CLEAN ARCHITECTURE

Roman Efremenko

IMPORTANT

We speak about how code parts of your program interact with each other and don't talk about how you should organize your code on disk. I understand that it is an important question where you should put your code, but it is not the main topic of this presentation. However, I will say a couple of words about it.

Backend programmer

AGENDA

LOR

History of architecture patterns from 2002 to 2012.

Clean architecture scheme

Look at the scheme of Clean architecture and identify interaction flow

Basic blocks

Look at the basic blocks of clean architecture and set the restrictions

Pros and cons

Talk about pros and cons for Clean architecture and when we should use it

Software architecture refers to the high-level structure of a software system, which encompasses the design and organization of its components or modules, the relationships between them, and the principles guiding their interactions.


СЛАБОУМИЕ
И
ОТВАГА

Coupling & Cohesion

Low Coupling

Modules should be as independent as possible from other modules, so that changes to module don't heavily impact other modules.

High Cohesion

High cohesion implies that the elements within a module have a strong and focused relationship, and they work together to achieve a specific, well-defined purpose.

GREAT ARCHITECTURE

Great software architecture achieves a balance between low coupling and high cohesion. This balance leads to systems that are flexible, maintainable, and robust, making them more adaptable to changing requirements and easier to manage throughout their lifecycle

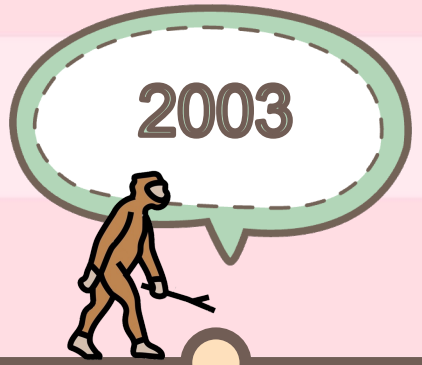
HEROES

ARCHITECTURE

TIMELINE

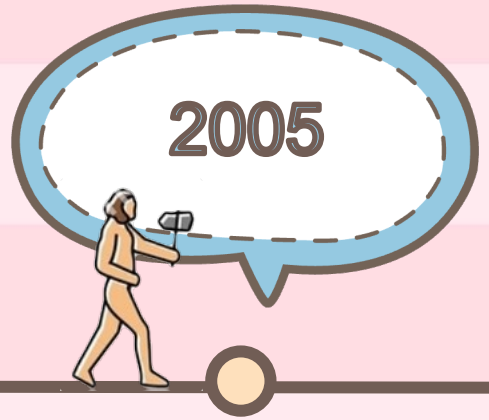
N-Layered Architecture

Martin Fowler start popularized N-Layered architecture emerged as a solution to manage the growing complexity of applications.



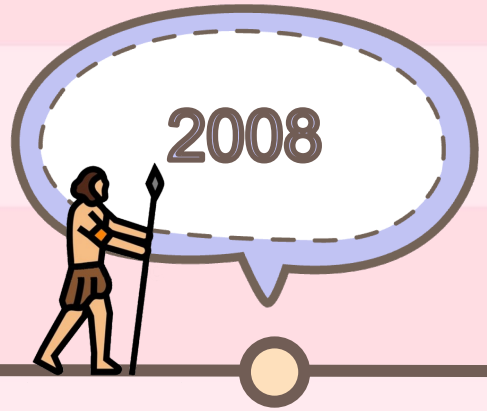
Domain Driven Design

Eric Evans advent of Domain-Driven Design (DDD). This introduced a paradigm shift in software architecture by placing a stronger emphasis on the Domain Model



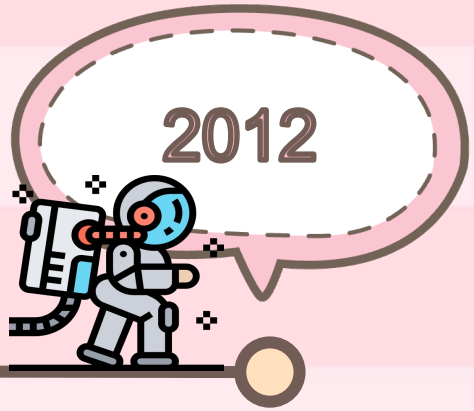
Hexagonal architecture

Alistair Cockburn got tired of rectangles, so he drew a hexagon and bring as new architecture also known as Ports and Adapters architecture.



Onion architecture

Jeffrey Palermo realizing his hatred of onions makes the whole world of programmers cry.



Clean architecture

Robert Martin, recognizing the hype around architecture and seizing the opportunity, consolidates all the ideas into one

