Domain Driven Design

One of the biggest challenges of microservices is to define the boundaries of individual services.

The general rule is that a service should do "one thing" — but putting that rule into practice requires careful thought.

Microservices should be designed around business capabilities, not horizontal layers such as data access or messaging. In addition, they should have loose coupling and high functional cohesion.

Microservices are loosely coupled if you can change one service without requiring other services to be updated at the same time

A microservice is cohesive if it has a single, well-defined purpose, such as managing user accounts or tracking delivery history.

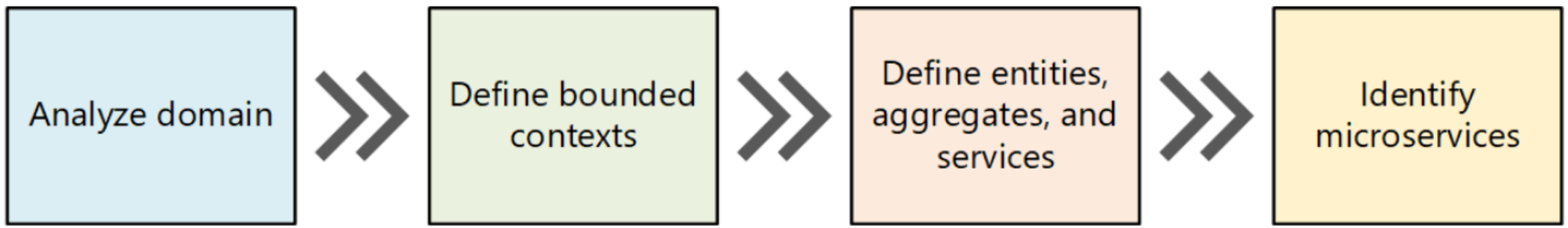
A service should encapsulate domain knowledge and abstract that knowledge from clients.

For example, a client should be able to schedule a drone without knowing the details of the scheduling algorithm or how the drone fleet is managed.

Domain-driven design (DDD) provides a framework that can get you most of the way to a set of well-designed microservices. DDD has two distinct phases, strategic and tactical.

In strategic DDD, you are defining the large-scale structure of the system. Strategic DDD helps to ensure that your architecture remains focused on business capabilities.

Tactical DDD provides a set of design patterns that you can use to create the domain model. These patterns include entities, aggregates, and domain services. These tactical patterns will help you to design microservices that are both loosely coupled and cohesive.


Analyze domain
Define bounded contexts
Define entities, aggregates, and services
Identify microservices

Domain Driven Design

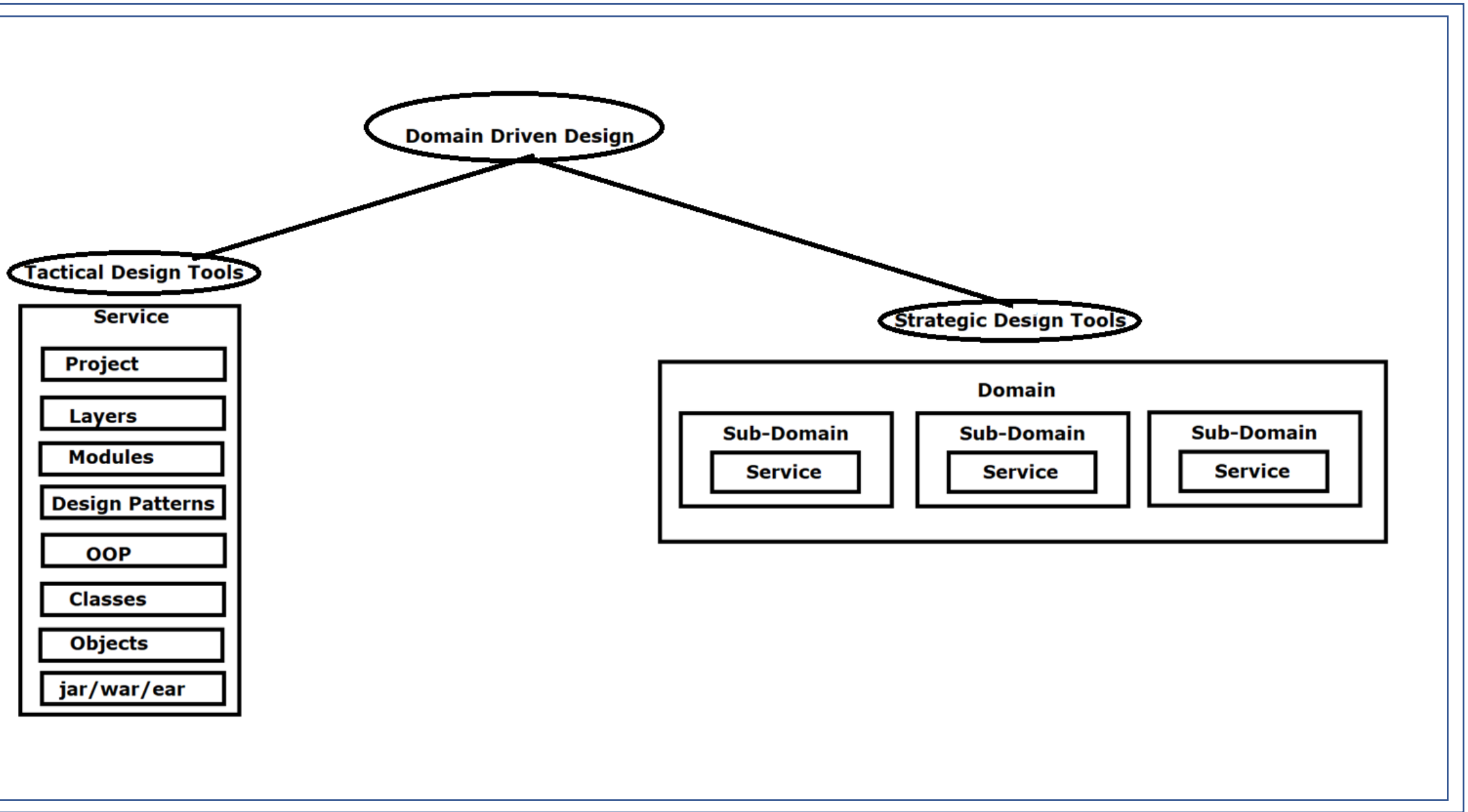
- It is a way of looking the software from top to down.
- When we are **developing a software our focus should** not **be** primarily on technology, it should be primarily **on business** or whatever activity we are trying to assist with the software, the domain.
- Specifically we approach that by trying to develop models of that domain and make our software conformed to that.

For most software projects, the primary focus should be on the domain and domain logic

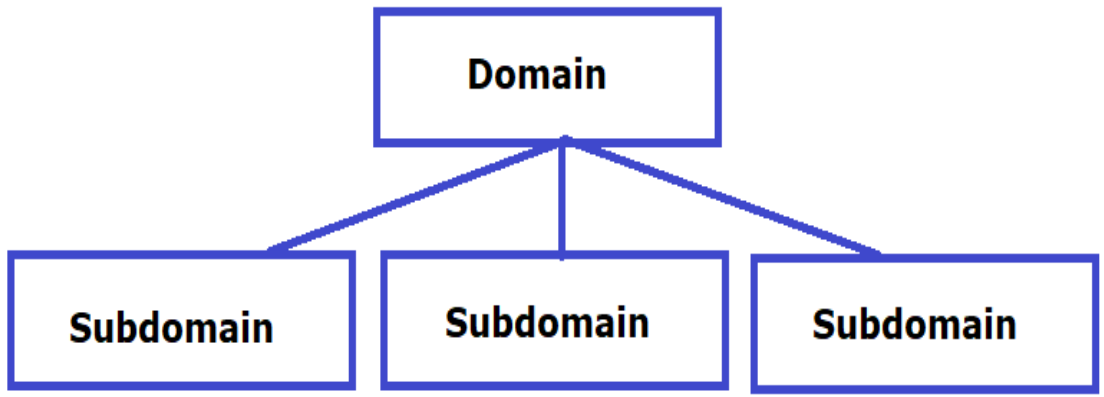
Domain-driven design is an approach to software development for complex needs by connecting the implementation to an evolving model.

Domain-driven design is predicated on the following goals:

- placing the project's primary **focus on the core domain and domain logic**;
- basing complex **designs on a model of the domain**;
- initiating a creative **collaboration between technical and domain experts** to iteratively refine a conceptual model that addresses particular domain problems.


Domain Driven Design
Tactical Design Tools
Service
Project
Layers
Modules
Design Patterns
OOP
Classes
Objects
jar/war/ear
Strategic Design Tools
Domain
Sub-Domain
Service
Sub-Domain
Service
Sub-Domain
Service

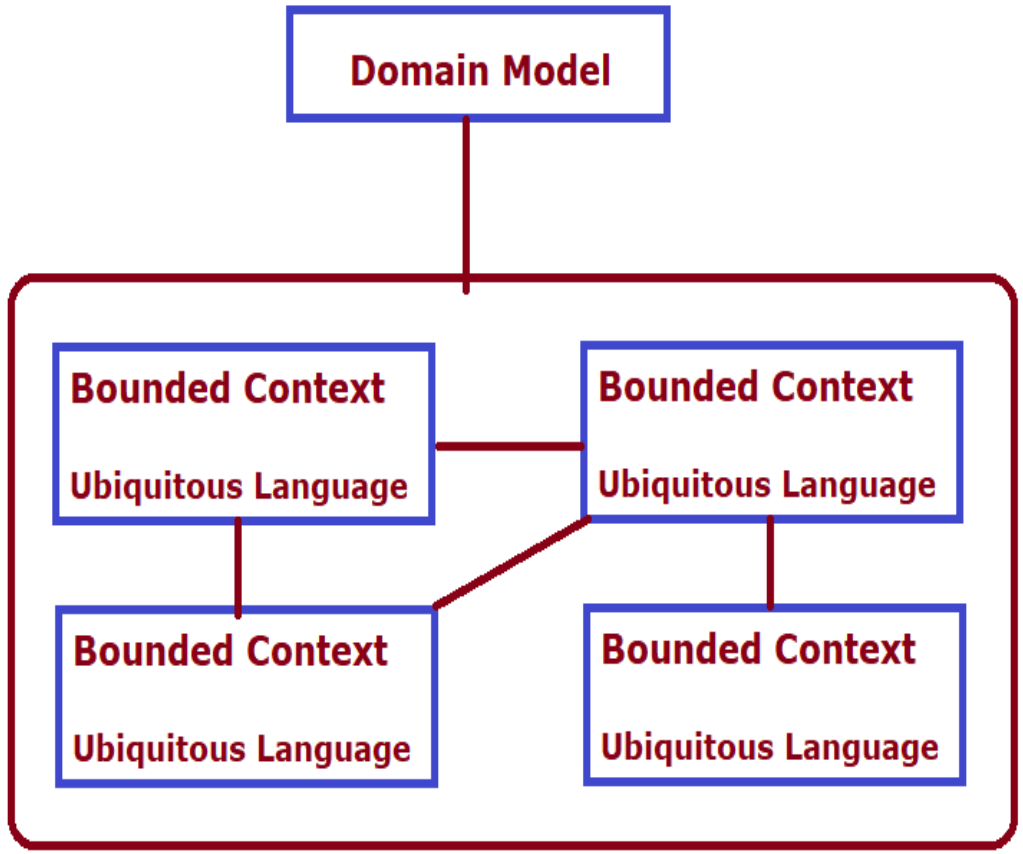
Problem Space



Types of Subdomain:

Core Sub-Domain
Supporting Sub-Domain
Generic Subdomains

Solution Space



Object Oriented Design : Think in terms of Objects

Strategic Design : Think in terms of Contexts

Core Domain

The core domain is so **critical and fundamental to the business** that it gives you a competitive advantage and is a foundational concept behind the business.

Ex : Order processing, loan processing, payment service etc.,

Supporting Subdomain

These types of pieces are also necessary because they **help perform ancillary or, well, supporting functions** related directly to what the business does.

Ex : system manages all the banners, landing pages, and other creative materials produced by our design studio

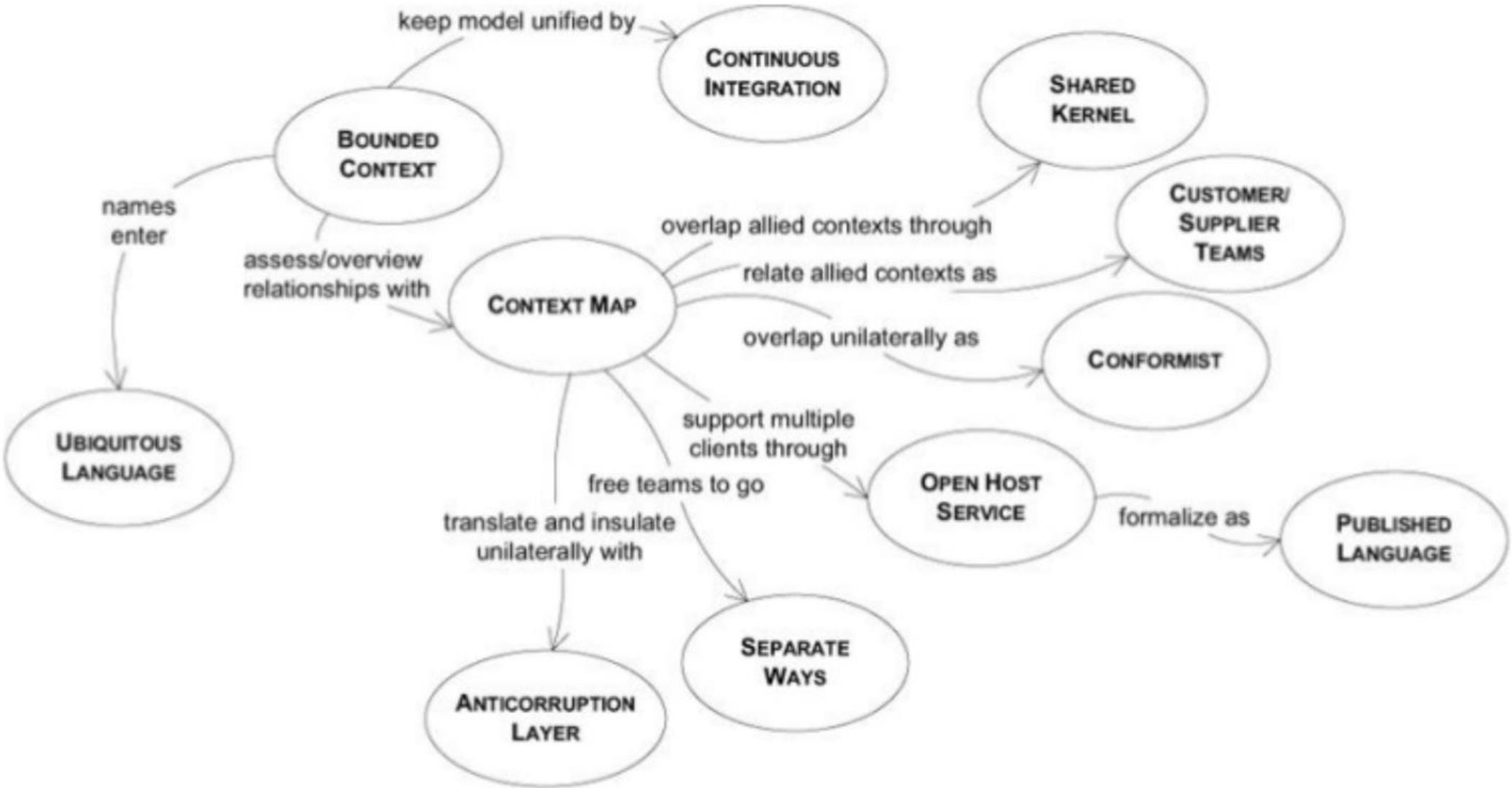
Generic Subdomains

Generic subdomains are those things that all companies do the same way. Those **are considered "solved" problems**.

Ex : identity and access management, ERP module , Invoicing & reporting

Strategic Design

- It would be preferable to **have a single, unified model**.
- It is **a set of principles for maintaining model integrity.**



Strategic Design Patterns

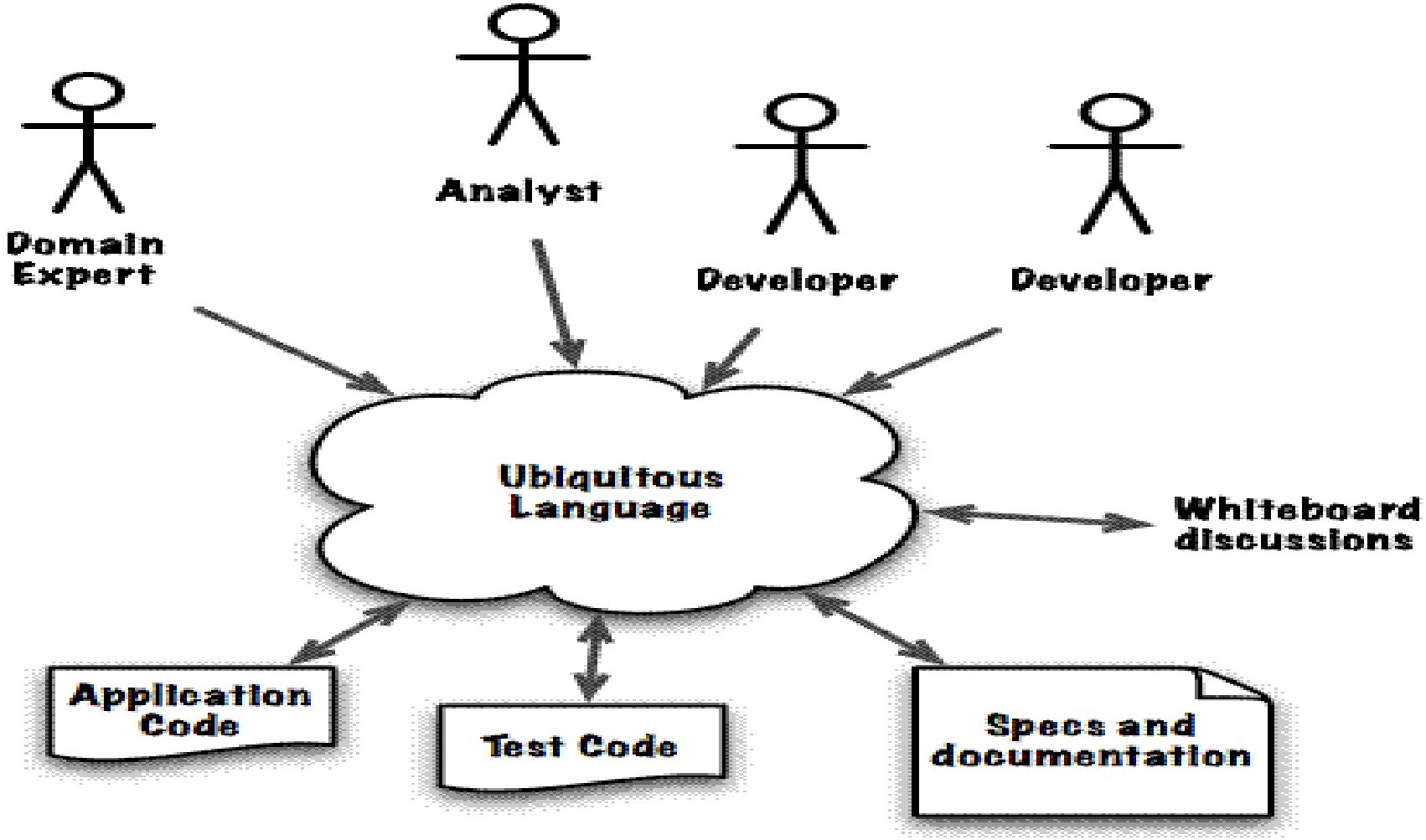
The Ubiquitous Language

The Need for a Common Language

- **Developer think** in terms of **inheritance, composition** etc. And they talk like that all the time.

