개요

클린 아키텍처[1]의 5부 아키텍처에서는 원칙이라고 분류하여 설명하지는 않는다. 물론 YAGNI(You Aren't Going to Need It), KISS(Keep It Simple Stupid), DRY(Don't Repeat Yourself)에 대해서 언급하기는 하지만 이 원칙은 아키텍처 설계와 관련되지만 객체 지향 설계에서도 채용되는 개념으로 클린 아키텍처에서 말하는 아키텍처 설계의 핵심 원칙이라고 하기는 어렵다. 여기서는 클린 아키텍처 책의 4부에서는 이야기 하는 주요 개념들을 정리해 본다. 즉, 우선은 15장 부터 22장까지의 내용을 정리하고자 한다. 다른 장들은 전체적인 개념을 바탕으로 상세화 하여 설명한다고 볼 수 있어 여기서는 다루지 않겠다.

언급하는 주요 항목은 프로덕트 생명 주기 (개발, 배포 운영)를 고려하되, 소프트웨어 시스템을 지금까지 논의 한 컴포넌트로 분리하는 것인데, 이를 경계(Boundary)라고 하고 아키텍처 설계는 결국은 어떻게 경계를 나누는가에 대한 것이다. 소프트웨어를 시스템에서 정책(Policy)을 가장 핵심적인 요소로 식별하고 동시에 세부 사항(Detail)은 정책에 무관하게 만들어 이를 결정하는 일은 연기할 수 있게 한다. 더 구체적으로 보면, 정책, 레벨, 그리고 각각의 업무 원칙들을 어떻게 구획을 나누어 컴포넌트로 분리하는지에 대한 결합 분리(Decoupling, 디커플링)에대해 설명한다. 책에서 결국 주장하고자 하는 클린 아키텍처는 핵심 업무 규칙에 가장 엔터프라이즈 업무 규칙을 엔터티로 구분하고 이를 사용하여 구현하는 어플레케이션 업무 규칙을 유스 케이스 계층으로 본다. 이를 외부에서 사용하기 편하게 변환할 인터페이스 어댑터가 그 다음 계층이며 세부 사항인 외부 인터페이스를 프레임워크와 드라이버로 구분한다.

책에서는 좋은 아키텍처는 시스템이 모노리틱 구조(Monolithic Structure)로 태어나서 단일 파일로 배포되더라도, 이후에는 독립적으로 배포 가능한 단위들의 집합으로 성장하고 또 독립적인 서비스나 마이크로서비스 수준까지 성장할 수 있도록 만들어져야 한다고 이야기 한다. 각각의 상세 사항을 살펴 보자.

15장 아키텍처란

이책의 영어 원제는 Clean Architecture: a craftsman's guide to software structure and design 이다. 즉, 소프트웨어 크래트맨쉽을 강조하는 저자의 경험으로 소프트웨어 구조와 설계에 대한 가이드로서 아키텍처를 논하는 것이라 볼수 있다. 그래서, 저자는 아키텍트가 개발을 계속해야 한다고 한다. 이 부분에 대해서는 이견이 있을 수 있다.  
  
저자는 소프트웨어 시스템의 아키텍처란 시스템을 구축했던 사람들이 만들어낸 시스템의 형태라고 한다. 이 형태는 개발, 배포, 운영, 유지보수되도록 만들어진다고 한다. 이러한 일을 용이하게 만들기 위해서는 가능한 한 많은 선택지를, 가능한 한 오래 남겨두는 전략을 따라야 한다고 이야기 한다.

