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

[hgraca](https://herbertograca.com/author/hgraca/" \o "Posts by hgraca)  [Architecture](https://herbertograca.com/category/development/architecture/), [Development](https://herbertograca.com/category/development/), [Series](https://herbertograca.com/category/development/series/), [The Software Architecture Chronicles](https://herbertograca.com/category/development/series/the-software-architecture-chronicles/), [Uncategorized](https://herbertograca.com/category/uncategorized/) November 16, 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 about 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.

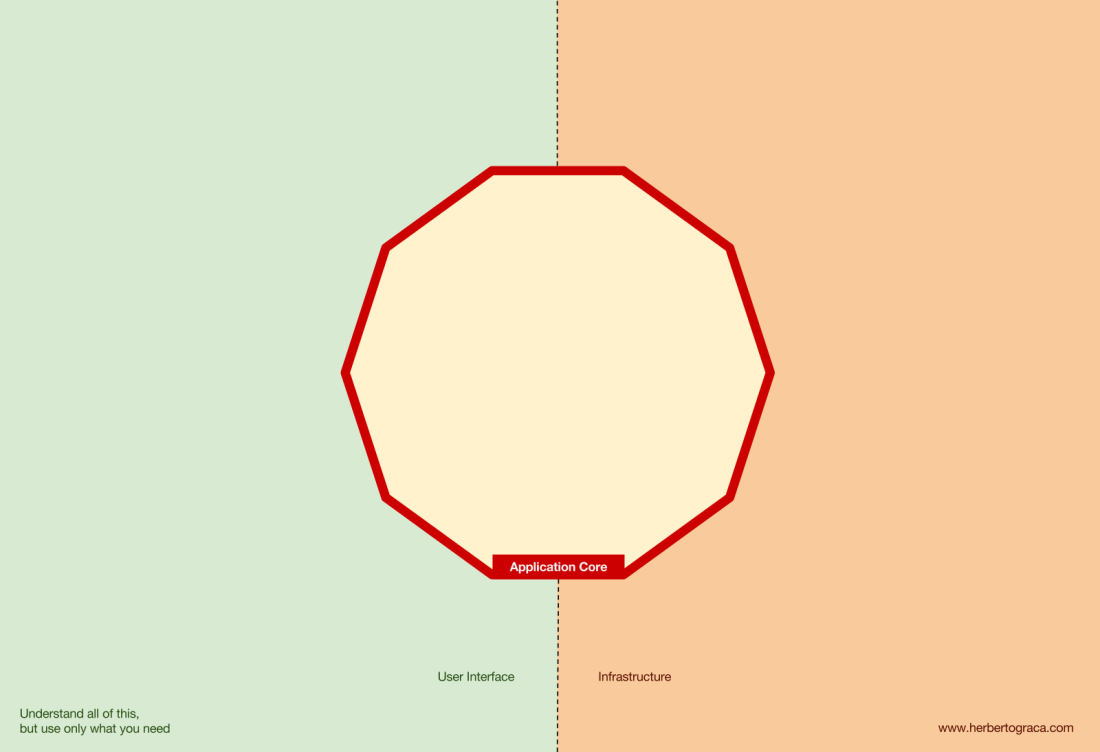
* [Fundamental blocks of the system](https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/#fundamental-blocks-of-the-system)
* [Tools](https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/#tools)
* [Connecting the tools and delivery mechanisms to the Application Core](https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/#connecting-the-tools-and-the-application-core)
  + [Ports](https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/#ports)
  + [Primary or Driving Adapters](https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/#primary-or-driving-adapters)
  + [Secondary or Driven Adapters](https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/#secondary-or-driving-adapters)
  + [Inversion of control](https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/#inversion-of-control)
* [Application Core Organisation](https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/#application-core-organisation)
  + [Application Layer](https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/#application-layer)
  + [Domain Layer](https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/#domain-layer)
    - [Domain Services](https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/#domain-services)
    - [Domain Model](https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/#domain-model)
* [Components](https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/#components)
  + [Decoupling the components](https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/#decoupling-the-components)
    - [Triggering logic in other components](https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/#triggering-logic-in-other-components)
    - [Getting data from other components](https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/#getting-data-from-other-components)
      * [Data storage shared between components](https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/#data-storage-shared-between-components)
      * [Data storage segregated per component](https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/#data-storage-segregated-per-component)
* [Flow of control](https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/#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.

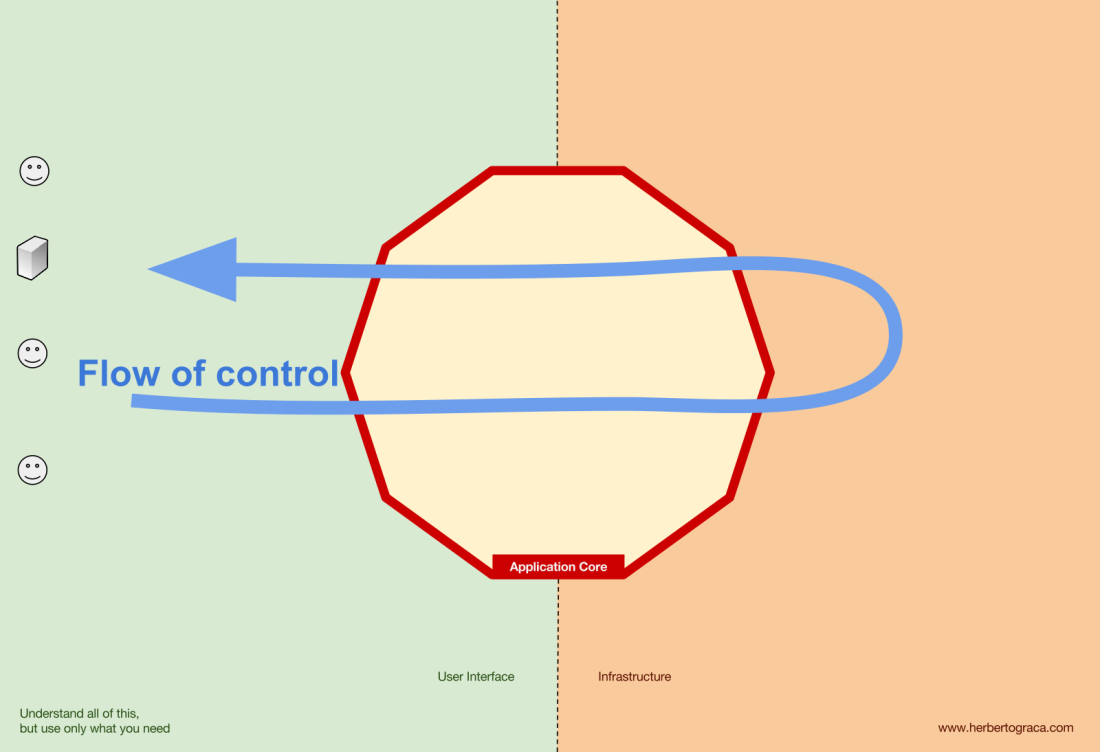
Furthermore, [**Ports & Adapters**](https://herbertograca.com/2017/09/14/ports-adapters-architecture/) architecture explicitly identifies three fundamental blocks of code in a system:

* 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.



