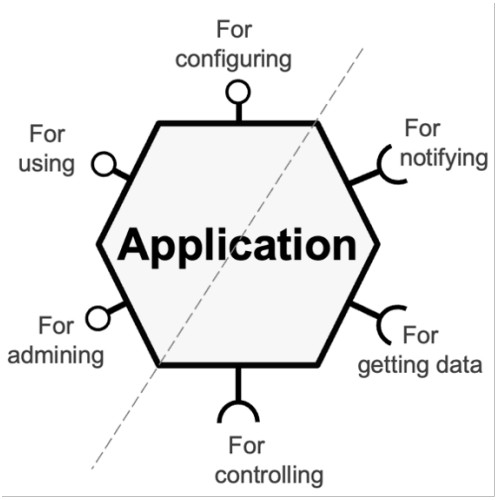
The updated pages to…

Hexagonal Architecture Explained

How the Ports & Adapters architecture simplifies your life, and how to implement it

Updated 1st Edition



Alistair Cockburn

Juan Manuel Garrido de Paz

.

ISBN 979-8-9985862-0-0 for paperback
ISBN 979-8-9985862-1-7 for ePub
Humans and Technology Press
5325 20th Ave S
Gulfport, FL 33707
v1.1b 20250420-1012 for paper&ePub books

Other books by Alistair Cockburn

1997 *Surviving Object Oriented Projects*
https://www.amazon.com/Surviving-Object-Oriented-Projects-Alistair-Cockburn/dp/0201498340

2000 *Writing Effective Use Cases*
https://www.amazon.com/Writing-Effective-Cases-Alistair-Cockburn/dp/0201702258

2001 *Agile Software Development (1st ed)*
https://www.amazon.com/Agile-Software-Development-Alistair-Cockburn/dp/0201699699

2002 *Patterns for Effective Use Cases*
https://www.amazon.com/Patterns-Effective-Cases-Software-Development/dp/0201721848

2003 *People and Methodologies in Software Development (Dr. Philos.)*
https://web.archive.org/web/20140329203845/http://alistair.cockburn.us/People+and+methodologies+in+software+development

2004 *Crystal Clear: A human-powered methodology for small teams*
https://www.amazon.com/Crystal-Clear-Human-Powered-Methodology-Development-ebook/dp/B00B8UX6K2

2006 *Agile Software Development: The cooperative game (2nd ed)*
https://www.amazon.com/Agile-Software-Development-Cooperative-Game-ebook/dp/B0027976NG

2021 *Design in Object Technology: Class of 1994*
https://www.amazon.com/Design-Object-Technology-Class-Object-Oriented-ebook/dp/B09GPP9K1L

2022 *Design in Object Technology: The Annotated Class of 1994*
https://www.amazon.com/Design-Object-Technology-Annotated-Class-ebook/dp/B0BFJYTRFP

2022 *Love Trio Trio del Amor (selected poems)*
https://www.amazon.com/Love-Trio-Amor-Alistair-Cockburn-ebook/dp/B0BPCCHZCG

2024 *Unifying User Stories, Use Cases, Story Maps (preview ed.)*
https://www.amazon.com/Unifying-User-Stories-Cases-Story-ebook/dp/B0D4JSQ5DY
ePub directly: https://store7710079.company.site/Unifying-User-Stories-Use-Cases-Story-Maps-epub-p655931612

2024 *Hexagonal Architecture Explained (preview ed)*
https://www.amazon.com/Hexagonal-Architecture-Explained-Alistair-Cockburn/dp/173751978X

2025 *Hexagonal Architecture Explained, Updated 1st ed.*
ePub directly: https://store7710079.company.site/Hexagonal-Architecture-Explained-ePub-p655931616

Kudos for the book

I just wanted to say thank you for hexagonal architecture.

My team used to do it for a while and we finally we got it right. Making changes to the services we hexagonalized properly feels really good and easy.The same changes kept giving us headaches in other services.

Your recent book recently helped us to gain confidence that we are doing it right and to see where ideas from other patterns were mixed in by people blogging on the topic.

Michael Kutz
Software Engineer at REWE digital GmbH

* * * * *

The publication of this book has been a great joy for several reasons. One of them is personal, as you might have guessed if you follow this blog. The other reason is that we finally have an authoritative reference guide to the pattern.

it is a very complete and detailed work. You can consider it a must-have reference both theoretically and practically, as it offers a fairly comprehensive implementation guide.

If you're really interested in understanding it and, possibly, using it in your projects, the book is the best source available, and it also includes the few original references you could find online.

Fran Iglesias,
Staff Software Engineer at Qualifyze

* * * * *

Been deep in Hexagonal Architecture lately—pure gold from @TotherAlistair & Juan Manuel Garrido de Paz. Highly recommend (get the book!)

Eugene F. Barker

* * * * *

I found the book to be very simple and practical. In fact, I used a few of its ideas in some refactoring I'm doing at work and they made a real difference.

What's more, it contains some detailed DDD discussion and its relation to Ports and Adapters!

