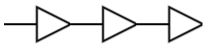
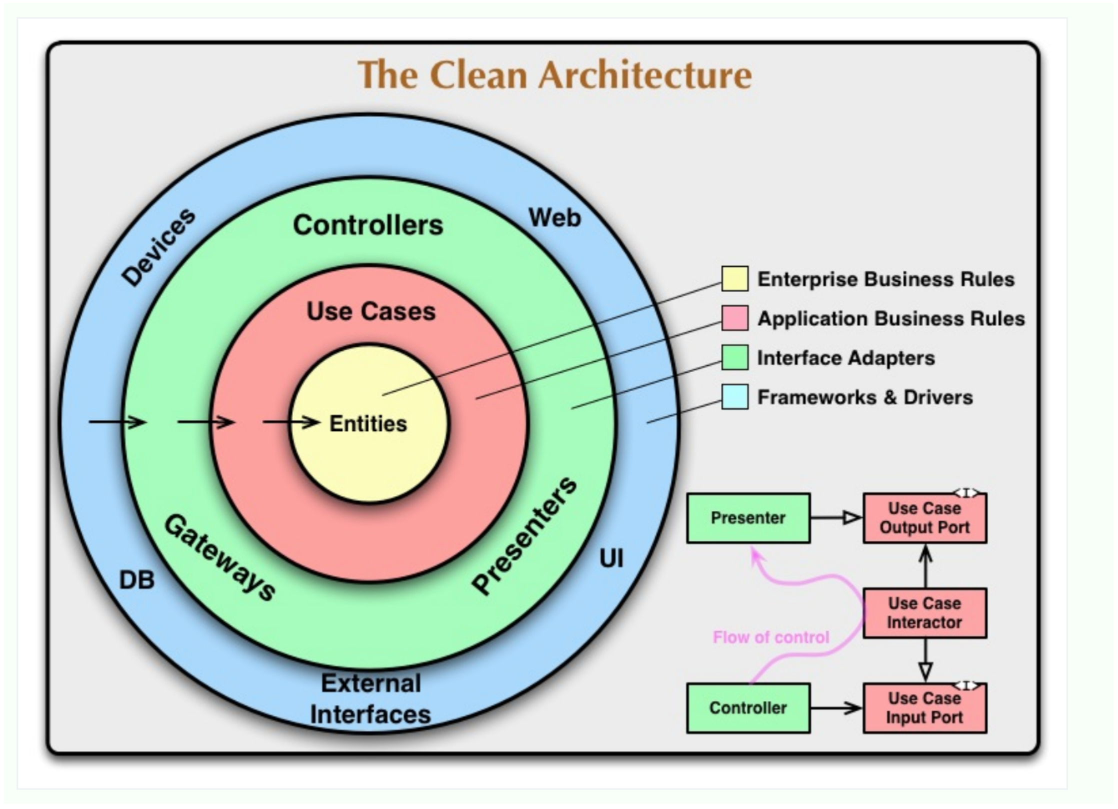
The Clean Architecture

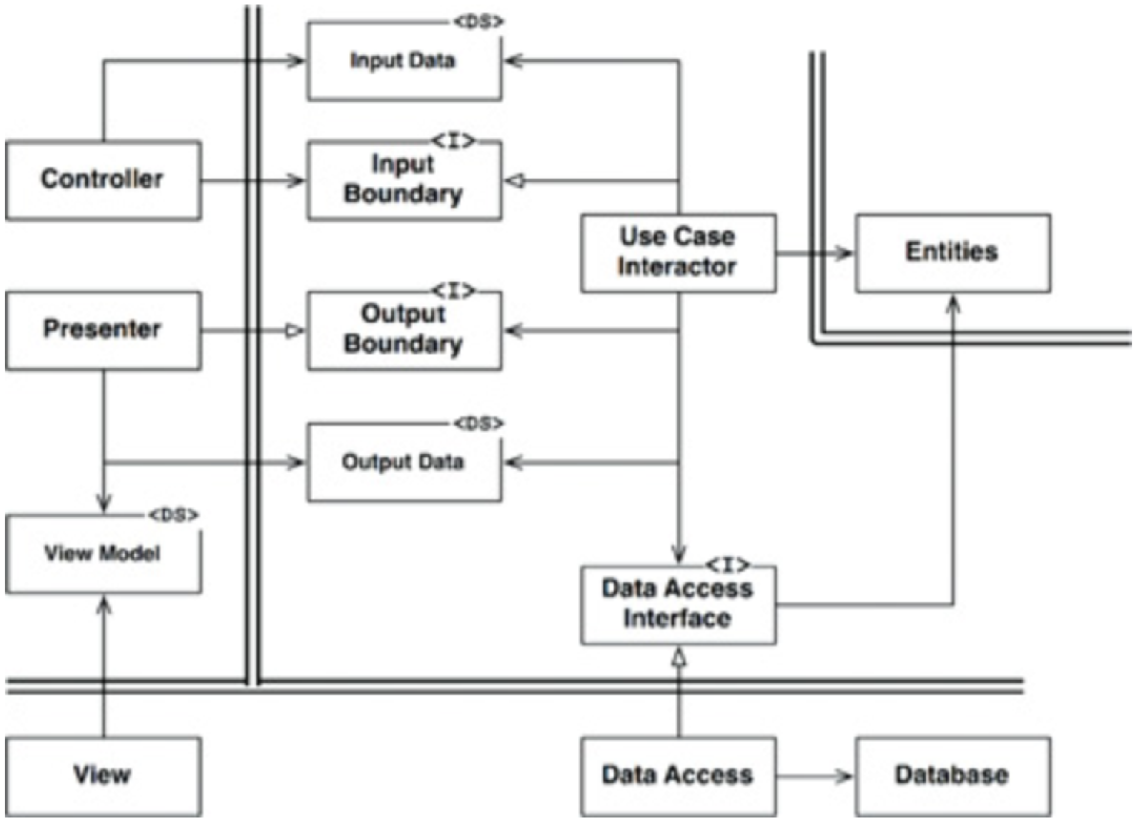
https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

