Ports & Adapters Architecture

hgraca Architecture, Development, Series, The Software Architecture Chronicles September 14, 2017September 15, 2017 7 Minutes

This post is part of The Software Architecture Chronicles (https://herbertograca.com/2017/07/03/the-software-architecture-chronicles/), a series of posts about Software Architecture (https://herbertograca.com/category/development/series/software-architecture/). In them, I write about what I've learned on Software Architecture, how I think of it, and how I use that knowledge. The contents of this post might make more sense if you read the previous posts in this series.

The Ports & Adapters Architecture (http://alistair.cockburn.us/Hexagonal+architecture) (aka Hexagonal Architecture (http://alistair.cockburn.us/Hexagonal+architecture)) was thought of by Alistair Cockburn and written down on his blog in 2005. This is how he defines its goal in one sentence:

> *Allow an application to equally be driven by users, programs, automated test or batch scripts, and to be developed and tested in isolation from its eventual run-time devices and databases.*
>
> *Alistair Cockburn 2005, Ports and Adapters (http://alistair.cockburn.us/Hexagonal+architecture)*

I've seen some articles talking about the Ports & Adapters Architecture that said a lot about layers. However, I haven't read anything about layers in the original Alistair Cockburn post.

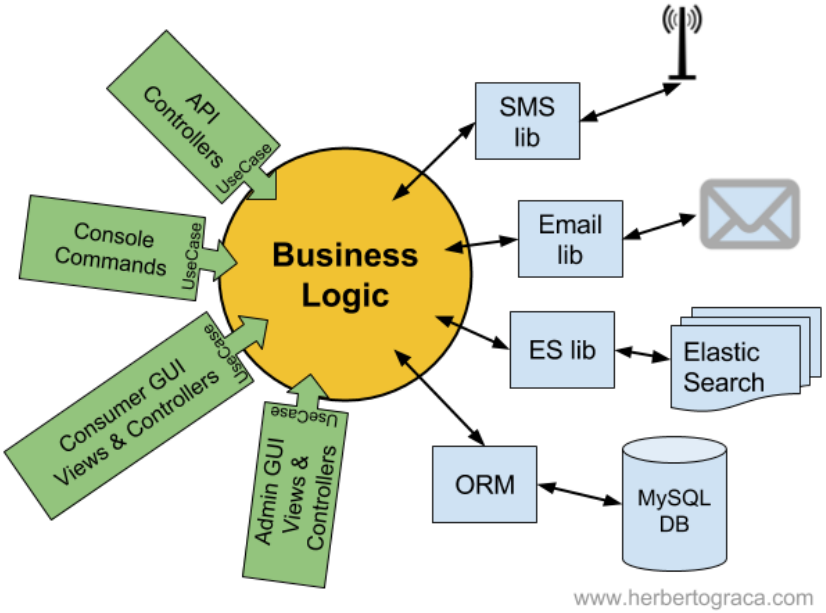
The idea is to think about our application as the central artefact of a system, where all input and output reaches/leaves the application through a port that isolates the application from external tools, technologies and delivery mechanisms. The application should have no knowledge of who/what is sending input or receiving its output. This is intended to provide some protection against the evolution of technology and business requirements, which can make products obsolete shortly after they are developed, because of technology/vendor lock-down.

In this post, I'm going to dive into the following subjects:

The problems of the traditional approach

The traditional approach will likely bring us problems both on the front-end side and on the backend side.

On the front-end side we end up having leakage of business logic into the UI (ie. when we put use case logic in a controller or view, making it unreusable in other UI screens) or even leakage of the UI into the business logic (ie. when we create methods in our entities because of some logic we need in a template).