GANG OF FOUR

Creational patterns

1. Abstract factory
2. Builder
3. Factory method
4. Prototype
5. Singleton
6. …

Structural patterns

1. Adapter
2. Bridge
3. Composite
4. Decorator
5. Façade
6. Proxy
7. …

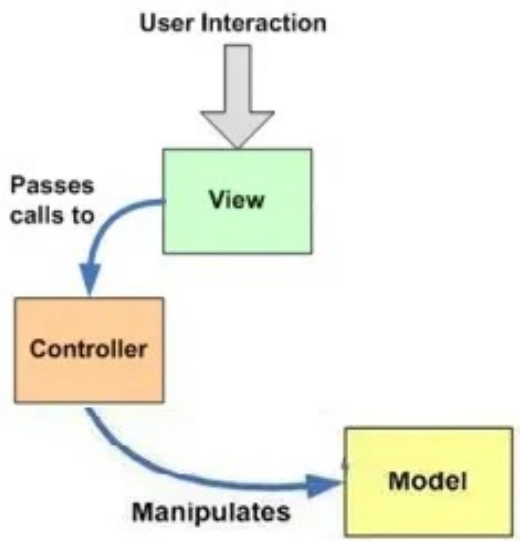
Behavioral patterns

1. Command
2. Interpreter
3. Visitor
4. Mediator
5. Strategy
6. …

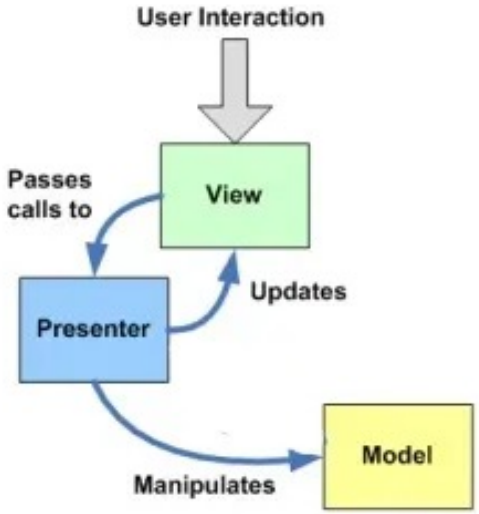
BIG BALL OF MUD

1979

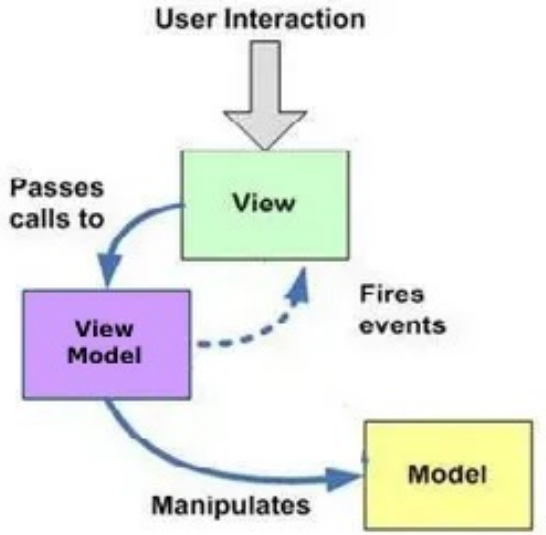
First attempts to organize



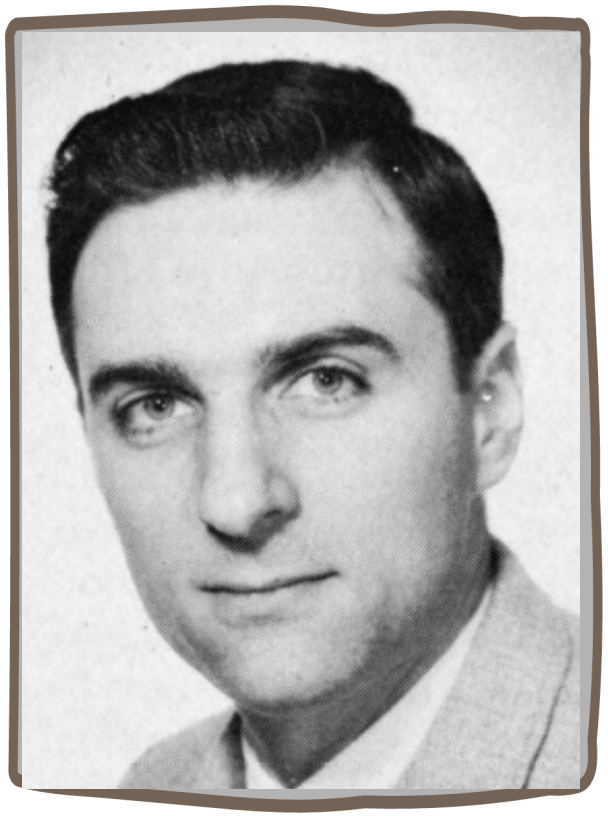
Model-View-Controller



Model-View-Presenter

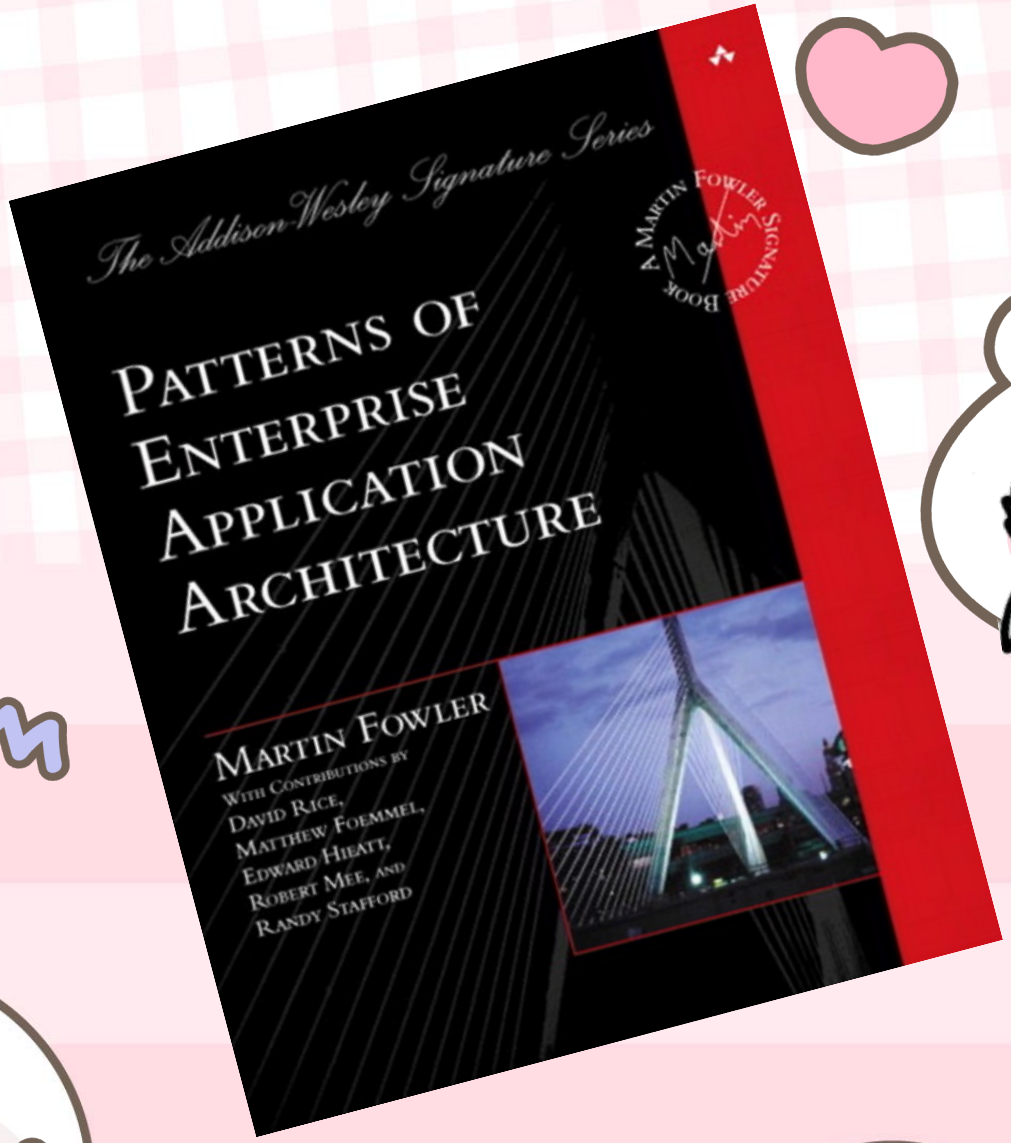


Model - View - ViewModel



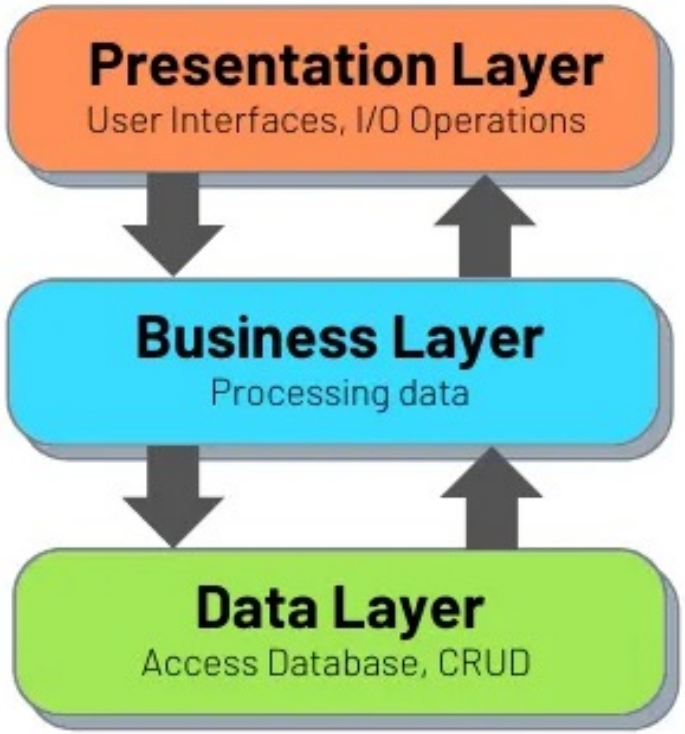
Melvin Conway

Organizations which design systems (in the broad sense used here) are constrained to produce designs which are copies of the communication structures of these organizations.

