

DDD, Hexagonal, Onion, Clean, CQRS, ... How I put it all together

hgraca Architecture, Development, Series, The Software Architecture Chronicles, Uncategorized
November 16, 2017December 2, 2017 17 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.

After graduating from University I followed a career as a high school teacher until a few years ago I decided to drop it and become a full-time software developer.

From then on, I have always felt like I need to recover the “lost” time and learn as much as possible, as fast as possible. So I have become a bit of an addict in experimenting, reading and writing, with a special focus on software design and architecture. That's why I write these posts, to help me learn.

In my last posts, I've been writing about many of the concepts and principles that I've learned and a bit about how I reason about them. But I see these as just pieces of big a puzzle.

Today's post is about how I fit all of these pieces together and, as it seems I should give it a name, I call it **Explicit Architecture**. Furthermore, these concepts have all “*passed their battle trials*” and are used in production code on highly demanding platforms. One is a SaaS e-com platform with thousands of web-shops worldwide, another one is a marketplace, live in 2 countries with a message bus that handles over 20 million messages per month.

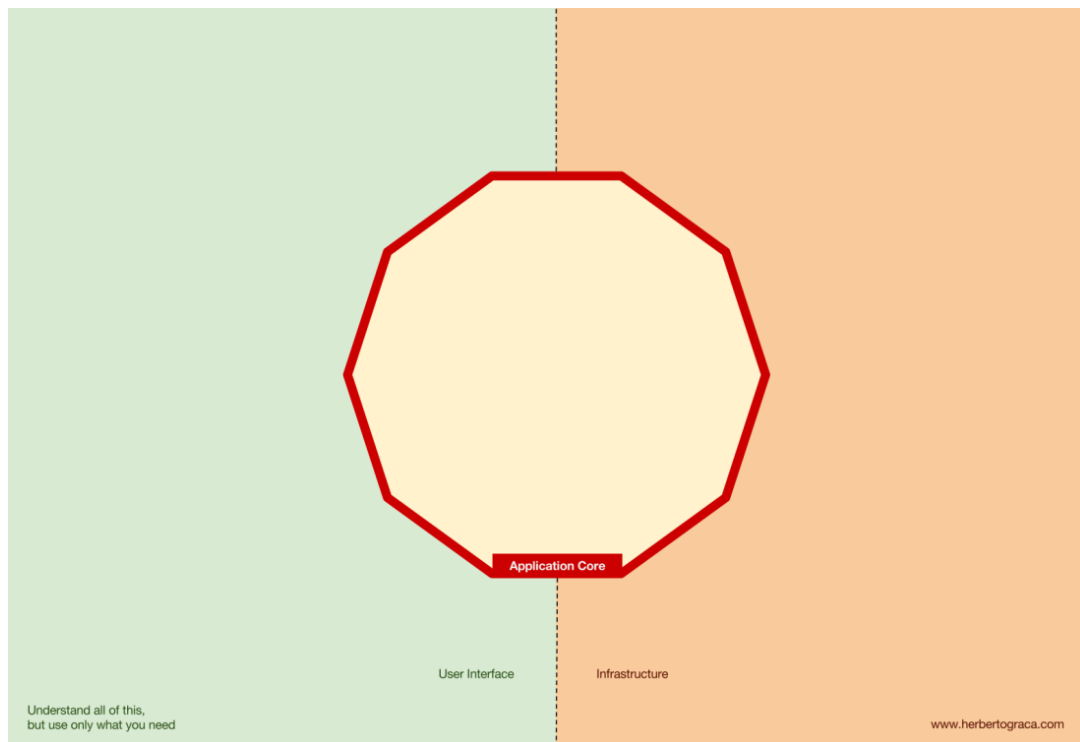
- Fundamental blocks of the system
- Tools
- Connecting the tools and delivery mechanisms to the Application Core
 - Ports
 - Primary or Driving Adapters
 - Secondary or Driven Adapters
 - Inversion of control
- Application Core Organisation
 - Application Layer
 - Domain Layer
 - Domain Services
 - Domain Model
- Components
 - Decoupling the components
 - Triggering logic in other components
 - Getting data from other components
 - Data storage shared between components
 - Data storage segregated per component
- Flow of control

Fundamental blocks of the system

I start by recalling **EBI** (<https://herbertograca.com/2017/08/24/ebi-architecture/>) and **Ports & Adapters** (<https://herbertograca.com/2017/09/14/ports-adapters-architecture/>) architectures. Both of them make an explicit separation of what code is internal to the application, what is external, and what is used for connecting internal and external code.

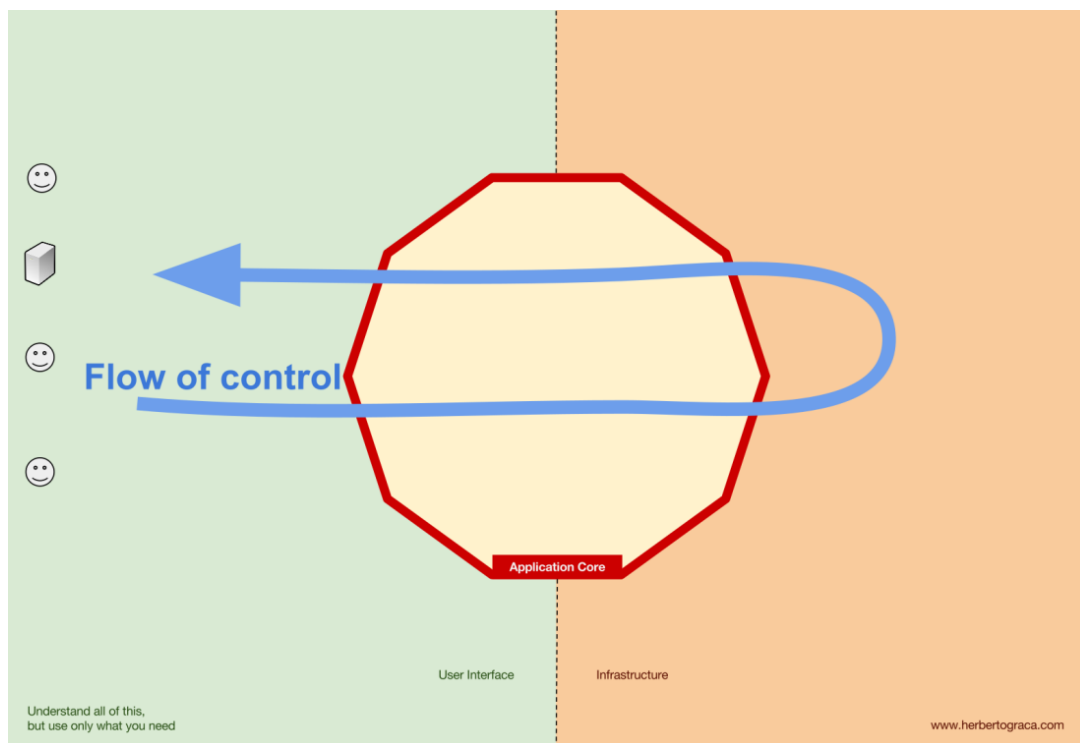
Furthermore, **Ports & Adapters** (<https://herbertograca.com/2017/09/14/ports-adapters-architecture/>) architecture explicitly identifies three fundamental blocks of code in a system:

- What makes it possible to run a **user interface**, whatever type of user interface it might be;
- The system **business logic**, or **application core**, which is used by the user interface to actually make things happen;
- **Infrastructure** code, that connects our application core to tools like a database, a search engine or 3rd party APIs.



The application core is what we should really care about. It is the code that allows our code to do what it is supposed to do, it IS our application. It might use several user interfaces (progressive web app, mobile, CLI, API, ...) but the code actually doing the work is the same and is located in the application core, it shouldn't really matter what UI triggers it.

As you can imagine, the typical application flow goes from the code in the user interface, through the application core to the infrastructure code, back to the application core and finally deliver a response to the user interface.



Tools

Far away from the most important code in our system, the application core, we have the tools that our application uses, for example, a database engine, a search engine, a Web server or a CLI console (although the last two are also delivery mechanisms).



While it might feel weird to put a CLI console in the same “bucket” as a database engine, and although they have different types of purposes, they are in fact tools used by the application. The key difference is that, while the CLI console and the web server are used to **tell our application to do something**, the database engine is

told by our application to do something. This is a very relevant distinction, as it has strong implications on how we build the code that connects those tools with the application core.

Connecting the tools and delivery mechanisms to the Application Core

The code units that connect the tools to the application core are called adapters (Ports & Adapters Architecture (<https://herbertograca.com/2017/09/14/ports-adapters-architecture/>)). The adapters are the ones that effectively implement the code that will allow the business logic to communicate with a specific tool and vice-versa.

The adapters that **tell** our application to do something are called **Primary or Driving Adapters** while the ones that are **told** by our application to do something are called **Secondary or Driven Adapters**.

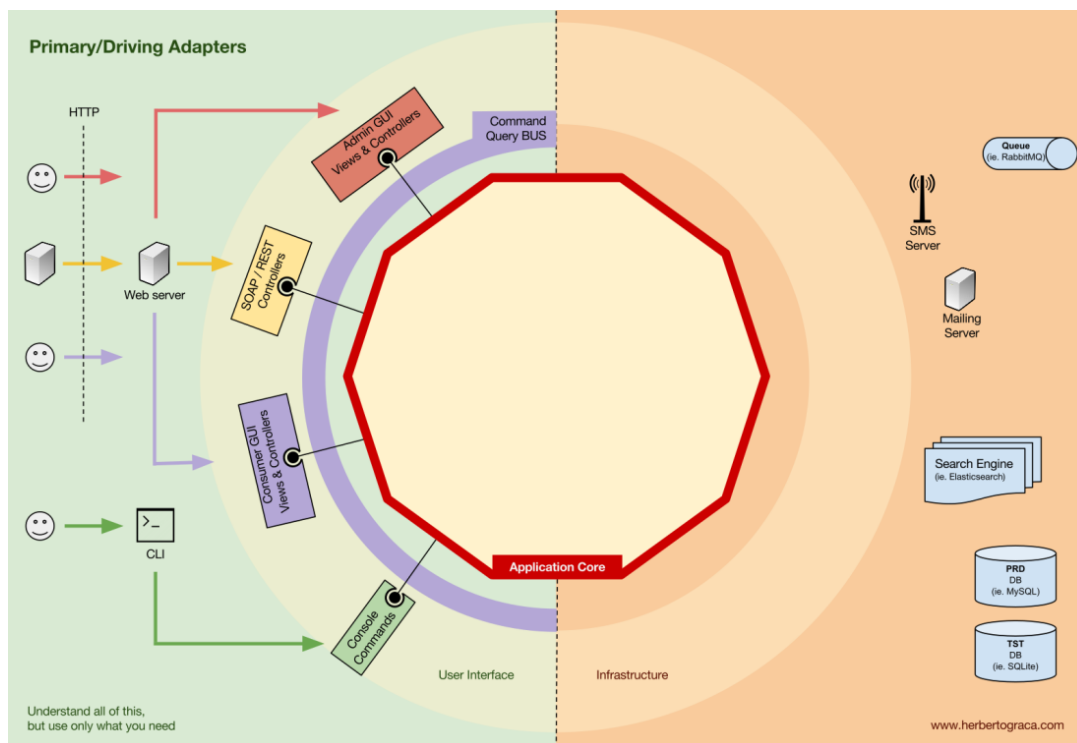
Ports

These *Adapters*, however, are not randomly created. They are created to fit a very specific entry point to the Application Core, a **Port**. A port is **nothing more than a specification** of how the tool can use the application core, or how it is used by the Application Core. In most languages and in its most simple form, this specification, the Port, will be an Interface, but it might actually be composed of several Interfaces and DTOs.

