CHAPTER 3

A Functional Architecture

Here's our next challenge: how should we translate our understanding of the domain into a software architecture, especially one that is based on functional programming principles?

We really shouldn't be doing too much thinking about architecture at this point, because we still don't really understand the system yet—we are at the peak of our ignorance! The best use of our time is to do things that reduce this ignorance: Event Storming, interviews, and all the other best practices around requirements gathering.

On the other hand, it's good to have a rough idea of how we are going to implement our domain model as software. In a fast-paced development cycle, we often need to start implementing some of the domain before we have understood the rest of it, so we'll need to have some plan for fitting the various components together even before they're built. And there's a lot to be said for creating a crude prototype—a "walking skeleton"—that demonstrates how the system will work as a whole. Early feedback on a concrete implementation is a great way to discover gaps in your knowledge.

In this chapter we'll take a brief look at a typical software architecture for a functionally oriented domain model. We'll look at how DDD concepts such as *bounded contexts* and *Domain Events* might be translated into software, and we'll sketch out the approach to implementation that we'll use in the rest of this book.

Software architecture is a domain in its own right, of course, so let's follow our own advice and use a "ubiquitous language" when talking about it. We'll use the terminology from Simon Brown's "C4" approach,[1] whereby a software architecture consists of four levels that can be described as follows:

1. http://static.codingthearchitecture.com/c4.pdf

- The "system context" is the top level, representing the entire system.
- The system context comprises a number of "containers," which are deployable units such as a website, a web service, a database, and so on.
- Each container in turn comprises a number of "components," which are the major structural building blocks in the code.
- Finally, each component comprises a number of "classes" (or in a functional architecture, "modules") that contain a set of low-level methods or functions.

One of the goals of a good architecture is to define the various boundaries between containers, components, and modules, such that when new requirements arise, as they will, the "cost of change" is minimized.

Bounded Contexts as Autonomous Software Components

Let's start with the concept of a "bounded context" and how it relates to an architecture. As we saw earlier, it's important that a context is an *autonomous* subsystem with a *well-defined boundary*. Even with those constraints, though, we have a number of common architectural styles to choose from.

If the entire system is implemented as a single monolithic deployable (a single container using the C4 terminology above), a bounded context could be as simple as a separate module with a well-defined interface, or preferably, a more distinct component such as a .NET assembly. Alternatively, each bounded context could be deployed separately in its own container—a classic service-oriented architecture. Or we could go even more fine-grained and make each individual *workflow* into a standalone deployable container—a microservice architecture.

At this early stage, however, we do not need to commit to a particular approach. The translation from the *logical* design to the *deployable* equivalent is not critical, as long as we ensure that the bounded contexts stay decoupled and autonomous.

We stressed earlier that it's important to get the boundaries right, but of course, this is hard to do at the beginning of a project, and we should expect that the boundaries will change as we learn more about the domain. It's a lot easier to refactor a monolith, so a good practice is to build the system as a monolith initially and refactor to decoupled containers only as needed. There's no need to jump straight to microservices and pay the "microservice premium"[2]

2. https://www.martinfowler.com/bliki/MicroservicePremium.html

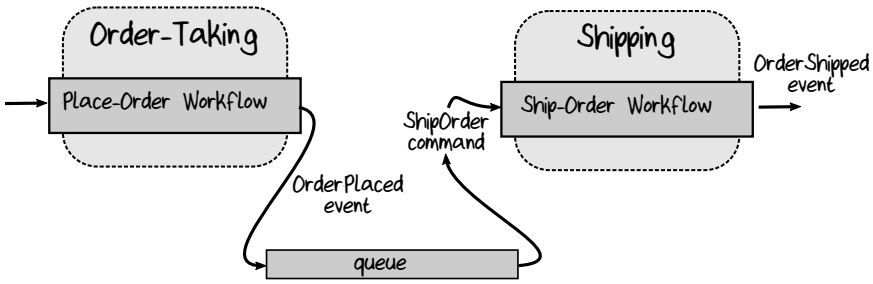
(the extra burden on operations) unless you are sure the benefits outweigh the drawbacks. It's tricky to create a truly decoupled microservice architecture: if you switch one of the microservices off and anything else breaks, you don't really have a microservice architecture, you just have a distributed monolith!

