20~24장: 아키텍처

20장. 업무 규칙

핵심 규칙과 핵심 데이터는 본질적으로 결합되어 있기 때문에 객체로 만들 좋은 후보가 된다.

이러한 유형의 객체를 **엔티티(Entity)**라고 부른다.

📍 엔티티

- 컴퓨터 시스템 내부의 객체
- 핵심 업무 데이터를 기반으로 동작하는 일련의 조그만 핵심 업무 규칙을 구체화
- 업무 데이터를 직접 포함하거나 핵심 업무 데이터에 쉽게 접근 가능

엔티티 생성 시에는, 핵심 업무 데이터와 핵심 업무 규칙을 하나로 묶어서 별도의 소프트웨어 모듈로 만든다.

📍 유스케이스

- 자동화된 시스템이 사용되는 방법을 설명
    - 수동 환경에서는 사용 불가
- 유스케이스는 사용자가 제공해야 하는 입력, 사용자에게 보여줄 출력, 그리고 해당 출력을 생성하기 위한 처리 단계를 기술
- 애플리케이션에 특화된(application-specific) 업무 규칙을 설명
- 엔티티 내부의 핵심 업무 규칙을 어떻게, 그리고 언제 호출 할지 명시하는 규칙을 담는다.
- 사용자 인터페이스를 기술하지 않는다.
- 시스템이 사용자에게 어떻게 보이는지 설명하지 않는다.
- 유스케이스는 객체다.
    - 애플리케이션에 특화된 업무 규칙을 구현하는 하나 이상의 함수를 제공한다.

왜 엔티티는 고수준이고 유스케이스는 저수준일까?

- 유스케이스는 단일 애플리케이션에 특화되어 있어, 해당 시스템의 입력과 출력에 가깝게 위치하기 때문
- 엔티티는 수많은 다양한 애플리케이션에서 사용될 수 있도록 일반화된 것이므로, 각 시스템의 입력이나 출력에서 더 멀리 떨어져 있음

유스케이스는 엔티티에 의존하지만 엔티티는 유스케이스에 의존하지 않음

📍 요청 및 응답 모델

유스케이스는 입력 데이터를 받아서 출력 데이터를 생성한다.

- 유스케이스 클래스의 코드가 HTML이나 SQL에 대해 알게하면 안된다.
- 의존성을 제거하는 것은 매우 중요하다.
- 요청 및 응답 모델이 독립적이어야 한다.
    - 두 객체의 목적이 완전히 다르기 때문에 추후 다른 이유로 변경될 수 있기 때문이다.
    - 위반할 경우 공통 폐쇄 원칙과 단일 책임 원칙을 위배

결론

- 업무 규칙은 소프트웨어 시스템이 존재하는 이유다.
- 업무 규칙은 UI나 데이터베이스 같은 저수준의 관심사로 인해 오염되어서는 안되며, 원래 그대로 모습을 유지해야 한다.
- 업무 규칙을 표현하는 코드는 반드시 시스템의 심장부에 위치해야 하며, 덜 중요한 코드는 이 심장부에 플러그인 되어야 한다.
- 업무 규칙은 시스템에서 가장 독립적이며 가장 많이 재사용할 수 있는 코드여야 한다.

21장. 소리치는 아키텍처

📍 아키텍처의 테마

이바 야콥슨(ivar Jacobson)이 소프트웨어 아키텍처에 대해 쓴 저서 《Object Oriented Software Engineering》을 읽어보자.

- 이 책의 부제가 유스케이스 주도 접근법(Use Case Driven Approach)이라는 점을 주목하자.
- 야콥슨은 소프트웨어 아키텍처는 시스템의 유스케이스를 지원하는 구조라고 지적했다.

아키텍처는 프레임워크에 대한 것이 아니다.

- 프레임워크는 사용하는 도구일 뿐, 아키텍처가 준수해야 할 대상이 아니다.
- 아키텍처를 프레임워크 중심으로 만들어버리면 유스케이스가 중심이 되는 아키텍처는 절대 나올 수 없다.

좋은 아키텍처는 유스케이스를 그 중심에 두기 때문에, 프레임워크나 도구, 환경에 전혀 구애받지 않고 유스케이스를 지원하는 구조를 아무런 문제 없이 기술할 수 있다.