Rubyists might be especially pleased to find examples of how to implement that architecture in Ruby (I definitely was!).
I strongly recommend it.

Hemal Varambhia
Senior Technical Coach

* * * * *

About the preview edition:

“It gives interesting insights not just on how the pattern can be implemented, but also on its story and the design considerations that revolve around it.”

About the additions in the updated 1st edition:

“I find that the additions you did are very valuable not just in terms of understanding the pattern, but also in terms of understanding how it fits with the existing literature, related patterns, testing strategies etc.”

Eleonora Ciceri

Preface

This is the full 1st edition

In early 2024, after working on this book for nearly 5 years, Juan and I (Alistair) felt the pressure to get it out "Now!". We decided to publish a *preview edition*, containing all the critical information, but possibly not in the best order, or needing better explanations in places.

That decision turned out to be prescient. Juan passed away very suddenly, just three weeks before the book was set to go to production. The preview edition came out barely in time for Alistair's visit to his home town, Sevilla, an emotional event, for sure. Happily, the book contains Juan's best thoughts up to that time.

Since then, I have watched numerous discussions online, I have taught to the book, found a few more topics for the FAQ section, and found one – almost humerously wrong – error in the first code sample. It is a credit to Juan that we argued so ferociously over the content that the content itself remains stable.

There are no significant changes to the original version, mostly I added a few pages of extra notes and fixed minor mistakes. The new text is marked with this solid gray bar down the side.

Since this update is from me alone, I will freely use the word 'I' and will feel free to add small anecdotes to make your reading a little more fun.

Alistair, April 10, 2025

(Preface to the *Preview* edition)

Juan and I feel it important enough to get this into your hands that we are publishing this edition before what normally constitutes fine tuning the book: sending to reviewers incorporating their changes, creating an index of key words, tuning page layout and so on. That process would take up to another year, and we feel you need this information today.

This edition has all the information we have at hand as of April 2024, in the best order we can think of. In other words, you can use it. Following

1.1. Copy this code

The Ports & Adapters architecture, first documented in 2005 as the "Hexagonal Architecture" pattern, demands this:

> *Create your application to work without either a UI or a database so you can run automated regression-tests against it, change connected technologies, protect it from leaks between business logic and technologies, work when the database becomes unavailable, and link applications together without any user involvement.*

The most surprising part of implementing it is this requirement:

> *"Never explicitly name any external object or technology. Always take a parameter for any external object or technology you wish to access."*

That requirement has a weak and a strong implementation. In the "weak" implementation, the programmer knows that the database will use SQL (for example), and without tying to a particular database, still expresses the interface in SQL. While technically meeting the rules of the Ports & Adapters architecture, that still handcuffs the system to SQL, which is not what we are after.

To get a full, or "strong" implementation of the Ports & Adapters architecture, we need:

> *"The app cannot know anything about the external technology."*

That is, the Service Provider Interface (SPI) or "driven port" is expressed purely in terms of concepts that make sense in the language of the domain. It can't even know that there is a database, let alone SQL.

The easiest way to show this is with a bit of code. The code is much simpler than all the discussions of why the code looks that way.

Therefore, to get started, replicate this code snippet in your larger system. This Java code shows the interface definitions explicitly:

```
interface ForCalculatingTaxes {
   double taxOn(double amount);
}
```

```
interface ForGettingTaxRates {
   double taxRate(double amount);
}
```

```
class TaxCalculator implements ForCalculatingTaxes {
   private ForGettingTaxRates taxRateRepository;
   public TaxCalculator(ForGettingTaxRates taxRateRepository) {
      this.taxRateRepository = taxRateRepository;
   }
   public double taxOn(double amount) {
      return amount *  taxRateRepository. taxRate( amount );
   }
}
```

```
class FixedTaxRateRepository
      implements ForGettingTaxRates {
   public double taxRate(double amount) {
      return 0.15;
   }
}
```

```
class Main {
   public static void main(String[] args) {
      ForGettingTaxRates taxRateRepository = new
                                          FixedTaxRateRepository();
      ForCalculatingTaxes myCalculator = new TaxCalculator(
                                             taxRateRepository );
      System.out.println( myCalculator.taxOn( 100 ) );
   }
```

The preview edition contained a mistake. Without studying the previous code, see if you can find it in this original version:

```
interface ForCalculatingTaxes {
   double taxOn(double amount);
}
```

```
interface ForGettingTaxRates {
   double taxRate(double amount);
}
```

```
class TaxCalculator implements ForCalculatingTaxes {
   private ForGettingTaxRates taxRateRepository;
   public TaxCalculator(ForGettingTaxRates taxRateRepository) {
      this.taxRateRepository = taxRateRepository;
   }
   public double taxOn(double amount) {
      return amount *  taxRateRepository. taxRate( amount );
   }
}
```

```
class FixedTaxRateRepository
      implements ForGettingTaxRates {
   public double taxRate(double amount) {
      return 0.15;
   }
}
```

```
class Main {
   public static void main(String[] args) {
      ForGettingTaxRates taxRateRepository = new
                                              FixedTaxRateRepository();
      TaxCalculator myCalculator = new TaxCalculator(
                                                  taxRateRepository );
      System.out.println( myCalculator.taxOn( 100 ) );
   }
```

I could say, “almost comical,” because Ricardo Guzmán Velasco (@RGVgamedev on Twitter) found it at the book launch. He came up and said he didn’t understand why I needed the driving port declaration. I went to explain, pulled my finger down the code, and went, “Crap.” He found the mistake within minutes of launch. Sigh.

