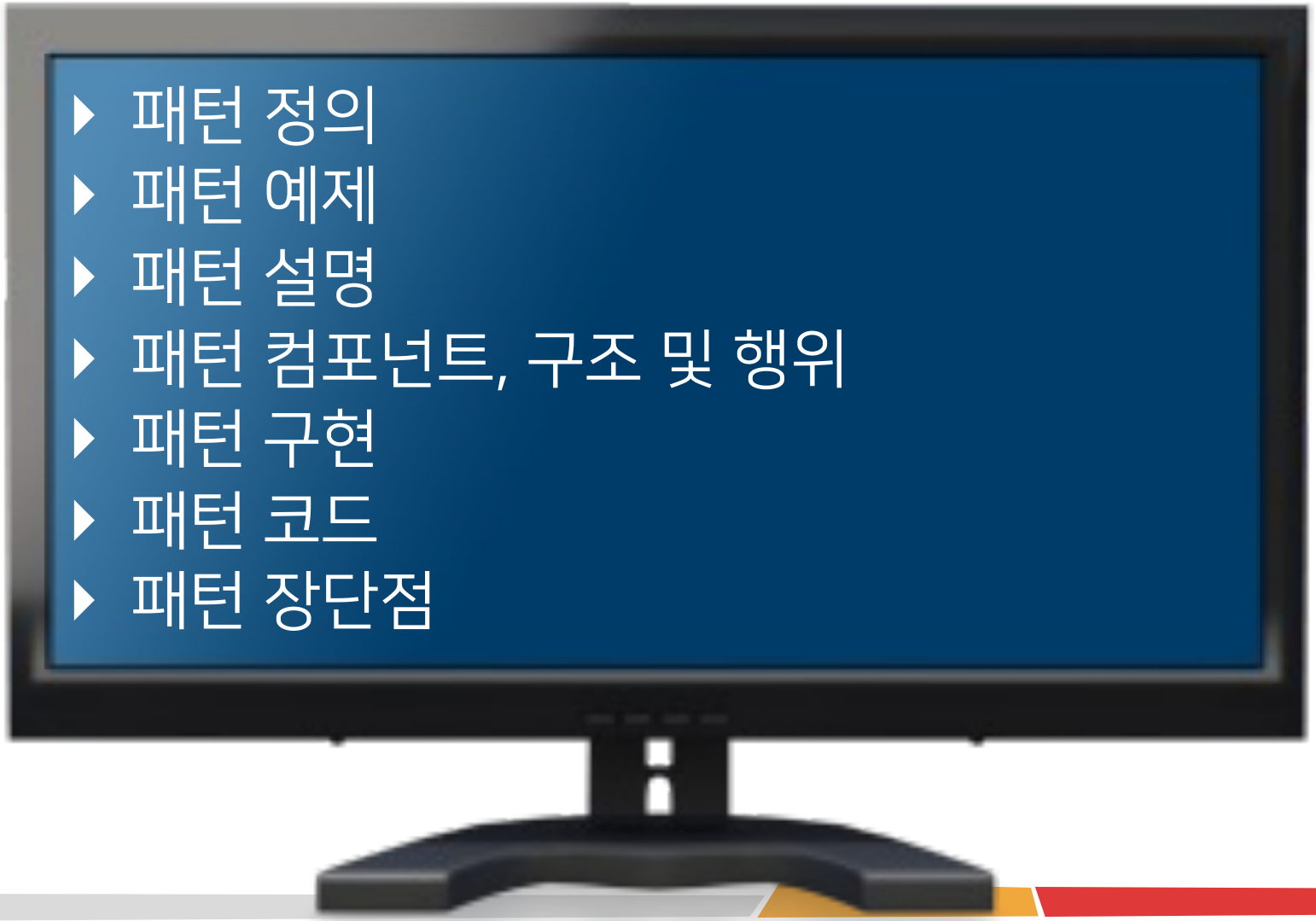
Spring 2023

소프트웨어 아키텍처 패턴: Layers

Seonah Lee

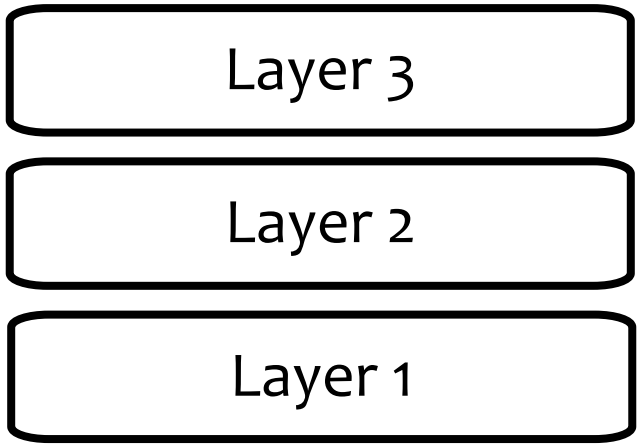
Gyeongsang National University

Layers 패턴



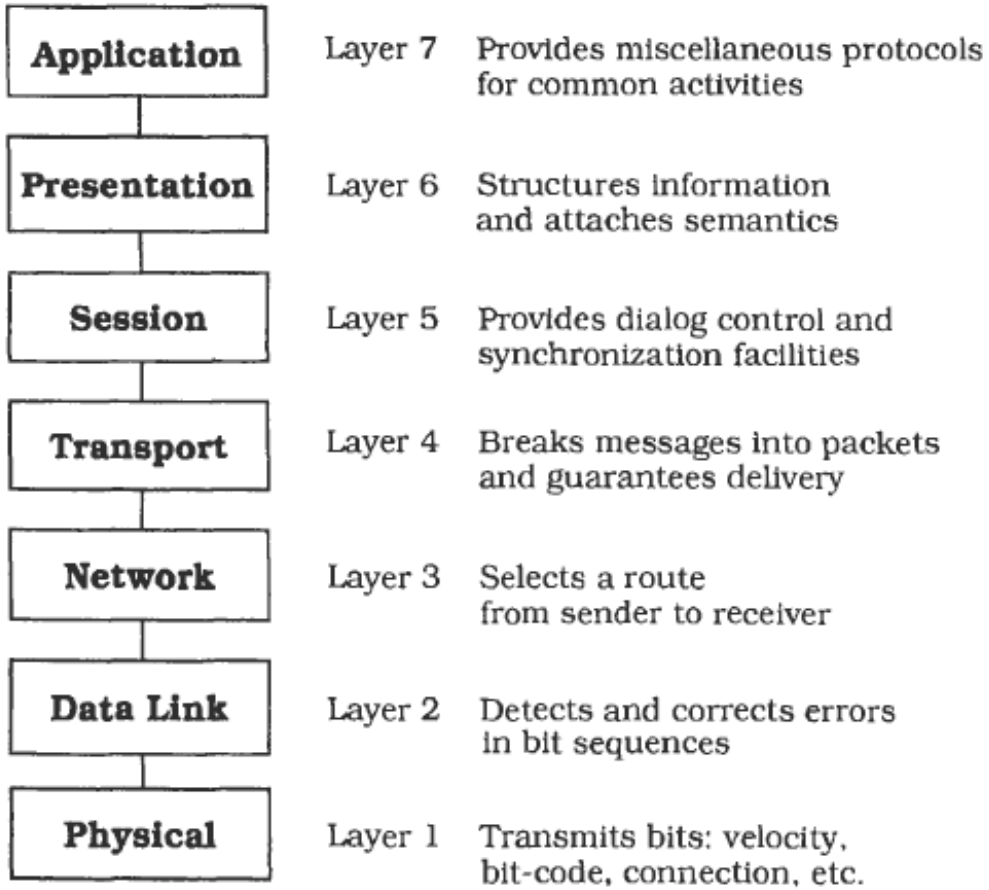
Layers Pattern: Definition

- 정의
  - 특정 추상 레벨에 있는 서브태스크들끼리 서로 묶어서 하나의 그룹으로 분류
  - 하위 수준의 이슈를 상위 수준에 이슈와 분리시켜 소프트웨어의 재사용성을 높임



