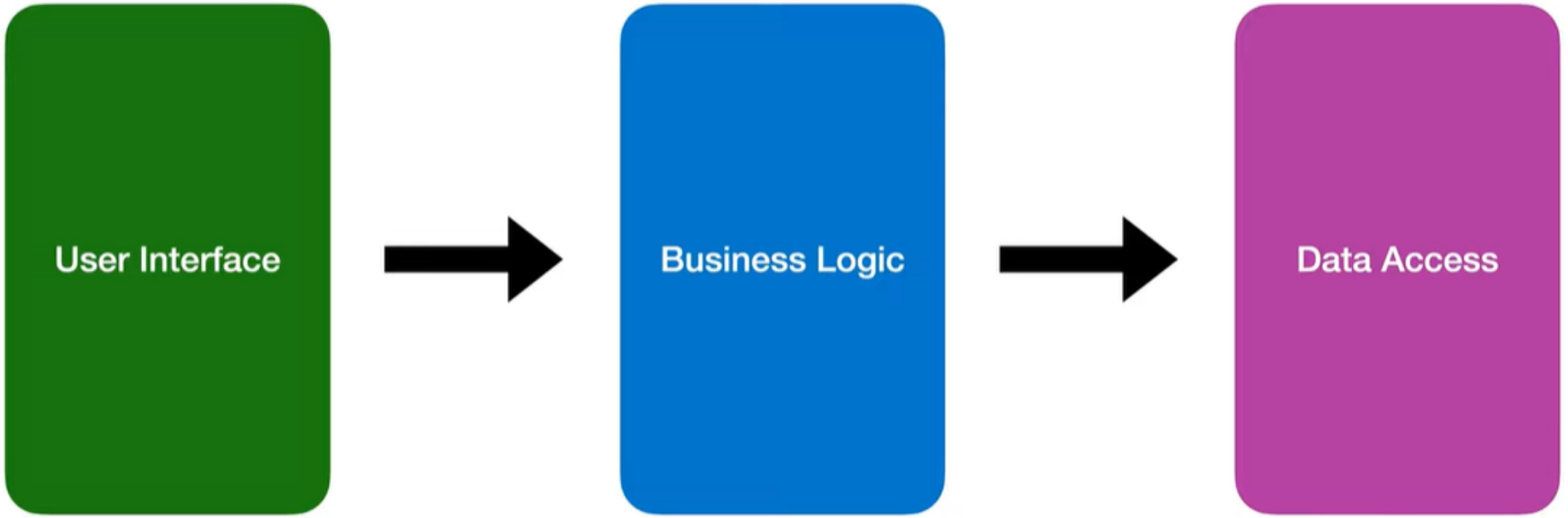
Hexagonal, Onion & Clean Architecture - Drawing Boxes