* 개발: 팀 구조가 다르다면 아키텍처 관련 결정에서도 차이가 난다. 이를 [**콘웨이의 법칙**](https://technical-leader.tistory.com/70?category=1060019)으로 연결하여 설명하였다.
* 배포: 배포 비용이 높을수록 시스템의 유용성은 떨어진다. 따라서 소프트웨어 아키텍처는 시스템을 단 한 번에 쉽게 배포할 수 있도록 만드는 데 그 목표를 두어야 한다.
* 운영: 아키텍처가 시스템 운영에 미치는 영향은 개발, 배포, 유지보수에 미치는 영향보다는 덜 극적이다. 운영에서 겪는 대다수의 어려움은 단순히 하드웨어를 더 투입해서 해결할 수 있다. 이는 인터넷 서비스를 기준으로 하는 이야기로 보이며, 실재로 휴대폰과 같은 제품을 목표라면 제품의 비용이 적게 해야 할 수 있다. 하드웨어는 값싸고 인력은 비싸다는 말이 뜻하는 바는 운영을 방해하는 아키텍쳐가 개발, 배포, 유지보수를 방해하는 아키텍쳐보다는 비용이 덜 든다는 것이다. 즉, 인력/하드웨어 보다는 비용 측면에서 고려해야 하지 않을까? 상황에 따라 사람이 비쌀 때도 혹은 프러덕트의 다른 요소의 비용이 높을 수도 있다고 생각한다.
* 유지보수: 유지보수는 모든 측면에서 봤을 때 소프트웨어 시스템에서 비용이 가장 많이 든다. 새로운 기능은 끝도 없이 행진하듯 발생하고 뒤따라서 발생하는 결함은 피할 수 없으며, 결함을 수정하는 데도 엄청난 인적 자원이 소모된다.

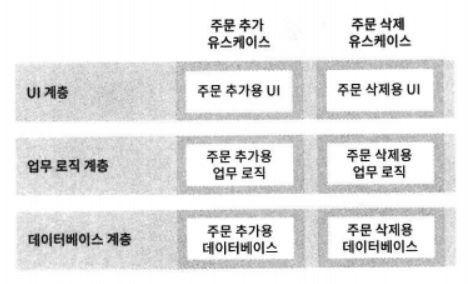
모든 소프트웨어 시스템은 주요한 두 가지 구성요소로 분해할 수 있다. 바로 **정책(Polciy)**과 **세부사항(Detail)**이다. 정책 요소는 모든 **업무 규칙(Business Rules)**과 업무 절차(Procuedures)를 구체화 한다. 정책이란 시스템의 진정한 가치가 살아 있는 곳이다. 세부사항은 사람, 외부 시스템, 플그래머가 정책과 소통할 대 필요한 요소지만, 정책이 가진 행위에는 조금도 영향을 미치지 않는다. 이러한 세부 사항에는 입출력 장치, 데이터 베이스, 웹 시스템, 서버, 프레임워크, 통신 프로토콜 등이 있다.

아키텍트의 목표는 시스템에서 정책을 가장 핵심적인 요소로 식별하고 동시에 세부사항은 정책에 무관하게 만들 수 있는 형태의 시스템을 구축하는 데 있다. 세부사항을 결정하는 일은 미루거나 연기할 수 있게 된다. 요점을 파악했을 것이다. 세부사항에 몰두하지 않은 채 고수준의 정책을 만들 수 있다면, 이러한 세부사항에 대한 결정을 오랫동안 미루거나 연기할 수 있다. 이러한 결정을 더 오래 참을 수 있다면, 더 많은 정보를 얻을 수 있고, 이를 기초로 제대로 된 결정을 내릴 수 있다. 선택 사항을 더 오랫동안 열어 둘 수 있다면 더 많은 실험을 해볼 수 있고 더 많은 것을 시도할 수 있다. 그리고 결정을 더 이상 연기할 수 없는 순간이 닥쳤을 대 이러한 실험과 시도 덕분에 더 많은 정보를 획득한 상태일 것이다. 좋은 아키텍트는 결정되지 않은 사항의 수를 최대화 한다.

16장 독립성.

좋은 아키텍처는 결국은 요구 사항을 지원해야 한다. 클린 아키텍처에서도 이 요구 사항을 시스템의 유스케이스를 지원해야 한다고 이야기 한다.  여기에 추가로 아키텍처는 이전에 이야기 한 프러덕트의 라이프 싸이클 중에서 유지 보수를 제외한 운영, 개발, 배포 관점에서도 지원해야 한다고 이야기 한다.  
  
어쩌 보면, 아키텍처는 시스템의 행위에 그다지 큰 영향을 주지 않는다. 다른 말로 행위와 관련하여 아키텍처가 열어 둘 수 있는 선택사항은 거의 없다. 좋은 아키텍처가 행위를 지원하기 위해 할 수 있는 일 중 에서 가장 중요한 사항은 행위를 명확히 하고 외부로 드러내며, 이를 통해 시스템이 지닌 의도를 아키텍처 수준에서 알아 볼 수 있게 만드는 것이다.  
이 부분은 뒤에도 나오지만, "소리치는 아키텍처"로 다르게 표현하기도 한다.  
  
