Component-plus-Strategy generalizes Ports-and-Adapters

Alistair Cockburn
Humans and Technology Technical Report 2022.01 (v3a, 2023-06-01)

Component + Strategy allows you to configure a subsystem to fit into slightly different environments. *Hexagonal Architecture* aka *Ports & Adapters* is a specific version of it that allows you to isolate a system from external technologies, vary those external technologies, and test the system in isolation from those technologies.

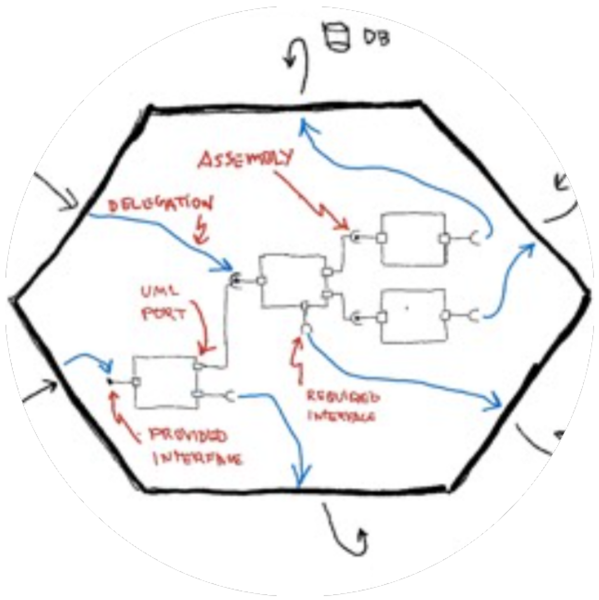

Table of Contents:

1. Warmup:

You quite naturally pass an object into a function so that the function can ask that object for more information or tell it to do something. This is normal object-oriented design.

For example, suppose you are programming a coffee machine that operates from recipes, you might pass in a recipe object to the drink-maker, so that the drink maker can get from it the sequence of ingredients to dispense.

Your code would look something like:

```
recipe = RecipeLibrary.find( "mochaccino" );
drinkmaker.make( recipe )
```

and inside the drinkmaker:

```
foreach step in recipe {
        dispenser = step.ingredient
        quantity - step.quantity
        dispenser.dispense( quantity )
}
```

Figure 1: The inevitable coffee machine

Although this is all really normal, it turns out to be quite subtle. It has been written about a lot, and given lots of fancy names.

First of all, we have **parameterized the recipes**, meaning, we choose which one to use according to the argument we pass in to the function. This is a really basic way to program, and should be fairly understandable. The main reason I mention it is that I want to be able to say in a bit: "parameterize the secondary actors." All I mean with that is that you pass in an argument that identifies which one to use.

It turns out we have just implemented the *Strategy* pattern. Many programmers don't use *Strategy* consciously, because it seems complicated in the *Design Patterns* book. So although they might use it reflexively, they don't describe their designs this way.

The Strategy pattern, very briefly, looks like this:

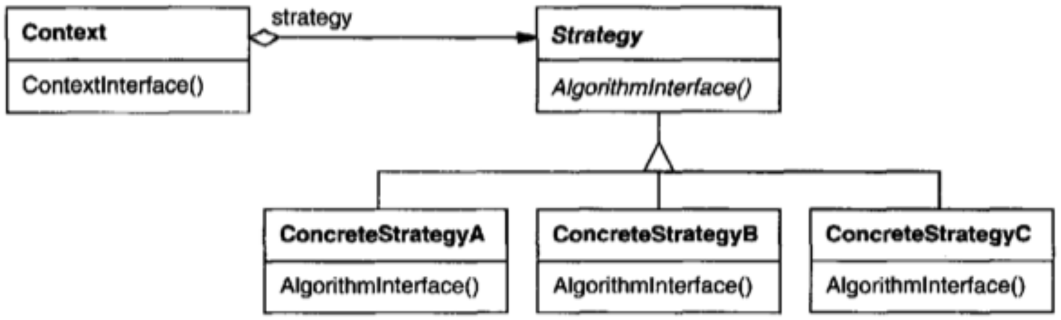

Figure 2: The *Strategy* pattern

The *Strategy* pattern only says that an object ("Context" here) has in its hands one of a set of possible objects that all respond to the same function call. Pretty ordinary polymorphism going on there.

In this drawing, the diagram shows the concrete strategies as *subclasses* of the top strategy class. But that only is needed in some languages. In languages such as Ruby, Smalltalk, and others, the concrete strategies only have to meet the function call interface, there is no need for an abstract superclass over them. This becomes important later.