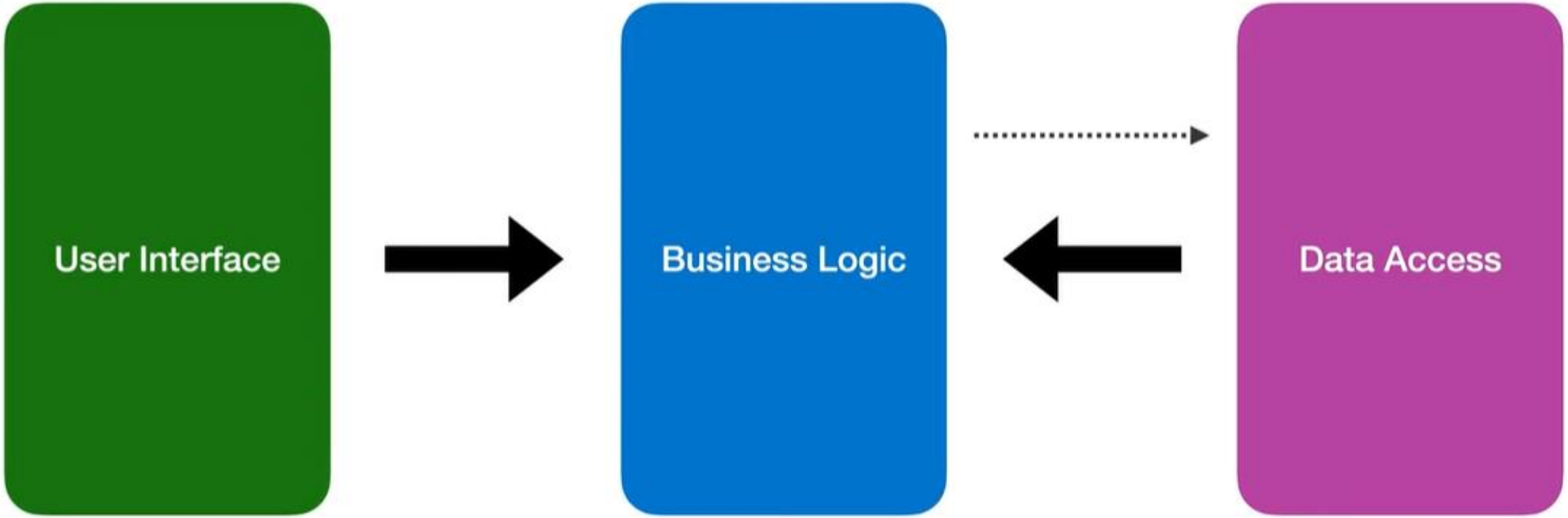
https://www.youtube.com/watch?v=JubdZIdLQ4M

N-Tier / Layered Architecture



A user interface calls into a business logic layer, which in turn calls into data access.

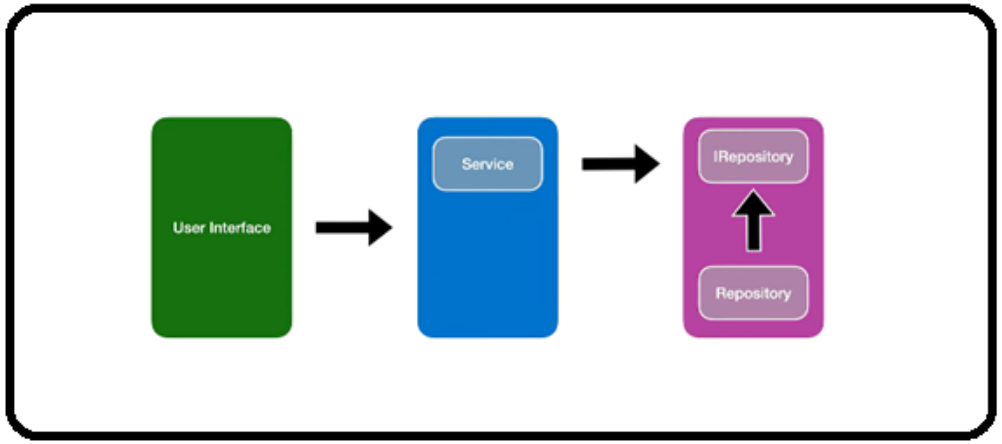
The UI knows that the business logic exists and it depends on it.
The business logic knows of the data access layer too.
The data layer however does not know that other layer exist.

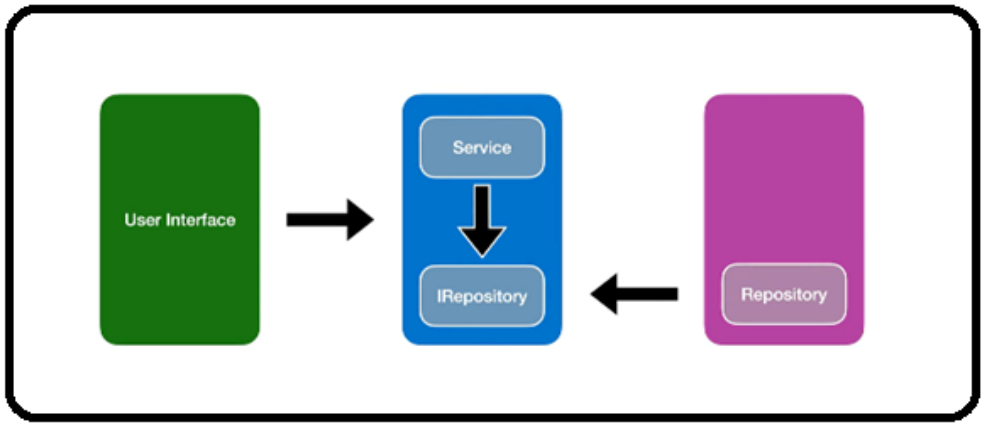


When we invert the dependency between the business logic and the data access. The business logic still makes calls into data access layer.

