Domain Driven Design

Zero to Hero

Fabrício Rissetto

Fabrício Rissetto

Software Engineer

fabriciorissetto.com

fabriciorissetto

fabriciorissetto

fabriciorissetto@gmail.com

creditas

Schedule

- Basic Concepts
- Ubiquitous Language
- Domain Expert
- Domain Model
- Architectures types
  - DDD,
  - Smart UI,
  - …
- **Strategic Design**
  - Bounded Context
  - Context Maps
  - Domain Events
  - Event Storming
- **Tactical Design** (building blocks)
  - Layered Architecture
  - Entities
  - Value Objects
  - Aggregates
  - Domain Services
  - Factories
  - Repositories
- CQRS
- SOA
  - Event Driven Architecture
- Event Sourcing
- Final Thoughts

Domain Expert

Ubiquitous Language

What is a **Domain Model**?

Domain == problem

Model == solution

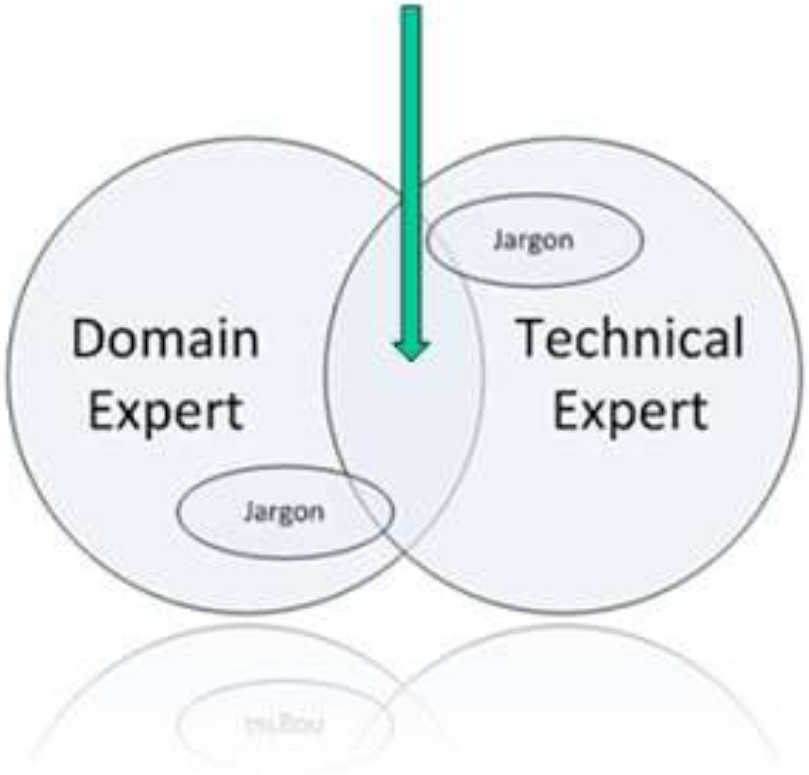
Distilled knowledge

“The model isn’t just the knowledge in a domain expert's head; it is a rigorously organized and selective abstraction of that.”
- Eric Evans

Can be expressed in several ways:

- **Code**
- Diagrams
- Written Documentation

Ubiquitous Language



C#

```
public class Livro
{
    public int Id { get; private set; }
    public int Titulo { get; private set; }
    public Autor Autor { get; private set; }
    public bool Revisado { get; private set; }
    public DateTime DataRevisao { get; private set; }
    public Usuario Revisor { get; private set; }

    public Livro(string titulo, Autor autor)...

    public void Revisar(Usuario revisor)
    {
        if (Revisado)
            throw new InvalidOperationException("O livro já foi revisado.");

        Revisado = true;
        DataRevisao = DateTime.Now;
        Revisor = revisor;
    }

    public void Publicar()...
}
```

C#

```
public class Livro
{
    public int Id { get; private set; }
    public int Titulo { get; private set; }
    public Autor Autor { get; private set; }
    public bool Revisado { get; private set; }
    public DateTime DataRevisao { get; private set; }
    public Usuario Revisor { get; private set; }

    public Livro(string titulo, Autor autor)...

    public void Revisar(Usuario revisor)
    {
        if (Revisado)
            throw new InvalidOperationException("O livro já foi revisado.");

        Revisado = true;
        DataRevisao = DateTime.Now;
        Revisor = revisor;
    }

    public void Publicar()...
}
```

C#

```
public class Livro
{
    public int Id { get; private set; }
    public int Titulo { get; private set; }
    public Autor Autor { get; private set; }
    public bool Revisado { get; private set; }
    public DateTime DataRevisao { get; private set; }
    public Usuario Revisor { get; private set; }

    public Livro(string titulo, Autor autor)...

    public void Revisar(Usuario revisor)
    {
        if (Revisado)
            throw new InvalidOperationException("O livro já foi revisado.");

        Revisado = true;
        DataRevisao = DateTime.Now;
        Revisor = revisor;
    }

    public void Publicar()...
}
```

C#

```
public class Livro
{
    public int Id { get; private set; }
    public int Titulo { get; private set; }
    public Autor Autor { get; private set; }
    public bool Revisado { get; private set; }
    public DateTime DataRevisao { get; private set; }
    public Usuario Revisor { get; private set; }

    public Livro(string titulo, Autor autor)...

    public void Revisar(Usuario revisor)
    {
        if (Revisado)
            throw new InvalidOperationException("O livro já foi revisado.");

        Revisado = true;
        DataRevisao = DateTime.Now;
        Revisor = revisor;
    }

    public void Publicar()...
}
```

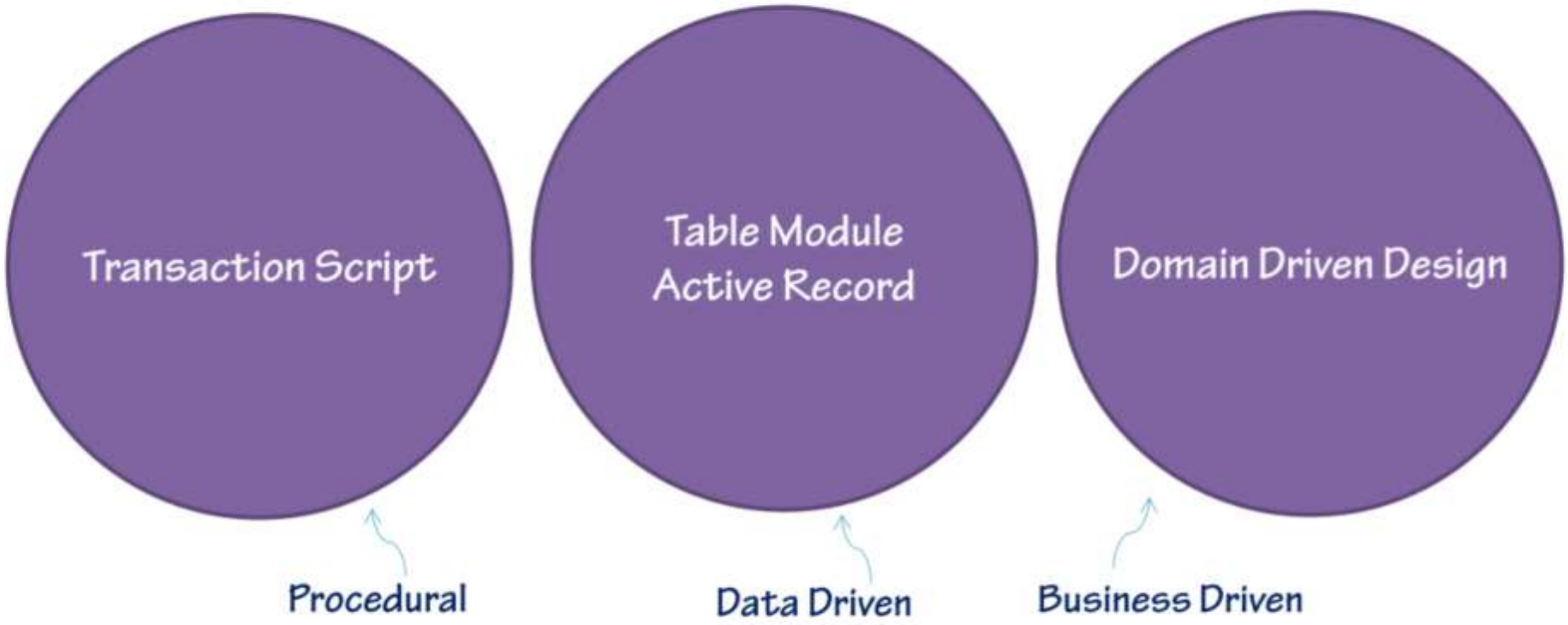
```ruby
class Livro
  attr_reader :id, :titulo, :autor, :revisado, :data_revisao, :revisor

  def initialize(titulo, autor)
    @titulo = titulo
    @autor = autor
  end

  def revisar(revisor)
    fail "O livro já foi revisado" unless revisado

    @revisado = true
    @data_revisao = Time.now
    @revisor = revisor
  end

  def publicar()
    # ...
  end

end
```

Ruby

Before we start

- An iterative process
- Business involvement (domain experts)
- Being fluent with Object-Oriented (OO) programming paradigm
- Being familiar with SOLID principles

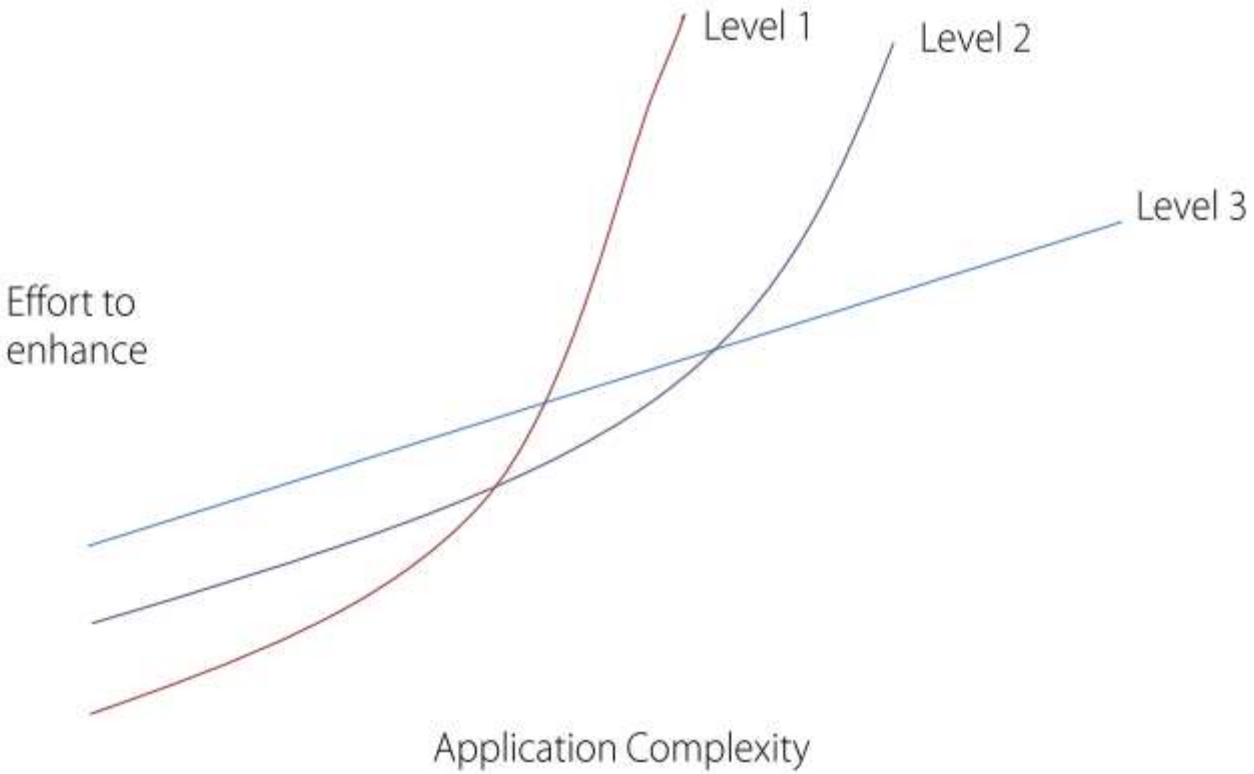
What's your focus?



Architecture Selection

MVP	Flagship product
Junior team	Senior Team
Small team	Large Team
Simple Domain	Complex domain
Tight timeline	Flexible timeline
Short lifespan	Long-term
No security concerns	Security matters
Little chance for reuse	Known need for reuse

Choice by Complexity



Schedule

- ~~Basic Concepts~~
- ~~Ubiquitous Language~~
- ~~Domain Expert~~
- ~~Domain Model~~
- ~~Architectures types~~
  - ~~DDD~~,
  - ~~Smart UI~~,
  - ~~...~~
- **Strategic Design**
  - Bounded Context
  - Context Maps
  - Domain Events
  - Event Storming
- **Tactical Design** (building blocks)
  - Layered Architecture
  - Entities
  - Value Objects
  - Aggregates
  - Domain Services
  - Factories
  - Repositories
- CQRS
- SOA
  - Event Driven Architecture
- Event Sourcing
- Final Thoughts