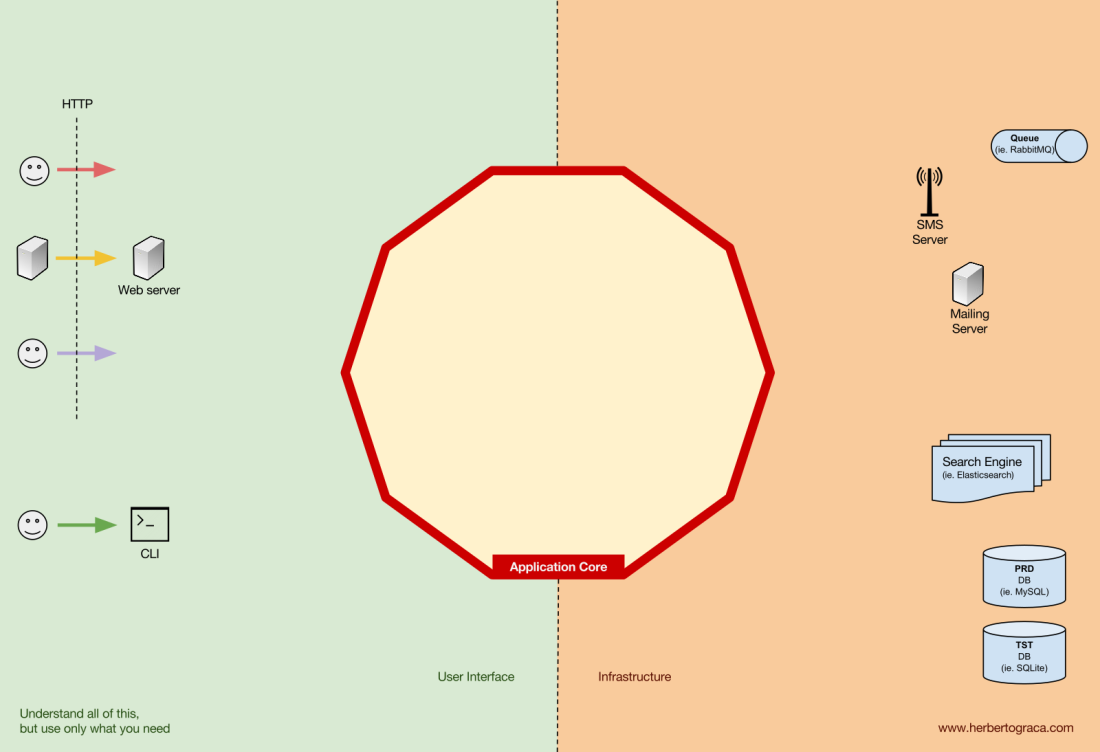
The application core is what we should really care about. It is the code that allows our code to do what it is supposed to do, it IS our application. It might use several user interfaces (progressive web app, mobile, CLI, API, …) but the code actually doing the work is the same and is located in the 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**.

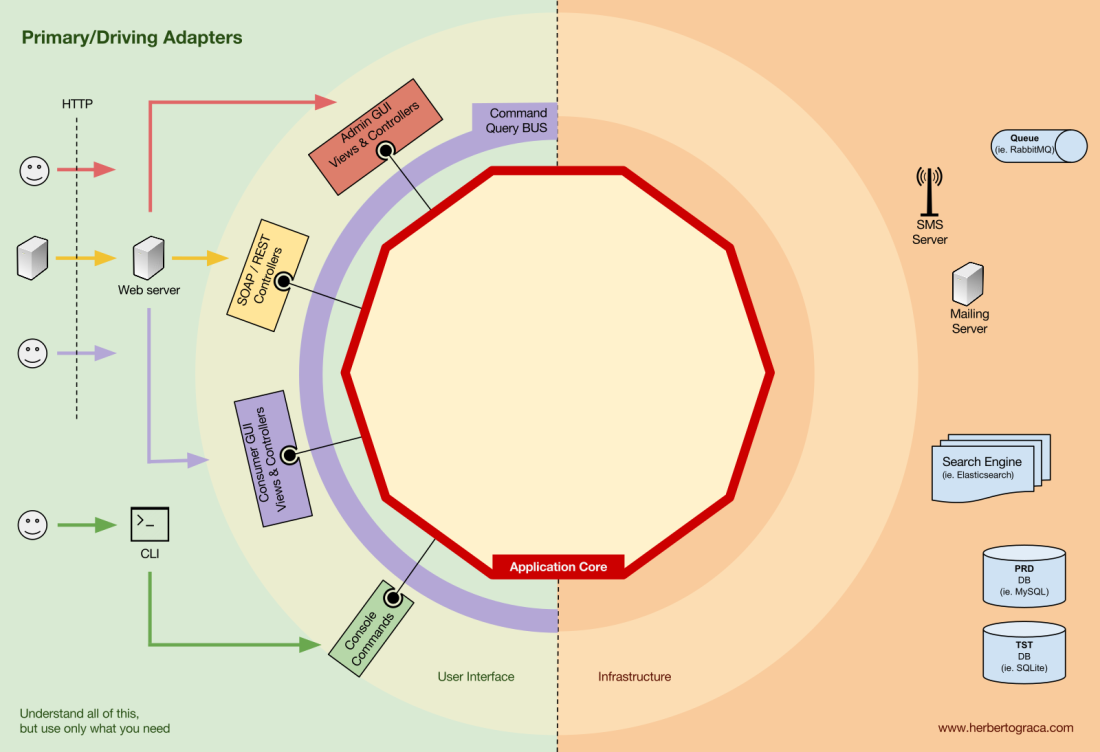
**Ports**

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.**



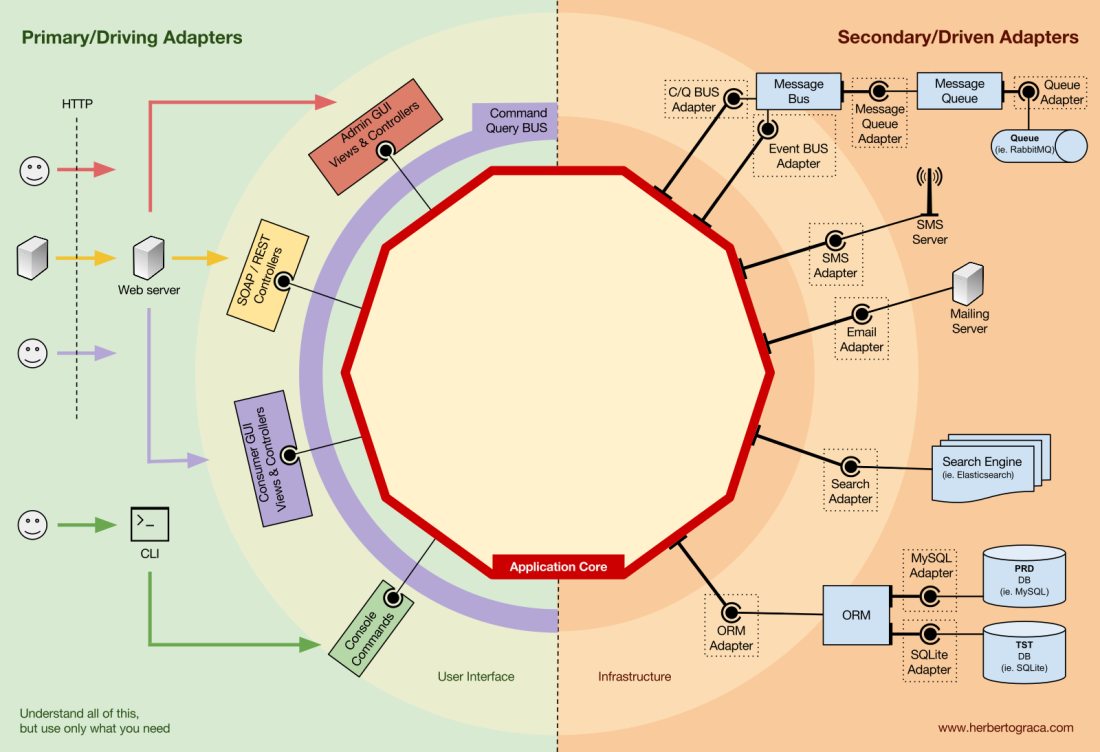
In other words, our Driving Adapters are Controllers or Console Commands who are injected in their constructor with some object whose class implements the interface (Port) that the controller or console command requires.

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).