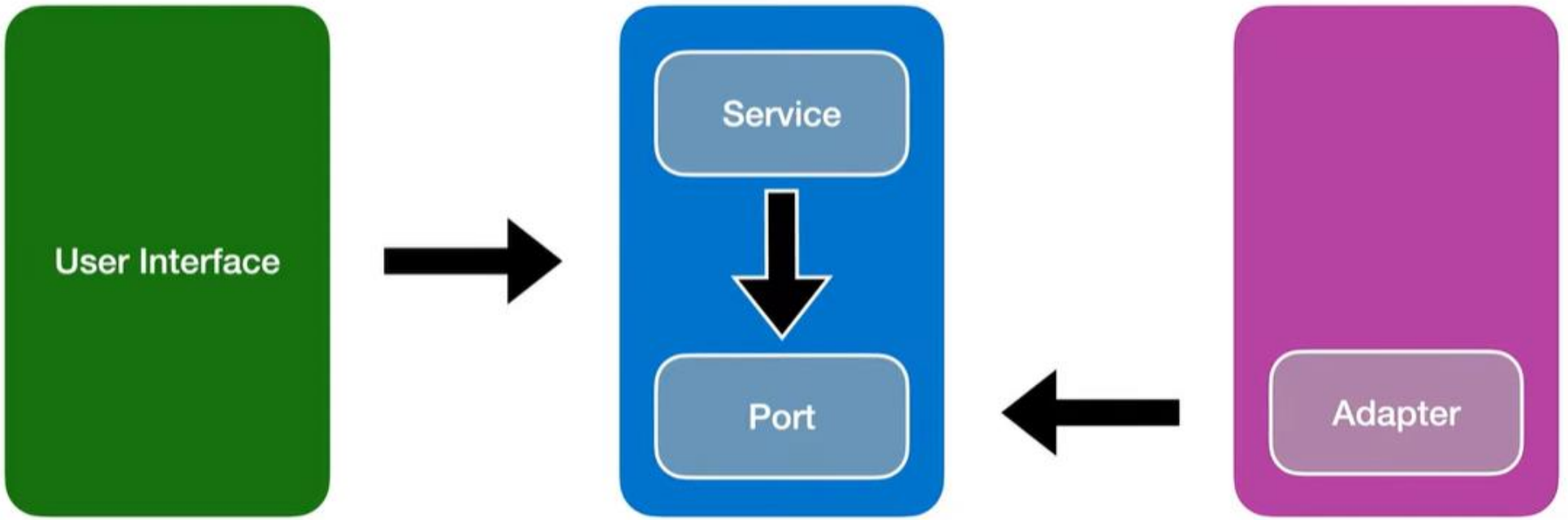
But it does not know what it's calling.

The business logic layer defines an interface that it expects to be implemented in the data access layer.



