Extracted from:

From Objects to Functions

Build Your Software Faster and Safer with Functional Programming and Kotlin

The Pragmatic Bookshelf

Dallas, Texas

The Pragmatic Programmers

From Objects to Functions

Build Your Software Faster and Safer with Functional Programming and Kotlin

Uberto Barbini

edited by Adaobi Obi Tulton

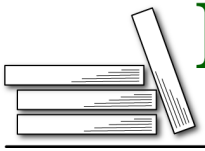
From Objects to Functions

Build Your Software Faster and Safer with Functional Programming and Kotlin

Uberto Barbini

The Pragmatic Bookshelf

Dallas, Texas



For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Adaobi Obi Tulton
Copy Editor: Karen Galle
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-845-1
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—September 2023

Separating the Domain from the Infrastructure

One of the general principles of good software design is that each module of our system should have only one reason to change. For this reason, we want to keep separate the logic that forms the business domain model from the technical implementation. We also want to keep our domain completely pure and without side effects, so we can easily test it and safely re-use its functions everywhere we need to.

An interesting question is how to distinguish what is business logic from what isn't? The main difference is at language level: business logic is something you can discuss with the business people, without using any technical terms such as serialization formats, network protocols, and so on. This can be hard if your business model is something technical, for example, if the product you are developing is a cloud platform.

On an even more general level, you can look at what you want to decouple in the design of your application. For example, do you really want to mix HTTP routing with JSON parsing and reporting logic? Probably not. Keeping all these aspects separate is the foundation of a clean design.

The Hub

So how can we keep the domain pure and separated from the rest? We'll use an interface to wrap all the domain logic and keep it separated from the external adapter. This approach is called Port and Adapter. It has been proposed by Alistair Cockburn,[2] and even if it's based on the object-oriented paradigm, it also works very well with functional programming.

The Port and Adapter Pattern

Also known as Hexagonal Architecture, it's a software architecture pattern that aims to create software systems that are flexible, maintainable, and adaptable to change. The primary goal of this pattern is to create a separation between the core business logic of a system and the external dependencies it relies on, such as databases, web services, and user interfaces.

The idea is to introduce a layer of abstraction between the core logic and its external dependencies, known as ports and adapters. Ports define the interfaces through which the core logic communicates with the external dependencies, while adapters implement

2. https://alistaircockburn.com/Component%20plus%20strategy.pdf

those interfaces and provide the actual communication with the external dependencies.

Overall, the Port and Adapter pattern is a powerful tool for creating maintainable and adaptable software systems, particularly in complex domains where external dependencies are numerous and subject to change.

You may wonder, why do we need yet another abstraction? Isn't it simpler if we just connect our domain functions to the HTTP layer?

Edsger W. Dijkstra said, "Being abstract is something profoundly different from being vague…The purpose of abstraction is not to be vague but to create a new semantic level in which one can be absolutely precise."

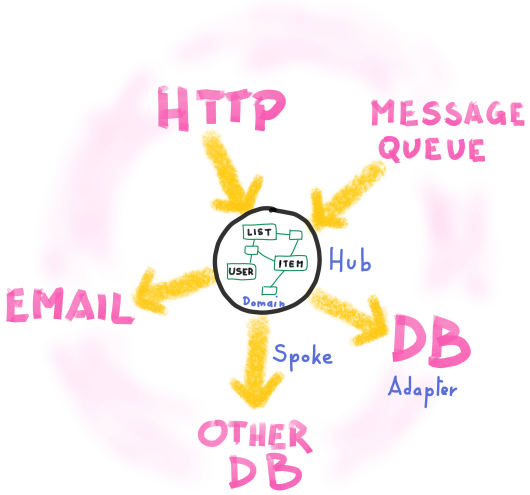
It may be simple to connect domain functions to the HTTP layer directly, but after a while, it will become very hard to separate one from the other. Introducing an interface to separate the domain from the adapters will allow us to keep the separation precise.

This domain-wrapping interface works like a *hub*, since it stays in the center of our application, and it's connected to the external by many functions that work like the spokes of a wheel.

