

# Hexagonal Architecture로 구현한 Clean Architecture

~ Get Your Hands Dirty on  
Clean Architecture ~

2021. 08.

Translated & composed by kimsh

# 목차

세미나의 목표와 선수 지식

전통적인 레이어 구조

클린 아키텍처의 구현체인 헥사고널 아키텍처  
패키지 구조

Port & Adaptor :영속 영역

매핑 전략

지름길

레이어 격리를 유지하기 위한 장치

뭐가 가장 나은 선택일까?

# 세미나의 목표와 선수 지식

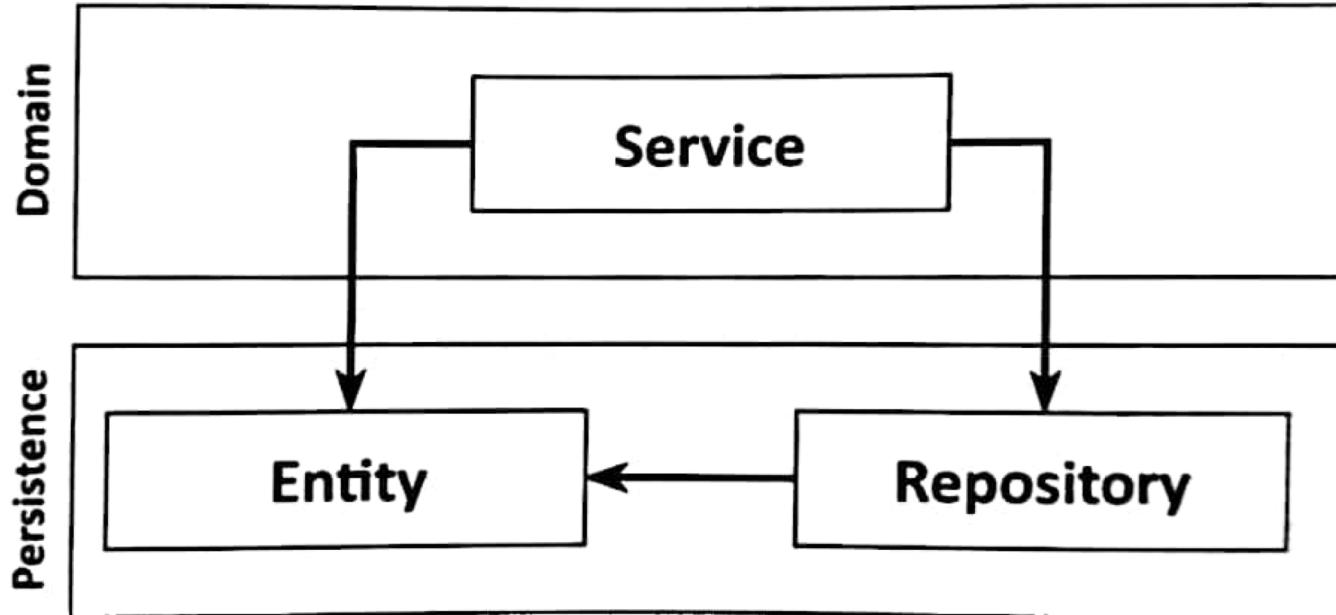
## 목표

클린 아키텍처를 코드 레벨에서 적용한 헥사고널 아키텍처를 분석하고,  
이를 바탕으로 디코드 아키텍처의 개선 방향에 대해 고려해본다.

## 선수 지식

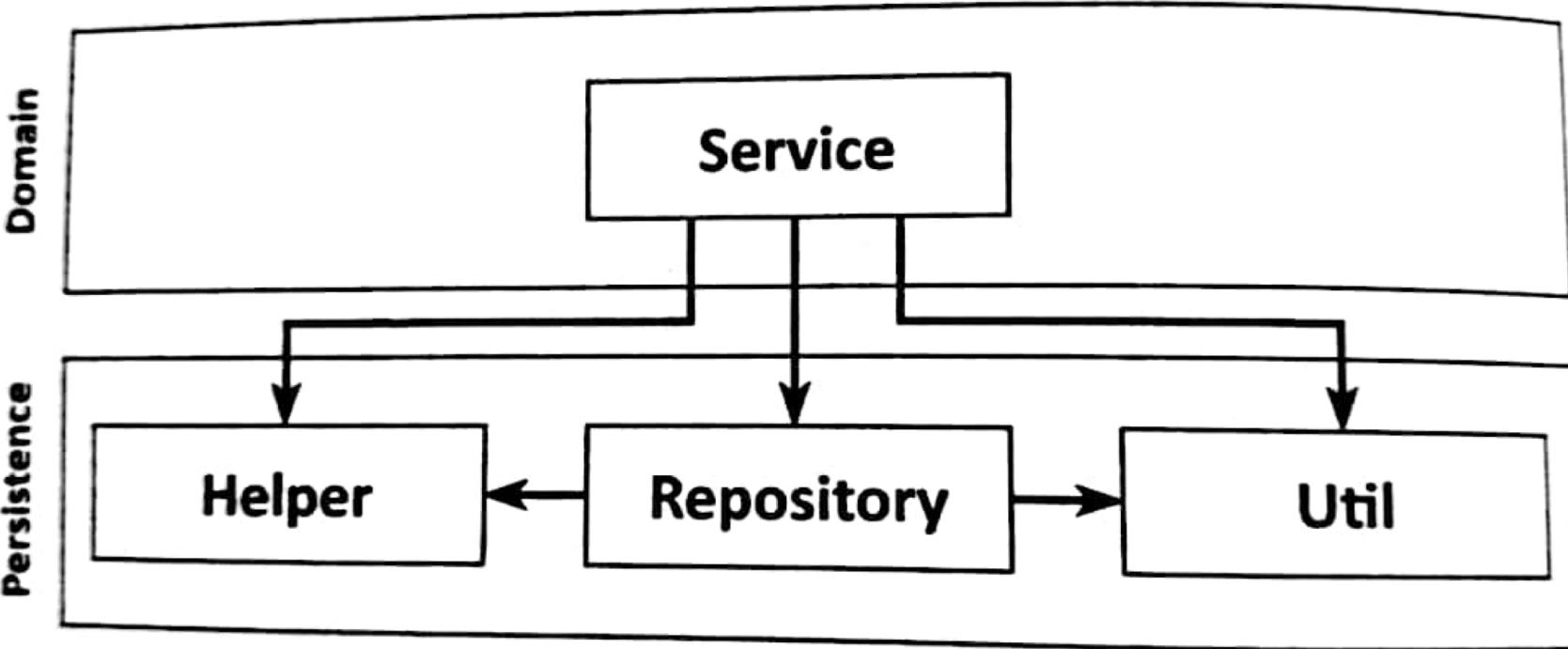
'클린 아키텍처'를 읽었거나 구글링 등을 통해 어느정도 이해하고 있다.  
실제로 클린 아키텍처를 적용하려는 시도를 해봤다.