📍 아키텍처의 목적

좋은 소프트웨어 아키텍처는 개발 환경 문제나 도구에 대해서는 결정을 미룰 수 있도록 만든다.

- 프레임워크는 열어둬야할 선택사항이다.

좋은 아키텍처는 **유스케이스에 중점**을 두며, **지엽적인 관심사에 대한 결합을 분리**시킨다.

웹은 전달 메커니즘(입출력 장치)이며, 애플리케이션 아키텍처에서도 그와 같이 다뤄야 한다.애플리케이션이 웹을 통해 전달된다는 사실은 세부사항이며, 시스템 구조를 지배해서는 절대 안 된다.

📍 하지만 웹은?

근본적인 아키텍처를 고치지 않아도 시스템은 어떤 플랫폼을 통해서라도 전달될 수 있어야 한다.

프레임워크는 도구일 뿐, 삶의 방식은 아니다

어떻게 하면 유스케이스에 중점을 둔 채 그대로 보존할 수 있는지를 생각하라.

프레임워크가 아키텍처의 중심을 차지하는 일을 막을 수 있는 전략을 개발하라.

📍 테스트하기 쉬운 아키텍처

- 아키텍처가 유스케이스를 최우선으로 하고 프레임워크와 적당한 거리를 둔다면, 필요한 유스케이스 전부에 대해 단위 테스트를 할 수 있어야 한다.
- 엔티티 객체는 반드시 오래된 방식의 간단한 객체(plain old object)여야 한다.

- 프레임워크, 데이터베이스, 등 복잡한 것들에 의존해서는 안 된다.
- 유스케이스가 엔티티 객체를 조작해야 한다.
- 프레임워크로 인한 어려움을 겪지 않고도 반드시 있는 그대로 테스트 할 수 있어야 한다.

결론

아키텍처는 시스템을 이야기해야 하며, 시스템에 적용한 프레임워크에 대해 이야기해서는 안 된다.

- 세부사항을 당장 고려하지 않은 상태에도 시스템이 동작할 수 있어야 한다.

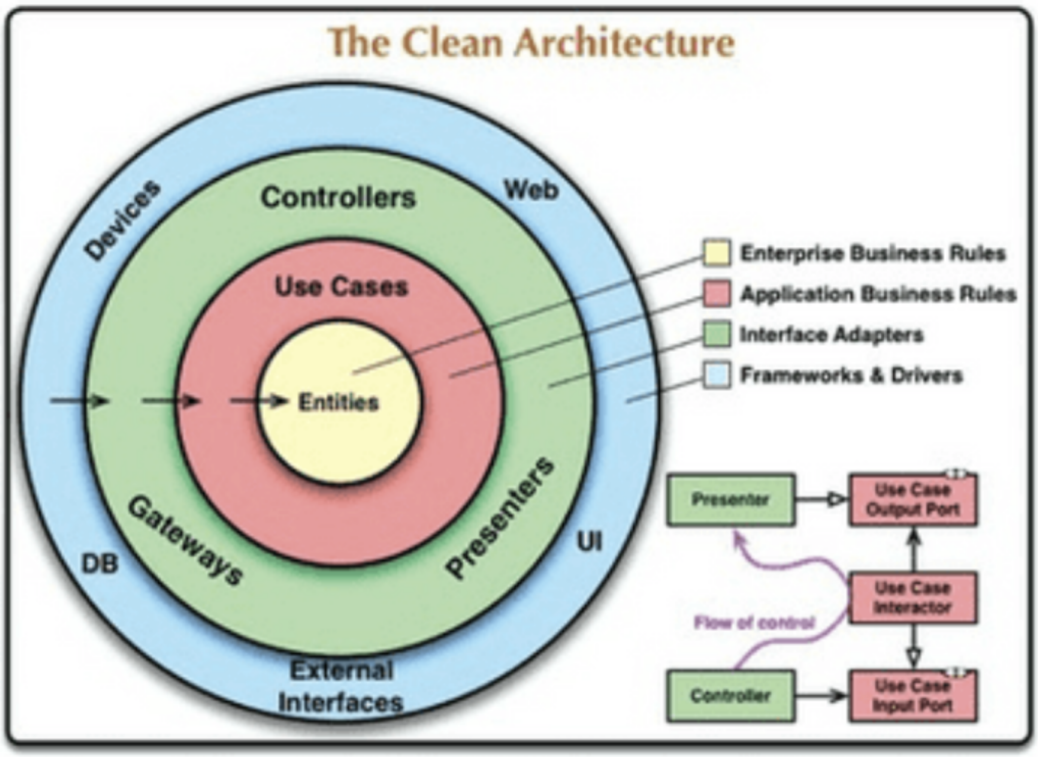
22장. 클린 아키텍처

다양한 아키텍처의 공통적인 목표는 관심사의 분리(DI)이다.