The Two Sides of Domain-Driven Design

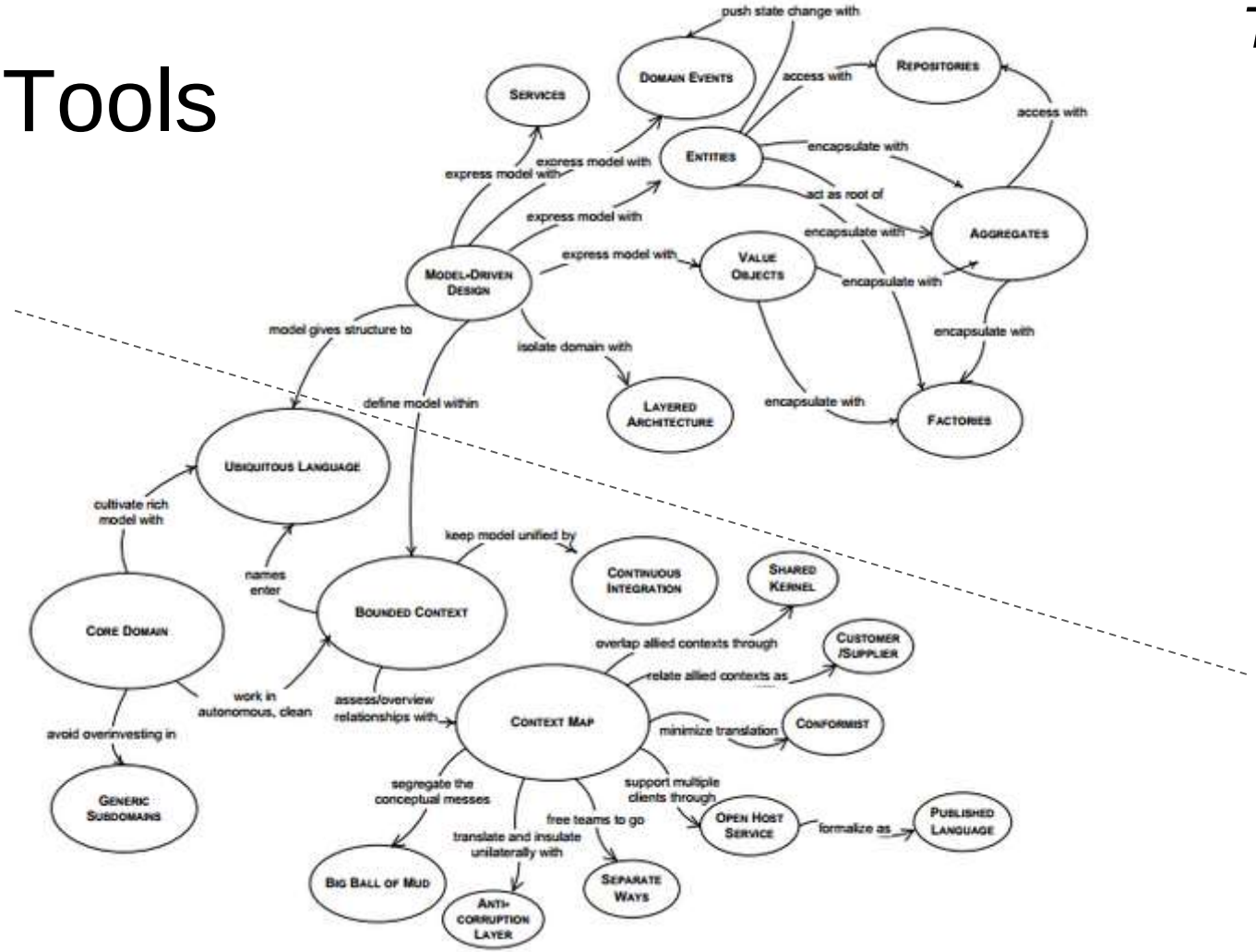
Tactical

vs

Strategic

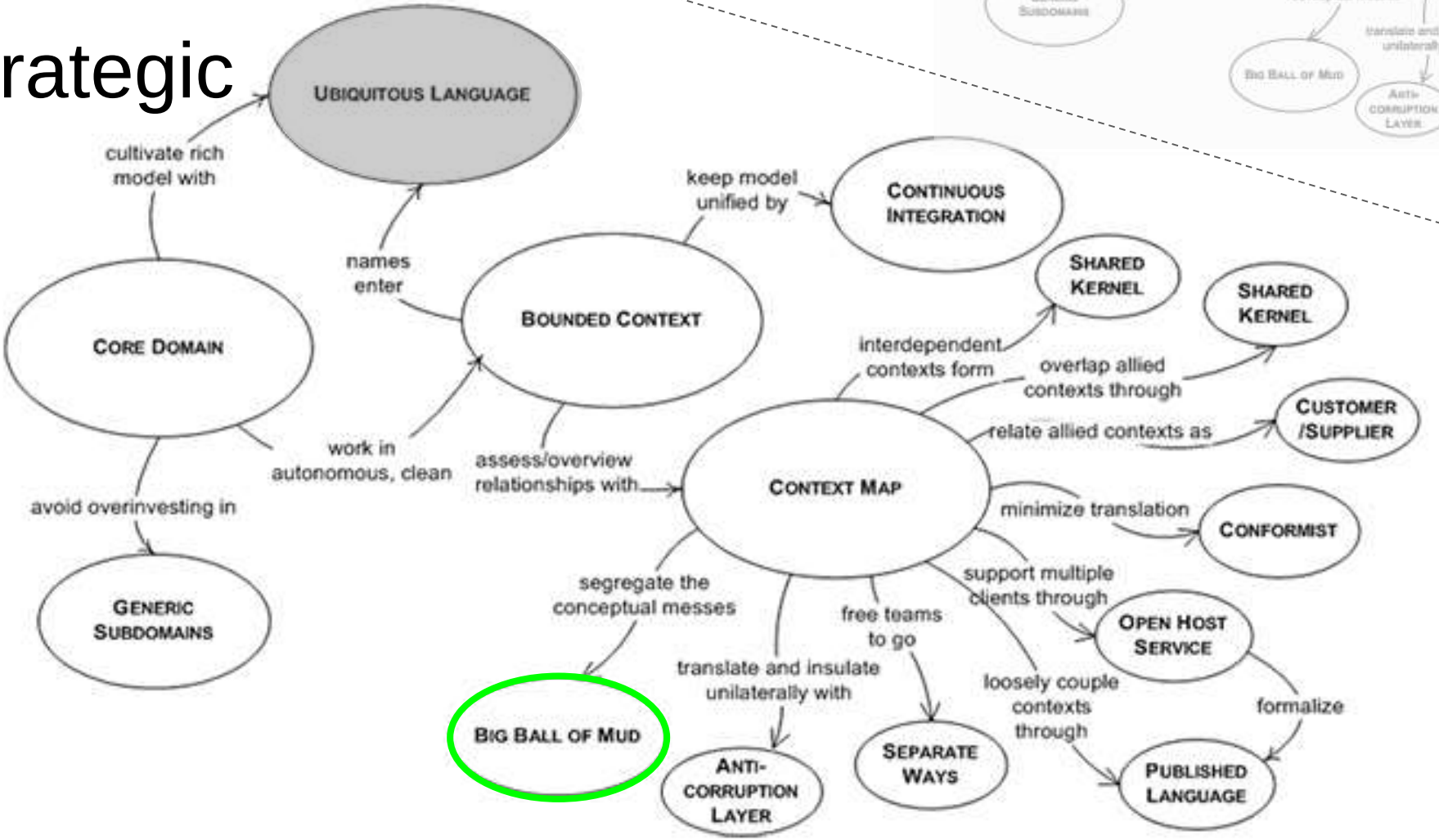
DDD Tools

Tactical

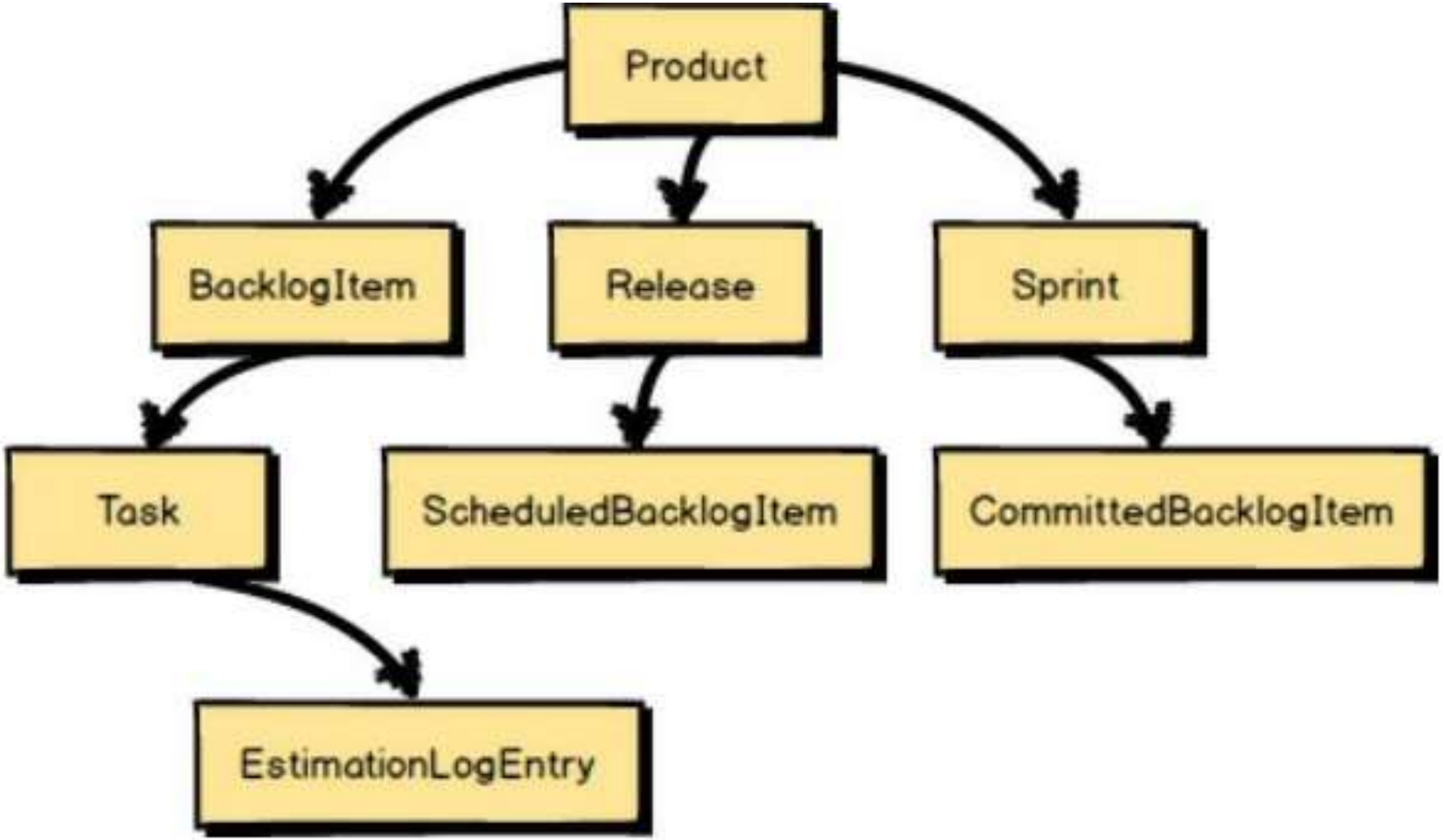


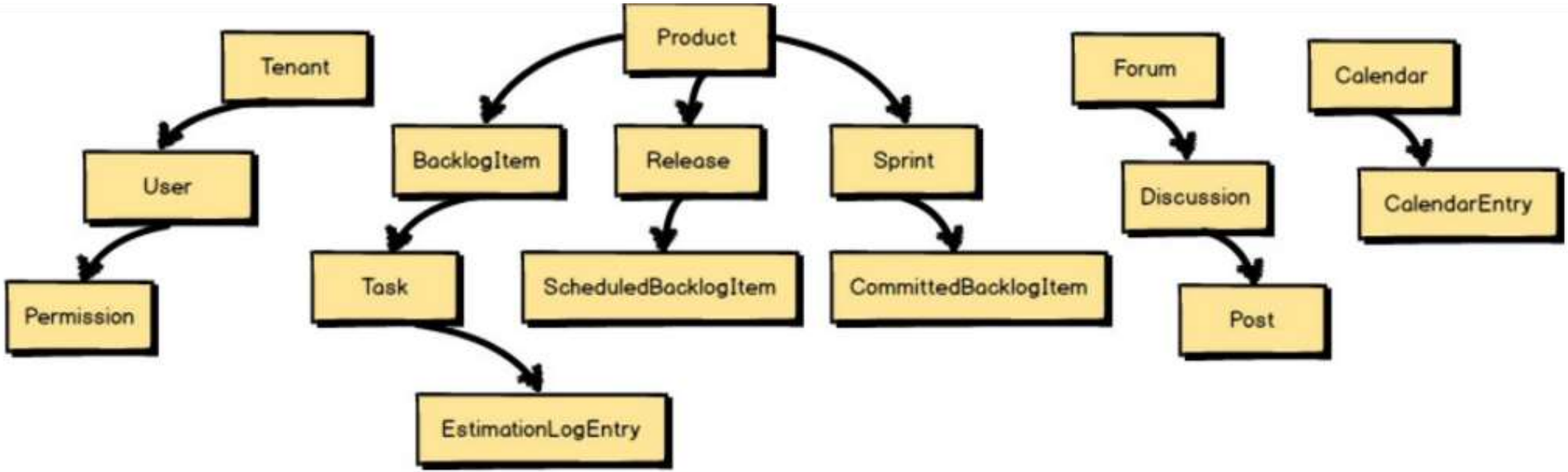
Strategic

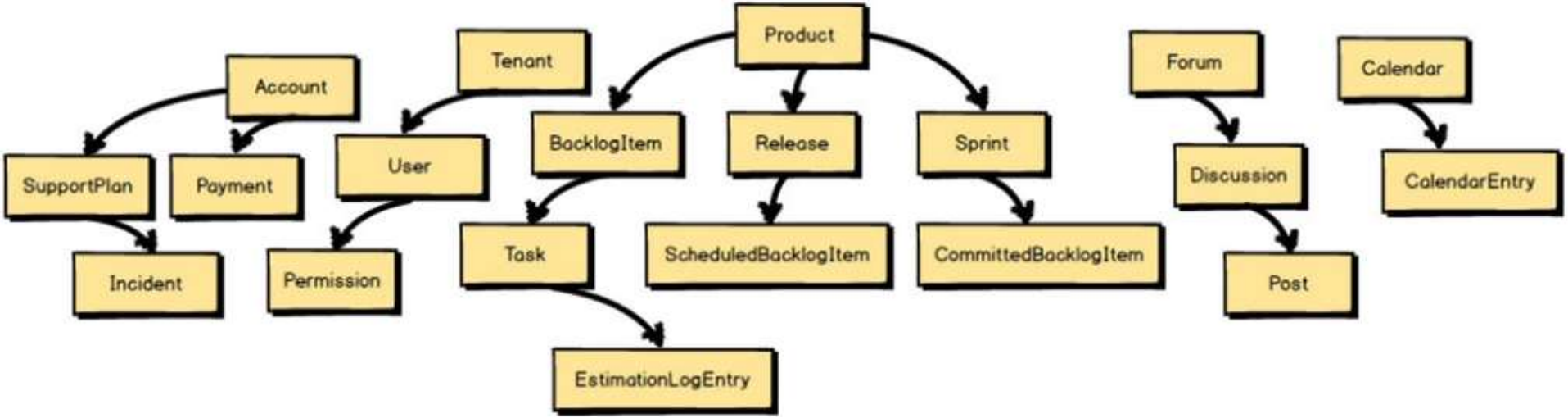
Strategic

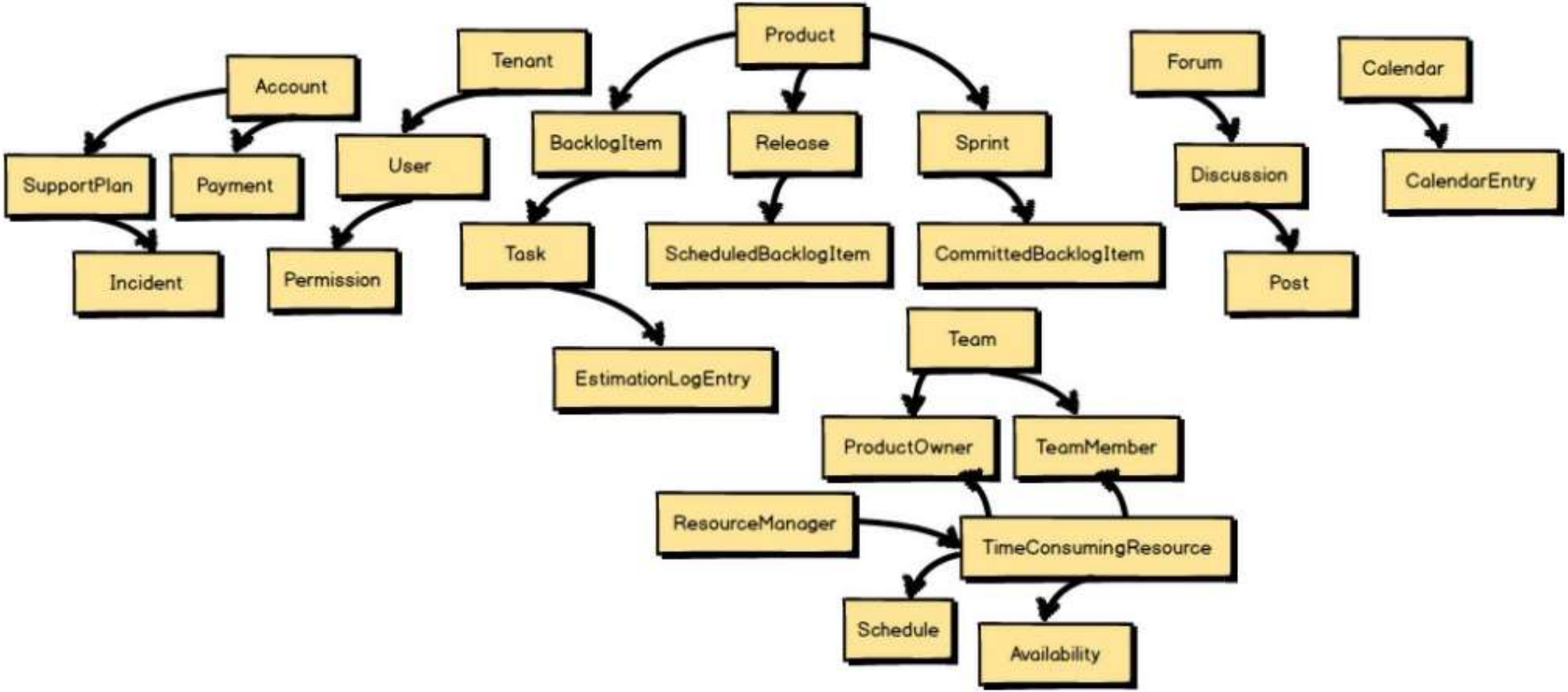

UBIQUITOUS LANGUAGE
cultivate rich model with
keep model unified by
CONTINUOUS INTEGRATION
names enter
BOUNDED CONTEXT
CORE DOMAIN
SHARED KERNEL
SHARED KERNEL
interdependent contexts form
overlap allied contexts through
CUSTOMER /SUPPLIER
relate allied contexts as
work in autonomous, clean
assess/overview relationships with
CONTEXT MAP
avoid overinvesting in
minimize translation
CONFORMIST
GENERIC SUBDOMAINS
segregate the conceptual messes
support multiple clients through
OPEN HOST SERVICE
free teams to go
translate and insulate unilaterally with
loosely couple contexts through
formalize
BIG BALL OF MUD
ANTI-CORRUPTION LAYER
SEPARATE WAYS
PUBLISHED LANGUAGE