What's cool about *Strategy* is that that polymorphism not only saves a bunch of 'if' statements, but the Context doesn't know or care which it has at the time of the call.

- Context may have calculated which one it needed earlier - for example, it may earlier have decided to use a time-optimal search or a space-optimal search, and obtained the appropriate search algorithm from somewhere, stuffed that search algorithm object into a safe place, and when needed, invoked whatever it had stored away.

- Or, the Context object might never know which concrete strategy object it is calling. Something, somewhere else, made that decision, and passed it in as a parameter. This is what we did with the recipe object.

The *Strategy* pattern doesn't tell us how the concrete strategy got loaded into the Context object. That is outside the scope of the pattern. As we discuss patterns in this article, we will pay attention to this - what does the pattern legislate versus which is outside the scope of the pattern.

Thirdly, we have just used what is known in UML as a **Required Interface**. The drinkmaker declares what calls it will make to its argument-collaborators, and they have to implement that. This is exactly what the *Strategy* pattern shows, above, although it is not evident to the casual reader that that is what is going on.

But we're not done with the example, yet. It also turns out that we have implemented the *Dependency Injection* pattern, one of the implementation possibilities of ***Configurable Receiver*** [https://alistaircockburn.com/Articles/Configurable-Receiver].

Wait! What?!

Let's draw a picture of the drinkmaker using UML notation.

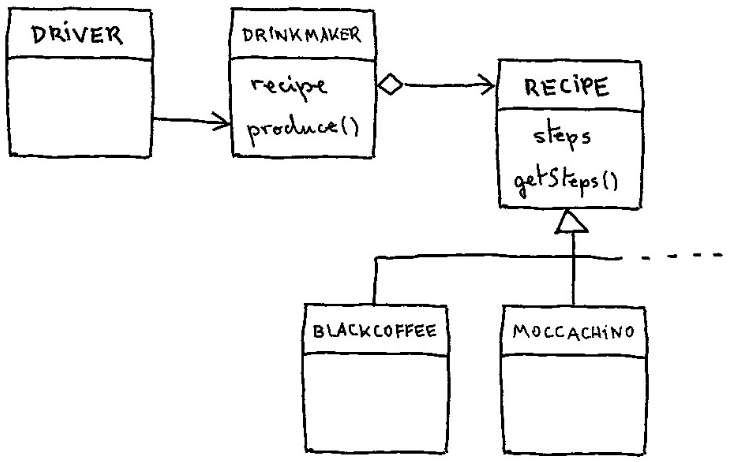

Figure 3: The drinkmaker example
(Image courtesy of Juan Manuel Garrido de Paz)

The simple arrowheads show a calling, or *uses* relation of the driver to the drinkmaker. The driver selects the recipe and passes it to the drinkmaker using what UML calls the drinkmaker's **Provided Interface.**

The open triangle shows an *implements* relation. Each recipe must implement the **Required Interface** of the drinkmaker. These are the same two arrowheads as in the *Strategy* diagram, except that the Strategy diagram does not show an outer driver calling the Context object, because that is out of scope of the pattern.

Because driver passes the recipe into the drinkmaker, the drinkmaker knows nothing about those other objects at programming time. It has no code-level dependencies on them. All knowledge it needs it obtains as needed during program execution. We like this, from a maintenance, testing, and reuse perspective.

To end this warmup section, what I am wanting to show here is how normal it is to pass an object as an argument to a function for further investigation, and how that simple act implements all of: **parameterized collaborator**, **Configurable Receiver, Dependency Injection, Strategy** and **Required Interface**. That's a lot of buzzwords for a fairly normal design practice

*   *   *   *   *

2. Introducing components and ports

I only just discovered in 2022(!) that UML contains a thing called **Component**, which has a **Provided Interface** or API on the driver side, and a **Required Interface** on the collaborator side. Further, **Component** has a thing called a **Port**, which is just a requirement that anything that plugs into the component must honor a protocol.

The UML spec says a Component is, "a modular unit with well-defined Interfaces that is replaceable within its environment".

> "A Component specifies a formal contract of the services that it provides to its clients and those that it requires from other Components or services in the system in terms of its provided and required Interfaces."