The mistake is giving myCalculator type TaxCalculator, that is, typing with the class instead of the interface. With that mistake, the interface definition at the top is meaningless.

What followed over the next months was interesting. Some people wrote in and said that the interface declaration was important:

* Convention: It is the standard programming convention in languages that have that feature to type by interface, not class.
* The interface declaration is intended to provide the minimum interface that we want to expose.
* If every client couples to TaxCalculator, you lose the freedom to change its implementation. If you create another ForCalculatingTaxes implementation, you have to change all clients when you want to switch the implementation.
* The purpose of type-checking is to catch a certain class of errors at compile time. Declaring the type as the class and not the interface defeats the purpose of typing. You lose the safety you thought you were getting.

Others wrote to say that there is no real problem in typing the variable with the class because for an app, the public methods are probably exactly the interface it should export, and you’re unlikely to make a second implementation of the app. Shoutout to Nicky Ramone (@nickyramone77) and Chris F Carroll (@chrisfcarroll.bsky.social) for these insights.

For them, the interface declaration at the top is unnecessary, which means the published code is still not right.

In the end, it seems there are two reasonable schools of thought, each with its own defenders.

In one, declare and use the interface declaration:

```
interface ForCalculatingTaxes {
   double taxOn(double amount);
}
```

```
class TaxCalculator implements ForCalculatingTaxes {
   … (public methods) …
}
```

```
class Main {
   …
      ForCalculatingTaxes myCalculator = new TaxCalculator(
                                                taxRateRepository );
}
```

In the other, don't declare it. Just use the class:

```
~~interface ForCalculatingTaxes {~~
~~   double taxOn(double amount);~~
~~}~~          (don't write this code)
```

```
class TaxCalculator {
   … (public methods) …
}
```

```
class Main {
   …
      TaxCalculator myCalculator = new TaxCalculator(
                                                taxRateRepository );
   }
```

My mistake was having a foot in each camp, declaring the interface and then not using it.

In your life, decide which way you prefer to write.

The difficulty of naming

We sweated over naming. The thing is, there are three things to talk about: actor, adapter, and port. We need two adjectives for each and tried all of these: driving/driven, inbound/outbound, primary/secondary, API/SPI, left/right.

In the preview edition we used driving/driven and primary/ secondary. Some people found these terms difficult to use, and wrote inbound/outbound or API/SPI instead.

Only *driving/driven* and *primary/secondary* apply to all three, actor, adapter and port. You can talk about a driving actor, a driving adapter, a driving port, and similarly for the driven side. You can also talk about a primary actor, a primary adapter, a primary port, and similar for secondary.

But you can't say "inbound actor" and "outbound actor." Similarly, "API actor", "SPI actor" make no sense.

Personally, I (Alistair) don't mind synonyms. If you like inbound/ outbound port, and inbound/outbound adapter, that's fine. For the ports, having "API ports" and "SPI ports" makes sense, since ports are just interfaces anyway.

Where you might find "inbound" and "outbound" most useful is in naming your folders. Alistair never liked seeing two folders next to each other called Driven Adapters and Driving Adapters. They are just too similar. Calling them Inbound Adapters and Outbound Adapters seems like a good idea. Alistair has also seen "Provided/Required" and "Controllers/Providers." (More on folder structure in Chapter 4.8: Where do I put my files?)

In this book we stick with driving/driven and primary/ secondary, so that we can apply the same adjective to actor, adapter and port. But in your life, feel free to use inbound /outbound for your ports and adapters, if you like, or API /SPI for your ports if that's all you're talking about.

Weak versus strong conformance to the pattern

You can implement this pattern in a legal but weak way. Suppose you know that the database will use SQL. Without tying to a particular database, you still express the driven port in SQL. While technically meeting the rules of the architecture, that still ties your system to SQL, which is not what we are after.

To get a proper, or strong implementation of the Ports & Adapters architecture,

> the app cannot know *anything* about the external technology.

That is, the driven port is expressed purely in terms of concepts that make sense in the application language. It can't even know that there a database, let alone an SQL one.

3.3. The BlueZone example

The BlueZone is Juan's full example of how the pattern works.