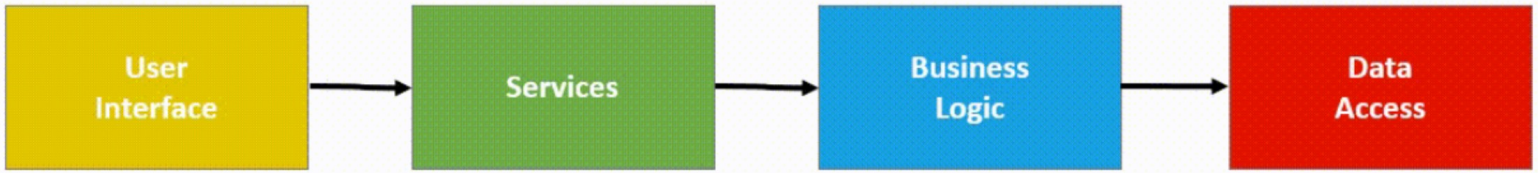


Martin Fowler

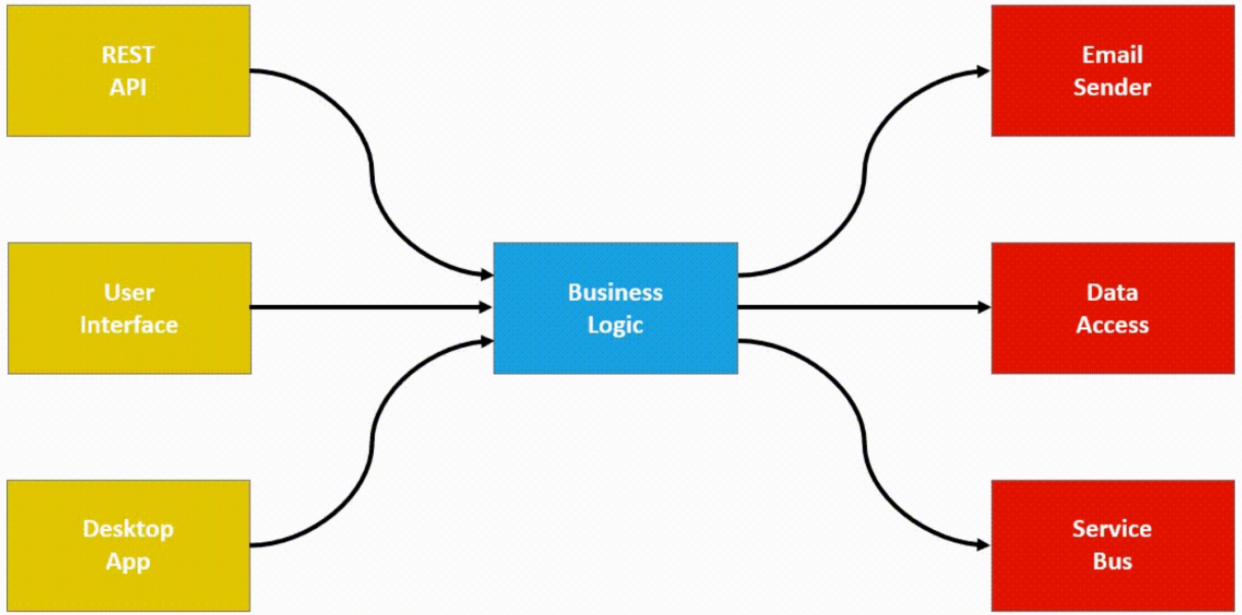
LAYERED

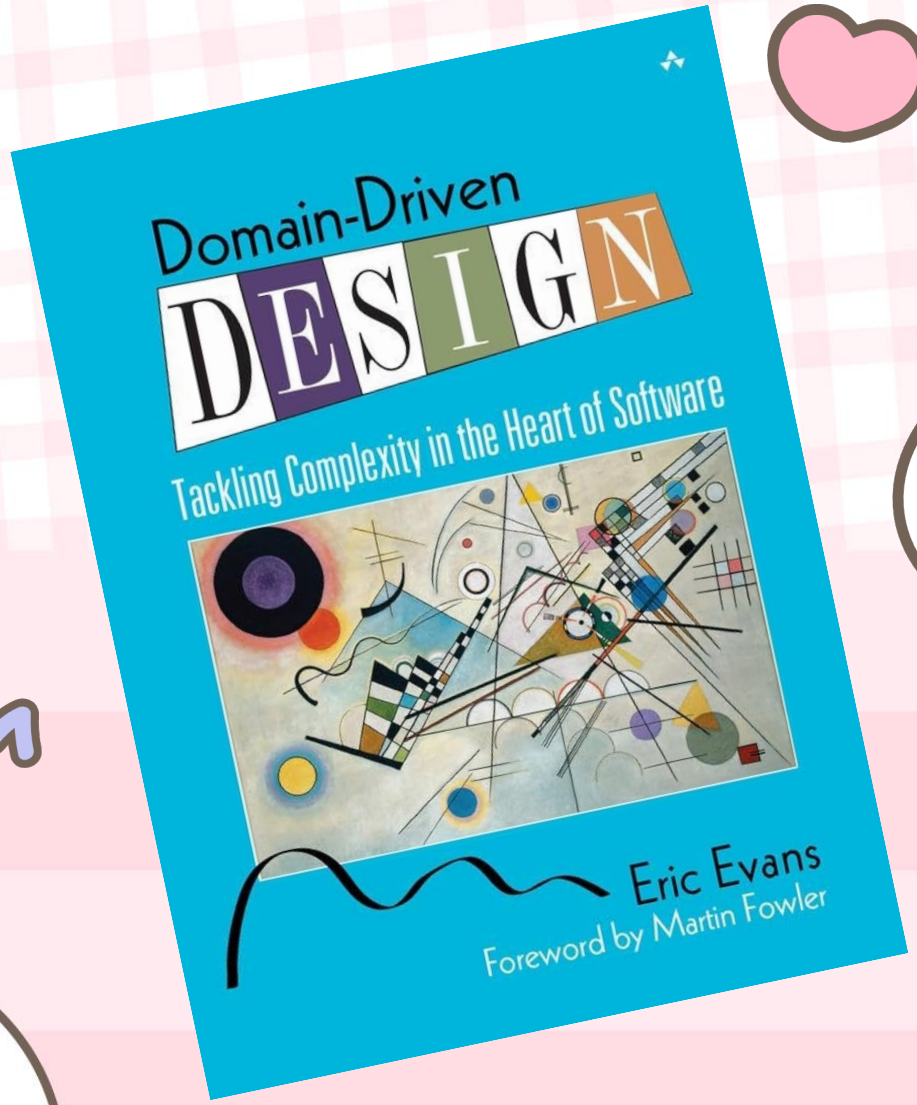

Presentation Layer
User Interfaces, I/O Operations
Business Layer
Processing data
Data Layer
Access Database, CRUD

LAYERED



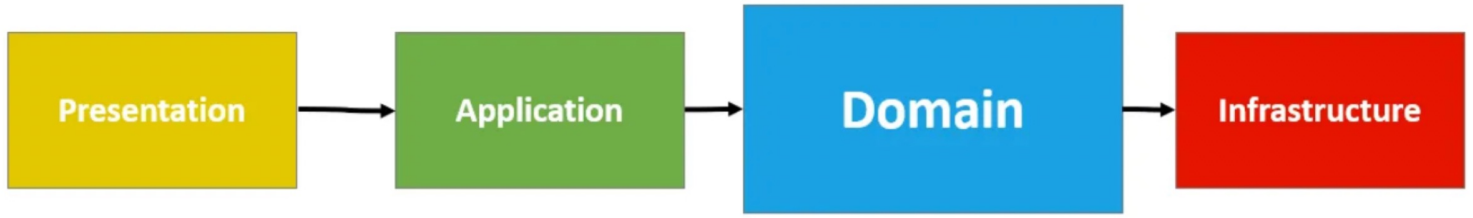
LAYERED


REST
API
User
Interface
Desktop
App
Business
Logic
Email
Sender
Data
Access
Service
Bus


Domain-Driven
DESIGN
Tackling Complexity in the Heart of Software
Eric Evans
Foreword by Martin Fowler