- But the **Domain Experts** usually know nothing about any of that. They have no idea about software libraries, frameworks, persistence, in many case not even databases. They **know about their specific area of expertise.**

- It is absolutely necessary to **develop a model of the domain** by having the **software specialists work with the domain experts;**



The Ubiquitous Language should be the **only language** used to express a model. Everybody in the team should be able to agree on every specific term without ambiguities and no translation should be needed

Creating the Ubiquitous Language

How can we start building a language?

Here is a **hypothetical dialog between a software developer and a domain expert** in the **inventory management & Reorder Point Management Project**

Bounded Context

In DDD, a **subdomain in the problem space** is mapped to a **bounded context in the solution space**.

A bounded context is an area of the application that requires its **own ubiquitous language** and its own architecture. Or, put another way, a bounded context is a boundary within which the ubiquitous language is consistent. **A bounded context** can have **relationships to other bounded contexts**.

Ubiquitous Language is a concept from Domain-Driven Design (DDD) that promotes a shared and consistent language between business stakeholders, developers, and all team members involved in a project. This helps ensure everyone has a common understanding of the domain and its intricacies. Here's a simplified conversation in the context of a sales domain, adhering to Ubiquitous Language principles:

Salesperson: Hey, I've been talking to the client, and they're interested in our product.

Product Manager: That's great news! What features are they specifically looking for?

Salesperson: They're really focused on the inventory management and reporting capabilities. They want real-time updates on stock levels.

Developer: So, you mean they're interested in the Inventory Control subdomain, right?

Salesperson: Yes, that's correct. They also mentioned wanting to set up automated reorder points.

Business Analyst: Sounds like they need a Reorder Point Management module. How frequently do they want the system to check stock levels?

Salesperson: They're looking for daily checks with email alerts when inventory drops below the reorder point.

Designer: Got it, so we'll need to design an email notification system. What kind of reports are they expecting?

Salesperson: They want monthly sales reports and real-time stock status dashboards.

Data Analyst: We'll need to create a Sales Reporting subdomain for those monthly sales reports. For real-time dashboards, we can integrate with the Inventory Control subdomain.

Product Manager: It seems like the client is mainly interested in the Inventory Control and Sales Reporting subdomains. Let's prioritize those for development.

In this conversation, the team members use a common language to discuss the sales domain, referring to specific subdomains, modules, and features.

This ensures that everyone understands the client's requirements and can work together effectively to deliver a solution that meets the client's needs.

This shared language is a fundamental aspect of Domain-Driven Design and helps prevent misunderstandings and miscommunication within the team.

The outcome of the conversation in the given context would typically be as follows:

Understanding Client Requirements: The Salesperson communicates the client's interest in the project, focusing on inventory management and reporting.

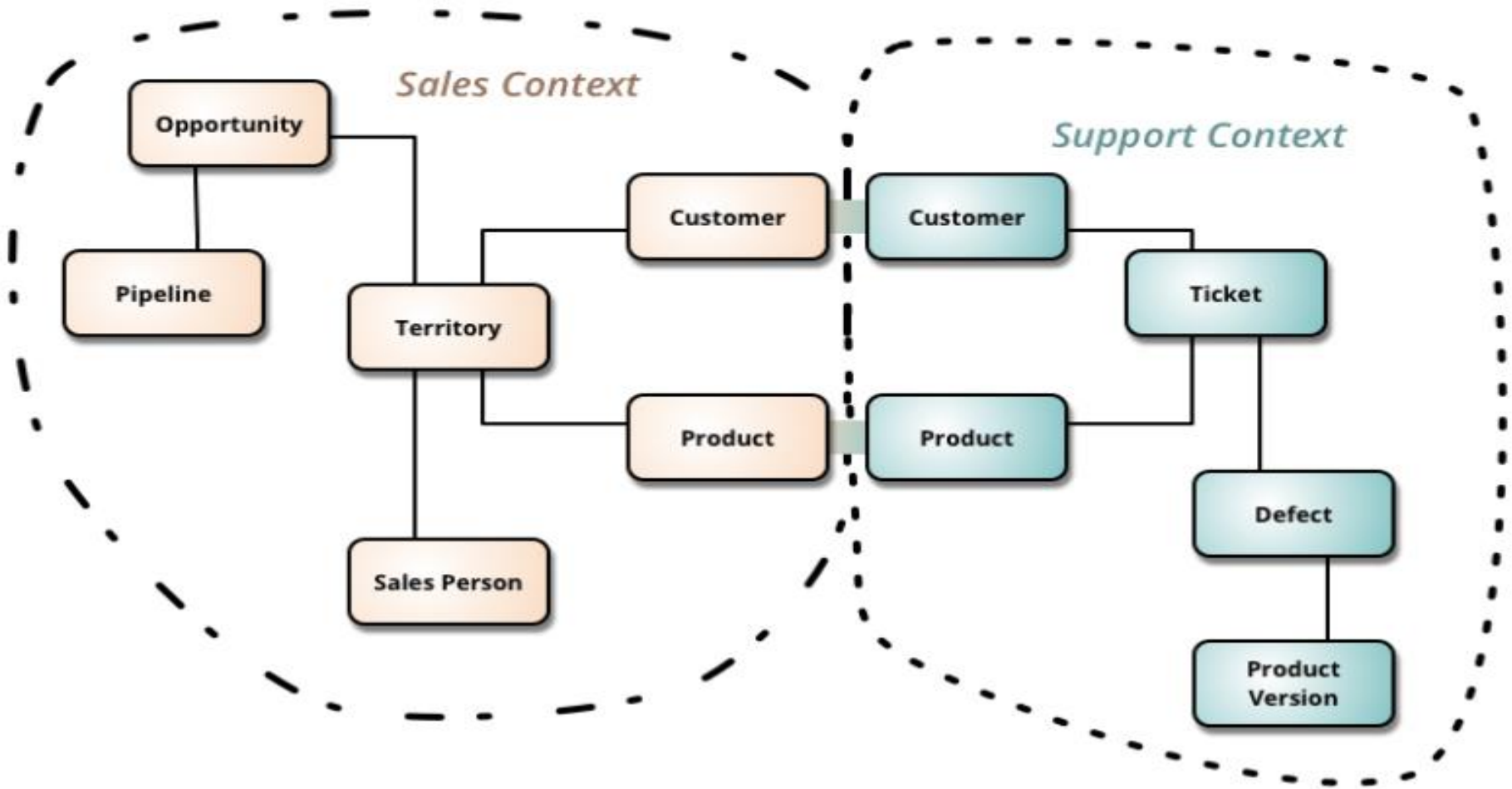
Identifying Specific Needs: The team, including the Product Manager, Developer, Business Analyst, Designer, and Data Analyst, engages to gather more specific information about the client's requirements.

Defining Subdomains: Through the conversation, it becomes clear that there are specific subdomains within the project, such as "Inventory Control" and "Sales Reporting."

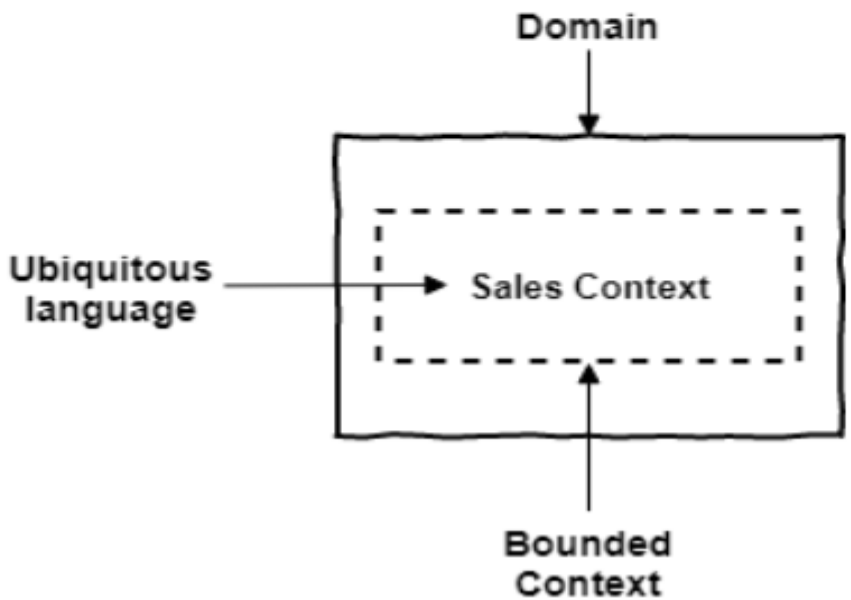
Prioritization: The team discusses the client's priorities and decides to focus on developing features related to the "Inventory Control" and "Sales Reporting" subdomains.

The outcome of this conversation is a shared understanding among team members regarding the client's needs, the specific focus areas of the project, and the initial steps to be taken, which may include further detailed requirements gathering, project planning, and development efforts. This shared understanding is crucial for effective collaboration and successful project delivery in the context of Domain-Driven Design and Ubiquitous Language.

Bounded Context is a central pattern in Domain-Driven Design. It is the focus of DDD's strategic design section which is all about dealing with large models and teams. DDD deals with large models by dividing them into different Bounded Contexts and being explicit about their interrelationships.



Ubiquitous Language is modeled within a Limited context, where the terms and concepts of the business domain are identified, and there should be no ambiguity.

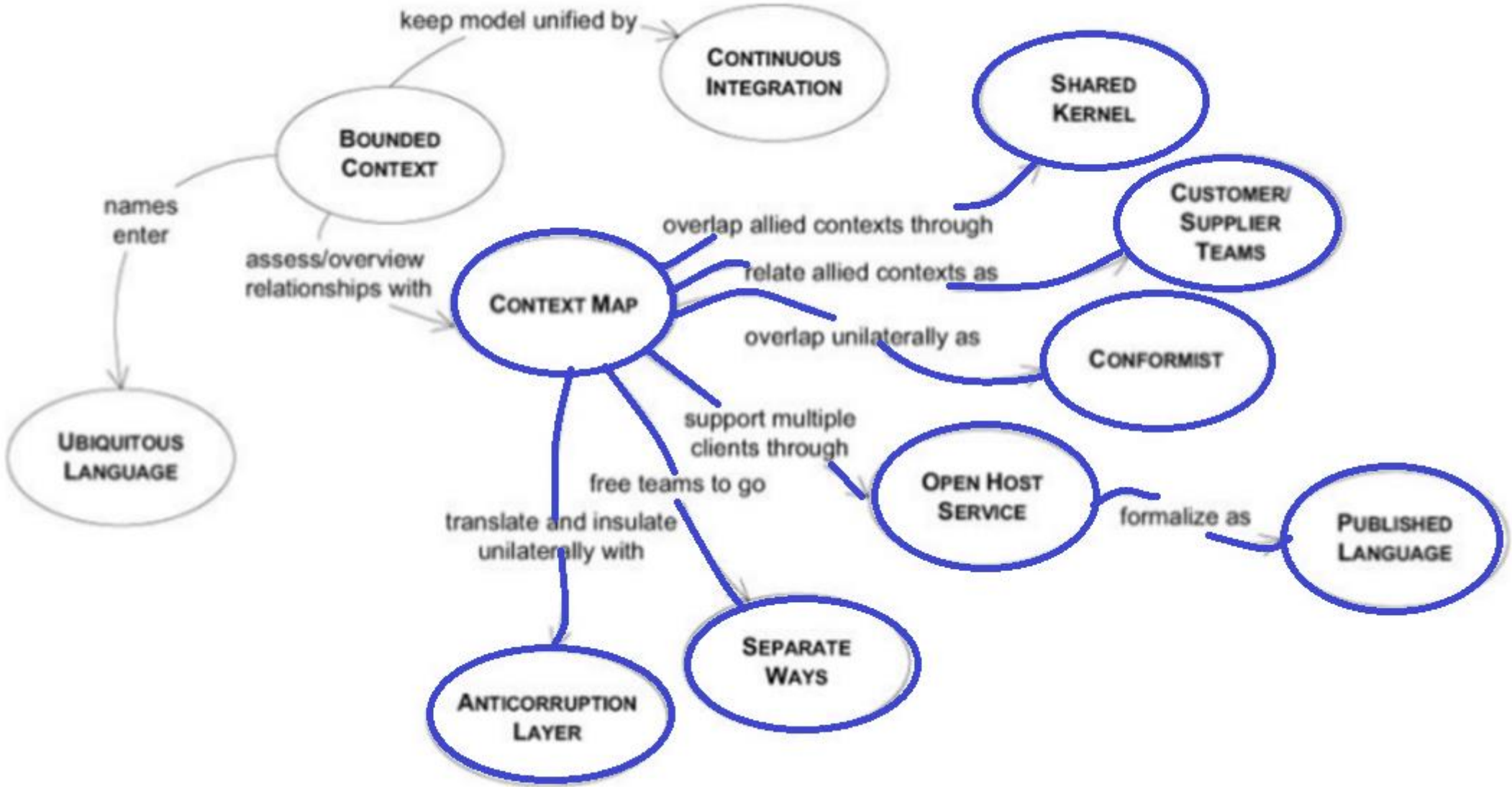


Bounded Context Communication: Any design has two common parts, abstraction of the data model and communicates with other parts of the system.

By Bounded context we separate the data model in a simple term abstracting the commonalities in the business but **How one Bounded context communicate with others?**

Here the concept of **Context Map** stepped in, Using Context map we can discover How one Context depends on other Bounded Context:

- Like are two Context has strong dependencies.
- or one domain sends a confirmation message to another domain(Conformist)
- or may use a shared kernel/Shared model,


keep model unified by
CONTINUOUS INTEGRATION
BOUNDED CONTEXT
names enter
UBIQUITOUS LANGUAGE
assess/overview relationships with
CONTEXT MAP
overlap allied contexts through
SHARED KERNEL
relate allied contexts as
CUSTOMER/ SUPPLIER TEAMS
overlap unilaterally as
CONFORMIST
support multiple clients through
OPEN HOST SERVICE
formalize as
PUBLISHED LANGUAGE
free teams to go
translate and insulate unilaterally with
ANTICORRUPTION LAYER
SEPARATE WAYS

Domain : Online Student management system

Online Student management system is where:

- ❑ Student can register to the site & choose course
- ❑ Pay the Course fee
- ❑ Student will be tagged to a batch
- ❑ Teacher & Student is notified about the batch details

We have to identify the bounded context of the different domain related to this business logic.

There are 4 bounded contexts Registration, Payments, Scheduler, and Notification.

1. **Registration process:** Which takes care of Registration of Student.

2. **Payment System:** Which will process the Course fee and publish online payment status.

3. **Batch Scheduling:** Upon confirmation of payment, this function checks the Teacher availability, batch availability and based on that create a batch and assign the candidates or update an existing batch with the candidate.

4. **Notification System:** It will notify Teacher and Student about the timings and slot information.

Suppose we have three bounded contexts: OrderContext, PaymentContext, ShipmentContext and these context have Order entity.

In Domain-Driven Design (DDD), whether we need a separate "Order" entity class for each bounded context or a shared context depends on the specific requirements and design of your system. The decision should be based on the level of divergence in the definitions and behaviour of the "Order" entity within each bounded context.

Here are a few considerations to help you decide:

1. Shared Order Entity:

Use a shared "Order" entity if the core properties and behaviors of the order remain consistent across all three bounded contexts (Order, Payment, and Shipment).

This approach simplifies the overall design by having a single definition of the "Order" entity that can be used consistently across contexts.
Any variations or context-specific behaviour can be handled through context-specific services or policies.

2. Separate Order Entities:

Consider separate "Order" entities if the definitions or behaviors of an order significantly differ across contexts.
This separation may be necessary when each context has its own interpretation of what an order represents, and this cannot be effectively addressed through shared properties and behavior.

It provides greater flexibility for modelling context-specific requirements and can avoid complexities that may arise from trying to accommodate all contexts in a single shared entity.

In some cases, a **hybrid approach** might also be applicable. You could have a shared "Order" entity that includes core properties and behaviors common to all contexts, while each context extends this shared entity to accommodate context-specific properties and behaviors.

Shared Kernel :

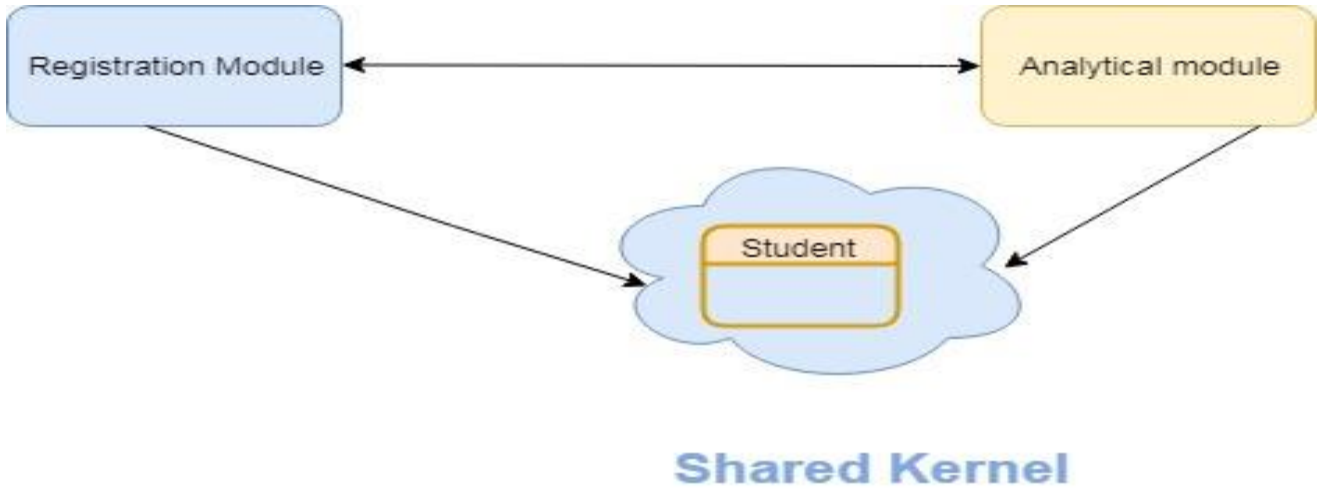
Shared kernel talks about a partnership relation where two or many teams shared common data model/ value object;

It reduces the code duplication as different context use that common model, but that common model/value object is very sensitive;

Any changes major/minor should be agreed upon all the parties unless it would break other parties code.

Example : Analytical module is used to analysis which courses are most chosen by the students, which students are chosen more than five courses etc, so that module works with Student model, course model.

Analytics module can share Registration modules student model, and they also agreed upon any changes on student model.



Customer/ Supplier:

Generally this is the common relationship between two contexts, where a context consumers or depend on data from another context, the context which produces data marked as upstream and the context which consumes data called downstream.

- **Upstream as the leader:**

In this type of relationship, the upstream team is in a **commendable position**, that team does not care about the downstream team, as they producing the data and downstream team need to change their model based on the data structure produces by upstream.

Example : Payment application decide what information in which structure they provide and Notification module consumes that data structure.

Downstream as the Leader:

In some cases, the relationship is revert although upstream is produces data, it must have to follow the rule, the data structure for downstream, in this scenario downstream is in a **commendable position**.