It's important to note that **the Ports (Interfaces) belong inside the business logic**, while the adapters belong outside. For this pattern to work as it should, it is of utmost importance that the Ports are created to fit the Application Core needs and not simply mimic the tools APIs.

Primary or Driving Adapters

The Primary or **Driver Adapters wrap around a Port** and use it to tell the Application Core what to do. **They translate whatever comes from a delivery mechanism into a method call in the Application Core.**



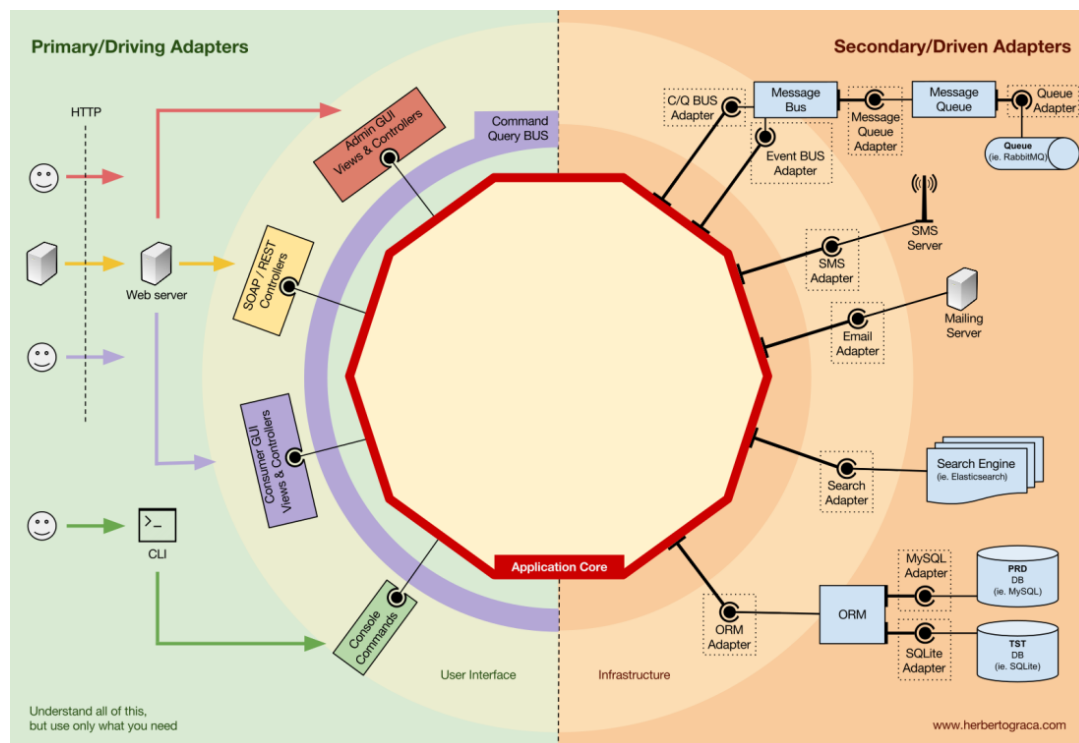
In other words, our Driving Adapters are Controllers or Console Commands who are injected in their constructor with some object whose class implements the interface (Port) that the controller or console command requires.

In a more concrete example, a Port can be a Service interface or a Repository interface that a controller requires. The concrete implementation of the Service, Repository or Query is then injected and used in the Controller.

Alternatively, a Port can be a Command Bus or Query Bus interface. In this case, a concrete implementation of the Command or Query Bus is injected into the Controller, who then constructs a Command or Query and passes it to the relevant Bus.

Secondary or Driven Adapters

Unlike the Driver Adapters, who wrap around a port, **the Driven Adapters implement a Port**, an interface, and are then injected into the Application Core, wherever the port is required (type-hinted).



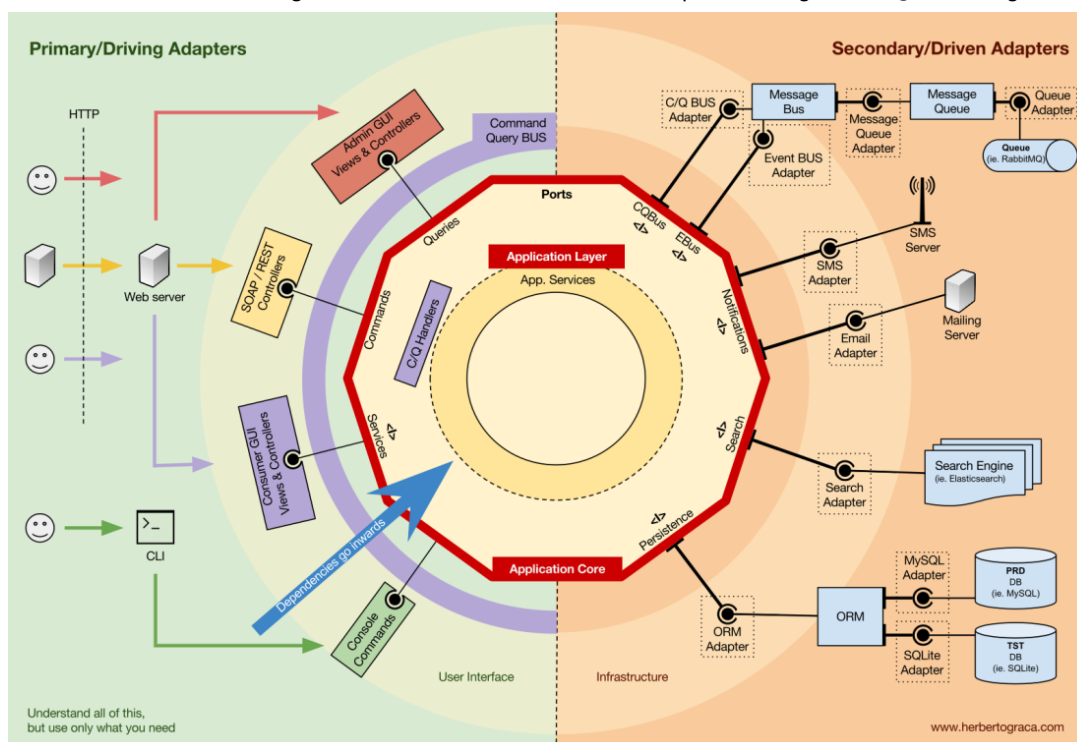
For example, let's suppose that we have a naive application which needs to persist data. So we create a persistence interface that meets its needs, with a method to *save* an array of data and a method to *delete* a line in a table by its ID. From then on, wherever our application needs to save or delete data we will require in its constructor an object that implements the persistence interface that we defined.

Now we create an adapter specific to MySQL which will implement that interface. It will have the methods to save an array and delete a line in a table, and we will inject it wherever the persistence interface is required.

If at some point we decide to change the database vendor, let's say to PostgreSQL or MongoDB, we just need to create a new adapter that implements the persistence interface and is specific to PostgreSQL, and inject the new adapter instead of the old one.

Inversion of control

A characteristic to note about this pattern is that the adapters depend on a specific tool and a specific port (by implementing an interface). But our business logic only depends on the port (interface), which is designed to fit the business logic needs, so it doesn't depend on a specific adapter or tool.



This layer contains Application Services (and their interfaces) as first class citizens, but it also contains the Ports & Adapters interfaces (ports) which include ORM interfaces, search engines interfaces, messaging interfaces and so on. In the case where we are using a Command Bus and/or a Query Bus, this layer is where the respective Handlers for the Commands and Queries belong.

The Application Services and/or Command Handlers contain the logic to unfold a use case, a business process. Typically, their role is to:

1. use a repository to find one or several entities;
2. tell those entities to do some domain logic;
3. and use the repository to persist the entities again, effectively saving the data changes.

The Command Handlers can be used in two different ways:

1. They can contain the actual logic to perform the use case;
2. They can be used as mere wiring pieces in our architecture, receiving a Command and simply triggering logic that exists in an Application Service.

