Implementing Domain-Driven Design in Python(Django)

PUG Tehran 2024

Who am I?

Milad Koohi(@miladkoohi)

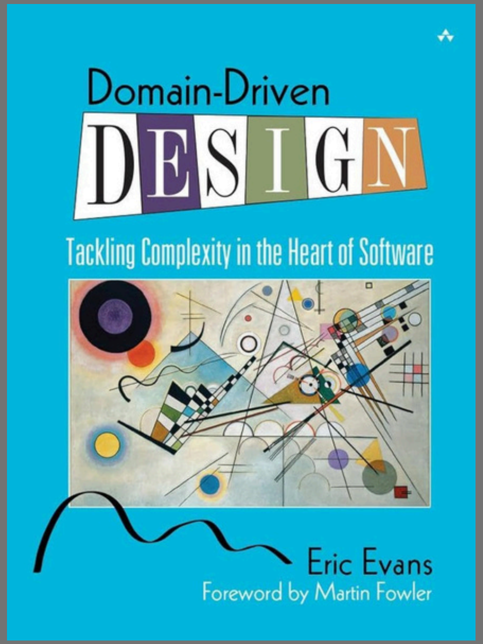
full-stack, previously .net + net core, 4-year XP in web dev

been doing Python/Django + Software Engineer since 2020

local meetup co-organizer, tutor

What is DDD, why and when?

Domain-Driven ~~Development~~ Design


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

A major software design approach, focusing on modelling software to match a domain according to input from that domain's experts. (Source: Wikipedia)

As a developer, you're not the expert.

It's an approach for conception, not a software recipe, nor a framework. It's language-agnostic. Theorized by Eric Evans.

Business domain at the center

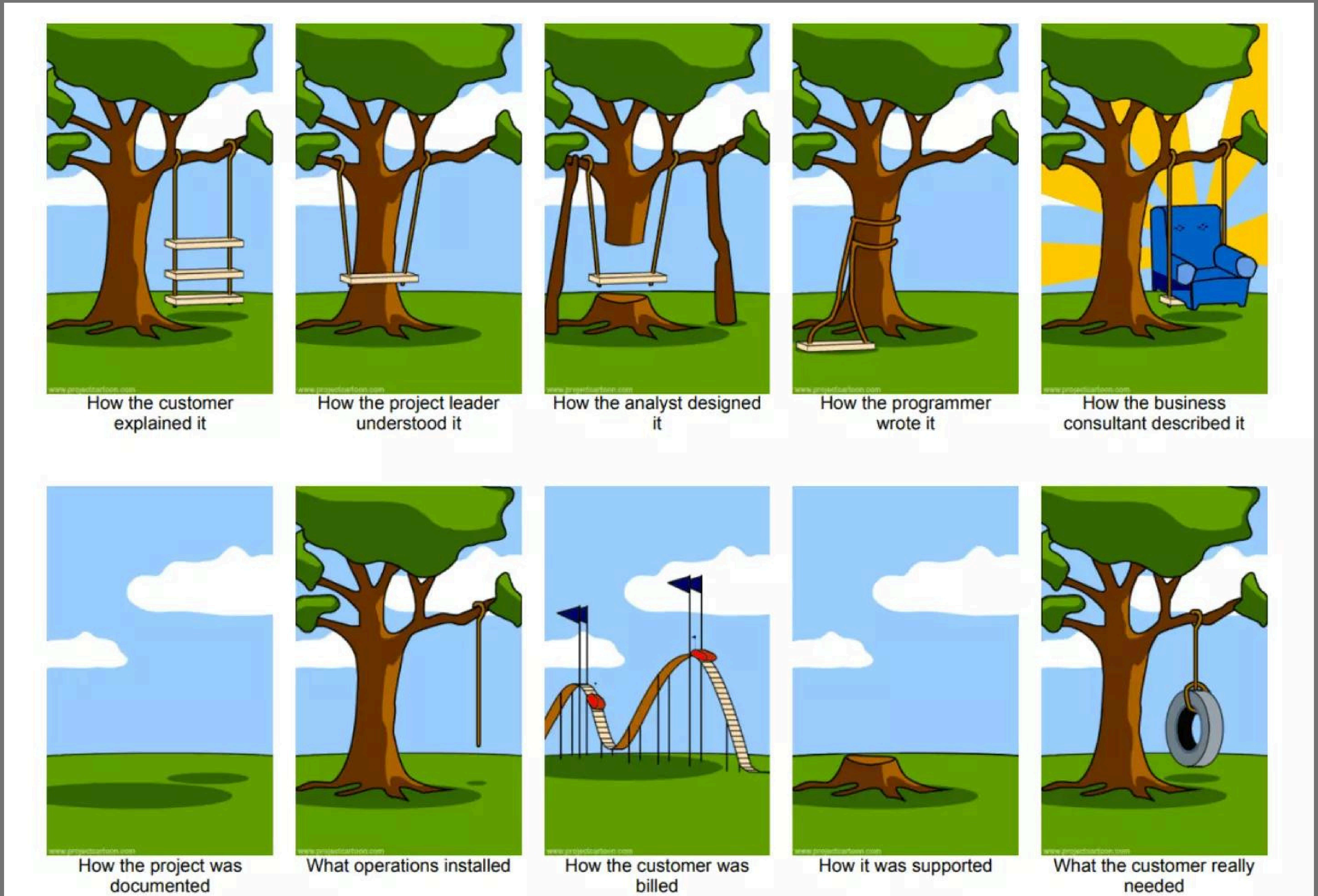
- As you're not the expert, you must speak the same language as domain's experts, have the same terms.
- If needed (because sometimes they're speaking a weird language), a glossary can be created
- Your code must reflect these terms
- If your domain is expressed in French, then yes, your DDD code should be in French