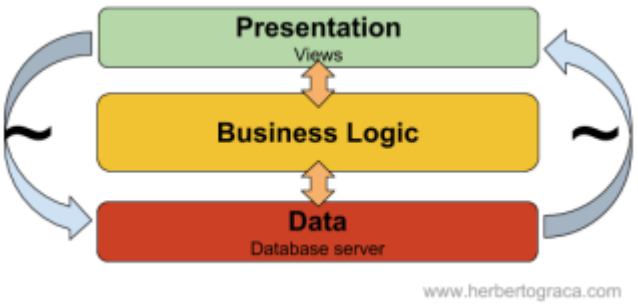


On the back-end side, we can have leakage of the external libraries and technologies into the business logic because we might end up referencing them directly by type hinting, subclassing, or even instantiating the libraries classes inside our business logic.

Evolving from the Layered Architecture

By 2005, thanks to EBI (https://herbertograca.com/2017/08/24/ebi-architecture/) and DDD (https://herbertograca.com/2017/09/07/domain-driven-design/), we knew already that **what is really relevant in the system are the inner layers**. Those layers are the ones where all the business logic lives (or should live), they are the real differential to our competitors. That is the real "application".

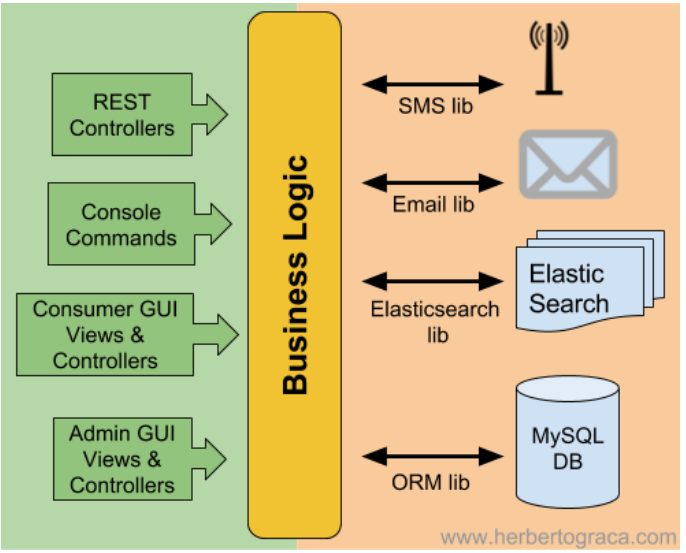
But at some point, Alistair Cockburn realised that **the top and bottom layers, on the other hand, were simply entry/exit points to/from the application**. Although they are actually different, they have nevertheless very similar objectives and there is symmetry in the design. Moreover, if we want to isolate our application inner layers, we can do it using those entry/exit points, in a similar fashion.

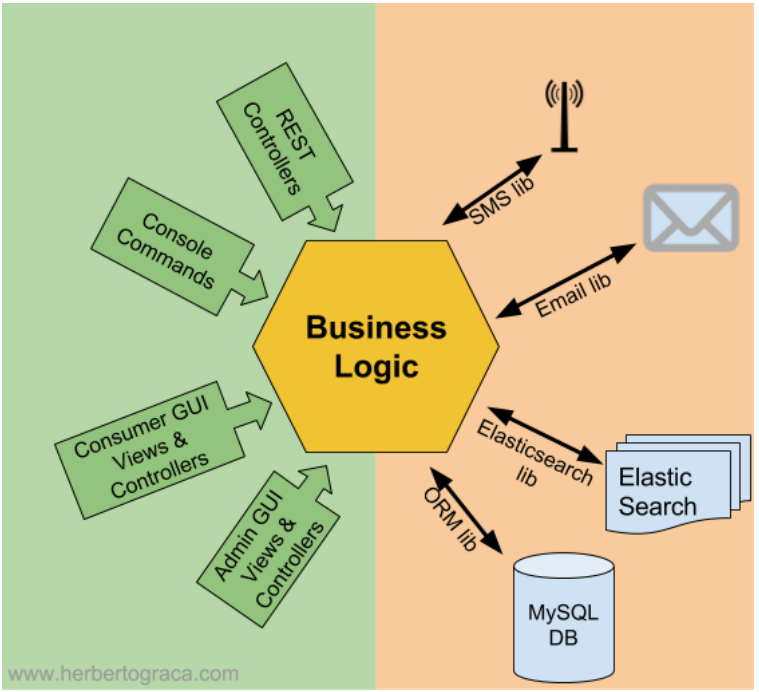


To escape the typical layering diagram, we will represent this two sides of the system as left and right, instead of top and bottom.