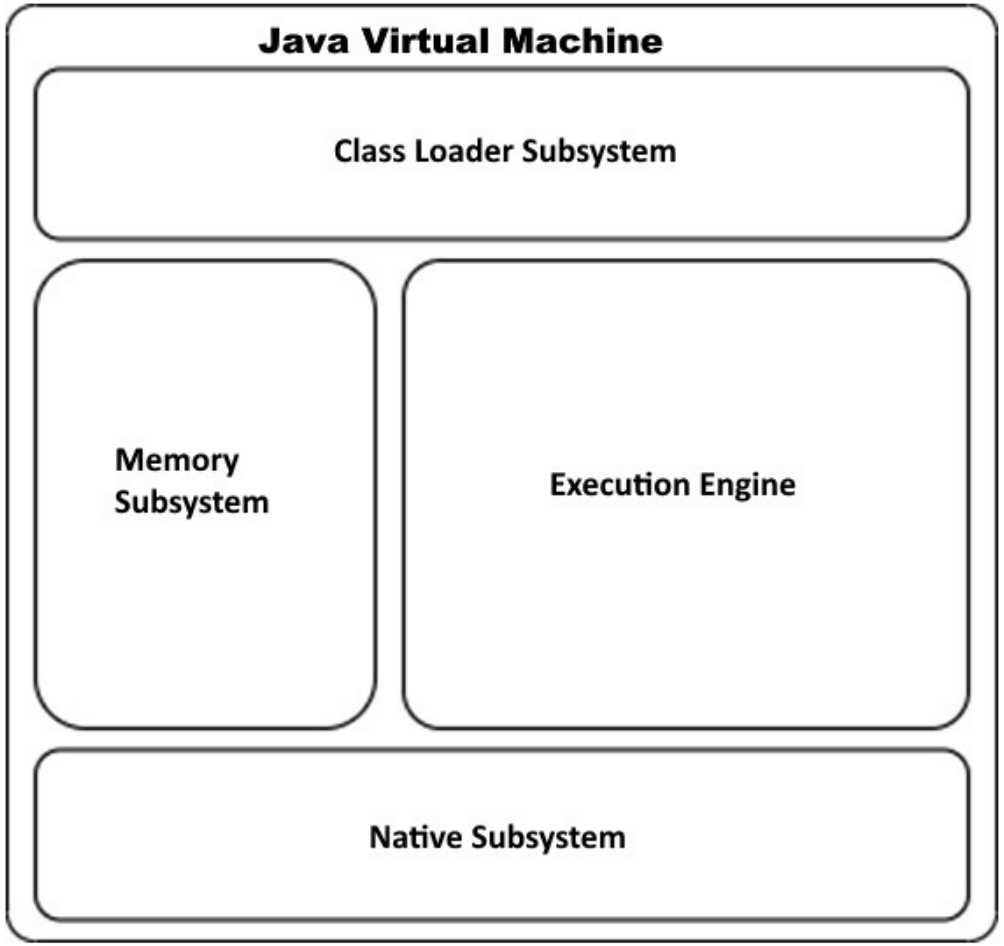
Layers Pattern: Example

- 네트워크 프로토콜 아키텍처 **(e.g. OSI 7 layer)**



Layers Pattern: Example

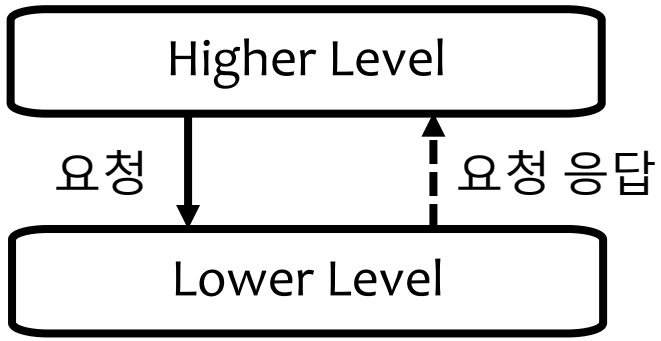
- 가상 머신 (e.g. interpreters, JVM)



Layers Pattern: Description

- 정황 **(Context)**
  - 시스템의 규모가 커서 분해할 필요가 있을 경우

- 문제 **(Problem)**
  - 하위 레벨과 상위 레벨 이슈가 서로 혼재해 있다는 점이 주된 특징인 시스템 설계
    - 통신 흐름의 일반적인 형태



  - 시스템의 기능이 수직적으로 나눠져 있거나 수평적인 경우와 혼재

Layers Pattern: Description

- 해법 (Solution)
  - 시스템의 상호연동 관계가 있는 모듈들을 모아 계층으로 추상화
    - 최상위 계층 : **Layer N**
    - 최하위 계층: **Layer 1**
  - **Layer J**가 제공하는 대부분의 서비스는 **Layer J-1**이 제공하는 서비스로 구성
    - 다른 계층의 서비스를 사용해서는 안됨