Communicating Between Bounded Contexts

How do bounded contexts communicate with each other? For example, when the order-taking context has finished processing the order, how does it tell the shipping context to actually ship it? As we've seen earlier, the answer is to use events. For example, the implementation might look like this:

- The Place-Order workflow in the order-taking context emits an OrderPlaced event.
- The OrderPlaced event is put on a queue or otherwise published.
- The shipping context listens for OrderPlaced events.
- When an event is received, a ShipOrder command is created.
- The ShipOrder command initiates the Ship-Order workflow.
- When the Ship-Order workflow finishes successfully, it emits an OrderShipped event.

Here's a diagram for this example:



You can see that this is a completely decoupled design: the upstream component (the order-taking subsystem) and the downstream component (the shipping subsystem) are not aware of each other and are communicating only through events. This kind of decoupling is critical if we want to have truly autonomous components.

The exact mechanism for transmitting events between contexts depends on the architecture we choose. Queues are great for buffered asynchronous communication and so would be the first choice for an implementation with microservices or agents. In a monolithic system, we can use the same queuing approach internally, or just use a simple direct linkage between the upstream component and the downstream component via a function call. As always, we don't need to choose right now, as long as we design the components to be decoupled.

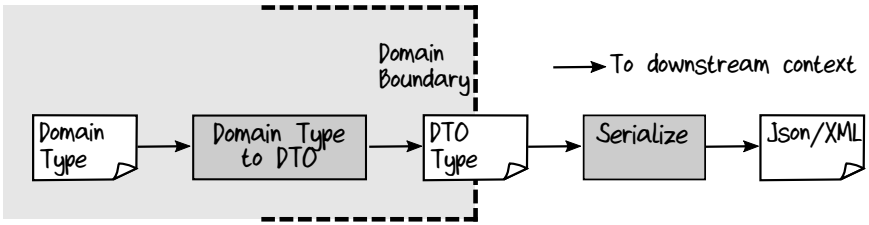
As for the handler that translates events (such as OrderPlaced) to commands (such as ShipOrder), it can be part of the downstream context (living at the boundary of the context), or it can be done by a separate router[3] or process manager[4] running as part of the infrastructure, depending on your architecture and where you want to do the coupling between events and commands.

Transferring Data Between Bounded Contexts

In general, an event used for communication between contexts will not be just a simple signal but will also contain all the data that the downstream components need to process the event. For example, the OrderPlaced event might contain the complete order that was placed. That gives the shipping context all the information it needs to construct a corresponding ShipOrder command. (If the data is too large to be contained in the event, some sort of reference to a shared data storage location can be transmitted instead.)

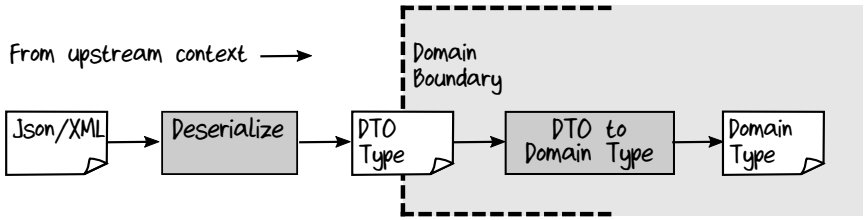
The data objects that are passed around may be superficially similar to the objects defined inside the bounded context (which we'll call *domain objects*), but they are *not* the same; they are specifically designed to be serialized and shared as part of the intercontext infrastructure. We will call these objects *Data Transfer Objects* or DTOs (although that term originated outside of DDD,[5] and I am using it slightly differently here). In other words, the OrderDTO contained in an OrderPlaced event will contain most of the same information as an Order domain object, but it will be structured differently to suit its purpose. (The *Serialization* chapter goes into detail on how to define DTOs.)