Although we can identify two symmetrical sides of the application, **each side can have several entry/exit points**. For example, an API and an UI are two different entry/exit points on our left side of the application, while an ORM and a search engine are two different entry/exit points on the right

side of our application. To represent that our application has several entry/exit points, we will draw our application diagram with several sides. **The diagram could have been any polygon with several sides**, but the choice turned out to be a hexagon. Hence the name "Hexagonal Architecture".





The Ports & Adapters Architecture (http://alistair.cockburn.us/Hexagonal+architecture) solves the problems identified earlier by using an abstraction layer, implemented as a port and an adapter.

What is a Port?

A port is a consumer agnostic entry and exit point to/from the application. In many languages, it will be an interface. For example, it can be an interface used to perform searches in a search engine. In our application, we will use this interface as an entry and/or exit point with no knowledge of the concrete implementation that will actually be injected where the interface is defined as a type hint.

What is an Adapter?

An adapter is a class that transforms (adapts) an interface into another.

For example, an adapter implements an interface A and gets injected an interface B. When the adapter is instantiated it gets injected in its constructor an object that implements interface B. This adapter is then injected wherever interface A is needed and receives method requests that it transforms and proxies to the inner object that implements interface B.

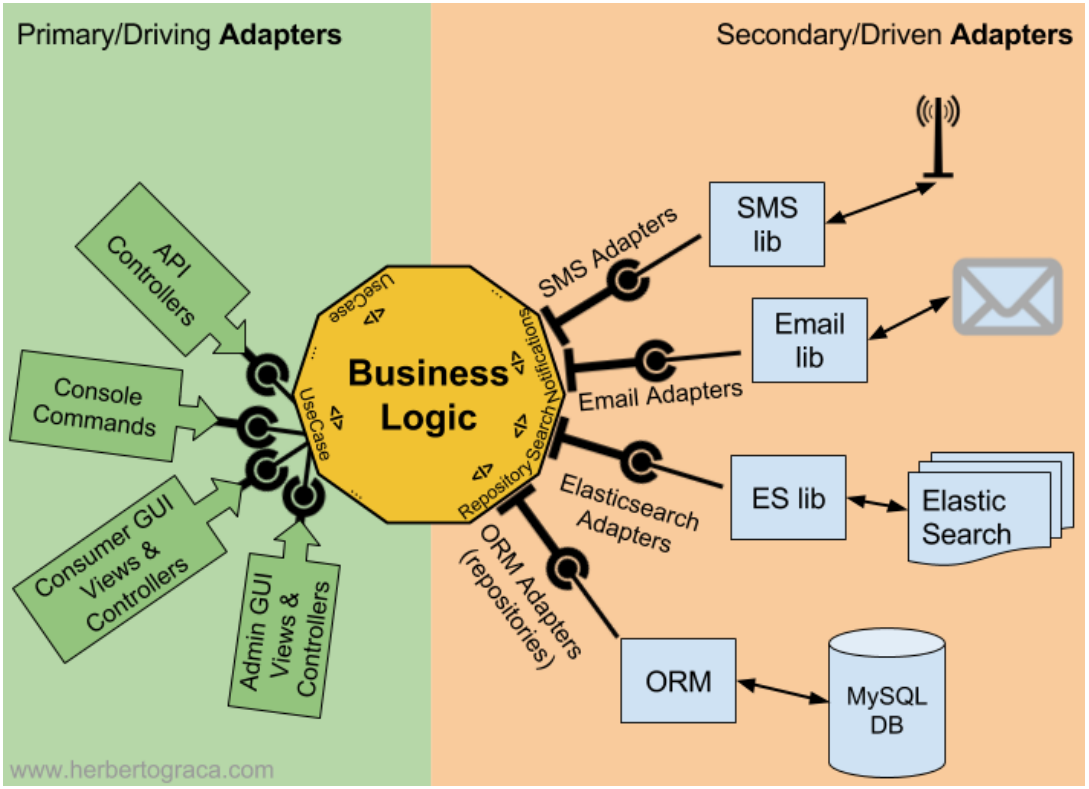
If I managed to confuse you, no worries, I give a more concrete example further below. 🙂

Two different types of adapters

The adapters on the left side, representing the UI, are called the **Primary** or **Driving Adapters** because they are the ones to start some action on the application, while the adapters on the right side, representing the connections to the backend tools, are called the **Secondary** or **Driven Adapters** because they always react to an action of a primary adapter.