# 전통적인 레이어 구조



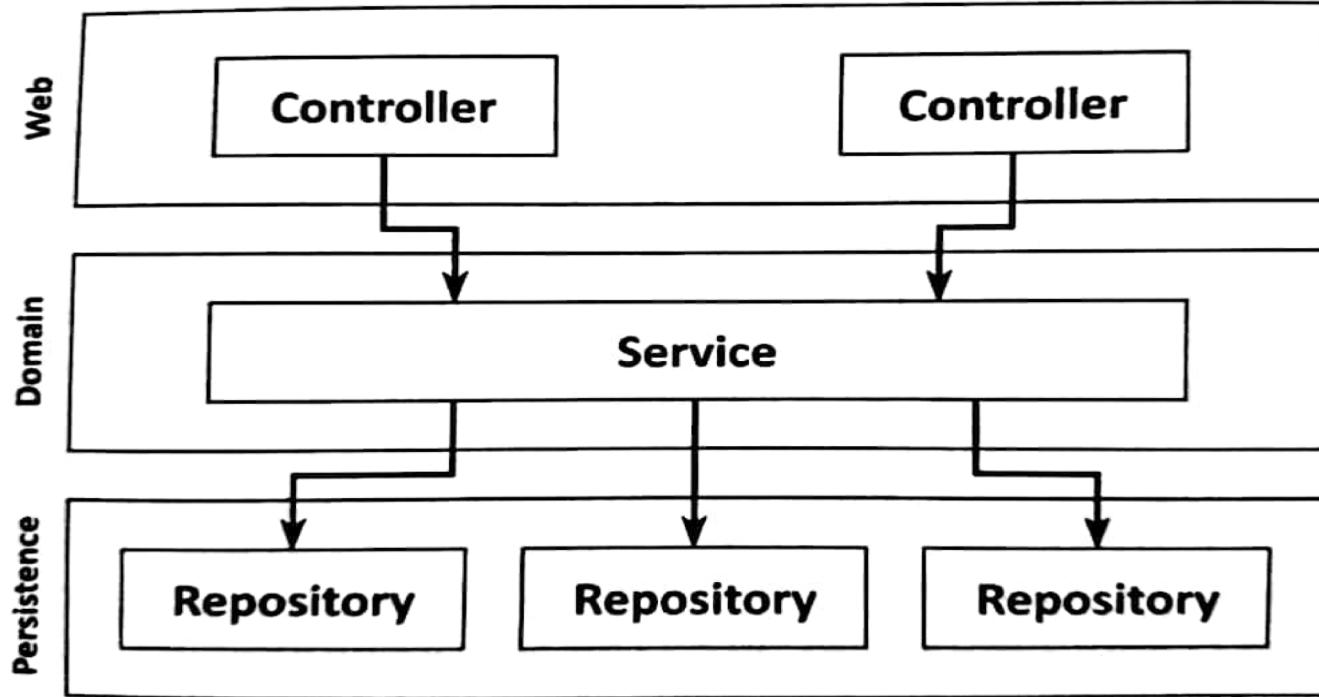
직관적이고 효율적이며 빠르다  
하지만 데이터베이스가 중심이라서  
데이터베이스나 테이블이 바뀌면 모든걸 수정해야 한다

# 전통적인 레이어 구조



데이터베이스에 계속해서 의존하며, 비대해지는 영속 영역  
Helper와 Util의 의존성도 심해지다보니  
리팩토링과 수정도 더욱 어려워 짐

# 전통적인 레이어 구조



서비스 클래스 하나에 2,000줄  
끝없이 이어지는 private 메서드와 참조 클래스들  
영등포시장 아키텍처...

# 전통적인 레이어 구조

어떤 외부 영역(클라이언트, 데이터베이스)에도 의존하지 않는  
단단하고 자유로운 애플리케이션 영역을  
구현하고 싶고, 해야 한다.

?

# 클린 아키텍처의 구현체인 헥사고널 아키텍처

The following figure shows how such an architecture might look on an abstract level:

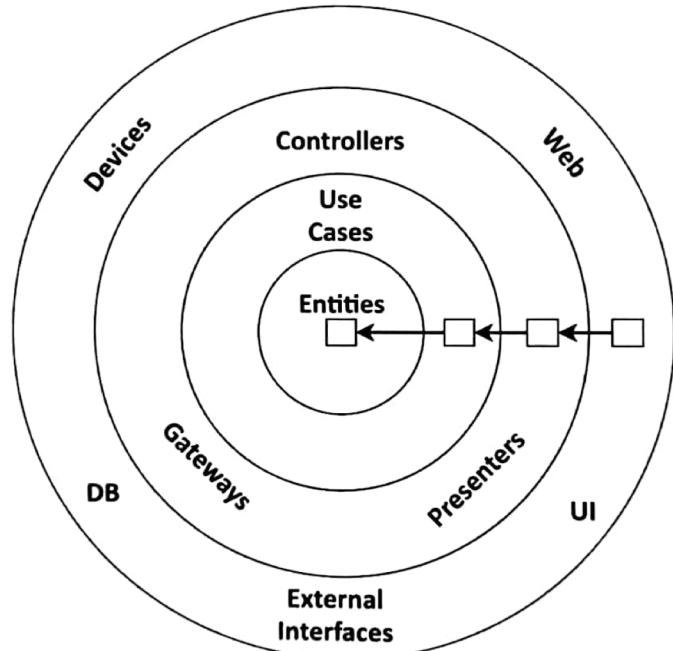


Figure 2.3: In a clean architecture, all dependencies point inward toward the domain logic.