Here is the UML picture for a component

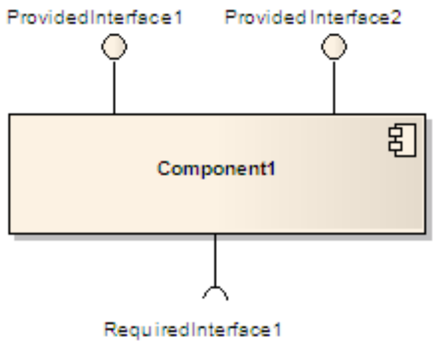

Figure 4: A UML Component with Provided and Required interfaces

A key property of components that is relevant to this article is that they can be nested - components inside of components - at any number of levels. This allows you to construct subsystems out of individual components and other subsystems. When we compare this to the *Ports & Adapters* or *Hexagonal Architecture* pattern, which doesn't nest, we will see this as a key difference between the two.

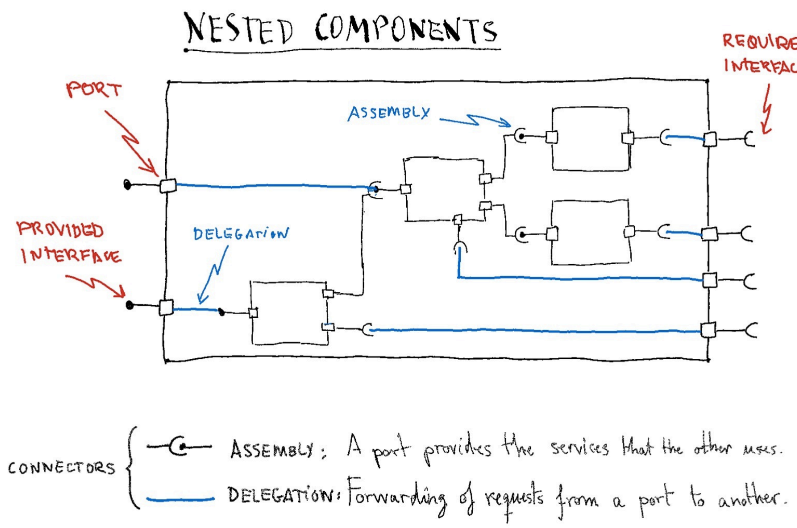
NESTED COMPONENTS
REQUIRED INTERFACE
PORT
ASSEMBLY
PROVIDED INTERFACE
DELEGATION
CONNECTORS
ASSEMBLY: A port provides the services that the other uses.
DELEGATION: Forwarding of requests from a port to another.

Figure 5: Components can be nested
(Image courtesy of Juan Manuel Garrido de Paz)

(formerly: https://www.uml-diagrams.org/component-diagrams/component-diagram-overview.png

Finally, we must note that we have shifted from a pure *modeling* discussion to one that includes *packaging*. The packaging is conceptual at the first hand, because we are asserting that a collection of things has a boundary and specified set of ways to interact with it. It may also be physical, in terms of being a stored or deployable unit.

What we're going to do now is a bit usual, we're going to blend a packaging discussion with a modeling discussion into one pattern. We are going to configure our Component with a Strategy

But first we have to ask: *Strategy*? or *Adapter*?

*   *   *   *   *

3. Strategy, Adapter, or both?

The *Adapter* pattern is a special case of the *Strategy* pattern in which the concrete strategy will make some adjustments for interface compatibility and then call another service to take care of the request. The big difference between the two is that *Adapter* has an additional level of indirection. The strategy may or may not do all its work itself, but we intend the adapter to connect to something else.

A *Strategy* object can, of course, do all this - that is outside the pattern definition - but we *expect* the *Adapter* to do this.

Now, I know the names are different, but for a moment I just want to look at the structure of the code, because we'll make use of that.

