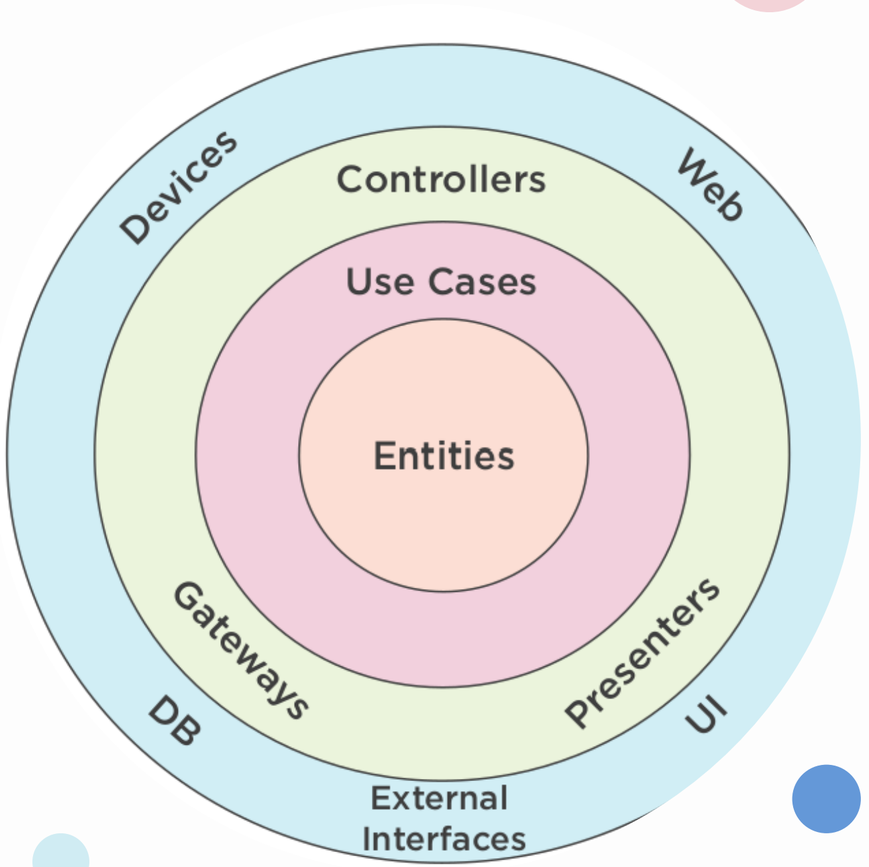
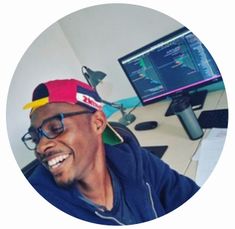
Clean Architecture:
Patterns, Practices and Principles





Régis ATEMENGUE

@regis_ate www.regisatemengue.com

Course objectives

Understanding clean Architecture.

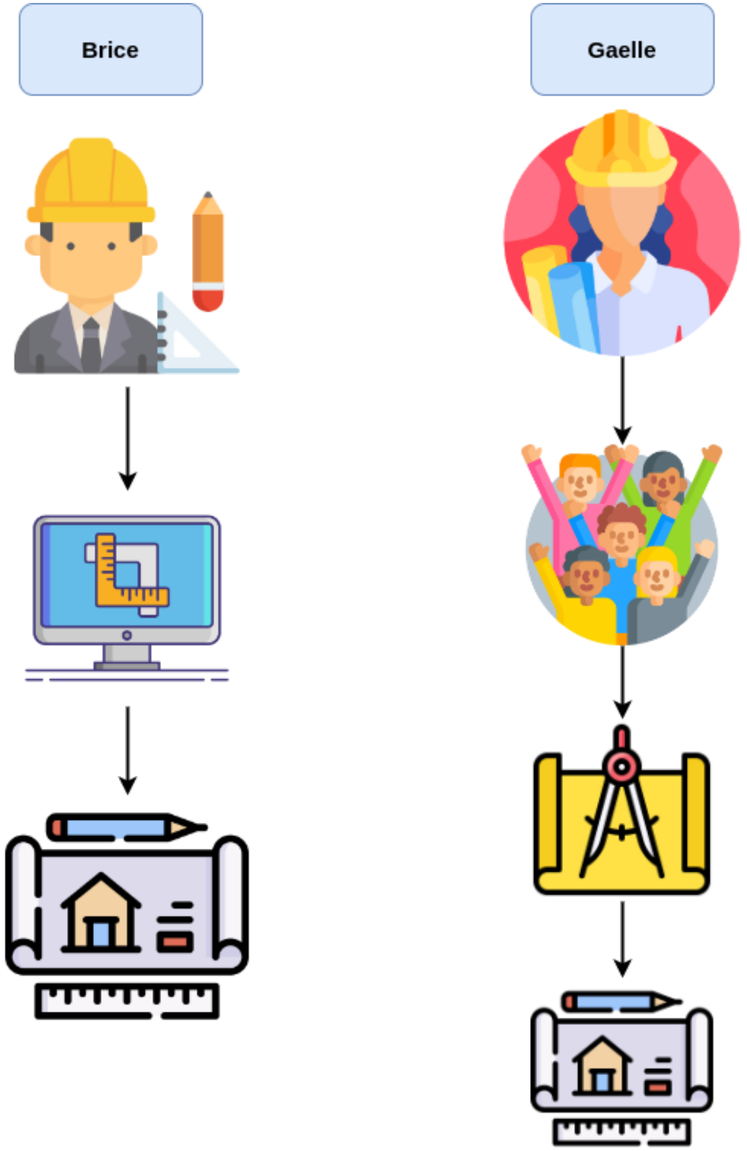
use best practice to build maintainable and testable applications

The Allegory

Imagine for me, if you will, that we have two architects. We'll call Brice and Gaelle. Brice and Gaelle have roughly the same years of experience, the same job title, and will both cost the same amount to hire. We hire them both to build two new buildings for our company in two different cities

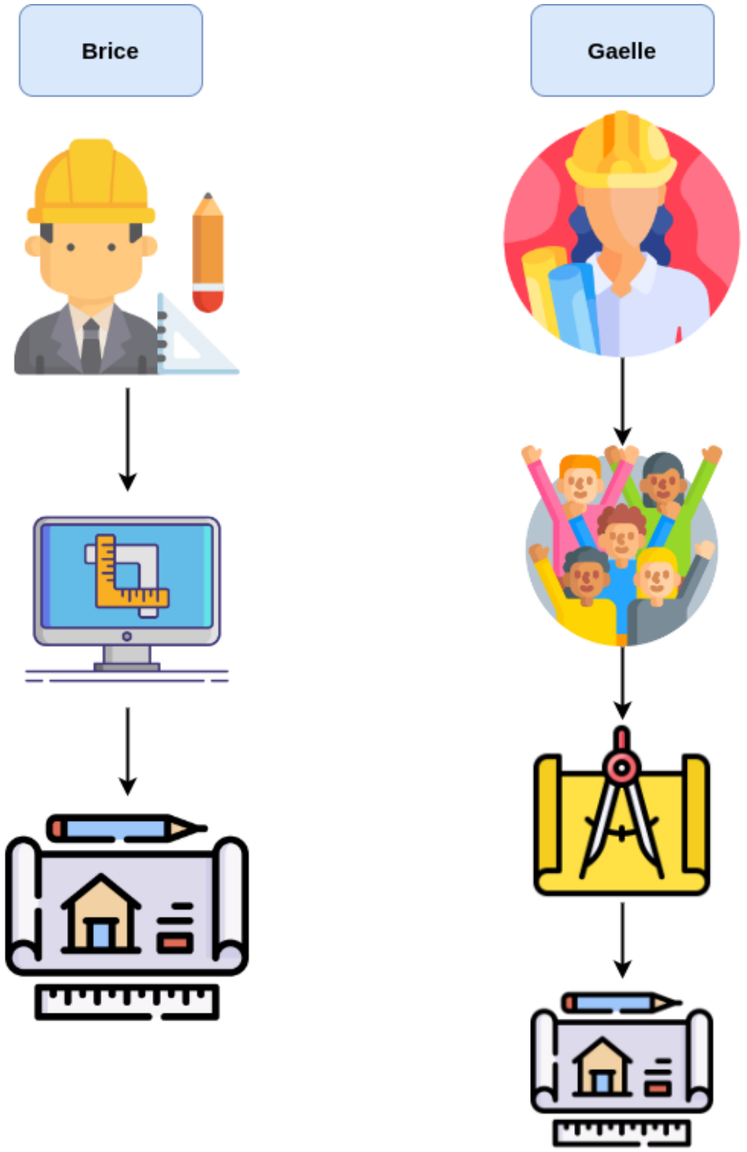
Brice is an expert in classical architecture. He is constantly pushing the boundaries of classical architectural design using cutting-edge technology. In addition, he uses all of his favorite tools and techniques on each of his new projects. His design is elegant. It's a classical design, yet it's state of the art, and he designed it just the way he likes it. In fact, his design completely wows the board of directors. Once he's finished he hands the blueprints to the construction crew, wishes them well, and moves on to design his next masterpiece.

Then, once the building is completed, he shows up to the grand opening ceremony to reap the rewards of his accomplishment.



The Allegory

Gaelle, however, has a very different approach to architecture. First, she starts by chatting with the future inhabitants of the building, that is the employees, the maintenance crew, and she even talks to the construction foreman. She finds out what their needs are, what they find valuable, and what the experience level of the construction crew is. Then she designs the architecture of the building with the inhabitants in mind, considering their needs with each decision that is made. Once she finishes the blueprints she doesn't just hand them off to the construction crew and disappear. Instead, she works side by side with them each day helping them understand and implement the design, gathering feedback, and adjusting the blueprints as necessary, as they learn better ways to solve problems that arise.



Software Architecture

software architecture is simply the combination of **application architecture** and **system architecture,** again in relation to structure and vision. In other words, it's anything and everything related to the design of a software system; from the structure of the code and understanding how the whole software system works at a high level, to how that software system is deployed onto infrastructure.

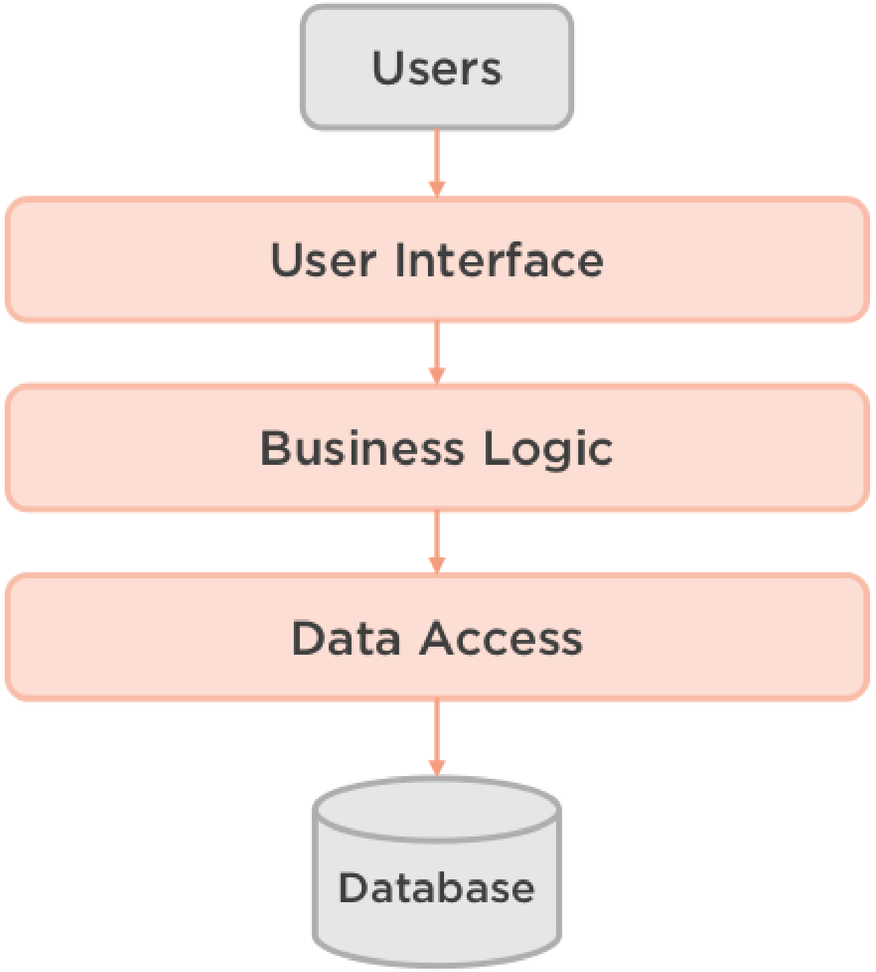
High-Level

Structure

Layers

Components

Relationships



Messy vs. Clean Architecture

SPAGHETTI

LASAGNA

What Is Bad Architecture?

Complex

Incoherent

Ridged

Brittle

Untestable

Unmaintainable

SPAGHETTI

First, bad or messy architecture looks and feels like spaghetti to us. It's just a messy pile of code with an occasional meatball thrown in here or there. It's almost impossible to navigate from the start of one noodle to the end of that same noodle, and it's very difficult to add or replace a noodle without disrupting all of the neighboring noodles

What Is Good Architecture?