    - 단, **Layer J** 내에 있는 서비스는 **Layer J**에 있는 다른 서비스에 종속될 수 있음

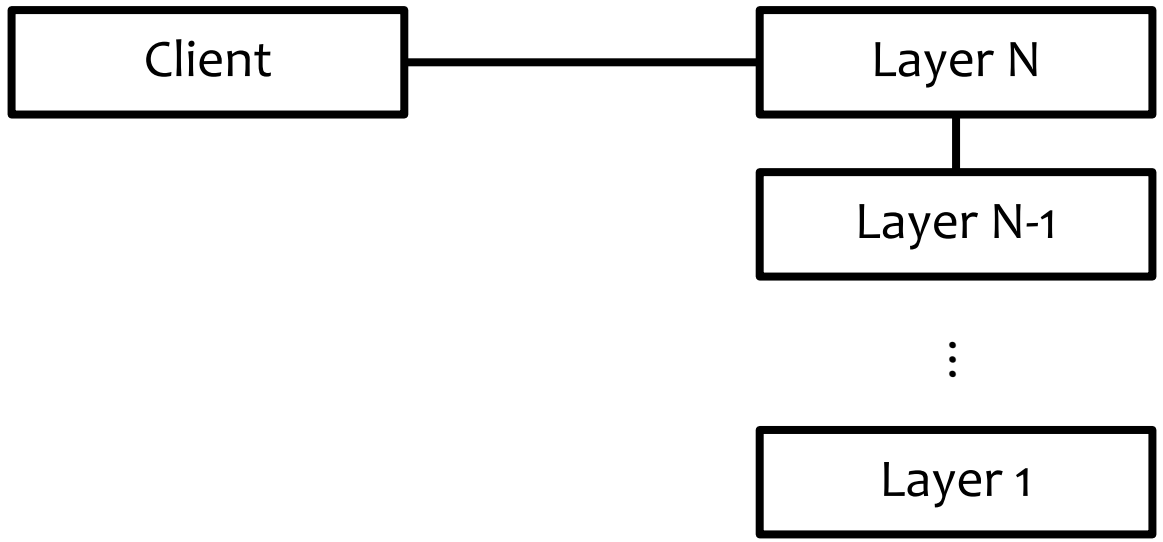
Layers Pattern: Components

- 피해야 할 경우
  - **Layer J**가 더 분해할 필요가 있는 복합 시스템으로 정의해서는 안됨
  - **Layer J**가 **Layer J+1**에서 넘겨진 요청을 **Layer J-1**로 넘길 요청으로 변환하는 역할만 수행 **(Layer J**가 주요하게 수행하는 일이 없음**)**

Layer J
• Layer J+1에 의해 사용되는 서비스 제공 • Layer J-1에 서브태스크 위임

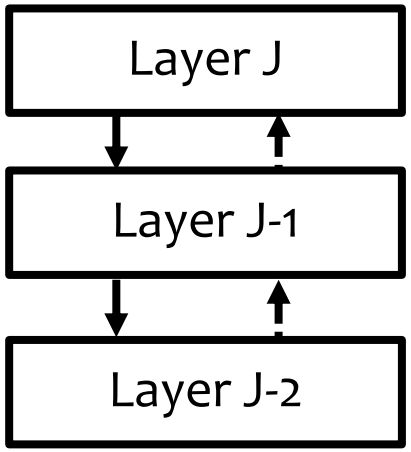
Layers Pattern: Structure

- **J** 레이어의 상위 레이어들은 **J** 레이어의 하위 레이어에 직접 액세스할 수 없도록 함

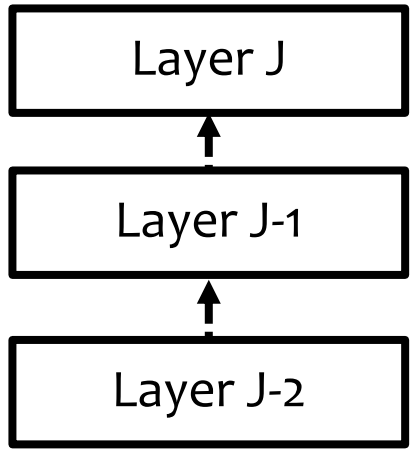


Layers Pattern: Behavior

- 시나리오 I*실습
  - **Top-down communication**
    - 하위 레이어로 요청**(request)**
    - 상위 레이어로 요청 응답
    - 정보나 제어 흐름