아키텍처를 구조적인 분리로 보면, 계층(Layer)와 유스 케이스(Use Case) 차원에서 분리로 볼 수 있다. 계층 디커플링/결합 분리 (Decoupling Layers) 측면에서 보면, 사용자 인터페이스(User Interface, UI)가 변경되는 이유는 업무 규칙(Business Rule)과는 아무런 관련이 없다. 만약 유스케이스가 두 가지 요소를 모두 포함한다면, 뛰어난 아키텍트는 유스케이스에서 UI부분과 업무 규칙 부분을 서로 분리하고자 할 것이다.

업무 규칙은 그 자체가 애플리케이션과 밀접한 관련이 있거나, 혹은 더 범용적일 수도 있다. 예를 들어 입력 필드 유효성 검사는 애플리케이션 자체와 밀접하게 관련된 업무 규칙이다. 계좌의 이자 계산이나 재고품 집계는 업무 도메인에 더 밀접하게 연관된 업무 규칙이다. 이들 서로 다른 두 유형의 규칙은 각자 다른 속도로 그리고 다른 이유로 변경될 것이다. 이들 규칙은 서로 분리하고, 독립적으로 변경할 수 있도록 만들어야 한다. 즉, 분리/디커플링 해야 한다.  
  
서로 다른 이유로 변경되는 것에 유스케이스 그 자체가 포함된다. 이를 유스 케이스 디커플링 (Decoupling Use Cases)이라고 한다. 주문 입력 시스템에서 주문을 추가하는 유스케이스는 주문을 삭제하는 유스케이스와는 틀림없이 다른 속도로 그리고 다른 이유로 변경된다. 유스케이스는 시스템을 분할하는 매우 자연스러운 방법이라고 설명한다.



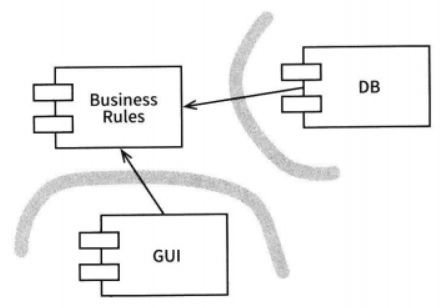
결합 분리/디커플링(Decoupling)[1]

이렇게 분리된 컴포넌트를 서로 다른 서버에서 실행해야 하는 상황이라면, 이들 컴포넌트가 단일 프로세서의 동일한 주소 공간에 함께 상주하는 형태로 만들어져서는 안 된다. 분리된 컴포넌트는 반드시 독립된 서비스가 되어야 하고 일종의 네트워크를 통해 서로 통신해야 한다. 많은 아키텍트가 이러한 컴포넌트를 '서비스(Service)' 또는 '마이크로서비스(Microservice)라고 하는데, 그 구분 기준은 모호한 면이 있다.  실제로 서비스에 기반한 아키텍처를 흔히들 서비스 지향 아키텍처(Service-oriented architecture SOA)라고 부른다.

## 17장 경계: 선 긋기

소프트웨어 아키텍처는 선을 긋는 기술이며, 저자는 이 선을 경계(boundary)라고 부른다. 경계는 소프트웨어 요소를 서로 분리하고, 경계 한편에 있는 요소가 반대편에 있는 요소를 알지 못하도록 막는다. 아키텍트가 내리는 경계의 결정이 핵심적인 업무 로직을 오염시키지 못하게 만들려는 목적으로 쓰인다. 그러면서, 이케텍트의 목표는 필요한 시스템을 만들고 유지하는 데 드는 인적 자원을 최소화하는 것이라는 사실을 상기하자. 로버트 마틴은 인적 자원이 가장 비싼 자원으로 보고 있다.

어떤 종류의 결정이 이른 결정일까? 바로 시스템의 업무 요구사항, 즉 유스케이스와 아무런 관련이 없는 결정이다. 프레임워크, 데이터베이스, 웹 서버, 유틸리티 라이브러리, 의존성 주입에 대한 결정 등이 여기 포함된다.  
  