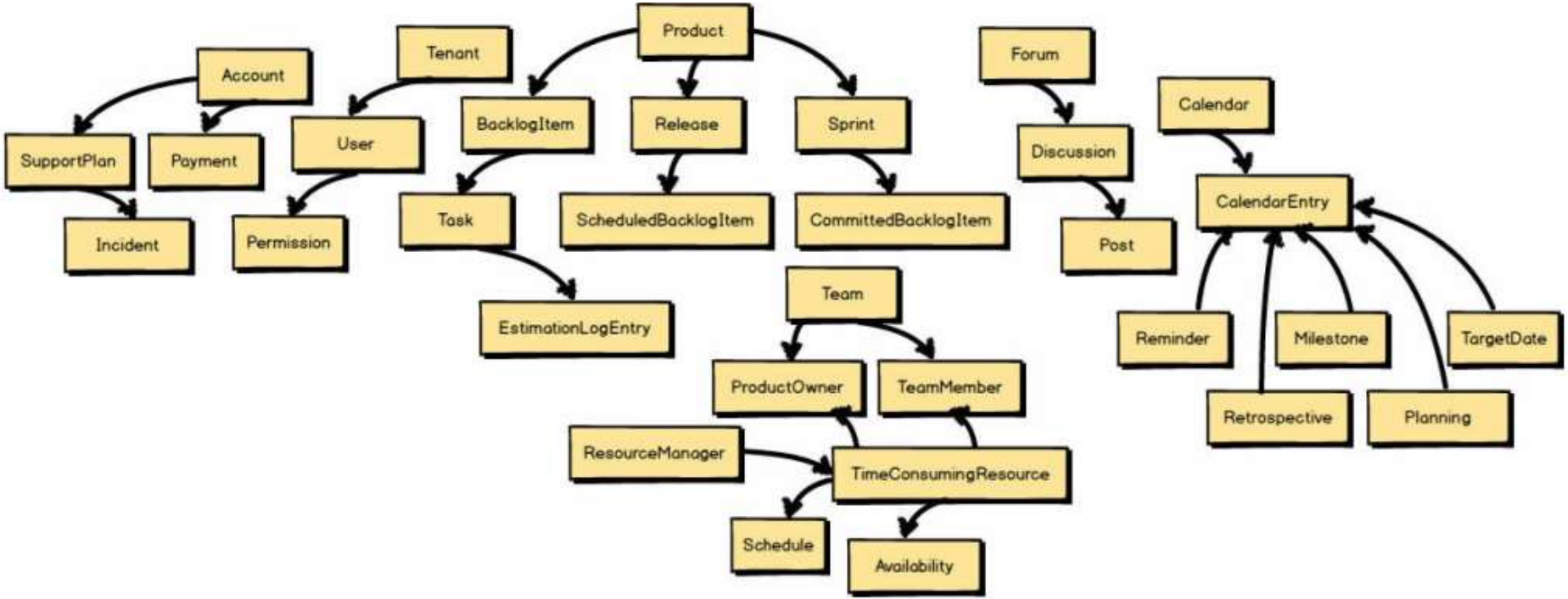
Big Ball of Mud

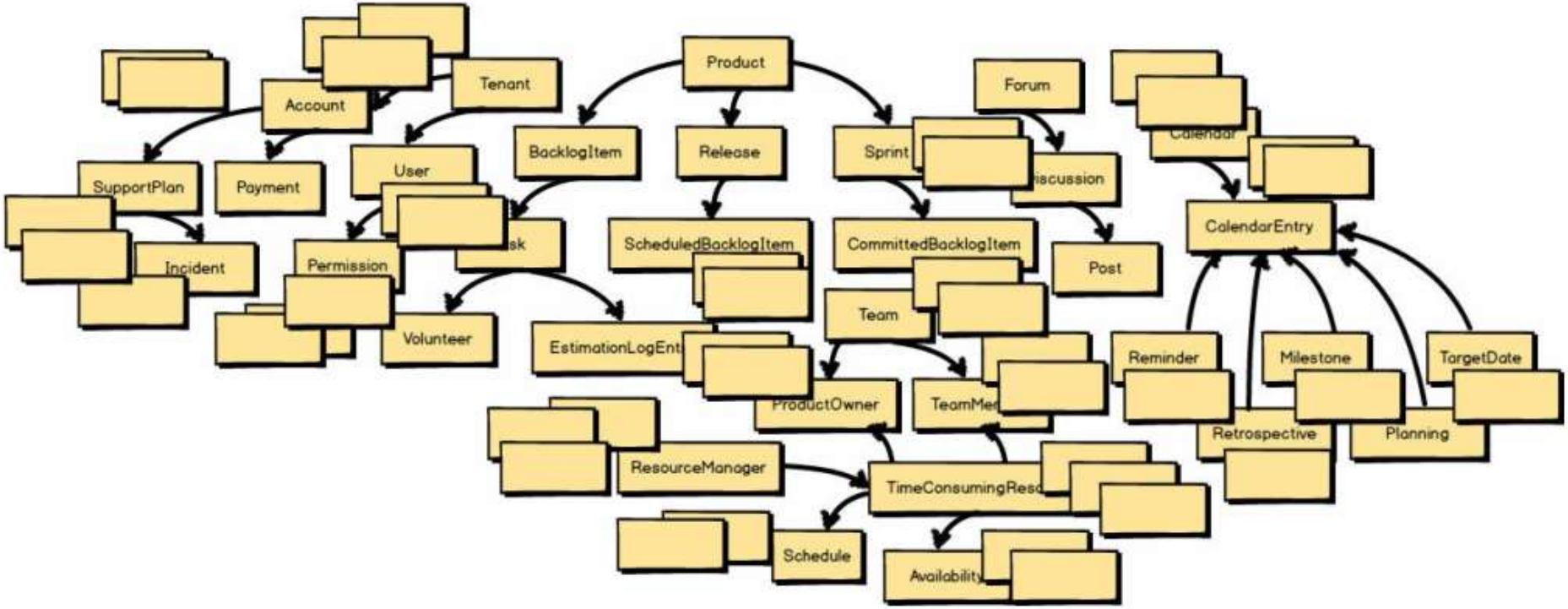

Product
BacklogItem
Release
Sprint
Task
ScheduledBacklogItem
CommittedBacklogItem
EstimationLogEntry


Tenant
User
Permission
Product
BacklogItem
Release
Sprint
Task
ScheduledBacklogItem
CommittedBacklogItem
EstimationLogEntry
Forum
Discussion
Post
Calendar
CalendarEntry

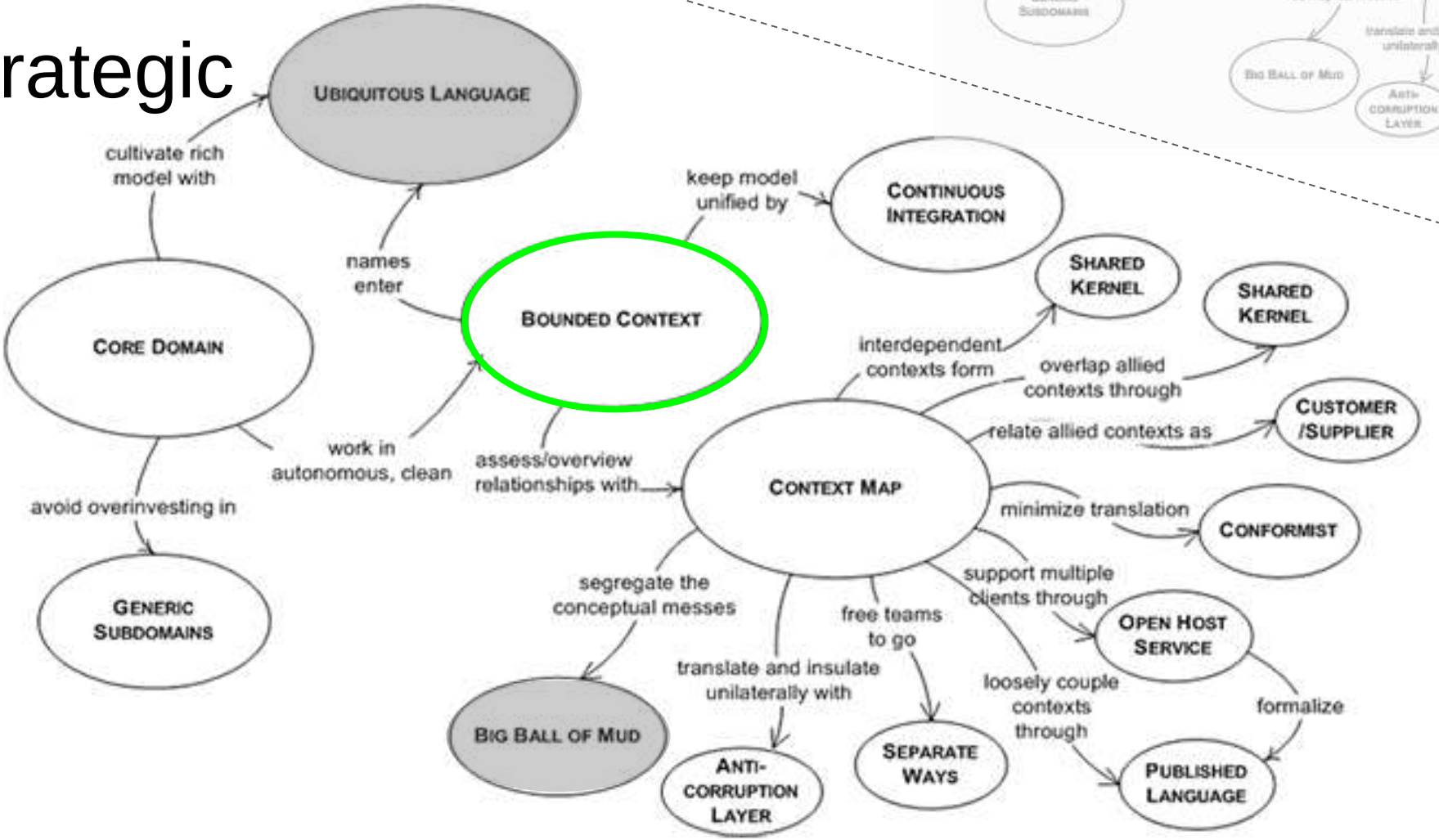

Account
SupportPlan
Payment
Incident
Tenant
User
Permission
Product
BacklogItem
Release
Sprint
Task
ScheduledBacklogItem
CommittedBacklogItem
EstimationLogEntry
Forum
Discussion
Post
Calendar
CalendarEntry


Product
Tenant
Account
Forum
Calendar
BacklogItem
Release
Sprint
User
SupportPlan
Payment
Discussion
CalendarEntry
Task
ScheduledBacklogItem
CommittedBacklogItem
Incident
Permission
Post
Team
EstimationLogEntry
ProductOwner
TeamMember
ResourceManager
TimeConsumingResource
Schedule
Availability

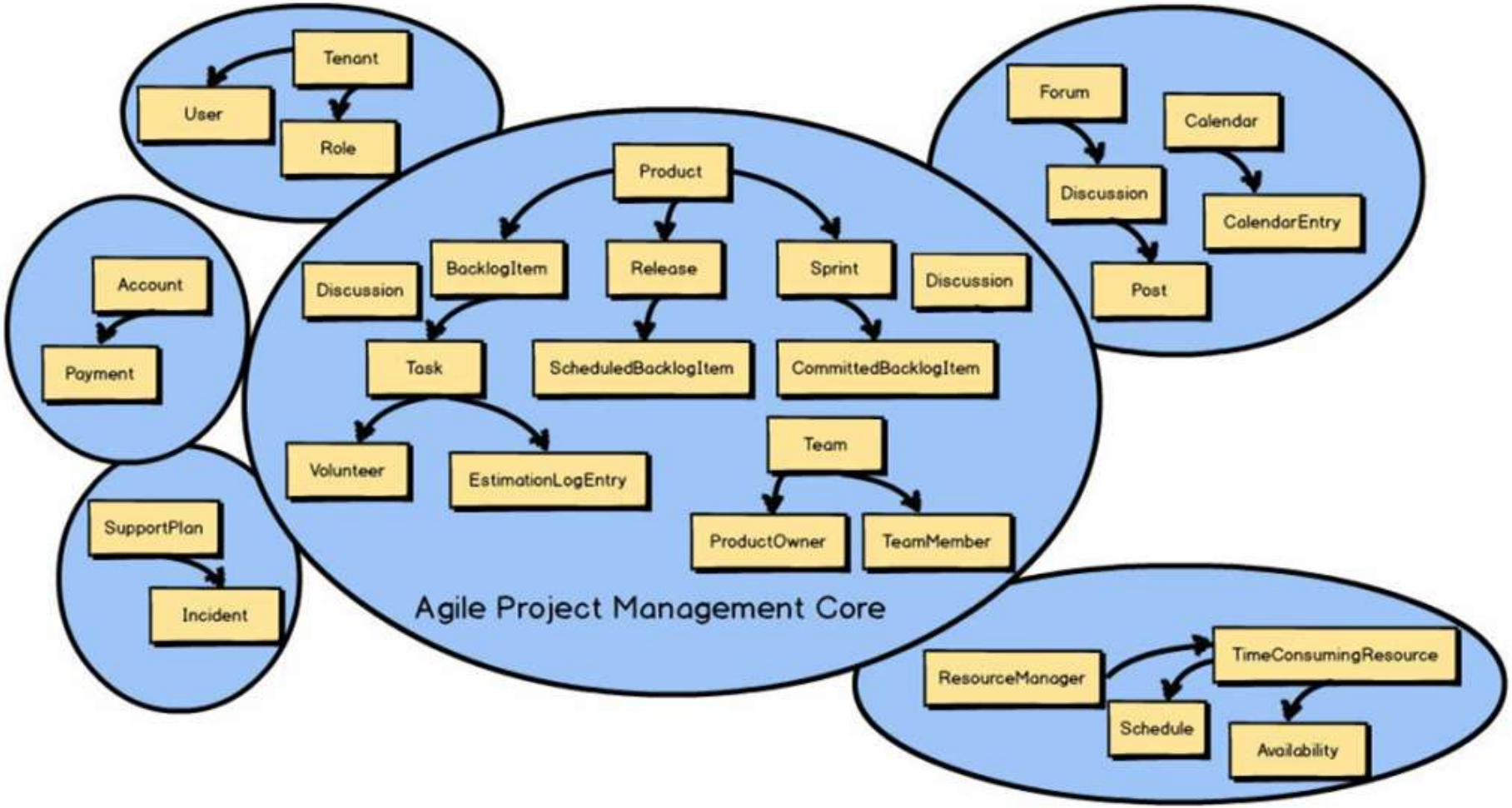

Account
Tenant
Product
Forum
Calendar
SupportPlan
Payment
User
BacklogItem
Release
Sprint
Discussion
CalendarEntry
Incident
Permission
Task
ScheduledBacklogItem
CommittedBacklogItem
Post
EstimationLogEntry
Team
Reminder
Milestone
TargetDate
ProductOwner
TeamMember
Retrospective
Planning
ResourceManager
TimeConsumingResource
Schedule
Availability

