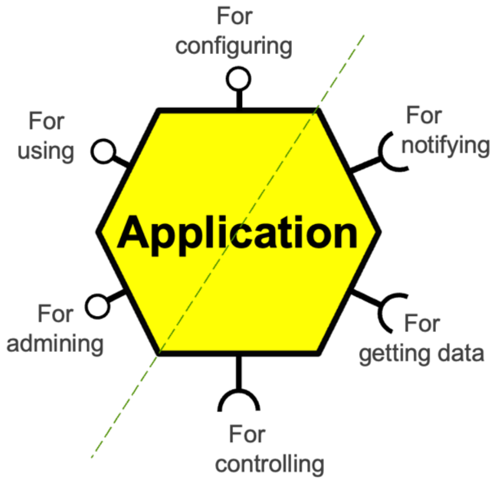
The code from the Preview Edition of

Hexagonal Architecture Explained

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



Alistair Cockburn

Juan Manuel Garrido de Paz

ISBN 978-1-7375197-8-2
Humans and Technology Press
32 W 200 S #504
Salt Lake City, UT 84101
V0.9a 20240507-1836 for paper books

Acknowledgements

From Alistair

I am immensely grateful to Juan Manuel Garrido de Paz, without whom this book could never have been written. Of all the people I have conversed with, Juan had the sharpest, deepest, most accurate understanding of the pattern. He saw its relationship to UML components and the *required* interface years before I did.

He was relentless in his quest to understand and describe the pattern. He provided code for me to study and include. We argued incessantly, but only ever in pursuit of the truth. Once we found it, we were once again in complete agreement.

Juan was also a relentless fan of FC Huelva:

Juan at Huelva, 2024

1.1. Copy this code

All the pages in this book only serve to help you replicate this code snippet in your larger system. Here is some Java code to show you 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 Ruby code shows how dynamic languages create the same system with no interface definitions:

```
class TaxCalculator
   def initialize( tax_rate_repository )
      @tax_rate_repository = tax_rate_repository
  end

    def tax_on( amount )
      amount * @tax_rate_repository.tax_rate( amount )
   end
end

class FixedTaxRateRepository
   def tax_rate( amount )
     0.15
   end
end



tax_rate_repository = FixedTaxRateRepository.new
my_calculator = TaxCalculator.new( tax_rate_repository )
puts my_calculator.tax_rate( 100 )
```

3.1. The simplest example: the tax calculator

We present this example in four parts.

1. First, we show just the tax_on() function in use. We use the simplest rate repository, which returns a fixed rate. This is the first step in creating architecture with as little code as possible.

   From this point forward, the drivers can change and grow, the port interfaces can change and grow, the business logic contained within the app can change and grow, and the repositories can change and grow, without any need to change the fundamental architecture.

   We chose to use Java for this first part, because it helps show the ports being defined and used.
2. Second, we provide the same code in Ruby. This illustrates how the code looks when it's not necessary to declare the ports or interfaces.
3. The third shows using setter injection instead of dependency injection. That is to say, the driven actor is not set in the constructor, but in a setter method. This allows the driven actor to be changed at any time.
4. The fourth stage shows a different configurator, one that uses dependency lookup instead of dependency injection for the configurator.

   The tax calculator is augmented to hook to different rate repositories for different countries. The configurator is now a "rate repository broker" that says what rate repository to use for any country requested. This permits different tax tables for different countries, or for adapters to connect to the official tax authority for different countries.

   Of course, we need a second port for this broker. It follows that we then need a test double for it, as well as a configurator for the broker.

The simplest tax calculator (Java)

In this simple example of the tax calculator, the configurator main() passes the receiver to the sender at object creation time. It creates the FixedTaxRateRepository, the receiver, and sends it to the TaxCalculator as part of its constructor.

For early development, I use an in-code tax rate repository with just one fixed tax rate. We chose to begin with Java, to show the interfaces.

```
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 ) );
   }
```

Notes:

“ForCalculatingTaxes” is the driving port, with, just for the moment, only one function offered: “taxOn(amount)”.

“ForGettingTaxRates” is the driven port. It require every repository to support the function “taxRate(amount)”.

The TaxCalculator implements the driving port, and uses the driven port.

The FixedRateRepository implements the driven port, as every rate repository would do.

Main acts as both the configurator and, in this tiny case, also the driving actor. The first two lines act as the configurator, creating the FixedRateRepository and feeding it to the TaxCaculator at creation time. The last line is using the driving port as any user might.