- 시나리오 II
  - **Bottom-up communication**
    - 상위 레이어로 통지**(notification)**
    - 데이터가 타고 올라감



Layers Pattern: Realization

- 구현 순서
  1. 레이어 별로 모듈을 묶는 추상 기준을 정의
     - 예: 체스 게임 애플리케이션

게임 전체 전략
방어를 하는 구체적인 전술
체스 말들의 기본 이동
비숍(Bishop)과 같은 체스 게임의 기본 요소

  2. 추상 기준에 따라 레이어을 몇 레벨로 나눌지 결정
     - 레이어가 너무 많으면 ➔ 불필요한 부하**(Overhead)**
     - 레이어가 너무 적으면 ➔ 조잡한 구조

Layers Pattern: Realization

- 구현 순서
  3. 레이어마다 역할 부여 및 태스크 할당
     - 최상위 레이어: 클라이언트에 의해 감지되는 전체 시스템 태스크
     - 다른 모든 레이어: 상위 레이어의 보조 역할; 상위 레이어의 인프라 제공
  4. 레이어 별 제공 서비스를 상세히 정의
     - 상위 레이어: 광범위한 적용성을 확보하도록 확장
     - 기반 레이어: 가능한 가볍게 유지

     } 재사용의 역 피라미드 구조 (Inverted pyramid of reuse)
  5. 레이어 별 상세 인터페이스 정의
     - **Layer J**의 서비스를 제공하는 인터페이스 설계 및 캡슐화

Layers Pattern: Realization

- 구현 순서
  6. 시스템 기능이 레이어에서 동작하는 것이 가능한지 확인
     - 예: 유스케이스 시나리오를 시뮬레이션 하는 방식
  7. 레이어 내부에 대한 구조 정의
     - 복잡할 경우 레이이어를 독립된 컴포넌트로 나눔
  8. 인접한 레이어 간의 통신 방식 정의
     - 푸시 모델: **Layer J**가 **Layer J-1** 호출 시 모든 정보를 서비스 호출의 일부분으로 전달
     - 풀 모델: **Layer J-1**이 스스로의 필요에 따라 **Layer J**에서 정보를 가져옴(콜백 사용)
  9. 예외 처리 방식을 정의

Layers Pattern: Implementation

```
main() {
    DataLink dataLink;
    Transport transport;
    Session session;

    transport.setLowerLayer(&dataLink);
    session.setLowerLayer(&transport);

    session.L3Service();
}
```

Layers Pattern: Implementation

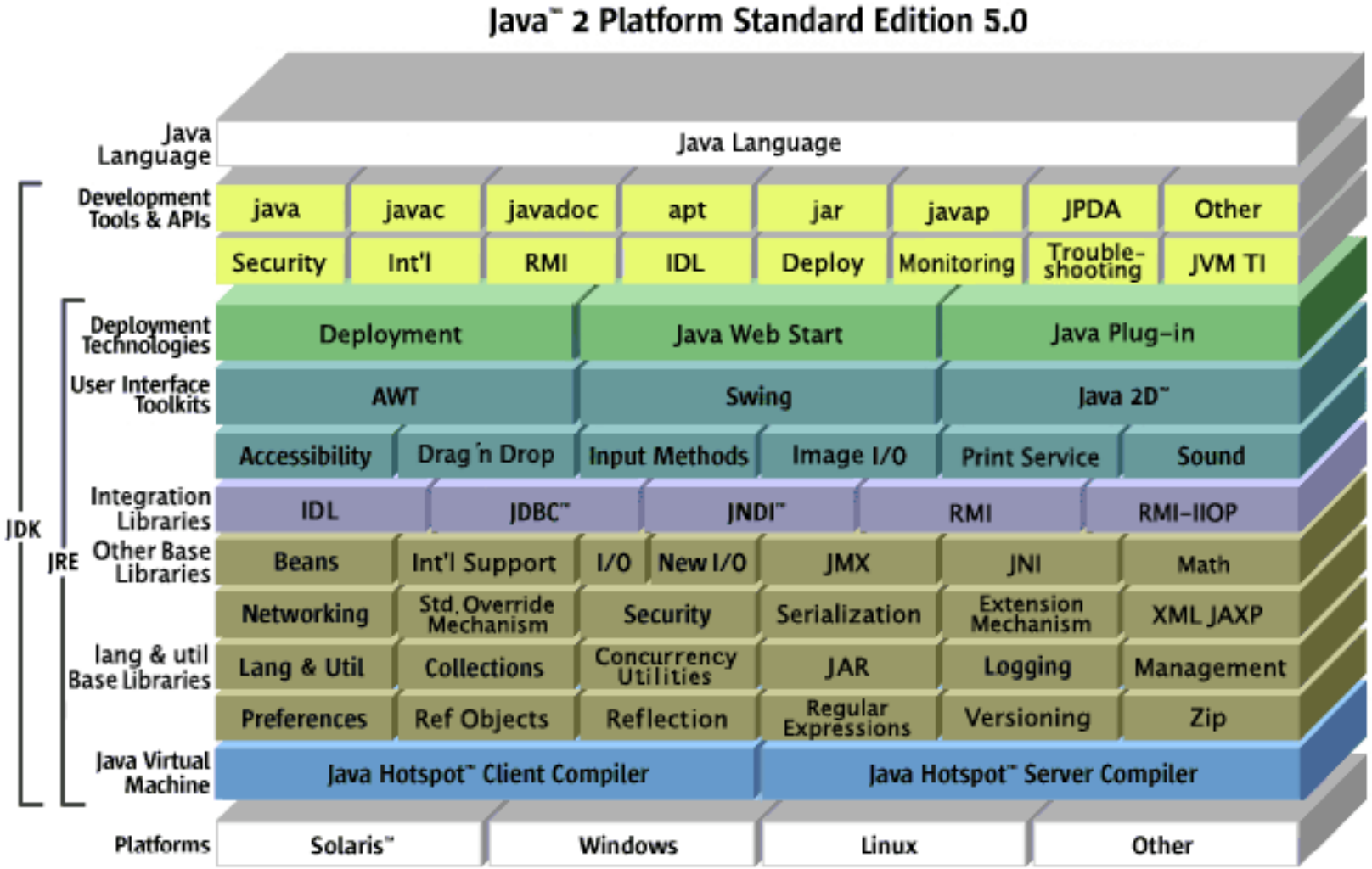
```
#include <iostream.h>

class L1Provider {
public:
     virtual void L1Service() = 0;
};
class L2Provider {
public:
     virtual void L2Service() = 0;
     void setLowerLayer(L1Provider *l1) {level1 = l1;}
protected:
     L1Provider *level1;
};
class L3Provider {
public:
     virtual void L3Service() = 0;
     void setLowerLayer(L2Provider *l2) {level2 = l2;}
protected:
     L2Provider *level2;
};
```

