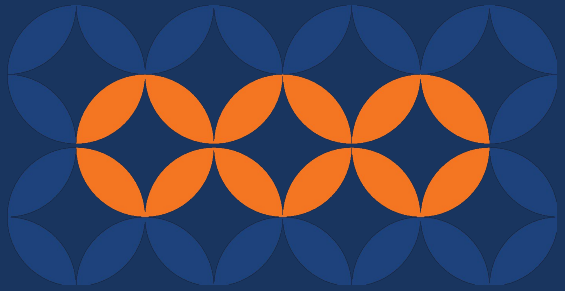
alterra
academy



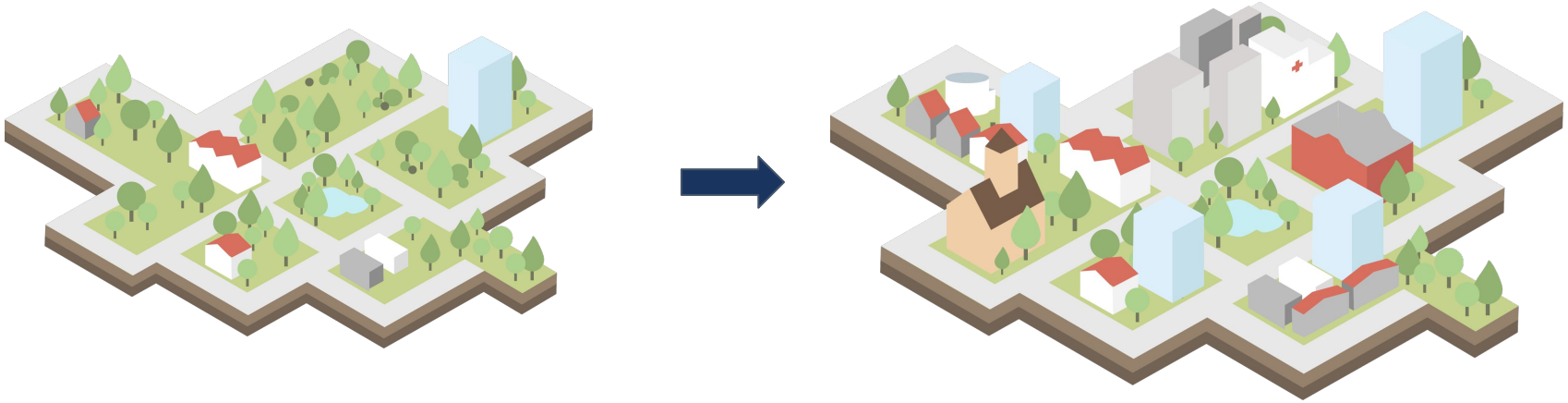
Clean Architecture

“the separation of concerns using layers”

Mission statement:

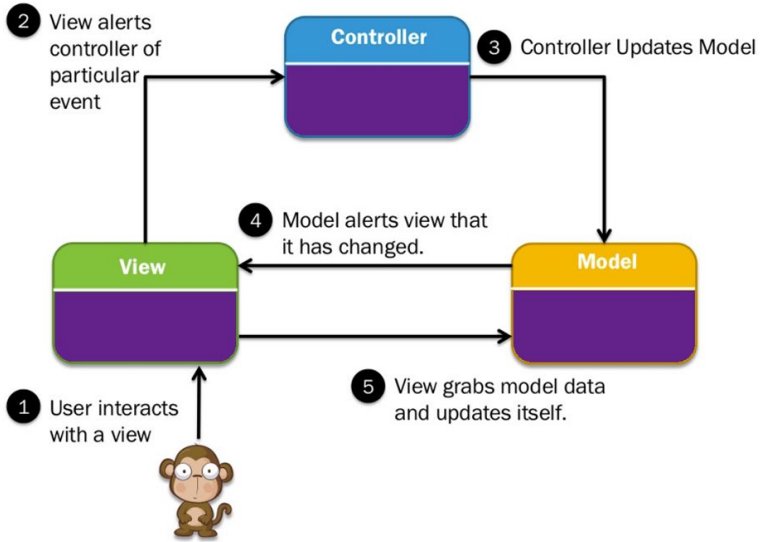
Good architecture is a separation of concern using layer to build a modular, scalable, maintainable and testable application.

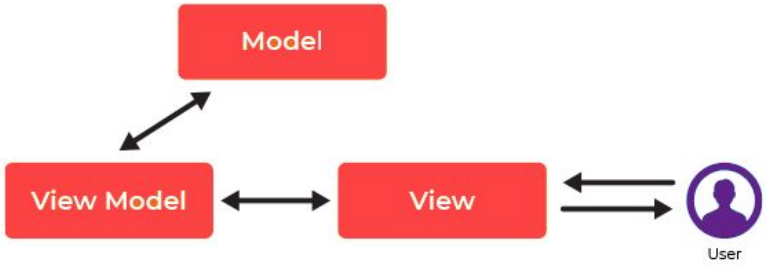
Why should I care?



