Table of Contents

Where and How Does DDD Fit?

> We won't be distracted by comparison if we are captivated with purpose.
>
> — Bob Goff

Software architecture refers to the fundamental structures of a software system and the discipline of creating such structures and systems. Over the years, we have accumulated a series of architecture styles and programming paradigms to help us deal with system complexity. In this chapter we will examine how DDD can be applied in a manner that is complementary to these architecture styles and programming paradigms. We will also look at how/where it fits in the overall scheme of things when crafting a software solution.

At the end of this chapter, you will gain an appreciation of a variety of architecture style and programming paradigms, along with some pitfalls to watch out for, when applying them. You will also understand the role that DDD plays in augmenting each of these.

Architecture Styles

Domain-driven design presents a set of architecture tenets in the form of the strategic and tactical design elements. This enables decomposing large, potentially unwieldy business subdomains into well-factored, independent bounded contexts. One of the great advantages of DDD is that it does not require the use of any specific architecture. However, the software industry has been using a plethora of architecture styles over a period of the last several years. Let's look at how DDD can be

used in conjunction with a set of popular architecture styles to arrive at better solutions.

Layered Architecture

The layered architecture is one of the most common architecture styles where the solution is typically organized into four broad categories: **presentation**, **application**, **domain** and **persistence**. Each of the layers provides a solution to a particular concern it represents as shown here:

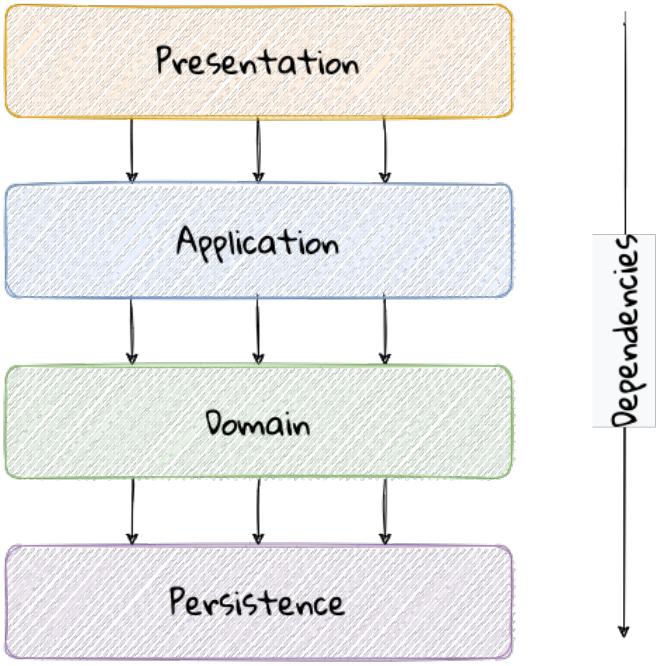

Figure 1. Essence of a layered architecture.

The main idea behind the layered architecture is a separation of concerns — where the dependencies between layers are unidirectional (from the top to the bottom). For example, the domain layer can depend on the persistence layer, not the other way round. In addition, any given layer typically accesses the layer immediately beneath it without bypassing layers in between. For example, the presentation layer may access the domain layer only through the application layer.

This structure enables looser coupling between layers and allows them to evolve independently of each other. The idea of the layered architecture fits very well with domain-driven design's tactical design elements as depicted here:

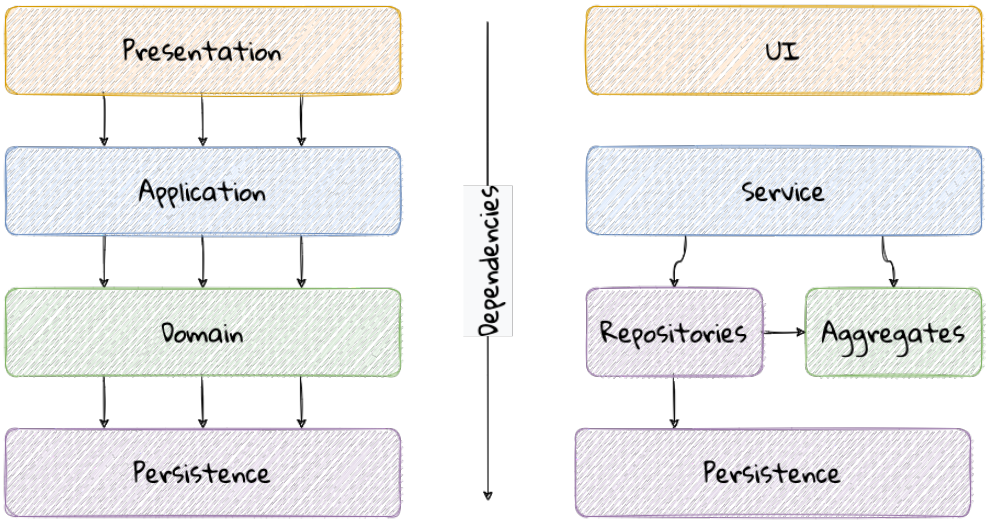

Figure 2. Layered architecture mapped to DDD's tactical design elements.

DDD actively promotes the use of a layered architecture, primarily because it makes it possible to focus on the domain layer in isolation of other concerns like how to information gets displayed, how end-to-end flows are managed, how data is stored and retrieved, etc. From that perspective, solutions that apply DDD tend to naturally be layered as well.

Notable variations

A variation of the layered architecture was invented by Alistair Cockburn, which he originally called the *hexagonal architecture*[1] (alternatively called the ports and adapters architecture). The idea behind this style was to avoid inadvertent dependencies between layers (as could occur in the layered architecture), specifically between the core of the system and the peripheral layers. The main idea here is to make use of interfaces (*ports*) exclusively within the core to enable modern drivers such as testing and looser coupling. This allows the core to be developed and evolved independently of the non-core parts and the external dependencies. Integration with real-world components such as a database, file systems, web services, etc. is achieved through concrete implementations of the *ports* termed as *adapters*. The use of interfaces within the core enables much easier testing of the core in isolation of the rest of the system using mocks and stubs. It is also common to use dependency injection frameworks to dynamically swap out implementations of these interfaces when working with the real system in an end-to-end environment. A visual representation of the hexagonal architecture is shown here:

