5부. 아키텍처(하)

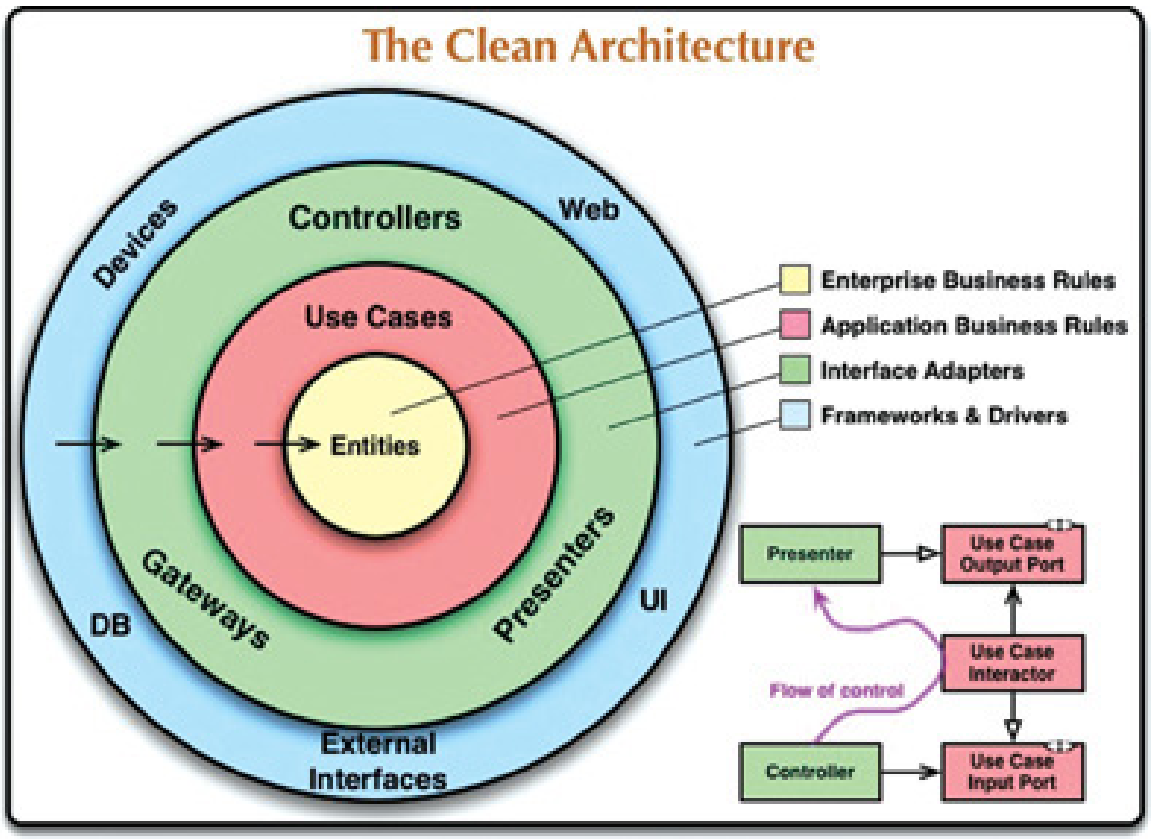
22장. 아키텍처

육각형 아키텍처, DCI(Data, Context and Intersection), BCE(Boundary-Control-Entity)의 공통적인 목표는 관심사의 분리(Separation of Concerns).

> **관심사의 분리란?**
> 컴퓨터 프로그램을 구별된 부분으로 **분리**시키는 디자인 원칙.

위 아키텍처는 모두 시스템이 다음과 같은 특징을 가지길 기대한다.

- 프레임워크 독립성: 아키텍처는 다양한 기능의 라이브러리를 제공하는 소프트웨어, 즉 프레임워크의 존재 여부에 의존하지 않는다.
- 테스트 용이성: 업무 규칙은 UI, 데이터베이스, 웹 서버, 또는 여타 외부 요소가 없이도 테스트할 수 있다.
- UI 독립성: 시스템의 나머지 부분을 변경하지 않고도 UI를 쉽게 변경할 수 있다.
- 데이터베이스 독립성: 오라클이나 MS SQL 서버를 몽고DB나 레디스로 교체할 수 있다.
- 모든 외부 에이전시에 대한 독립성: 실제로 **업무 규칙**은 외부 세계와의 인터페이스에 대해 **전혀 알지 못한다.**



의존성 규칙

각가의 동심원은 소프트웨어에서 서로 다른 영역을 표현한다. 보통 안으로 들어갈수록 고수준의 소프트웨어가 된다. 바깥쪽 원은 메커니즘이고 안쪽 원은 정책이다.

> *소스 코드 의존성은 반드시 안쪽으로, 고수준의 정책을 향해야 한다.*

내부에 원에 속한 요소는 외부의 원에 속한 어떤 것도 알지 못한다. 특히 내부의 원에 속한 코드는 외부의 원에 선언된 어떤 것에 대해서도 그 이름을 언급해서는 절대 안된다.

같은 이유로, 외부의 원에 선언된 데이터 형식도 내부의 원에서 절대로 사용해서는 안된다. 특히 그 데이터 형식이 외부의 원에 있는 프레임워크가 생성한 것이라면 더더욱 사용해서는 안된다. 우리는 외부 원에 위치한 어떤 것도 내부의 원에 영향을 주지 않기를 바란다.

엔티티

엔티티는 전사적인 핵심 업무 규칙을 캡슐화한다. 엔티티는 메서드를 가지는 객체이거나 일련의 데이터 구조와 함수의 집합일 수도 있다, 기업의 다양한 애플리케이션에서 엔티티를 재

사용할 수만 있다면, 그형태는 그다지 중요하지 않다.

전사적이지 않은 단순한 단일 애플리케이션을 작성하고 있다면 엔티티는 해당 어플리케이션의 업무 객체가 된다. 이 경우 엔티티는 가장 일반적이며 고수준의 규칙을 캡슐화한다. 외부의 무언가가 변경되더라도 엔티티가 변경될 가능성은 지극히 낮다.

유스케이스

유스케이스 계층의 소프트웨어는 애플리케이션에 특화된 업무 규칙을 포함한다. 또한 유스케이스 계층의 소프트웨어는시스템의 모든 유스케이스를 캡술화하고 구현한다.

엔티티가 자신의 핵심 업무 규칙을 사용해서 유스케이스의 목적을 달성하도록 이끈다.

이 계층에서 발생한 변경이 엔티티에 영향을 줘서는 안된다.

하지만 운영 관점에서 애플리케이션이 변경된다면 유스케이스의 영향을 받으며, 따라서 이 계층의 소프트웨어에도 영향을 줄 것이다. 유스케이스의 세부사항이 변하면 이 계층의 코드 일부는 분명 영향을 받을 것이다.

인터페이스 어댑터

인터페이스 어탭터 계층은 일련의 어댑터들로 구성된다. 어댑터는 데이터를 유스케이스와 엔티티에게 가장 편리한 형식에서 데이터베이스나 웹 같은 외부 에이전시에게 가장 편리한 형식으로 변환한다.

프레임워크와 드라이버

가장 바깥쪽 계층은 일반적으로 데이터베이스나 웹 프레임워크 같은 프레임워크나 도구들로 구성된다. 일반적으로 이 계층에서는 안쪽 원과 통신하기 위한 접합 코드 외에는 특별히 더 작성해야할 코드가 그다지 많지 않다.