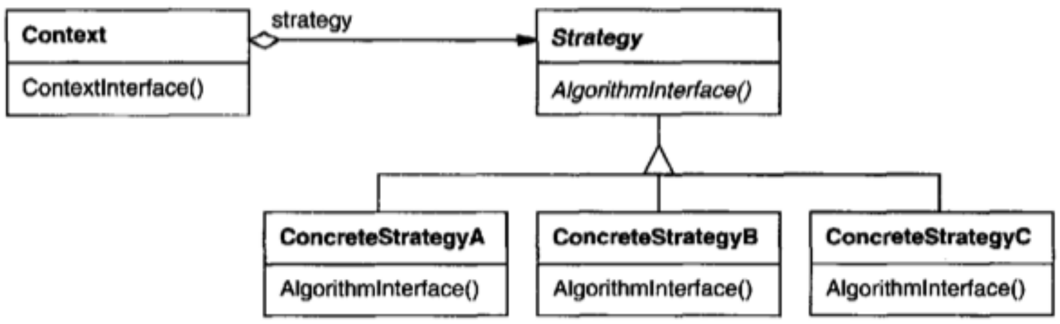

Figure 6: The *Strategy* pattern again

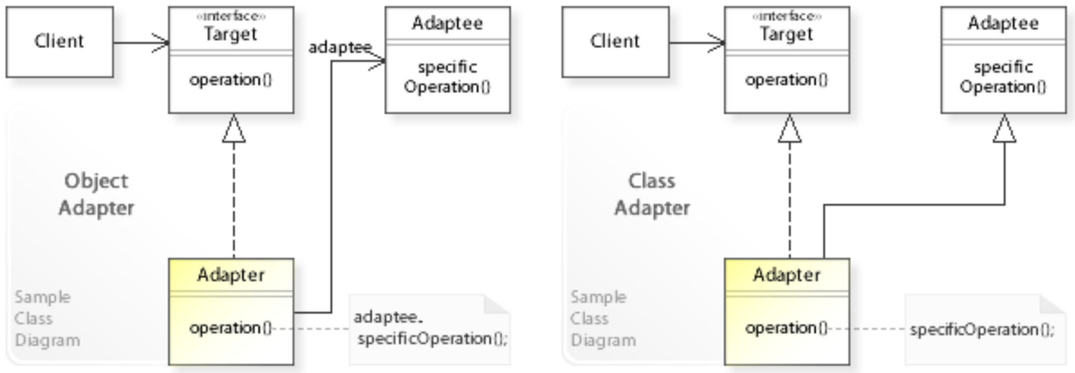

Figure 7: The *Adapter* pattern

http://www.w3sdesign.com/GoF_Design_Patterns_Reference0100.pdf

You might notice that in the diagrams, the *Adapter* picture only shows one adapter, where the *Strategy* picture shows several, we imagine swappable, concrete strategies. But this is only in the drawing. In ordinary working, we are quite likely to send a message out to a web channel, a text message, or something else, and vary that during program execution. So, there are just as likely to be several concrete adapters classes that get called and swapped at program configuration time or run time.

Where this similarity between the patterns this becomes useful is that you can combine Strategies and Adapters. The following example from shows them together. The first indicated strategy might do all the work itself, the second uses a third object to complete the work.

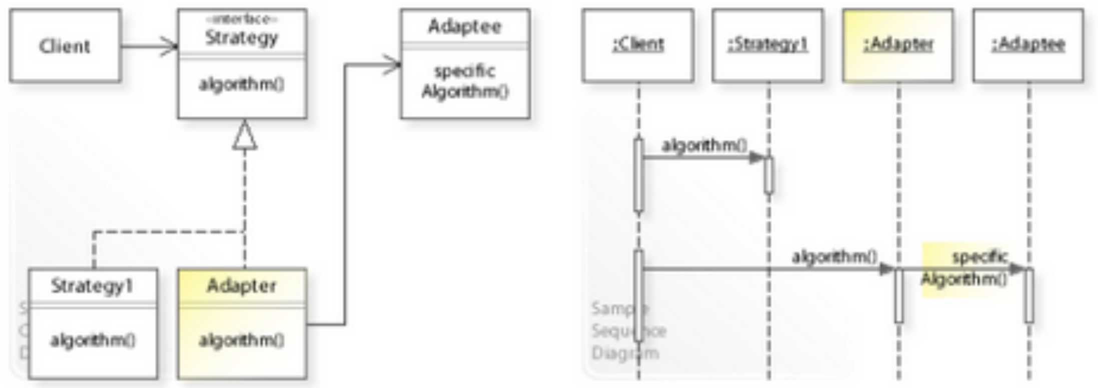

Figure 8: Using *Strategy* and *Adapter* together
http://www.w3sdesign.com/GoF_Design_Patterns_Reference0100.pdf

We will make use of this combined pattern in testing our component.

For the above reasons, I call the pattern in this article *Component + Strategy*. I will specialize it to the *Ports & Adapters* or *Hexagonal Architecture* pattern.

*   *   *   *   *

4. Component + Strategy : The pattern

Component + Strategy allows you to configure a subsystem to fit into slightly different environments.

Because we have a packaging element connected to a modeling element, I show the pattern in two diagrams, a component diagram with an object hanging off it, and a class diagram showing the component as a single class, even though it probably consists of many.

First, just to get us used to looking at them, here is the *Strategy* pattern shown as a component diagram.