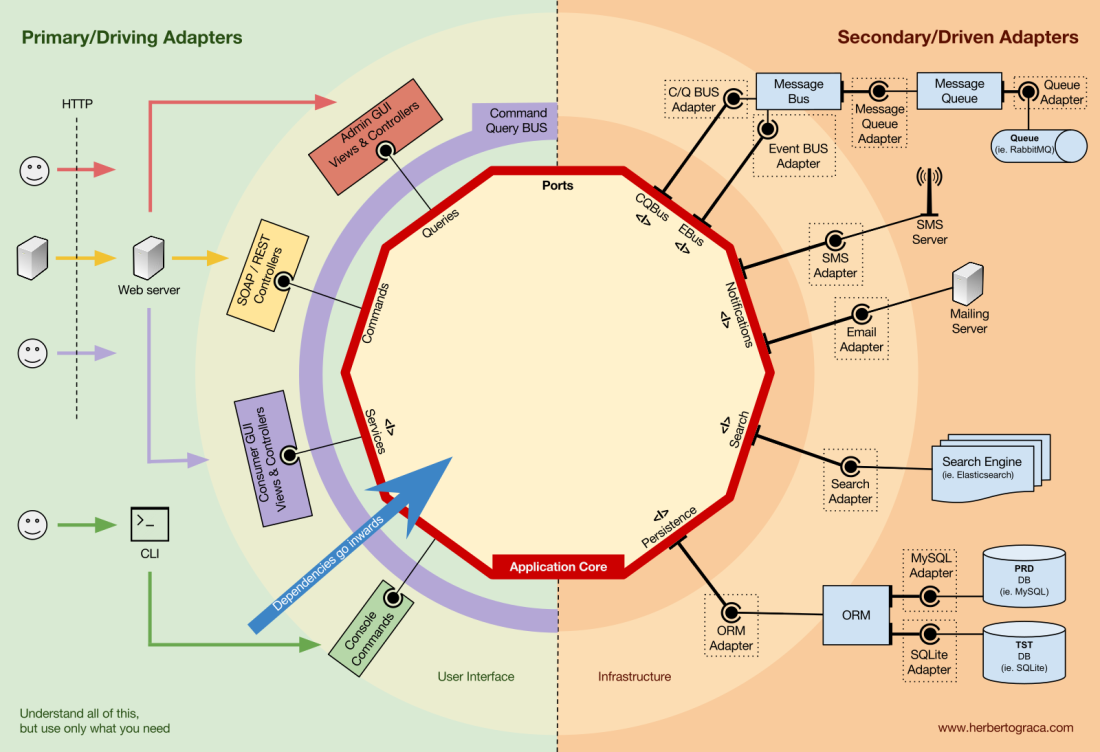
For example, let’s suppose that we have a naive application which needs to persist data. So we create a persistence interface that meets its needs, with a method to *save* an array of data and a method to *delete* a line in a table by its ID. From then on, wherever our application needs to save or delete 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 means the direction of dependencies is towards the centre, it’s the **inversion of control principle at the architectural level**.

Although, again, **it is of utmost importance that the Ports are created to fit the Application Core needs and not simply mimic the tools APIs.**

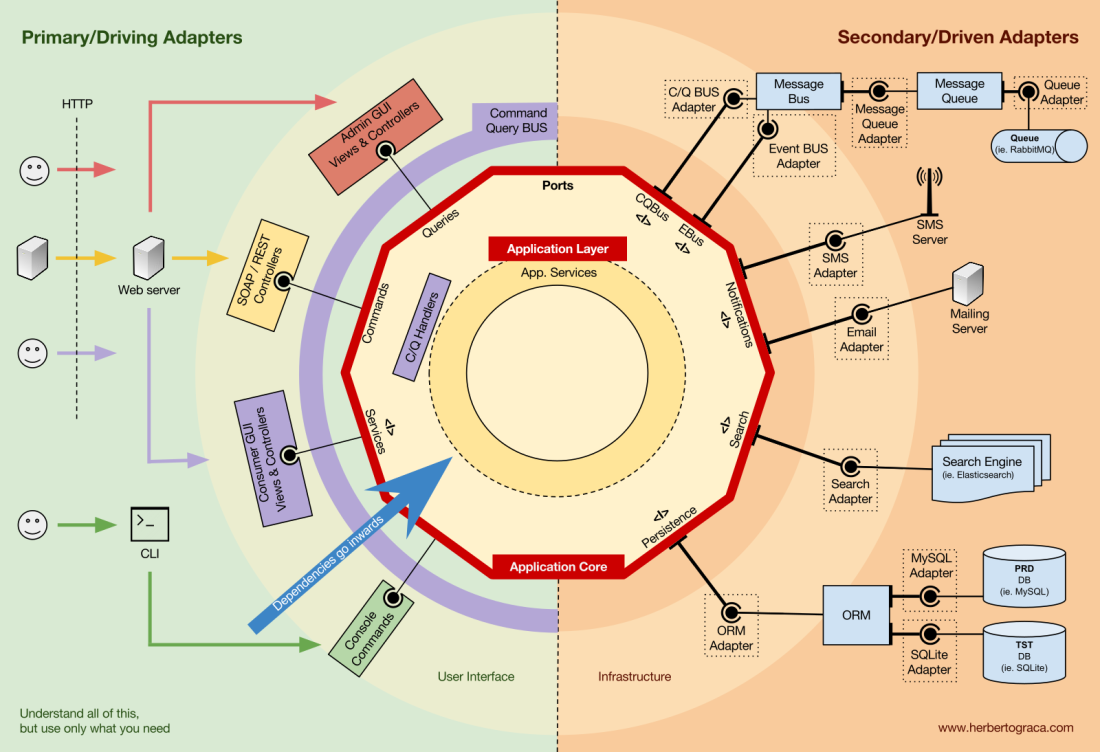
**Application Core organisation**

The [Onion Architecture](https://herbertograca.com/2017/09/21/onion-architecture/) picks up the DDD layers and incorporates them into the [Ports & Adapters Architecture](https://herbertograca.com/2017/09/14/ports-adapters-architecture/). Those layers are intended to bring some organisation to the business logic, the interior of the Ports & Adapters “hexagon”, and just like in Ports & Adapters, the dependencies direction is towards the centre.

**Application Layer**

The use cases are the processes that can be triggered in our Application Core by one or several User Interfaces in our application. For example, in a CMS we could have the actual application UI used by the common users, another independent UI for the CMS administrators, another CLI UI, and a web API. These UIs (applications) could trigger use cases that can be specific to one of them or reused by several of them.

The use cases are defined in the Application Layer, the first layer provided by DDD and used by the Onion Architecture.



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.

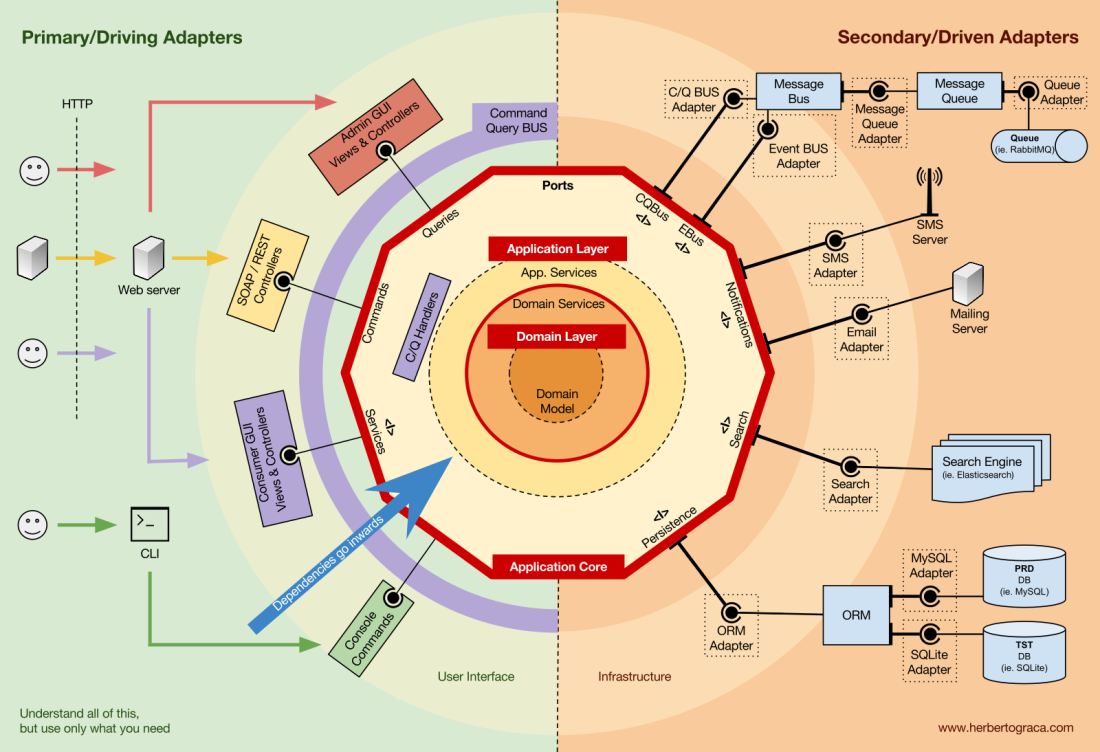
Which approach to use depends on the context, for example:

* 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.