The simplest tax calculator (Ruby)

This code accomplishes the same work as the prior Java example, but in Ruby. The ports and interfaces don’t have to be declared in Ruby, which makes it difficult to see where they are.

```
class TaxCalculator

   def initialize( tax_rate_repository )
      @tax_rate_repository = tax_rate_repository
  end

    def tax_on( amount )
      amount * @tax_rate_repository.tax_rate( amount )
   end
end

class FixedTaxRateRepository
   def tax_rate( amount )
     0.15
   end
end

tax_rate_repository = FixedTaxRateRepository.new
my_calculator = TaxCalculator.new( tax_rate_repository )
puts my_calculator.tax_rate( 100 )
```

Using a setter instead of constructor argument (Java)

In this example, we build upon previous examples to demonstrate that it’s not required to set the driven actors in the constructor arguments (constructor injection). In fact, it’s just as valid to use a setter method (setter injection). This allows the driven actor to be changed at any time.

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

class TaxCalculator {
   private ForGettingTaxRates taxRateRepository;
   public void setTaxRateRepository(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();
      myCalculator.setTaxRateRepository( taxRateRepository );
      System.out.println(myCalculator.taxOn( 2000 ));
   }
}
```

Using a broker instead of a fixed rate repository (Ruby)

The fourth adjustment shows the use of *dependency lookup* instead of *configurable receiver* to configure the driven actor.

We introduce a rate repository broker, which will tell the calculator what rate repository to use for any given country. This allows different rate repositories to handle the different tax rate tables in various countries, or to have an adapter that connects directly to a country's official tax authority.

To do this, we introduce a second port, "ForGettingCountryBasedTaxRateRepository". This port requires one function, RepositoryForCountry(country).

Now that we have a second port, we need to be able to test it. There needs to be a test double as well as a production rate repository broker. This in turn means we need a configurator for the broker port. Here we use constructor injection to set the broker to use at the time the tax calculator is created.

We chose to show this code in Ruby because it's easier to see the intention. We'll then use Java to show the second port beng declared.

```
class RateRepositoryBroker
  def initialize
    @tax_rate_repository_FR = TaxRateRepositoryFR.new
    @tax_rate_repository_US = TaxRateRepositoryUS.new
  end
  def repository_for( country )
    if country == "US"      return @tax_rate_repository_US
    elsif country == "FR"  return @tax_rate_repository_FR
    else  return nil
    end
  end
end
```

```
class TaxCalculator
  def initialize( repository_broker )
    @my_rate_repository_broker = repository_broker
  end

  def tax_on( country, amount )
    tax_rate_repository = @my_rate_repository_broker.repository_for( 
country )
    amount * tax_rate_repository.tax_rate( amount )
  end
end

class TaxRateRepositoryFR
  def tax_rate( amount )
    0.30
  end
end

class TaxRateRepositoryUS
  def tax_rate( amount )
    0.15
  end
end

my_tax_rate_broker = RateRepositoryBroker.new
my_calculator = TaxCalculator.new( my_tax_rate_broker )
puts my_calculator.tax_rate( "FR", 2000 )
puts my_calculator.tax_rate( "US", 2000 )puts my_calculator.tax_rate( 100 
)
```

3.2. Another simple example, the web-hexagon

In the 2010s, Alistair started building a custom content management system in Ruby. To do this, he needed to install and connect to a web service as the driving actor. He began with the simplest app possible: just returning the input multiplied by a number from a repository.

Output = Input * database[Input].

Ruby allows you to return two values, so Alistair had it return both the tax rate and the result of the multiplication.

The first test uses a mock repository. This is enough to establish the Ports & Adapters architecture. Developing the architecture further with different external technologies, he added Rack for the web on the input side and a flat file for the repository.

After growing the app some more, he simplified it back down to the smallest serviceable example, to show how simple the code is. See https://github.com/totheralistair/SmallerWebHexagon

Just the app

The app gets configured with the secondary actor through its constructor, setting the repository to use. Because this is coded in Ruby, there are no declarations for the ports.

```
class SmallerWebHexagon

  def initialize rater
    @rater = rater     # the database port needs configuring
  end

  def rate_and_result  value
    rate = @rater.rate(value)
    result = value * rate
    return rate, result
  end
end
```

The first repository: an in-memory repository:

To make the tests a bit interesting, we use two tax rates.

```
class InCodeRater

def rate value
   case
     when value <= 100
       1.01
     when value > 100
       1.5
   end
 end

end
```

The first test: test-harness to app to in-memory rater

In this first test, you can see the InCodeRater being passed in with the constructor.

