Course: Domain Driven Design & Microservices for Architects

Section: DDD Tactical Patterns

http://acloudfan.com/

Contact: raj@acloudfan.com

Discount Link to course:
https://www.udemy.com/course/domain-driven-design-and-microservices/?referralCode=C5DCD3C4CC0F0298EC1A

DDD Tactical Design Patterns

Strategic patterns

Decomposition of complex domains

Tactical patterns

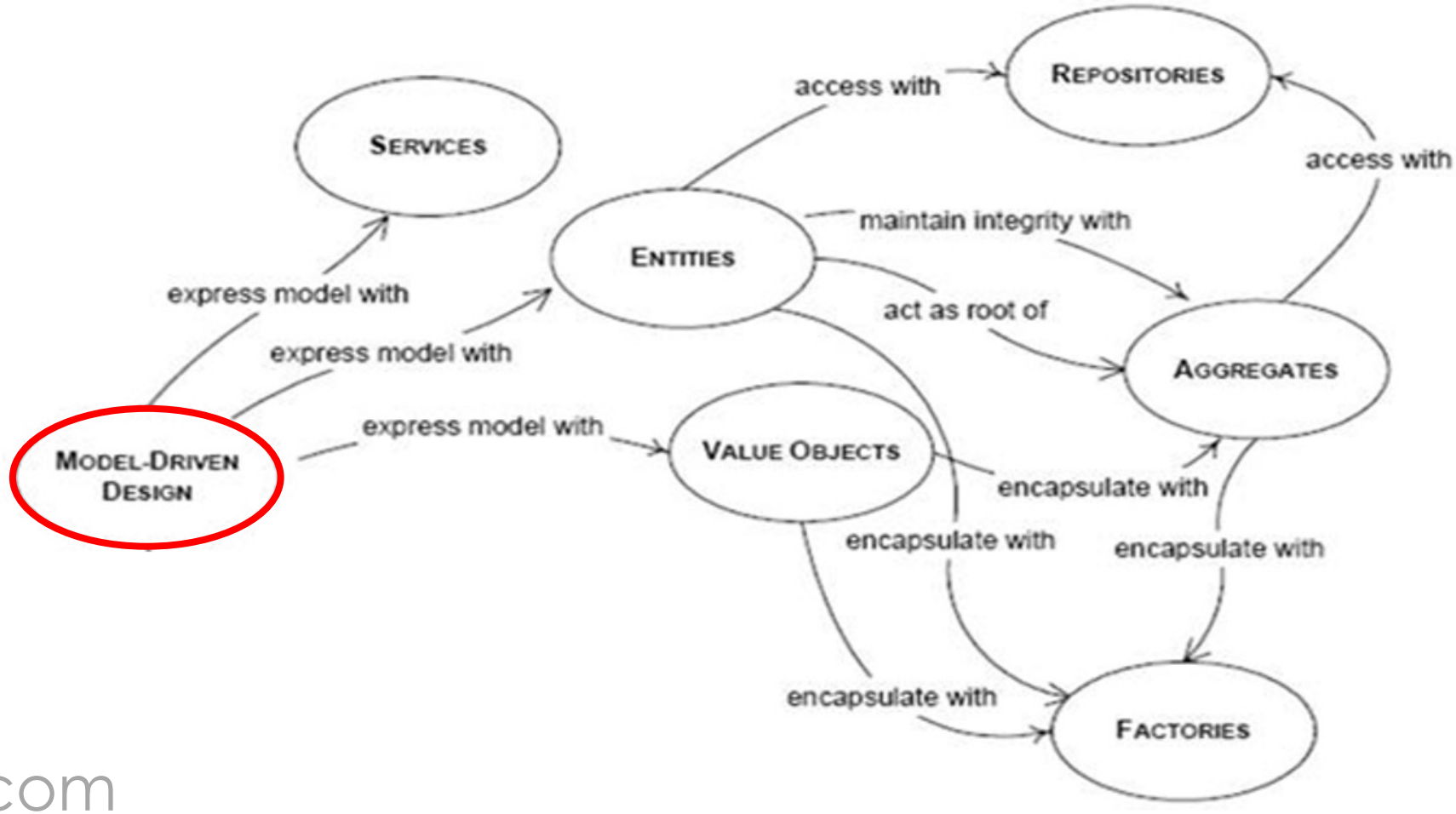
Used for modeling & realization of microservices

Model Driven Design

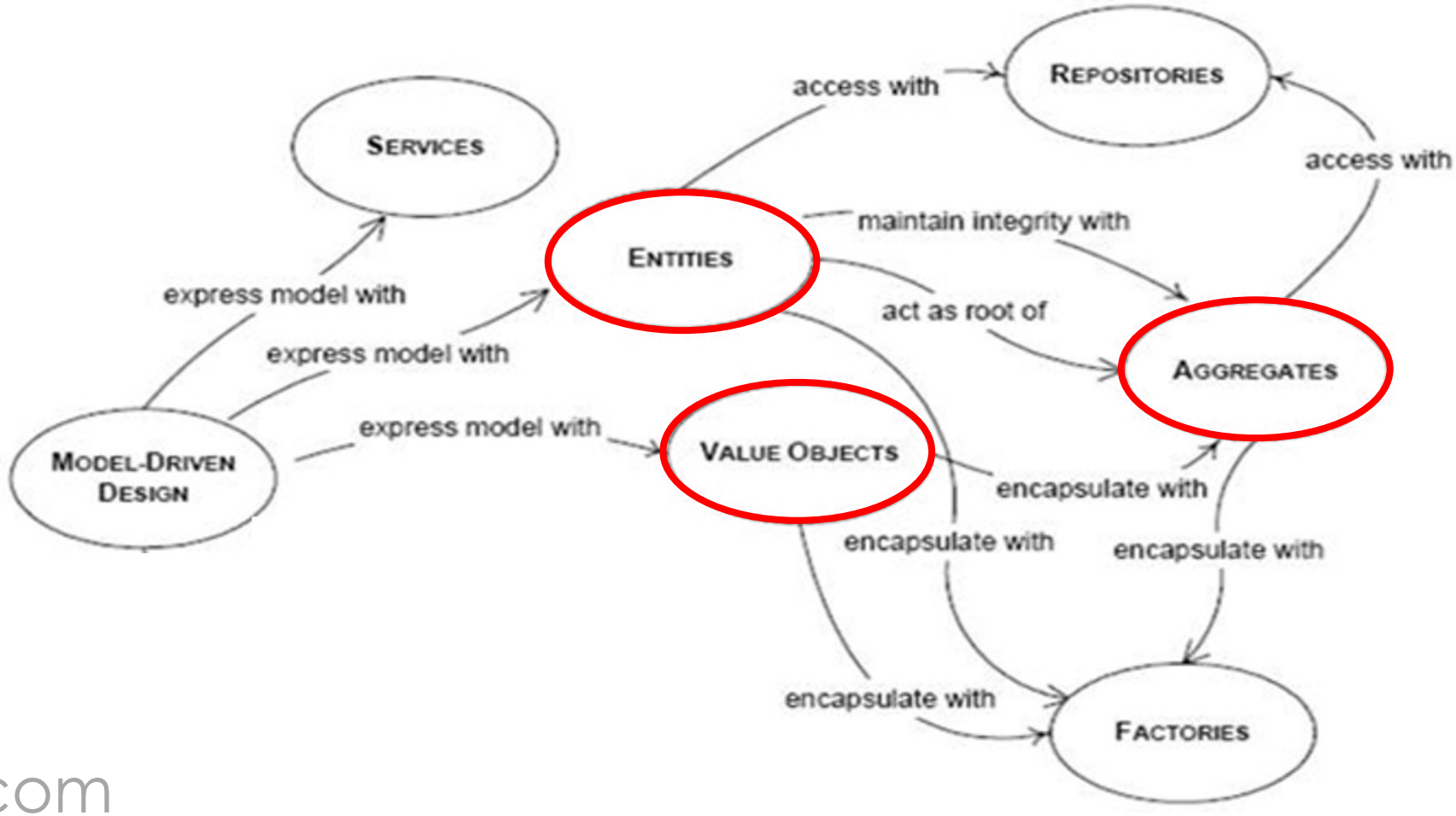
Model Driven Design provides a framework for the realization of Systems modeled using the Domain Driven Design approach

Tactical patterns are the building blocks

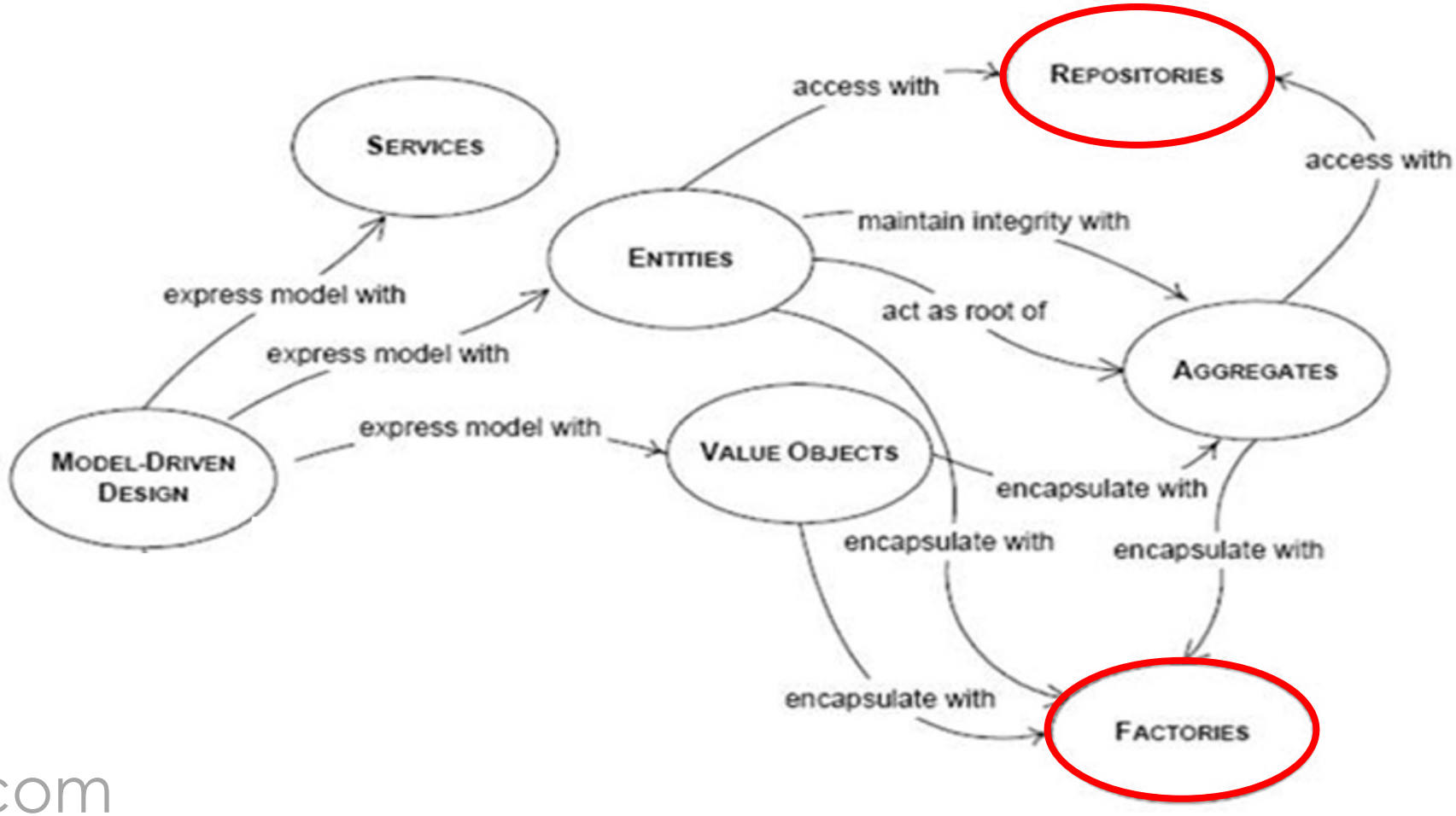
Model Driven



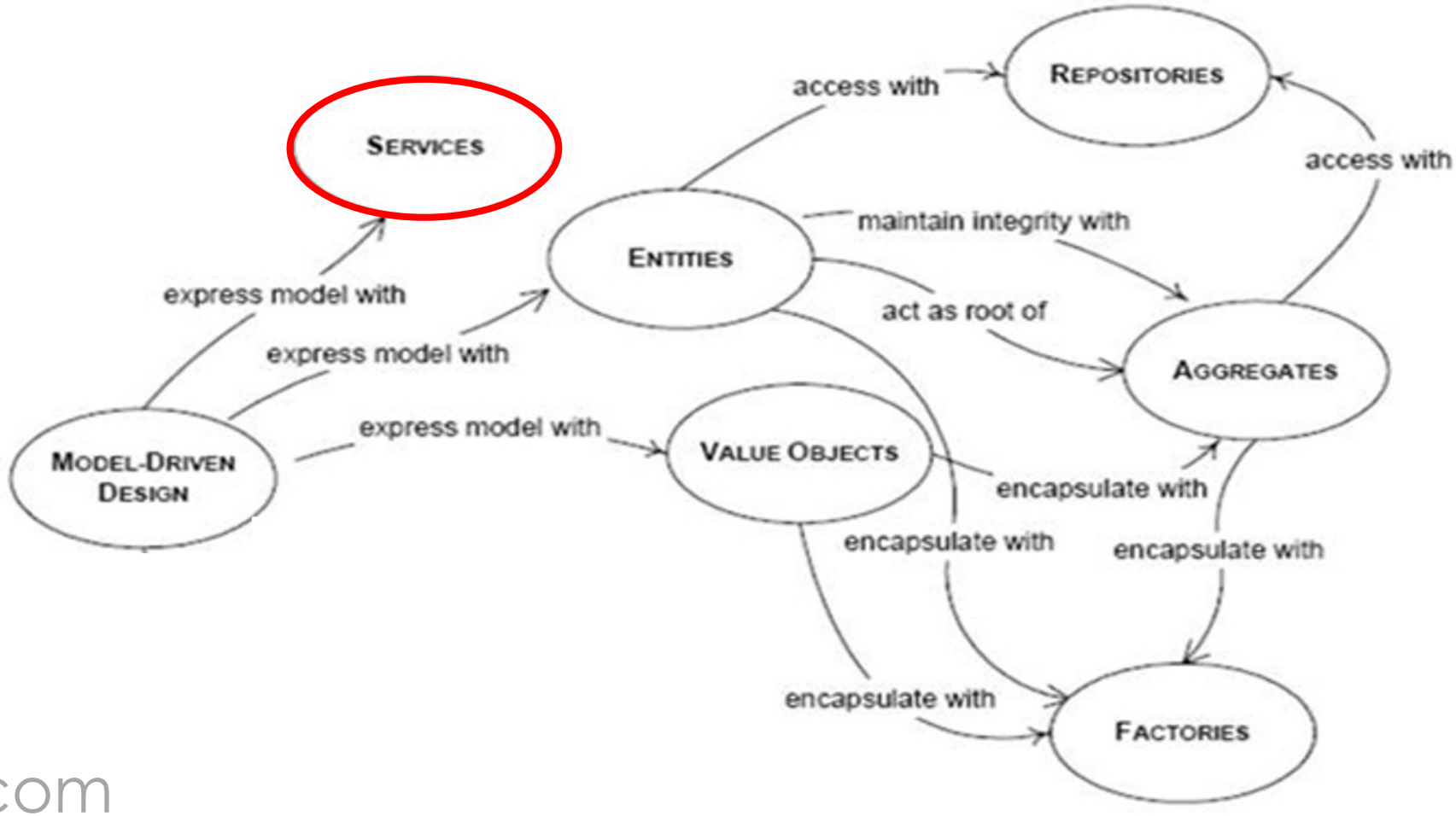
Domain Objects



Creation & Persistence



Services


SERVICES
REPOSITORIES
access with
access with
ENTITIES
maintain integrity with
express model with
act as root of
express model with
AGGREGATES
express model with
MODEL-DRIVEN DESIGN
VALUE OBJECTS
encapsulate with
encapsulate with
encapsulate with
encapsulate with
FACTORIES

Section objective

1. Entity, Value, Aggregate Objects
2. Repository pattern
3. Domain, Application, Infrastructure Services
4. Anemic & Rich Model
5. Case Study : Working ACME Sales Model (UML & JAVA)

Building Blocks for Microservices

Tactical Patterns

What is a domain?

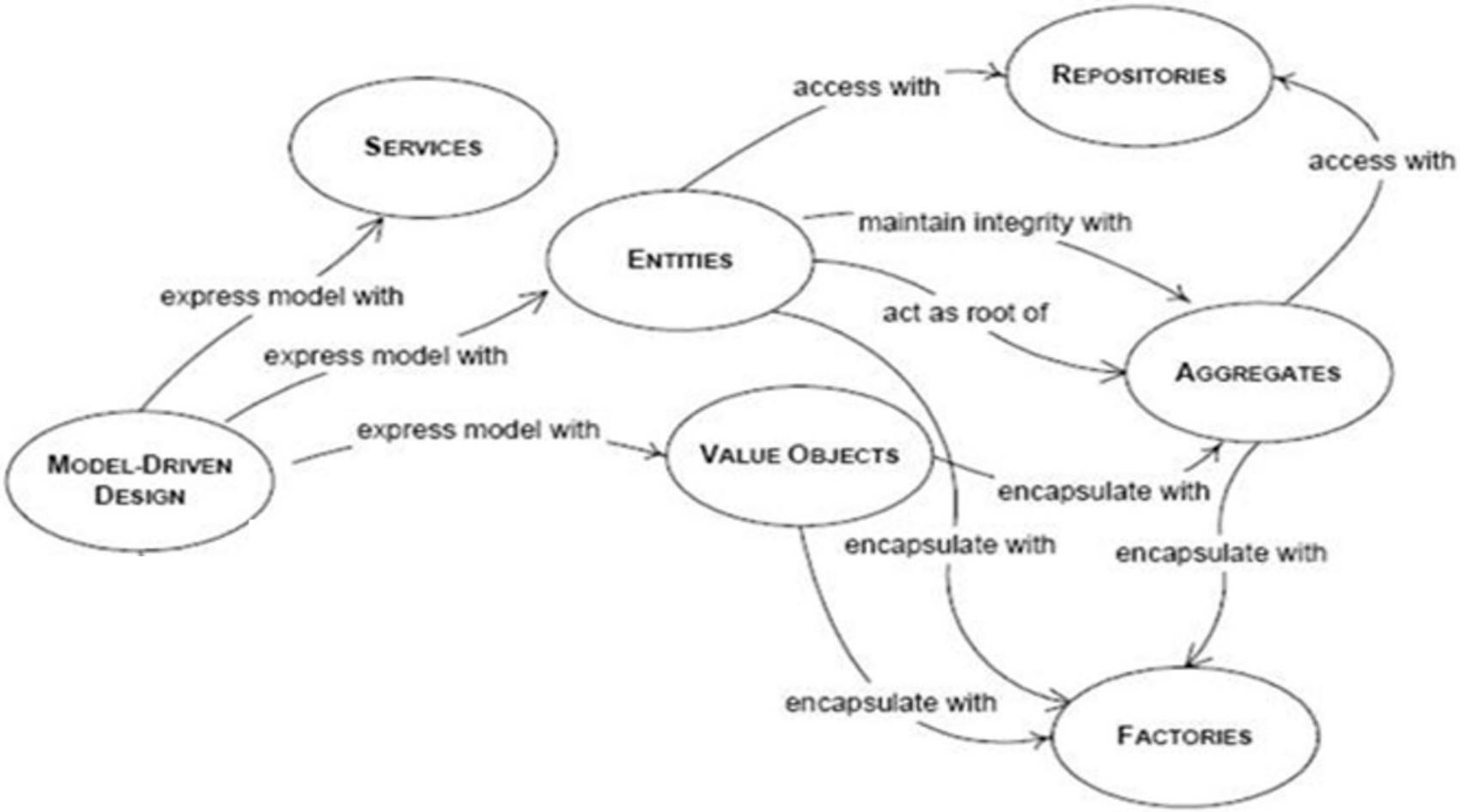
…

…

Domain Objects

Domain objects a.k.a. Business objects are the foundational elements for defining the concepts in a domain model

Domain Objects



Entities

A DDD tactical pattern

Entity objects

Characteristics of Entities

Business Logic

Entity

An Entity represents a uniquely identifiable business object that encapsulates attributes and a well-defined domain behavior

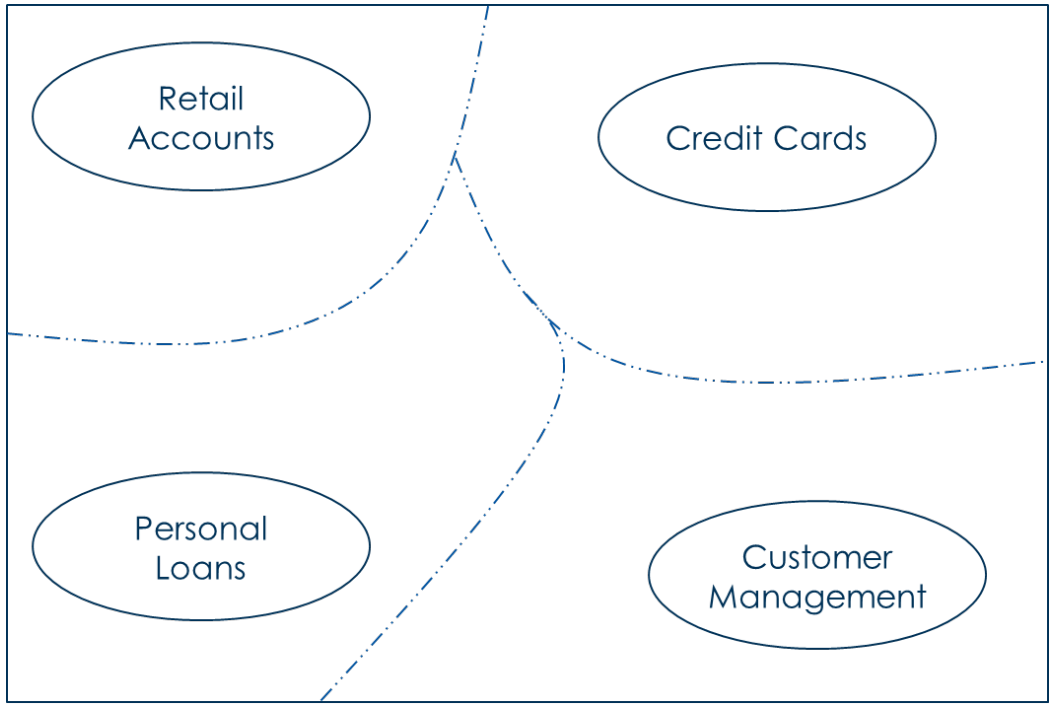
Banking	Account	Credit Card	Transaction
Retail	Order	Product	Invoice

Unique Identity

An entity is uniquely identified within a Bounded Context

Checking Account <<Entity>>
Account_Number ….

Loan Account <<Entity>>
Loan_Account_Num ….

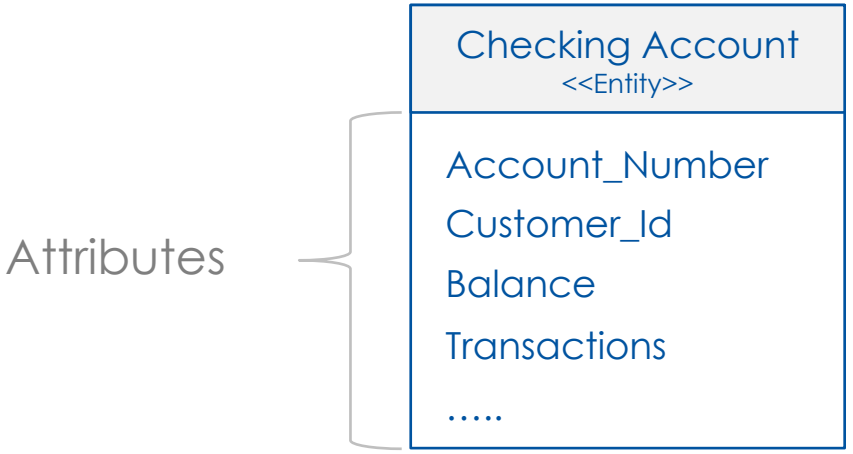


Credit Card Account <<Entity>>
Credit_Card_Number …

Entity Attributes

An entity has a set of attributes

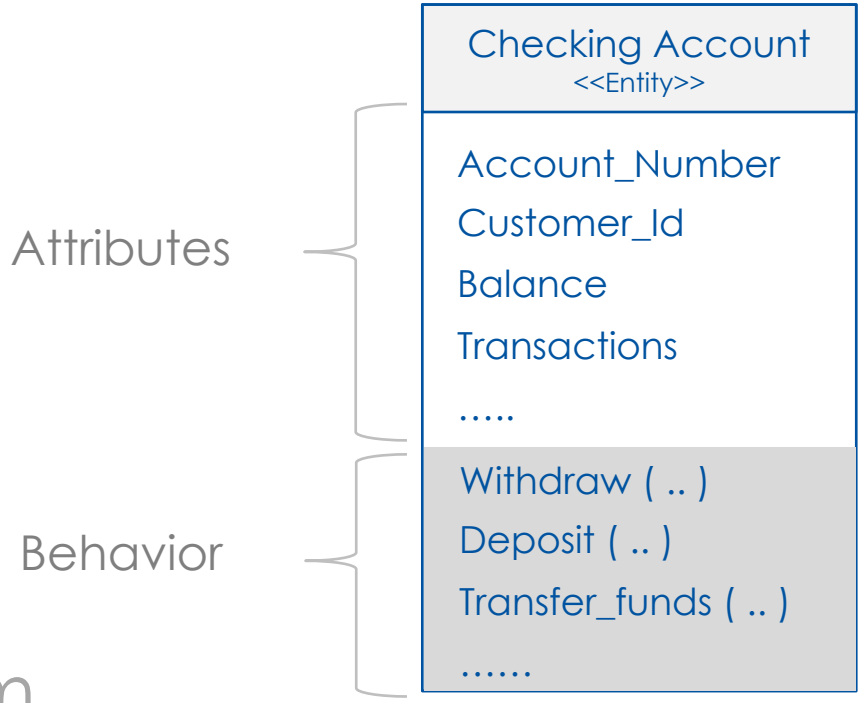
- Attributes are defined as per the Ubiquitous Language



Entity Behavior

An entity has a behavior i.e., business logic

- Entity's state is managed by way of operations



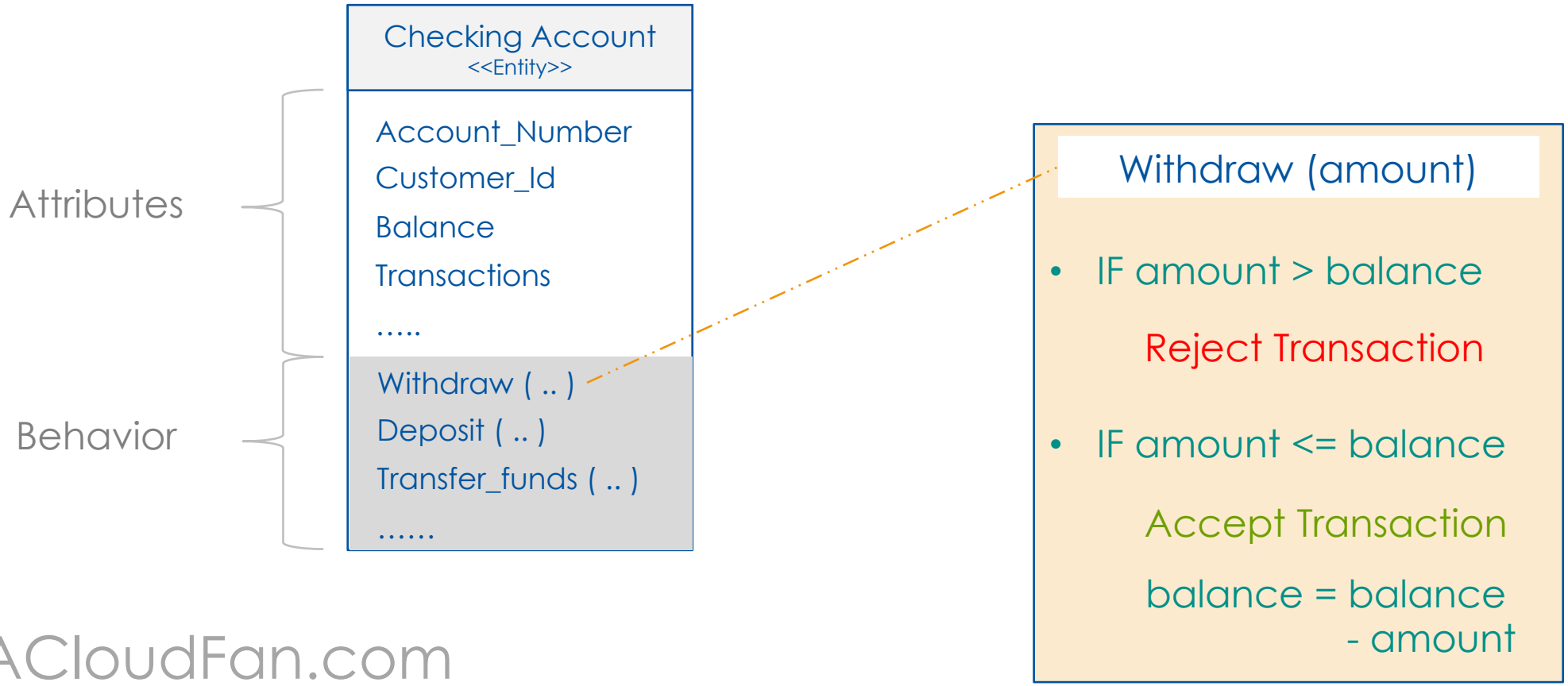
What is Business Logic?

a.k.a. Domain Logic

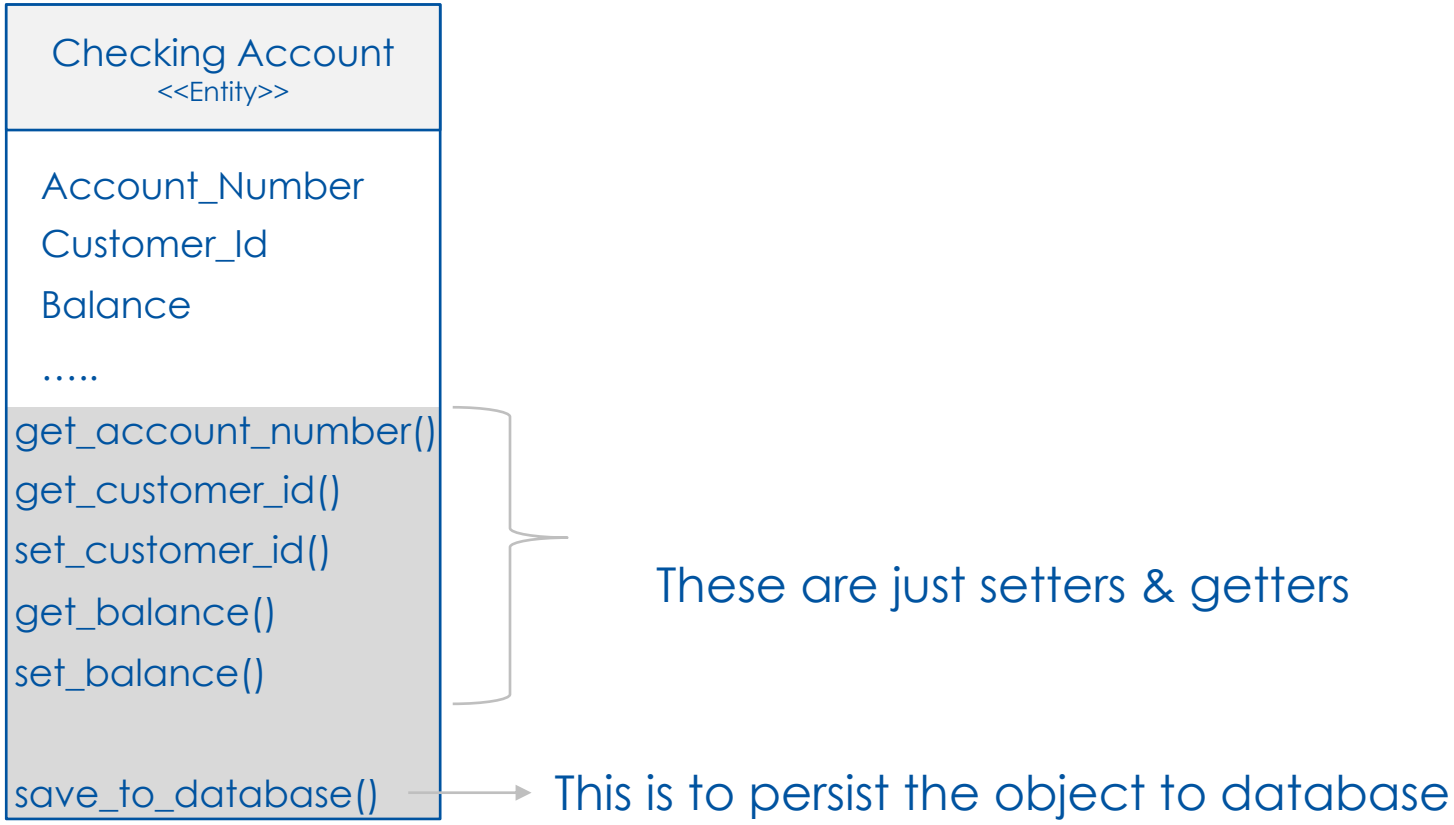
- Business Rules — E.g., Withdrawal will fail if Balance < Withdrawal Amount
- Validations — E.g., Withdrawal amount cannot be <= 0
- Calculations — E.g., Calculate the compound interest for the account
- Operation/Logic — E.g., Withdrawal transaction logic

Example: Business Logic | Behavior

The behavior implements the business logic



Quick Quiz



Does this Entity expose business logic?