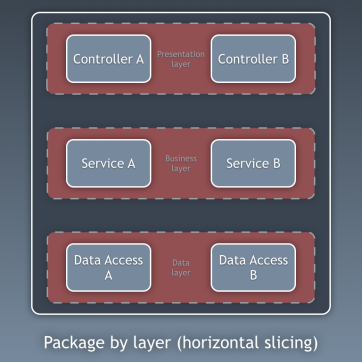
**Domain Model**

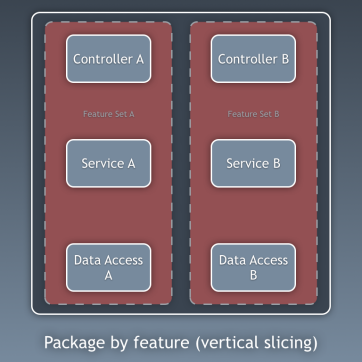
In the very centre, depending on nothing outside it, is the Domain Model, which contains the business objects that represent something in the domain. Examples of these objects are, first of all, Entities but also Value Objects, Enums and any objects used in the Domain Model.

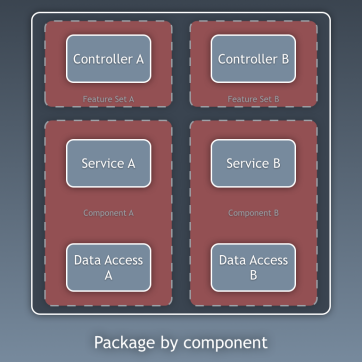
The Domain Model is also where Domain Events “live”. These events are triggered when a specific set of data changes and they carry those changes with them. In other words, when an entity changes, a Domain Event is triggered and it carries the changed properties new values. These events are perfect, for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

**Components**

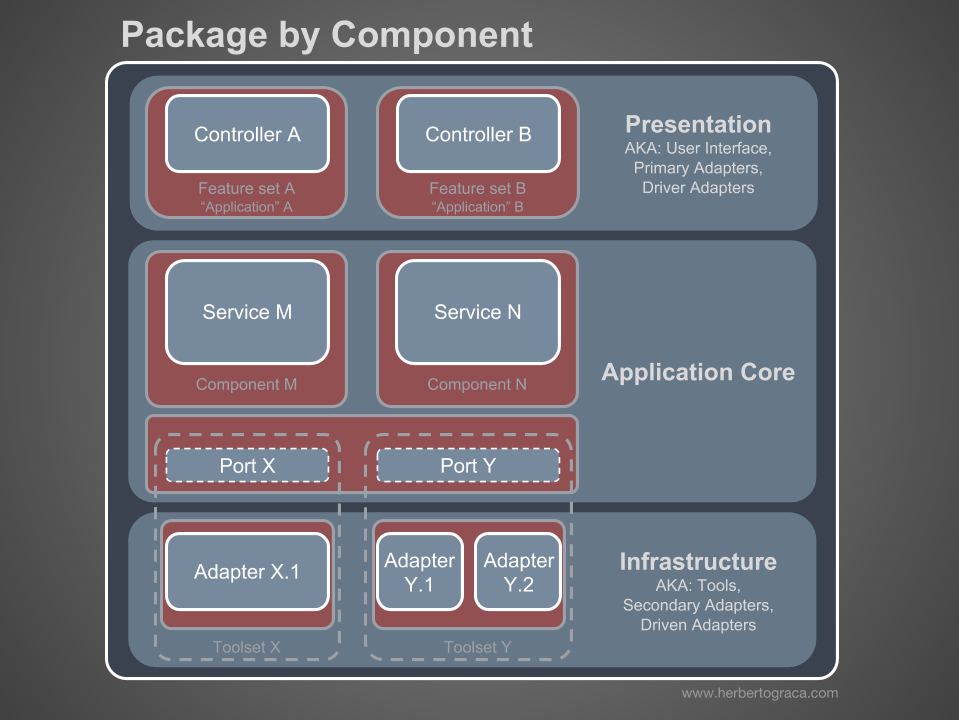
So far we have been segregating the code based on layers, but that is the fine-grained code segregation. The coarse-grained segregation of code is at least as important and it’s about segregating the code according to sub-domains and [***bounded contexts***](http://ddd.fed.wiki.org/view/welcome-visitors/view/domain-driven-design/view/bounded-context), 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-by-layer.png)

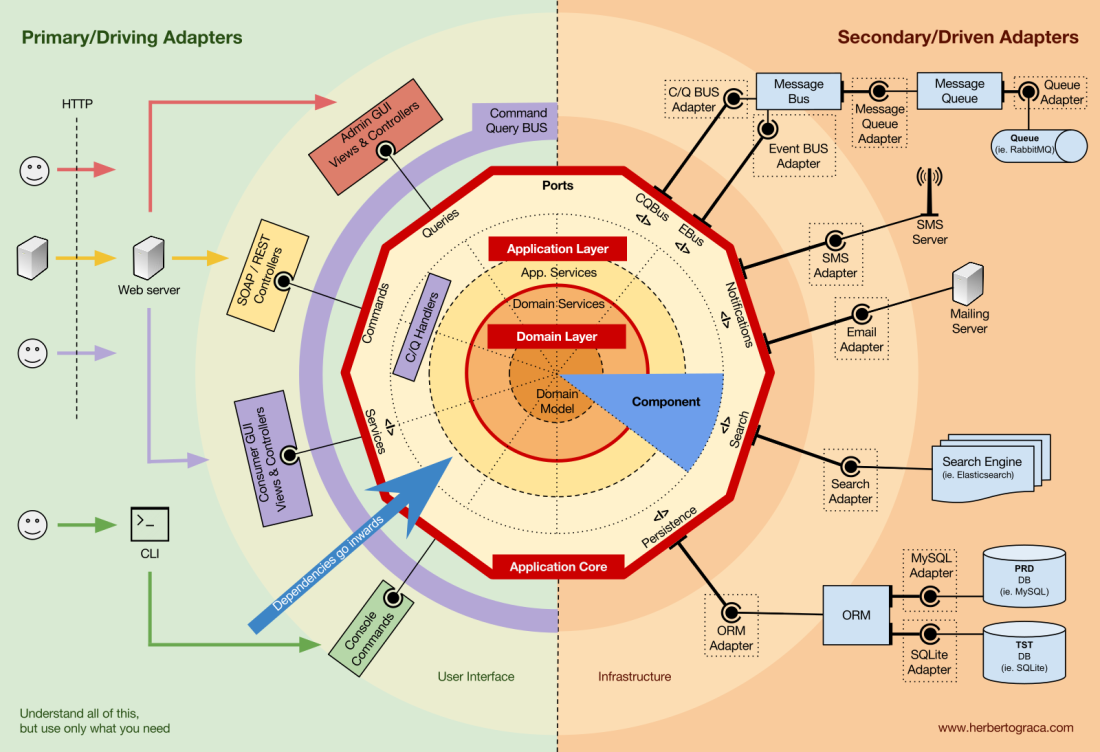
[](https://herbertograca.files.wordpress.com/2017/11/20150308-package-by-feature.png)

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



hese 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  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.