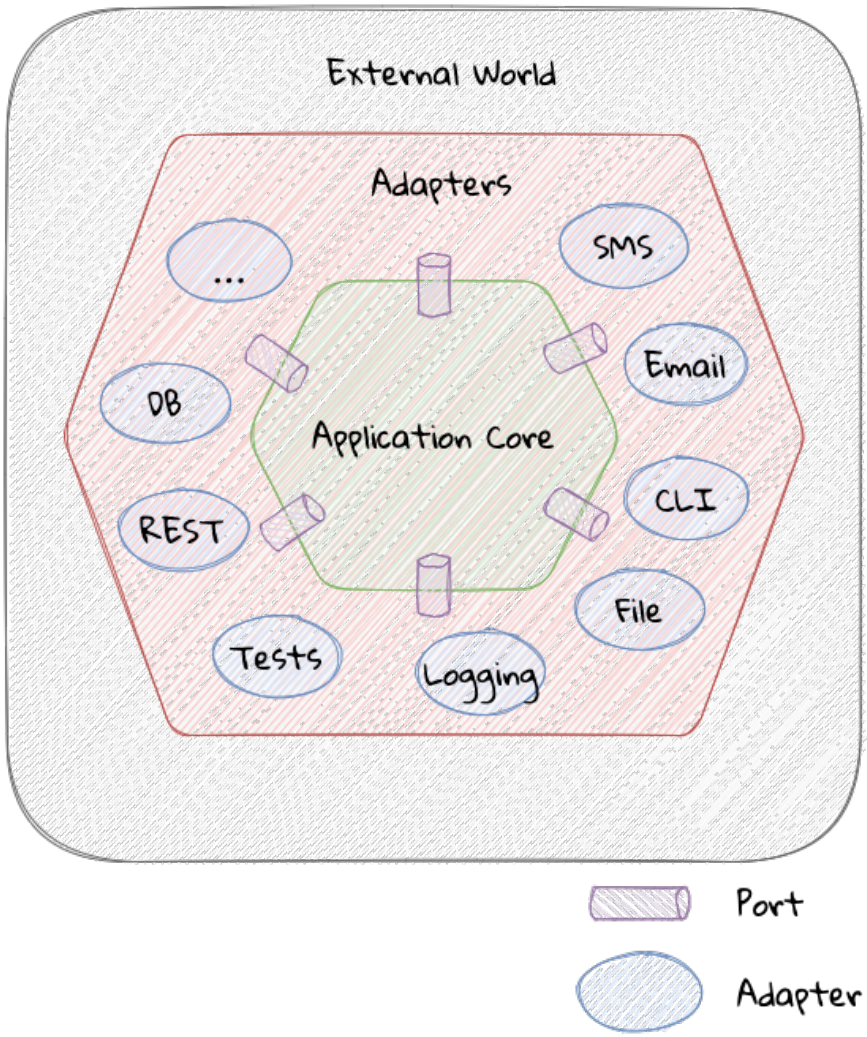

Figure 3. Hexagonal architecture

It turns out that the use of the term hexagon in this context was purely for visual purposes — not to limit the system to exactly six types of ports.

Similar to the hexagonal architecture, the onion architecture[2], conceived by Jeffrey Palermo is based on creating an application based on an independent object model within the core that can be compiled and run separately from the outer layers. This is done by defining interfaces (called ports in the hexagonal architecture) in the core and implementing (called adapters in the hexagonal architecture) them in the outer layers. From our perspective, the hexagonal and onion architecture styles have no perceptible differences that we could identify.

A visual representation of the onion architecture is shown here:

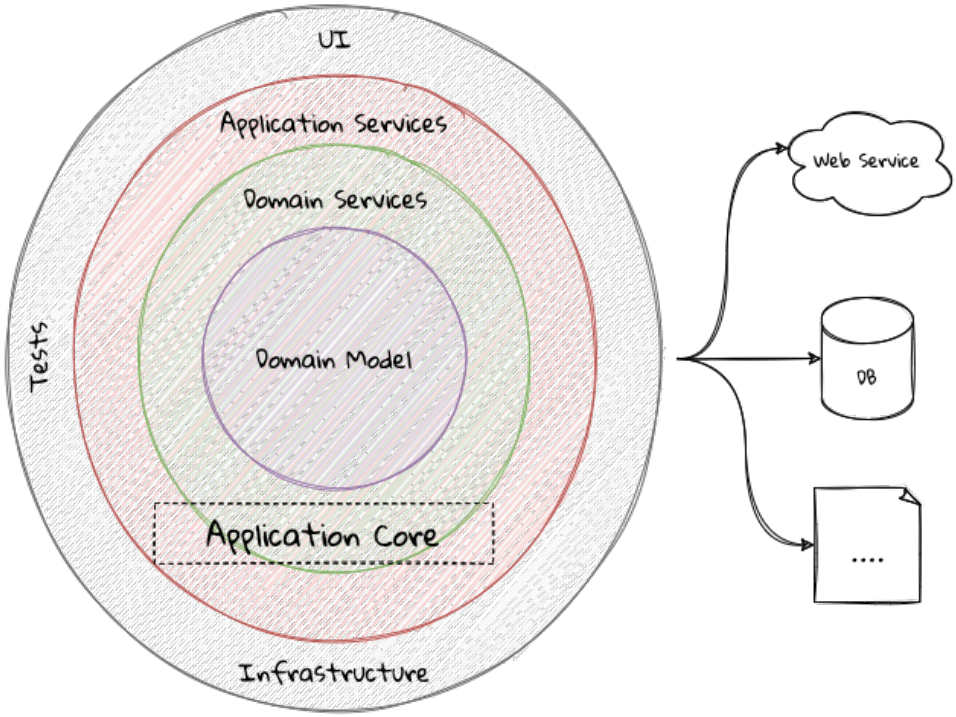

Figure 4. Onion architecture

Yet another variation of the layered architecture, popularized by Robert C. Martin (known endearingly as Uncle Bob) is the clean architecture. This is based on adhering to the SOLID principles[3] also perpetrated by him. The fundamental message here (just like in the case of hexagonal and onion architecture) is to avoid dependencies between the core — the one that houses business logic and other layers that tend to be volatile (like frameworks, third-party libraries, UIs, databases, etc).