Domain? Why? What about data?

- Yes, but for complex processes, the code can look like spaghetti
- When your model changes, you risk an alteration of the process
- Sometimes you code things you don't understand (and you're translating it wrong)
- A model-driven architecture represents a developer's way of thinking, and teams can change over time

Django or rest ... is a model-driven
(or data-driven ?) framework !

Domain? Why? What about data?



How the customer explained it

How the project leader understood it

How the analyst designed it

How the programmer wrote it

How the business consultant described it

How the project was documented

What operations installed

How the customer was billed

How it was supported

What the customer really needed

When to use DDD?

Project must be maintained in the long-term (5 years +)

Lots of business domains

You have time money and software crafting experience

When not to use DDD?

App is a simple CRUD

Business is already answered by third-party (CMS, e-commerce, etc.)

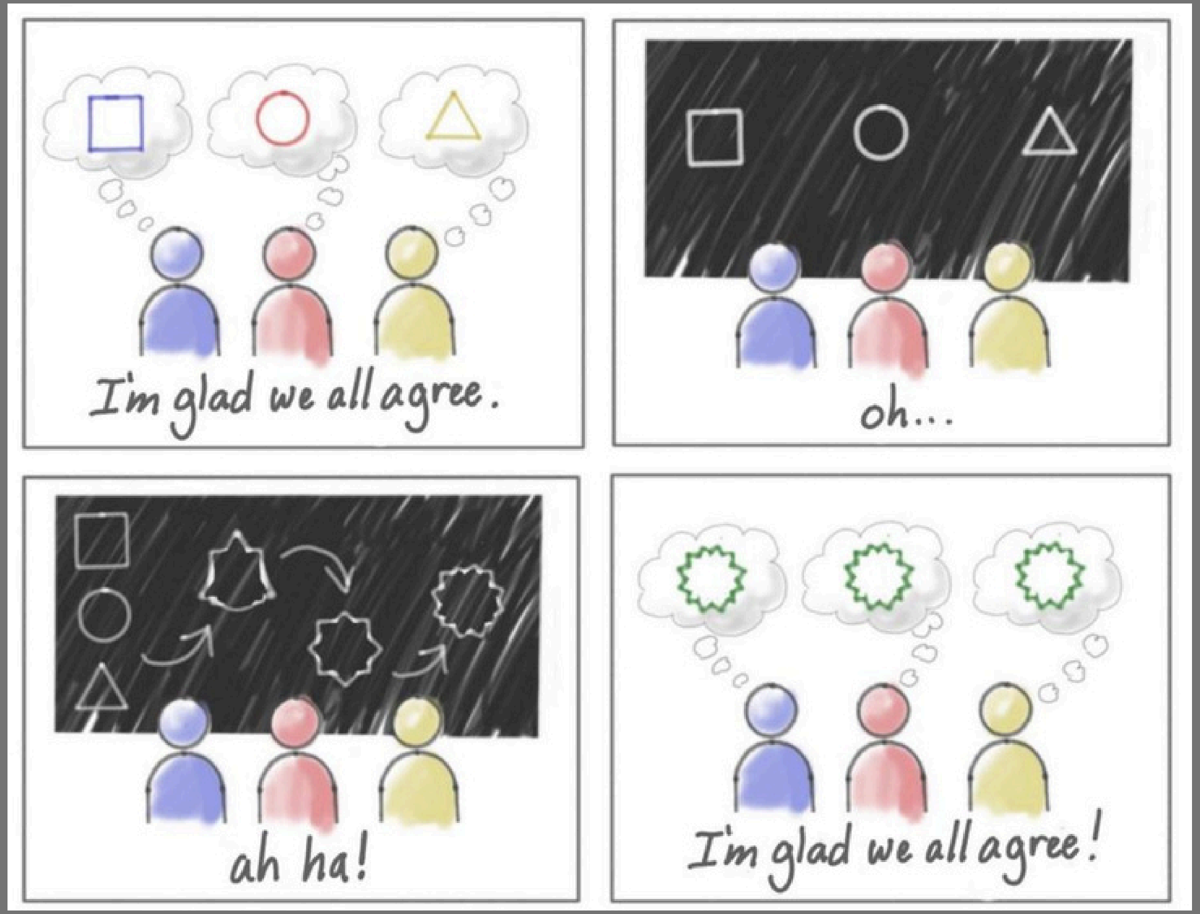
No maintenance needed over time, no evolution