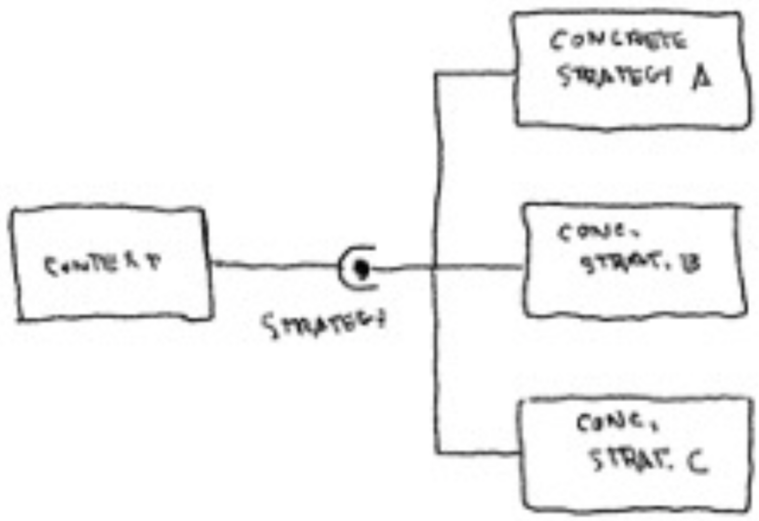

Figure 9: *Strategy* as a component diagram
(Image courtesy of Juan Manuel Garrido de Paz)

Next, here is *Component + Strategy* as a component diagram:

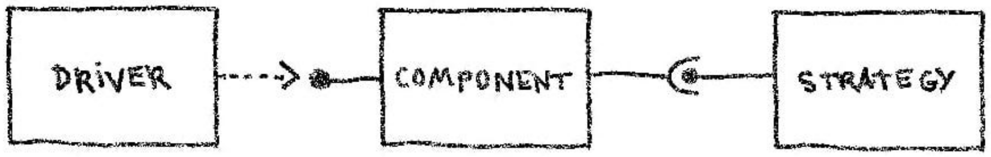

Figure 10: *Component + Strategy* as a component diagram
(Image courtesy of Juan Manuel Garrido de Paz)

Finally, here is *Component + Strategy* as a class diagram

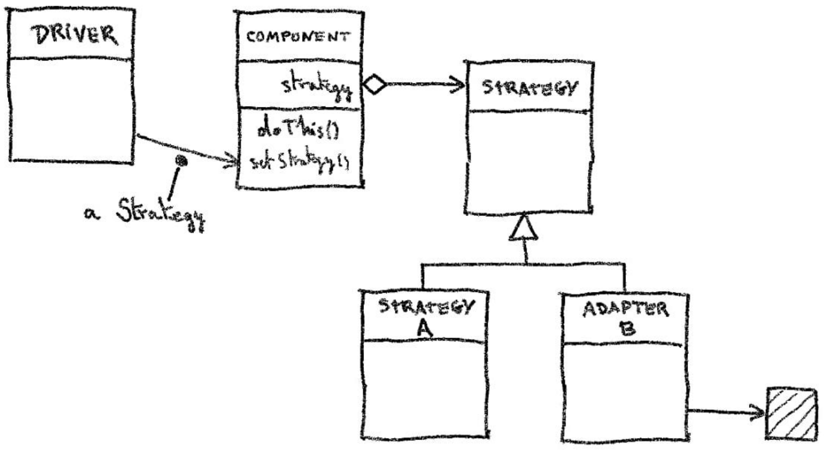

Figure 11: *Component + Strategy* as a class diagram
(Image courtesy of Juan Manuel Garrido de Paz)

One of the benefits of using *Component + Strategy* is that by declaring the component boundary explicitly, you can supply a test double as the strategy for one of external actors and thus test the component in isolation. Then for production use, supply an adapter to do the real connection.

And now our Strategy-Adapter discussion becomes relevant. The test double might not be connected to a test database. If it is not, then the test double fits the definition of a *Strategy* object, as we discussed above. If it is connected to a test database, then it is arguably an adapter. Personally, I am not fussed which way you call it, I don't consider that argument worth the time fighting over it. I am only going into this detail here because this is the pattern definition, and I would like to be as accurate as possible with the terms.

In the end, I am choosing the name *Component + Strategy* instead of *Component + Adapter* because *Strategy* is the more general of the two.