Eric Evans

DDD



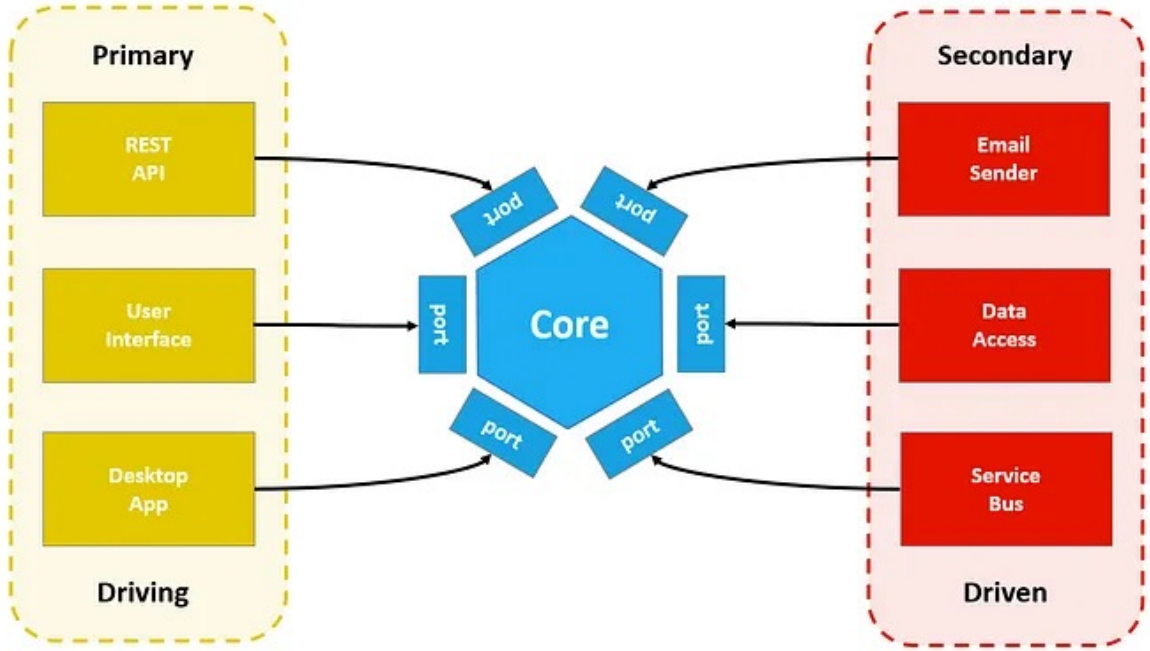
CONCEPT

1. Bounded context
2. Ubiquitous language
3. Value object
4. Entity
5. Aggregate
6. Domain service
7. Command handlers
8. ...

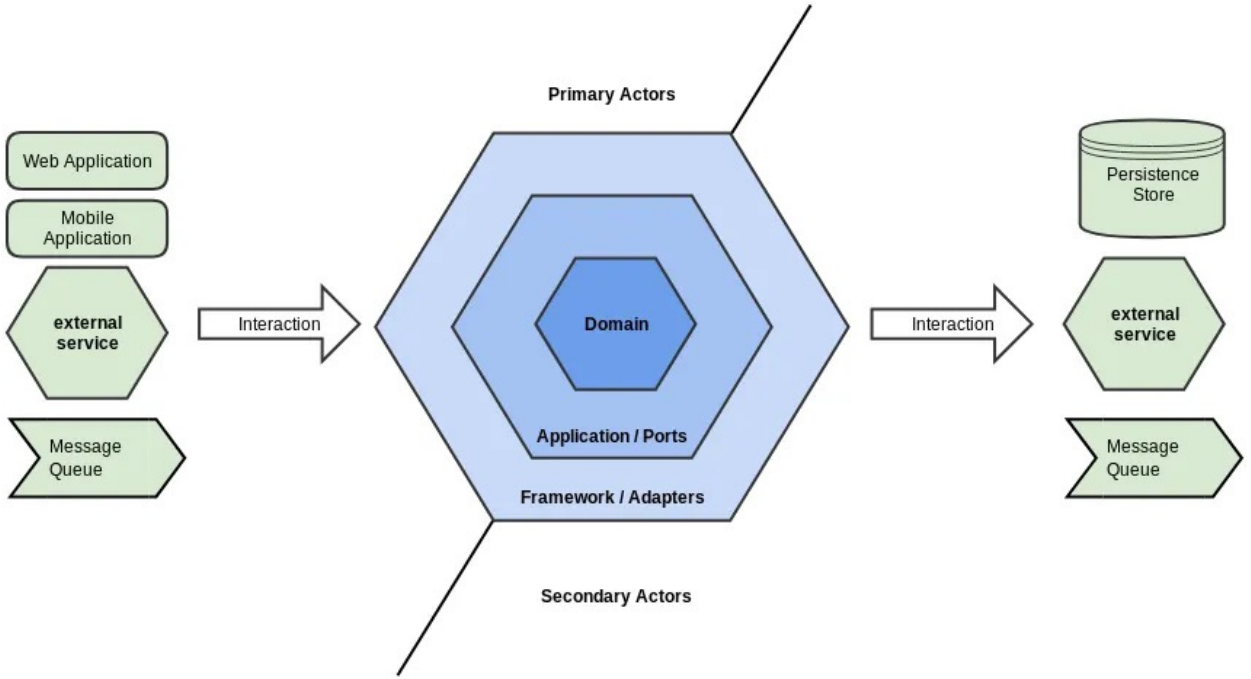
HEXAGONAL

Alistair Cockburn

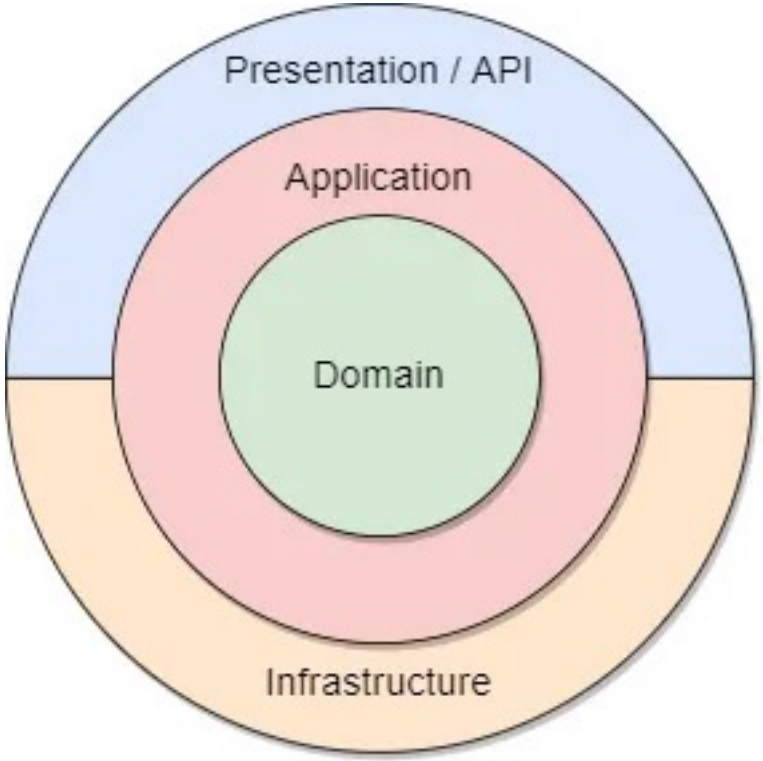
HEXAGONAL


Primary
REST API
User Interface
Desktop App
Driving
port
port
port
Core
port
port
port
Secondary
Email Sender
Data Access
Service Bus
Driven

HEXAGONAL

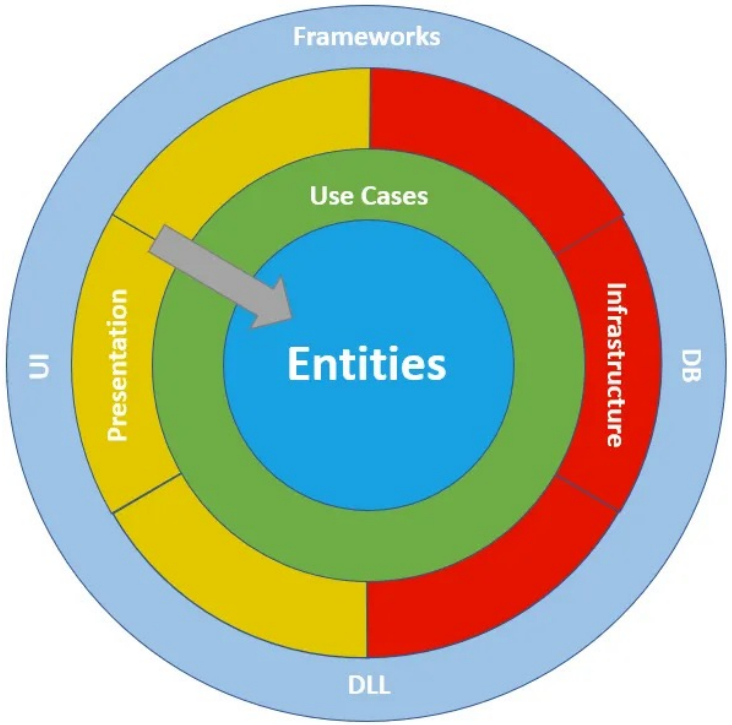

Primary Actors
Web Application
Mobile Application
external service
Message Queue
Interaction
Domain
Application / Ports
Framework / Adapters
Secondary Actors
Interaction
Persistence Store
external service
Message Queue