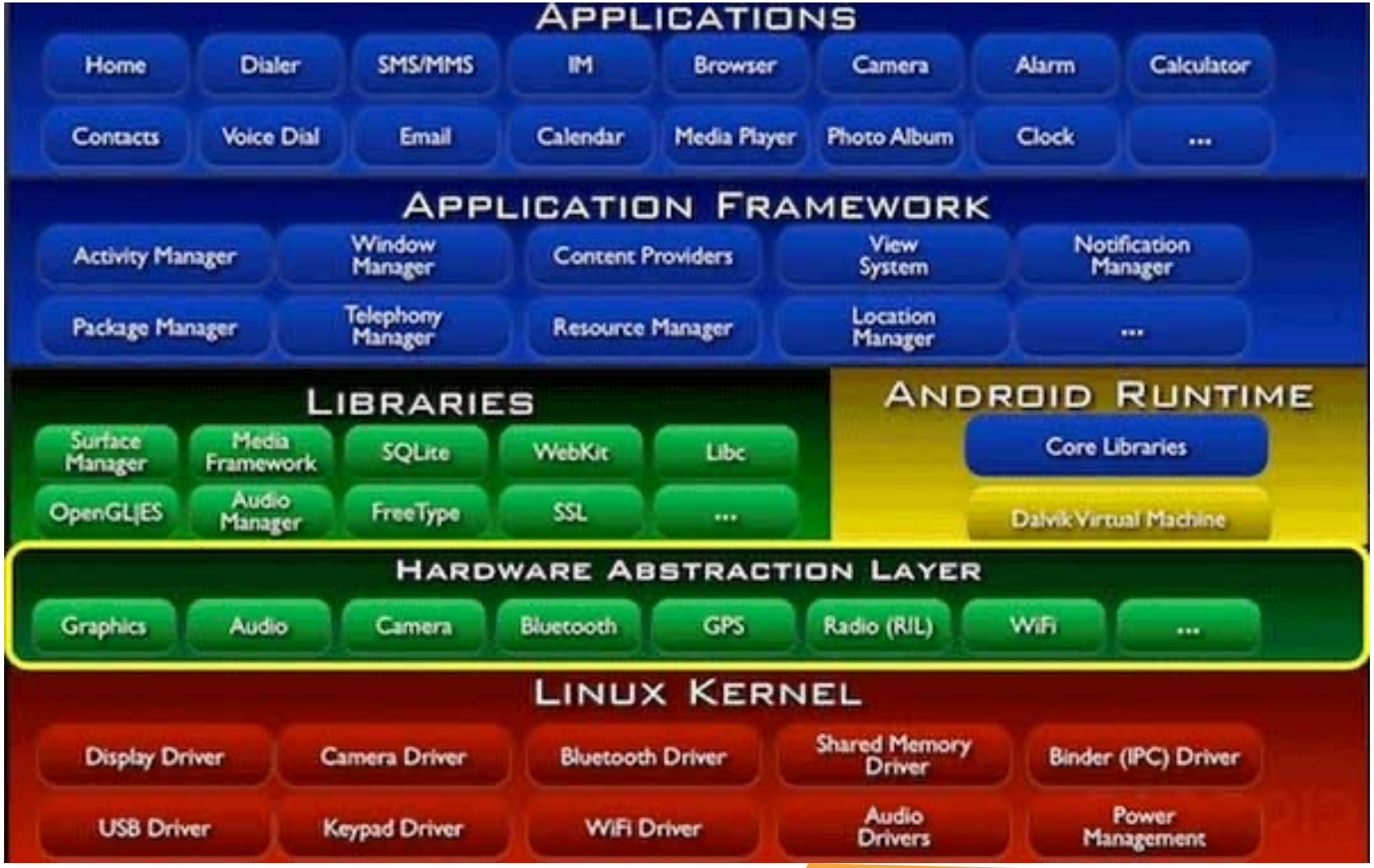
Layers Pattern: Implementation

```
class DataLink : public L1Provider {
public:
     virtual void L1Service(){
          cout << "L1Service doing its job" << endl;}
};
class Transport : public L2Provider {
public:
     virtual void L2Service() {
          cout << "L2Service starting its job" << endl;
          level1->L1Service();
          cout << "L2Service finishing its job" << endl;}
};
class Session : public L3Provider {
public:
     virtual void L3Service() {
          cout << "L3Service starting its job" << endl;
          level2->L2Service();
          cout << "L3Service finishing its job" << endl;}
};
```

