galvanize

Module 2

Week 7 | Day 2 | **Aggregates & Factory Pattern**

Goals for the Week

Monday

* Modeling a hotel management system
* **Domain-Driven Design**: Entities & Value Objects
* JSON
* **Lab:** Intro to Conference GO

Tuesday

* Building a blog & shopping cart
* **Domain-Driven Design**: Aggregates & Factory Pattern
* Build a JSON Library
* **Lab:** Build your own JSON library

Wednesday

* RESTful API's
* **Domain-Driven Design**: Bounded Contexts
* **Lab:** RESTfulize your app!

Thursday

* HTTP
* More REST
* **Domain-Driven Design**: Anti-Corruption Layers
* **Lab:** Integrate 3rd Party data

GALVANIZE | HACK REACTOR BY GALVANIZE

Today's Agenda

Aggregates & Factories: Design Patterns

Domain-Driven Design

1. **Activity:** Finding aggregates in Hotel Model
2. **Theory:** Aggregates & Factory Pattern
3. **Code-Along:** Build a Blog

AFTERNOON
Technical Practice:

Building Your Own JSON Library

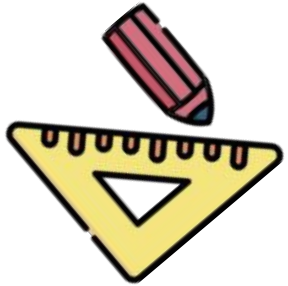
slido

Join at slido.com
#056358

GALVANIZE

slido

Why do engineers use Domain-Driven Design?



GALVANIZE

+ BENEFITS

1. **Flexibility** - built for evolution of business processes & tech stacks
2. **Communication Between Business People and the Development Team**
   Philosophy: *"The domain is more important than the tech stack or UI"*
3. **Team Driven** - modular organization of code. *Hexagonal architecture*.
4. **Business Logic Lives in One Place**

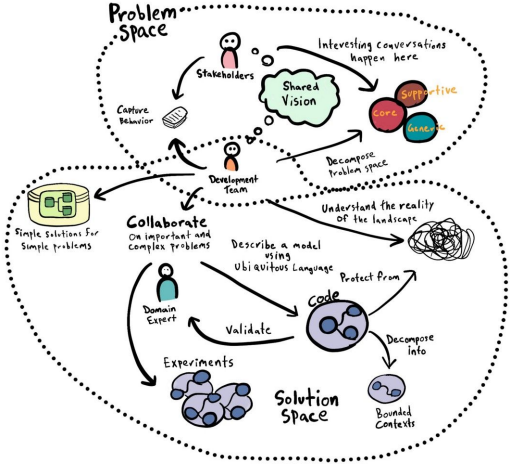
– DRAWBACKS

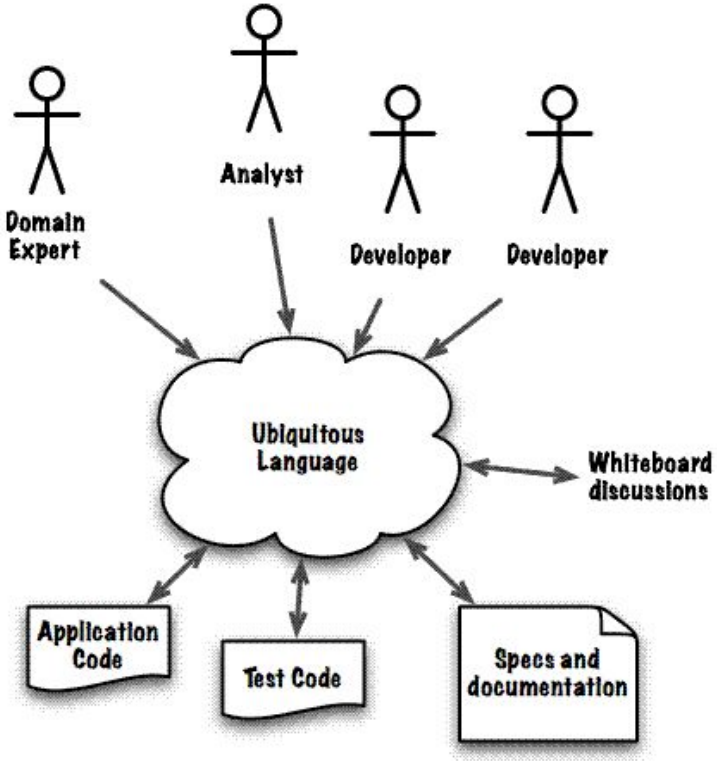
1. **Time & Effort** - most of the time on the project spent in conversation with domain experts to understand and model business logic.
2. **Large Learning Curve -** DDD includes many difficult to understand principles, patterns, and processes.
3. **DDD should not be used on every project** - best for complex domains; not a good choice for small projects.

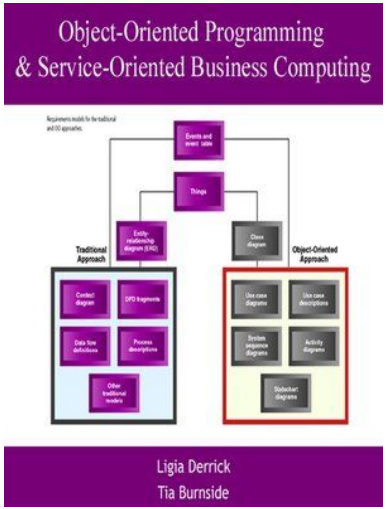
1. Benefits & Drawbacks of DDD

Tackling complex and evolving systems by modeling modular business logic, in consultation with Domain Experts.

Object-Oriented ALL THE THINGS!