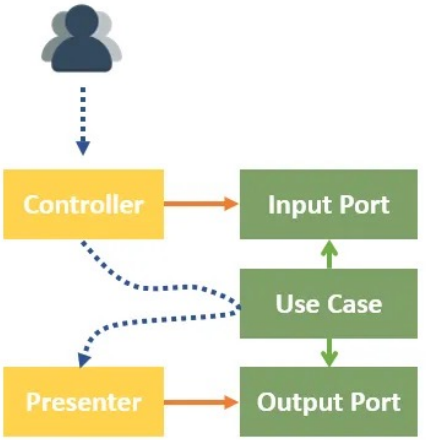
ONION



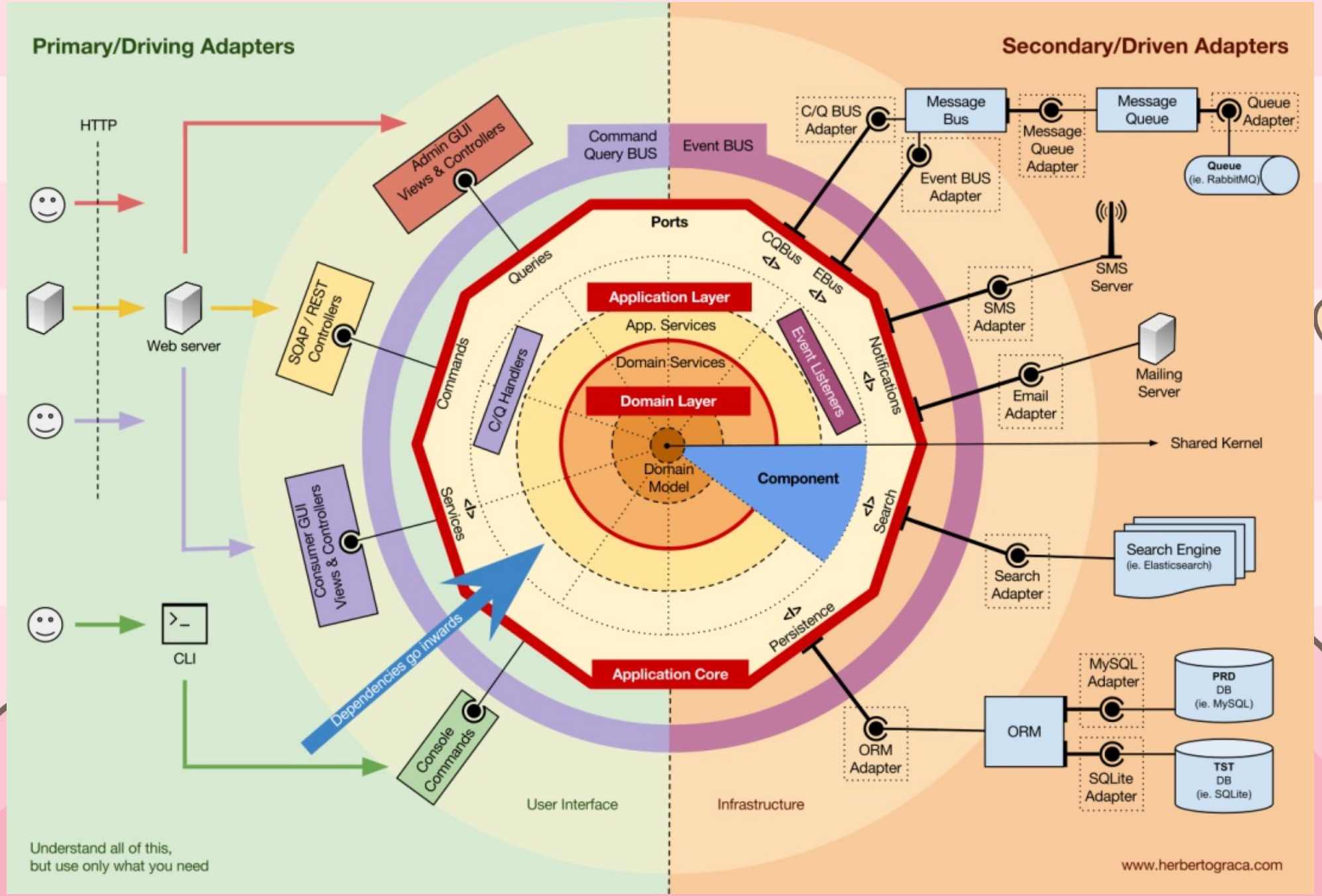
Jeffrey Palermo

CLEAN





Robert Martin


Primary/Driving Adapters
Secondary/Driven Adapters
HTTP
Web server
CLI
Admin GUI Views & Controllers
SOAP / REST Controllers
Consumer GUI Views & Controllers
Console Commands
Command Query BUS
Event BUS
Ports
Queries
Commands
C/Q Handlers
Application Layer
App. Services
Domain Services
Domain Layer
Domain Model
Event Listeners
Services
Persistence
Search
Notifications
CQBus
EBus
Component
Application Core
Dependencies go inwards
C/Q BUS Adapter
Message Bus
Event BUS Adapter
Message Queue Adapter
Message Queue
Queue Adapter
Queue (ie. RabbitMQ)
SMS Adapter
SMS Server
Email Adapter
Mailing Server
Shared Kernel
Search Adapter
Search Engine (ie. Elasticsearch)
ORM Adapter
ORM
MySQL Adapter
PRD DB (ie. MySQL)
SQLite Adapter
TST DB (ie. SQLite)
User Interface
Infrastructure
Understand all of this, but use only what you need
www.herbertograca.com

DOMAIN

Value object

A value object is a type of domain object that represents a simple or atomic value in the domain

Aggregate

Cluster of domain objects that can be treated as a single unit. It encapsulates entities and value objects which conceptually belong together.

Entity

An entity is a type of domain object that represents a complex or composite concept or entity in the domain

Domain Service

A object that performs some domain-specific operation or logic that does not belong to any entity or value object

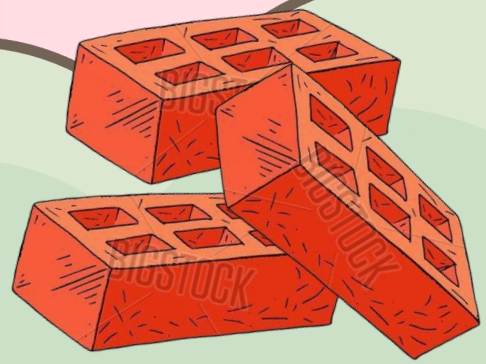
Event

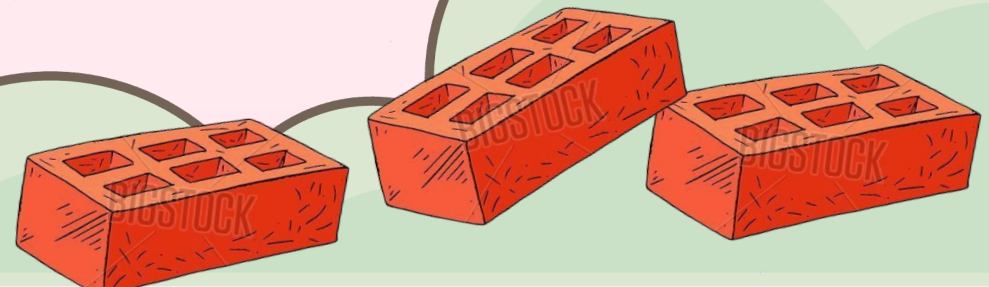
A object that represents something meaningful or significant that happened in the domain.

VALUE OBJECT

A value object is a type of domain object that represents a simple or atomic value in the domain

- Should not allow its attributes to be changed after creation
- Implement equality based on its attributes, rather than on its identity or reference
- Should be safe to share across multiple domain objects or contexts, without causing side effects or inconsistencies
- Explicitly defines and enforces important constraints (invariants).