어떤게 선을 그을까? 그리고 언제 그을까? 이와 관련된 답으로는 관련이 있는 것과 없는 것 사이에 선을 긋는 것이 기본 규칙이라 할 수 있다.  GUI는 업무 규칙과는 관련 없기 때문에, 이 둘 사이에는 반드시 선이 있어야 한다. 데이터베이스는 GUI와는 관련이 없으므로, 이 둘 사이에도 반드시 선이 있어야 한다. 데이터베이스는 업무 규칙과 관련이 없으므로, 이 둘 사이에도 선이 있어야 한다.



업무 규칙(Business Rules)에 플러그인으로 연결하기[1]

두 컴포넌트 사이에 이러한 경계선을 그리고 화살표의 방향이 Business Rules를 향하도록 만들었으므로, Busines Reules에서는 어떤 종류의 데이터 베이스도 사용할 수 있음을 알 수 있다. Database 컴포넌트는 다양한 구현체로 교체될 수 있으며, Business Rules는 조금도 개의치 않는다. GUI는 다른 종류의 인터페이스로 얼마든지 교체할 수 있으며 BusinessRules는 전혀 개의치 않는다는 사실을 알 수 있다. 데이터베이스와 GUI에 대해 내린 두 가지 결정을 하나로 합쳐서 보면 컴포넌트 추가와 관련한 일종의 패턴, 즉, 플러그인 아키텍처가 만들어진다.

## 18장 경계 해부학(Boundary Anatomy)

위와 같이 분리된 일련의 소프트웨어 컴포넌트와 그 컴포넌트들을 분리하는 경계를 통해서 시스템 아키텍처가 정의 된다. 런타임에 경계를 횡단한다함은 그저 경계 한쪽에 있는 기능에서 반대편 기능을 호출하여 데이터를 전달하는 일에 불과하다. 적절한 위치에서 경계를 횡단하게 하는 비결은 소스 코드 의존성 관리에 있다.  
  
단일체로 변역하는 모노리틱(Monolithic) 구조는 쉽게 이야기 하면, 단일 실행 파일에 지나지 않는다. 이 파일은 정적으로 링크된 C/C++ 프로젝트이거나, 실행 가능한 jar 파일로 묶인 일련의 자바 클래스 파일이거나, 단일 .EXE파일로 묶인 일련의 .NET 바이너리등일 것이다.  
  
배포형 컴포넌트(library, jar, DLL), 스레드, 로컬 프로세스(OS상의 Process), 서비스(물리적으로도 분리된 컴퓨팅 리소스에서 동작) 등으로 컴포넌트들을 분리하는 방법들이 있을 수 있다.

## 19장 정책과 수준

소프트웨어 시스템이란 정책(Policy)을 기술한 것이다. 실제로 컴퓨터 프로그램의 핵심부는 이게 전부다. 컴포터 프로그램은 각 입력을 출력으로 변환하는 정책을 상세하게 기술한 설명서다. 대다수의 주요 시스템에서 하나의 정책은 이 정책을 서술하는 여러 개의 조그만 정책들로 쪼갤 수 있다. 예를 들어 집계와 관련된 업무 규칙을 처리하는 방식을 서술하는 조그만 정책이 있을 수 있다. 그리고 특정 보고서를 어떤 포맷으로 만들지 서술하는 또 다른 정책이 있을 수 있다. 또한 입력 데이터를 어떻게 검증할지를 서술하는 정책이 있을 수 있다.  
  
수준(Level)을 엄밀하게 정의하자면 **'입력과 출력까지의 거리'**다. 시스템의 입력과 출력 모두로 부터 멀리 위치할 수록 정책의 수준은 높아진다. 입력과 출력을 다르는 정책이라면 시스템에서 최하위 수준에 위치한다.  
  
정책을 컴포넌트로 묶는 기준은 정책이 변경되는 방식에 달려 있다는 사실을 상기하자. 단일 책임 원칙(SRP)와 공통 폐쇄 원칙(CCP)에 따르면 동일한 이유로 동일한 시점에 변경되는 정책은 함께 묶인다. 고수준 정책, 즉 입력과 출력에 서부터 멀리 떨어진 정책은 저수준 정책에 비해 덜 빈번하게 변경되고, 보다 중요한 이유러 변경되는 경향이 있다.저수준 정책, 즉 입력과 출력에 가까이 위치한 정책은 더 빈번하게 변경되며, 보다 긴급성을 요하며, 덜 중요한 이유로 변경되는 경향이 있다.