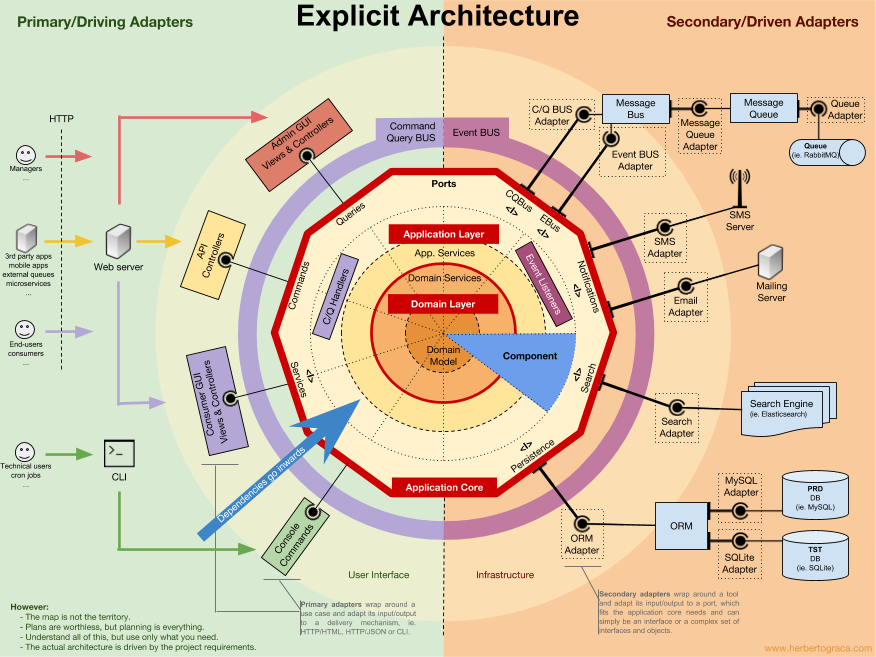


**Decoupling the components**

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!

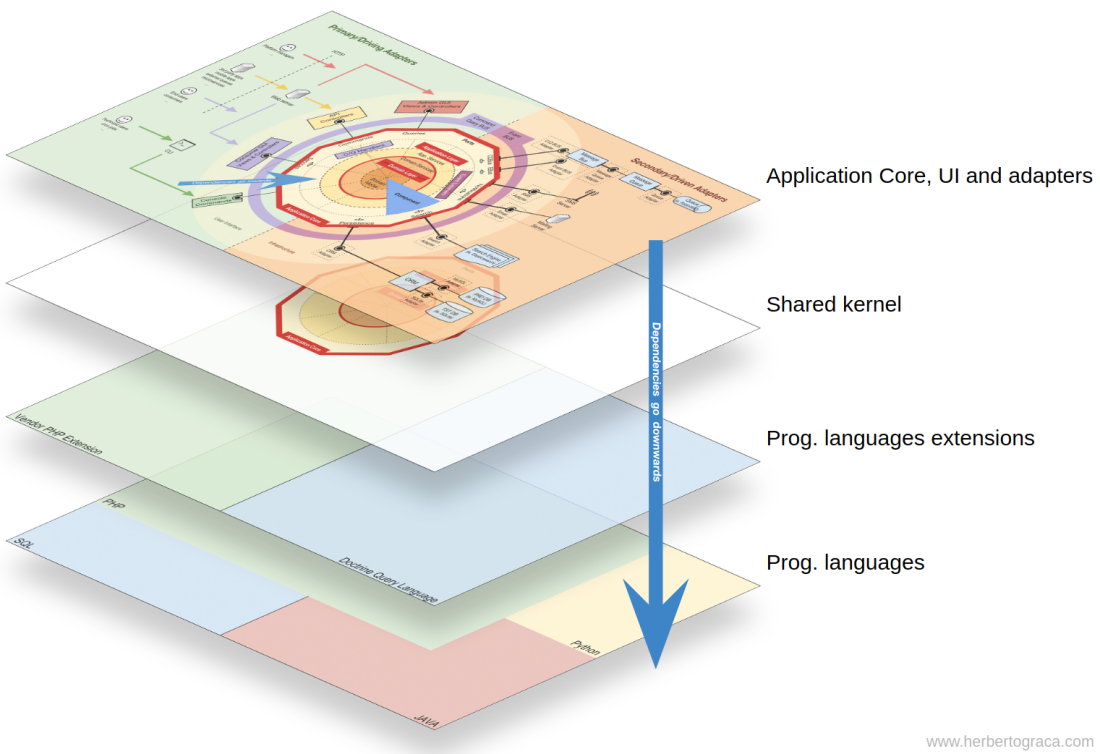
[](https://drive.google.com/open?id=1E_hx5B4czRVFVhGJbrbPDlb_JFxJC8fYB86OMzZuAhg)

**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. Read more about this in my followup post: [More than concentric layers](https://herbertograca.com/2018/07/07/more-than-concentric-layers/).



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**

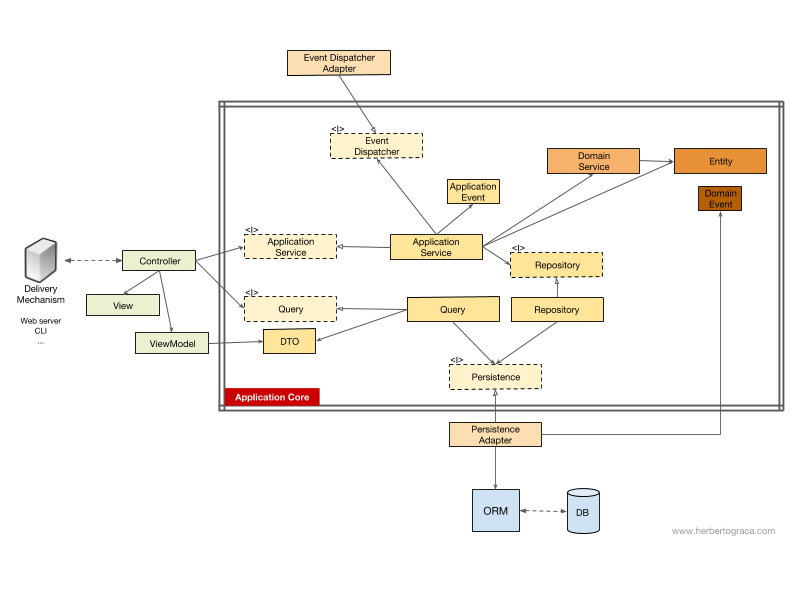
As I said above, the flow of control goes, of course, from the user into the Application Core, over to the infrastructure tools, back to the Application Core and finally back to the user. But how exactly do classes fit together? Which ones depend on which ones? How do we compose them?

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

**Without a Command/Query Bus**

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. Tkx to [MorphineAdministered](https://www.reddit.com/user/MorphineAdministered" \t "_blank) 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. ThisViewModel 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:

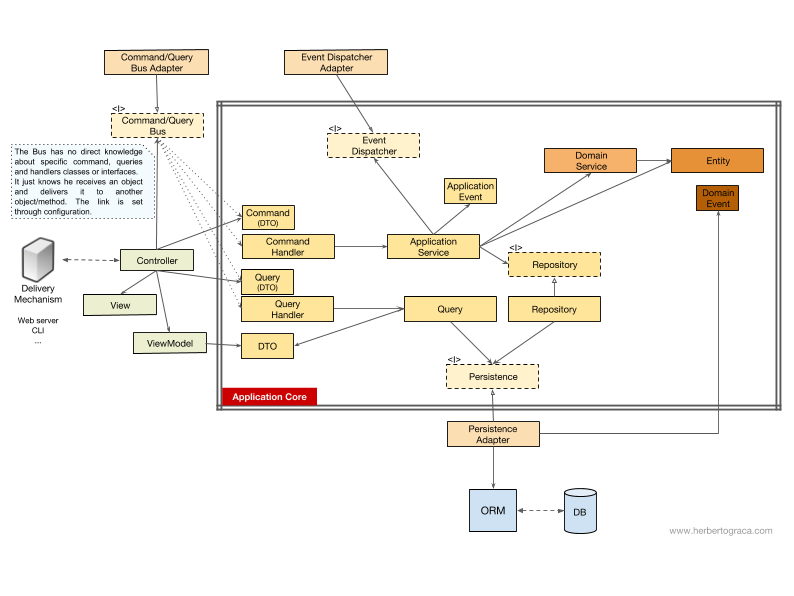
* The persistence interface is an abstraction layer over the ORM so we can swap the ORM being used with no changes to the Application Core.
* The repository interface is an abstraction on the persistence engine itself. Let’s say we want to switch from MySQL to MongoDB. The persistence interface can be 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.