Note (2025) Juan kept evolving his code, creating two designs in two repositories. A book like this can't keep up with the changes, so in this chapter I'll outline one of his designs, and let you compare the two designs he left. Check:

https://github.com/jmgarridopaz/bluezone

https://github.com/HexArchBook/bluezone_pro

BlueZone allows car drivers to use a web UI to pay for parking at various zones in a city. Different colored lines on the road indicate different parking rates; for example, central downtown is more expensive than a few blocks out. After possibly looking up the rates of different zones, the driver buys a ticket for a zone for a set time, paying by various means.

The parking inspector will check whether parked cars have paid correctly for their zone and time.

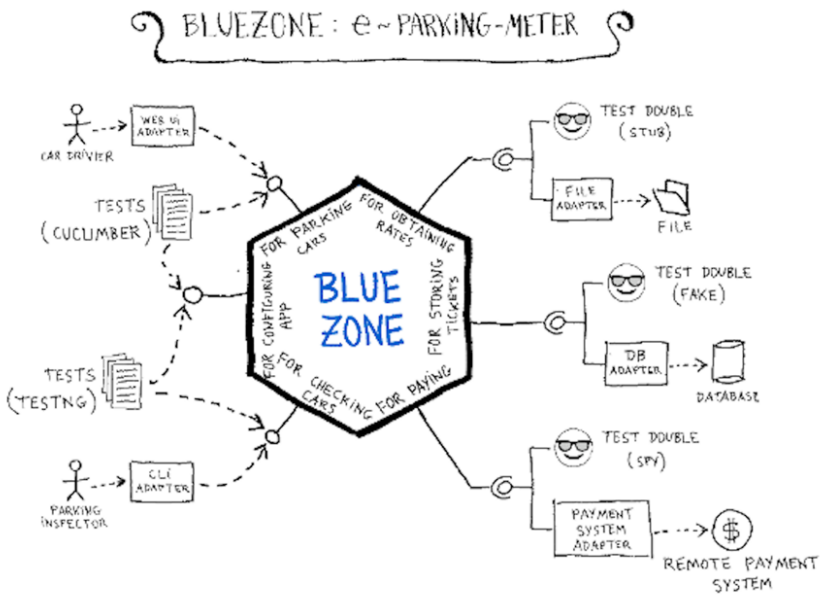


Figure 3.1. The actors in the BlueZone example

Suppose you have two driving ports, one for a user calculating taxes and another for the admin person doing general admin things.

The folders (and also the port and interface) names will be "for_calculating_taxes" and "for_admin_purposes." Actual naming and coding is up to your personal standards, they are not part of the pattern.

For the Test Cases folder, organize as you like.

In the driving adapters and driven adapters folders, make a subfolder for each adapter.

Naming your folders.

As described in the glossary, chapter 2.1, you have several choices for how to name the folders. In this book, we continue to write “Driven/Driving Ports” and “Driven/Driving Adapters.” However, some people find those words confusing, so they call the folders “Inbound/Outbout Ports” and “Inbound/Outbound Adapters.” A recent proposal was to write: “Provided Interfaces” and ‘Required Interfaces.” Feel free to use any of these alternatives if you like.

Figure 4.6 shows three ways of setting them up. The top one shows driving/driven port definitions inside the app project. The second has the ports into their own folder. In the third, I show how it looks if you call them Inbound/Outbound. Your choice. These decisions are not mandated by the pattern.

Folder structures that don't match the intentions of the pattern cause confusion and even damage to the project. Create clarity in your project by implementing them in one of these ways.

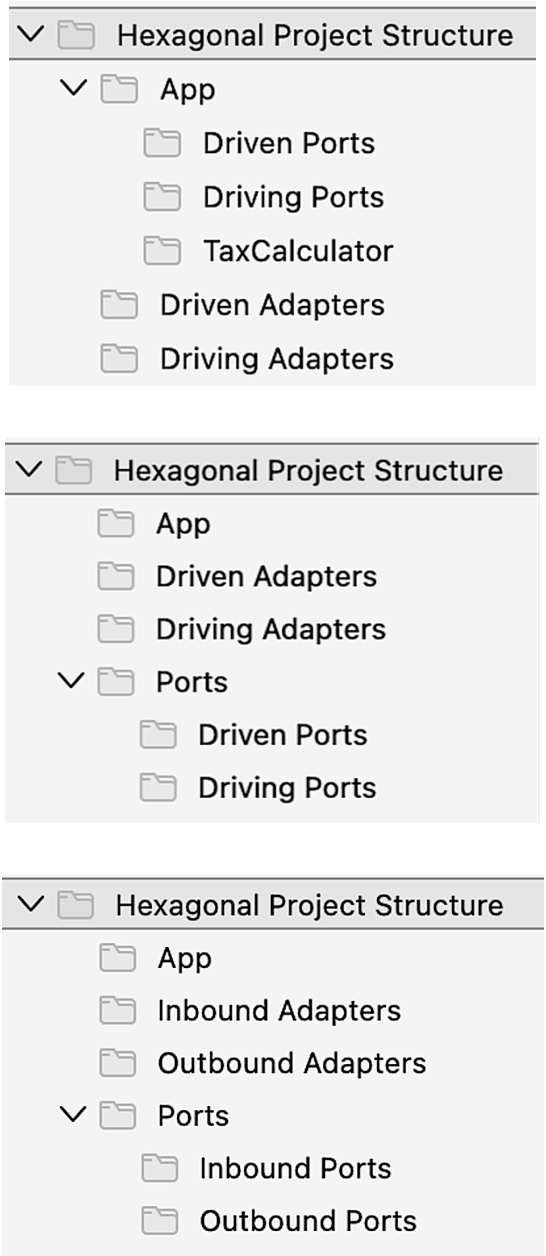


Figure 4.6. Possible folder/project structures.