At the boundaries of the upstream context then, the domain objects are converted into DTOs, which are in turn serialized into JSON, XML, or some other serialization format:



At the downstream context, the process is repeated in the other direction: the JSON or XML is deserialized into a DTO, which in turn is converted into a domain object as shown in the figure on page 47.

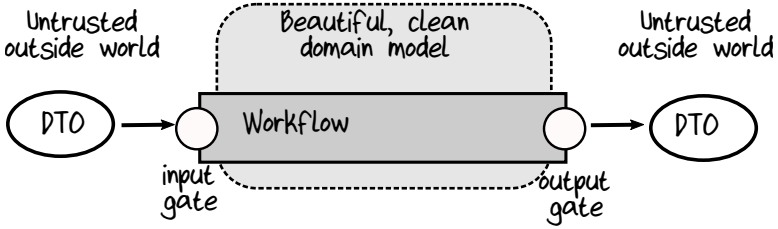
3. http://www.enterpriseintegrationpatterns.com/patterns/messaging/MessageRouter.html
4. https://www.slideshare.net/BerndRuecker/long-running-processes-in-ddd
5. https://martinfowler.com/eaaCatalog/dataTransferObject.html



In practice, the top-level DTOs that are serialized are typically event DTOs, which in turn contain child DTOs, such as a DTO for Order, which in turn contains additional child DTOs (such as a list of DTOs representing OrderLines).

Trust Boundaries and Validation

The perimeter of a bounded context acts as a "trust boundary." Anything inside the bounded context will be trusted and valid, while anything outside the bounded context will be untrusted and might be invalid. Therefore, we will add "gates" at the beginning and end of the workflow that act as intermediaries between the trusted domain and the untrusted outside world.



At the input gate, we will *always* validate the input to make sure that it conforms to the constraints of the domain model. For example, say that a certain property of an Order must be non-null and less than fifty characters. The incoming OrderDTO will have no such constraints and could contain anything, but after validation at the input gate, we can be sure that the Order domain object is valid. If the validation fails, then the rest of the workflow is bypassed and an error is generated. (The *Serialization* chapter covers this kind of DTO validation.)

The job of the output gate is different. Its job is to ensure that private information doesn't leak out of the bounded context, both to avoid accidental coupling between contexts and for security reasons. For example, there's no need for the shipping context to know the credit card number used to pay for an order. In order to do this, the output gate will often deliberately "lose" information (such as the card number) in the process of converting domain objects to DTOs.

Contracts Between Bounded Contexts

We want to reduce coupling between bounded contexts as much as possible, but a shared communication format always induces some coupling: the events and related DTOs form a kind of contract between bounded contexts. The two contexts will need to agree on a common format for them in order for communication to be successful.

So who gets to decide the contract? There are various relationships between the contexts, and the DDD community has developed some terms for the common ones:

- A *Shared Kernel* relationship is where two contexts share some common domain design, so the teams involved must collaborate. In our domain, for example, we might say that the order-taking and shipping contexts must use the same design for a delivery address: the order-taking context accepts an address and validates it, while the shipping context uses the same address to ship the package. In this relationship, changing the definition of an event or a DTO must be done only in consultation with the owners of the other contexts that are affected.
- A *Customer/Supplier* or *Consumer Driven Contract*[6] relationship is where the downstream context defines the contract that they want the upstream context to provide. The two domains can still evolve independently, as long as the upstream context fulfills its obligations under the contract. In our domain, the billing context might define the contract ("this is what I need in order to bill a customer") and then the order-taking context provides only that information and no more.
- A *Conformist* relationship is the opposite of consumer-driven. The downstream context accepts the contract provided by the upstream context and adapts its own domain model to match. In our domain, the order-taking context might just accept the contract defined by the product catalog and adapt its code to use it as is.