*   *   *   *   *

5. Ports & Adapters (Hexagonal Architecture) revisited

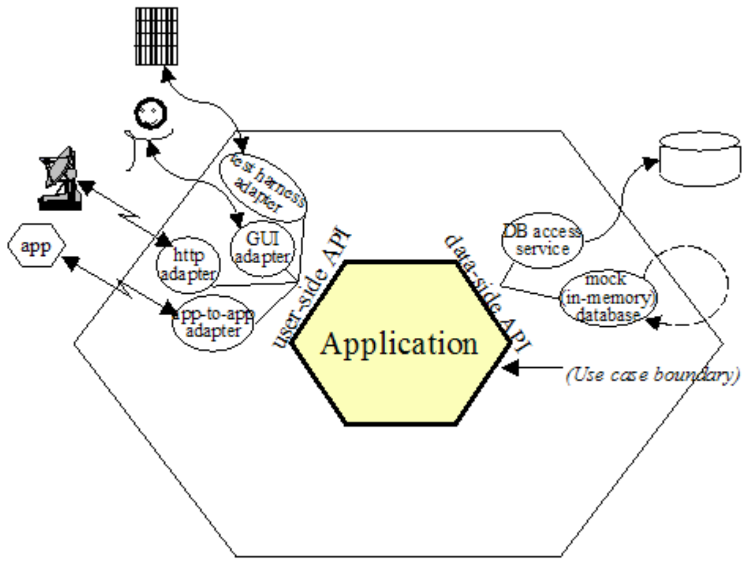

Figure 12: *Ports & Adapters* aka *Hexagonal Architecture*

Given the above, we can now see that the *Ports & Adapters* also known as *Hexagonal* architecture is a specific use of *Component + Strategy* where the component boundary is placed just in front of external technologies. *Adapter* objects are supplied for each port to adjust to the **Provided Interface** or **Required Interface** of the component.

- In the case of direct module-to-module interaction where the interfaces are compatible, no adapter may be needed.

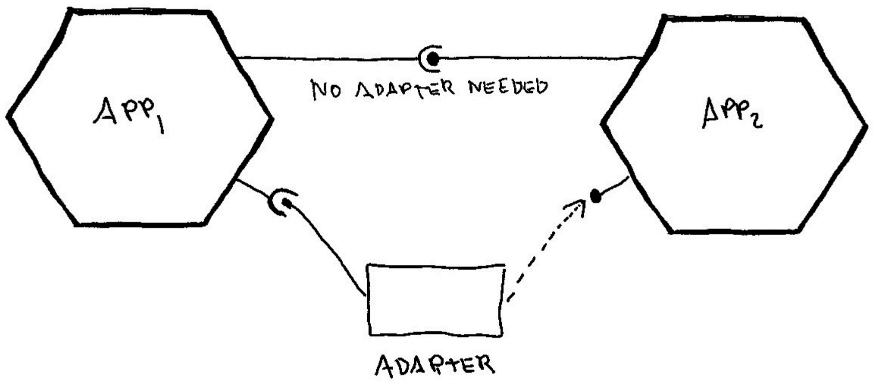

Figure 13: *Apps interacting with and without needing adapters*
(Image courtesy of Juan Manuel Garrido de Paz)

- In testing, test doubles may be either *Strategy* or *Adapter* objects.

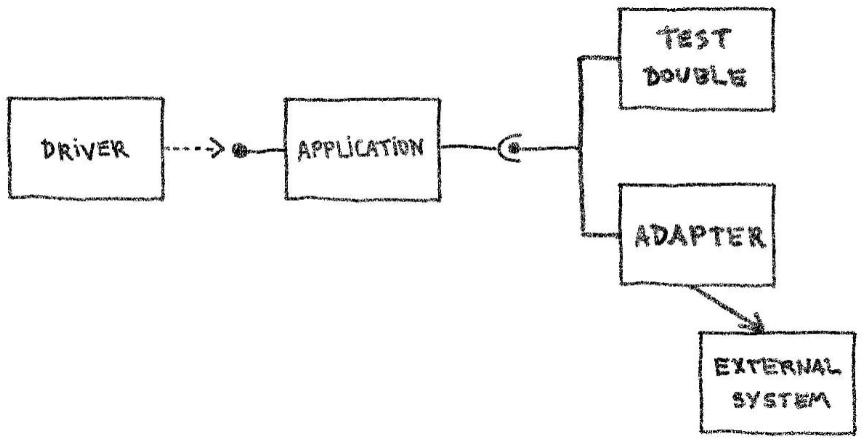

Figure 14: *Ports & Adapters* as component diagram showing test double
(Image courtesy of Juan Manuel Garrido de Paz)

One of the differences between the two patterns is that a key intention of *Ports & Adapters* is to protect against external technology changes. Hence, its boundary is placed only at the technology boundary. It is not nestable, unlike *Component + Strategy*, which is designed to be nested.