5.5. Layered, onion, clean, hexagonal: what is the difference?

The Ports & Adapters architecture differs from layered, onion and clean architectures in two ways:

- Ports & Adapters has only two layers: the inside (the app), and the outside (everything else).
- Ports & Adapters requires that you organize the external actors so they connect to specific ports.

But let's look at conventional layered architectures first. In a layered architecture, you separate code by concerns and arrange them from "higher" and "lower," such that higher-level items call or have a dependency upon lower-level ones. More abstract concerns like policy objects are placed higher in the architecture, while hardware and drivers sit on the bottom. The policy items have dependency on the drivers and hardware.

Ports & Adapters, onion, and clean architectures all put the application and domain *below* the UI and infrastructure, as Figure 5.2 illustrates. This makes them appear upside down compared to traditional layered architecture pictures.

The inside of the app with the policy items is on the bottom. Everything else is *above* it, pointing downward. That is because the app can't have a compile-time dependency on anything else.

Inside the upper layer, the "outside", you may have any number of layers of your own choosing. Those decisions are outside the Ports & Adapters architecture and are your personal choices.

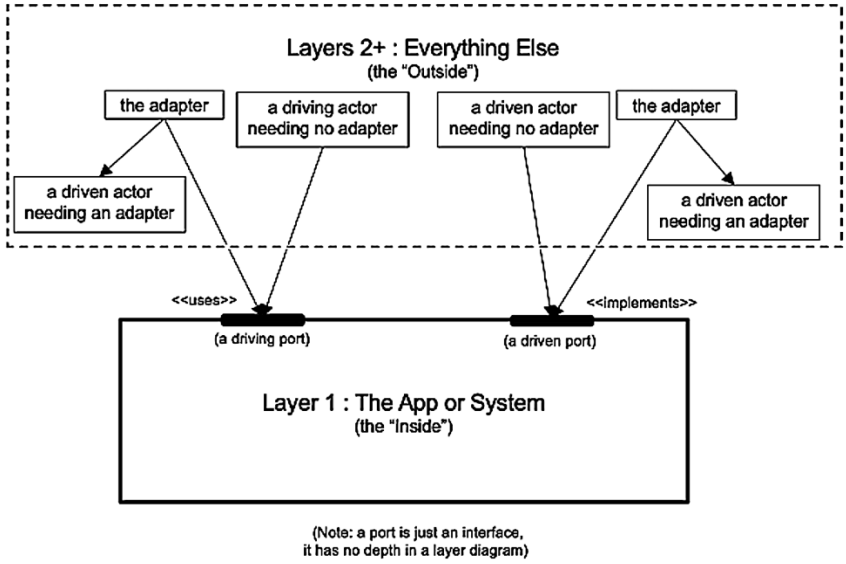


Figure 5.2. Ports & Adapters only specifies two layers: inside and outside.

Figures 5.3 and 5.4 show why the Ports & Adapters architecture looks strange when you are used to a layered architecture.

Figure 5.3 shows an invoicing system with a GUI and a database. On the left is the usual 3-layer architecture with dependencies pointing downward. The execution calls also go down.

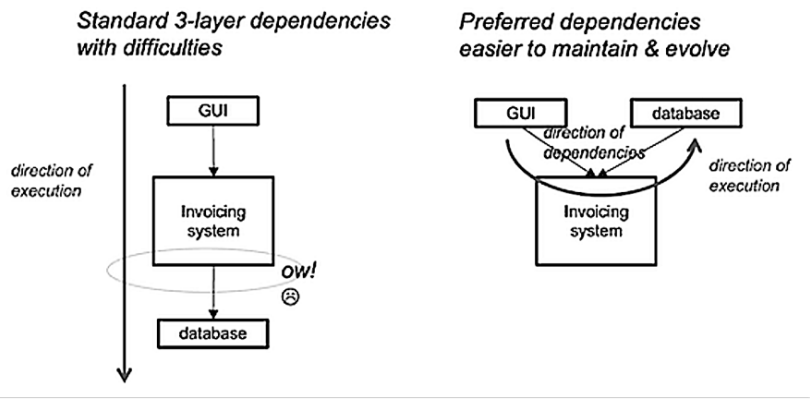


Figure 5.3. Order of dependencies and execution in 3-layer versus Ports & Adapters architectures.

On the right is the Ports & Adapters architecture. As with Figure 5.2, the invoicing system is on the bottom, with both the GUI and the database (or its adapter) having a compile-time dependency on the invoicing system.

