painting by Jyoti Jagtap

Domain-Driven Design And Microservices Explained with Examples

Learn to clarify and focus the boundaries of your system's architecture

Sandeep Jagtap

This book is for sale at http://leanpub.com/domaindrivendesignandmicroservicesexplained

This version was published on 2023-02-01

To My Mother and Father. Thanks for everything!

Contents

#Summary

All working code examples are available at
https://github.com/ddd-workshop-org

*** Note: this book is in Draft. Contents mentioned below will be covered when I finish writing this book. So current contents may not match one to one to what is listed below as chapters ***

This book is designed to talk about the relationship of DDD to microservices architecture. As we go through DDD concepts using code examples, we will make connections to microservices architecture.

Chapter 1 talks about the rationale behind DDD. I will talk about my experiences using it on real projects. It will also cover the relationship between XP, which is arguably one of the most useful Agile methodologies, and DDD. Then it covers briefly how DDD is related to OO and the functional paradigm,and how existing design patterns and SOLID principles apply in the context of DDD. It then covers the basics of microservices architecture.

Chapter 2 covers the concept of Domain Event which is the basis for Event Driven Architecture. It covers good practices for Domain Events and how to model them. It talks about real life examples like Banking Ledger and on how Domain Events are useful. It covers the state based model vs Domain Event based models and what are advantages of later.

Chapter 3 and 4 covers the Entity and Value Object concepts respectively which are the tactical design patterns. And good modeling practices for both along with code samples.
It covers when Domain Object may be Entity and same may become Value Object.

Chapter 5 covers Domain Model and what it means. It then covers two known architecture styles (Layered/Tiered architecture and Hexagonal Architecture) commonly used in microservices architecture or any web application. It will then detail out on how the Domain Layer of DDD fits into these two architectures using Code examples.

Chapter 6 covers Domain Service as one of the not so known patterns and it's quite useful when modeling business processes which do not fit naturally as Entity or Value Object. Domain Services are not the same as Application services in Layered Architecture.

Chapter 7 covers Aggregate which is the most important tactical pattern. We will see how Entity becomes Aggregate and will learn what are good modeling practices for Aggregates.

Then in Chapter 8 we will see how Aggregates are related to microservices. We will see code of microservice using the Domain model which we have so far and adding Layered Architecture on top of it. We will also look at what happens when Aggregate is split and what does that mean to our microservices architecture. We will cover microservices communication patterns and Sagas. We will cover the relation of Aggregates and Domain Services to REST resources.

In Chapter 9 and Chapter 10 we will cover the most important strategic pattern called a bounded context and its relationship to microservices architecture using code examples. We will discuss subdomains and their mapping to bounded contexts. This chapter will help with how the given problem domain can be split into subdomains and how that helps with overall microservices architecture. We will also cover team topologies and how bounded context can be useful for those.

In Chapter 11 we will cover Ubiquitous language and how that applies only in a given bounded context. We will also discuss why the Canonical domain model and just one ubiquitous language does not work across the entire problem domain. We will discuss good practices on Ubiquitous language.

Chapter 12 covers the relationships between different bounded contexts and what that means to the business domain problem we are solving and how it affects our microservices architecture.

Chapter 13 covers how we can apply what we learnt with DDD to existing codebases or existing microservices.

Chapter 14 covers one of the very old techniques called the modular monolith and how to apply that to modern microservices architecture and how it can benefit during microservices development.

And then in Chapter 15 we will cover Event Sourcing based on the Domain Event concept which we learnt. And understand the CQRS architecture pattern.

In *Chapter 16* we will cover different types of architectural style that we can apply to Domain Models and bounded Contexts. We will see when you use which style and tradeoff for them.

Chapter 17 covers techniques of finding bounded contexts which can be used when we start discovery or inception for new projects or modernization of existing legacy systems. We will cover Event Storming and Domain Storytelling which are two popular techniques

Chapter 18 talks about Data Mesh and how DDD helps with building Data Meshes. We will cover challenges with the existing data landscape and things like data lake and data warehouse and how Data Mesh is trying to solve it.

Chapter 19 Microfrontends are gaining popularity on the UI side and we will see how DDD helps with building micro frontends.

Chapter 20 Overall Summary

* * *

Introduction to Domain-Driven Design

Welcome to Domain Driven Design!

What is Domain Driven Design aka DDD

It is a term coined by Eric Evans in his seminal and timeless Book which came out in 2003. It is Domain centric approach and collection of patterns, principles and practices.

DDD is very useful in building microservices architecture, Data Mesh, Team topologies and Modular monoliths.

Domain Driven Design got lot of attention after microservices architecture became popular style of architecture after era of SOAP based webservices.

Why use Domain Driven Design

DDD helps with mapping production-code closely to the business domain.
It helps manage Domain Complexity and write more maintainable code.

It also supports Extensibility (Ability to add new features with lesser efforts). And helps with overall architecture - High Cohesion and loose coupling.

"Complexity in software is the result of inherent domain complexity (essential) mixing with technical complexity (accidental)". - Eric Evans

When to use Domain Driven Design

Any Complex Domain.
Example: E Commerce. Maintaining inventory, Fulfillment, distribution, shipping, Product Catalog, Returns and Refunds, Invoicing, Marketplace.

It is not very useful for simple CRUD applications (when there is not much business logic).

Examples of not so complex domains - Building UI, BFFs and services which talk to core/legacy services(running on mainframes, etc).

DDD and Friends

Iterative approach is core to DDD Philosophy. DDD says Design/Model/Code evolves over a period of time as we know more about Domain and Business!

DDD complements Object Oriented Paradigm (and Functional Paradigm).

This book covers applying DDD using Object Oriented Paradigm.

DDD gives formal names to some of the concepts like Entity, Repository, Anemic domain, service dependencies, Domain model that you might be already using.

All things in Object Orieneted programming would still apply - GOF Design Patterns, SOLID,Law of Demeter, DRY (within bounded context - more on that later). Also all functional programming concepts would still apply - Immutability, higher order functions,etc.

In my view DDD is Object Oriented++.

Basics of microservices

There is an excellent article by Martin Fowler and James Lewis on microservices. https://martinfowler.com/articles/microservices.html

Most of industry have moved from SOAP based Webservices architecture to microservices architecture as default style of architecture.

Microservices are everywhere. **Following are few of challenges I have seen with them**

Either too fine-grained or too coarse-grained.

Some microservices become to big..almost like monolith.

microservices are dependent on each other for user journey to complete.

Identifying whether something is a microservice is not easy.

Ending up building tightly coupled microservices.

Some of above challenges can be addressed with Domain-Driven Design. We will learn about some ways of dealing with above challenges using DDD in this book.

Domain Events - Understanding building blocks of DDD

This book takes an approach of covering DDD concepts bottom-up. That means - we will learn from basic tatical patterns and then reach to strategic patterns of DDD.

Let's understand the building blocks of DDD using simple use cases.

Use case 1

Add a "Apple Pencil" to a Cart

Note:
Please do not create a User class.
Please do not create ProductCategory, Variant, Colour, etc classes.

Let's code this use case. One sample implementation could be -

```java
package com.ddd_bootcamp.domain;

import java.util.ArrayList;
import java.util.List;

public class Cart {
    private List<Product> products = new ArrayList<>();

    public void add(Product product) {
        products.add(product);
    }
}

package com.ddd_bootcamp.domain;

public class Product {
    private String name;

    public Product(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

Use case 2

Add a “Sony Wireless headphone” to a Cart

Let’s code this use case. One sample implementation could be -
We can actually use same code from Use case 1. Beacuse “Sony Wireless headphone” is just another product, so no code change needed.

Use case 3

Add 2 quantities of “Apple Pencil” to a Cart.

Let’s code this use case. One sample implementation could be -

```java
package com.ddd_bootcamp.domain;

import java.util.ArrayList;
import java.util.List;

public class Cart {
    private List<Item> items = new ArrayList<>();

    public void add(Item item) {
        items.add(item);
    }
}

package com.ddd_bootcamp.domain;

public class Item {
    private Product product;
    private int quantity;

    public Item(Product product, int quantity) {
        this.product = product;
        this.quantity = quantity;
    }
}
```

How did we come up with class Item in above code? Was it a Developers deciding on that name or something else? When we are not sure about what should we name the classes, it is hint that we may missing some domain concept. Talking to Domain Experts, Business Analysts helps in this case. So assuming in this case, Domain Expert tells us that concept of Product and multiple quantities is called as Item. And hence we code our solution using Item class. This takes us to concept of Ubiquitous language in DDD. Ubiquitous language is roughly using same terms and names that Domain Experts use in our code. We will talk more about Ubiquitous language in upcoming chapters.

Use case 4

Remove already added Item "Apple Pencil" (with all its quantities) from Cart.

Let's code this use case. One sample implementation could be -

```java
package com.ddd_bootcamp.domain;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;
import java.util.stream.Collectors;

public class Cart {
    private List<Item> items = new ArrayList<>();

    public void add(Item item) {
        items.add(item);
    }

    public void remove(Item item) {
        items.remove(item);
    }

    public void removeX(Product product) {
        items.removeIf(item -> item.getProduct().equals(product));
    }
}
```

So we have added new behaviour to Cart class to remove the Item by passing Item or by passing a Product as parameter to a method.

Let's get to our next use case.

Use case 5

As a business User, I would like to know which Products (Product’s name) were removed from Cart.
–

Note – Please do not create a business User class. You could have a method on Cart class or some other class, which returns list of Product names which were removed from Cart.

Let’s code this use case. There could be mutiple implementations for it. Mainitaining list of removed products, using observers, etc. Let’s look at solution using concept called as Domain Events.

```
package com.ddd_bootcamp.domain;

import com.ddd_bootcamp.domain.events.DomainEvent;
import com.ddd_bootcamp.domain.events.ItemAddedToCartEvent;
import com.ddd_bootcamp.domain.events.ItemRemovedFromCartEvent;

public class Cart {

    private List<DomainEvent> events = new ArrayList<>();
    private List<Item> items = new ArrayList<>();

    public void add(Item item) {
        ItemAddedToCartEvent itemAddedEvent =
                new ItemAddedToCartEvent(item.getProductName(),
                        item.getQuantity());

        apply(itemAddedEvent);
    }

    public void remove(Item item) {
        ItemRemovedFromCartEvent itemRemovedFromCartEvent =
                new ItemRemovedFromCartEvent(item.getProductName());

        apply(itemRemovedFromCartEvent);
    }

    private void apply(ItemAddedToCartEvent event) {
        events.add(event);
        items
        .add(new Item(new Product(event.getProductName(),event.getQuantity()));
    }

    private void apply(ItemRemovedFromCartEvent event) {
```

```java
        events.add(event);
        items.remove(this.items.stream()
        .filter(item->item.getProductName().equals(event.getProductName()))
        .findFirst().get());
    }

    public Set<String> removedProductNames() {
        return events.stream()
                .filter(event -> event instanceof ItemRemovedFromCartEvent)
                .map(event -> ((ItemRemovedFromCartEvent) event)
                .getProductName()).collect(Collectors.toSet());
    }

}

package com.ddd_bootcamp.domain.events;
public interface DomainEvent {}


package com.ddd_bootcamp.domain.events;

public class ItemAddedToCartEvent implements DomainEvent {
    private String productName;
    private int quantity;

    public ItemAddedToCartEvent(String productName, int quantity) {
        this.productName = productName;
        this.quantity = quantity;
    }

}

package com.ddd_bootcamp.domain.events;

public class ItemRemovedFromCartEvent implements DomainEvent {
    private String productName;

    public ItemRemovedFromCartEvent(String productName) {
        this.productName = productName;
    }

}
```

DDD Concept 1 - Domain Event

Above code shows how we can solve above use case 5 using Domain Events.

Domain Event - Capture something important, that happened in the Domain.

It should be of importance to Domain Experts and Business folks.
Domain Events are immutable. They are named in past tense as event has already happened.

Event cannot be deleted. Event cannot be updated.

Example : Credit and Debit Events in bank ledger.

"Accountants do not use Pencil." - Greg Young in his talk on CQRS and Event Sourcing.

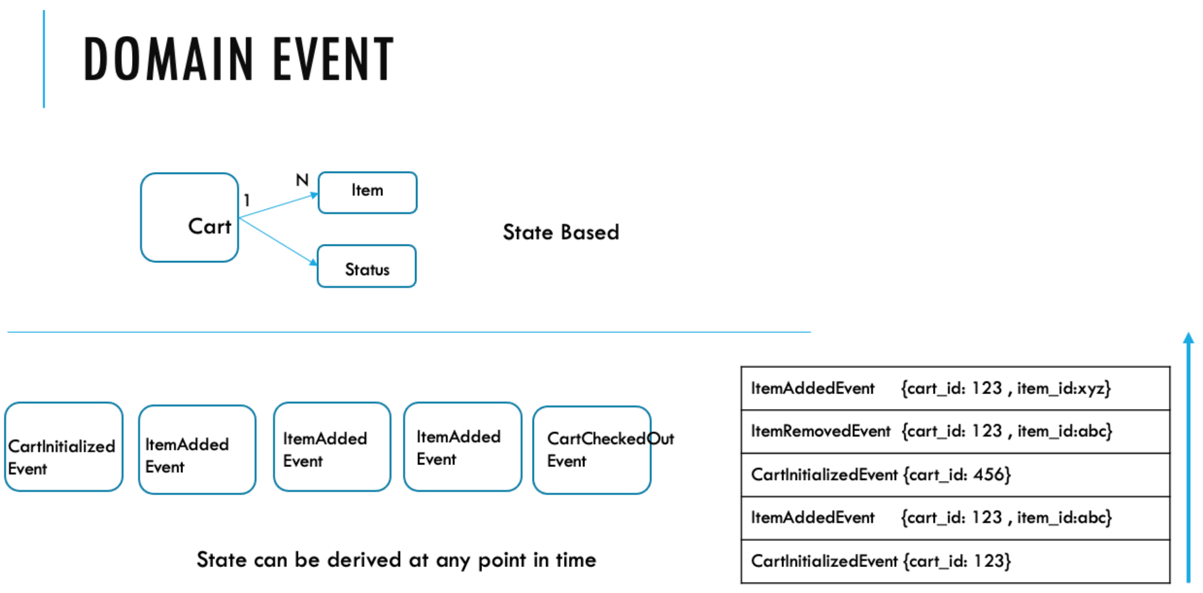
If Cashier makes mistake while paying you…then they will add new credit or debit transaction but will never delete or update existing transaction.

Is CartUpdatedEvent a good Domain Event ? You can check that with Domain experts!
Does name of Event tell intent ? Does it tell what changed on Cart ?

Domain Experts may be confused by name CartUpdatedEvent - Was Item Added?, Was Item Removed?, Was Cart Checked Out?

Domain Events helps with building loosely coupled, scalable systems. More on that in upcoming chapters.

We will see code which uses Domain Events using Axon framework (DDD + CQRS + Event Sourcing) in upcoming chapters.


DOMAIN EVENT
Cart
1
N
Item
Status
State Based
CartInitialized Event
ItemAdded Event
ItemAdded Event
ItemAdded Event
CartCheckedOut Event
State can be derived at any point in time
ItemAddedEvent {cart_id: 123 , item_id:xyz}
ItemRemovedEvent {cart_id: 123 , item_id:abc}
CartInitializedEvent {cart_id: 456}
ItemAddedEvent {cart_id: 123 , item_id:abc}
CartInitializedEvent {cart_id: 123}

Domain Event

Now lets move on to learning next concept in Domain-Driven Design.

In above diagram, model on top can be derived from playing events shown in bottom of diagram, in the order in which they happened.

So top model may loose information when Item is removed from Cart. But bottom model will have akll information anytime needed in future for new business requirements that may come up.

Entity - Understanding building blocks of DDD

Let’s move to understanding other concepts in DDD by solving some use cases.

Use case 6

As a business User, I would like to differentiate between two Carts, even if they contain same Item (both carts have Product “Sony Wireless headphone” with 1 quantity)
–
Note –
Two carts where items are same, equality on Carts should return false
Cart cart1 = Cart();
Cart cart2 = Cart();
Item item1 = new Item(new Product(”Sony Wireless headphone”), 1);
Item item2 = new Item(new Product(”Sony Wireless headphone”), 1);
cart1.add(item1);
cart2.add(item2);
cart1.equals(cart2) ⇒ should return false