이처럼 모든 소스 코드 의존성의 방향이 고수준 정책을 향할 수 있도록 정책을 분리했다면 변경의 영향도를 줄일 수 있다. 시스템의 최저 수준에서 중요하지 않지만 긴급한 변경이 발생하더라도, 보다 높은 위치의중요한 수준에 미치는영향은 거의 없게 된다.

## 20장 업무 규칙

**업무 규칙(Business Rules)**은 사업적으로 수익을 얻거나 비용을 줄일 수 있는 규칙 또는 절차다. 더 엄밀하게 말하면 컴퓨터 상으로 구현했는지와 상관없이, 업무 규칙은 사업적으로 수익을 얻거나 비용을 줄일 수 있어야 한다. 심지어 사람이 수동으로 직접 수행하더라도 마찬가지다. 이러한 규칙을 **핵심 업무 규칙(Criticial Business Rule)**이라고 부를 것이다. 왜나하면 이들 규칙은 사업 자체에 핵심적이며, 규칙을 자동화하는 시스템이 업서라도 업무 규칙은 그대로 존재하기 때문이다. 핵심 업무 규칙은 데이터를 요구하는데, 책에서는 이를 **핵심 업무 데이터(Critical Business Data)**라고 부르고 있다.

핵심 규칙과 핵심 데이터는 본질적으로 결합되어 있기 때문에 객체로 만들 좋은 후보가 된다. 우리는 이러한 유형의 객체를 **엔티티(Entity)**라고 부른다.  
  
유스케이스  
자동화된 시스템이 동작하는 방법을 정의하고 제약함으로써수익을 더거나 비용을 줄이는 업무 규칙도 존재한다. 바로 이것이 유스케이스(Use case)다. 유스케이스는 **애플리케이션에 특화된 업무 규칙(Application-specific Business Rules)**을 설명한다.

## 21장: 소리치는 아키텍처

여러분의 애플리케이션 아키텍처는 뭐라고 소리치는가? 상위 수준의 디렉터리 구조, 최상위 패키지에 담긴 소스 파일을 볼 때, 시스템의 본질을 이야기 하는가? 즉,  "핼스 케어 시스템이야" 또는 "재고 관리 스스템이야"라고 소리치는가? 아니면 사용하고 있는 디테일인 프레임워크에 대해서 이야기 하고 있는가? 예를 들자면, "레일스(Rails)야", "스프링(Spring)/하이버네이트(Hibernate)야", 아니면 "ASP야"라고 소리치는가?  
  
책에서는 좋은 아키텍처는 유스케이스를 그 중심에 두기 때문에, 프레임워크나 도구 환경에 전혀 구애받지 않고 유스케이스를 지원하는 구조를 아무런 문제 없이 기술 할 수 있다고 이야기 한다. 좋은 소프트웨어 아키텍처는 프레임워크, 데이터베이스, 웹 서버, 그리고 여타 개발 환경 문제나 도구에 대해서는 결정을 미룰 수 있도록 만든다.

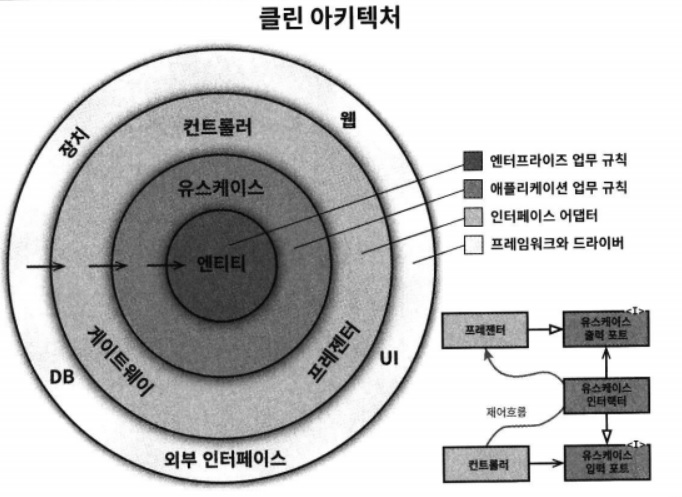
프레임워크는 열어 둬야 할 선택사항이다. 좋은 아키텍처는 프로젝트의 훨씬 후반까지 레일스, 스프링, 하이버네이트, 톰갯(Tomcat), MySQL에 대한 결정을 하지 않아도 되도록 해준다. 프레임워크, 데이터베이스, 웹 서버, 여타 개발환경 문제나 도구에 대한 결정을 미룰 수 있어야 한다. 프레임워크는 매우 강력하고 상당히 유용할 수 있다. 하지만, 프레임워크는 도구일 뿐, 삶의 방식은 아니다.