Entity - Bounded Context Relationship

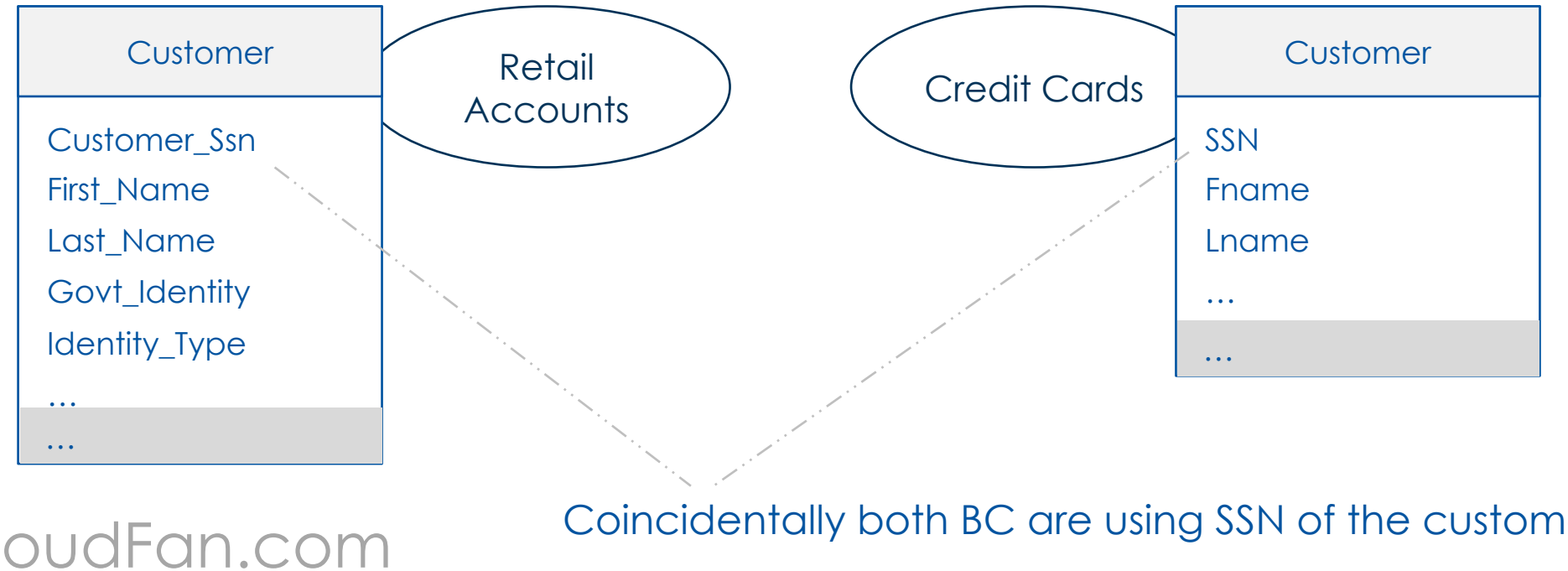
An entity is meaningful within a Bounded Context

- Definition of the entity is NOT guaranteed to stay the same

Entity - Bounded Context Relationship

An entity is meaningful within a Bounded Context

- The identity of the entity *may* cross over to other BCs



Persistence

An entity is persisted in a long-term storage

- Data in the persistent storage represents current state of entity

Account_Number	Customer_Id	Balance	…	…
1617738749	117-11-111	$3000		
1744688499	111-37-589	$128.72		
8653884268	274-14-113	$11,322.81		
…	…	…	…	…

Quick Review

Entities are meaningful within a Bounded Context

Entities are unique within a Bounded Context

Definition of entity consist of attributes and behavior

Entities are persisted in long term storage

Value Objects

A DDD tactical pattern

Value Object

Characteristics

Lifecycle

Value Objects

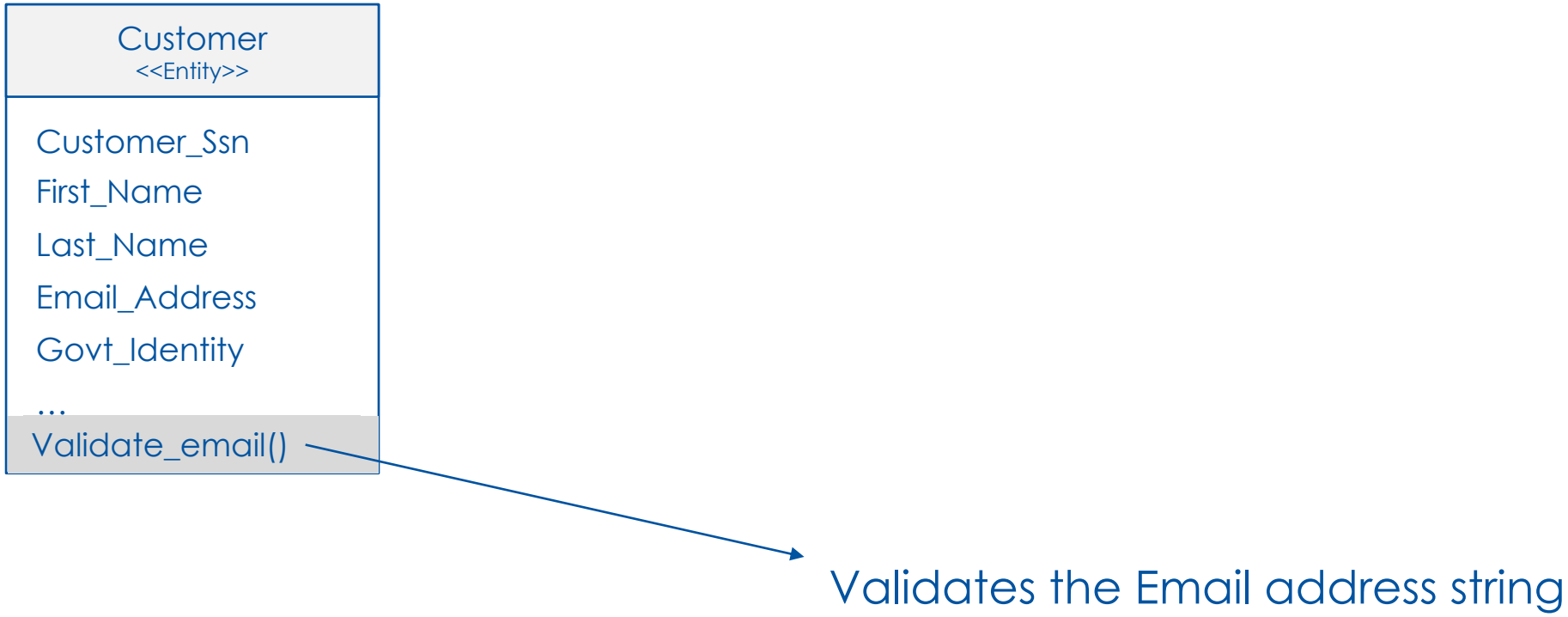
Value objects unlike Entity objects have no conceptual identity in the Bounded Context

Value object attributes | behavior does not map directly to the core concepts in the Bounded Context

Quick Quiz

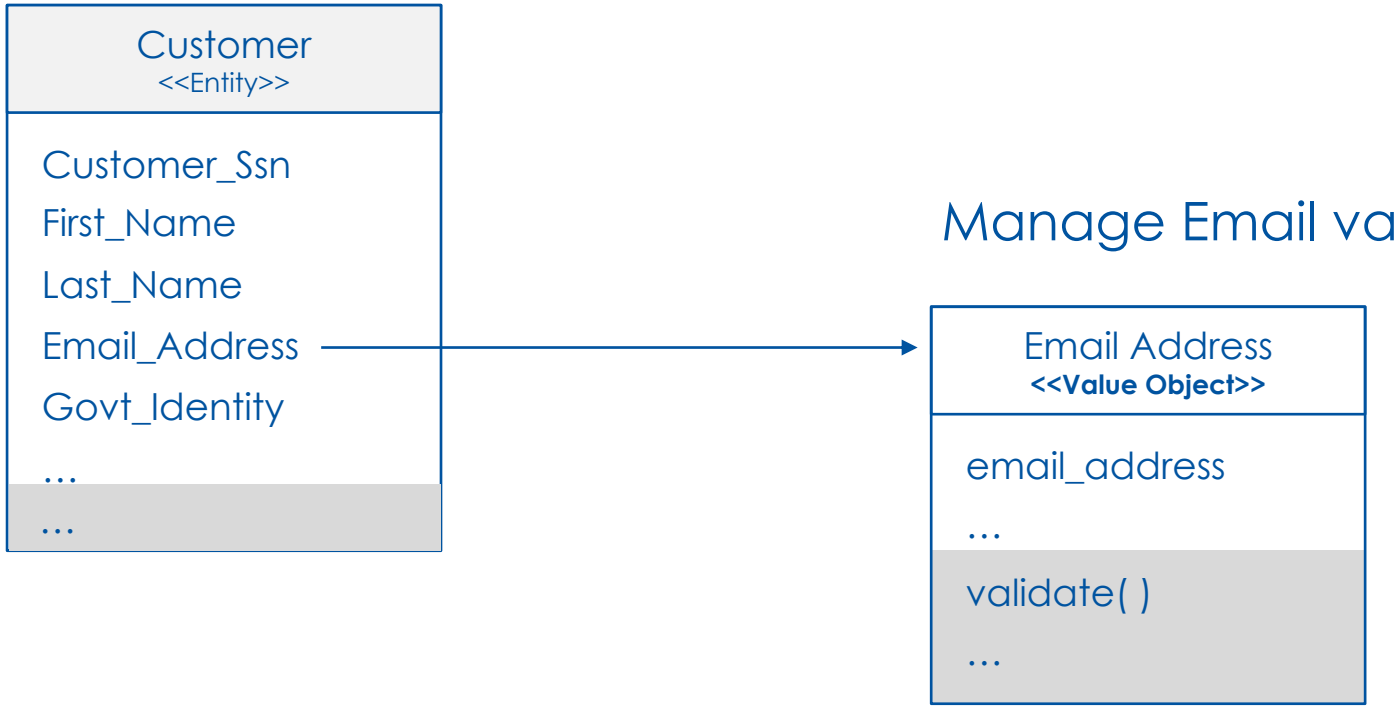
Where would you put the validation logic for the Email address?

Option#1: Email Validation Logic in Customer <<Entity>>



This is a Technical validation NOT related to any business concept!!!

Option#2: Email Address as Value Object

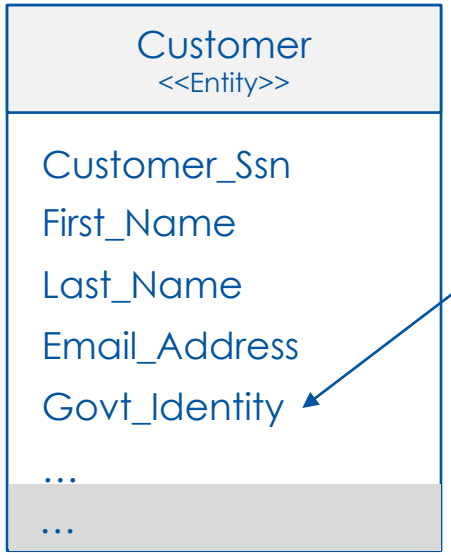


Manage Email validation in an independent object

NO business logic from Bounded Context perspective

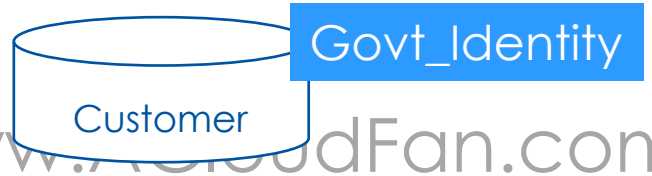
A Value Object keeps the Entity object cleaner | simpler

Example: Government Identity



Customer MUST produce a valid Government issued identity document

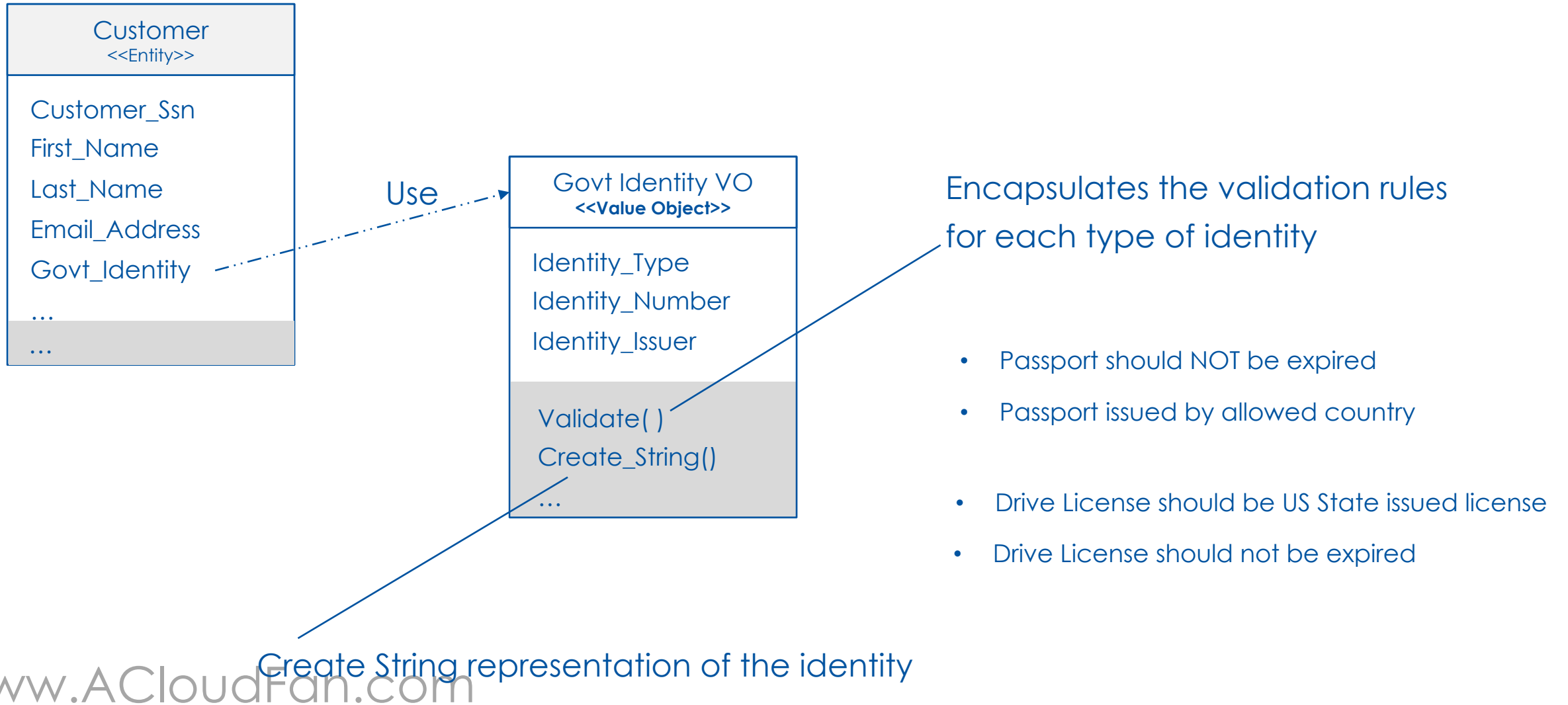
Customer entity stores identity information as a string



US-Passport# 12345678

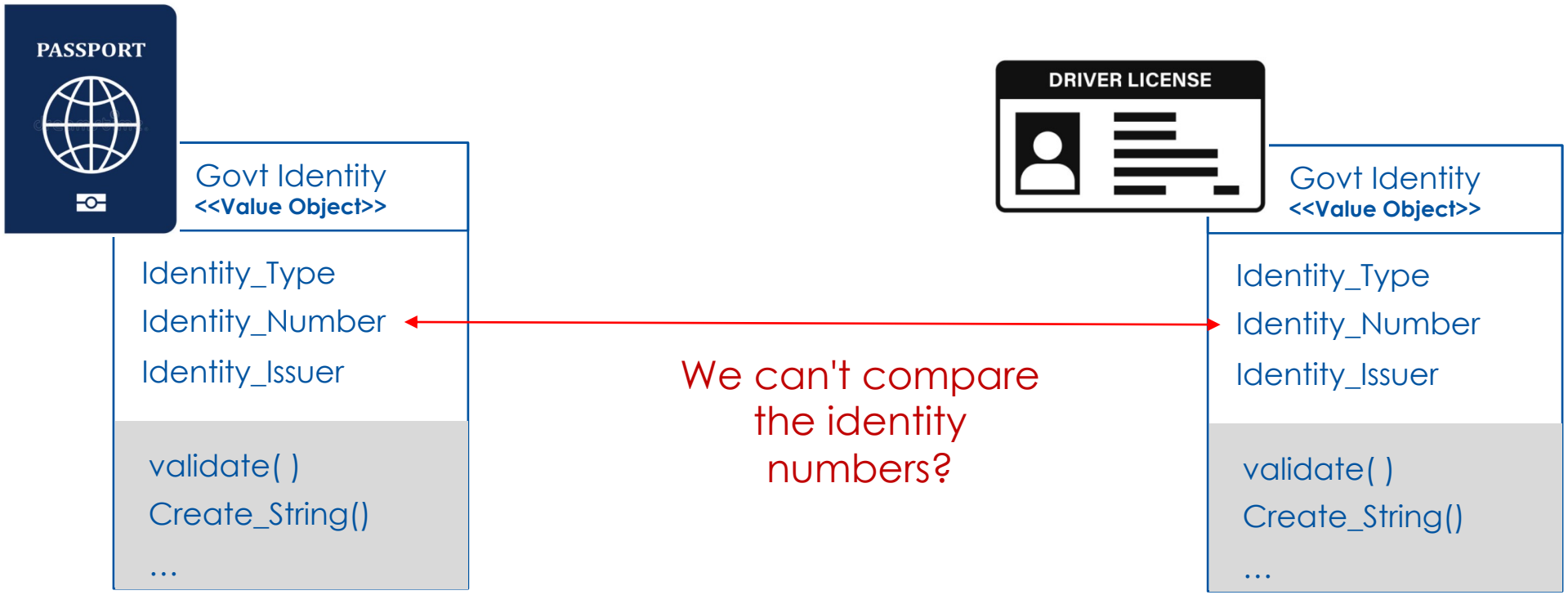
NJ-DL# 5678901

Attributes & Behavior



Value Object Identity

Does NOT have a unique identity; equality check is based on attributes



Value Object Persistence

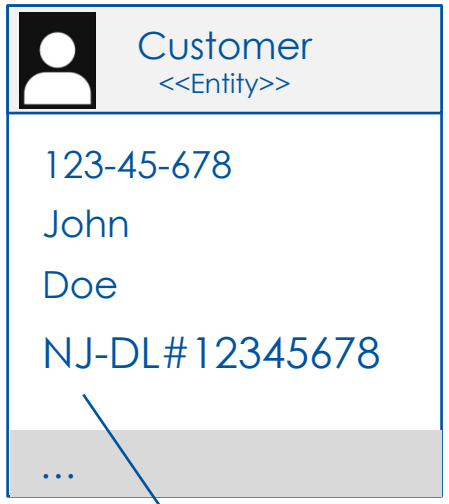
Not persisted as independent object in a database

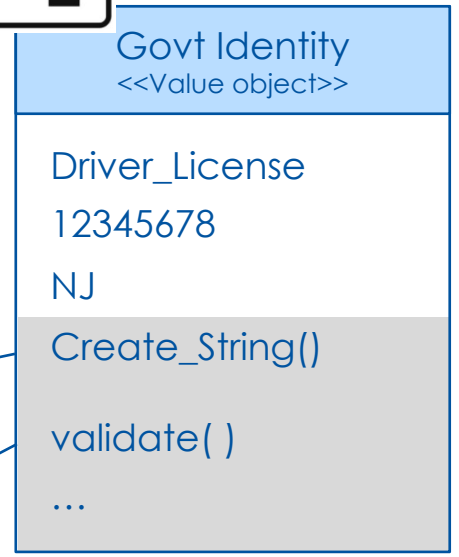
- Persisted as part of Entity Object
- OR not persisted to the long term storage

Lifecycle

Value objects have a short lifespan; NOT persisted in storage

Identity			
123-45-678	John	Doe	NJ-DL#12345678
…	….	…	….





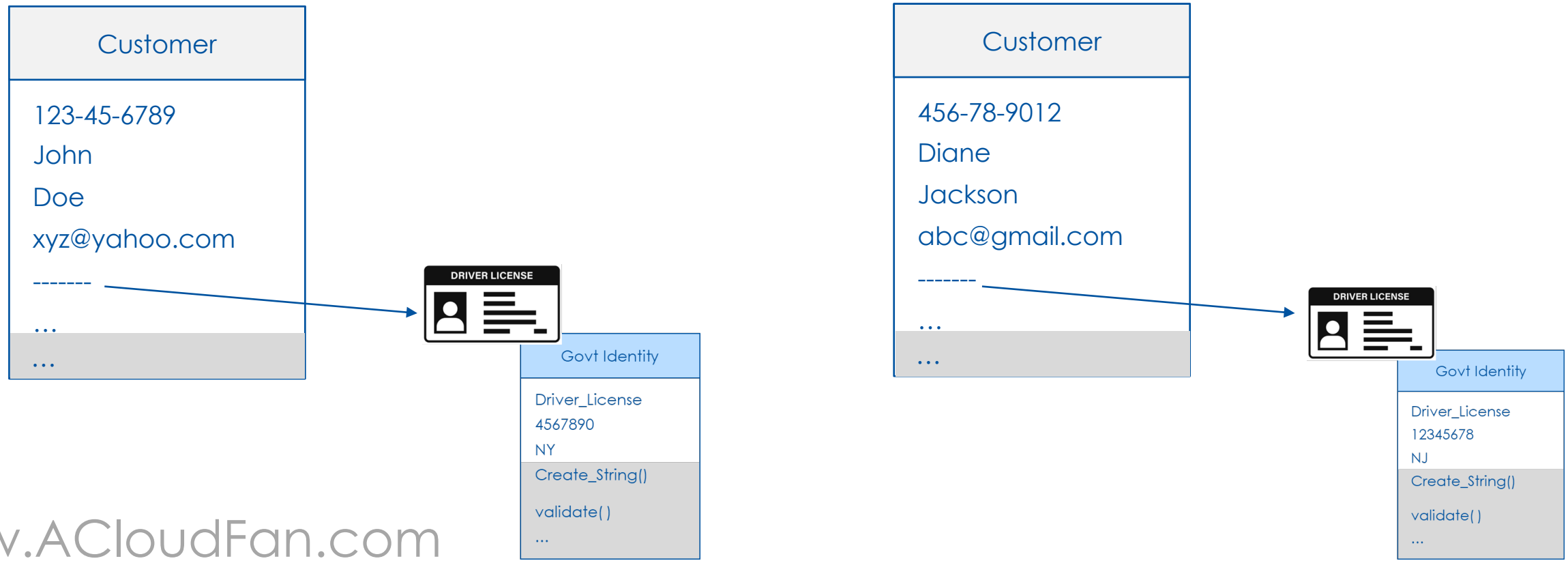
Immutability

Value objects are immutable

- All attributes put together gives a meaning to the Value Object
- To keep the code simple, create a new instance instead of re-using

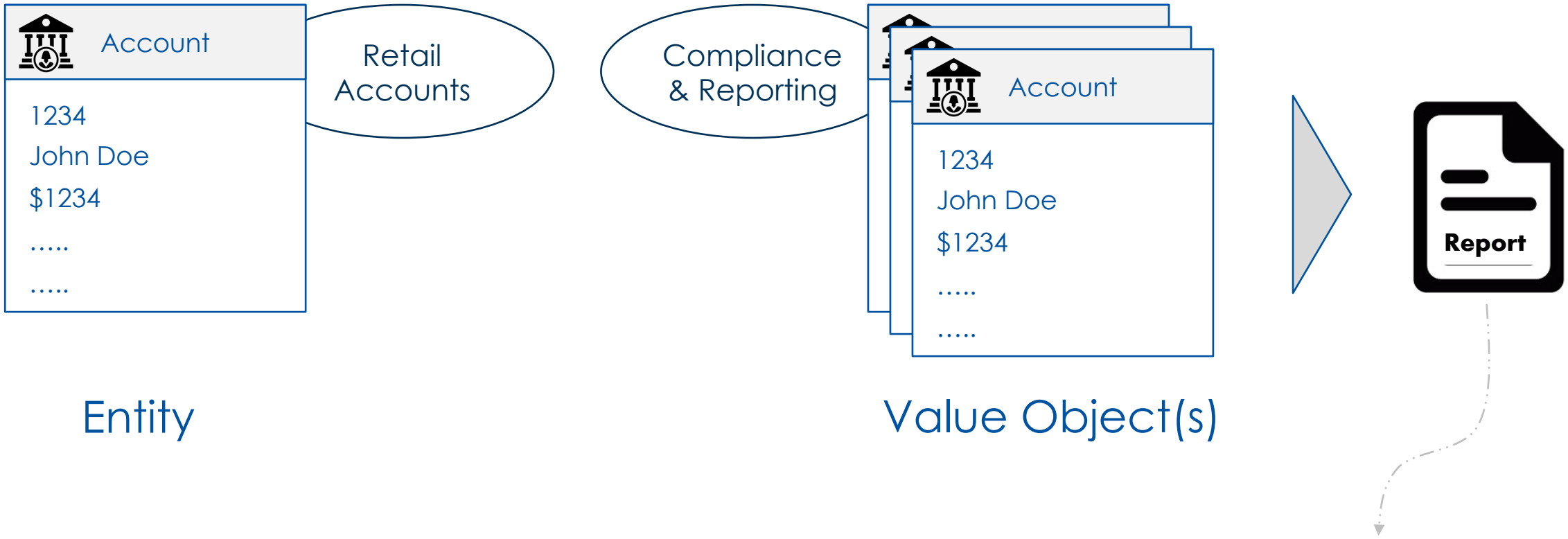
Scope of Value Object

Value objects have meaning only in the context of an Entity



Value Object & BC

Value objects in one BC may be an Entity in another BC !!



- Individual A/c does not matter from reporting perspective

Quick Review

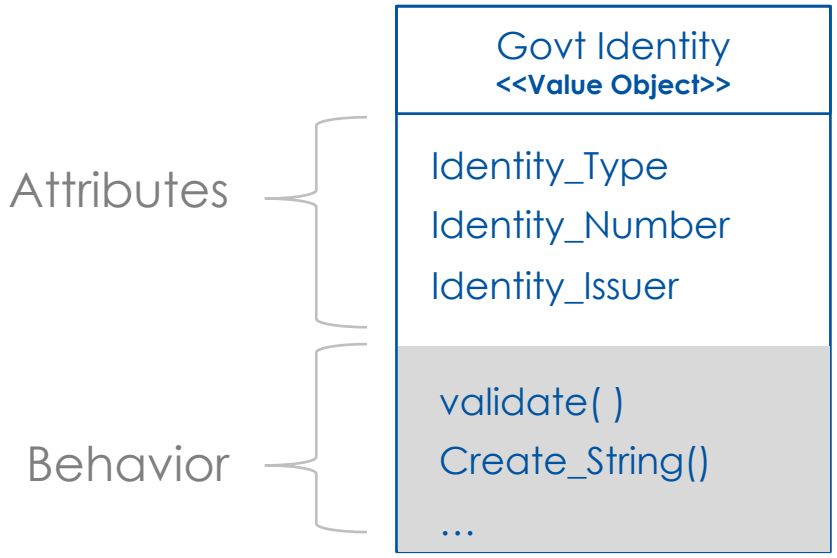
Value object does not have a unique identity

Value object is not persisted in a DB as an independent object

An Entity in one BC may be a Value Object in another BC & vice versa

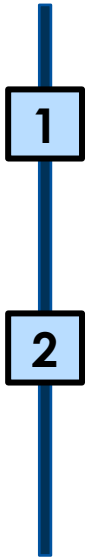
Attributes & Behavior

Like the Entities, the Value Objects have attributes & behavior



Exercise: ACME Sales Model

Identify Entities and Value Objects in ACME Sales | Products Model



Identify Entities

Identify Value Objects

John, Travel Advisor

IT Lead

We are in the business of selling vacation packages. The sale process starts with a Customer calling us. Based on Customer's desires we select the packages and describe it. If customer shows interest we start a proposal for the selected package.

Packages have a suggested retail price but our product team also puts out offers that we can apply to the packages. These offers are essentially the discounts based on various criteria.

Once the customer commits to the proposal, we gather the Pax details i.e., passengers details. All the information is gathered into a purchase order and then we get the Payment information submit the proposal for reservation and if everything goes fine we receive the Booking confirmation.

Vacation Package

Customer

Proposal

Offer

Pax or Passenger

Purchase Order

Payment Information

Booking Confirmation

Customer

Pax
or
Passenger

Offer

Vacation
Package

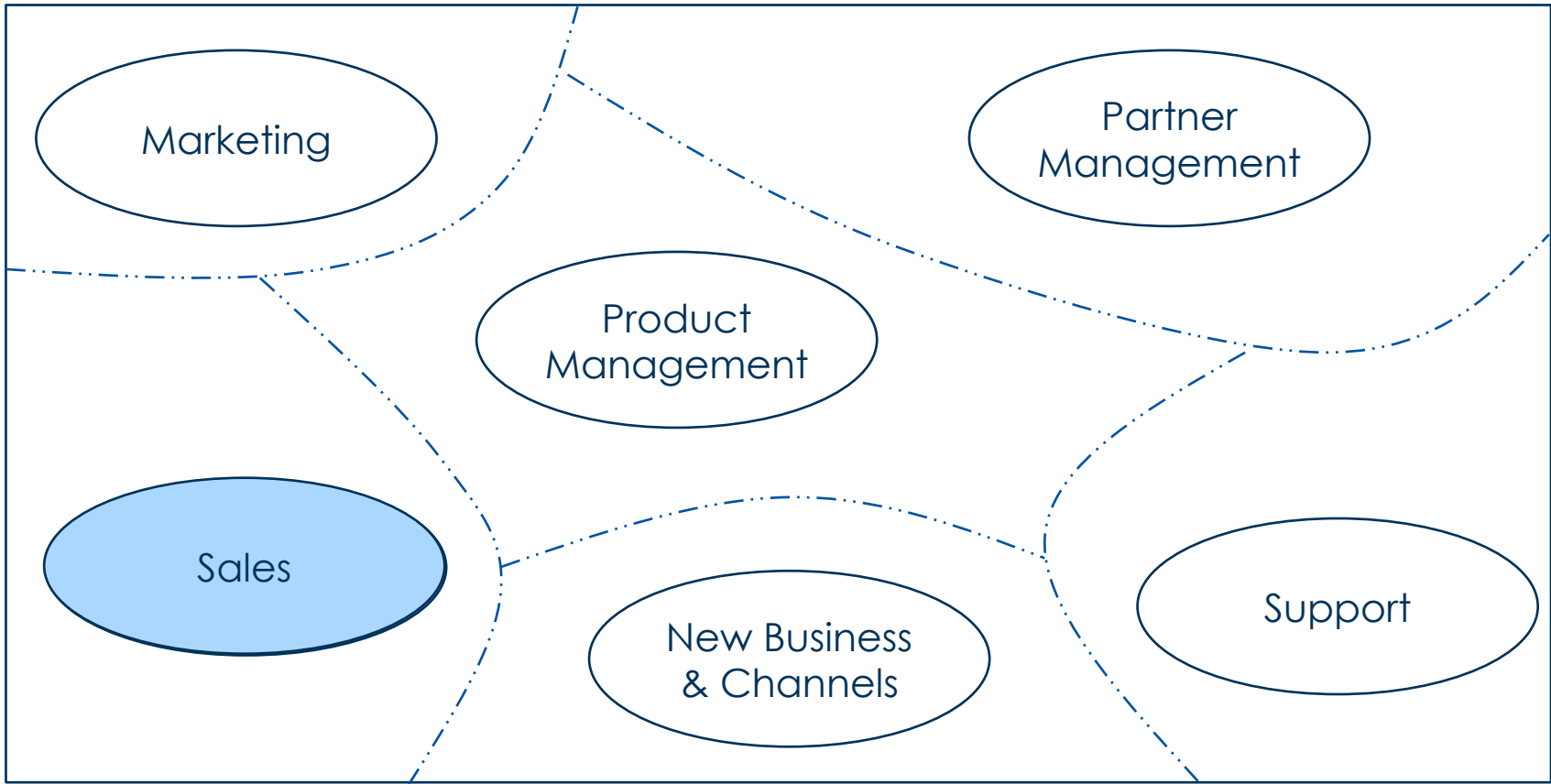
PS: Additional Value Objects may be added

Identify the Entities & Value Objects in Sales context

ACME Travel

Focus is on the Sales Context

- Entities | VO are meaningful within a context
- Value Objects may not have a conceptual identity in the BC



Customer

An individuals who initiates the sale process.

- Identified by an email or phone number
- Limited information about the customer is needed in sales context

Customer

Unique Identity

Persisted in storage

Customer <<Entity>>
Phone Number Email First Name Middle Name Last Name Contact Information .....

Customer

An individuals who initiates the sale process.

- Customer's validate physical address needs to be captured
- Address is not a core concept

Customer

Verify address using third party e.g., USPS

Does this VO need to be Persisted?

YES - as part of the **Customer <<Entity>>**

Address <<Value Object>>
Address Line1 Address Line 2 City State Zipcode

Customer

An individuals who initiates the sale process.

- Customer's Email address & Phone needs to be validated | stored

Customer

Email Address good?

Phone Number good?

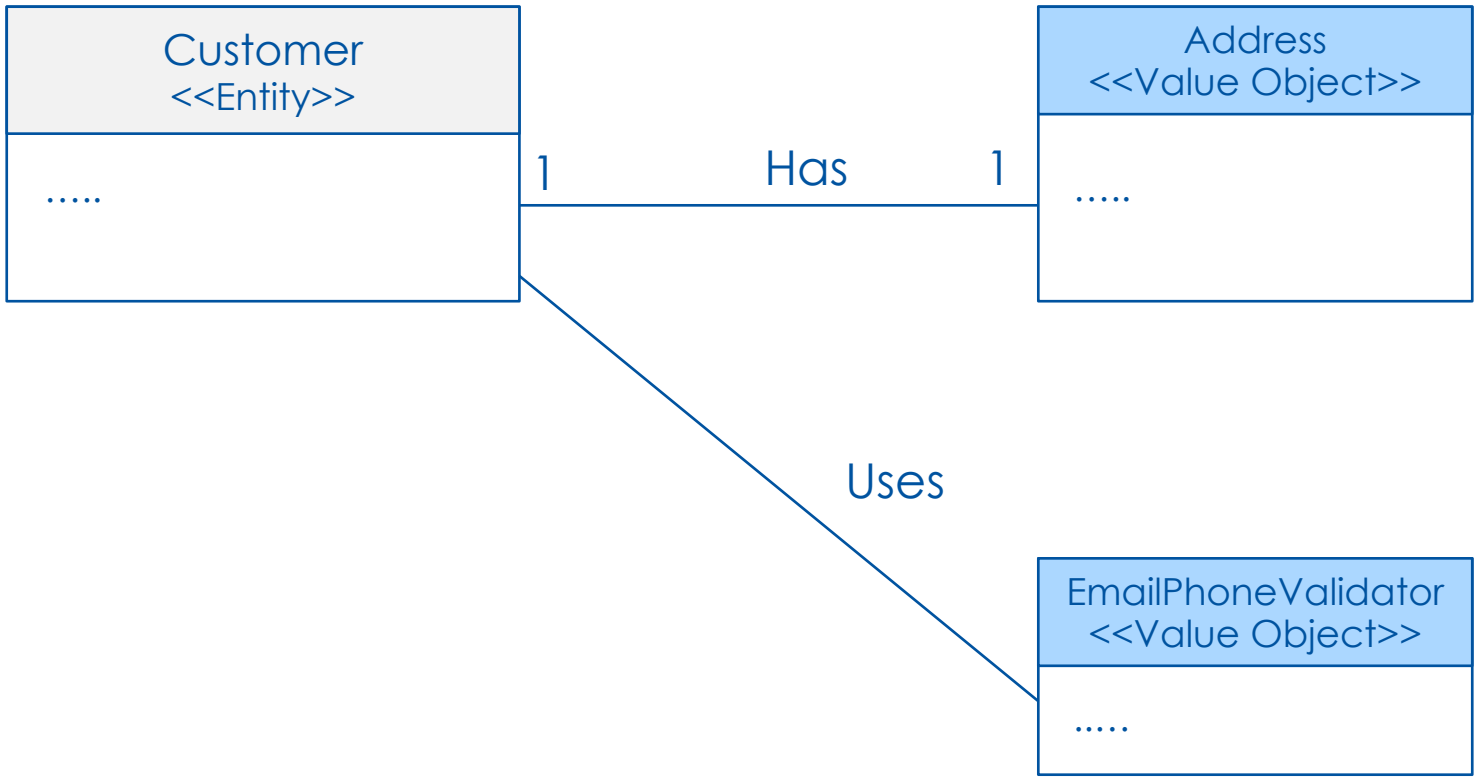
Does this VO need to be Persisted?

NO

EmailPhoneValidator <<Value Object>>
EmailAddress PhoneNumber …

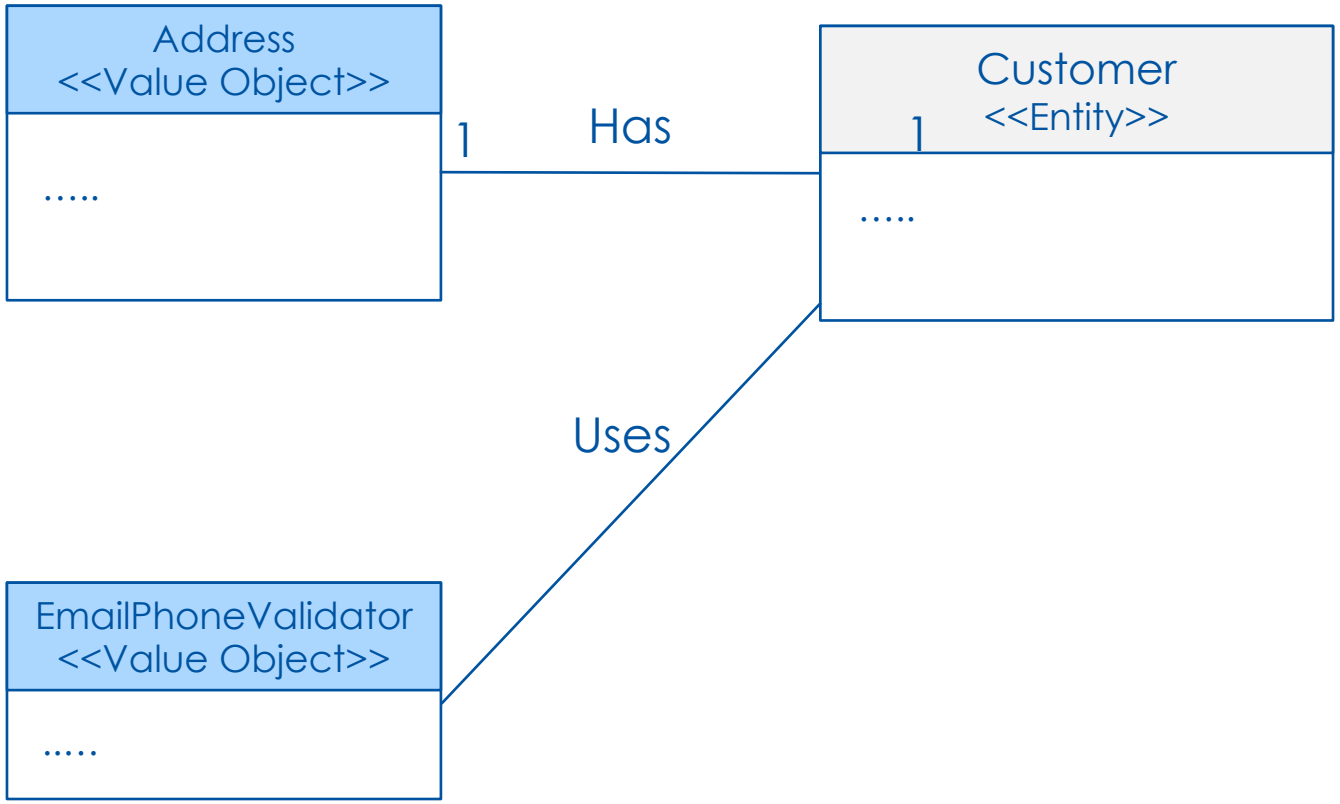
Customer

An individuals who initiates the sale process.



Customer

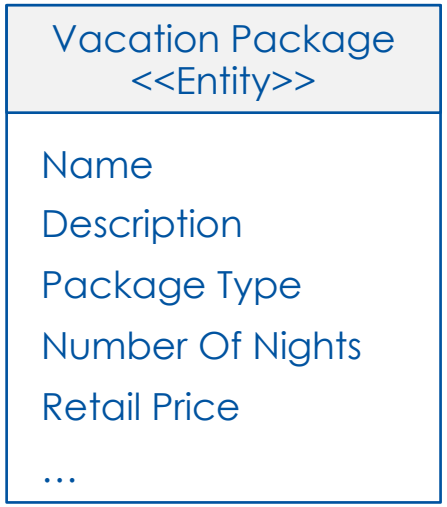
An individuals who initiates the sale process.