Example : Student registration system we need to submit Form 16 to government as a tax payee so our payment module has to submit form 16 data to Government exposed API but government API has certain rules and data structure for submitting form 16 data, so although government API is downstream it has total control, our Payment module should communicate with down steam in such a way so it can fulfil downstream rules.

Downstream as the Leader

The customer-supplier relation works best when both the parties upstream and downstream are aligned with the work both party agreed upon the interfaces and change in the structure.

In case of any changes in the contract both parties will do a discussion synchronize their priority backlogs and agreed upon the changes, If one party does nor care upon another party then every time contract will be broken and it is tough to maintain a customer-supplier relationship.

Conformist:

Sometimes, there is a relation between two parties in such a way that downstream team always dependent on an upstream team and they **can't do a mutual agreement** with upstream about requirements.

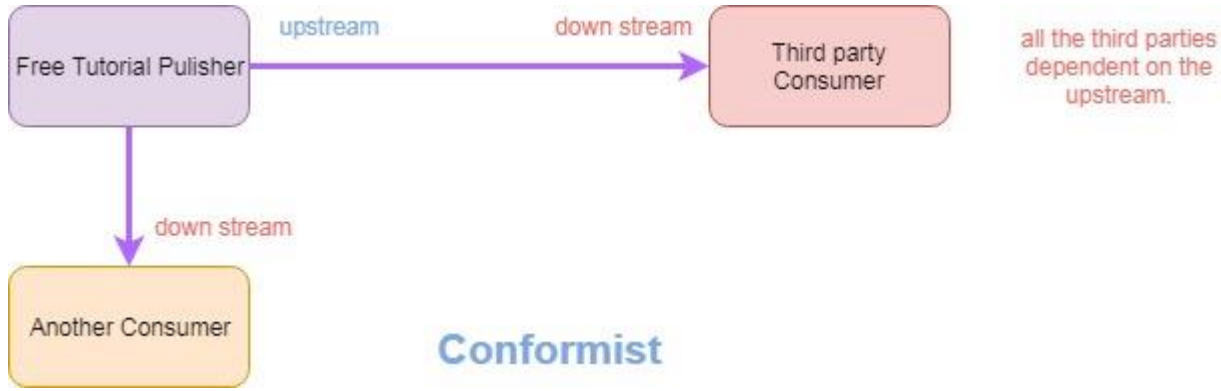
The **upstream** is not aligned with downstream and does not care they are **free to change there published endpoint** or contract any time and not taking any request from downstream.

It is happens when **Upstream team is an external system** or under a different management hierarchy , and many downstream systems are registered with it so it can't give a priority to any downstream, rather then all downstream system must be aligned with upstream contact and data structures.

Example : Free tutorial module is used by all students or other applications can embed them in their application.

Here Free tutorial module acts as upstream and independent of any other third party app who consumes our free tutorials.

We can’t give any priority to them and we don't have any contract with them if we change the contracts or data structures it is other third parties duty to change their application accordingly to consume our free tutorials. Other parties are acting as a conformist.



Anti Corruption Layer:

When two system interacts if we consume the **data directly from upstream** we **pollute** our **downstream system** as upstream data structure leak through the downstream so if the upstream become polluted our downstream too as it imitates the upstream data while consuming.

So it is a good idea while consume data from third party or from a legacy application always use a translation layer where the **upstream data translate to downstream** data structure before fed in to downstream.

This will help to resist the data leakage from upstream, if upstream contract changes it does not pollute downstream internal system only Translation layer has to be changed in order to adopt new data structure from upstream and convert it into downstream data structure, **this technique is called Anti-corruption layer**.

Anti-corruption layers save the downstream system from upstream changes.

Example : Notification module can implement an ACL while consuming data from payment module so if payment module data structure changes only ACL layers affected.

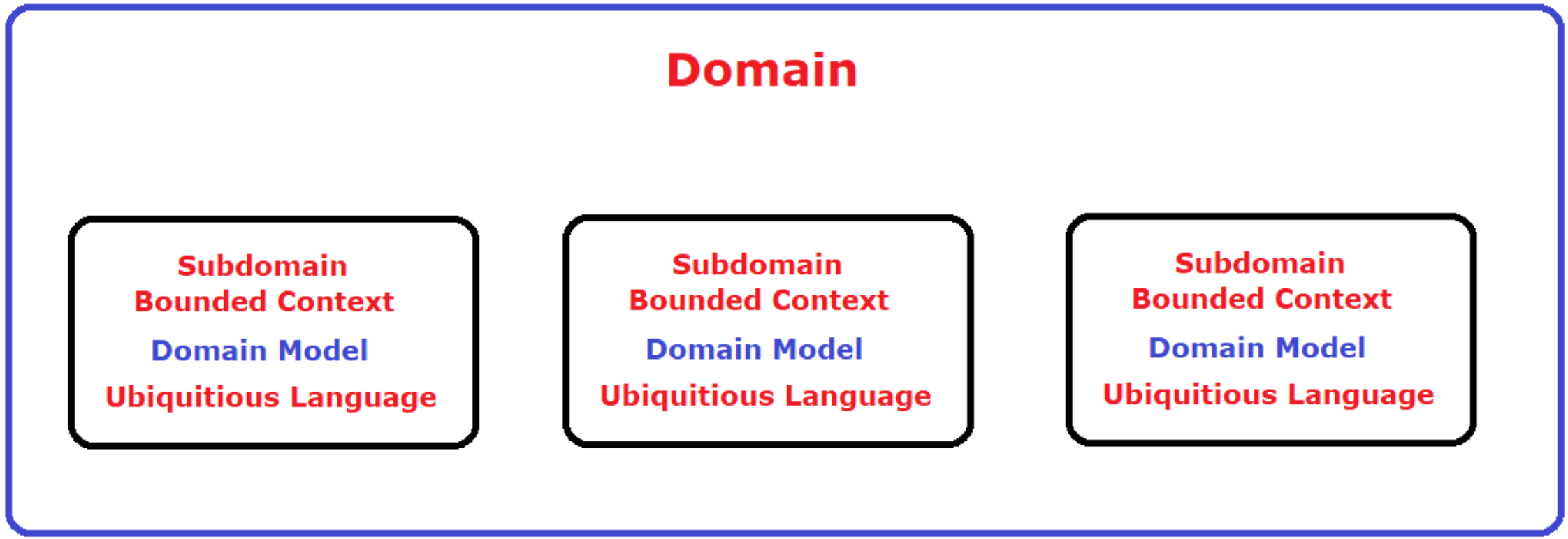
Open Host : In some cases, your Domain API needs to be accessed by many other services like our Free Tutorial Publisher module, Many external or internal domains want to consume this service, so as **Upstream** it should be **hosted** as a service and maintains a protocol and **service contract** like REST and JSON structure so another system can consume the data.

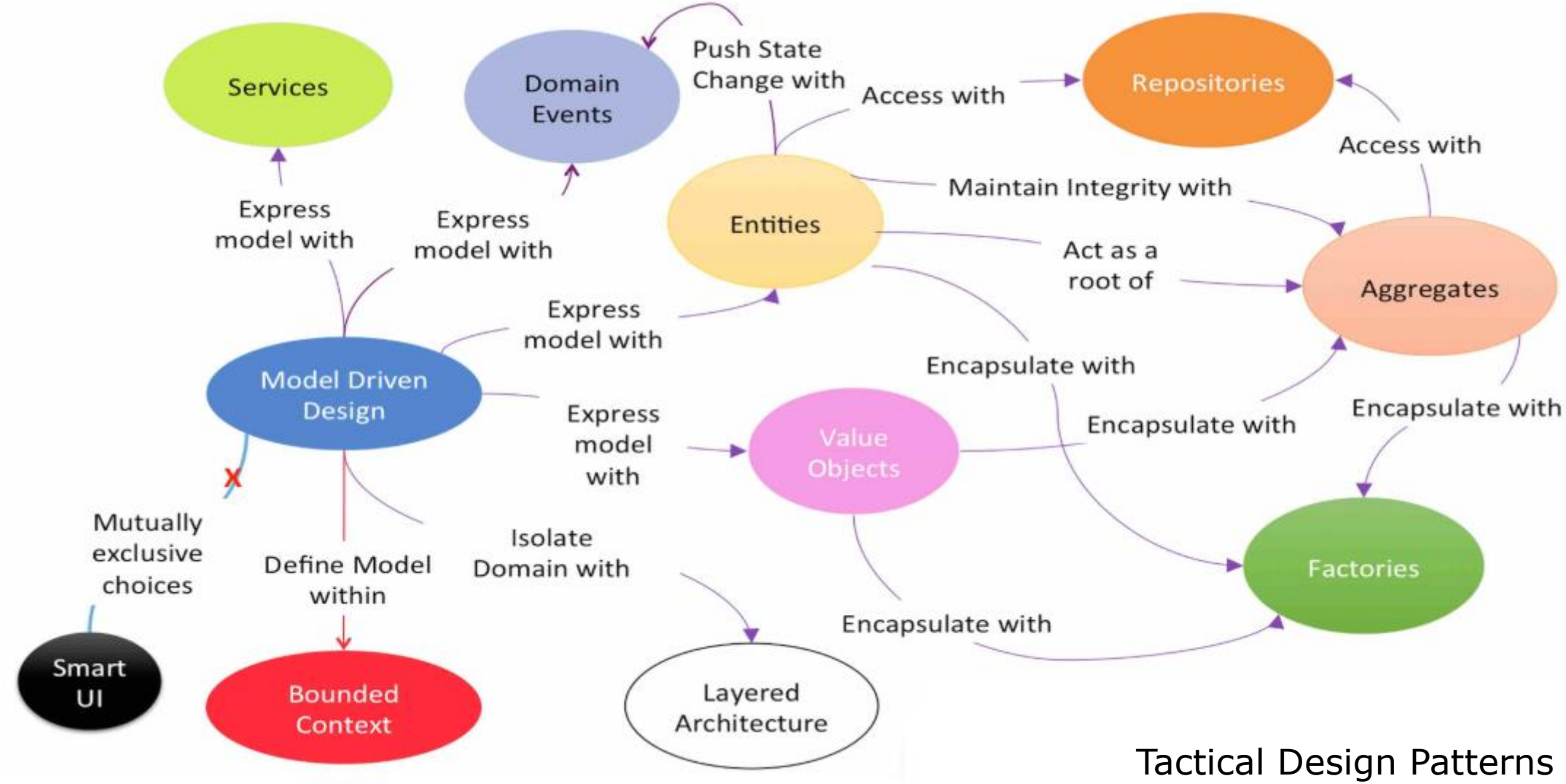
Published Language: Often two or more system receive and send messages among themselves, in that case, a common language will be needed for the transformation of the data from one system to another like XML, JSON we call that structure as Published language.

Tactical design tools

- Tactical design tools are **concerned with implementation details**.
- Generally takes care of components inside a bounded context.
- A de facto standard in development world
- **Tactical patterns are services, entities, repositories, factories etc.,**
- Tactical patterns is **expected to change** during product development.