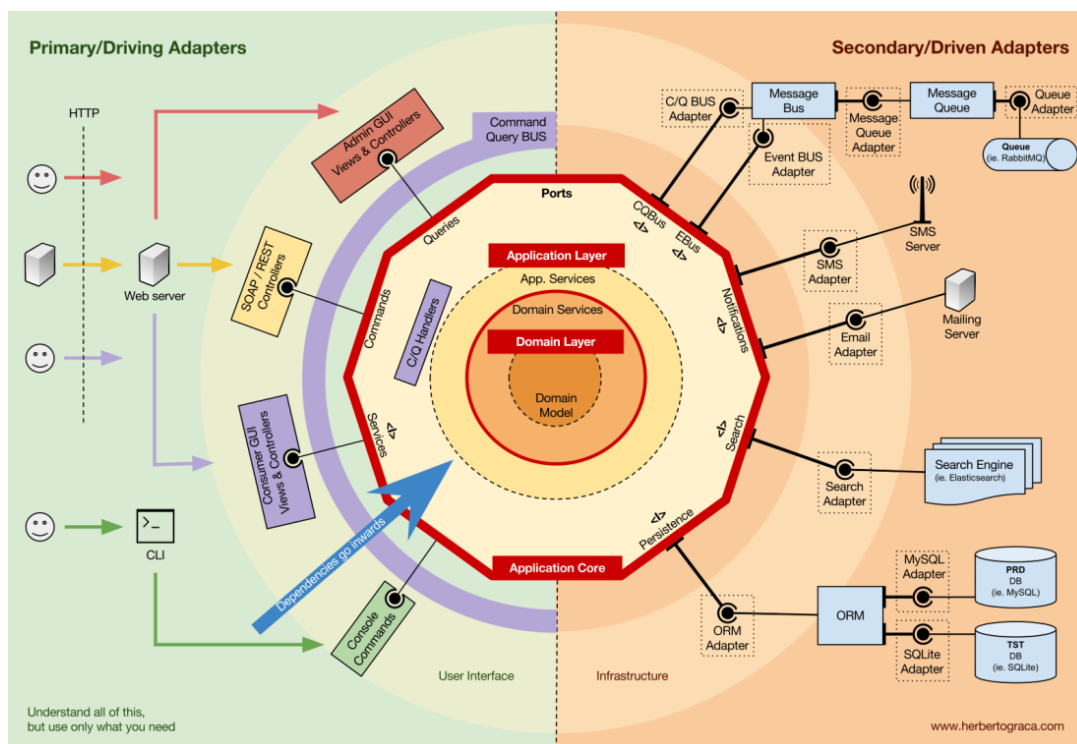
Which approach to use depends on the context, for example:

- Do we already have the Application Services in place and are now adding a Command Bus?
- Does the Command Bus allow specifying any class/method as a handler, or do they need to extend or implement existing classes or interfaces?

This layer also contains the triggering of **Application Events**, which represent some outcome of a use case. These events trigger logic that is a side effect of a use case, like sending emails, notifying a 3rd party API, sending a push notification, or even starting another use case that belongs to a different component of the application.

Domain Layer

Further inwards, we have the Domain Layer. The objects in this layer contain the data and the logic to manipulate that data, that is specific to the Domain itself and it's independent of the business processes that trigger that logic, they are independent and completely unaware of the Application Layer.



Domain Services

As I mentioned above, the role of an Application Service is to:

1. use a repository to find one or several entities;
2. tell those entities to do some domain logic;
3. and use the repository to persist the entities again, effectively saving the data changes.

However, sometimes we encounter some domain logic that involves different entities, of the same type or not, and we feel that that domain logic does not belong in the entities themselves, we feel that that logic is not their direct responsibility.

So our first reaction might be to place that logic outside the entities, in an Application Service. However, this means that that domain logic will not be reusable in other use cases: domain logic should stay out of the application layer!

The solution is to create a Domain Service, which has the role of receiving a set of entities and performing some business logic on them. A Domain Service belongs to the Domain Layer, and therefore it knows nothing about the classes in the Application Layer, like the Application Services or the Repositories. In the other hand, it can use other Domain Services and, of course, the Domain Model objects.

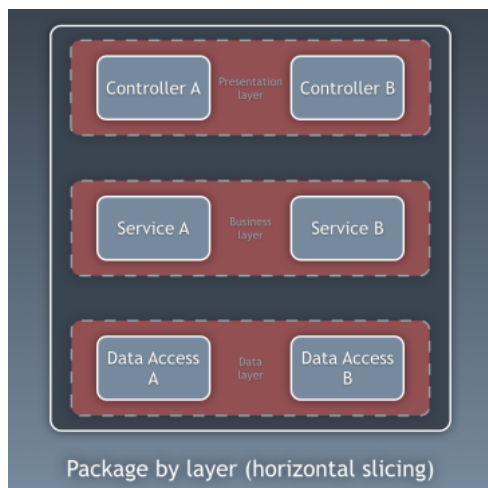
Domain Model

In the very centre, depending on nothing outside it, is the Domain Model, which contains the business objects that represent something in the domain. Examples of these objects are, first of all, Entities but also Value Objects, Enums and any objects used in the Domain Model.

The Domain Model is also where Domain Events “live”. These events are triggered when a specific set of data changes and they carry those changes with them. In other words, when an entity changes, a Domain Event is triggered and it carries the changed properties new values. These events are perfect, for example, to be used in Event Sourcing.

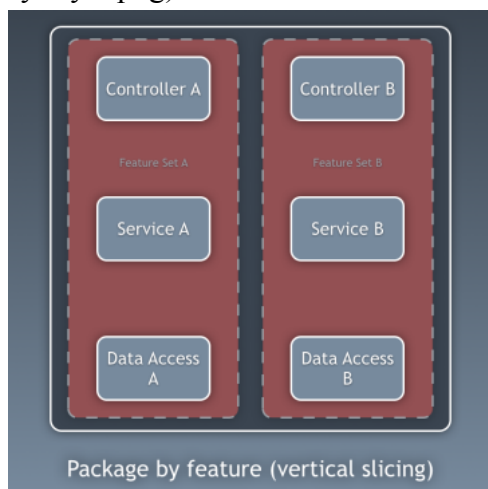
Components

So far we have been segregating the code based on layers, but that is the fine-grained code segregation. The coarse-grained segregation of code is at least as important and it’s about segregating the code according to sub-domains and ***bounded contexts*** (<http://ddd.fed.wiki.org/view/welcome-visitors/view/domain-driven-design/view/bounded-context>), following Robert C. Martin ideas expressed in ***screaming architecture*** (<https://8thlight.com/blog/uncle-bob/2011/09/30/Screaming-Architecture.html>). This is often referred to as “Package by feature” or “Package by component” as opposed to “Package by layer“, and it’s quite well explained by Simon Brown in his blog post “Package by component and architecturally-aligned testing (http://www.codingthearchitecture.com/2015/03/08/package_by_component_and_architecturally_aligned_testing.html)“:



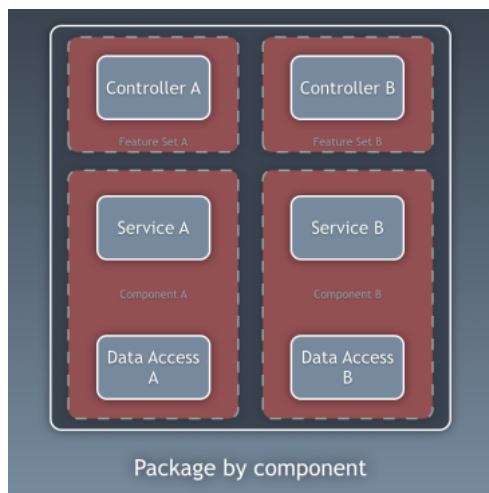
([https://herbertograca.files.wordpress.com/2017/11/20150308-package-](https://herbertograca.files.wordpress.com/2017/11/20150308-package-by-layer.png)

by-layer.png)



([https://herbertograca.files.wordpress.com/2017/11/20150308-package-](https://herbertograca.files.wordpress.com/2017/11/20150308-package-by-feature.png)

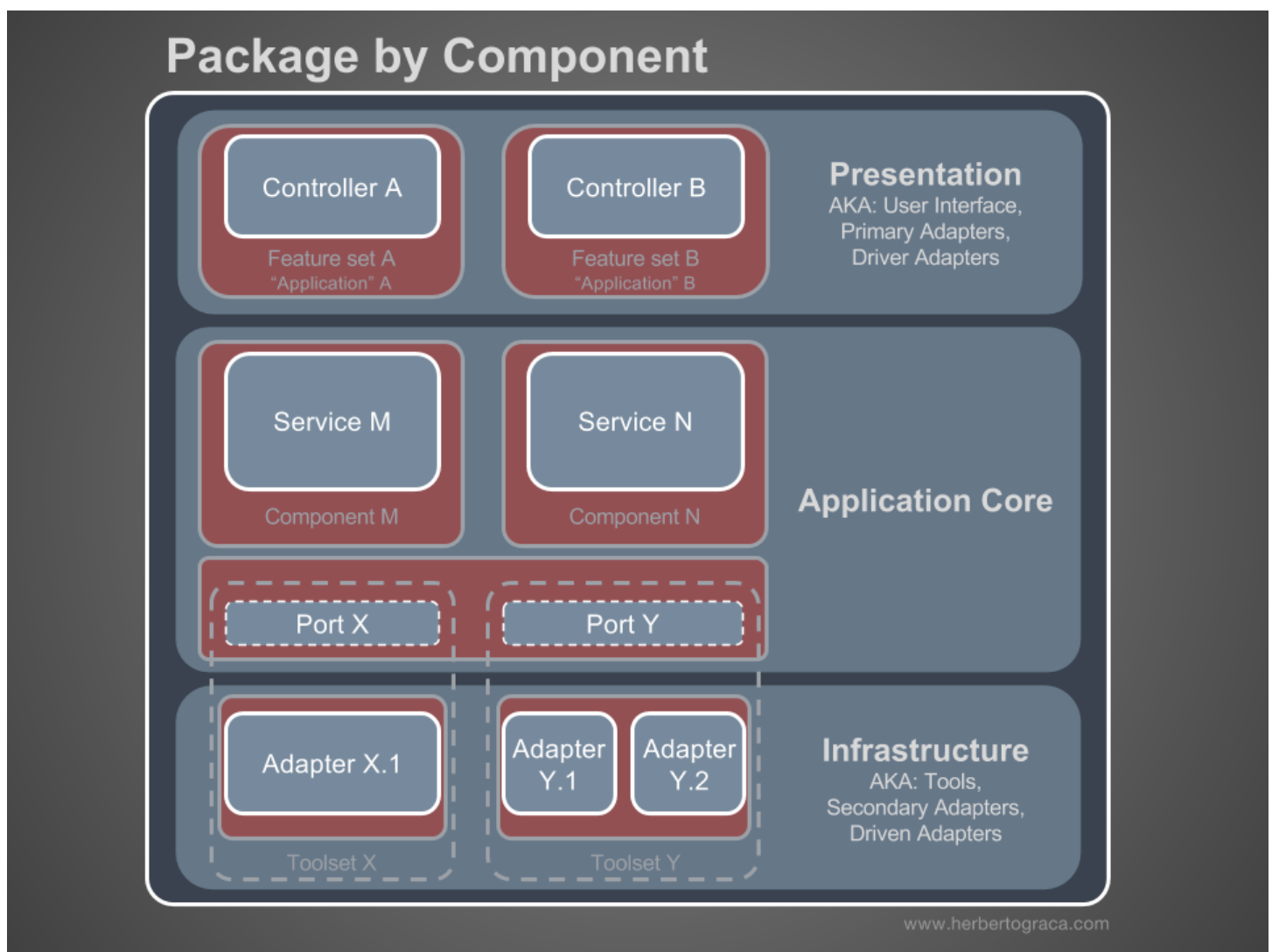
by-feature.png)