There is also a difference on how the ports/adapters are used:

- On the **left side**, the adapter depends on the port and gets injected a concrete implementation of the port, which contains the use case. On this side, **both the port and its concrete implementation (the use case) belong inside the application**;
- On the **right side**, the adapter **is** the concrete implementation of the port and is injected in our business logic although our business logic only knows about the interface. On this side, **the port belongs inside the application, but its concrete implementation belongs outside** and it wraps around some external tool.



What are the benefits?

Using this port/adapter design, with our application in the centre of the system, allows us to keep the application isolated from the implementation details like ephemeral technologies, tools and delivery mechanisms, making it easier and faster to test and to create a reusable proof of concept.

Implementation and technology isolation

Context

We have an application that uses SOLR as a search engine and we use an open source library to connect to it and perform searches.

Traditional approach

Using a traditional approach, we will use that library classes directly in our code base, as type hints, instances and/or superclasses of our implementations.

Ports and adapters approach

Using ports and adapters we will create an interface, let's call it UserSearchInterface, which we will use in our code when needed as a type hint. We will also create the adapter for SOLR, which will implement that interface, let's name it UserSearchSolrAdapter. This implementation is a wrapper for the SOLR library, so it gets the library injected and uses it to implement the methods specified in the interface.

Problem

At some point, we want to switch from SOLR to Elasticsearch. Moreover, for the same search, sometimes we want to use SOLR and other times we want to use Elasticsearch, with that decision being made at runtime.

If we used the traditional approach, we will have to search and replace the usage of the SOLR library for the Elasticsearch library. However, that is not a simple search and replace: the libraries have different ways of being used, different methods with different inputs and outputs, so replacing the libraries will not be a trivial task. And using one library instead of another one, at runtime, won't even be possible.

However, if we used Ports & Adapters, we just need to create a new adapter, let's name it UserSearchElasticsearchAdapter, and inject it instead of the SOLR adapter, maybe just by changing a config in the DIC. To inject a different implementation at run time, we can use a Factory to decide which adapter to inject.

Delivery mechanisms isolation

In a similar fashion to the previous example, let's say we have an application that needs a web GUI, a CLI and a web API. We also have some functionality that we want to make available in all three UIs, let's call that functionality *UserProfileUpdate*.

Using Ports & Adapters, we would implement this functionality in an application service method and think of it as a use case. This service would implement an interface specifying the methods, inputs and outputs.

Each UI version would then have a controller (or console command) that would use that interface to trigger the logic desired and would be injected with the concrete implementation of the service. Here, the Adapter is actually the controller (or CLI Command).

We could then completely change the UI knowing that we would not affect the business logic.

Testing

In both the previous examples, testing becomes easier with Ports and Adapters Architecture. In the first examples, we can mock or stub the interface (Port) and test our application without using SOLR nor Elasticsearch.

In the second example, we can test all the UIs in isolation from our application, and our use cases in isolation from the UI by simply giving our service some input and asserting the results.

Conclusion

The way I see it, Ports & Adapters Architecture has only one **goal: isolate the business logic from the delivery mechanisms and tools used by the system**. And it does so by using a common programming language construct: *interfaces*.

On the **UI** side (the driving adapters), we create **adapters that use our application interfaces**, ie. controllers.

On the **infrastructure** side (the driven adapters), we create **adapters that implement our application interfaces**, ie. repositories.

That's all there really is to it!