HACK REACTOR
BY GALVANIZE

Example: Hotel Reservation Software

Departments / People:

- Finance department
- Cleaning staff
- Concierge
- Management

Example: Hotel Reservation Software

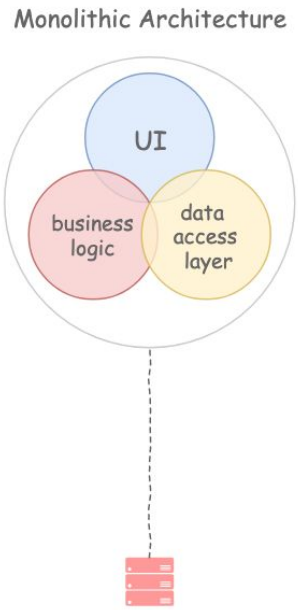
Business Problems:

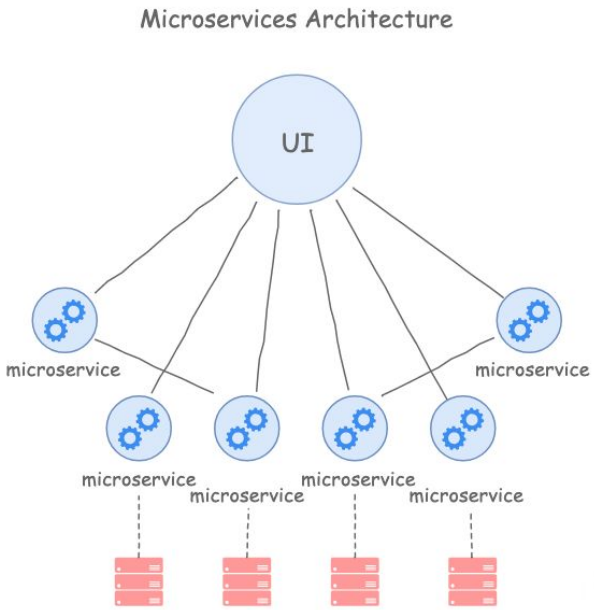
1. Billing – Add up all the charges for services with taxes.

2. Reservations - Assign rooms, schedule cleaning services.

3. Payroll - Pay employees for the services and hours they work.

Make them into Microservices!





slido

What are some of the goals of Domain Driven Design?

Select one or many answers below:

GALVANIZE

Keywords

1

Domain – the field or industry in which a company operates.

2

Domain Expert – the people who know most about their domain or subdomain.

3

Subdomain - in DDD, a domain is broken up into modular subdomains, each with its own domain expert.

4

Strategic Approach

5

Tactical Approach

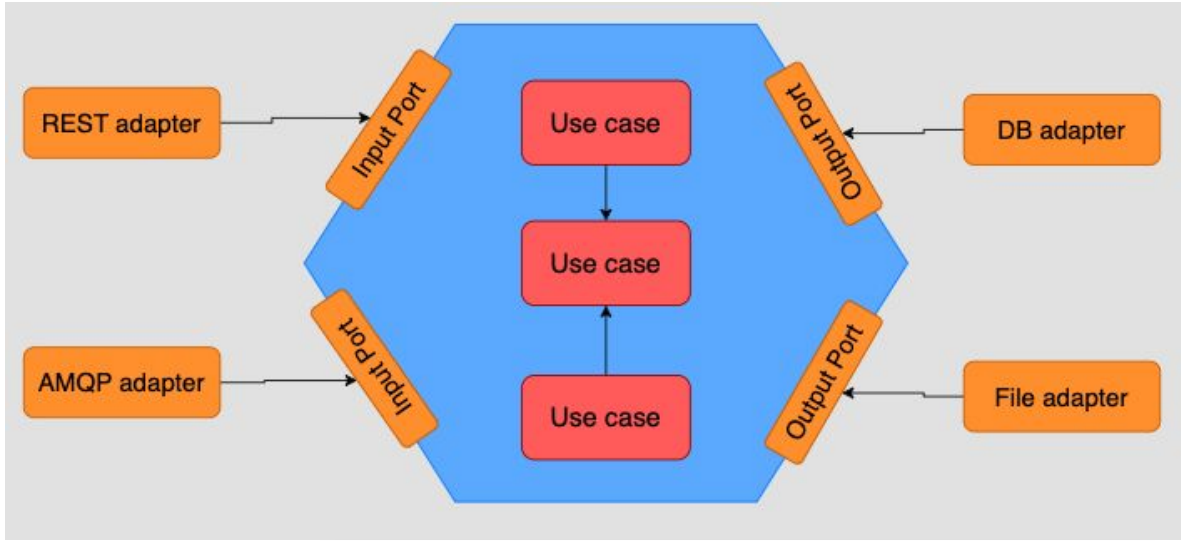
6

Hexagonal Architecture

Hexagonal Architecture

A.K.A. **Ports and Adapters Architecture**

- Separates the business logic from the outside world
- **Ports** - interfaces for apps or databases to pass or receive data from outside entities
- **Adapter** - used to connect two layers using ports, translating data in a safe way
- Inner & Outer Layers: Each layer does not know how the other layers work



By **Alistair Cockburn**, a founder of the Agile movement, which deeply influences DevOps.

+ BENEFITS

- **Independence** - each layer is independent of all others, and can be worked on by separate teams.
- **Quick Maintainability & Unit Testing** - changes in one layer do not impact another. Allows for continuous integration and deployment across teams.