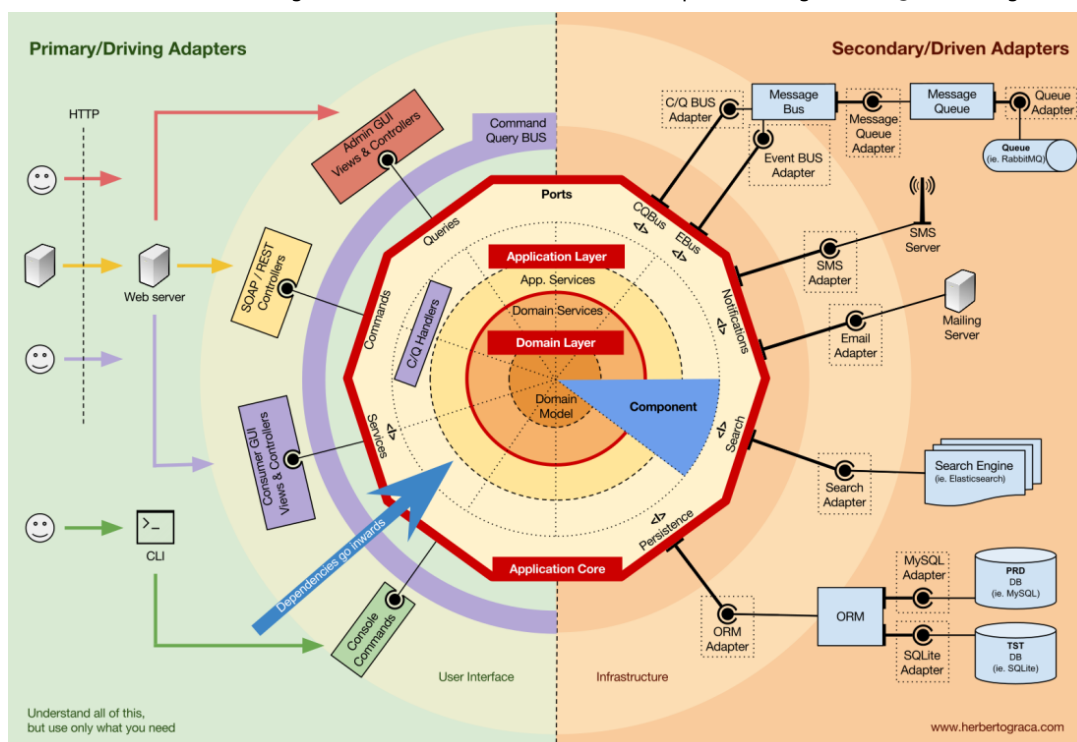
([https://herbertograca.files.wordpress.com/2017/11/20150308-package-](https://herbertograca.files.wordpress.com/2017/11/20150308-package-by-component.png)

[by-component.png](#))

I am an advocate for the “*Package by component*” approach and, picking up on Simon Brown diagram about *Package by component*, I would shamelessly change it to the following:



These sections of code are cross-cutting to the layers previously described, they are the **components** (<https://herbertograca.com/2017/07/05/software-architecture-premises/>) of our application. Examples of components can be Authentication, Authorization, Billing, User, Review or Account, but they are always related to the domain. Bounded contexts like Authorization and/or Authentication should be seen as external tools for which we create an adapter and hide behind some kind of port.

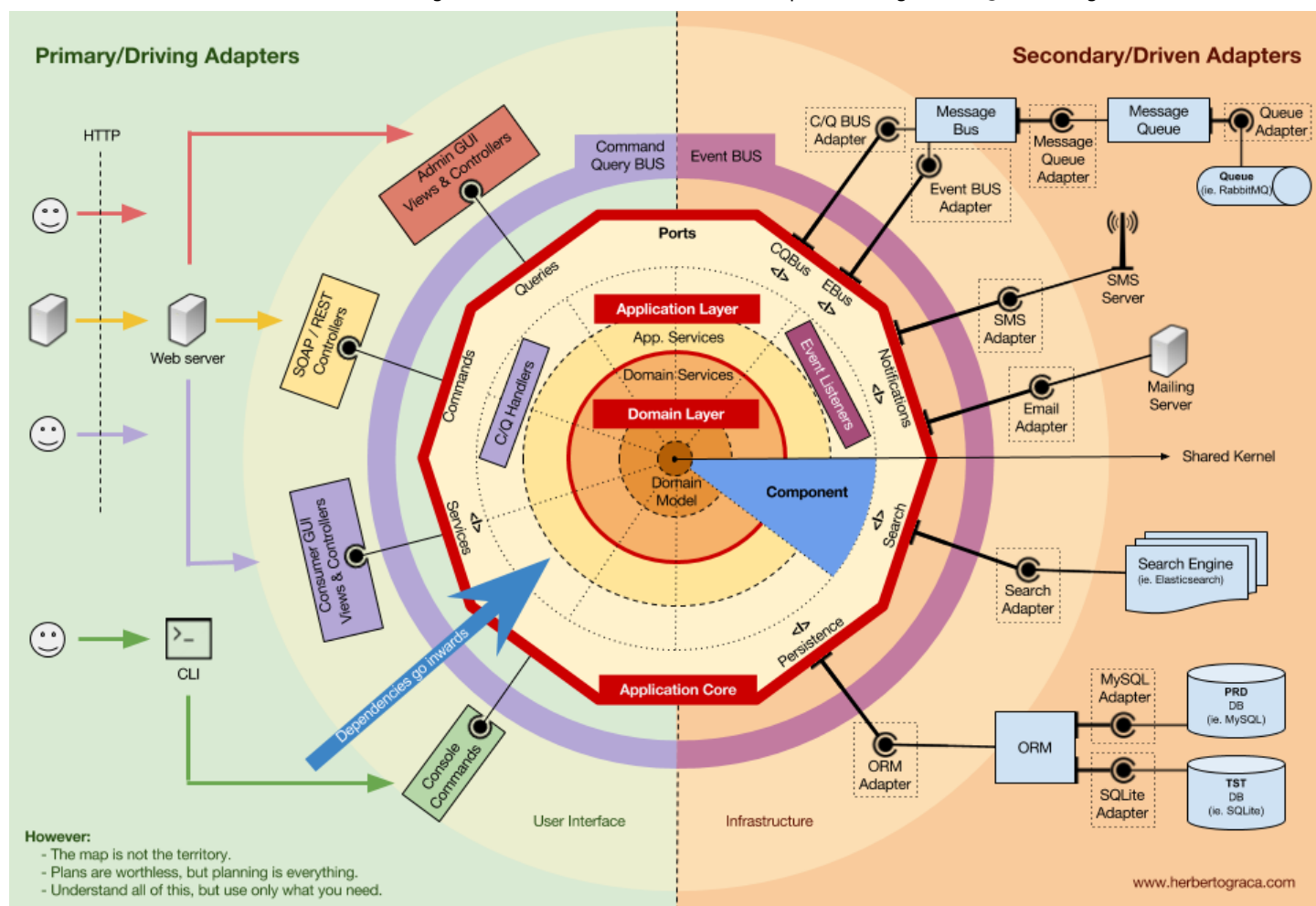


Decoupling the components

Just like the fine-grained code units (classes, interfaces, traits, mixins, ...), also the coarsely grained code-units (components) benefit from low coupling and high cohesion.

To decouple classes we make use of Dependency Injection, by injecting dependencies into a class as opposed to instantiating them inside the class, and Dependency Inversion, by making the class depend on abstractions (interfaces and/or abstract classes) instead of concrete classes. This means that the depending class has no knowledge about the concrete class that it is going to use, it has no reference to the fully qualified class name of the classes that it depends on.

In the same way, having completely decoupled components means that a component has no direct knowledge of any another component. In other words, it has no reference to any fine-grained code unit from another component, not even interfaces! This means that Dependency Injection and Dependency Inversion are not enough to decouple components, we will need some sort of architectural constructs. We might need events, a shared kernel, eventual consistency, and even a discovery service!



Triggering logic in other components

When one of our components (component B) needs to do something whenever something else happens in another component (component A), we can not simply make a direct call from component A to a class/method in component B because A would then be coupled to B.

However we can make A use an event dispatcher to dispatch an application event that will be delivered to any component listening to it, including B, and the event listener in B will trigger the desired action. This means that component A will depend on an event dispatcher, but it will be decoupled from B.

Nevertheless, if the event itself “lives” in A this means that B knows about the existence of A, it is coupled to A. To remove this dependency, we can create a library with a set of application core functionality that will be shared among all components, the Shared Kernel (<http://ddd.fed.wiki.org/view/welcome-visitors/view/domain-driven-design/view/shared-kernel>). This means that the components will both depend on the Shared Kernel but they will be decoupled from each other. The Shared Kernel will contain functionality like application and domain events, but it can also contain Specification objects, and whatever makes sense to share, keeping in mind that it should be as minimal as possible because any changes to the Shared Kernel will affect all components of the application. Furthermore, if we have a polyglot system, let's say a micro-services ecosystem where they are written in different languages, the Shared Kernel needs to be language agnostic so that it can be understood by all components, whatever the language they have been written in. For example, instead of the Shared Kernel containing an Event class, it will contain the event description (ie. name, properties, maybe even methods although these would be more useful in a Specification object) in an agnostic language like JSON, so that all components/micro-services can interpret it and maybe even auto-generate their own concrete implementations.

This approach works both in monolithic applications and distributed applications like micro-services ecosystems. However, when the events can only be delivered asynchronously, for contexts where triggering logic in other components needs to be done immediately this approach will not suffice! Component A will need to make a direct HTTP call to component B. In this case, to have the components decoupled, we will need a discovery service to which A will ask where it should send the request to trigger the desired action, or alternatively make the request to the discovery service who can proxy it to the relevant service and eventually return a response back to the requester. This approach will couple the components to the discovery service but will keep them decoupled from each other.

Getting data from other components

The way I see it, a component is not allowed to change data that it does not “own”, but it is fine for it to query and use any data.

Data storage shared between components

When a component needs to use data that belongs to another component, let's say a billing component needs to use the client name which belongs to the accounts component, the billing component will contain a query object that will query the data storage for that data. This simply means that the billing component can know about any dataset, but it must use the data that it does not “own” as read-only, by the means of queries.