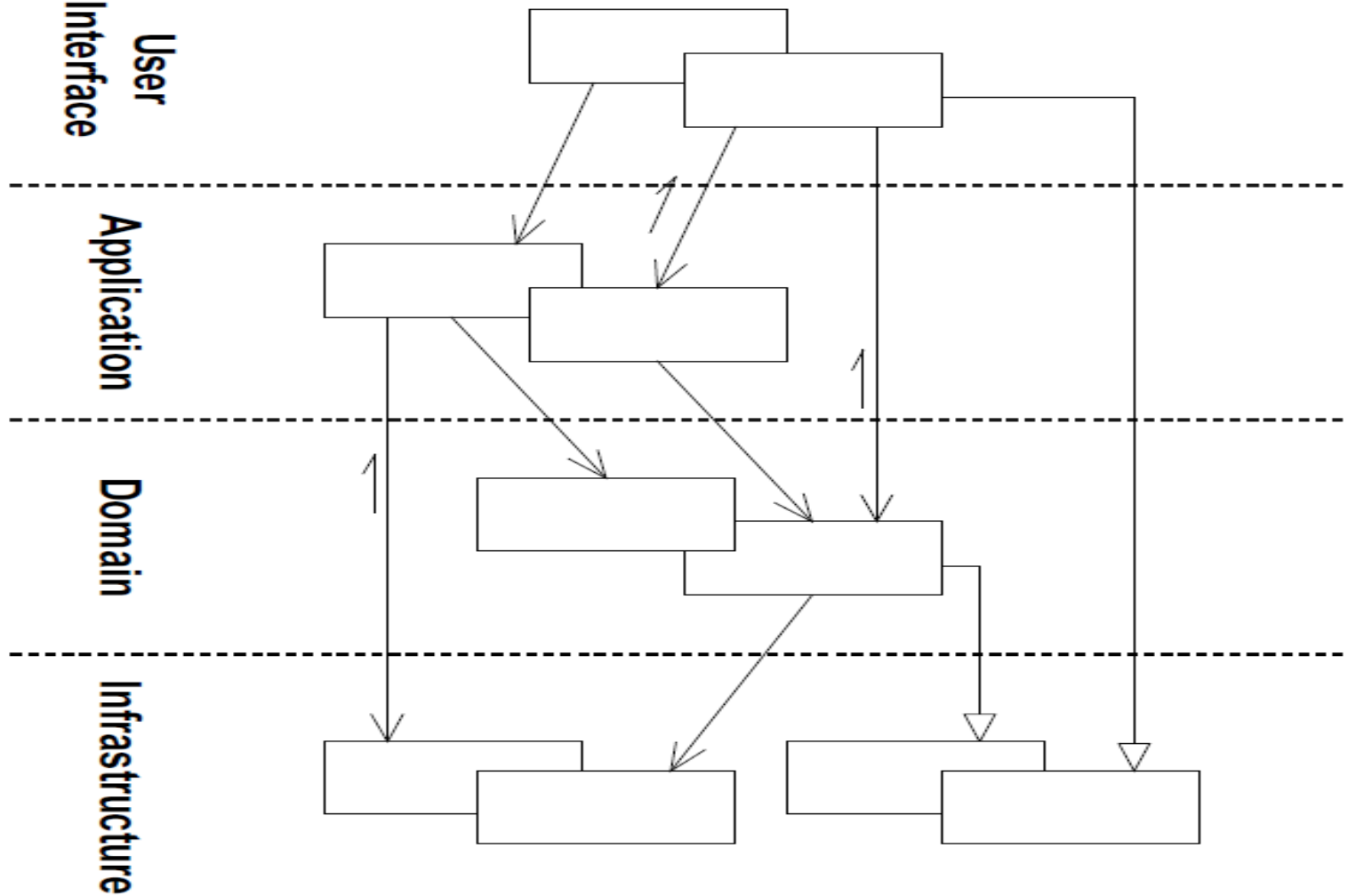
Model Driven Design



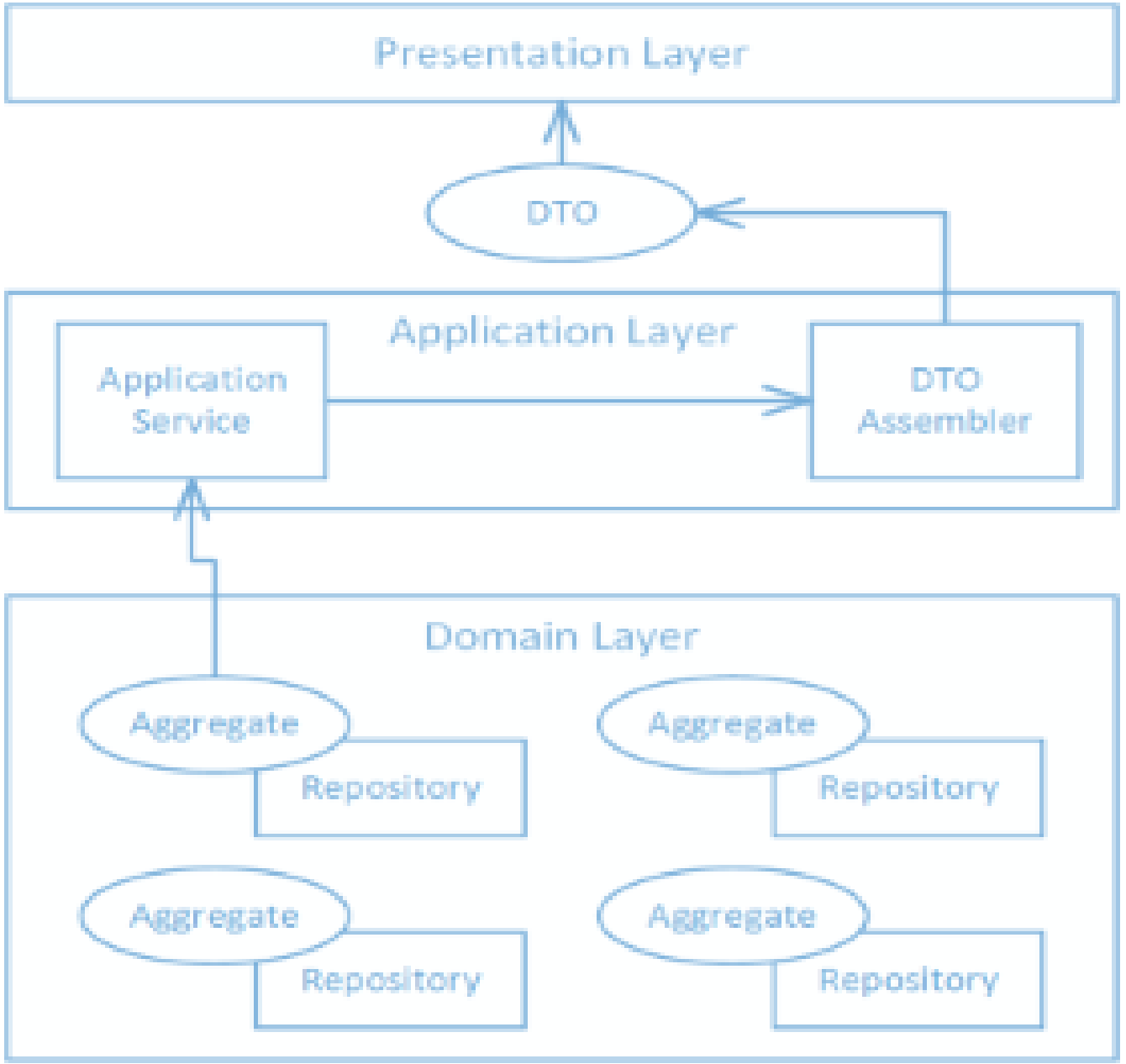


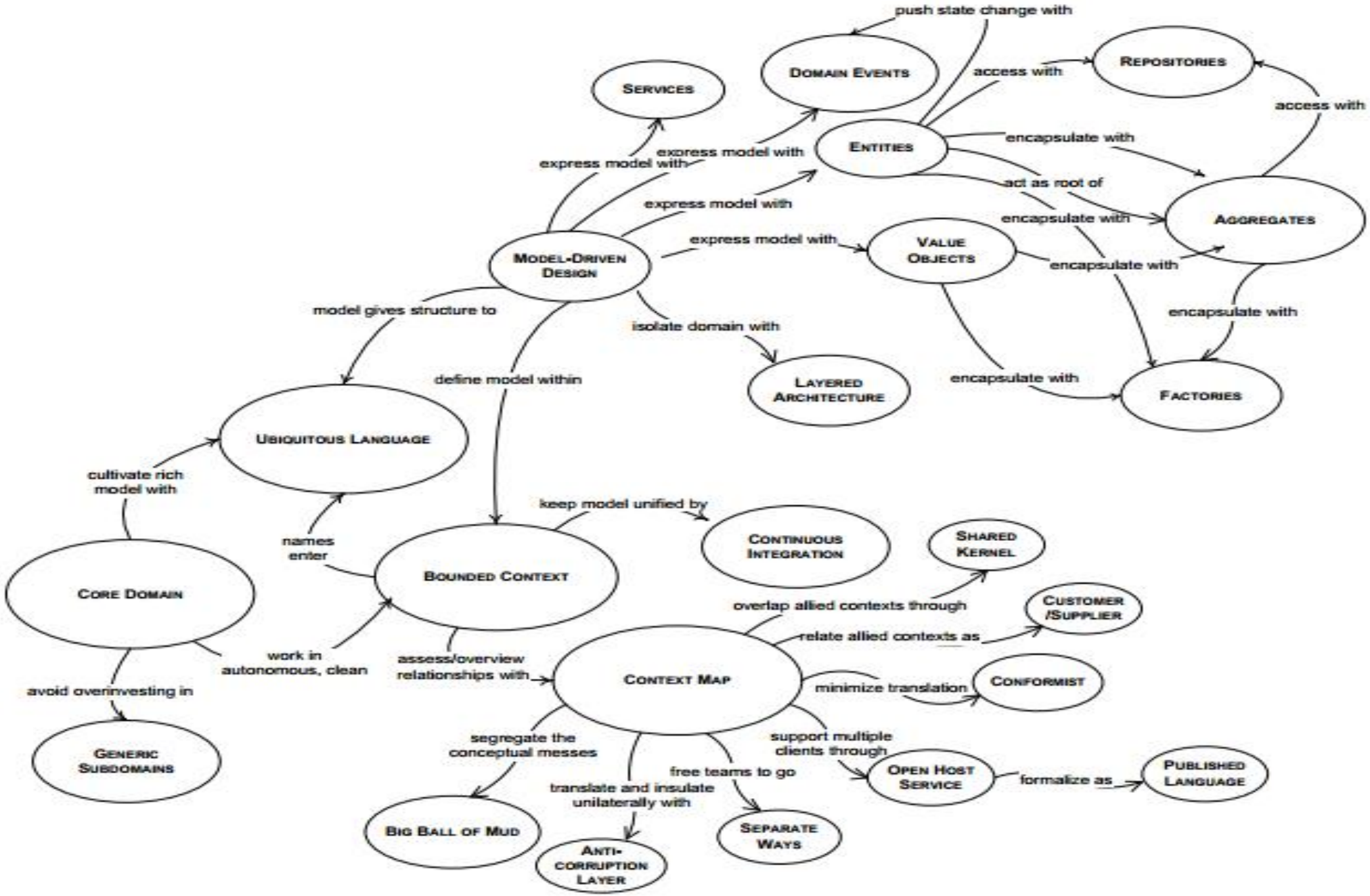
Tactical Design Patterns

Layered Architecture



User Interface **(aka Presentation Layer)**	Responsible for showing information to the user and interpreting the user's commands. The external actor might sometimes be another computer system rather than a human user.
Application Layer	Defines the jobs the software is supposed to do and directs the expressive domain objects to work out problems. The tasks this layer is responsible for are meaningful to the business or necessary for interaction with the application layers of other systems. This layer is kept thin. It does not contain business rules or knowledge, but only coordinates tasks and delegates work to collaborations of domain objects in the next layer down. It does not have state reflecting the business situation, but it can have state that reflects the progress of a task for the user or the program.
Domain Layer **(aka Model Layer)**	Responsible for representing concepts of the business, information about the business situation, and business rules. State that reflects the business situation is controlled and used here, even though the technical details of storing it are delegated to the infrastructure. *This layer is the heart of business software.*
Infrastructure Layer	Provide generic technical capabilities that support the higher layers: message sending for the application, persistence for the domain, drawing widgets for the UI, etc. The infrastructure layer may also support the pattern of interactions between the four layers through an architectural framework.


Presentation Layer
DTO
Application Layer
Application Service
DTO Assembler
Domain Layer
Aggregate
Repository
Aggregate
Repository
Aggregate
Repository
Aggregate
Repository


push state change with
DOMAIN EVENTS
access with
REPOSITORIES
SERVICES
access with
express model with
express model with
ENTITIES
encapsulate with
act as root of
express model with
encapsulate with
AGGREGATES
express model with
MODEL-DRIVEN DESIGN
VALUE OBJECTS
encapsulate with
model gives structure to
isolate domain with
encapsulate with
LAYERED ARCHITECTURE
define model within
encapsulate with
FACTORIES
UBIQUITOUS LANGUAGE
cultivate rich model with
keep model unified by
CONTINUOUS INTEGRATION
SHARED KERNEL
names enter
BOUNDED CONTEXT
CORE DOMAIN
overlap allied contexts through
CUSTOMER /SUPPLIER
relate allied contexts as
work in autonomous, clean
assess/overview relationships with
CONTEXT MAP
minimize translation
CONFORMIST
avoid overinvesting in
support multiple clients through
segregate the conceptual messes
free teams to go
OPEN HOST SERVICE
formalize as
PUBLISHED LANGUAGE
GENERIC SUBDOMAINS
translate and insulate unilaterally with
BIG BALL OF MUD
SEPARATE WAYS
ANTI-CORRUPTION LAYER

During the strategic phase of DDD, you are mapping out the business domain and defining bounded contexts for your domain models.

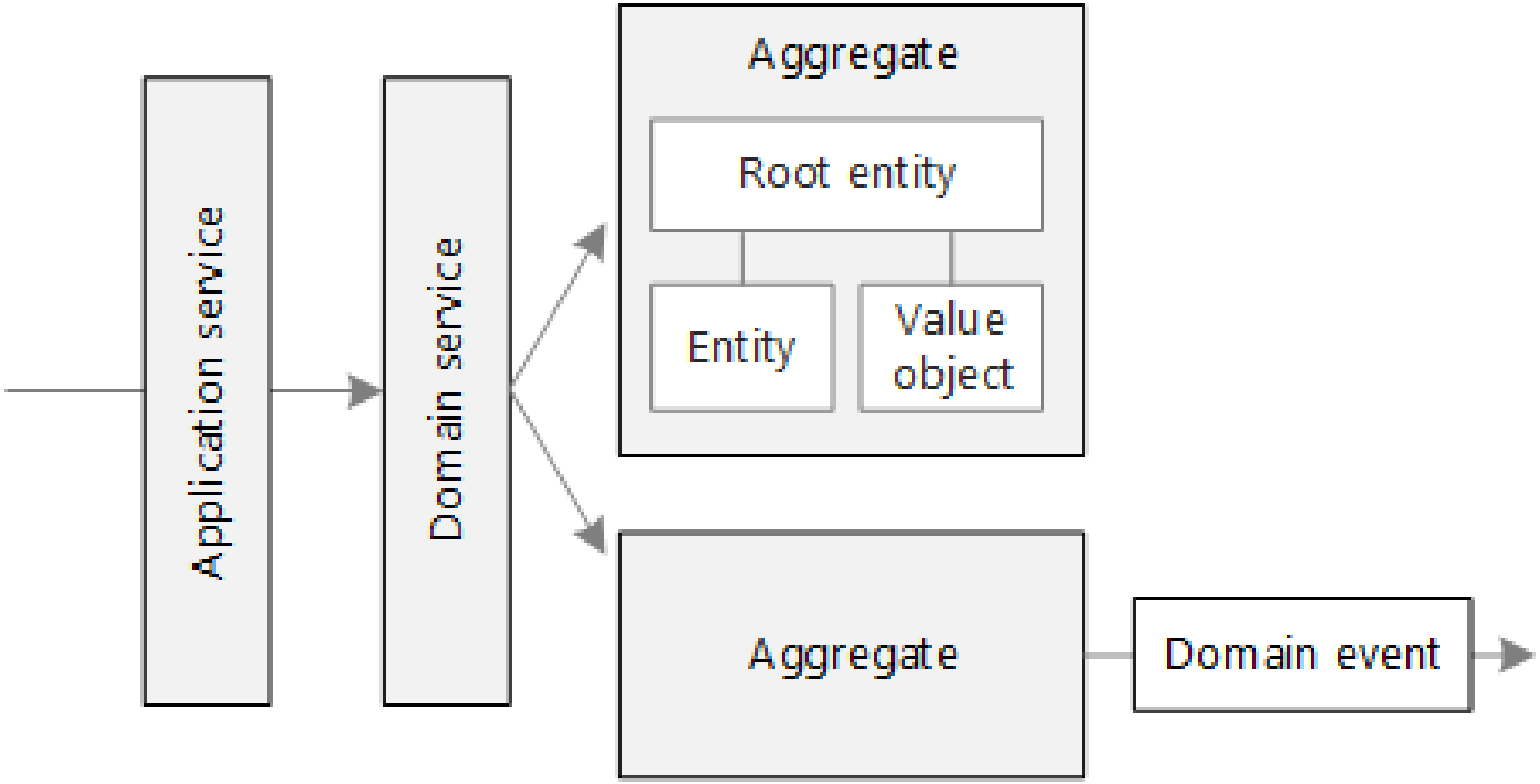
Tactical DDD is when you define your domain models with more precision.

The tactical patterns are applied within a single bounded context.

In a microservices architecture, the entity and aggregate patterns are very important.

Applying these patterns will help us to identify natural boundaries for the services in our application.

As a general principle, a microservice should be no smaller than an aggregate, and no larger than a bounded context.


Application service
Domain service
Aggregate
Root entity
Entity
Value object
Aggregate
Domain event

Domain-driven design leads to entities

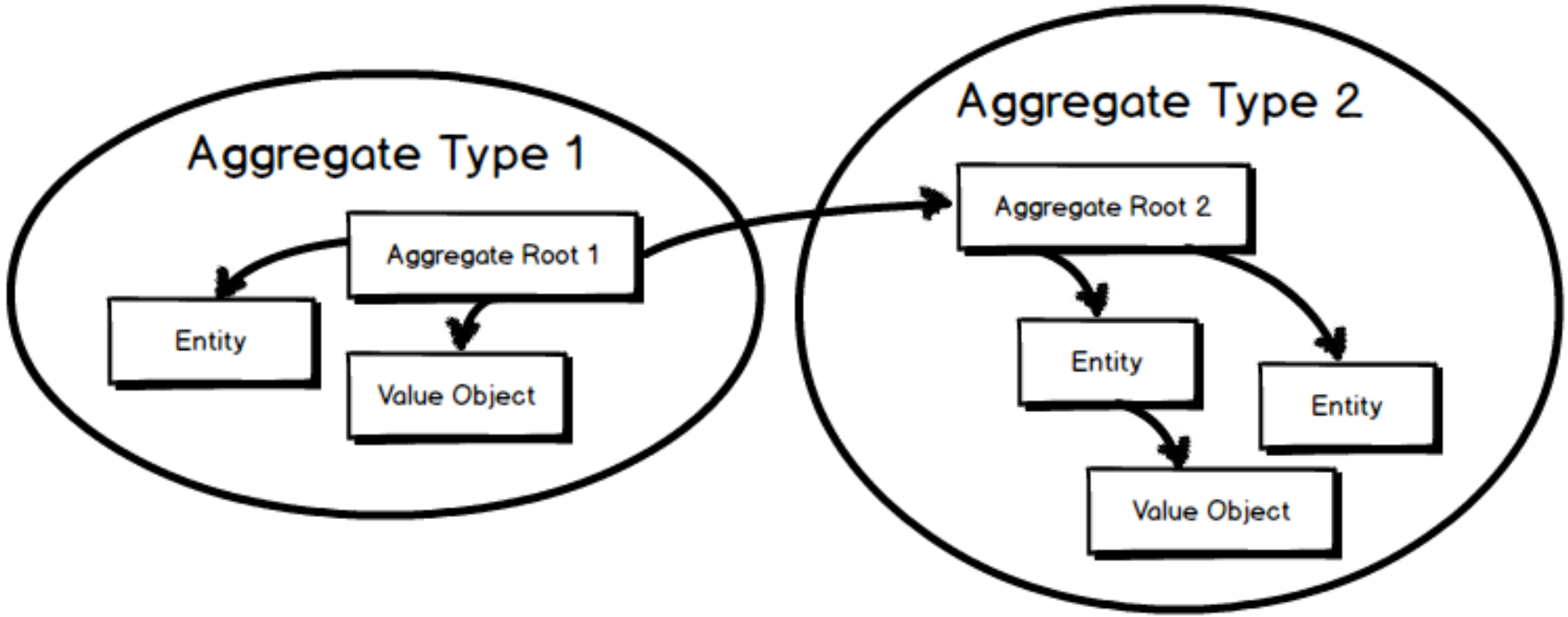
From the approach in domain-driven design, you get the following objects, among others:

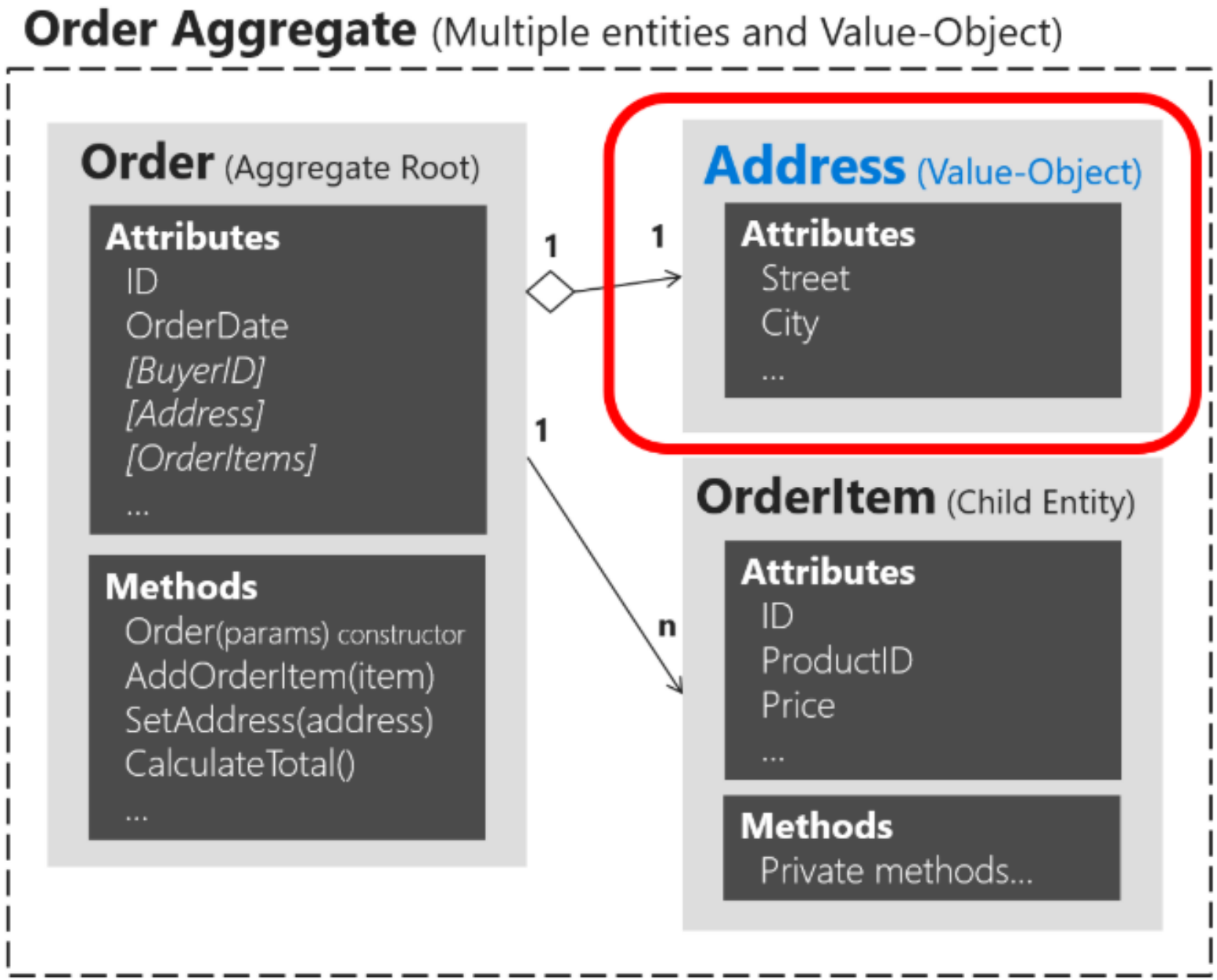
Entity : "An object that is not defined by its attributes, but rather by a thread of continuity and its identity."

Value Objects : "An object that contains attributes but has no conceptual identity. They should be treated as immutable."

Aggregate : "A collection of objects that are bound together by a root entity, otherwise known as an aggregate root. The aggregate root guarantees the consistency of changes being made within the aggregate by forbidding external objects from holding references to its members."

Repository : "Methods for retrieving domain objects should delegate to a specialized Repository object such that alternative storage implementations may be easily interchanged."


Aggregate Type 1
Aggregate Root 1
Entity
Value Object
Aggregate Type 2
Aggregate Root 2
Entity
Entity
Value Object


Order Aggregate (Multiple entities and Value-Object)
Order (Aggregate Root)
Attributes
ID
OrderDate
[BuyerID]
[Address]
[OrderItems]
...
Methods
Order(params) constructor
AddOrderItem(item)
SetAddress(address)
CalculateTotal()
...
1
1
Address (Value-Object)
Attributes
Street
City
...
1
n
OrderItem (Child Entity)
Attributes
ID
ProductID
Price
...
Methods
Private methods...

A **value object** is a small object that represents a simple entity whose equality is not based on identity: i.e. two value objects are equal when they have the same value, not necessarily being the same object.

Examples of value objects are objects representing an address, amount of money or a date range.

Value objects should be immutable; this is required for the implicit contract that two value objects created equal, should remain equal. It is also useful for value objects to be immutable, as client code cannot put the value object in an invalid state or introduce buggy behaviour after instantiation.

Value objects are among the building blocks of DDD.

```
// entity:
class Person {
    PersonId id; // global identity
    FirstName firstName;
    LastName lastName;
    Address address;
}
// value objects:
class PersonId {
    Long value;
}
class FirstName {
    String value;
}
class LastName {
    String value;
}
class Address {
    String street;
    String streetNo;
    String city;
    String postalCode;
}
```

As the value objects have no identity, we compare them together by simply comparing all the values they contain:

```
// Address
@Override
public boolean equals(Object o) {
    // basic checks and casting cut out for brevity
    return Objects.equals(street, address.street) &&
            Objects.equals(streetNo,
address.streetNo) &&
            Objects.equals(city, address.city) &&
            Objects.equals(postalCode,
address.postalCode);
}
@Override
public int hashCode() {
    return Objects.hash(street, streetNo, city,
postalCode);
}
```

Usually, we also make/treat the value objects as immutable, i.e. instead of changing the value objects, we create new instances that wrap the new values:

```
// wrong:
this.address.setStreet(event.street);

// good:
this.address = new Address(event.street, ...);
```

Immutable types are handy, as they can be easily shared between different objects and returned by the entities without the risk of compromising consistency.

From the conceptual perspective, it makes sense to create a new instance of a value object when the value changes, as we’re literally assigning a new value.

The code gets more expressive:
// without:
Map<Long, String>
// with:
Map<PersonId, PhoneNumber>

Repository is a pattern used to separate the domain model from the data access code. It acts as a bridge between the domain model and the data source (usually a database) and provides a set of methods for performing data access operations without exposing the underlying storage details to the domain logic.

Here are the key aspects and purposes of a Repository in DDD:

Abstraction of Data Access: A Repository abstracts the data access code, allowing the domain model to work with domain objects rather than directly dealing with database queries or storage mechanisms. This separation enhances the maintainability and testability of the domain model.