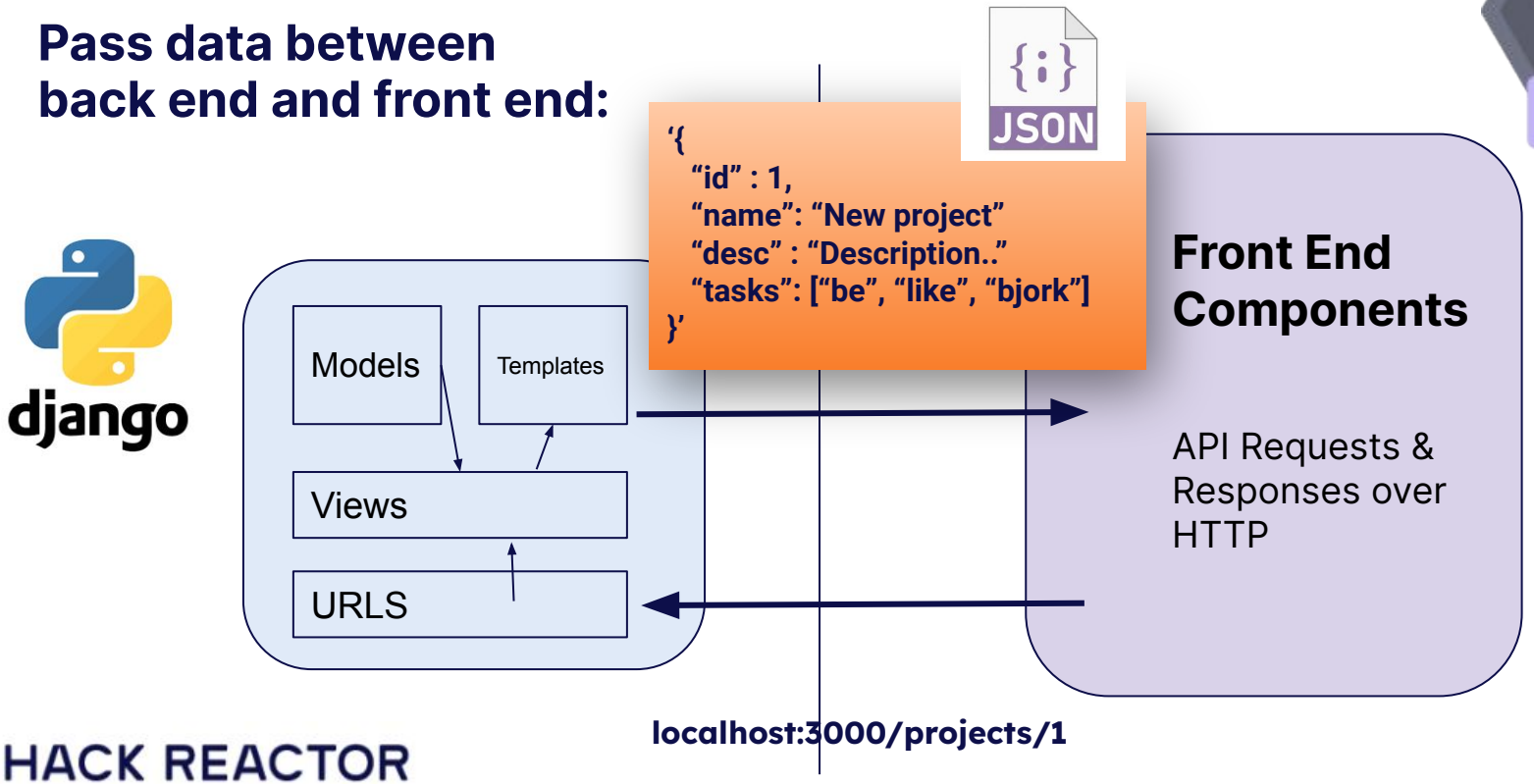
slido

What is the format in which data can be passed over HTTP from the Django backend and server to the user's browser?

Construct an example of one of these!

JSON

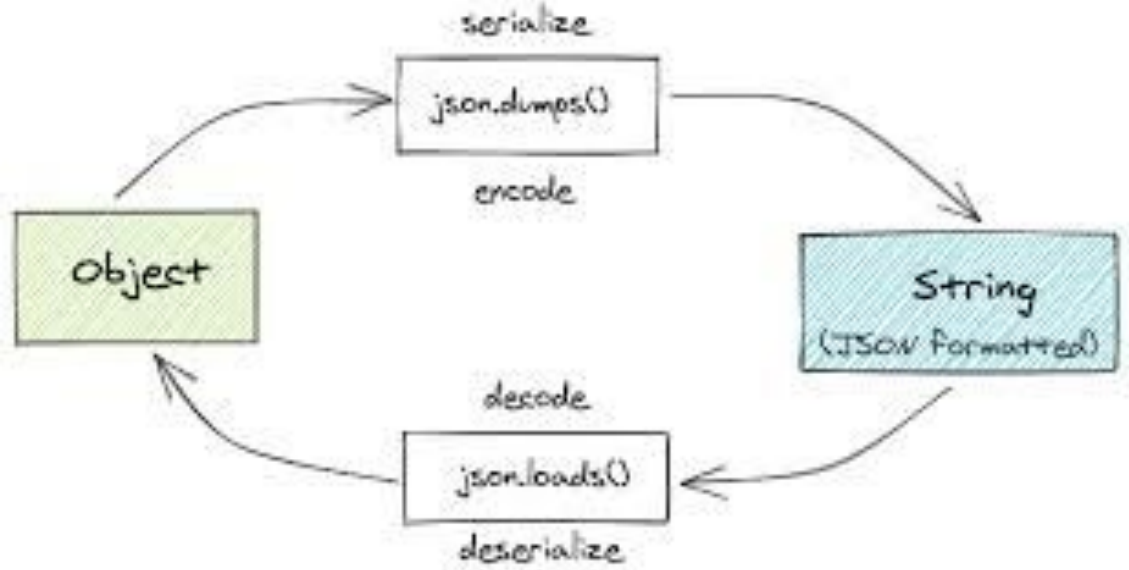
- **Pass data between back end and front end:**