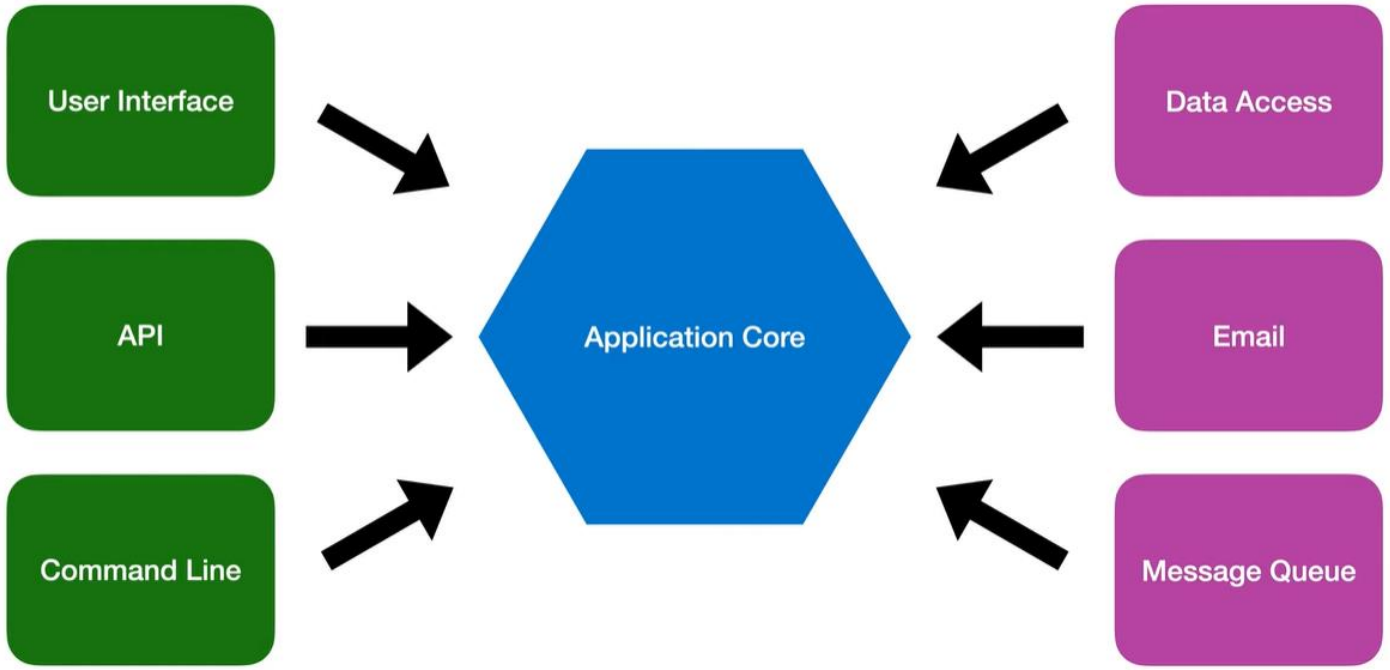


Normally IRepository interface living alongside the repository implmentation, but simply moving this interface means we invert the direction of dependency between these layers.