Working on a programming project is similar to planning and building a residential district. If you know that the district will be expanding in the near future, you need to keep space for future improvements. Without that, you will be forced to destroy buildings and streets to make space for new buildings

MVC and MVVM





https://www.guru99.com/mvc-vs-mvvm.html

alterra academy

CURRENT CONDITION

Every team has his own structure

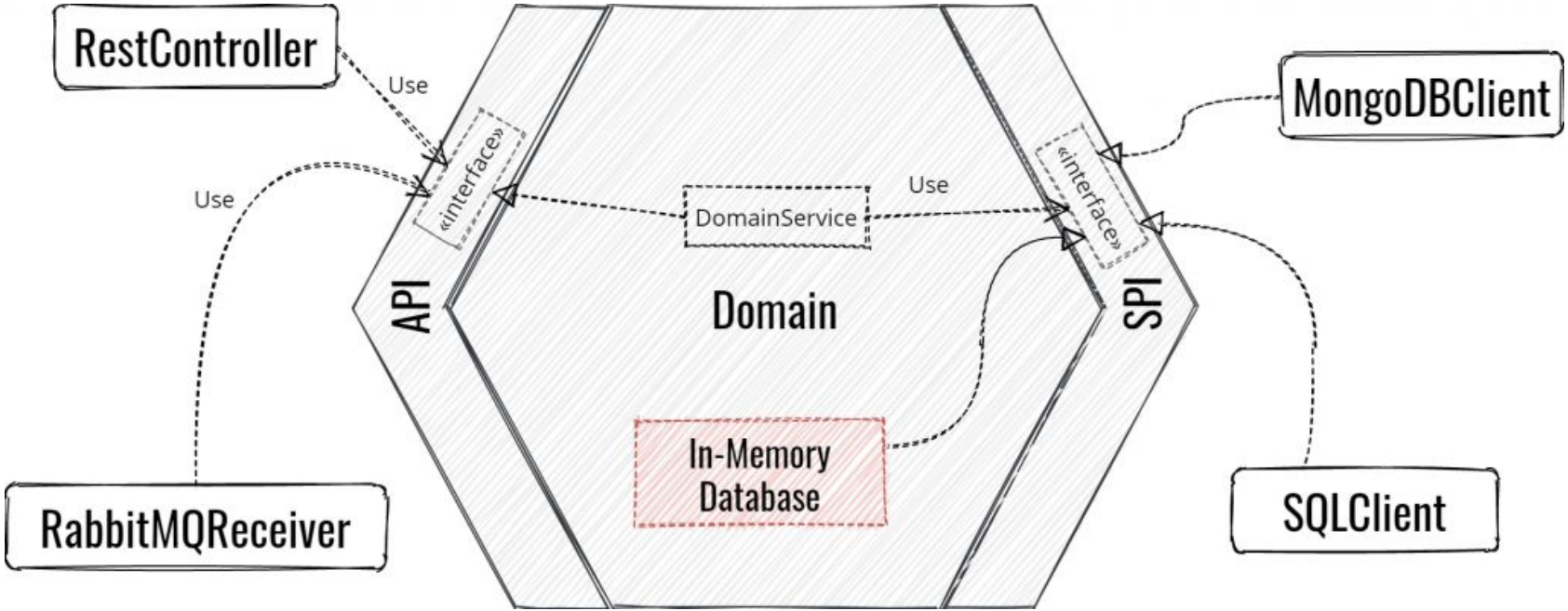
➔ *How to maintain this?*

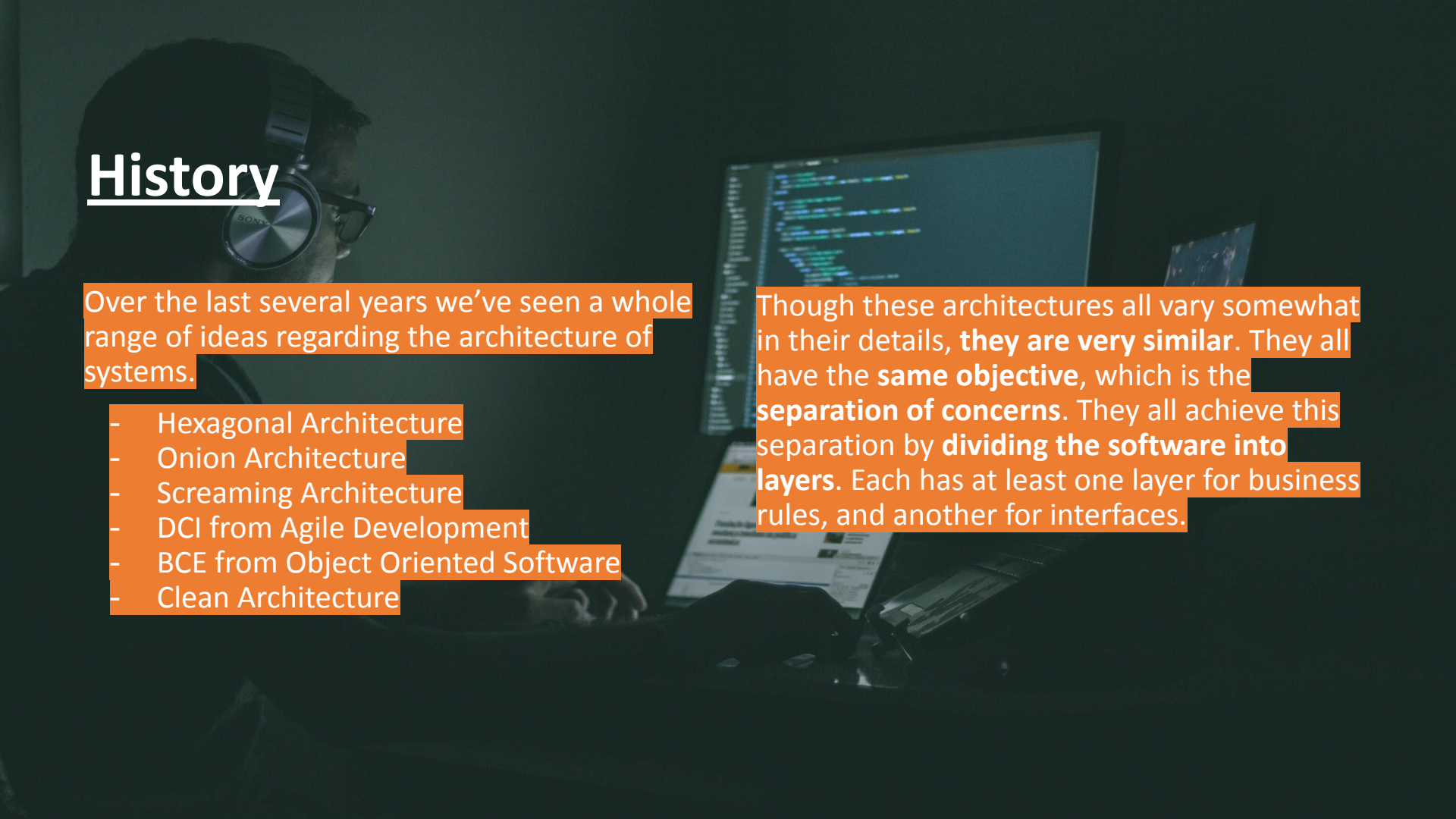
➔ *Mobility issue* → it's hard to transfer knowledge on business domain and plus with unique code structure we have extra time and brain to learn that from top to down.

➔ *Another issue* → ex: it's hard to implement the *unittest* especially test that has connection with database and then we hardly choose the *integrate test* instead.

How can we fix that mess condition?

Hexagonal Architecture one of the best solution.





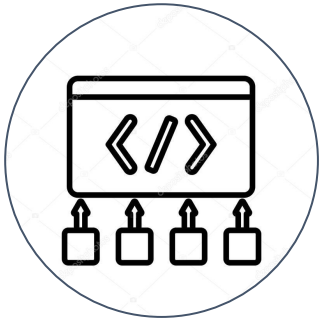
History

Over the last several years we've seen a whole range of ideas regarding the architecture of systems.

- Hexagonal Architecture
- Onion Architecture
- Screaming Architecture
- DCI from Agile Development
- BCE from Object Oriented Software