What is surprising is that the execution sequence goes in the opposite direction of the dependencies on the driven side. The invoicing system still sends calls to the database, but the database (or its adapter) has the compile-time dependency on the invoicing system. This is different to a layered architecture.

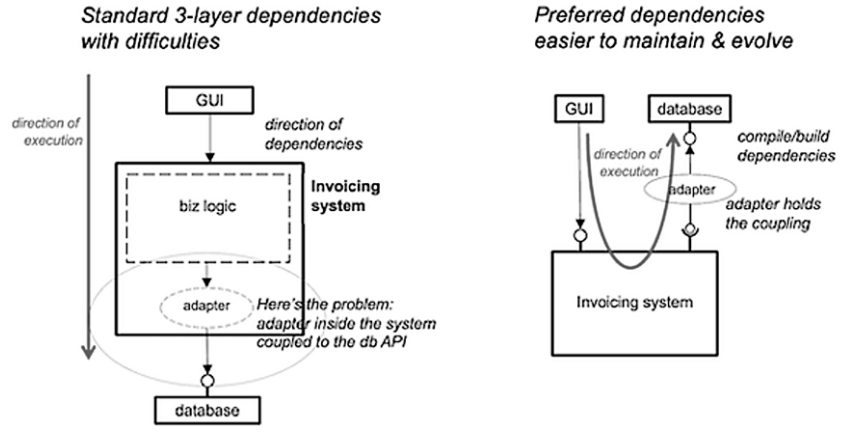


Figure 5.4. Moving the adapter outside and making it dependent on both the system and the database.

Figure 5.4 shows the adapters. The database being purchased has its own published interface, which doesn't match the domain interface of the system under design. An adapter is needed. Usually, that adapter is considered part of the system being designed. Both the compile-time dependencies and the execution flow go from the business logic to the adapter, to the database. This is shown on the left side.

The right side shows the dependencies and execution in the Ports & Adapters architecture. Note that the adapter is outside the system.

- The system publishes its driven port specification (the hook going upward in the drawing);
- the adapter has a compile-time dependency on and implements that interface (the ball fits into the hook);
- the adapter also has a compile-time dependency on and uses the database defined interface.

Note that neither the invoicing system nor the database depend on each other – they are independent. The adapter depends on both of them. The execution flows from the invoicing system to the adapter to the database (and back again).

James Grenning's Embedded IoT

In a parallel evolution, James Grenning (another author of the Agile Manifesto) developed the exact same architecture as Ports & Adapters for systems involving hardware. We found our two designs identical, just using different words. Notable to me was his referring to the driven adapters as the "Service Abstraction Layer," which seems just right.

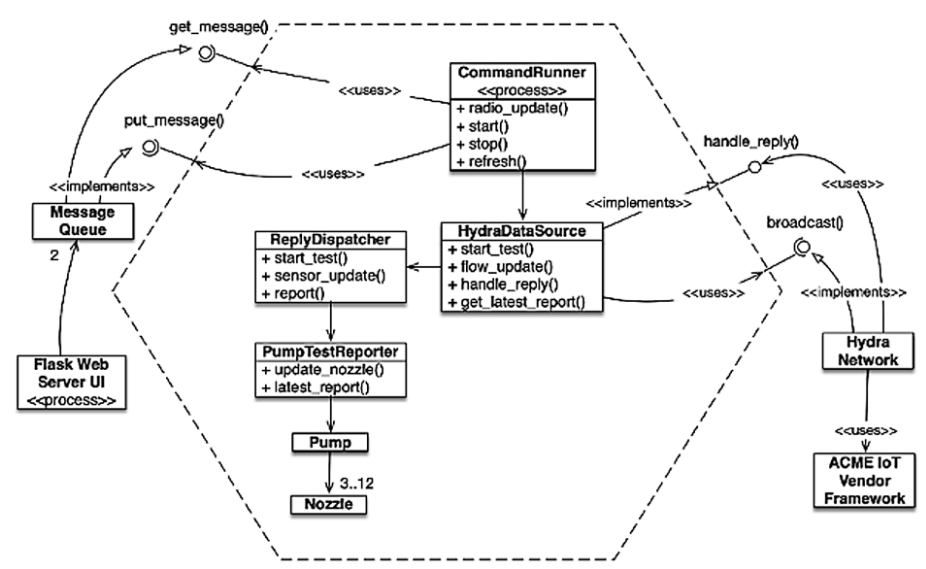


Figure 5.5. Grenning's IoT system.

His sample diagram confused me for a bit, because the 'get' to the message queue is a driven port! That worried me, until he told me that the message queue is polled: the app sends a 'get' request every second. So the ports are correct.

See his full writeup, with code in Python, in Chapter 8 of the forthcoming *Clean Code: A Handbook of Agile Software Craftsmanship, 2nd Ed* (2025).