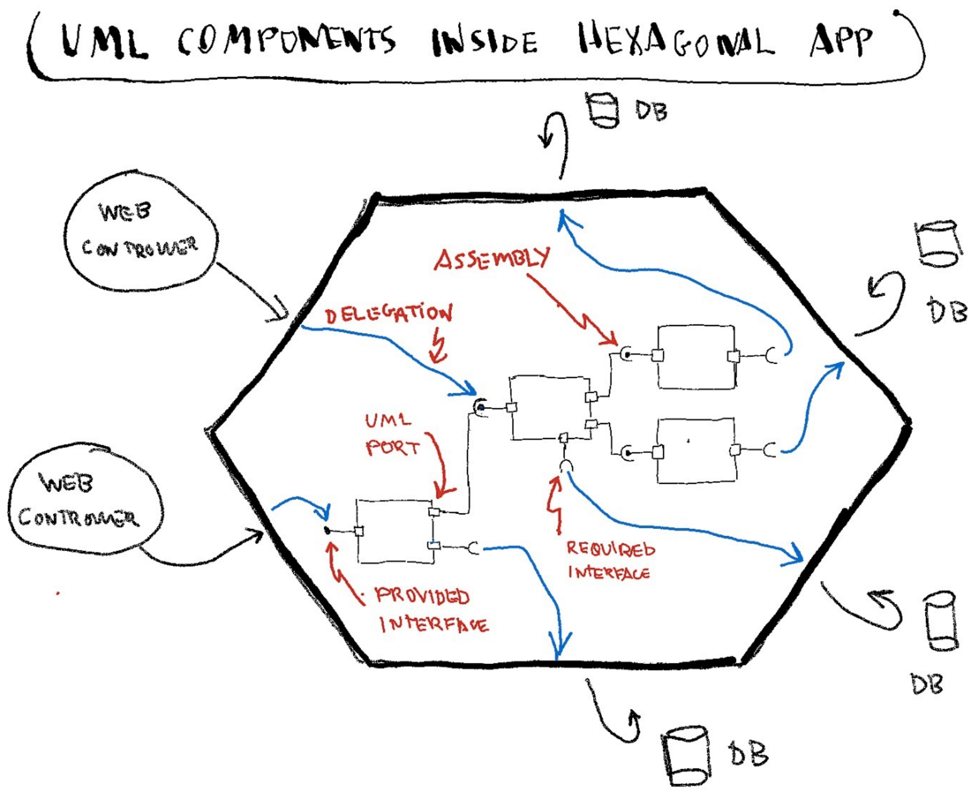

Figure 15: *Components* within *Ports & Adapters*
(Image courtesy of Juan Manuel Garrido de Paz)

Testing should be the same for *Ports & Adapters* as for *Component + Strategy*. Place a test driver or test double at each port to test the component in isolation.

For the definition of *Hexagonal Architecture*, see https://alistair.cockburn.us/hexagonal-architecture/

* * * * *

6. The Hidden Fourth Object, the Configurator

All of the preceding diagrams and discussions skipped over an important question:

How do all these objects come to know of each other?

The *Strategy* diagram doesn't show how the Context object came to know which concrete strategy to use. That is outside the scope of the pattern. The same is true for both *Component + Strategy* and *Ports & Adapters*.

However, sooner or later there has to be some module or code that knows all the players and introduces them to each other. That's where source-code dependencies lie. This is the *Configurator* object. There are a few other solutions, but this one is the most common.

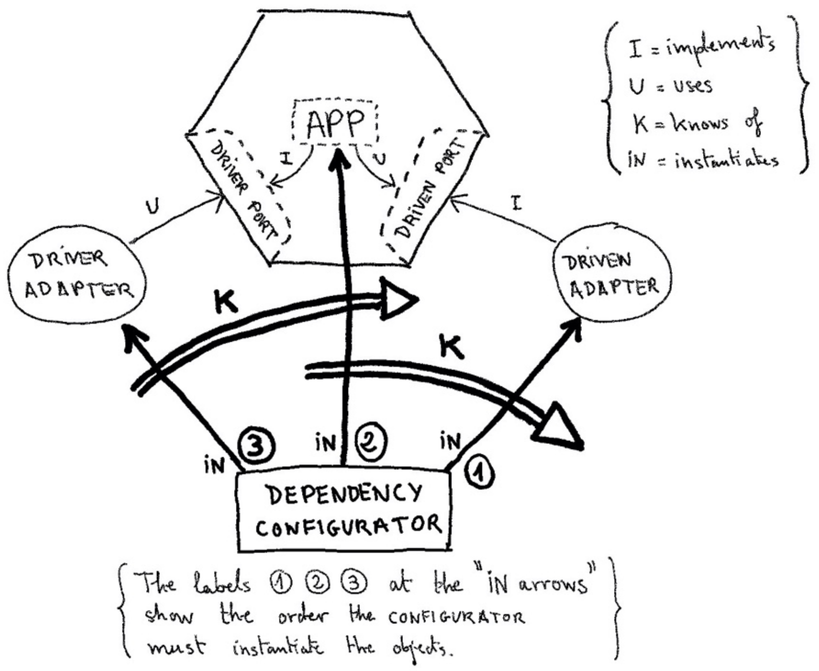

Figure 16: The *Configurator* sets up the knowledge paths
(Image courtesy of Juan Manuel Garrido de Paz)