Vacation Package

This is the product that Acme sells

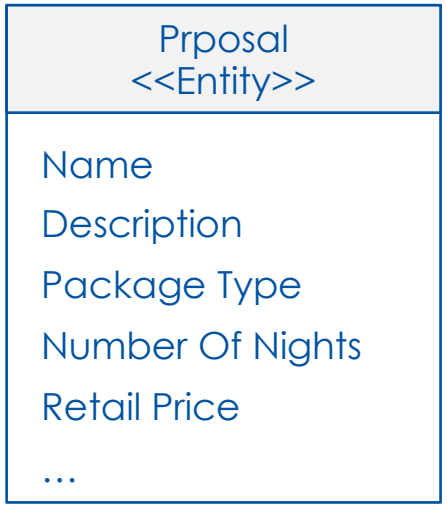
- Uniquely Identified by a name e.g., LV3NIGHT, BAHA5NCRUISE
- Package have a type
  - Resort
  - Hotel & Air ticket
  - Cruise
  - Hotel, Air ticket & Car rental
- Package have a fixed number of nights
- Package has a retail price



Proposal

This is created for specific vacation package | pricing

- Proposal contains information needed for pricing the package
- Proposals are valid for 14 days
- Multiple proposals may be created; unique identity



Offer

Discount applied to the proposals

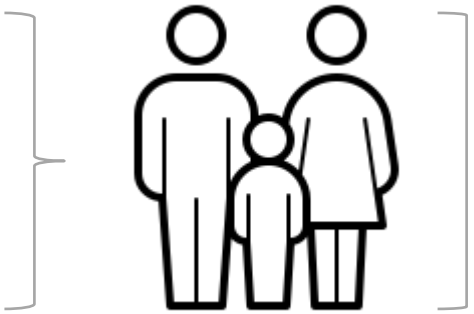
- Managed by Acme Product team
- Have a unique identity assigned by Product team
- Applicable only in the context of Proposal
- Business Rule : Maximum 2 offers can be applied

Pax
or
Passenger

This person is the traveler

Jane calls to buy the package

Customer



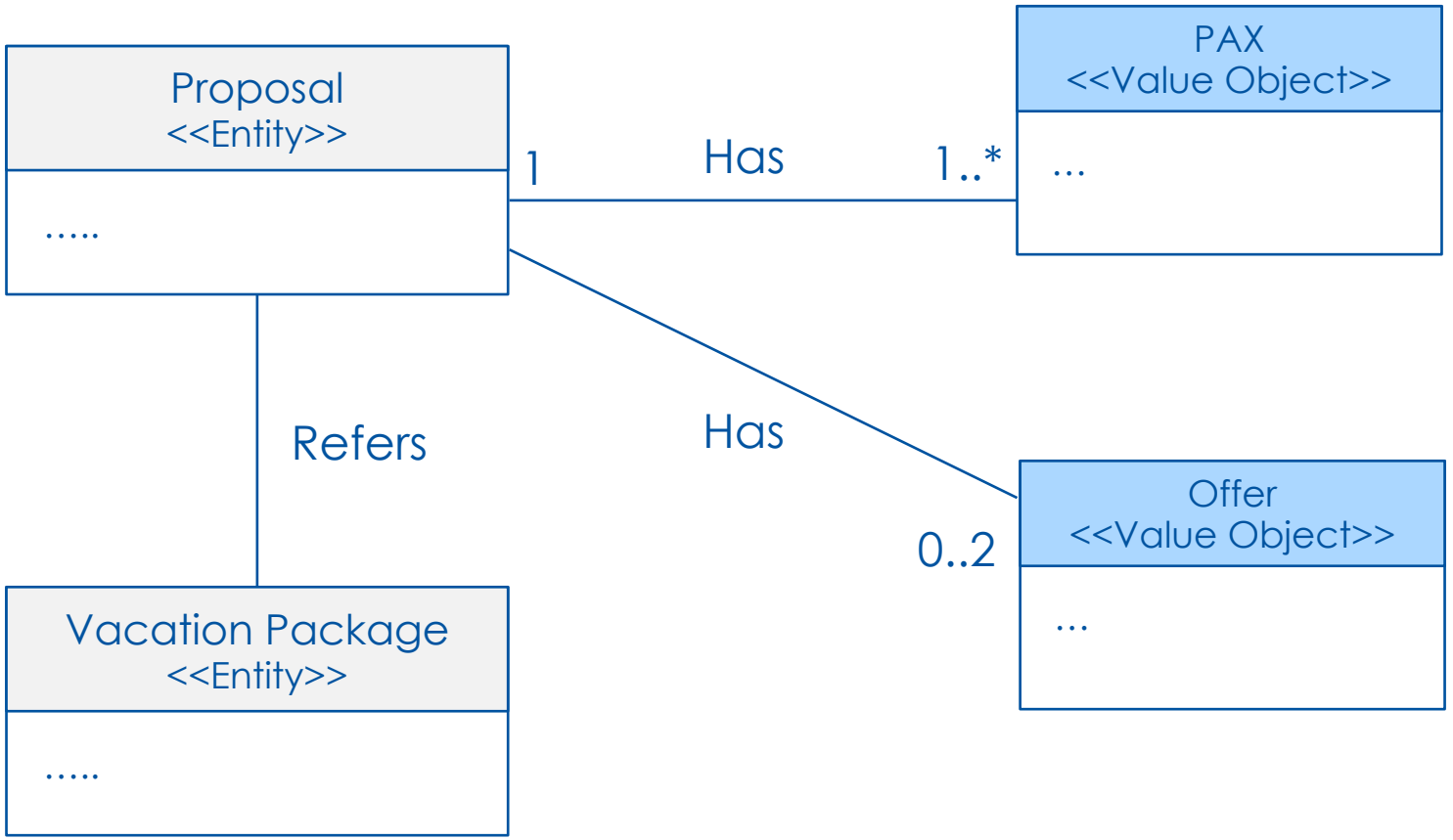
Jane, Jack (spouse) & Peter (son) will be the PAX

- Pax does not need to be uniquely identified in the context of proposal
- Pax are an important part of proposal from pricing perspective

Pax <<Value Object>>
Name Age …

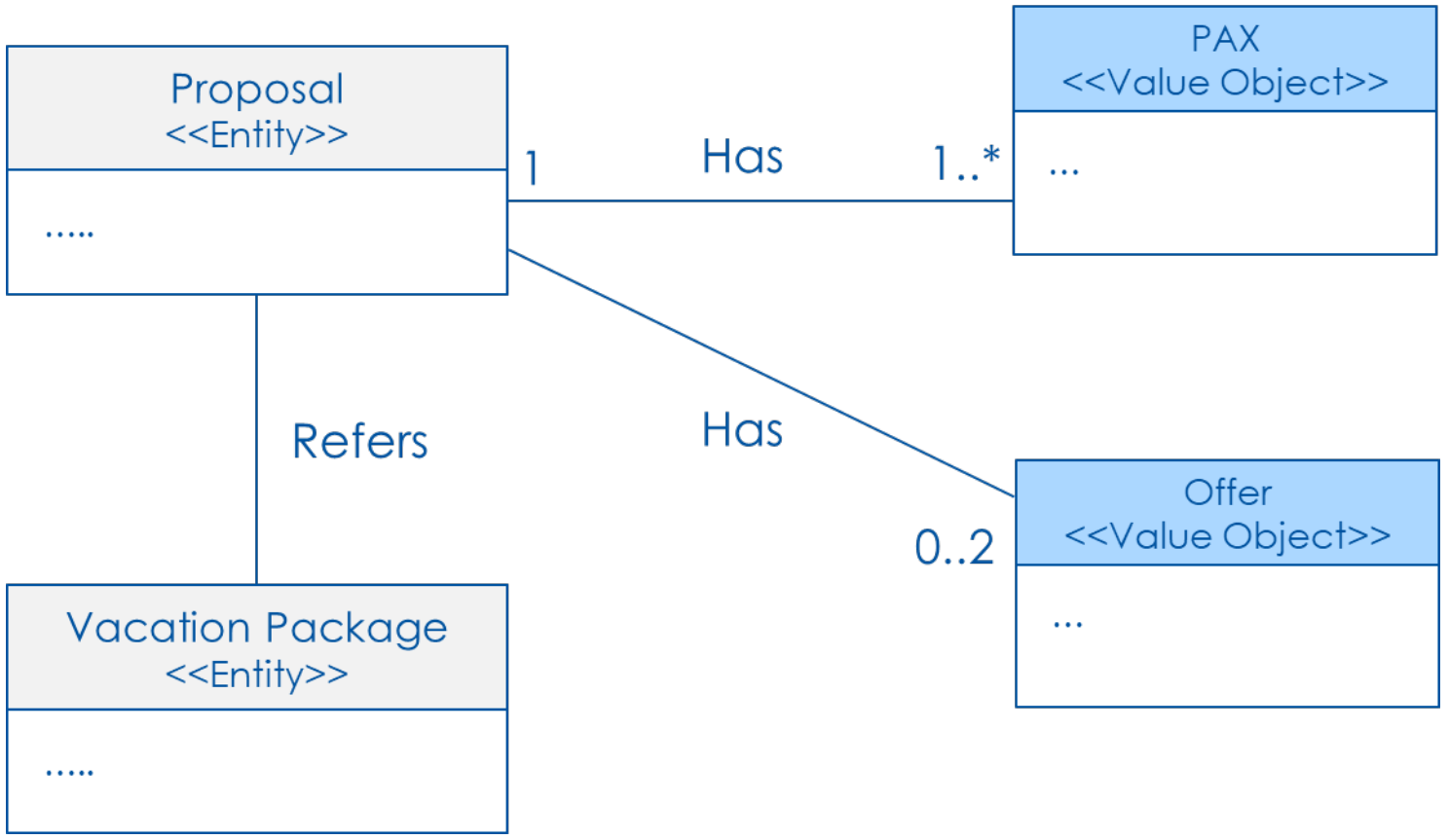
Proposal

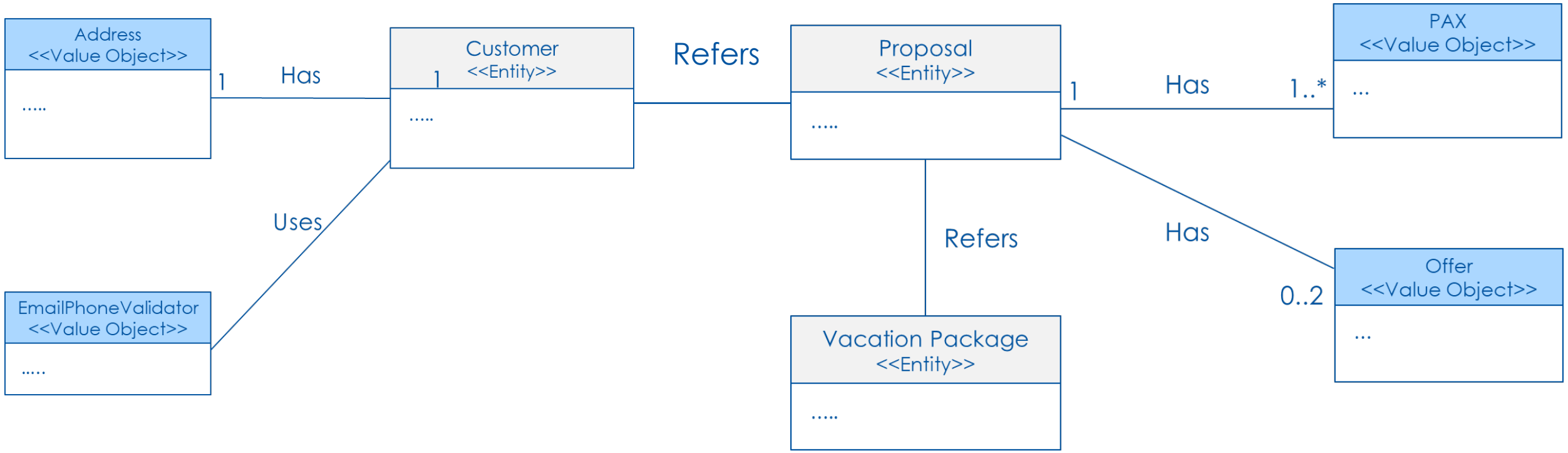
This is created for specific vacation package | pricing



Proposal

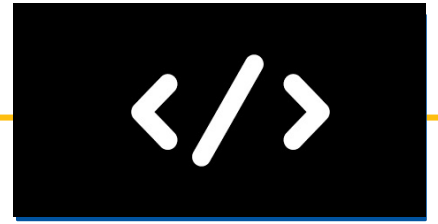
This is created for specific vacation package | pricing





ACME Model in UML & JAVA

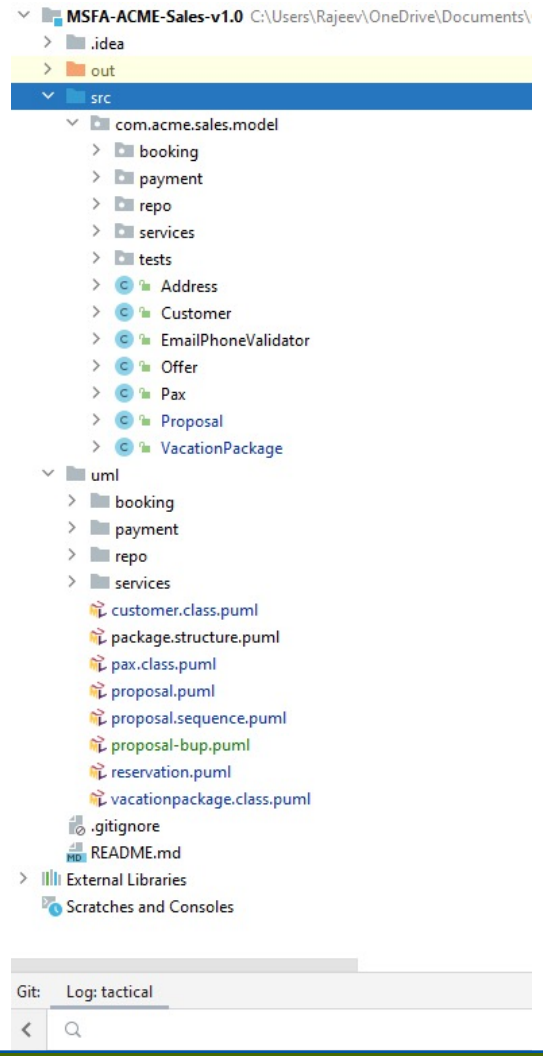
Build the ACME Sales Entities and Value Objects in UML & JAVA



1. Customer, VacationPackage, Proposal
2. Pax, Address, EmailPhoneValidator
3. Relationships

Objective

Model the core objects in the ACME "Sales domain"



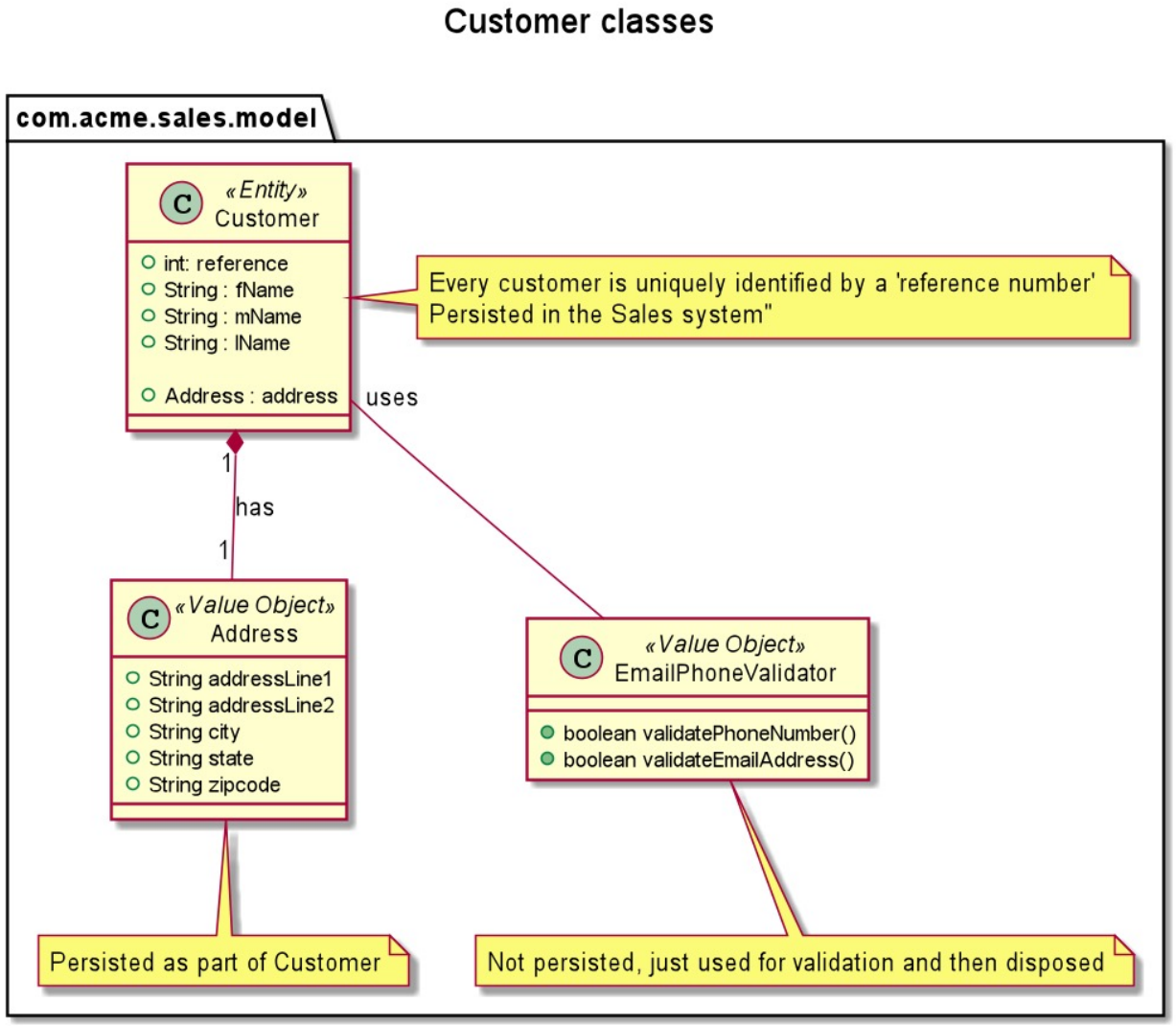
Code & Model files are available in the git branch tactical

Self walk through of the code (*optional*)

Note: Model kept simple so that it is easy to follow

Think of this model as the "First Draft"

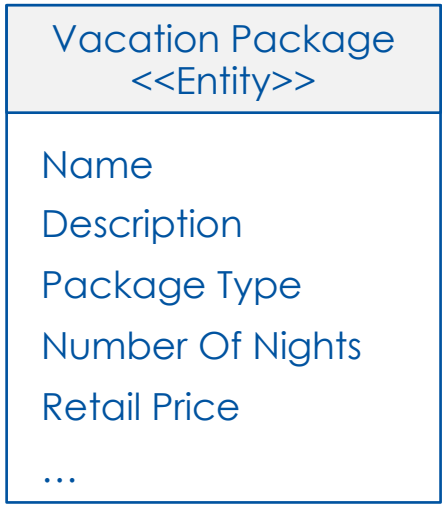
Model Walkthrough



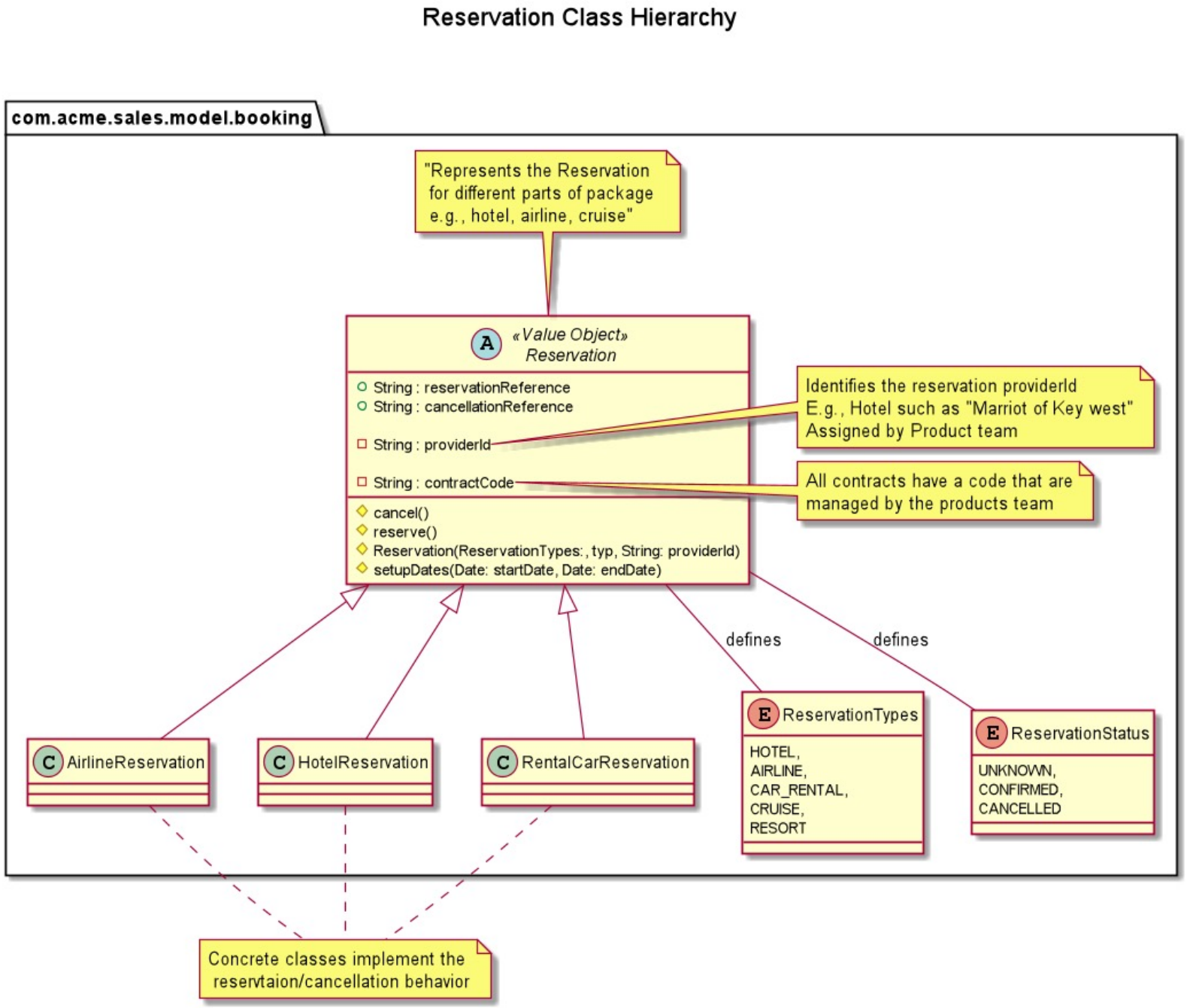
Vacation Package

This is the product that Acme sells

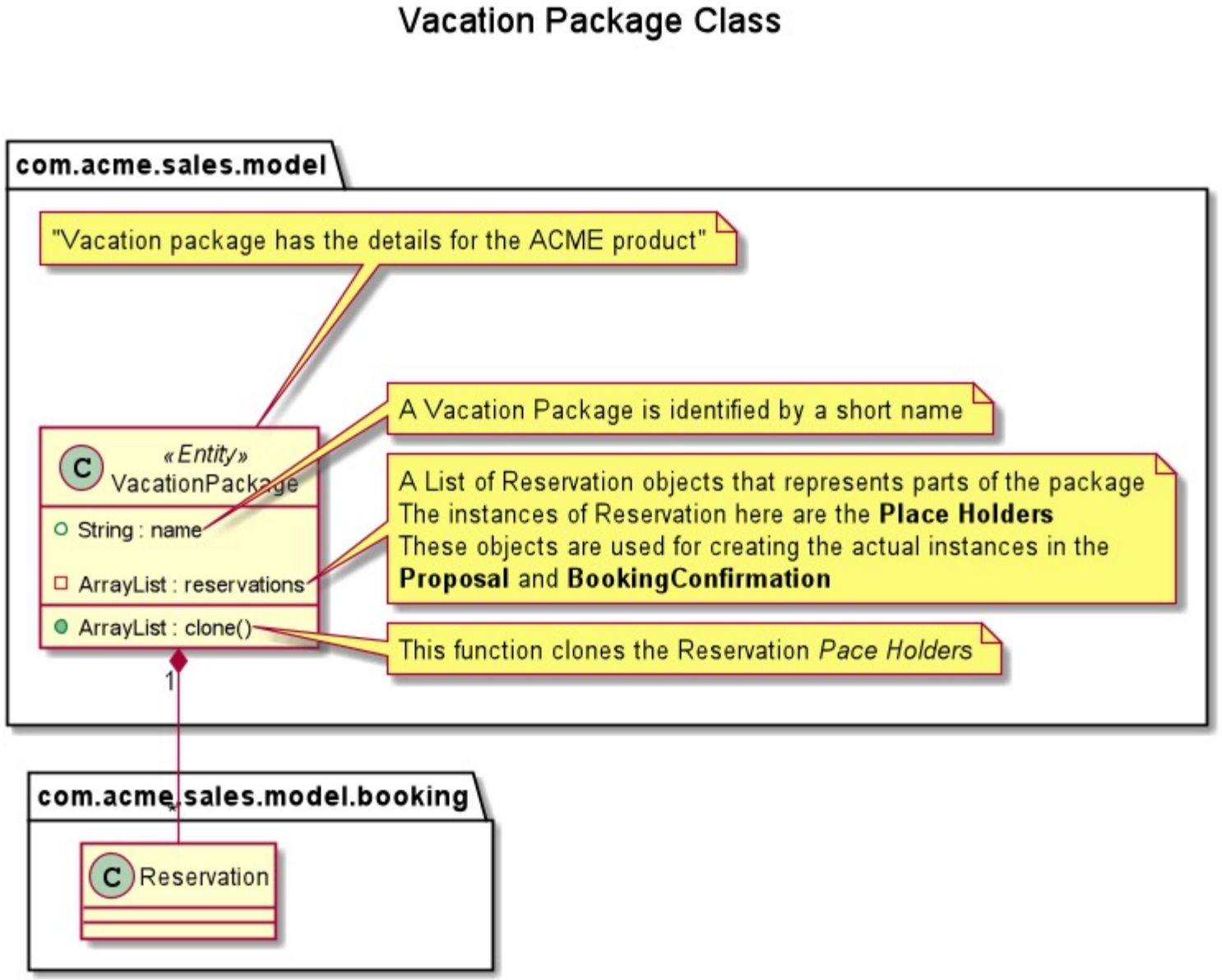
- Uniquely Identified by a name e.g., LV3NIGHT, BAHA5NCRUISE
- Package have a type
  - Resort
  - Hotel & Air ticket
  - Cruise
  - Hotel, Air ticket & Car rental
- Package have a fixed number of nights
- Package has a retail price



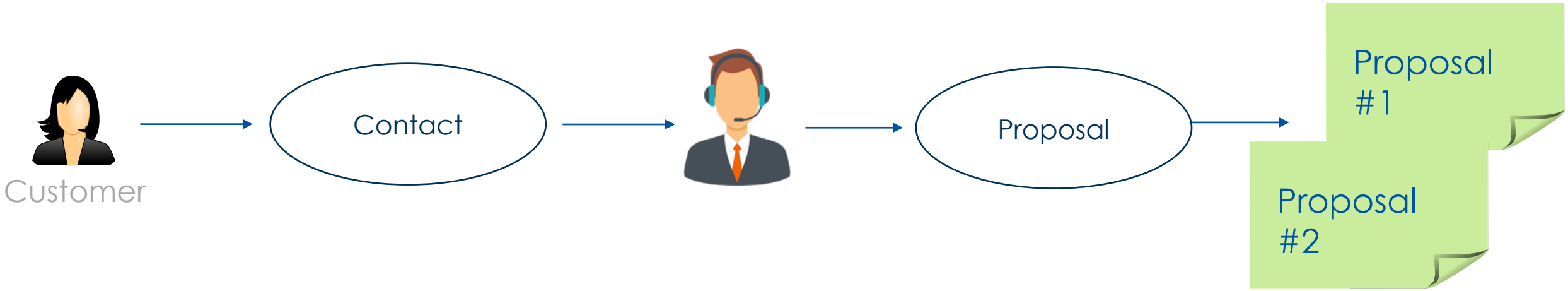
Entities



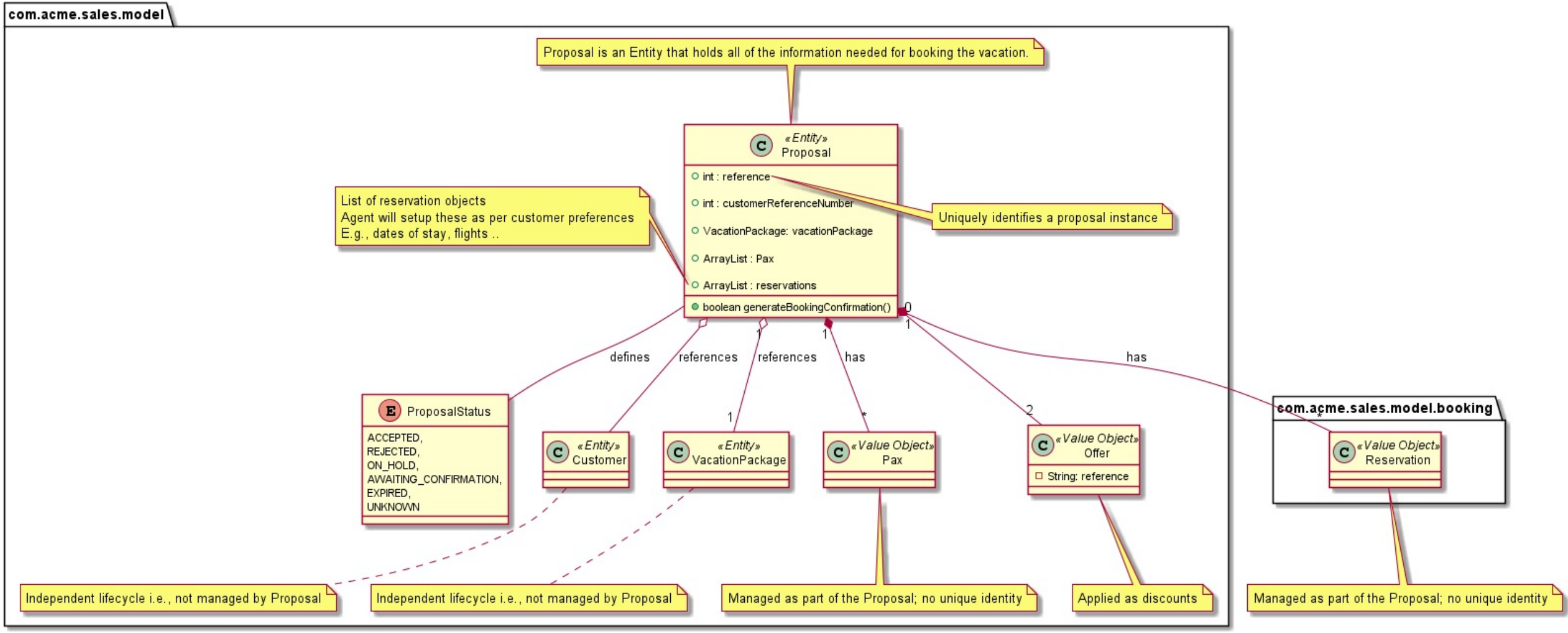
Vacation Package



Sales Use Cases



Proposal Class



Aggregates & Factories

Aggregate and Factory DDD tactical patterns

What is an aggregate?

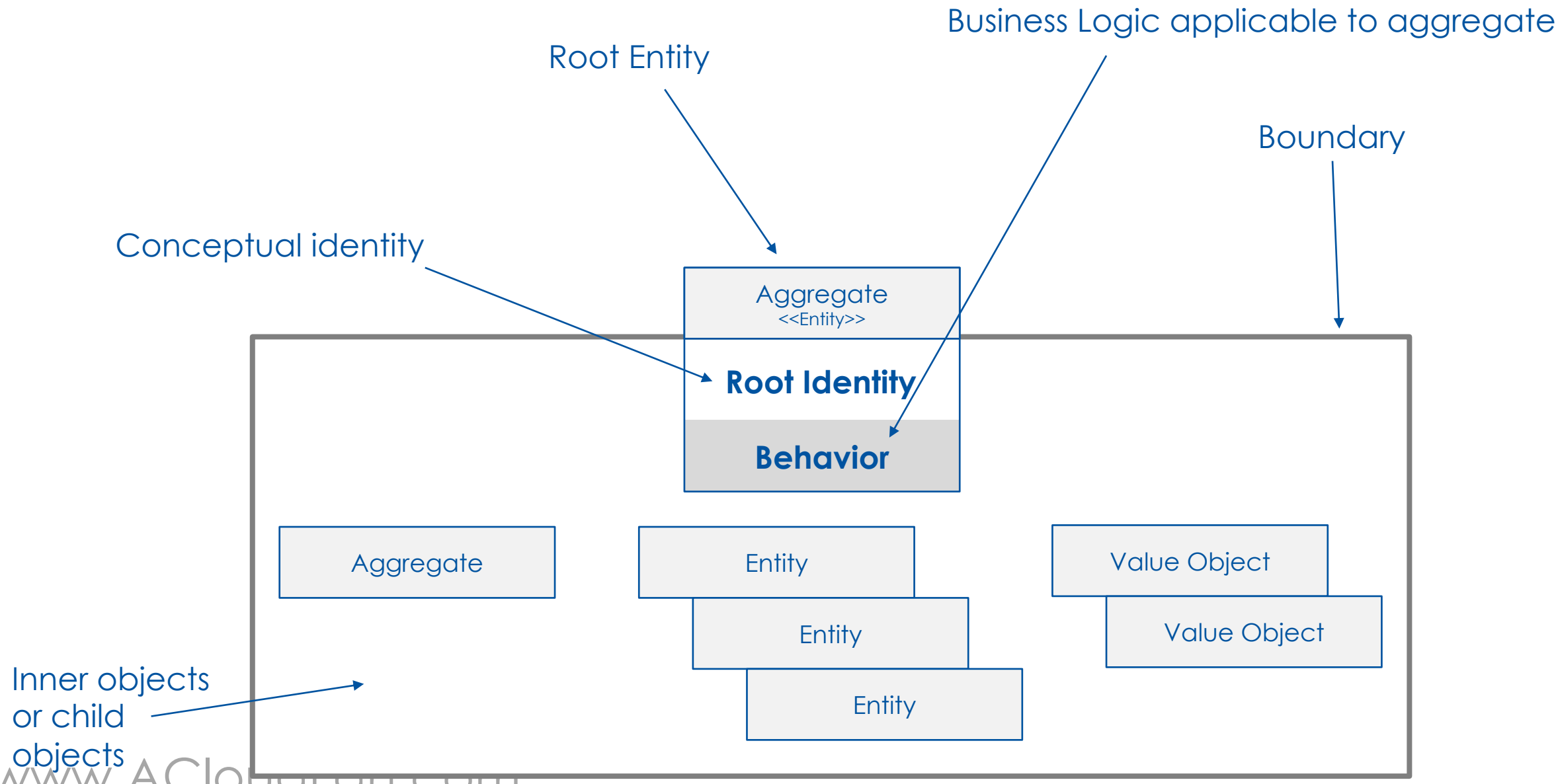
Parts of an Aggregate

Factory pattern

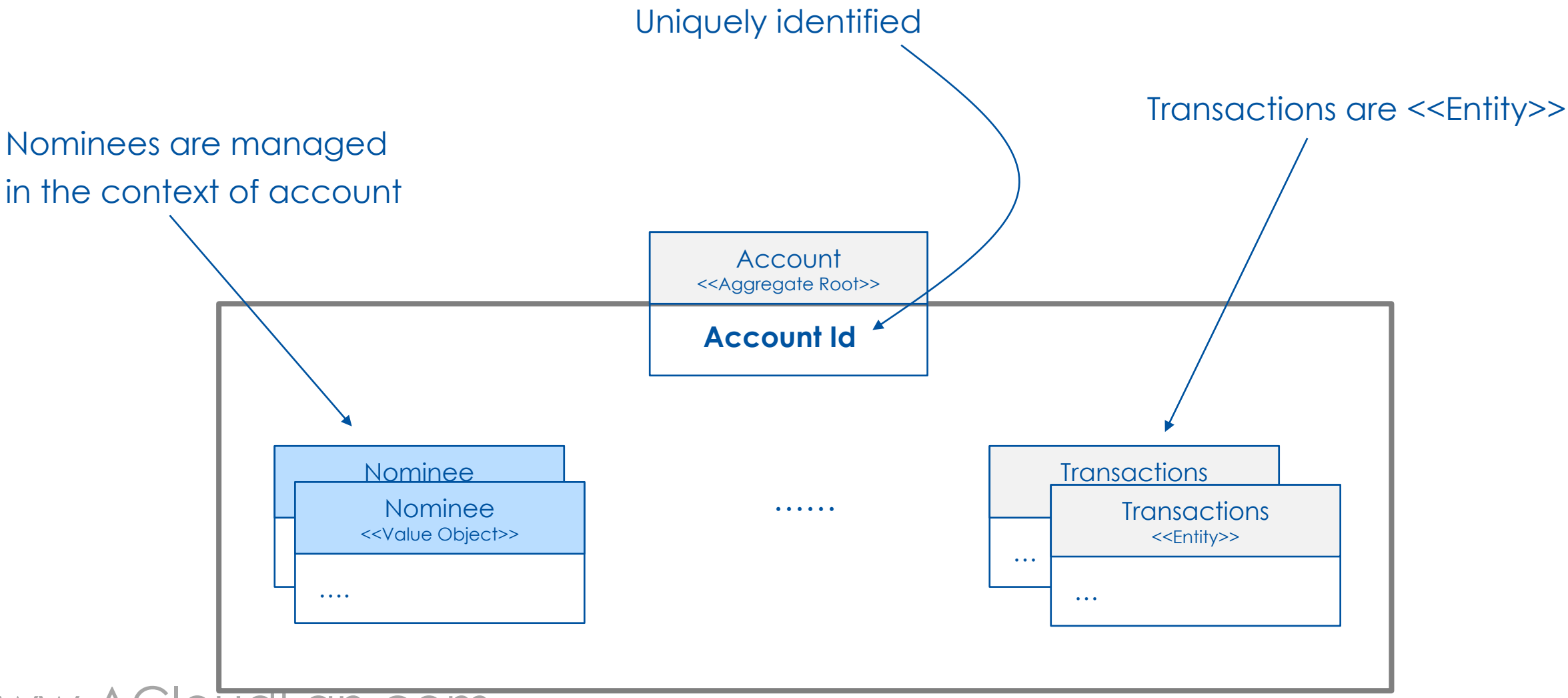
Aggregate

An Aggregate object is a cluster of entities and value objects that are viewed as a unified whole from the domain concepts and data perspective

Aggregate



Aggregate (Bank Account)



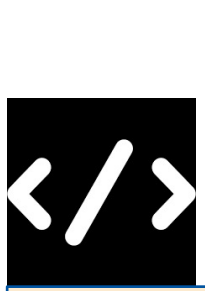
It Is an Entity

Also referred to as an Aggregate | Aggregate Root

Aggregate with 0 inner objects (children)?