```java
package com.ddd_bootcamp.domain;

import com.ddd_bootcamp.domain.events.DomainEvent;
import com.ddd_bootcamp.domain.events.ItemAddedToCartEvent;
import com.ddd_bootcamp.domain.events.ItemRemovedFromCartEvent;

public class Cart implements Entity<Cart> {

    private CartId cartId;
    private List<DomainEvent> events = new ArrayList<>();
    private List<Item> items = new ArrayList<>();

    public Cart() {
        cartId = CartId.generateCartId();
    }

    public void add(Item item) {
        ItemAddedToCartEvent itemAddedEvent =
                new ItemAddedToCartEvent(item.getProductName(),
                        item.getQuantity());

        apply(itemAddedEvent);
    }


    public void remove(Item item) {
        ItemRemovedFromCartEvent itemRemovedFromCartEvent =
                new ItemRemovedFromCartEvent(item.getProductName());

        apply(itemRemovedFromCartEvent);
    }

}
```

DDD Concept 2 - Entity

Cart is an Entity.
Definition of Entity taken from Eric Evans DDD Blue book-
"Many objects are not fundamentally defined by their attributes but rather by thread of continuity and Identity"[1]
"Entity has life cycle and even passes through multiple forms"[2]
"Object defined primarily by its identity is called an Entity"[3]
A person, city, car, lottery ticket or bank transaction.[4]
Seat booking in stadium. Seat and Attendee as Entities.[5]
If it is general admission, then Seat need not be Entity.[6]
Thread of continuity means that there is some life cycle and
thus a state machine associated with Cart e.g. Items are added to Cart, they are removed from Cart and finally Cart is checkedout. Thus there is lifecycle associated with it.
Various Strategies for generating Identity for entities
e.g., Amazon order id, SSN in US, etc.
Identity of Entity will be governed by business requirments.
e.g. Amazon order id should be human readable so that user while taking to Amazon customer care can read it out easily.
Imagine if we used UUID for amazon order id!

Credits and References -
1, 2, 3, 4, 5, 6 - Domain-Driven Design: Tackling Complexity in the Heart of Software Eric Evans (Chapter 5)

Let's move on to next concept in DDD.

Value Object - Understanding building blocks of DDD

Use case 7

Add Price to a Product . We need support for only one currency, say - USD.
–
Note –
For modelling currency - Please use java.util.Currency (Currency.getInstance("USD")) or what your programming language provides, if none just use String for now.
new Product("Apple Pencil", new Price(...));

```java
package com.ddd_bootcamp.domain;

import com.ddd_bootcamp.domain.events.DomainEvent;
import com.ddd_bootcamp.domain.events.ItemAddedToCartEvent;
import com.ddd_bootcamp.domain.events.ItemRemovedFromCartEvent;


public class Cart implements Entity<Cart> {

    private CartId cartId;
    private List<DomainEvent> events = new ArrayList<>();
    private List<Item> items = new ArrayList<>();

    public Cart() {
        cartId = CartId.generateCartId();
    }

    public void add(Item item) {
        ItemAddedToCartEvent itemAddedEvent =
                new ItemAddedToCartEvent(item.getProductName(),
                        item.getQuantity(), item.getProductPrice());

        apply(itemAddedEvent);
    }

    public void remove(Item item) {
        ItemRemovedFromCartEvent itemRemovedFromCartEvent =
                new ItemRemovedFromCartEvent(item.getProductName());

        apply(itemRemovedFromCartEvent);
    }

    //....

}

package com.ddd_bootcamp.domain;

import java.math.BigDecimal;
import java.util.Currency;

public class Price implements ValueObject<Price> {
    private BigDecimal value;
    private Currency currency;

    public Price(BigDecimal value, Currency currency) {
        this.value = value;
        this.currency = currency;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        Price price = (Price) o;

        if (!value.equals(price.value)) return false;
        return currency.equals(price.currency);
    }

    @Override
    public int hashCode() {
        int result = value.hashCode();
        result = 31 * result + currency.hashCode();
        return result;
    }

    @Override
    public boolean sameValueAs(Price other) {
        if (this == other) return true;
        if (!value.equals(other.value)) return false;
        return currency.equals(other.currency);
    }
}
```

DDD Concept 3 - Value Object

Price is Value Object.
Definition of Value Object taken from Eric Evans DDD Blue book -
"Many objects have no conceptual identity. These objects describe the characteristic of the thing."[1]
E.g., two markers of same color and same shape. If one is lost, a drawing can continue by replacing it with another marker.[2]
"Value objects describe the things. They are elements of design that we care about only for what they are and not who or which they are."[3]
Design Value Object as Immutable (cannot be changed once created) whenever possible. This is not mandatory. There might be cases where Value object needs to be mutable.
Credits and References -
1, 2, 3 - Domain-Driven Design: Tackling Complexity in the Heart of Software Eric Evans (Chapter 5)

Is it value object or entity

Depends on business requirement (Domain you are in) Who is asking ?
100 Dollar Bill For Central Bank /For You?
100 Dollar Bill – Value Object for you. Same 100 Dollar Bill may be Entity for Central bank,
they would care about serial No, which printing press it was printed at, etc.
Example of Seat booking in stadium. Seat and Attendee as Entities.
If it is general admission, then Seat will be Value Object. [1]

Credits and References :
1 - Domain-Driven Design: Tackling Complexity in the Heart of Software Eric Evans (Chapter 5)

Domain Service - Understanding building blocks of DDD

Use case 8

As a Business User, I would like to price my product 10% below competitor price (competitor price is available for product) .

–

Note – Assume that HashMap/Dictionary of Competitor Product Name and price is available. Competitor Product name matches 1 to 1 with our Product Name.
Please do not create pricing discount logic inside Product class. Pass discounted price while creating Product.
new Product("Apple Pencil",Price(discountedPrice, Currency.getInstance("USD")));
new Product("Apple Pencil",Price(125, Currency.getInstance("USD"))) – 125 is already discount price

```java
package com.ddd_bootcamp.domain.domain_service;

import com.ddd_bootcamp.domain.Price;

import java.math.BigDecimal;
import java.util.Currency;

public class CompetitorBasedPricer {

    static Map<String, Price> competitorPrices = new HashMap();
    private static int discountPercentage = 10;

    static {
        competitorPrices.put("Apple Pencil",
        new Price(new BigDecimal(100), Currency.getInstance("USD")));

        competitorPrices.put("Sony Wireless headphone",
        new Price(new BigDecimal(10), Currency.getInstance("USD")));
    }

    public static Price getPrice(String productName) {
        Price price = competitorPrices.get(productName);
        return price.reduceByPercent(discountPercentage);
    }
}

package com.ddd_bootcamp.domain;

import java.math.BigDecimal;
import java.util.Currency;
import java.util.Objects;

public class Price implements ValueObject<Price>{
    private BigDecimal value;
    private Currency currency;

    public Price(BigDecimal value, Currency currency) {
        this.value = value;
        this.currency = currency;
    }

    public Price reduceByPercent(int percentage) {
        return new Price(value.subtract(value.divide(new BigDecimal(percentage))), C\
urrency.getInstance("USD"));
    }

    //...
}
```

DDD Concept 3 - Domain Service

Sometimes it is not just a thing or not a noun. You can model verbs or business processes as Domain Service.

1. CompetitorBasedPricer.
2. Fund transfer from One Bank Account to Other Bank Account[1]

Transfer functionality -

```
class FundTransfer  {
    public static transfer(Account from, Account to)  {/...}
}
```

Bank Account Entity :

```
class Account {
    public void credit() {}
    public void debit()  {}
}
```

Credits and References :
1 - Domain-Driven Design: Tackling Complexity in the Heart of Software Eric Evans (Chapter 5)

Please Note that Domain Service is not an application service.

Domain Service resides in Domain Layer and is part of Domain Model. We can create a package called domain_service and put it inside domain package.

Application Service should not have business logic. Application Service is just a Adapter outside Domain Layer in Hexagonal architecture, and a layer in 3-tier or n-tier architecture outside Domain layer.

If some behaviour is not naturally fitting into Entity then only consider using Domain Service.

Refactoring/Improving existing codebase or existing microservices

1. Look for domain/business logic which has leaked into application services or controllers or data access/repository layers.
2. Move logic to Domain Layer/Domain Model using Entity, Value Object, and Domain Services. Use Language of Domain which is already known to Domain experts for naming Entity, Value Object and Domain Services and even Domain Events.
3. Use of Domain Events is optional, you can do DDD without using Domain Events as well.
4. Look for refactoring to use Domain Language used by Domain Experts (Product Owner, Domain Expert, Domains Specialist, Business Analyst, etc)
5. Try to draw a picture or diagram of Domain Model of your existing codebase, app, or microservice.

6. Domain Layer/ Domain Model classes under domain package should not talk to anything external like databases, file systems, other microservices, etc.

7. Use technique like Strangler Fig Pattern to slowly change one microservice at a time to use Domain Driven Design [1]

Credits and References:
1 https://martinfowler.com/bliki/StranglerFigApplication.html

Recap of what we covered so far

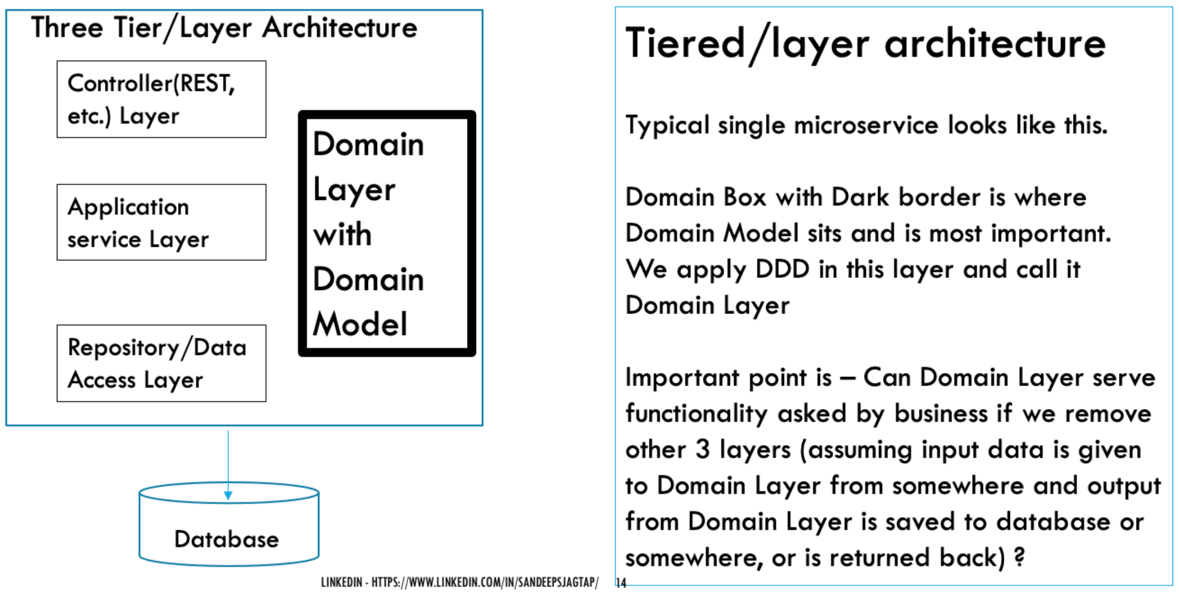
- DDD is iterative and embraces XP. Model/Code/Design evolves over a period of time as we know more about domain.
- DDD can be implemented in OO or functional paradigm
- Domain Layer in 3 tier and Hexagonal architecture
- Domain Model – Domain Model resides in Domain Layer
- Domain Event
- Entity
- Value Object
- Domain Service

What is Domain Layer / Domain Model

Lets try to understand what a Domain Model or Domain Layer is.

The code used in previous chapter with java’s package com.ddd_bootcamp.domain makes a Domain layer/ Domain Model.

We all are familiar with three tier/n-tier/layered architecture which is used in monolith, microservices or SOAP based services.


Three Tier/Layer Architecture
Controller(REST, etc.) Layer
Application service Layer
Repository/Data Access Layer
Domain Layer with Domain Model
Database
LINKEDIN - HTTPS://WWW.LINKEDIN.COM/IN/SANDEEPSJAGTAP/
14

Tiered/layer architecture

Typical single microservice looks like this.

Domain Box with Dark border is where Domain Model sits and is most important. We apply DDD in this layer and call it Domain Layer

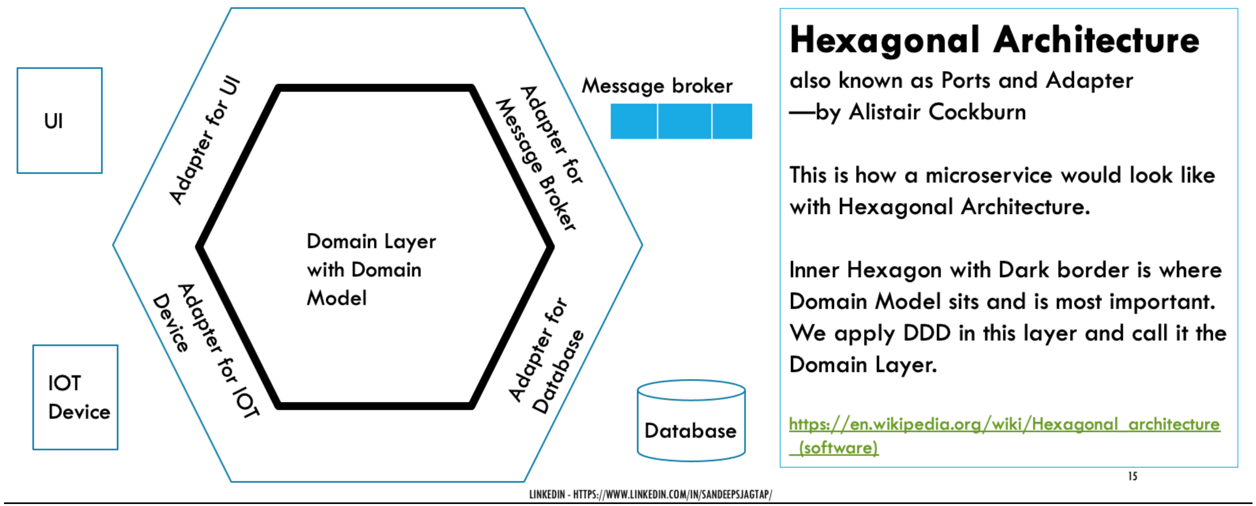
Important point is – Can Domain Layer serve functionality asked by business if we remove other 3 layers (assuming input data is given to Domain Layer from somewhere and output from Domain Layer is saved to database or somewhere, or is returned back) ?

Three tier architecture

The most important layer is the Domain Layer here, the vertical box in the diagram above. A good litmus test for the current code base is if we remove Controller, Application Services/Facades and Repository/DAO layer, are we still able to achieve functionality that Business people want us to implement.

If our Domain Layer is able to represent all functionality needed by business given we provide Domain Layer needed input data. Domain Layer is what Domain Driven Design focuses on. Domain Layer takes the data given and then does necessary things to achieve business functionality, Domain Layer should not care about what modified state is written to the database or something else like Kafka messaging system.
Domain layer takes input and does NOT care about where input data came from.Domain layer returns data and does NOT care where that data is being saved or sent to e.g DB, Message broker

There is a popular architecture style called Hexagonal Architecture also known as Ports and Adapters.


UI
Adapter for UI
Adapter for Message Broker
Message broker
Domain Layer with Domain Model
Adapter for IOT Device
IOT Device
Adapter for Database
Database
Hexagonal Architecture
also known as Ports and Adapter
—by Alistair Cockburn
This is how a microservice would look like with Hexagonal Architecture.
Inner Hexagon with Dark border is where Domain Model sits and is most important. We apply DDD in this layer and call it the Domain Layer.
https://en.wikipedia.org/wiki/Hexagonal_architecture_(software)
15
LINKEDIN - HTTPS://WWW.LINKEDIN.COM/IN/SANDEEPSJAGTAP/

Hexgonal architecture

The most important layer is the Domain Layer here and is represented by inner hexagon. We can keep adding new adapter as needed. If we want to sent domain events to Kafka then we can write new adapter. Adapter in hexagonal architecture as similar to application service or respository layers in three tier/n-tier/layered architecture.

DDD helps with UI as well Database not influensing the design of domain layer/model.

Each microservice can use Layered Architecture or Hexagonal Architecture.

In CQRS architecture style (which we will look at in upcoming chapters),we have Write side Domain model and Read side Model.
Read model is influenced by UI requirements/Consumers.
Write side model should be not be influenced by UI and Database.

Understading Aggregates and relationship to microservices

Lets try to understand what aggregates are and how they are related to microservices using following code examples.

Use case 9

Create Order(with Products) when Cart is checked out. Also, Mark Cart as checked out .
–
Note –
While Creating Order please do not use Item class but use Product class. Flatten out products in Item, that means, if Cart has Item with Product "Apple Pencil" and Quantity two, then create two Product objects for "Apple Pencil" and add them to Order's product list.
Order order = new Order(List<Product> products)

Possible Solutions:

```java
 package com.ddd_bootcamp.domain;

import com.ddd_bootcamp.domain.events.*;

public class Cart implements Entity<Cart> {

    private CartId cartId;
    private List<DomainEvent> events = new ArrayList<>();
    private List<Item> items = new ArrayList<>();
    private boolean isCheckedOut;

    public Cart() {
        cartId = CartId.generateCartId();
    }

    public void add(Item item) {
        ItemAddedToCartEvent itemAddedEvent =
                new ItemAddedToCartEvent(item.getProductName(),
                        item.getQuantity(), item.getProductPrice());

        apply(itemAddedEvent);
    }

    public void remove(Item item) {
        ItemRemovedFromCartEvent itemRemovedFromCartEvent =
                new ItemRemovedFromCartEvent(item.getProductName());

        apply(itemRemovedFromCartEvent);
    }

    public Order checkOut() {
        List<CartItem> cartItems = items.stream().map(item ->
                new CartItem(item.getProductName(),
                        item.getProductPrice(),
                        item.getQuantity())).collect(Collectors.toList());

        apply(new CartCheckedOutEvent(cartItems));

        List<Product> products = items.stream().flatMap(item ->
                item.getFlattenedProducts().stream()).collect(Collectors.toList());
        return new Order(products);
    }

    public void checkOut1() {
        List<CartItem> cartItems = items.stream().map(item ->
                new CartItem(item.getProductName(),
                        item.getProductPrice(),
                        item.getQuantity())).collect(Collectors.toList());

        apply(new CartCheckedOutEvent(cartItems));
    }
    //....

    private void apply(CartCheckedOutEvent event) {
        events.add(event);
        this.isCheckedOut = true;
    }

}

package com.ddd_bootcamp.domain;

```

```
import java.util.UUID;

public class CartId {
    private UUID randomUUID;

    public static CartId generateCartId() {
        return new CartId(UUID.randomUUID());
    }

    private CartId(UUID randomUUID) {
        this.randomUUID = randomUUID;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        CartId cartId = (CartId) o;
        return randomUUID.equals(cartId.randomUUID);
    }

    @Override
    public int hashCode() {
        return randomUUID.hashCode();
    }
}

package com.ddd_bootcamp.domain;

import java.util.List;

public class Order {
    private List<Product> products;

    public Order(List<Product> products) {
        this.products = products;
    }
}

package com.ddd_bootcamp.domain.domain_service;

import com.ddd_bootcamp.domain.Cart;
```

```java
import com.ddd_bootcamp.domain.Item;
import com.ddd_bootcamp.domain.Order;
import com.ddd_bootcamp.domain.Product;

import java.util.List;
import java.util.stream.Collectors;

public class CheckOutService {

    public static Order checkOut(Cart cart) {
        cart.checkOut1();
        List<Item> items = cart.getItems();
        List<Product> products = items.stream().flatMap(item ->
                item.getFlattenedProducts().stream()).collect(Collectors.toList());
        return new Order(products);
    }
}

package com.ddd_bootcamp.application;

import com.ddd_bootcamp.domain.Cart;
import com.ddd_bootcamp.domain.Item;
import com.ddd_bootcamp.domain.Price;
import com.ddd_bootcamp.domain.Product;
import com.ddd_bootcamp.domain.domain_service.CheckOutService;
import com.ddd_bootcamp.domain.domain_service.CompetitorBasedPricer;

import java.math.BigDecimal;
import java.util.Currency;
import java.util.List;

public class Application {
    public static void main(String[] args) {
        Cart cart = new Cart();
        Product headphone = new Product("Sony Wireless headphone",
                new Price(BigDecimal.valueOf(10), Currency.getInstance("USD")));
        Item headphoneItem = new Item(headphone, 1);
        cart.add(headphoneItem);
        Product applePencil = new Product("Apple Pencil",
                new Price(BigDecimal.valueOf(100), Currency.getInstance("USD")));
        Item applePencilItem = new Item(applePencil, 2);
        cart.add(applePencilItem);
        System.out.println("Cart checked out = " + cart.checkOut());
        //System.out.println("Cart checked out = " +    CheckOutService.checkOut(car\
t));

}
```

Let's switch to other use case - different domain (Banking)

Use case 10

Customer and Bank Account. (Not related to e-commerce domain)
When Customer's Address is updated, update all her Bank Accounts addresses as well. (This is invariant, or business rules or consistency rule given by business)
–
Note –
Customer is Entity. Account is Entity.
Customer has List of bank accounts.
Customer has Address. Account has Address
Address has one attribute called city.
Address is Value Object
– customer.updateAddress(new Address("Mumbai"))

Possible Solutions:

```
package com.ddd_bootcamp.domain;

import java.util.ArrayList;
import java.util.List;
import java.util.UUID;

/**
 *
 * Code Problem 10:
 * Customer and Bank Account. ( Not related to e-commerce domain)
 * When Customer's Address is updated, update all her Bank Accounts addresses as wel\
l.
 * ( This is invariant, or business rules or consistency rule given by business)
 */

/**
 *
 * Database transactions -  ACID
 *
 * ACID =>  Entity says I will take care of  AC ( Atomic and Consistent).
 * then
 * Entity -> upgrades to -> Aggregate
 *
 * Aggregate and Aggregate root are Customer
 *
 * Aggregate root - Entity which is at top
 *
 *
 */
public class Customer {
    private UUID customerId;
    private Address address;
    private List<Account> accounts = new ArrayList<>();

    public Customer(Address address) {
        this.customerId = UUID.randomUUID();
        this.address = address;
    }

    public void addAccount(Account account) {
        accounts.add(account);
        account.updateAddress(this.address);
    }

    //Logical transaction start
    //Physcial transaction is started by Application Service or Adapters.
    //When Customer's Address is updated,
    // update her all Bank Accounts address as well.
    public void updateAddress(Address address) {
        this.address = address;
        accounts.forEach(account -> {
            account.updateAddress(address);
        });
    }
    //Logical transaction end

}

package com.ddd_bootcamp.domain;

import java.util.UUID;

```

```java
//Entity
public class Account {

    private UUID accountId;
    private Address address;

    public Account() {
        this.accountId =  UUID.randomUUID();
    }
    protected void updateAddress(Address address) {
        this.address = address;
    }

}


package com.ddd_bootcamp.domain;

//Value Object
public class Address {
    private String city;

    public Address(String city) {
        this.city = city;
    }
}
```

Aggregates

Let's try to understand what Aggregates are…

Detour - Transactions ACID ⇒
A - ATOMICITY— Either it happens, or nothing happens. When we add 2 quantity of Apple Pencil to Cart, then either both quantities get added or nothing happens.

C - CONSISTENCY— Transaction does not negatively affect/corrupt existing data. All business rules are satisfied when we add 2 quantity of Apple Pencil to Cart ; e.g., total price of Cart would reflect price of 2 quantity of Apple Pencil which was added to Cart.

I - ISOLATION— How two transactions see data if they are changing same thing.
D - DURABILITY— When process/memory dies, Data is stored to disk.

ACID – Strong Consistency ⇒ Given I & D , when we apply A & C to Entity, then it becomes Aggregate.

When Entity takes addtional responsibility of managing atomicity and consistency, then it becomes Aggregate.

Customer was Entity and it become aggrregate as it is managing it own Address Value Object and List of Account Entities.

Account still remains as Entity as it is managed by Customer Aggregate.

Definition of Aggregate taken from Eric Evans DDD Blue book:
"Objects have life cycles" [1]
Challenge: "Maintaining Integrity/invariants throughout life cycle" [2]
"Aggregate defines clear ownership and boundaries" [3]
"Aggregate is cluster of associated object that we treat as a unit for purpose of data changes" [4]
E.g. Cart Entity.
"Aggregate root is at top and only entity through which Aggregate can be accessed and has global identifier" [5]
"Other Objects inside aggregate can have local identifier and are NOT accessible outside aggregate directly. Aggregate root controls access to them from outside world" [6]
"Invariants/consistency rules applied within Aggregate will be enforced with the completion of each transaction" [7]
"Aggregate is boundary of atomicity" [8]

E.g. - Cart total price should match with sum of all items in the cart. Thus car is an Aggregate.

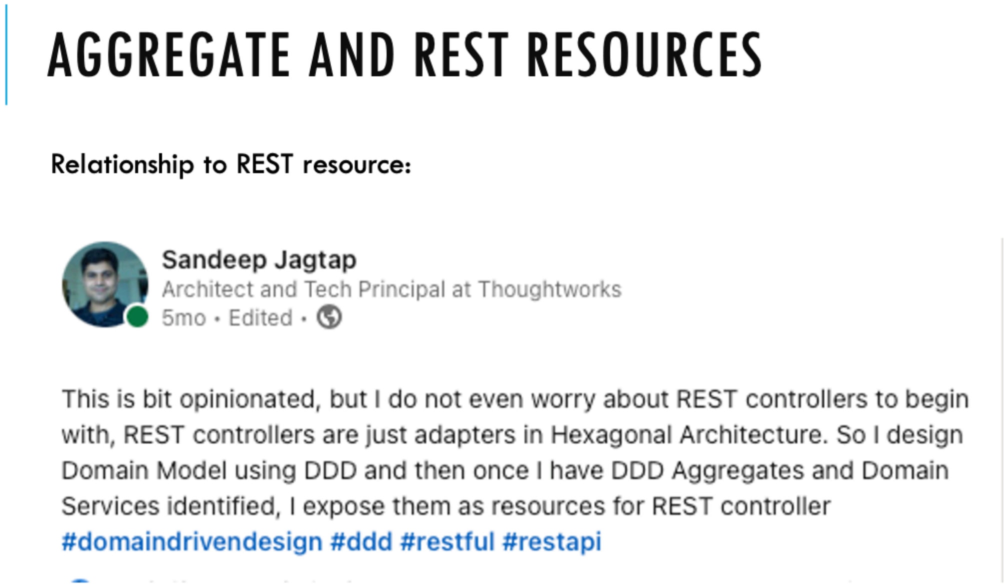
Credits and References -
1 to 8 - Domain-Driven Design: Tackling Complexity in the Heart of Software Eric Evans (Chapter 6)

Relationship to microservices

When we find an Aggregate, we have potentially found a new microservice
(Rough guideline to use)

Also when we find a Domain Service, we have potentially found a new microservice (Rough guideline to use)

And we get REST resources for free!



AGGREGATE AND REST RESOURCES

Relationship to REST resource:

Sandeep Jagtap
Architect and Tech Principal at Thoughtworks
5mo • Edited •

This is bit opinionated, but I do not even worry about REST controllers to begin with, REST controllers are just adapters in Hexagonal Architecture. So I design Domain Model using DDD and then once I have DDD Aggregates and Domain Services identified, I expose them as resources for REST controller
#domaindrivendesign #ddd #restful #restapi

REST and DDD

We already have domain package design using DDD concept, now let's just add layered architecture on top of it. We can also use Hexagonal Architecture instead of Layered Architecture as well.

Lets see how Customer microservice would look like!

Please check at https://github.com/ddd-workshop-org/ddd-customer-service

```java
package com.ddd_bootcamp.threetier.controller;

import com.ddd_bootcamp.domain.Account;
import com.ddd_bootcamp.domain.Address;
import com.ddd_bootcamp.domain.Customer;
import com.ddd_bootcamp.domain.CustomerId;
import com.ddd_bootcamp.threetier.applicationservice.CustomerAppService;
import com.ddd_bootcamp.threetier.controller.resource.CustomerResource;
import com.ddd_bootcamp.threetier.controller.viewModel.AccountRequest;
import com.ddd_bootcamp.threetier.controller.viewModel.AddressRequest;
import com.ddd_bootcamp.threetier.controller.viewModel.CustomerRequest;
import org.springframework.web.bind.annotation.*;

import java.util.UUID;

@RestController
public class CustomerController {

    private final CustomerAppService customerAppService;

    public CustomerController(CustomerAppService customerAppService) {
        this.customerAppService = customerAppService;
    }

    @PostMapping("/customers")
    public CustomerResource create(@RequestBody CustomerRequest request) {
        System.out.println("request = " + request);

        Customer customer = customerAppService.createCustomer(
                new Customer(new Address(request.getAddressRequest().getCity())));

        return CustomerResource.from(customer);
    }

    @PutMapping("/customers/{customerId}/accounts")
    public CustomerResource createAccount(@RequestBody AccountRequest request, @Path\
Variable String customerId) {