## 22장: 클린 아키텍처

책에서는 다른 아키텍처 접근 방법에 대해서 설명한다. 이 접근 법들도 목표가 같은데, 바로 관심사의 분리(Seperation of concerns)다. 모두 소프트웨어를 계층으로 분리함으로써 이 관심사의 분리라는 목표를 달성한다고 한다.

이들 아키텍처는 모두 시스템이 다음과 같은 특징을 지니도록 만든다.  
. 프레임워크 독립성  
. 테스트 용이성  
. UI 독립성  
. 데이터베이스 독립성  
. 모든 외부 에이전시에 대한 독립성

아래 다이어그램은 이들 아키텍처 전부를 실행 가능한 하나의 아이디어로 통합하려는 시도라고 이야기 한다. 즉, 핵심 개념도로 볼 수 있으며 지금까지 이야기 한 부분에 대한 큰 그림으로 이해할 수 있다.



클린 아키텍처[1]

### **의존성 규칙**

클린 아키텍어 다이어 그램에서 안으로 들어갈 수록 고수준의 소프트웨어가 된다. 바깥쪽 원은 메커니즘이고 안쪽 원은 정책이다.  여기서 가장 중요한 규칙은 의존성 규칙(Dependency Rule)이다. 즉, 소스코드 의존성은 반드시 안쪽으로, 고수준의 정책을 향해야 한다. 내부의 원에 속한 요소는 외부의 원에 속한 어떤 것도 알지 못한다. 함수, 클래스, 변수, 그리고 소프트웨어 엔티티로 명명되는 모든 것이 포함된다. 같은 이유로, 외부의 원에서 선언된 데이터 형식도 (Data Type)도 내부의 원에서 절대로 사용해서는 안 된다.

### **엔티티**

엔티티는 전사적인 핵심 업무 규칙을 캡슐화한다. 엔티티는 메서드를 가지는 객체이거나 일련의 데이터 구조와 함수의 집합이다. 운영 관점에서 특정 애플리케이션에 무언가 변경이 필요하더라도 엔티티 계층에는 절대로 영향을 주어서는 안 된다.

### **인터페이스 어댑터**

인터페이스 어댑터(Interface Adapter) 계층은 일련의 어댑터들로 구성된다. 어댑터는 데이터를 유스케이스와 인터티에게 가장 편리한 형식에서 데이터베이스나 웹 같은 외부 에이전시에게 가장 편리한 형식으로 변환한다. GUI의 MVC 아키텍처를 모두 포괄한다. 프레젠터(Presenter), 뷰(View), 컨트롤러(Controller)는 이 계층에 속한다. 모델은 그저 이 계층에서 사용되는 데이터 구조로 보면 된다.

### **프레임워크와 드라이버**

클린 아키텍처 그림에서 가장 바깥쪽 계층은 일반적으로 데이터베이스나 웹 프레임워크 같은 프레임워크나 도구들로 구성 된다. 이는 안쪽 원과 통신하기 위한 접합 코드로 볼 수 있다.

### **원은 네 개여야만 하나?**

이 다이어 그램은 개념을 설명하기 위한 하나의 예시로, 실재 시스템에서는 더 많은 원이 필요할 수도 있다. 하지만, 어떤 경우에도 의존성 규칙은 적용된다. 소스 코드 의존성은 항상 안쪽으로 향한다. 안쪽으로 이동할 수록 추상화와 정책의 수준은 높아진다. 가장 바깥쪽 원은 저수준의 구체적인 세부사항으로 구성된다.