Dump to String

- You can transform Python dictionaries into JSON strings, using:

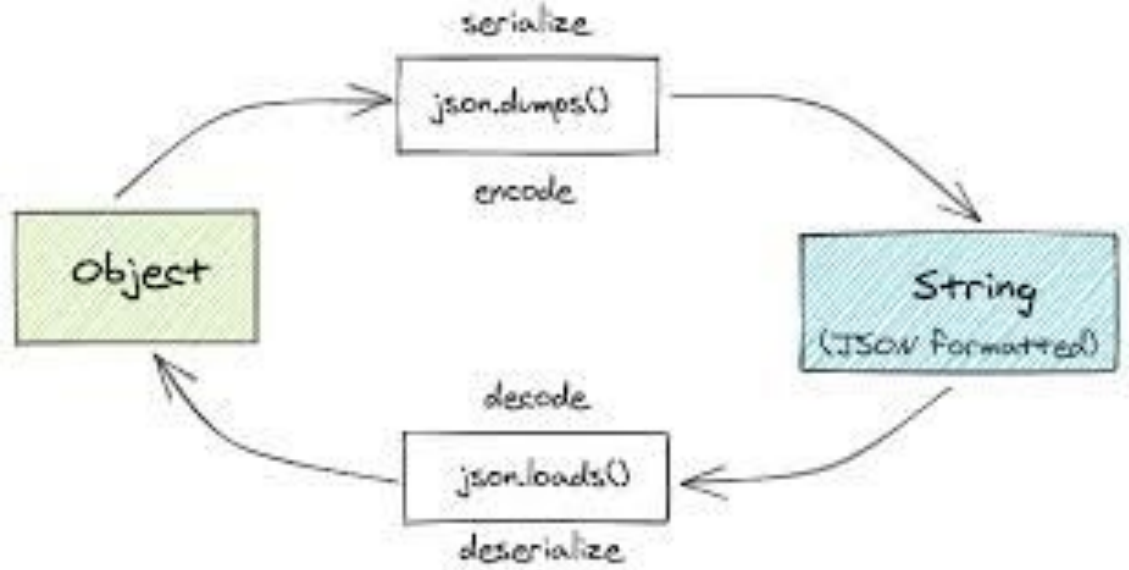
JSON.dumps( )



Load to Dictionary

- **You can transform JSON strings into Python dictionaries, using:**

JSON.loads( )

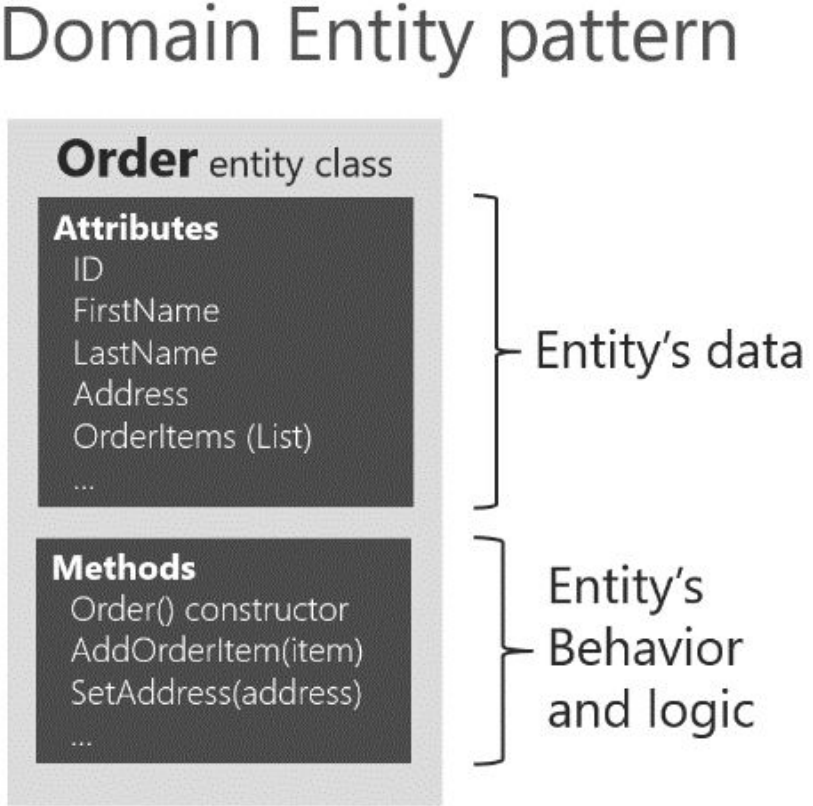


slido

What's the difference between an entity and a value object in DDD?