        Customer customer = customerAppService.addAccount(new CustomerId(UUID.fromSt\
ring(customerId)), new Account());

        return CustomerResource.from(customer);
    }

    @PutMapping("/customers/{customerId}/address")
    public CustomerResource updateAddress(@RequestBody AddressRequest request, @Path\
Variable String customerId) {

        Customer customer = customerAppService.updateAddress(new CustomerId(UUID.fro\
mString(customerId)),
                new Address(request.getCity()));

        return CustomerResource.from(customer);
    }

}


package com.ddd_bootcamp.threetier.applicationservice;

```

```
import com.ddd_bootcamp.domain.Account;
import com.ddd_bootcamp.domain.Address;
import com.ddd_bootcamp.domain.Customer;
import com.ddd_bootcamp.domain.CustomerId;
import com.ddd_bootcamp.threetier.repository.CustomerRepository;
import org.springframework.stereotype.Service;

import javax.transaction.Transactional;

@Service
public class CustomerAppService {

    private CustomerRepository customerRepository;

    public CustomerAppService(CustomerRepository customerRepository) {
        this.customerRepository = customerRepository;
    }

    @Transactional
    public Customer createCustomer(Customer customer) {
        Customer savedCustomer = customerRepository.save(customer);
        return savedCustomer;
    }

    @Transactional
    public Customer updateAddress(CustomerId customerId, Address address) {
        Customer customer = customerRepository.find(customerId);
        customer.updateAddress(address);
        return customerRepository.save(customer);
    }

    @Transactional
    public Customer addAccount(CustomerId customerId, Account account) {
        Customer customer = customerRepository.find(customerId);
        customer.addAccount(account);
        return customerRepository.save(customer);
    }
}


package com.ddd_bootcamp.threetier.repository;

import com.ddd_bootcamp.domain.Customer;
import com.ddd_bootcamp.domain.CustomerId;
import org.springframework.stereotype.Repository;

import java.util.HashMap;
import java.util.Map;

//Customer Repository or Data Access Layer
//Using in-memory map instead of db for demo
@Repository
public class CustomerRepository {

    private Map<CustomerId, Customer> customerMap = new HashMap<>();

    public Customer save(Customer customer) {
        System.out.println("in Repository customer = " + customer);
        customerMap.put(customer.getCustomerId(), customer);
        return customer;
    }

    public Customer find(CustomerId customerId) {
        return customerMap.get(customerId);
    }
}
```

Splitting Aggregates

Consider the following made-up situation :

We are a very successful bank after 1 year, with millions of Customers and Accounts.

We use Postgres db server and have already used vertical scaling (more RAM,CPU) and partitioning. As now we have billions of records in Customer and Account tables, we see that transaction which goes across above two tables is very very slow (customer.updateAddress() method)

Business and End users are not happy.

What are our options ?

Choices
1. Use two Postgres database servers one for customer tables, another for accounts table. Now we need to use distributed transaction (2 phase commit)

2. Use Domain Event from Customer Aggregate. Customer aggregate does not deal with address update for Accounts. Account becomes its own aggregate.
3. No Domain Events, but customer microservice calls account microservice via http or other similar mechanisms, Each microservice has its own Postgres database server
4. Other options …

With Option 2 and 3 above, we end up Spitting Customer Aggregate into Customer Aggregate and Account Aggregate.

So we get two microservices - customer-microservice and account-microservice.

Please refere to following code

Customer-microservice with Customer as Aggregate and account-microservice with Account as Aggregate

https://github.com/ddd-workshop-org/ddd-customer-only-service

and

https://github.com/ddd-workshop-org/ddd-account-service

Other scenarios in which Aggregates may be split

We would need to get an agreement from the Business team as business rules (of keeping customer and their account addresses in sync) will be eventually consistent.

We saw in the above example that for a technical challenge, we had to split an aggregate into two aggregates.

This could also happen due to functional requirements, with new changes coming in, we may have to split an Aggregate.

Bounded Contexts

Use case 11

Calculate Total cost for the Order.
Calculate shipping cost using weight of the Product.
Total cost = cost of all products in order + (weightIngrams of each product *0.1)

One possible solution would be to add weightInGrams to Product class and use the Product class both Cart and Order class. And this is where DDD's bounded context comes into picture.

Please find working code at:

https://github.com/ddd-workshop-org/ddd-sample-uc11

Online ecommerce -
Product in Shopping cart and Product in Product Catalog/Order are different (even if we are talking about same product say, "iPad Pencil").

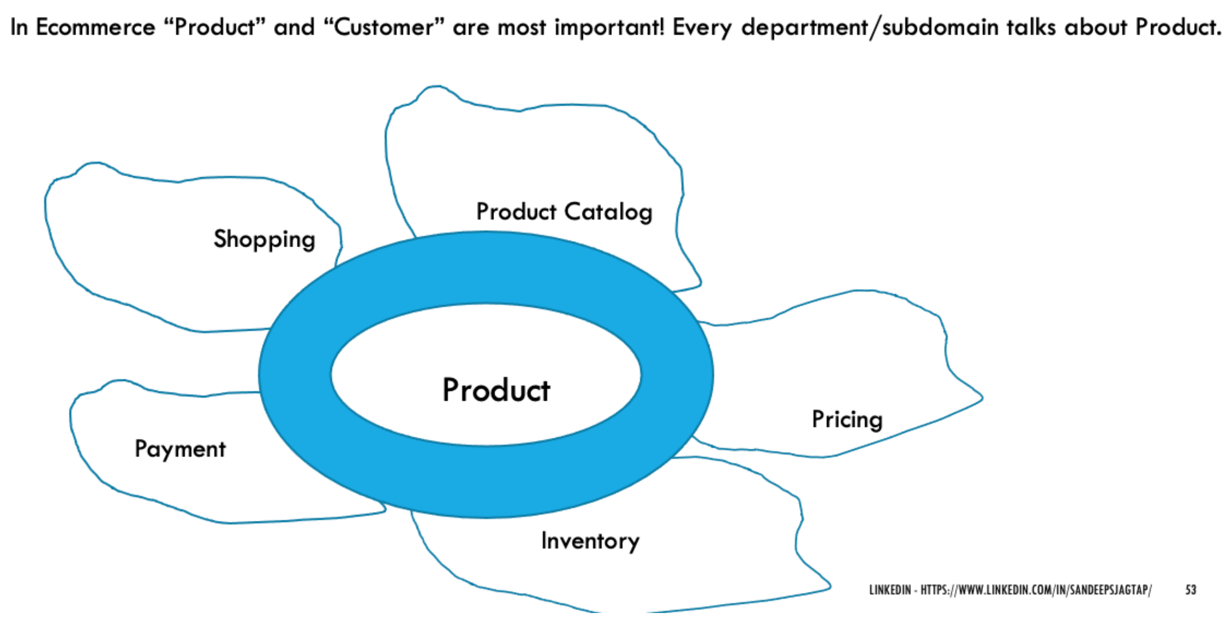
OO - Possibly - Reuse Product class.

DDD - Product in Shopping cart is different than Product in Inventory. You are better off with having two different Product classes.

Bounded context is thus a Linguistic boundary.

"Good fences make good neighbours."
Reference https://en.wikipedia.org/wiki/Mending_Wall

Bounded Context is about boundaries.


In Ecommerce "Product" and "Customer" are most important! Every department/subdomain talks about Product.
Product Catalog
Shopping
Product
Pricing
Payment
Inventory
LINKEDIN - HTTPS://WWW.LINKEDIN.COM/IN/SANDEEPSJAGTAP/
53

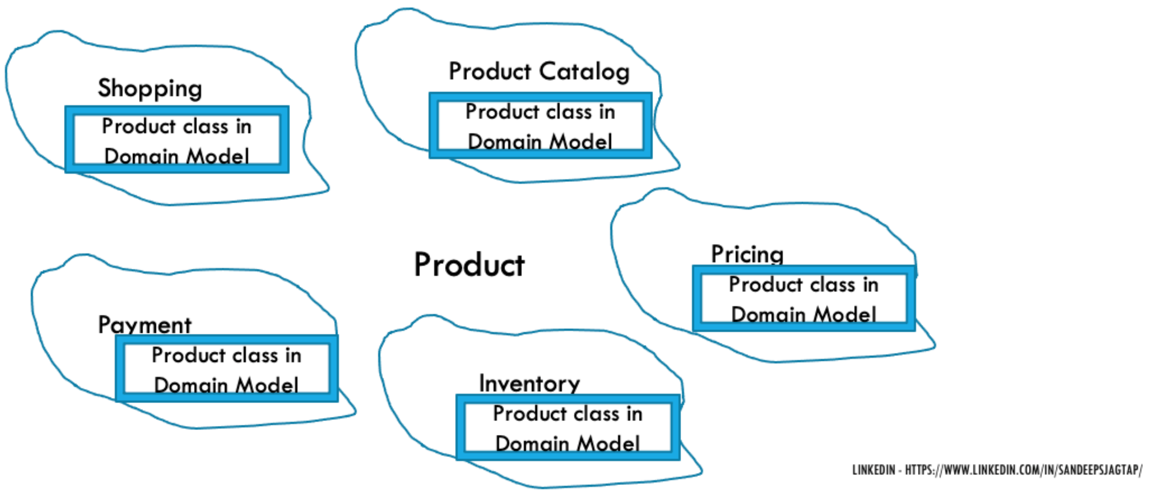
One big model

Shared Product class among Shopping, Shipping, Pricing, Product Catalog, Inventory, & Payment

```
 class Product {

    //methods/behaviors related to Shopping…..

    // methods/behaviors related to Payment…..

    // methods/behaviors related to Product Catalog…..

    // methods/behaviors related to Inventory…..