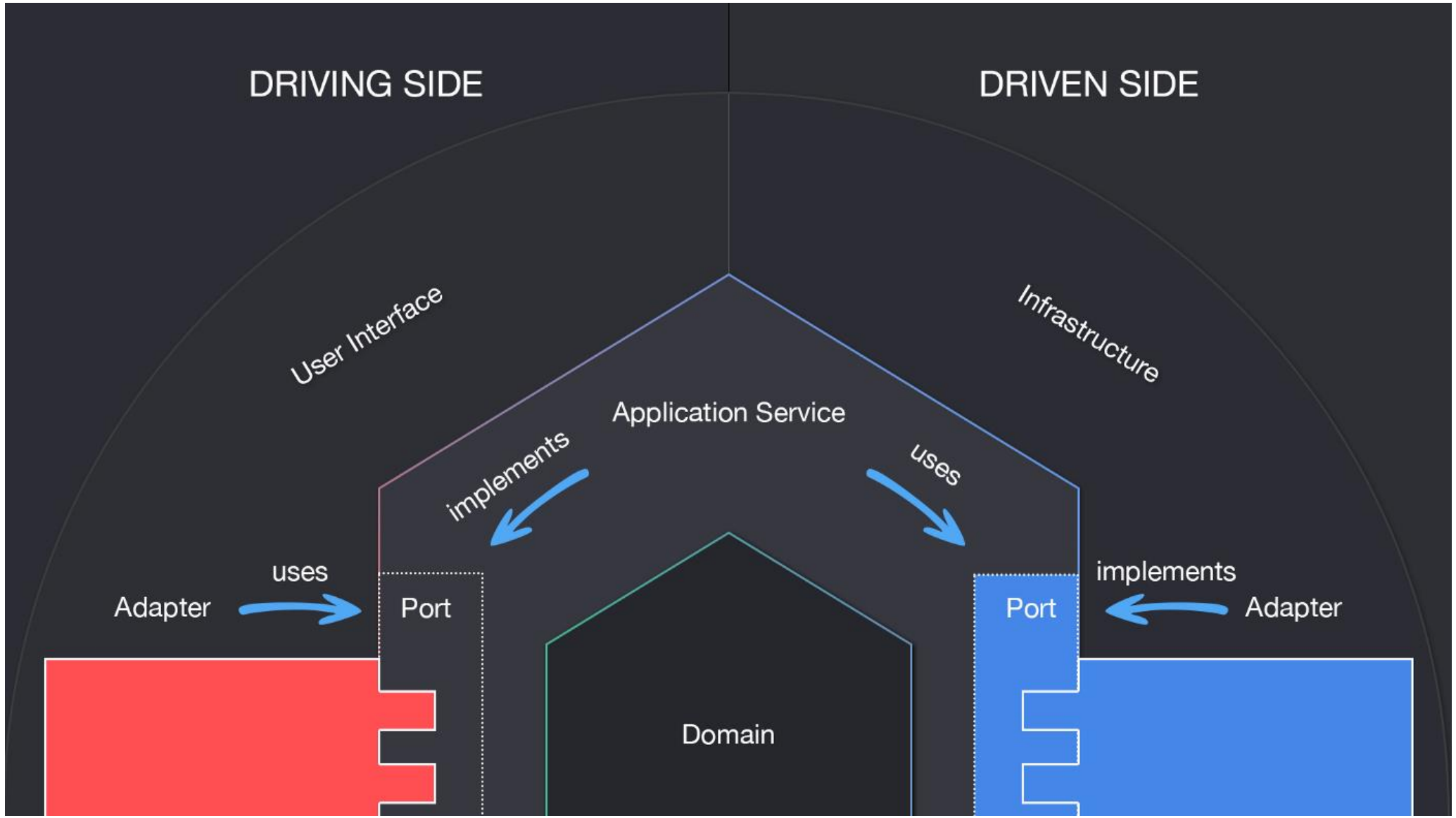
The interface is called a "port". The data access layer knows how to implement that port by using an "adapter".

This architecture became knows as "Ports and Adapters", but it was originally called the "Hexagonal Architecture".



The core logic has no dependencies; it is pure code with no networks and no reference to the outside world, which makes it very easy to write tests for.

On the left we can add adapters for other presentations such as an API or a command line interface. These are called the "Primary" or the "Driving" adapters because they drive, or they make calls into, the core. On the right hand side we have the adapters for the infrastructure, such as first sending emails or using a message queue. These are called the "Secondary“ or the "driven" adapters because of the inverted control; it is the core that drives them.


DRIVING SIDE
DRIVEN SIDE
User Interface
Infrastructure
Application Service
implements
uses
uses
Adapter
Port
Port
implements
Adapter
Domain

Driving Side vs Driven Side

Driving (or primary) actors are the ones that initiate the interaction, and are always depicted on the left side.

For example, a driving adapter could be a controller which is the one that takes the (user) input and passes it to the Application via a Port.

Driven (or secondary) actors are the ones that are “kicked into behaviour” by the Application. For example, a database Adapter is called by the Application so that it fetches a certain data set from persistence.

- Ports will be (most of the time, depending on the language you choose) represented as interfaces in code.

- Driving Adapters will use a Port and an Application Service will implement the Interface defined by the Port, in this case both the Port’s interface and implementation are inside the Hexagon.

- Driven adapters will implement the Port and an Application Service will use it, in this case the Port is inside the Hexagon, but the implementation is in the Adapter, therefore outside of the Hexagon.