Data storage segregated per component

In this case, the same pattern applies, but we have more complexity at the data storage level. Having components with their own data storage means each data storage contains:

- A set of data that it owns and is the only one allowed to change, making it the single source of truth;
- A set of data that is a copy of other components data, which it can not change on its own, but is needed for the component functionality, and it needs to be updated whenever it changes in the owner component.

Each component will create a local copy of the data it needs from other components, to be used when needed. When the data changes in the component that owns it, that owner component will trigger a domain event carrying the data changes. The components holding a copy of that data will be listening to that domain event and will update their local copy accordingly.

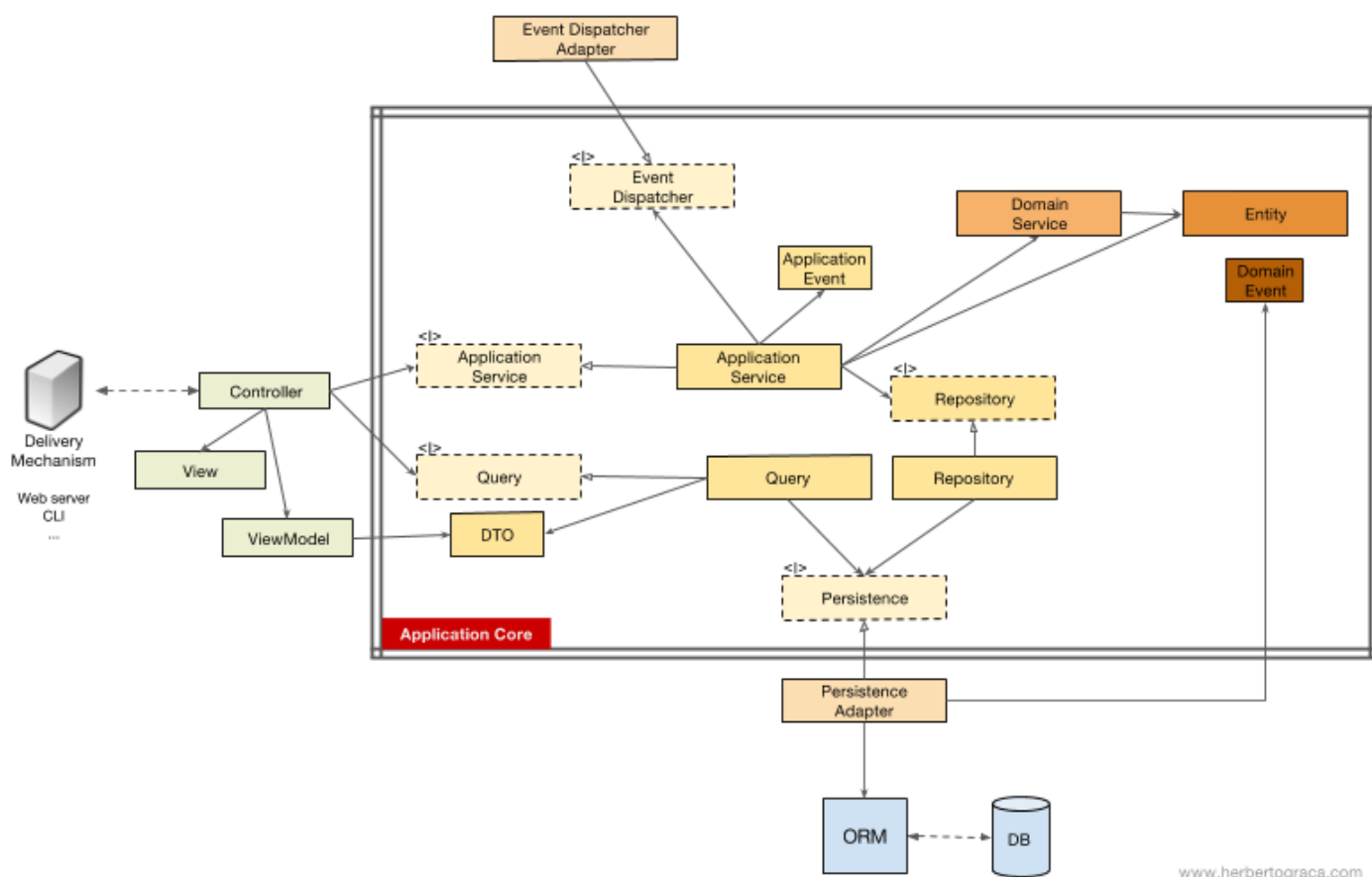
Flow of control

As I said above, the flow of control goes, of course, from the user into the Application Core, over to the infrastructure tools, back to the Application Core and finally back to the user. But how exactly do classes fit together? Which ones depend on which ones? How do we compose them?

Following Uncle Bob, in his article about Clean Architecture, I will try to explain the flow of control with UMLish diagrams...

In the case we do not use a command bus, the Controllers will depend either on an Application Service or on a Query object.

[EDIT – 2017-11-18] I completely missed the DTO I use to return data from the query, so I added it now. Tks to MorphineAdministered (<https://www.reddit.com/user/MorphineAdministered>) who pointed it out (https://www.reddit.com/r/PHP/comments/7dcz8k/ddd_hexagonal_onion_clean_cqrs_how_i_put_it_all/dpy6va4/) for me.



In the diagram above we use an interface for the Application Service, although we might argue that it is not really needed since the Application Service is part of our application code and we will not want to swap it for another implementation, although we might refactor it entirely.

The Query object will contain an optimized query that will simply return some raw data to be shown to the user. That data will be returned in a DTO which will be injected into a ViewModel. This ViewModel may have some view logic in it, and it will be used to populate a View.

The Application Service, on the other hand, will contain the use case logic, the logic we will trigger when we want to do something in the system, as opposed to simply view some data. The Application Services depend on Repositories which will return the Entity(ies) that contain the logic which needs to be triggered. It might also depend on a Domain Service to coordinate a domain process in several entities, but that is hardly ever the case.

After unfolding the use case, the Application Service might want to notify the whole system that that use case has happened, in which case it will also depend on an event dispatcher to trigger the event.

It is interesting to note that we place interfaces both on the persistence engine and on the repositories. Although it might seem redundant, they serve different purposes:

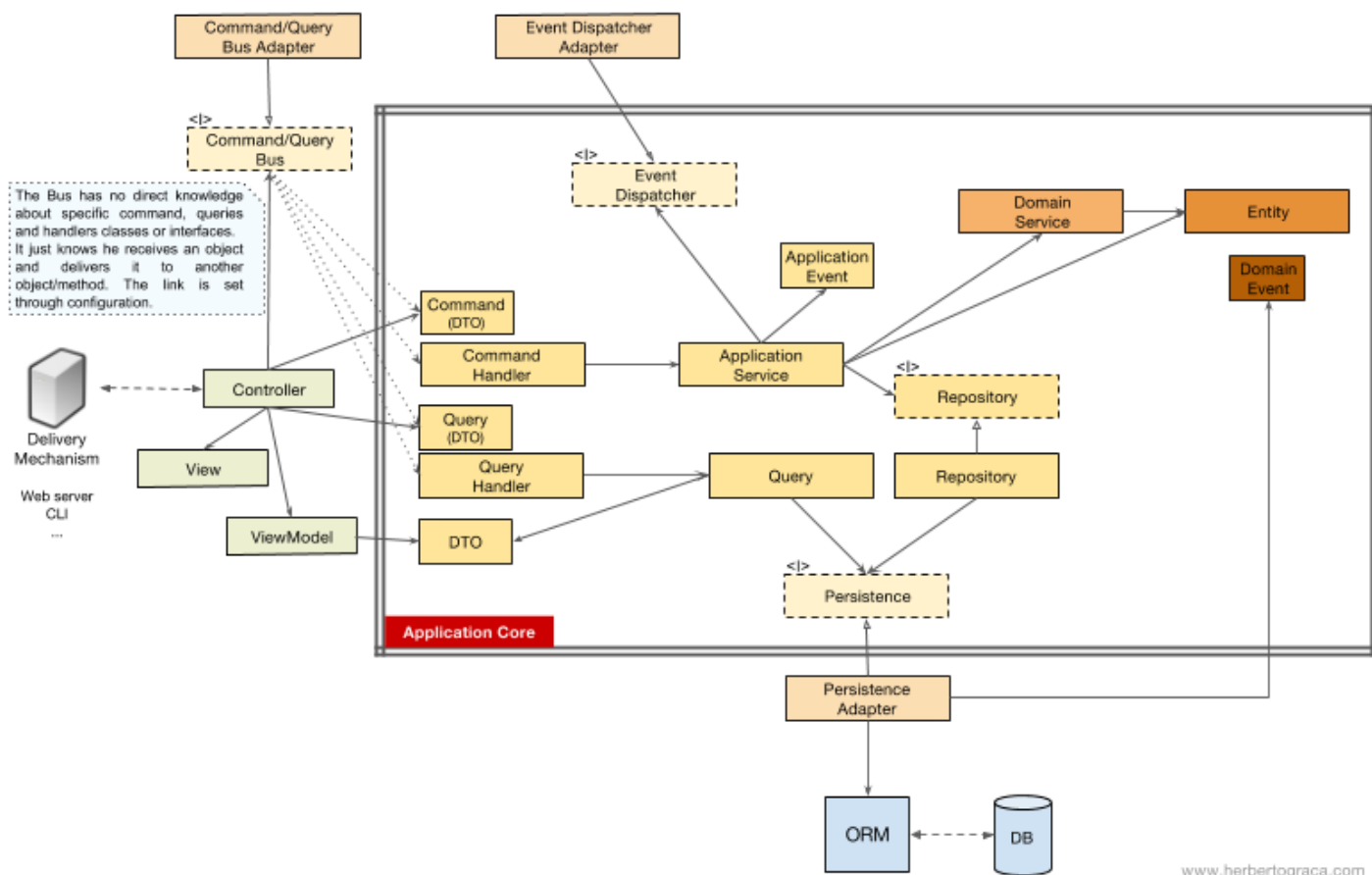
- The persistence interface is an abstraction layer over the ORM so we can swap the ORM being used with no changes to the Application Core.
- The repository interface is an abstraction on the persistence engine itself. Let's say we want to switch from MySQL to MongoDB. The persistence interface can be the same, and, if we want to continue using the same ORM, even the persistence adapter will stay the same. However, the query language is completely different, so we can create new repositories which use the same persistence mechanism, implement the same repository interfaces but builds the queries using the MongoDB query language instead of SQL.

With a Command/Query Bus

In the case that our application uses a Command/Query Bus, the diagram stays pretty much the same, with the exception that the controller now depends on the Bus and on a command or a Query. It will instantiate the Command or the Query, and pass it along to the Bus who will find the appropriate handler to receive and handle the command.