ENTITY

An entity is a type of domain object that represents a complex or composite concept or entity in the domain

- Contain Domain business logic
- Have an identity that defines it and makes it distinguishable from others
- Can contain other objects, such as other entities or value objects.
- Are responsible for collecting all the understanding of state and how it changes in the same place
- Responsible for the coordination of operations on the objects it owns
- Domain entities data should be modelled to accommodate business logic, not some database schema.
- Must be consistent - maintain its invariants and rules at all times, regardless of its state changes

AGGREGATE

An aggregate is a cluster of related entities and value objects that form a unit of consistency and transactional integrity

- Aggregates help to simplify the domain model by gathering multiple domain objects under a single abstraction.
- Aggregate root has global identity (UUID / GUID / primary key). Entities inside the aggregate boundary have local identities, unique only within the Aggregate.
- Aggregate root is a gateway to entire aggregate. Any references from outside the aggregate should **only** go to the aggregate root.
- Any operations on an aggregate must be transactional operations.
- Only Aggregate Roots can be obtained directly with database queries. Everything else must be done through traversal.
- Similar to Entities, aggregates must protect their invariants through entire lifecycle. When a change to any object within the Aggregate boundary is committed, all invariants of the whole Aggregate must be satisfied. (*Consistency Boundary*).
- Aggregates can publish Domain Events

DOMAIN EVENT

A domain event is an object that represents something meaningful or significant that happened in the domain. A domain event captures the state and context of the event, and may trigger some actions or reactions in response to it.

- Domain events are just messages pushed to an in-memory Domain Event dispatcher.
- Domain Events are used to prevent coupling between domain aggregates
- Domain Events are ordinarily immutable, as they're a record of something in the past
- Domain Events describe what happened in the domain, using a clear and expressive name and attributes
- Domain Events should be published or broadcasted to interested parties or subscribers, using an event bus, a message broker, or other mechanisms
- Domain Events may trigger some actions or reactions in response to it, such as updating the domain state, sending notifications, invoking services, etc.

INTEGRATION EVENT

An integration event is an out-of-process communication, where a Domain Event is sent to an external domain (bounded context), for example, to an external microservice.

- Integration Events usually should be published only after all Domain Events finished executing and saving all changes to the database
- Integration event publishing using external message broker/ event bus like Kafka or Rabbit MQ

SERVICE

A domain service is an object that performs some domain-specific operation or logic that is not a natural responsibility of an ENTITY or VALUE OBJECT.

- Domain Service should not have any state or dependencies of its own, and should only rely on the parameters or inputs provided to it
- Domain Service are used when putting the logic on a particular Entity would break encapsulation and require the Entity to know about things it really shouldn't be concerned with.
- Domain services operate only on types belonging to the Domain. They contain meaningful concepts that can be found within the Ubiquitous Language. They hold operations that don't fit well into Value Objects or Entities.

APPLICATION

Usecase

Usecases are used to orchestrate the steps required to fulfill the commands imposed by the client.

CQRS

CQRS is a design principle that states that a method is either a COMMAND that performs an action OR a QUERY that returns data to the caller, but never both

Ports

A port is an interface that facilitates communication between an application and an external system using a specific protocol.

USECASE

Application Services (also called "Workflow Services", "Use Cases", "Interactors", etc.) are used to orchestrate the steps required to fulfill the commands imposed by the client

- Typically used to orchestrate how the outside world interacts with your application and performs tasks required by the end users
- Contain no domain-specific business logic
- Operate on scalar types, transforming them into Domain types.
- Uses ports to declare dependencies on infrastructural services/adapters required to execute domain logic
- Fetch domain Entities/Aggregates (or anything else) from database/external APIs
- Execute domain logic on those Entities/Aggregates. In case of working with multiple Entities/Aggregates, use a Domain Service to orchestrate them.
- Should not depend on other application services since it may cause problems (like cyclic dependencies)

CQRS

CQRS is a design principle that states that a method is either a COMMAND that performs an action OR a QUERY that returns data to the caller, but never both

Command

- Command is an object that signals user intent, for example *CreateUserCommand*
- Commands are used for state-changing actions, like creating new user and saving it to the database. Create, Update and Delete operations are considered as state-changing.
- Data retrieval is responsibility of Queries, so Command methods should not return business data.
- To execute a command you can use a Command Bus instead of importing a service directly. This will decouple a command Invoker from a Receiver, so you can send your commands from anywhere without creating coupling.

CQRS

CQRS is a design principle that states that a method is either a COMMAND that performs an action OR a QUERY that returns data to the caller, but never both

Query

- Query is just a data retrieval operation and should not make any state changes (like writes to the database, files, third party APIs, etc.)
- Similarly to Commands, Queries can use a Query Bus if needed

PORT

A port is an interface that facilitates communication between an application and an external system using a specific protocol.

- Ports are basically just interfaces that define what has to be done and don't care about how it's done.
- Ports should be created to fit the Domain needs, not simply mimic the tools APIs
- When designing ports, remember the Interface segregation principle. Split large interfaces into smaller ones when it makes sense, but also keep in mind to not overdo it when not necessary.
- Ports can also help to delay decisions. The Domain layer can be implemented even before deciding what technologies (frameworks, databases etc.) will be used.
- Mock implementations can be passed to ports while testing. Mocking makes your tests faster and independent of the environment.

INFRASTRUCTURE

Repository

A repository is an abstraction that provides access to the aggregates or entities of a domain model.

Controller

Controllers handling user input and coordinating the flow of data between the user and the underlying business logic

DTO

DTO is a design pattern used to transfer data between software application subsystems or layers that may have different structures

REPOSITORY

A repository is an abstraction that provides access to the aggregates or entities of a domain model. A repository hides the details of how the aggregates or entities are stored, retrieved, or persisted, and provides a collection-like interface to manipulate them

- Repository define an interface that specifies what operations are available for the aggregates or entities, without exposing how they are implemented.
- Repository provide one or more implementations that realize the interface using different technologies or mechanisms, such as databases, files, web services, etc

CONTROLLER

This layer is responsible for translating data between the external world (such as user input from the UI) and the use cases or application-specific business rules. The controller receives user input, processes it if necessary, and then delegates the work to the appropriate use case or interactor.

- Convert data from the use cases into a format suitable for the external world
- Controller define an interface to communicate between external framework and business layer

DTO

A repository is an abstraction that provides access to the aggregates or entities of a domain model. A repository hides the details of how the aggregates or entities are stored, retrieved, or persisted, and provides a collection-like interface to manipulate them