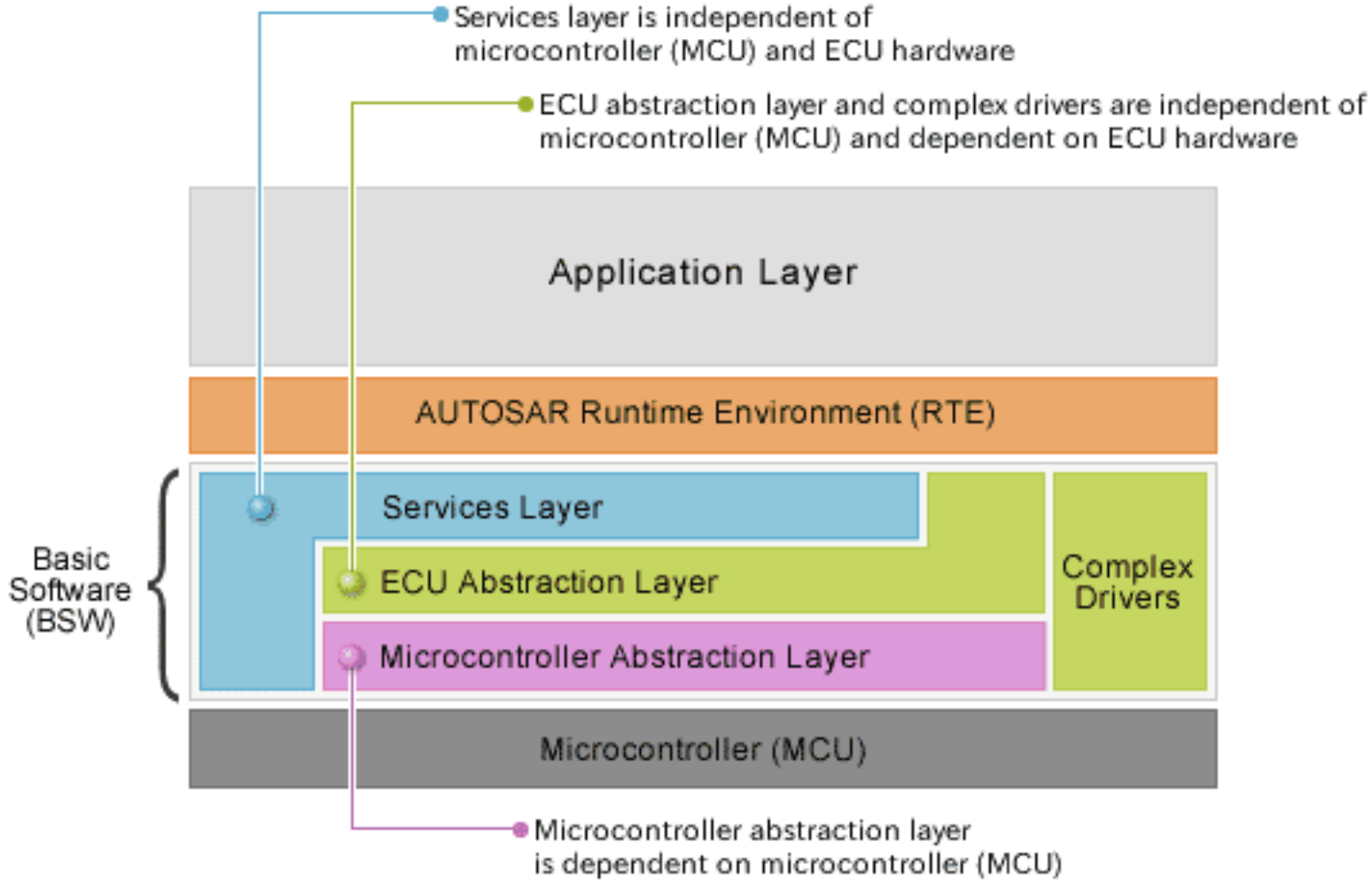
Layers Pattern: Case Studies



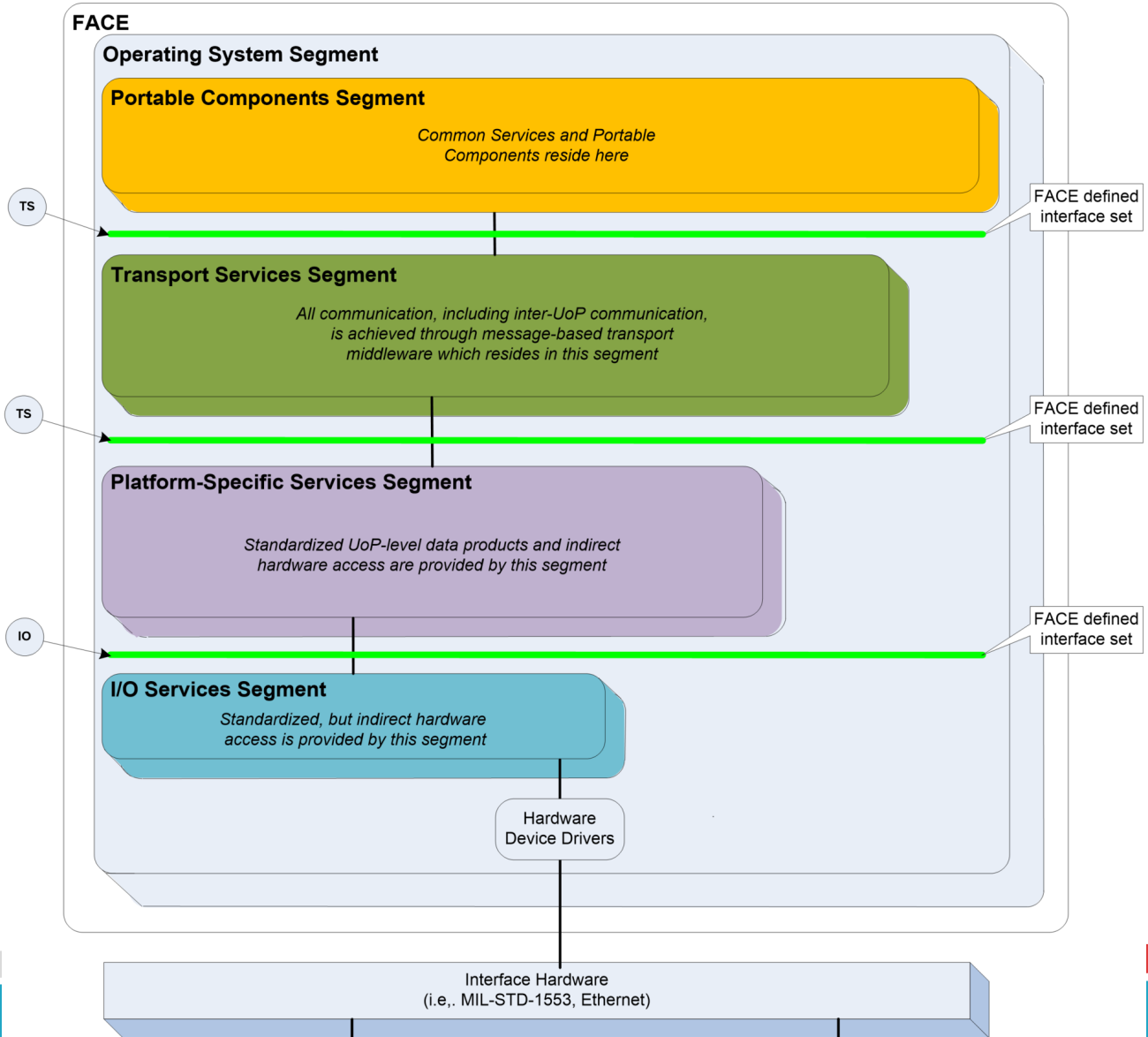
Layers Pattern: Case Studies



Layers Pattern: Case Studies

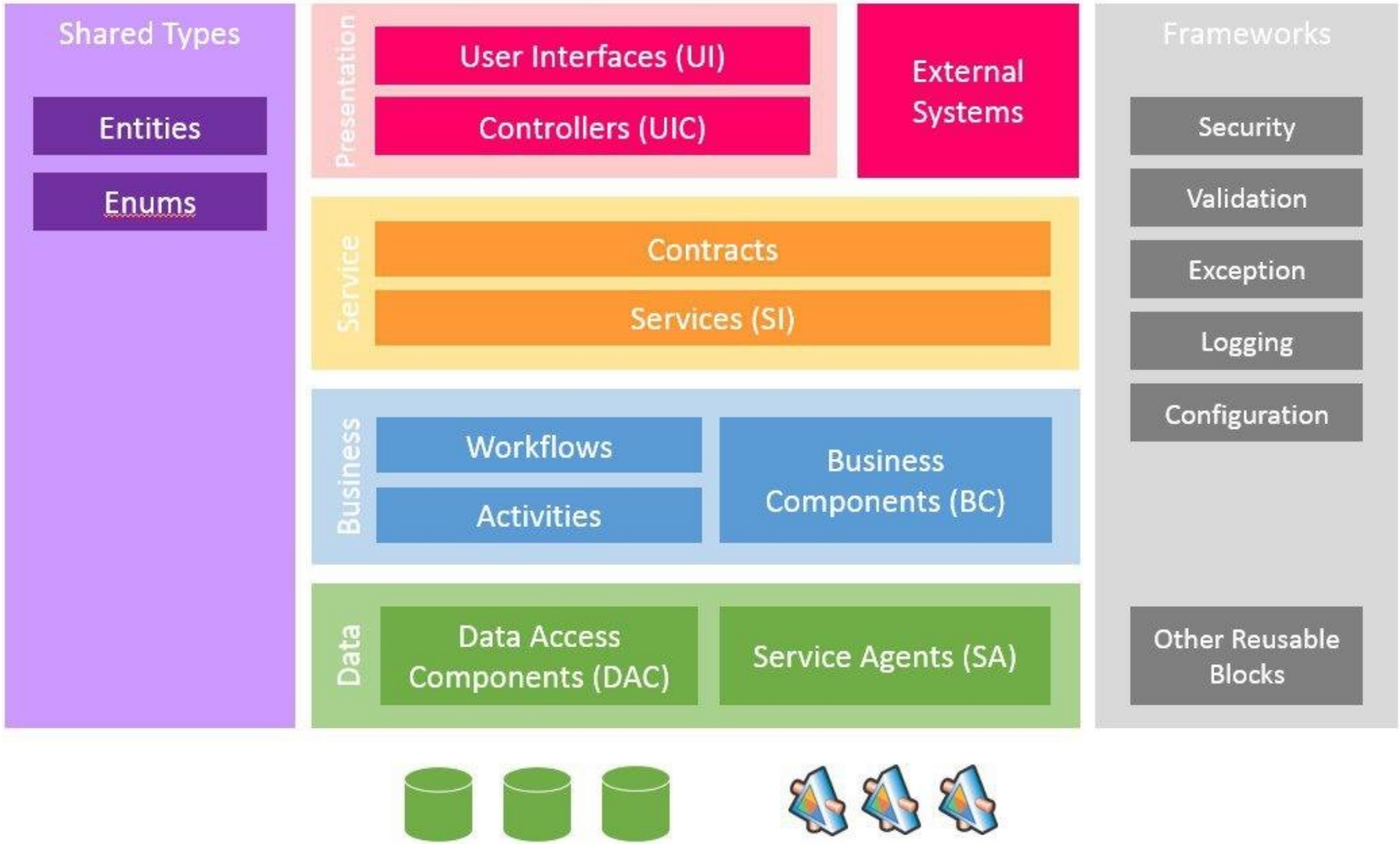


Layers Pattern: Case Studies



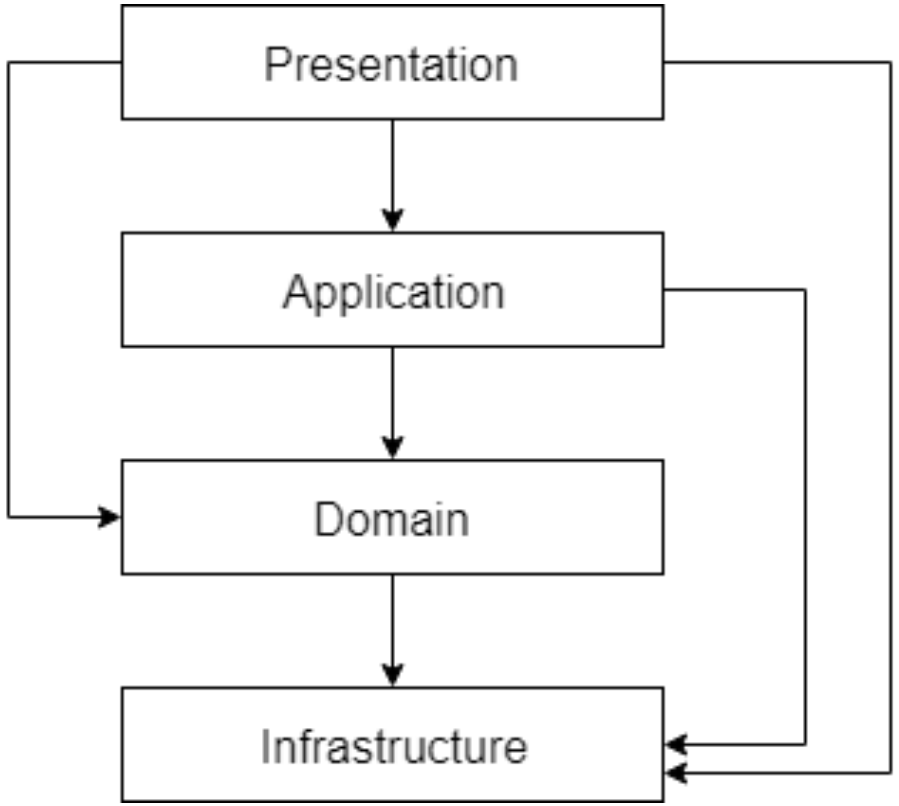
Source: Open Group

Layers Pattern: Case Studies



Layers Pattern: Case Studies