프레임워크나 드라이버 계층은 모든 세부사항이 위치하는 곳이다. 웹은 세부사항이다. 데이터베이스는 세부사항이다. 우리는 이러한 것들은 모두 외부에 위치시켜서 피해를 최소화한다.

원은 네 개 여야만하나

아니다.

경계 횡단하기

우선 제어흐름에 주목해 보자. 제어흐름은 컨트롤러에서 시작해서, 유스케이스를 지난 후, 프레젠터에게 실행되면서 마무리한다. 다음으로 소스 코드 의존성도 주목해서 보자. 각 의존성은 유스케이스를 향해 안쪽을 가리킨다.

이처럼 제어흐름과 의존성의 방향이 명백히 반대여야 하는 경우, 대체로 DIP를 사용하여 해결한다.

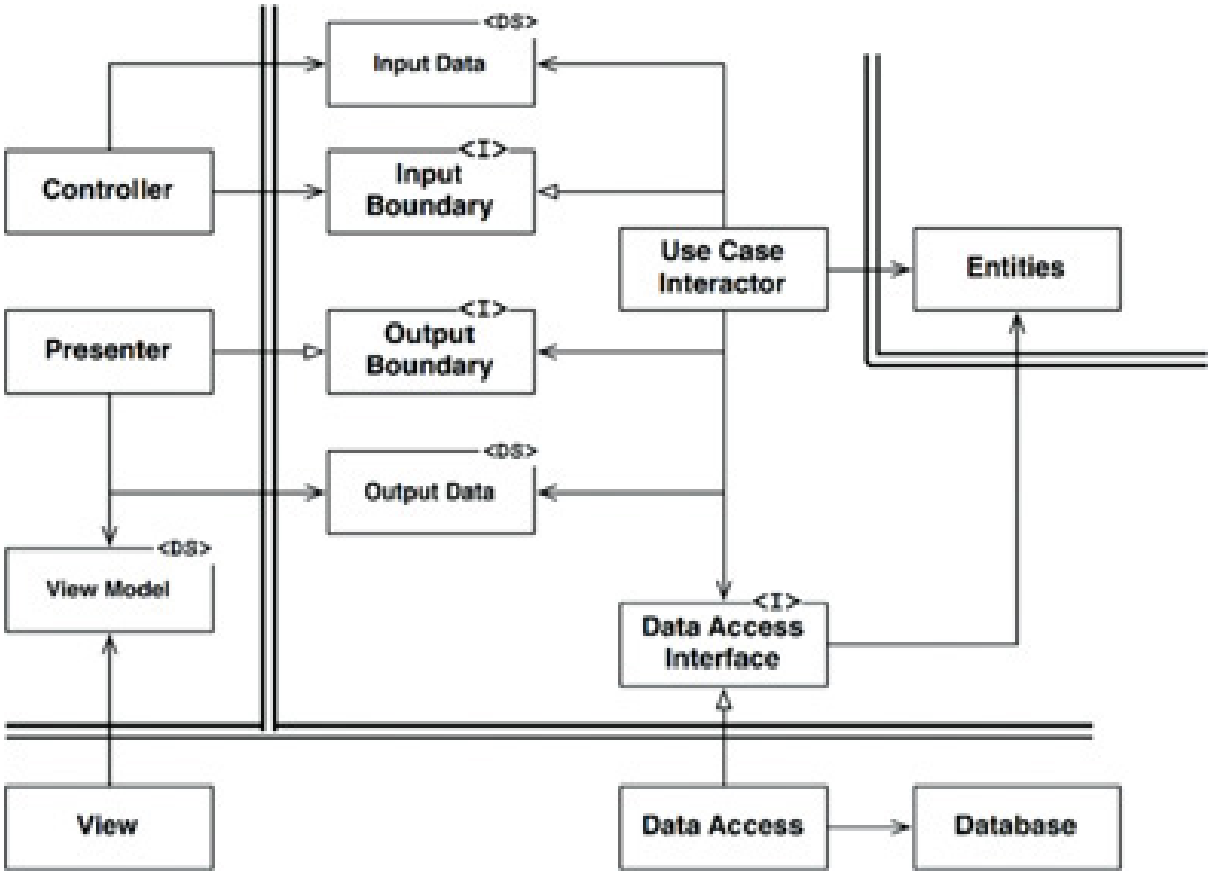
경계를 횡단하는 데이터는 어떤 모습인가

경계를 가로지르는 데이터는 흔히 간단한 데이터 구조로 이루어져 있다. 기본적인 구조체나 간단한 DTO 등 원하는 대로 고를 수 있다.

엔티티 객체나 데이터 로우 등을 전달해서는 안된다. 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.

예시



결론

이상의 간단한 규칙들은 준수하는 일은 어렵지 않으며, 향후에 겪을 수많은 고통거리를 덜어줄 것이다. 소프트웨어를 계층으로 분리하고 의존성 규칙을 준수한다면 본질적으로 테스트하기 쉬운 시스템을 만들게 될 것이며, 그에 따른 이점을 누릴 수 있다. 데이터베이스나 웹 프레임워크와 같은 시스템의 외부 요소가 구식이 되더라도, 이들 요소를 야단스럽지 않게 교체할 수 있다.

23장 프렌제터와 험블 객체

험블 객체 패턴

험블 객체 패턴은 디자인 패턴으로, 테스트하기 어려운 행위와 테스트하기 쉬운 행위를 단위 테스트 작성자가 분리하기 쉽게 하는 방법으로 고안되었다. 아이디어는 매우 단순하다. 행위들을 두 개의 모듈 또는 클래스로 나눈다. 이들 모듈 중 하나가 험블이다.

가장 기본적인 본질은 남기고, 테스트하기 어려운 행위를 모두 험블 객체로 옮긴다. 나머지 모듈에는 험블 객체에 속하지 않은, 테스트하기 쉬운 행위를 모두 옮긴다.

프레젠터와 뷰

뷰는 험블 객체이고 테스트하기 어렵다. 이 객체에 포함된 코드는 가능한 한 간단하게 유지한다.

프레젠터는 테스트하기 쉬운 객체다. 프레젠터의 역할은 애플리케이션으로부터 데이터를 받아 화면에 표현할 수 있는 포맷으로 만드는 것이다. 이를 통해 뷰는 데이터를 화면으로 전달하는 간단한 일만 처리하도록 만든다.