It is, though, curious to note that this same idea was already published 13 years before (https://herbertograca.com/2017/08/24/ebi-architecture/), although not explicitly emphasising the objective of isolating the tools and delivery mechanisms from the core of the application.

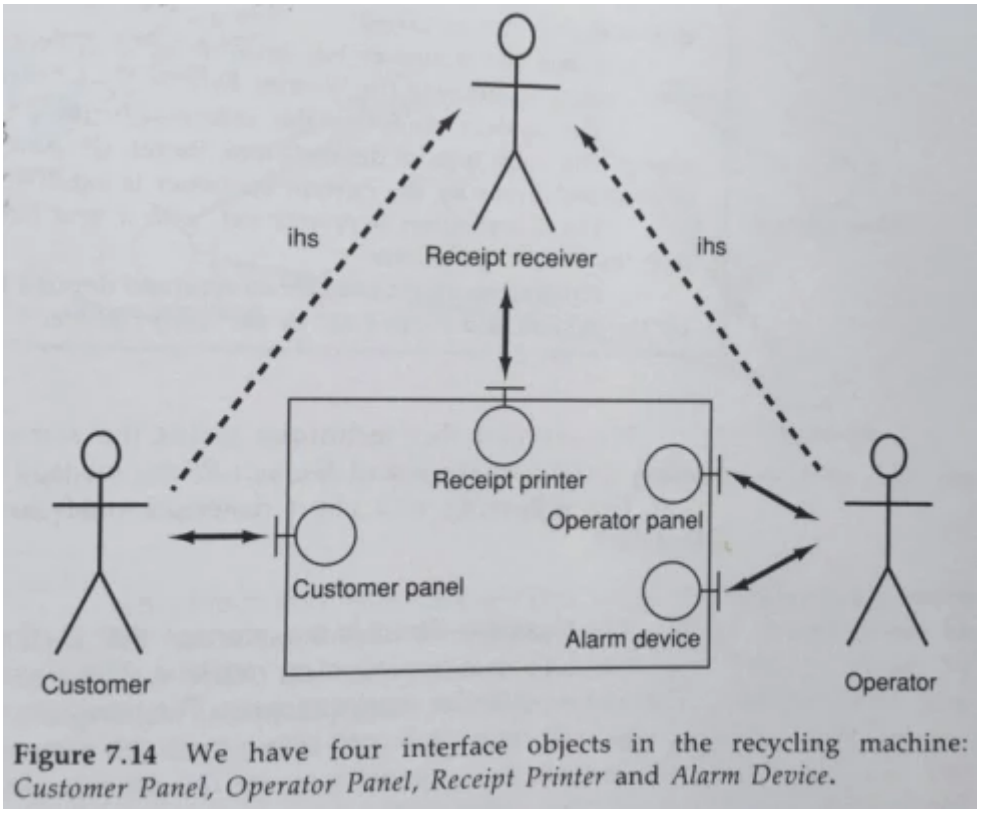


Figure 7.14 We have four interface objects in the recycling machine: *Customer Panel, Operator Panel, Receipt Printer* and *Alarm Device.*

Ivar Jacobson 1992, pp. 171

Any interaction of the system with an actor goes through a Boundary object. As Jacobson describes, an actor can be a human user like a customer or an administrator (operator), but it might also be a non-human "user" like an alarm or a printer, which corresponds to the *Driving Adapters* and *Driven Adapters* of Ports & Adapters Architecture.

Sources

1992 – Ivar Jacobson – Object-Oriented Software Engineering: A use case driven approach (https://www.amazon.com/Object-Oriented-Software-Engineering-Driven-Approach/dp/0201403471)

200? – Alistair Cockburn – Hexagonal Architecture (http://wiki.c2.com/?HexagonalArchitecture)

2005 – Alistair Cockburn – Ports and Adapters (http://alistair.cockburn.us/Hexagonal+architecture)

2012 – Benjamin Eberlei – OOP Business Applications: Entity, Boundary, Interactor (https://beberlei.de/2012/08/13/oop_business_applications_entity_boundary_interactor.html)