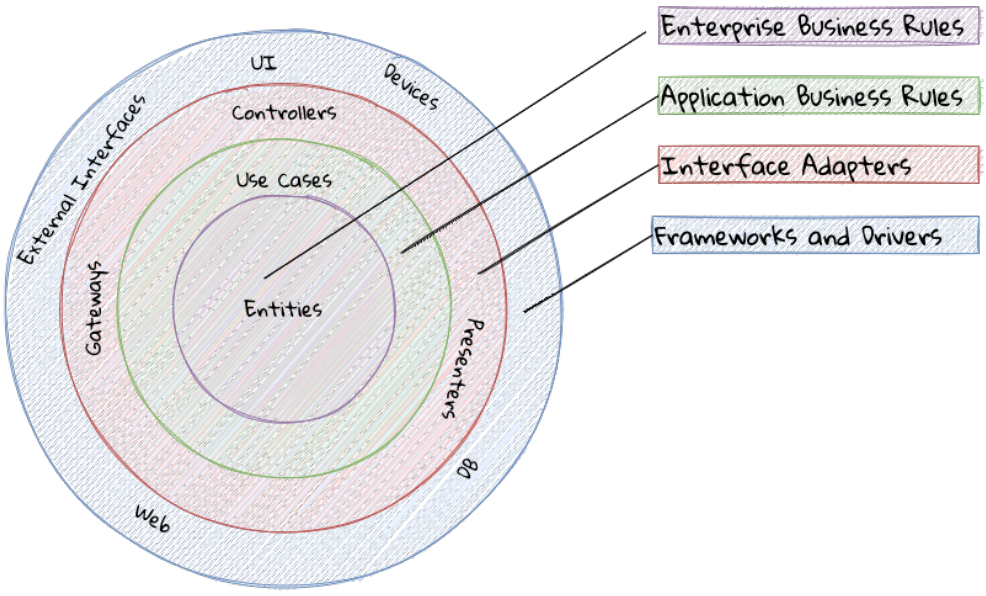

Figure 5. Clean architecture

All these architecture styles are synergistic with DDD's idea of developing the domain model for the core subdomain (and by extension its bounded context) independently of the rest of the system.

While each of these architecture styles provide additional guidance in terms of how to structure a layered architecture, you will need to be cognizant of the same considerations we described above as part of the conversation on the layered architecture.

However, any architecture approach we choose comes with its set of tradeoffs and limitations. We discuss some of these here.

Considerations

Layer cake anti-pattern

Sticking to a fixed set of layers provides a level of isolation, but in simpler cases, it may prove overkill without adding any perceptible benefit other than adherence to an agreed on architectural guidelines. In the layer cake anti-pattern, each layer merely proxies the call to the layer beneath it without adding any value. The example below illustrates this scenario that is fairly common:

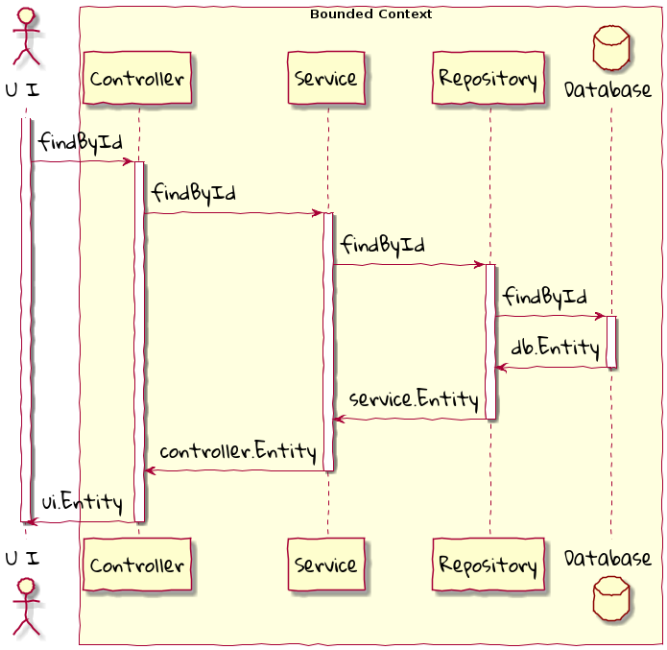

Figure 6. Example of the **layer cake** *anti-pattern to find an entity representation by ID*

Here the `findById` method is replicated in every layer and simply calls the method with the same name in the layer below with no additional logic. This introduces a level of accidental complexity to the solution. Some amount of redundancy in the layering may be unavoidable for the purposes of standardization. It may be best to re-examine the layering guidelines if the *layer cake* occurs prominently in the codebase.

Anemic translation

Another variation of the layer cake we see commonly is one where layers refuse to share input and output types in the name of higher isolation and looser coupling. This makes it necessary to perform translations at the boundary of each layer. If the objects being translated are more or less structurally identical, we have an *anemic translation*. Let's look at a variation of the `findById` example we discussed above.

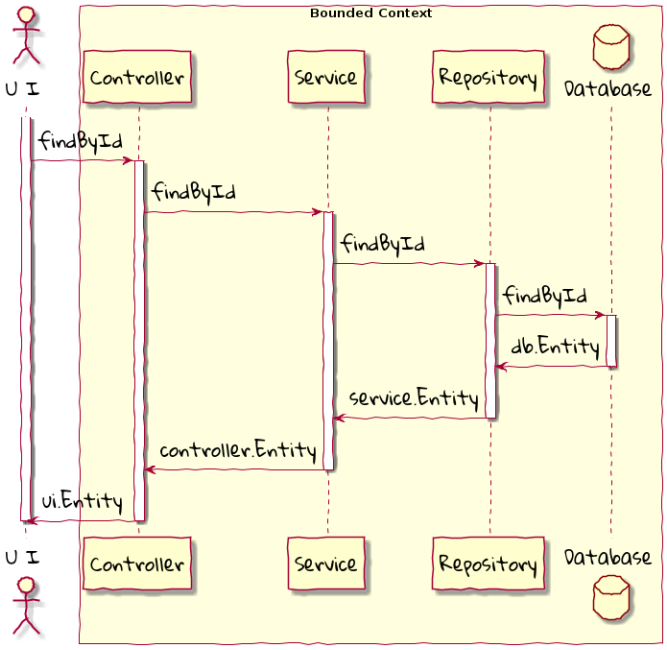

Figure 7. Example of the **anemic translation** *anti-pattern to find an entity representation by ID*

In this case, each layer defines a `Entity` type of its own, requiring a translation between types at each layer. To make matters worse, the structure of the `Entity` type may have seemingly minor variations (for example, `lastName` being referred to as `surname`). While such translations may be necessary across bounded contexts, teams should strive to avoid the need for variations in names and structures of the same concept within a single bounded context. The intentional use of the **ubiquitous language** helps avoid such scenarios.

Layer bypass

When working with a layered architecture, it is reasonable to start by being strict about layers only interacting with the layer immediately beneath it. As we have seen above, such rigid enforcements may lead to an intolerable degree of accidental complexity, especially when applied generically to a large number of use-cases. In such scenarios, it may be worth considering consciously allowing one or more layers to be bypassed. For example, the `controller` layer may be allowed to work directly with the `repository` without using the `service` layer. For example, we have found it useful to use a separate set of rules for *commands* versus *queries*.