Source: "Clean Architecture" by Robert C. Martin

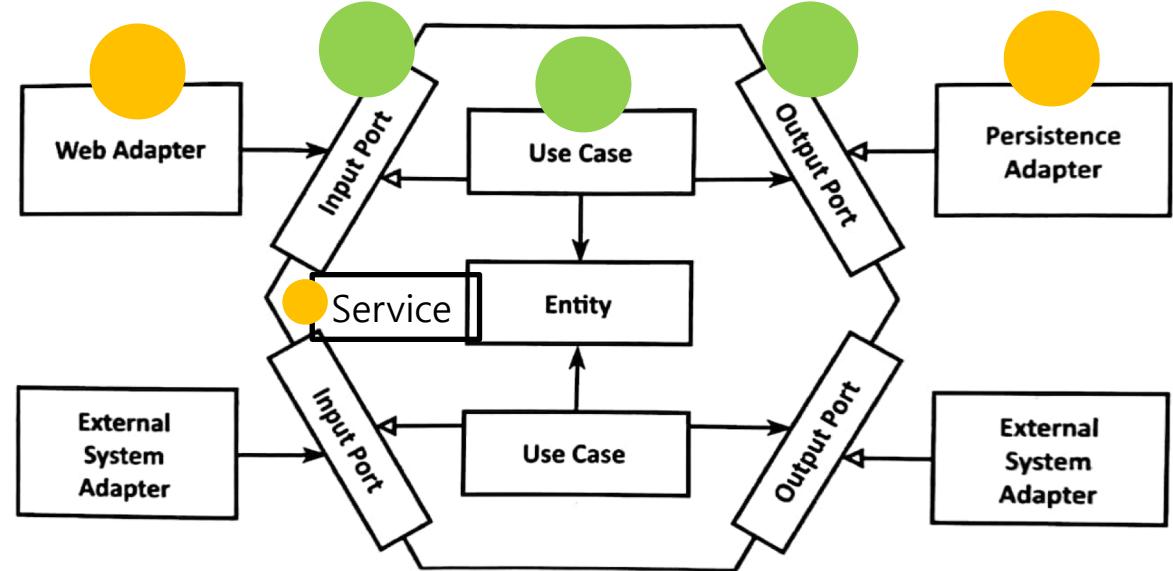


Figure 2.4: A hexagonal architecture is also called a "ports-and-adapters" architecture since the application core provides specific ports for each adapter to interact with

의존 방향은 오직 내부로, 각 레이어는 인터페이스로 연결

# 패키지 구조

```
1 buckpal
2   └── account
3     └── adapter
4       └── in
5         └── web
6           └── o AccountController
7         └── out
8           └── persistence
9             └── o AccountPersistenceAdapter
10            └── o SpringDataAccountRepository
11   └── domain
12     └── + Account
13     └── + Activity
14   └── application
15     └── o SendMoneyService
16   └── port
17     └── in
18       └── + SendMoneyUseCase
19     └── out
20       └── + LoadAccountPort
21       └── + UpdateAccountStatePort
```

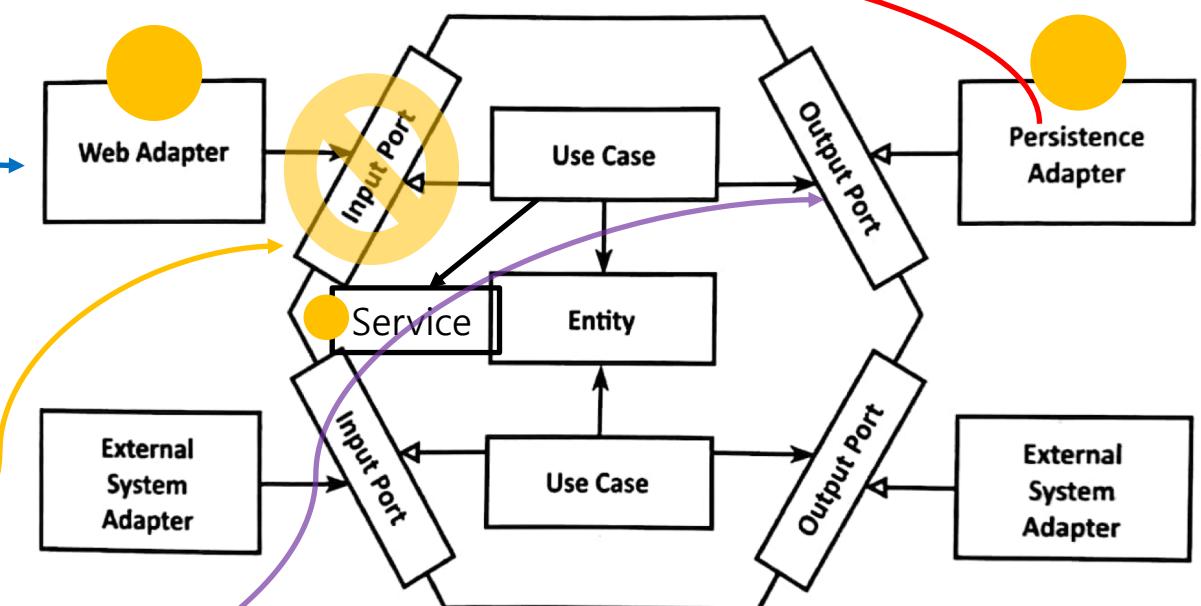


Figure 2.4: A hexagonal architecture is also called a "ports-and-adapters" architecture since the application core provides specific ports for each adapter to interact with

Figure 10.2: Package structure with visibility modifiers.

# Port & Adaptor : 영속 영역

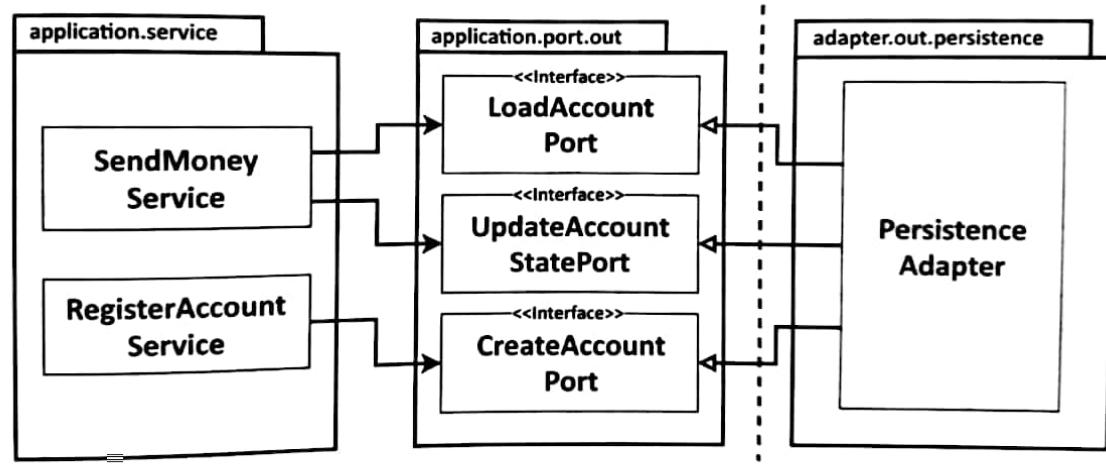


Figure 6.3: Applying the Interface Segregation Principle removes unnecessary dependencies and makes the existing dependencies more visible