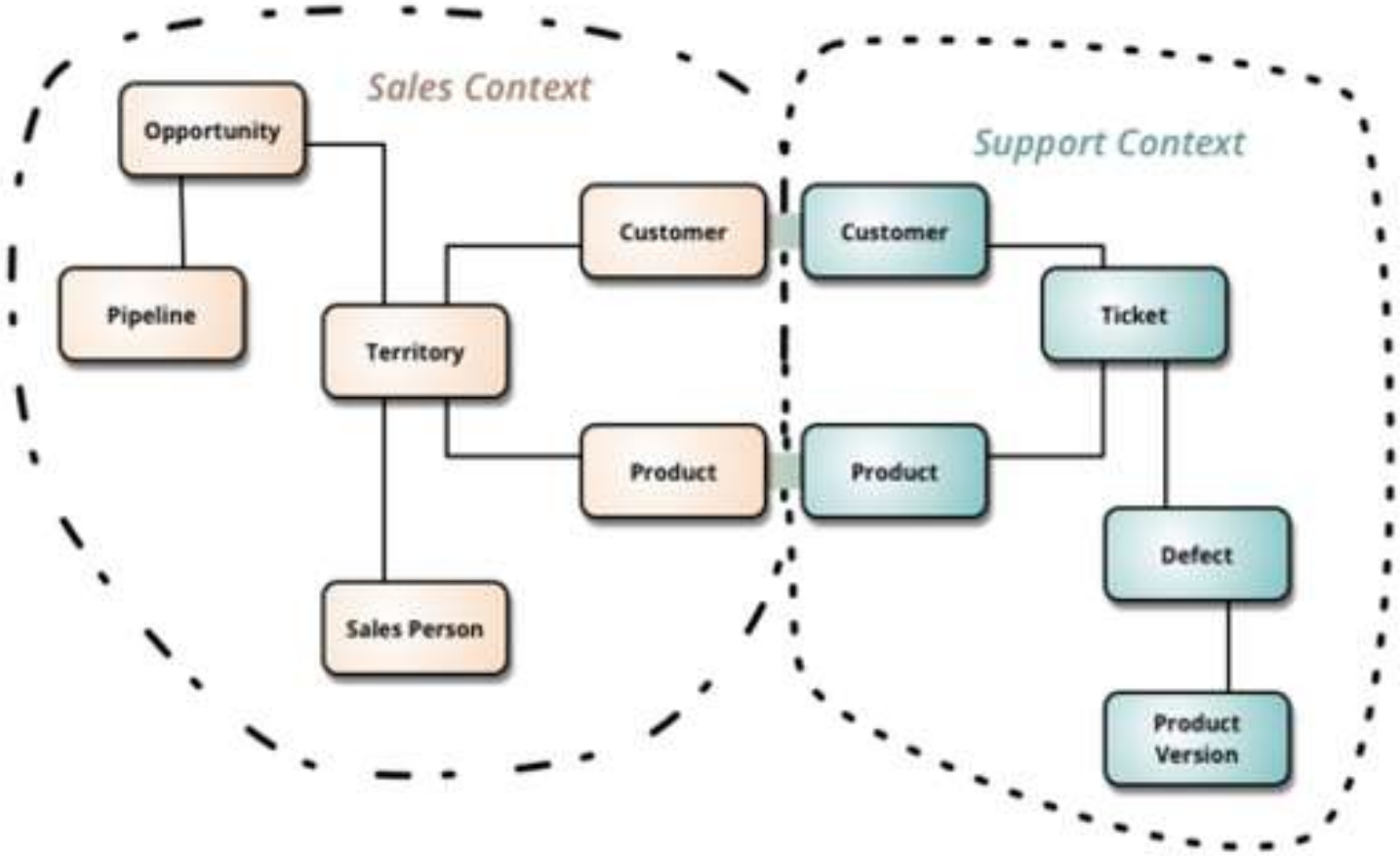

Tenant
Account
Product
Forum
SupportPlan
Payment
User
BacklogItem
Release
Sprint
Incident
Permission
ScheduledBacklogItem
CommittedBacklogItem
Post
CalendarEntry
Volunteer
Team
Reminder
Milestone
TargetDate
ProductOwner
Retrospective
Planning
ResourceManager
Schedule

Strategic


UBIQUITOUS LANGUAGE
cultivate rich model with
names enter
BOUNDED CONTEXT
keep model unified by
CONTINUOUS INTEGRATION
CORE DOMAIN
work in autonomous, clean
assess/overview relationships with
CONTEXT MAP
interdependent contexts form
SHARED KERNEL
overlap allied contexts through
SHARED KERNEL
relate allied contexts as
CUSTOMER /SUPPLIER
minimize translation
CONFORMIST
avoid overinvesting in
GENERIC SUBDOMAINS
segregate the conceptual messes
BIG BALL OF MUD
translate and insulate unilaterally with
ANTI-CORRUPTION LAYER
free teams to go
SEPARATE WAYS
support multiple clients through
OPEN HOST SERVICE
loosely couple contexts through
PUBLISHED LANGUAGE
formalize

Effective design
with
Bounded Contexts

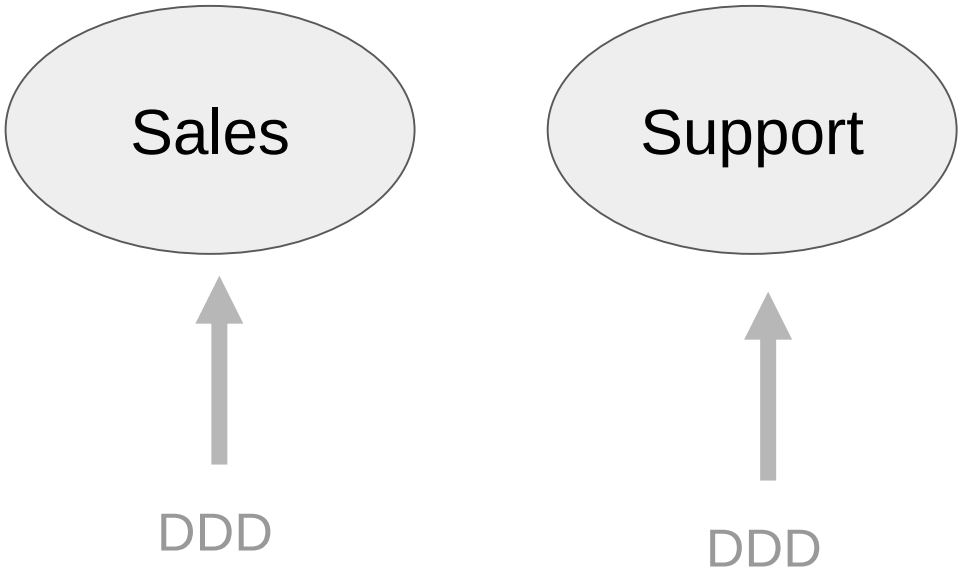

Tenant
User
Role
Forum
Calendar
Discussion
CalendarEntry
Post
Account
Payment
Product
Discussion
BacklogItem
Release
Sprint
Discussion
Task
ScheduledBacklogItem
CommittedBacklogItem
Volunteer
EstimationLogEntry
Team
ProductOwner
TeamMember
SupportPlan
Incident
Agile Project Management Core
ResourceManager
TimeConsumingResource
Schedule
Availability


Sales Context
Opportunity
Pipeline
Territory
Customer
Product
Sales Person
Support Context
Customer
Ticket
Product
Defect
Product Version

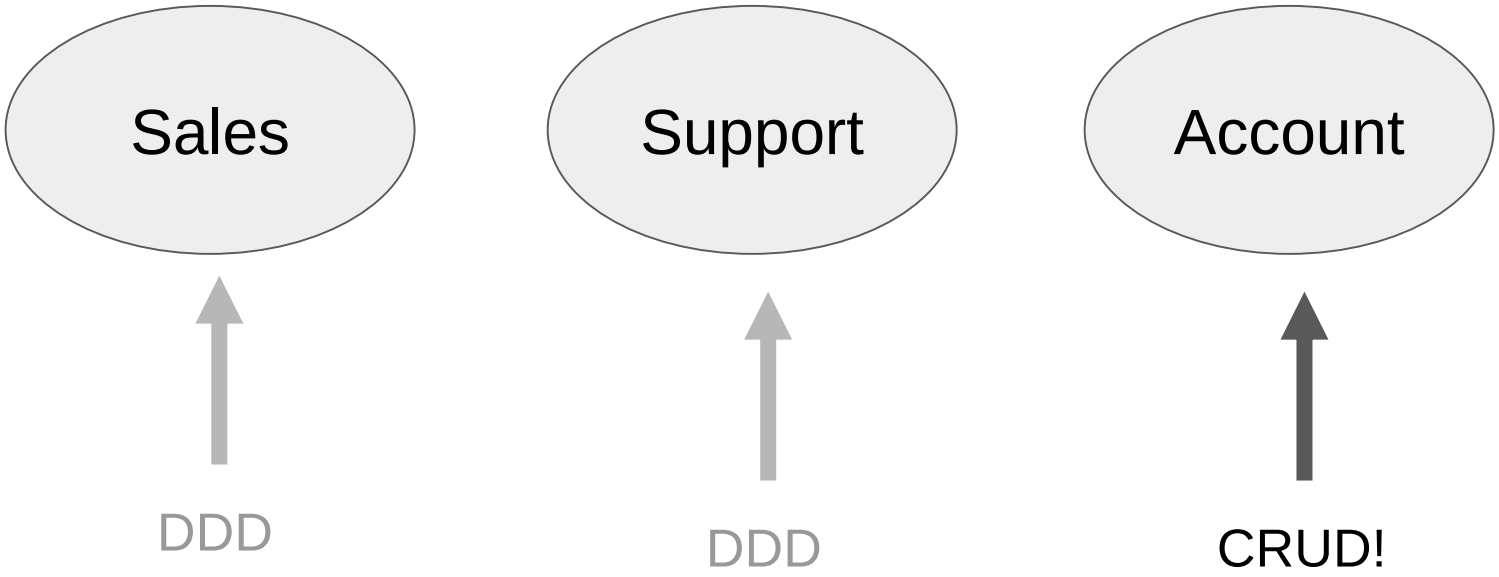
Contexts aren’t just folders or “packages”

Everything could be a context

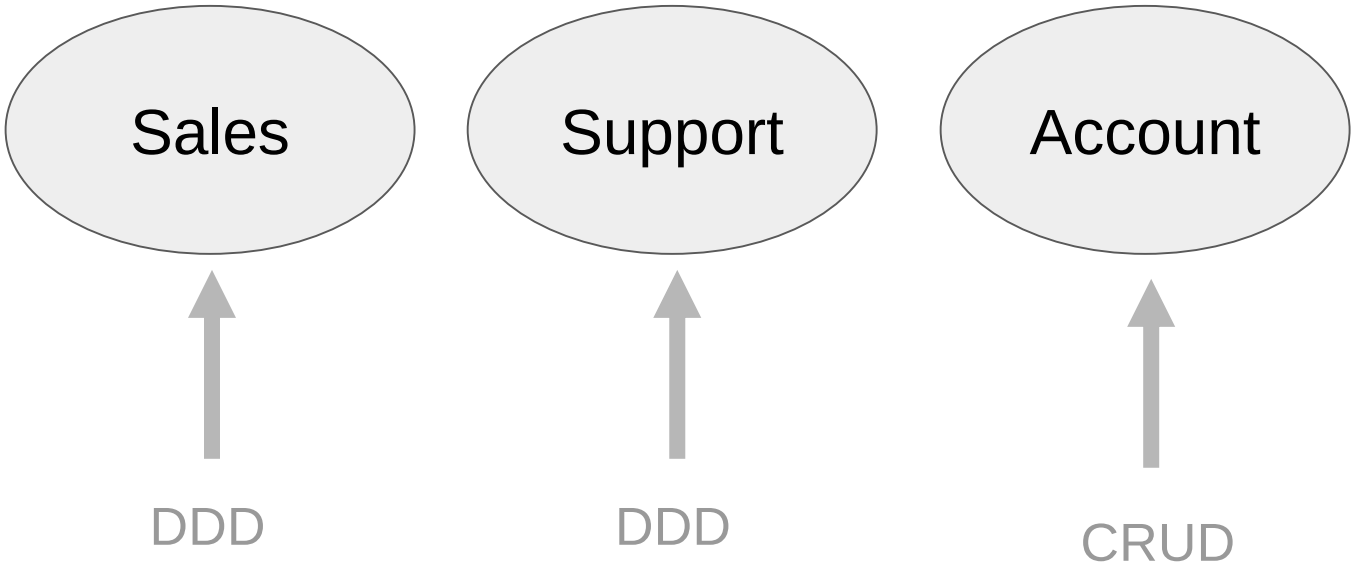
Even a microservice...