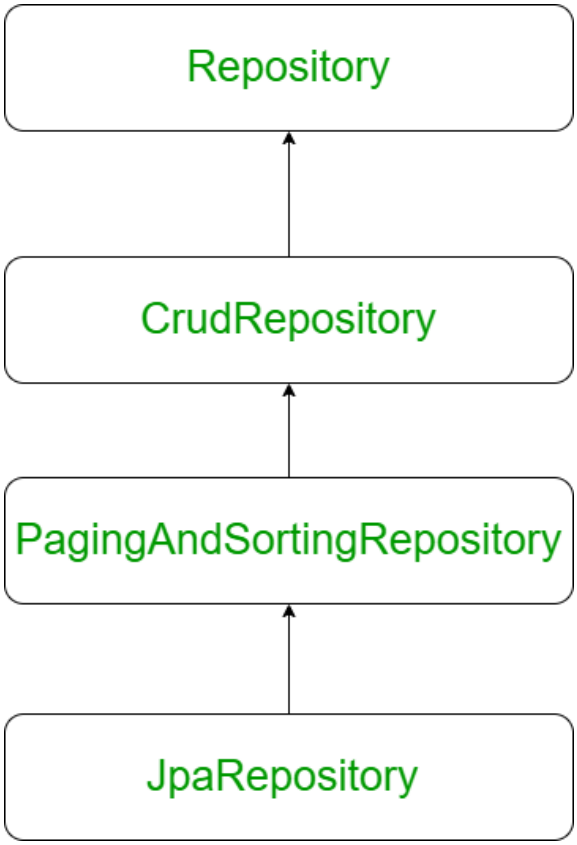
테스트와 아키텍처

테스트 용이성은 좋은 아키텍처가 지녀야 할 속성으로 오랫동안 알려져 왔다. 험블 객체 패턴이 좋은 예인데, 행위를 테스트하기 쉬운 부분과 테스트하기 어려운 부분으로 분리하면 아키텍처 경계가 정의되기 때문이다. 프레젠터와 뷰 사이의 경계는 이러한 경계의 중 하나이며, 이 밖에도 수 많은 경계가 존재한다.

데이터베이스 게이트웨이

유스케이스 인터랙터와 데이터베이스 사이에는 데이터베이스 게이트웨이가 위치한다. 이 게이트웨이는 다형적 인터페이스로, 애플리케이션이 데이터베이스에 수행하는 CRUD 작업과 관련된 모든 메소드를 포함한다.

다시 한번 말하지만 유스케이스 계층은 SQL을 허용하지 않는다. 따라서 유스케이스 계층에 필요한 메소드를 제공하는 게이트웨이 인터페이스를 호출한다.



JPA의 데이터베이스 게이트웨이

서비스 리스너

서비스 리스너는 외부로 부터 데이터를 수신하는 객체를 말하며, 수신된 데이터를 애플리케이션에서 사용할 수 있게 간단한 데이터 구조로 포맷을 변경하는 역할을 한다. 그 데이터 구조는 서비스 경계를 가로질러 내부로 전달된다.

결론

각 아키텍처 경계마다 경계 가까이 숨어 있는 험블 객체 패턴을 발견할 수 있을 것이다. 경계를 넘나드는 통신은 거의 모두 간단한 데이터 구조를 수반할 때가 많고 대개 그 경계는 테스트하기 어려운 무언가와 테스트하기 쉬운 무언가로 분리될 것이다. 그리고 이러한 아키텍처 경계에서 험블 객체 패턴을 사용하면 전체 시스템의 테스트 용이성을 크게 높일 수 있다.

24장. 부분적 경계

많은 경우에, 뛰어난 아키텍트라면 이러한 경계를 만드는 비용이 너무 크다고 판단하면서도, 한편으로는 나중에 필요할 수도 있으므로 이러한 경계에 필요한 공간을 확보하기 원할 수도 있다.

애자일 커뮤니티에서는 미래를 예견해서 경계를 긋는 행위에 대해 YAGNI(Yor Aren`t Going to Need it.)원칙을 위배하기 때문이다. 하지만 아키텍트라면 이 문제를 검토하면서 “그래, 하지만 어쩌면 필요 할지도,”라는 생각이 들 수도 있다. 만약 그렇다면 부분적 경계(Partial Boundary)를 구현해볼 수 있다.

마지막 단계를 건너뛰기

부분적 경계를 샐성하는 방법 하나는 독립적으로 컴파일하고 배포할 수 있는 컴포넌트를 만들기 위한 작업은 모두 수행한 후, 단일 컴포넌트에 그대로 모아만 두는 것이다. 쌍방향 인터페이스도 그 컴포넌트에 있고, 입력·출력 데이터 구조도 거기에 있으며, 모든 것이 완전히 준비되어 있다. 하지만 이 모두를 단일 컴포넌트로 컴파일해서 배포한다.

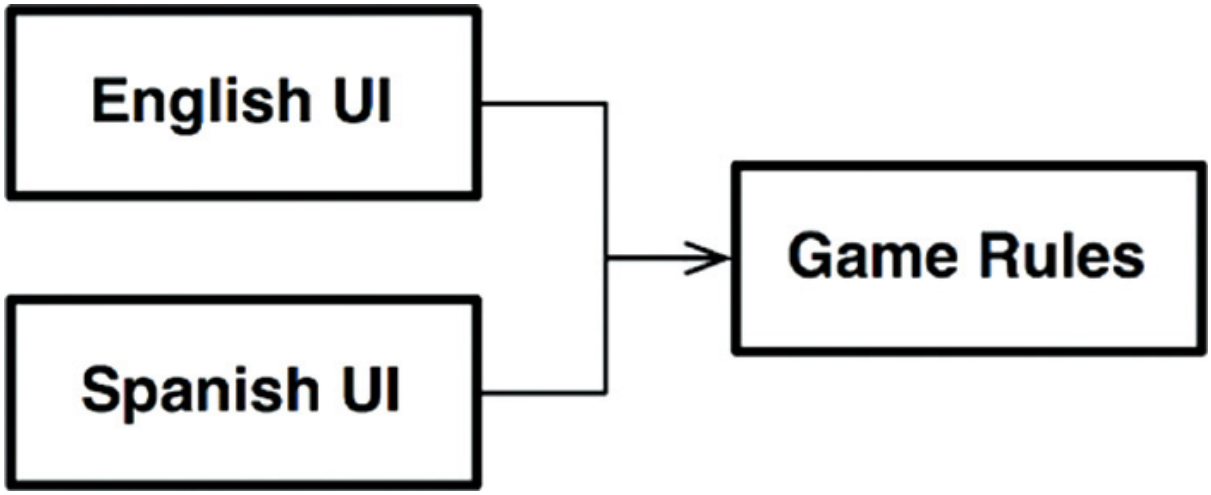
결론

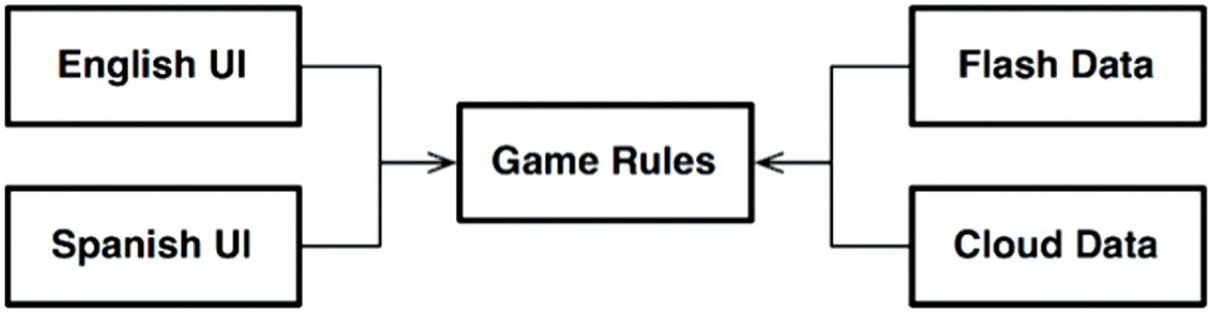
아키텍처 경계를 부분적으로 구현하는 방법에는 전략 패턴과 퍼사드 패턴이 있다.