- Information System



Layers Pattern: Benefits

- 레이어 별 연동을 한정할 수 있어 **Loosely coupled** 원칙을 지킬 수 있음
- 레이어 재사용 가능
  - 기존에 있던 레이어를 재사용하여 개발 노력과 결함을 줄일 수 있음
  - 유지보수성이나 이식성이 필요한 시스템에 적용하기 좋은 패턴임
- 종속성을 국지적으로 최소화함
  - 레이어 별로 변화에 대한 영향력을 한정할 수 있음
  - 코딩이나 테스트를 레이어 별로 진행할 수 있음
- 교환 가능성 높임
  - 인터페이스 정의가 잘 되어 있다면 레이어를 통째로 교체할 수 있음

Layers Pattern: Liabilities

- 레이어 동작 변경 시 단계별 재작업 필요
  - 보기에 국지적인 변경이 여러 레이어에 걸쳐 많은 재작업을 수행할 가능성 있음
- 효율이 낮음
  - 계층의 원칙을 지키기 위해 각 계층을 모두 거침 ➔ 성능 측면에 불이익을 받을 수 있음
- 불필요한 작업 수행
  - 성능: 여러 상위 레벨 요청을 처리하기 위해 동일한 비트 시퀀스를 여러 번 읽음
  - 변경: 계층을 구분하기 어렵고 잘못 구분할 경우 설계 수정이 빈번히 발생할 수 있음
- 계층의 적절한 개수 및 규모를 정의하는 것이 어려움
  - 너무 적은 갯수의 레이어 ➔ 재사용성, 가변성, 이식성 활용이 어려움
  - 너무 많은 갯수의 레이어 ➔ 복잡성(Complexity) 증가, 과도한 부하(Overhead)

Question?

Seonah Lee
saleese@gmail.com