Simple

Understandable

Flexible

Emergent

Testable

Maintainable

LASAGNA

Clean architecture looks and feels like lasagna to us. It has nice, consistent, horizontal noodle boundaries that uniformly divide the various layers of filling. When a piece is too big we cut it into smaller components with crisp, clean, orthogonal lines of demarcation, and it has a nice presentation layer riding on top of an edible stack of functionality.

Bad and Good Architecture

So, in more technical terms, what is bad or messy architecture? Well, it's an architecture that is **complex**, but due to accidental complexity rather than necessary complexity. It's **incoherent** in the sense that the parts don't seem like they fit together. It's **rigid**, that is the architecture resists change or makes it difficult to evolve the architecture over time. it's brittle, touching a part of the code over here might break another part of the code somewhere else, it's **untestable**, that is, you'd really like to write unit tests and integration tests, but the architecture fights you each step of the way, and ultimately, all of these things lead to an architecture that's **unmaintainable** over the life of the project.

On the other hand, we know we have good, clean architecture when it is **simple** or at least it's only as complex as is necessary, and that complexity is not accidental. It's **understandable**, that is it's **easy** to reason about the software as a whole, it's **flexible**, we can easily adapt the system to meet changing requirements, it's emergent, the architecture evolves over the life of the project, it's **testable**, the architecture makes **testing easier**, not harder, and ultimately all of this leads to an architecture that's more maintainable over the life of the project.

What Is Clean Architecture?

Clean architecture is that it's architecture that is designed for the **inhabitants** of the architecture, not for the architect or for the machine. Clean architecture is a philosophy of architectural essentialism. It's about focusing on what is truly essential to the software's architecture versus what is just an implementation detail. By designing for the inhabitants we mean the people that will be living within the architecture during the life of the project.

This means the users of the system, the **developers building** the system, and the **developers maintaining** the system. By not designing for the architect we mean that the architect should put aside his or her own desires, preferences, and wishes, and only consider what is best for the inhabitants of the architecture with each decision that is made. By not designing for the machine we mean that we should optimize the architecture first for the needs of the inhabitants, that is the users and the developers, and only optimize for the machine when the cost of performance issues to the users, who are inhabitants of the architecture, outweighs the benefit of a clean design to the developers who are also inhabitants of the architecture.

Why Invest in Clean Code?

Cost/benefit

The primary justification for investing in clean architecture is mainly a cost-benefit argument. Our goal as software architects is to minimize the cost of creating and maintaining the software while we maximize the benefit that is the business value that the software provides.

Minimize cost

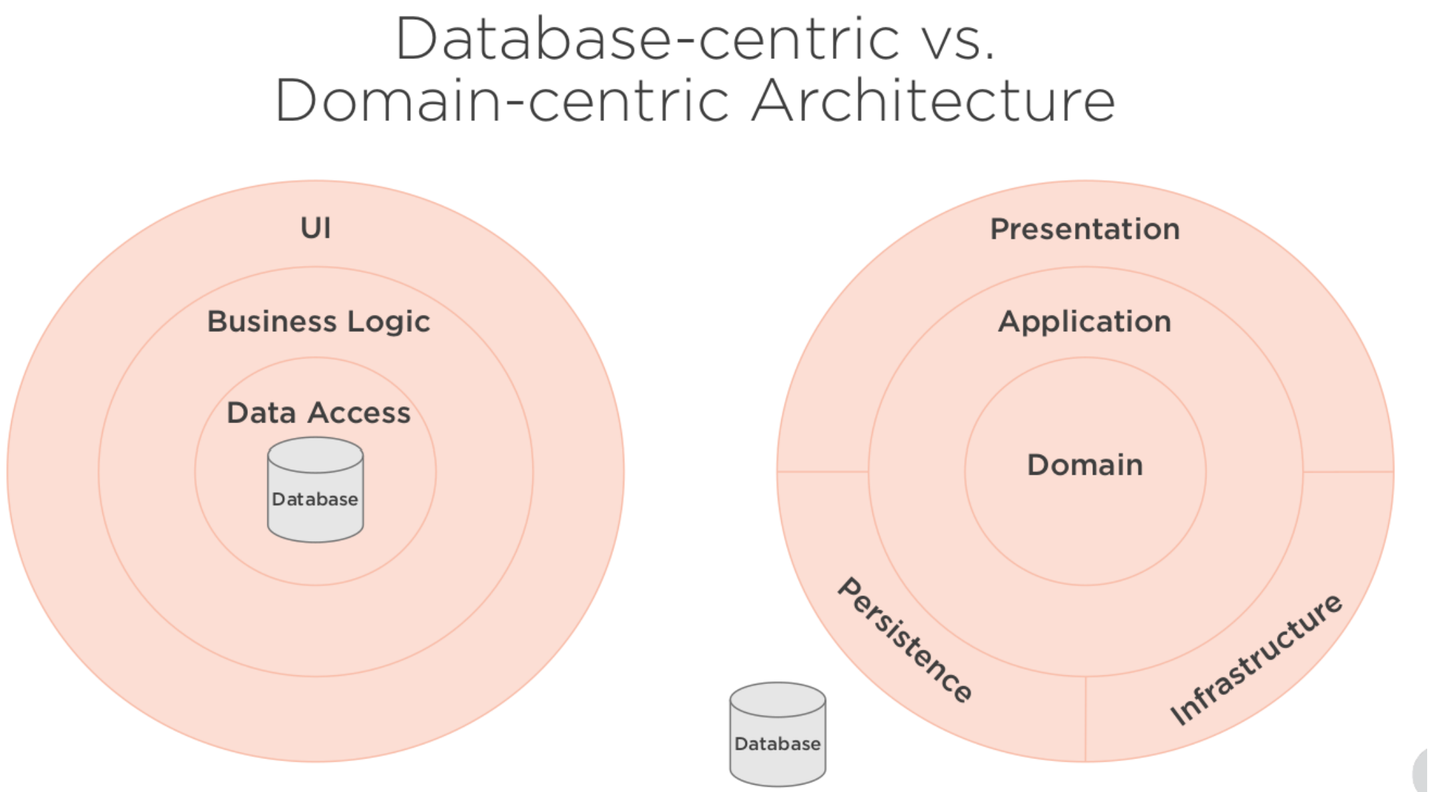
The second, clean architecture builds only what is necessary when it is necessary. We attempt to create only the features and corresponding architecture that are necessary to solve the immediate needs of the users in order of each features perceived business value. We attempt to do this without creating any accidental complexity, unnecessary features, premature performance optimizations, or architectural embellishments. This helps to reduce the cost of creating the system. Third, clean architecture optimizes for maintainability

Maximize value

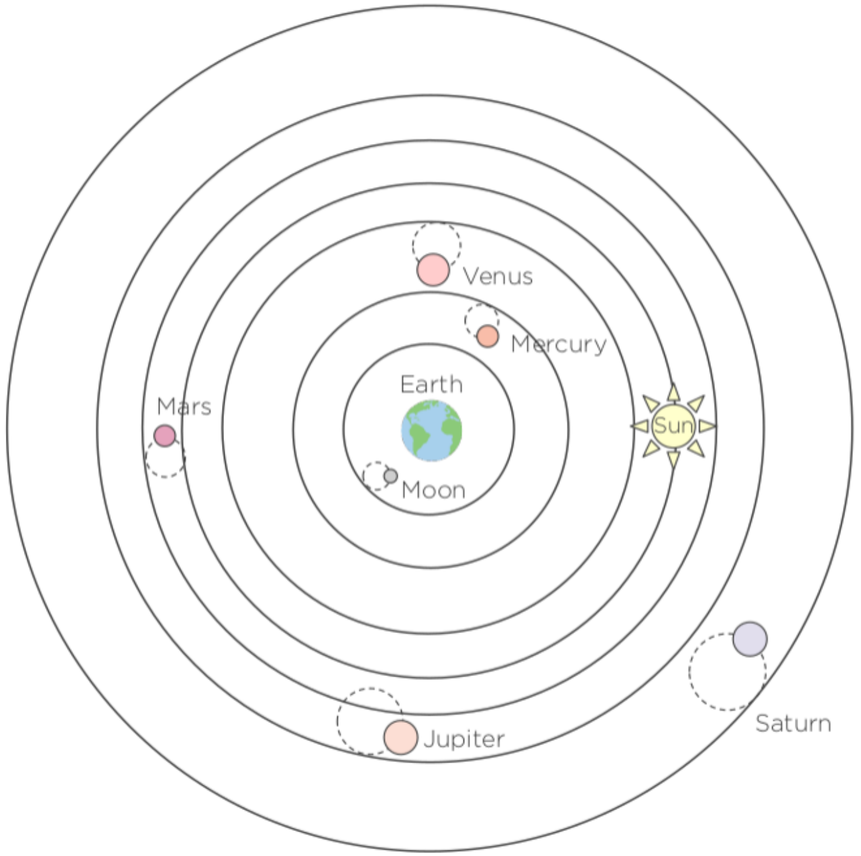
For an average enterprise application with a sufficiently long lifecycle, say 10 years or so, we spend significantly more time and money maintaining the system than we do creating it This focus on value-adding and cost-reducing activities attempts to maximize the return on investment of the software as a whole. In addition, there are several other value-adding and cost-reducing benefits that clean architecture provides;

Maximize ROI

Domain-Centric Architecture

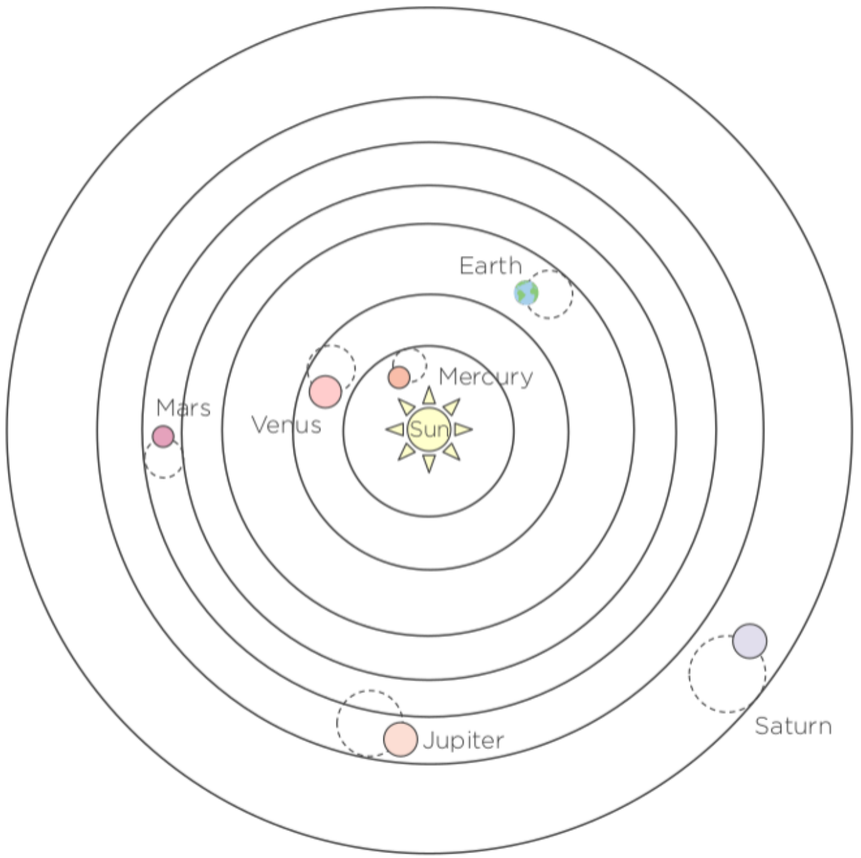


Domain-Centric Architecture



Geocentric Model

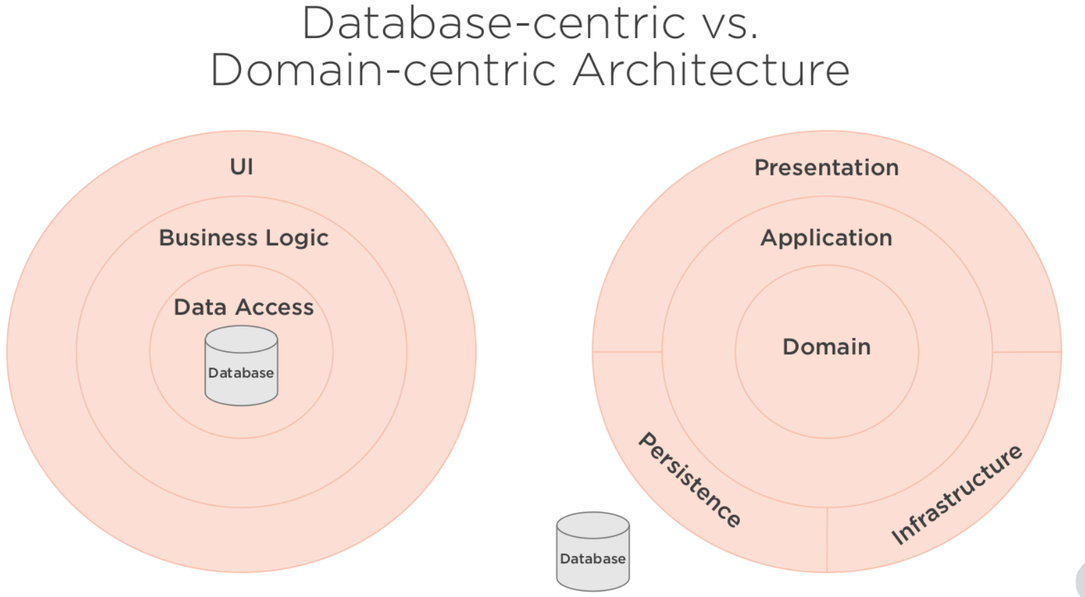
Classique



Heliocentric Model

Nicolaus Copernicus

Domain-Centric Architecture



The evolution of thinking is taking place in the world of software architecture. We have the classic three-layer database-centric architectures. Its main characteristic is that the user interface, the business logic, and the data access layer revolve around the database. The database is **essential** and is therefore at the heart of this architecture.