Sales
Support
DDD
DDD

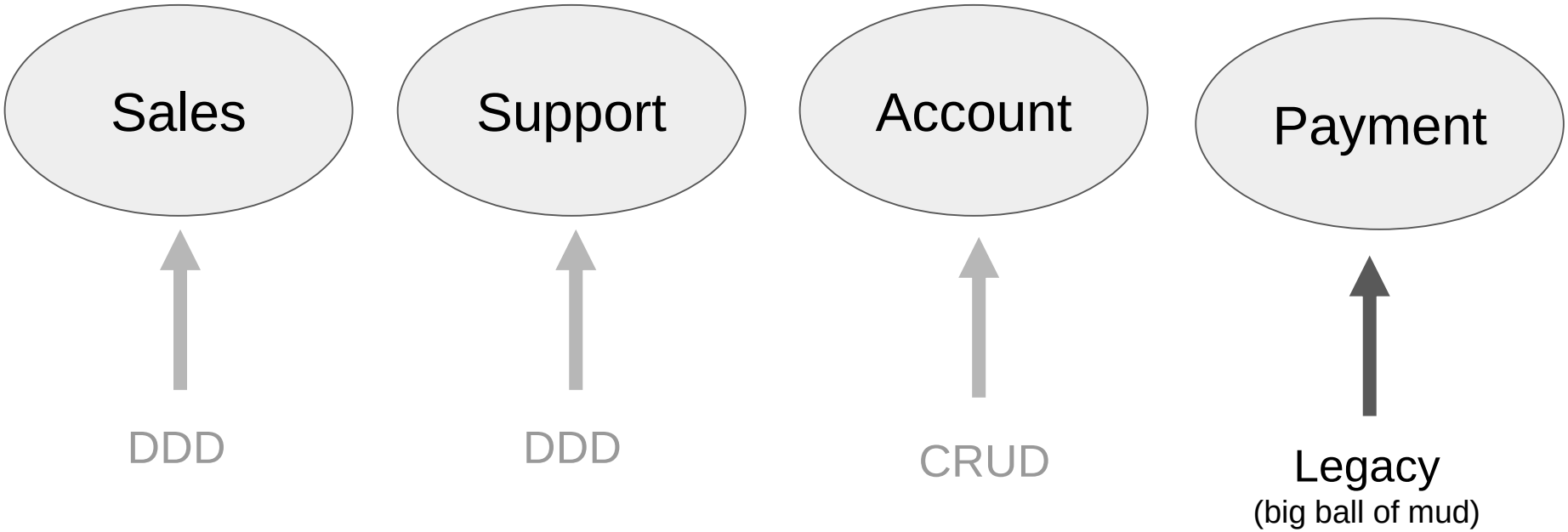
Data-driven Context



Big Ball of Mud as a Context



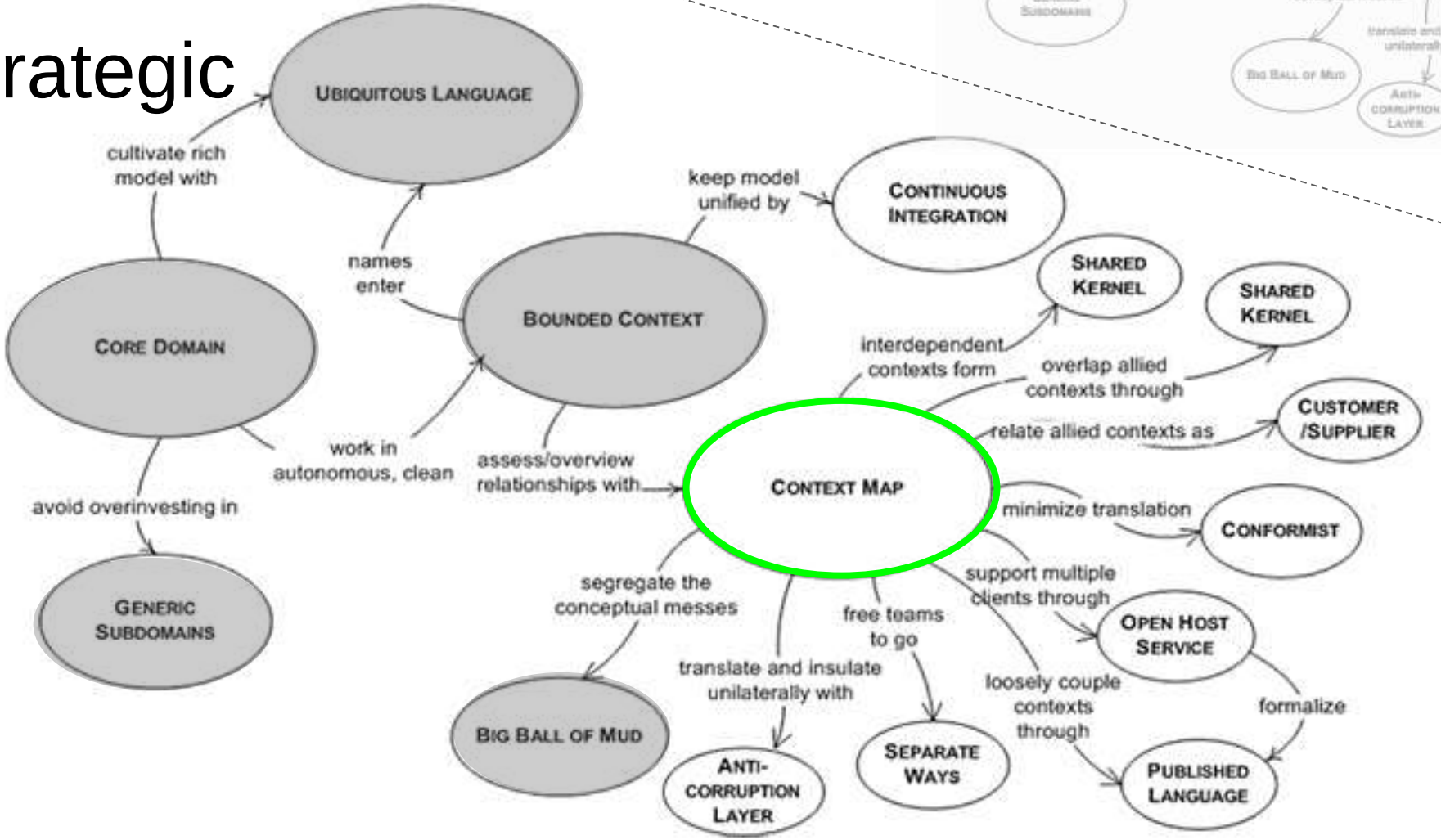
Big Ball of Mud as a Context



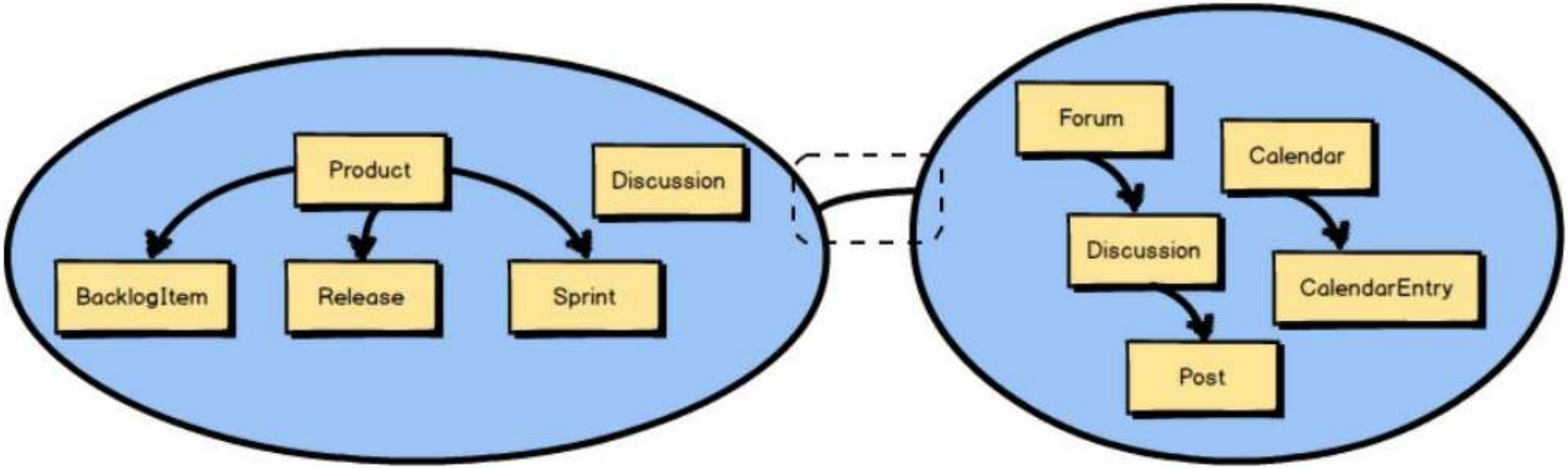
Squads

Bounded Contexts

Strategic

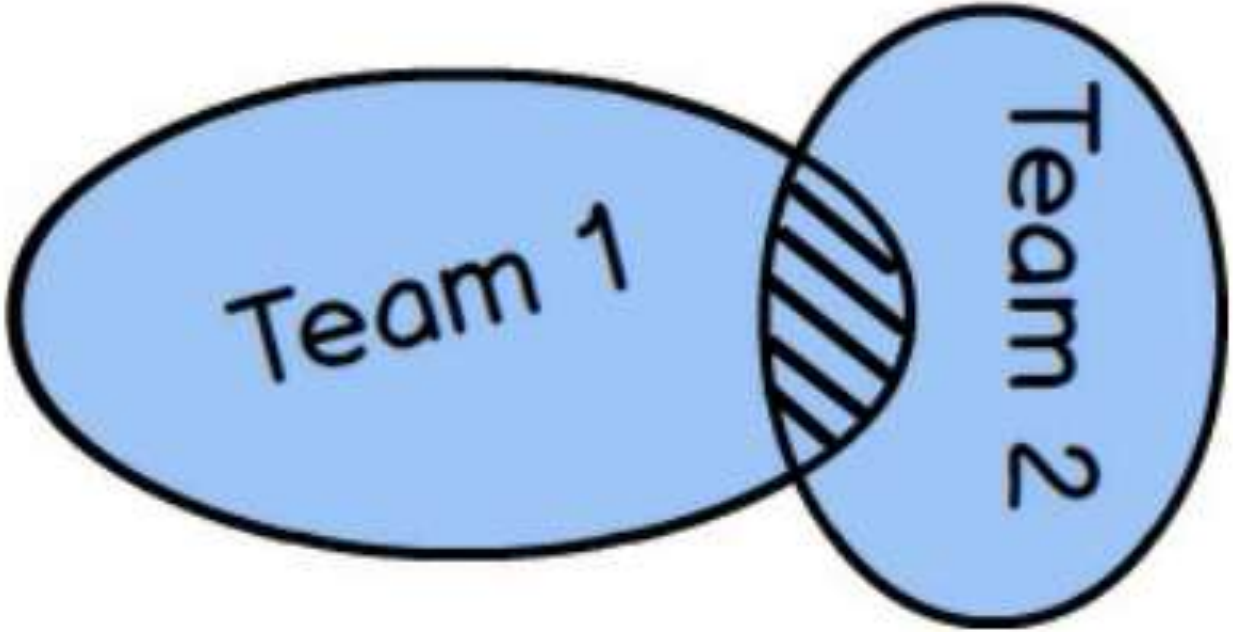

UBIQUITOUS LANGUAGE
cultivate rich model with
names enter
BOUNDED CONTEXT
keep model unified by
CONTINUOUS INTEGRATION
CORE DOMAIN
work in autonomous, clean
assess/overview relationships with
CONTEXT MAP
interdependent contexts form
SHARED KERNEL
overlap allied contexts through
SHARED KERNEL
relate allied contexts as
CUSTOMER /SUPPLIER
minimize translation
CONFORMIST
avoid overinvesting in
GENERIC SUBDOMAINS
segregate the conceptual messes
BIG BALL OF MUD
translate and insulate unilaterally with
ANTI-CORRUPTION LAYER
free teams to go
SEPARATE WAYS
support multiple clients through
OPEN HOST SERVICE
loosely couple contexts through
PUBLISHED LANGUAGE
formalize

Contex Map

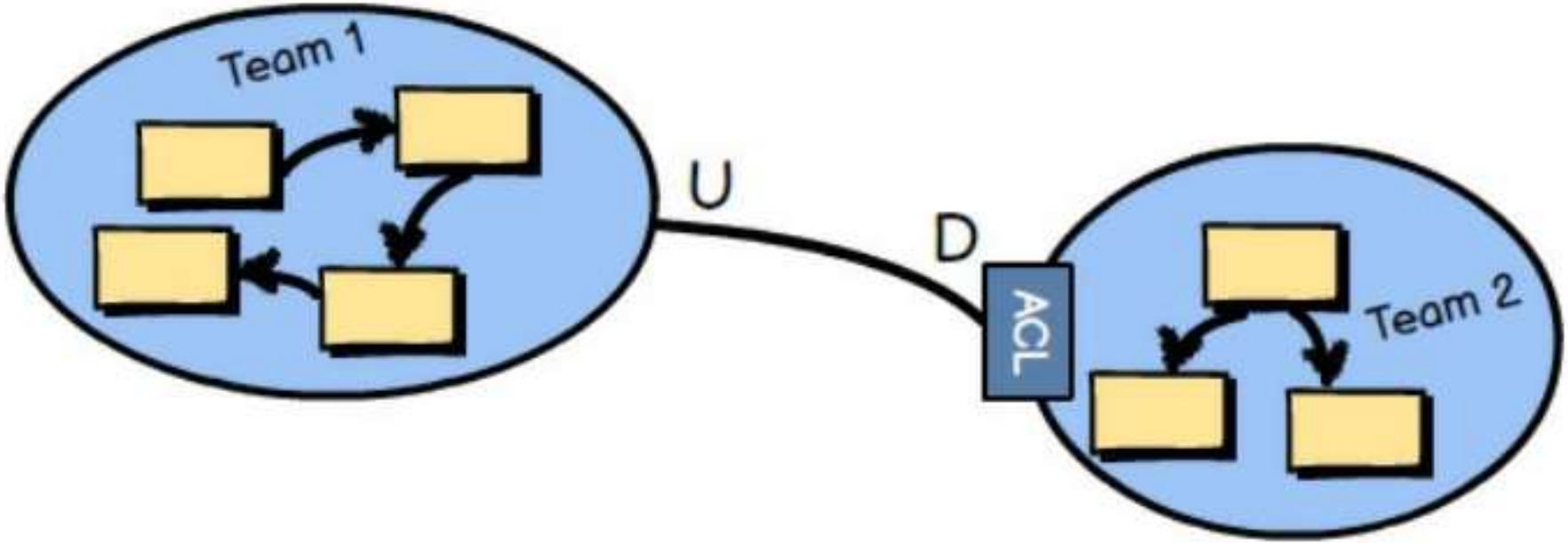

Product
Discussion
BacklogItem
Release
Sprint
Forum
Calendar
Discussion
CalendarEntry
Post

Contex Map - Shared Kernel
Contex Map - Customer/Supplier
Contex Map - Conformist
Contex Map - Partner
Contex Map - Open Host Service
Contex Map - Published Language

Contex Map - Shared Kernel

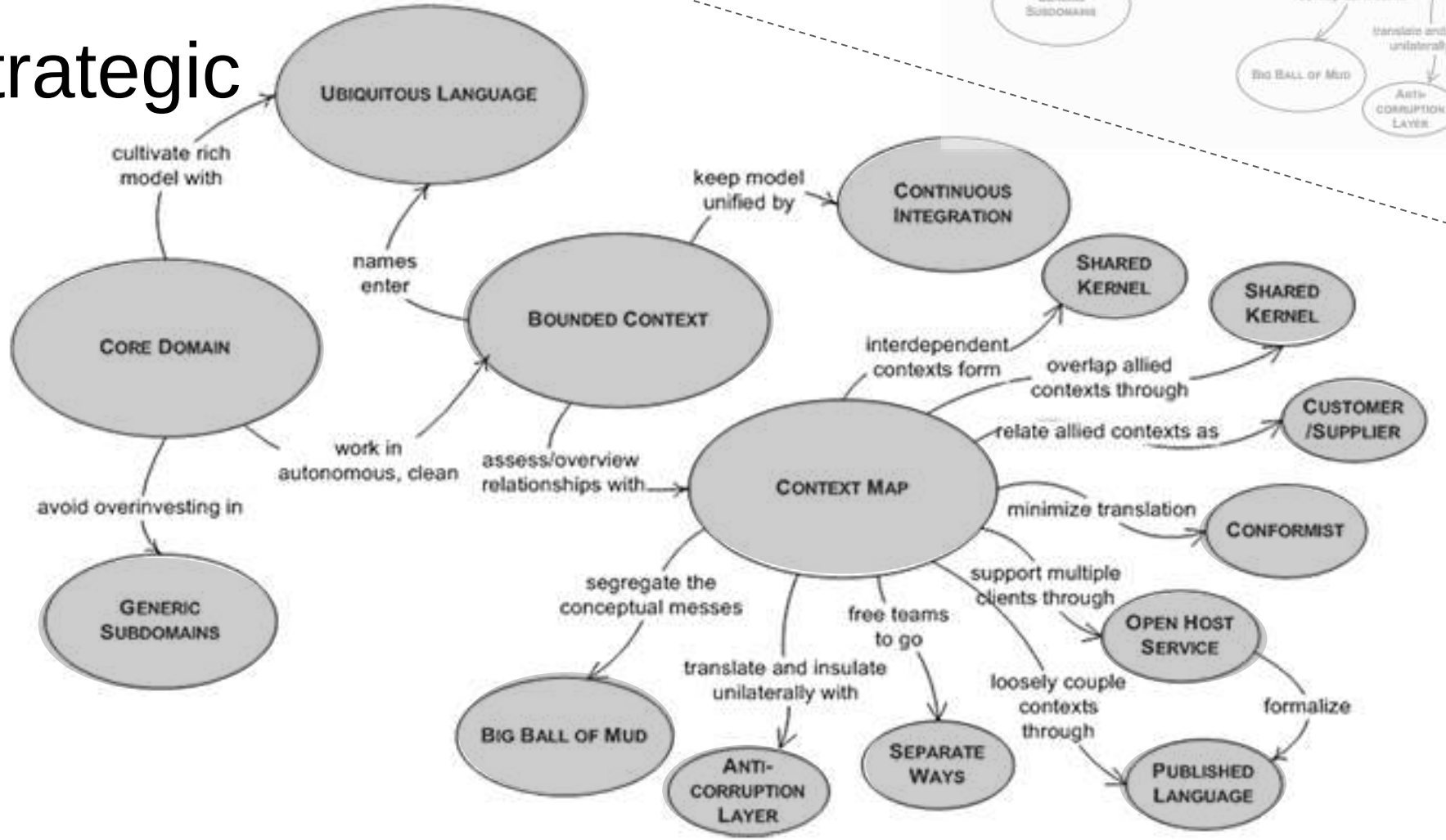


Contex Map - Anticorruption Layer



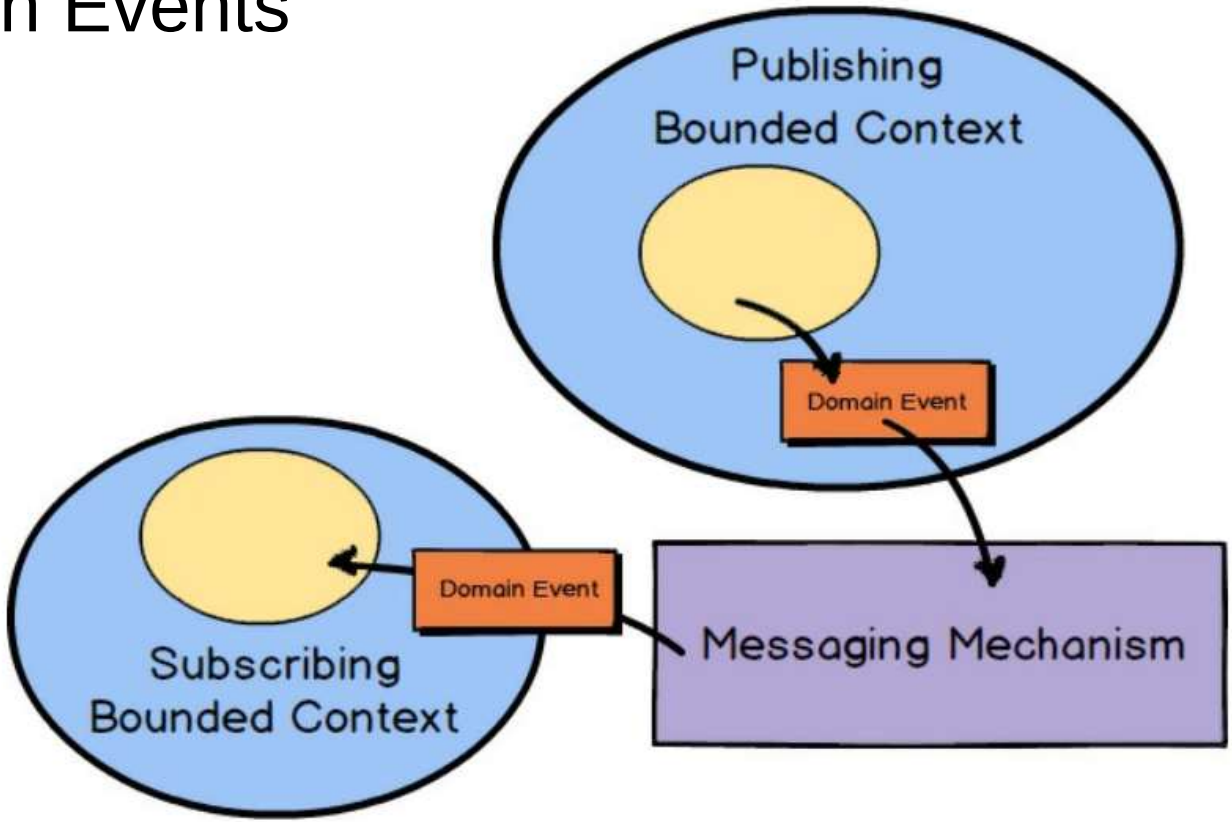
Integration Approaches

Strategic


UBIQUITOUS LANGUAGE
cultivate rich model with
CORE DOMAIN
avoid overinvesting in
GENERIC SUBDOMAINS
names enter
BOUNDED CONTEXT
work in autonomous, clean
keep model unified by
CONTINUOUS INTEGRATION
assess/overview relationships with
CONTEXT MAP
interdependent contexts form
SHARED KERNEL
overlap allied contexts through
SHARED KERNEL
relate allied contexts as
CUSTOMER /SUPPLIER
minimize translation
CONFORMIST
support multiple clients through
OPEN HOST SERVICE
formalize
PUBLISHED LANGUAGE
loosely couple contexts through
SEPARATE WAYS
free teams to go
translate and insulate unilaterally with
ANTI-CORRUPTION LAYER
segregate the conceptual messes
BIG BALL OF MUD