Dependency Inversion in the Hexagonal Architecture Context

Agile Software Development Principles:

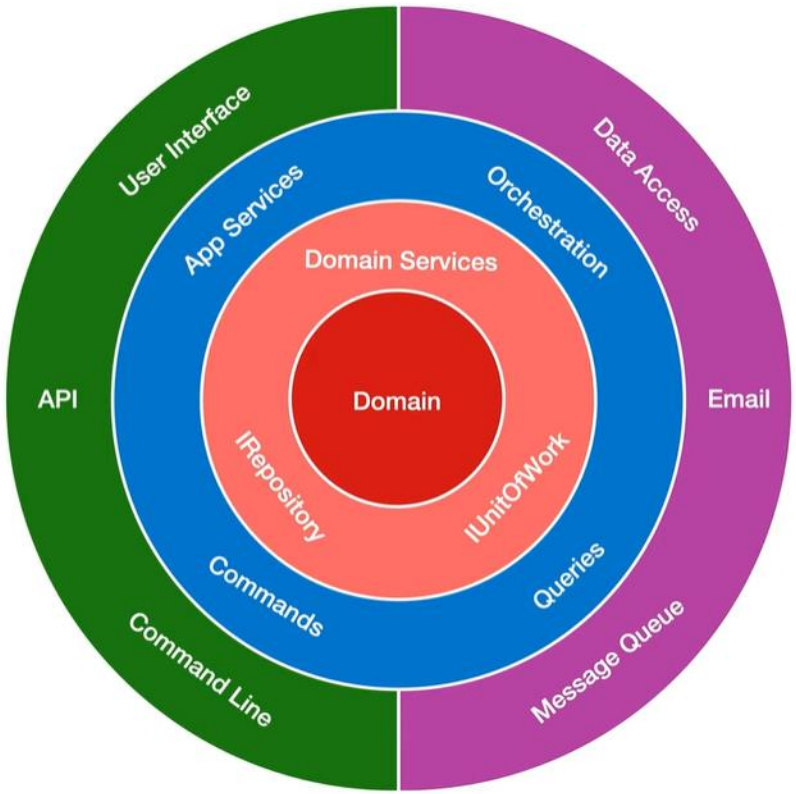
- High-level modules should not depend on low-level modules. Both should depend on abstractions.

- Abstractions should not depend on details. Details should depend on abstractions.

The left and right sides of the Hexagon contain 2 different types of actors, Driving and Driven where both Ports and Adapters exist.

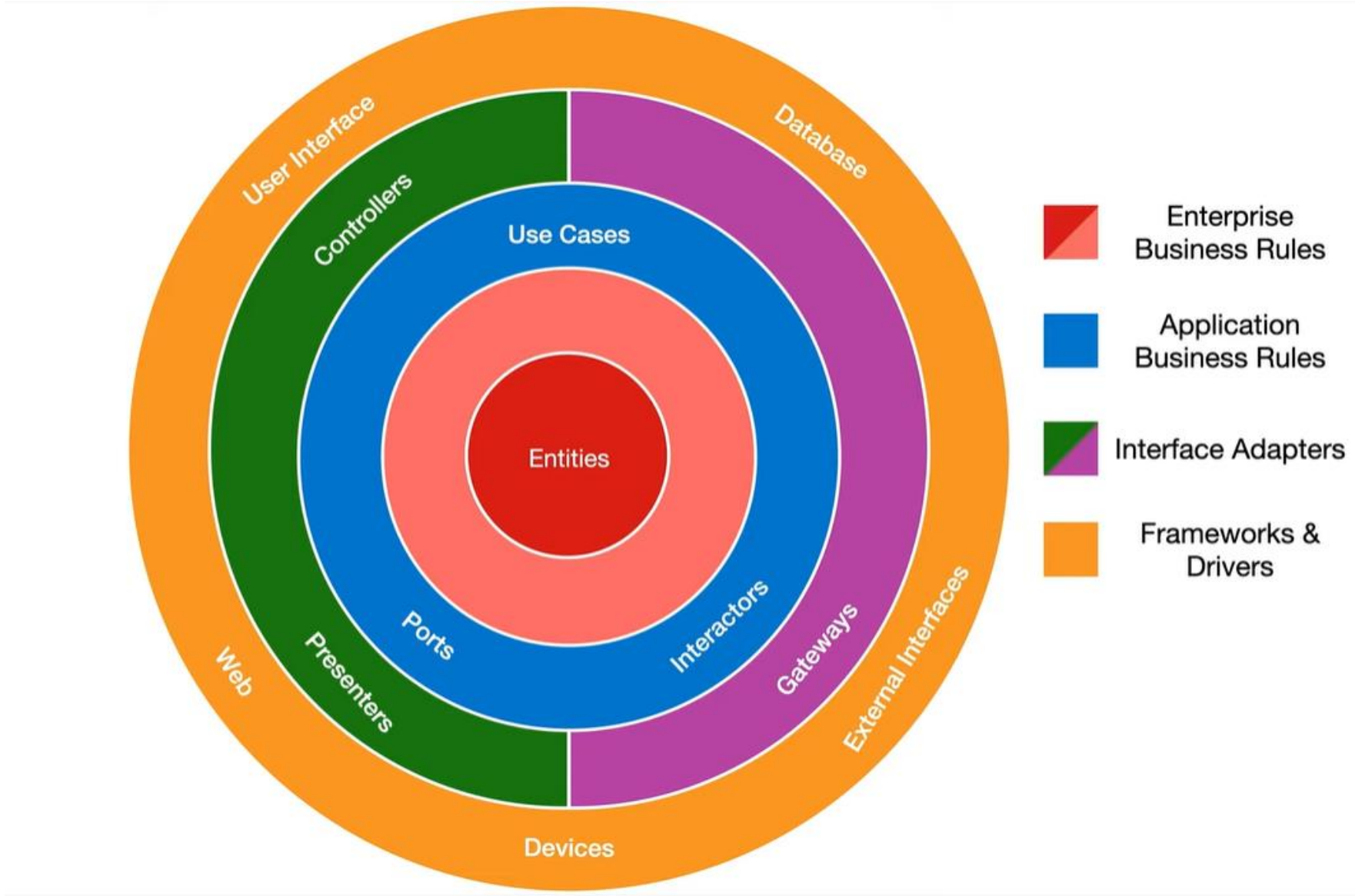
On the Driving side, the Adapter depends on the Port, which is implemented by the Application Service, therefore the Adapter doesn't know who will react to it's invocations, it just knows what methods are guaranteed to be available, therefore it depends on an abstraction.

