Overengineering in Concentric Architectures

Clean Architecture, Onion Architecture, and Hexagonal Architecture (aka Ports-and-Adapters) have become the norm for backend system design today. Powerful influencers have promoted these architectures without stressing enough that they are (overly)complex, exhaustive blueprints that need to be simplified to the actual problem at hand. Applying these architectures 'by the book' can lead to overengineering that causes useless development effort and risks and can impede a deep strategical refactoring on a different axis tomorrow. The purpose of this article is to point to common places in which simplifications can be made to these software architectures, explaining the tradeoffs involved.

Throughout the history of software architectures, many influencers have stressed the importance of keeping the peripheral **Integration** concerns away from the central **Domain** area that handles the main complexity addressed by the application. This culminated with the rise of Domain-Driven Design that emphasized keeping your Domain supple, constantly looking for ways to deepen it, distill it and refactor it to better ways of modeling your problem.

The 3 styles of architecture above can be referred to as "Concentric Architectures", as they organize code not in layers but in rings, concentric around the Domain. Note: in this article, the terms *layer* and *ring* are used interchangeably.

The pragmatic variations in this article might help you simplify the design of your application, but each **requires the entire team to understand code smells and design values** (cohesion, coupling, DRY, OOP, keep core logic in an agnostic domain). Any move you plan to take, start small and constantly weigh simplifications vs heterogeneity introduced in the code base.

Useless Interfaces

Throughout my career, I've heard different variations of the ideas below (**all are wrong**):

> *Domain Entities must be used only through the interfaces they implement*
>
> *Every Layer must expose and consume only interfaces*. (in Layered Architecture)
>
> *Application Layer must implement Input Port interfaces*. (in Hexagonal Architecture)

Besides the superfluous interface files that have to be kept in sync with changes in the signatures of their implementation, navigating code in such a project can be both mysterious (is there another implementation?) as well as frustrating (ctrl-click on a method might throw you into an interface instead of the method definition).

But why create more interfaces in the first place?

The excessive use of interfaces may originate from a very old Design Principle: "*Depend on Abstractions, rather than Implementations*". The **first level of understanding** of this principle suggests using more interfaces and abstract classes to allow swapping in a different implementation via polymorphism at runtime. The **second level of understanding** is much more powerful but has nothing to do with `interface` keyword: it speaks about breaking down the problem into subproblems (abstractions) on top of which to build a simpler, cleaner solution (eg TCP/IP stack). But let's focus on the former for this discussion.

Using interfaces allows swapping in a different implementation at runtime:

alternative implementations of the same contract (aka Strategy Design Pattern)

chains of "Filters" (aka Chain of Responsibility Pattern), eg web/security filters

test Fakes (almost eradicated by modern mocking frameworks): MyRepoFake implements IMyRepo

enriched variations of implementations (Decorator Pattern): Collections.*unmodifiableList*()

proxies to concrete implementations (Note: JVM-based languages do NOT require interfaces to proxy concrete classes)

All of the above are ways to increase the flexibility of your design using polymorphism. However, it is a mistake to introduce these patterns before they are actually used. In other words, be wary of thoughts like "*Let's introduce an interface there just in case*", as this is the very definition of the Speculative Generality code smell. Designing in advance perhaps made more sense decades ago when editing code required a lot of effort. But tools have evolved enormously since then. Using the refactoring tools of a modern IDE (eg IntelliJ), you can extract an interface from an existing class and replace the class references with the new interface in the entire codebase, in a matter of seconds with almost no risk. Having such a tool should encourage us to extract interfaces only when really needed.

Question any interface with a single implementation, in the same module.

As a matter of fact, the only reason to tolerate an interface with a single implementation is when the interface resides in a different compilation unit than its implementation:

1. In a library used by your clients, eg `my-api-client.jar`
2. Interface in an inner ring, implementation in an outer ring = Dependency Inversion Principle

The dependency Inversion Principle (a hallmark of any concentric architecture) allows the Inner Ring (eg Domain) to call methods in an Outer Ring (eg Infrastructure), but without being coupled to its implementation. For example, a Service inside the Domain Ring can retrieve information from another system by calling methods of an Interface declared in the Domain Ring but implemented in the outer Infrastructure Ring. In other words, Domain is able to call Infrastructure but without seeing the actual method implementation invoked. The direction of the call (Domain→Infra) and the direction of the code dependency (Infra→Domain) are inverted, thus its name, “Dependency Inversion”.

Here’s one more (tragi-comic) argument in favor of interfaces that I once heard:
– Victor, we need interfaces to clarify the public contract of our classes!
– But what’s wrong with looking at the class structure for public methods? I replied
– Yeah… But there are over 50 public methods in that class, so we like to have them listed in a separate file, grouped nicely.