Aggregates and Entities: In DDD, the Repository primarily deals with Aggregates and Entities, which are fundamental building blocks of the domain model. Aggregates are consistency boundaries that group together Entities and Value Objects. Repositories are typically associated with specific Aggregates or Entity types.

CRUD Operations: Repositories provide methods for creating, reading, updating, and deleting domain objects. These methods are usually tailored to the needs of the specific domain and the associated Aggregate.

Persistence Ignorance: A well-designed Repository is often persistence-agnostic, meaning it doesn't have specific knowledge of how data is stored or retrieved. This allows for flexibility in choosing different data storage technologies, such as relational databases, NoSQL databases, or in-memory storage.

Query Methods: Repositories may also offer query methods for retrieving domain objects based on specific criteria. These query methods help in finding objects based on the domain's needs without exposing low-level query details.

Transaction Management: Repositories can help manage transactions by ensuring that data operations are performed within a single unit of work. This is important for maintaining data consistency.

Caching and Performance Optimization: Repositories can also incorporate caching mechanisms to improve data access performance, but this depends on the specific requirements of the application.

Domain and application services:

In DDD terminology, a service is an object that implements some logic without holding any state.

domain services, which encapsulate domain logic, and application services, which provide technical functionality, such as user authentication or sending an SMS message.

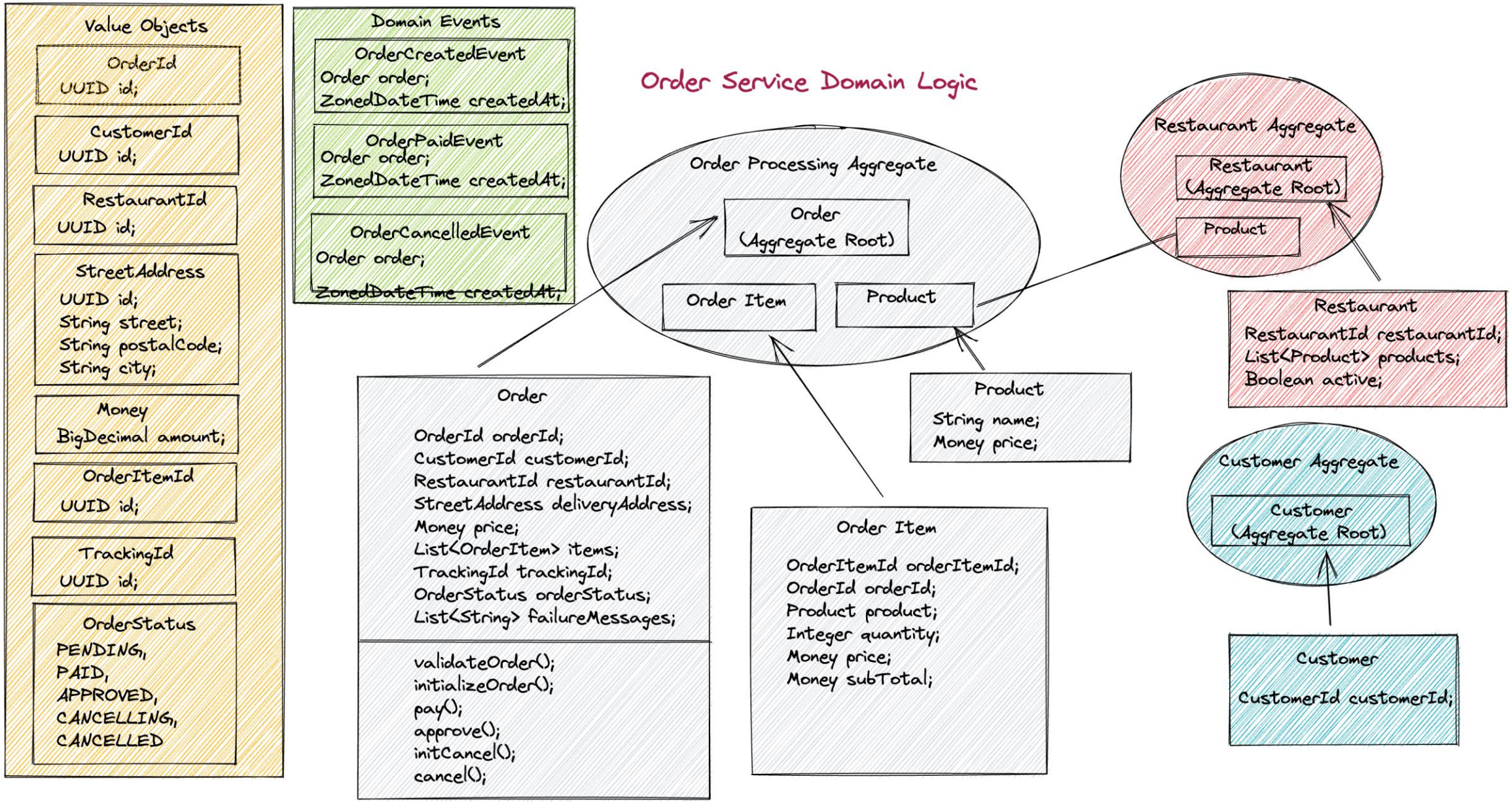
Domain services are often used to model behaviour that spans multiple entities.

Domain events

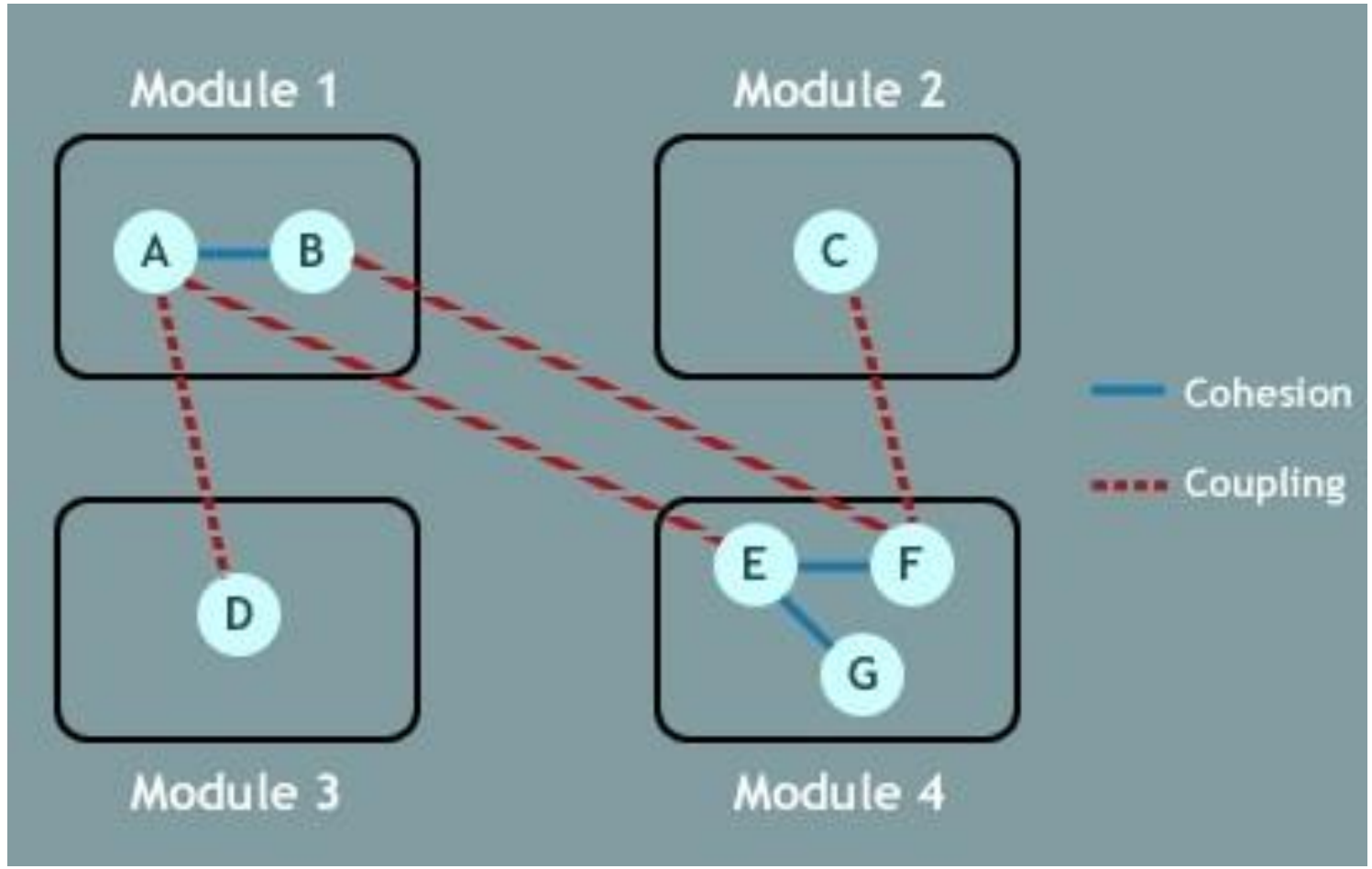
Domain events can be used to notify other parts of the system when something happens. As the name suggests, domain events should mean something within the domain.

For example, "a record was inserted into a table" is not a domain event. "A delivery was cancelled" is a domain event. Domain events are especially relevant in a microservices architecture.

Because microservices are distributed and don't share data stores, domain events provide a way for microservices to coordinate with each other.


Value Objects
OrderId
UUID id;
CustomerId
UUID id;
RestaurantId
UUID id;
StreetAddress
UUID id;
String street;
String postalCode;
String city;
Money
BigDecimal amount;
OrderItemId
UUID id;
TrackingId
UUID id;
OrderStatus
PENDING,
PAID,
APPROVED,
CANCELLING,
CANCELLED
Domain Events
OrderCreatedEvent
Order order;
ZonedDateTime createdAt;
OrderPaidEvent
Order order;
ZonedDateTime createdAt;
OrderCancelledEvent
Order order;
ZonedDateTime createdAt;
Order Service Domain Logic
Order Processing Aggregate
Order
(Aggregate Root)
Order Item
Product
Restaurant Aggregate
Restaurant
(Aggregate Root)
Product
Restaurant
RestaurantId restaurantId;
List<Product> products;
Boolean active;
Order
OrderId orderId;
CustomerId customerId;
RestaurantId restaurantId;
StreetAddress deliveryAddress;
Money price;
List<OrderItem> items;
TrackingId trackingId;
OrderStatus orderStatus;
List<String> failureMessages;
validateOrder();
initializeOrder();
pay();
approve();
initCancel();
cancel();
Product
String name;
Money price;
Order Item
OrderItemId orderItemId;
OrderId orderId;
Product product;
Integer quantity;
Money price;
Money subTotal;
Customer Aggregate
Customer
(Aggregate Root)
Customer
CustomerId customerId;

Cohesion and Coupling


Module 1
Module 2
A
B
C
D
E
F
G
Module 3
Module 4
Cohesion
Coupling

Cohesion	Coupling
Cohesion is the indication of the relationship within **module**.	**Coupling** is the indication of the relationships between modules.
Cohesion shows the module's relative **functional** strength.	Coupling shows the relative **independence** among the modules.
Cohesion is a degree (quality) to which a component / module focuses on the **single** thing.	Coupling is a degree to which a component / module is connected to the **other** modules.
While designing you should strive for **high cohesion** i.e. a cohesive component/ module focus on a single task (i.e., **single-mindedness**) with little interaction with other modules of the system.	While designing you should strive for **low coupling** i.e. **dependency** between modules should be less.
Cohesion is the kind of natural extension of data hiding for example, **class** having all members visible with a package having default visibility.	Making private fields, private methods and non public classes provides loose coupling.
Cohesion is **Intra – Module** Concept.	Coupling is **Inter -Module** Concept.

Our aim is to design Microservices that are autonomous, ie. have a low coupling with other services, have well defined interfaces, and implement a single business capability, ie. have high cohesion.

A successful equation is to keep high cohesion within a microservice and loose coupling between microservices.

Hexagonal Architecture

Ports and Adapters Pattern

A hexagonal architecture simplifies deferring or changing technology decisions.

It allows to isolate the core business of an application and automatically test its behaviour independently of everything else

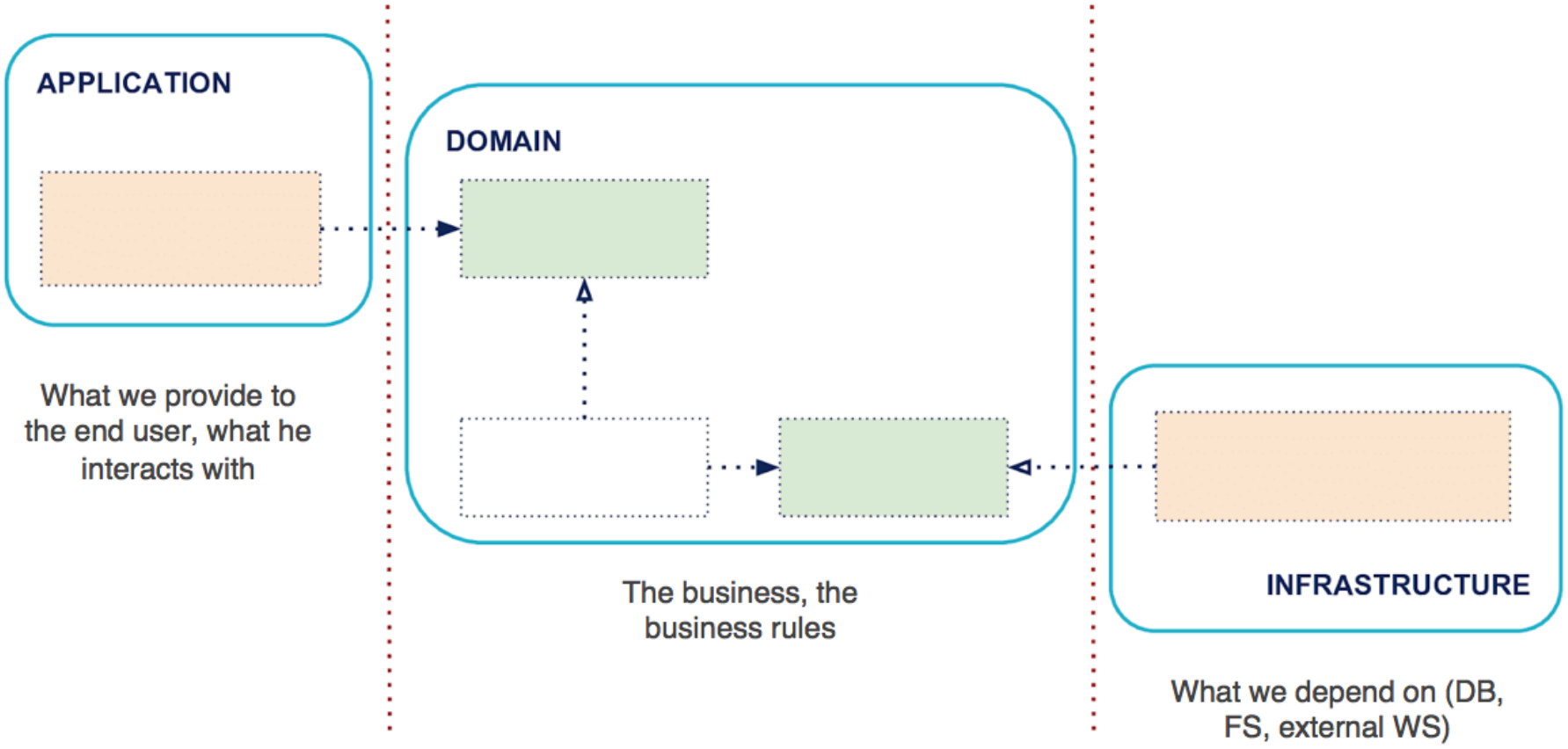
We want to change to a different framework? Write a new adapter. You want to use a database, instead of storing data in files? Again, write an adapter for it.

Principles of Hexagonal Architecture

The hexagonal architecture is based on three principles and techniques:

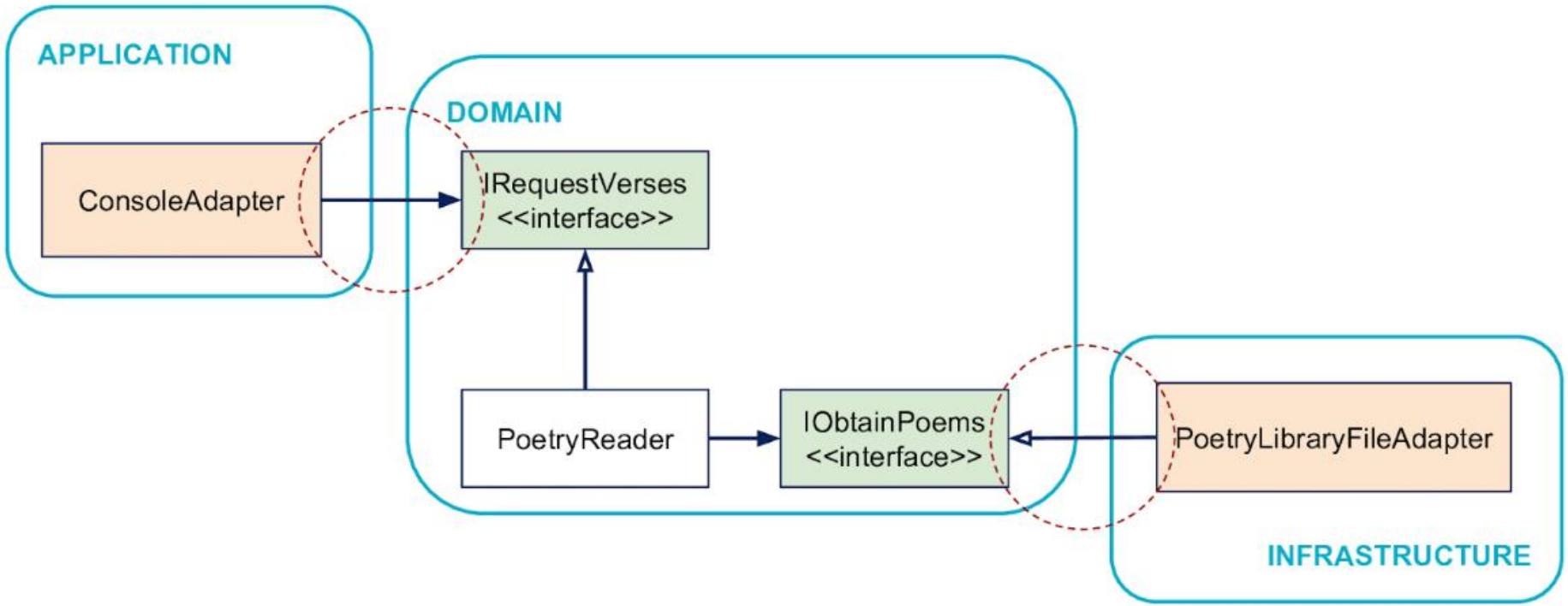
- ❑ Explicitly separate Application, Domain, and Infrastructure
- ❑ Dependencies are going from Application and Infrastructure to the Domain
- ❑ We isolate the boundaries by using Ports and Adapters

Principle: Separate Application, Domain and Infrastructure



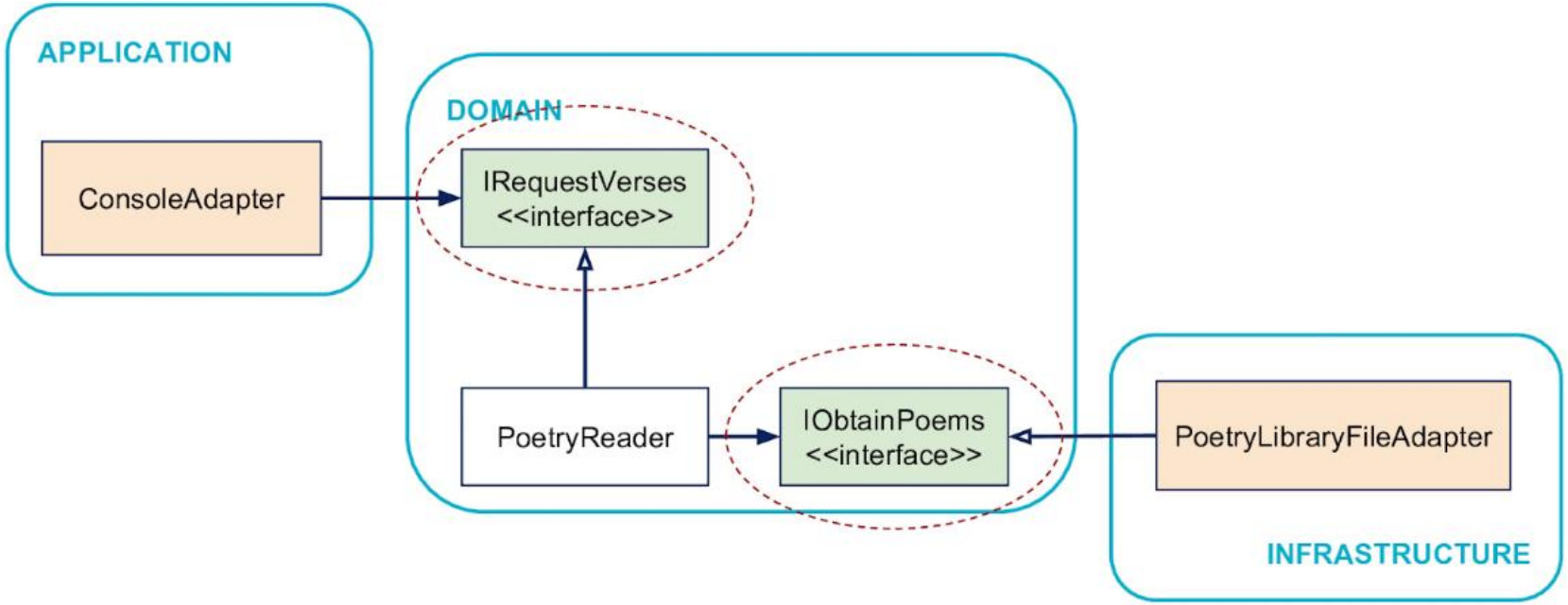
Principle: dependencies go inside

The program can be controlled both by the console and by tests, there is no concept of a console in the Domain. The Domain does not depend on the Application side, it is the Application side that depends on the Domain.