A new perspective has changed the way many of us look at our architecture. Rather than having the database at the center of our architecture, some of us are putting the **domain at the center,** and making the database just an implementation detail outside of the architecture. **Here the domain is essential, and the database is just a detail.**

“The first concern of the architect is to make sure that the house is usable, it is not to ensure that the house is made of brick.”

- Uncle Bob

Essential Vs Detail

Space is **essential**

Usability is **essential**

Building material is a **detail**

Ornamentation is a **detail**

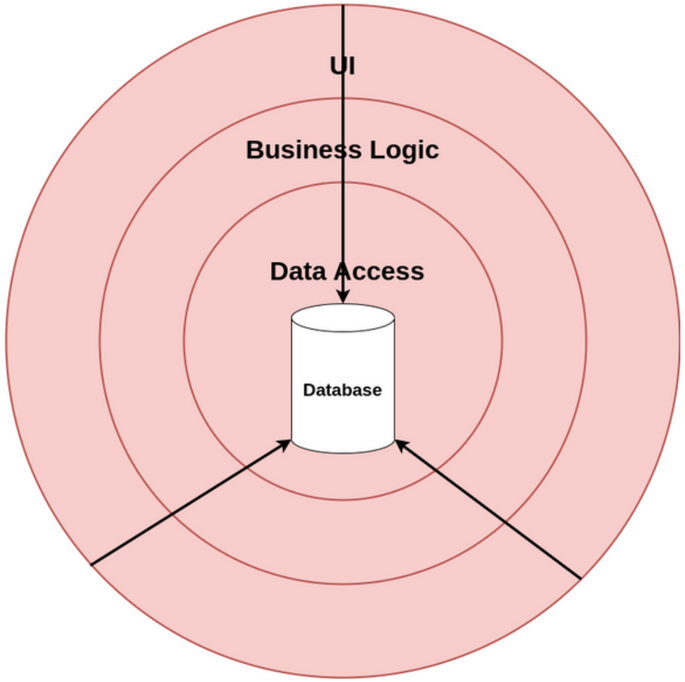
Domain is **essential**

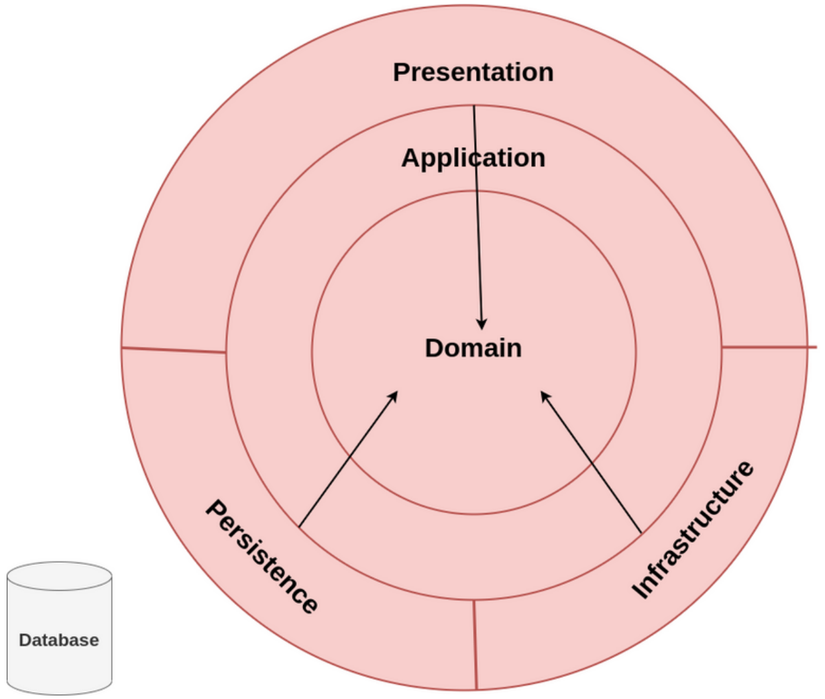
Use cases are **essential**

Presentation is a **detail**

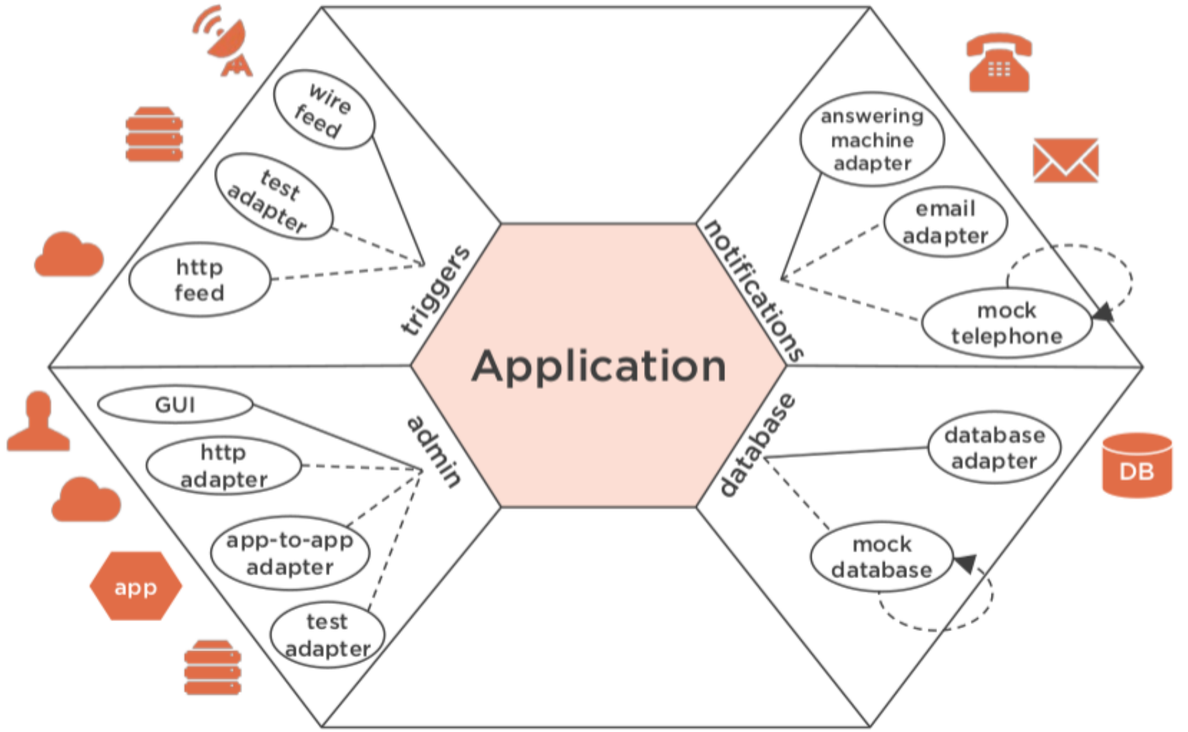
Persistence is a **detail**

Domain-Centric Architecture vs. Database Centric





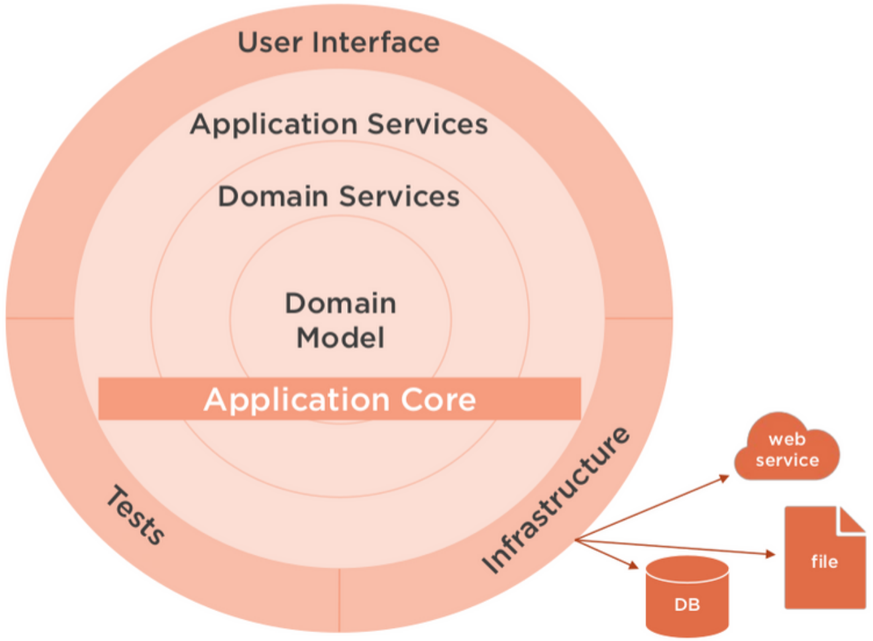
With database-centric architectures the database is essential, so the database is at the center of the application, and all dependencies point towards the database. With domain-centric architectures the domain and use cases are essential, and the presentation and persistence are just a detail, so the domain is at the center of the application wrapped in an application layer and all dependencies point towards the domain



Original source: http://alistair.cockburn.us/Hexagonal+architecture

'L'architecture hexagonale d'Alistair Cockburn. Il s'agit d'une architecture en couches avec la couche d'application, et donc transitivement, le domaine au centre de l'architecture. En outre, il s'agit d'une architecture de plugins qui comprend des ports et des adaptateurs. Essentiellement, les couches externes de l'architecture adaptent la couche d'application interne aux différents supports de présentation, supports de persistance et systèmes externes. Vous pouvez exécuter, et donc tester, une isolation dans cette architecture d'application complète sans interface utilisateur, sans base de données et sans dépendances externes.

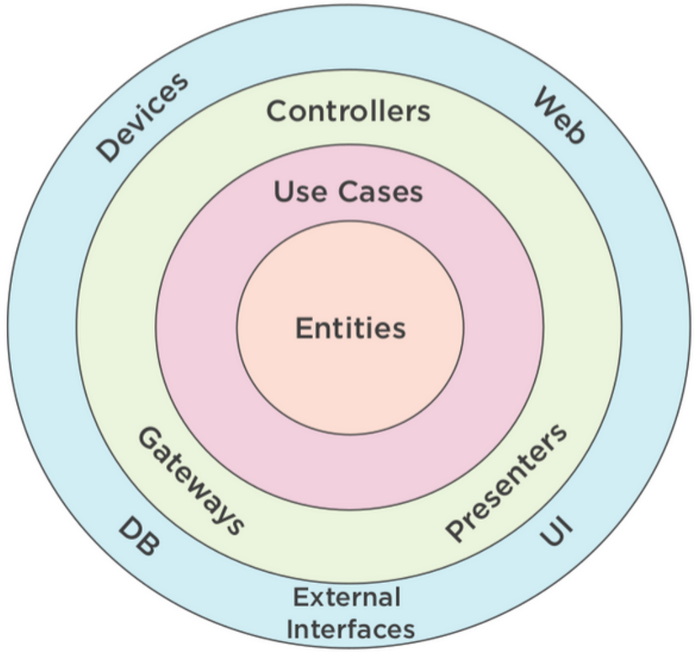
Onion Architecture

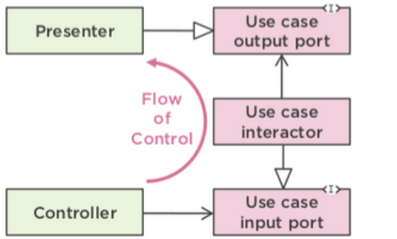


Original source: http://jeffreypalermo.com/blog/the-onion-architecture-part-2/

The onion architecture by Jeffrey Palermo. This is also a layered architecture with the domain at the center surrounded by an application layer. The outer layers consist of a thin UI as a presentation layer, and an infrastructure layer, which includes persistence. In addition, all dependencies point toward the center of the architecture, that is no inner layer knows about any outer layer. Once again, you can test this application architecture in isolation without a UI, a database or any external dependencies.