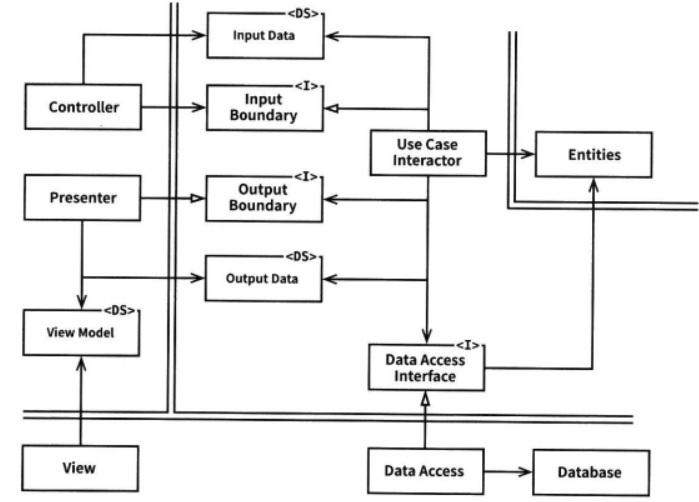
### **경계 횡단하기**

그림의 우측 하단 다이어그램에 원의 경계를 횡단하는 방법을 보여 주는 예시가 있다. 컨트롤러와 프레젠터가 다음 계층에 속한 유스케이스와 통신하는 모습을 확인할 수 있다. 우선 제어흐름에 주목해 보자. 제어흐름은 컨트롤러에서 시작해서, 유스케이스를 지난후, 프레젠터에서 실행되면서 마무리 된다. 이와 상반되게 의존성은 반대이다.이와 같은 경우, 대체로 의존성 역전 원칙을 사용하여 해결한다.

경계를 가로지르는 데이터는 흔히 간단한 데이터 구조로 이루어져 있다. 기본적인 구조체나 간단한 데이터 전송 객체(Data Transfer Object)등 원하는 대로 고를 수 있다. 또는 함수를 호출할 때 간단한 인자를 사용해서 데이터로 전달할 수도 있다. 그게 아니라면 데이터를 해시맵으로 묶거나 객체로 구성할 수도 있다.

### **전형적인 시나리오**

이 클린 아키텍처의 예로서 전형 적인 웹 서비스 아키텍처를 아래 그림에서 살펴 볼 수 있다. 웹 서버는 사용자로 부터 입력 데이터를 모아서 좌측 상단의 Controller로 전달한다. Controller는 데이터를 평범한 자바객체(POJO)로 묶은 후, InputBoundary 이터페이스를 통해 UseCaseInsteractor로 전달한다. UseCaseInteractor는 이 데이터를 해석해서 Entities가 어떻게 춤출지를 제어하는데 사용한다.



전형적 웹 서비스 구조[1]

## 결론

클린 아키텍처에서 주장하는 가장 핵심 적인 것을 보통 클린 아키텍처 다이어 그램에서 이야기 하는 4가지 영역의 분리라고 볼 수 있다. 서론에서도 이야기 했듯이 아키텍처의 구조적인 측면에서 경계(Boundary)를 만드는 것이 가장 중요한 일일 수 있다. 다른 말로 관심사의 분리(Separation of Concern)이라고도 했다.

이 분리의 주된 목적은 변경이 필요할 때, 영향을 받지 않는 부분을 분리하자는 것이다. 이렇게 하기 위해 가장 먼저 분리 한 것은 정책(Policy)와 세부 사항(Detail)이다. 그림에서는 프레임워크/드라이버와 나머지가 분리되었다고 볼 수 있다. 정책에서는 업무 규칙 중에서도 유스 케이스에서 더 핵심적인 것을 분리해서 엔터프라이즈 업무 규칙과 유스 케이스 업무 규칙이 분리되고 이 것들이 외부의 세부 사항과 연결하기 위한 영역으로 인터페이스 어댑터를 둔 것으로 볼 수 있다.

이 구조에서 가장 중요한 원칙이 의존성 규칙이라고 했다. 즉, 내부 영역이 외부 영역에 의존하면 안되고 몰라야 한다는 것이다. 혹시 제어 흐름이 반대인 것은 의존성 역전 원칙(DIP)를 적용하는 것도 언급하였다. 이렇게 하면서, 세부 사항들은 늦게 결정하도록 하면서 여러 시험을 통해 더 좋은 선택을 할 수 있게 한다고 이야기 하고 이 것이 좋은 아키텍처, 클린 아키테처라고 주장한다.

애자일과 소프트웨어 크래프트맵쉽을 중시하는 저자의 철학과 연결되는 것이 제품/프러덕트가 모노리틱 구조(Monolithic Structure)로 태어나고, 성장하고, 분화 한다고 이야기로 보인다. 이렇게 분화한 것이  독립적인 서비스(Service Oriented Architecture, SOA)나 마이크로서비스(Microservice Architecture)가 된다고 한다.