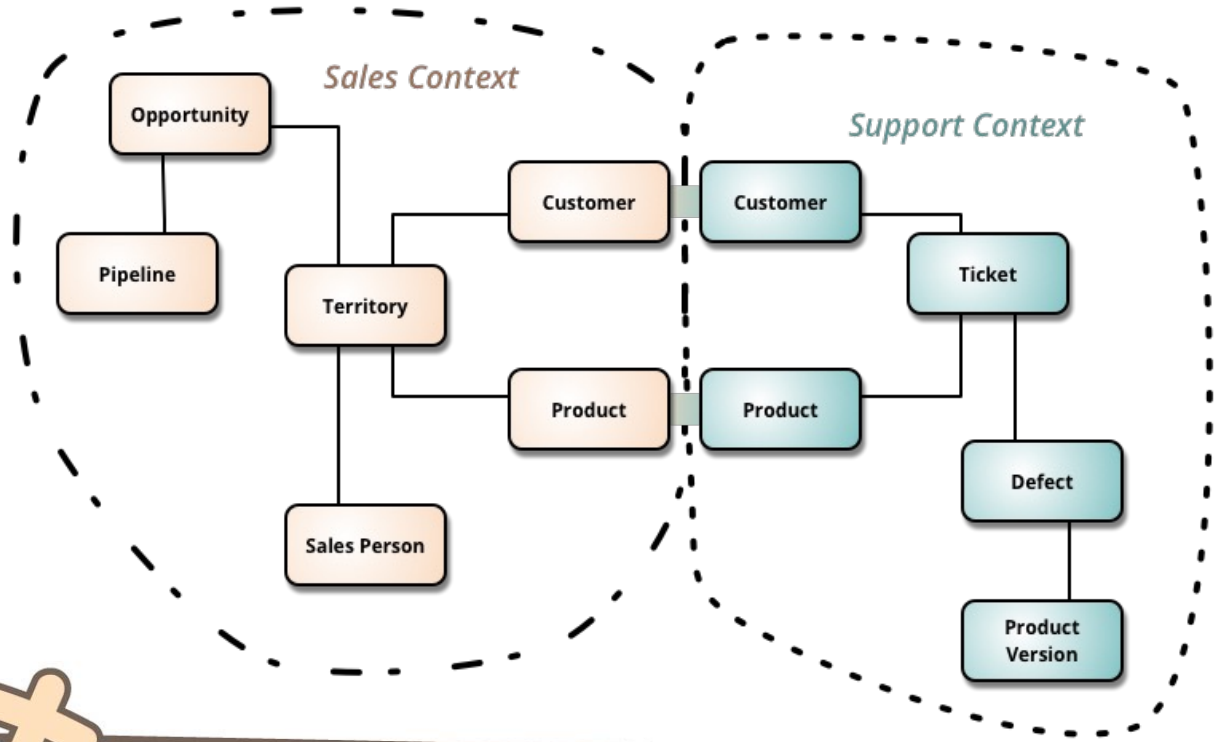
- DTOs help in transferring data between the domain layer and other layers without exposing the internal details of the domain objects.
- We use DTO to translate data between these different models, preventing the corruption of the domain layer.
- DTOs provide a way to expose specific data from these domain objects without compromising their immutability or exposing internal details.

UBIQUITOUS LANGUAGE

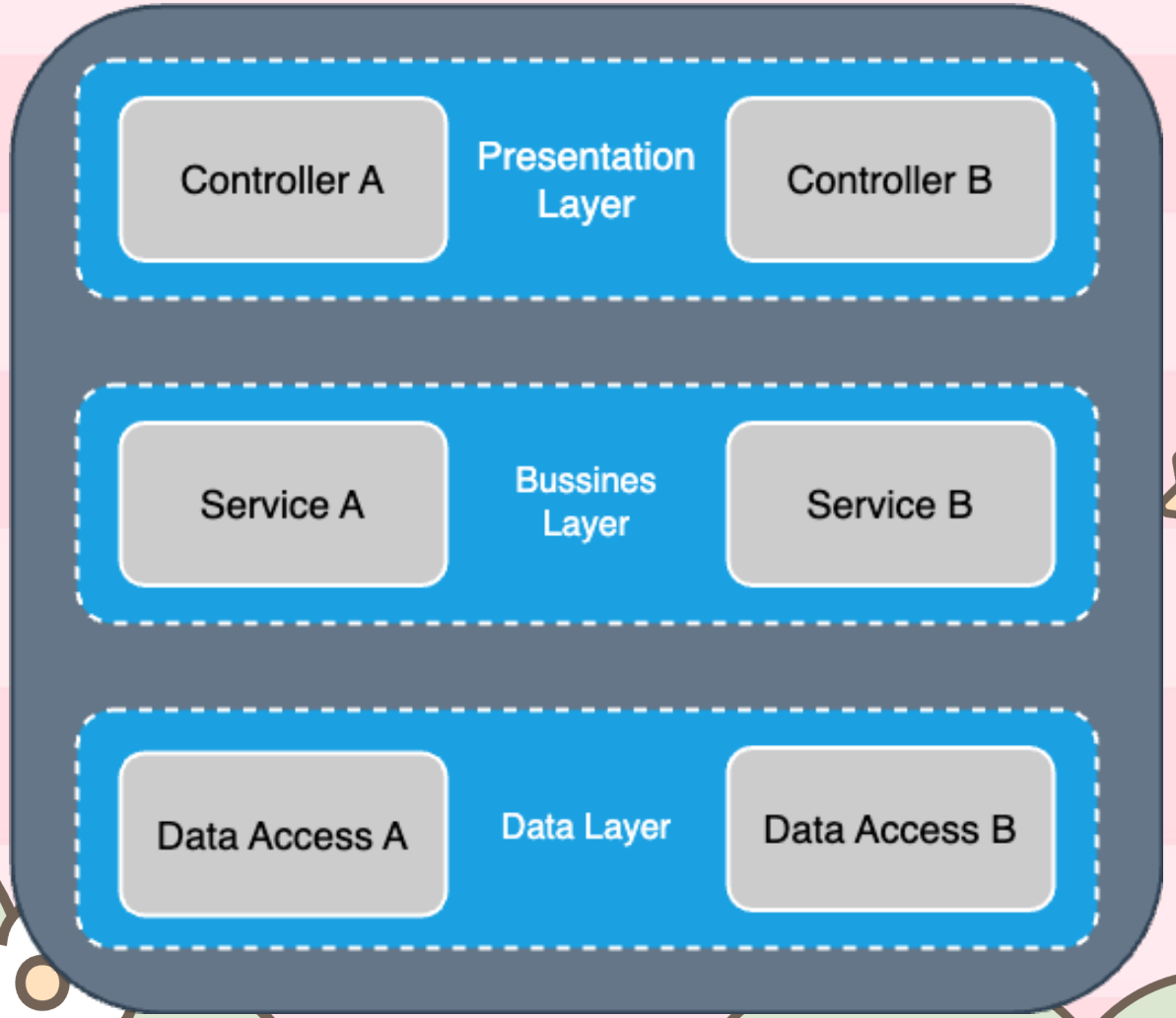
A ubiquitous language is a shared language used by all stakeholders to communicate about the domain. This involves defining a common vocabulary and set of concepts that can be used by developers, domain experts, and stakeholders to discuss the domain.

- The language should be based on the domain model and its behavior, and should be easy to understand and use.
- The language should be used consistently across the entire development process, from requirements gathering to design, implementation, and testing. This helps ensure that everyone involved in the project is on the same page and can communicate effectively