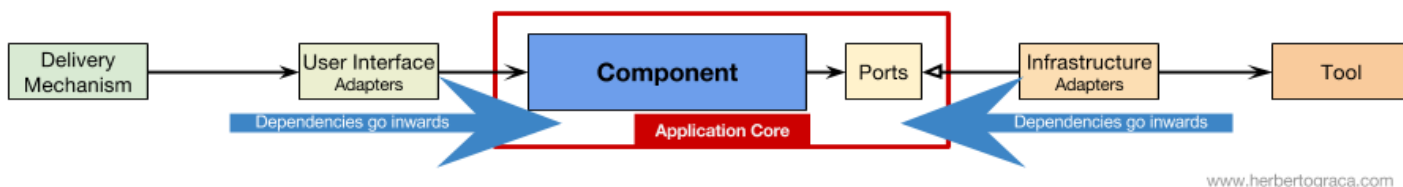
In the diagram below, the Command Handler then uses an Application Service. However, that is not always needed, in fact in most of the cases the handler will contain all the logic of the use case. We only need to extract logic from the handler into a separated Application Service if we need to reuse that same logic in another handler.

[**EDIT – 2017-11-18**] I completely missed the DTO I use to return data from the query, so I added it now. Tlx to MorphineAdministered (<https://www.reddit.com/user/MorphineAdministered>) who pointed it out (https://www.reddit.com/r/PHP/comments/7dcz8k/ddd_hexagonal_onion_clean_cqrs_how_i_put_it_all/dpy6va4/) for me.



You might have noticed that there is no dependency between the Bus and the Command, the Query nor the Handlers. This is because they should, in fact, be unaware of each other in order to provide for good decoupling. The way the Bus will know what Handler should handle what Command, or Query, should be set up with mere configuration.

As you can see, in both cases all the arrows, the dependencies, that cross the border of the application core, they point inwards. As explained before, this a fundamental rule of Ports & Adapters Architecture, Onion Architecture and Clean Architecture.



(<https://docs.google.com/drawings/d/1DGip9qyBpRHPDPKRJoXdElw1DXwmJoR-88Qvtf6hBNA/edit?usp=sharing>)

Conclusion

The goal, as always, is to have a codebase that is loosely coupled and high cohesive, so that changes are easy, fast and safe to make.

Plans are worthless, but planning is everything.

Eisenhower

This infographic is a concept map. Knowing and understanding all of these concepts will help us plan for a healthy architecture, a healthy application.

Nevertheless:

The map is not the territory.

Alfred Korzybski

Meaning that **these are just guidelines! The application is the territory, the reality, the concrete use case where we need to apply our knowledge, and that is what will define what the actual architecture will look like!**

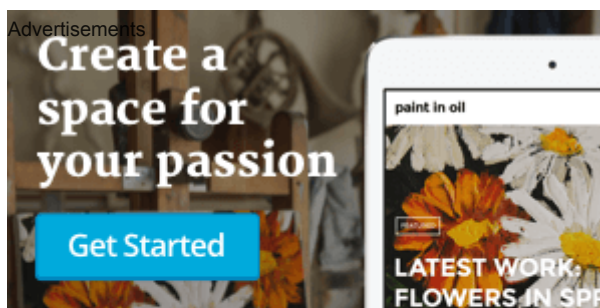
We need to understand all these patterns, but we also always need to think and understand exactly what our application needs, how far should we go for the sake decoupling and cohesiveness. This decision can depend on plenty of factors, starting with the project functional requirements, but can also include factors like the time-frame to build the application, the lifespan of the application, the experience of the development team, and so on.

This is it, this is how I make sense of it all. This is how I rationalize it in my head.

However, how do we make all this explicit in the code base? That's the subject of my next post about how I reflect the architecture and domain, in the code.

Last but not least, thanks to my colleague Francesco Mastrogioacomo (<https://www.linkedin.com/in/francescomastrogioacomo/>), for helping me make my infographic look nice. 😊

Advertisements



Create a space for your passion

Get Started

AOC I2281FWH 21.5" HDM...

477,17 zł

Kup teraz

Benq 23.8" GW2470HE LE...

Tagged:

architecture,
explicit architecture,
software architecture,
software development



Published by hgraca

[View all posts by hgraca](#)

44 thoughts on “DDD, Hexagonal, Onion, Clean, CQRS, ... How I put it all together”

Pingback: [The Software Architecture Chronicles – @herbertograca](#)

thedilab says:

November 17, 2017 at 02:43

Have you ever implemented such patterns in any projects?

🗨 Reply

hgraca says:

November 17, 2017 at 07:13

Yes, I have.

In big enterprise projects, for example one is a SaaS ecom platform with thousands of web-shops worldwide, another one is a marketplace, live in 2 countries with a message bus that handles over 20 million messages per month.

🗨 Reply

thedilab says:

November 17, 2017 at 07:59

Sounds awesome!

Regards,

Xu

tnviet says:

November 17, 2017 at 10:33

Thanks for your articles, I understand more about the patterns now.

But it seems all the backend patterns. How the frontend fit on the architecture? For example, in the single page application (React, Angular, ...) where we have components with their own business logic (like validation, hide/show field based on properties? Should the UI have their own domain model/services?

🗨 Reply

hgraca says:

November 18, 2017 at 16:44

Well, I'm not a frontend, but I feel these concepts are quite generic and usable in the frontend as well. Just the other day I was discussing with the react native developer working with me and we decided to create an adapter for the backend API his application is using because if we need to switch to another API version he can easily create another adapter and have it working.

When it comes to the domain in the frontend, it depends. If it really is just a frontend app with no backend and domain logic, then we can create a domain model as well. But in a very small app, I don't know if it is worth it, I can only judge case by case, we can not go into the hammer and nail approach.

As for “validation, hide/show field based on properties”, those are not domain concerns, those are just functionality used by the frontend. I would even regard them as language constructs that can just be used wherever. For example, PHP does not have a native function to create a Uuid, but it is a very innocuous thing, so I would just create a function to generate them and treat it as if it was part of the PHP core itself. At any point I could just change its implementation to use Uuid v3 or v4, or whatever.

I hope this helps, although as I said, I have no experience in frontend.

🗯 Reply

Patrick Roza says:

November 17, 2017 at 15:41

Love your post. Excellent reference material for some active topics for me.

This is how I've been architecting software projects for the past few years and I've been extremely happy with the results:

- Clean
- Extendable
- Flexible
- Predictable
- Scalable

At danger of treading into “Hammer & Nail” territory, I found this can be applied to pretty much every type of project, be it Desktop UI, Console, Web App, API, regardless of delivery mechanism, you name it. and virtually any programming language and framework.

I never liked how frameworks and libraries would take over your project structure and dependencies, as well as infiltrate your lingo and dictating your choices.

But with this approach it has become just a detail, sitting at the outside boundary of your app.

🗯 Reply

Ray Clanan says:

November 17, 2017 at 19:33

Great article, I can't wait to read more from you.

🗯 Reply

Pingback: [Clean architecture links | lessthan12ms.com](#)

Dmitriy says:

November 18, 2017 at 11:18

Wow, that is an article! Thanks for putting such a huge amount of work to compose it.

I have a lot of questions about proper architecture but the most struggling one which is bothering me in my work is authorization.

Would you mind to put some light on how you'd design authorization across all the layers and components? You mentioned that component itself can be of the purpose of authorization. How does it all work together?

🗯 Reply

hgraca says:

November 18, 2017 at 16:25

Tkx. 😊

I would use an ACL module. I would try to find an ACL module, a tool, that I could just create an adapter for it and protect the routes according to what user is logged in, and some configuration files specifying what type of user has access to each route.

I think Authorization and Authentication were not a good example, it's a flop, sorry. The Components should be Domain related, they are BoundedContexts, and Authorization and Authentication are not domain. I will replace those examples immediately.

🔒 Reply

Peter says:

November 19, 2017 at 17:45

I think you need to make the distinction between authorising and authenticating (making sure this person is who claim to be and that he can actually access the part of the application he is trying to access) and managing you authentication and authorization (user registration, user login, managing user roles, ...)

The latter can be seen as a (sub)domain of your application. However this is usually pretty generic and you can easily use an existing component to handle all of it, because why spend precious time reinventing and redesigning the wheel. Unless you have very specific needs which you can't find in the existing components.

The former is in my opinion a concern of your (driving) adapter. It is (usually) not your application logic that should check whether the user is logged in, or has access to this specific command. It is the duty of your web controller (or even web framework) to handle this. I see it this way, because depending on where the request comes from, your access rules may vary. For example, when using the CLI, you could assume the user doesn't need to be logged in, as he is an admin of your system. Or you may want some functionality to be accessible not only to a person logged in to your system, but also to someone logged in into an external admin system (like someone from a helpdesk who can edit a users account).

Sometimes it might seem to be a domain rule that some things can only be altered by the user who created this thing, but more often than not, this business rule doesn't hold up when you start thinking about the part of the system, which can not be seen be a regular user.

Dmitriy says:

November 20, 2017 at 12:13

I can't find a place in my head for a proper authorization model. While ACL is a good separation (if I understand it correctly) – like if you have some particular role attached to you, then you can post to a blog, if don't then you cant.

In my practice, there were more complex rules which transcend layers (app and domain layers.). You might need a specific role to make a domain action but also you need to meet some specific domain criteria.

For example, “You can post a comment if you have a “client” role AND you’ve purchased this product in the past”. This kind of rules involves specific domain logic to be executed.

So when you saying that “Authorization and Authentication are not domain”, I kind of disagree about authorization. Because Authorization can involve domain rules. Right?

I really want to learn effective ways to organize authorization logic in clean architecture. While I do understand all this layer&ports&adapters paradigm I feel like authorization part is often left undisclosed in many blog posts and books.

p.s. I haven't read the latest Uncle Bob's book on the subject. Perhaps there are some hints about authorization.

hgraca says:

December 1, 2017 at 13:32

Sorry, I've missed your comment last week.

But to answer you, what you are talking about is not exactly Authorization in terms that I was talking about.

I was referring to Authorization in the sense of "Who can access what" platform-wide, which is basically a configuration that is set up in some config file, although to make it scalable we should use a generic ACL component.

What you are referring to is indeed business rules, not authorization in the sense of a platform-wide set of rules.

So those business rules do not belong in a generic Authorization component, it belongs in the component that needs it.

Going back to your example "You can post a comment if you have a 'client' role AND you've purchased this product in the past", we only need this when the consumer issues a command to post a comment.