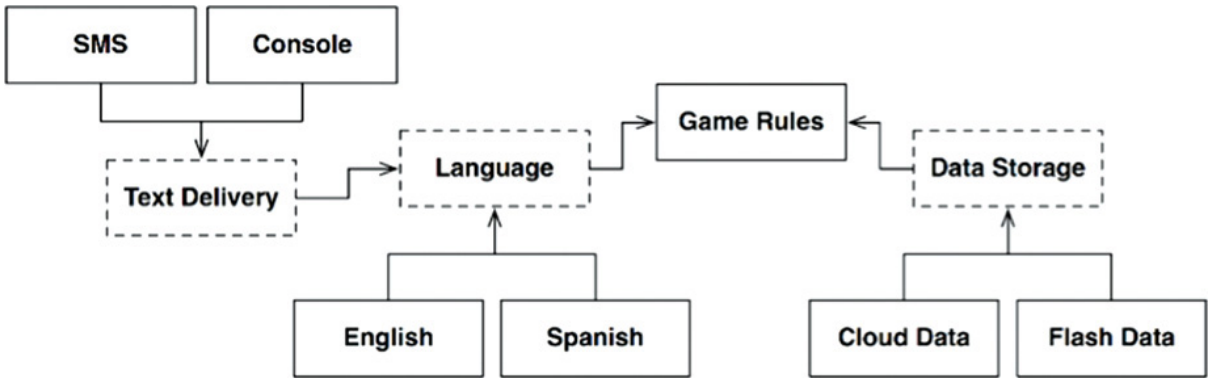
25장. 계층과 경계

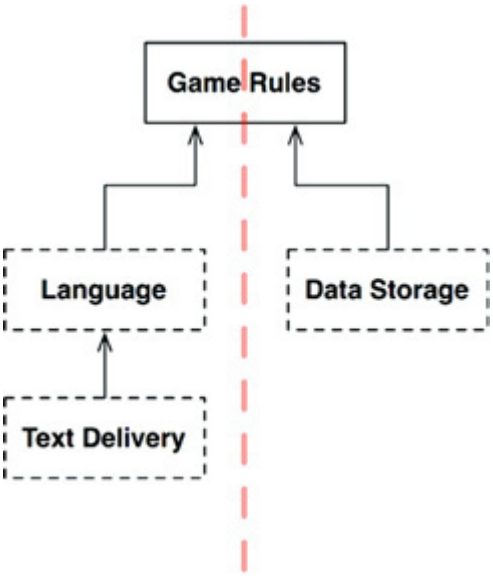
시스템이 세 가지 컴포넌트(UI, 엄무 규칙, 데이터베이스)로만 구성된다고 생각하기 쉽다. 몇몇 단순한 시스템에서는 이 정도로 충분하다. 하지만 대다수의 시스템에서 컴포넌트의 개수는 이보다 훨씬 많다.

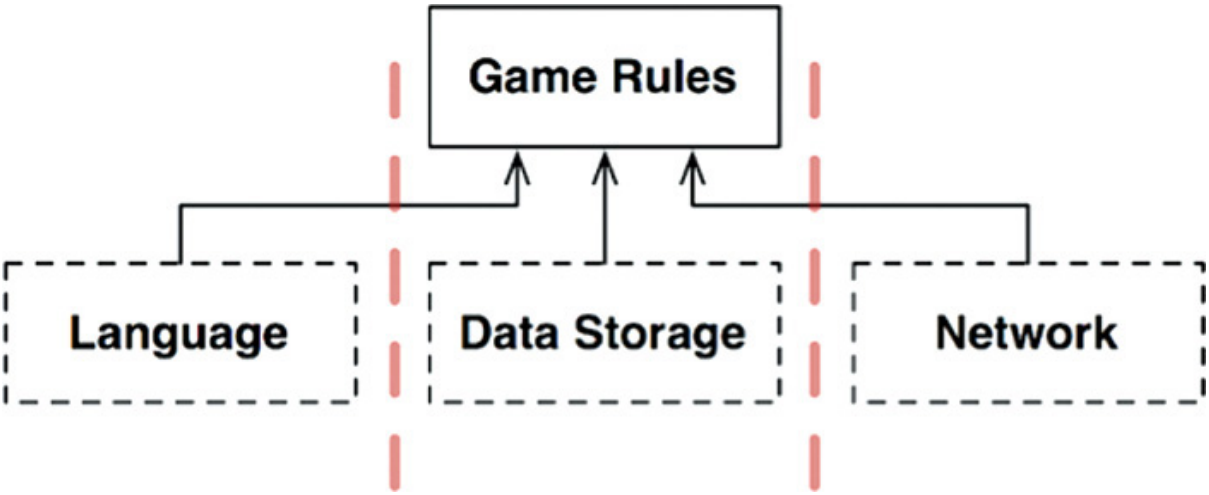
책에서는 움퍼스 사냥 게임이란 소프트웨어의 아키텍처를 설계하며 레이어 아키텍처를 넘어서서 계층과 경계를 긋는 방법에 대해 설명한다. 그 변화의 흐름은 아래와 같다.