    // methods/behaviors related to Shipping…..
}
```

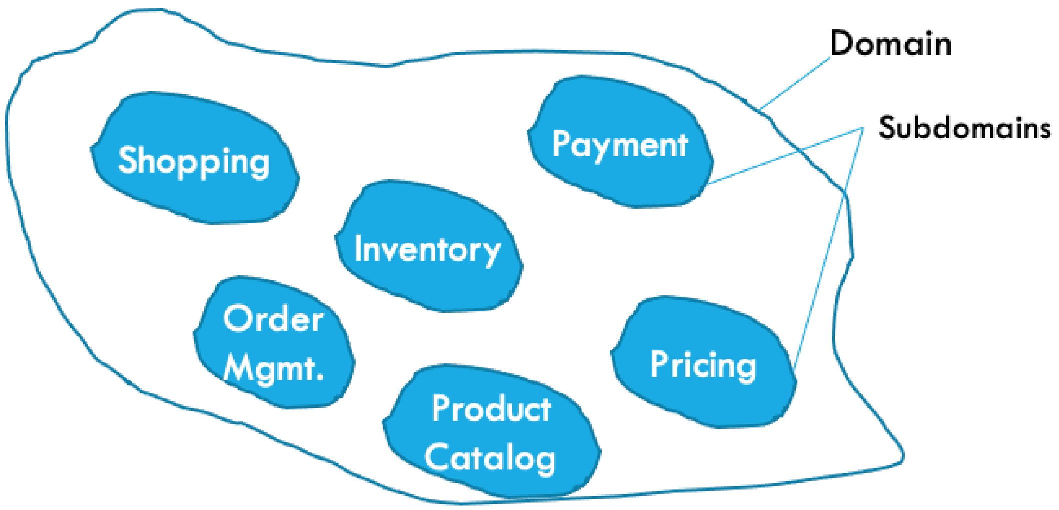
As we can see, one big canonical model for Product class could be a bad design.

Seperate product class in Each bounded context.


Shopping
Product class in Domain Model
Product Catalog
Product class in Domain Model
Product
Pricing
Product class in Domain Model
Payment
Product class in Domain Model
Inventory
Product class in Domain Model
LINKEDIN - HTTPS://WWW.LINKEDIN.COM/IN/SANDEEPSJAGTAP/

better-design-of-product

Subdomains



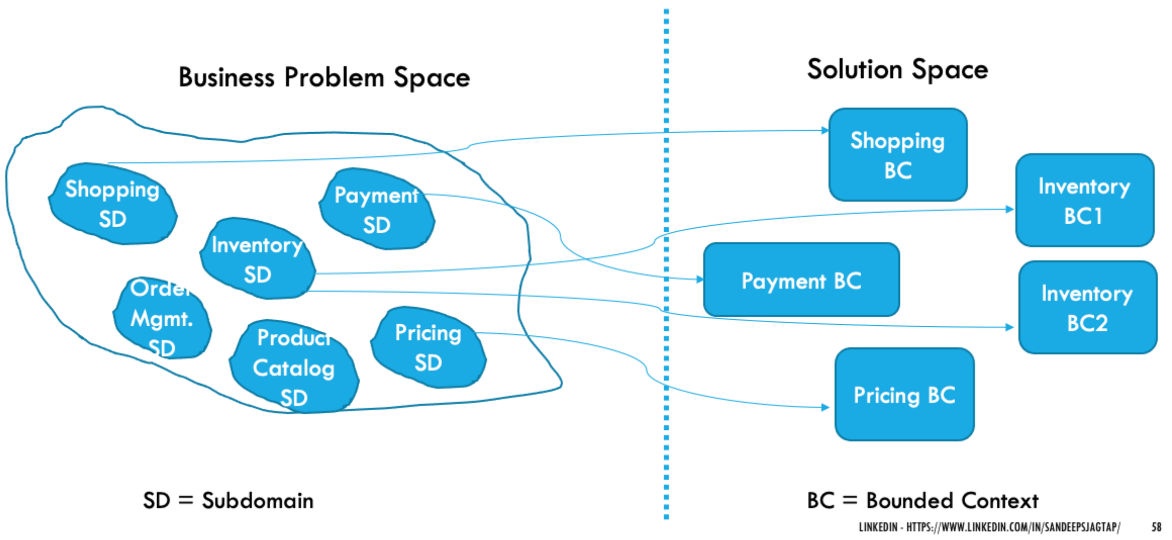
subdomains

E.g., Ecommerce is a Domain.
Then Shopping, Order Mgmt., Inventory, Product Catalog, Pricing, Payment could be subdomains.
Subdomains match roughly to departments in a given organization's Domain.

Three types of Subdomain
Core – Core to business

Supporting – Supporting subdomain like reporting
Generic – Readymade solutions might be available (open source or paid)

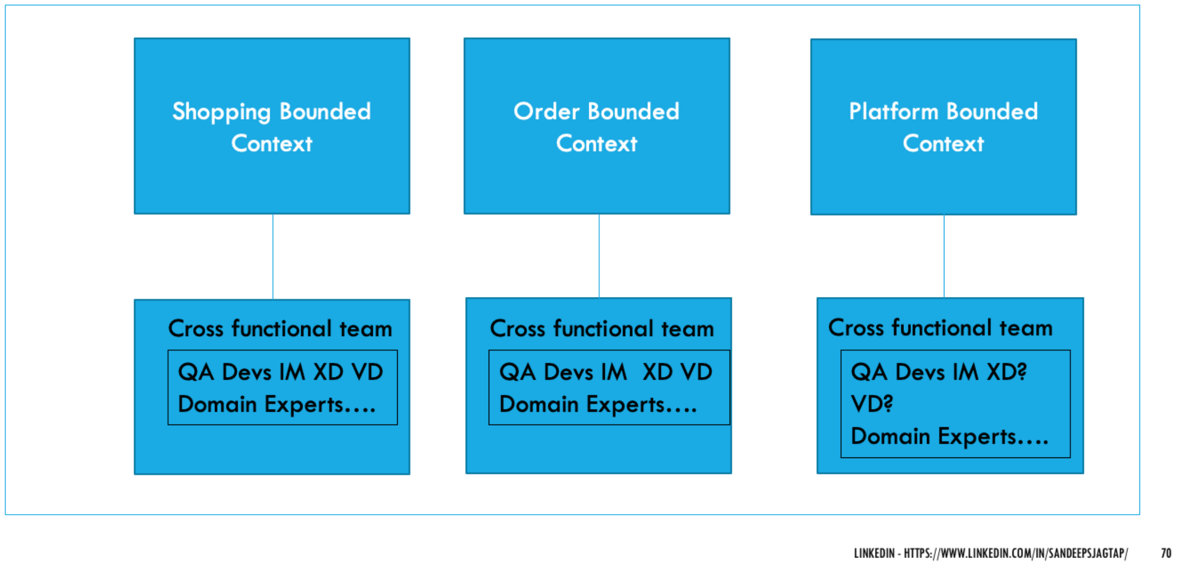
How Subdomain and Bounded Context Map to each other



sd-bc

Bounded Context and team organization

Each bounded context can have its own team


Shopping Bounded Context
Order Bounded Context
Platform Bounded Context
Cross functional team
QA Devs IM XD VD Domain Experts….
Cross functional team
QA Devs IM XD VD Domain Experts….
Cross functional team
QA Devs IM XD? VD? Domain Experts….
LINKEDIN - HTTPS://WWW.LINKEDIN.COM/IN/SANDEEPSJAGTAP/
70

bounded contexts and teams

Bounded Context and its relation to microservices

Bounded Context is linguistic boundary. So each Bounded Context has its own ubiquitous language.

It may happen that you have mutiple aggregates or domain services with same bounded context because ubiquitous language is same.

An Aggregate maps roughly to a microservice, that means a Bounded context with multiple Aggregates can have mutiple microservices each mapping to an aggregate.

Pat Helland in his paper “Life beyond distributed transactions” talks about “Think about Almost-Infinite Scaling of Applications
To frame the discussion on scaling, this paper presents
an informal thought experiment on the impact of
almost-infinite scaling. I assume the number of
customers, purchasable entities, orders, shipments,
health-care-patients, taxpayers, bank accounts, and all
other business concepts manipulated by the application
grow significantly larger over time. Typically, the
individual things do not get significantly larger; we
simply get more and more of them. It really doesn’t
matter what resource on the computer is saturated first,
the increase in demand will drive us to spread what
formerly ran on a small set of machines to run over a
larger set of machines”

In the example which we have, Cart and Order are Aggregates and if we put them in same microservices, then there are chances that someone may end up storing both of them in one ACID transaction to database and with that some day when number of Carts and Orders grow up then we end up with slow performing system. So it is better to map a Aggregate to microservice and better design will be to map Cart Aggregate to one microservice and Order Aggregate to other microservice.

Team Topologies

If you are working on large projects with say more than 30+ people, how to organize teams becomes a bit of a challenge.

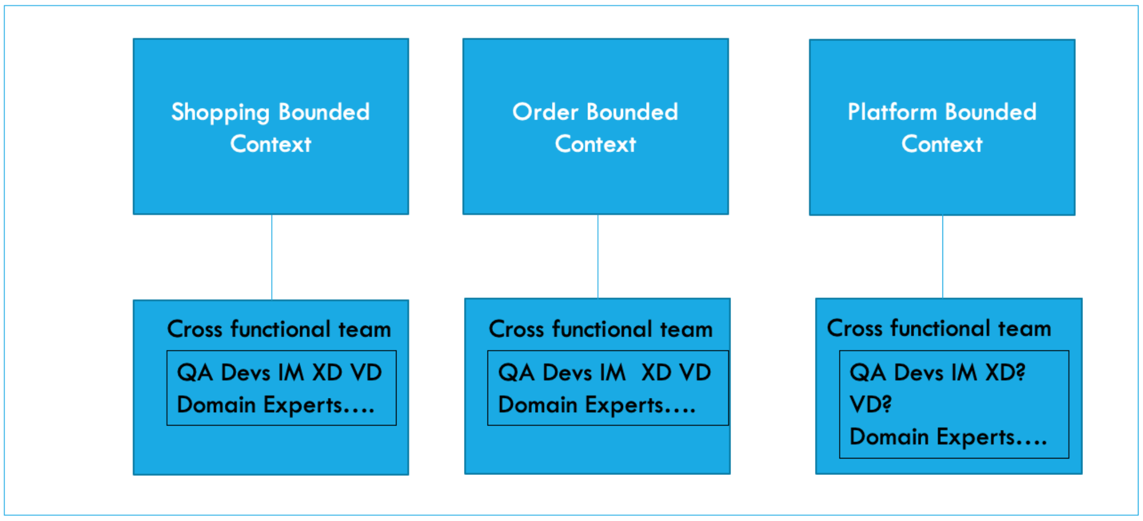
And DDD's bounded context may help solve that.

Once we find a bounded context, we can put a cross-functional team to work on each bounded context. Team Tologies book by Matthew Skelton , Manuel Pais talks about fours types of the teams 1) Stream aligned 2) Platform 3) Complicated subsystem 4) Enabling. It also talks about three types of interaction modes in above 4 team types 1) X-as service 2)Collaboration 3)Facilitating

Cross-functional team working on bounded context is most likely Stream aligned team type in team topologies.
If we are building a Insurance Marketplace platform then we have two domains in the play

1. Insurance Domain - As we are selling various types of insurance products.
2. Technology/Platform domain as we are building the ability to onboard new insurance products from various vendors on this platform in a configurable way(without needing much code development).

The Platform team in diagram below is most likely the Platform team type of Team topologies.



Cross functional Team per bounded context

Ubiquitous Language

Word Ubiquitous means "Universal", but that does not mean entire domain like e-commerce will have just one Ubiquitous Language. That will be bad design as we have seen in previpus chapters.

Developer and Domain Experts talking in business language.

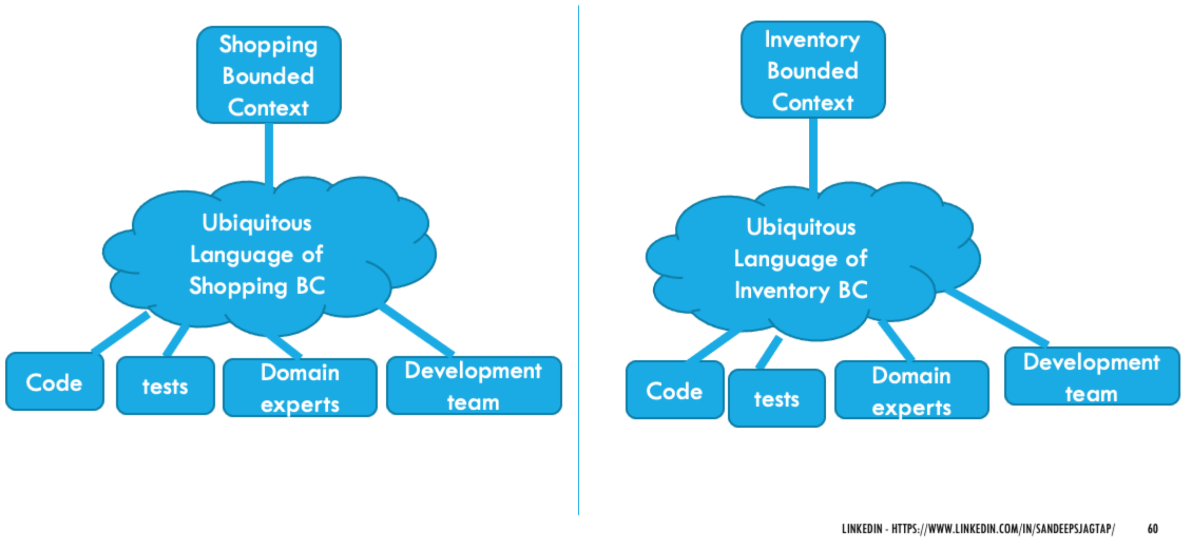
Vocabulary of Ubiquitous language includes name of classes, Domain Events and prominent operations, documentations.

Language includes terms to discuss rules which are made explicit in the design.