So, the way I see it, this is a specific rule for this specific use case (ie. "CreateProductComment"), of a specific component (ie. "ReviewComponent").

We need to validate this rule but it is very specific.

Doing it quick and dirty, we can simply put an if statement in the command handler (or application service method).

Doing it a bit more scalable, we can create an authorization service specific to this component which will contain all the validation rules for all commands in the component.

Doing it a bit better, we could have a configuration setup saying that a use case (command, or application service method) prior to running needs to be validated with some other class/method.

If we are using a Command Bus, this would be something that could be done there, with a validation middleware.

Depending on our concrete needs, we could also use a Specification pattern where we would put fine-grained business rules, in such a way that we could eventually create complex business rules in specification objects by composing them from other specification objects.

In any case, both the use case code and its validation rules belong together, in the same component, and it is not what is generally referred to as Authorization.

Dmitriy says:

December 3, 2017 at 07:35

(I was not able to reply directly to the message you've replied to me with. Maybe the blog reached maximum comment depth)

Authorization

I see you separate the two: generic ACL authorization and business rules. It makes total sense to me to have a separated authorizer class attached to a command handler. In fact, I practiced it and it worked very well.

So to follow your logic, generic ACL lets administrator to manually assign roles to users and allow roles to perform certain actions. Once a user is allowed to make a command only then domain rules will perform additional validation against the command and maybe reject it due to some unmet

restrictions. It is like a 2-level authorization.

Performance

I was recently asked on my blog (<https://lessthan12ms.com/authorization-and-authentication-in-clean-architecture/#comment-3629170994>) how to deal with performance and code separation. For example, an authorizer class and command handler both can query the same data to make the work. But since they are independent, we may end up with multiple database queries.

How do you design such things in your apps? Can you give some guidance on improving performance in “clean architecture” apps?

hgraca says:

December 3, 2017 at 09:42

Glad my explanation helped you 😊

About the performance related to repeating queries in different contexts of the application, I think the Application Core should not concern itself with it. I think that is something for the ORM to deal with.

Doctrine, for example, uses a 1st level cache that stores the objects that were already fetched from the DB. It also keeps track of what queries were executed and what objects they returned. This 1st level cache is up during the request and only in the end it flushes all changes to the DB. So it takes care of repeating queries.

Doctrine also has a 2nd level cache where it stores objects in redis and reuses that cache through several requests.

Nevertheless, what I think we need to be very careful with is the N+1 problem when we create our queries. There is not much the ORM can do to help us with that except for telling us that a query is being repeated.

Tomasz Sadza says:

November 18, 2017 at 11:47

Hi, may I ask about final infographic in scalable version (vector pdf or svg) to printout on the wall in my developers office ? It's very good and I think I may be usefull many times. Thank you

🗨 Reply

hgraca says:

November 18, 2017 at 16:13

Sure, here you go: [https://drive.google.com/open?](https://drive.google.com/open?id=1E_hx5B4czRVFVhGJbrbPDlb_JFxJC8fYB86OMzZuAhg)

[id=1E_hx5B4czRVFVhGJbrbPDlb_JFxJC8fYB86OMzZuAhg](https://drive.google.com/open?id=1E_hx5B4czRVFVhGJbrbPDlb_JFxJC8fYB86OMzZuAhg)

🗨 Reply

alyaros says:

November 18, 2017 at 17:14

First I would like to say that this is a very good summary of all the concepts (DDD, CQRS, P&A etc...) into one coherent whole solution and also to take the opportunity and say that I really enjoy reading your blog posts.

I'm familiar with the ideas mentioned in the post for several years now and also I've been following those topics in various places: blogs, books, conferences and more and also tested/implemented some of them myself in various projects to some extent.

There 2 main issues that bother me the most regarding this and I'll be very happy to hear your opinion.

1) Creating many abstraction and separations (Domain objects, Domain services, App services, DTO's ,

Persistence adapters etc..) is adding a lot of entities to maintain and also many mappers between them – this creates A lot of boilerplate code that is hard to maintain. For example I have at the minimum 3 flavors of the same entity – one for the primary adapters (i.e. return User to the UI\service-client), second for the domain, third for the persistence adapter. (Usually it's simple 1:1 mapping but obviously not always)

2) Such abstractions creates significant performance penalty mainly on data driven applications. For example let's look on the separation between domain and persistence. We aspire to keep the business code isolated inside the domain and treating the persistence as storage mechanism – we can save or getBy (filter) but we try to keep the business decision\logic inside the domain – doing so can lead to a very inefficient usage of the persistence (I.e. DB) – for example instead of creating a more complex query that will do some of the calculation on the DB itself we fetch a lot of data, translate it to domain, perform the calculation and return the result. The question is how you prevent leakage of business rules into the persistence adapter while still keeping the process efficient as possible.

🗨 Reply

hgraca says:

November 19, 2017 at 23:12

Hi,

Thank you, I'm glad you have been enjoying my posts. 😊

Regarding your questions:

1) Yes, you are absolutely right. Doing all this creates more code.

However, I don't think it has high maintainability costs as once it is done, it just sits there. If we need to change it, it's better to change in that one place, as opposed to everywhere where the actual tool would be explicitly used. But on the other hand, it does give us the advantage of easily changing how we handle the actual tool behind the port/adapter.

For example, at my current company we had an adapter for the symfony event dispatcher and all our core only knew about the interface in front of the adapter. When we decided to start using async events, we just had to create the adapter for the async events and add it to the list of adapters being used by the actual dispatcher being used in our application.

The same thing goes for the cases where we want to upgrade some library version. If we have an adapter, we just need to write a new adapter, the application core will stay the same.

These practices give a lot of stability to the application core.

Nevertheless, Software Architecture is all about tradeoffs. So, if the project at hand is not going to be long-lived, or the company is still in a stage of lean-startup, and creating these ports/adapters will be too time-consuming, maybe it's better to just use the tools directly throughout the codebase. Also keep in mind that there are ports/adapters easier to create than others, so maybe we can create the ones that are easier to create and not the most difficult ones.

In the end, it's all about tradeoffs. The important is that we know all these patterns, because they can help us at some point, but we use only what is worth using in a specific context.

2) Picking up your example “for example instead of creating a more complex query that will do some of the calculation on the DB itself we fetch a lot of data, translate it to domain, perform the calculation and return the result. The question is how you prevent leakage of business rules into the persistence adapter while still keeping the process efficient as possible.”

I think that the actual domain is the fact that we use the result of that specific calculation, not the way we actually calculate it. So, I would create a query object with a class name and method name that accurately reflects the calculation being made from the domain point of view, and have the DB make that calculation for me and just return the result. Since we only want to get data, we don't need to use an ORM to hydrate the data into entities. We only need to hydrate entities if we want them to “do”

something, as opposed to just give use some data to show our users.

I use query objects when I just want to get data. A query object only has one query inside.

I use repositories pretty much as collections. A repository always returns an entity or set of entities.

I hope this helps.

🗯 Reply

Oleksandr Sova says:

December 30, 2017 at 03:24

Thank you for your great post/article. Wish to have it couple years earlier. Everything described amazing but still have one question: at previous comment you told about that when you do a Query and you only need the data you do it avoiding ORM and other stuff. That is a way that I usually do. For instance I have a controller that depends on, lets say *LastMonthDeposits* – *contract of a query, which is implemented by LastMonthDepositsPostgreSQL which depends DIRECTLY on connection. And at schema your query depends on contract of persistence which is implemented by persistence adapter which depends on whole ORM and so on. So why would anyone choose to go all the way down through ORM instead of be specific? And in terms of PHP – why would anyone use adapter over doctrine/orm instead of direct doctrine/dbal (query level cache is it's responsibility) in case of Query, not Command handling?*

Yes, I have read that it's just a guideline and so on but the main thing that I'd like to know is what were the preconditions/circumstances to emit such particular guideline?

hgraca says:

December 30, 2017 at 13:26

Hi, tkx.

If I understand correctly, your question is “Why does the query object implementation depend on the persistence port, whose implementation uses the ORM/DBAL, instead of the connection directly?”

There are 2 things I wanna point out to answer your question:

1.

The reason is the same as usual, by making the query object implementation depend on the port, we can change or completely swap the persistence implementation without changing the query object implementation (as long as the query language is the same). For example, if we use a port/adaptor we can add a line of code to the adapter to log all queries being made. In the other hand, if we inject the connection in all our query objects, to add the same logging we will have to add that line of code to all the query objects and inject that logger into all of those objects.

2.

The implementation of the persistence port does not need to be the same. So, for commands, you can use an implementation (adapter) that wraps around the ORM but, for the query objects, you can use an implementation that wraps around the DBAL, or the connection itself. This way, you can skip the ORM and/or DBAL overhead when you don't need it and still have the flexibility that the port/adaptor brings us.

Did this answer your question?

Stephane says:

November 19, 2017 at 13:48

Thanks for the article.

Great subject.

Would you be willing to present that at Wajug user group (in belgium)?

We're looking for great speakers and great subjects.

Have a look at our upcoming/past events:

<http://www.wajug.be/>

<http://www.wajug.be/pastEvents>

Hope to hear from you,

Stephane

🗨 Reply

hgraca says:

November 19, 2017 at 23:22

Thank you!

I feel flattered now!! 😊

Liege is a bit out of my way though...

But I still wanna write a few more posts and then if I decide to make some talks I will send you guys an email.

Thank you for the invite, I really appreciate it.

🗨 Reply

Stephane Rondal says:

November 20, 2017 at 09:15

You're welcome, I really liked your post.

No problem, contact me when you're ready for some talks.

It will be a pleasure to host you.

And Liege is not that far from Amsterdam 😊

In the meantime, good continuation with your excellent articles.

Cheers,
Stephane

png hai says:

November 21, 2017 at 10:27

Reblogged this on [Pikachu's Corner](#).

🗨 Reply

Pingback: [November-2017.php – Jesper Jarlskov's blog](#)

Tiago Brito (@tlfbrito) says:

December 5, 2017 at 00:55

Thanks for this article. Very detailed.

🗨 Reply

Radek Maziarka says:

December 12, 2017 at 23:25

I am amazed by this post. Great work 😊

🗨 Reply

Pingback: [DDD, Hexagonal, Onion, Clean, CQRS, ... How I put it all together – Java Việt Nam](#)

Juan says:

December 19, 2017 at 01:34

Hello. Great post. Congratulations.

How do you deal with secondary ports and components?

Do you consider a port being part of the component, or is the port a separate layer?