This can be a slippery slope. To continue maintaining a level of sanity, teams should consider the use of a lightweight architecture governance tool like **ArchUnit**[4] to make agreements explicit and afford quick feedback. A simple example of how to use ArchUnit for this purpose is shown here:

```
class LayeredArchitectureTests {
    @ArchTest
    static final ArchRule layer_dependencies_are_respected_with_exception =
layeredArchitecture()

            .layer("Controllers").definedBy("..controller..")
            .layer("Services").definedBy("..service..")
            .layer("Domain").definedBy("..domain..")
            .layer("Repository").definedBy("..repository..")

            .whereLayer("Controllers").mayNotBeAccessedByAnyLayer()
            .whereLayer("Services").mayOnlyBeAccessedByLayers("Controllers")
            .whereLayer("Domain").mayOnlyBeAccessedByLayers("Services", "Repository",
"Controllers")
            .whereLayer("Repository")
                .mayOnlyBeAccessedByLayers("Services", "Controllers"); ①
}
```

① The Repository layer can be accessed by both the Services and Controllers layers — effectively allowing Controllers to bypass the use of the Services layer.

Vertical slice architecture

The layered architecture and its variants described above, provide reasonably good guidance on how to structure complex applications. The vertical slice architecture championed by Jimmy Boggard recognizes that it may be too rigid to adopt a standard layering strategy for all use cases across the entire application. Furthermore, it is important to note that business value cannot be derived by implementing any of these horizontal layers in isolation. Doing so will only result in unusable inventory and lots of unnecessary context switching until all these layers are connected. Therefore, the vertical slice architecture proposes *minimizing coupling between slices, and maximizing coupling in a slice*[5] as shown here:

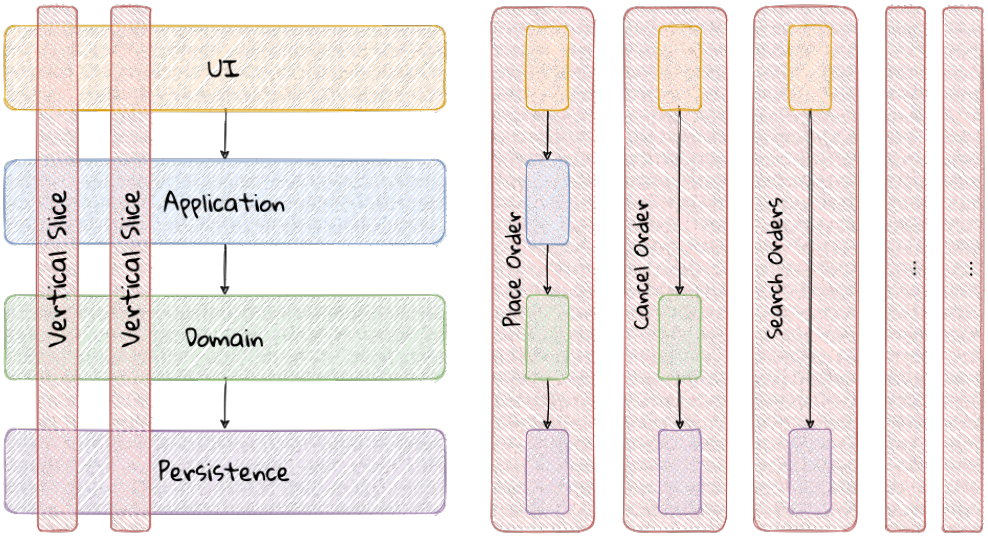

Figure 8. Vertical slice architecture

In the example above, *place order* might require us to coordinate with other components through

the application layer, apply complex business invariants while operating within the purview of an ACID transaction. Similarly, *cancel order* might require applying business invariants within an ACID transaction without any additional coordination — obviating the need for the application layer in this case. However, *search orders* might require us to simply fetch existing data from a query optimized view. This style makes use of a horses for courses approach to layering that may help alleviate some anti-patterns listed above when implementing a plain vanilla layered architecture.

Considerations

The vertical slice architecture affords a lot of flexibility when implementing a solution — taking into consideration the specific needs of the use-case being implemented. However, without some level of governance, this may quickly devolve to the big ball of mud with layering decisions being made seemingly arbitrarily based on personal preferences and experiences (or lack thereof). As a sensible default, you may want to consider using a distinct layering strategy for commands and queries. Beyond that, non-functional requirements may dictate how you may need to deviate from here. For example, you may need to bypass layers to meet performance SLAs for certain use cases.

When used pragmatically, the vertical slice architecture does enable applying DDD very effectively within each or a group of related vertical slices — allowing them to be treated as bounded contexts. We show two possibilities using the *place order* and *cancel order* examples here:

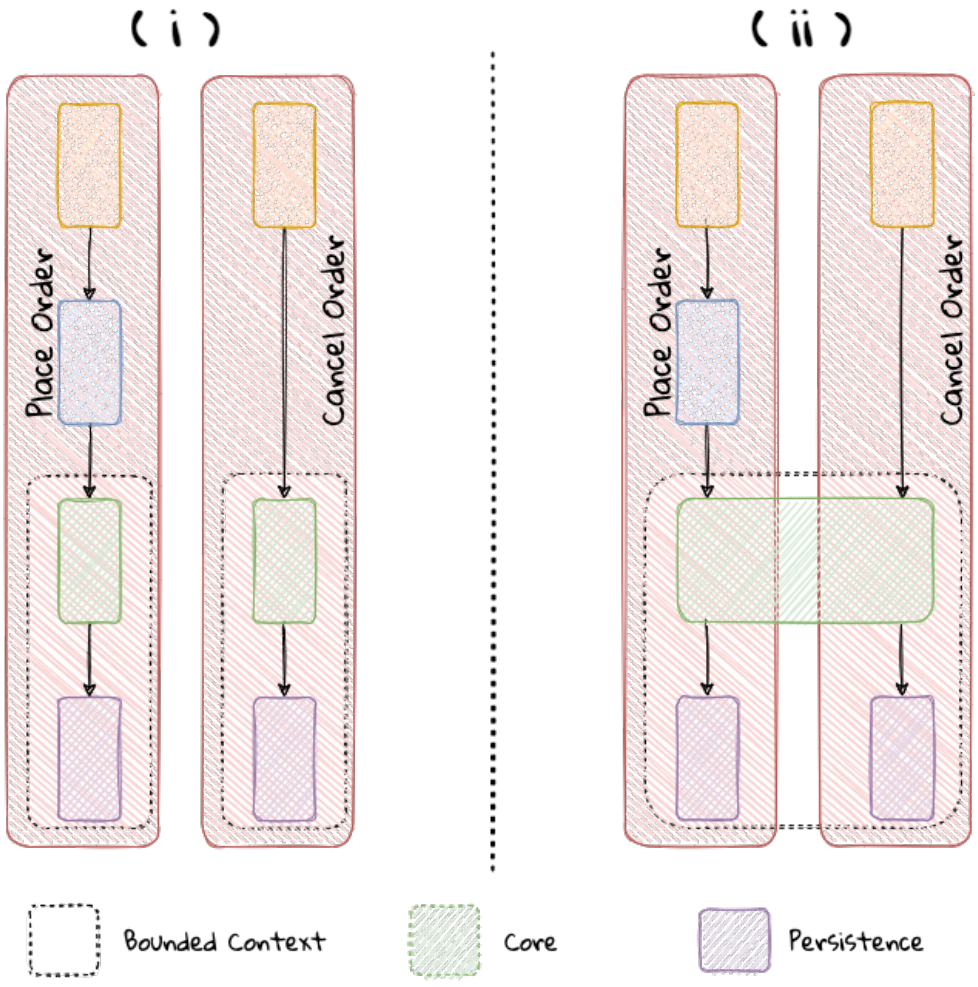

Figure 9. Vertical slices used to evolve bounded contexts

In example (i) above, *place order* and *cancel order*, each use a distinct domain model, whereas in

example (ii), both use cases share a common domain model and by extension become part of the same bounded context. This does pave the way to slice functionality when looking to adopt the serverless architecture along use case boundaries.

Service Oriented Architecture (SOA)

Service Oriented Architecture (SOA) is an architectural style where software components expose (potentially) reusable functionality over standardized interfaces. The use of standardized interfaces (such as SOAP, REST, gRPC, etc. to name a few) enables easier interoperability when integrating heterogeneous solutions as shown here:

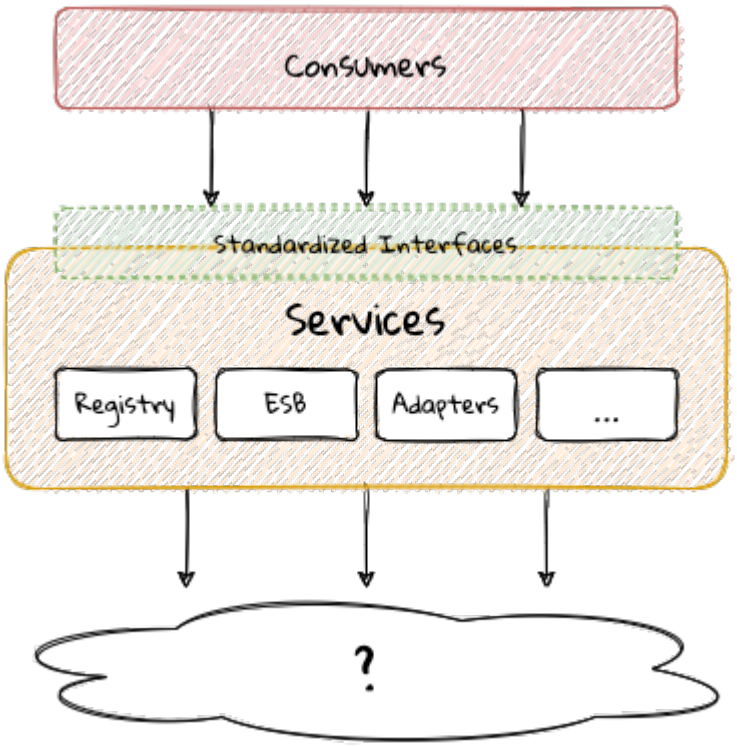

Figure 10. SOA: Expose reusable functionality over standard interfaces.

Previously, the use of non-standard, proprietary interfaces made this kind of integration a lot more challenging. For example, a retail bank may expose inter-account transfer functionality in the form of SOAP web services. While SOA prescribes exposing functionality over standardized interfaces, the focus is more on integrating heterogeneous applications than on implementing them.

Considerations

At one of the banks we worked at, we exposed a set of over 500 service interfaces over SOAP. Under the covers, we implemented these services using EJB 2.x (a combination of stateless session beans and message-driven beans) hosted on a commercial J2EE application server which also did double duty as an enterprise service bus (ESB). These services largely delegated most if not all the logic to a set of underlying stored procedures within a single monolithic Oracle database using a canonical data model for the entire enterprise! To the outside world, these services were *location transparent*, stateless, *composable* and *discoverable*. Indeed, we advertised this implementation as an example of SOA, and it would be hard to argue that it was not.

This suite of services had evolved organically over the years with no explicit boundaries, concepts

from various parts of the organization and generations of people mixed in, each adding their own interpretation of how business functionality needed to be implemented. In essence, the implementation resembled the dreaded big ball of mud which was extremely hard to enhance and maintain.

The intentions behind SOA are noble. However, the promises of reuse, loose coupling are hard to achieve in practice given the lack of concrete implementation guidance on component granularity. It is also true that SOA means many things[6] to different people. This ambiguity leads to most SOA implementations becoming complex, unmaintainable monoliths, centered around technology components like a service bus or the persistence store or both. This is where using DDD to solve a complex problem by breaking it down into subdomains and bounded contexts can be invaluable.

Microservices architecture

In the last decade or so, microservices have gained quite a lot of popularity with lots of organizations wanting to adopt this style of architecture. In a lot of ways, microservices are an extension of service-oriented architectures — one where a lot of emphasis is placed on creating focused components that deal with doing a limited number of things and doing them right. Sam Newman, the author of the *Building Microservices* book defines microservices as *small*-sized, independently deployable components that maintain their own state and are **modeled around a business domain**. This affords benefits such as adopting a horses for courses approach when modeling solutions, limiting the blast radius, improved productivity and speed, autonomous cross-functional teams, etc. Microservices usually exist as a collective, working collaboratively to achieve the desired business outcomes, as depicted here:

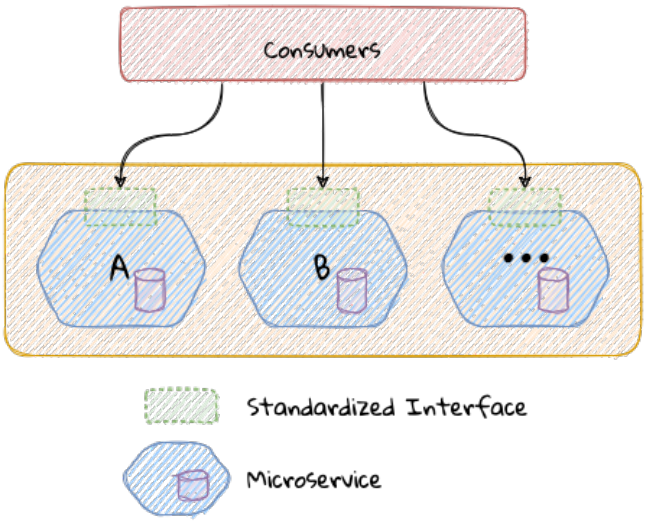

Figure 11. A microservices ecosystem

As we can see, SOA and microservices are very similar from the perspective of the consumers in that they access functionality through a set of standardized interfaces. The microservices approach is an evolution of SOA in that the focus now is on building smaller, self-sufficient, independently deployable components with the intent of avoiding single points of failure (like an enterprise database or service bus), which was fairly common with a number of SOA-based implementations.