Anti-Corruption Layers

Often when communicating with an external system, the interface that is available does not match our domain model at all. In this case, the interactions and data need to be transformed into something more suitable for use inside

6. https://www.infoq.com/articles/consumer-driven-contracts

the bounded context, otherwise our domain model will become "corrupted" by trying to adapt to the external system's model.

This extra level of decoupling between contexts is called an *Anti-Corruption Layer* in DDD terminology, often abbreviated as "ACL." In the diagram above, the "input gate" often plays the role of the ACL—it prevents the internal, pure domain model from being "corrupted" by knowledge of the outside world.

That is, the Anti-Corruption Layer is not primarily about performing validation or preventing data corruption, but instead acts as a translator between two different languages—the language used in the upstream context and the language used in the downstream context. In our order-taking example, then, we might have an Anti-Corruption Layer that translates from "order-taking" vocabulary to "shipping" vocabulary, allowing the two contexts, each with their own vocabulary, to evolve independently.

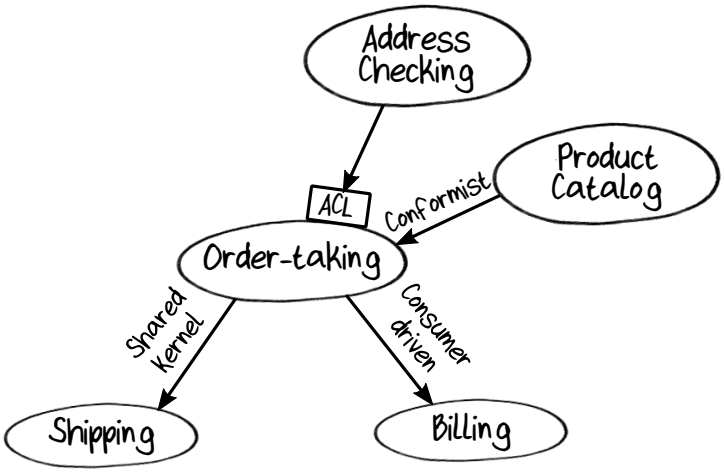
A Context Map with Relationships

Let's say that we have progressed with our design and have now decided what the relationships between our contexts are:

- The relationship between the order-taking and shipping contexts will be a "Shared Kernel," meaning that they will jointly own the communications contract.
- The relationship between order-taking and billing will be a "Consumer-Driven Contract" one, meaning that the billing context determines the contract and the order-taking system will supply the billing context with exactly the data it needs.
- The relationship between order-taking and the product catalog will be a "Conformist" one, meaning that the order-taking context will submit to using the same model as the product catalog.
- Finally, the external address checking service has a model that's not at all similar to our domain, so we'll insert an explicit Anti-Corruption Layer into our interactions with it. This is a common pattern when using a third-party component. It helps us avoid vendor lock-in and lets us swap to a different service later.

A context map of our domain showing these kinds of intercontext relationships is shown in the figure on page 50.

You can see that the context map is no longer just showing purely technical relationships between contexts, but is now also showing the relationships between the *teams* that own the contexts and how we expect them to



collaborate (or not!). Deciding on how the domains interact is often just as much an organizational challenge as it is a technical one, and some teams have used the so-called "inverse Conway maneuver"[7] to ensure that the organization structure is aligned with the architecture.

Workflows Within a Bounded Context

In our discovery process, we treated business workflows as a mini-process initiated by a command, which generated one or more Domain Events. In our functional architecture, each of these workflows will be mapped to a single function, where the input is a command object and the output is a list of event objects.