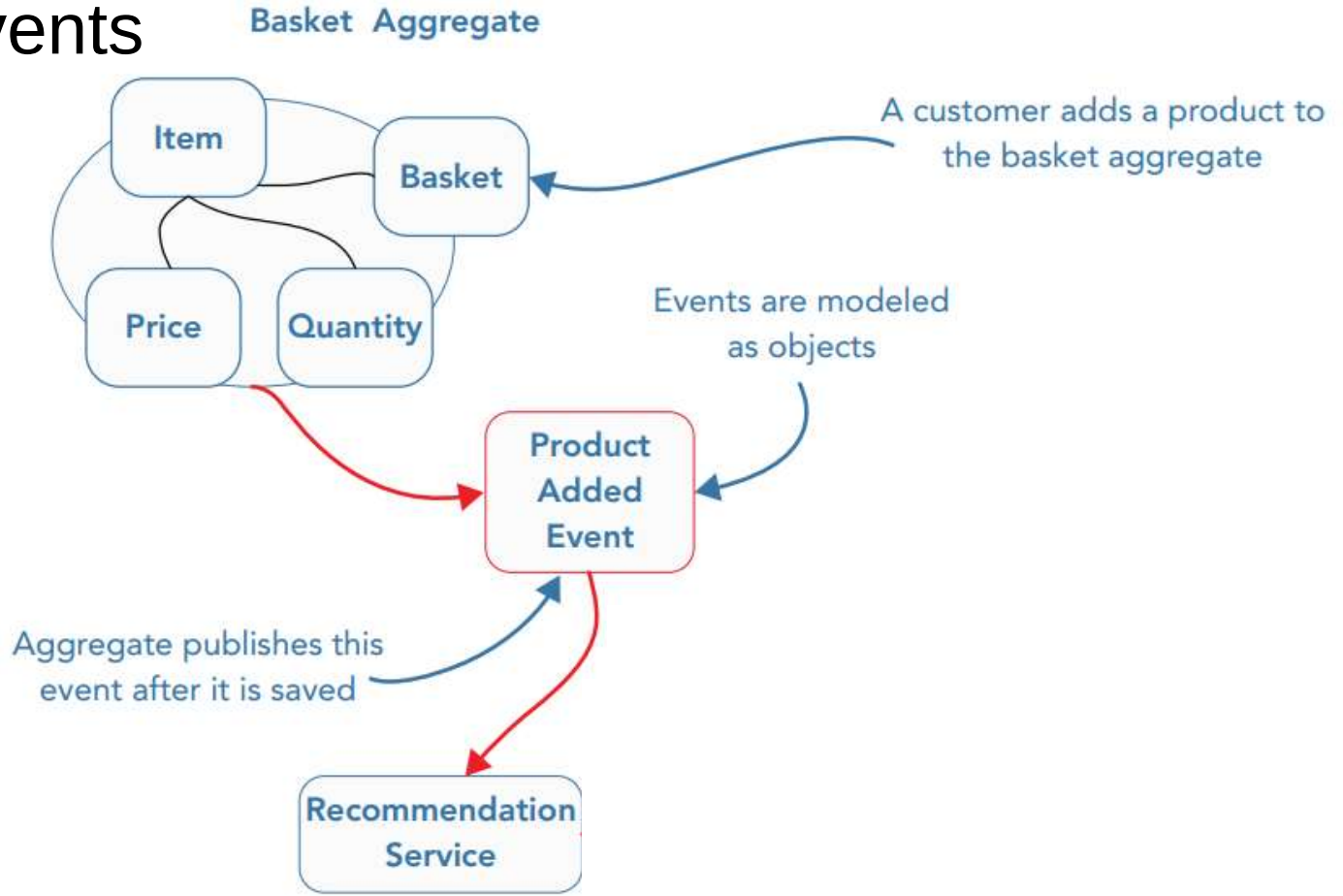
Integrating between bounded contexts using Domain Events

Domain Events



http://udidahan.com/2009/06/14/domain-events-salvation/

Domain Events

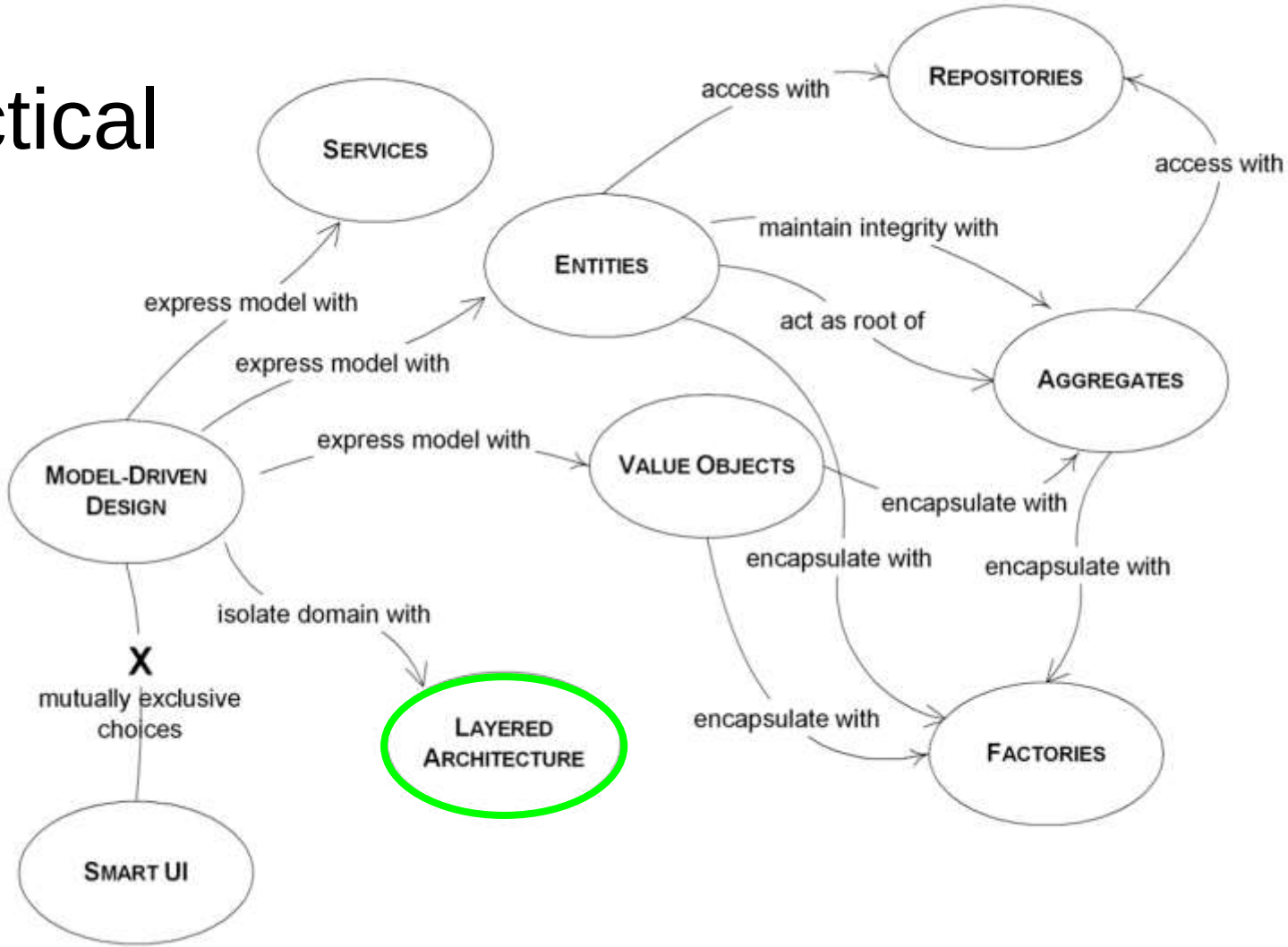


Schedule

- ~~Basic Concepts~~
- ~~Ubiquitous Language~~
- ~~Domain Expert~~
- ~~Domain Model~~
- ~~Architectures types~~
  - ~~DDD~~,
  - ~~Smart UI~~,
  - ~~...~~
- ~~**Strategic Design**~~
  - ~~Bounded Context~~
  - ~~Context Maps~~
  - ~~Domain Events~~
  - ~~Event Storming~~

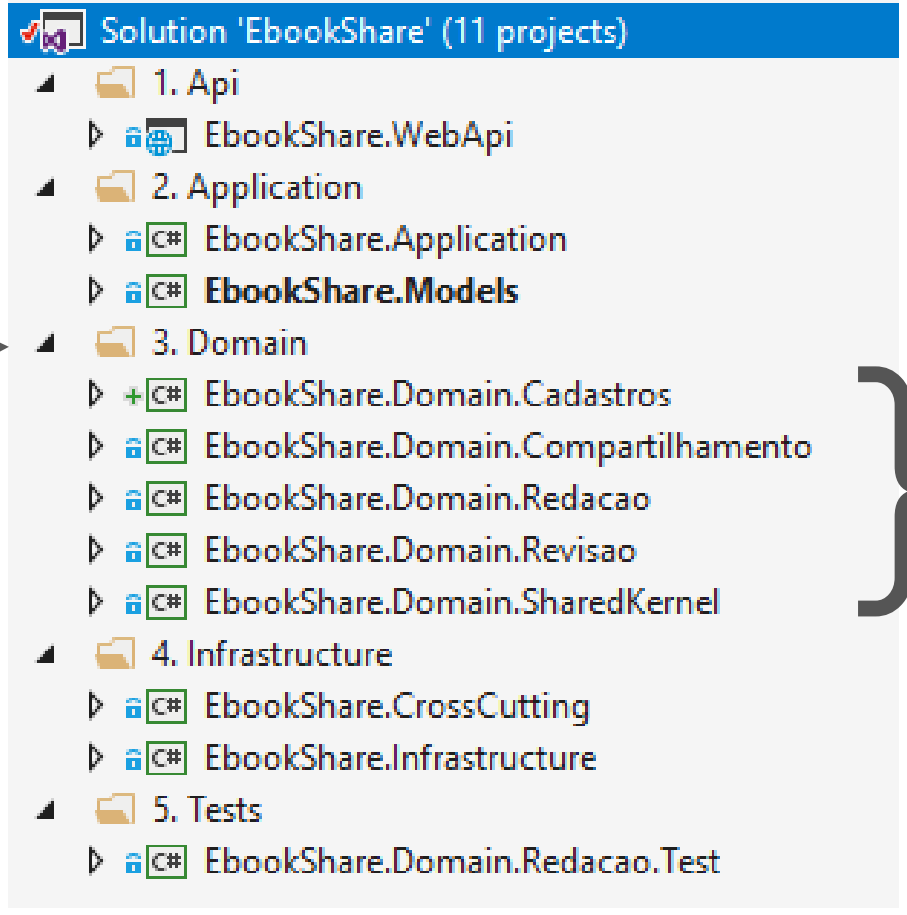
- **Tactical Design** (building blocks)
  - Layered Architecture
  - Entities
  - Value Objects
  - Aggregates
  - Domain Services
  - Factories
  - Repositories
- CQRS
- SOA
  - Event Driven Architecture
- Event Sourcing
- Final Thoughts

Tactical


SERVICES
access with
REPOSITORIES
access with
ENTITIES
maintain integrity with
express model with
act as root of
express model with
AGGREGATES
express model with
MODEL-DRIVEN DESIGN
VALUE OBJECTS
encapsulate with
encapsulate with
encapsulate with
isolate domain with
X
mutually exclusive choices
LAYERED ARCHITECTURE
encapsulate with
FACTORIES
SMART UI