Onion and Clean

Onion and Clean have the same dependency structure as Ports & Adapters. The two differences are that they don't require the specification of ports, and they do call for additional layers that Ports & Adapters doesn't. Figures 5.5 and 5.6 show these layers.

Once you have implemented Ports & Adapters, you are welcome to add the layers of clean and onion – or not. Those decisions are outside Ports & Adapters. Your choice.

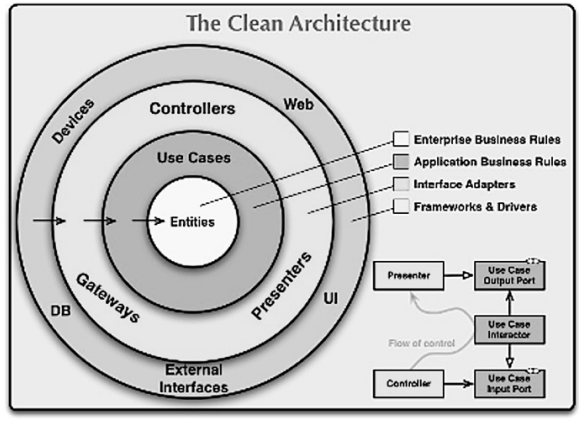


Figure 5.6. Clean architecture
https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

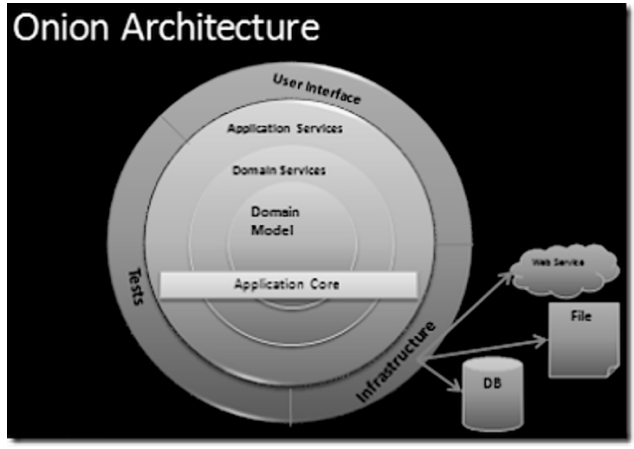


Figure 5.7. Onion architecture
https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/

If you're feeling bombarded by drawings right now, don't worry. Recall the wisdom of David Adamo Jr: The code is simpler than the drawings.

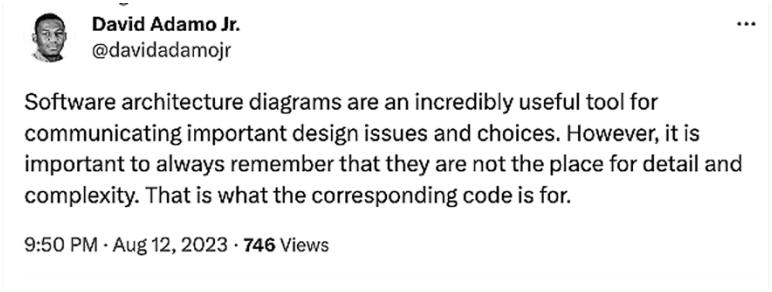


Figure 5.8. Architecture drawings are not code: https://twitter.com/davidadamojr/status/1690541235918753792

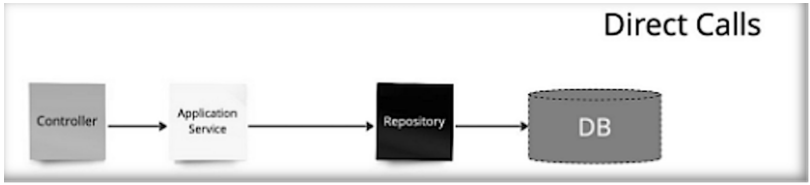
Remember, in Ports & Adapters you are free to organize the inside of the app in any way you like, and the things outside the app in any way you like.

Just put ports in place. Oh, and write those tests.

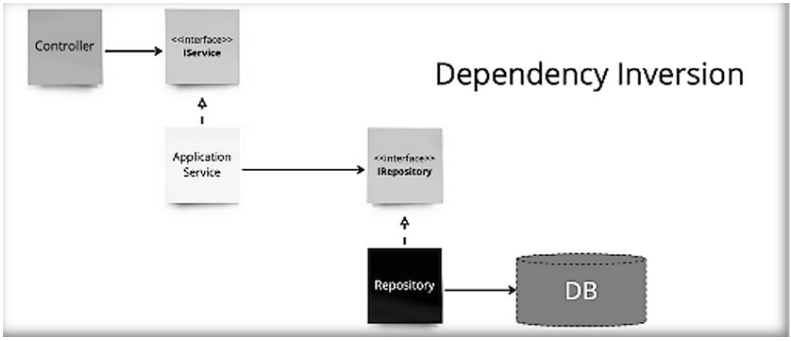
Moving from Layered to Ports & Adapters.