– me: (speechless)…
– Oh, and btw, the implementation class has over 2000 lines of code. 😱
I think it’s obvious that the useless interface was NOT their **real** problem…

In conclusion:

An interface deserves to exist ***if and only if****:*

1. ***it has more than one implementation*** *in the project, or*
2. *it is used to implement* ***Dependency Inversion*** *to protect an Inner Ring, or*
3. *it is packaged in a* ***client library***

If an interface does not match either of the arguments above, consider destroying it (eg by using the “Inline” refactoring in IntelliJ).

But let’s come back to the opening quotes:

Interfaces for Domain Entities – are wrong! None of the 3 reasons above apply:

1. Having alternative implementations of Entities is absurd
2. No code is more precious than your Domain (Dep Inversion against Domain?!)

3. Exposing Entities in your API is a very dangerous move

Input Port interfaces in Hexagonal Architecture – are wrong!

1. They have a single implementation (the Application itself)
2. The API controller does NOT need to be protected (it is an outer ring)
3. If you expose the Input Ports interfaces directly, that’s no longer the classic Hexagonal Architecture.

Strict Layers

The proponents of this approach state that:

> *“Each layer/ring should only call the immediate next one”*

For this argument, let’s assume these 4 layers/rings:

1. Controller
2. Application Service
3. Domain Service
4. Repository

If we enforce Strict Layers, Controller(1) should only talk to Application Service(2) → Domain Service(3) → Repository(4). In other words, Application Service(2) is NOT allowed to talk directly to the Repository(4); the call must always go through a Domain Service(3). Taking this decision leads to boilerplate methods like this:

```
class CustomerService {
  ..
  public Customer findById(Long id) {
    return customerRepo.findById(id);
  }
}
```

The code above is almost a textbook example of a Code Smell called Middle Man, as this method represents “indirection without abstraction” – it does not add any new semantic (abstraction) to the method to which it delegates (customerRepo.findById). The method above does not improve code clarity, instead, it just adds one extra ‘hop’ in the call chain.

The (few) proponents of Strict Layers today argue that this rule requires taking fewer decisions (desirable for mid-junior teams), and reduces the coupling of the upper layers. However, one key role of the Application Service is to orchestrate the use case, so high coupling to the parts involved is usually expected. To put it another way, it is a design goal of the Application Service to host orchestration logic to allow lower-level components to become less coupled to one another.

The alternative to Strict Layers is called “**Relaxed Layers**“, and it allows skipping layers as long as the calls go in the same direction. For example, an Application Service(2) is free to call a Repository(4) directly, while at other times it might call through a Domain Service(3) first, if there’s any logic to push into the DS. This can lead to less boilerplate but does require a habit of refactoring to continuously extract logic growing complex into Domain Services.

One-liner REST Controller Methods

For more than a decade, we all agreed that:

> *The responsibility of the REST Controller layer is to handle the HTTP concerns and then delegate all logic to the Application Service*

Indeed, that made perfect sense 10-20 years ago. When generating HTML webpages on the server side (think .jsp + Struts2), that called for quite a lot of ceremony in the Controller. But today, if they talk over HTTP, our apps and microservices usually only expose a REST API, pushing all the screen logic into a front-end/mobile app. Furthermore, the frameworks (eg Spring) we use today evolved so much that if used carefully can reduce the responsibility of the controller to just a series of annotations.

The HTTP-related responsibility of a REST Controller today can be summarized as:

- Map HTTP requests to methods – via annotations (@GetMapping)
- Authorize user action – via annotations (@Secured, @PreAuthorized)
- Validate the request payload – via annotations (@Validated)

Read/Write HTTP request/response headers – best done via a web Filter

Set the Response status code – best done in a global exception handler (@RestControllerAdvice)

Handle file upload/download – not fancy anymore, but the only really ugly HTTP-related thing that's left

Note: The examples provided above are from Spring Framework (the widest-used framework in Java), but there are equivalents for almost all features in other Java frameworks and other web-potent languages.

Therefore, unless you are uploading some files or doing some other HTTP kung-fu (WHY?!), there should be no more HTTP-related logic in your REST Controllers – the framework extinguished it. If we assume the data conversion (Dto ↔ Domain) does NOT happen in the Controller, but in the next layer, for example in the Application Layer, then the methods of a REST Controller will be one-liners, delegating each call to a method with a similar name in the next layer:

```
@RestController
class WhiteController {
  ..
  @GetMapping("/white")
  public WhiteDto getWhite() {
    return whiteService.getWhite();
  }
}
```

Oh no! It’s again the “Middle Man” code smell we saw before – boilerplate code.

If what you read above matches your setup, you can consider merging your controller with the next layer (eg Application Service).