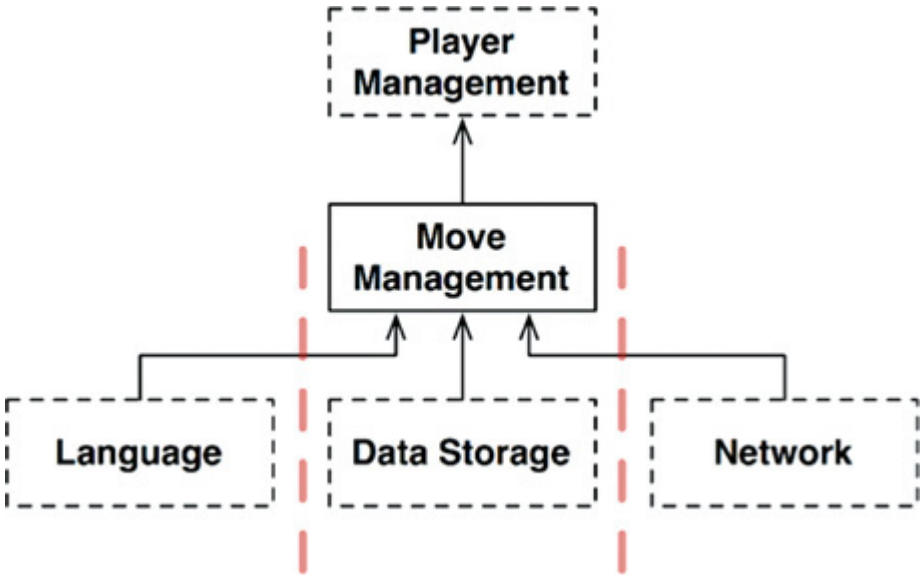

English UI
Spanish UI
Game Rules

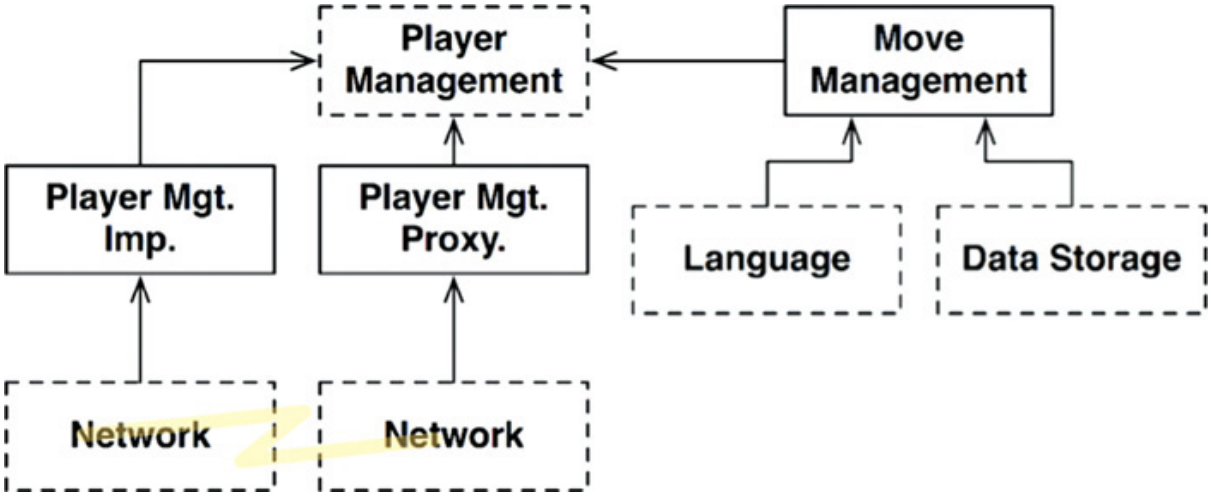

English UI
Spanish UI
Game Rules
Flash Data
Cloud Data


SMS
Console
Text Delivery
Language
Game Rules
Data Storage
English
Spanish
Cloud Data
Flash Data


Game Rules
Language
Data Storage
Text Delivery


Game Rules
Language
Data Storage
Network





결론

이 예시를 통해 아키텍처 경계가 어디에나 존재한다는 사실을 알 수 있다. 아키텍처로서 우리는 아키텍처 경계가 언제 필요한지를 신중하게 파악해내야 한다. 또한 우리는 이러한 경계를 제대로 구현하려면 비용이 많이 든다는 사실도 인지하고 있어야 한다.

YAGNI가 말하는 철학은 “추상화가 필요하리라고 미리 예측해서는 안된다” 이다. 이 문구에는 지혜가 담겨 있는데, 오버 엔지니어링이 언더 엔지니어링보다 나쁠 때가 훨씬 많기 때문이다. 다른 한편으로는 어떤 아키텍처 경계도 존재하지 않는 상황에서 경계가 정말로 필요하

다는 사실을 발견한 경우, 그때서야 경계를 추가하려면 비용이 많이 들고 큰 위험을 감수해야 한다.

소프트웨어 아키텍트는 이러한 미래를 내다보고 현명하게 판단해야 한다. 비용을 산정하고 아키텍처의 경계를 어디에 둬야할지, 완벽하게 경계를 구현할 것과 부분적으로 구현할 것, 그리고 무시할 것에 대해 현명하게 결정해야 한다.

이건 일회적인 판단으로 끝날 수 없으며 지속적인 관찰을 통해 결정 해야 한다.

26장 메인(Main) 컴포넌트

메인 컴포넌트

모든 시스템에는 최소한 하나의 컴포넌트가 존재하고, 이 컴포넌트가 나머지 컴포넌트를 생성하고, 조정하며, 관리한다. 나는 이 컴포넌트를 메인 이라고 부른다.

궁극적인 세부사항

메인 컴포넌트는 궁극적인 세부사항으로, 가장 낮은 수준의 정책이다. 메인은 시스템의 초기 진입점이다. 운영체제를 제외하면 어떤 것도 메인에 의존하지 않는다. 메인은 모든 팩토리와 전략 그리고 시스템 전반을 담당하는 나머지 기반 설비를 생성한 후, 시스템에서 더 높은 수준을 담당하는 부분으로 제어권을 넘기는 역할을 맡는다.

메인을 지저분한 컴포넌트 중에서도 가장 지저분한 컴포넌트라고 생각하자.

요지는 메인은 클린 아키텍처에서 가장 바깥 원에 위치하는, 지저분한 저수준 모듈이라는 점이다. 메인은 고수준의 시스템을 위한 모든 것을 로드한 후 제어권을 고수준의 시스템에게 넘긴다.

결론

메인은 애플리케이션의 플러그인이라고 생각하자. 메인은 초기 조건과 설정을 구성하고, 외부 자원을 모두 수집한 후, 제어권을 애플리케이션의 고수준 정책으로 넘기는 플러그인이다.

메인은 플러그인이므로 메인 컴포넌트를 애플리케이션의 설정 별로 하나씩 두도록 하여 둘 이상의 메인 컴포넌트를 만들 수도 있다.

27장. ‘크고 작은 모든’ 서비스들

SOA와 MSA의 인기의 배경에는 다음과 같은 사실(?)이 있다.

- 서비스를 사용하면 **상호 결합이 철저하게 분리**되는 것처럼 보인다. 나중에 보겠지만, 이는 **일부만 맞는 말이다.**
- 서비스를 사용하면 **개발과 배포 독립성**을 지원하는 것처럼 보인다. 나중에 보겠지만,. 이 역시도 **일부만 맞는 말이다.**

서비스 아키텍처?

시스템의 아키텍처는 **의존성 규칙을 준수하며 고수준의 정책을 저수준의 세부사항으로 분리하는 경계에 의해 정의**된다. 단순히 애플리케이션의 행위를 분리할 뿐인 서비스라면 값비싼 함수 호출에 불과하며, 아키텍처 관점에서 꼭 중요하다고 볼 수는 없다.

- **저수준의 세부사항은?** OS, 서버, 컨테이너, 디비 등 여러가지

서비스의 이점?

결합 분리의 오류

각 서비스는 서로 다른 프로세스에서, 심지어는 서로 다른 프로세서에서 실행된다. 따라서 서비스는 다른 서비스의 변수에 직접 접근할 수 없다. 그리고 모든 서비스의 인터페이스는 반드시 잘 정의되어 있어야 한다.

이 말에는 어느 정도 일리가 있지만, 꼭 그런 것만은 아니다. 물론 서비스는 개별 변수 수준에서는 각각 결합이 분리된다. 하지만 프로세서 내의 또는 **네트워크 상의 공유 자원** 때문에 결합될 가능성이 여전히 존재한다. 더욱이 서로 **공유하는 데이터**에 의해 이들 서비스는 **결합**되어 버린다.