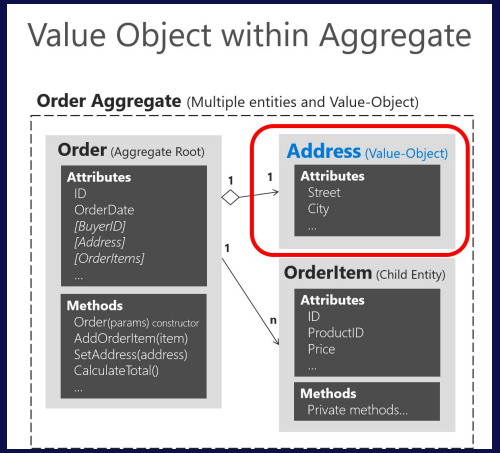
2. Entities

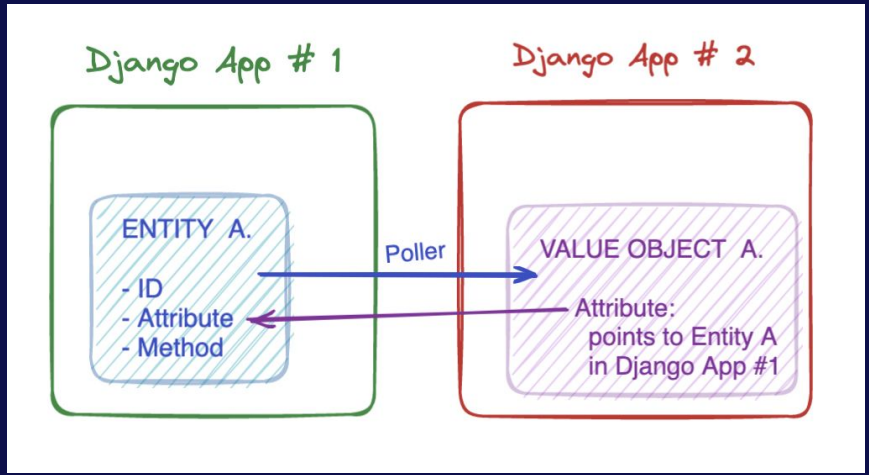
- A **representation of an object in the domain** with attributes that likely to change over time.
- **NEEDS A UNIQUE IDENTIFIER**
  "An object primarily defined by its identity is called an Entity."
  (Eric Evans)
- Should only be created with a constructor.



3. Value Objects

- Has **static value** (*immutable*)
  - Its attributes ARE its value.
  - Can be used to “describe” entities.
- Does not have an ID.
- In this module, we will be using VO’s as references to entities, to pass information about entities between subdomains.
  - Like a foreign key between models.





3. Value Objects

- Has **static value** (*immutable*)
    - Its attributes ARE its value.
    - Can be used to “describe” entities.

Don’t worry, it’ll be alright!

Don't worry, it'll be alright!

3 minute break

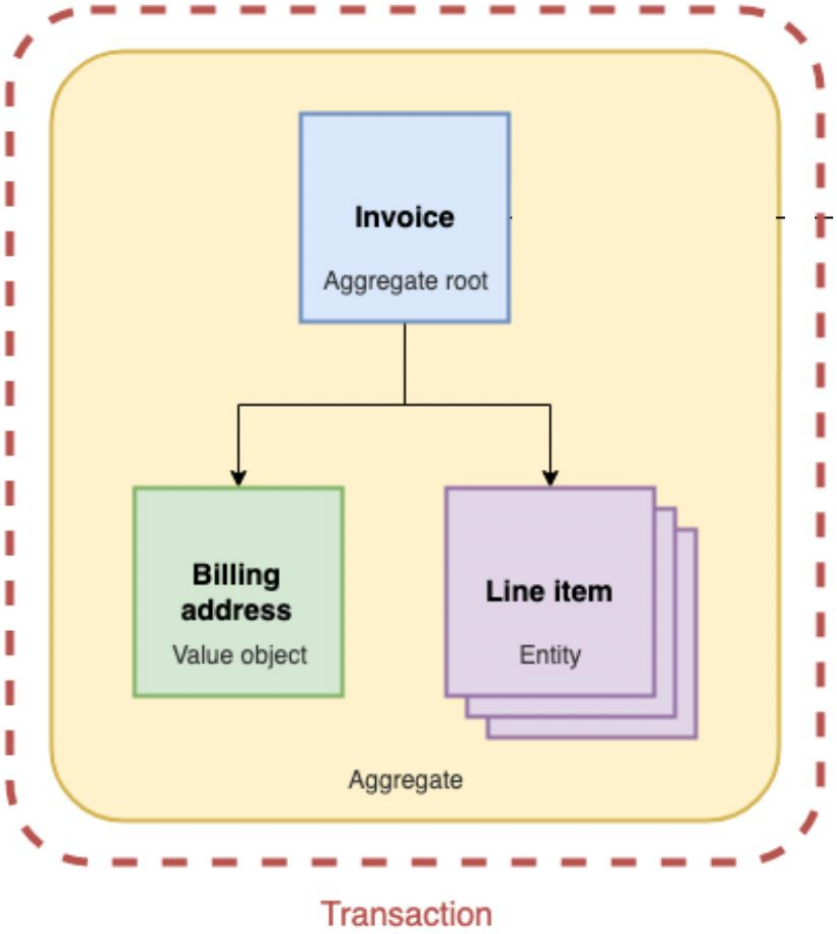
Building Blocks of Domain-Driven Design

1. **Ubiquitous Language**
2. **Entities**
3. **Value Objects**
4. **Aggregates**
   - Aggregate Root
5. **Bounded Context**
6. **Anti-Corruption Layer**

Aggregates

An aggregate is a cluster of entities and value objects that are treated as a single unit.

