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

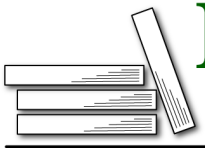
From Objects to Functions

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

Starting a New Story to Modify a List

Let's consider the second story: adding an item to a list. Writing a DDT isn't difficult, but starting from scratch can be a bit daunting. The trick is to think in terms of a use-case scenario: how would we describe it using small actors' steps? To make the DDT compile we can add the steps as empty functions in the actor until we're satisfied with the scenario code.

```
(1) class ModifyAToDoListDDT: ZettaiDDT(allActions()){

        val ann by NamedActor(::ToDoListOwner)

        @DDT
(2)     fun `The list owner can add new items`() = ddtScenario {

            setup {
(3)             ann.`starts with a list`("diy", emptyList())
            }.thenPlay(
(4)             ann.`can add #item to #listname`("paint the shelf", "diy"),
                ann.`can add #item to #listname`("fix the gate", "diy"),
                ann.`can add #item to #listname`("change the lock", "diy"),
(5)             ann.`can see #listname with #itemnames`("diy", listOf(
                  "fix the gate", "paint the shelf", "change the lock"))

(6)         ).wip(LocalDate.of(2023,12,31), "Not implemented yet")

        }
    }
```

(1) The name of the test class reflects the user story name.

(2) Each test represents a scenario of the story.

(3) We start with a list without items.

(4) Then, we call a new step to add an item three times.

(5) Finally, we check that the list has the three items.

(6) We mark the test as work-in-progress until it passes.

Note how we marked the new test with the `wip` method at the end. That's short for work-in-progress, since, as we said in the first chapter, our DDT won't pass until we finish the implementation at the end of this chapter.

Joe asks:

Why Do We Use Randomly Generated Values in the Unit Tests but Specific Examples in the DDTs?

This is a very good question. In the case of our unit tests, we want to make sure that we cover all corner cases and our functions are pure and total. For this reason, randomly generated values give us more confidence in the results, and since the tests are very fast, we can run them many times.

When it comes to tests that simulate actual usage scenarios, such as our DDTs, we're primarily interested in the cooperation between the development team and business stakeholders. Communication among different teams is simpler when discussing in terms of concrete examples, rather then in terms of logical relations, like the property tests assertions.

Work-in-Progress

In a normal unit test, we expect the test to fail for only a few minutes, and we'll definitely not commit it if it's not green. So, having a failing unit test isn't recommended.

Conversely, for DDTs and all end-to-end tests, it's acceptable for them to stay broken for days. They'll pass only when the story is completely finished.

We can see here a symmetrical principle: DDTs are the first tests we write and they'll be the last to pass. It's also a good practice to draft all the DDTs we need to complete the story when discussing with the business experts; in this way, we can quickly validate their utility and be sure not to forget important details. We can then proceed to implement and make them pass one by one, rechecking them often with the stakeholders.

To do this, we need a clear way to mark the tests we're working on, letting them run but ignoring the failure. We can also specify a tentative date to complete them, after which the test won't be ignored anymore. Then, if we forgot to fix them in time, they'll break the build, and it will remind us to complete them or to delete them if they aren't needed anymore.

Without the WIP notation, we have to remember which tests are supposed to fail and which aren't. This isn't a big issue right now since we have only two acceptance tests, but as soon as we start having many tests, the "accepted failures" would make the report very confusing.

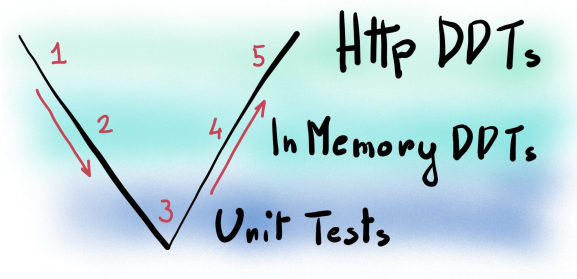
Domain-Driven Test Process

The process of working with DDTs looks like a V:

1. We start with the `Http` version first so we can sort out the "plumbing" of our architecture.

2. When we arrive at the point where we need some domain logic, we switch to InMemory/DomainOnly DDTs and let them guide us in modeling the domain.
3. Then, we develop the needed components one by one, using unit tests.
4. After that, we fix the DomainOnly DDTs until they pass.
5. Finally, we return to the Http DDT and make sure the final infrastructure is working as expected.