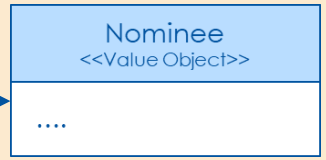
Procedural Programming

Operating on the child objects

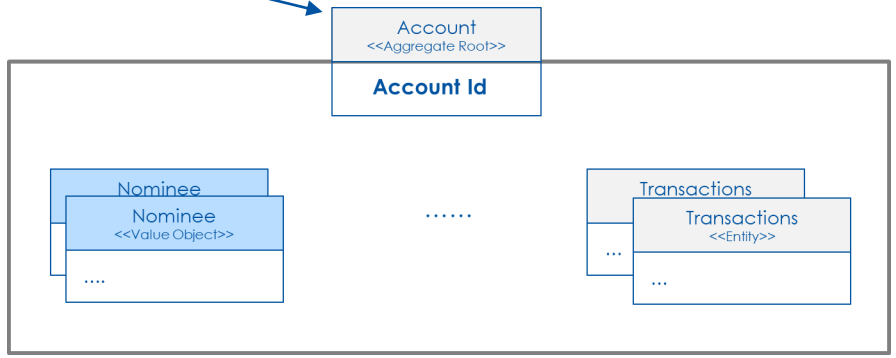


- Get Nominee from aggregate
  OR Access it directly from storage
- Operate **directly** on the object

setNomineePct()



getNominee()

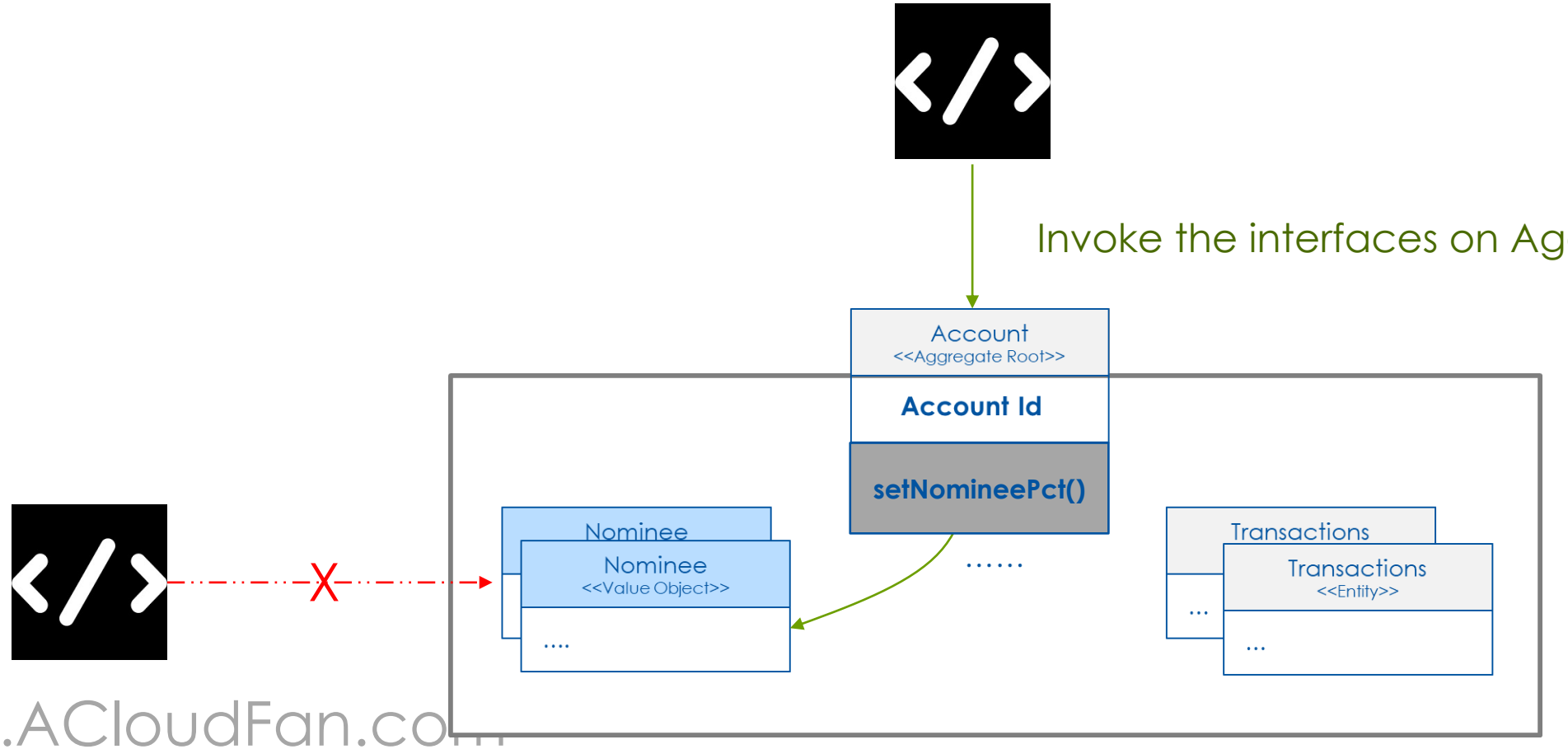


This is Procedural Programming !!!

DO NOT do this ☺

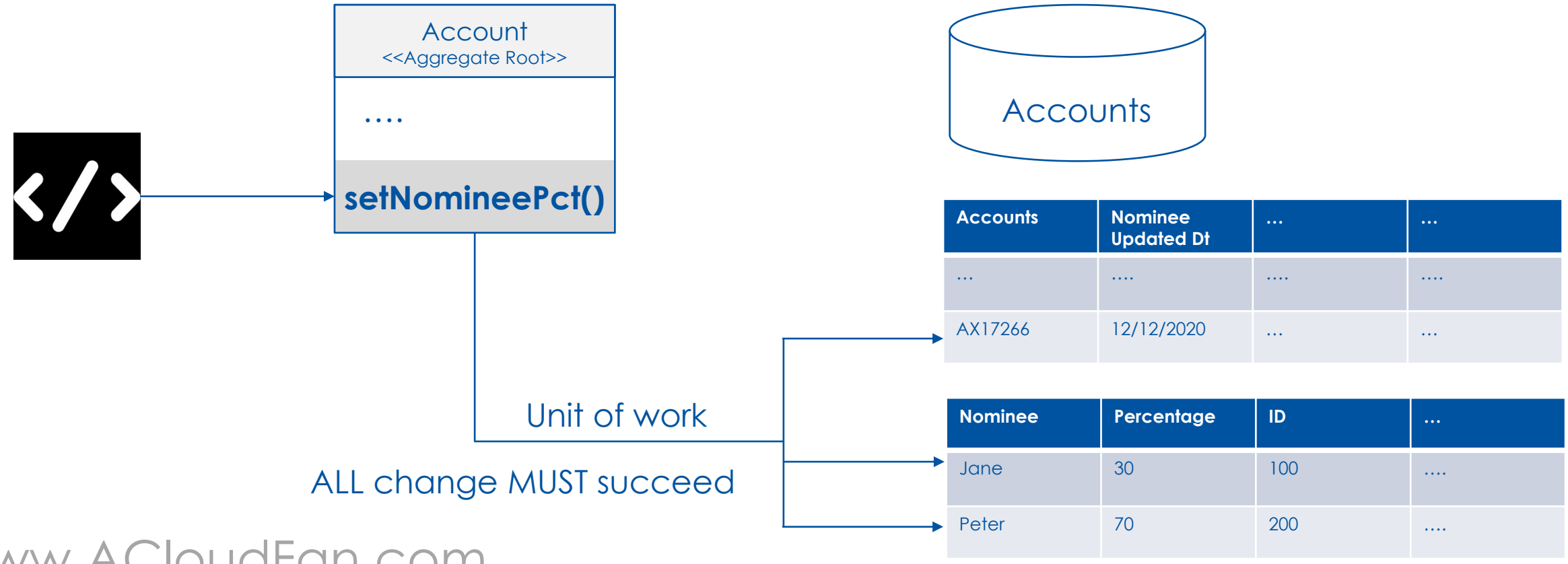
Aggregate Access

Aggregate provides interfaces for operations on Inner objects



Atomic Persistence

The aggregate is inserted | updated atomically

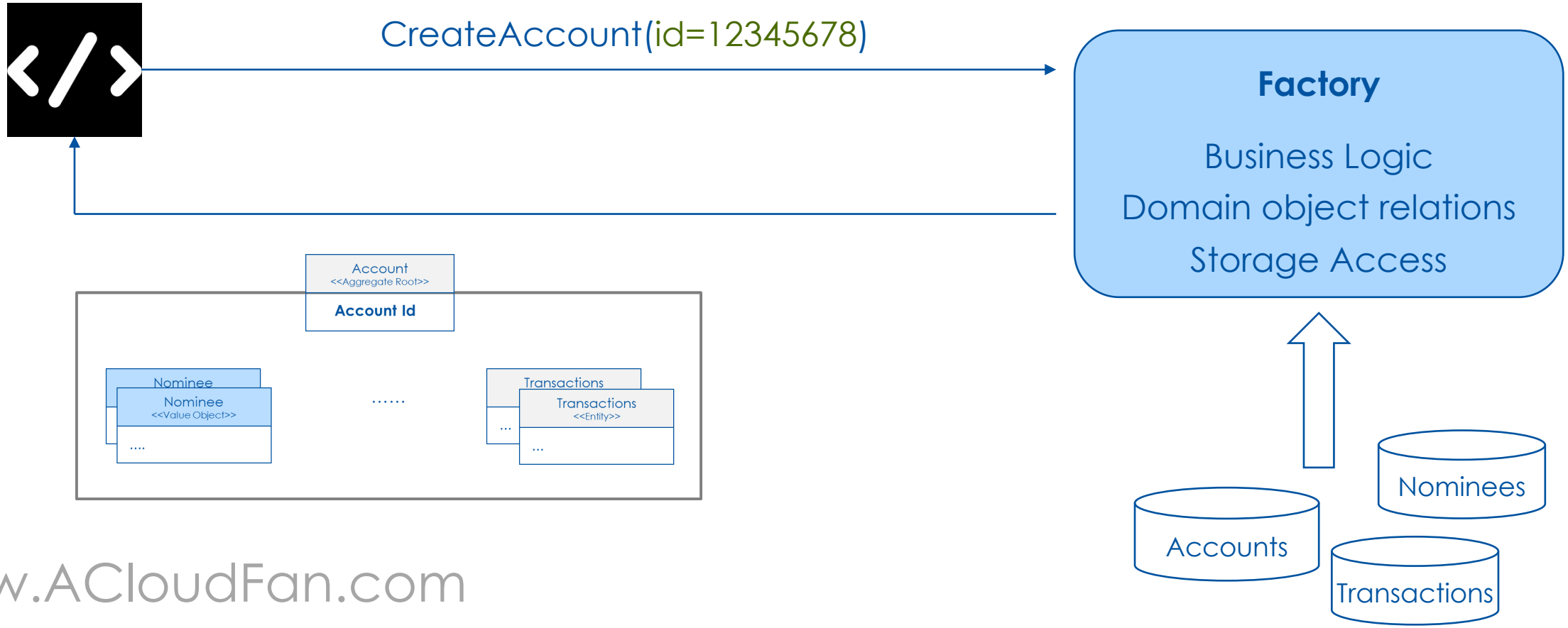


Accounts	Nominee Updated Dt	...	...
...	....	....	....
AX17266	12/12/2020	...	...

Nominee	Percentage	ID	...
Jane	30	100	....
Peter	70	200	....

Factory Pattern

A component for building complex domain aggregates



Quick Review

An Aggregate may contain Aggregates | Entities | Value Objects

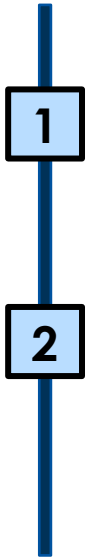
Aggregate MUST encapsulate behavior to manage inner objects

All changes to Aggregates are saved atomically

Factory pattern is used for creating complex Aggregates

Exercise : ACME Sales Use Case

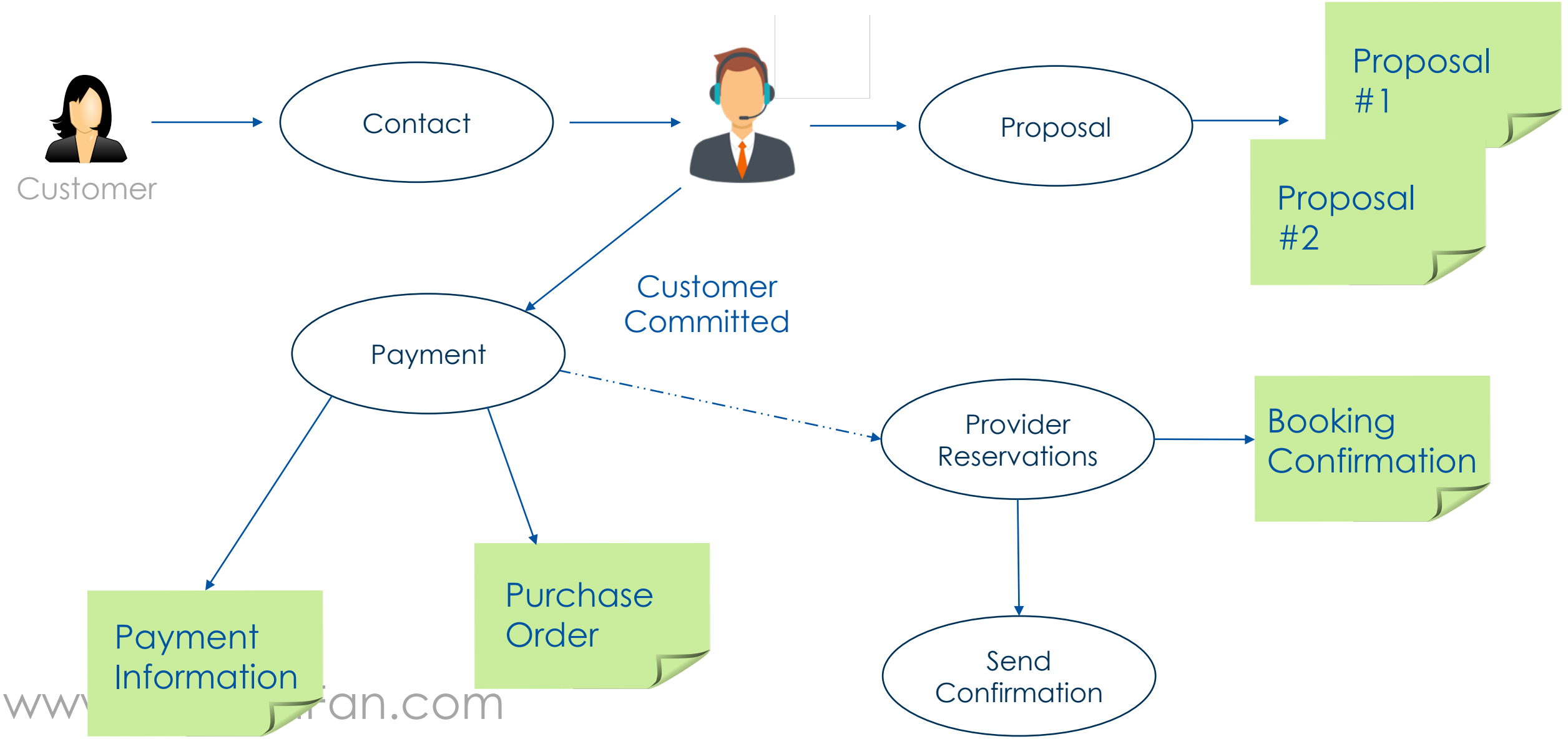
Identify Aggregates in the model



Sales use case

Booking Confirmation Aggregate

Sales Use Cases



Booking Confirmation

Contains all information about the reservation

Payment Confirmation

Pax

Reservation

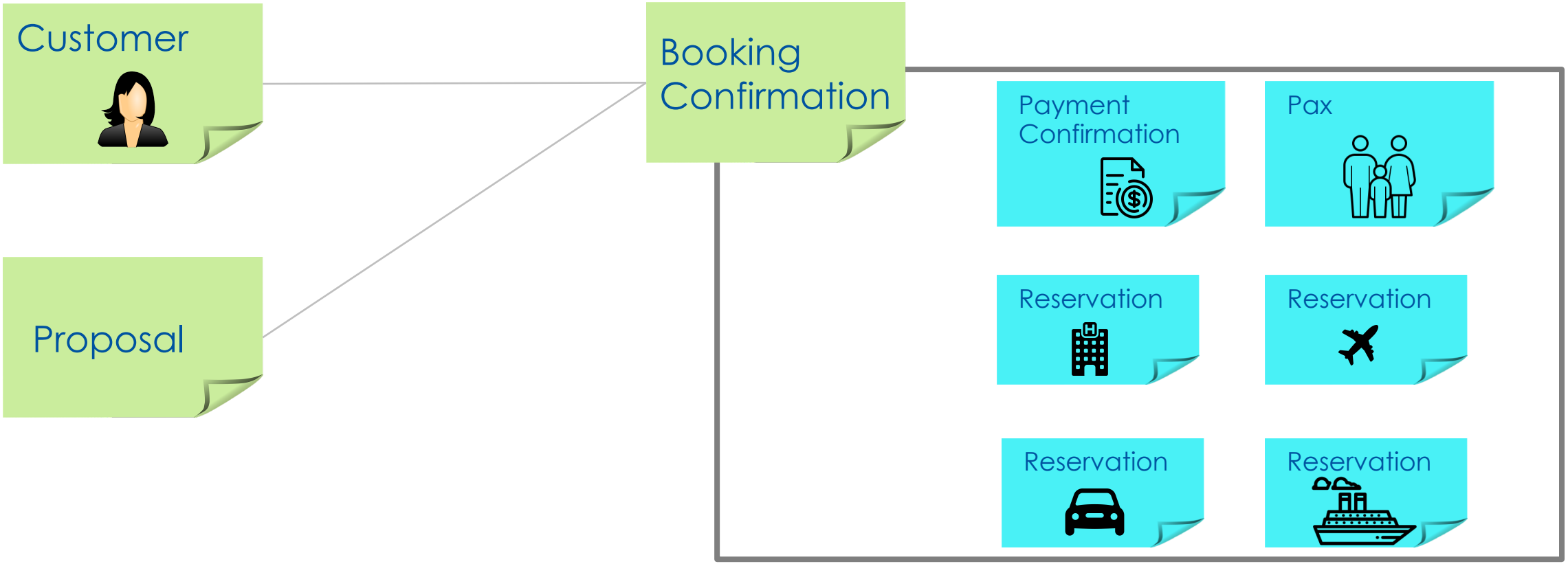
Reservation

Reservation

Reservation

.....

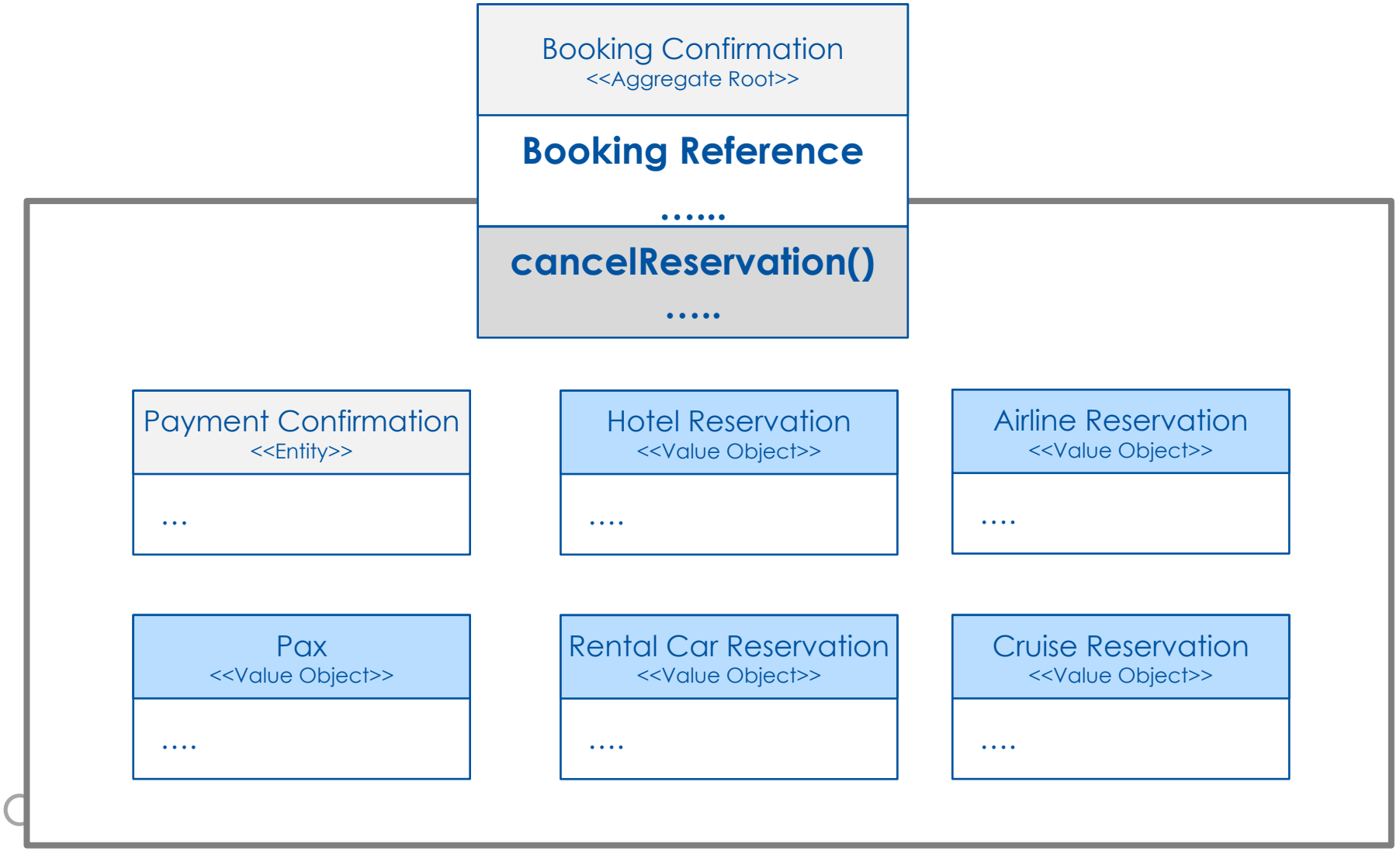
Booking Confirmation : Aggregate



Inner objects meaningful ONLY in the context of "Booking Confirmation"

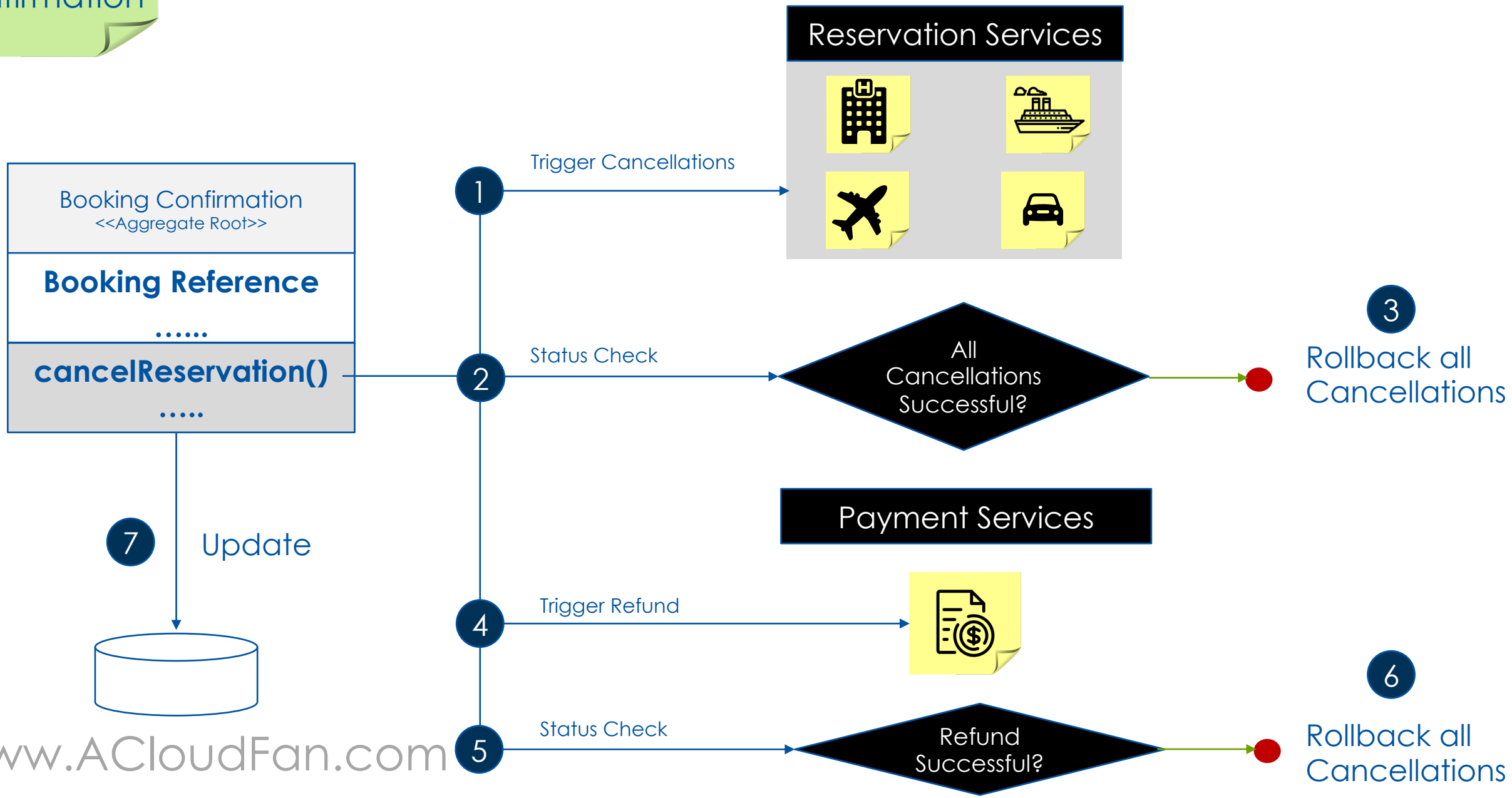
Booking Confirmation

Changes made only via exposed aggregate functions



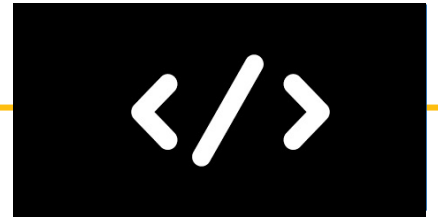
Booking Confirmation

Changes are atomic



ACME Model Aggregate

UML design for ACME Aggregate



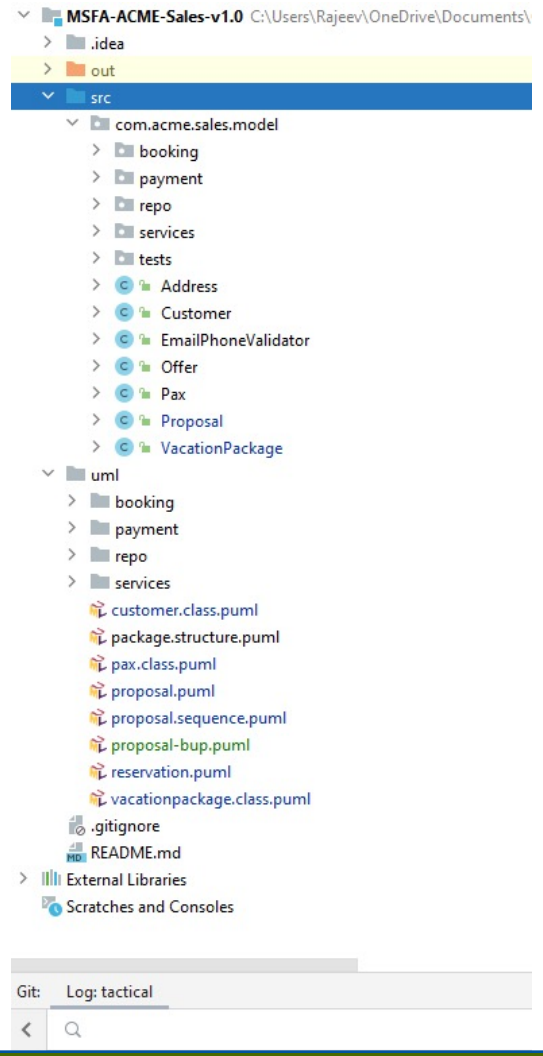
PaymentConfirmation <<Entity>>

BookingConfirmation <<Aggregate Root>>

BookingConfirmation state diagram

Objective

Model the core objects in the ACME "Sales domain"



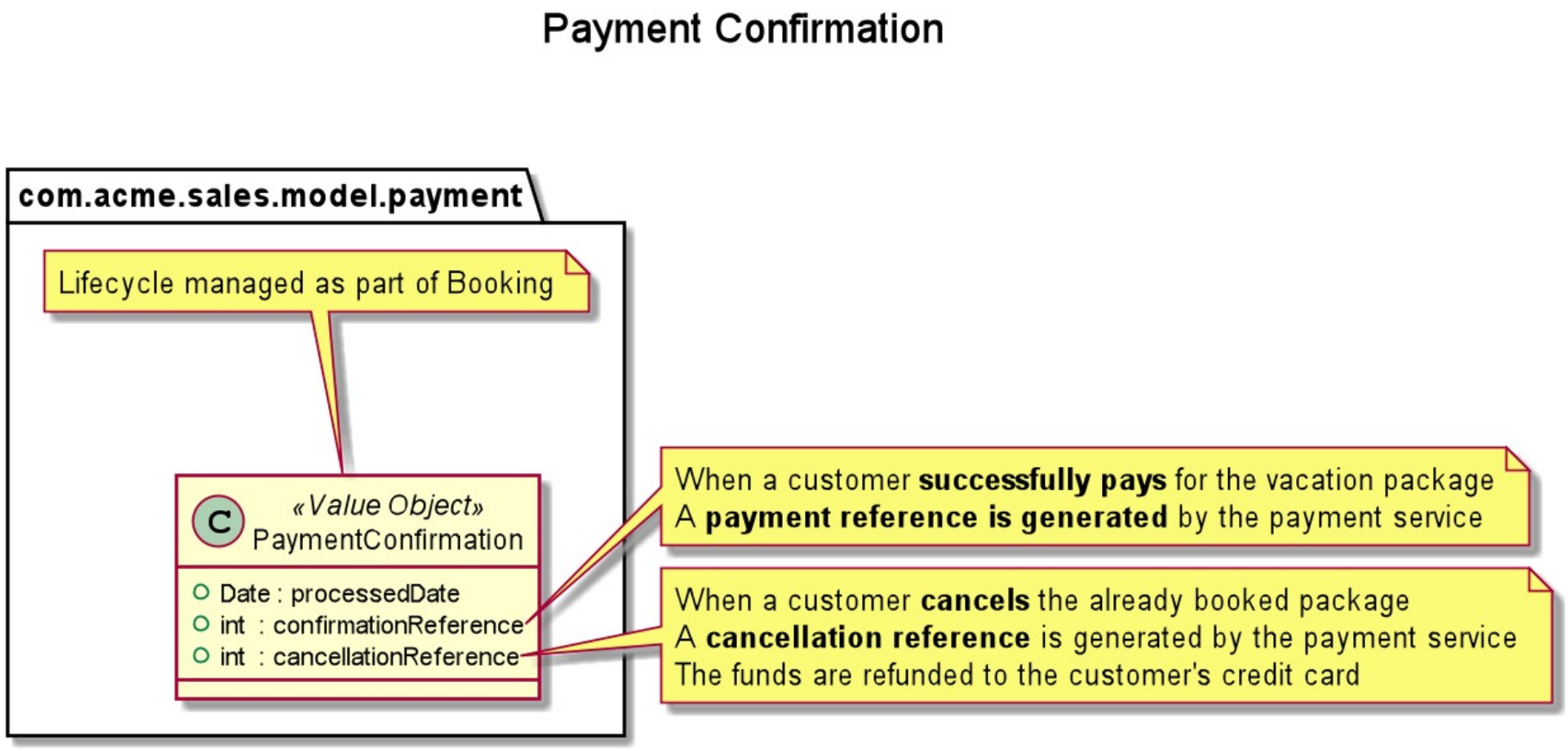
Code & Model files are available in the git branch tactical

Self walk through of the code (*optional*)

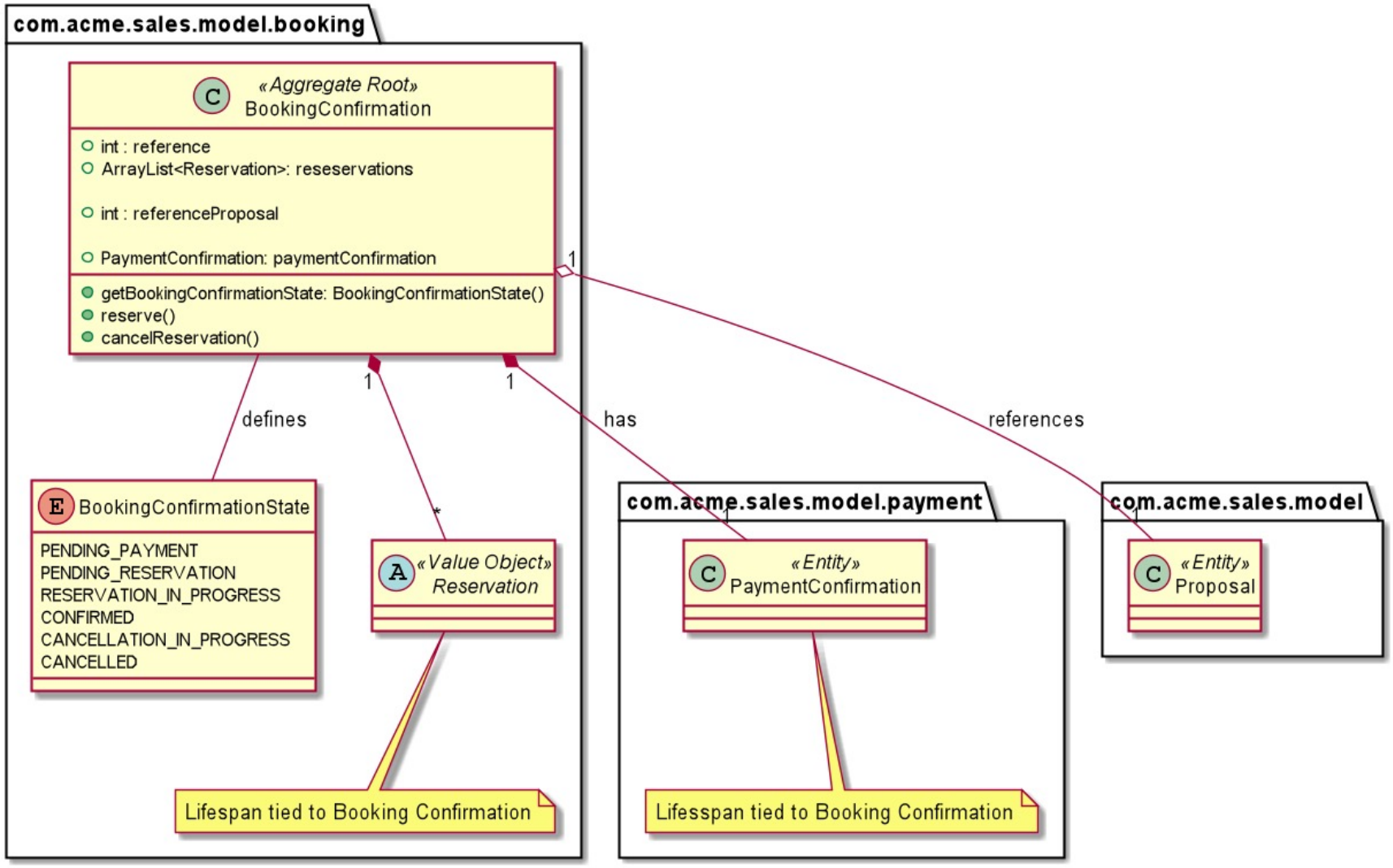
Note: Model kept simple so that it is easy to follow

Think of this model as the "First Draft"

Payment Confirmation



Booking Confirmation Class



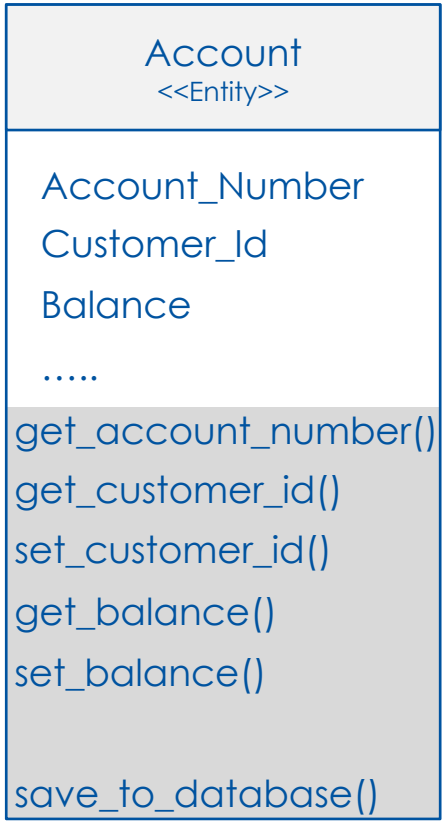
Model Behavior

Common terms used for describing the domain models

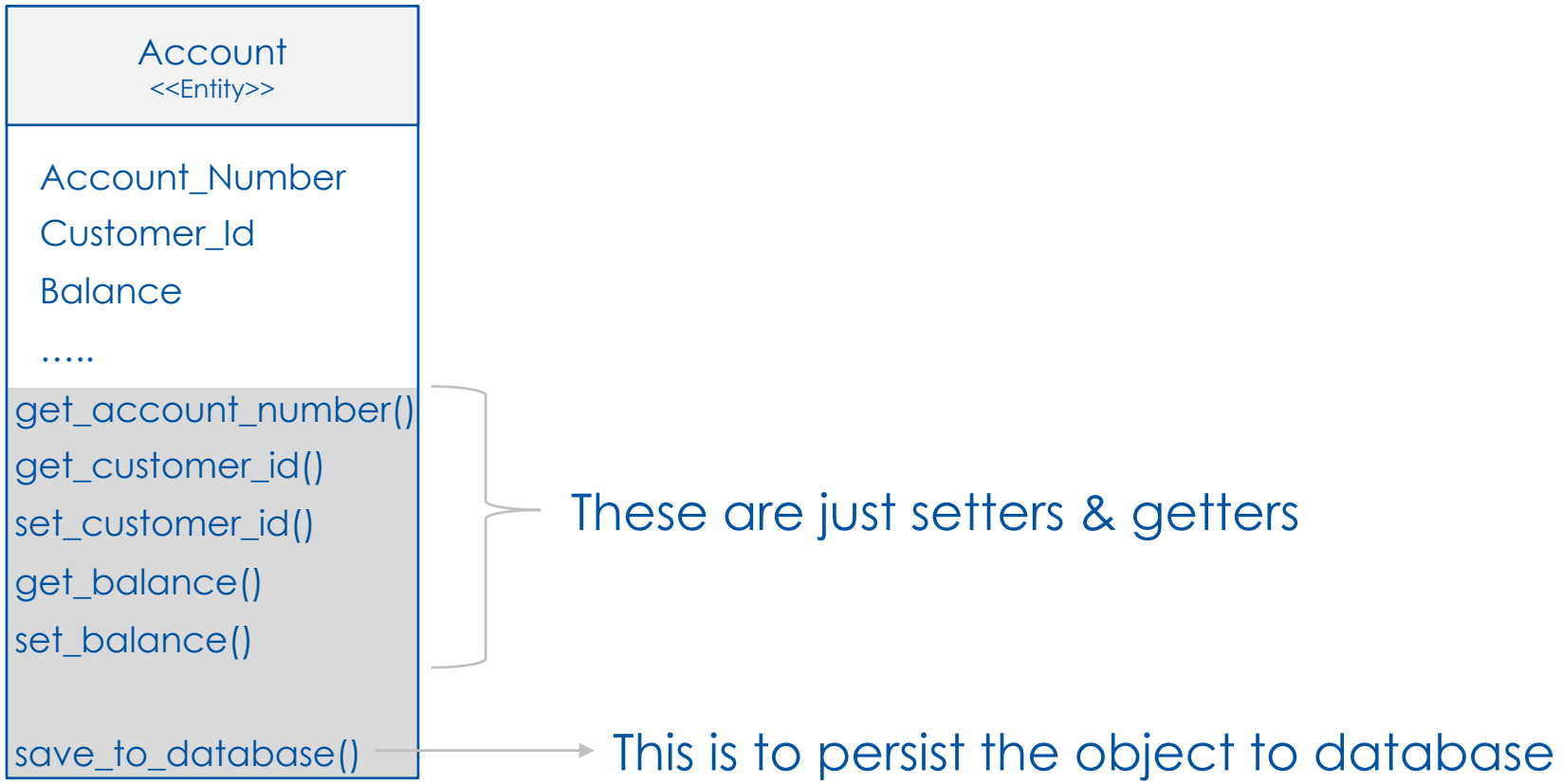
Anemic Models

Symptoms of an anemic model

Rich Models



Does this Entity expose business logic?