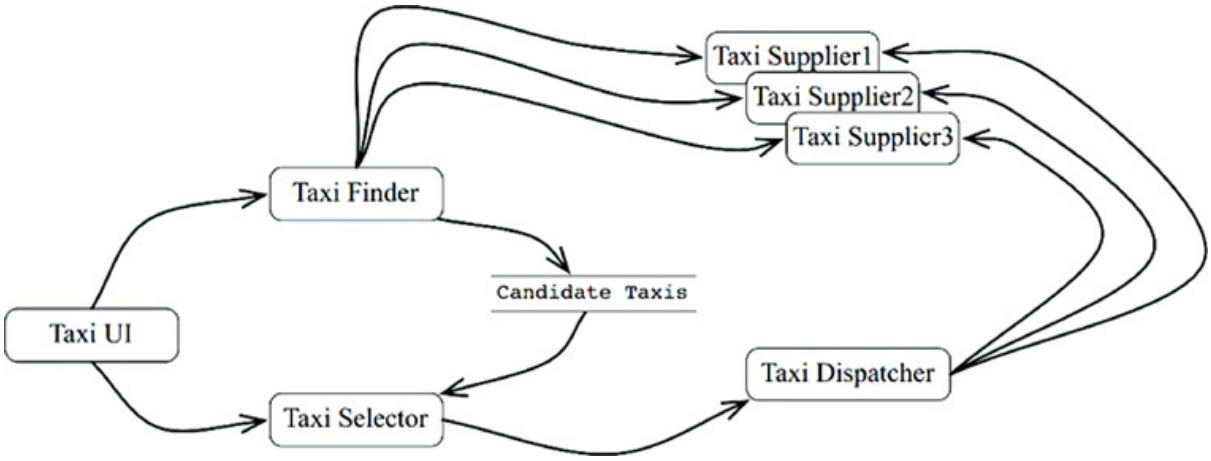
개발 및 배포 독립성의 오류

서비스를 사용함에 따라서 예측되는 또 다른 이점은 전담팀이 서비스를 소유하고 운영한다는 점이다. 그래서 데브옵스 전략의 일환으로 전담 팀에서 각 서비스를 작성하고, 유지보수하며, 운영하는 책임을 질 수 있다.

이러한 개발 및 배포 독립성은 확장 가능한 것으로 간주된다. 대규모 엔터프라이즈 시스템을 독립적으로 개발하고 배포 가능한 수십, 수백, 수천 개의 서비스들을 이용하여 만들 수 있다고 믿는다. 시스템의 개발, 유지보수, 운영 또한 비슷한 수의 독립적인 팀 단위로 분할할 수 있다고 여긴다.

다 맞지는 않다. 첫째로, 오랜 역사를 통해 증명된 바에 따르면 서비스는 확장 가능한 시스템을 구축하는 유일한 선택지가 아니다. 둘째로, 결합 분리의 오류에 따르면 서비스라고 해서 항상 독립적으로 개발하고, 배포하며, 운영할 수 있는 것은 아니다. 데이터나 행위에서 어느 정도 결합되어 있다면 결합된 정도에 맞게 개발, 배포, 운영을 조정해야만 한다.

야옹이 문제

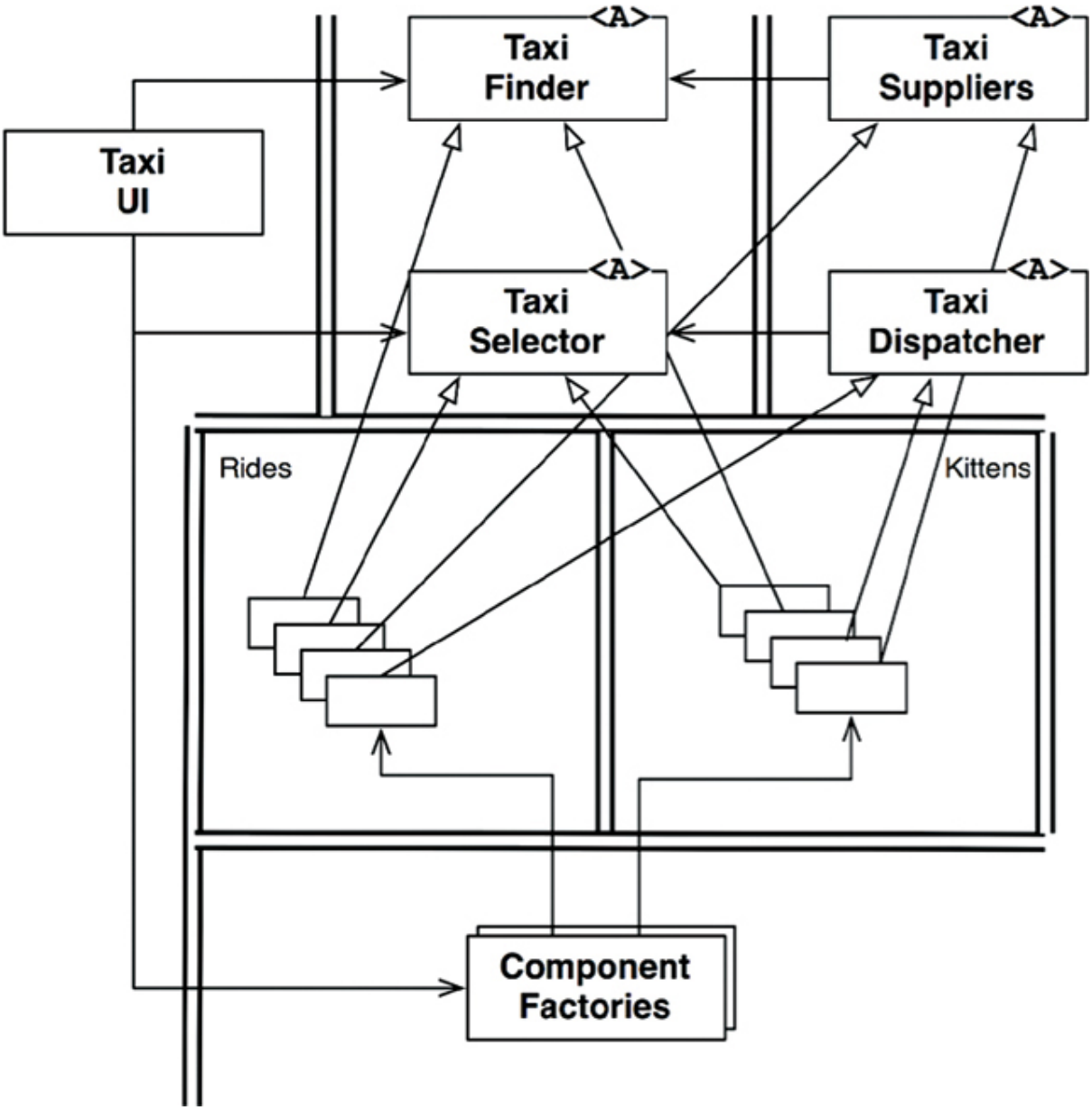


책에서는 택시를 찾고 배정해주는 프로그램을 예시로 들었다. 서비스 단위로 분리되어 있어 결합 되어 있지 않고 개발과 배포에 독립적일 것으로 예상 되었다.

하지만 현실은 이 프로그램에 야옹이 배달 기능이 추가되면서 모든 서비스가 결합되어 있다는 사실과 독립적으로 개발하거나 유지 될 수 없다는 것을 알게 되었다.

이게 바로 횡단 관심사(cross-cutting concern)가 지닌 문제다. 모든 소프트웨어 시스템은 서비스 지향이든 아니든 이 문제에 직면하게 마련이다. 위와 같은 서비스 다이어그램에서 묘사된 것과 같은 종류의 기능적 분해는 새로운 기능이 기능적 행위를 횡단하는 상황에 매우 취약하다.