- Clean Architecture

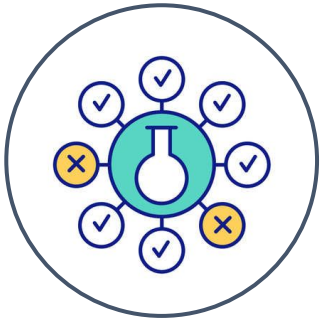
Though these architectures all vary somewhat in their details, **they are very similar**. They all have the **same objective**, which is the **separation of concerns**. They all achieve this separation by **dividing the software into layers**. Each has at least one layer for business rules, and another for interfaces.

The constraint before designing the Clean Architecture are :



Independent of Frameworks

This allows you to use such frameworks as tools, rather than having their limited constraints.



Testable

The business rules can be tested without any other stuff.

Independent of UI

The UI can change easily, without changing the rest of the system.

Independent of Database

Your business rules are not bound to the database, you can swap out the database easily.

Independent of any external

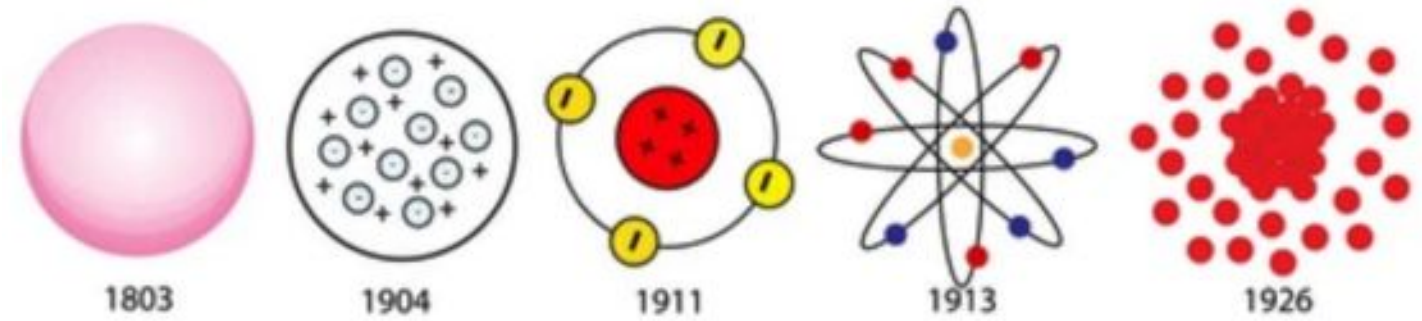
In fact your business rules simply don't know anything at all about the outside world

The benefit

- a standard structure, so it’s easy to find your way in the project,
- faster development in the long term,
- mocking dependencies becomes trivial in unit tests,
- easy switching from prototypes to proper solutions (e.g., changing in-memory storage to an SQL database).