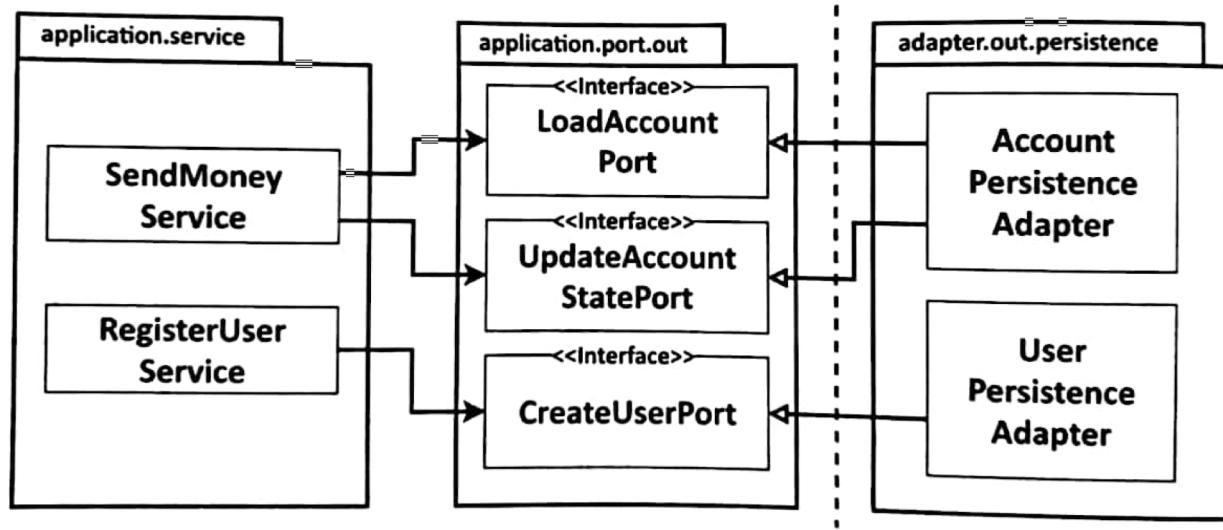


Figure 6.4: We can create multiple persistence adapters, one for each aggregate

구현체는 한 레퍼지토리(= 영속 어댑터)로 만들더라도,  
포트 인터페이스를 분리하면 충분한 격리를 할 수 있다.  
또한 각 포트의 이름을 명확하게 Load, Update, Create로 나눔으로써  
누구든지 쉽게 코드를 이해하고, 서비스와 영속 어댑터의 관계도 파악할 수 있다.

# 매핑 전략 - 노 매핑

## The "No Mapping" Strategy

The first strategy is actually not mapping at all:

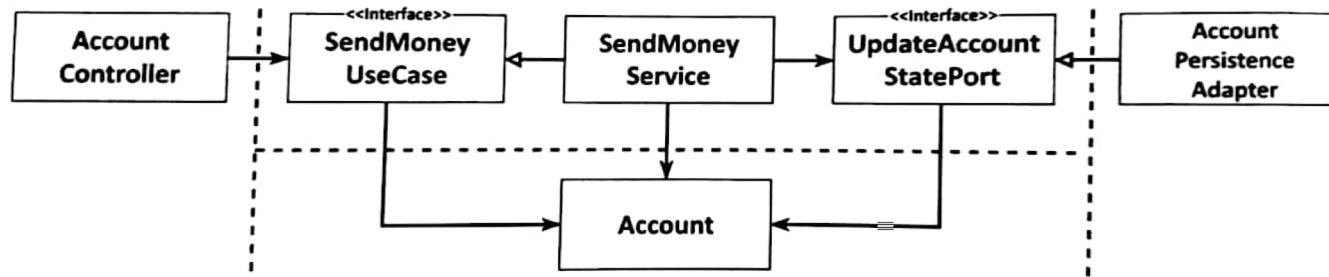


Figure 8.1: If the port interfaces use the domain model as the input and output model,  
we don't need to map between layers

컨트롤러, 서비스, 레퍼지토리가 모두 한 엔티티를 의존하고 있다. 문제점은,

1. 엔티티가 JSON 변환과 ORM의 애노테이션을 모두 갖게 된다.
2. 클라이언트나 레퍼지토리에만 필요한 특수한 필드가 모두 섞이므로 null 처리에 신경써야 한다.
3. 엔티티를 수정할 이유가 여러 개가 되므로 '단일 책임 원칙'을 위배한다.  
하지만?

모든 애플리케이션은 처음엔 '단순한 CRUD'로 시작한다.

어떤 비즈니스 로직은 이후에 복잡한 모델로 바뀌기도 하지만 영원히 바뀌지 않는 로직도 있다.

그러니 일단 노 매핑 전략으로 시작하는 것도 괜찮은 선택이다.

나중에 비즈니스 로직이 복잡해지면? 그때 적절한 매핑 전략으로 리팩토링하면 된다.

# 매핑 전략 – 투 웨이 매핑

## The "Two-Way" Mapping Strategy

A mapping strategy where each layer has its own model is what I call the "two-way" mapping strategy, outlined in the following figure:

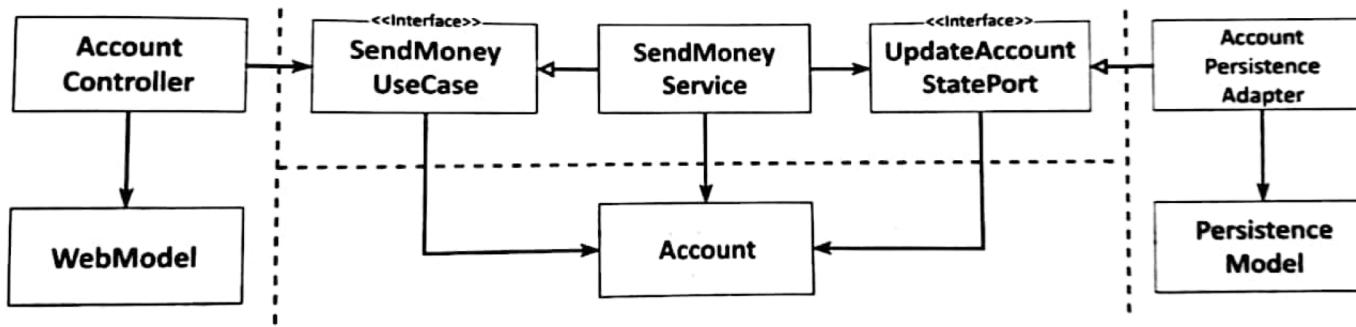


Figure 8.2: With each adapter having its own model, the adapters are responsible for mapping their model into the domain model and back

각 레이어가 자신의 모델 객체를 가지고 있다.