App is "too simple" or no complex business

Your dev team is too junior

Common patterns

Ubiquitous language



- Use the right terms across your domain
- Define them at the beginning, they must come from the business but they can be discussed
- A term can have another meaning in another context

Bounded contexts

Separate your business into small parts,
each one solving a problem.

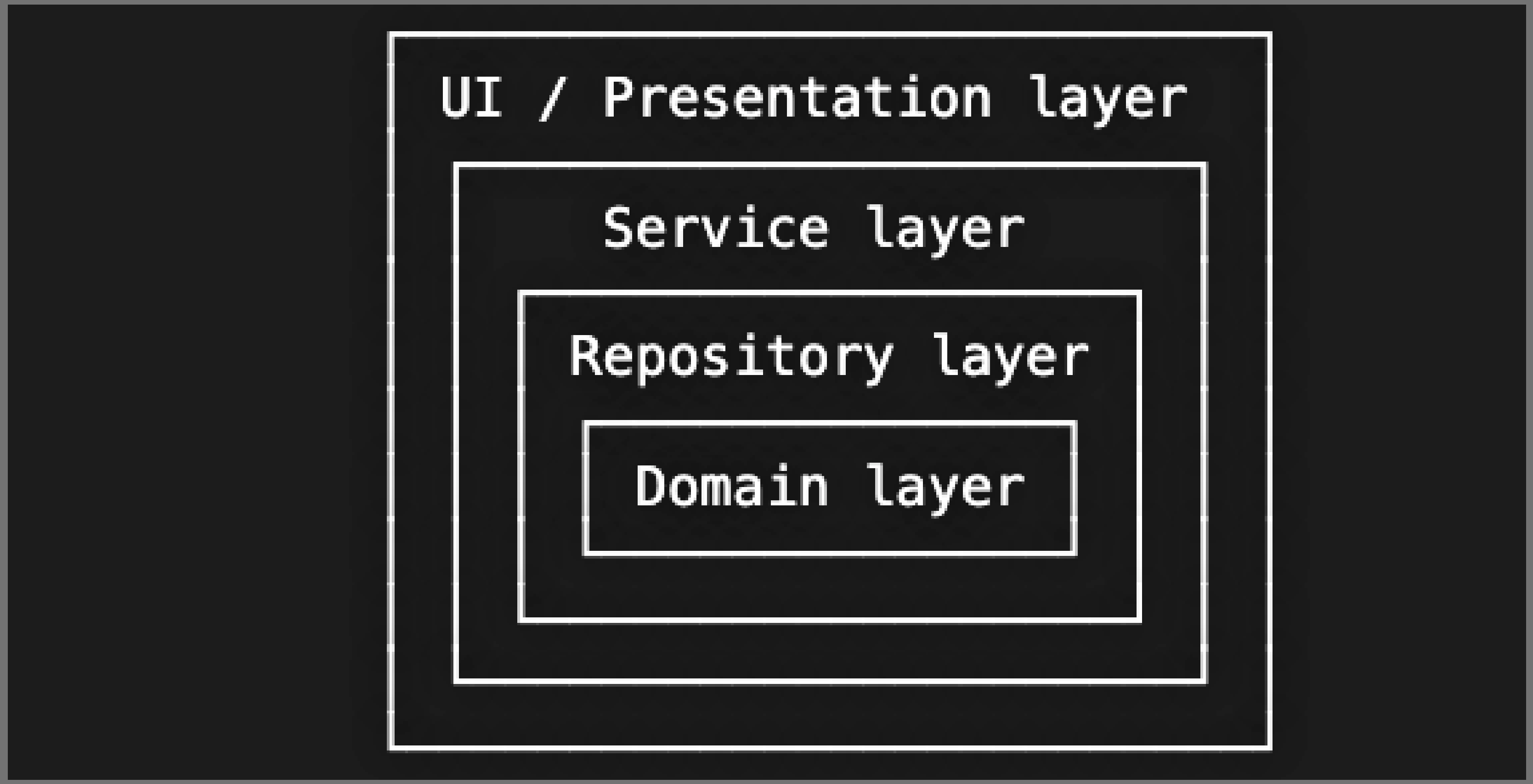
1. means that an object can have different name in different contexts
   - User is a customer in the commerce domain
   - User is an account in the invoicing domain
   - User is a recipient in the shipping domain
2. in the code, bounds usually means namespace
3. does not mean contexts cannot communicate, but data may have to be translated between
4. data exposure can depend on context (even if close in storage)
   - payment mode is only needed in invoice domain
   - address could be different in invoice and shipping domains