Milestones

Show where you are in the process and what's left to tackle



CA Layer

We will define classic 3-layer architecture (we could have more layers).

- *(optional)* ***Entities layer -*** *business objects as they reflect the concepts that your app manages*
- ***Use Case - Domain layer*** *- contains business logic*
- ***Controller - Presentation layer*** *- presents data to a screen and handle user interactions*
- ***Drivers - Data layer*** *- manages application data eg. retrieve data from the network, manage data cache*

How it works

Approach 1

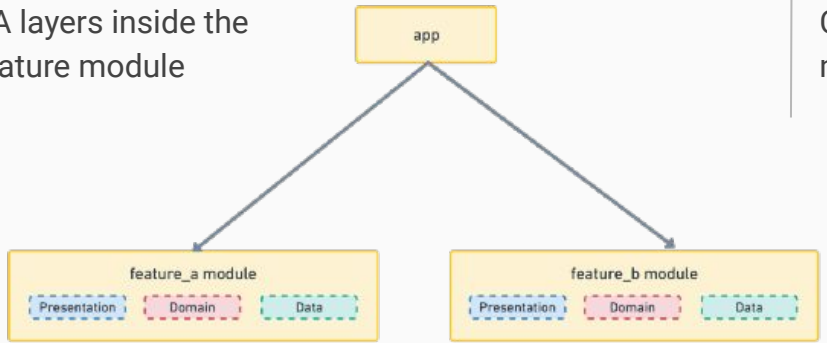
CA layers in a single module

Approach 2

One CA layer per module

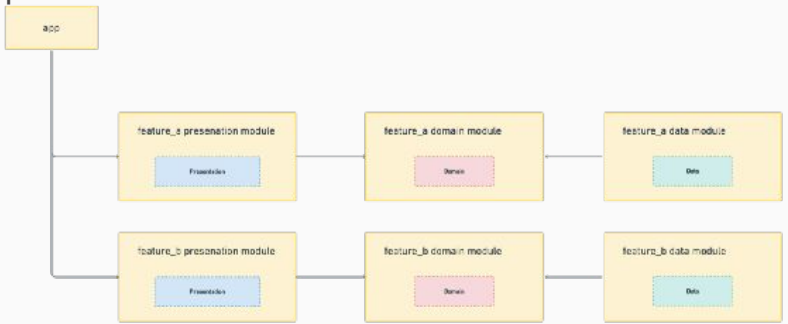
Approach 3

CA layers inside the feature module



Approach 4

CA layers per feature modules



What about Domain:

DDD
(Domain Driven Design)

“I don’t know about the topic and I want to know to be able to do my job properly”

The Two Views

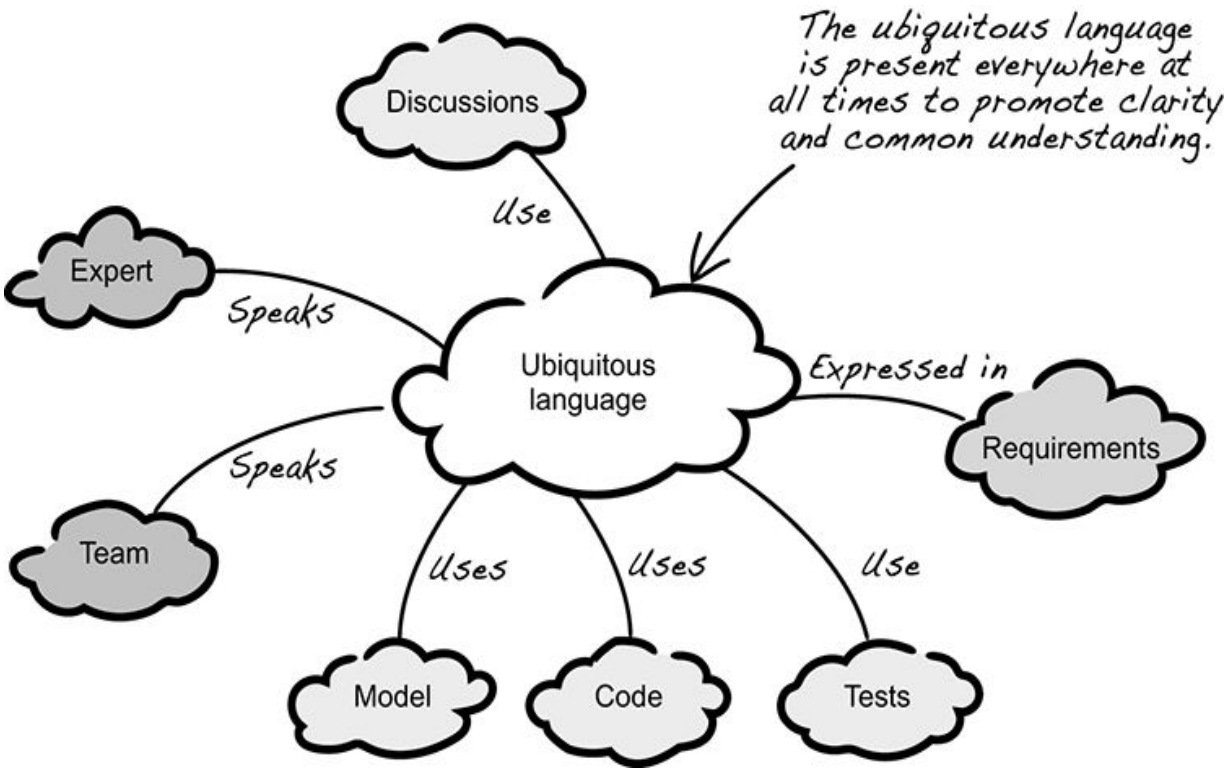
- Software Engineer
- Experts in something we want to solve (product owner, project manager, client)