가장 단순하게 클린 아키텍처의 규칙을 지킨다.

하지만?

너무나 많은 보일러플레이트 코드를 만들어야 한다.

매핑 프레임워크를 사용한다 하더라도... 매핑 과정에서 발생하는 에러를 해결하는 것도 우리를 지치게 만든다.  
그리고 컨트롤러 – 서비스 – 레퍼지토리를 연결하려면 어차피 모든 경계에서 도메인 모델을 참조할 수 밖에 없다.

# 매핑 전략 - 풀 매핑

## The "Full" Mapping Strategy

Another mapping strategy is what I call the "full" mapping strategy, sketched in the following figure:

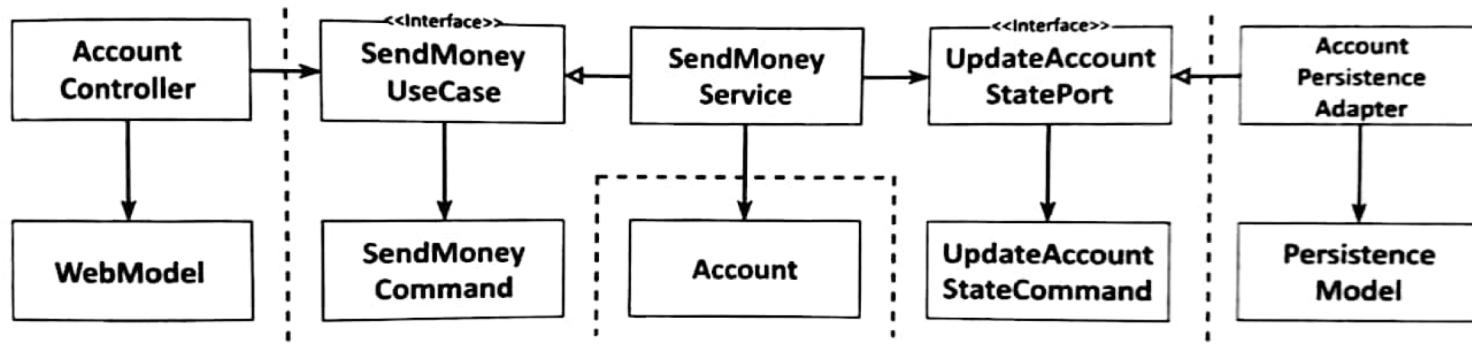


Figure 8.3: With each operation requiring its own model, the web adapter and application layer each map their model into the model expected by the operation they want to execute

도메인 모델을 완전히 격리하기 위해서 command 객체를 도입한다.

컨트롤러가 요청을 받으면 서비스가 명확하게 알 수 있는 커맨드(=명령) 객체로 변환해서 전달한다.

서비스는 커맨드 객체의 요청에 맞춰서 비즈니스 로직을 수행한다.

DB가 필요하다면 다시 영속 어댑터에게 커맨드 객체로 데이터를 전달한다.

이 전략은 컨트롤러 -> 서비스에서는 상당히 유용하지만,  
서비스 -> 레퍼지토리에서는 이득보다는 매팅 오버헤드가 더 커서 추천하지 않는다.

# 매핑 전략 – 원 웨이

## The "One-Way" Mapping Strategy

There is yet another mapping strategy with another set of pros and cons – the "one-way" strategy sketched in the following figure:

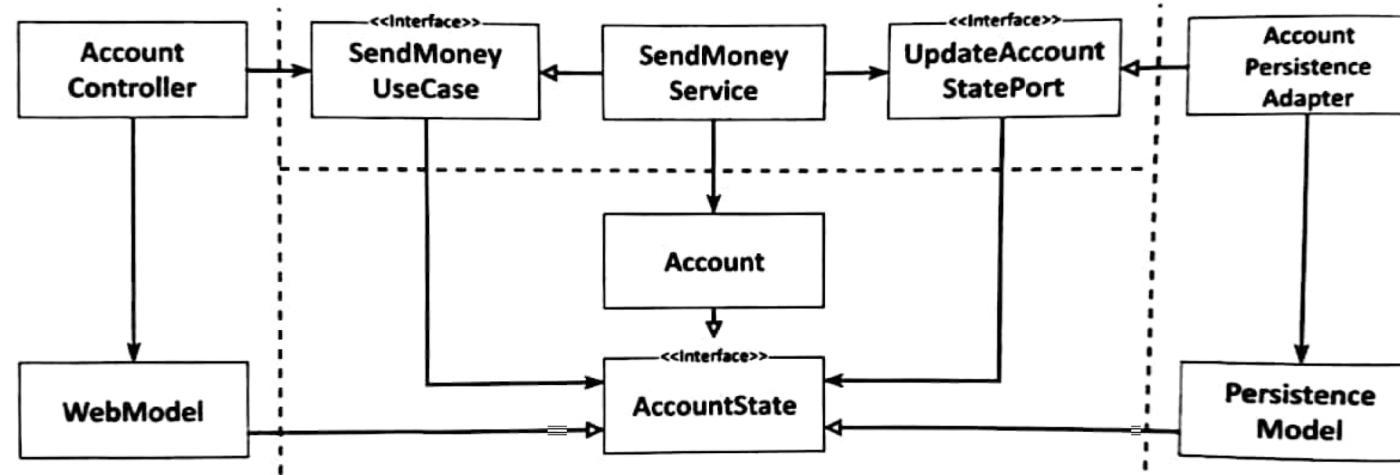


Figure 8.4: With the domain model and the adapter models implementing the same "state" interface, each layer only needs to map objects it receives from other layers – one way

이런 재미있는 매핑 전략도 있습니다.

상태만 저장하는 도메인 모델 하나를 쓰지만 인터페이스를 이용해서 각 레이어가 도메인 모델을 조작하는 방법을 제한하는 거죠.

# 매핑 전략

어떤 매핑 전략을 사용해야 할까?

"It depends."

한 애플리케이션에 여러 매핑 전략이 혼재해 있다면  
왠지 불편하고 하나로 통일하고 싶습니다.

하지만 통일하기 위해 불필요한 작업을 더 해야한다면 통일할 이유가 있을까요?