The Clean Architecture

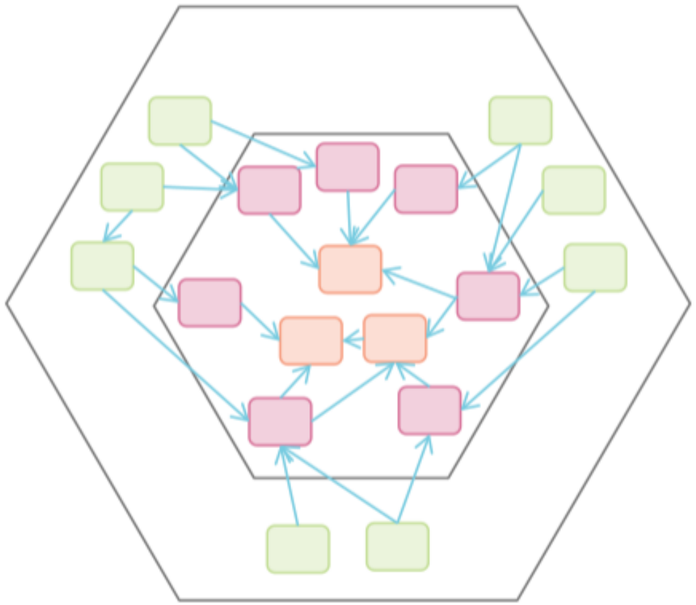




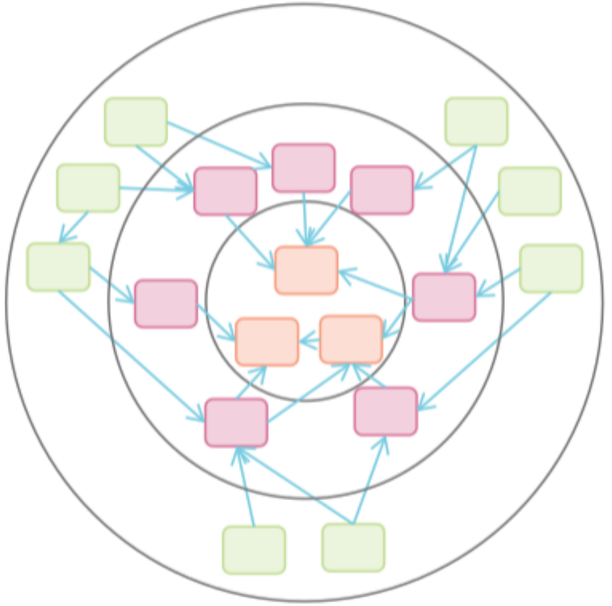
Original source: http://blog.8thlight.com/uncle-bob/2012/08/13/the-clean-architecture.html

the clean architecture by Uncle Bob. Once again, it's a layered architecture with the domain, that is the entities, at the center surrounded by an application layer, that is the use cases. The outer layer consists of ports and adapters adapting the application core to the external dependencies via controllers, gateways, and presenters. In addition, Uncle Bob goes one step further by incorporating Ivar Jacobson's BCE architecture pattern to explain how the presentation layer and the application layer should be wired up.

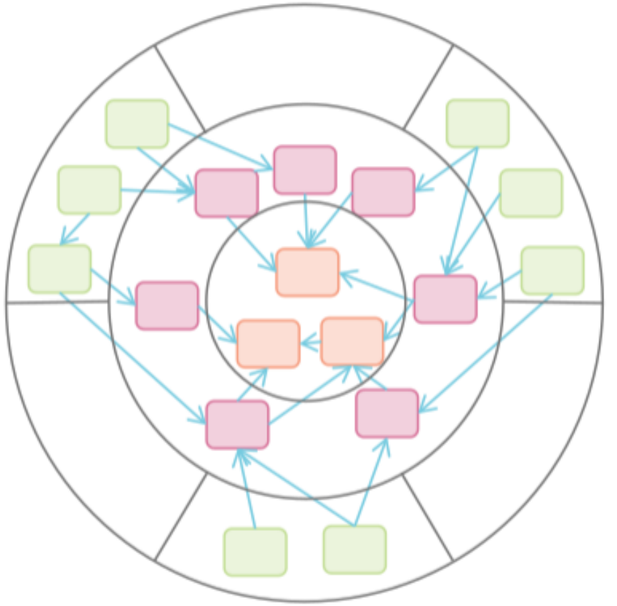
It's All the Same Thing



Hexagonal



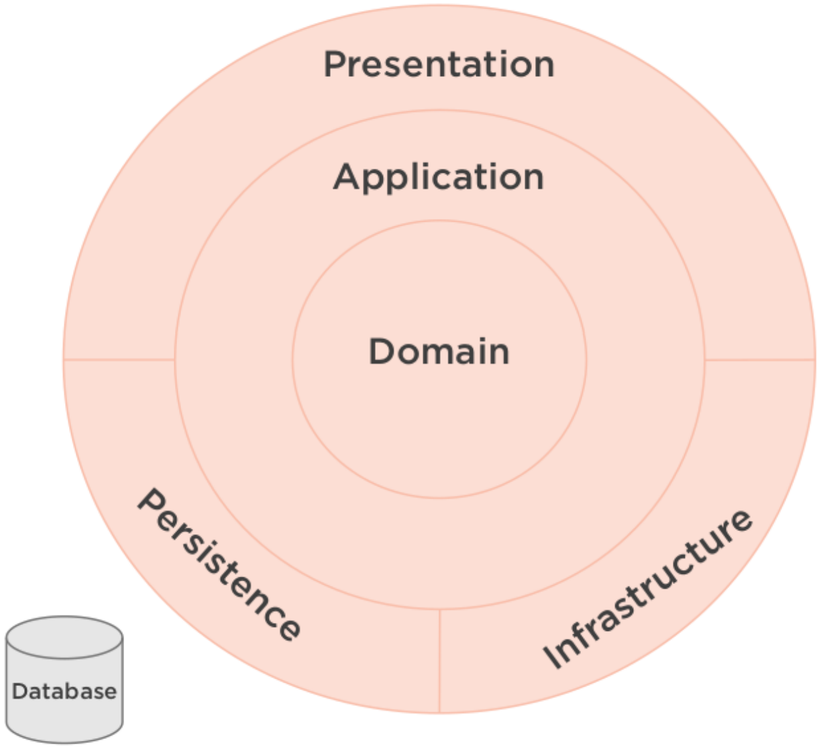
Onion



Clean

Why Use Domain-centric Architecture?

Firstly, **the focus is on the domain**, which is essential for the architecture's inhabitants, i.e. users and developers, and offers several advantages and cost reductions for our software. Secondly, there is **less coupling** between domain logic and implementation details, such as presentation, database, and operating system. This allows the system to be more flexible and adaptable, and we can much more easily evolve the architecture over time. Thirdly, using a domain-centric architecture allows us t**o incorporate Domain-Driven Design,(DDD)** which is an excellent set of strategies developed by Eric Evans for managing business domains with a high degree of complexity.



Domain-Driven Design(DDD)

Domain-Driven Design is an approach to software development that centers the development on programming a domain model that has a rich understanding of the processes and rules of a domain

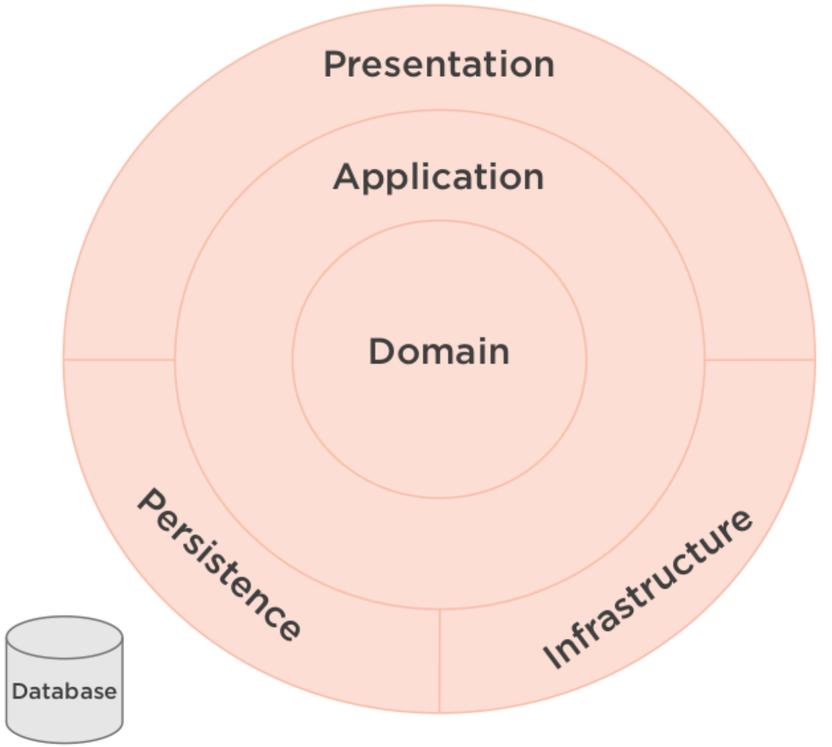
Martin Fowler

Value Proposition of DDD

Principles and patterns to solve difficult problems	History of success with complex projects
Aligns with practices from our own experience	Clear, readable, testable code that represents the domain

why would we not want to use a domain-centric architecture?

First, **change is difficult.** Most developers come out of college having only been taught the traditional three-layer database-centric architecture. In addition, it may also be the only architectural model that the architect knows well enough to offer guidance on. Second, it **requires more thought to implement a domain-centric desig**n. You need to know what classes belong in the domain layer, and what classes belong in the application layer rather than just throwing everything in a business logic layer. Third, it has a **higher initial cost** to implement this architecture compared to a traditional three layer database-centric architecture.



Clean Microservice Design Principle

Clean Microservice Design Principle

The clean microservice design promotes object-oriented design with separation of concerns achieved by dividing software into layers using t**he dependency inversion principle (programming against interfaces).**

Clean microservice design comes with the following benefits:

- Not tied to any single framework
- Not tied to any single API technology like REST or GraphQL
- Unit testable
- Not tied to a specific client (works with web, desktop, console, and mobile clients) • Not tied to a specific database
- Not dependent on any specific external service implementation

Clean Microservice Design Principle

Important Design Principles

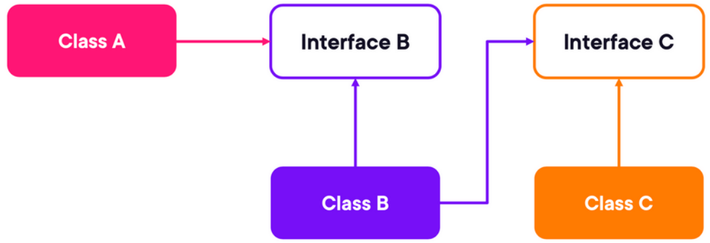
- Dependency inversion
- Separation of concerns
- Single responsibility
- DRY
- Persistence ignorance

Dependency Inversion: Program Against Interfaces Principle (Generalized Dependency Inversion Principle)

Do not write programs where internal dependencies are concrete object types—instead, program against interfaces. An exception to this rule is data classes with no behavior (not counting simple getters/setters)

Typical Approach

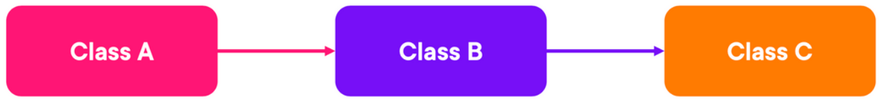
Adding Dependency Inversion



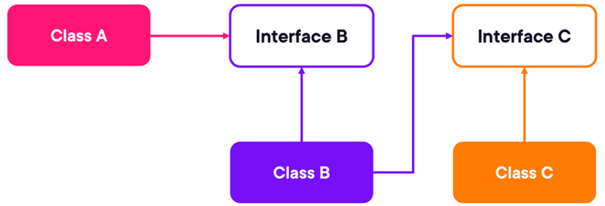
Dependency Inversion: Program Against Interfaces Principle (Generalized Dependency Inversion Principle)

An interface is used to define an abstract base type. Various implementations can be introduced that implement the interface. When you want to change the behavior of a program, you create a new class that implements an interface and then use an instance of that class. In this way, you can practice the open-closed principle. You can think of this principle as a prerequisite for using the **open-closed principle** effectively. The program against interfaces principle is a generalization of the dependency inversion principle from the **SOLID** principles:

Typical Approach



Adding Dependency Inversion



Dependency Inversion: Program Against Interfaces Principle (Generalized Dependency Inversion Principle)

The dependency inversion principle is a methodology for **loosely coupling** software classes.

When following the principle, the conventional dependency relationships from high-level classes to low-level classes are reversed, thus making the high-level classes independent of the low-level implementation detail

The dependency inversion principle states:
• High-level classes should not import anything from low-level classes
• Abstractions (= interfaces) should not depend on concrete implementations (classes) • Concrete implementations (classes) should depend on abstractions (= interfaces)

Dependency Inversion: Program Against Interfaces Principle (Generalized Dependency Inversion Principle)

The dependency inversion principle is a methodology for **loosely coupling** software classes.