Ubiquitous Language applies within bounded context and
NOT across bounded contexts

Bounded contexts exists because they have the Ubiquitous language. When langauge changes we get new bounded context.

Each bounded context has its own Ubiquitous Language.



Shopping Bounded Context
Ubiquitous Language of Shopping BC
Code
tests
Domain experts
Development team
Inventory Bounded Context
Ubiquitous Language of Inventory BC
Code
tests
Domain experts
Development team
LINKEDIN - HTTPS://WWW.LINKEDIN.COM/IN/SANDEEPSJAGTAP/
60

Ubiquitous language per bounded context

Context Maps

Context Map talks about relationship of one Bounded Context (BC) with another BC

Context Map is important as it helps in understanding –

1. Global View of Domain's BCs
2. How one BC can affect deliverables of other BC
3. Helps with understanding dynamics between different BCs. (Organizational issues, etc.)

Relationship Types -

: **Anticorruption Layer**

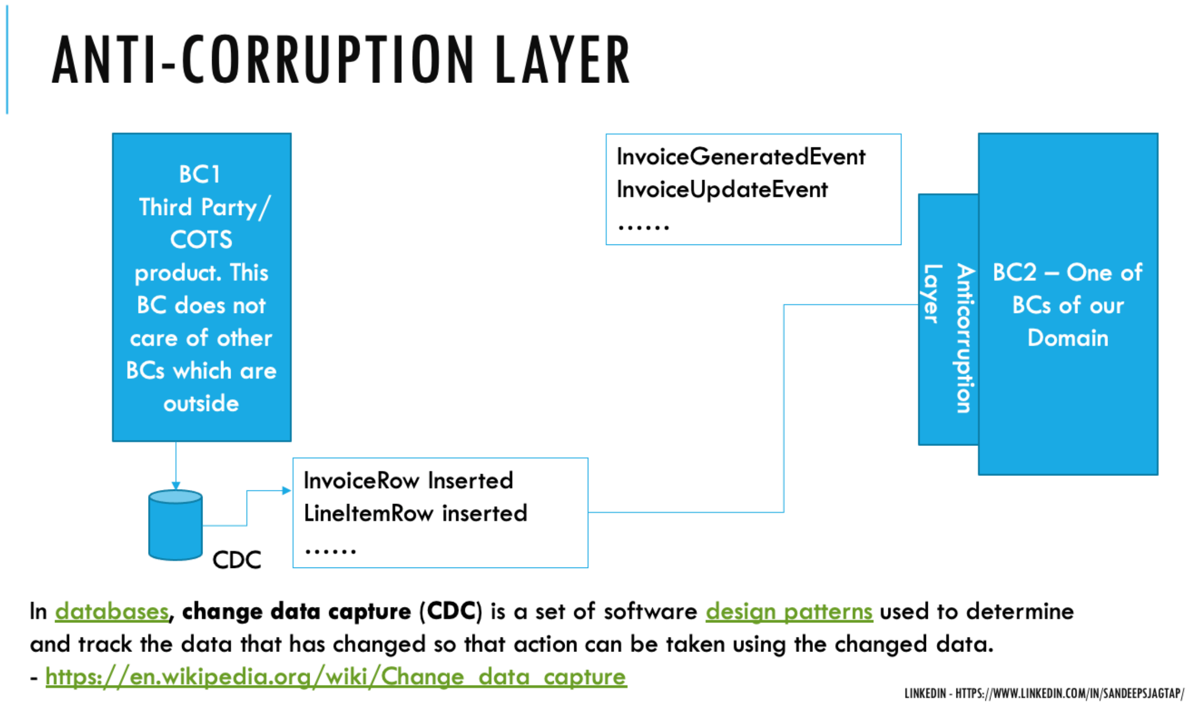
Customer Supplier

Conformist

Published Language

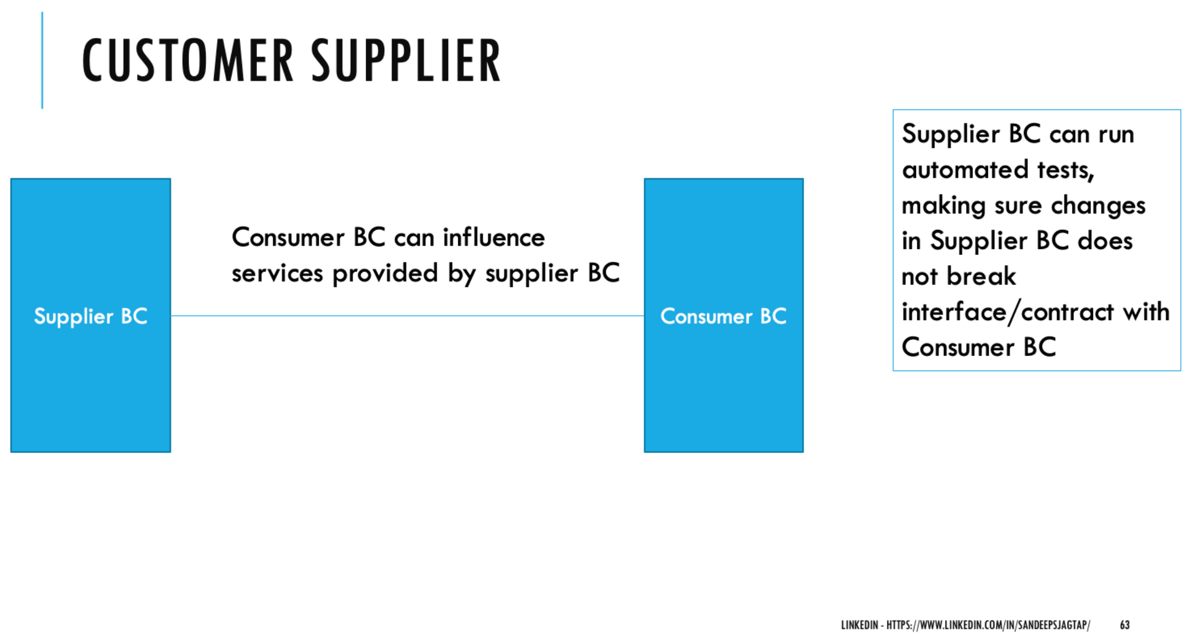
Open Host Service

Shared Kernel


ANTI-CORRUPTION LAYER
BC1 Third Party/ COTS product. This BC does not care of other BCs which are outside
InvoiceGeneratedEvent
InvoiceUpdateEvent
……
Anticorruption Layer
BC2 – One of BCs of our Domain
CDC
InvoiceRow Inserted
LineItemRow inserted
……
LINKEDIN - HTTPS://WWW.LINKEDIN.COM/IN/SANDEEPSJAGTAP/

In databases, **change data capture** (**CDC**) is a set of software design patterns used to determine and track the data that has changed so that action can be taken using the changed data.
- https://en.wikipedia.org/wiki/Change_data_capture

Anticorruption Layer

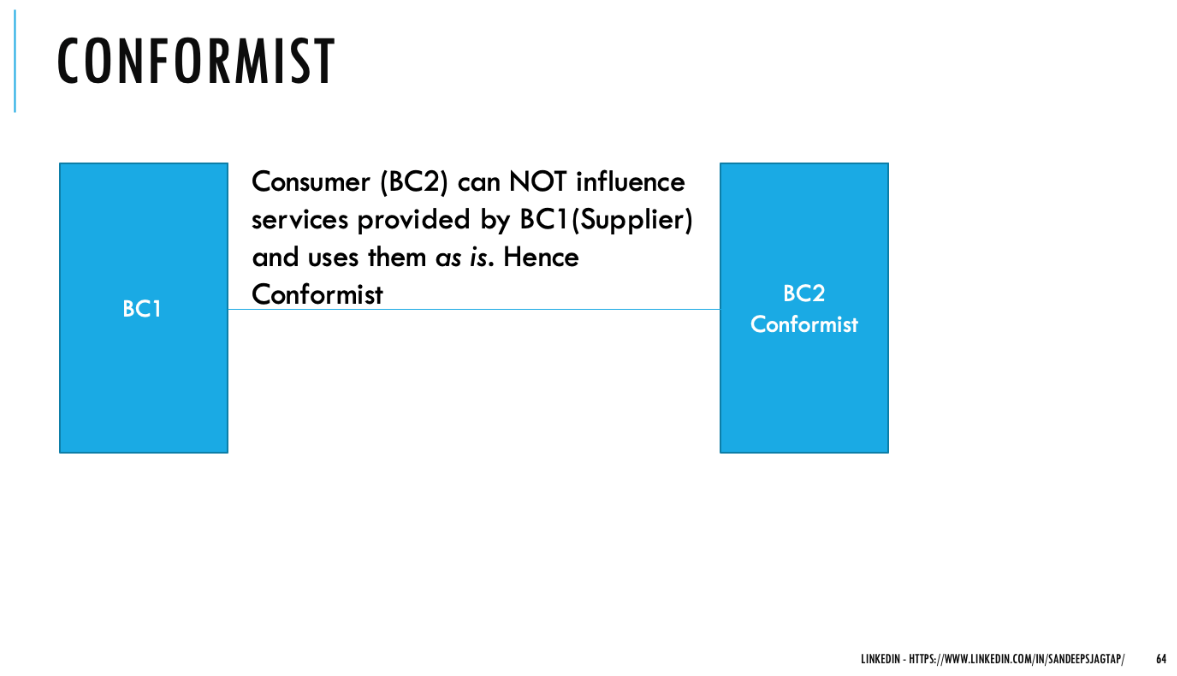


CUSTOMER SUPPLIER

Consumer BC can influence services provided by supplier BC

Supplier BC can run automated tests, making sure changes in Supplier BC does not break interface/contract with Consumer BC

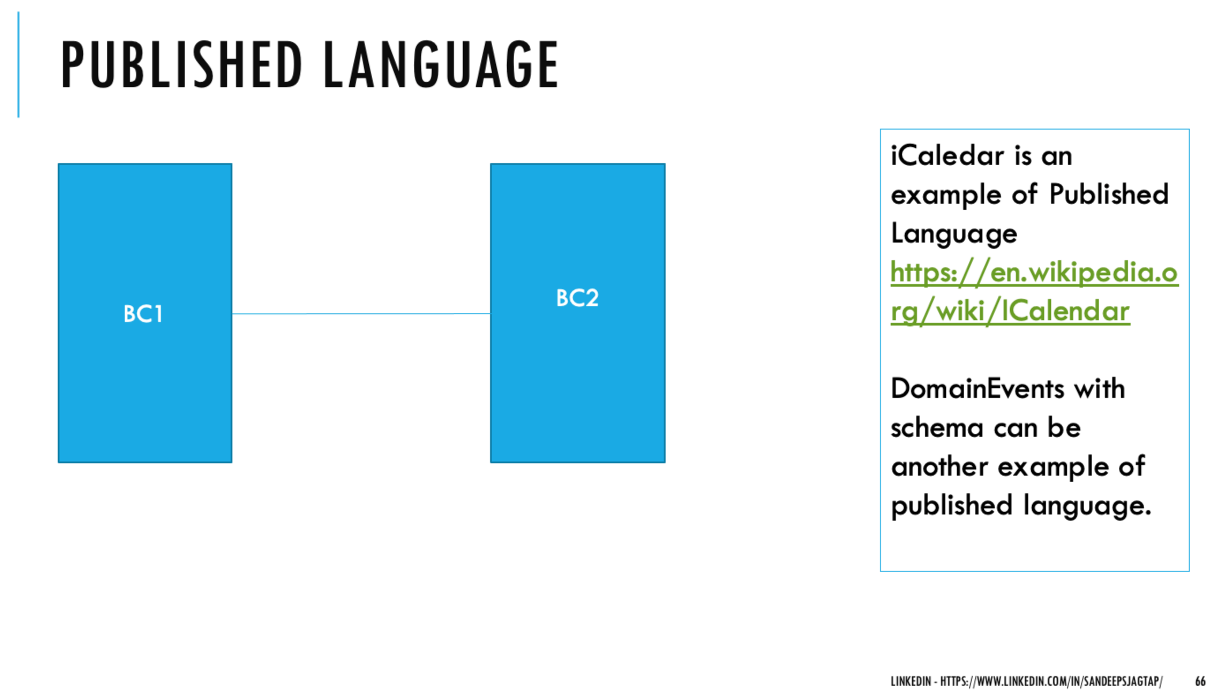
Customer Supplier



CONFORMIST

Consumer (BC2) can NOT influence services provided by BC1(Supplier) and uses them *as is*. Hence Conformist

Conformist

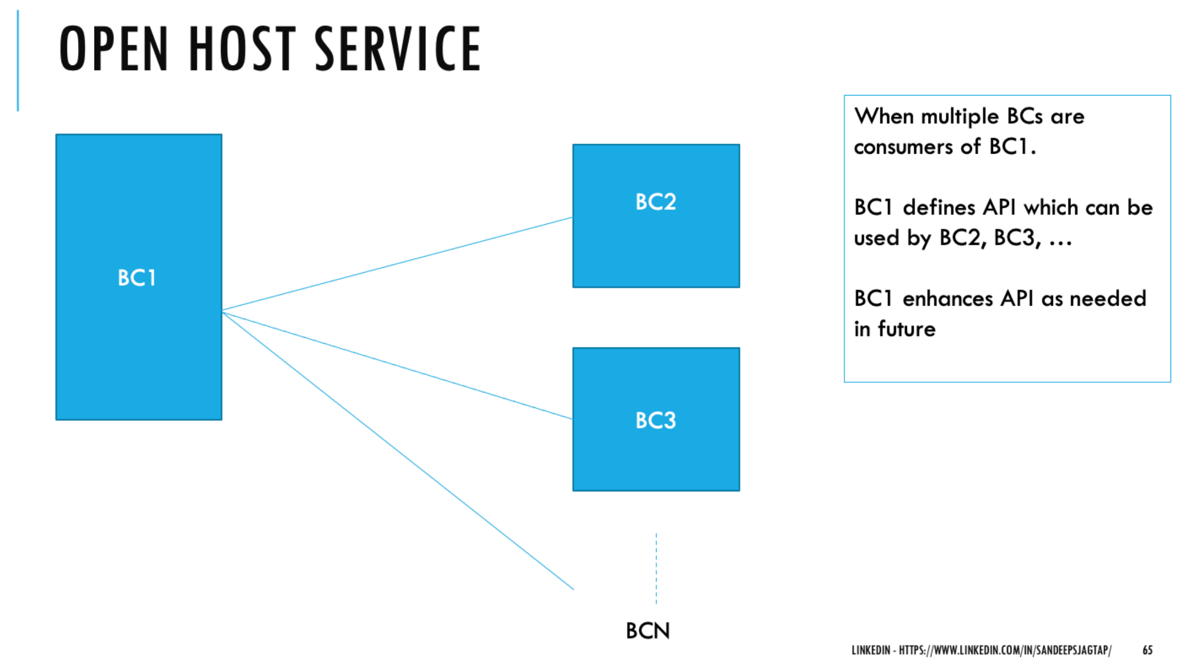

PUBLISHED LANGUAGE

iCaledar is an example of Published Language https://en.wikipedia.org/wiki/ICalendar

DomainEvents with schema can be another example of published language.

LINKEDIN - HTTPS://WWW.LINKEDIN.COM/IN/SANDEEPSJAGTAP/ 66

Published Languag


OPEN HOST SERVICE

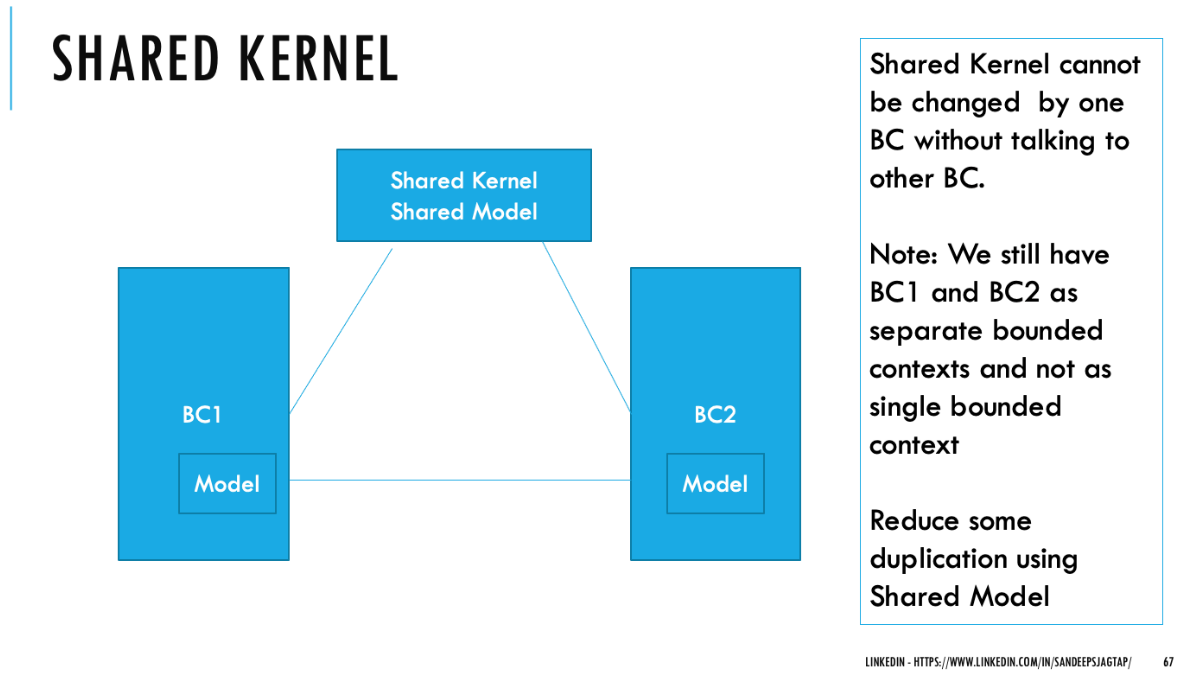
When multiple BCs are consumers of BC1.

BC1 defines API which can be used by BC2, BC3, …

BC1 enhances API as needed in future

LINKEDIN - HTTPS://WWW.LINKEDIN.COM/IN/SANDEEPSJAGTAP/ 65

Open Host Service



SHARED KERNEL

Shared Kernel
Shared Model
BC1
Model
BC2
Model

Shared Kernel cannot be changed by one BC without talking to other BC.

Note: We still have BC1 and BC2 as separate bounded contexts and not as single bounded context

Reduce some duplication using Shared Model

LINKEDIN - HTTPS://WWW.LINKEDIN.COM/IN/SANDEEPSJAGTAP/ 67

Shared Kernel

Please check also

https://contextmapper.org

Modular Monoliths

Modular monolith is a powerful concept!

Identifying (micro)service boundaries is a challenge and becomes relatively easy as we learn more about the domain. So we wanted to try a modular monolith approach. As the module matures and we see it becoming first class microservice, we can take it out of the modular monolith very easily. With modular monoliths it is easy and cheaper to add, remove modules.
This helps greatly to come up with correct boundaries for (micro) services as we go along delivering project.

Each module should have its own db migrations. And modules should not access other modules database tables directly.

One module should access other modules through controllers only (making in process call instead of http call). Keeping interaction at controller level avoids one module depending on the domain layer of another module as Controller would return view models.

We wanted to choose an approach so that we can have different modules/services in one place but not as a one service and we can easily extract them out if needed.

Summary

Each module has its own Domain classes.
Each module has its own DB or DB Schema. And DB Migration.
Modules talk to each other via published APIs
One module is not allowed to access DB of other module.
One module can not access other modules classes directly unless they are published as API to be used.

Keeping interaction at controller level avoids having one module depending on the domain layer of another module, as Controller would return view models.

Please refer to following working code:

Customer microservice with Customer as Aggregate containing module for Account with account as Aggregate. Example of Modular monolith or microservices with additional modules and each of module may become a microservice in future