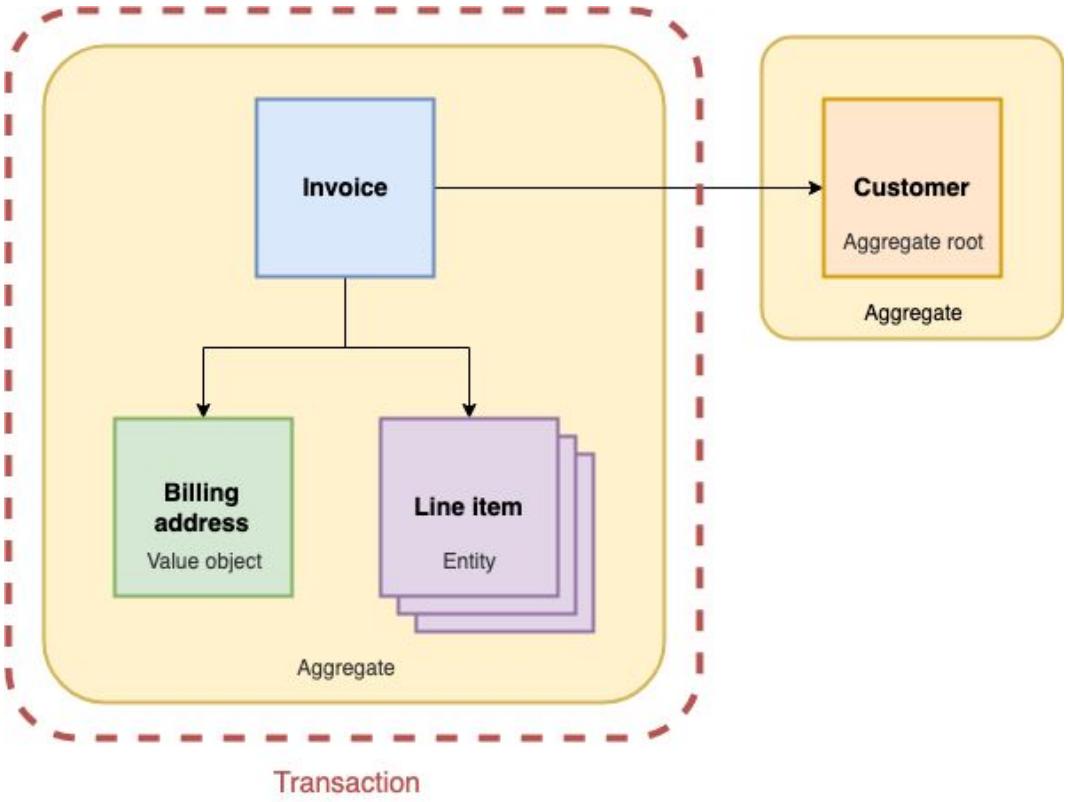
- Has a transactional boundary / interface.
- You can only access elements inside the aggregate through the aggregate root.



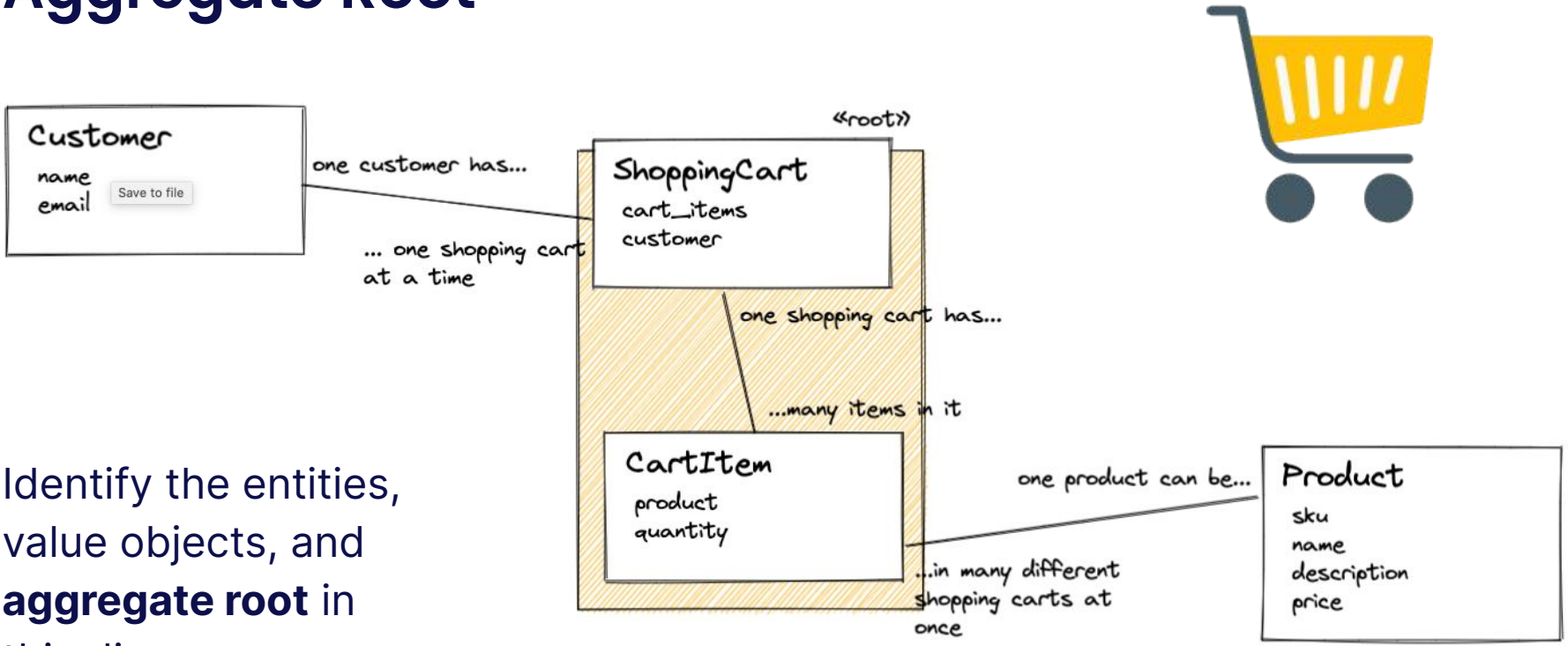
Aggregate Root

An entity or aggregate that controls access to all the other elements it owns.

- Orchestrates logic for other aggregates, value objects, and entities it contains.
- Should always be an entity or an aggregate that contains an entity. *(Must have an ID)*



Aggregate Root



Identify the entities, value objects, and **aggregate root** in this diagram.