On the Driven side, the Application Service is the one that depends on the Port, and the Adapter is the one that implements the Port's Interface, effectively inverting the dependency since the 'low-level' adapter (i.e. database repository) is forced to implement the abstraction defined in the application's core, which is 'higher-level'.



The onion architecture divides the application core into with other layers with the dependencies always pointing inwards.

The adapters form an outer layer for all of the infrastructure. Sometimes split into two to separate the presentation layer for the driving or primary adapters from the back end infrastructure at the center with no dependencies is domain model the core business objects. It's common to follow Domain-Driven Design practices for this layer. The other layers in between may vary typically just outside our domain models and our data abstractions such as IRepository interface and maybe an IUnitOfWork interface. But, only the abstractions; these are ports that are expected to be implemented by adapters in the infrastructure layer. Outside of this are the services classes, or the commands and queries. If you are following CQRS patterns, and this is when we can orchestrate interactions between domain objects such as by handling domain events.


User Interface
Database
Controllers
Use Cases
Entities
Ports
Interactors
Gateways
Presenters
Web
External Interfaces
Devices
Enterprise Business Rules
Application Business Rules
Interface Adapters
Frameworks & Drivers

Clean architecture extends this a little further bringing in concepts from other architectures to form a single actionable idea. It gives more meaningful names to these layers.

The center is for enterprise-wide business rules, the entities, the core domain objects. These are the least likely to change. There is no distinction for the domain services in clean architecture, but there is no strict rule on the number of layers either, so we can keep a separate layer if we want to.

The application layer for application specific business rules described as usecases. Here we see some ports for the services interfaces and commands and query models and the term "interactors" for the implementations.

The infrastructure layer contains interface adapters. These are described in more abstract terms. "Controllers" (like in MVC) handle the requests from an API or UI. "Presenters", which convert the response of a use case interactor to a view model. And then "gateways", which convert data from external services, the most common example being the repositories.

Clean architecture also defines a layer outside of these for the frameworks and the drivers; the database, the UI, the web, devices, and external interfaces. There won't be much code in here; it is just a glue that wires everything together.