They speak different languages, how we combine both perspective?

Ubiquitous Languange

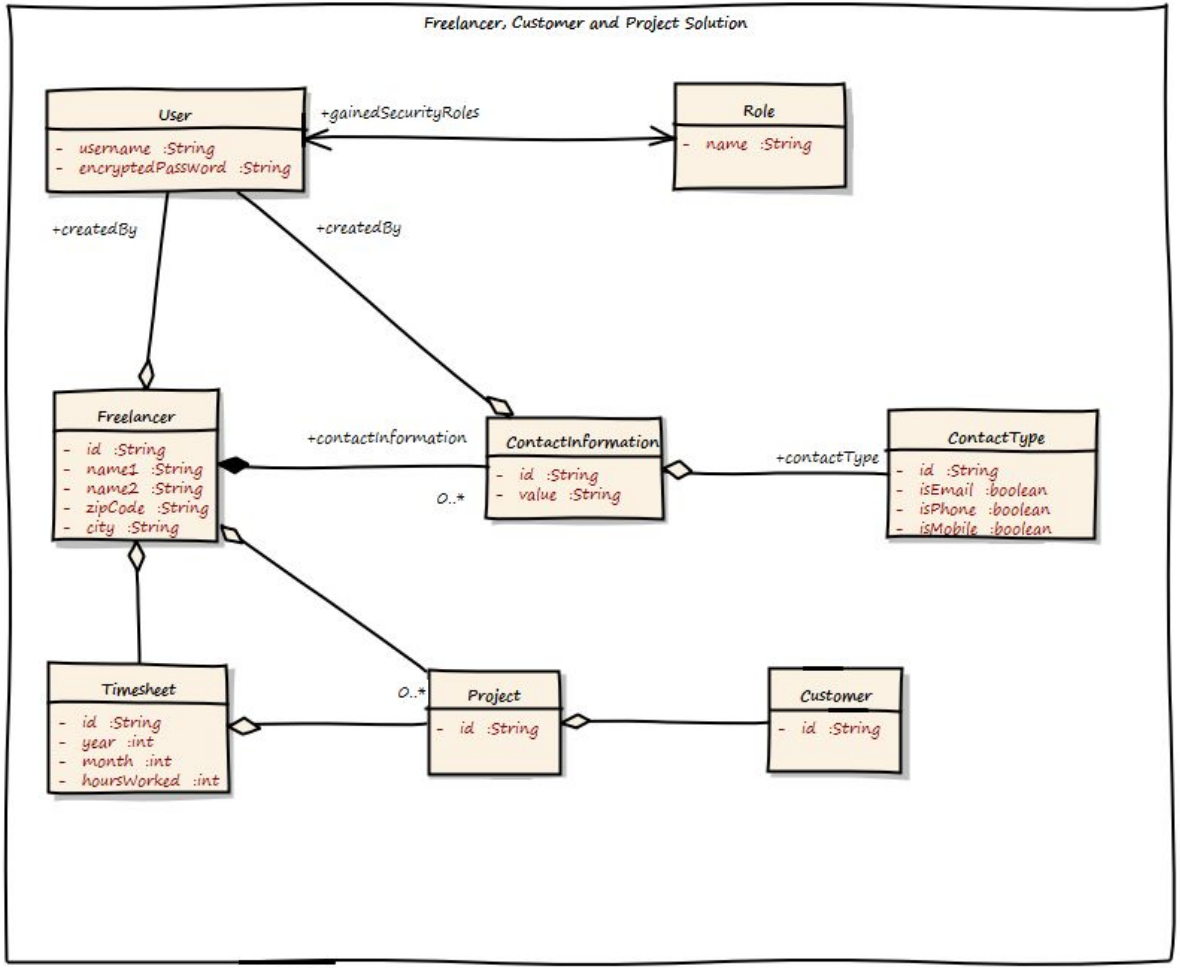
Long story short after asking with expert, exploration, find the work domain, modeling, and brainstorming finally we have something in common within the perspective.