Considerations

While microservices have definitely helped, there still exists quite a lot of ambiguity when it comes

to answering how big or small[7] a microservice should be. Indeed, a lot of teams seem to struggle to get this balance right, resulting in a distributed monolith[8] — which in a lot of ways can be much worse than even the single process monolith from the SOA days. Again, applying the strategic design concepts of DDD can help create independent, loosely coupled components, making it an ideal companion for the microservices style of architecture.

Event-Driven Architecture (EDA)

Irrespective of the granularity of components (monolith or microservices or something in between), most non-trivial solutions have a boundary, beyond which there may be a need to communicate with external system(s). This communication usually happens through the exchange of messages between systems, causing them to become coupled with each other. Coupling comes in two broad flavors: *afferent* — who depends on you and *efferent* — who you depend on. Excessive amounts of efferent coupling can make systems very brittle and hard to work with.

Event-driven systems enable authoring solutions that have a relatively low amount of efferent coupling by emitting events when they attain a certain state without caring about who consumes those events. In this regard, it is important to differentiate between message-driven and event-driven systems as mentioned in the *Reactive Manifesto*:

Message-driven versus Event-driven

> A message is an item of data that is sent to a specific destination. An event is a signal emitted by a component upon reaching a given state. In a message-driven system addressable recipients await the arrival of messages and react to them, otherwise lying dormant. In an event-driven system notification listeners are attached to the sources of events such that they are invoked when the event is emitted. This means that an event-driven system focuses on addressable event sources while a message-driven system concentrates on addressable recipients.
>
> — Reactive Manifesto

In simpler terms, event-driven systems do not care who the downstream consumers are, whereas in a message-driven system that may not necessarily be true. When we say event-driven in the context of this book, we mean the former.

Typically, event-driven systems eliminate the need for point-to-point messaging with the ultimate consumers by making use of an intermediary infrastructure component usually known as a message broker, event bus, etc. This effectively reduces the efferent coupling from *n* consumers to 1. There are a few variations on how event-driven systems can be implemented. In the context of publishing events, Martin Fowler talks about two broad styles (among other things) — event notifications and event-carried state transfer in his What do you mean by "event-driven"?[9] article.

Considerations

One of the main trade-offs when building an event-driven system is to decide the amount of state (payload) that should be embedded in each event. It may be prudent to consider embedding just enough state indicating changes that occurred as a result of the emitted event to keep the various opposing forces such as producer scaling, encapsulation, consumer complexity, resiliency, etc. We will discuss the related implications in more detail when we cover implementing events in Chapter 5.

Domain-driven design is all about keeping complexity in check by creating these independent bounded contexts. However, independent does not mean isolated. Bounded contexts may still need to communicate with each other. One way to do that is through the use of a fundamental DDD building block — domain events. Event-driven architecture and DDD are thus complementary. It is typical to make use of an event-driven architecture to allow bounded contexts to communicate while continuing to loosely coupled with each other.

Command Query Responsibility Segregation (CQRS)

In traditional applications, a single domain, data/persistence model is used to handle all kinds of operations. With CQRS, we create distinct models to handle updates (commands) and enquiries. This is depicted in the following diagram:

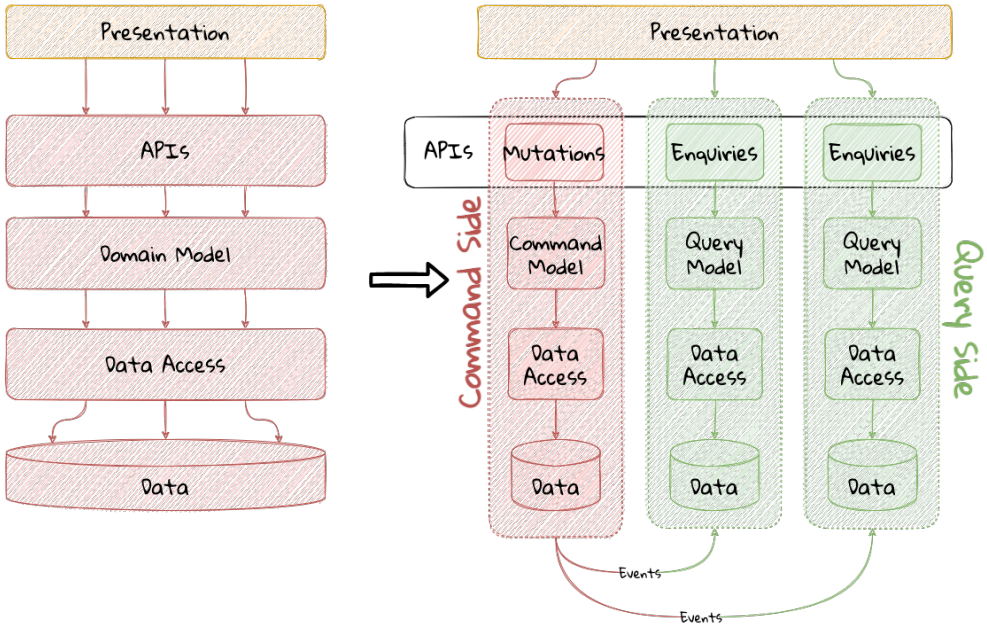

Figure 12. Traditional versus CQRS Architecture

We depict multiple query models above because it is possible (but not necessary) to create more than one query model, depending on the kinds of query use cases that need to be supported.

For this to work predictably, the query model(s) need to be kept in sync with the write models (we will examine some of the techniques to do that in detail later.

Considerations

The traditional, single-model approach works well for simple, CRUD-style applications, but starts to become unwieldy for more complex scenarios. We discuss some of these scenarios below:

Volume imbalance between read and writes

In most systems, read operations often outnumber write operations by significant orders of magnitude. For example, consider the number of times a trader checks stock prices vs. the number of times they actually transact (buy or sell stock trades). It is also usually true that write operations are the ones that make businesses money. Having a single model for both reads and writes in a system with a majority of read operations can overwhelm a system to an extent where write performance can start getting affected.