> *Merge Controllers with Application Services, when developing a REST API.*

Yes, I mean annotating as HTTP endpoints (@GetMapping..) the methods in the first layer holding logic, be it Application Service or just “Service” (in our example, in WhiteService).

< *awkward silence* >

I know, it goes against some very old habits we followed until just a while ago. Probably the more years you’ve spent in the field, the more strange this decision feels. But times have changed. In a system exposing a REST API, things can (and should) be simplified more.

In the workshops I run, I’ve often encountered architectures that allowed the REST Controller to contain logic: mapping, more elaborate validations, orchestration logic, and even bits of business logic. This is an equivalent solution, in which the Application Service was technically merged ‘up’ with the Controller in front of it. Both are equally good: (a) having a Controller do bits of application logic or (b) exposing an Application Service as a REST API.

⚠️ There is one trap though: **don't accumulate complex business rules in the REST API component**. Instead, constantly look for cohesive bits of domain logic to move to the Domain Model (eg in an Entity) or into a Domain Service.

One last note: if you are heavily annotating your REST endpoint methods for documentation purposes (OpenAPI stuff), you could consider extracting an interface and moving all the REST annotations to it. The Application Service will then implement that interface and take over that metadata.

Mock-full Tests

Unit Testing is king. Professional developers thoroughly unit-test their code. Therefore:

> *Each layer should be tested by mocking the layer below*

When the architecture mandates **Strict Layers** or allows **One-liner REST Controller Methods** (even if data conversion is performed in the Controllers), a rigorous team would ask the obvious question: should those ~~stupid~~ one-liner methods be unit-tested? How? Testing such silly methods with mocks would lead to 5x times larger test code than tested code. Worse, it would *feel* useless – what are the chances that a bug would occur inside such a method?

People might get frustrated ("*Testing sucks!*🤬") or worse, relax their testing strictness ("*Let's not test THIS ONE*🤫").

In reality, what you are facing is **honest feedback from your tests** – your system is over-engineered. A seasoned TDD practitioner will quickly pick up the idea, but others will have a hard time accepting it: “**when testing is hard, the production design can be improved**“.

For completeness, I’d like to add one more suggestion to the classic statement above:

> *When testing is hard, the production design can be improved, or you’re testing too fine-grained.*

If your integration tests fully cover that method, do you really need to *unit* test it in isolation? Read about Honeycomb Testing for microservices and explore the idea that Unit Testing is just a necessary evil. When testing microservices, test from outside-in: start with Integration Testing > then Unit Testing (to cover corner cases).

Separate Application DTOs vs REST DTOs

When studying the Clean Architecture by Uncle Bob, many people get the impression that the data structures exposed by the REST Endpoints should be different structures from the objects exposed by the Application ring (let’s call them “ApplicationDto”). That is, have an additional set of classes, transformed from one into the other. It’s usually quick to realize the huge price of that decision and most engineers give up. Others, however, insist, and every field they add later to a data structure has to be added now to 3 data structures: one in the Dto sent/received as JSON, one in the ApplicationDto object, and one in the Domain Entity that persists the data.

#Don't do this!

Instead:

> *Propagate REST API DTOs into the Application layer.*

Yes, the Application Service would have a harder time working with API models, but that's one more reason to keep it light, stripped of heavy domain complexity.

But is there a valid reason to have an ApplicationDto separated from the REST API DTO?

Yes, there is. In case the same use-case is exposed over 2 or more channels: for example over REST, Kafka, gRPC, WSDL, RMI, RSock, Server-side HTML (eg Vaadin,..) etc. For those use cases, it makes sense to have your Application Service speak its own data structures (language) via its public methods. Then, the REST Controller will convert them to/from REST API DTO, while (eg) the gRPC endpoint will convert to/from its own protobuf objects. When faced with such scenarios, a pragmatic engineer will probably apply this technique only for the several use cases that require it.

Separate Persistence from Domain Model

We finally reach one of the most heated debates around Clean/Onion Architecture:

> ***Should I allow Persistence concerns to pollute my Domain Model?***

In other words, should I annotate with @Entity my Domain Model and let my ORM framework save/fetch instances of my Domain Model objects?

If you answer **NO** to the questions above, then you need to:

- create a copy of all your Domain Entities outside of the Domain, eg CustomerModel (Domain) vs CustomerEntity (Persistence)
- create interfaces for repositories in the Domain that only retrieve/save Domain Entities, eg customerRepo.save(CustomerModel)
- implement the repo interfaces in the infrastructure by **converting Domain Entities to/from ORM Entities**

In short: pain! bugs! frustration!
It's one of the most expensive decisions to take, as it effectively increases the code 4 times or

more for CRUD operations.

I’ve met teams that took this path and paid the price above, but all of them were regretting their decision 1-2 years later, spare a few exceptions.