2014 – Fideloper – Hexagonal Architecture (http://fideloper.com/hexagonal-architecture)

2014 – Philip Brown – What is Hexagonal Architecture? (http://culttt.com/2014/12/31/hexagonal-architecture/)

2014 – Jan Stenberg – Exploring the Hexagonal Architecture (https://www.infoq.com/news/2014/10/exploring-hexagonal-architecture)

2017 – Grzegorz Ziemoński – Hexagonal Architecture Is Powerful (https://dzone.com/articles/hexagonal-architecture-is-powerful)

2017 – Shamik Mitra – Hello, Hexagonal Architecture (https://dzone.com/articles/hello-hexagonal-architecture-1)

Tagged:
Alistair Cockburn,
Hexagonal Architecture,
Ports & Adapters Architecture,
software architecture,
software developement

Published by hgraca

View all posts by hgraca

42 thoughts on “Ports & Adapters Architecture”

Pingback: where do you put methods relating to multiple instances of your model? - Get Code Solution
Pingback: Архитектура современных корпоративных Node.js-приложений – LvBoard
Pingback: The 4DEVELOPERS Katowice 2019 conference - Yumasoft
Asanka Abeysinghe (@asankama) says:
October 27, 2020 at 21:29
Great post, I’ve published parallel concepts in two industry papers [1] Cell-based Architecture (CBA) and [2] Layered & Segmented Architecture.
[1] https://github.com/wso2/reference-architecture/blob/master/reference-architecture-cell-based.md
[2] https://github.com/wso2/reference-architecture/blob/master/reference-architecture-layered-segmented.md

↳ Reply
Pingback: Un test peut en cacher un autre – Tests d’intégration – P1 | OCTO Talks !
Pingback: Архитектура современных корпоративных Node.js-приложений / Блог компании Яндекс / Хабр
Pingback: Архитектура современных корпоративных Node.js-приложений — monobono.ru
Pingback: Un test peut en cacher un autre – Tests d’intégration | OCTO Talks !
Pingback: Un test peut en cacher un autre – Un peu de théorie | OCTO Talks !
Dominik Mimler says:
February 18, 2019 at 07:46
Hello, I’m using hexagonal architecture for my new project. Now I have a web api component which is a driving adapter, and a knx bus, which should listen for events on the bus, and if one occurs, react on it (start a use case). I’m not sure where to place the abstraction and the concrete

implemantation of the knx bus. I think it's an driving adapter, so it should be on the left side. Maybe a console application whit a while loop, an if an event occurs, it calls an use case.

Additionally I will call read and write on the knx bus in the use case, that means, I have to implement the knx bus in the infrastructure layer also. Is this an problem, because of code duplication?

One way to resolve this problem is to implement the bus in the infrastructure layer, and in the application I have an interface with an event in the use case, which raises when the bus has changed. The console app calls the use case, which listens to the bus.

Which solution do you suggest?

Thank you
Dominik

↳ Reply

hgraca says:
February 18, 2019 at 08:08
Whats a knx bus?

↳ Reply

Dominik Mimler says:
February 21, 2019 at 08:10
A bus for home automation (in office buildings, it's used very often).

↳ Reply

hgraca says:
February 21, 2019 at 08:26
It feels to me that, the API controllers are the driving adapters, the knx bus is infrastructure which can be used by the driving adapters.
If u build a cli, it should be as a driving adapter as well.

Im not sure this answers your question…

↳ Reply

Dominik Mimler says:
February 21, 2019 at 08:49
It's difficult, but I think, the concrete implementation of the knx bus, should be made in infrastructure layer (because a driving adapter, the main loop in the cli uses it). I make an use case, which emits an event when the knx bus sends a signal.

↳ Reply

Jordey says:
March 25, 2019 at 13:44
I read your articles with a lot of pleasure, thank you. I've one question.
What is the difference with the Eric Evans layered architecture combined with Bob Martin's dependency inversion? It seems to be similar. With this approach: use cases go into the application layer, UI is using the application layer handlers (use cases/orchestration) via an interface/abstraction. The domain layer will use the repository via an abstractions as well. Using abstractions via an interfaces will result naturally in an adapter pattern.