The hub defines and abstracts upon the boundaries between the domain and the technical layer—in a very precise and oddly satisfying way. The domain stays inside the hub and only communicates with the rest of the application using specific functions. I learned this approach while working with Nat Pryce. It also makes it easier to test each component separately.[3]

So far, we have only implemented a single story, but for the sake of explanation, let's consider a domain that has to connect with two databases, a message queue, an email server, and the HTTP routes. Here is a diagram to illustrate our implementation of ports and adapter architecture with hub and spokes:

3. http://www.natpryce.com/articles/000772.html



In this conceptual diagram, anything inside the hub is domain related, functionally pure, and can communicate with external components only using its "spokes"—the arrows around it. In this way, the business logic can change without any change on the technical layers, like our HTTP functions. At the same time, if we need to change a technical detail, we don't have to touch the business logic at all.

Note that some arrows are pointing inward and some pointing outward from the hub; in the diagram, the arrows follow the direction of the call. Inward arrows are mapped on methods of the hub that can be called by the outer adapters. Outward arrows represent the dependency methods that the domain logic inside the hub needs to call.

Plug the Hub into Zettai

Going back to our application, we said that we should put our business logic inside the hub; what does this mean in concrete terms? We defined four functions in the previous chapter. Which ones are part of the domain in Zettai?

We can list them here with their signature:

Function Name	Function Type	Hub/Spoke
extractListData	(Request)-> Pair<User, ListName>	spoke
fetchListContent	(Pair<User, ListName>) -> ToDoList	hub
renderHtml	(ToDoList) -> HtmlPage	spoke

Function Name	Function Type	Hub/Spoke
createResponse	(HtmlPage) -> Response	spoke

If we consider our types: User, ListName, and ToDoList are part of our domain model, while HtmlPage, Request, and Response are technical details of our implementation. If unsure, a simple test is to check which names can emerge if we describe our business to a nontechnical person.

The functions that have a domain type both as input and output are part of our domain; in our case, there is only one. The others are part of the spokes that connect the hub with the external world.

So, we create an equivalent to the fetchListContent function inside the hub. Let's start writing a test to see how we would like to use our hub:

```
@Test
fun `get list by user and name`() {
    val hub = ZettaiHub(listMap)

    val myList = hub.getList(user, list.listName)

    expectThat(myList).isEqualTo(list)
}
```

Let's proceed, defining our ZettaiHub interface. We need only one function from the inside of the hub to the external, the one to retrieve a ToDoList:

```
interface ZettaiHub {
    fun getList(user: User, listName: ListName): ToDoList?
}
```

When implementing the hub interface, it's important to keep in mind two things:

- The inside of the hub should stay functionally pure, without any side effects and external interactions.
- The hub needs external functions to complete the functionality, so we need to provide them from the outside.

Now we can implement the hub for the ToDoList. It gets the map of to-do lists and users in the constructor and implements our function:

```
class ToDoListHub(val lists: Map<User, List<ToDoList>>): ZettaiHub {

    override fun getList(user: User, listName: ListName): ToDoList? =
        lists[user]
        ?.firstOrNull { it.listName == listName }

}
```

Joe asks:
Isn't It Premature to Create the Hub Interface Now?

It may seem excessive to create an interface with a single method and a single implementation, but it's simpler and faster to enforce a clean design from the beginning rather than retrofit it later in a poorly designed application.

The key point here is that we aren't trying to guess future needs; that would be against the principles of lean development. What we're doing is defining and respecting a principle (namely the separation between domain and infrastructure) from the very beginning, in the simplest possible way.

Then we pass the hub to the Zettai class constructor, in lieu of the map of lists:

```
data class Zettai(val hub: ZettaiHub): HttpHandler{

    //rest of the methods...

    fun fetchListContent(listId: Pair<User, ListName>): ToDoList =
        hub.getList(listId.first, listId.second)
        ?: error("List unknown")

}
```

Note that the Actions interface of our acceptance tests has a method with exactly the same signature. This isn't a coincidence; if we're keeping our tests close to the domain, the actor's actions will be quite similar to the methods of the hub.

Now we have good acceptance tests and a clean design with the domain separated from the adapters. We can take advantage of this fact and add another tool to our toolkit.