Shared kernel

- A shared kernel can be created for objects that are used everywhere
- It can be present at multiple levels of the project
- It prevents duplication
- Beware of it inflating if you put too much in it!

Common examples : user info, addresses, notifications

Onion architecture



some DDD architectures use hexagonal or clean architecture

Vocabulary

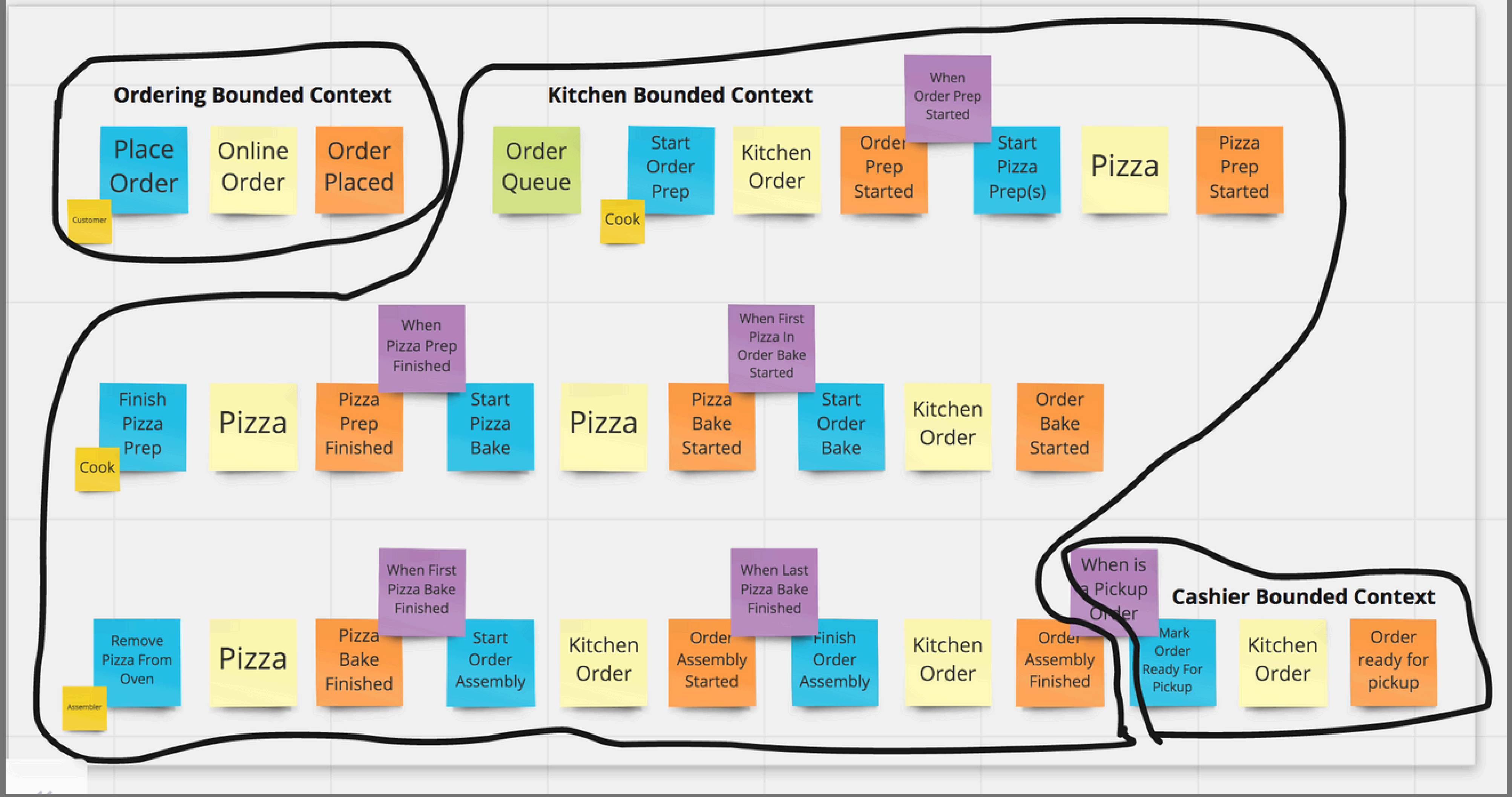
- Use case : some action, responding to a query (read) or command (write) made by someone
- Aggregate : a root domain model entity, entry point to manipulate data, masking other domain entities
- Repository : your way to the data, loading or saving aggregates, or fetching DTO
- DTO : Data-Transfer-Object, a dumb data class used to I/O from context
- Domain service : useful when dealing with multiple or complex things
- Validator : prevents manipulating a wrong state in your domain

Use cases

Also called "application services", basically a user story

Special technique : Event storming

Get everyone in the business around a table to state on what should the application do (take this time to also establish the glossary)



Our take

A brief comment on Python

- Use at least python 3.7 for dataclasses
- attrs is a nice library for default attribute factories, frozen and slotted classes
- Using typing makes it easier, mypy (with django-stubs) is your friend after that
- Avoid using any of Django in ddd directory (apart maybe gettext_lazy and timezone)