The arrows below should be

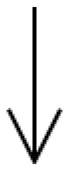


As they indicate Source Code dependency from Lower-level details to higher-level abstractions. The arrows are not indicative of the execution flow. You can see below

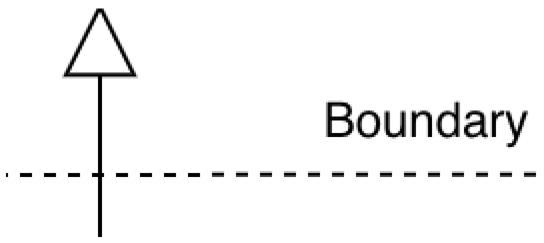




Call flow (from caller to end of arrow being called)



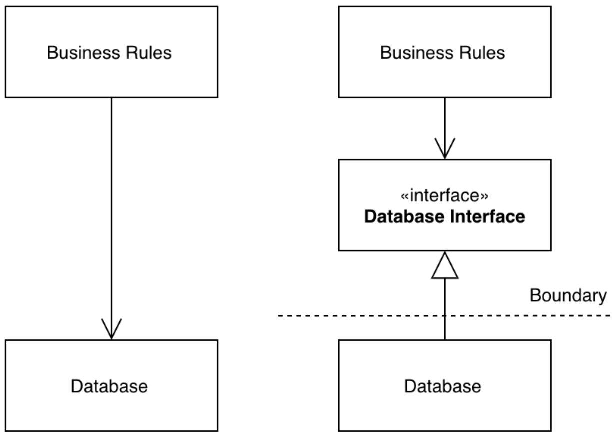
Code dependency: The Origin of the arrow is the low-level detail (implementation) aware of specifics: datbase type: Postgress, MySQL, web framework: Tomcat, Jetty that implements the functionality required by the destination (after crossing the boundary) that is the high level detail (interface), Use case or entity



You should this as an application of the Dependency Inversion lower-level details to higher-level abstractions.

Setting boundaries

Boundaries are lines that separate software elements. They separate things that matter from things that don’t, i.e. high-level components from low-level components. If a high-level component depends on a low-level component at the source level, changes in the low-level components will spread to the high-level component. Therefore, we place a boundary between the two, using polymorphism to invert the logic flow. This is the **dependency inversion principle** in the SOLID principles.



Q&A

Does a data repository belongs to domain or infra?

The question is tricky because a repository should have two parts:

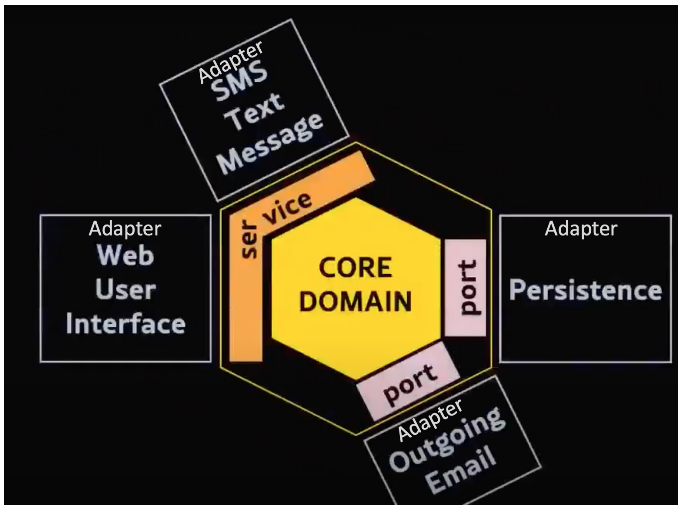
1. Definition of the interface belongs to domain.
2. Implementation of the interface belongs to infra.

What is the difference between Infraestructure and adapters

Infraestructure is anything that is required for our app to work but it is ouside of the main service that we are building. For example: For our Microservice to work, we need to have a shellscript to start up a REDIS database. Another example is deployment pipelines.