Need for multiple read representations

When working with relatively complex systems, it is not uncommon to require more than one representation of the same data.For example, when looking at personal health data, one may want to look at a daily, weekly, monthly view.While these views can be computed on the fly from the *raw* data, each transformation (aggregation, summarization, etc.) adds to the cognitive load on the system.Several times, it is not possible to predict ahead of time, the nature of these requirements.By extension, it is not feasible to design a single canonical model that can provide answers to all these requirements.Creating domain models specifically designed to meet a focused set of requirements can be much easier.

Different security requirements

Managing authorization and access requirements to data/APIs when working a single model can start to become cumbersome.For example, higher levels of security may be desirable for debit operations in comparison to balance enquiries.Having distinct models can considerably ease the complexity in designing fine-grained authorization controls.

More uniform distribution of complexity

Having a model dedicated to serve only command-side use cases means that they can now be focused towards solving a single concern.For query-side use cases, we create models as needed that are distinct from the command-side model.This helps spread complexity more uniformly over a larger surface area — as opposed to increasing the complexity on the single model that is used to serve all use cases.It is worth noting that the essence of domain-driven design is mainly to work effectively with complex software systems and CQRS fits well with this line of thinking.

When working with a CQRS based architecture, choosing the persistence mechanism for the command side is a key decision.When working in conjunction with an event-driven architecture, one could choose to persist aggregates as a series of events (ordered in the sequence of their occurrence).This style of persistence is known as event sourcing.We will cover this in more detail in Chapter 5 in the section on event-sourced aggregates.

Serverless Architecture

Serverless architecture is an approach to software design that allows developers to build and run services without having to manage the underlying infrastructure.The advent of AWS Lambda service has popularized this style of architecture, although several other services (like S3 and DynamoDB for persistence, SNS for notifications, SQS for message queuing etc.) have existed long before Lambda was launched.While AWS Lambda provided a compute solution in the form of Functions-as-a-Service (FaaS), these other services are just as essential, if not more, in order to benefit from the serverless paradigm.

In conventional DDD, bounded contexts are formed by grouping related operations around an aggregate, which then informs how the solution is deployed as a unit — usually within th confines of a single process.With the serverless paradigm, each operation (task) is required to be deployed as an independent unit of its own as distributed components.This requires that we look at how we model aggregates and bounded contexts differently — now centered around individual tasks as opposed to a group of related tasks.

Does that mean that the principles of DDD to arrive at a solution do not apply anymore?While serverless introduces an additional dimension of having to treat finely-grained deployable units as first-class citizens in the modeling process, the overall process of applying DDD's strategic and tactical design continue to apply.We will examine this in more detail in Chapter 12 when we refactor the solution we build throughout this book to employ a serverless approach.

Big ball of mud

Thus far, we have examined a catalog of named architecture styles along with their pitfalls and how applying DDD can help alleviate them.On the other extreme, we may encounter solutions that lack a perceivable architecture, infamously termed as the *big ball of mud*.

> A BIG BALL OF MUD is haphazardly structured, sprawling, sloppy, duct-tape and bailing wire, spaghetti code jungle. We've all seen them. These systems show unmistakable signs of unregulated growth, and repeated, expedient repair. Information is shared promiscuously among distant elements of the system, often to the point where nearly all the important information becomes global or duplicated. The overall structure of the system may never have been well-defined. If it was, it may have eroded beyond recognition. Programmers with a shred of architectural sensibility shun these quagmires. Only those who are unconcerned about architecture, and, perhaps, are comfortable with the inertia of the day-to-day chore of patching the holes in these failing dikes, are content to work on such systems.
>
> — Brian Foote and Joseph Yoder

Although Foote and Yoder advise avoiding this style of architecture at all costs, software systems that resemble the big ball of mud continue to be a day-to-day inevitability for a lot of us. The

strategic and tactical design elements of DDD provide a set of techniques to help deal with and recover from these near-hopeless situations in a pragmatic manner without potentially having to adopt a big bang approach. Indeed, the focus of this book is to apply these principles to prevent or at least delay further devolution towards the big ball of mud.

Which architecture style should you use?

As we have seen, there are a variety of architecture styles one can lean on to when crafting a software solution. A lot of these architecture styles share quite a few common tenets. It can become difficult to claim conformance to any single architecture style. DDD, with its emphasis on breaking down complex business problems into subdomains and bounded contexts, enables the use of more than one approach across bounded contexts. We would like to make a special mention of the vertical slice architecture because it places an emphasis on dividing functionality along specific business outcomes and thus more naturally to DDD's ideas of subdomains and bounded contexts. In reality, one may find the need to extend and even deviate from pedantic definitions of architecture styles in order to meet real-world needs. But when we do make such compromises, it is important to do so **intentionally** and make it unambiguously clear why we are making such a decision (preferably using some lightweight mechanism such as ADRs[10]). This is important because it may become hard to justify this to others and even ourselves when we look at it in the future.

In this section, we have examined popular architecture styles and how we can amplify their effectiveness when used in conjunction with DDD. Now let's look at how DDD can complement the use of existing programming paradigms.

Programming paradigms

The tactical elements of DDD introduce a specific vocabulary (aggregates, entities, value objects, repositories, services, factories, domain events, etc.) when arriving at a solution. At the end of the day, we need to translate these concepts into running software. Over the years, we have employed a variety of programming paradigms including procedural, object-oriented, functional, aspect-oriented, etc. Is it possible to apply DDD in conjunction with one or more of these paradigms? In this section, we will explore how some common programming paradigms and techniques help us express the tactical design elements in code.

Object-oriented programming

On the surface of it, DDD seems to simply replicate a set of OO terms and call them using different names. For example, the central concepts of tactical DDD such as `aggregates`, `entities` and `value objects` could simply be referred to as objects in OO terms. Others like `services` may not have a direct OO analog. So how does one apply DDD in an object-oriented world? Let's look at a simple example:

```
interface PasswordService {
    String generateStrongPassword();
    boolean isStrong(String password);
    boolean isWeak(String password);
}

class PasswordClient {
    private PasswordService service;

    void register(String userEnteredPassword) {
        if (service.isStrong(userEnteredPassword)) {
            //...
        }
    }
}
```

OO purists will be quick to point out that the `PasswordService` is procedural and that a `Password` class might be needed to encapsulate related behaviours. Similarly, DDD enthusiasts might point out that this is an anemic domain model implementation. An arguably better object-oriented version might look something like:

```
class Password {
    private final String password;

    private Password(String password) {
        this.password = password;
    }

    public boolean isStrong() { ... }
    public boolean isWeak() { ... }
    public static Password generateStrongPassword() { ... }
    public static Password passwordFrom(String password) { ... }

}

interface PasswordService {
    Password generateStrongPassword();
    Password createPasswordFrom(String userEntered);
}

class PasswordClient {
    private PasswordService service;

    void register(String userEnteredPassword) {
        Password password = service.createPasswordFrom(userEnteredPassword);
        if (password.isStrong()) {
            // ...
        }
    }
}
```

In this case, the `Password` class stops exposing its internals and exposes the idea of a strong or weak password in the form of behavior (the `isStrong` and `isWeak` methods). From an OO perspective, the second implementation is arguably superior. If so, shouldn't we be using the object-oriented version at all times? As it turns out, the answer is nuanced and depends on what the consumers desire and the ubiquitous language used in that context. If the concept of the `Password` is in common usage within the domain, it perhaps warrants introducing such a concept in the implementation as well. If not, the first solution might suffice even though it seems to violate OO principles of encapsulation.