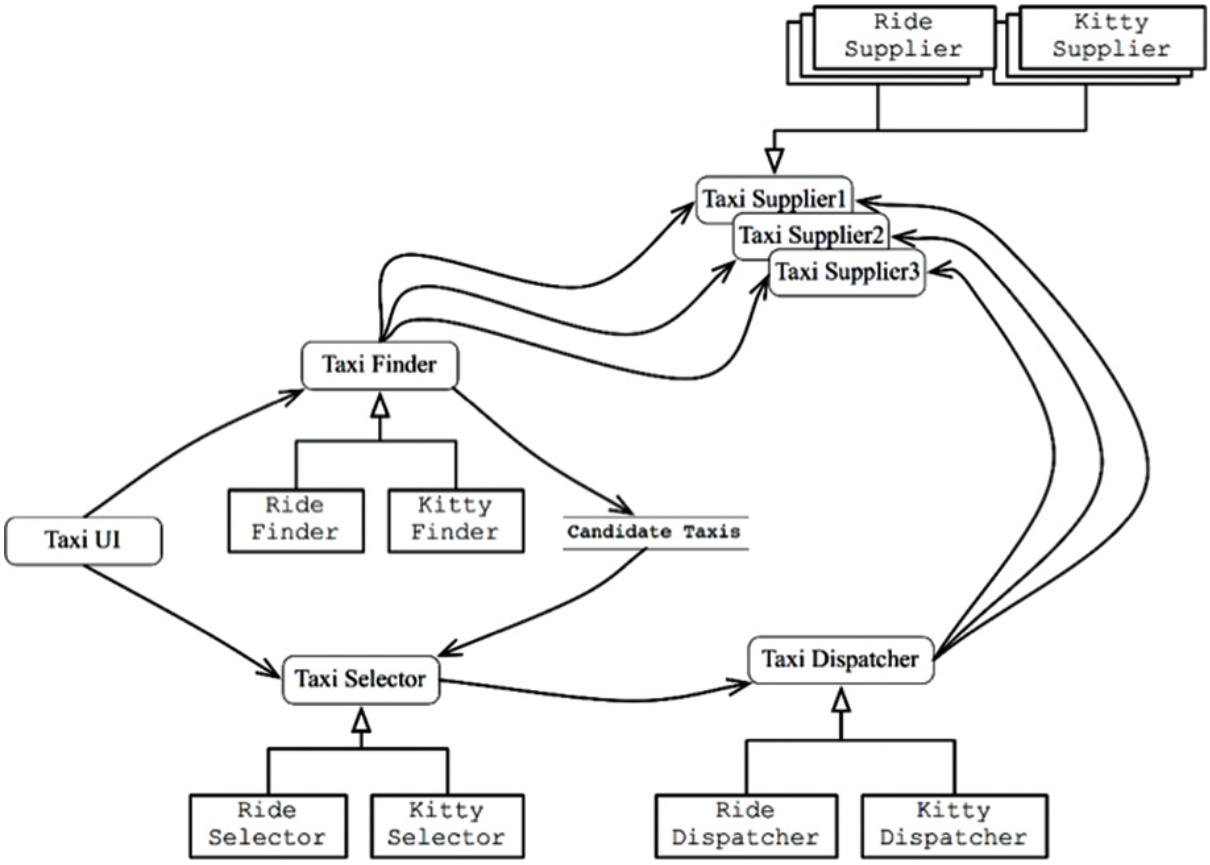
객체가 구출하다



이러한 기능 추가에 대응하기 위해 기존의 배차 기능은 **Rides 컴포넌트**로 추출하고, 야옹이에 대한 신규 기능은 **Kittens 컴포넌트**에 들어갔다. 이 두 컴포넌트는 기존 컴포넌트들에 있는 추상 기반 클래스를 템플릿 메소드나 전략 패턴 등을 이용해서 오버라이드 한다. 두 개

의 신규 컴포넌트는 의존성 규칙을 준수하고 있으며, 동시에 UI의 제어 하에 팩토리가 생성한다느 점도 주목하자.

결과적으로 배차와 관련된 기능과 야옹이에 대한 기능은 각각 변경되고 배포될 수 있게 되었다. 결합이 분리되고 독립적으로 개발하여 배포될 수 있게 된 것이다.



컴포넌트 기반 서비스

서비스는 SOLID 원칙대로 설계할 수 있으며 컴포넌트 구조를 갖출 수도 있다. 이를 통해 서비스 내의 기존 컴포넌트들을 변경하지 않고도 새로운 컴포넌트를 추가할 수 있다.

횡단 관심사

아키텍처 경계가 서비스 사이에 있지 않다는 사실이다. 오히려 서비스를 관통하며, 서비스를 컴포넌트 단위로 분할한다.

모든 주요 시스템이 직면하는 횡단 관심사를 처리하려면, 위 다이어그램에서 보듯이, 서비스 내부는 의존성 규칙도 준수하는 컴포넌트 아키텍처로 설계해야한다. 이 서비스들은 시스템의 아키텍처 경계를 정의하지 않는다. **아키텍처 경계를 정의하는 것은 서비스 내에 위치한 컴포넌트**다.

결론

서비스는 시스템의 확장성과 개발 가능성 측면에서 유용하지만, 그 자체로는 아키텍처적으로 그리 중요한 요소는 아니다. 시스템의 아키텍처는 시스템 내부에 그어진 경계와 경계를 넘나드는 의존성에 의해 정의된다. 시스템의 구성 요소가 통신하고 실행되는 물리적인 메커니즘에 의해 아키텍처가 정의되는 것이 아니다.

서비스는 단 하나의 아키텍처 경계로 둘러싸인 단일 컴포넌트로도 가능하고, 혹은 여러 아키텍처 경계로 분리된 다수의 컴포넌트로 구성할 수도 있다.

28장. 테스트 경계

테스트는 시스템의 일부이며, 아키텍처에 관여한다.

시스템 컴포넌트인 테스트

테스트는 태생적으로 의존성 규칙을 따른다. 테스트는 세부적이며 구체적인 것으로, 의존성은 항상 테스트 대상이 되는 코드를 향한다. 실제로 테스트는 아키텍처에서 가장 바깥쪽 원으로 생각할 수 있다. 시스템 내부의 어떤 것도 테스트에는 의존하지 않으며, 테스트는 시스템의 컴포넌트를 향해, 항상 원의 안쪽으로 의존한다.

테스트는 시스템 컴포넌트 중에서도 가장 고립되어 있다.

테스트는 다른 모든 시스템 컴포넌트가 반드시 지켜야 하는 모델을 표현해준다.

테스트를 고려한 설계

깨지기 쉬운 테스트는 시스템을 뻣뻣하게 만든다는 부작용을 낳을 때가 많다. 시스템에 가한 간단한 변경이 대량의 테스트 실패로 이어진다는 사실을 알게 되면, 개발자는 그러한 변경을 하지 않으려 들 것이다.

이 문제를 해결하려면 테스트를 고려해서 설계해야 한다. 소프트웨어 설계의 첫 번째 규칙은 언제나 같다. 변동성이 있는 것에 의존하지 마라.

테스트 API

테스트가 모든 업무 규칙을 검증하는데 사용할 수 있도록 특화된 API를 만들면 된다.

구조적 결합

구조적 결합은 테스트 결합 중에서 가장 강하며, 가장 은밀하게 퍼져 나가는 유형이다. 모든 상용 클래스에 테스트 클래스가 각각 존재하고, 또 모든 상용 메서드에 테스트 메서드 집합이 각각 존재하는 테스트 스위트가 있다고 가정해보자. 이러한 테스트 스위트는 애플리케이션 구조에 강하게 결합되어 있다.