Principle: boundaries are isolated with interfaces

The application code drives the business code through an interface (here IRequestVerses) defined in the business code. And the business code drives the infrastructure through an interface also defined in the business code (IObtainPoems). These interfaces act as explicit insulators between inside and outside.



Draw a boundary around the business logic. The hexagon. Anything inside the hexagon must be free from technology concerns.

The outside of the hexagon talks with the inside only by using interfaces, called ports. Same the other way around. By changing the implementation of a port, you change the technology.

Isolating business logic inside the hexagon has another benefit. It enables writing fast, stable tests for the business logic. They do not depend on web technology to drive them.

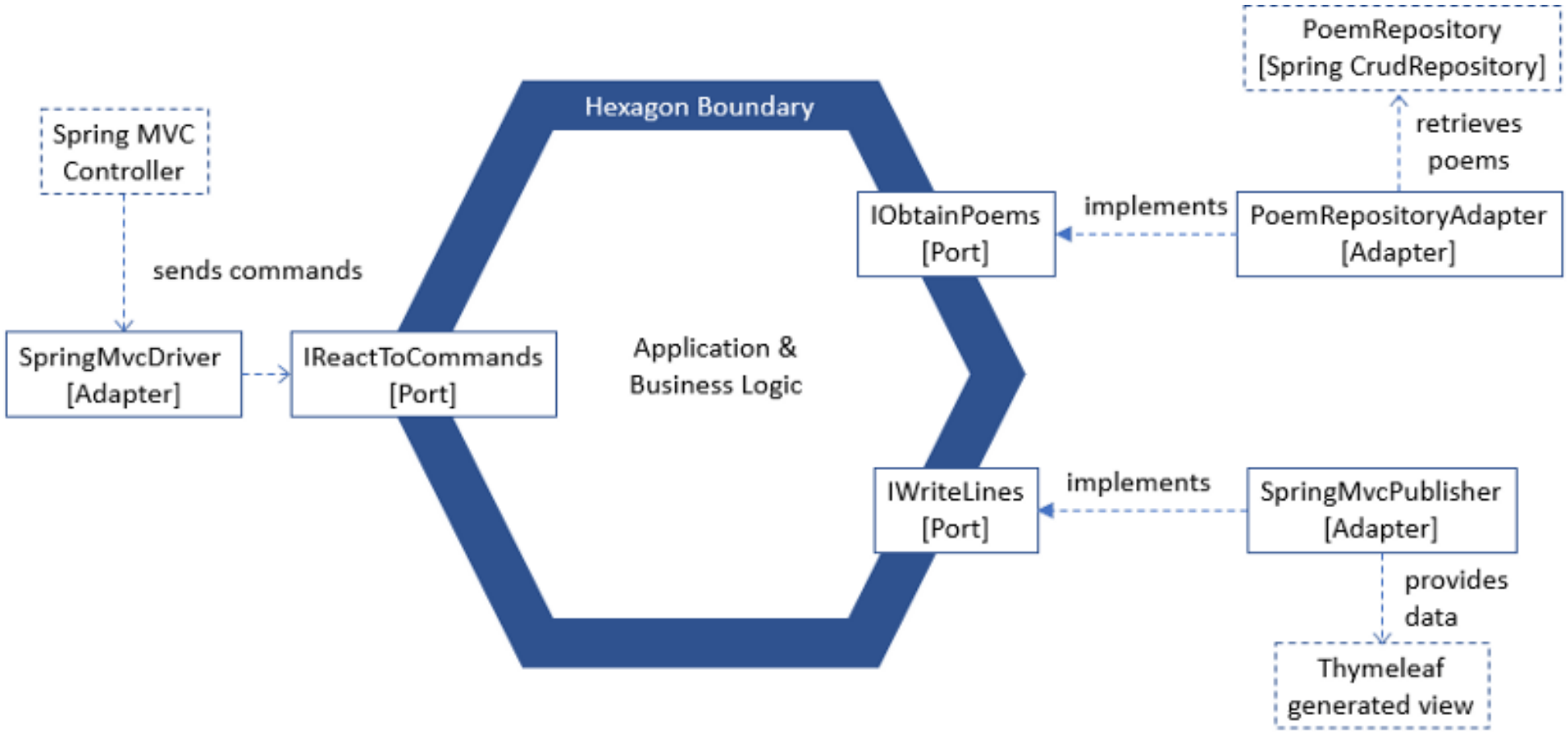
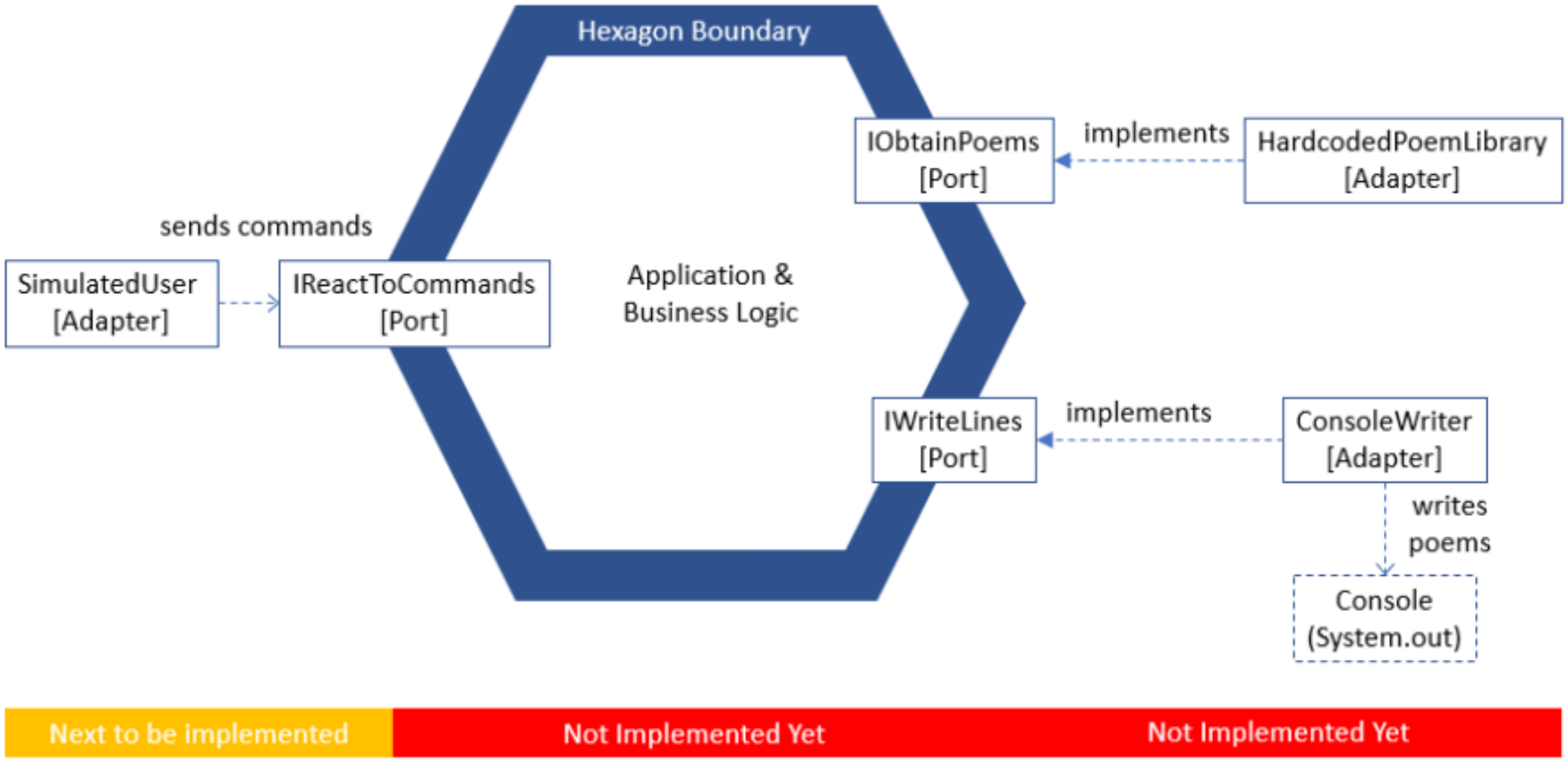
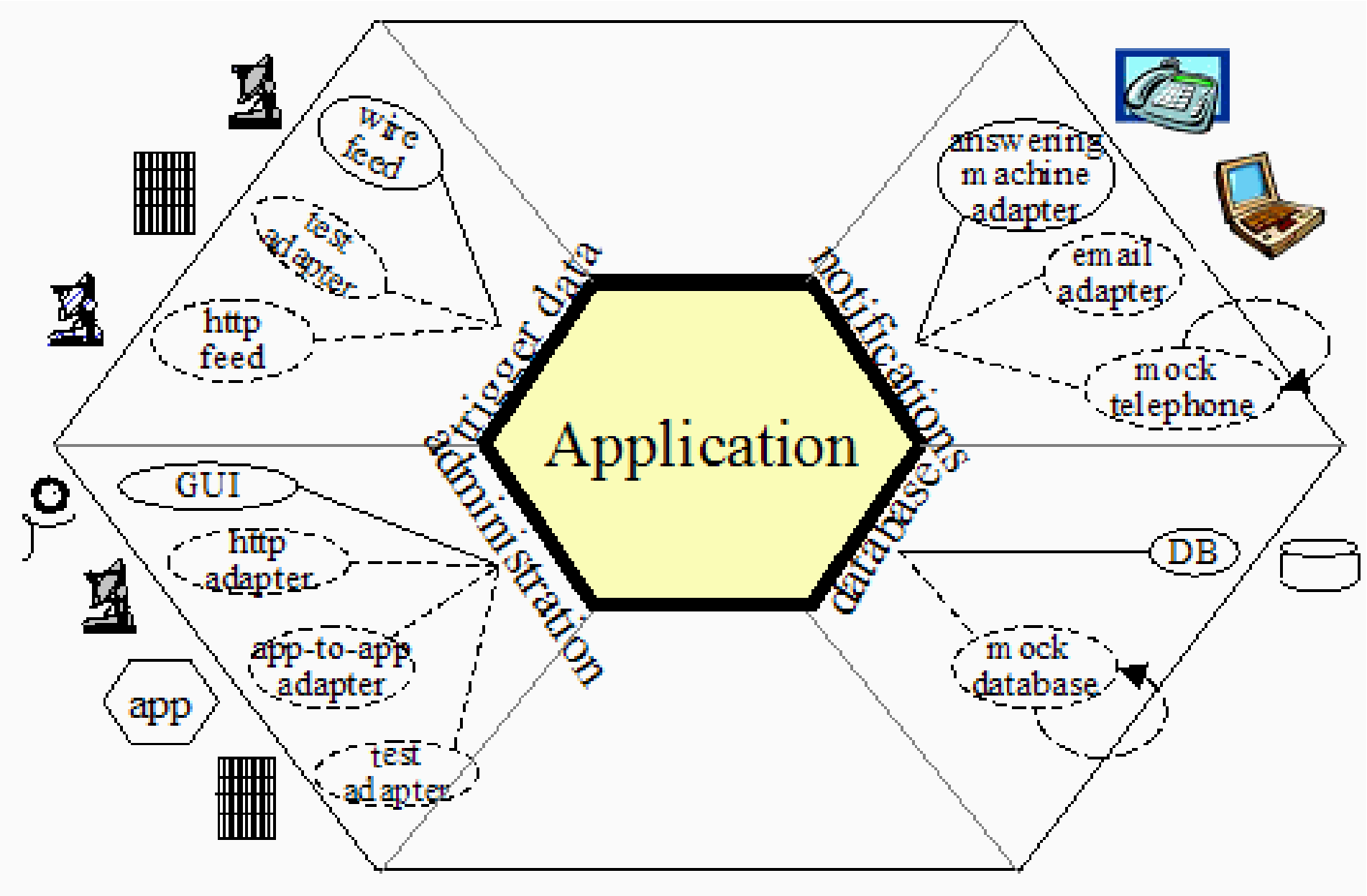

Hexagon Boundary
Application &
Business Logic
Spring MVC
Controller
sends commands
SpringMvcDriver
[Adapter]
IReactToCommands
[Port]
IObtainPoems
[Port]
implements
PoemRepositoryAdapter
[Adapter]
PoemRepository
[Spring CrudRepository]
retrieves
poems
IWriteLines
[Port]
implements
SpringMvcPublisher
[Adapter]
provides
data
Thymeleaf
generated view

Diagram shows Spring MVC technology as boxes with dotted lines, ports and adapters as solid boxes, and the hexagon without its internals.

An adapter translates between a specific technology and a technology free port. The SpringMvcDriver adapter on the left forwards commands to the IReactToCommands port. Because the adapter actively uses the port, it's called a driver adapter. IReactToCommands is called a driver port. Its implementation is inside the hexagon

On the right side, the SpringMvcPublisher adapter implements the IWriteLines port. This time, the hexagon calls the adapter through the port. That's why SpringMvcPublisher is called a driven adapter. And IWriteLines is called a driven port.


Hexagon Boundary
IObtainPoems
[Port]
implements
HardcodedPoemLibrary
[Adapter]
sends commands
SimulatedUser
[Adapter]
IReactToCommands
[Port]
Application &
Business Logic
IWriteLines
[Port]
implements
ConsoleWriter
[Adapter]
writes
poems
Console
(System.out)
Next to be implemented
Not Implemented Yet
Not Implemented Yet


wire feed
test adapter
http feed
trigger data
administration
GUI
http adapter
app-to-app adapter
app
test adapter
Application
notifications
databases
answering machine adapter
email adapter
mock telephone
DB
mock database

Ports

Ports with their provided implementations (Adapters).

PORT	ADAPTERS
HTTP Server	Jetty, Servlet
Templates	Pebble
Serialization Formats	JSON, YAML
Settings	Environment, Files, Resources

Microservices Patterns

Event Sourcing

The basic idea of Event Sourcing is to store every change in the state of an application as events instead of only storing the current state. The current state can be constructed by applying all past events.

Unlike the traditional approach with a relational database, event sourcing does not persist the current state of a record, but instead stores the individual changes as a series of deltas that led to the current state over time.

The procedure is similar to the way a bank manages an account, for example. The bank does not save the current balance. Instead, it records the deposits and withdrawals that occur over time. The current balance can then be calculated from this data:

+500 (deposit)
+200 (deposit)
 -300 (withdraw)

= 400 (balance)

Optimizing performance

A replay becomes more and more complex as the number of events that need to be replayed increases.

At first glance, the use of event sourcing seems to lead to read accesses becoming increasingly slow.

Since events are always only added at the end of the existing list and existing events are never changed, a replay calculated once will always produce the very same result for a certain point in time.

We can take advantage of this situation by saving the currently calculated state as a so-called **snapshot**.

The entire history does not always have to be played back all along the way.

Usually it is sufficient to start from the last snapshot and only look at the events that have been saved since then.

- **Event Sourcing** is a technique for reliably updating state and publishing events that overcomes limitations of other solutions.

- The design concepts for an event-driven architecture, using event sourcing, align well with microservices architecture patterns.

- Snapshots can improve performance when querying aggregates by combining all events up to a certain point in time.

- Event sourcing can create challenges for queries, but these are overcome by following CQRS guidelines and materialized views.

- Event sourcing and CQRS do not require any specific tools or software, and many frameworks exist which can fill in some of the low-level functionality.

How does Event Sourcing work?