No - It does not expose any Business Logic !!

Anemic Model

> A domain model composed of entities that do not exhibit behavior i.e., operations applicable to the domain concepts are missing

Rich Domain Model is opposite of Anemic Domain Model

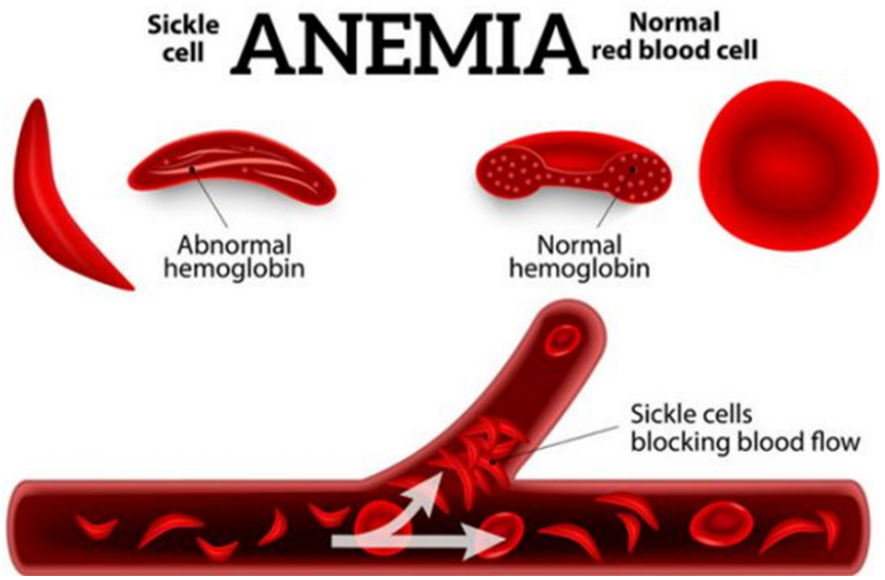
Martin Fowler coined the term *Anemic Models*

What is Anemia?

Anemia is a condition where the body does not have enough red blood cells to carry enough oxygen to various tissues and organs.

Symptoms:

- Fatigue
- Weakness
- Pale skin
- Cold hands and feet.
- Irregular heartbeats
- Shortness of breath
- Dizziness
- Chest pains



Anemic = Person suffering from Anemia

Analogy used for Models that may lead to unmanageable & complex code

Symptoms of Anemic models

#1 Entities lack the behavior

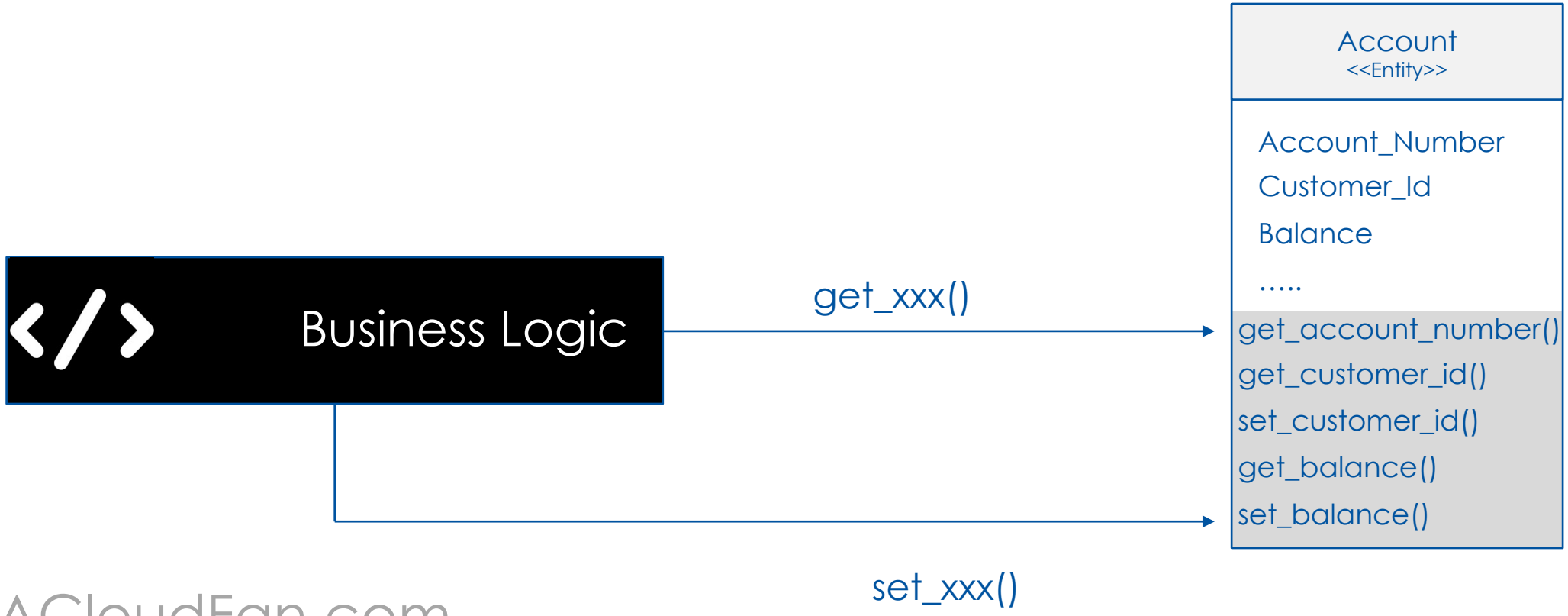
#2 Entity exposes functions ONLY for CRUD operations

#3 Business Logic is implemented outside the Domain Objects

Best Practice is to always go for a "Rich Domain Model"

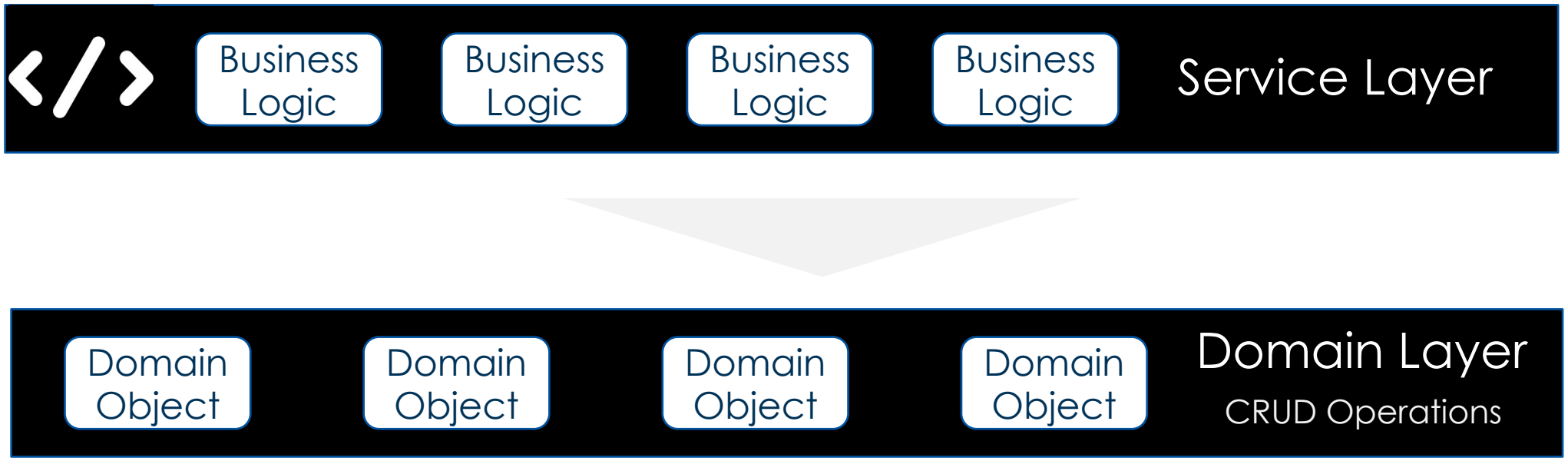
Anemic Models Symptoms

#3 Business Logic is implemented outside the Domain Objects



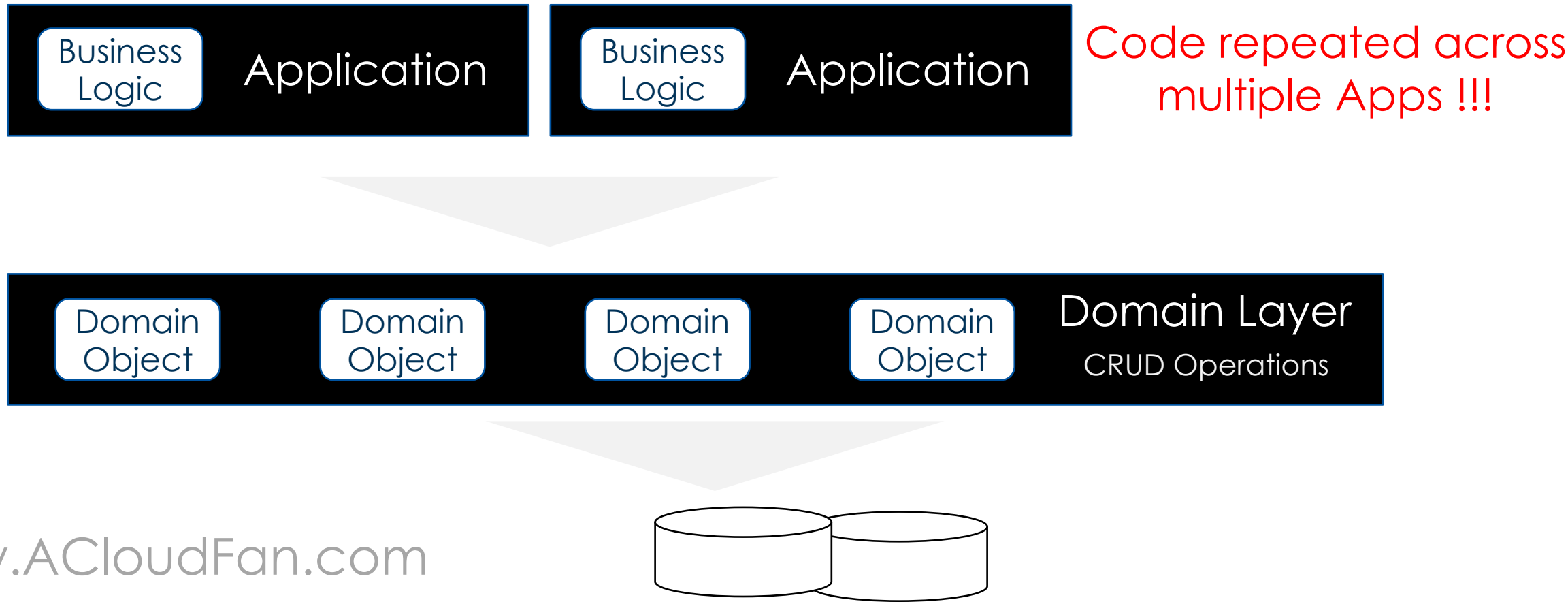
Anemic Models Symptoms

#3 Business Logic in shared services



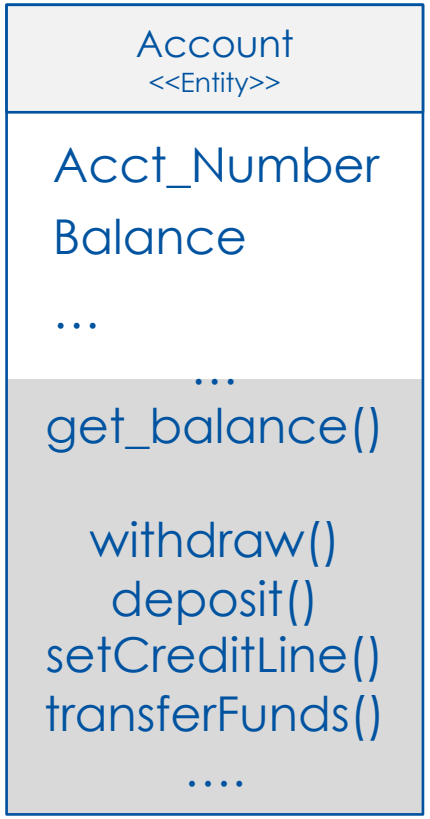
Anemic Models Symptoms

#3 Business Logic embedded in Apps as procedural code



Rich Model implements the Model Behavior

Business Logic is implemented as inherent part of the entity



- Business Logic is one place
- Data Integrity is maintained

Implements the domain concepts
i.e., Withdrawal from account

Is an Anemic Models always Bad ?

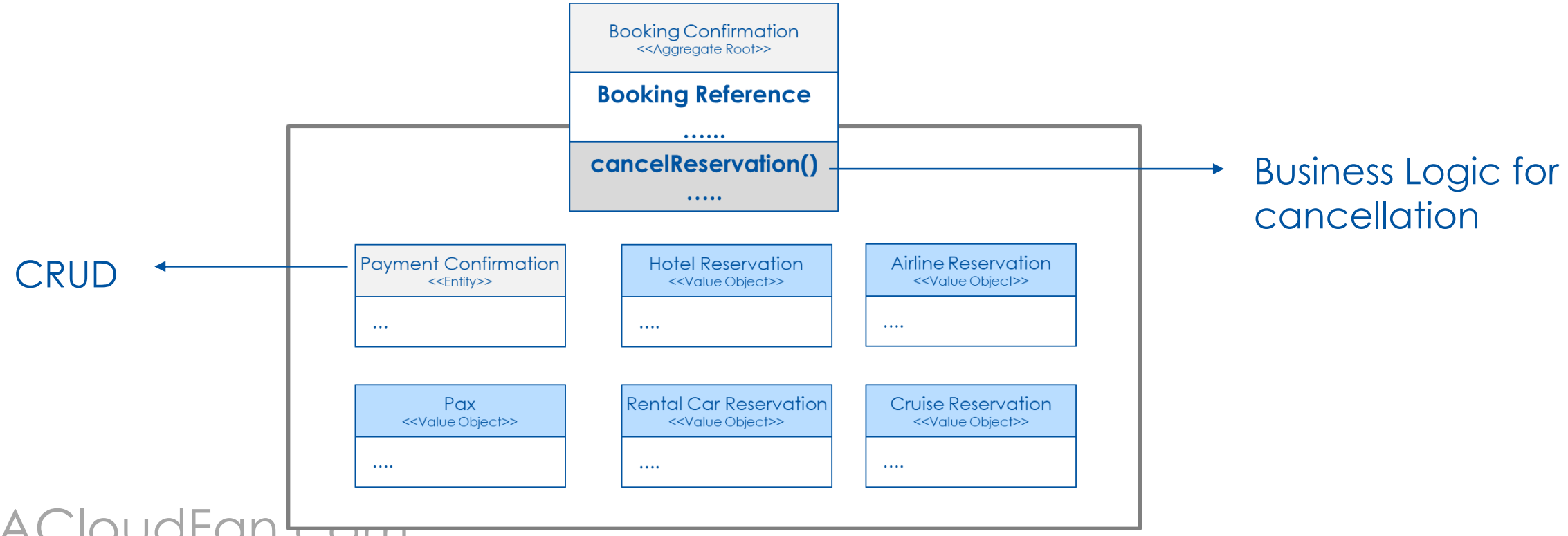
Not Really, especially in case of simple domain problems

- No or Light Business Logic; Infrequently changing business logic
- Generic Data Services e.g., CRUD API
- Shared logic that does not belong in a single model entity

"Anemic Models" may be Anti Pattern in some situations but NOT all !!!

Entity model with no behavior

An Aggregate Object implements the behavior



Considerations

- Consistency
- Don't Repeat Yourself (DRY)
- Maintainability
- Complexity

Quick Review

Anemic Model

Entities with no behavior

Symptoms of Anemic Model

- Entities with ONLY CRUD functions
- Business Logic in external components

Is Anemic Model always an anti-pattern?

- Depends on the use case | requirements | other factors

Raj's opinion

Repository Pattern

Domain Objects are persisted to databases using a Repository

What is a Repository?

Characteristics of a Repository

Realization options

Repository

A Repository object acts as a collection of Aggregate Objects in memory. It hides the storage level details needed for managing & querying the state of the Aggregate in the underlying data tier.

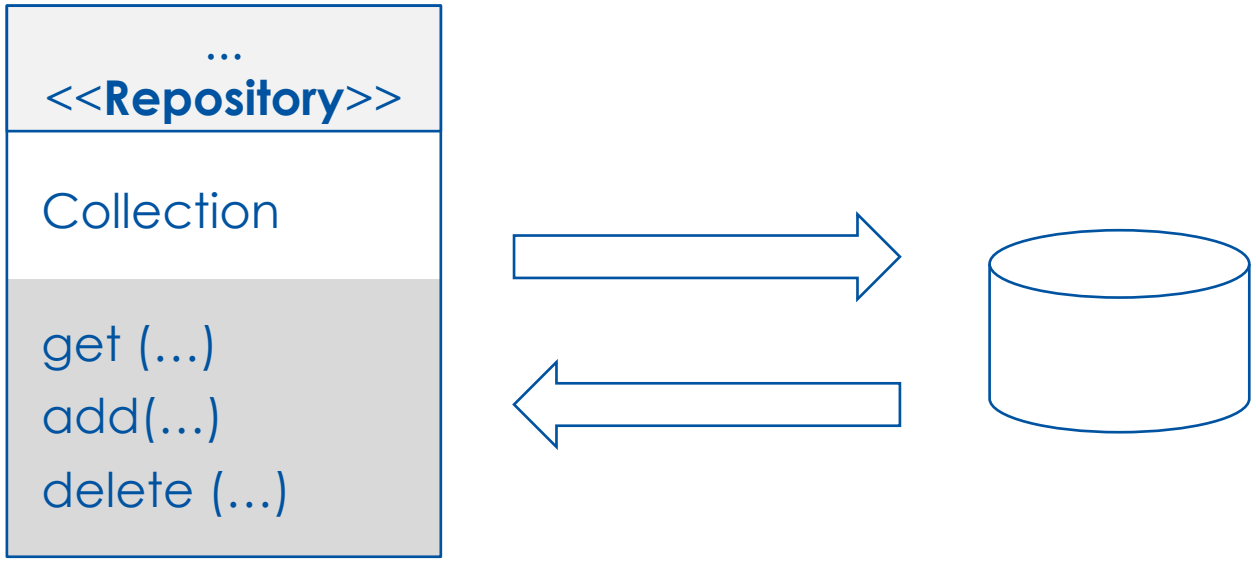
Repository

Manage | Query the state of aggregate

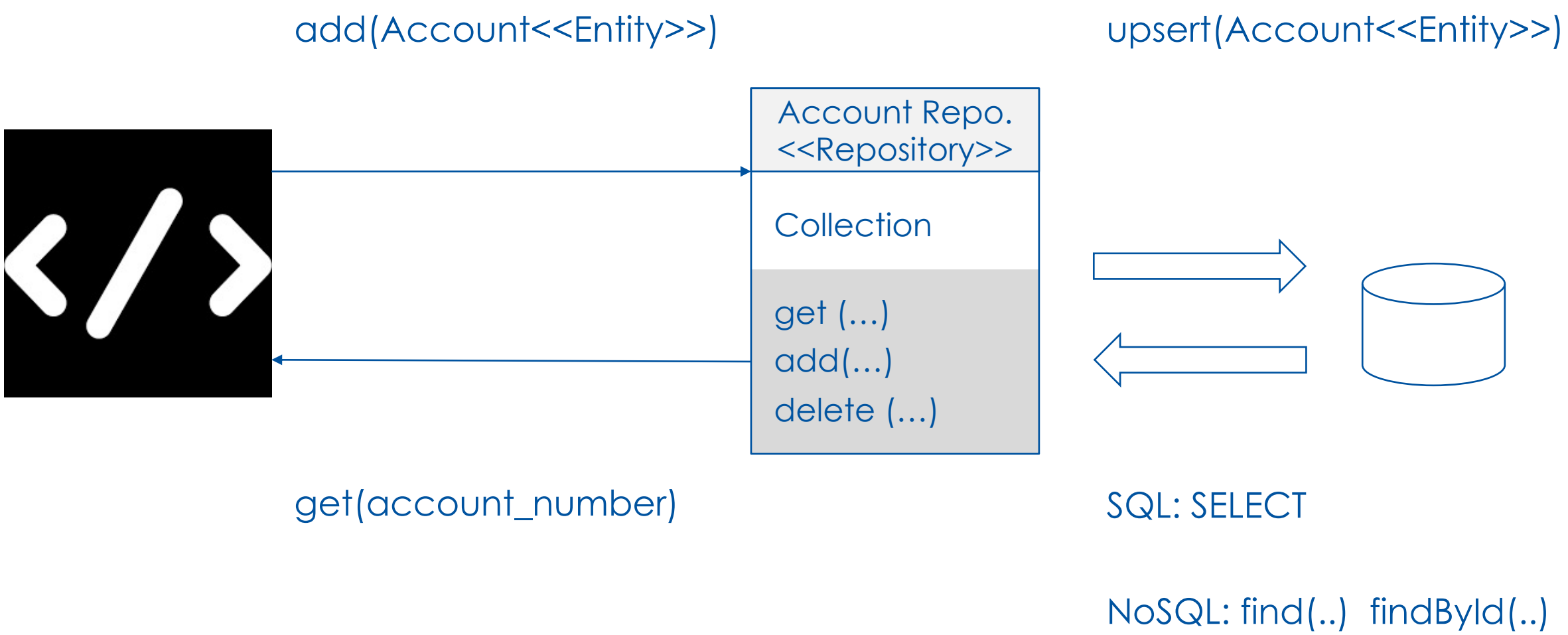
Create **U**pdate = add(<Aggregate>)

Retrieve = getByid(..)

Delete = removeById(…)



Example: Account Repository



Caller of repository has NO knowledge of Data tier

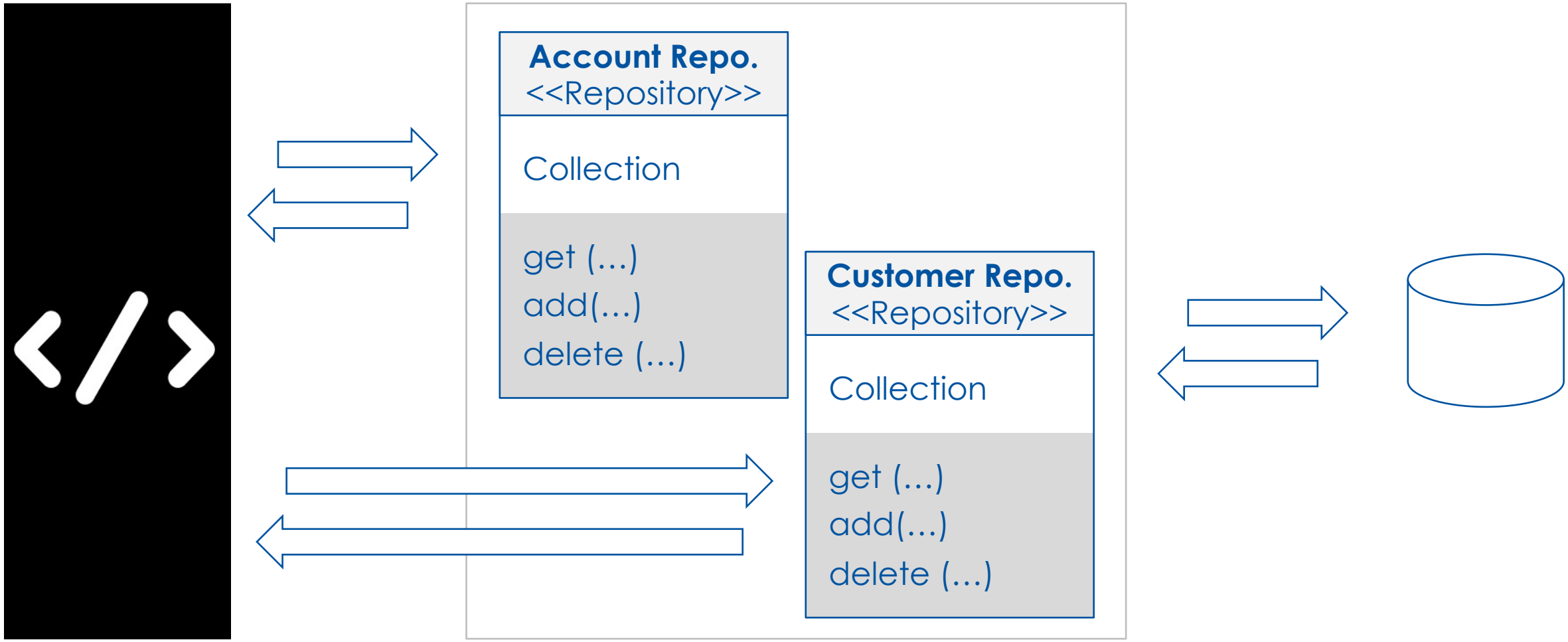
Characteristic of Repository

#1 Created on per Aggregate basis

#2 May expose higher level behavior | functions

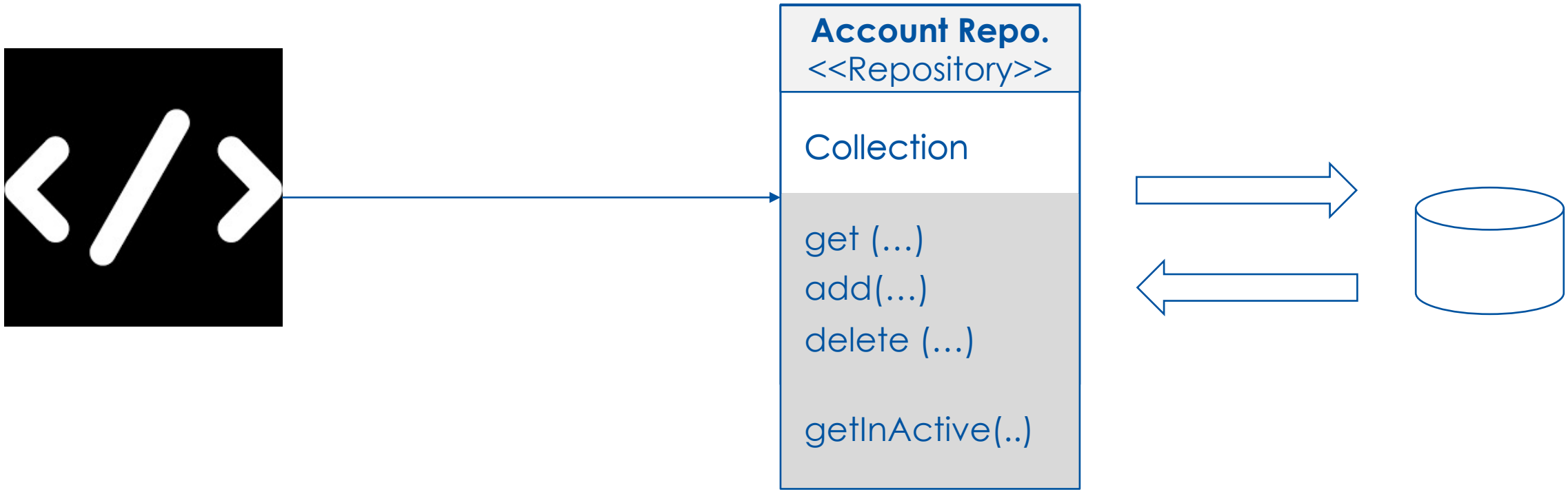
#3 Persistence operations are Atomic

#1 One Repository per Aggregate



Managed as part of the Domain Layer

#2 May expose Higher Level functions

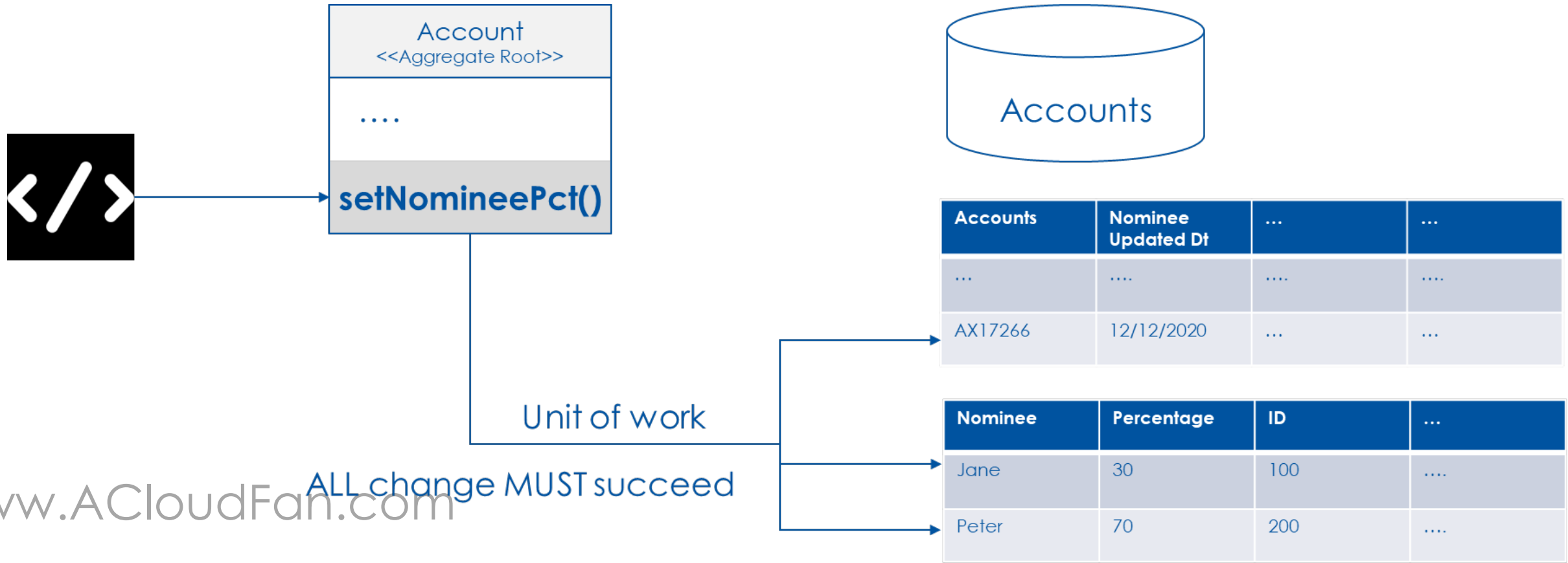


Get All Accounts that have not been active after the date: '12-31-99'

SELECT *
WHERE last_used < '12-31-99'

#3 Persistence operations are Atomic

The aggregate is inserted | updated | deleted atomically



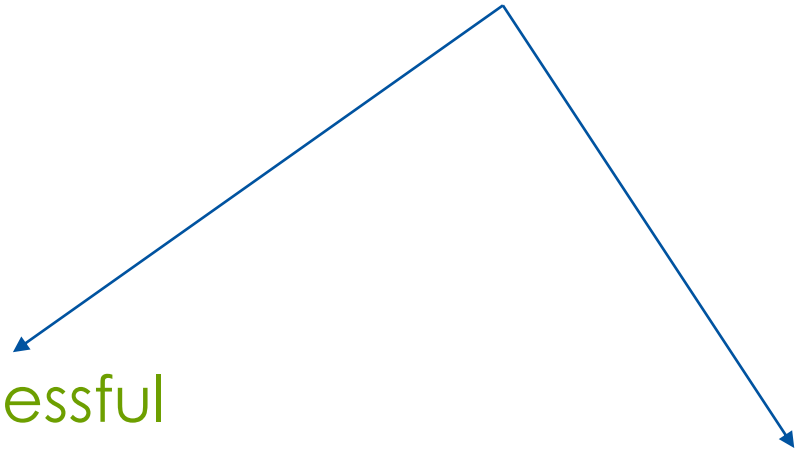
Accounts	Nominee Updated Dt	...	...
...	....	....	....
AX17266	12/12/2020	...	...

Nominee	Percentage	ID	...
Jane	30	100	....
Peter	70	200	....