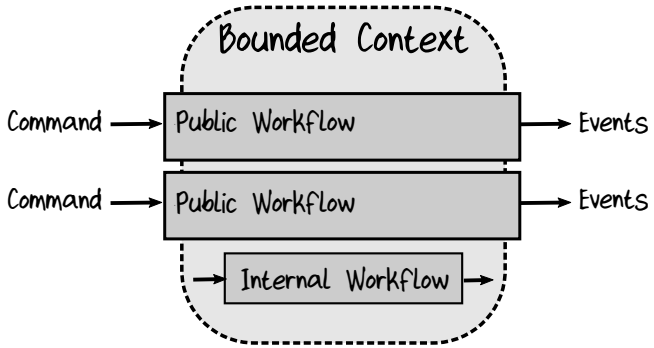
When we create diagrams of the design, we represent workflows as little pipes with an input and output. Public workflows (those that are triggered from outside the bounded context) are shown as "sticking out" a little over the boundary as shown in the figure on page 51.

A workflow is always contained within a single bounded context and never implements a scenario "end-to-end" through multiple contexts. The *Modeling Workflows as Pipelines* chapter goes into detail on how to model workflows.

Workflow Inputs and Outputs

The input to a workflow is always the data associated with a command, and the output is always a set of events to communicate to other contexts. In our order-placing workflow, for example, the input is the data associated with a PlaceOrder command and the output is a set of events such as the OrderPlaced event.

7. http://bit.ly/InverseConwayManeuver

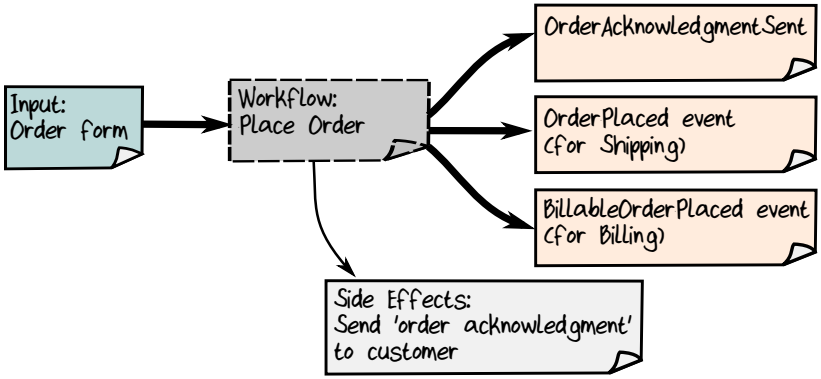


But remember that we have now determined that there is a "customer/supplier" relationship with the billing context. That means that, rather than sending a generic OrderPlaced event to the billing context, we need to send *only* the information that billing needs and no more. For example, this might just be the billing address and the total amount to bill but not the shipping address or the list of items.

This means we will need to emit a new event (BillableOrderPlaced say) from our workflow, with a structure that might look something like this:

```
data BillableOrderPlaced =
    OrderId
    AND BillingAddress
    AND AmountToBill
```

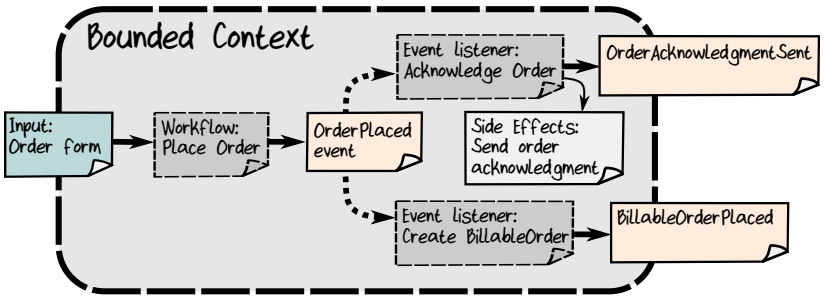
We might also want to emit an OrderAcknowledgmentSent event as well. With these changes, our earlier diagram of the workflow on page 29 is misleading and we need to update it:



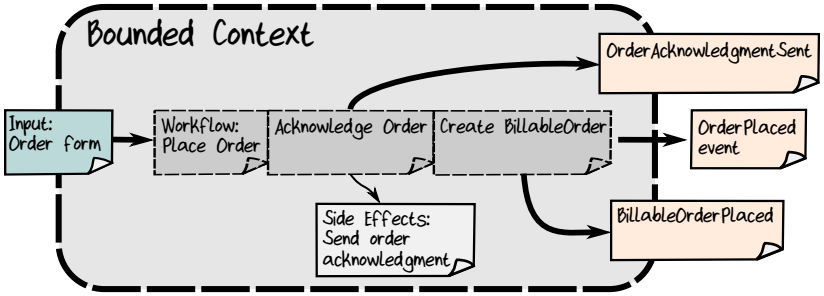
In the preceding diagram, it's important to note that a workflow function does not "publish" Domain Events—it simply returns them. How they get published is a separate concern.

Avoid Domain Events Within a Bounded Context

In an object-oriented design, it is common to have Domain Events raised internally within a bounded context. In that approach, a workflow object raises an OrderPlaced event. Next a handler listens for that event and sends the order acknowledgment, then another handler generates a BillableOrderPlaced event, and so on. It might look like this:



In a functional design, we prefer not to use this approach because it creates hidden dependencies. Instead, if we need a "listener" for an event, we just append it to the end of workflow like this:

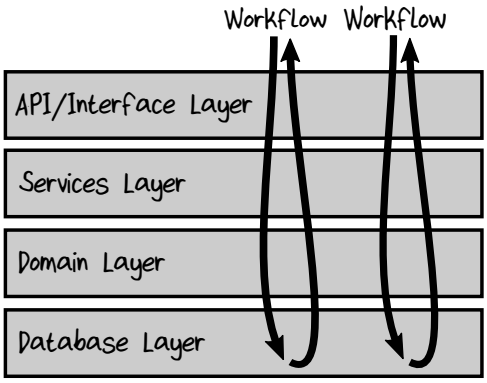


This approach is more explicit—there are no global event managers with mutable state—and therefore it's easier to understand and maintain. We'll see how this works in practice in the Implementation chapter on page 161 and also in the Evolving A Design chapter on page 265.

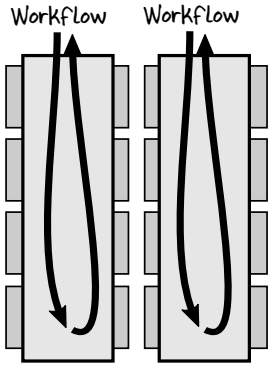
Code Structure Within a Bounded Context

Now let's look at how the code is structured within a bounded context.

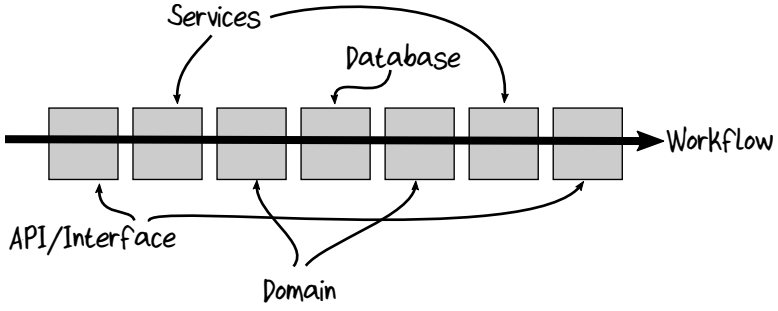
In a traditional "layered approach," the code is divided into layers: a core domain or business logic layer, a database layer, a services layer, and an API or user interface layer (or some variant of these). A workflow will start at the top layer, work its way down to the database layer, and then return back to the top as shown in the figure on page 53.



This approach has many problems, however. One particular issue is that it breaks the important design principle of "code that changes together belongs together." Because the layers are assembled "horizontally," a change to the way that the workflow works means that you need to touch every layer. A better way is to switch to "vertical" slices, where each workflow contains all the code it needs to get its job done, and when the requirements change for a workflow, only the code in that particular vertical slice needs to change as shown in the figure.