- 계층으로 분리하여 관심사를 분리한다.
- 각 아키텍처는 최소한 업무 규칙을 위한 계층 하나와, 사용자와 시스템 인터페이스를 위한 또 다른 계층 하나를 반드시 포함한다.

아키텍처들은 모두 시스템이 다음과 같은 특징을 지니도록 만든다.

1. 프레임워크 독립성 아키텍처는 프레임워크의 존재 여부에 의존하지 않는다. 이를 통해 프레임워크가 지닌 제약사항 안으로 시스템을 집어넣는게 아니라, 프레임워크를 도구로 사용할 수 있다.
2. 테스트 용이성업무 규칙은 UI, 데이터베이스, 웹 서버, 외부 요소 없이 테스트할 수 있다.
3. UI 독립성시스템의 나머지 부분을 변경하지 않고도 UI를 쉽게 변경할 수 있다. 예를 들어 업무 규칙을 변경하지 않고도 웹 UI를 콘솔 UI로 대체할 수 있다.
4. 데이터베이스 독립성 오라클이나 MS SQL서버를 몽고DB(MongoDB), 빅테이블(BigTable), 카우치DB(CauchDB) 등으로 교체할 수 있다. 업무 규칙은 데이터베이스에 결합되지 않는다.
5. 모든 외부 에이전시에 대한 독립성실제로 업무 규칙은 외부 세계와의 인터페이스에 대해 전혀 알지 못한다.



- 위 다이어그램은 아키텍처들 전부를 실행 가능한 하나의 아이디어로 통합하려는 시도다.

의존성 규칙

위 그림에서

- 안으로 향할수록 고수준의 소프트웨어가 된다.
- 바깥쪽 원은 메커니즘이고, 안쪽 원은 정책이다.

이러한 아키텍처가 동작하도록 하는 가장 중요한 규칙은 **의존성 규칙**이다.

소스 코드 의존성은 반드시 안쪽으로, 고수준의 정책을 향해야 한다.

- 내부의 원에 속한 요소는 외부의 원에 속한 어떤 것도 알지 못한다.
    - 함수, 클래스, 변수, 소프트웨어 엔티티로 명명되는 모든 것을 언급해선 안된다.
- 외부의 원에 선언된 데이터 형식도 내부의 원에서 절대로 사용해서는 안 된다.

📍 엔티티

- 전사적인 핵심 업무 규칙을 캡슐화
- 전사적이지 않고 단순한 애플리케이션을 작성한다면 엔티티는 해당 애플리케이션의 업무 객체가 된다.
- 운영 관점에서 특정 애플리케이션에 무언가 변경이 필요하더라도 엔티티 계층에는 절대로 영향을 주어서는 안 된다.

📍 유스케이스

- 애플리케이션에 특화된 업무 규칙을 포함하며 모든 유스케이스를 캡슐화하고 구현한다.
- 엔티티로 들어오고 나가는 데이터 흐름을 조정
- 엔티티가 자신의 핵심 업무 규칙을 사용해서 유스케이스의 목적을 달성하도록 이끈다.
- 이 계층에서 발생한 변경이 엔티티에 영향을 주면 안 되며, 외부 요소에서 발생한 변경이 이 계층에 영향을 주면 안된다.
    - 관심사로부터 격리되어있다.
    - 하지만 운영 관점에서 애플리케이션이 변경된다면 유스케이스가 영향을 받고, 따라서 이 계층의 소프트웨어에도 영향을 줄 것이다.