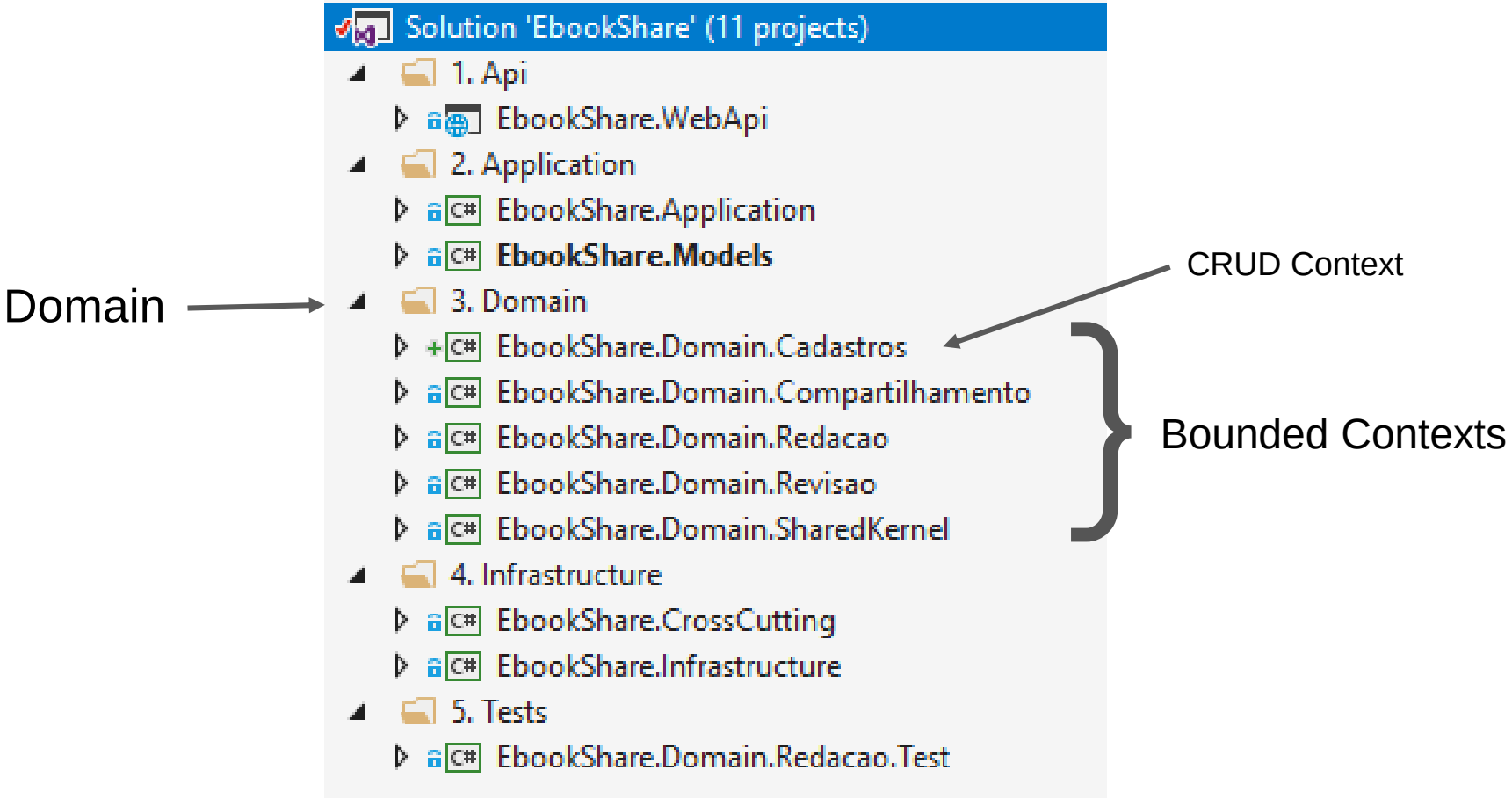
Layered Architecture

Domain

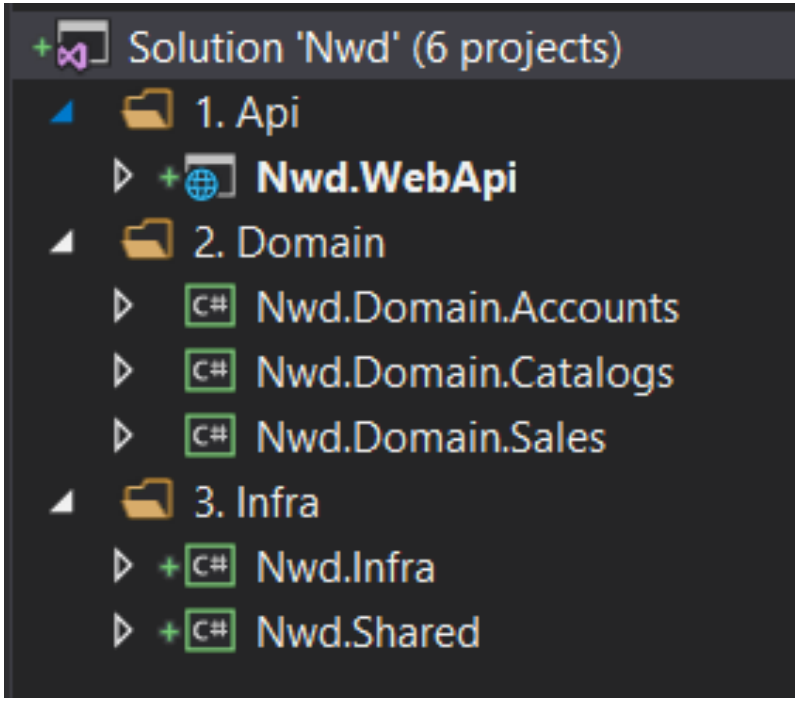


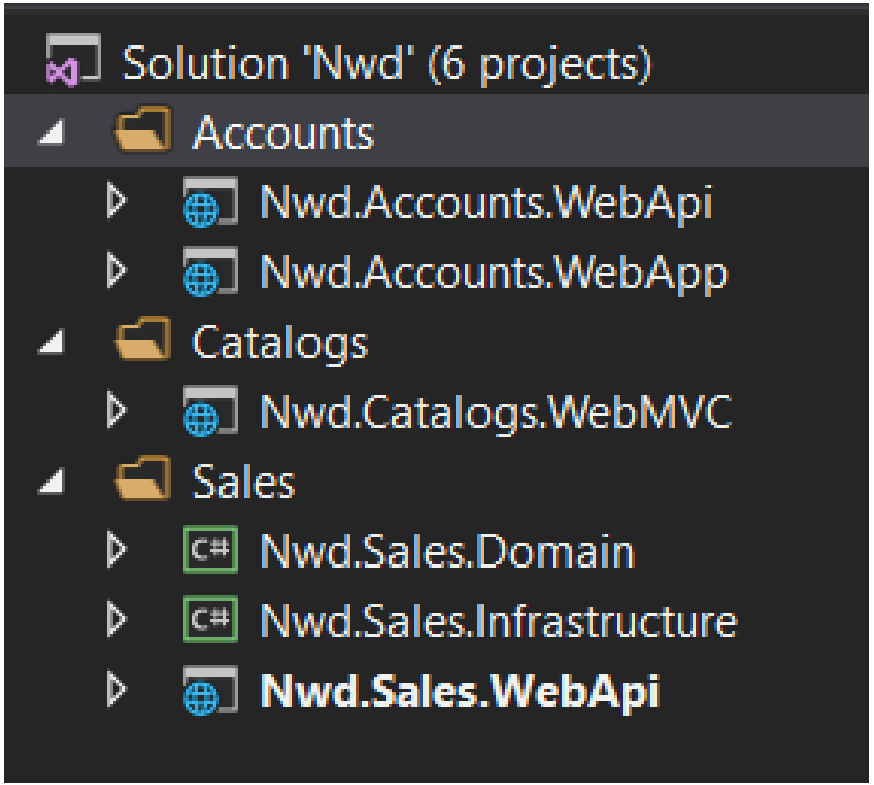
Bounded Contexts

Layered Architecture

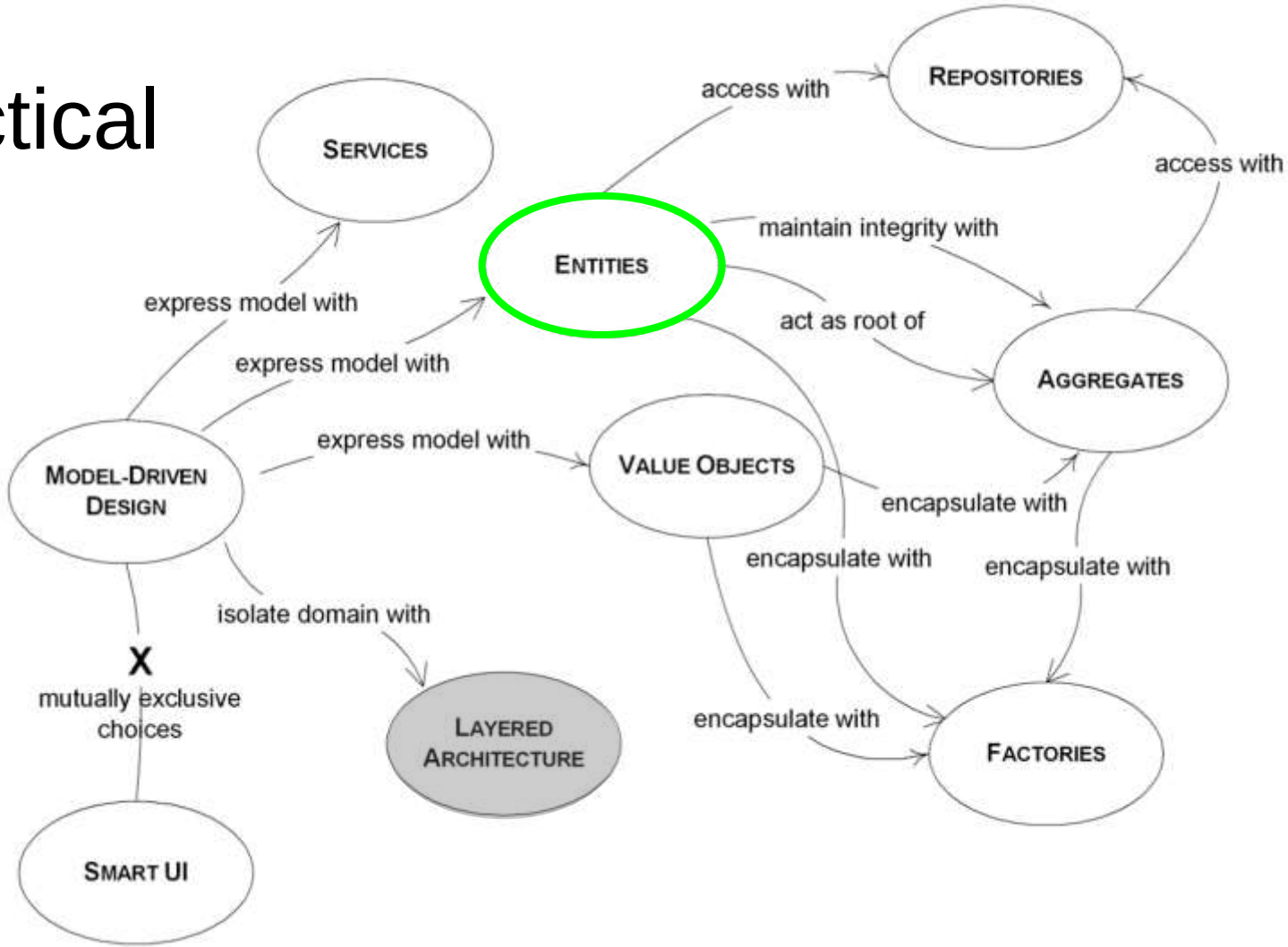


Layered Architecture

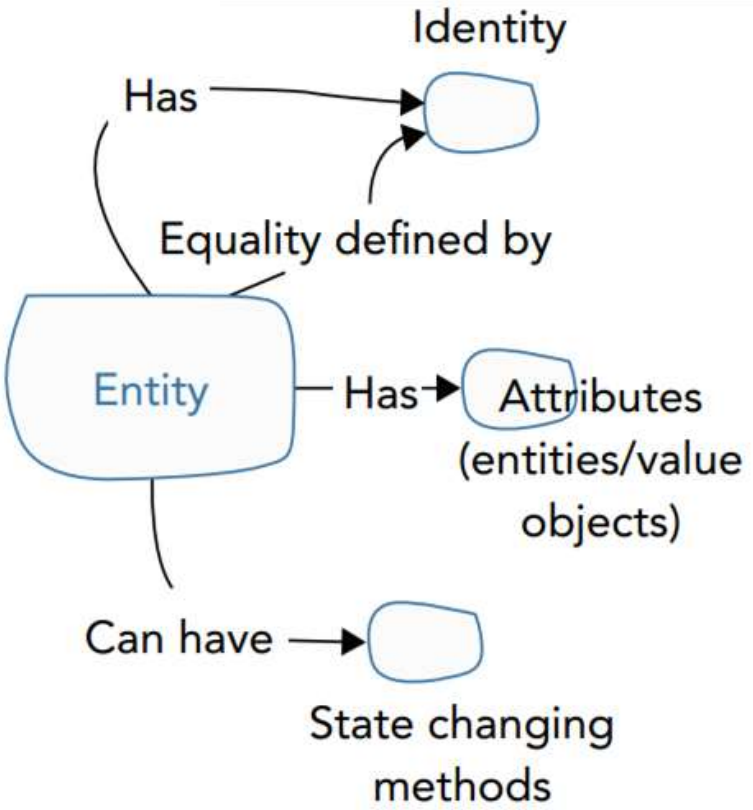




Tactical

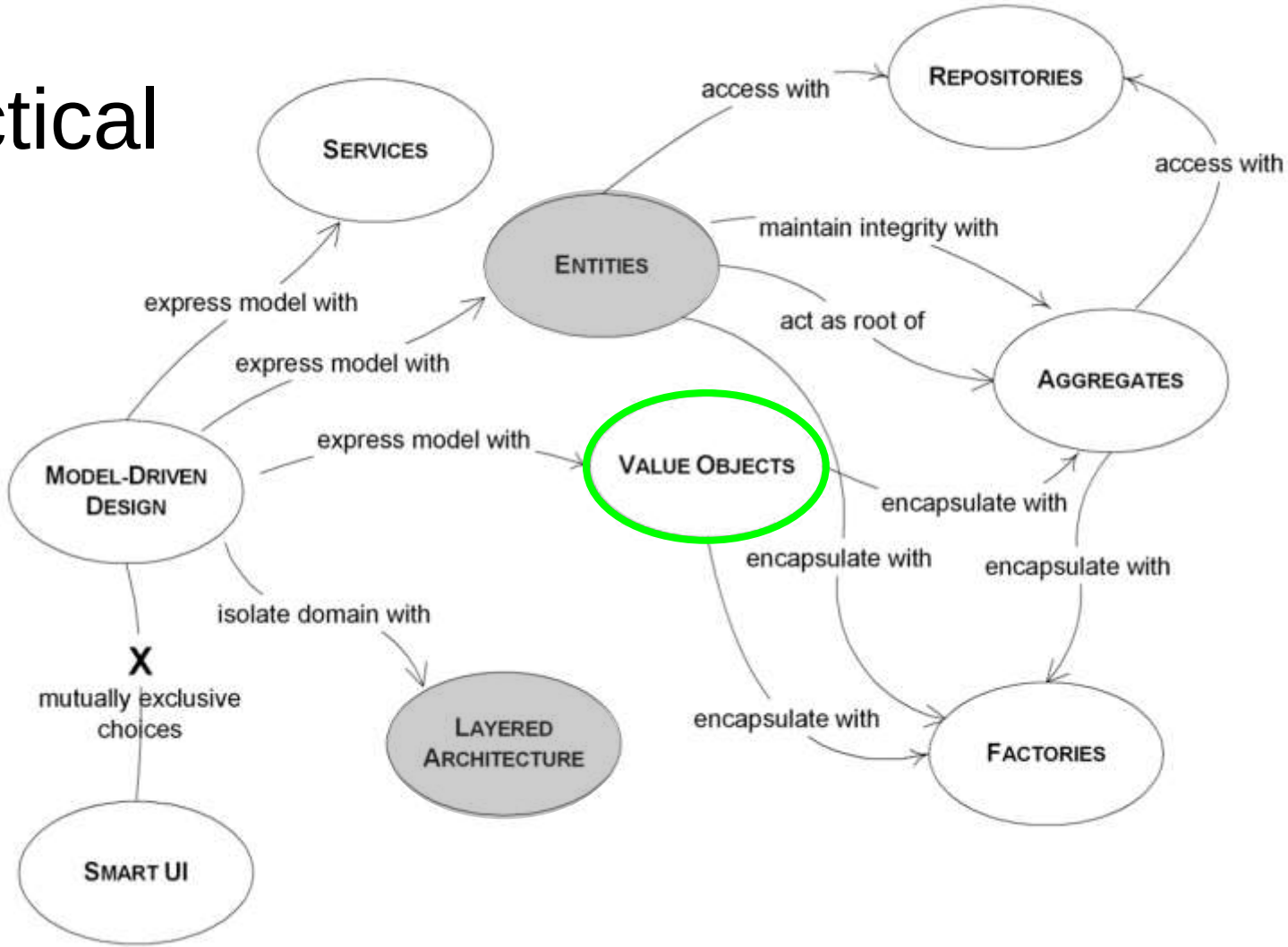

SERVICES
access with
REPOSITORIES
access with
ENTITIES
maintain integrity with
express model with
act as root of
express model with
AGGREGATES
express model with
MODEL-DRIVEN DESIGN
VALUE OBJECTS
encapsulate with
encapsulate with
encapsulate with
isolate domain with
X
mutually exclusive choices
LAYERED ARCHITECTURE
encapsulate with
FACTORIES
SMART UI

Entities

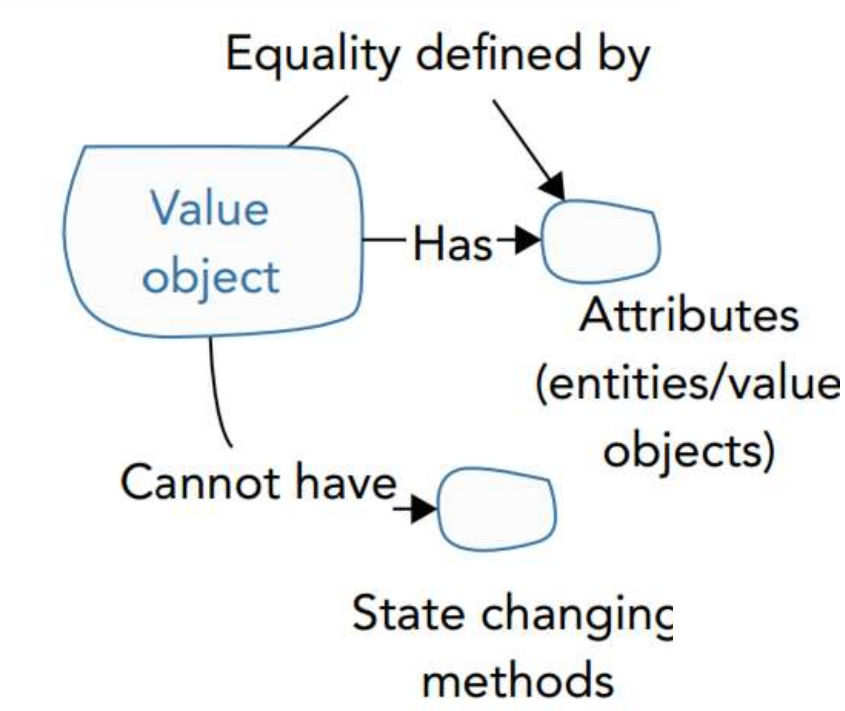

Identity
Has
Equality defined by
Entity
Has
Attributes
(entities/value
objects)
Can have
State changing
methods

```ruby
class Item < Infra::Models::ApplicationRecord
  after_initialize :validate

  belongs_to :product

  attribute :quantity, :integer

  def validate
    validate_quantity!
    validate_product_presence!
  end

  def change_quantity(quantity)
    validate_quantity!
    self.quantity = quantity
  end

  private

  def validate_quantity!
    raise Exceptions::BusinessException.new('Product must be greater than zero')  unless self.quantity.
  end

  def validate_product_presence!
    raise Exceptions::BusinessException.new('Product must be informed')  unless self.product.present?
  end
```