This is still not ideal, though. To see this, let's stretch a workflow into a horizontal pipe and look at the layers in that way.

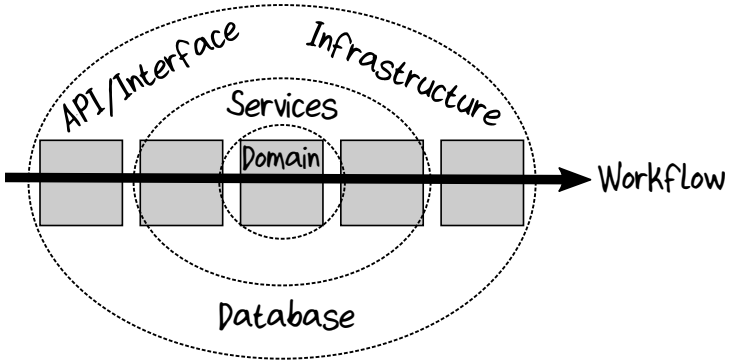


It's clear that the layers are intermingled in a way that makes understanding the logic (and testing it) unnecessarily complicated.

The Onion Architecture

Let's instead put the domain code at the center and then have the other aspects be assembled around it using the rule that each layer can only depend

on inner layers, not on layers further out. That is, *all dependencies must point inward*. This is called the "Onion Architecture."[8]



Other similar approaches exist, such as the Hexagonal Architecture[9] and the Clean Architecture.[10]

In order to ensure that all dependencies point inward, we will have to use the functional equivalent of dependency injection, which is discussed in *Implementation: Composing a Pipeline*.

Keep I/O at the Edges

A major aim of functional programming is to work with functions that are predictable and easy to reason about without having to look inside them. In order to do this, we will try to work with immutable data wherever possible and try to ensure that our functions have *explicit* dependencies instead of hidden dependencies. Most importantly, we will try to avoid side effects in our functions, including randomness, mutation of variables outside the function, and most importantly, any kind of I/O.

For example, a function that reads or writes to a database or file system would be considered "impure," so we would try to avoid these kinds of functions in our core domain.

But then how *do* we read or write data? The answer is to push any I/O to the edges of the onion—to access a database, say, only at the start or end of a workflow, not inside the workflow. This has the additional benefit of forcing us to separate different concerns: the core domain model is concerned only with business logic, while persistence and other I/O is an infrastructural concern.

8. http://jeffreypalermo.com/blog/the-onion-architecture-part-1/
9. http://alistair.cockburn.us/Hexagonal+architecture
10. https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html

In fact, the practice of shifting I/O and database access to the edges combines very nicely with the concept of *persistence ignorance* that we introduced in the previous chapter. You can't model your domain using a database if you can't even access the database from inside the workflow! (The *Persistence* chapter discusses the use of databases in more detail.)

Wrapping Up

We've been introduced to a few more DDD-related concepts and terminology in this chapter, so let's summarize them in one place:

- A *Domain Object* is an object designed for use only within the boundaries of a context, as opposed to a Data Transfer Object.
- A *Data Transfer Object*, or *DTO*, is an object designed to be serialized and shared between contexts.
- *Shared Kernel*, *Customer/Supplier*, and *Conformist* are different kinds of relationships between bounded contexts.
- An *Anti-Corruption Layer*, or *ACL*, is a component that translates concepts from one domain to another in order to reduce coupling and allow domains to evolve independently.
- *Persistence Ignorance* means that the domain model should be based only on the concepts in the domain itself and should not contain any awareness of databases or other persistence mechanisms.

What's Next

We've now got an understanding of the domain and a general approach to designing a solution for it, so we can move on to the challenge of modeling and implementing the individual workflows. In the next few chapters, we'll be using the F# type system to define a workflow and the data that it uses, creating compilable code that is still understandable by domain experts and non-developers.