테스트 API의 역할은 애플리케이션의 구조를 테스트로부터 숨기는데 있다. 이렇게 만들면 상용 코드를 리팩터링하거나 진화시키더라도 테스트에는 전혀 영향을 주지 않는다. 또한 테스트를 리팩터링하거나 진화시킬 대도 상용 코드에는 전혀 영향을 주지 않는다.

결론

테스트를 시스템의 일부로 설계하지 않으면 테스트는 깨지기 쉽고 유지보수하기 어려워지는 경향이 있다.

29장. 클린 임베디드 아키텍처

더그 슈미트의 명언

> *"소프트웨어는 닳지 않지만, 펌웨어나 하드웨어는 낡아 가므로 결국 소프트웨어도 수정해야 한다."*

저자의 의견을 더하면?

> *소프트웨어는 닳지 않지만, 펌웨어와 하드웨어에 대한 의존성을 관리하지 않으면 안으로부터 파괴돌 수 있다.*

임베디드 엔지니어가 아닌 엔지니어들 또한 펌웨어를 작성한다. 임베디드 엔지니어가 아닌 당신도 코드에 SQL을 심어 놓거나 개발하는 코드 전반에 플랫폼 의존성을 퍼뜨려 놓는다면, 본질적으로 펌웨어를 작성하는 셈이다. 안드로이드 앱 개발자 역시 업무 로직을 안드로이드 API로부터 분리하지 않는다면 펌웨어를 작성하는 셈이다.

앱-티튜드 테스트

켄트 백은 소프트웨어를 구축하는 세 가지 활동을 다음과 같이 기술했다.

1. 먼저 동작하게 만들어라. 소프트웨어가 동작하지 않는다면 사업은 망한다.
2. 그리고 올바르게 만들어라. 코드를 리팩토링해서 당신을 포함한 나머지 사람들이 이해할 수 있게 만들고 요구가 변경되거나 요구를더 잘 이해하게 되었을 때 코드를 개선할 수 있게 만들어라
3. 그리고 빠르게 만들어라. 코드를 리팩토링해서 **요구되는 성능**을 만족시켜라.

프로그래머가 오직 앱이 동작하도록 만드는 일만 신경 쓴다면 자신의 제품과 고용주에게 몹쓸 짓을 하는 것이다.

책에서 나오는 코드를 보면 특정 임베디드 장치에서만 테스트할 수 있음을 알시하는 파일 구조를 포함하고 있다.

특정 임베디드 장치에 의존하는 코드들

하드웨어, 전원, IO 등

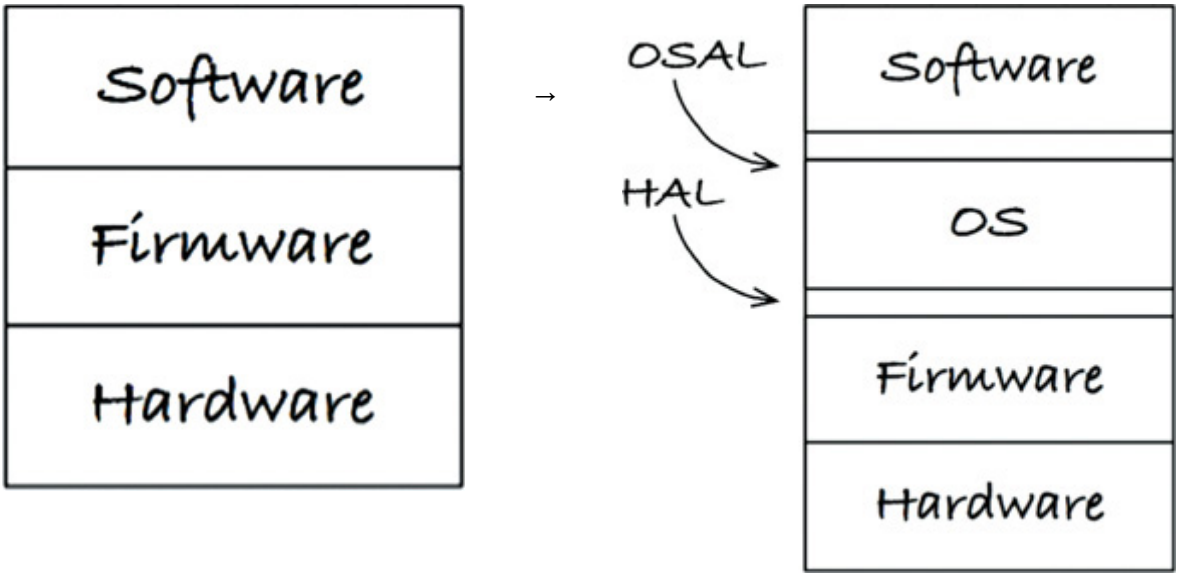
```
ISR(TIMER1_vect) { ... }
ISR(INT2_vect) { ... }
void btn_Handler(void) { ... }
float calc_RPM(void) { ... }
static char Read_RawData(void) { ... }
void Do_Average(void) { ... }
void Get_Next_Measurement(void) { ... }
void Zero_Sensor_1(void) { ... }
void Zero_Sensor_2(void) { ... }
void Dev_Control(char Activation) { ... }
char Load_FLASH_Setup(void) { ... }
void Save_FLASH_Setup(void) { ... }
void Store_DataSet(void) { ... }
float bytes2float(char bytes[4]) { ... }
void Recall_DataSet(void) { ... }
void Sensor_init(void) { ... }
void uC_Sleep(void) { ... }
```

임베디드가 지닌 특수한 문제 중 하나는 타깃-하드웨어 병목 현상이다. 임베디드 코드가 클린 아키텍처 원칙과 실천법을따르지 않고 작성된다면, 대개의 경우 코드를 테스트할 수 있는 환경이 해당 특정 타깃으로 국한될 것이다. 그리고 그 타깃이 테스트가 가능한 유일한 장소라면 타깃-하드웨어 병목 현상이 발생하여 진척이 느려질 것이다.

병목현상을 나누는 방법은 아래와 같다.

계층

최초에 왼쪽에서 오른쪽으로 개선해나가는 과정을 설명했다. 핵심은 다른 레이어에 직접 의존하지 않고 중간에 추상 레이어를 둠으로써 간접적으로 의존하도록 했다.



계층형 아키텍처는 인터페이스를 통해 프로그래밍하자는 발상을 기반으로 한다. 모듈들이 서로 인터페이스를 통해 상용작용한다면 특정 서비스 제공자를 다른 제공자로 대체할 수 있다.

오직 구현체에서만필요한 데이터 구조, 상수, 타입 정의들로 인터페이스를 헤더 파일을 어지럽히지 마라. 이는 단순히 어수선해지는 문제로 끝나지 않고, 결국 원치 않는 의존성을 만들어 낼 것이다.

구현 세부사항의 가시성을 제한하라. 구현 세부상은 변경될거라고 가정하라. 세부사항을 알고 있는 부분이 적을수록 추적하고 변경해야 할 코드도 적어진다.

DRY 원칙: 조건부 컴파일 지시자를 반복하지 마라.

결론

특정 하드웨어에 의존하지 않는 아키텍처와 코드로 프로그래밍 하는 것이 소프트웨어 수명을 늘리는데 도움이 된다. 추상 레이어를 통해 경계선을 긋고, 적절한 캡슐화를 통해 구현 세부사항의 공개를 최소화 하자.