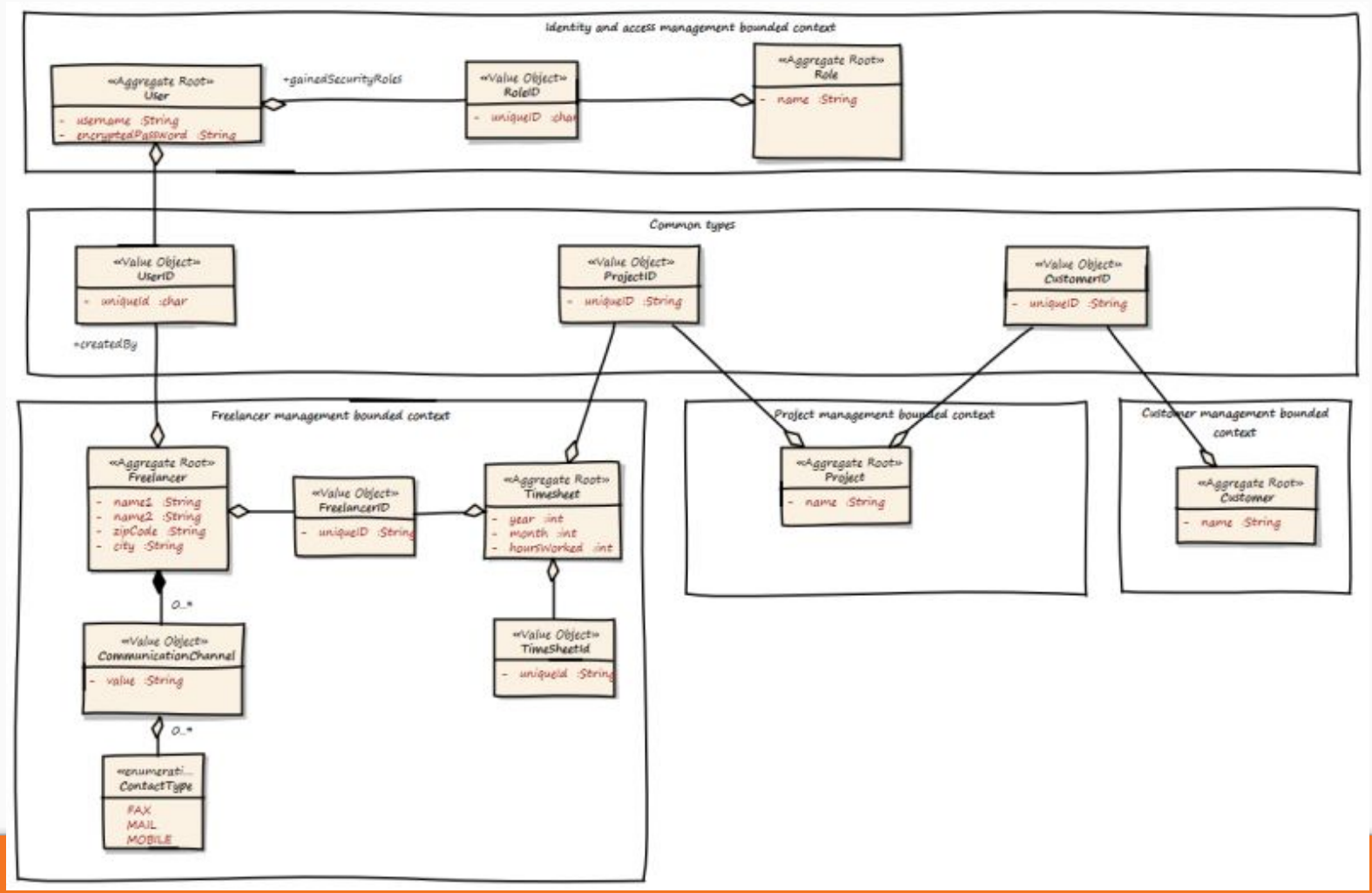
alterra academy



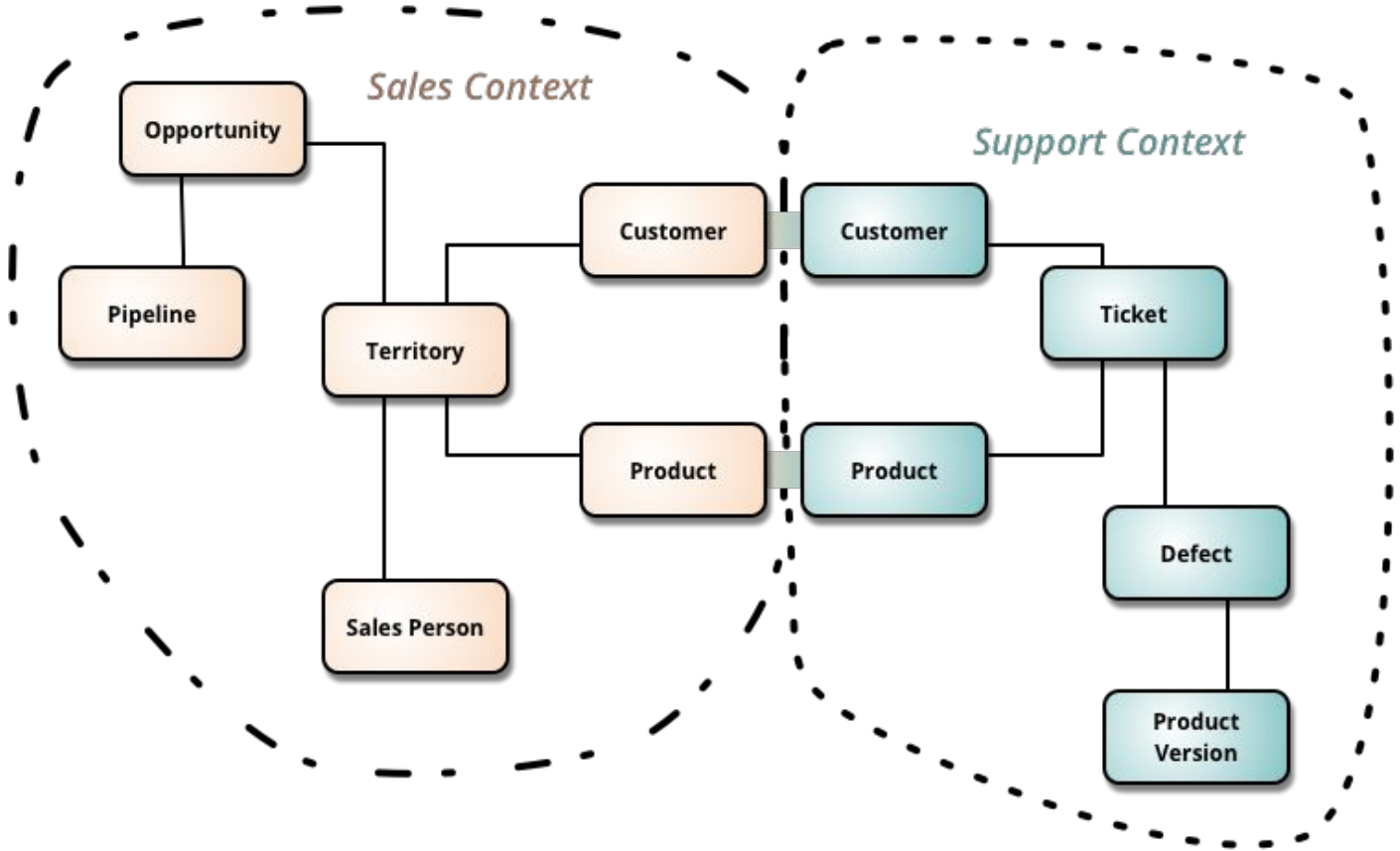
You need to go slow, to go fast

Ensure that you solve valid problem in the optimal way. After that implement the solution in a way that your business will understand without any extra translation from technical language needed.