For example, if each component owns its repository the secondary adapter should depends on all components (the whole app core, not just the ports).

But if you put ports in a separate layer, the you break package by component design.

Thanks, Juan.

🗨 Reply

hgraca says:

December 19, 2017 at 11:23

Tkx for your nice words, I appreciate it. 😊

As for your question, I don't understand what you mean by 'secondary ports and components'. But maybe this will make it clear:

If my application uses an ORM, I will create a Persistence port and an adapter for the ORM. That port will be used by all components that need the ORM. The ports belong to the application core, its an entry/exit point to the application core, not to a specific component.

Repositories are specific to a component, si I place them in its specific component. They depend on the ORM port, it is injected in the repository constructor.

The tricky thing with repositories is that they are also specific to the query language. They implement a repository interface, but then there can be different implementations per query language. A query language depends on the persistence engine used, it can be SQL, Mongo Query Language, Elasticsearch query language, Doctrine Query Language, ... whatever.

This means that a repository, although not an adapter around the ORM, it is an adapter for the query language, and it might need to know some specifics about what is outside the Application Core, it definitely needs to know what language it is using, and if using DQL it needs the doctrine query builder to be injected. Unless we create our own query language, like Doctrine did, but I think that is going way too far.

Did this answer your question?

🗨 Reply

Oleksandr Sova says:

December 30, 2017 at 03:50

Hello. Why do you think that *put ports in a separate layer, the you break package by component design?*

Your packages/modules may strictly require other module which in this case may be it's bounadry.

Uncle Bob described it like "plug-in approach". So consider next packages relation structure:

```
+-----+ +-----+ +-----+
| Infrastructure | | UseCase | | App impl |
| module where |==requires==>| boundary |<==requires==| where |
| ControllerA | | where is | | ConcreteA |
| belongs to | | < I > ContractA | | belongs to |
+-----+ +-----+ +-----+
```

Infrastructure do knows about boundary and only about it as a contract but dependencies are still inwards. So ControllerA –depends on–> ContractA implemented by ConcreteA. What is breaking this way? Packages may require other packages to be a part of it so it's still top level package by component. And packages may be re-used at other packages/components so it's way like plugin. As for example – to publish an IDE plugin you don't have to publish whole IDE with it.

🗨 Reply

Juan says:

December 31, 2017 at 05:51

Hi, I explain it in my reply to hgraca (<https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/#comment-753>). In my implementation of hexagonal architecture, I have a jar for every port and another jar for the core (hexagon = core + ports). Ports are the boundary of the hexagon. The outside world (adapters) access the inside through ports. I have a jar for every adapter. Core depends on ports (interfaces), and every adapter depends on the port that it implements. Ports isolate the core from the outside. So say you have ComponentA and ComponentB in the core. RepositoryA and RepositoryB are interfaces belonging to the persistence port for accessing the database, so they both belong to the “persistence port”. But this breaks “package by feature” (because RepositoryA should belong to ComponentA, and RepositoryB to ComponentB). My solution was to add a RepositoryA class in ComponentA, that just gets injected the RepositoryA interface of the port, and forwards every method call to the port. Same for B.

Apologize to hgraca, this is your blog, not mine (I don't have any), but I've answered as the question was about something I said.

Regards,
Juan.

hgraca says:

December 31, 2017 at 07:41

No need to apologize Juan, the more ppl participate the better.

One of the reasons i created the blog was to give back to the community, and we are all part of it 😊

Juan says:

December 19, 2017 at 13:40

Thanks for your answer, I appreciate it a lot too, because the architecture of my project is very similar to yours, I'm a fan of ports and adapters architecture and DDD. When I saw this post I said: “this is amazing, perfect”. And the picture of the whole architecture is awesome. In general, your posts are great. I say it because it's what I really think.

By secondary ports I mean what you call just ports. They belong to the core but I split them physically apart (core = domain model + secondary ports). Domain model depends on secondary ports. They are interfaces for talking to external actors (systems, humans, services, databases or whatever). The language of the port methods signatures are the language of the core. A secondary adapter implements (depends on) the secondary port for a specific external actor and technology. I said Repository (persistence port) as an example of secondary port. In my case I split the core in modules (DDD concept), more or less is what you call component. A module has one or more aggregates, each one with its repository interface. This aggregates repo interfaces should be the secondary port, but if I place them (one per aggregate) in the port, then I break the “package by feature” design. So I decided to add kind of a “port proxy” in the domain model, that gets the port injected and forward the method calls to the port. By this I add complexity and duplication. But if I don't do it, then the persistence adapter (repo implementation) should depend on the whole domain model, and that's not correct from my point of view. Every secondary adapter must depend just on the secondary port that it implements.

I think your answer (“Repositories are specific to a component, si I place them in its specific component. They depend on the ORM port, it is injected in the repository constructor”) is more or less what I mean.

Thank you very much.

PD: Just one more question: do you have a persistence model in the persistence port, different from the domain model, and a translator between the two? Components of the domain model depend on persistence port and persistence adapter depends on the port too. Port doesn't depend on anything, so it knows nothing about the domain model.

🗨 Reply

hgraca says:

December 19, 2017 at 19:50

Thank you again for your comment, quite interesting how you do see it, and how similarly we do it! 😊

I'm not quite sure what you mean by a persistence model. However, I can say that the persistence port indeed knows nothing of the domain although all ports should be designed to fit what the domain needs as opposed to what would be easier to implement by the adapter. Otherwise, the adapter is just a wrapper around the tool. In other words, we should first design the port as it would be easier to use by the components, and then implement the adapter.

In my concrete case, I use Doctrine (similar to Hibernate) which is an ORM that uses a data-mapper pattern, as opposed to an active-record pattern, so I can indeed have a data model completely different from the domain model, although that is not usually the case, and the mapping from domain model to data model is set up with configuration files understood by the ORM itself.

Did this answer your question? 😊

🗨 Reply

Juan says:

December 19, 2017 at 20:36

I totally agree with you.

By persistence model I mean the model mapped to the database. I use jpa and xml mapping, in order to not pollute the entities with annotations. Again, I think it is similar to yours.

Of course you did answer my question.

Thanks again.

I'm looking forward to see your next articles.

Regards, Juan.

jantonioreyes says:

January 15, 2018 at 14:09

Great article.

Thanks for this post.

I've a doubt about just one point, where you're talking about put repository interfaces in the Application Layer. According to DDD we use to put them in Domain, so if they belong to Application, where they must be implemented?

If interfaces are in Application, and implementations in Infrastructure, must Infrastructure have dependencies from Domain and Application layers?

Thank you so much, and keep it going!

Regards.

🗨 Reply

hgraca says:

January 15, 2018 at 23:52

Hi, tkx for your comment.

I put both the repository interface and the implementation(s) in the application layer.

I see the repository interfaces as an abstraction over a query language. So there can be different implementations of the same repository interface, each implementation using a different query language, ie one implementation with DQL, another with SQL, another with Mongo QL, ...

Each implementation gets injected a persistence interface which lives and is implemented in the infrastructure layer. The persistence interface implementation injected can be different for all different repositories or it can be the same by using a strategy pattern.

I hope this was clear...

In my next post i have an example of this implementation. 😊

🗨 Reply

Juan says:

January 16, 2018 at 00:54

Hi hgraca, I don't agree with you at this point. I think that neither putting repo interfaces nor their implementations in the application layer is right from a DDD point of view.

Repo interfaces belong to the domain, one Repo per Aggregate Root. And the implementation should live in the infrastructure layer, implementing the repo methods (named using the UL) with languages according to the storage (SQL, Mogo, etc).

The application layer, as the direct client of the domain, uses the repositories interfaces.

Persistence is not a cross-cutting concern, like transactions, logging, exception handling, etc, so persistence shouldn't live in the application layer.

Well that's my point of view from what I've been learning about DDD. Maybe I'm wrong.

In response to jantonioreyes, I wanted to say that in DDD, infrastructure depends on both application layer and domain model. In order to implement cross-cutting concerns, the interface lives in the application layer and the implementation lives in the infrastructure.

Regards, Juan.

hgraca says:

January 16, 2018 at 07:47

That's fair enough 😊

However, if you place repository interfaces in the domain layer, that means that an Entity could eventually use a repository. I think that should never be the case. So by putting the repository interface in the application layer, you explicitly say that a domain object should never use a repository.

Collections, on the other hand, yes. We can have a User and a Role entities in a 1-* relation, and then the User will contain a collection of Roles, and that collection can contain business rules inside.

I have never encountered a case where a domain object (entity, value obj, collection, specification, ...) would need to know about persistence.

A use case, on the other hand, needs to know to what entities it applies to, needs to get them from persistence, tell them to do something, and persist them again.

Furthermore, I don't think the repository implementation should live in the infrastructure because a repository is specific to a component, it should not be used in other components. A repository is not cross cutting through components, unlike the persistence interface which will be used by all components.

This is how I see it anyway 😊

jantonioreyes says:

January 16, 2018 at 10:58

Hi,

I agree that entities should never use a repository, but not entirely sure about its use in Domain Services, f.e. when applying business rules regarding some type of data validation (stored in database) before a command execution.

However, if you place repository in Application layer, the use of repositories is restricted in Domain, but they could be used from Presentation, as it can access to all Application functionality. And in a DDD context, I don't think this is really correct, as any access from UI should be through Application Services or Command/Queries Handlers.

Anyway, I don't agree with @Juan about that infrastructure depends on both application layer and domain model in DDD. From my point of view, dependencies in DDD do in this way:

Application → Domain ← Infrastructure

Where domain is the core of everything.

It'll be interesting to know your opinions about all the things discussed here.

Regards.

🗨 Reply

Juan says:

January 16, 2018 at 12:31

Hi @jantonioreyes. Infrastructure must depend on Application Layer in order to implement the application logic that needs to deal with technology or frameworks.

For example, transactions are a cross-cutting concern that should live in the application layer, and the implementation (with Spring framework for example) should live in the infrastructure.

Sending a notification is another example of application logic. The interface lives in the application layer and the implementation (using email, sms or whatever) lives in the infrastructure.

Look at the layered architecture diagram using DIP (page 124 in the book "Implementing DDD" by Vaughn Vernon). Infrastructure depends on all layers.

I don't use layered architecture though. I use hexagonal architecture, but it's similar: application logic belongs to the inside of the hexagon, and it can use ports (interfaces that belongs to the hexagon too). This ports are implemented by adapters (the infrastructure, the outside of the hexagon).

Regards, Juan.

🗨 Reply

[Blog at WordPress.com.](#)