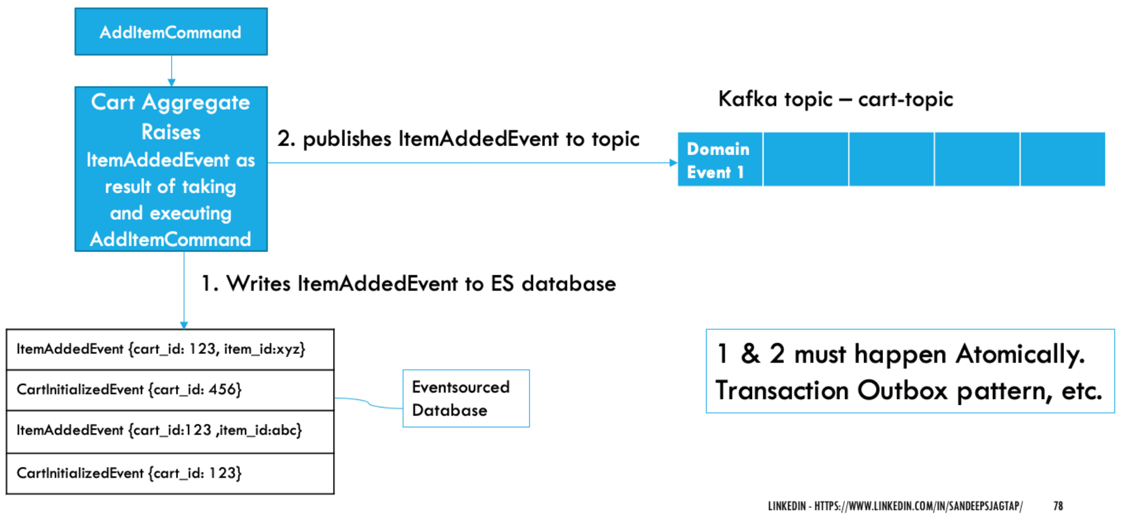
Code[1]

[1] https://github.com/ddd-workshop-org/ddd-modular-monolith-customer-account.git

Event Sourcing

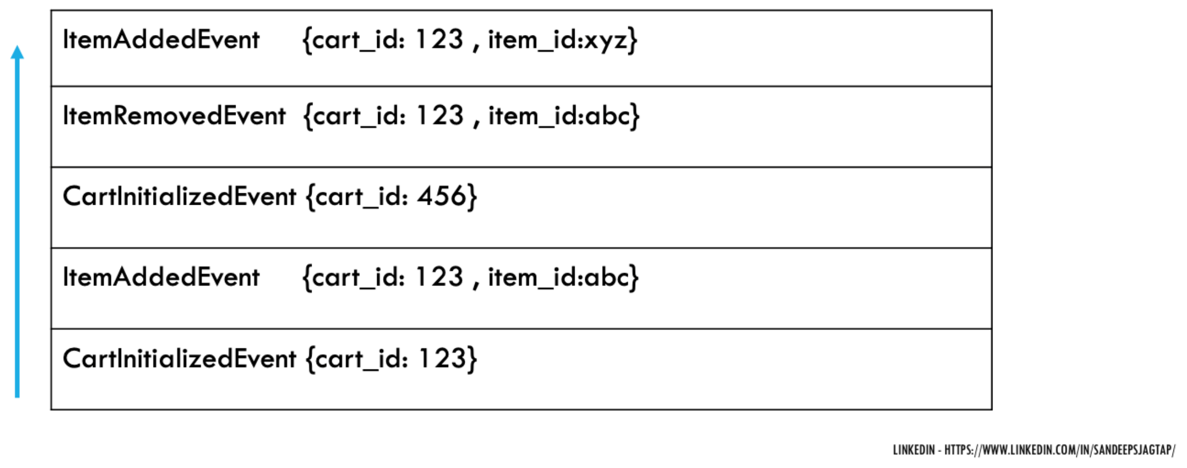
Event Sourcing is an Architectural Pattern in which state of application is being determined by sequence of events - Wikipedia

Events are Immutable, Facts, Named in past tense. Events have happened and can not deleted or modified.
Few examples of Event Sourced Aggregates are — Bank Account, General Ledgers, Append only Logs of Kafka
Lets look at Cart Aggregate in diagrams below. When AddItemCommand is sent to CartAggregate, Aggregate will ensure that all invariants are met,it will generate an Event called ItemAddedToCartEvent. Event will be stored in EvenStore and it will be published on message bus for external world to consume it.


AddItemCommand
Cart Aggregate Raises ItemAddedEvent as result of taking and executing AddItemCommand
2. publishes ItemAddedEvent to topic
Kafka topic – cart-topic
Domain Event 1
1. Writes ItemAddedEvent to ES database
ItemAddedEvent {cart_id: 123, item_id:xyz}
CartInitializedEvent {cart_id: 456}
ItemAddedEvent {cart_id:123 ,item_id:abc}
CartInitializedEvent {cart_id: 123}
Eventsourced Database
1 & 2 must happen Atomically. Transaction Outbox pattern, etc.
LINKEDIN - HTTPS://WWW.LINKEDIN.COM/IN/SANDEEPSJAGTAP/
78

Event Sourcing

Events will be saved in EventStore as shown below.


ItemAddedEvent {cart_id: 123 , item_id:xyz}
ItemRemovedEvent {cart_id: 123 , item_id:abc}
CartInitializedEvent {cart_id: 456}
ItemAddedEvent {cart_id: 123 , item_id:abc}
CartInitializedEvent {cart_id: 123}
LINKEDIN - HTTPS://WWW.LINKEDIN.COM/IN/SANDEEPSJAGTAP/

Event Sourcing - with events

CQRS

Separate Write side from Read side. Many systems have different requirements for read vs write. Many system are read heavy than write heavy.
Use commands to change Aggregate on write side.

Use read side to query current state of Aggregate. Build Read side by consuming Events published by Aggregate (write side).

Using Event sourcing and replaying events new read store can be built at any time in future.

Read side can use eventual consistency or weaker forms of it (CAP) using non-sql database. e.g Riak, cassandra, mongodb, neo4j

Write side has to be Strong consistency (ACID) — Relational DB Postgres, mysql, CocrochDB, Spanner

Using Event Sourcing and CQRS , you are already in land of Distributed System and microservices.

E.g. frameworks
http://www.axonframework.org
http://doc.akka.io/docs/akka/current/java/persistence.html

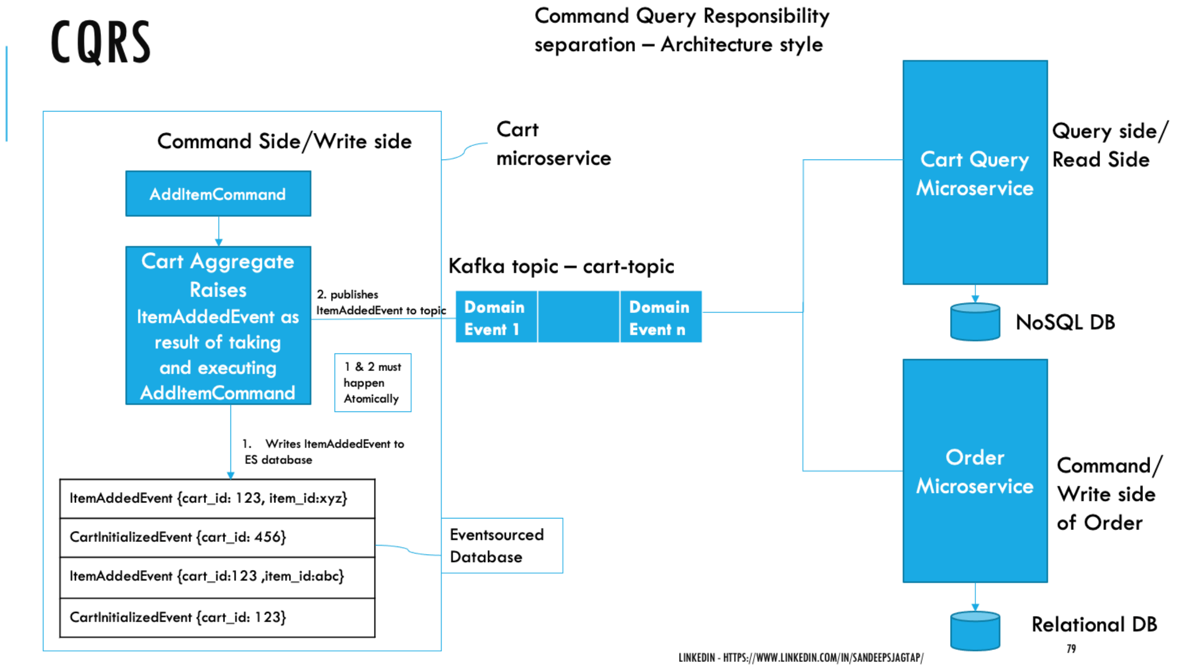
CQRS can be used without Event Sourcing

Separate Write side model and Read side Model

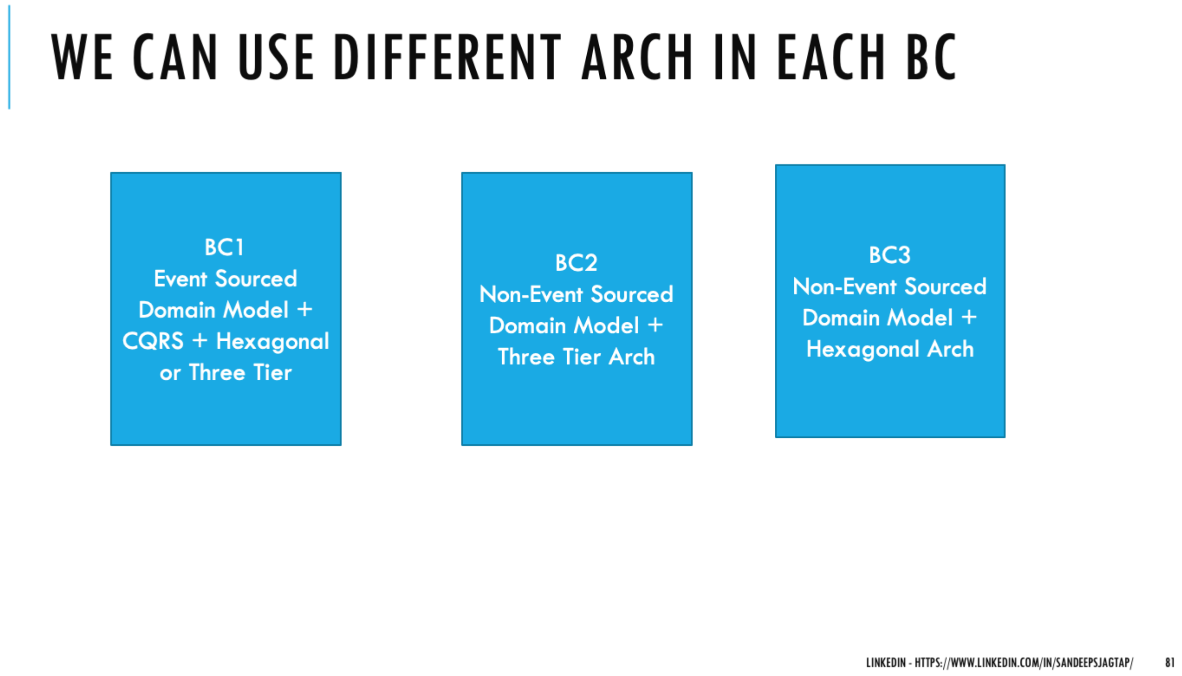
E.g., Hibernate HQL Queries are hard because we are trying to use Write Model to serve Read/Query requirements.

Working Code examples - DDD + CQRS + Event sourcing :
Code — Work in progress (all repos starting with eshop)

https://github.com/sandeepjagtap/eshop-shopping
https://github.com/sandeepjagtap/eshop-shopping-query


CQRS
Command Query Responsibility separation – Architecture style
Command Side/Write side
Cart microservice
AddItemCommand
Cart Aggregate Raises ItemAddedEvent as result of taking and executing AddItemCommand
2. publishes ItemAddedEvent to topic
1 & 2 must happen Atomically
1. Writes ItemAddedEvent to ES database
ItemAddedEvent {cart_id: 123, item_id:xyz}
CartInitializedEvent {cart_id: 456}
ItemAddedEvent {cart_id:123 ,item_id:abc}
CartInitializedEvent {cart_id: 123}
Eventsourced Database
Kafka topic – cart-topic
Domain Event 1
Domain Event n
Cart Query Microservice
Query side/ Read Side
NoSQL DB
Order Microservice
Command/ Write side of Order
Relational DB
LINKEDIN - HTTPS://WWW.LINKEDIN.COM/IN/SANDEEPSJAGTAP/
79

CQRS



WE CAN USE DIFFERENT ARCH IN EACH BC

BC1
Event Sourced Domain Model + CQRS + Hexagonal or Three Tier

BC2
Non-Event Sourced Domain Model + Three Tier Arch

BC3
Non-Event Sourced Domain Model + Hexagonal Arch

LINKEDIN - HTTPS://WWW.LINKEDIN.COM/IN/SANDEEPSJAGTAP/ 81

Different Bounded Context can use different architectures

Finding Bounded Contexts

Two Techniques:

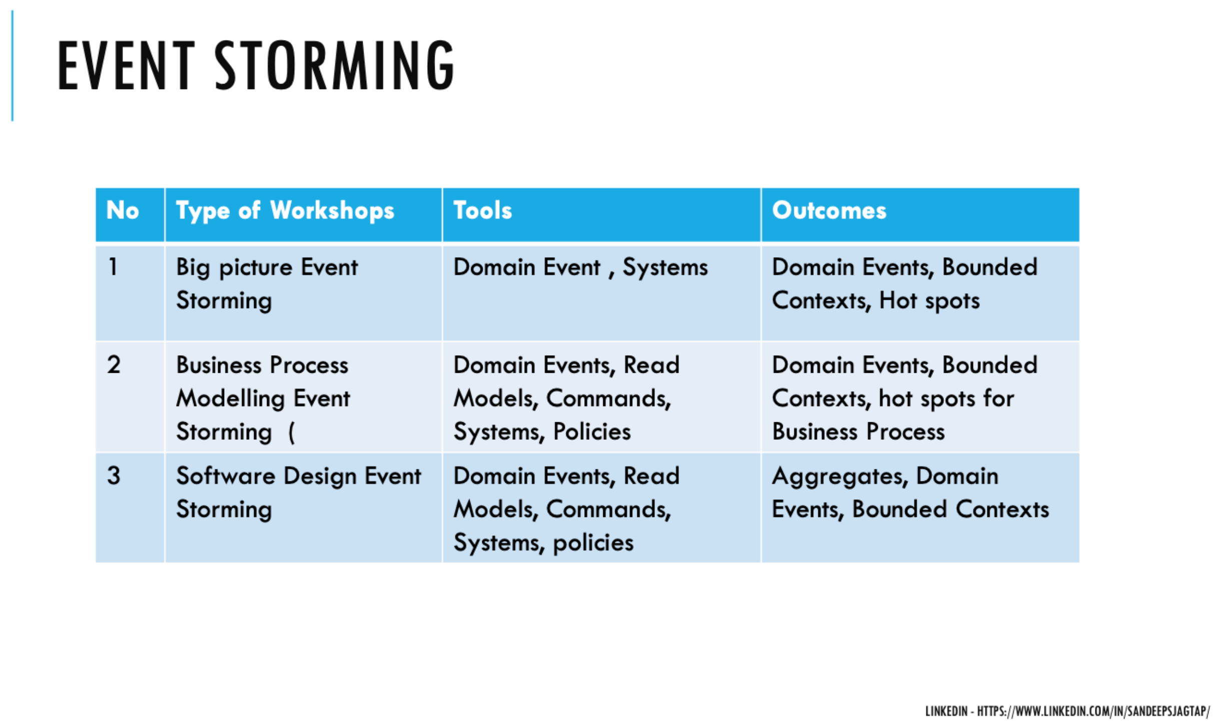
1. Event Storming

2 Domain Storytelling

Event Storming

https://www.eventstorming.com/. by Alberto Brandolini
Event storming is one of the techniques for finding initial bounded contexts using Domain Events.
It is a workshop run with Domain Experts (Product Owners, Domain Specialist, Business Analyst), technologists, XD, VD, and other key stake holders
Potential outcomes of Event Storming

1. Initial list of bounded context.
2. Domain Events.
3. Aggregates.

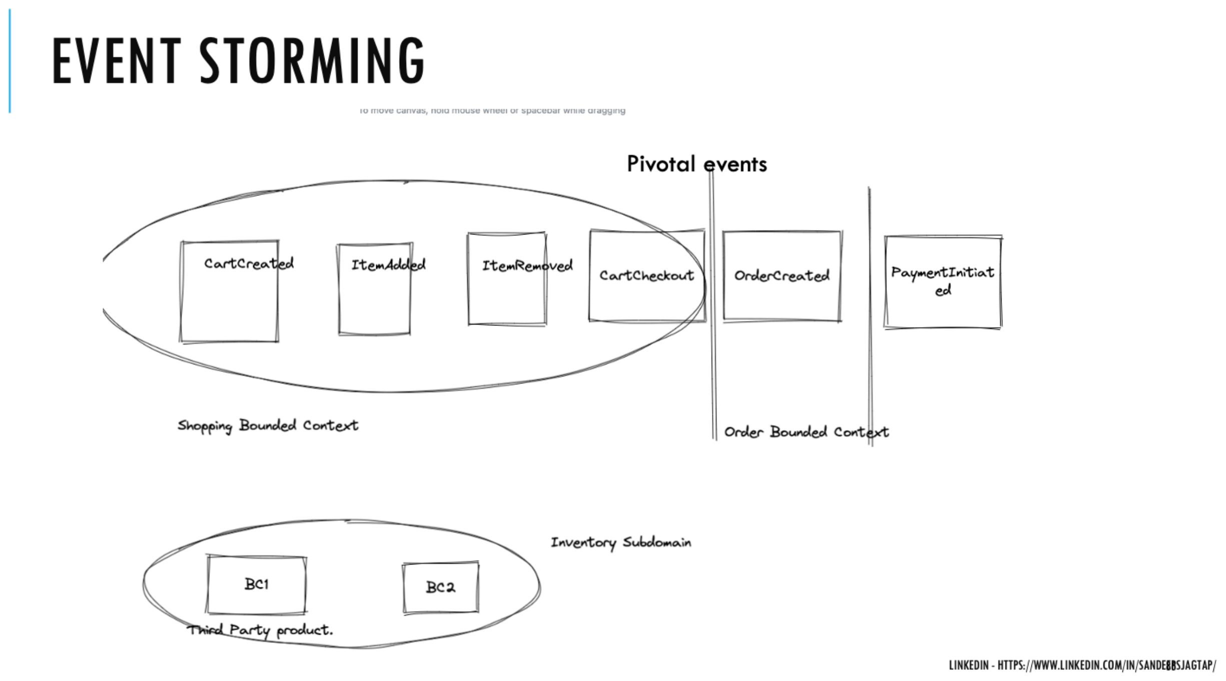


EVENT STORMING

No	Type of Workshops	Tools	Outcomes
1	Big picture Event Storming	Domain Event , Systems	Domain Events, Bounded Contexts, Hot spots