To start with, though, we need to understand what *type* means to functional programmers and how it is different from *class* in object-oriented design. That's the topic of the next chapter.

Part II

Modeling the Domain

In this second part, we'll take one workflow from the domain and model it in a functional way. We'll see how the functional decomposition of a domain differs from an object-oriented approach, and we'll learn how to use types to capture requirements. By the end of this part, you'll know how to write concise code that does double-duty: first as readable documentation of the domain but also as a compilable framework that the rest of the implementation can build upon.

CHAPTER 4

Understanding Types

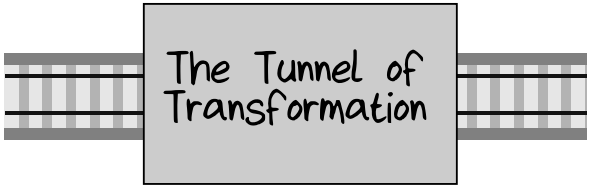
In the second chapter, we captured the domain-driven requirements for a single workflow of the order-taking system. The next challenge is to convert those informal requirements into compilable code.

The approach we are going to take is to represent the requirements using F#'s "algebraic type system." In this chapter we'll learn what algebraic types are, how they are defined and used, and how they can represent a domain model. Then, in the next chapter, we'll use what we've learned to accurately model the order-placing workflow.

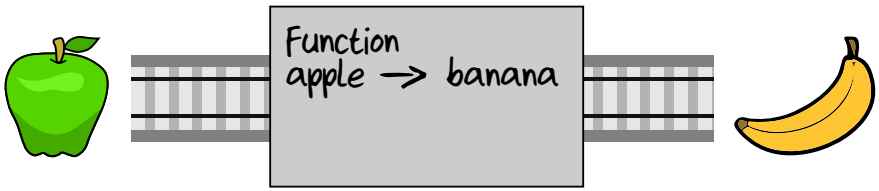
Understanding Functions

Before we can understand types, we need to understand the most basic concept in functional programming—a function.

If you remember your high-school mathematics, a function is a kind of black box with an input and an output. You can imagine it as a bit of railroad track, with a Tunnel of Transformation sitting on it. Something goes in, is transformed somehow, and comes out the other side.



For example, let's say that this particular function turns apples into bananas. We describe a function by writing down the input and output, separated by an arrow, as shown in the figure on page 60.



Type Signatures

The `apple -> banana` description is called a *type signature* (also known as a function signature). This particular signature is simple, but type signatures can get very complicated. Understanding and using type signatures is a critical part of coding with F#, so let's make sure we understand how they work.

Here are two functions: `add1` adds 1 to its single input `x`, and `add` adds its two inputs, `x` and `y`:

```
let add1 x = x + 1   // signature is: int -> int
let add x y = x + y  // signature is: int -> int -> int
```

As you can see, the `let` keyword is used to define a function. The parameters are separated by spaces, without parentheses or commas. Unlike C# or Java, there is no `return` keyword. The last expression in the function definition is the output of the function.

Even though F# cares about the types of the inputs and outputs, you rarely need to explicitly declare what they are, because in most cases the compiler will infer the types for you automatically.[1]

- For `add1`, the inferred type of `x` (before the arrow) is `int` and the inferred type of the output (after the arrow) is also `int`, so the type signature is `int -> int`.
- For `add`, the inferred type of `x` and `y` is `int` and the inferred type of the output (after the last arrow) is also `int`. `add` has two parameters, and each parameter is separated by an arrow, so the type signature is `int -> int -> int`.

If you are using an IDE such as Visual Studio, hovering over the definition of a function will show you its type signature, but since this is a book, we'll put the type signature in a comment above the definition when we need to make it clear. It's just a comment and isn't used by the compiler.

Functions that consist of more than one line are written with an indent (like Python). There are no curly braces. Here's an example:

1. https://fsharpforfunandprofit.com/posts/type-inference/