Adapters are written in the same language as your Use Cases. They are in the same git respository. They import your Application (Use Cases) code and use them using your programming language (typescript, java) invocation style: functions in javascript, import clases and intanstiate: java.

Hexagonal architecture



Adapters: It is an anti-corruption layer that separate our Business services abstraction from the specific channel adaptations we need to interact with. There are two types of adapters:

- Adapters that call the application: getUserInfo
- Adapters that are being called by the application: sendEmail, saveUser, findUser

Any communication (response to a service, or trigger a port call) always use DTOs (data transfer object) or primitive data. . The rule is that domain objects should not leave the hexagon. Adapters are outside of the Hexagon because they should be a different deployable component. For example in java they could be a different jar. In node they could be integrated using "npm install smsAdapter".

Port: Interfaces and DTOs

Services: Interfaces and DTOs

DTOs are just fields with getters and setters and without core domain logic

The conversion of clean architecture to Hexagonal architecture is as follow:

Clean architecture	Hexagonal
famework and drivers	✗
Interface adapters	adapters
use cases	core domain
enterprise business rules	core domain
Input port	Service
Output port	port

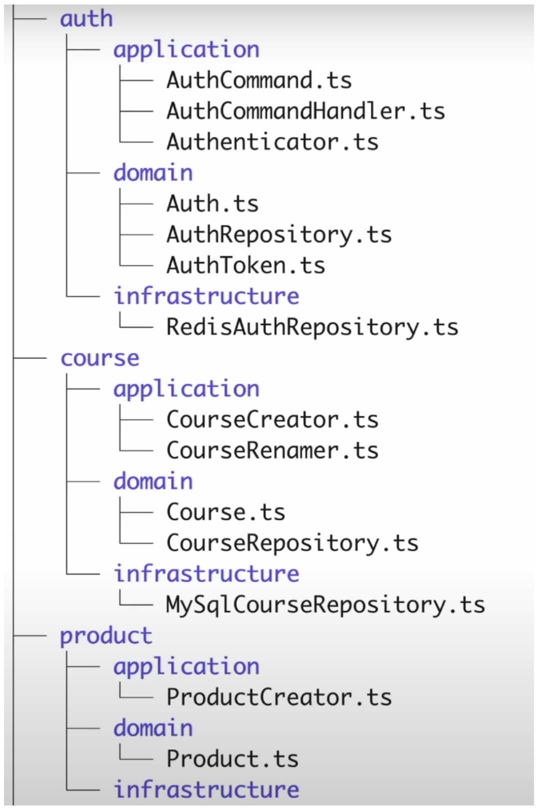
Folder Structure of a project separated in different domains

infrastructure: Adapters

application: core domain (at use case level), services and ports

domain: business core abstraction rules

The first division is at feature or domain level. This keep all the logic of a vertical in the same place, so it is easier to read. Second level is a technical architecture level.



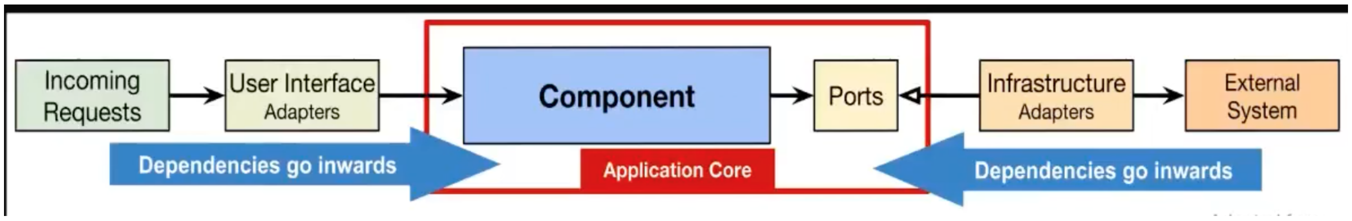
```
├── auth
│   ├── application
│   │   ├── AuthCommand.ts
│   │   ├── AuthCommandHandler.ts
│   │   └── Authenticator.ts
│   ├── domain
│   │   ├── Auth.ts
│   │   ├── AuthRepository.ts
│   │   └── AuthToken.ts
│   └── infrastructure
│       └── RedisAuthRepository.ts
├── course
│   ├── application
│   │   ├── CourseCreator.ts
│   │   └── CourseRenamer.ts
│   ├── domain
│   │   ├── Course.ts
│   │   └── CourseRepository.ts
│   └── infrastructure
│       └── MySqlCourseRepository.ts
├── product
│   ├── application
│   │   └── ProductCreator.ts
│   ├── domain
│   │   └── Product.ts
│   └── infrastructure
```

How do we deal with utils libraries. We can create another domain and put it as part of domain folder

-- share

   |-- domain

         |-- utils.ts


Incoming Requests
User Interface Adapters
Component
Ports
Infrastructure Adapters
External System
Dependencies go inwards
Application Core
Dependencies go inwards