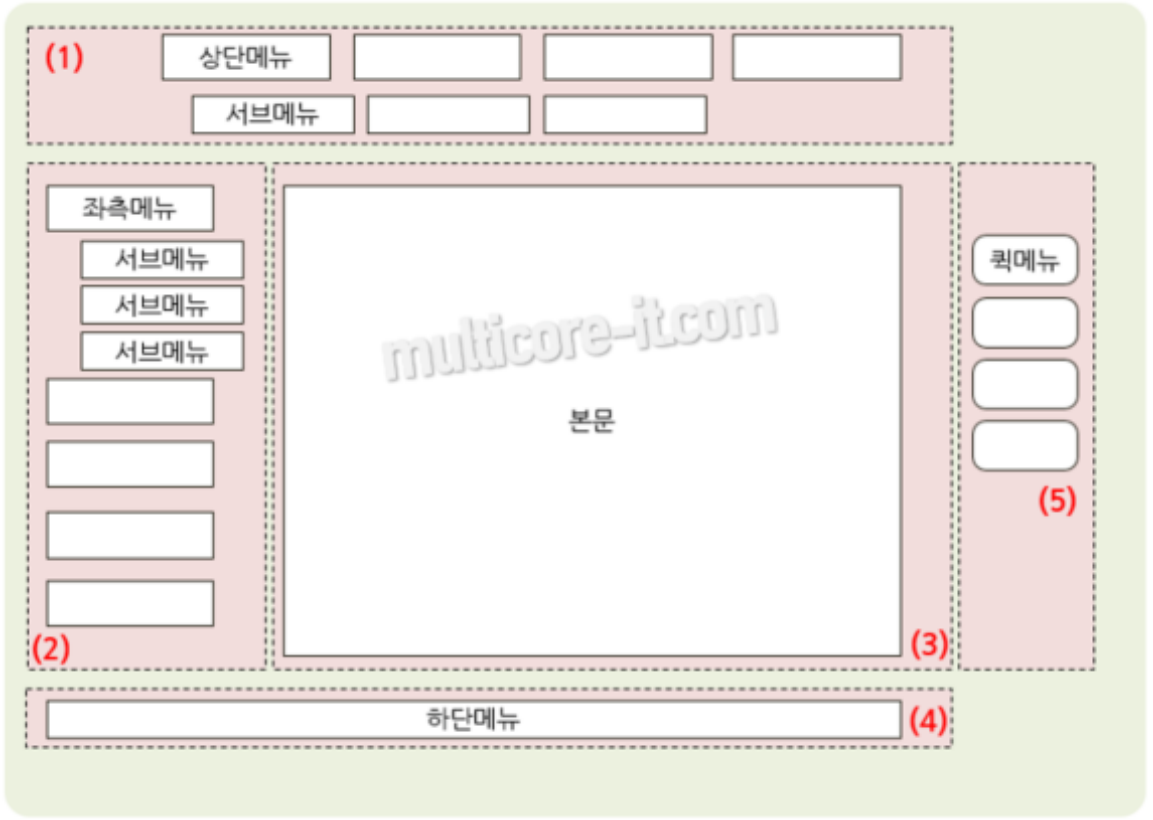
물리적 모델링

인터페이스 설계

시스템 인터페이스 설계는 소프트웨어 모듈 사이의 인터페이스를 정의하거나 또는 외부 시스템 사이의 인터페이스를 정의하는 부분이고,
사용자 인터페이스는 사용자와 컴퓨터 소프트웨어 사이의 인터페이스를 정의하는 부분

- **인터페이스 설계**

. 시스템 인터페이스(SI, System Interface) 설계, 사용자 인터페이스(UI, User Interface) 설계



사용자 인터페이스(UI, User Interface)

구현(코딩)

- 구현(코딩)

- 코딩 단계의 작업은 설계한 기능을 프로그램으로 구현하는 일에만 초점을 두지 않아야 한다.
  가독성을 높이고, 재사용성 및 유지보수를 편하게 하는 관점에서 작업할 필요가 있다.
- 코딩을 하다보면 '문제가 될 가능성이 있는 부분' 또는 '중복된 부분'이 생기거나, 또는 로직이 복잡해질 위험성 등이 존재
  따라서 중간중간 리팩토링을 수행해서 보다 단순하게 코드를 만들어 줄 필요가 있다.
  리팩토링 : 기능을 변경하지 않고, 코드의 내부 구조를 개선하는 작업

T h a n k y o u

TECHNOLOGY

Lorem ipsum dolor sit amet, consectetur adipisicing elit. Velit ex
ullicabo ipsum, labore sed tempora ratione asperiores des
cipserat bore sed tempora rati jgert one bore sed tem!