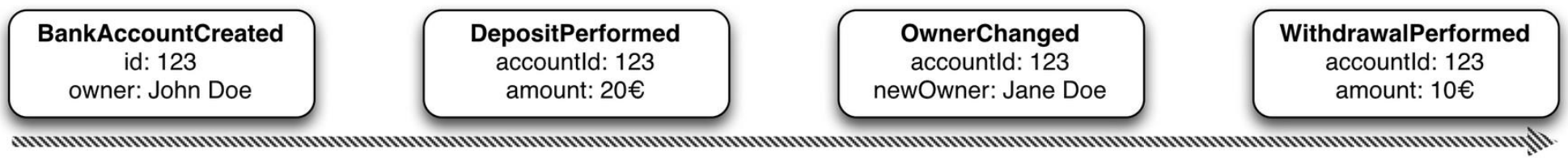
Saving objects

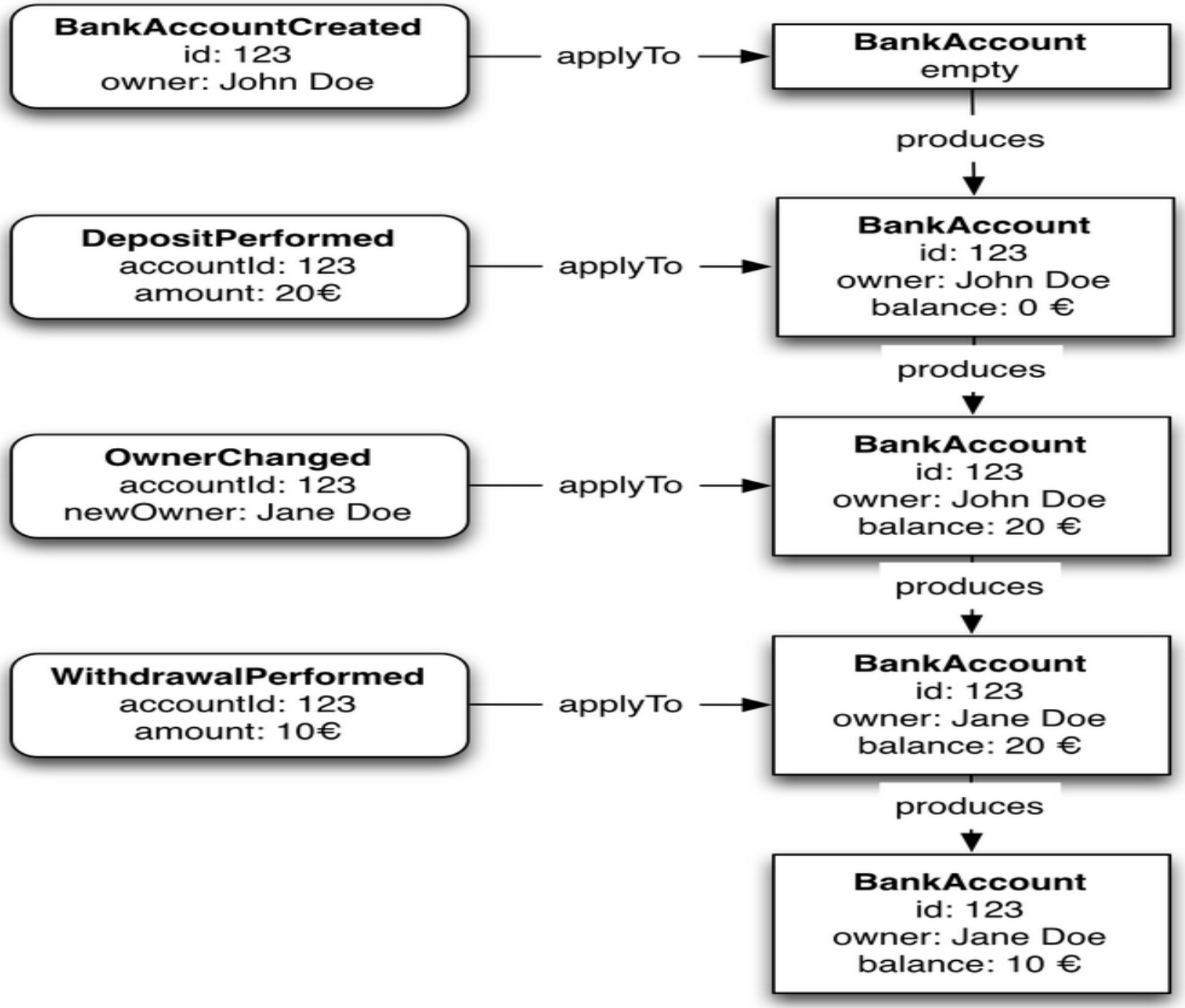
- Create an event for every state change of the object:

- Persist this stream of events, preserving event order

Restoring objects

Subsequently apply the events from the respective `EventStream` to a "blank" object instance




BankAccountCreated
id: 123
owner: John Doe
applyTo
BankAccount
empty
produces
DepositPerformed
accountId: 123
amount: 20€
applyTo
BankAccount
id: 123
owner: John Doe
balance: 0 €
produces
OwnerChanged
accountId: 123
newOwner: Jane Doe
applyTo
BankAccount
id: 123
owner: John Doe
balance: 20 €
produces
WithdrawalPerformed
accountId: 123
amount: 10€
applyTo
BankAccount
id: 123
owner: Jane Doe
balance: 20 €
produces
BankAccount
id: 123
owner: Jane Doe
balance: 10 €

Benefits and Drawbacks of Event Sourcing

A major benefit of event sourcing is that it reliably publishes events whenever the state of an aggregate changes. It is a good foundation for an event-driven microservices architecture.

Also, because each event can record the identity of the user that made the change, event sourcing provides an audit log that is guaranteed to be accurate.

Another benefit of event sourcing is that it stores the entire history of each aggregate

The main drawback of event sourcing is that querying the event store can be challenging.

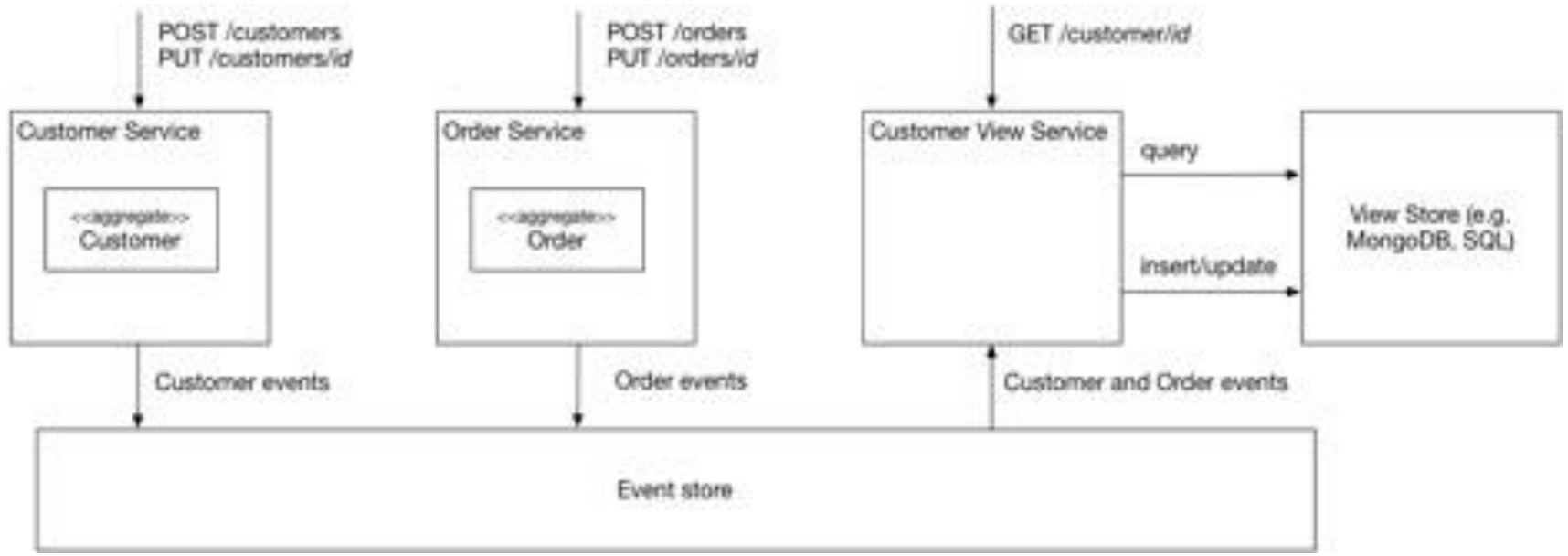
SELECT * FROM CUSTOMER WHERE CREDIT_LIMIT < ? AND c.CREATION_DATE > ?. There isn’t a column containing the credit limit. Instead, we must use a more complex and potentially inefficient query that has a nested SELECT to compute the credit limit by folding events that set the initial credit and adjust it.

Implementing Queries Using CQRS

A good way to implement queries is to use an architectural pattern known as Command Query Responsibility Segregation (CQRS). CQRS, as the name suggests, splits the application into two parts.

The first part is the command-side, which handles commands (e.g. HTTP POSTs, PUTs, and DELETEs) to create, update and delete aggregates. These aggregates are, of course, implemented using *event sourcing*.

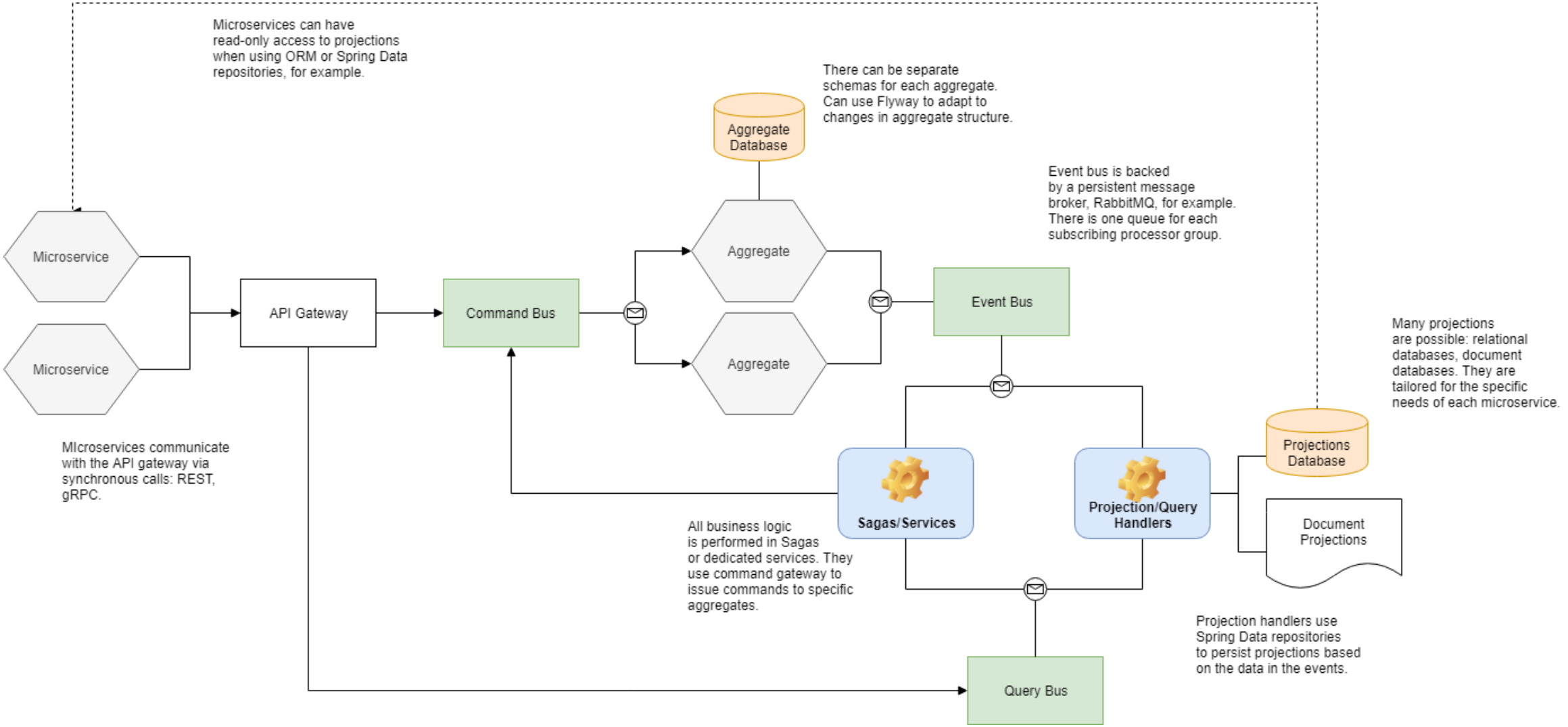
The second part of the application is the query side, which handles queries (e.g. HTTP GETs) by querying one or more materialized views of the aggregates. The query side keeps the views synchronized with the aggregates by subscribing to events published by the command side.


POST /customers
PUT /customers/id
POST /orders
PUT /orders/id
GET /customer/id
Customer Service
<<aggregate>>
Customer
Order Service
<<aggregate>>
Order
Customer View Service
query
insert/update
View Store (e.g. MongoDB, SQL)
Customer events
Order events
Customer and Order events
Event store

The Customer View Service subscribes to the Customer and Order events published by the command-side services.

It updates a view store that is implemented using MongoDB. The service maintains a MongoDB collection of documents, one per customer.

Each document has attributes for the customer details. It also has an attribute that stores the customer’s recent orders. This collection supports a variety of queries


Microservices can have read-only access to projections when using ORM or Spring Data repositories, for example.
There can be separate schemas for each aggregate. Can use Flyway to adapt to changes in aggregate structure.
Aggregate Database
Event bus is backed by a persistent message broker, RabbitMQ, for example. There is one queue for each subscribing processor group.
Microservice
Microservice
API Gateway
Command Bus
Aggregate
Aggregate
Event Bus
Many projections are possible: relational databases, document databases. They are tailored for the specific needs of each microservice.
MIcroservices communicate with the API gateway via synchronous calls: REST, gRPC.
Projections Database
Sagas/Services
Projection/Query Handlers
Document Projections
All business logic is performed in Sagas or dedicated services. They use command gateway to issue commands to specific aggregates.
Projection handlers use Spring Data repositories to persist projections based on the data in the events.
Query Bus

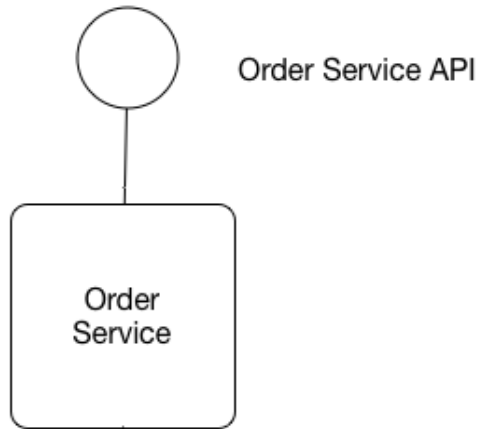
Pattern: Database Server per service

Context

Let’s imagine you are developing an online store application using the Microservice architecture pattern.

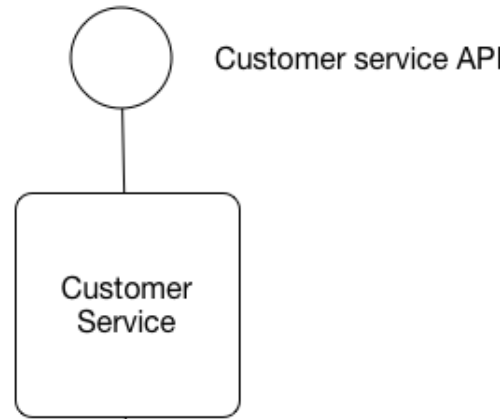
Most services need to persist data in some kind of database.

For example, the Order Service stores information about orders and the Customer Service stores information about customers.



ORDER table

ID	CUSTOMER_ID	STATUS	TOTAL	…
4567	234	ACCEPTED	84044.30	…



CUSTOMER table

ID	CREDIT_LIMIT	…
234	100000	…

Problem

What’s the database architecture in a microservices application?

Forces

Services must be loosely coupled so that they can be developed, deployed and scaled independently

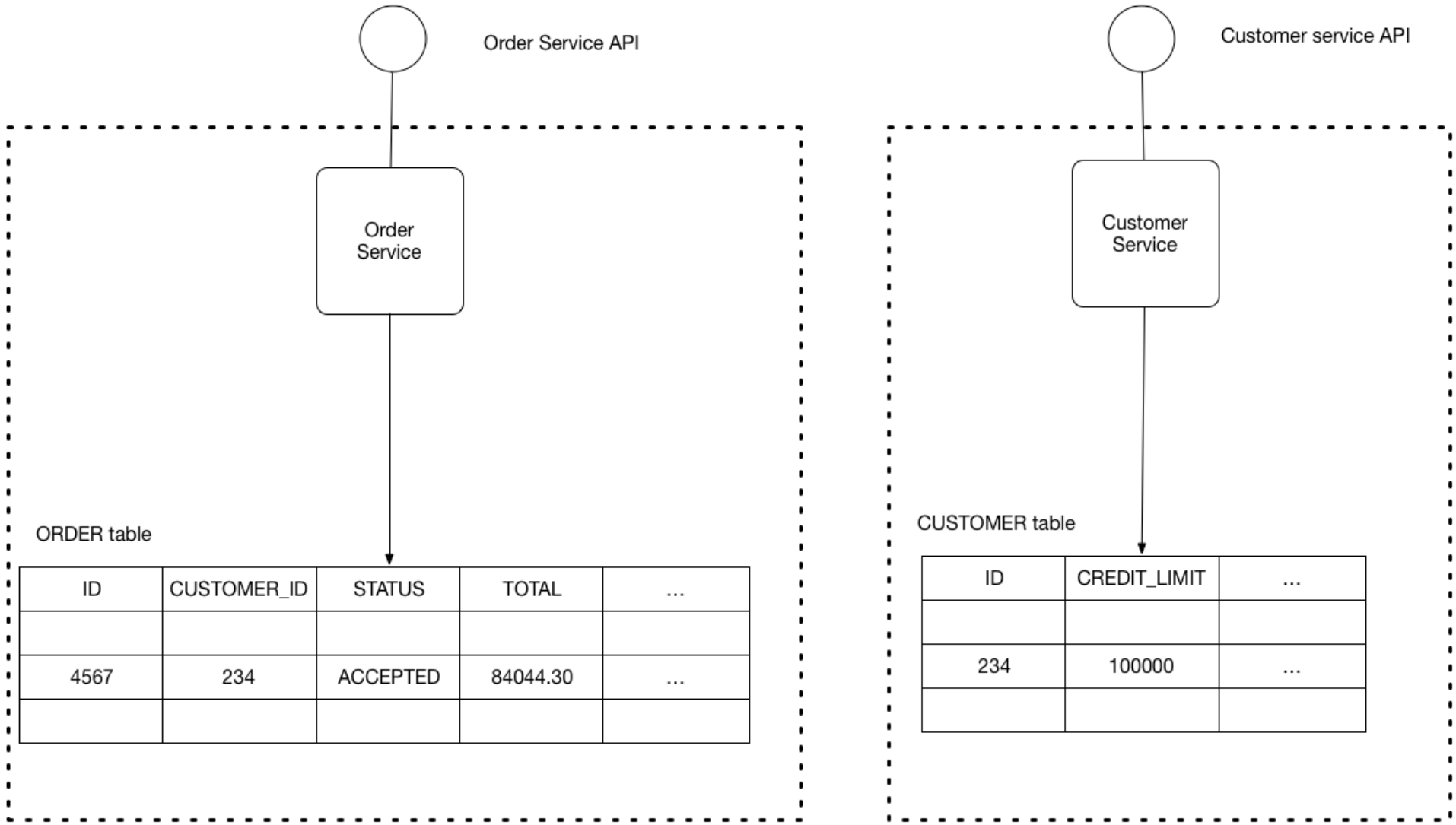
Some business transactions must enforce invariants that span multiple services. For example, the Place Order use case must verify that a new Order will not exceed the customer’s credit limit. Other business transactions, must update data owned by multiple services.

Some business transactions need to query data that is owned by multiple services. For example, the View Available Credit use case must query the Customer to find the creditLimit and Orders to calculate the total amount of the open orders.

Some queries must join data that is owned by multiple services. For example, finding customers in a particular region and their recent orders requires a join between customers and orders.

Different services have different data storage requirements. For some services, a relational database is the best choice. Other services might need a NoSQL database such as MongoDB, which is good at storing complex, unstructured data, or Neo4J, which is designed to efficiently store and query graph data.

Solution : Keep each microservice's persistent data private to that service and accessible only via its API.



There are a few different ways to keep a service's persistent data private. You do not need to provision a database server for each service. For example, if you are using a relational database then the options are:

Private-tables-per-service – each service owns a set of tables that must only be accessed by that service

Schema-per-service – each service has a database schema that's private to that service

Database-server-per-service – each service has it's own database server.