#3 Persistence operations are Atomic

Repository carries out Insert | Update | Delete as a transaction

- All changes carried out under a unit of work



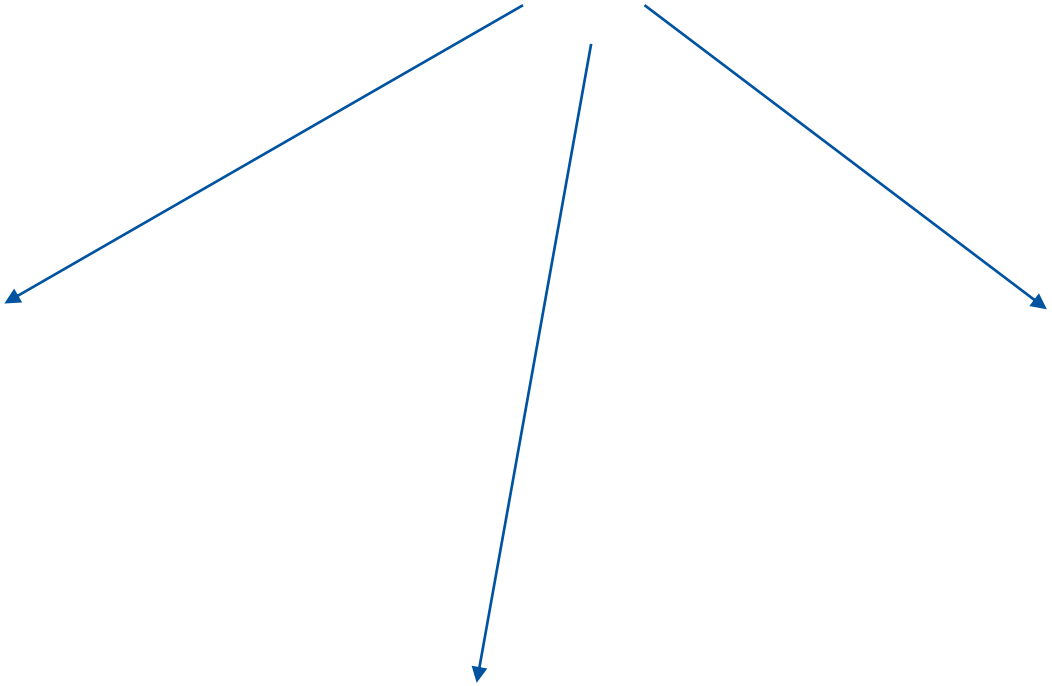
All changes to DB Successful

No Change to Any Entity in DB

In case of any DB op Failure

Repository Benefits

Keeps the Domain model independent of the storage layer



Storage Model

- Table structures
- Column names

Technology

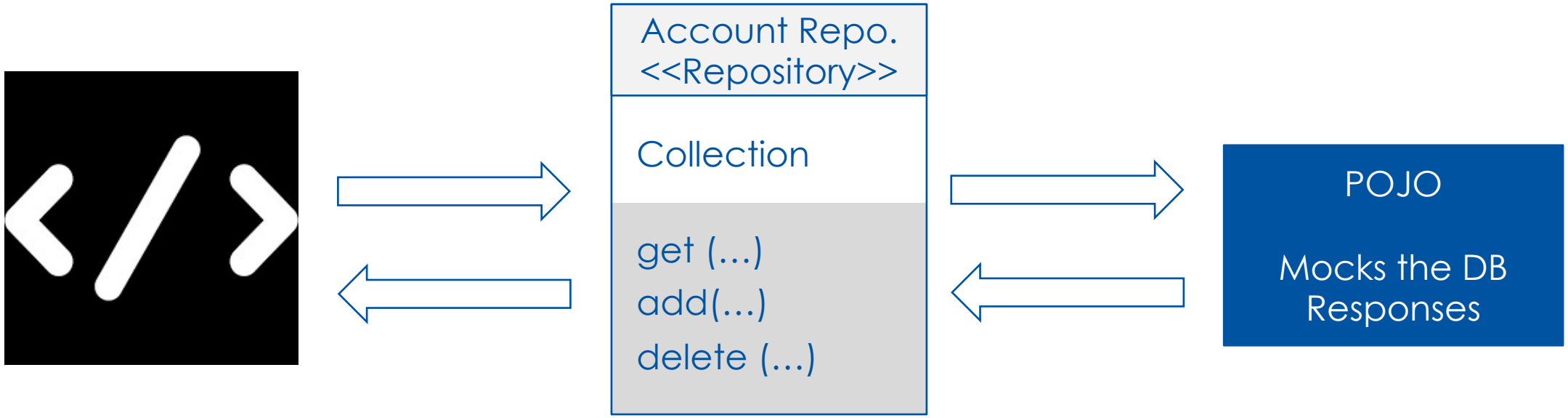
- RDBMS or NoSQL
- File System

Infrastructure

- JDBC or SDK
- Read Replica for queries

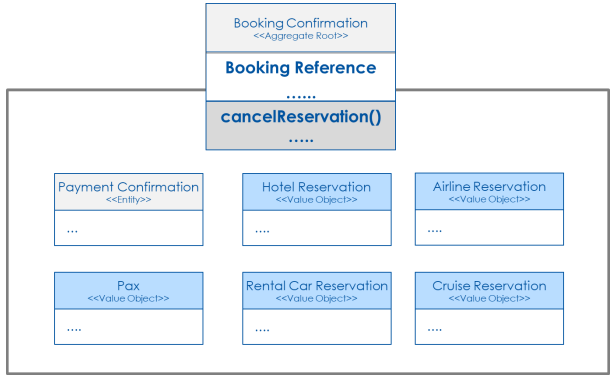
Repository Benefits

Unit testing and Mocking



Repository - Impact on performance

Large Aggregate may impact performance

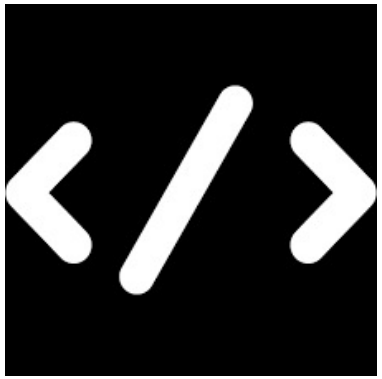


- Object creation may require multiple DB ops
- Joins across multiple tables

Leverage caching

Repository - Impact on performance

Impact on Criteria based Queries



- Partial result set needed in UI/UX app
- Data from multiple aggregates

OK to expose additional queries outside of the Repository

PS: CQRS pattern covered later

Repository - Managing the mapping complexity

Complex code is needed for mapping between domain & DB

Consider using ORM/Frameworks

Quick Review

Repository Object - Makes domain model independent of DB layer

- Aggregate operations are Atomic

- Used for Unit Testing & Mocking

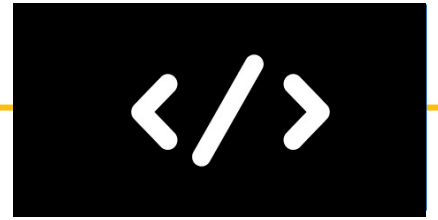
Performance Concerns - Expose higher level query functions

- Use caching solutions such as Redis/Memcached

- OK to expose Queries outside of Repository

Hands On : ACME Sales Model

UML model for the Repository for ACME Sales Aggregates | Entities



1. Repository objects
2. Test code walkthrough
3. Test flow in action

John, Travel Advisor

We are in the business of selling vacation packages. The sale process starts with a Customer calling us. Based on Customer's desires we select the packages and describe it. If customer shows interest we start a proposal for the selected package.

Customer Specifications

Vacation Package

Customer

Proposal

Offers

Purchase Order

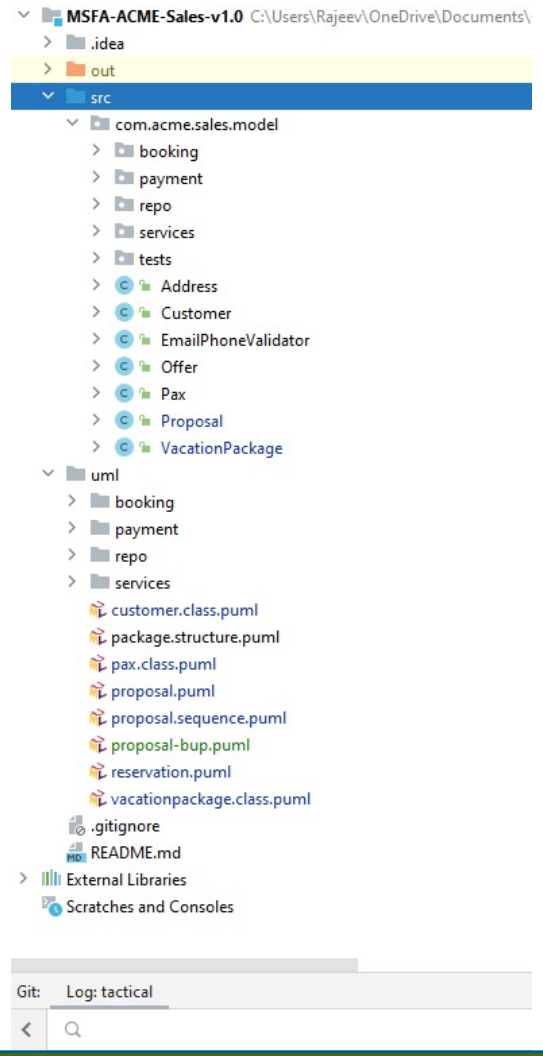
Pax or Passenger

Payment Information

Booking Confirmation

Objective

Demonstrate the idea behind Repository



Code & Model files are available in the git branch tactical

Self walk through of the code (*optional*)

Intent is NOT to teach/promote Java

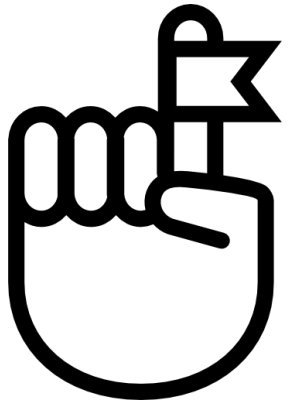
Note: Code is kept simple so that it is easy to follow

E.g., Error handling is minimal to reduce lines of code & to highlight key aspects in the implementation

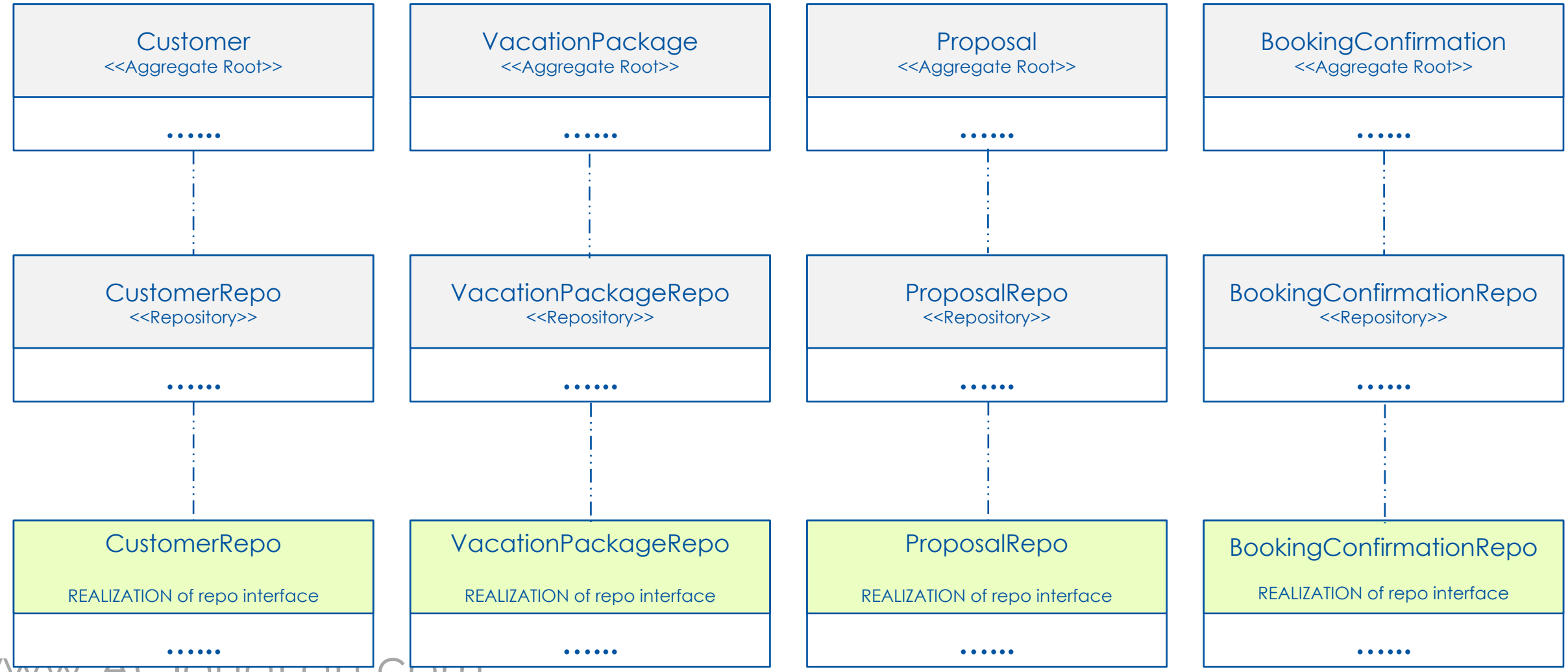
Aggregates & Repos

An Entity with 0 inner object is an Aggregate Root

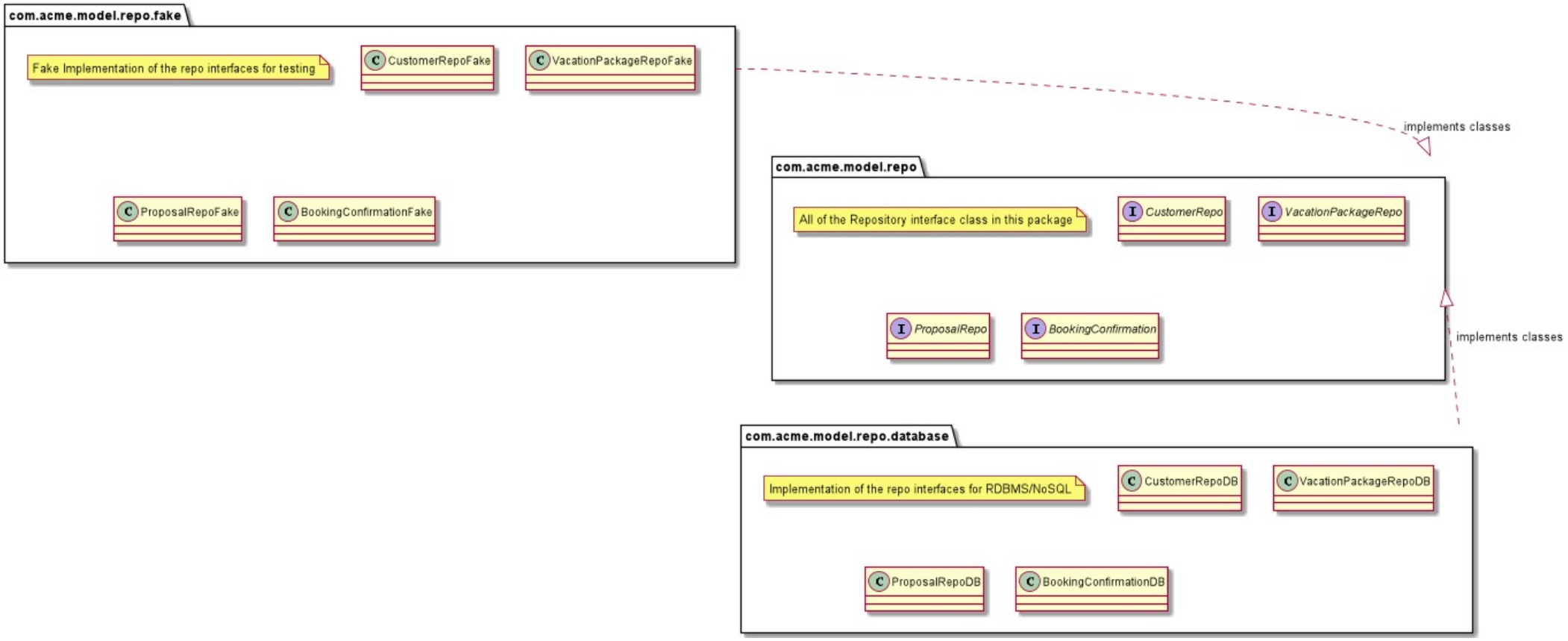
One to one Relationship



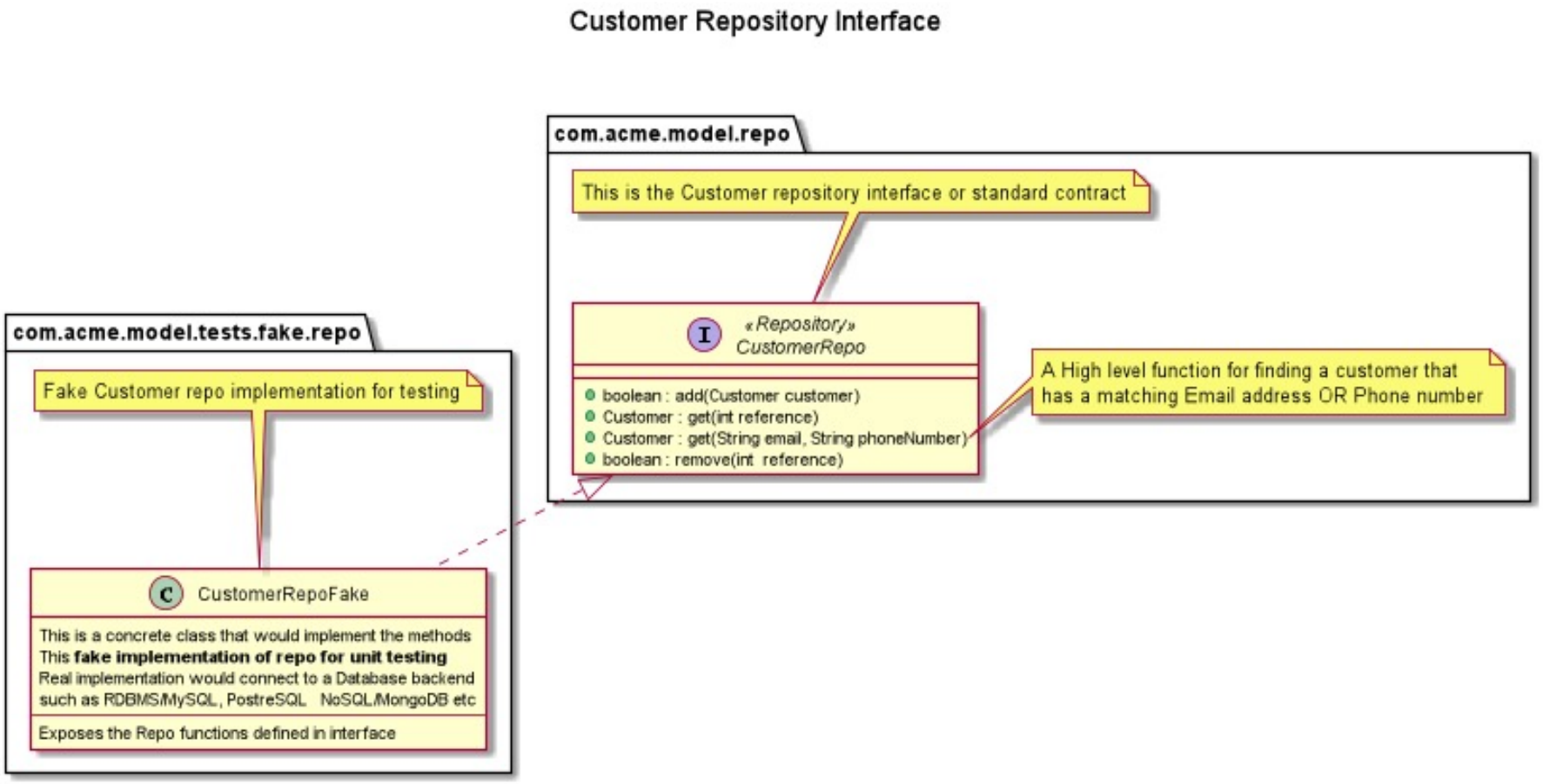
Aggregates in ACME model

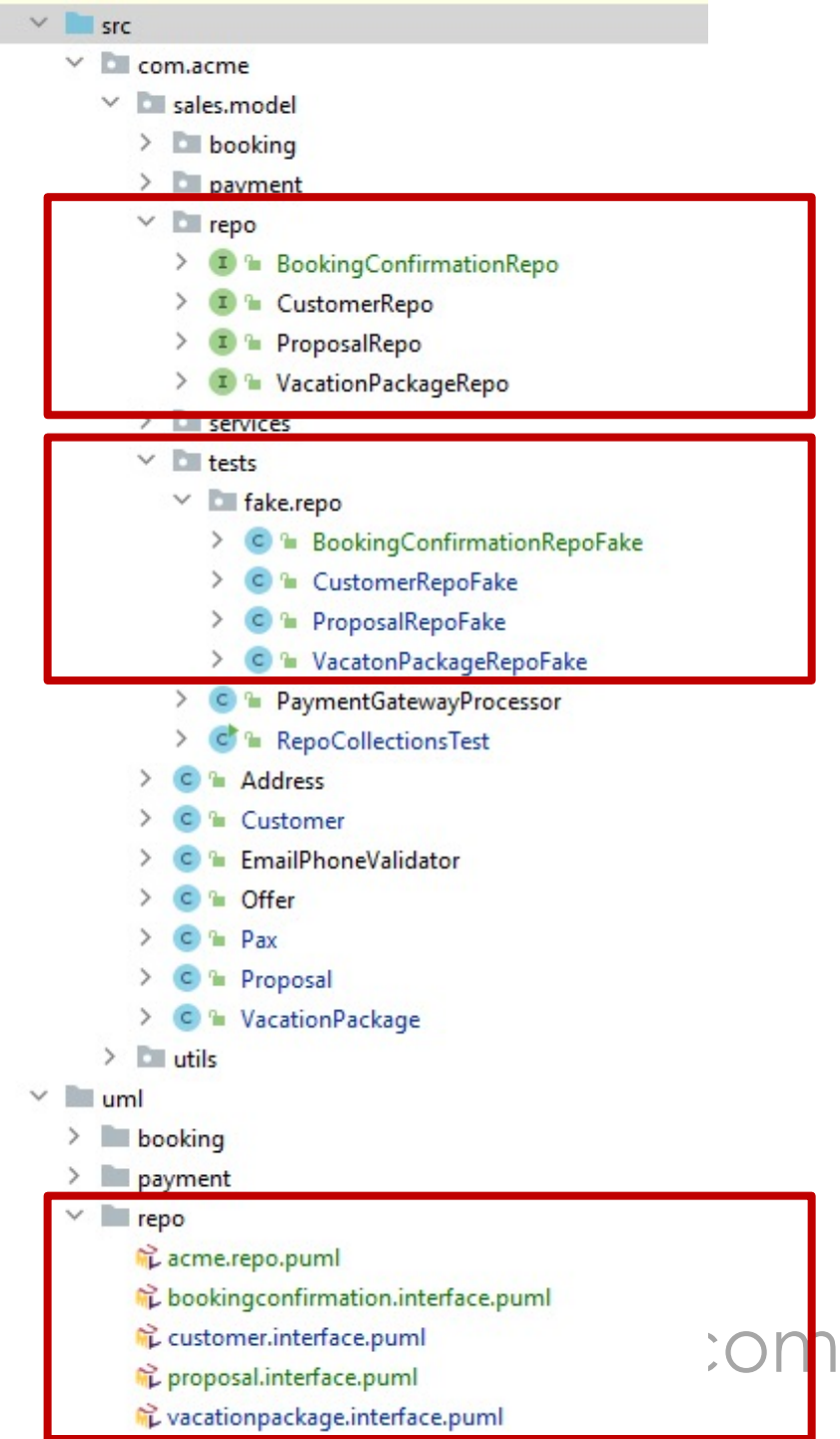


Model



Customer Repo Interface



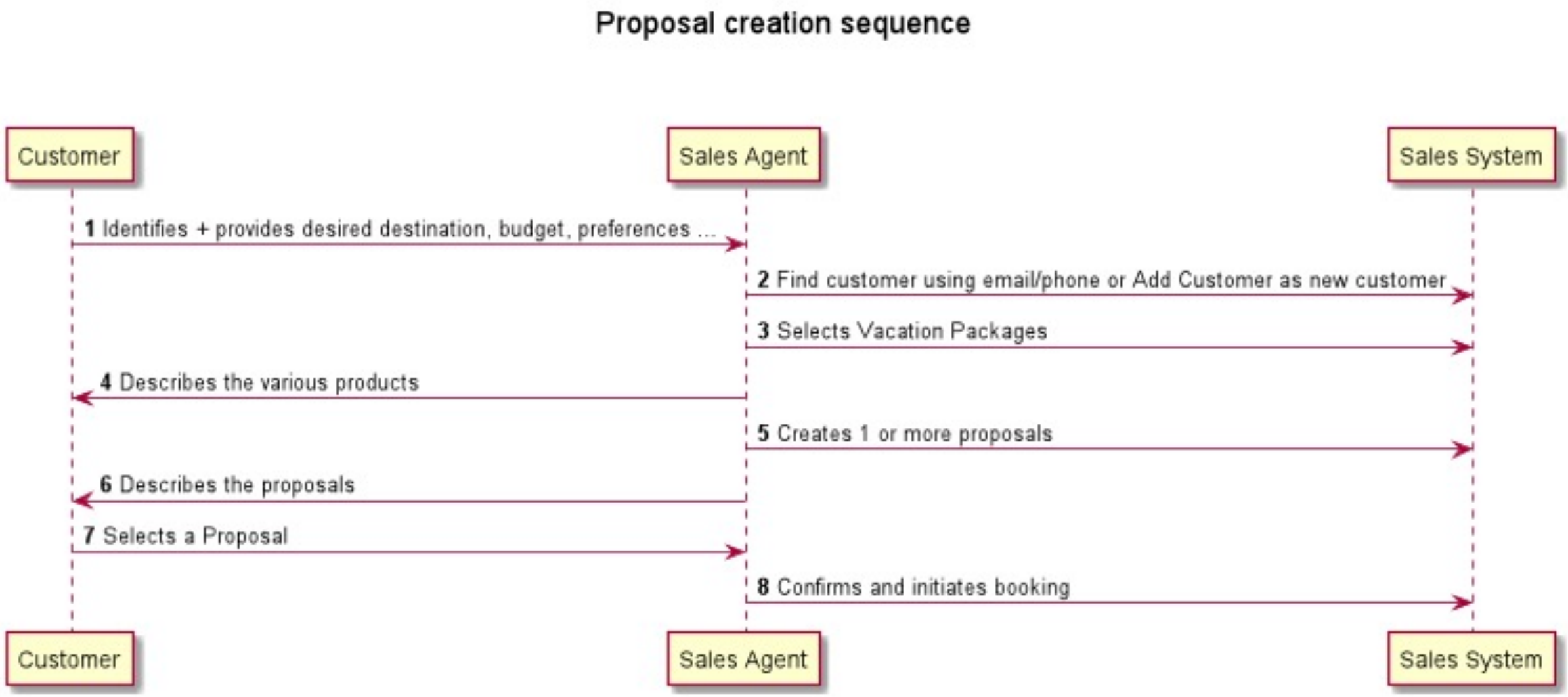


JAVA Repository object code

JAVA Fake Repository object implementation

Repository object models

Sequence Testing using FAKE repo implementation



Domain Services

Implementation of a behavior that is associated with multiple objects

1. Domain Service Pattern
2. Characteristics of Domain Services

SavingAccount
<<Entity>>

CheckingAccount
<<Entity>>

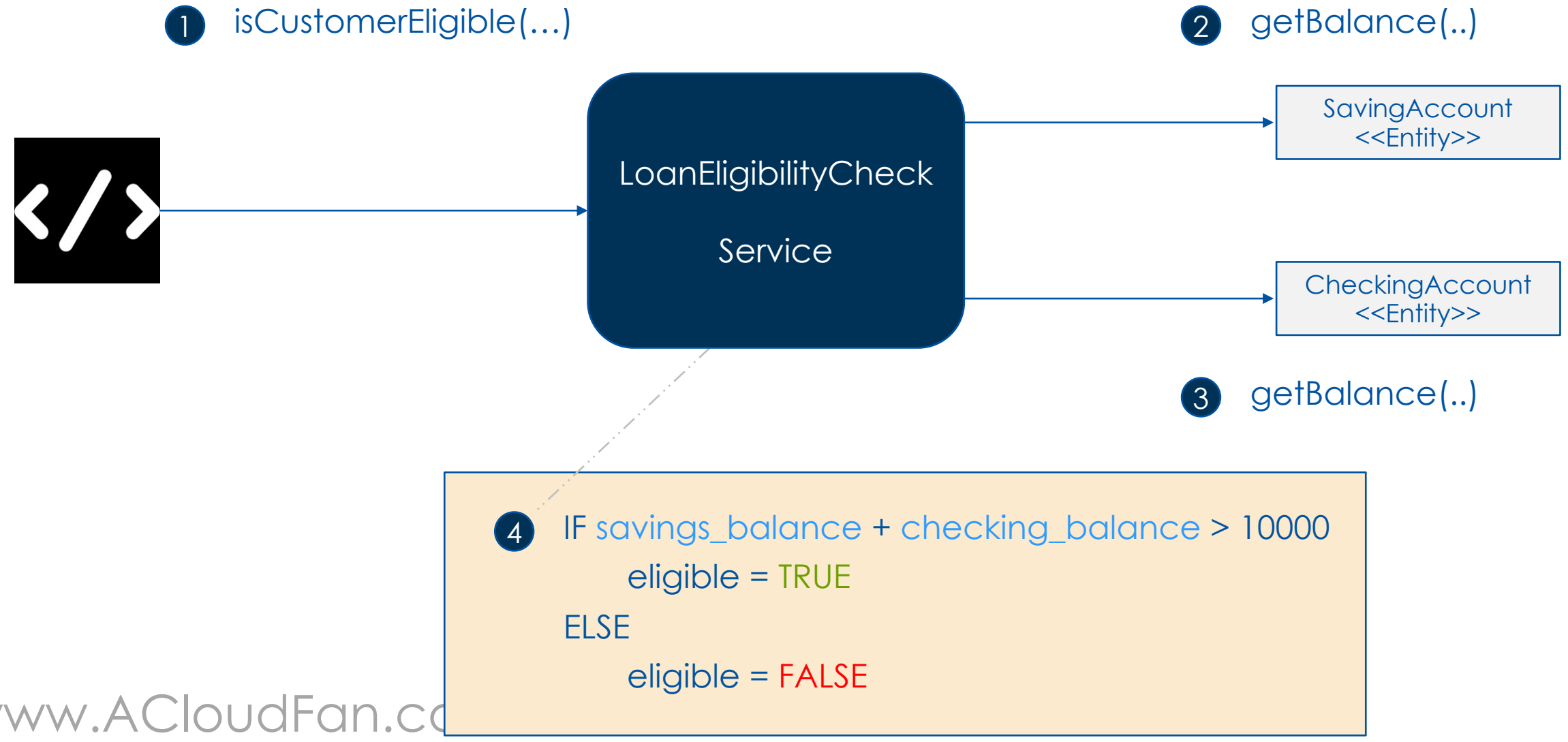
LoanAccount
<<Entity>>

Requirement:

Bank offers Loan to customers ONLY if their combined balance in bank > $10000

Where would you build this functionality?

Domain Service



Domain Service

A Domain Object that implements the Domain functionality (or concept) that may not be modeled (naturally) as a behavior in any domain entity or value object

Characteristics of Domain Services

#1 Business Behavior (i.e., Business Logic) for the Domain

#2 Stateless

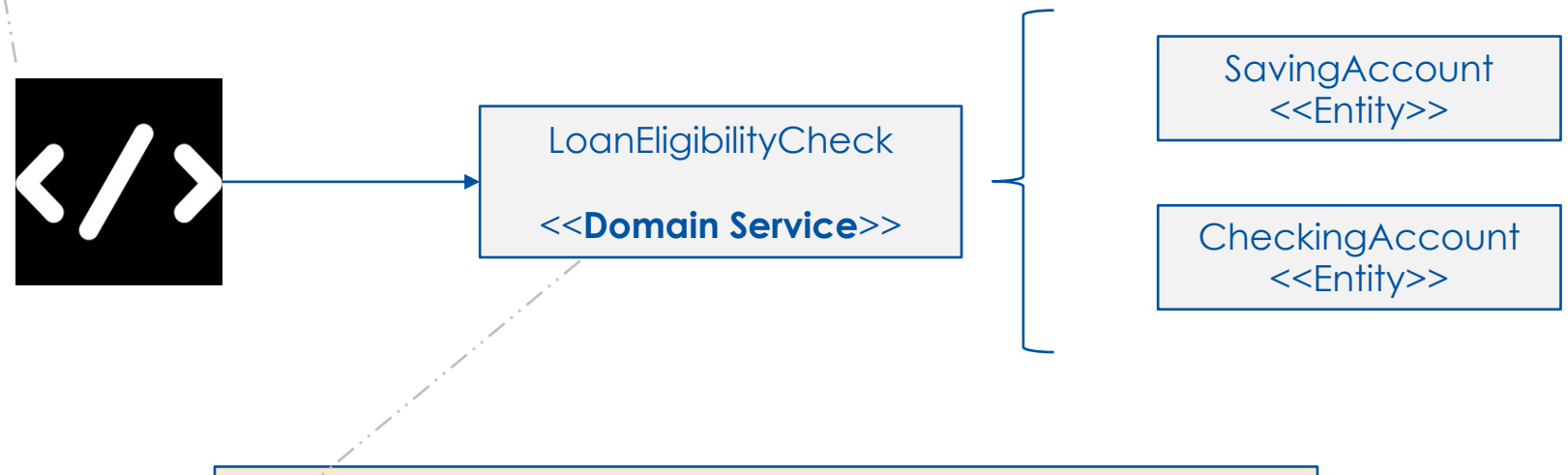
#3 Highly cohesive

#4 May interact with other Domain Services

#1 Domain Service has Business Behavior

Domain Service is aware of the domain objects

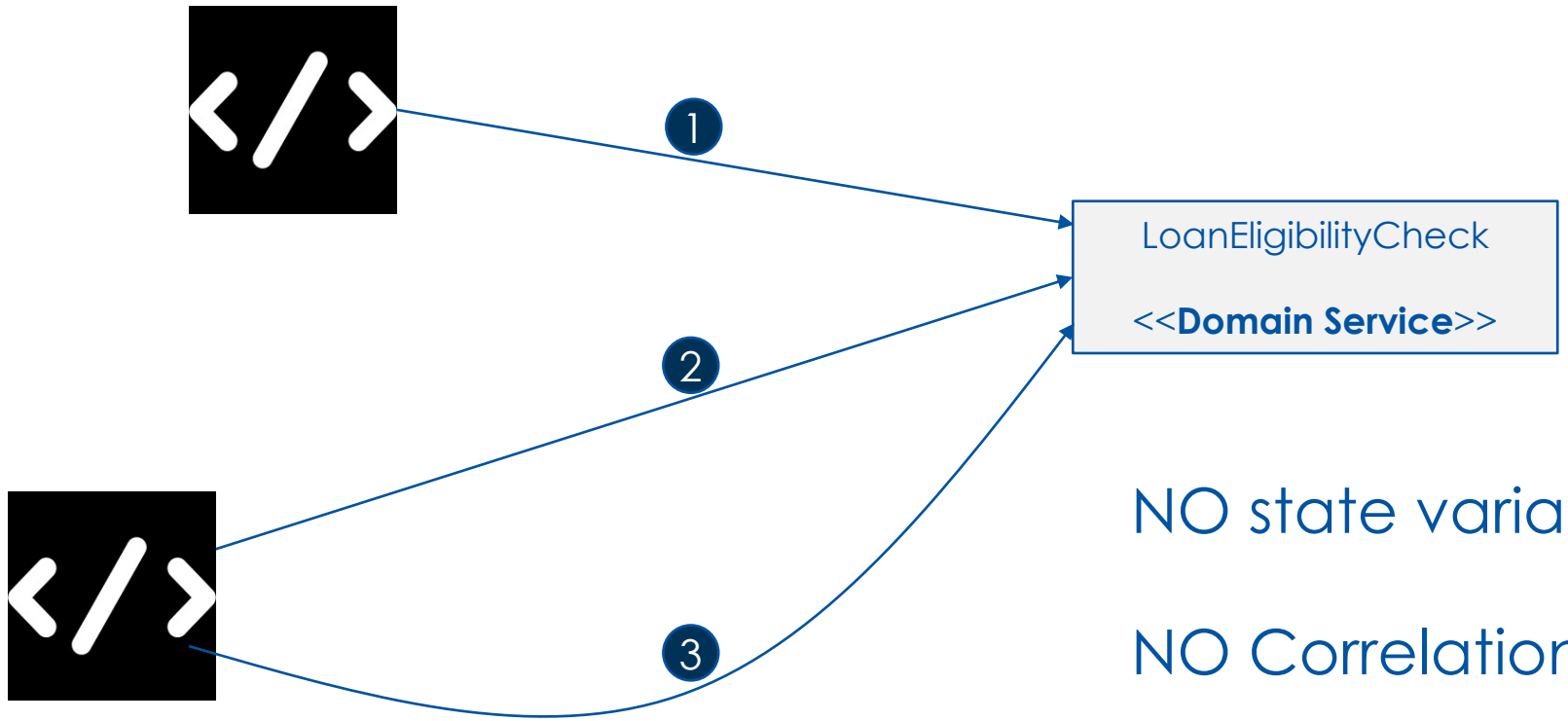
NO behavior awareness



- Business Rules
- Calculations
- Workflow
- Decision logic

#2 Stateless

Domain Service does not maintain state between calls



NO state variables or persistence

NO Correlation between calls

#3 Highly Cohesive

Does one and only one thing

LoanEligibilityCheck

<<**Domain Service**>>

Check for Loan Eligibility

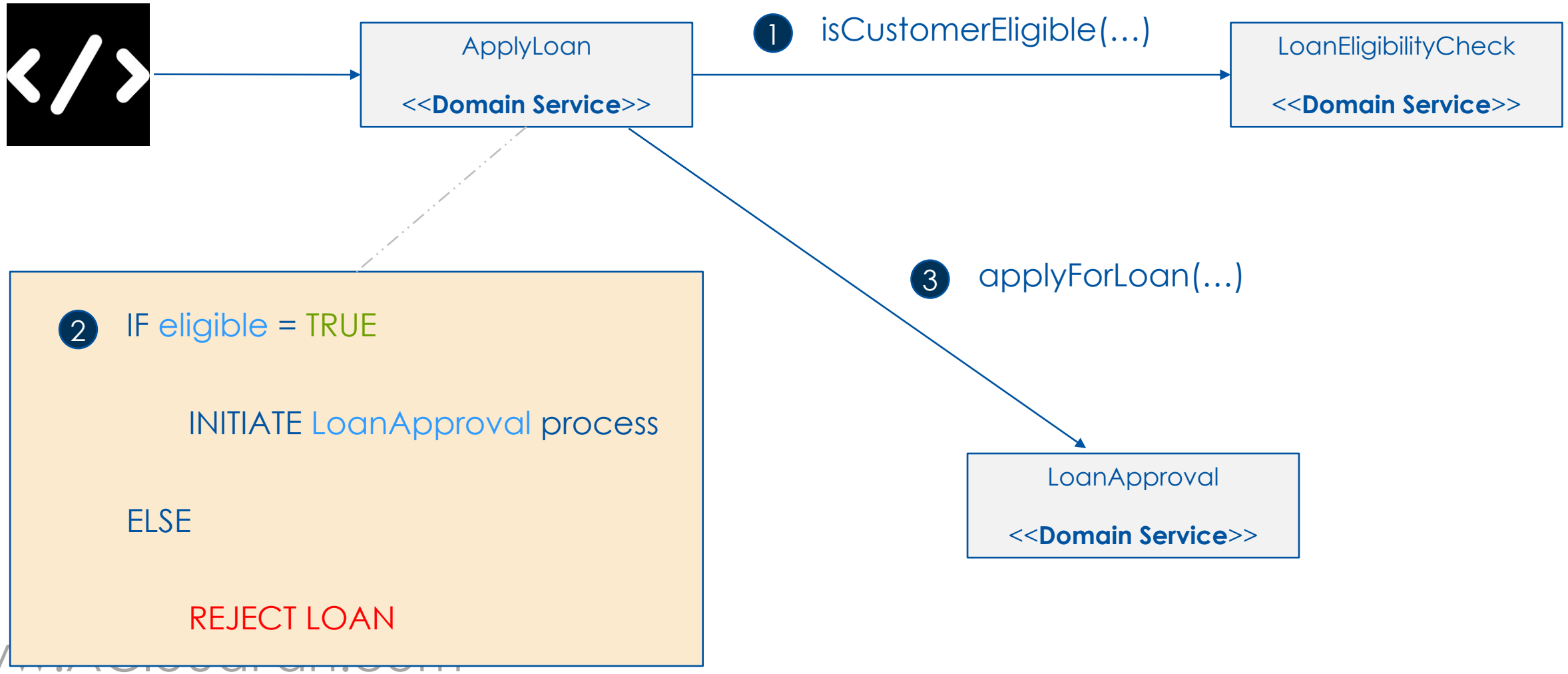
LoanApproval*

<<**Domain Service**>>

Rules & Flow for loan application

** Depends on credit history from all accounts*

#4 May interact with other Domain Services



Domain Service invocations

Services are technology agnostic

- May be POJO function calls
- May be over a network protocol such as HTTP/s, MQ

Quick Review

Domain Service implements Domain Behavior

- That does not fit naturally in other Entities and Value Objects

- Stateless
- Highly Cohesive
- May invoke other Domain Service(s)

Application Services

Implementation of a behavior that is not a core domain behavior

1. Application Service Pattern
2. Characteristics of Application Services

SavingAccount
<<Entity>>

CheckingAccount
<<Entity>>

Customer
<<Entity>>

Requirement:

CustomerPortfolio

Create capability that would consolidate the data into Customer Portfolio object and return to caller

Would you implement it as Domain Service?