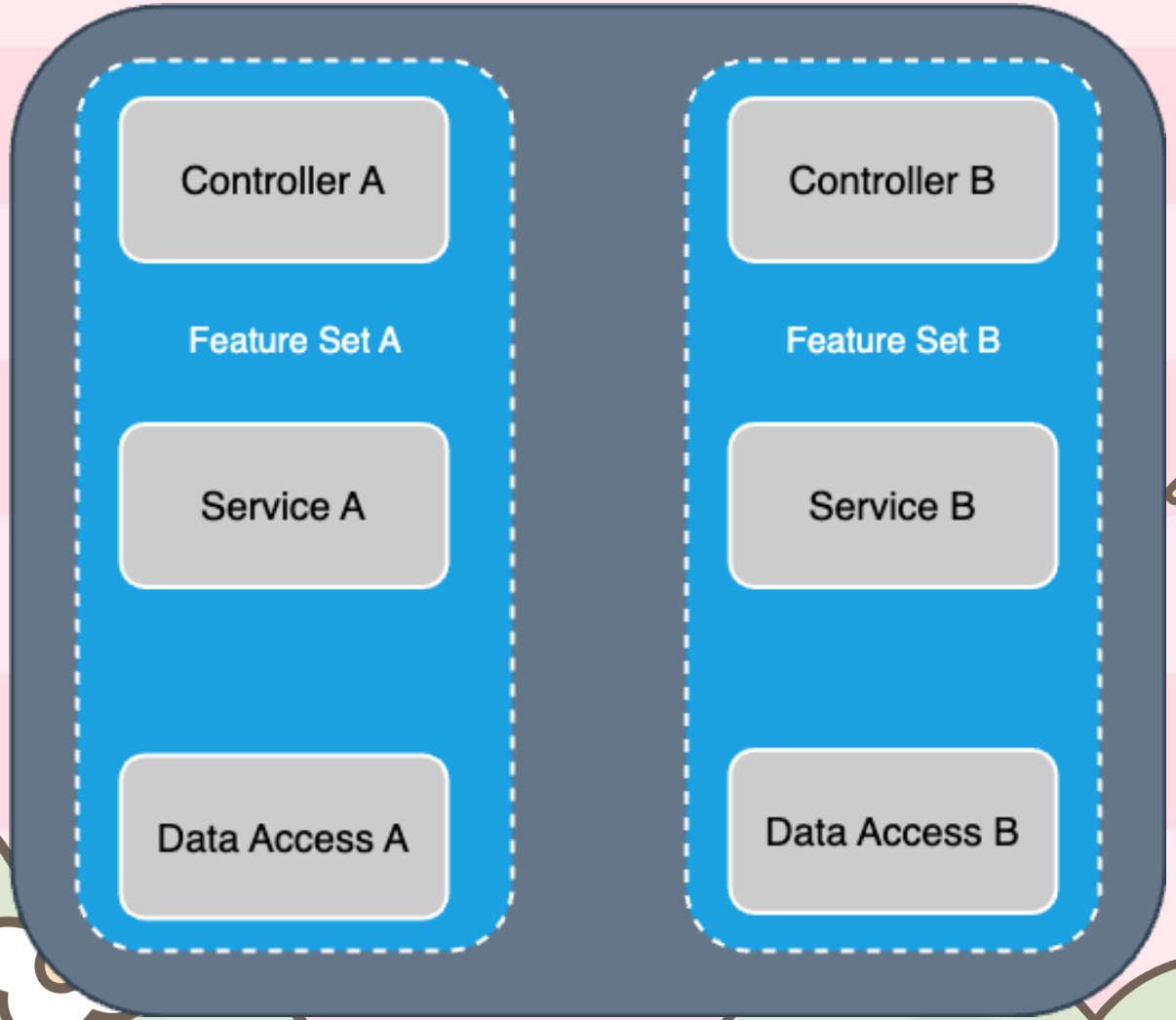
BOUNDED CONTEXT

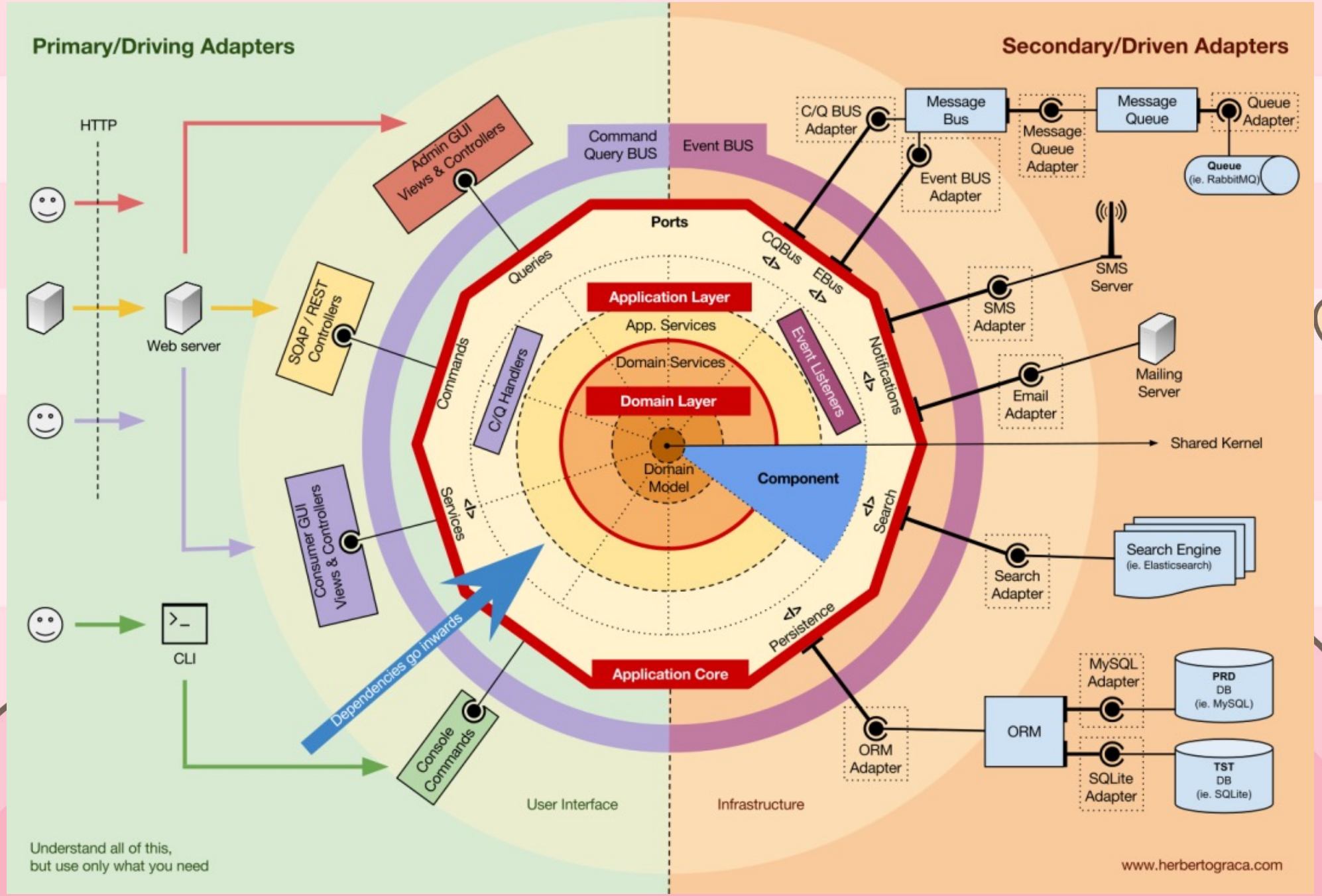


HORIZONTAL SLICING



VERTICAL SLICING




Primary/Driving Adapters
Secondary/Driven Adapters
HTTP
Web server
CLI
Admin GUI Views & Controllers
SOAP / REST Controllers
Consumer GUI Views & Controllers
Console Commands
Command Query BUS
Event BUS
Ports
Queries
Commands
C/Q Handlers
Application Layer
App. Services
Domain Services
Domain Layer
Domain Model
Event Listeners
Services
Persistence
Search
Notifications
CQBus
EBus
Component
Application Core
Dependencies go inwards
C/Q BUS Adapter
Message Bus
Event BUS Adapter
Message Queue Adapter
Message Queue
Queue Adapter
Queue (ie. RabbitMQ)
SMS Adapter
SMS Server
Email Adapter
Mailing Server
Shared Kernel
Search Adapter
Search Engine (ie. Elasticsearch)
ORM Adapter
ORM
MySQL Adapter
PRD DB (ie. MySQL)
SQLite Adapter
TST DB (ie. SQLite)
User Interface
Infrastructure
Understand all of this, but use only what you need
www.herbertograca.com

ADVANTAGES

1. **Communication:** improves communication between developers, managers, and domain experts by providing a shared language and understanding of the business domain
2. **Independence of Frameworks:** The core business logic is kept separate from external frameworks. This independence allows for easier updates or changes to the frameworks without affecting the core application logic.
3. **Testability:** The architecture encourages the use of unit tests, making it simpler to verify the correctness of individual components and ensuring that changes don't introduce unintended side effects.
4. **Maintainability:** Due to the clear separation of concerns, it is easier to locate and update specific parts of the system without affecting the entire codebase. This leads to improved maintainability over time.
5. **Scalability:** Clean Architecture provides a scalable structure, allowing the application to grow without introducing complexities. As the system grows over time, the difficulty in adding new features remains constant and relatively small.

DISADVANTAGES

1. **Initial Learning Curve:** Adopting Clean Architecture may require developers to learn new concepts and adapt to a different way of structuring code, leading to an initial learning curve.
2. **Increased Boilerplate Code:** Clean Architecture can result in more initial boilerplate code compared to simpler architectures, which might be perceived as a drawback, especially for small projects.
3. **Over-Engineering:** In some cases, applying Clean Architecture principles to small or straightforward projects might be considered over-engineering, introducing unnecessary complexity.
4. **Development Time:** Can be complex, especially for large, complex domains. It requires a significant amount of time and effort to create a well-designed domain model and to implement the domain logic.

Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live

Martin Golding

QUESTIONS?