Aggregate Root

Methods on the **aggregate root** to create add or remove cart items.

```
class ShoppingCart:
    def __init__(self, customer):
      self.customer = customer
      self.items = []

    def remove_item_by_index(self, index):
        self.items.pop(index)
        # more interesting code here

    def remove_item_by_sku(self, sku):
        for index, item in enumerate(self.items):
            if item.sku == sku:
                self.items.pop(index)
                break
        # interesting code here

    # Other interesting code
```

Factory Method

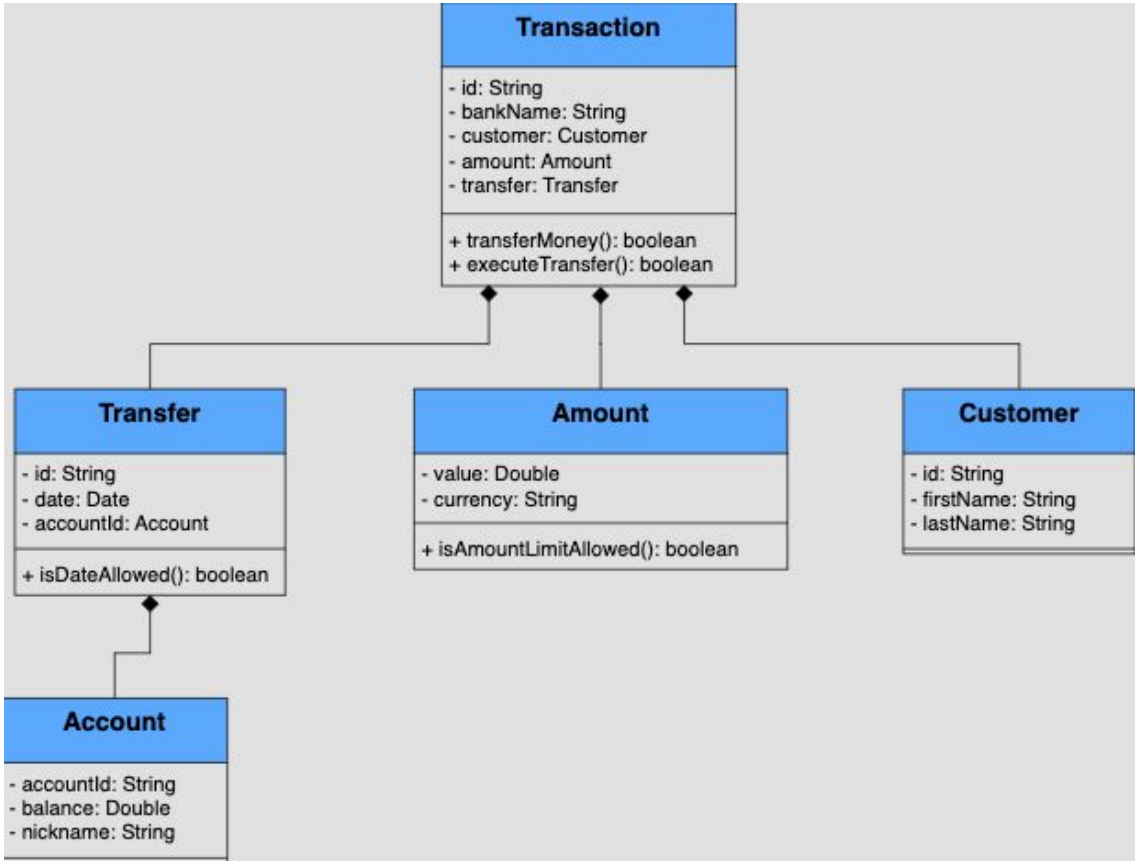
A tactical **design pattern** responsible for the creation of complex objects.

```
class ShoppingCart:
    @staticmethod
    def createShoppingCart(self, customer):
        cart = ShoppingCart(customer)
        cart.tax = TaxCalculator(customer.location)
        return cart
```

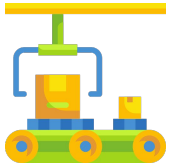
GALVANIZE

Aggregate Root

Identify the entities, value objects, and **aggregate root** in this diagram.