↳ Reply

hgraca says:

March 26, 2019 at 12:31
The difference is that layered architecture allows the entities to know/use repositories, while the ports & adapters, onion and clean (uncle bob) architectures are concentric and disallow entities to know/use repositories or anything outside the domain layer.

↳ Reply

Hamdi says:
February 5, 2019 at 15:36
Very good article about Port & Adapters,

I just want to ask a question, let's say that we will use ASP.NET MVC on left side and want to use Port & Adapters for reasons explained in your article. Will Driving Adapters return a ASP.NET MVC neutral response and will we translate this response to a form suitable for ASP.NET MVC like a view, is it correct?

Regards for your effort,

↳ Reply

hgraca says:
February 5, 2019 at 16:23
Tkx.

Short answer, yes.

Long answer:

Well, there are 2 different issues.
The driving adapters are the MVC controllers, so
1. The main idea is to do separation of concerns and regard the controllers as a translator for the HTTP request payload to whatever the user wants to do in the application core. This will also abstract the application from the delivery mechanism because we will be able to use the same logic with a CLI command instead of a controller;
2. Another idea of P&A is to abstract from the tools used, and since the framework is a tool, we can abstract from it. In that case, our controllers and views know nothing about the framework, they will know only about an abstraction layer between our application and the framework. However this can be quite difficult and counter productive to do, it depends on the project at hand, so be careful with your choices. 🙂

Good luck.

↳ Reply

Hamdi says:
February 6, 2019 at 09:30
Thanks for your response,

In your long answer's 2nd item, you wrote,

"In that case, our controllers and views know nothing about the framework"

From this, I understand that we will have view and controller parts/concepts in our application also, (inside hexagon), is this correct?

↳ Reply

Hamdi says:

February 6, 2019 at 09:39
Hi again,

Could you supply a sample of left-side port interface you developed in your previous projects ?

↳ Reply

hgraca says:
February 6, 2019 at 12:46
On the left side i dont find interfaces useful, as they would refer to core application commands/services and those we don't usually replace for another equivalent implementation, as we could do with a tool, we usually simply refactor or change our core application commands/services.
This is not finished, namely the frontend needs to be changed, but you can get an idea:
https://github.com/hgraca/explicit-architecture-php

↳ Reply

Hamdi says:
February 7, 2019 at 08:39
Thanks a lot for your kindly response,

Regards,

↳ Reply

hgraca says:
February 6, 2019 at 12:40
No, not inside the hexagon. The controllers are the driver adapters, they adapt the delivery mechanism (HTTP) to our application use cases (commands or application services) which live inside the hexagon.
Another delivery mechanism could be the CLI, and then we would have console commands as adapters, adapting the CLI to our application use cases which are the same as i mentioned above and therefore live in the hexagon.

↳ Reply

Pingback: Как правильно работать с исключениями в DDD – CHEPA website

Miguel Alfonso (@AlfonsoUno2) says:
September 18, 2018 at 10:51
Good read! I'm just wondering if you have the `UserSearchInterface` type. What are the public interface(methods) and how do you deal with criteria/search parameters?

Another example is `IEmailSender` , if for example I want to implement `SendGrid` and `MailChimp` how will you design the `IEmailSender` public methods? SendGrid and MailChimp might have different types/implementation of `EmailMessage`

↳ Reply

hgraca says:
September 18, 2018 at 14:00
Hi, tkx.

Well, the interface method signatures should be created according to what the Application Core needs, it is irrelevant how the underlying tool works.

It's the adapter role to adapt what the Application Core needs (which is reflected in the interface) to the underlying tool.

In ur 2nd example, and not knowing how either of them work from the top of my head, i would say u probably need to implement your own message value object to be used by the Application Core, which will be translated by the adapter into whatever the tool needs.

↳ Reply

Pingback: Ports and Adapters Architecture – SS Tech Digital

Valihan says:
June 17, 2018 at 08:12
Thanks a lot for this article. Finally, could wrap my head around how to do the primary adapter and port part.