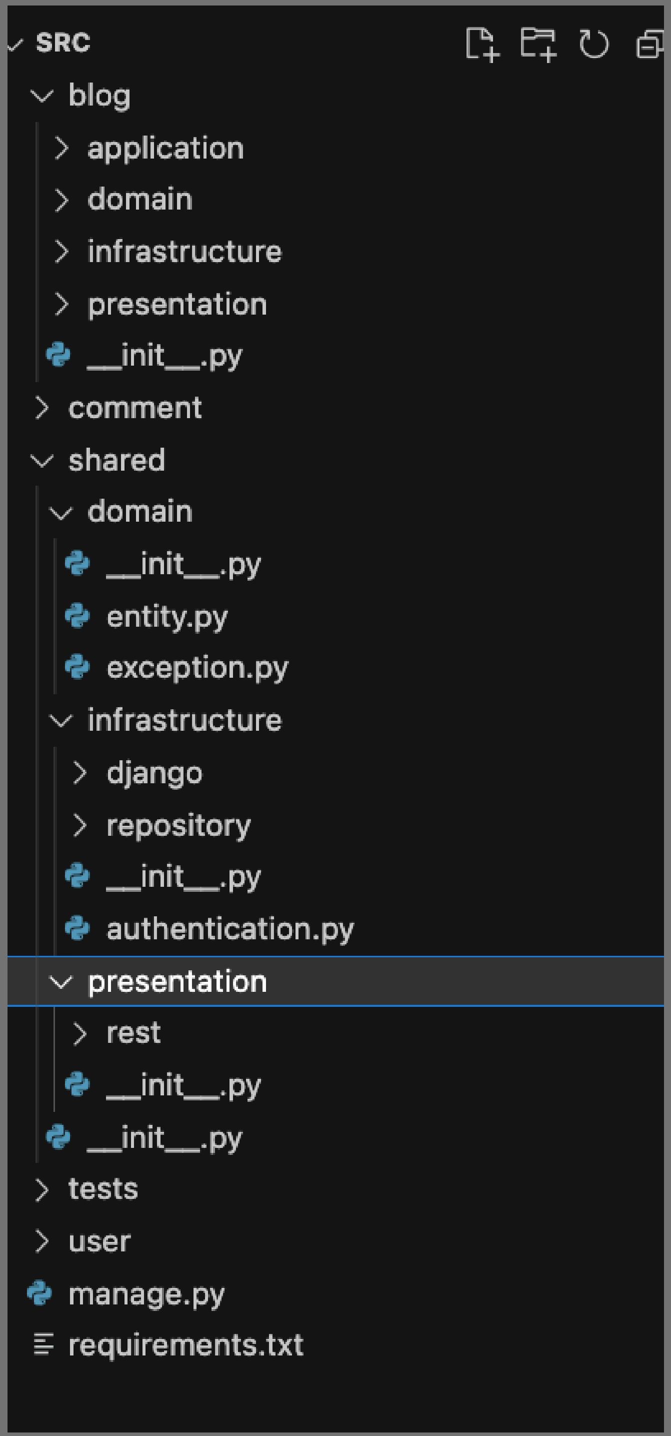
An example of an aggregate

- it's a **root entity** (could be a single model in Django, or multiple, or a part)
- it has an **identity** (usually PK, may be an uuid)
- it has a **state** (data attributes) and some behaviors (methods encapsulating the business)
- that may contain complex data (through ValueObjects)
- some behavior can ensure its state is valid before modifying it
- it can be loaded and saved **through a repository**

An example of repository

- Based on a declared interface, hence swappable:
- can have an in-memory version for unit tests
- or even a SQLAlchemy version to drop Django's ORM 😱
- It should have 3 default methods : get, save (update or create), and delete, but can have other methods (search, get_by_X)
- Can return an aggregate (or a collection of) or DTOs

Software Architect



```
SRC
  blog
    application
    domain
    infrastructure
    presentation
    __init__.py
  comment
  shared
    domain
      __init__.py
      entity.py
      exception.py
    infrastructure
      django
      repository
      __init__.py
      authentication.py
    presentation
      rest
      __init__.py
    __init__.py
  tests
  user
  manage.py
  requirements.txt
```

```
src
├──── shared
│    ├──── domain
│    └──── infrastructure
│     ├──── django
│     └──── repository
│     ├──── mapper
│     └──── rdb
├──── blog
│    ├──── application
│    │    └──── use_case
│    │    ├──── command
│    │    └──── query
│    ├──── domain