2	Business Process Modelling Event Storming (	Domain Events, Read Models, Commands, Systems, Policies	Domain Events, Bounded Contexts, hot spots for Business Process
3	Software Design Event Storming	Domain Events, Read Models, Commands, Systems, policies	Aggregates, Domain Events, Bounded Contexts

LINKEDIN - HTTPS://WWW.LINKEDIN.COM/IN/SANDEEPSJAGTAP/

Event storming



EVENT STORMING

LINKEDIN - HTTPS://WWW.LINKEDIN.COM/IN/SANDEEPSJAGTAP/

Event storming

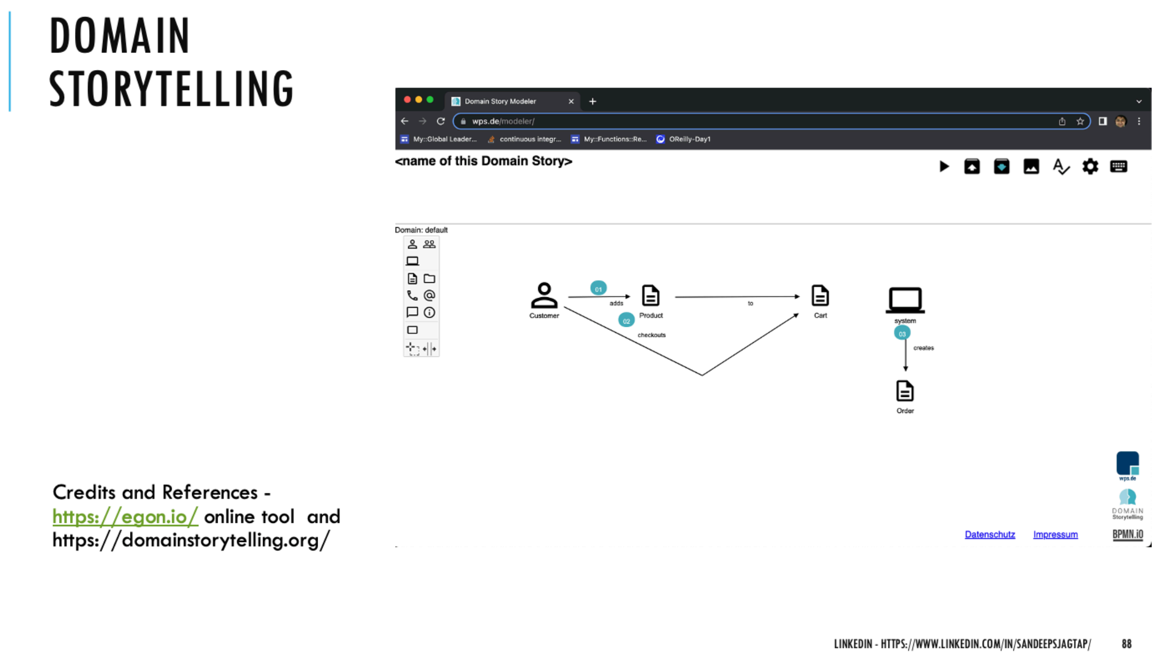
Domain Storytelling

https://domainstorytelling.org/

Domain Storytelling is one of the techniques for finding initial bounded contexts.
It is workshop run with Domain Experts (Product Owners, Domain Specialist, Business Analyst), technologists, XD, VD, and other key stake holders
Potential outcomes of Domain Storytelling

1. Initial list of bounded context.
2. Initial Domain Model

Domain Storytelling
https://www.youtube.com/watch?v=Y1ykXnl6r7s



DOMAIN STORYTELLING

<name of this Domain Story>

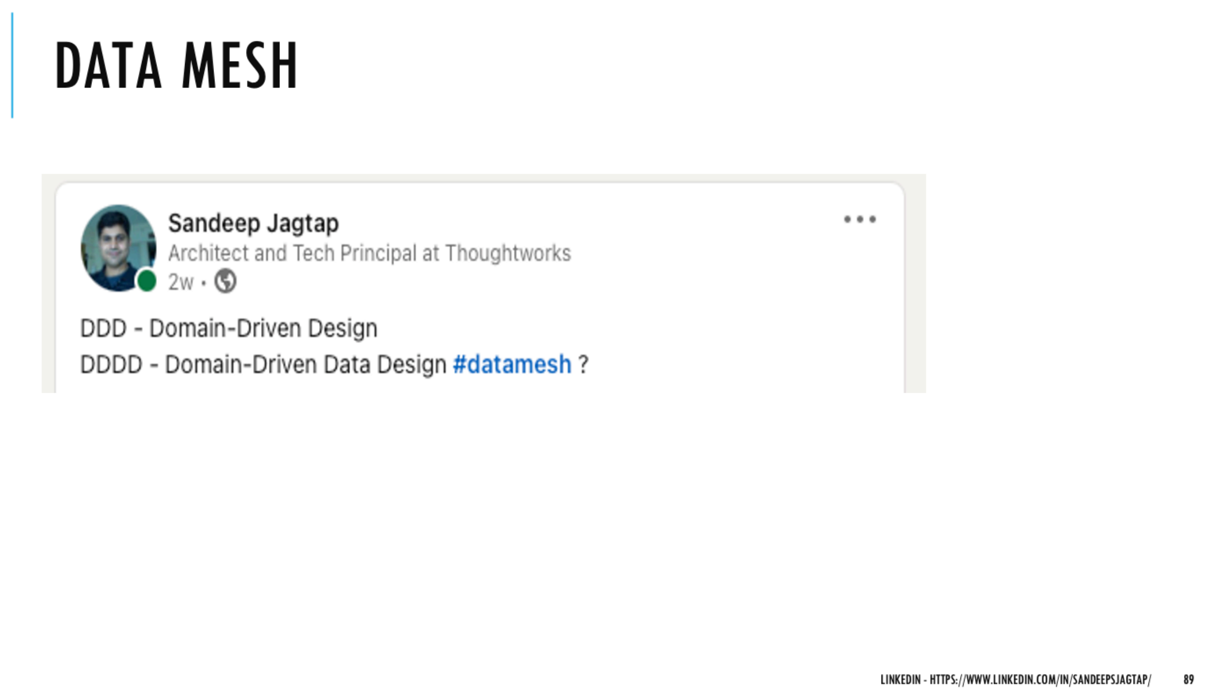
Domain: default

Credits and References - https://egon.io/ online tool and https://domainstorytelling.org/

LINKEDIN - HTTPS://WWW.LINKEDIN.COM/IN/SANDEEPSJAGTAP/ 88

Domain Storytelling

DDD and Relation to Data Mesh



DATA MESH

Sandeep Jagtap
Architect and Tech Principal at Thoughtworks
2w •

DDD - Domain-Driven Design
DDDD - Domain-Driven Data Design #datamesh ?

LINKEDIN - HTTPS://WWW.LINKEDIN.COM/IN/SANDEEPSJAGTAP/ 89

What is Data Mesh

Data Mesh https://www.oreilly.com/library/view/data-mesh/9781492092384/ Zhamak Dehghani

Data Lake are more like monoliths, and we may face issues like lack of quality, lack of ownership, and too much decency on centralized team

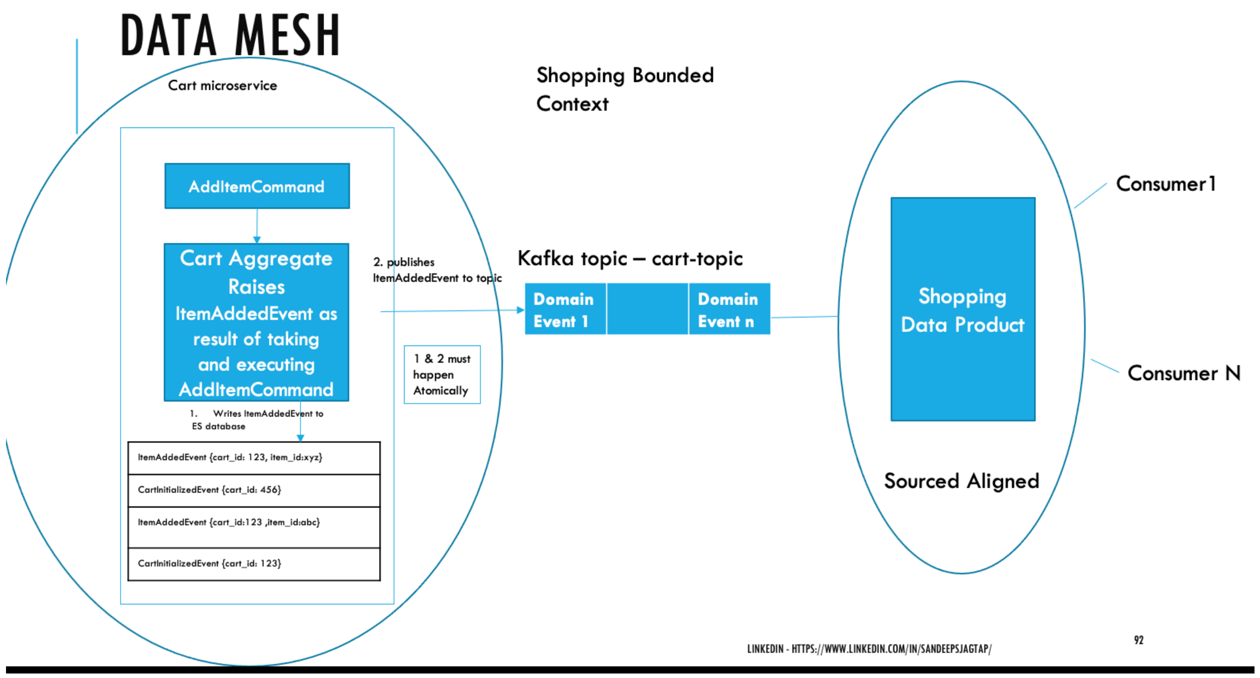
“Data mesh is a decentralized sociotechnical approach to share, access, and manage analytical data in complex and large-scale environments—within or across organizations.
Distributed data products oriented around domains and owned by independent cross-functional teams who have embedded data engineers and data product owners, using common data infrastructure as a platform to host, prep and serve their data assets”. - Zhamak Dehgani

Types of Data products
1. Source-aligned domain data:
Matches with existing bounded contexts. Analytical data reflecting the business facts generated by the bounded context. (native data product)
2. Aggregate domain data:
Analytical data that is an aggregate of multiple upstream domains (or bounded contexts). Be careful with not building monolith here.

3. Consumer-aligned domain data: Analytical data transformed to fit the needs of one or multiple specific use cases. (fit-for-purpose domain data)

We can use DomainEvents to build Data Mesh.
Microservice's Domain layer can publish events which can be consumed by Data Mesh Products.

We can also use CDC (Chnage Data Capture) as techniques in case of Legacy systems and consume them.


DATA MESH
Cart microservice
Shopping Bounded Context
AddItemCommand
Cart Aggregate Raises ItemAddedEvent as result of taking and executing AddItemCommand
2. publishes ItemAddedEvent to topic
1 & 2 must happen Atomically
1. Writes ItemAddedEvent to ES database
ItemAddedEvent {cart_id: 123, item_id:xyz}
CartInitializedEvent {cart_id: 456}
ItemAddedEvent {cart_id:123 ,item_id:abc}
CartInitializedEvent {cart_id: 123}
Kafka topic – cart-topic
Domain Event 1
Domain Event n
Shopping Data Product
Sourced Aligned
Consumer1
Consumer N
LINKEDIN - HTTPS://WWW.LINKEDIN.COM/IN/SANDEEPSJAGTAP/
92

Data Mesh and Bounded Context

github code links

All code examples are available at

code[2]

[2]https://github.com/ddd-workshop-org

References

DDD blue book - Eric Evans[3]

DDD Patterns Principles - Scott Millett and Nick Tune[4]

The Anatomy of Domain-Driven Design by Scott Millett[5]

Data inside vs data outside - Pat Helland[6]

Life beyond Distributed transactions Paper Pat Helland.

Hexagonal Architecture - Alistair Cockburn[7]

Strangler Fig - Martin Fowler[8]

Event Sourcing and CQRS - Greg Young

Data Mesh - Zhamak Dehghani[9]

Evenstorming - Alberto Brandolini[10]

Domain Storytelling - Stefan Hofer and Henning Schwentner [11]

Team Topologies by Matthew Skelton and Manuel Pais[12]

[3]https://learning.oreilly.com/library/view/domain-driven-design-tackling/0321125215/
[4]https://learning.oreilly.com/library/view/patterns-principles-and/9781118714706/
[5]https://leanpub.com/theanatomyofdomain-drivendesign
[6]http://cidrdb.org/cidr2005/papers/P12.pdf
[7]https://en.wikipedia.org/wiki/Hexagonal_architecture_(software)
[8]https://martinfowler.com/bliki/StranglerFigApplication.html
[9]https://www.oreilly.com/library/view/data-mesh/9781492092384/
[10]https://www.eventstorming.com/
[11]https://domainstorytelling.org/
[12]https://teamtopologies.com/book

Microfrontends and DDD

<To do>