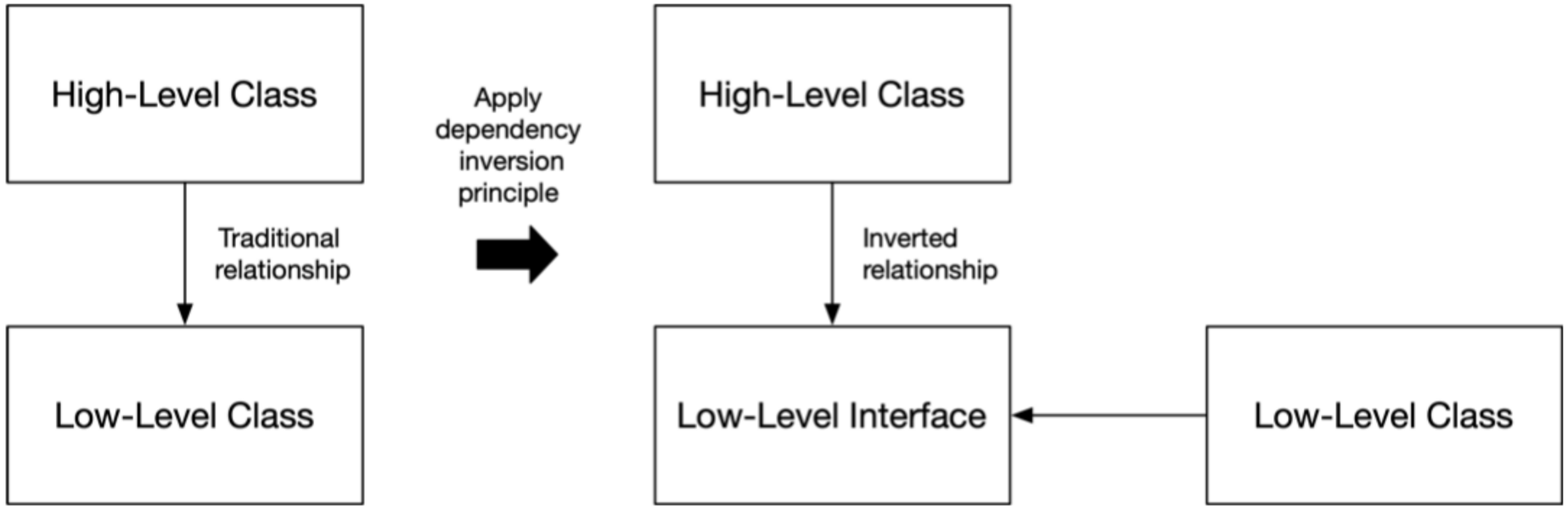
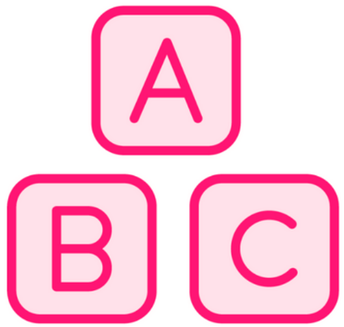


Fig 3.1 Dependency Inversion Principle

Separation of Concerns

Separation of Concerns is a software architecture design pattern/principle for separating an application into distinct sections, so each section addresses a separate concern. At its essence, Separation of concerns is about order. The overall goal of separation of concerns is to establish a well-organized system where each part fulfills a meaningful and intuitive role while maximizing its ability to adapt to change.

Separation of Concerns



Split into blocks of functionality
- Each covering a concern

More modular code
- Encapsulation within a module

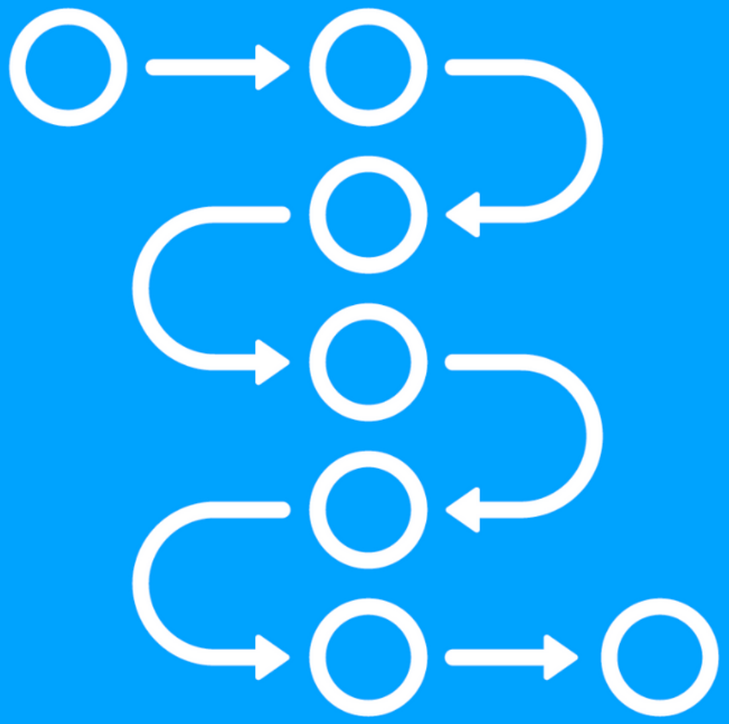
Typical layered application

Easier to maintain

Separation of Concerns

A software entity should have only single responsibility at its abstraction level.

A software system is at the highest level in the software hierarchy and should have a single dedicated purpose. For example, there can be an e-commerce or payroll software system. But there should not be a software system that handles both e-commerce and payroll-related activities. If you were a software vendor and had made an e-commerce software system, selling that to clients wanting an e-commerce solution would be easy. But if you had made a software system that encompasses both e-commerce and payroll functionality, it would be hard to sell that to customers wanting only an e-commerce solution because they might already have a payroll software system and, of course, don't want another one



DRY
(aka Don’t Repeat Yourself)

Less code repetition

Easier to make changes

Persistance Ignorance

The principle of Persistence Ignorance (PI) holds that classes modeling the business domain in a software application should not be impacted by how they might be persisted. Thus, their design should reflect as closely as possible the ideal design needed to solve the business problem at hand, and should not be tainted by concerns related to how the objects' state is saved and later retrieved.