차라리 각 매핑 전략을 사용할 가이드 라인을 정하는게 더 합리적일 것 같습니다.  
또한 각 전략에는 장, 단점도 있으니까요.

이런 경우엔 '노 매핑', 저런 경우에는 '투웨이 매핑'을 고려해보라는 식으로 말이죠.  
이렇게 계속 '리팩토링'하며 발전시키면 됩니다.

# 지름길

## Broken Windows Theory : 깨진 창문 이론

빈민가에서는 모든 차의 창문이 깨져 있지만,  
부자들이 사는 동네에서는 차가 다 깨끗하다.

왜 그럴까...? 부자 동네의 차를 일부러 부셔보면 어떨까?

놀랍게도... 사이드 미러 하나를 부셨을 뿐인데 다음날이 되니 차가 박살났다.

사람은 누구나 깨끗한 곳에 있으면 자기도 모르게 조심하게 되고,  
더러운 곳에 있다보면 쉽게 쓰레기를 버리게 된다.

# 지름길

하지만 살다 보면 가끔 창문을 깨야 할 때도 있다.  
어른의 사정이나, 프로토 타이핑 등을 위해서 불가피하게...

정말 급박한 상황이라면 지름길을 통해서 빠르게 개발하자.

하지만 창문이 더 깨지고 차가 박살 나는 걸 막기 위해  
다 같이 '임시로 갈 수 있는 지름길'을 명시적으로 정해 놓자.

'지금은 이 지름길을 사용할 수 있지만,  
이건 비포장 도로이니 나중에 꼭 다시 포장하자'

모두가 이렇게 인지하고 있다면 차가 완전히 박살 나는 건 막을 수 있다.

# 지름길

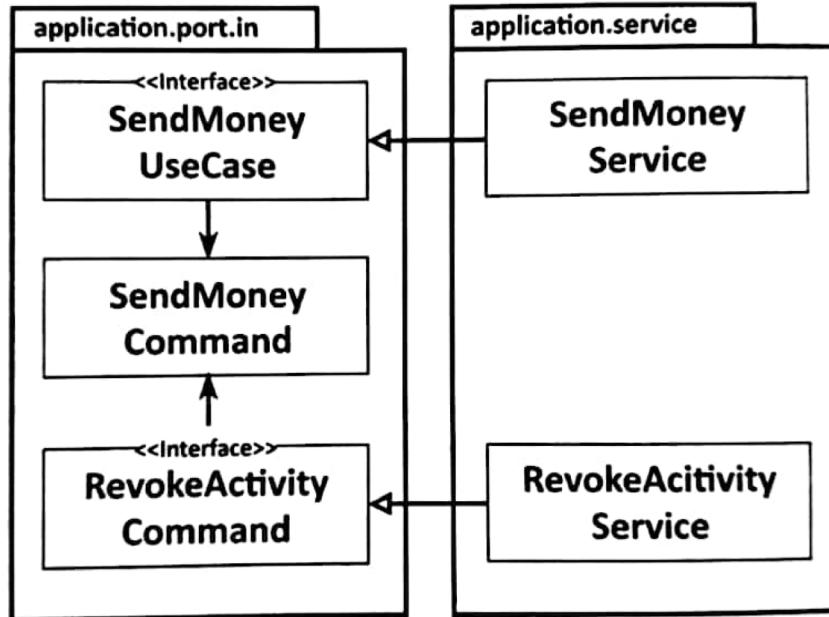


Figure 11.1: Sharing the input or output model between use cases leads to coupling between the use cases

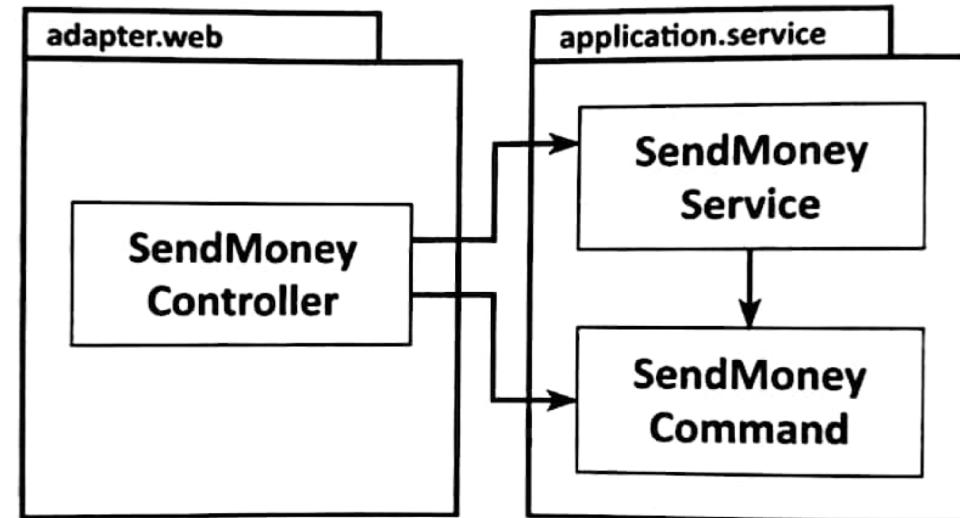


Figure 11.3: Without incoming ports, we lose clearly marked entry points to the domain logic