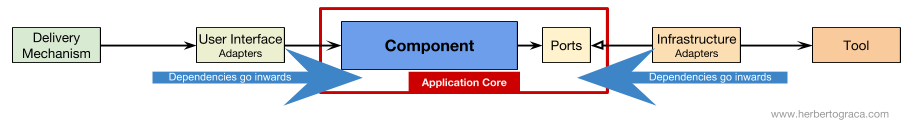
In the diagram below, the Command Handler then uses an Application Service. However, that is not always needed, in fact in most of the cases the handler will contain all the logic of the use case. We only need to extract logic from the handler into a separated Application Service if we need to reuse 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. Tkx to [MorphineAdministered](https://www.reddit.com/user/MorphineAdministered" \t "_blank) 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 of 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.

I expanded these ideas a bit more on a followup post: [More than concentric layers](https://herbertograca.com/2018/07/07/more-than-concentric-layers/).

However, how do we make all this explicit in the code base? That’s the subject of one of my next posts: how to reflect the architecture and domain, in the code.

Last but not least, thanks to my colleague [Francesco Mastrogiacomo](https://www.linkedin.com/in/francescomastrogiacomo/), for helping me make my infographic look nice.

**Translations**

* [Chinese](https://www.jianshu.com/p/d3e8b9ac097b), by Qinyusuain
* [Japanese](https://tagoto.atlassian.net/wiki/spaces/public/blog/2018/02/18/37061127/DDD+Hexagonal+Onion+Clean+CQRS), by Tagoto
* [Russian](http://chepa.net/all/2018/10/25/%D0%BF%D0%B5%D1%80%D0%B5%D0%B2%D0%BE%D0%B4-ddd-hexagonal-onion-clean-cqrs-%D0%BA%D0%B0%D0%BA-%D1%8F-%D1%81%D0%BE%D0%B1%D1%80%D0%B0%D0%BB-%D0%B2%D1%81%D1%91-%D1%8D%D1%82%D0%BE-%D0%B2/)
* [Vietnamese](https://edwardthienhoang.wordpress.com/2018/08/24/ket-hop-cac-mau-kien-truc-pattern-vao-trong-mot-ddd-hexagonal-onion-clean-cqrs/), by Edward Hien Hoang

**17.清晰架构(01): 融合 DDD、洋葱架构、整洁架构、CQRS...(译)**

[](https://www.jianshu.com/u/oSuHPw)

[qinyu](https://www.jianshu.com/u/oSuHPw) 关注

2018.08.27 16:52\* 字数 6849 阅读 1289评论 0喜欢 27

原文：[https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/](https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/" \t "_blank)

*这篇文章是[软件架构编年史](https://herbertograca.com/2017/07/03/the-software-architecture-chronicles/" \t "_blank)(*[*译*](https://www.jianshu.com/p/b477b2cc6cfa)*)的一部分，这部编年史由[一系列关于软件架构的文章](https://herbertograca.com/category/development/series/software-architecture/" \t "_blank)(*[*译*](https://www.jianshu.com/nb/28913609)*)组成。在这一系列文章中，我将写下我对软件架构的学习和思考，以及我是如何运用这些知识的。如果你阅读了这个系列中之前的文章，本篇文章的的内容将更有意义。*

大学毕业之后我做了一名高中老师，直到几年前我决定成为一名全职软件开发者。

从那时起，我时常觉得我必须找回“失去的”时间，尽力地多学一点，学快一点。所以我变得有一点沉迷于实验、阅读和写作，特别关注的就是软件设计和架构。帮助我学习正是我写下这些文章的初衷。

在之前的文章中，我记录了许多学到的概念和原则和我的一些思考。但我知道这只是管中窥豹。

今天的文章内容是我如何将这些碎片融合成一种新的架构，看起来我得给它起个名字，就叫做**清晰架构**吧。而且，这些概念全都经历过“炮火的洗礼”，在高水准的平台生产代码中得到了应用。其中一个是拥有数千家遍布全球的网上商店的 SaaS 电子商务平台，另一个是已经在两个国家上线的市场，它拥有可以每月处理超过两千万条消息的消息总线。

**系统的基本构建块**

我们从 [EBI 架构](https://herbertograca.com/2017/08/24/ebi-architecture/)以及[端口和适配器架构](https://www.jianshu.com/p/f39f4537857e)([译](https://www.jianshu.com/p/f39f4537857e))的回顾开始。它们都有清晰的代码划分，哪些代码在应用内部，哪些代码在外部，而哪些代码用来连接它们。

除此之外，[端口和适配器架构](https://www.jianshu.com/p/f39f4537857e)还明确地识别出了一个系统中的三个基本代码构建块：

* 运行**用户界面**所需的构建块，无论是哪种用户界面；
* 系统的**业务逻辑**，或者**应用核心**，用户界面要使用这个构建块达成目的；
* **基础设施**代码，这个构建块将我们的应用核心和诸如数据库、搜索引擎或第三方 API 这样的工具连接起来。

我们真正关心的应该是应用核心。这部分代码才是我们编写代码的目的，它们才是我们的应用。它们可能使用一些不同的用户界面(渐进式 Web 应用，移动应用、命令行、接口...)，但完成实际工作的代码是一模一样的，它们就在应用核心内部，它们不用关心是哪种用户界面触发了它们。

你可以想像得到，典型的应用控制流开始于用户界面中的代码，经过应用核心到达基础设施代码，又返回应用核心，最后将响应传达给用户界面。

**工具**

在远离我们系统中最重要的代码-——应用核心-——的地方，还有一些应用会用到的工具，例如数据库引擎、搜索引擎、Web 服务器或者命令行控制台(虽然最后两种工具也是传达机制)。

把命令行控制台和数据库引擎放在同一个“篮子”中感觉有点奇怪，尽管它们有着不用的用途，但它们实际都是应用使用的工具。关键的区别在于，命令行控制台和 Web 服务器***告诉我们*的应用它要做什么**，而数据库引擎是**由我们的应用来*告诉它*做什么**。这是针锋相对的差别，强烈地暗示着我们应该如何构建连接这些工具和应用核心的代码。

**将传达机制和工具连接到应用核心**

连接工具和应用核心的代码单元被称为适配器([端口和适配器架构](https://www.jianshu.com/p/f39f4537857e))。适配器有效地实现了让业务逻辑和特定工具之间可以相互通信的代码。

**告知我们**的应用应该做什么的适配器被称为**主适配器**或**主动适配器**，而那些由我们的应用**告知它**该做什么的适配器被称为**从适配器**或者**被动适配器**。

**端口**

而这些适配器并非是随意创建的。它们需要按照应用核心某个特定的入口的要求来创建，即**端口**。端口无外乎是一份工具如何使用应用核心或者如何被应用核心使用的**说明书**。这份说明书，即端口，在大多数语言里最简单的形式就是接口，但实际上也可能由多个接口和 DTO 组成。  
　  
**端口(接口)位于业务逻辑内部**，而适配器位于其外部，这一点要特别注意。要让这种模式按照设想发挥作用，端口按照应用核心的需要来设计而不是简单地套用工具的 API，这一点再怎么强调都不为过。

**主适配器或主动适配器**

**主适配器**或**主动适配器*包装*端口**并通过它告知应用核心应该做什么。**它们将来自传达机制的信息转换成对应用核心的方法调用**。

换句话说，我们的主动适配器就是 Controller 或者控制台命令，它们需要的接口(端口)由其他类实现，这些类的对象通过构造方法注入到 Controller 或者控制台命令。

再举一个更具体的例子，端口就是 Controller 需要的 Service 接口或者 Repository 接口。Service、Repository 或 Query 的具体实现被注入到 Controller 供 Controller 使用。

此外，端口还可以是命令总线接口或者查询总线接口。这种情况下，命令总线或者查询总线的具体实现将被注入到 Controller 中， Controller 将创建命令或查询并传递给相应的总线。

**从适配器或被动适配器**

和主动适配器包装端口不同，**被动适配器*实现*一个端口(接口)**并被注入到需要这个端口的应用核心里。

举个例子，假设有一个需要存储数据的简单应用。我们创建了一个符合应用要求的持久化接口，这个接口有一个*保存*数据数组的方法和一个根据 ID 从表中*删除*一行的方法。接口创建好之后，无论何时应用需要保存或删除数据，都应该使用实现了这个持久化接口的对象，而这个对象是通过构造方法注入的。