```
class TestRequests < Test::Unit::TestCase
 attr_accessor :app

 def test_it_works_with_in_code_rater
   p __method__

   @app = SmallerWebHexagon.new(InCodeRater.new)

   value_should_produce_rate 100, 1.01
   value_should_produce_rate 200, 1.5
 end
```

At this point the Ports & Adapters architecture has already been completed, there is a primary port for computing taxes, and a secondary one for getting tax rates. Note again that since Ruby does not require interfaces to be declared, the ports themselves are not explicitly visible.

Adding a second type of repository

To test that the architecture functions as intended, we create a file with the tax rates:

```
0   1.0
100 2.0
```

We add a file reader as the adapter:

```
class FileRater

  def initialize fn
    @rates = []
    File.open(fn) do |f|
      f.each_line do |line|
        @rates << line.split.map(&:to_f)
      end
    end
  end

  def rate value # ugly code but I only need to know it works
    case
      when value >= @rates[0][0] && value < @rates[1][0]
        rate = @rates[0][1]
      when value >= @rates[0][0]
        rate = @rates[1][1]
    end
  end

end
```

Finally, we add a test to ensure all this works. Personally, I like to use a different number in the file rater, so I can tell which rater is being activated.

```
 def test_it_works_with_file_rater
   p __method__

   @app = SmallerWebHexagon.new(FileRater.new('file_rater.txt'))

   value_should_produce_rate 10, 1.00
   value_should_produce_rate 100, 2.0
 end
```

Add a web interface at the front

Finally, we add the interface to Rack for web input.

```
class RackHttpAdapter

 def initialize(hex_app, views_folder)
   @app = hex_app
   @views_folder = views_folder
 end

 def call(env) # hooks into the Rack Request chain
   request = Rack::Request.new(env)
   value =  path_as_number(request)

   rate, result = @app.rate_and_result  value

   out = {
      out_action:   'result_view',
      value:  value,
      rate:   rate,
      result: result
   }

   template_path =
Pathname.new(@views_folder).join(out[:out_action]).sub_ext('.erb')
```

```
  page = html_from_template_file(template_path , binding)

  response = Rack::Response.new
  response.write(page)
  response.finish
 end
```

To test this, use the in-memory rater from the first test, then built the app to run live from a browser. This file is "config.ru".

```
# run the Smaller Web Hexagon from a browser

require './src/smaller_web_hexagon'
require './src/rack_http_adapter'
require './src/raters'

hex = SmallerWebHexagon.new(InCodeRater.new)
app = RackHttpAdapter.new(hex,"./src/views/")


run app
```

At this point we can drive the app from the tests or a browser, and get the reates from either the in-memory rater or the file.

The final test suite:

Here is the full set of tests:

```
require_relative '../src/smaller_web_hexagon'
require_relative '../src/rack_http_adapter'
require_relative '../src/raters'
require 'rack/test'
require 'rspec/expectations'
require 'test/unit'

# The first 2 tests check the primary adapter swaps, using direct API access
for the left
# The last test checks the secondary adapter swap, using Rack input.
# The config.ru file runs the real server stuff, for the final usage test.
# note about the tests, I made all the raters give different answers,
# so that I can see if they are hooked up wrong


class TestRequests < Test::Unit::TestCase
  attr_accessor :app

  def test_it_works_with_in_code_rater
    p __method__

    @app = SmallerWebHexagon.new(InCodeRater.new)

    value_should_produce_rate 100, 1.01
    value_should_produce_rate 200, 1.5
  end


  def test_it_works_with_file_rater
    p __method__

    @app = SmallerWebHexagon.new(FileRater.new('file_rater.txt'))

    value_should_produce_rate 10, 1.00
    value_should_produce_rate 100, 2.0
  end


  def test_runs_via_rack_adapter
    p __method__

    views_folder = '../src/views/'
    hex = SmallerWebHexagon.new (InCodeRater.new)
    app = RackHttpAdapter.new(hex, views_folder)

    request = Rack::MockRequest.new(app)
    response = request.request('GET', '/100') # sends the req through the
Rack call(env) chain
```

```
    out = {             # expected values
      out_action:   'result_view',
      value:  100,
      rate:   1.01,
      result: (100)*(1.01)
    }
    response.body.should == html_from_template_file(views_folder +
'result_view.erb' , binding)
  end


  def value_should_produce_rate value, exp_rate
    rate, result = @app.rate_and_result value

    rate.should == exp_rate
    result.should == value * exp_rate
  end

end
```

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.

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

"Looking at the screen of my laptop, I realized that it was full of code that didn't let me understand what it did regarding business logic. From that moment I began to search until I discovered the architecture that decouples the business logic from the frameworks: Hexagonal Architecture, more correctly called Ports & Adapters. From that moment until now, I haven't stopped reading and learning about this pattern."

Used by giants like Netflix and Amazon, the Hexagonal or Ports & Adapters architecture simplifies testing, protects against business logic leakage, supports changing technologies in long-running system, and lets you apply Domain Driven Design.

In this definitive book on the subject, pattern author Dr. Alistair Cockburn and Juan Manuel Garrido de Paz lay bare all of the intricacies of the pattern, providing sample code and answering your many frequently asked questions.

$40.00
ISBN 978-1-7375197-8-2
54000>
9 781737 519782

Hexagonal Architecture Explained

Alistair Cockburn & Juan Manuel Garrido de Paz