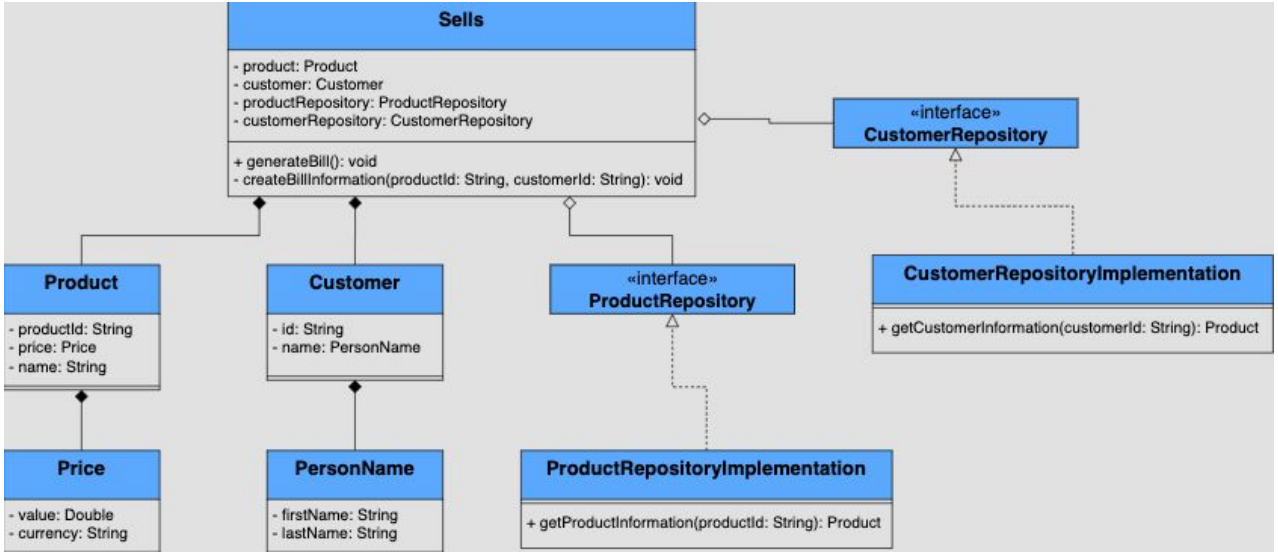


Factories



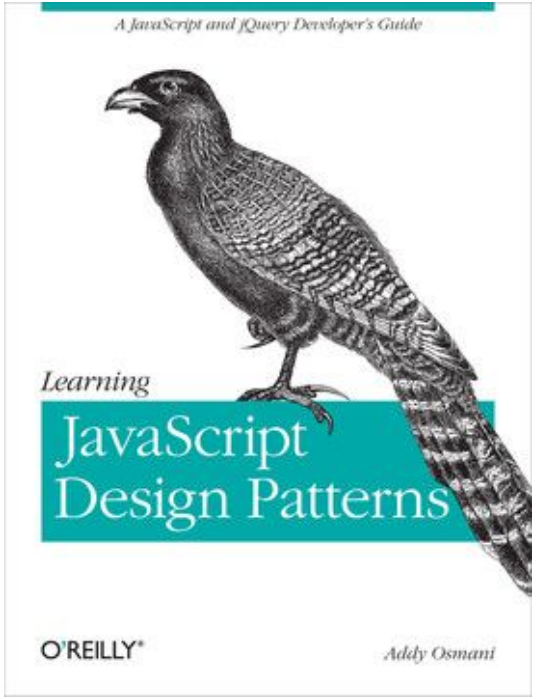
A tactical design pattern responsible for the creation of complex objects.

Isolates business logic from the complexity that may appear when new objects are instantiated.



When the Sells object is instantiated, it creates all of the dependencies it needs to execute its work. The **GenerateBill()** method prints all of the information related to a bill.

In software engineering, a **design pattern** is a general repeatable solution to a commonly occurring problem in software design.



patterns.dev

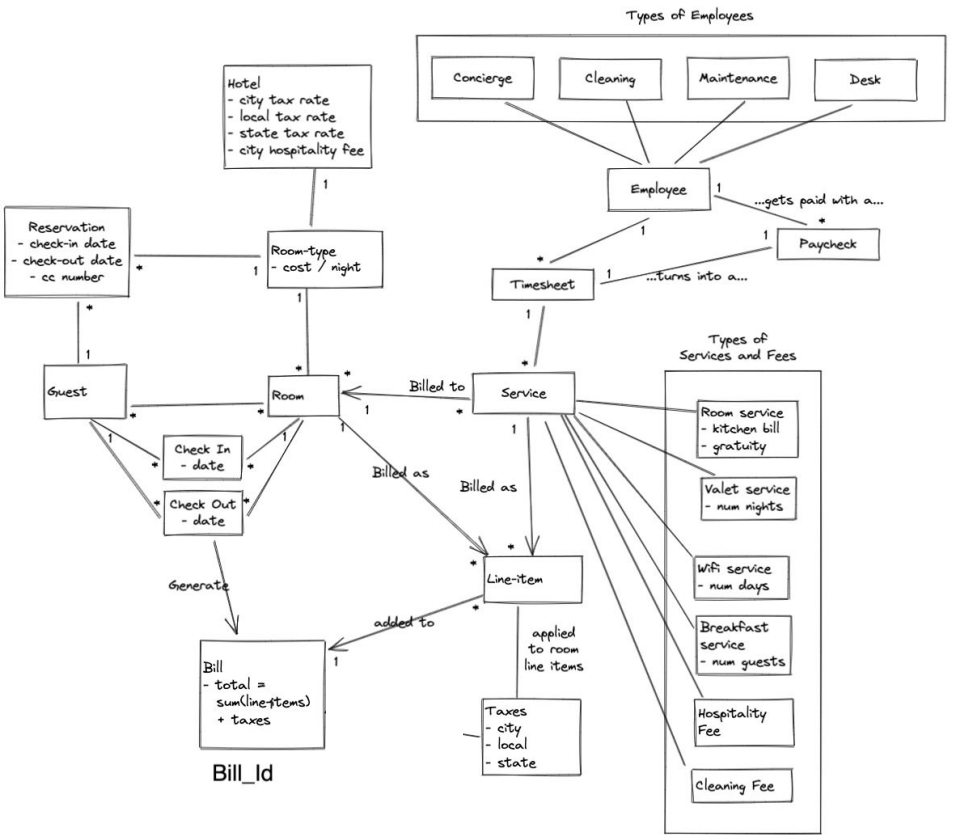
Design Patterns

Your Turn: Find Aggregates in the Hotel Billing App

Business Problems:

1. **Taxes** - Calculate the taxes and fees that can occur for rooms.

2. **Services** – Add up the services that can accrue on a bill,



HACK REACTOR BY GALVANIZE

GALVANIZE | HACK REACTOR BY GALVANIZE

Day 2

Aggregates & Factories: Design Patterns

Domain-Driven Design

1. **Activity:** Finding aggregates in Hotel Model
2. **Theory:** Aggregates & Factory Pattern
3. **Code Example:** Build a Blog with Aggregates

AFTERNOON
Technical Practice:

Building Your Own JSON Library