Our default position is to apply good OO practices as a starting point. However, it is more important to mirror the language of the domain as opposed to applying OO in a dogmatic manner. So we will be willing to compromise on OO purity if it appears unnatural to do so in that context. As mentioned earlier, clearly communicating the rationale for such decisions can go a long way.

Functional programming

Functions are a fundamental building block to code organization that exist in all higher order programming languages. Functional programming is a programming paradigm where programs are constructed by applying and composing functions. This is in contrast to imperative

programming that uses statements to change a program's state. The most significant differences stem from the fact that functional programming avoids side effects, which are used in imperative programming. Pure functional programming completely prevents side effects and forces immutability. Embracing a functional style when designing a domain model to be more declarative and express intent a lot more clearly while remaining terse. It also allows us to keep complexity in check by enabling us to compose more complex concepts by using simpler ones.The functional implementation allows us to use a language closer to the problem domain, while having the added benefit of also being terse. Consider a simple example where we need to find the item with the least inventory across all our warehouses using a functional style as shown here:

Functional example

```
class Functional {
    public static Optional<Item> scarcestItem(Warehouse... warehouses) {
        return Stream.of(warehouses)
                .flatMap(Warehouse::items)
                .collect(groupingBy(Item::name, summingInt(Item::quantity)))
                .entrySet().stream()
                .map(Item::new)
                .min(comparing(Item::quantity));
    }
}
```

The imperative style shown here does get the job done, but is arguably a lot more verbose and harder to follow, sometimes even for technical team members!

Imperative example

```
class Imperative {
    public static Optional<Item> scarcestItem(Warehouse... warehouses) {
        Collection<Item> allItems = new ArrayList<>();
        for (Warehouse warehouse : warehouses) {
            allItems.addAll(warehouse.getItems());
        }
        Map<String, Integer> itemNamesByQuantity = new HashMap<>();
        for (Item item : allItems) {
            final String name = item.name();
            final int quantity = item.quantity();
            if (itemNamesByQuantity.containsKey(name)) {
                itemNamesByQuantity.put(name, itemNamesByQuantity.get(name) +
quantity);
            } else {
                itemNamesByQuantity.put(name, quantity);
            }
        }
        final Map.Entry<String, Integer> min =
            Collections.min(itemNamesByQuantity.entrySet(), Map.Entry.
comparingByValue());
        return min != null ? Optional.of(new Item(min)) : Optional.empty();

    }
}
```

From a DDD perspective, this yields a few benefits:

- **Increase collaboration with domain experts** because the declarative style allows placing a bigger focus on the what, rather than the how. This makes it a lot less intimidating to technical and non-technical stakeholders alike to work with on an ongoing basis.
- **Better testability**: because the use of pure functions (those that are side effect free) makes it easier to create data-driven tests. This has also afforded us an additional benefit of less mocking/stubbing. These characteristics make tests that are a lot easier to maintain and reason about. This has the benefit of allowing even technical team members to visualize corner cases a lot earlier in the process.

Which paradigm should you choose?

DDD simply states that you should build your software around a domain model that represents the actual problem that the software is trying to solve. When encountered with complex real-life problems, often we will find it hard to conform to any single paradigm across the board. Looking to use a one-size-fits-all approach may work to one's detriment. Our experience indicates that we will need to make use of a variety of techniques in order to solve the problem at hand elegantly. Java is inherently an object-oriented language. But with the advent of Java 8, it has started to embrace a variety of functional constructs as well. This allows us to make use of a multitude of techniques to create elegant solutions. The most important thing is to agree on the ubiquitous language and allow it to guide the approach taken. It also largely depends on the talent and experience one has at their

disposal. Making use of a style that is foreign to a majority of the team will likely prove counter-productive. Although we haven't covered the procedural paradigm here in this text, there may be occasions where it might be the best solution given the current situation. As long as we are intentional about areas where we deviate from the accepted norm for a particular programming paradigm, we should be in a reasonably good place.

Summary

In this chapter, we covered a series of commonly used architecture patterns and how we can practice DDD when working with them. We also looked at common pitfalls and gotchas that one may need to be cognizant of when using these architectures. We also looked at popular programming paradigms and their influence on the tactical elements of DDD.

Additionally, you should have an appreciation of the various architecture styles that we need to employ when coming up with a solution. In addition, you should have an understanding of how DDD can play a role no matter which style of architecture you choose to adopt.

In the next section, we will look to apply all the learnings in this and previous chapters against a real-world business use case. We will apply both the strategic and tactical patterns of DDD to break a complex domain into subdomains, bounded contexts and iteratively build a solution using technologies that are based on the Java programming language.

Questions

1. What architecture style(s) are you using in your current ecosystem? Are you seeing any of the common pitfalls that we have covered in this chapter?
2. Do you see merit in employing a hybrid of the architecture approaches covered here?
3. What programming paradigms are you employing in your current ecosystem?
4. Are there areas where you have had to violate principles that are strictly aligned with any given paradigm or approach? Are these deviations well understood and documented?

[1] https://alistair.cockburn.us/hexagonal-architecture/

[2] https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/

[3] https://blog.cleancoder.com/uncle-bob/2020/10/18/Solid-Relevance.html

[4] https://www.archunit.org/

[5] https://jimmybogard.com/vertical-slice-architecture/

[6] https://martinfowler.com/bliki/ServiceOrientedAmbiguity.html

[7] https://martinfowler.com/articles/microservices.html#HowBigIsAMicroservice

[8] https://www.infoq.com/news/2016/02/services-distributed-monolith/

[9] https://martinfowler.com/articles/201701-event-driven.html

[10] https://www.thoughtworks.com/de-de/radar/techniques/lightweight-architecture-decision-records