现在我们创建了一个专门针对 MySQL 实现了该接口的适配器。它拥有保存数组和删除表中一行数据的方法，然后在需要使用持久化接口的地方注入它。

如果未来我们决定更换数据库供应商，比如换成 PostgreSQL 或者 MongoDB，我们只用创建一个专门针对 PostgreSQL 实现了该接口的适配器，在注入时用新适配器代替旧适配器。

**控制反转**

这种模式有一个特征值得留意，适配器依赖特定的工具和特定的端口(它需要提供接口的特定实现)。但业务逻辑只依赖按照它的需求设计的端口(接口)，它并不依赖特定的适配器或工具。

这意味着依赖的方向是由外向内的，这就是**架构层面的控制反转原则**。

再一次强调，**端口按照应用核心的需要来设计而不是简单地套用工具的 API**。

**组织应用核心的结构**

[洋葱架构](https://www.jianshu.com/p/d87d5389c92a)采用了 DDD 的分层，将它们融合进了端口和适配器架构。这种分层想要为位于[端口和适配器架构](https://www.jianshu.com/p/f39f4537857e" \t "_blank)“六边形”内的业务逻辑带来一种结构组织，和[端口与适配器架构](https://www.jianshu.com/p/f39f4537857e" \t "_blank)一样，依赖的方向也是由外向内。

**应用层**

在应用中，由一个或多个用户界面触发的应用核心中的过程就是用例。例如，在一个 CMS 系统中，我们可以提供普通用户使用的应用 UI、CMS 管理员使用的独立的 UI、命令行 UI 以及 Web API。这些 UI(应用)可以触发的用例可能是专门为它设计的，也可以是多个 UI 复用的。

用例定义在应用层中，这是 DDD 提供的第一个被[洋葱架构](https://www.jianshu.com/p/d87d5389c92a" \t "_blank)使用的层。

这个层包括了作为一等公民的应用服务(以及它们的接口)，也包括了[端口与适配器架构](https://www.jianshu.com/p/f39f4537857e" \t "_blank)中的接口，例如 ORM 接口、搜索引擎接口、消息接口等等。如果我们使用了命令总线和/或查询总线，命令和查询分别对应的处理程序也属于这一层。

应用服务和/或命令处理程序包含了展现一个用例，一个业务过程的逻辑。通常，它们的作用是：

1. 使用 Repostitory 查找一个或多个实体；
2. 让这些实体执行一些领域逻辑；
3. 再次使用 Repostitory 让这些实体持久化，有效地保存数据变化。

命令处理程序有两种不同使用方式：

1. 它们可以包含执行用例的实际逻辑；
2. 它们可以仅仅作为我们应用中的连接片段，接收命令然后简单地触发应用服务中的逻辑。

使用哪种方式是由上下文决定的，例如：

* 我们已经有了合适的应用服务，现在要做的是添加命令总线？
* 命令总线允许指定任意类/方法作为处理程序吗？还是说它们需要扩展已有的类或者实现已有的接口？

应用层还包括**应用事件**的触发，这也代表着某些用例的产出。这些事件触发的逻辑是用例的副作用，比如发送邮件、通知第三方 PAI、发送推送通知，或是发起属于其他应用组件的另一个用例。

**领域层**

继续向内一层就是领域层。这一层中的对象包含了数据和操作数据的逻辑，它们只和领域本身有关，独立于调用这些逻辑的业务过程。它们完全独立，对应用层完全无感知。

**领域服务**

如前所述，应用服务的作用是：

1. 使用 Repostitory 查找一个或多个实体；
2. 让这些实体执行一些领域逻辑；
3. 再次使用 Repostitory 让这些实体持久化，有效地保存数据变化。

然而，有时我们还会碰到某种领域逻辑，它涉及不同的实体。这些实体也许是同一个类型，也许不是，而且我们觉得这种领域领域逻辑并不属于这些实体，这种逻辑不是这些实体的直接责任。

所以，我们的第一反应也许是把这些逻辑放到实体外的应用服务中。然而，这意味着这些领域逻辑就不能被其它的用例复用。领域逻辑应该放在应用层之外！

解决方法是创建领域服务，它的作用是接收一组实体并对它们执行某种业务逻辑。领域服务属于领域层，因此它并不了解应用层中的类，比如应用服务或者 Repository[译注：Repository 属于应用服务层？？]。另一方面，它可以使用其他领域服务，当然还可以使用领域模型对象。

**领域模型**

在架构的正中心，是完全不依赖外部任何层次的领域模型。它包含了那些表示领域中某个概念的业务对象。这些对象的例子首先就是实体，还有值对象、枚举以及其它领域模型种用到的任何对象。

领域事件也“活在”领域模型中。当一组特定的数据发生变化时就会触发这些事件，而这些时间会携带这些变化的信息。换句话说，当实体变化时，就会触发一个领域事件，它携带着发生变化的属性的新值。这些事件可以完美地应用于事件溯源。

**组件**

目前为止，我们都是使用层次来划分代码，但这是细粒度的代码隔离。根据 Robert C. Martin 在[尖叫架构](https://8thlight.com/blog/uncle-bob/2011/09/30/Screaming-Architecture.html" \t "_blank)中表达的观点，按照子域和[限界上下文](http://ddd.fed.wiki.org/view/welcome-visitors/view/domain-driven-design/view/bounded-context)对代码进行划分这种粗粒度的代码隔离同样重要。这通常被叫做“*按特性分包*”或者“*按组件分包*”，和“*按层次分包*”相呼应。Simon Brown 的文章“[Package by component and architecturally-aligned testing](http://www.codingthearchitecture.com/2015/03/08/package_by_component_and_architecturally_aligned_testing.html)”很好地阐述了这种划分：

|  |  |  |
| --- | --- | --- |

我是“*按组件分包*”方式的坚定拥护者，在此我厚着脸皮将 Simon Brown *按组件分包*的示意图做了如下修改：

这些代码块在前面描述的分层基础上再进行了“横切”，它们是应用的[组件](https://herbertograca.com/2017/07/05/software-architecture-premises/" \t "_blank)([译](https://www.jianshu.com/p/df295f92fb52))。组件的例子包括、、账单、用户、评论或帐号，而它们总是都和领域相关。像认证和/或授权这样的限界上下文应该被看作外部工具，我们应该为它们创建适配器，把它们隐藏在某个端口之后。

**组件解耦**

和细粒度的代码单元(类、接口、特质、混合等等)一样，粗粒度的代码单元(组件)也会从高内聚低耦合中受益。

我们使用了依赖注入，**通过将依赖注入类而不是在类内部初始化依赖**；以及依赖倒置；**让类依赖抽象(接口和/或抽象类)而不是具体类**来解耦类。这意味着类不用知道它要使用的具体类的任何信息，不用引用所依赖的类的完全限定类名。

以同样的方式完全解耦的组件意味着组件不会直接了解其它任何组件的信息。换句话说，它不会引用任何来自其它组件的细粒度的代码单元，甚至都不会引用接口！这意味着依赖注入和依赖倒置对组件解耦是不够用的，我们还需要一些架构层级的结构。我们需要事件、共享内核、最终一致性甚至发现服务！

**触发其它组件的逻辑**

当一个组件(组件 A)中有事情发生需要另一个组件(组件B)做些什么时，我们不能简单地从组件 A 直接调用组件 B 中的类/方法，因为这样 A 就和 B 耦合在一起了。

但是我们可以让 A 使用事件派发器，派发一个领域事件，这个事件将会投递给任何监听它的组件，例如 B，然后 B 的事件监听器会触发期望的操作。这意味着组件 A 将依赖事件派发器，但和 B 解耦了。