↳ Reply

Pingback: O que eu vi do QconSP 2018 - Blog da Caelum: desenvolvimento, web, mobile, UX e Scrum

gerardszczepanski says:
February 10, 2018 at 15:28
Very nice article!
I found this very informative and helpful to understand Hexagonal architecture.

One thing, that gets me a little bit confused. You wrote that when we use layered architecture and we have to deal with change vendor specific library (lets say ElasticSearch) we have to change code since out logic is directly using vendor API. This can be handled by creating facade object which wraps vendor api calls and separate our business logic from vendor implementations. This approach for me is the same as creating output port and it's adapter implementation.
Am I right? Or there are some differences?

↳ Reply

hgraca says:
February 10, 2018 at 15:59
Tkx. 🙂

Well, if I understand you correctly, that facade is the adapter. It's methods are adjusted to what the codebase needs and it wraps around the tool.
I think this is good, but incomplete.

If we would switch from elasticsearch to solr, we would have 2 options:
1. Change the code in the facade, breaking the Open/Closed principle;
2. Create a new facade and replace the old one for the new one everywhere in our application core (since its a new one, the full qualified class name is different).

On the other hand, if we have an interface for that facade, we would create a new adapter (facade) implementing the same interface as the old one and tell the dependency injection container to inject the new one instead of the old one wherever its needed. The difference is subtle, but it is there. This is the main reason to use interfaces: isolate the client code from the concrete implementations.

Furthermore, we are thinking of this example as a very simple case of Interface/Implementation.
However, in many cases the port will not be just one interface, it will be a set of interfaces, classes and value objects that are used both by our application core and the adapter. In such cases I think the facade alone might not be enough to have a clean implementation.

↳ Reply

gerardszczepanski says:
February 11, 2018 at 13:05
Thank you for your explanation.

I meant that we could create interface for facade, which wraps usage of some library. If we want to change vendor lib in future, we can simply create new implementation of that interface, which wraps new library, and inject new implementation through DI container. We do not break OCP in this case and we follow Dependency Inversion principle as well.

I know it is very simple example, I actually work with hexagonal microservices, and I deal with more complex case scenarios. I just want to get this straight, because for me, we can also deal gracefully with vendor libraries in classic layered architecture as well as in hexagonal architecture. 🙂

↳ Reply

hgraca says:
February 11, 2018 at 13:07
Yes, then we are talking about exactly the same thing 🙂

↳ Reply

Pavel says:
January 26, 2018 at 16:14
Good article! I like it. It explains the theme more clear in comparison to some other articles I have read. And especial thanks for very good and clear pictures. They really shows the main concept and they are the best! :-). Keep going this way!

↳ Reply

Pingback: Ports & Adapters Architecture – Java Việt Nam
Pingback: DDD, Hexagonal, Onion, Clean, CQRS, … How I put it all together – Java Việt Nam
Pingback: DDD, Hexagonal, Onion, Clean, CQRS, … How I put it all together – @herbertograca

keth says:
November 13, 2017 at 18:50
I always thought that IOC and DI were already a thing since the late 90s:
https://avalon.apache.org/closed.html

↳ Reply

hgraca says:
November 13, 2017 at 23:40
Well, you made it clear that there were projects like the one you point out 🙂 but I don't believe it was common practice… I don't think they were yet a thing. For example, in 2002/3 I was doing JAVA in a big company and I don't remember anything about it being mentioned to me, nor reading about it neither in the web nor in the book I read back then.
But I might be wrong…
Thanks for your input, btw. I highly appreciate it.

↳ Reply

kshitijmeets says:
September 24, 2017 at 04:02
This reads like a lot of fancy words to describe IOC and dependency injection.

↳ Reply

hgraca says:

September 24, 2017 at 13:41
True, but remember that P&A was thought of back in 2005… 🙂
It was a new thing back then, IOC and DI were not out there yet, at least not like it is now days.

↪ Reply
Pingback: The Software Architecture Chronicles | @herbertograca