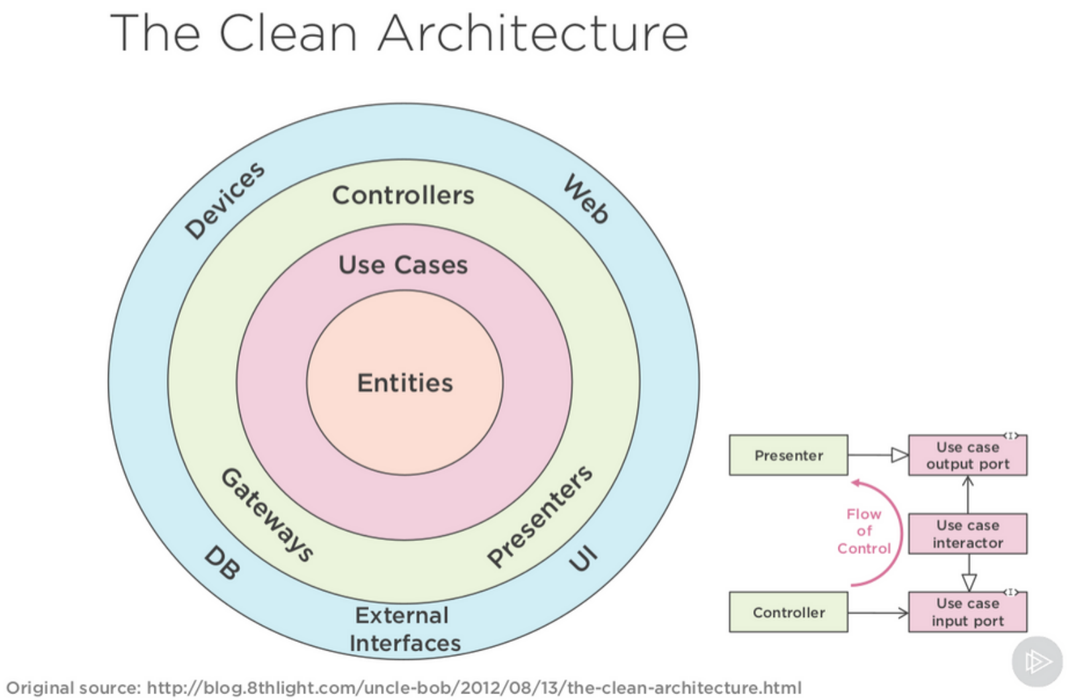
Persistence Ignorance

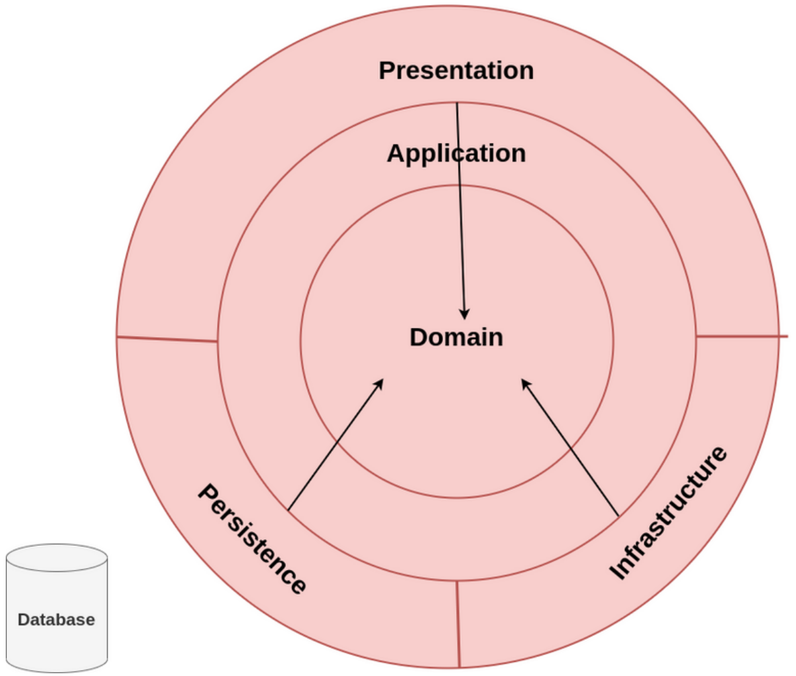
POCO

Domain classes shouldn't be impacted by how they are persisted

Typically requires base class or attributes

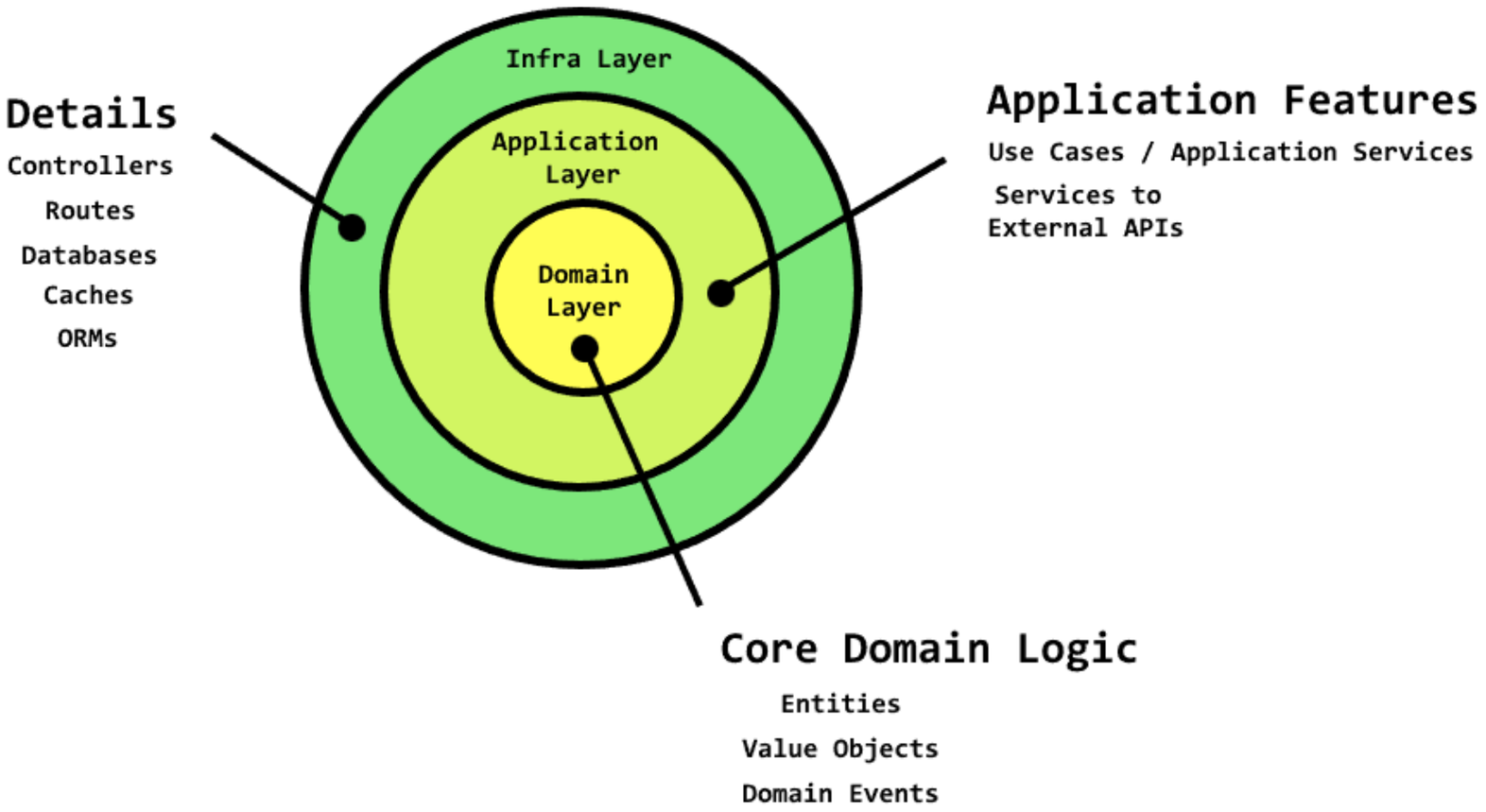
Architectural Layers



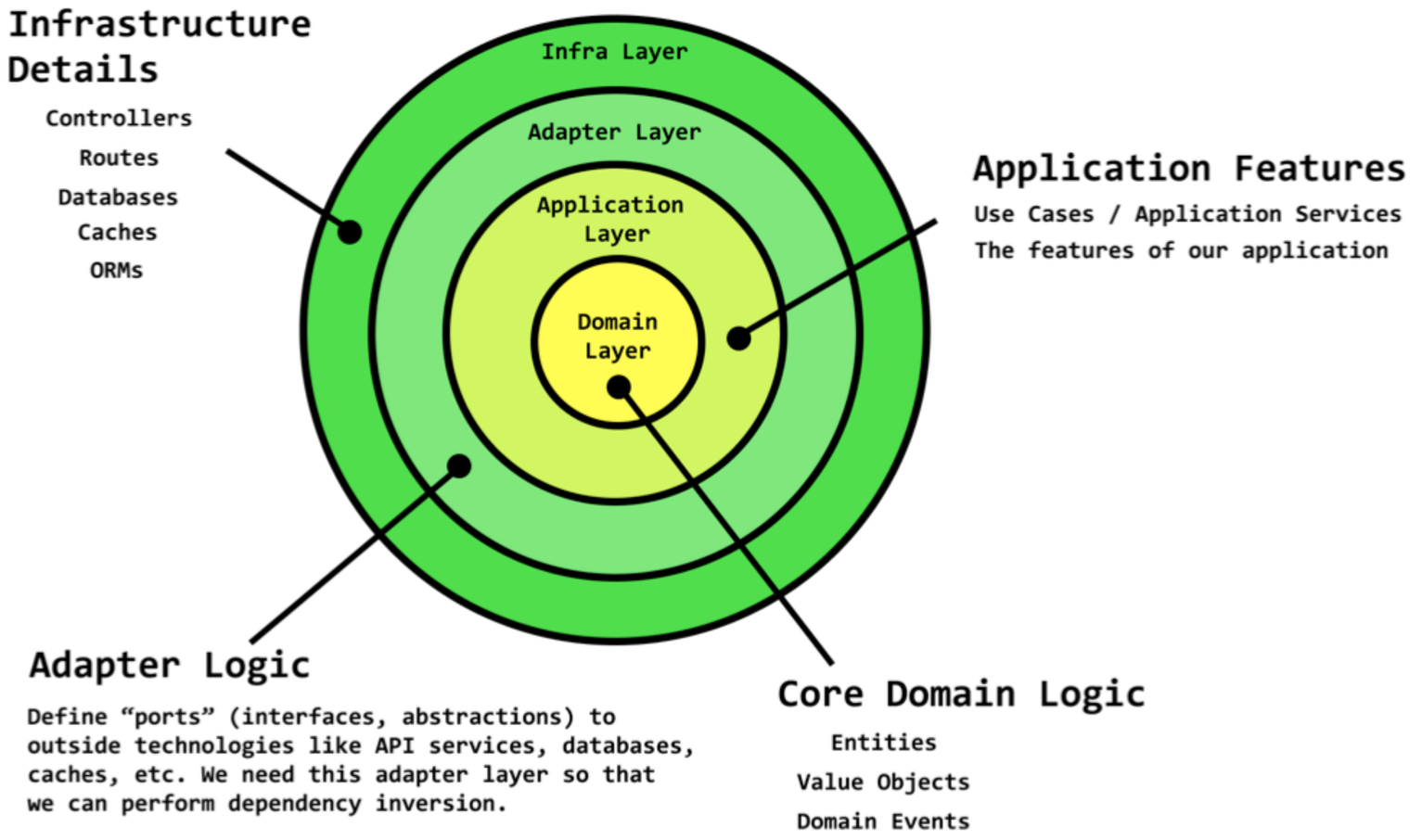


- The Domain, Application, and Infrastructure Layer
- The Dependency Rural

Domain Layers



Clean Structure

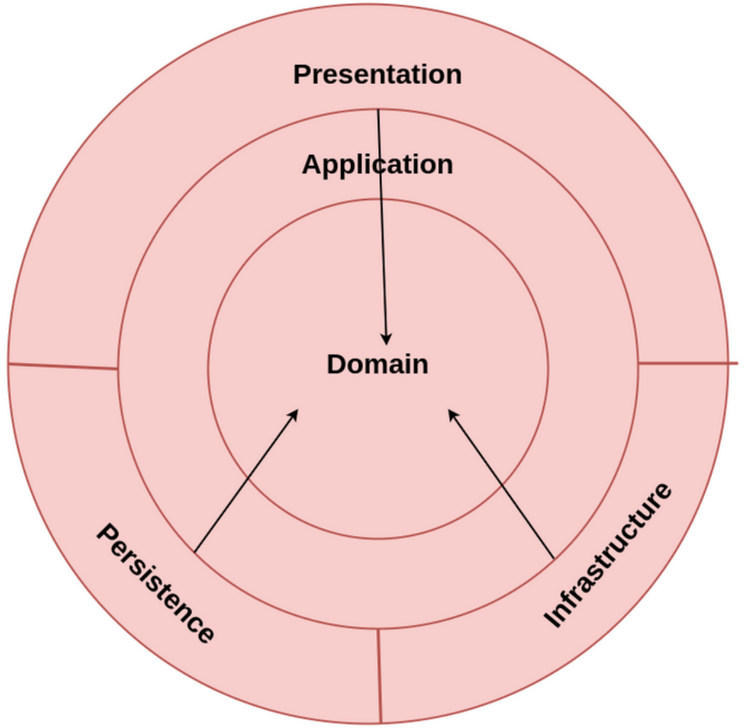


Domain Layers

The domain layer is also a layer without any infrastructure code. It contains:

- Entities
- Value objects
- Domain events
- Entity (write model) repository interfaces
- Domain services

These domain objects should be considered implementation details of the
application layer. In fact, most of these details should stay behind the application layer.



Domain Layers

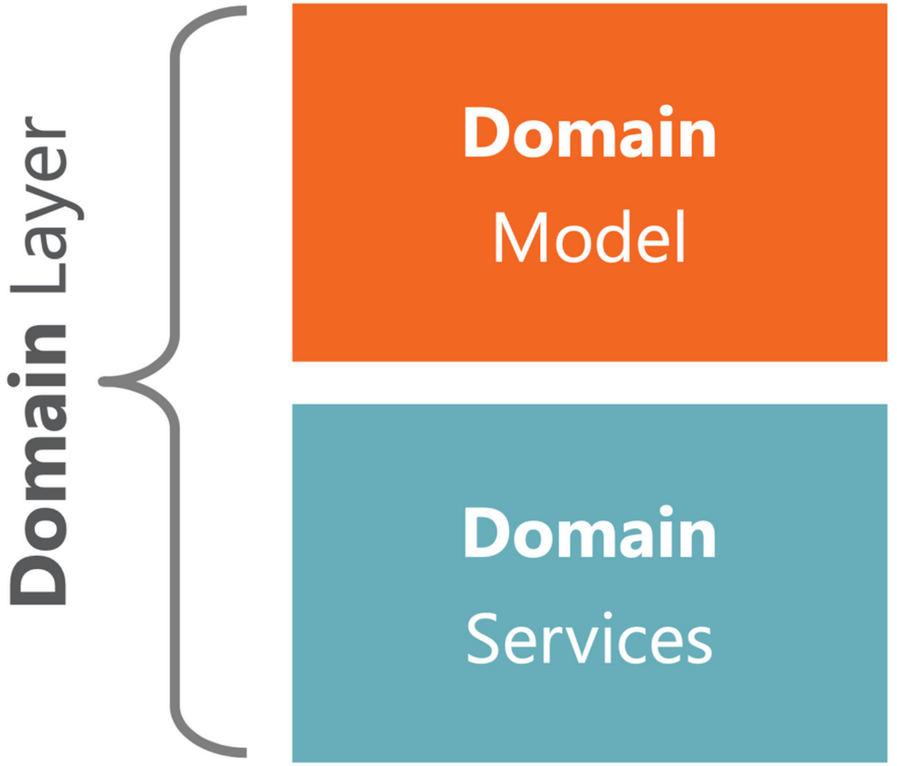
Implementing the Domain Layer
Classic Approach