# 지름길

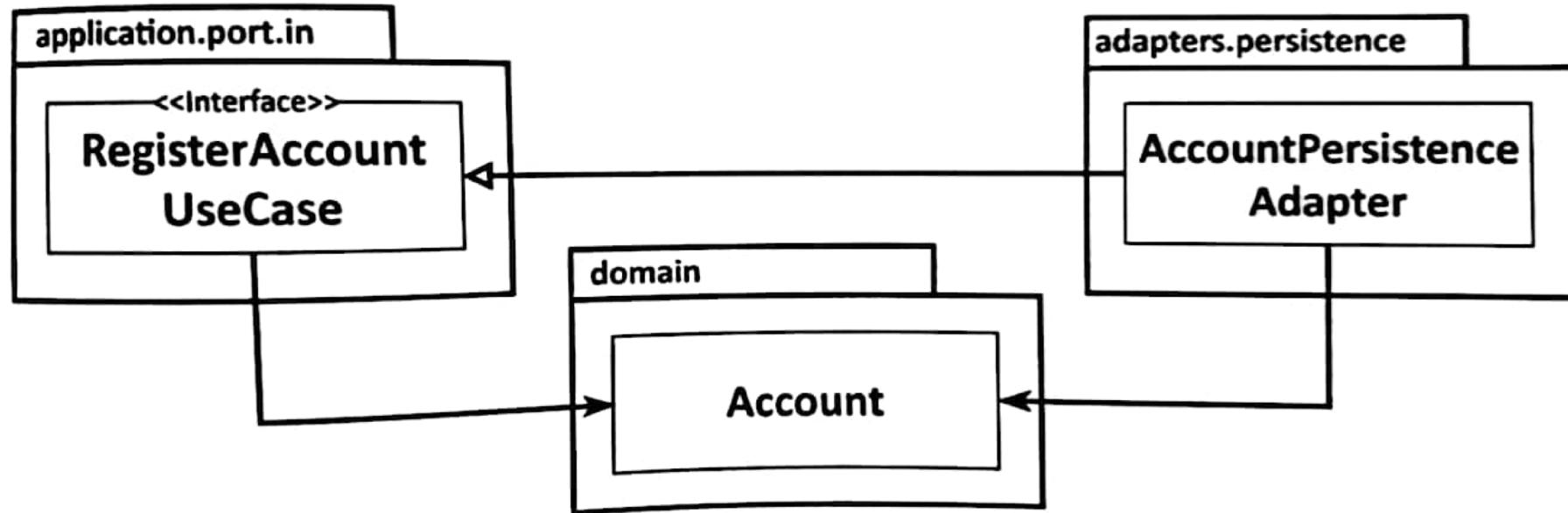


Figure 11.4: Without application services, we don't have a specified location for domain logic

왠지 php가 생각나네요...?

(Apache, Php, My-sql이여 영원하라!)

# 레이어 격리를 유지하기 위한 장치

## Boundaries and Dependencies

Before we talk about different ways of enforcing architecture boundaries, let's discuss where the boundaries lie within our architecture and what "enforcing a boundary" actually means:

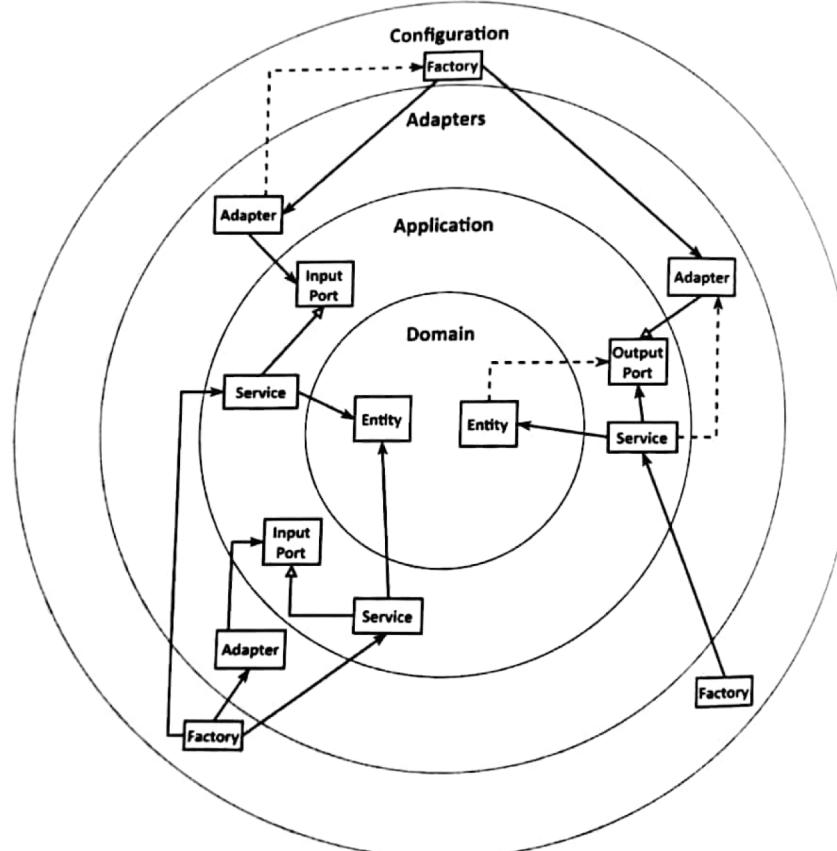
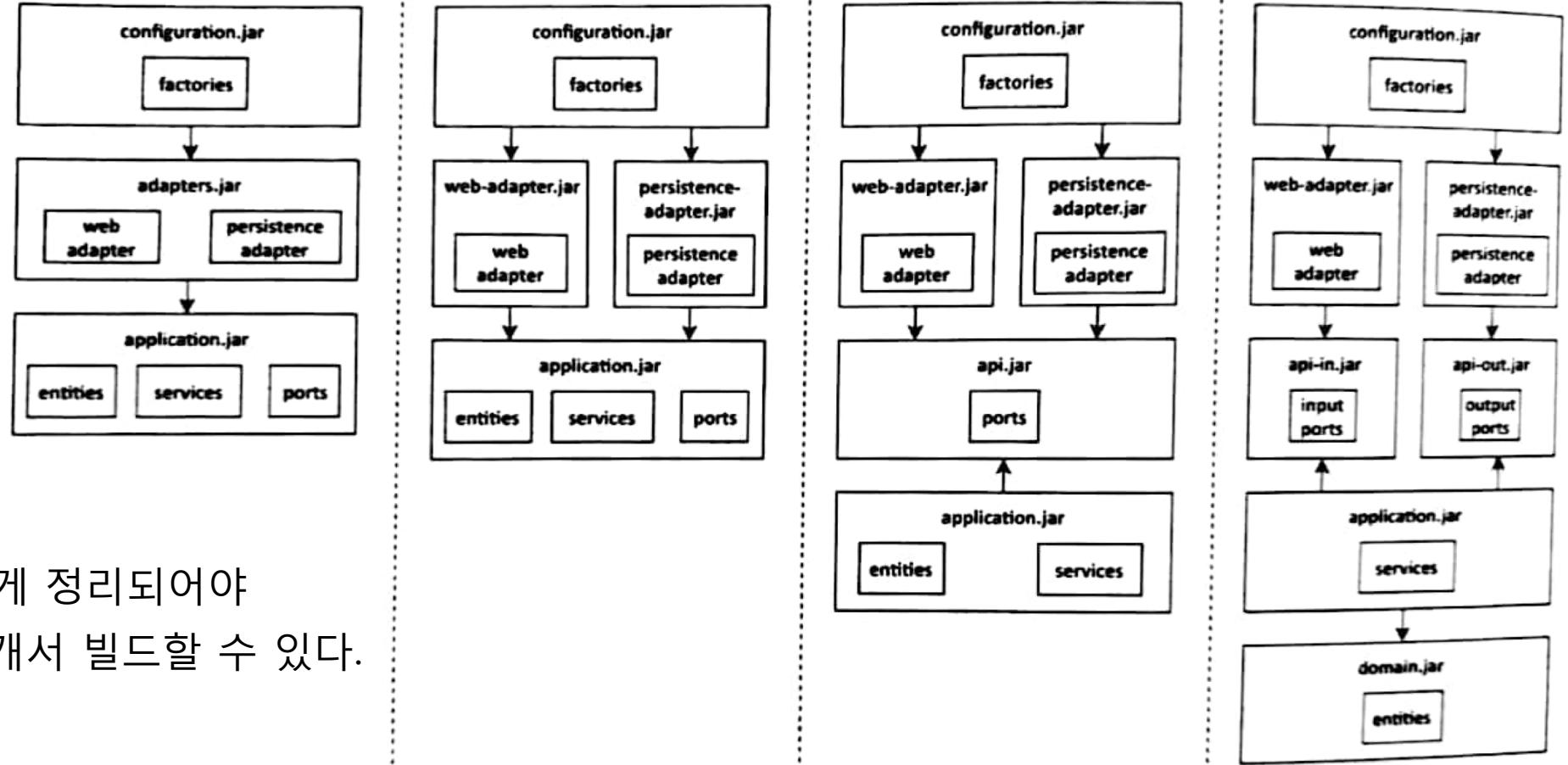