Resulting context

Using a Database-server-per-service has the following benefits:

Helps ensure that the services are loosely coupled. Changes to one service's database does not impact any other services.

Each service can use the type of database that is best suited to its needs. For example, a service that does text searches could use ElasticSearch. A service that manipulates a social graph could use Neo4j.

Using a Database-server-per-service has the following drawbacks:

Implementing business transactions that span multiple services is not straightforward. Distributed transactions are best avoided. Moreover, many modern (NoSQL) databases don’t support them.

The best solution is to use the Saga pattern. Services publish events when they update data. Other services subscribe to events and update their data in response.

Implementing queries that join data that is now in multiple databases is challenging.

There are various solutions:

API Composition - the application performs the join rather than the database. For example, a service (or the API gateway) could retrieve a customer and their orders by first retrieving the customer from the customer service and then querying the order service to return the customer's most recent orders.

Command Query Responsibility Segregation (CQRS) - maintain one or more materialized views that contain data from multiple services. The views are kept by services that subscribe to events that each services publishes when it updates its data.

For example, the online store could implement a query that finds customers in a particular region and their recent orders by maintaining a view that joins customers and orders. The view is updated by a service that subscribes to customer and order events.

three frameworks that presently support saga processing, namely Narayana LRA, Axon framework and Eventuate.io.

Narayana LRA

Narayana Long Running Actions is a specification developed by the Narayana team in the collaboration with the Eclipse MicroProfile initiative. The main focus is to introduce an API for coordinating long running activities with the assurance of the globally consistent outcome and without any locking mechanisms.

[https://github.com/eclipse/microprofile-sandbox/tree/master/proposals/0009-LRA]

Axon framework

Axon framework is Java based framework for building scalable and highly performant applications. Axon is based on the Command Query Responsibility Segregation (CQRS) pattern. The main motion is the event processing which includes the separated Command bus for updates and the Event bus for queries.

[http://www.axonframework.org/]

Eventuate.io

Eventuate is a platform that provides an event-driven programming model that focus on solving distributed data management in microservices architectures. Similarly to the Axon, it is based upon CQRS principles. The framework stores events in the MySQL database and it distributes them through the Apache Kafka platform.

[http://eventuate.io]

SAGA Pattern

Saga Pattern is a direct result of Database-per-service pattern.

In Database-per-service pattern, each service has its own database. In other words, each service is responsible only for its own data.

This leads to a tricky situation.

Some business transactions require data from multiple services. Such transactions may also need to update or process data across services. Therefore, a mechanism to handle data consistency across multiple services is required.

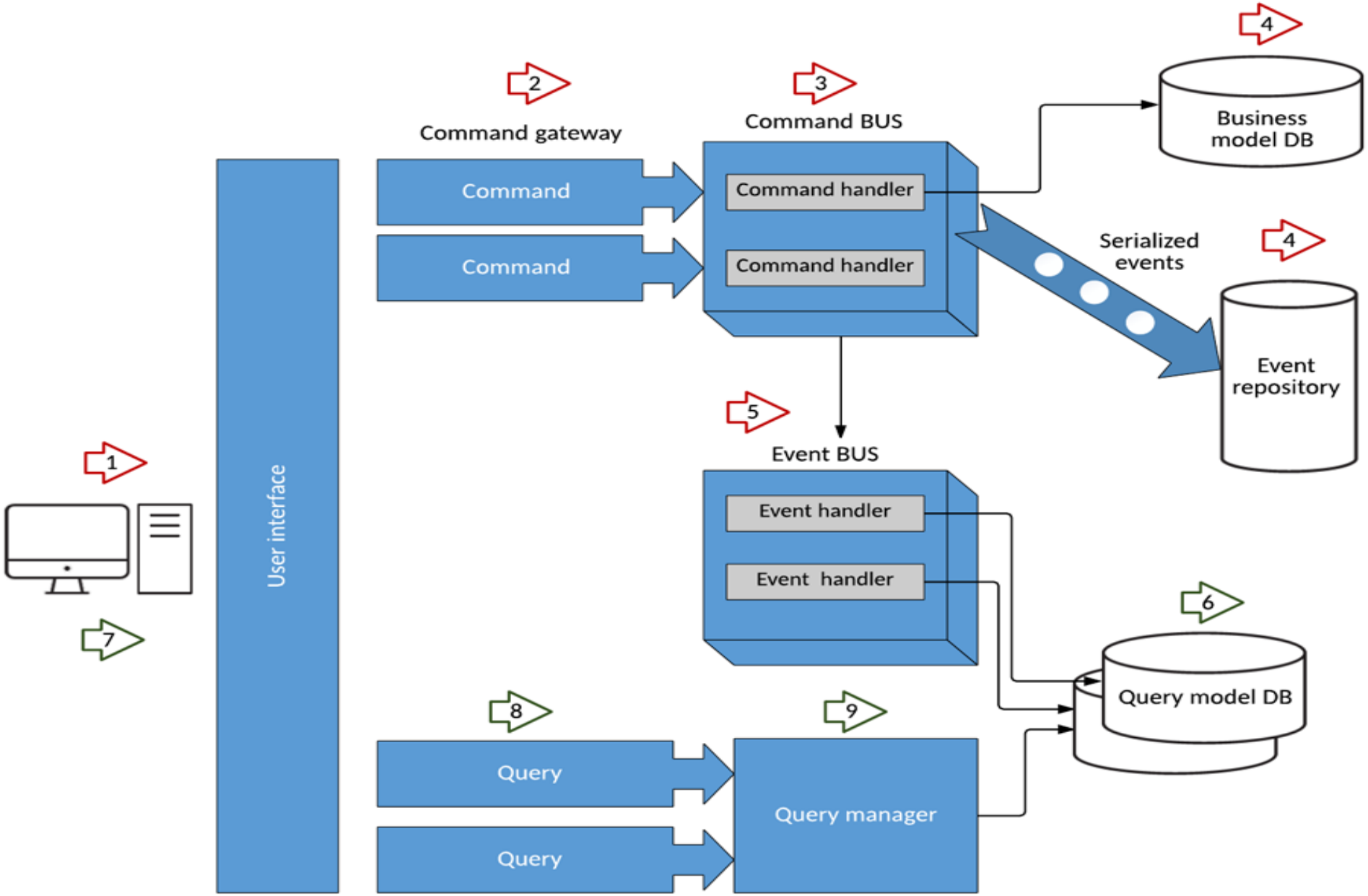
This situation or use-case forms the basis of the Saga Pattern.

Axon Framework is a microservices framework that makes it easy to build distributed systems. It provides great support for Spring Boot.

Axon works on the concept of commands and events.

Commands are user-initiated actions that can change the state of your aggregate.

Events are the actual changing of that state.


1
User interface
7
2
Command gateway
Command
Command
3
Command BUS
Command handler
Command handler
4
Business model DB
Serialized events
4
Event repository
5
Event BUS
Event handler
Event handler
6
Query model DB
8
Query
Query
9
Query manager

The **TargetAggregateIdentifier** annotation tells Axon that the annotated field is an id of a given aggregate to which the command should be targeted.

Defining the Saga Pattern

Saga Pattern proposes implementing distributed transactions in the form of Sagas.

A Saga is nothing but a sequence of local transactions. These local transactions are occurring at the service level. Whenever a local transaction occurs, it publishes a message or an event. Such an event is responsible for triggering the next transaction in the Saga.

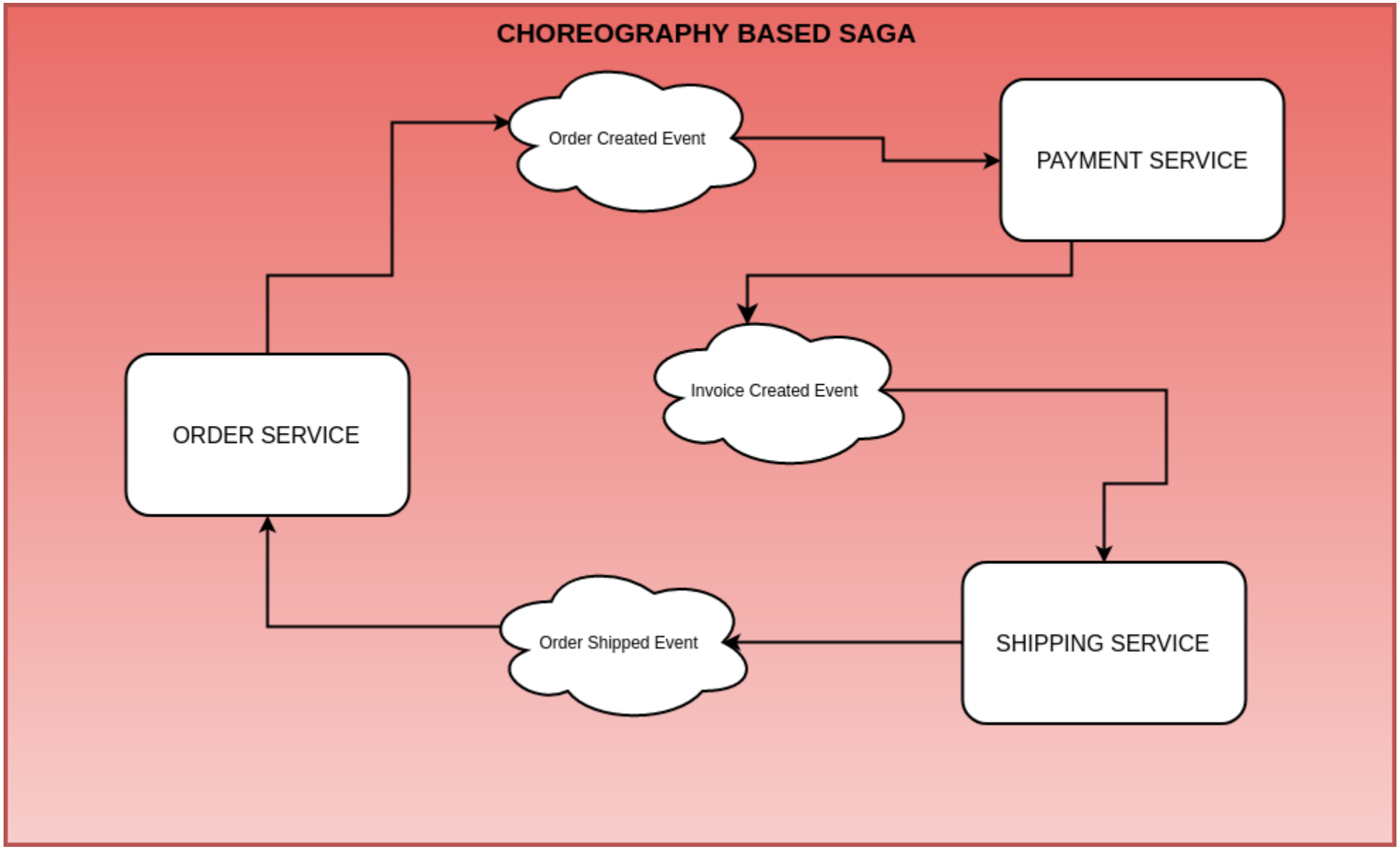
But then, you might ask what happens when a single transaction in the Saga sequence fails?

In that case, the Saga executes a series of compensating transactions. These transactions basically undo the changes made by the preceding transactions.

Types of Saga

There are basically two types of Saga. In other words, there are two ways in which you can implement the Saga Pattern.

Choreography-Based Saga - In this type of Saga Implementation, each service publishes one or more domain events. These domain events trigger local transactions in other microservices.

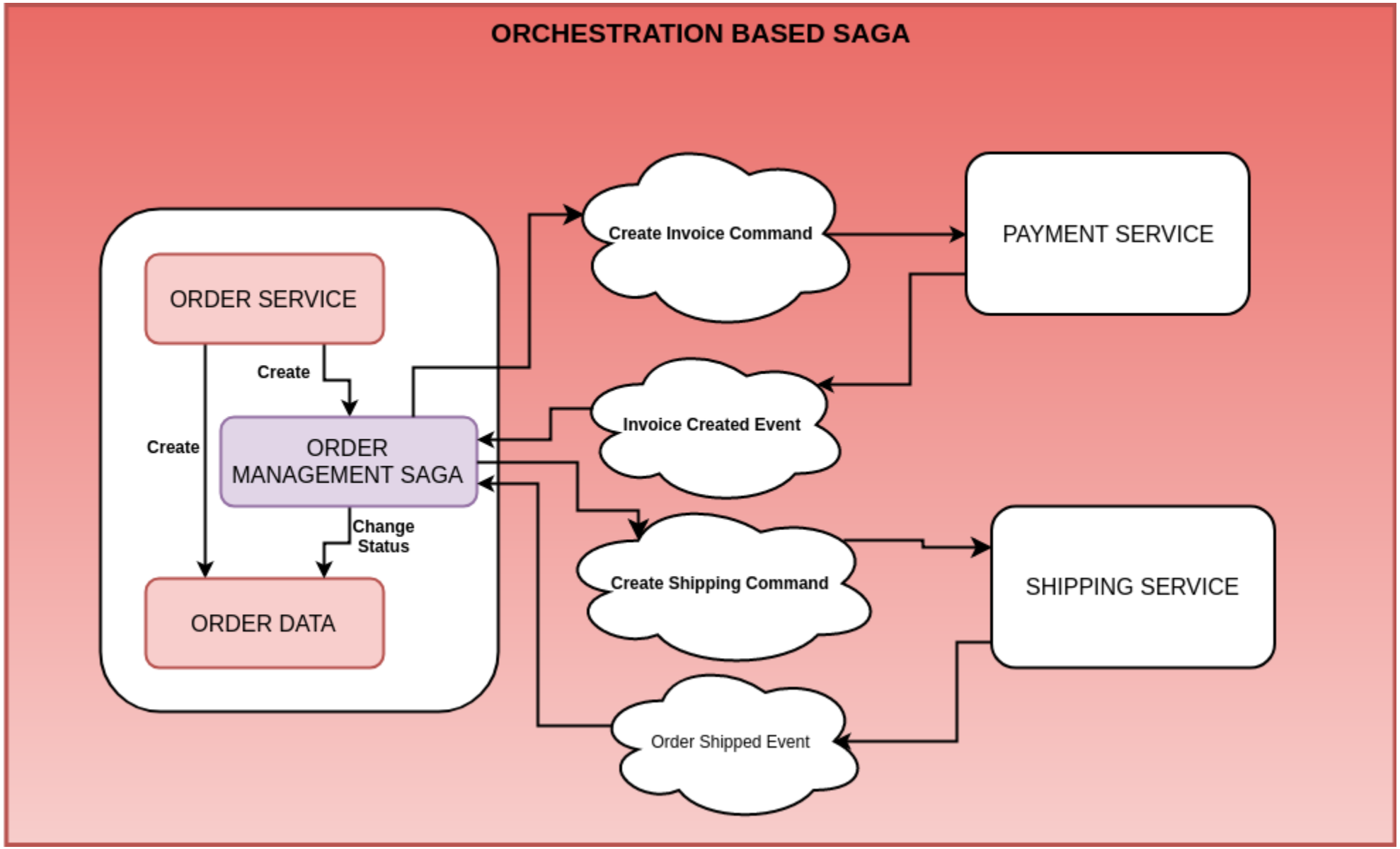

CHOREOGRAPHY BASED SAGA
Order Created Event
PAYMENT SERVICE
ORDER SERVICE
Invoice Created Event
Order Shipped Event
SHIPPING SERVICE

Let’s understand what is happening in the above example.

- **Order Service** is responsible for creating an **Order**. It also publishes an event for the same.
- The **Payment Service** listens to that event and creates an **Invoice**.
- When the **Invoice** is created, the **Shipping Service** creates the shipment.
- When the **Order** is shipped, the **Order Service** updates the status of the **Order**.

Orchestration-Based Saga

In Orchestration-Based Saga, there is an orchestrator. An orchestrator can also be thought of as a manager that directs the participant services to execute local transactions.


ORCHESTRATION BASED SAGA
ORDER SERVICE
Create
Create
ORDER MANAGEMENT SAGA
Change Status
ORDER DATA
Create Invoice Command
PAYMENT SERVICE
Invoice Created Event
Create Shipping Command
SHIPPING SERVICE
Order Shipped Event

Let’s understand what’s happening in this approach:

- **Order Service** creates an **Order**. Then, it also creates the Order Management Saga.

- The **Order Management Saga** sends a **Create Invoice Command** to the **Payment Service**.

- The **Payment Service** creates the **Invoice** and responds back to the **Order Management Saga**. Note that these responses can be totally asynchronous and message-driven as well.

- In the next step, the **Order Management Saga** issues the **Create Shipping Command** to the **Shipping Service**.

- The **Shipping Service** does the needful and creates the **Shipping**. It also replies back to the **Order Management Saga**.

- The **Order Management Saga** changes the status of the **Order** and ends the Saga’s life-cycle.

Orchestration-Based Implementation

The Axon Platform comprises of the core Axon Framework and also the Axon Server. The Axon Server facilitates communication between microservices.

Orchestration-Based Implementation

The Axon Platform comprises of the core Axon Framework and also the Axon Server. The Axon Server facilitates communication between microservices.

Building a complex software system

When building a complex software system, combining various architectural and design patterns can result in a robust, maintainable, and scalable solution.

Let's discuss how Domain-Driven Design (DDD), Event Sourcing, Command Query Responsibility Segregation (CQRS), and the Ports and Adapters (Hexagonal) pattern can work together:

1. Domain-Driven Design (DDD):

•DDD focuses on modeling the core business domain by defining entities, aggregates, value objects, and repositories. It encourages a shared understanding of the problem domain between developers and domain experts.

•DDD helps ensure that your software design reflects the real-world domain and enforces clear boundaries within the domain.

2. Event Sourcing:

- Event Sourcing is a pattern in which you capture all changes to an application's state as a sequence of events. This history of events can be used to rebuild the current state and analyze past states.

- Events are persisted and can serve as a source of truth, enabling audit trails, temporal querying, and resilience against failures.

3. Command Query Responsibility Segregation (CQRS):

- CQRS separates the read (query) and write (command) operations in an application.

- The command side handles state changes and is responsible for processing and storing events.

- The query side handles reading data from the event store or other storage to serve queries efficiently.

4. Ports and Adapters (Hexagonal) Pattern:

- Ports and Adapters (Hexagonal) pattern is about decoupling your application from external concerns and isolating the core business logic.

- It promotes a clear separation of application core (domain logic) and external interfaces (such as web APIs, databases, messaging systems).

- Ports are interfaces through which your application interacts with the external world, and Adapters implement those interfaces to connect with specific technologies.

Combining the Patterns:

Domain Model: Apply DDD to model the core business domain. Define your aggregates, entities, and value objects to capture the domain logic.

Event Sourcing: Implement Event Sourcing to track changes to your domain's state as a series of events. Each aggregate's state is reconstructed from its event history.

CQRS: Use CQRS to separate the command side (handling writes and events) from the query side (handling reads). The query side can efficiently serve read requests by querying the event store or other storage optimized for queries.

Ports and Adapters: Apply the Ports and Adapters pattern to isolate your application's core from external concerns. Your ports (interfaces) define how your application interacts with external systems, while adapters implement these interfaces to connect with specific technologies (e.g., a REST API, a message queue).

By combining these patterns, we create a highly adaptable, domain-focused architecture.

The domain model is enriched by DDD principles, events are used for auditing and temporal queries, and the application remains decoupled from external systems, allowing for easy testing and swapping out of adapters.

This combination can lead to a flexible and maintainable architecture, especially in complex domains or systems with evolving requirements.