Oliver Zihler published a wonderful article on Substack [https://codeartify.substack.com/p/from-layered-to-hexagonal-architecture] "From Layered to Hexagonal Architecture in 2 steps", which describes it clearly. Here are his figures. Read the article for his explanation if it is not evident how to interpret them.

Step 0: Your starting point:



Step 1: Invert dependencies:



Step 2: Make sure the ports are defined inside the app. Done.

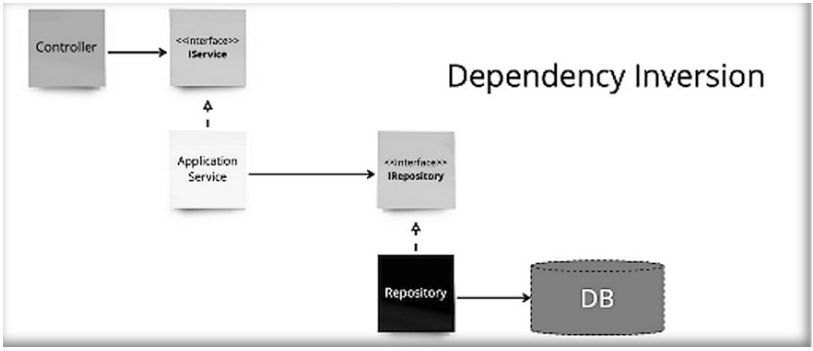


Figure 5.9. From Layered to Hexagonal Architecture in 2 steps
(Figures copyright Oliver Zihler)

6.1. The longer history

One of the last contributions Juan made to this book was to go back and clean up the timeline. I remain grateful to him for his attention to detail and correctness.

To be clear about one point about this history: I am not a systems programmer, I have always been an application programmer. But I grew up with Smalltalk's Model-View-Controller on the driving side, being able to swap drivers easily. I simply assumed and expected that I should be able to do the same on the driven side. I kept asking for this capability of the system architects and being told it wasn't possible. It was out of defense that I started asking myself, How should things be done so that it would be possible? I called it *shunt* at first, and then *loopback* (mocks weren't invented then), came up with the idea of how to do it, and finally in 2004-2005 was in a situation to write some code that did it.

In other words, I created this architecture so that I, as an application programmer, could have those safety /swapping features I needed to develop the application.

1988: Smalltalk and C

> Alistair unknowingly implemented Model-View-Controller in his Smalltalk prototype, but his C programmer didn't. When the need arose to change the source of inputs, that program had to be torn apart and rewritten.

At IBM Research in Switzerland, I had just learned Smalltalk for a new project, with a pre-doctoral student on my team who would implement what my Smalltalk prototype did into a properly fast diagram editor in C.

The Smalltalk tutorial had me code up a "talking parrot", to get us used to state machines. As it turned out, not to my knowledge at the time, that example used the Model-View-Controller architecture. When I made my first real program, I simply copied the talking parrot example and changed it to fit my needs. (There is a separate lesson in here about how to learn a new language, but we can leave that out for now.) As a result, I had the MVC structure in my code without knowing it.

Fin

I want to thank Christopher Hayes-Kossmann for copy-editing the preview edition. Hemal Varambhia make a spectacular gift proofing this updated edition with his eagle eyes. Hemal, Simone Giusso, Rob Jarratt, Ricardo Guzmán Velasco, Nicky Ramone, Chris Carroll, and Eleonora Ciceri did some really detailed reading and provided great feedback.

It is strange that I have been describing this pattern for 30 years. It is a really simple architecture to implement, and yet we are still discovering relationships to other people's work and other patterns. As someone commented, the code is much simpler than the descriptions of the pattern. It is for that reason that we show the code in the first and last sections of the book.

And finally, for me, Alistair, I feel like I lost half of my Hexagonal brain with the passing of Juan Manuel Garrido de Paz. He was the person I always wrote to when someone posed a new question or I had a doubt about a piece of code. His knowledge was encyclopedic, his method analytical. We debated incessantly until we agreed on an answer that satisfied us both.

Here is his favorite image he sent me during those discussions:

R.I.P. Juan Manuel Garrido de Paz. Thank you.

About the Authors

Dr. Alistair Cockburn (pronounced CO-BURN), known for his wild hair photo on LinkedIn, was named as one of the “42 Greatest Software Professionals of All Times" in 2020, as a world expert on object-oriented development, software architecture, project management, use cases and agile development. Since 2015 he has been working on expanding agile to cover every kind of initiative, including social impact project, governments, and families. For his latest work, see https://alistaircockburn.com/.

Juan Manuel Garrido de Paz (August 3, 1970 - April 18, 2024) won his Bachelor in Software Engineering at the Polytechnic University of Madrid. He became the world’s other leading authority on the Ports & Adapters pattern by probing and interacting with Dr. Alistair Cockburn over years. A senior developer for the government of Andalucía, his two passions were Hexagonal Architecture and Recreativo de Huelva Football Club. Sadly, Juan passed away just weeks before this book went to print. This book is dedicated to him and his life.