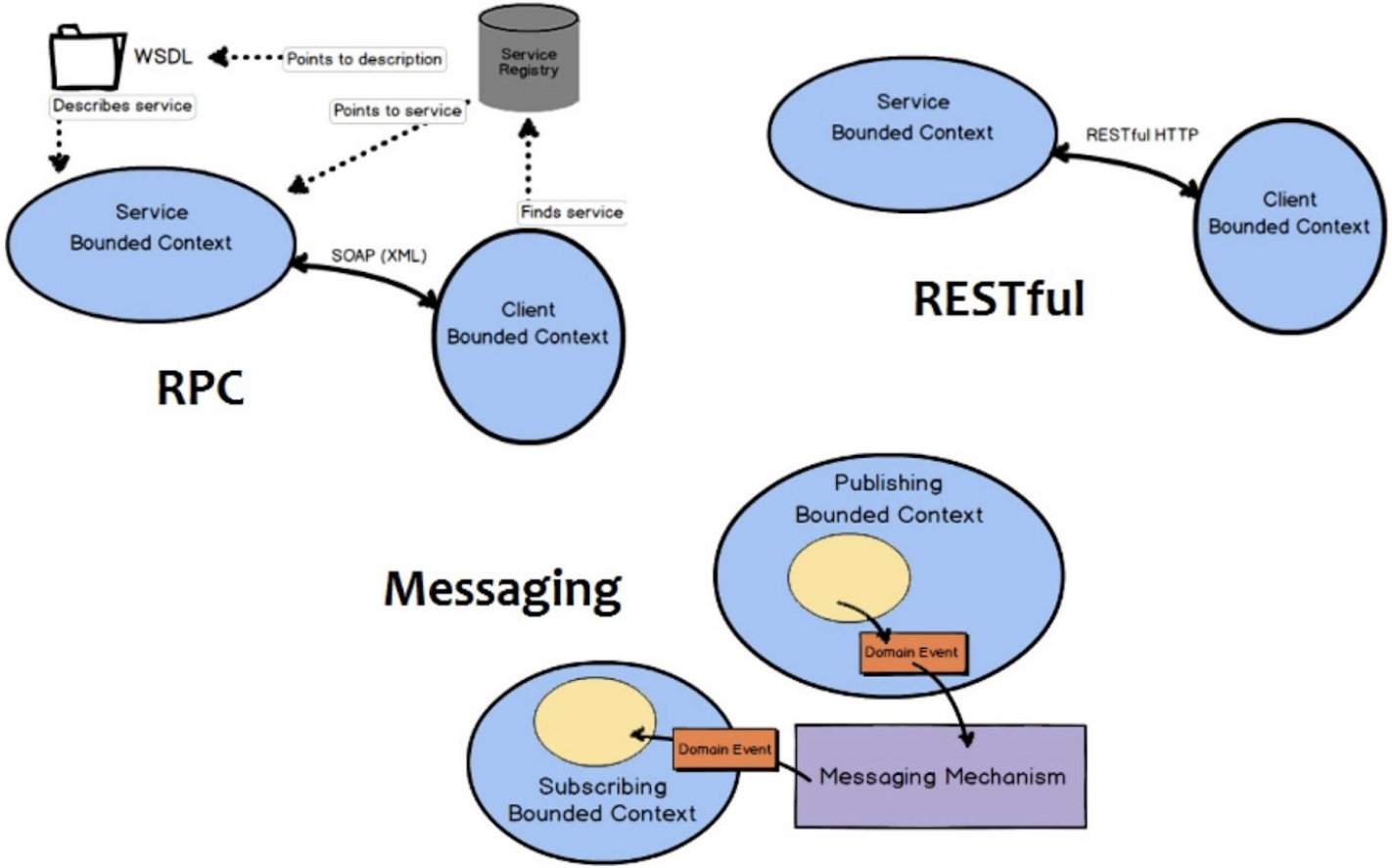



Identity and access management bounded context
«Aggregate Root»
User
- username :String
- encryptedPassword :String
+gainedSecurityRoles
«Value Object»
RoleID
- uniqueID :char
«Aggregate Root»
Role
- name :String
Common types
«Value Object»
UserID
- uniqueId :char
«Value Object»
ProjectID
- uniqueID :String
«Value Object»
CustomerID
- uniqueID :String
+createdBy
Freelancer management bounded context
«Aggregate Root»
Freelancer
- name1 :String
- name2 :String
- zipCode :String
- city :String
«Value Object»
FreelancerID
- uniqueID :String
«Aggregate Root»
Timesheet
- year :int
- month :int
- hoursWorked :int
0..*
«Value Object»
CommunicationChannel
- value :String
«Value Object»
TimeSheetId
- uniqueId :String
0..*
«enumerati...
ContactType
FAX
MAIL
MOBILE
Project management bounded context
«Aggregate Root»
Project
- name :String
Customer management bounded context
«Aggregate Root»
Customer
- name :String

Bounded Context



Communicate between domain



Use Case

alterra academy

This usually what we see…

- Add new product
- Edit product
- Delete product

This what we need to see….

- Owner need to add new product
- Owner need to archive old product
- Owner need to activate product
- Owner need to edit product
- Owner need to delete product which archive more than 1 year

Then we make a method represent the business logic, literally

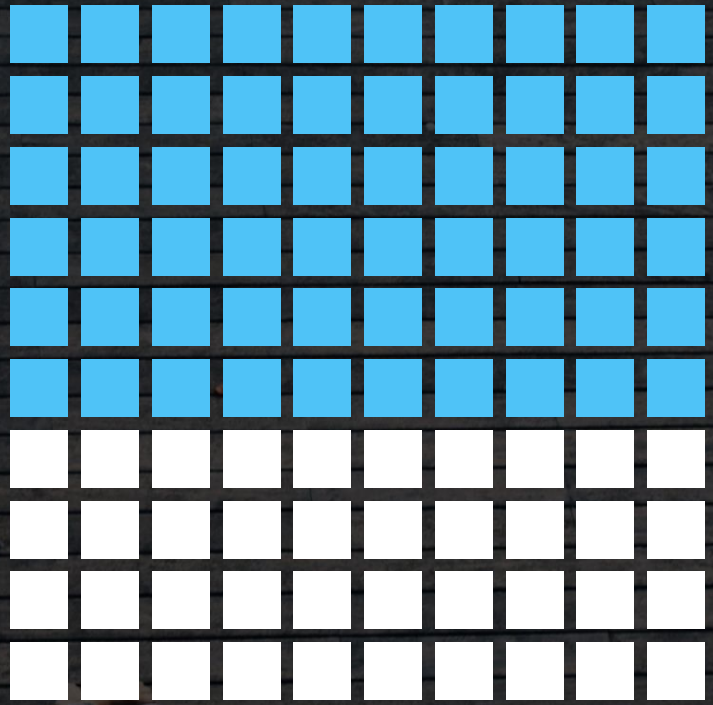
- Product.Create()
- Product.Archive()
- Product.Activate()
- Product.Update()
- Product.DeleteArchive(duration int)

Clean Architecture

is a software architecture

Domain Driven Design

is software design technique



They complete each other.