Object-oriented Model

Domain Services

Domain Layers

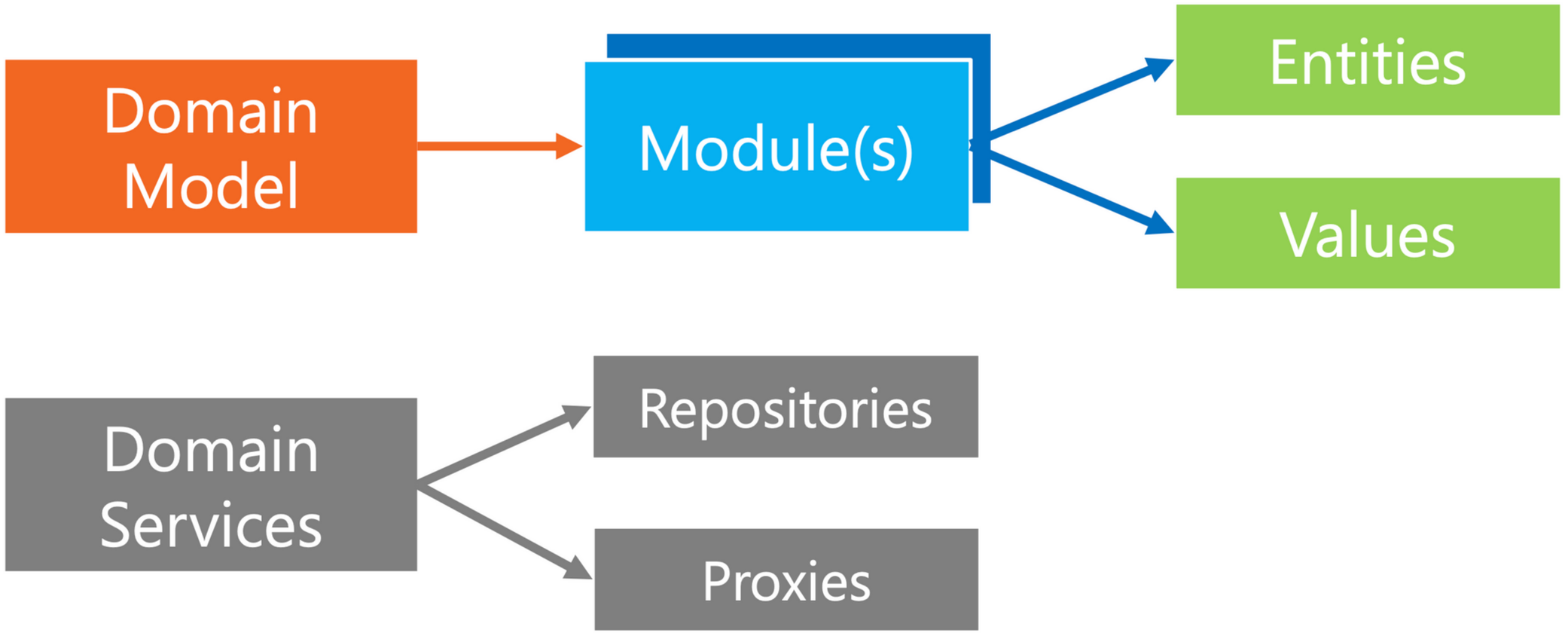
Domain Model Supporting Architecture



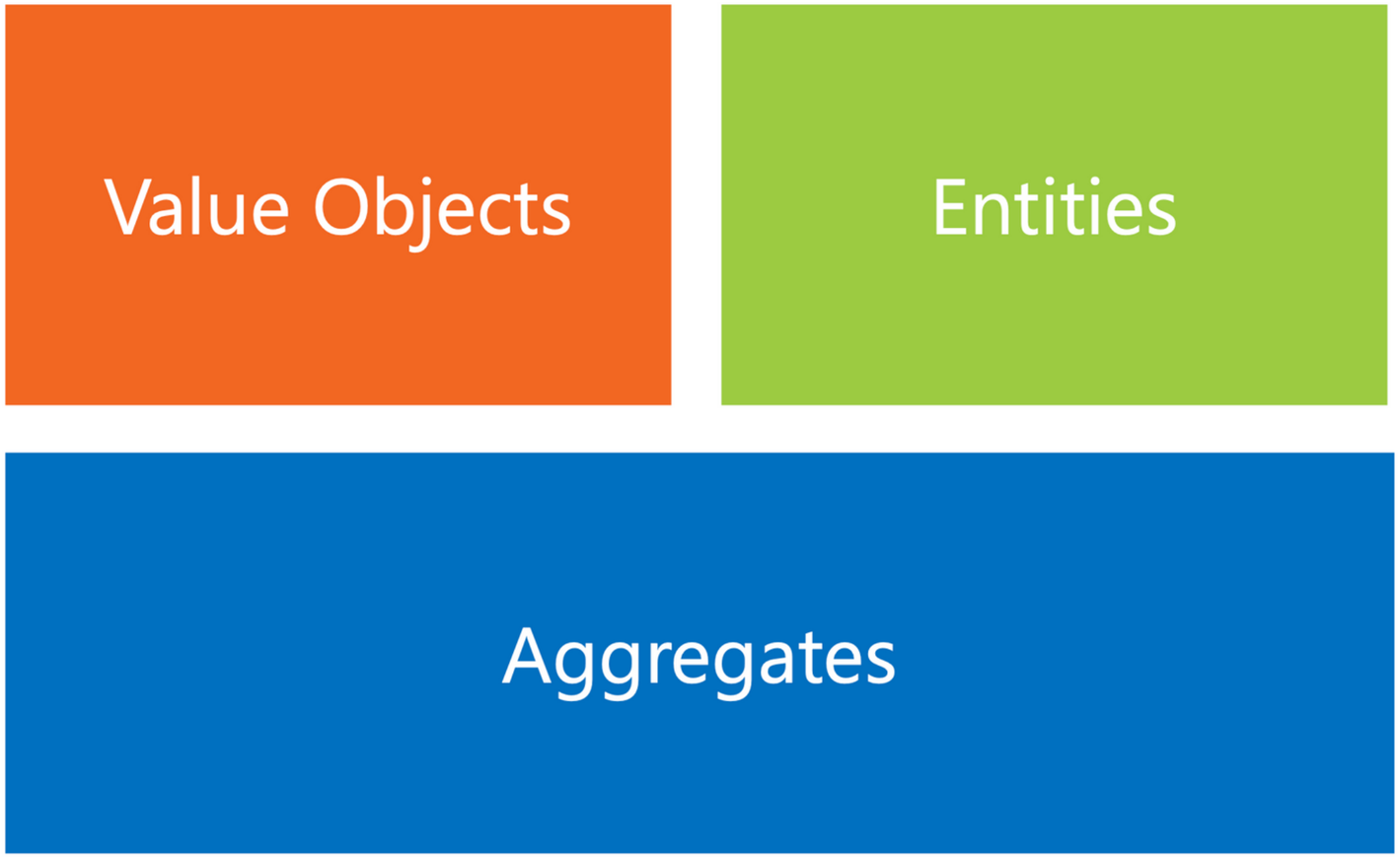
- Aggregates
- Entities
- Value types
- Factories

- Cross-aggregate behavior
- Repositories
- External services

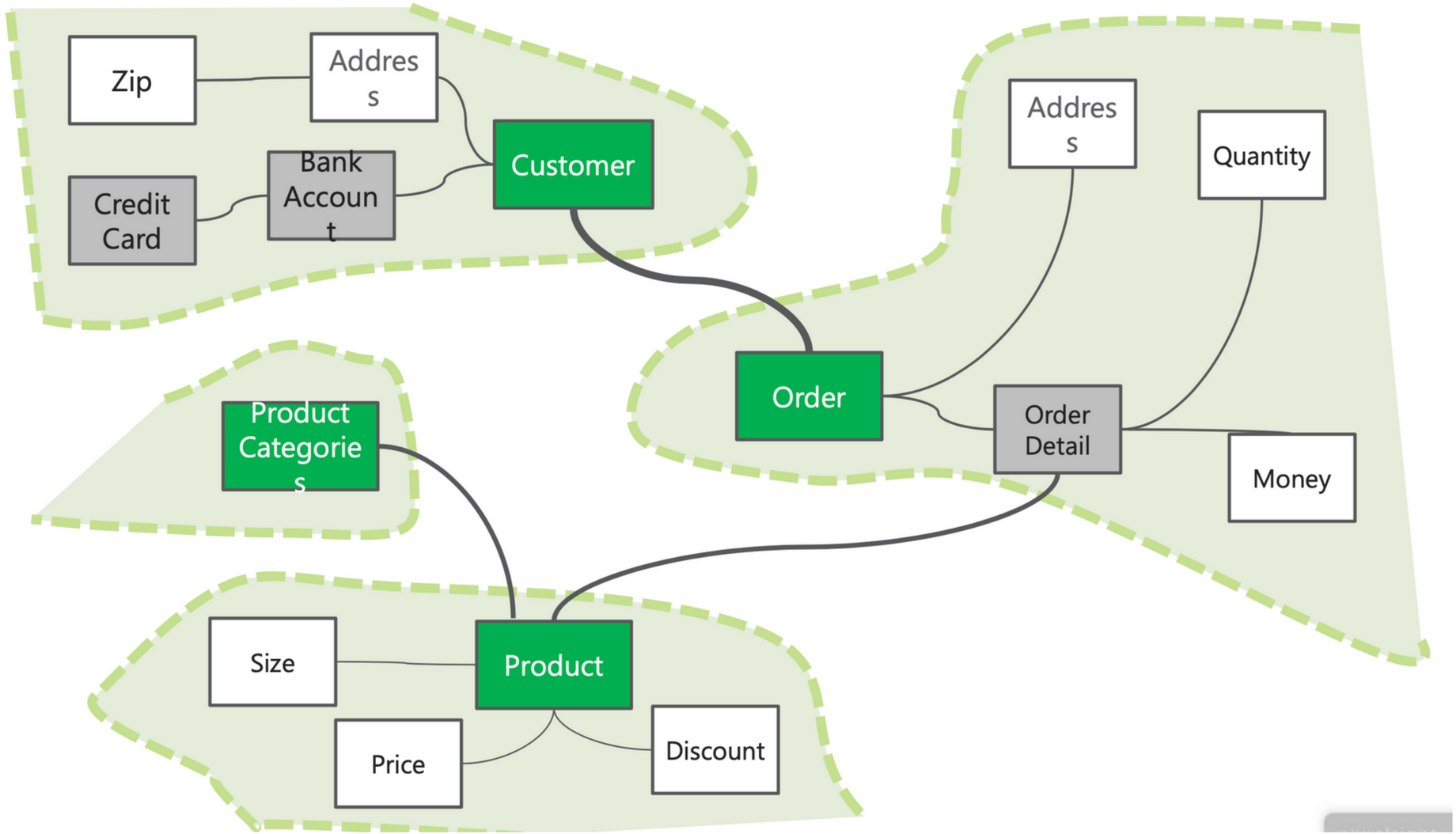
Domain Layer



Aspects of a Domain Model **Module**



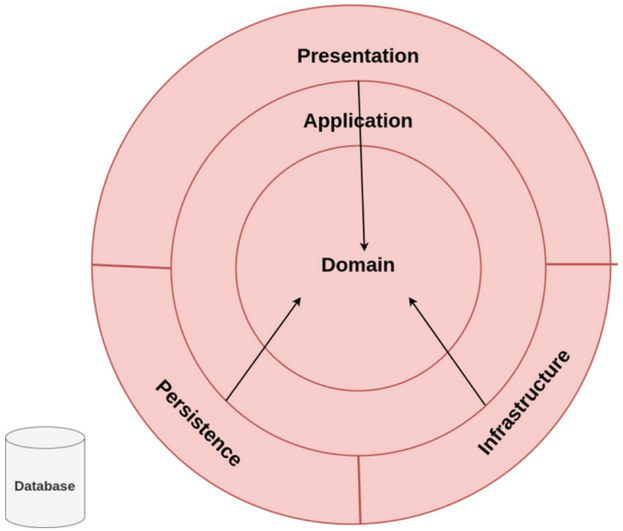
Domain Layers



Infrastructure Layers

Every bit of infrastructure code we've encountered so far is going to end up
in the infrastructure layer. This includes:

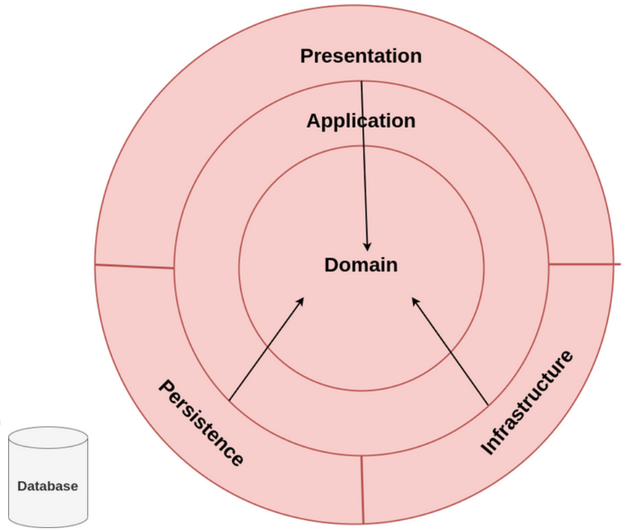
- Web controllers
- CLI commands
- Write and read model repository implementations
- Services that connect to external systems, like a remote API, or the file system
- Services that use the current time or generate random data



Infrastructure Layers

Every bit of infrastructure code we've encountered so far is going to end up
in the infrastructure layer. This includes:

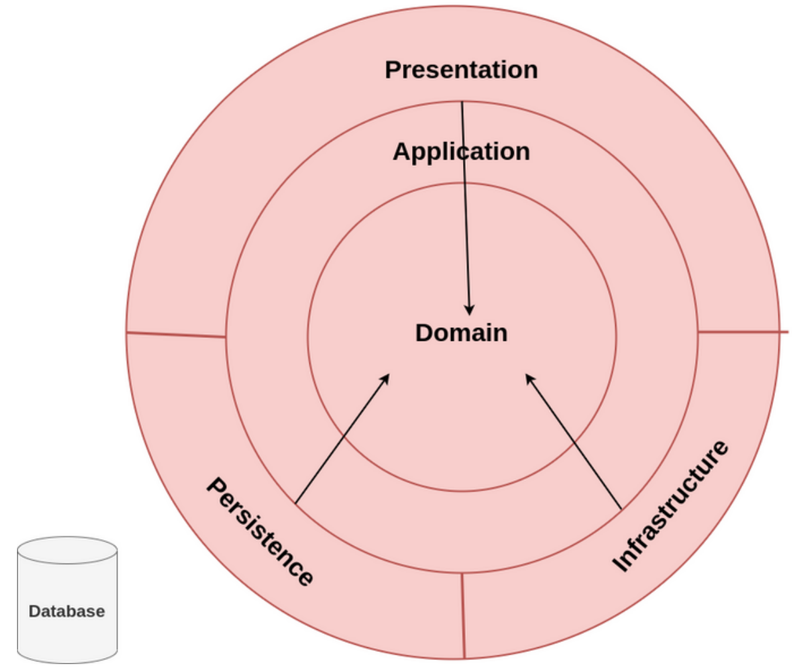
- Web controllers
- CLI commands
- Write and read model repository implementations
- Services that connect to external systems, like a remote API, or the file system
- Services that use the current time or generate random data



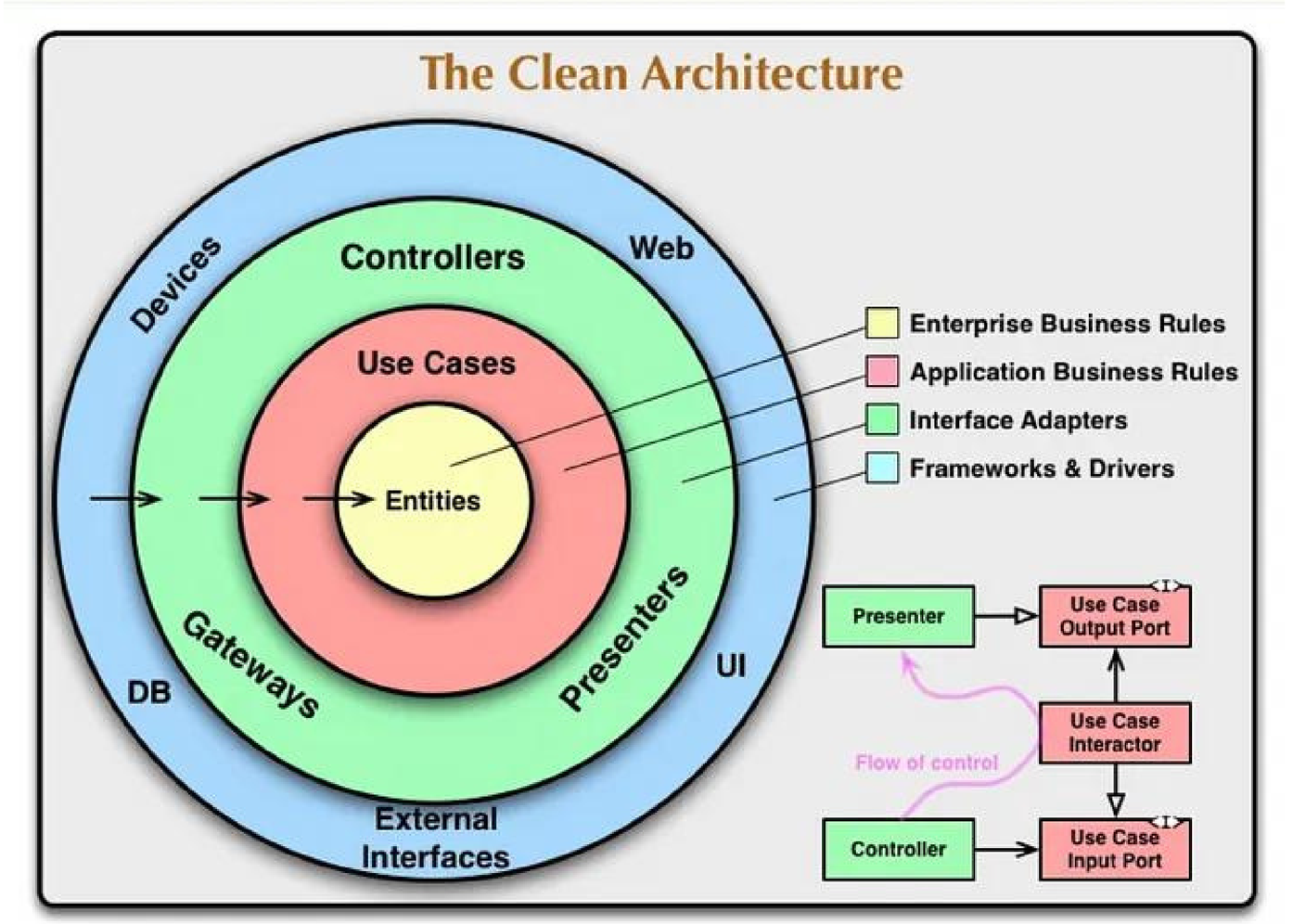
Application Layers

The application layer is the first layer that's free of infrastructure code. This
layer includes:

- Application services/command handlers, and command DTOs
- View model repository interfaces, and view model DTOs
- Event subscribers that listen to domain events and perform secondary
tasks
- Interfaces for infrastructure services



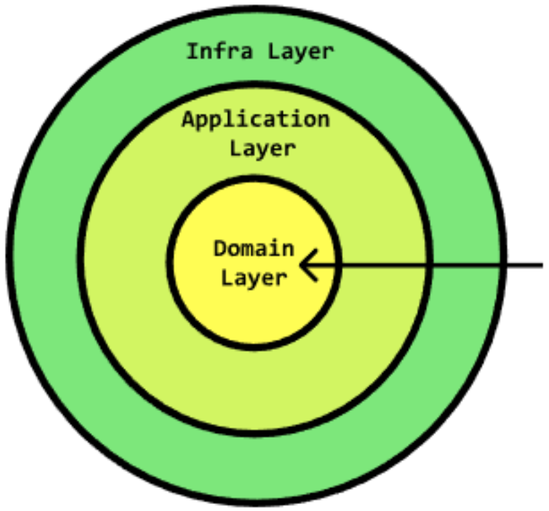
Clean Architecture

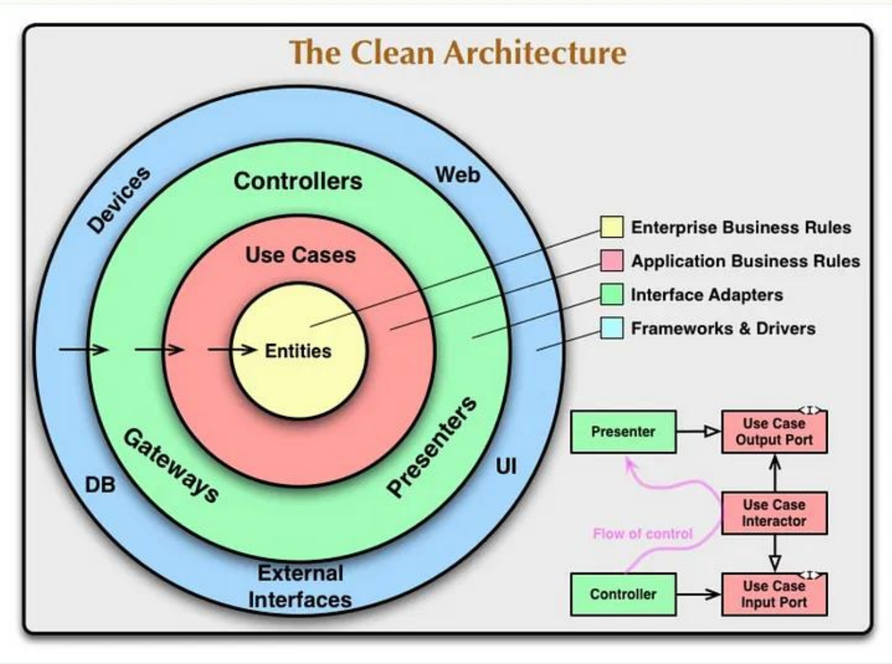


Dependencies Rule

The Dependency Rule states that the source code dependencies can only point inwards.

This means nothing in an inner circle can know anything at all about something in an outer circle. i.e. the inner circle shouldn't depend on anything in the outer circle. The Black arrows represented in the diagram show the dependency rule.





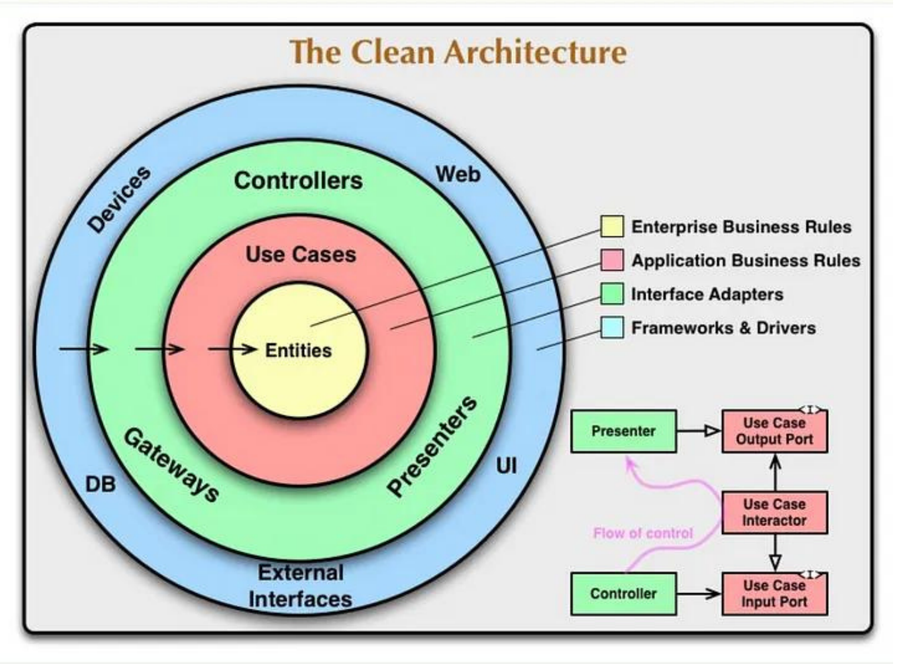
Frameworks and Drives

Software areas that reside inside this layer are

- User Interface
- Database
- External Interfaces (eg: Native platform API)
- Web (eg: Network Request)
- Devices (eg: Printers and Scanners)

Generally, you don't write much code in this layer, other than glue code than glue code that communicates to the next circle inward.

The frameworks and drivers layer is where all the details go. The web is a detail. The database is a detail. We keep these things on the outside where they can do little harm.

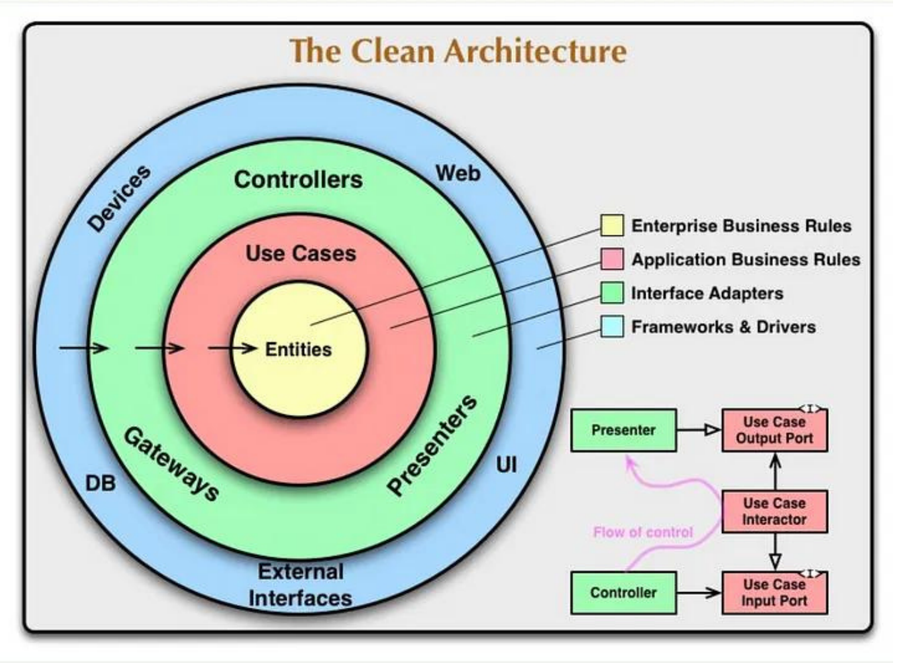


Interface Adapters

This layer holds

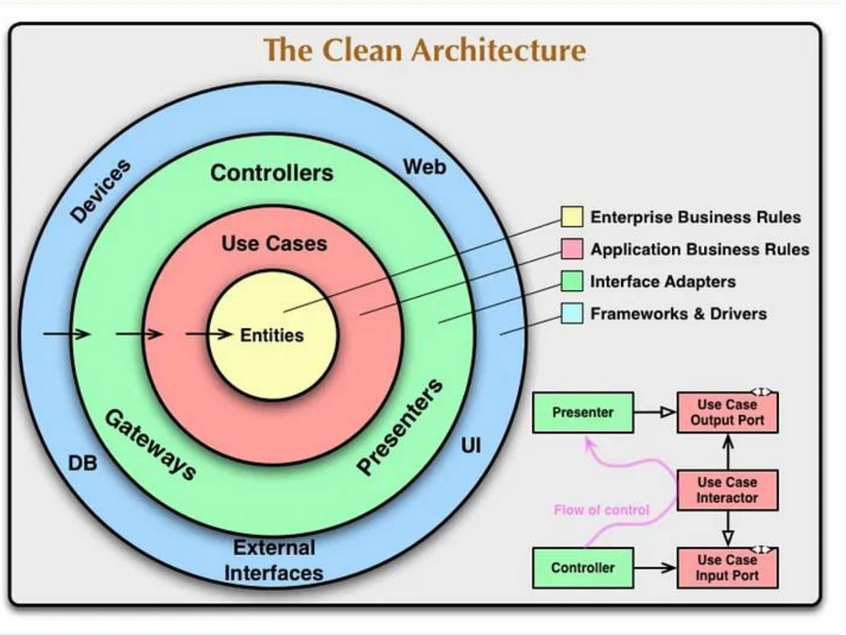
- Presenters (UI Logic, States)
- Controllers (Interface that holds methods needed by the application which is implemented by Web, Devices or External Interfaces)
- Gateways (Interface that holds every CRUD operation performed by the application, implemented by DB)

The software in the interface adapters layer is a set of adapters that convert data from the most convenient for some external agency such as the database or the web.



Entities

Entities encapsulate enterprise-wide Critical Business Rules. An Entity can be an object with methods, or it can be a set of data structures and functions.

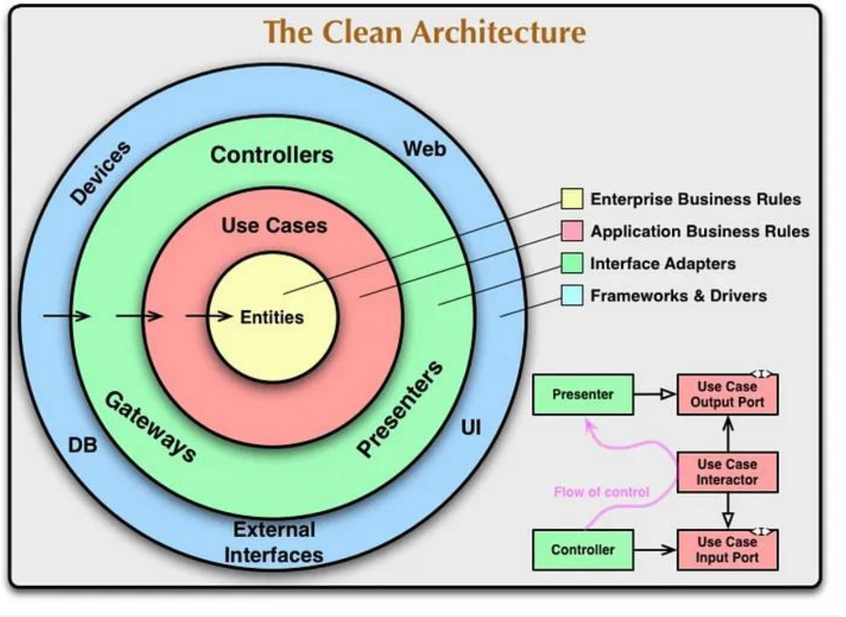


Enterprise Business Rules

This is the layer that holds core-business rules or domain-specific business rules. Also, this layer is the least prone to change.
Change in any outer layer doesn't affect this layer. Since Business Rules won't change often, change in this layer is very rare. This layer holds Entities.
An entity can either be a core data structure necessary for the business rules or an object with methods that hold business logic in it.

USE CASES

The software in the use cases layer contains application-specific business rules. Il encapsulates and implements all of the use cases of the system. These use cases orchestrate the flow of data to and from the entities, and direct those entities to use their critical Business Rules to achieve the goals of the uses.



DEMO