Figure 10.1: Enforcing architecture boundaries means enforcing that dependencies point in the right direction. Dashed arrows mark dependencies that are not allowed according to our architecture

의존성은 오직  
내부를 향합니다!

# 레이어 격리를 유지하기 위한 장치



의존성이 명확하게 정리되어야

각 레이어를 모듈로 쪼개서 빌드할 수 있다.

Figure 10.3: Different ways of dividing our architecture into multiple build artifacts to prohibit illegal dependencies

# 뭐가 가장 나은 선택일까? – Domain Is King

136 | Deciding on an Architecture Style

## The Domain is King

It should have become obvious in the previous chapters that the main feature of a hexagonal architecture style is that we can develop the domain code free from diversions such as persistence concerns and dependencies upon external systems.

*Evolving domain code free from external influence is the single most important argument for the hexagonal architecture style.*

This is why this architecture style is such a good match for Domain-Driven Design (DDD) practices. To state the obvious, in DDD the domain drives the development. And we can best reason about the domain if we don't have to think about persistence concerns and other technical aspects at the same time.

I would even go so far as to say that *domain-centric architecture styles such as the hexagonal style are enablers of DDD*. Without an architecture that puts the domain into the center of things, without inverting the dependencies toward the domain code, we have no chance of really doing DDD; the design will always be driven by other factors.

So, as a first indicator of whether to use the architecture style presented in this book or not, if the domain code is not the most important thing in your application, you probably don't need this architecture style.

어떤 외부 시스템 (DB 등등...)에도 의존하지 않는  
DDD를 구현해야 한다면 헥사고널 아키텍처가 좋은 대안이다.

# 뭐가 가장 나은 선택일까? – Experience is Queen

## Experience is Queen

We are creatures of habit. Habits automate decisions for us so we don't have to spend time on them. If there's a lion running toward us, we run. If we build a new web application, we use the layered architecture style. We have done it so often in the past that it has become a habit.

I'm not saying that this is necessarily a bad decision. Habits are just as good at helping us to make the right decision as they are at making a bad one. I'm saying that we are doing what we are experienced in. We are comfortable with what we have done in the past, so why should we change anything?

So, the only way to make an educated decision about an architecture style is by having experience in different architecture styles. If you are unsure about the hexagonal architecture style, try it out on a small module of the application that you are currently building. Get used to the concepts and get comfortable. Apply the ideas in this book, modify them, and add your own ideas to develop a style you are comfortable with.

This experience can then guide your next architecture decision.

인간은 습관의 동물이고, 습관을 따르면 빨리 달릴 수 있다.

등 뒤에 사자가 쫓아오고 있다면 머뭇거릴 것인가? 그럴 때는 습관에 따라 움직여야 살아남는다.

만약, 이런 급박한 상황에서도 핵사고널 아키텍처를 적용하고 싶다면 작은 모듈에서부터 시작해보자.

작은 모듈에서 익숙해졌다면 조금씩 더 적용해보자.

# 뭐가 가장 나은 선택일까? – It Depends

It Depends | 137

## It Depends

I would love to provide a list of multiple-choice questions to decide on an architecture style just like all those "Which Personality Type Are You?" and "What Kind of Dog Are You?" tests that regularly swirl around social media. I'm the "Defender" personality type and if I were a dog, I would apparently be a Pit bull.

But it isn't as easy as that. My answer to the question of which architecture style to choose remains the professional consultant's "It depends....". It depends on the type of software to be built. It depends on the role of the domain code. It depends on the experience of the team. And finally, it depends on being comfortable with a decision.

I hope, however, that this book has provided some sparks to help with the architecture question. If you have a story to tell about architecture decisions, with or without hexagonal architecture, I'd love to hear about it.

You can drop me an email at [tom@reflectoring.io](mailto:tom@reflectoring.io).

나는 결정을 내리기 보다 여러 선택지를 제시하는 걸 좋아한다. 아키텍처를 선택해야 할 때도 마찬가지다.

'어떤 종류의 소프트웨어를 만들어야 하는지',

'도메인 코드의 역할이 어떤 것인지',

'개발팀의 역량이 어느정도인지'를 고려해서 최선의 대안을 제시한다.

# 더 읽어볼 거리들



gradlew.bat first working example with use case and persistence adapter 2 years ago No packages published

Contributors 2 thombergs Tom Hombergs mdeinum Marten Deinum

Languages Java 100.0%

**Example Implementation of a Hexagonal Architecture**

CI passing

Get Your Hands Dirty on Clean Architecture  
A Hands-on Guide to Creating Clean Web Applications with Code Examples in Java  
Tom Hombergs

This is the companion code to my eBook [Get Your Hands Dirty on Clean Architecture](#). It implements a domain-centric "Hexagonal" approach of a common web application with Java and Spring Boot.

**Companion Articles**

- [Hexagonal Architecture with Java and Spring](#)
- [Building a Multi-Module Spring Boot Application with Gradle](#)

**Prerequisites**

- JDK 11
- this project uses Lombok, so enable annotation processing in your IDE

모든 소스 코드는

<https://github.com/thombergs/buckpal>

에서 볼 수 있습니다.