📍 인터페이스 어댑터

- 일련의 어댑터들로 구성
- 어댑터는 데이터를 유스케이스와 엔티티에게 가장 편리한 형식에서 데이터베이스나 웹 같은 외부 에이전시에게 가장 편리한 형식으로 변환
- 데이터를 엔티티와 유스케이스에게 가장 편리한 형식에서 영속성용으로 사용 중인 임의의 프레임워크가 이용하기에 가장 편리한 방식으로 변환
- 데이터를 외부 서비스와 같은 외부적인 형식에서 유스케이스나 엔티티에서 사용되는 내부적은 형식으로 변환하는 또 다른 어댑터가 필요

📍 프레임워크와 드라이버

- 데이터베이스나 웹 프레임워크 같은 프레임워크나 도구들로 구성
- 일반적으로 이 계층에는 안쪽원과 통신하기 위한 코드 외에 특별히 작성할 코드가 많지 않다.
- 세부사항이 위치하는 곳

- 외부에 위치시켜 피해를 최소화

원은 4개여야만 하나?

- 그럴필요는 없지만 항상 의존성 규칙은 적용된다.
- 소스코드 의존성은 안쪽을 향한다.
    - 안쪽으로 갈수록 추상화와 정책의 수준은 높아진다.
    - 가장 바깥쪽 원은 저수준의 구체적인 세부사항으로 구성된다.
    - 가장 안쪽 원은 가장 범용적이며 높은 수준을 가진다.

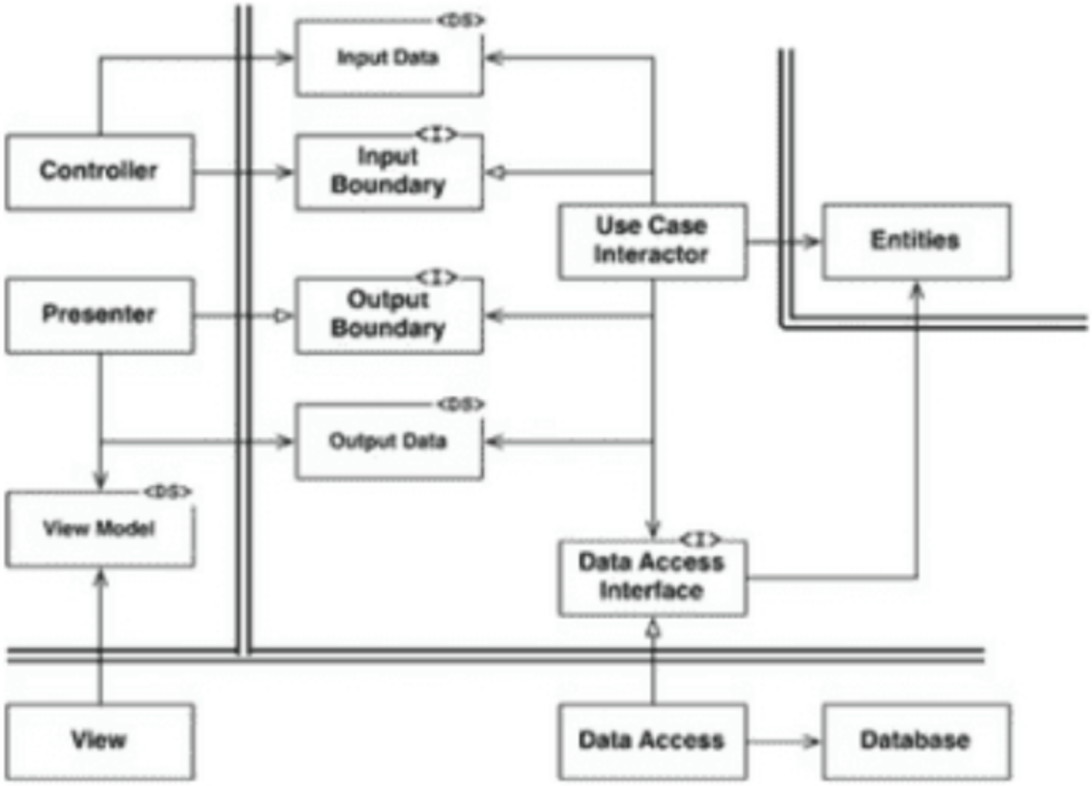
📍 경계 횡단하기

- 제어흐름과 의존성의 방향이 명백히 반대여야 하는 경우, 보통 의존성 역전 원칙을 사용하여 해결
- 아키텍처 경계를 횡단할 때도같은 기법을 사용할 수 있다.
    - 동적 다형성을 이용하여 소스 코드 의존성을 제어흐름과 반대로 만들 수 있고, 이를 통해 제어흐름이 어느 방향으로 흐르든 상관 없이 의존성 규칙을 준수할 수 있다.