然而，如果事件本身“活在” A 中，这将意味着 B 知道了 A 的存在，就和 A 存在耦合。要去掉这个依赖，我们可以创建一个包含应用核心功能的库，由所有组件共享，这就是[共享内核](http://ddd.fed.wiki.org/view/welcome-visitors/view/domain-driven-design/view/shared-kernel" \t "_blank)。这意味着两个组件都依赖共享内核，而它们之间却没有耦合。共享内核包含了应用事件和领域事件这样的功能，而且还包含规格对象，以及其它任何有理由共享的东西。记住共享内核的范围应该尽可能的小，因为它的任何变化都会影响所有应用组件。而且，如果我们的系统是语言异构的，比如使用不同语言编写的微服务生态，共享内核需要做到与语言无关的，这样它才能被所有组件理解，无论它们是用哪种语言编写的。例如，共享内核应该包含像 JSON 这样无关语言的事件描述(例如，名称、属性，也许还有方法，尽管它们对规格对象来说更有意义)而不是事件类，这样所有组件/微服务都可以解析它，还可以自动生成各自的具体实现。请在我的下一篇文章种了解更多内容：[超越同心圆分层](https://herbertograca.com/2018/07/07/more-than-concentric-layers/" \t "_blank)([译](https://www.jianshu.com/p/fcf5bb27a60b))

这种方法既适用于单体应用，也适用于像微服务生态系统这样的分布式应用。然而，这种方法只适用于事件异步投递的情况，在需要即时完成触发其它组件逻辑的上下文中并不适用！组件 A 将需要向组件 B 发起直接的 HTTP 调用。这种情况下，要解耦组件，我们需要一个发现服务，A 可以询问它得知请求应该发送到哪里才能触发期望的操作，又或是向发现服务发起请求并由发现服务将请求代理给相关服务并最终返回响应给请求方。这种方法会把组件和发现服务耦合在一起，但会让组件之间解耦。

**从其它组件获得数据**

我的看法是，组件不允许修改不“属于”它的数据，但可以查询和使用任何数据。

**组件之间共享的数据存储**

当一个组件需要使用属于其它组件的数据时，比如说账单组件需要使用属于账户组件的客户名字，账单组件会包含一个查询对象，可以在数据存储中查询该数据。简单的说就是账单组件知道任何数据集，但它只能通过查询只读地使用不“属于”它的数据。

**按组件隔离的数据存储**

这种情况下，这种模式同样有效，但数据存储层面的复杂度更高。

组件拥有各自的数据存储意味着每个数据存储都包含：

* 一组属于它的数据，并且只允许它自己修改这些数据，让它成为单一事实来源；
* 一组其它组件数据的副本，它自己不能修改这些数据，但组件的功能需要这些数据，而且一旦数据在其所属的组件中发生了变化，这些副本需要更新。

每个组件都会创建其所需的其它组件数据的本地副本，在必要时使用。当数据在其所属的组件中发生了变化，该组件将触发一个携带数据变更的领域事件。拥有这些数据副本的组件将监听这个领域事件并相应地更新它们的本地副本。

**控制流**

如前所述，控制流显然从用户出发，进入应用核心，抵达基础设施工具，再返回应用核心并最终返回给用户。但这些类到底是是如何配合的？哪些类依赖哪些类？我们怎样把它们组合在一起？

根据 Uncle Bob 在他关于整洁架构的文章中的说法，我来试着用 UML 图解释控制流...

**没有命令/查询总线**

如果没有命令总线，控制器要么依赖应用服务，要么依赖查询对象。

[**2017-11-18 编辑**] 之前我完全忘记了查询返回数据中的 DTO，现在我把它加了回来。谢谢[指出](https://www.reddit.com/r/PHP/comments/7dcz8k/ddd_hexagonal_onion_clean_cqrs_how_i_put_it_all/dpy6va4/" \t "_blank)我这处错误的[MorphineAdministered](https://www.reddit.com/user/MorphineAdministered)。

上图中我们使用了应用服务接口，尽管我们会质疑这并没有必要。因为应用服务是我们应用代码的一部分，而且我们不会想用另外一种实现来替换它，尽管我们可能会彻底地重构它。

查询对象包含优化过的查询，简单地返回一些给用户看的原始数据就好。这些数据将放在 DTO 中返回，并注入到 ViewModel。ViewModel 中可能有一些 View 逻辑，它被用来填充 View。

另一方面，应用服务还包含用例逻辑，不是浏览数据这么简单，我们需要在系统中做一些事情时触发这些逻辑。应用服务依赖 Repository 返回实体，这些实体中包含着需要触发的逻辑。它也可能依赖领域服务来整合多个实体来完成领域流程，但这种情况很少出现。

展开用例之后，应用服务可能想通知整个系统，这个用例已经发生了。这种情况下，它还要依赖事件派发器来触发事件。

很有意思的是我们在持久化引擎和资源库之上都放上了接口。尽管看起来有些多余，但它们服务于不用的目标：

* 持久化接口是 ORM 之上的抽象层，这样我们可以切换使用的 ORM 而不用修改应用核心。
* 资源库接口是持久化引擎自身的抽象。比方说我们想要从 MySQL 切换为 MongoDB。如果我们想继续使用同样的 ORM，持久化接口可以保持不变，甚至持久化适配器也可以保持不变。然而，两者的查询语言完全不同，所以，我们可以创建使用同样持久化机制的新资源库，可以实现相同的资源库接口，但使用 MongoDB 查询语言而不是 SQL 来构建查询。

**有命令/查询总线**

如果我们的应用使用了命令/查询总线，UML 图基本没有变化，唯一的区别是控制器现在会依赖总线、命令或查询。它将实例化命令或查询，将它们传递给总线。总线会找到合适的处理程序接收并处理命令。

在下图中，命令处理程序接下来将使用应用服务。然而，这不总是必须的，实际上大多数情况下，处理程序将包含用例的所有逻辑。只有在其它处理程序需要重用同样的逻辑时，我们才需要把处理程序中的逻辑提取出来放到单独的应用服务中。

[**2017-11-18 编辑**] 之前我完全忘记了查询返回数据中的 DTO，现在我把它加了回来。谢谢[指出](https://www.reddit.com/r/PHP/comments/7dcz8k/ddd_hexagonal_onion_clean_cqrs_how_i_put_it_all/dpy6va4/)我这处错误的[MorphineAdministered](https://www.reddit.com/user/MorphineAdministered)。

你可能注意到了，总线和命令查询，以及处理程序之间没有依赖。这是因为实际上它们之间应该互相无感知，才能提供足够的解耦。只有通过配置才能设置总线可以发现哪些命令，或者查询应该由哪个处理程序处理。

如你所见，两种情况下，所有跨越应用核心边界的箭头——依赖——都指向内部。如前所述，这是端口和适配器架构、洋葱架构以及整洁架构的基本规则。

**总结**

一如既往，这些架构的目标是得到高内聚低耦合的代码库，这样变化才会简单、快速和安全。

计划不名一文，但制订计划的过程至关重要。——艾森豪威尔

这份信息图是一份概念地图。了解并理解所有这些概念将帮助我们规划出健康的架构和应用。

不过：

地图并非疆域。——阿尔弗雷德·柯日布斯基

这只是一份指南！**应用才是你的疆域，现实情况和具体用例才是运用这些知识的地方，它们才能勾勒出实际架构的轮廓！**

**我们需要理解所有这些模式，但我们还时常需要思考和理解我们的应用需要什么，我们应该在追求解耦和内聚的道路上走多远。**这个决定可能受到许多因素的影响，包括项目的功能需求，也包括构建应用的时间期限，应用寿命，开发团队的体验等等因素。

就到这里，这是我对一切的理解。这就是我脑海中对这一切的梳理。

在下篇文章中我将更深入地展开这些话题: [超越同心圆分层](https://herbertograca.com/2018/07/07/more-than-concentric-layers/)([译](https://www.jianshu.com/p/fcf5bb27a60b))。

然而，我们如何将这些全部展现在代码库中呢？这是再下一篇文章的主题，我如何将架构和领域反映在代码之中。

最后，感谢我的同事[Francesco Mastrogiacomo](https://www.linkedin.com/in/francescomastrogiacomo/)，帮助我制作了漂亮的[信息图](https://drive.google.com/open?id=1E_hx5B4czRVFVhGJbrbPDlb_JFxJC8fYB86OMzZuAhg)。

*译者：我将这份信息图翻译成了中文版*

**小礼物走一走，来简书关注我**