│    │    ├──── entity
│    │    ├──── exception
│    │    └──── value_object
│    ├──── infrastructure
│    │    ├──── database
│    │    │    ├──── migrations
│    │    │    ├──── models
│    │    │    └──── repository
│    │    └──── django
│    └──── presentation
│    └──── rest
│    ├──── api
│    ├──── containers
│    ├──── request
│    └──── response
├──── user
└──── tests
```

Pros

- Technical "complexity" is concentrated in the ddd directory, with classic python
- DRY
- a use case could be re-used in multiple places in the UI
- business validators too
- optimize queries in the repository, model proxy and managers still have value
- Premature optimization is avoided
- Your code is well-structured, easily readable (even by a non-technical person)
- Typing and type checking (F*** yeah!)
- Easier maintenance
- Decoupling business from the framework, you know, if one day you stop loving Django...

Cons

- You lose the sugar (Rapid Application Development) of Django
- Auto-scaffolding is out (or you must rework it)
- Some queries (especially across multiple aggregate/domains) become complex
- ORM is a bit trickier to optimize
- No support for transactions, could be added, but tricky
- It takes more time, so more money

@miladkoohi

I have code experiences and my studies in the field of:
Python | Rust | Django | Flask | Clean codes | Clean software architecture Free software | DDD | TDD | DevOps | Data streaming

You can follow me and republish my content, I am ready to receive your comments