경계를 횡단하는 데이터는 어떤 모습인가

- 경계를 가로지르는 데이터는 흔히 간단한 데이터 구조로 이루어져 있다.
- 격리되어 있는 간단한 데이터 구조가 경계를 가로질러 전달된다는 사실이다.
- 데이터 구조가 어떤 의존성을 가져 의존성 규칙을 위배하게 되는 일은 바라지 않는다.
- 따라서 경계를 가로질러 데이터를 전달할 때, 데이터는 항상 내부의 원에서 사용하기에 가장 편리한 형태를 가져야만 한다.

전형적인 시나리오



의존성 방향에 주목하라. 모든 의존성은 경계선을 안쪽으로 가로지르며, 따라서 의존성 규칙을 수행한다.

결론

소프트웨어를 계층으로 분리하고 의존성 규칙을 준수하면 본질적으로 테스트하기 쉬운 시스템을 만들게 되며, 시스템의 외부 요소가 구식이 되더라도 쉽게 교체할 수 있다.

23장. 프레젠터와 험블 객체

📍 험블 객체 패턴

- 테스트하기 어려운 행위와 테스트하기 쉬운 행위를 분리하기 쉽게 하는 목적으로 고안
- 행위를 두개의 모듈이나 클래스로 나눈다. 이중 하나가 험블이다.
    - 가장 기본적인 본질은 남기고, 테스트 하기 어려운 행위를 모두 험블 객체로 옮긴다.

    - 나머지 모듈에는 험블 객체에 속하지 않은, 테스트하기 쉬운 행위를 모두 옮긴다.
- 험블 객체 패턴을 사용하면 두 부류의 행위를 프레젠터와 뷰로 분리할 수 있다.

📍 프레젠터와 뷰

뷰는 험블객체이고 테스트하기 어렵다.

- 뷰는 데이터를 GUI로 이동시키지만, 데이터를 직접 처리하지 않는다.

프레젠터는 테스트하기 쉬운 객체다.

- 프레젠터의 역할은 애플리케이션(server)으로부터 데이터를 받아 화면에 표현할 수 있는 포맷으로 만드는 것이다.

따라서 뷰는 데이터를 건들지 않고 화면으로 전달하기만 하면 된다.뷰는 뷰 모델의 데이터를 화면으로 로드할 뿐, 아무 역할이 없다. : **Humble**

📍 테스트와 아키텍처

좋은 아키텍처는 테스트하기 용이해야 한다.

📍 데이터베이스 게이트웨이

유스케이스 인터랙터와 데이터베이스 사이에는 데이터베이스 게이트웨이가 존재한다.

- 다형적 인터페이스로, 애플리케이션이 데이터베이스에 수행하는 CRUD 작업과 관련된 모든 메서드를 포함한다.

유스케이스 계층은 SQL을 허용하지 않는다.

- 유스케이스 계층은 필요한 메서드를 제공하는 게이트웨이 인터페이스를 호출한다.
    - 인터페이스의 구현체(험블 객체)는 데이터베이스 계층에 위치한다.
    - 구현체에서 직접 SQL을 사용하거나 데이터베이스에 대한 임의의 인터페이스를 통해 게이트웨이의 메서드에 필요한 데이터에 접근한다.

인터랙터는 애플리케이션에 특화된 업무 규칙을 캡슐화 하기 때문에 험블 객체가 아니다.

- 테스트하기 쉽다.
- 게이트웨이는 스텁(stub), 모의(mock)나 테스트 더블(test-double)로 적당히 교체할 수 있기 때문이다.

📍 데이터 매퍼

객체 관계 매퍼(Object Relational Mapper, ORM) 같은 건 존재하지 않는다고 한다.

- 사용자 입장에서 볼 때 단순히 오퍼레이션의 집합이다.

객체와 달리 데이터 구조는 함축된 행위를 가지지 않는 public 데이터 변수의 집합이다.

이러한 ORM 시스템은 어디에 위치해야 하는가?

- 데이터베이스 계층이다.
- ORM은 게이트웨이 인터페이스와 데이터베이스 사이에서 일종의 또 다른 험블 객체 경계를 형성한다.