In the following "V" diagram, we're now at phase one of the DDT to modify a list. We're going to add the new methods on the actors and the HTTP actions so we can make the test compile first.



Joe asks:

Why Are We Starting from the HTTP and Not from the Hub?

As we discussed in the first chapter, writing functionalities starting from the external layers (the UI) and going to the internal ones is called "outside-in" style, whilst writing it from the internal domain and proceeding to the external layers is called "inside-out" style.

Which style should we use? There is no unique answer and it really depends on our constraints and acceptance criteria. DDTs are defined by the user actions on the external layer of the system; that's why starting from the HTTP layer makes sense.

On the other hand, if we wanted to develop a specific algorithm to solve a problem but we didn't care about the external layers, it would make more sense to use the inside-out style.

Actor Step

To make it compile we need to add a new step to the actor:

```
data class ToDoListOwner(override val name: String):
                                            DdtActor<ZettaiActions>() {

    val user = User(name)

    fun `can add #item to #listname`(itemName: String, listName: String) =
        step(itemName, listName) {
            val item = ToDoItem(itemName)
            addListItem(user, ListName(listName), item)
        }

    //rest of the methods
}
```

As we saw, the words starting with # in the method name will be replaced with the actual values when we run the test. Each step is defined inside the step method of the DdtActor. The step itself calls the addListItem on the actions with the correct parameters.

This is a general pattern. The actor steps only contain logic to call the actions, but they don't interact with the application directly. Here, we're sending a command to the application, so we don't have any result to verify. In case of steps that query the status of the application—like in the case of the can see #listname with #itemnames step—we would verify also that the result is what we expect.

HTTP Actions Call

To continue, we need to add the addListItem method to the ZettaiActions interface, so we can implement it in the HTTP and domain instance.

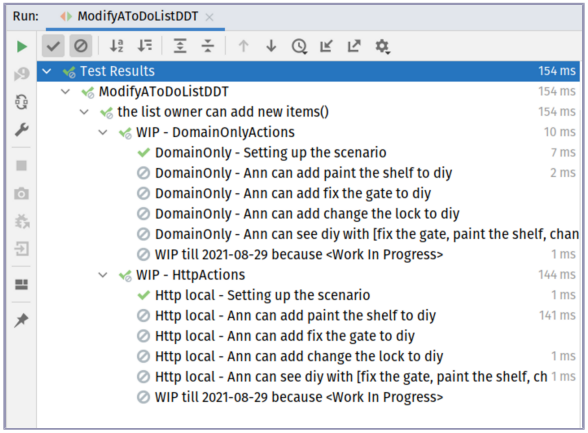
As we're in the first point of our "V" diagram, we'll leave the domain action with just a TODO in the implementation.

Instead, we'll start from the HTTP implementation. To simulate adding an item to a list, we need to submit an HTTP webform to the server with the item name and the item due date fields:

```
data class HttpActions(val env: String = "local"): ZettaiActions {

    override fun addListItem(user: User,
            listName: ListName, item: ToDoItem) {

        val response = submitToZettai(
            todoListUrl(user, listName),
            listOf( "itemname" to item.description,
                    "itemdue" to item.dueDate?.toString())
        )

        expectThat(response.status).isEqualTo(Status.SEE_OTHER)
    }
```

```
    private fun submitToZettai(path: String, webForm: Form): Response =
        client(log(
            Request(
                Method.POST,
                "http://localhost:$zettaiPort/$path")
            .body(webForm.toBody())))

//rest of methods...
}
```

Since the DDT now compiles, we can run it. This is how it looks in the IDE:



You can see that only the first step passes; the remaining steps are failing, but since the test is marked as work-in-progress, they are not failing the scenario.

Handle Different Pages

To progress on the implementation of our feature, we need a new HTTP endpoint to add a new item to a given list.

But, we have a new problem: so far we've created a function that returns an HTML page with a to-do list from a request, but this is only one among the many kinds of requests our web service will need to handle. How can we return different pages or API according to the details of the request?

We don't want to modify the function we already wrote; it's finished as it is and it shouldn't care about the other kinds of calls. We also don't want to write a function too specific with hardcoded values, because we couldn't reuse it. We want a generic reusable solution to this problem.

How can we combine together functions that handle different kinds of `Request` without changing our existing code? With another function, of course! More

precisely, we need a function that takes a collection of functions as input and returns a new function.

In order to learn how to code it, let's leave our Zettai app for a moment, and let's have a spike on a minimal web service for operating on some data to learn how to define routes using higher-order functions.