Would you implement it as a Domain Service?

Requirement:

CustomerPortfolio

Create capability that would consolidate the data into Customer Portfolio object and return to caller

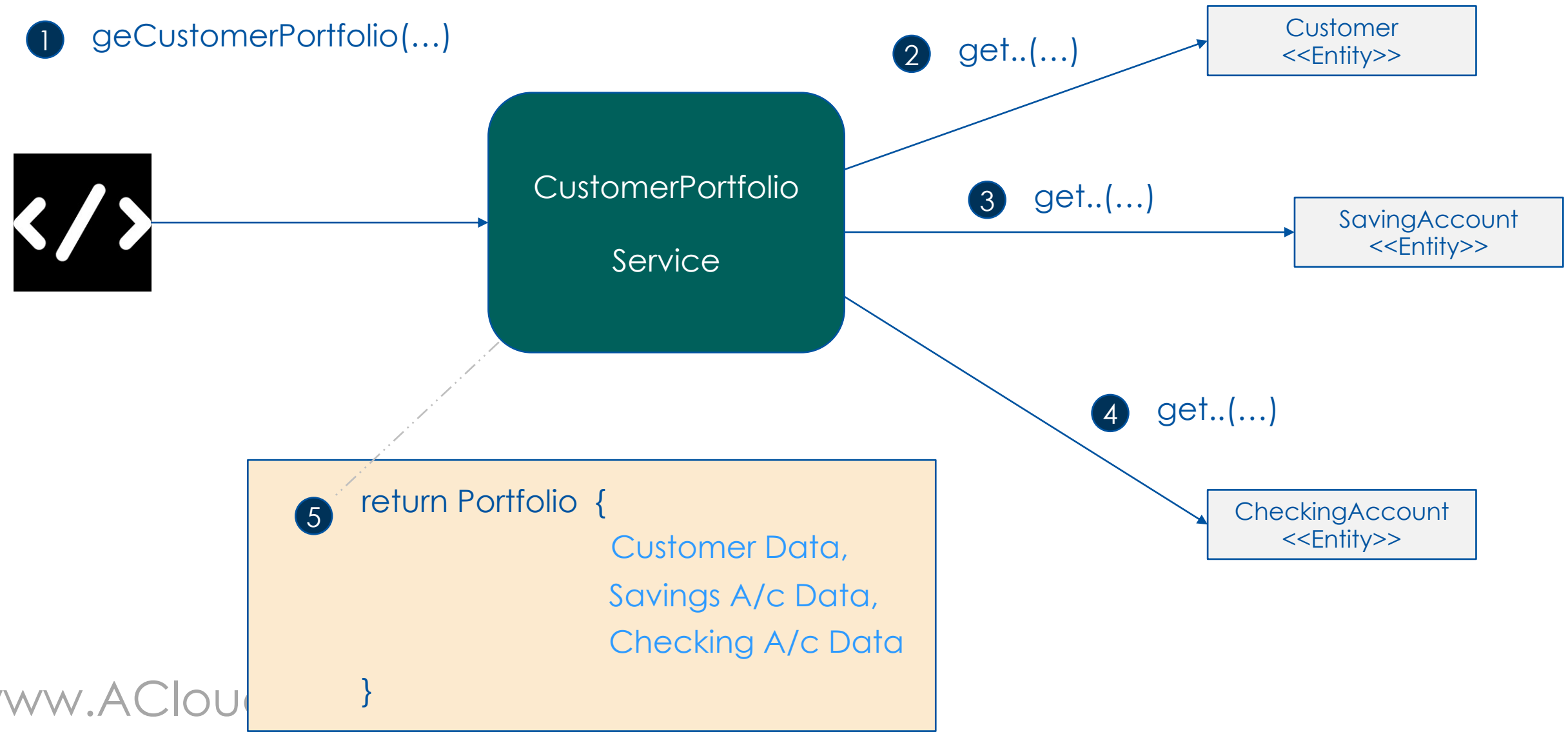
A Domain Object that implements the Domain functionality (or concept) that may not be modeled (naturally) as a behavior in any domain entity or value object

Customer Portfolio Service will not implement any domain functionality !!!

Application Service

A Domain Object that does NOT implement any Domain functionality but depends on other Domain Objects | Services for exposing higher-level domain functionality to the consumers external to the model.

Application Service = No Domain Logic



Characteristics of Application Service

#1 **NO Domain Logic**

#2 Stateless

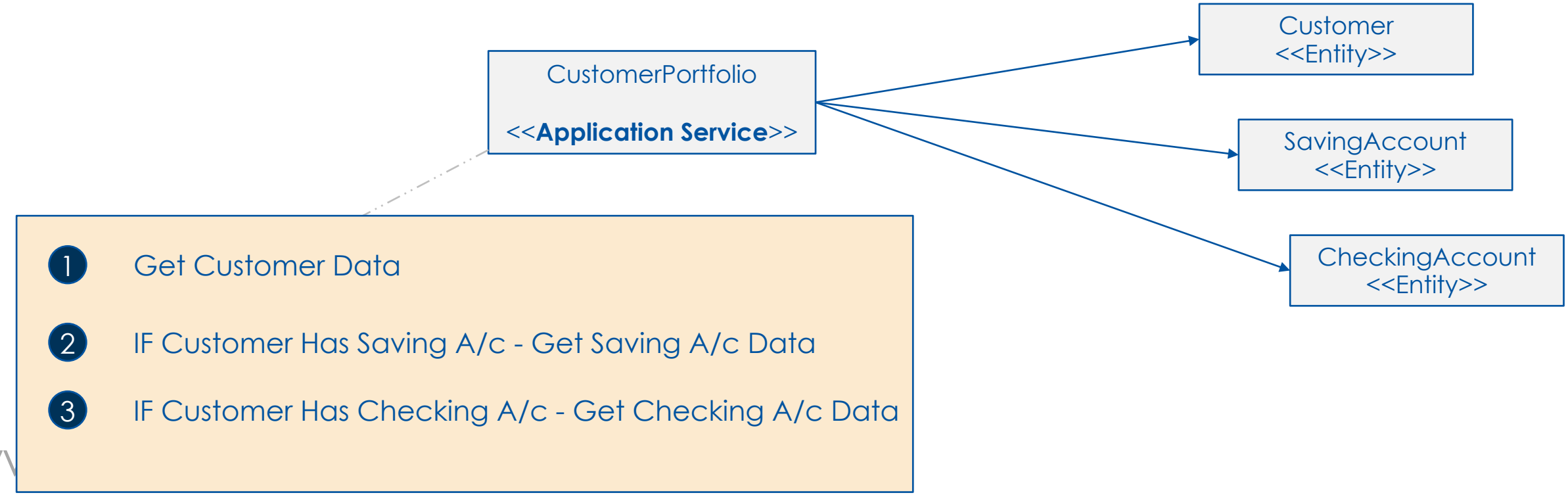
#3 Defines an external interface

#4 Exposed over network

#1 No Domain Logic

Depends on the Domain Objects for Domain Logic

- Orchestrates the execution of Domain Logic



#2 Stateless

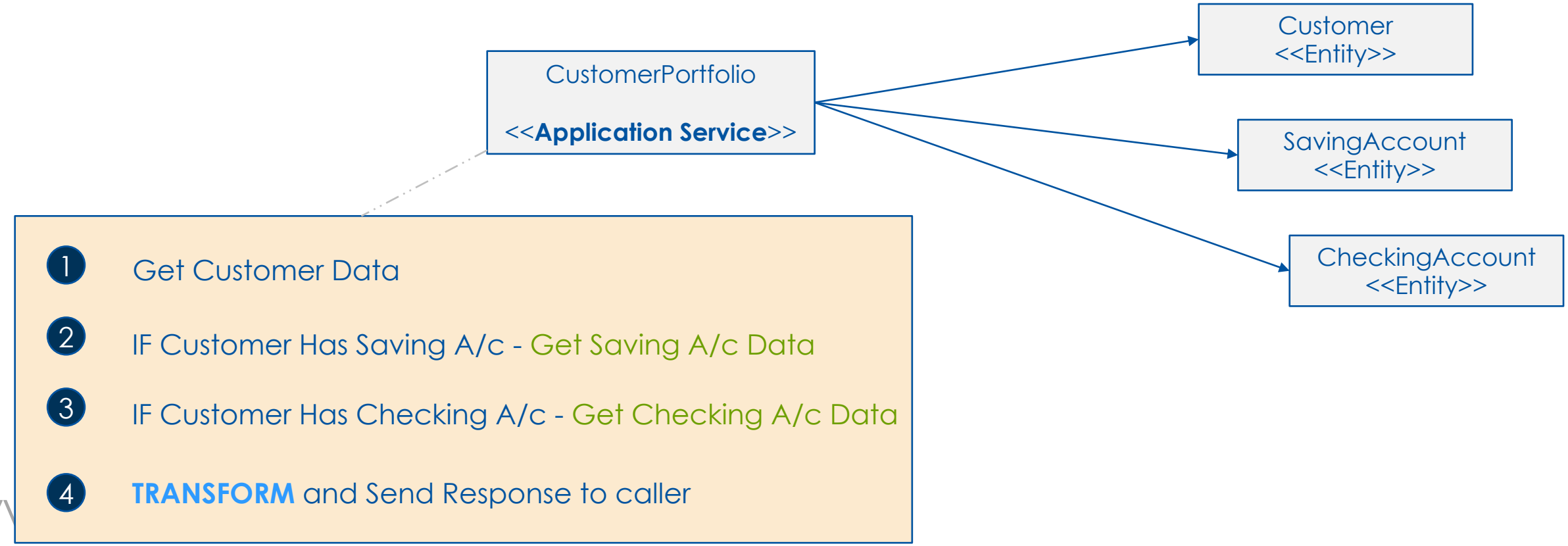
No State management carried out in the Application service

- No state variables or persistence of domain objects
- Depends on the Domain objects for persistence

#3 Defines an external interface

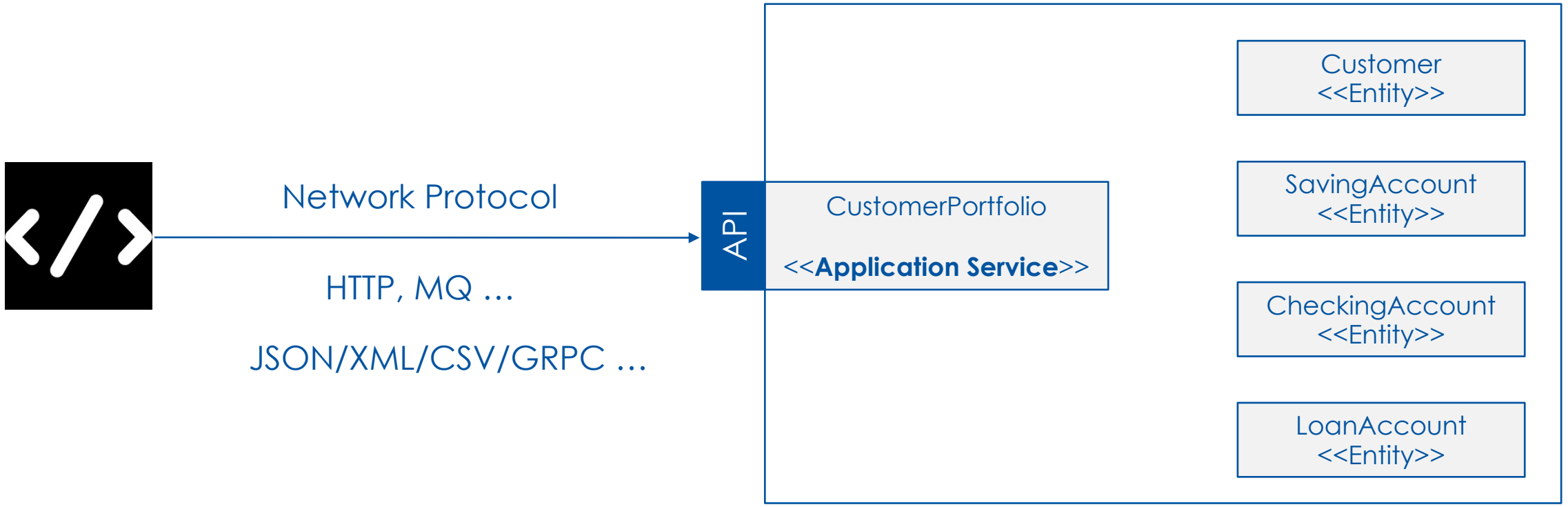
Exposes the interface used by outside world

- Interface doesn't need to be Entity | Value object



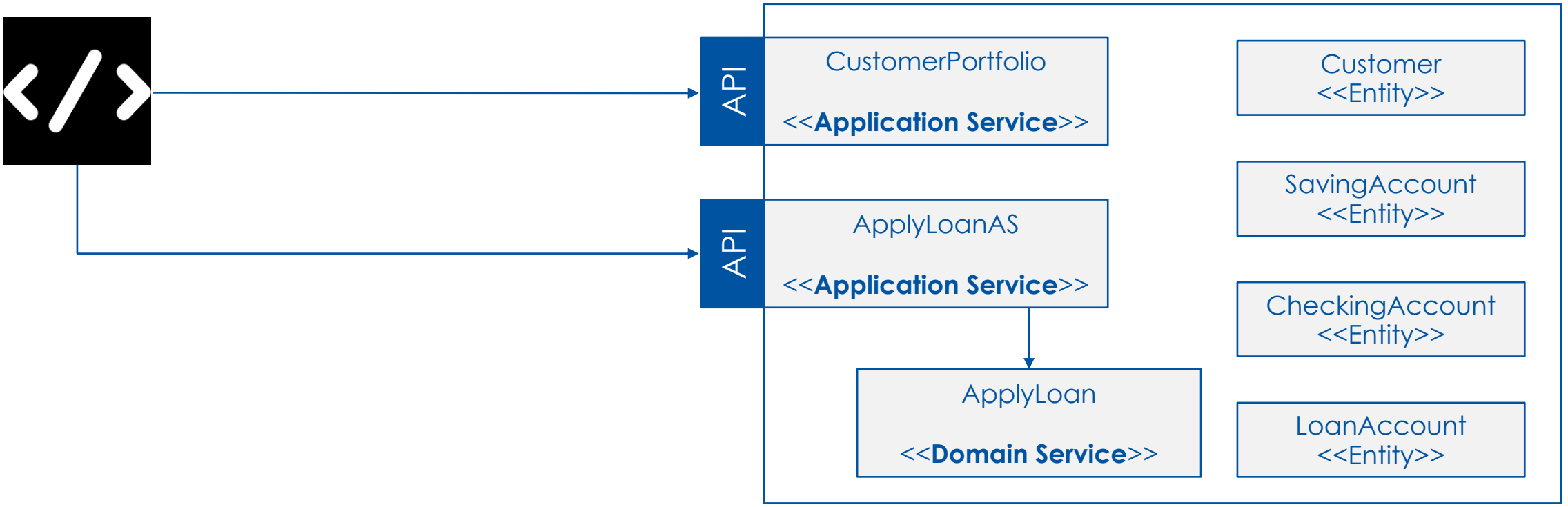
#4 Exposed over network

Exposes the interface over network protocol



Relationship with Domain Services

Application Service may expose a Domain Service



Quick Review

Application Service does NOT implement any Domain Behavior

- Orchestrates execution of domain logic in domain objects
- Exposes interface to external components
- Consumed over a network protocol such as HTTP, MQ

Infrastructure Services

Keeps the model independent of the infrastructure details | dependencies

Infrastructure Service Pattern

Characteristics of Infrastructure Services

Infrastructure Service

A service that interact with an external* resource to address a concern that is not part of the primary problem domain

It defines a contract used by the Domain Objects to interact with outside services.

** External to the model*

Examples : External Resources

- Logging system e.g., Fluentd, ElastiSearch
- Notifications e.g., Email, SMS
- Persistence mechanism e.g., Database, File system
- External APIs e.g., Google maps, Salesforce API

Characteristics of Infrastructure Services

#1 NO Domain Logic

#2 Single Responsibility

#3 Standard Interface | Contract

#1 NO Domain Logic

Provides an Infrastructure service NOT a Business service

- No direct dependency on Domain Objects
- Consumed by other Domain Objects | Services

#2 Single Responsibility

Addresses ONE and only one concern

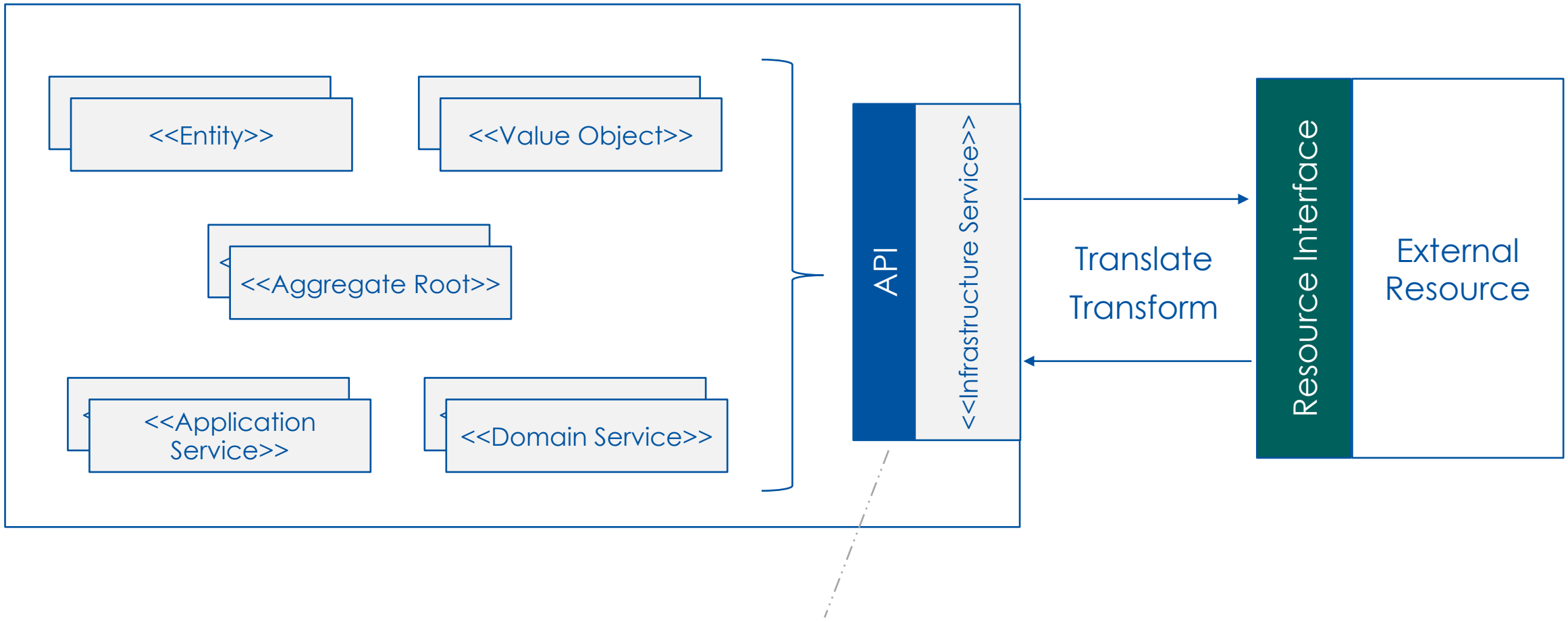
- Intent is to simplify implementation and make it easy to understand
- Example:

#3 Standard Interface | Contract

Defines a standard contract between model & external resource

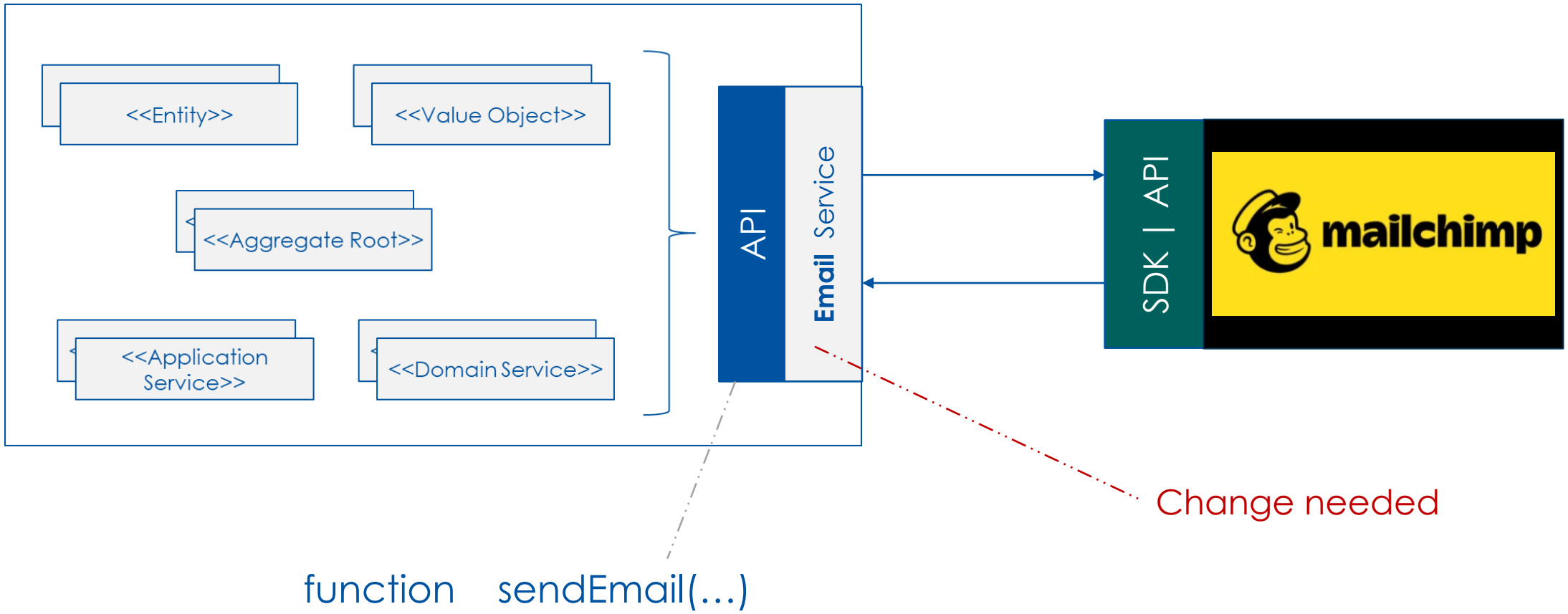
- Think - API meant for consumption by model objects & services
- Realization of API is in the "*Infrastructure Layer*"
- Makes the model, technology (& external service) agnostic

Example: Standard Interface



Email: function **sendEmail(…)**

Agnostic to External Resource | Realization



Domain Model is insulated from the external resource changes !!!

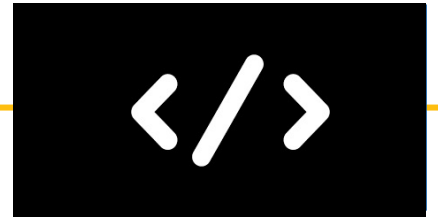
Quick Review

Infrastructure Service does NOT implement any Domain Behavior

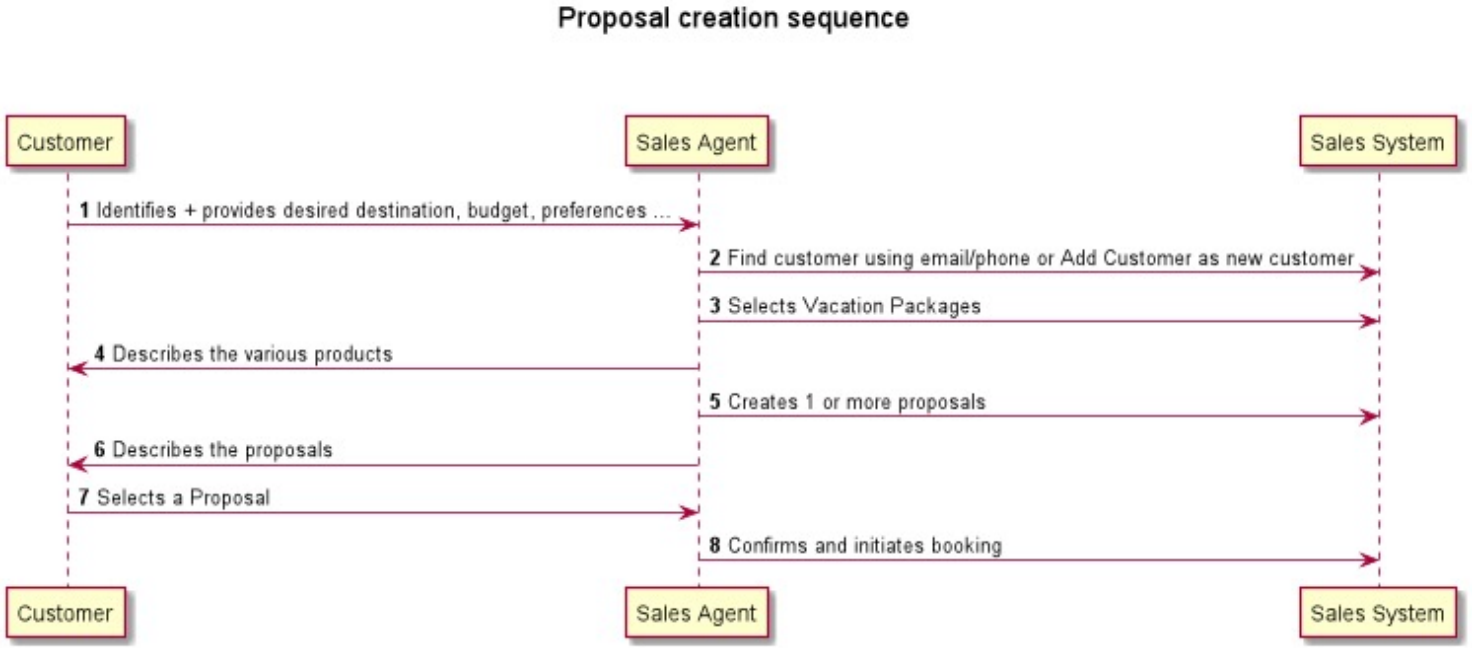
- Exposes external resources via standard interface / contract
- Insulates the domain model from changes in external services

Hands On : ACME Model (1 of 2)

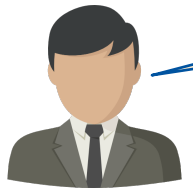
UML Model and JAVA code for a Domain Service



1. Flaw in "*Proposal Creation Sequence*"
2. Addressing the issue
3. Pricing service



We missed the Proposal Pricing step in the sequence !!!!



John, Travel Advisor

Vacation Package price is Per Person Price

It does not Include

- Taxes
- Surcharges

Let me add the term Surcharge and Per Person Price in our UL/Business vocabulary

I understand the tax but explain the Surcharge further

Apply surcharge wherever applicable

- Airport
- Rental Car
- Hotel
- Resort
- State tourism

We use spreadsheets to calculate the prices.

These spreadsheets get updated every quarter

Surcharge amount varies based Provider, State & Season

Proposal Price = Number of PAX x Price Per Person

\+ All Surcharges

\+ State Taxes

A. In the **Booking Confirmation <<entity>>**

B. In a **Domain Service**

C. In an **Application Service**

D. In an **Infrastructure Service**

How would you build pricing functionality?

Pricing functionality

A. In the **Booking Confirmation <<entity>>**

B. In a **Domain Service**

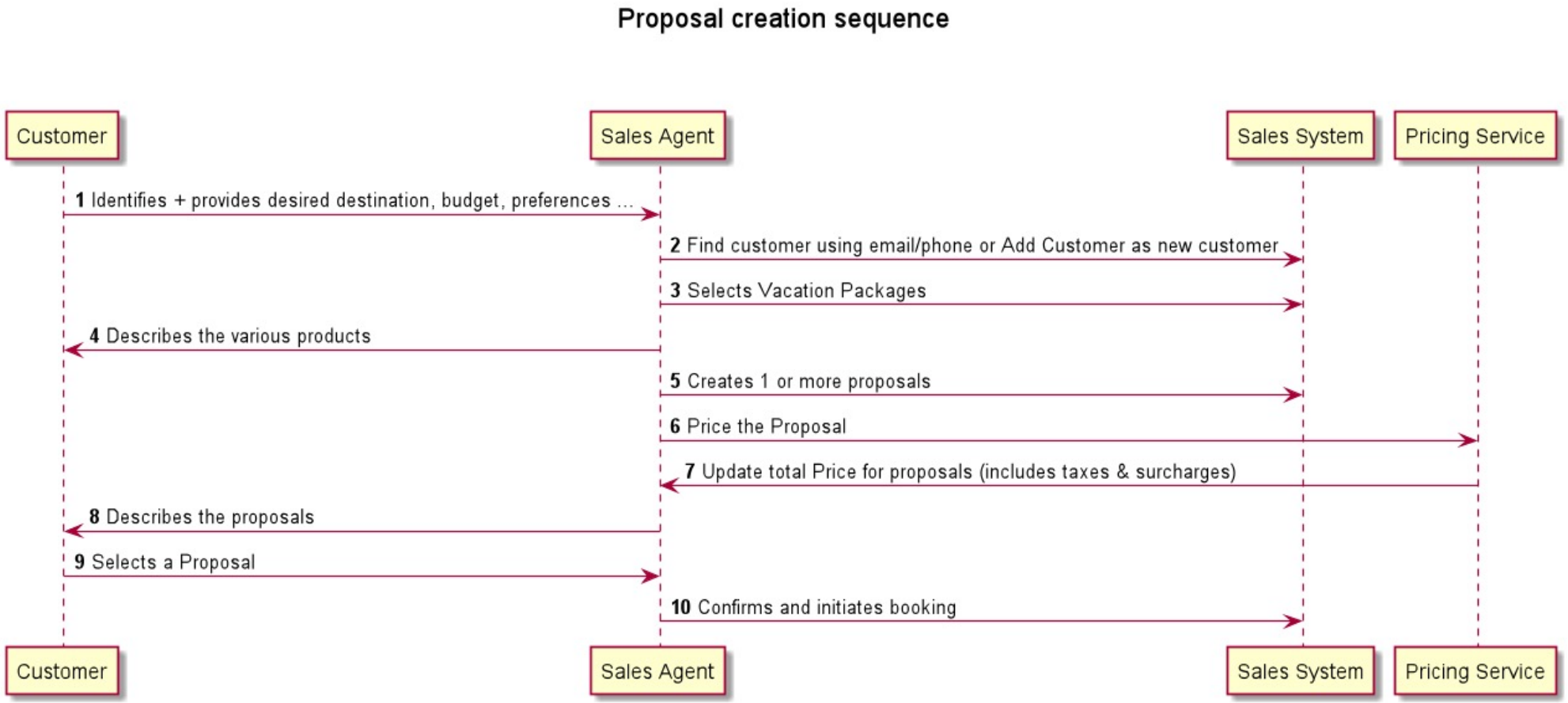
C. In an **Application Service**

D. In an **Infrastructure Service**

Price calculation does not fit in the entity naturally

These do NOT have Business Logic | Calculations

Sequence Diagram

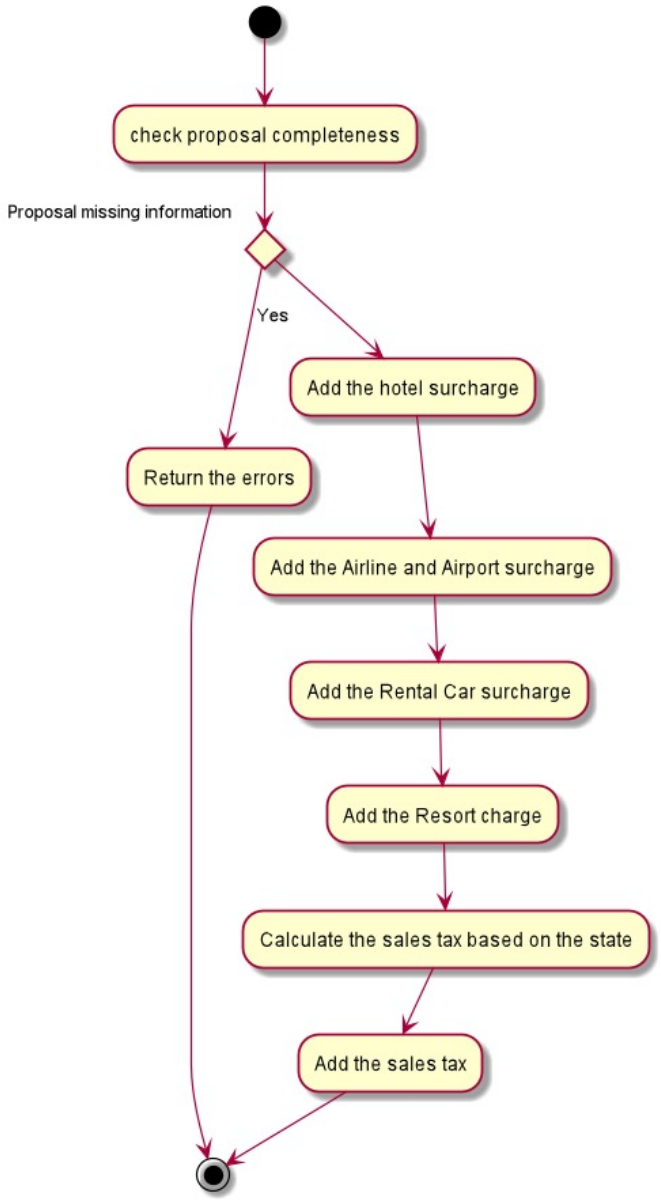


Business Logic modelling

Commonly carried out in

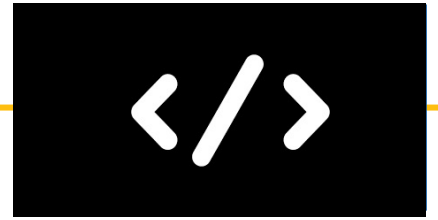
- Flow charts
- State diagrams
- Activity diagrams

Activity diagram



Hands On : ACME Model (2 of 2)

UML Model and JAVA code for Application & Infrastructure Services



Credit Card Service

Customer History Service

1. Service for **Credit Card processing via external vendor**

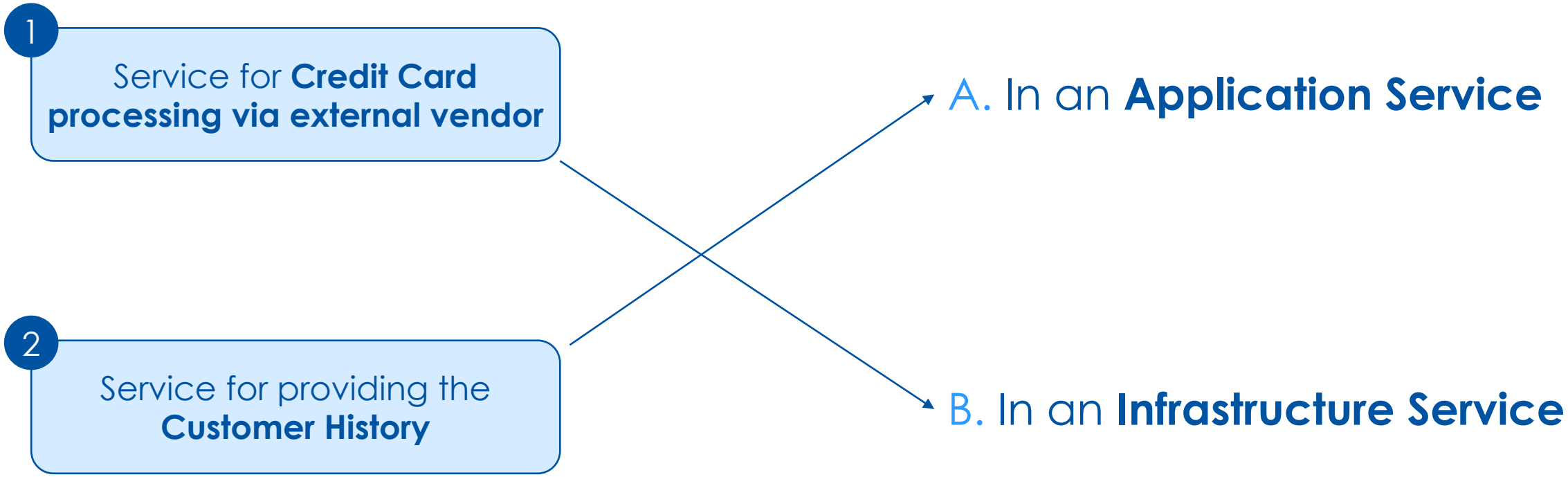
2. Service for providing the **Customer History**

A. In an **Application Service**

B. In an **Infrastructure Service**

Match the model with service type?