📍 서비스 리스너

애플리케이션이 다른 서비스(server)와 통신해야 하거나 일련의 서비스를 제공해야 한다면,

서비스 리스너(service listener)가 서비스 인터페이스로부터 데이터를 수신하고, 데이터를 애플리케이션에서 사용할 수 있게 간단한 데이터 구조로 포맷을 변경한다.

그 후 이 데이터 구조는 서비스 경계를 가로질러서 내부로 전달된다.

결론

- 각 아키텍처 경계마다 경계 가까이 숨어 있는 험블 객체 패턴을 발견할 수 있다.
- 경계를 넘나드는 통신은 거의 다 간단한 데이터 구조를 수반할 때가 많고 그 경계는 테스트하기 어려운 무언가와 쉬운 무언가로 분리된다.
- 이렇게 아키텍처 경계에서 험블 객체 패턴을 사용하면 전체 시스템의 테스트 용이성을 크게 향상시킬 수 있다.

24장. 부분적 경계

아키텍처를 완벽히 만드는 데는 당연히 비용이 많이 든다. 애자일 커뮤니티에 속한 사람 중 많은 이는 선행적인 아키텍처 설계를 탐탁치 않게 여긴다.

하지만 아키텍트는 "어쩌면 필요할지도." 라는 생각이 들 수도 있기 때문에 만약 그렇다면 `부분적 경계`를 구현해볼 수 있음.

📍 마지막 단계를 건너뛰기

부분적 경계를 생성하는 방법 하나는 독립적으로 컴파일하고 배포할 수 있는 컴포넌트를 만들기 위한 작업은 모두 수행한 후, 단일 컴포넌트에 그대로 모아만 두는 것임.

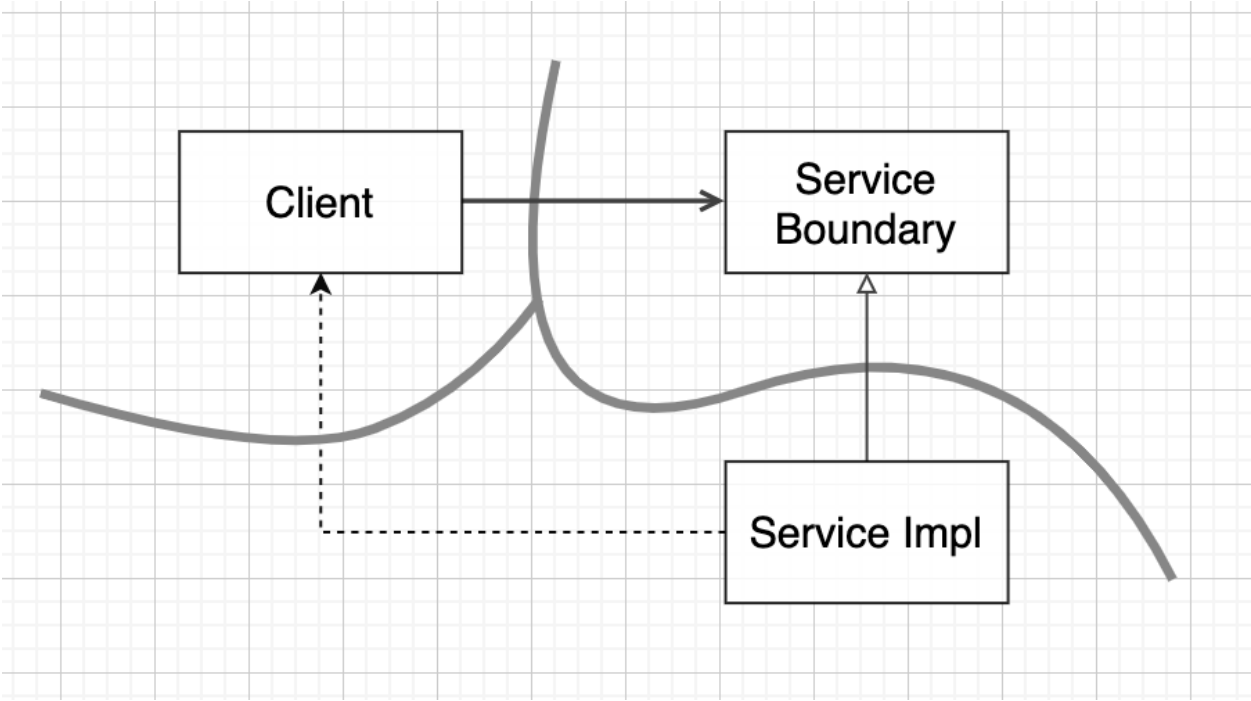
부분적 경계 전략을 기반으로 FitNess는 웹 서버 컴포넌트가 위키나 테스트 영역과는 분리되도록 설계했음. 새로운 웹 기반 애플리케이션을 만들 때 해당 웹 컴포넌트를 재사용할 수도 있다고 생각했기 때문. 그러나 시간이 흐르며, 별도로 분리한 웹 컴포넌트가 재사용 될 가능성은 전혀 없을 것임이 명백 해짐.