When you test at the system level, there is no UI and there are no databases. You have a test harness driving the **Provided Interface**, and a test double handling the requests at the **Required Interface**.
You write a module where you instantiate all three of them: the test harness, the test double, and the component, you tell the test harness to use the component, and send the test double in to the component as the concrete strategy at the **Required Interface**. Then you tell the test harness to go, and it all runs.

In early stages of development, each test case does all that wiring and then runs the specific test. Here, the configurator is inside each test case.

Then, for production use, all those same instantiations take place in the program build and startup. The startup module will instantiate the component, the UI, and the adapters to the relevant databases and other actors. Depending on your design, the configurator may pass in the strategy objects to the component or that may be the assignment for the UI or another module. In all cases, the configurator knows all the players and what they need.

Because the Configurator is outside the pattern definition - exactly how and where all that knowledge acquisition happens - we generally don't see the Configurator getting talked about. In order to make the patterns useful, though, we need to make it explicit.

For completeness, Juan Manuel Garrido de Paz was kind enough to contribute this Spring code that illustrates:

```
@Configuration
public class SpringDiscounterAppConfigurator {

  @Bean
  @ConditionalOnProperty(name = "for-managing-discounts", havingValue
= "test-cases")
  public Driver testCasesDriver ( ForDiscounting discounterApp ) {
    return new TestCases( discounterApp );
  }

  @Bean
  @ConditionalOnProperty(name = "for-managing-discounts", havingValue
= "cli")
  public Driver cliDriver ( ForDiscounting discounterApp ) {
    return new Console(discounterApp);
  }

  @Bean
  public ForDiscounting discounterApp (ForObtainingRates rateRepository )
{
    return new DiscounterApp ( rateRepository );
  }

  @Bean
  @ConditionalOnProperty(name = "for-obtaining-rates", havingValue =
"test-double")
  public ForObtainingRates testDoubleRateRepository() {
    return new StubRateRepository();
  }

  @Bean
```

```
    @ConditionalOnProperty(name = "for-obtaining-rates", havingValue =
"file")
    public ForObtainingRates fileRateRepository() {
        return new FileRateRepository();
    }

}
```

*   *   *   *   *

7. Tests or no tests?

One of the things that makes my blood freeze when seeing people claim they have implemented *Ports & Adapters* or *Hexagonal Architecture* is the absence of tests on both sides.

If you place hexagons everywhere, at different levels inside the system, then there will be too much repetition between the hexagon boundary tests and the system tests. Not being worth the time to write and maintain both, the team will likely stop writing tests for the inner hexagons, at which point it ceases to be a real Component.

One reason I like UML's *Component* is that simply by using the word 'component', you should feel obligated to write tests at all of the boundaries. I mean, it is called a "component" after all, and is intended to be placed in different systems and circumstances. Of course there should be tests at the declared boundaries.

Perhaps because the *Ports & Adapters Architecture* pattern never explicitly says it is a component, people think of it only as a conceptual interface, a "nice thought", but not really something to write tests at.

My hope is that by writing *Component + Strategy*, and then making it clear that *Ports & Adapters* is a special case of that general pattern, people will start to treat these boundaries as real system boundaries, and hence worth the trouble of writing tests to.

It is for this reason that I am adamant that *Ports & Adapters* aka *Hexagonal Architecture* is placed at the outer, technology boundary. At that interface, the tests are meaningful system tests, worth maintaining. The application becomes a component in the sense we intend, and gets its proper regression tests.

*   *   *   *   *

8. End Notes

The point of the opening example was to show how simple and ordinary our design was. We simply parameterized an external resource, then passed in an object that let us get the appropriate one at run time.

The difference between *Component + Strategy* and *Ports & Adapters* or *Hexagonal Architecture* is that *Ports & Adapters* is aimed at solving one very specific problem - changing external technologies (and testing) - whereas *Component + Strategy* is intended as a general subsystem-bounding effort.

I would like to see increased use of *Component + Strategy* as a packaging concept that allows arbitrary sub-sections of code to be protected by a test wall and configured to their environments.

Alistair Cockburn
Humans and Technology Technical Report 2022.01