Application | Infrastructure Service

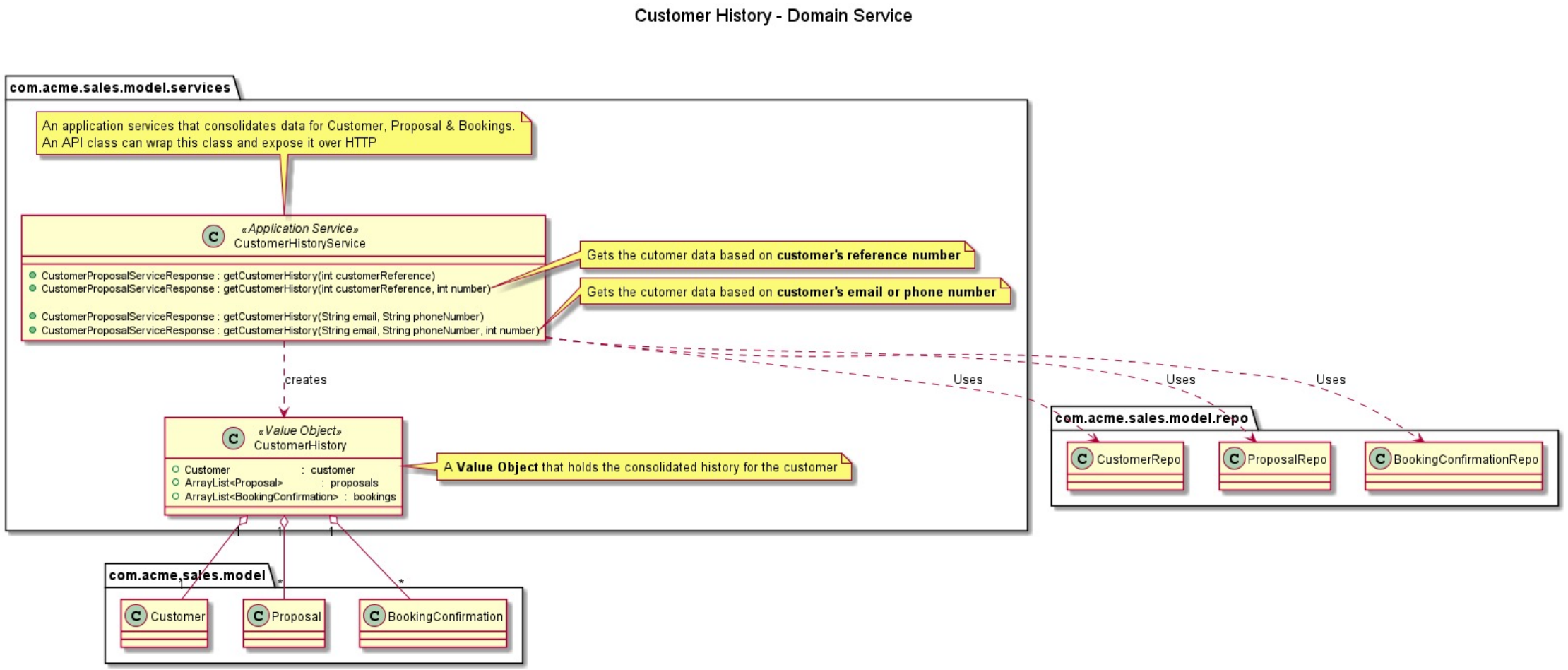


Customer History

Consolidates information for the customer

- Customer
- Proposals (recent 10)
- Past Booking Confirmations (recent 5)

Customer History



Payment Gateway Service

ACME uses 3rd party credit card services

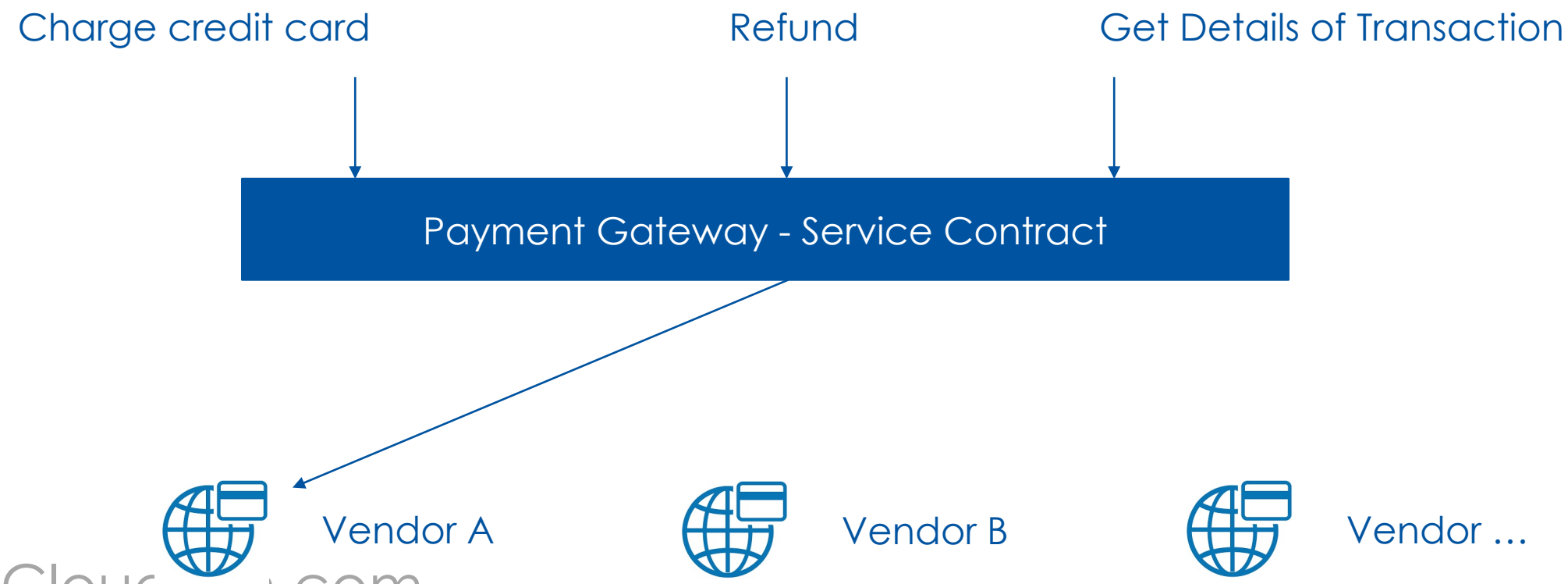
We pick up the payment vendor who gives us the best deal
i.e., low processing fees

For this reason we have switched the payment vendor 3 times in last 2 years

Every time we make a switch it's a hassle ☹ as changes need to be made to all systems

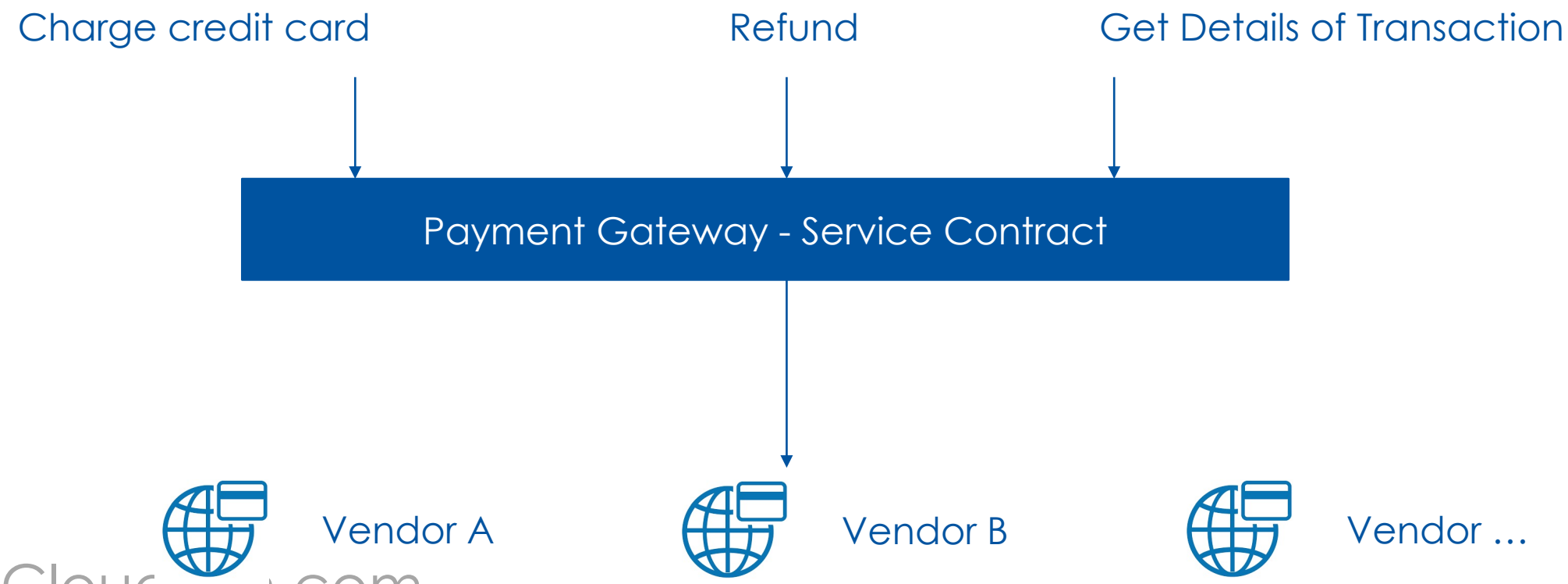
Payment Gateway Service

An Infrastructure Service that defines the standard contract



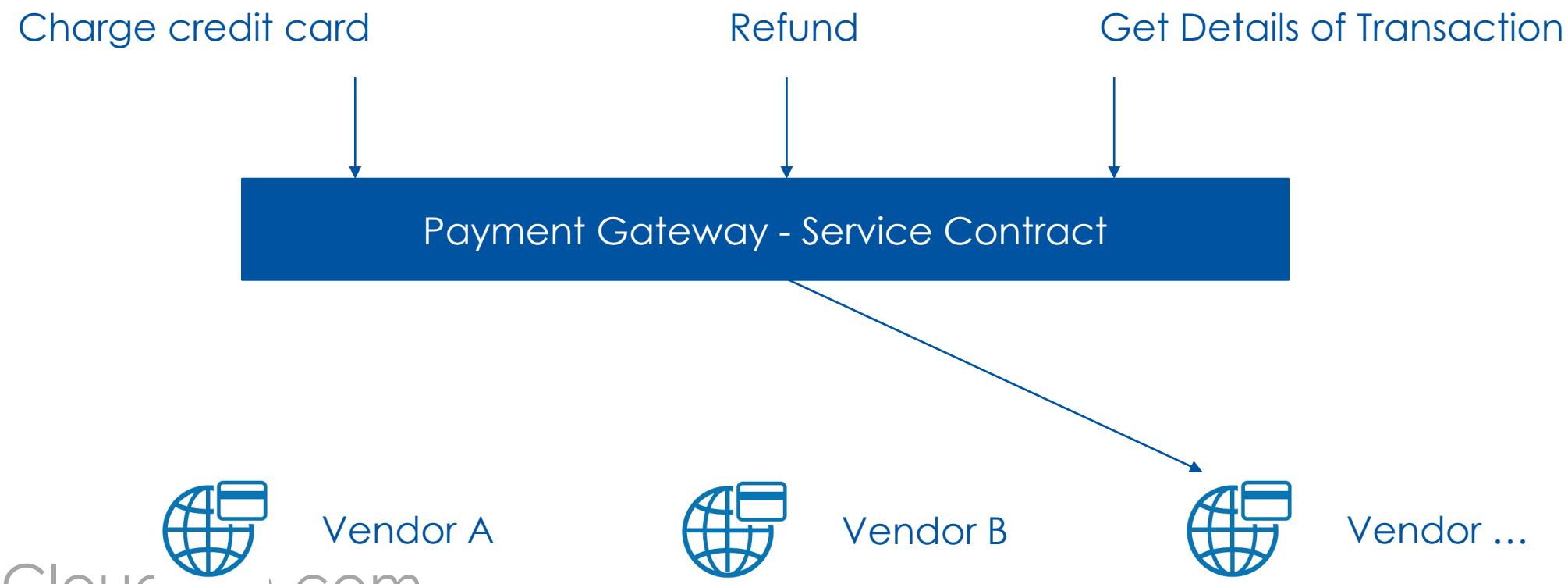
Payment Gateway Service

An Infrastructure Service that defines the standard contract

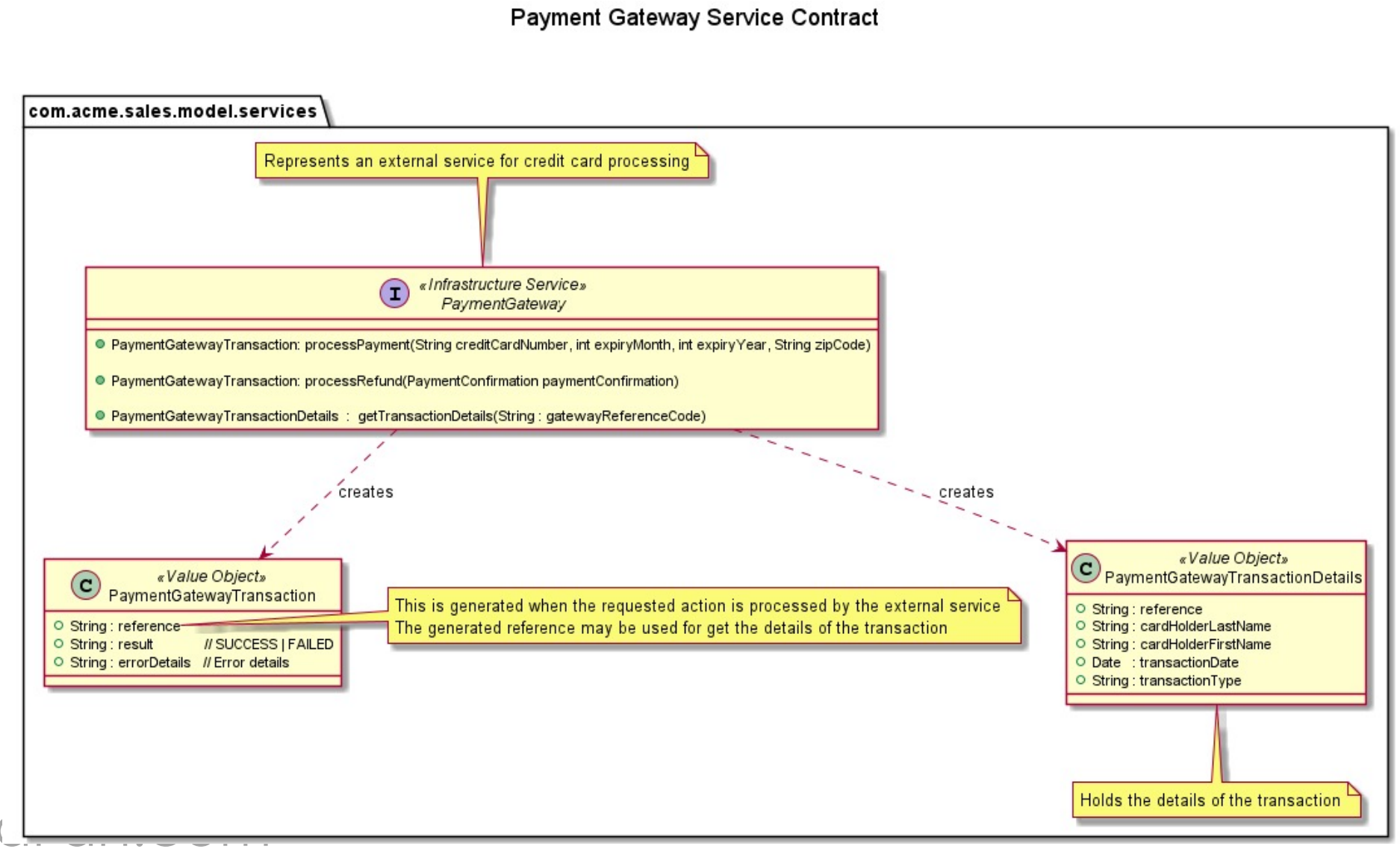


Payment Gateway Service

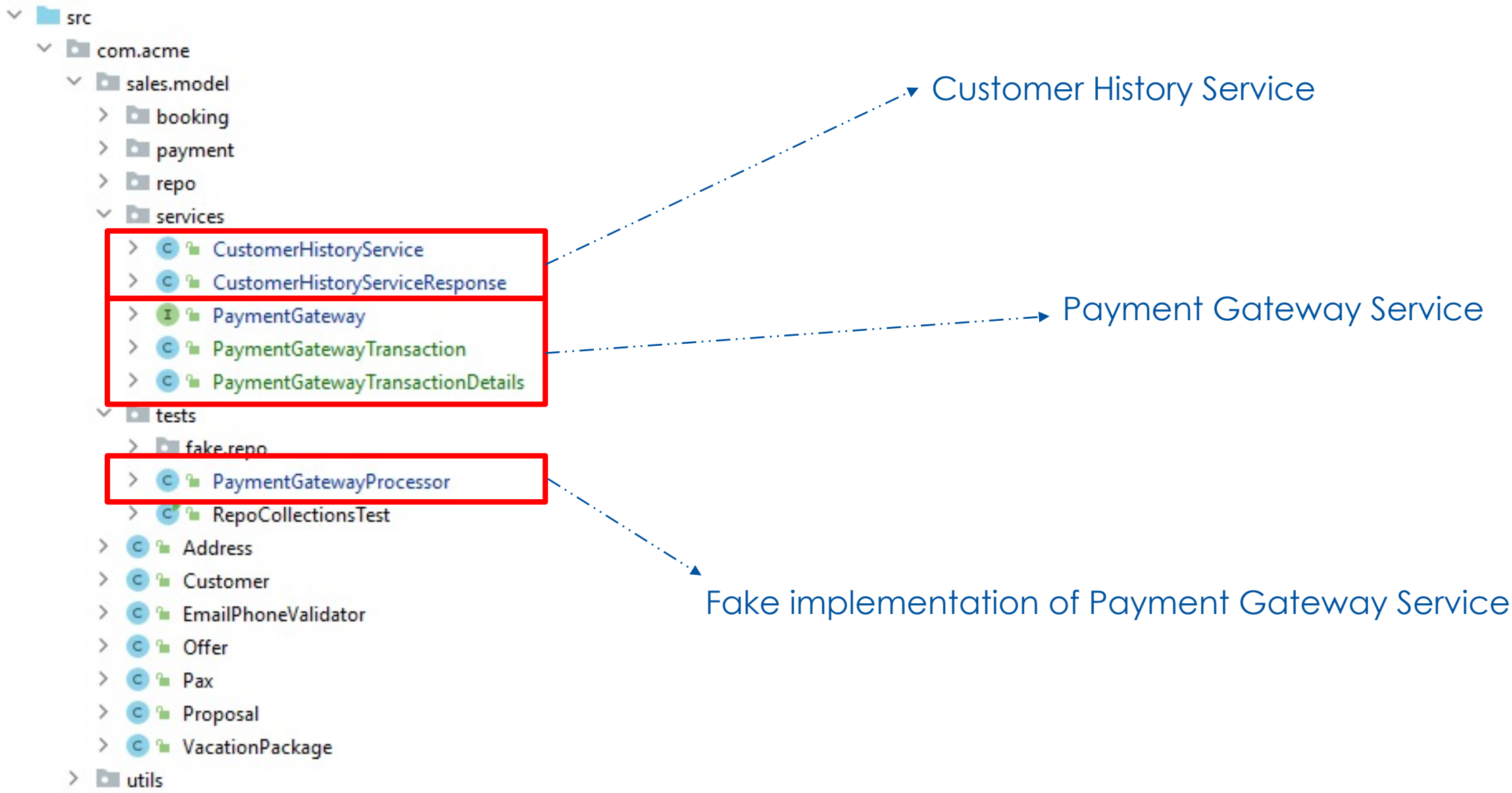
An Infrastructure Service that defines the standard contract



Payment Gateway Service

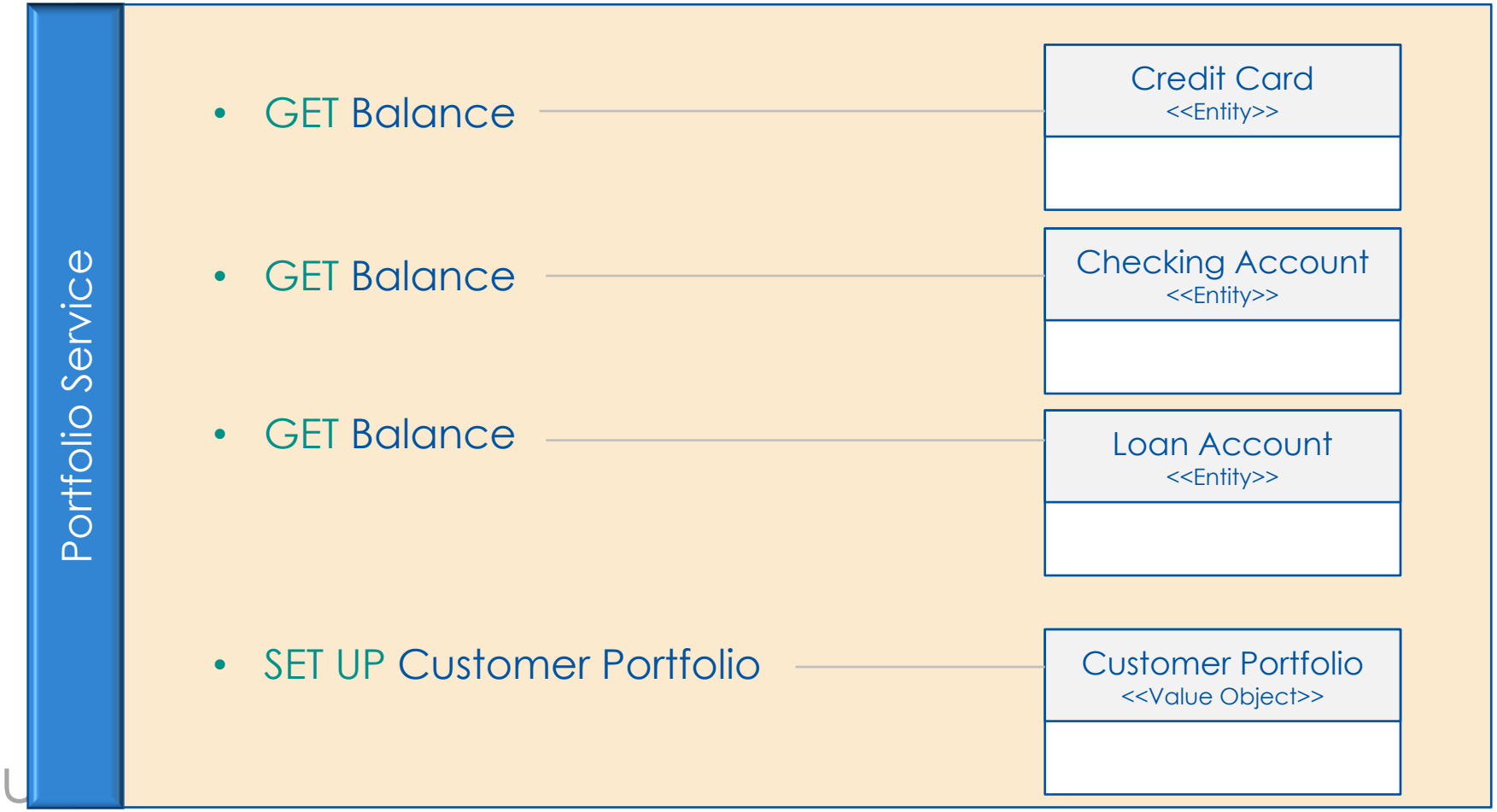


JAVA implementation



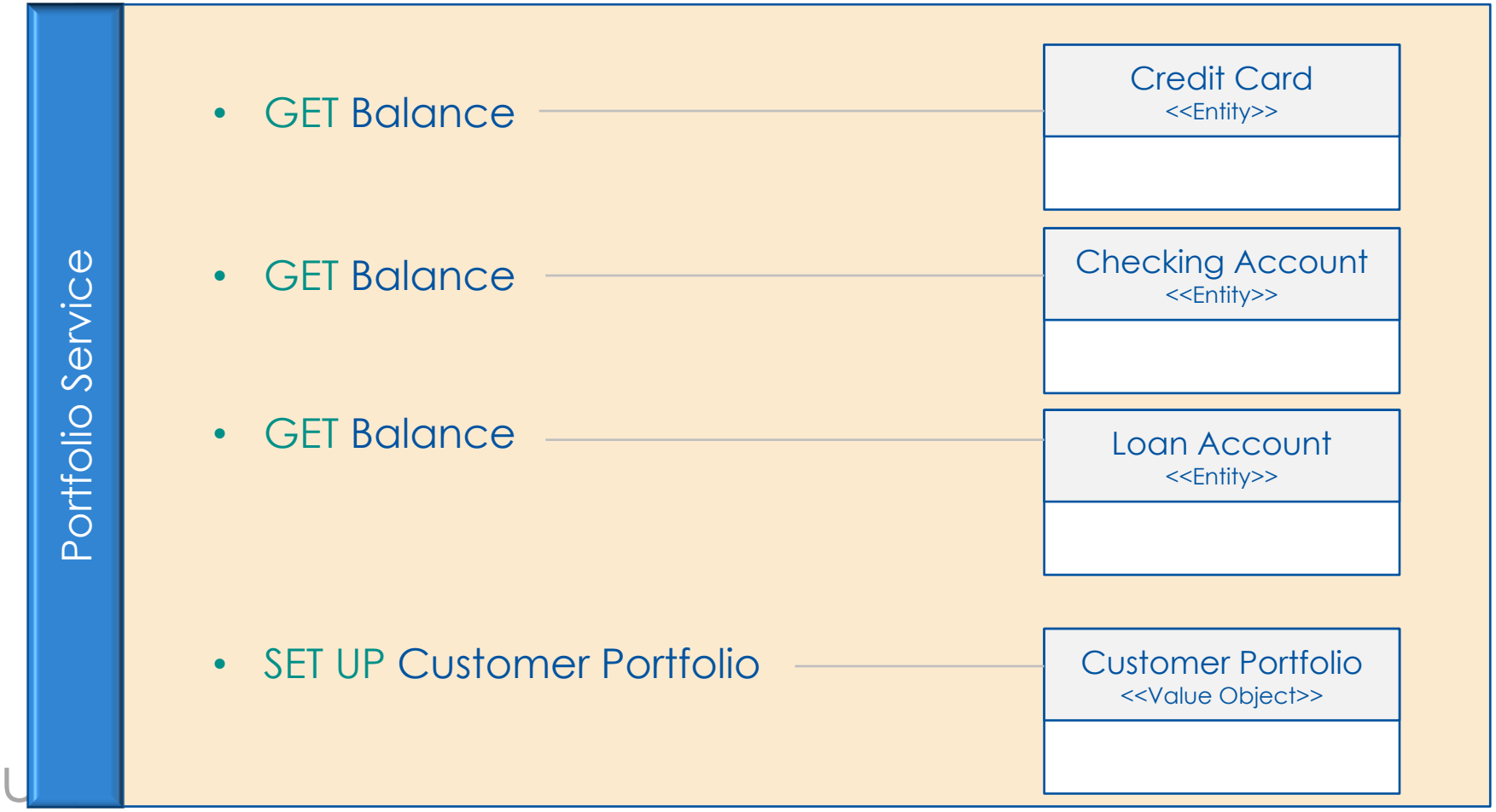
Domain Service

Implement as domain model aware independent components



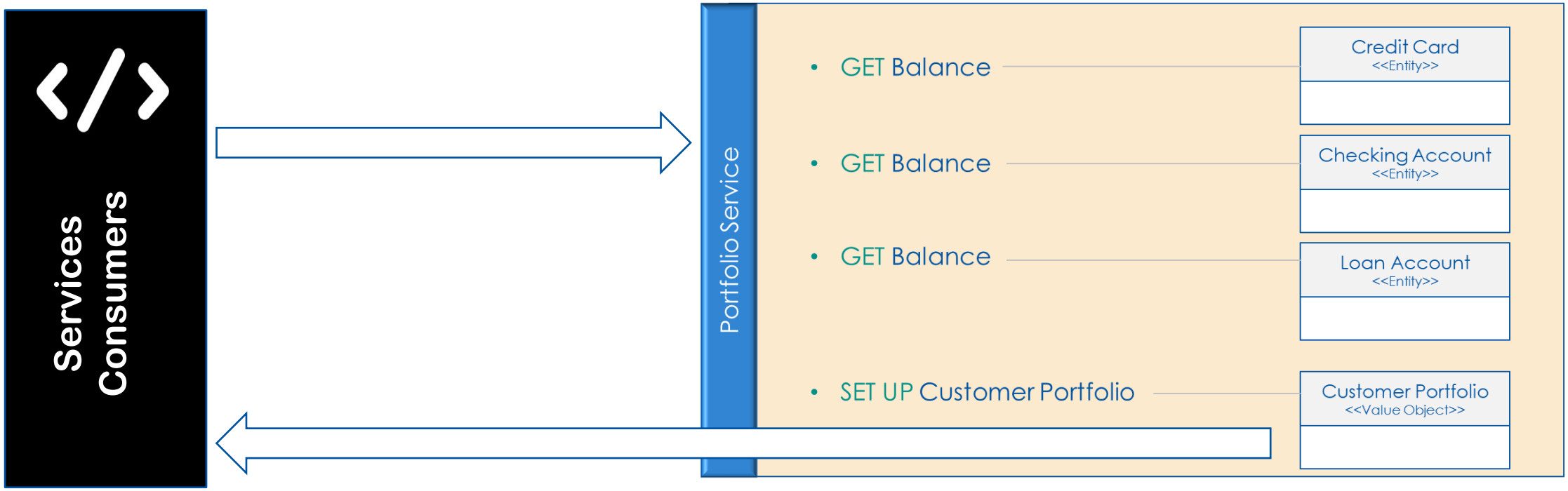
Services

Implemented as domain model aware independent components



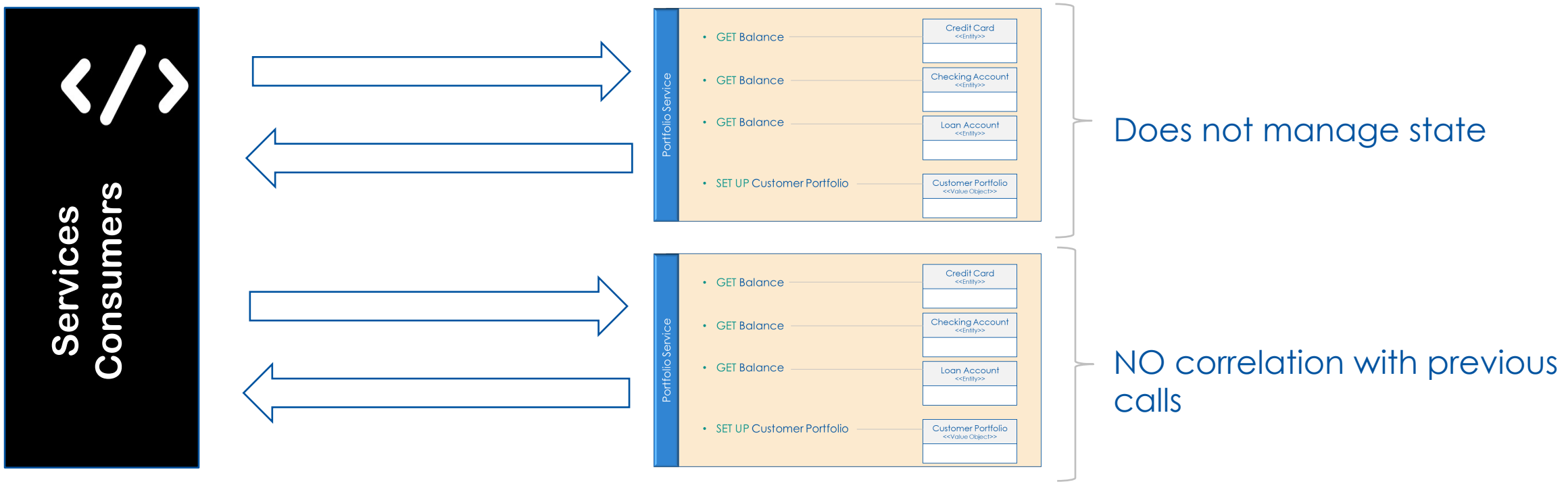
Blackbox

Client of service is unaware of the internals of the service



Stateless

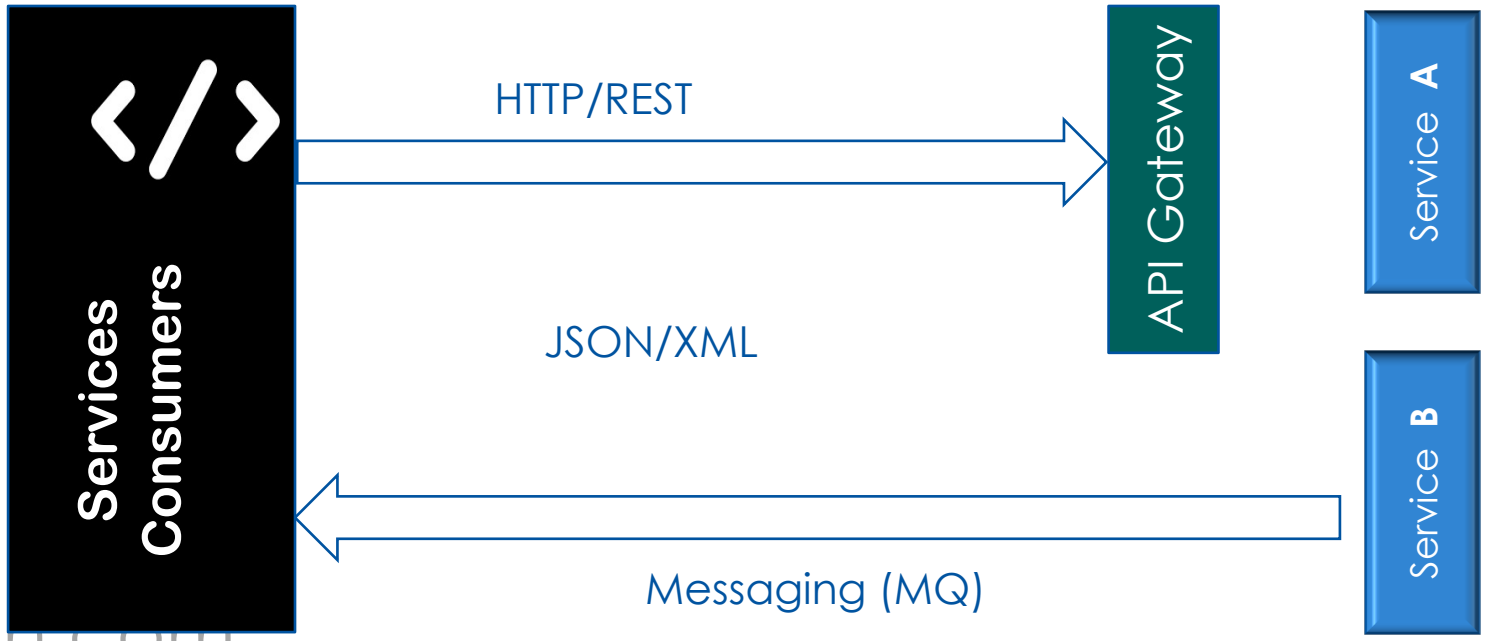
Services are stateless



Accessibility

Services are made available over the network

- Technology agnostic communication protocol



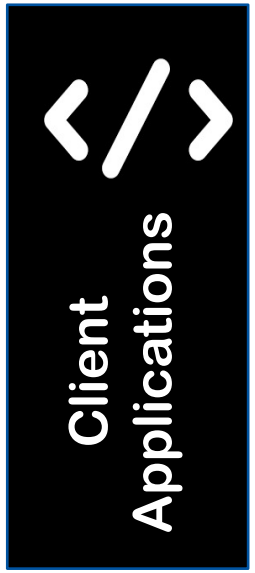
Creation & Management

Services created and managed as independent deployable units

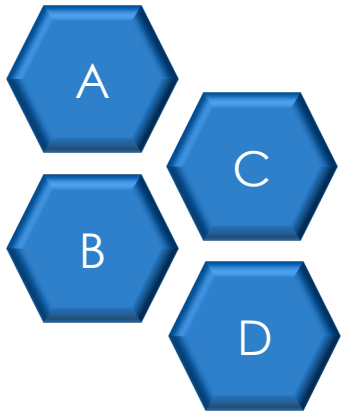
- Autonomous development
- Independent deployment
- Independent scaling

Services Consumer

Client applications and other microservices



- Web apps
- Mobile apps
- API
- 3rd Party integrations



- Dependent MS

Quick Review

Open Host Service (OHS)

Upstream BC exposes common services

Published Language (PL)

Common Language created managed by the team for OHS