웹 컴포넌트와 위키 컴포넌트 사이의 구분도 약화되기 시작함.

📍 일차원 경계

완벽한 아키텍처 경계는 양방향으로 격리된 상태를 유지해야하므로 쌍방향 Boundary 인터페이스를 사용함.

그러나 비용이 너무 많이듦.

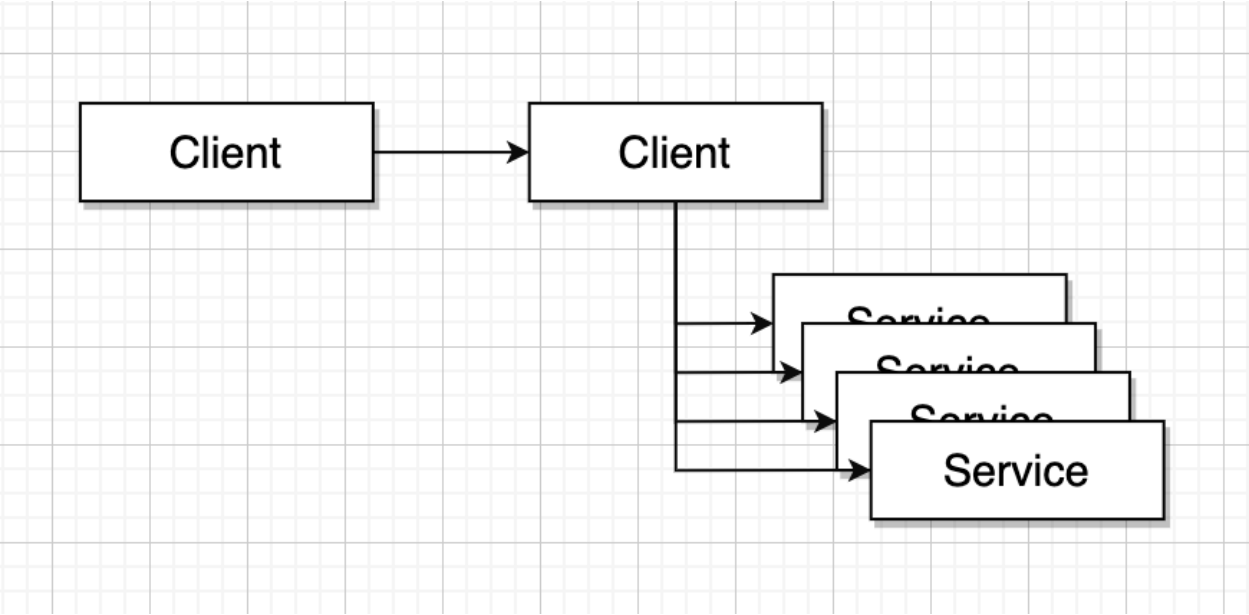


그림은 전통적인 전략 패턴임.

- Client는 Service Boundary를 사용하며 이는 ServiceImpl가 구현함.
- Client를 ServiceImpl로 부터 격리 시키고자 의존성 역전 원칙을 적용함.

그러나 쌍방향 인터페이스가 없고 개발자와 아키텍트가 제대로 훈련되어 있지 않다면, 전략 패턴은 위에 점선과 같은 비밀 통로가 생길 수 있음.

📍 퍼사드



훨씬 더 간단한 경계로써 모든 서비스 클래스를 메서드 형태로 정의하고 있는 Facade 클래스가 있음.

클라이언트는 서비스 클래스를 직접 접근할 수 없음.

그러나 정적 언어일 경우 클라이언트가 모든 서비스 클래스에 대해 추이 종속성을 가짐.

→ 서비스 클래스 하나가 변경되면 클라이언트도 무조건 재 컴파일 해야함.