But what makes respected tech leaders and architects take such an expensive decision?

What is the danger of using an ORM?

Using a framework as powerful as an ORM never comes for free.

Here are the main pitfalls of using an ORM:

1. **Magic Features**: auto-flush dirty changes, write-behind, lazy loading, transaction propagation, PK assignment, merge(), orphanRemoval, ...
2. **Non-obvious Performance Issues** that might hurt you later: N+1 Queries, lazy loading, fetching useless amounts of data, naive OOP modeling,..
3. **Database-centric Modeling Mindset**: thinking in terms of tables and Foreign Keys makes it harder to think about your Domain at higher levels of abstraction, finding better ways to remodel it (Domain Distillation)

It’s not prudent to just ignore the points above. After all, they concern the deepest, most precious part of your application: the Domain Model.

So here’s what you can do about them:

1) Magic Features: **Learn or Avoid.** The amount of surprise in the audience during my JPA workshops has always concerned me – people driving a Ferrari without a driver's license. After all, Hibernate/JPA's features are far more complex than the ones you find in Spring Framework. But somehow, everyone assumes they know JPA. Until magic strikes you with a dark bug.

You can also block some of the magic features, for example, detaching entities from Repo and not keeping transactions open, but such approaches usually incur a performance impact and/or unpleasant corner cases. So it's better to make sure your entire team masters JPA.

2) Monitor Performance early: there are commercial tools available to help a less experienced team detect typical performance issues associated with ORM usage, but the easiest way to prevent them (and learn at the same time) is to keep an eye on the generated SQL statements with tools like Glowroot, p6spy, or any other way to log the executed JDBC statements.

3) Model your Domain in Object-Oriented. Even if you are using incremental scripts (and you should!), be always on the lookout for model refactoring ideas, and try them out by allowing your ORM to generate the schema while exploring. Object-Oriented reasoning can always provide far better levels of Domain insight than thinking in terms of tables, columns and foreign keys.

To summarize, my experience showed me that it's cheaper and easier for a team in the long run to learn ORM than to run away from it. So allow ORM on your Domain but learn it!

Application Layer decoupled from Infrastructure Layer

In the original Onion Architecture, the Application Layer is decoupled from the Infrastructure Layer, aiming to keep ApplicationService logic separated from 3rd party APIs. Every time we want to get some data from a 3rd party API from the Application Layer, we would need to create new data objects + a new interface in the Application Layer, and then implement it in the Infrastructure (the Dependency Inversion Principle we mentioned earlier). That's quite a high price for decoupling.

But since most of our business rules are implemented in the Domain layer, the Application layer is left to orchestrate the use-cases, without doing much logic itself. Therefore, the risk of manipulating 3rd party DTOs in the Application Layer is tolerable, compared with the cost of decoupling.

Merge Application with Infrastructure Layer.
= "Pragmatic Onion" 😏

Merging the Application with Infrastructure Layer allows free access to 3rd party API DTOs from ApplicationServices.

This way, you end up with only 2 layers/rings: Application (including Infrastructure) and Domain.

The risk: if core business rules are not pushed into the Domain Layer, but kept in the Application Layer, their implementation might be polluted and become dependent on 3rd party DTOs and APIs. So the Application Services should be stripped of business rules even harder.

There is an interesting corner case here. If you lack persistent data (if you don't store much data), then you don't have an obvious Domain Model to map to/from the 3rd party data. In such systems, if there is serious logic to implement on top of those 3rd party objects, at some point it might be worth creating your own data structures to which to map the external DTOs, so that you have control over the structures you write your complex logic with.

Not Enough Domain Complexity

Like any tool, concentric architectures are not suitable for any software project. If the domain complexity of your problem is fairly low (CRUD-like), or if the challenge of your application is NOT in the complexity of its business rules, then the Onion/Hexagonal/Ports-Adapters/Clean Architecture might not be the best choice, and you might be better off with vertical slicing, anemic model, CQRS, or another type of architecture.

Conclusion

In this article, we looked at the following sources of overengineering, waste, and bugs when applying the Concentric Architectures (Onion/Hexagonal/Clean-)

– **Useless Interfaces** => remove them unless ≥2 implementations or Dependency Inversion

– **Strict Layers** => Relaxed Layers = allow calls to skip layers while going in the same direction

– **One-liner REST Controllers** => merge them with the next layer, but keep their complexity under control

– **Mock-ful Tests** => Collapse layers or Test a larger chunk

– **Separate Application<>Controller Dtos** => use a single set of objects unless multiple output channels

– **Separate Persistence from Domain Model** => don't! Use a single model, but learn the ORM magic, and performance pifalls.

– **Decouple Application from Infrastructure** => merge Application+Infrastructure Layer, but push harder business rules in the Domain