Tactical


SERVICES
access with
REPOSITORIES
access with
ENTITIES
maintain integrity with
express model with
act as root of
express model with
AGGREGATES
express model with
MODEL-DRIVEN DESIGN
VALUE OBJECTS
encapsulate with
encapsulate with
encapsulate with
isolate domain with
X
mutually exclusive choices
LAYERED ARCHITECTURE
encapsulate with
FACTORIES
SMART UI

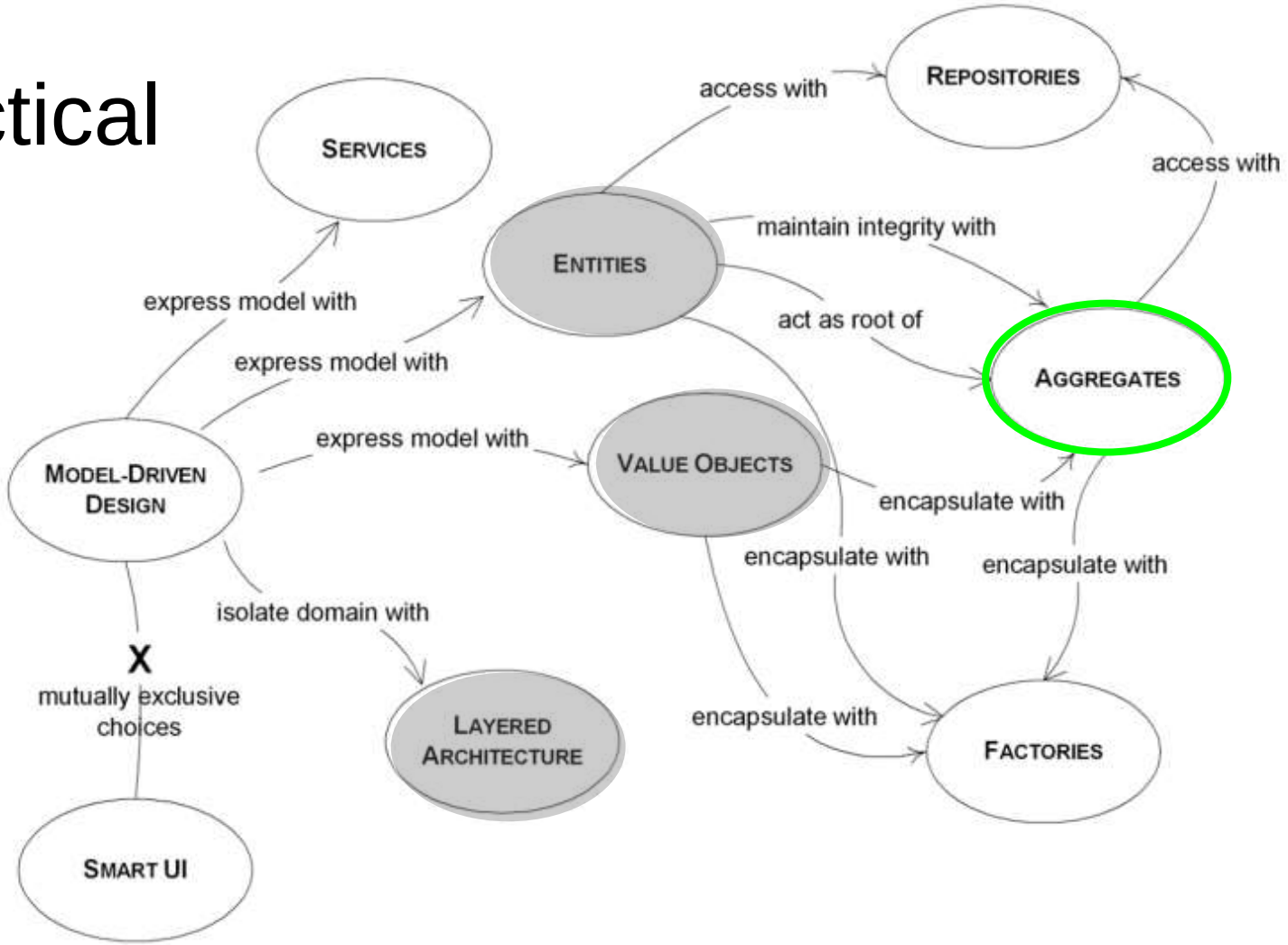
Value Objects



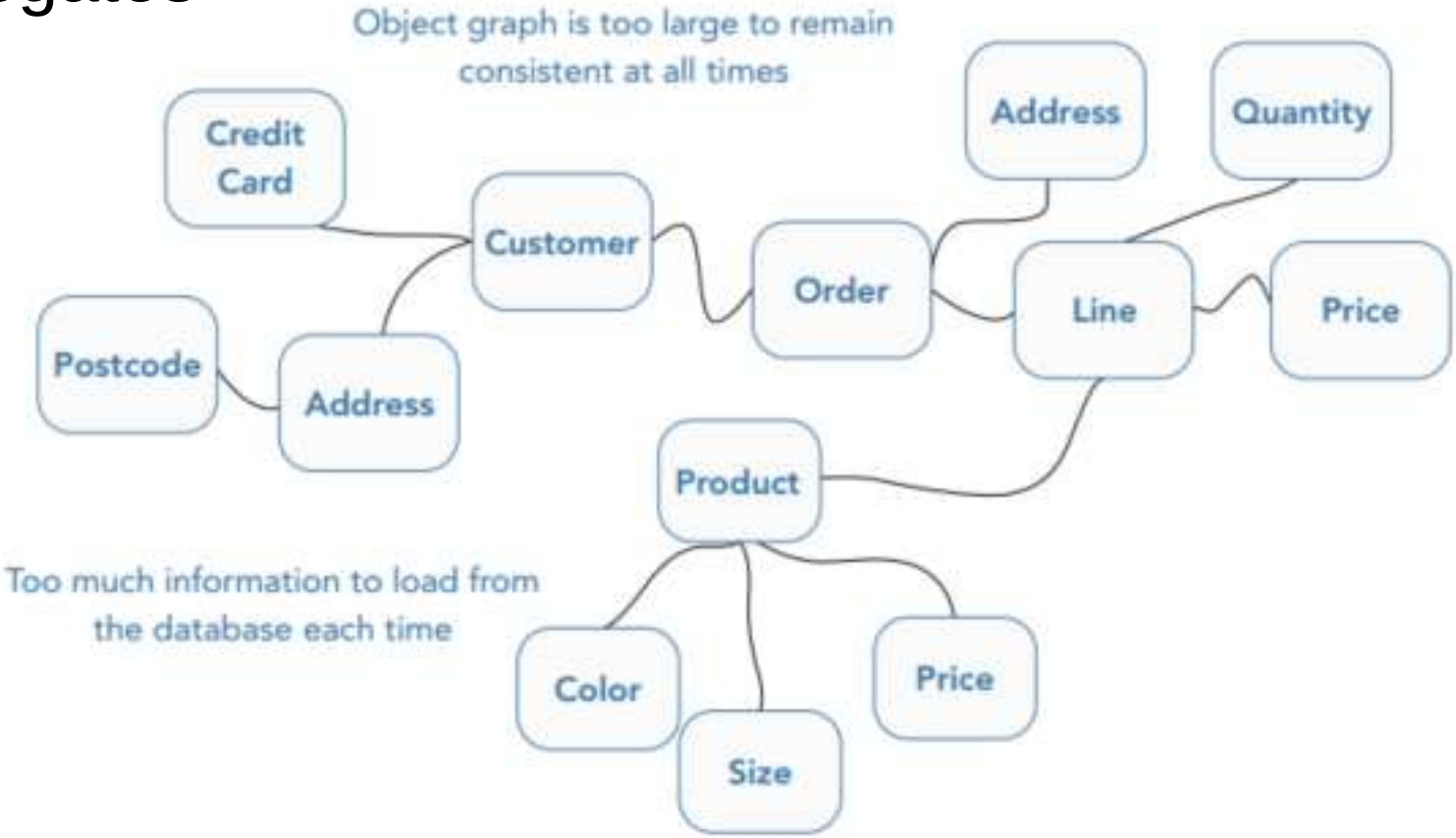
Mostrar money object… bem melhor pra mostrar vo

```
public class Endereco : ValueObject<Endereco>
{
    public Endereco(string logradouro, string cidade, string estado, Cep cep)
    {
        Logradouro = logradouro;
        Cidade = cidade;
        Estado = estado;
        Cep = cep;
    }

    public string Logradouro { get; private set; }
    public string Cidade { get; private set; }
    public string Estado { get; private set; }
    public Cep Cep { get; private set; }
}
```

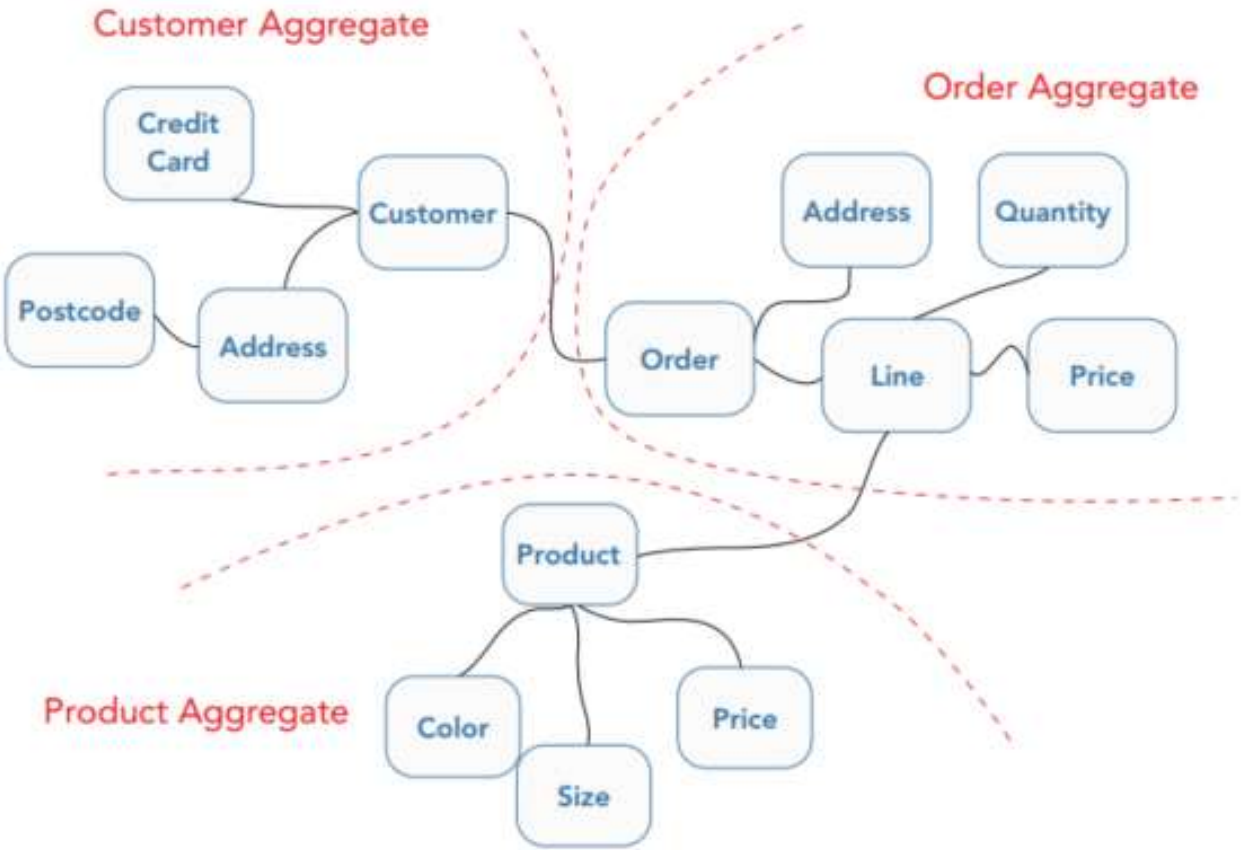
Tactical


SERVICES
access with
REPOSITORIES
access with
ENTITIES
maintain integrity with
express model with
act as root of
express model with
AGGREGATES
express model with
VALUE OBJECTS
MODEL-DRIVEN DESIGN
encapsulate with
encapsulate with
encapsulate with
isolate domain with
X
mutually exclusive choices
LAYERED ARCHITECTURE
encapsulate with
FACTORIES
SMART UI

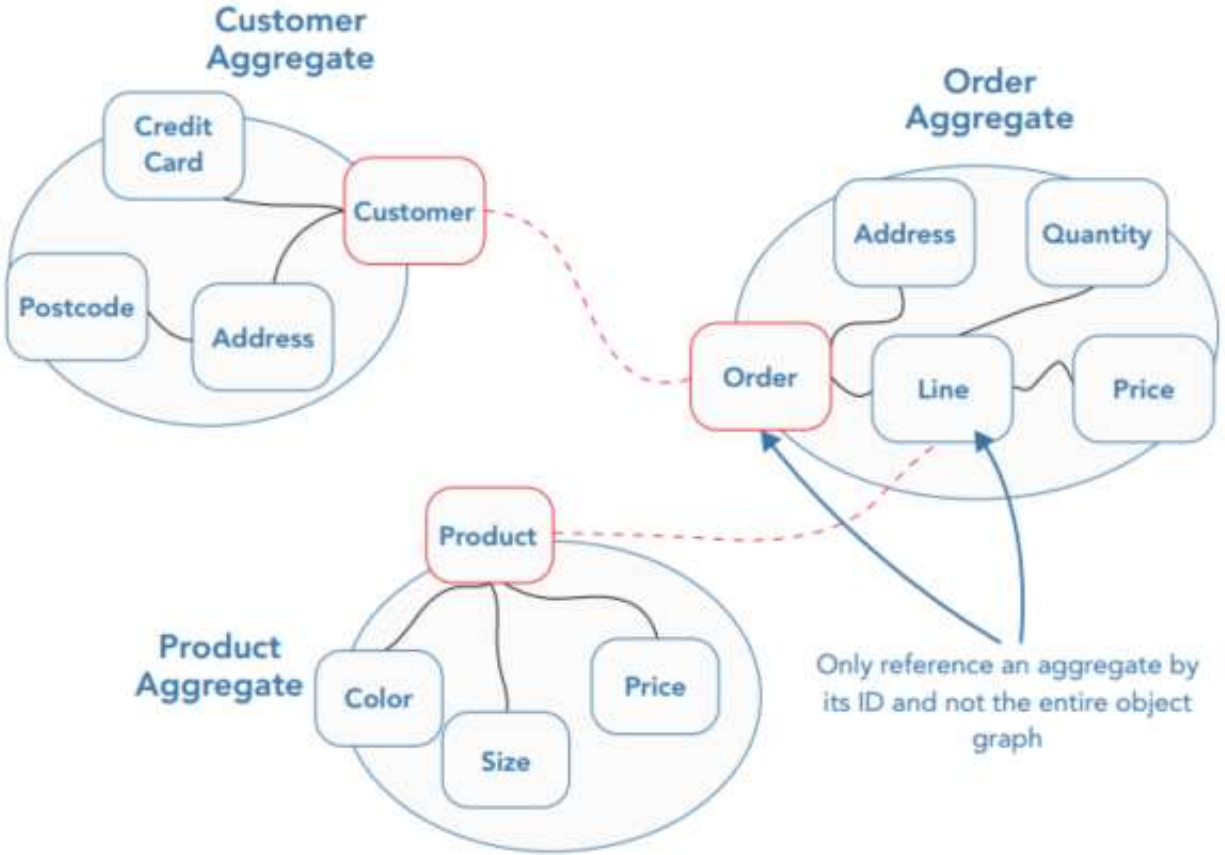
Aggregates


Object graph is too large to remain consistent at all times
Credit Card
Customer
Address
Quantity
Order
Line
Price
Postcode
Address
Product
Too much information to load from the database each time
Color
Size
Price

Aggregates

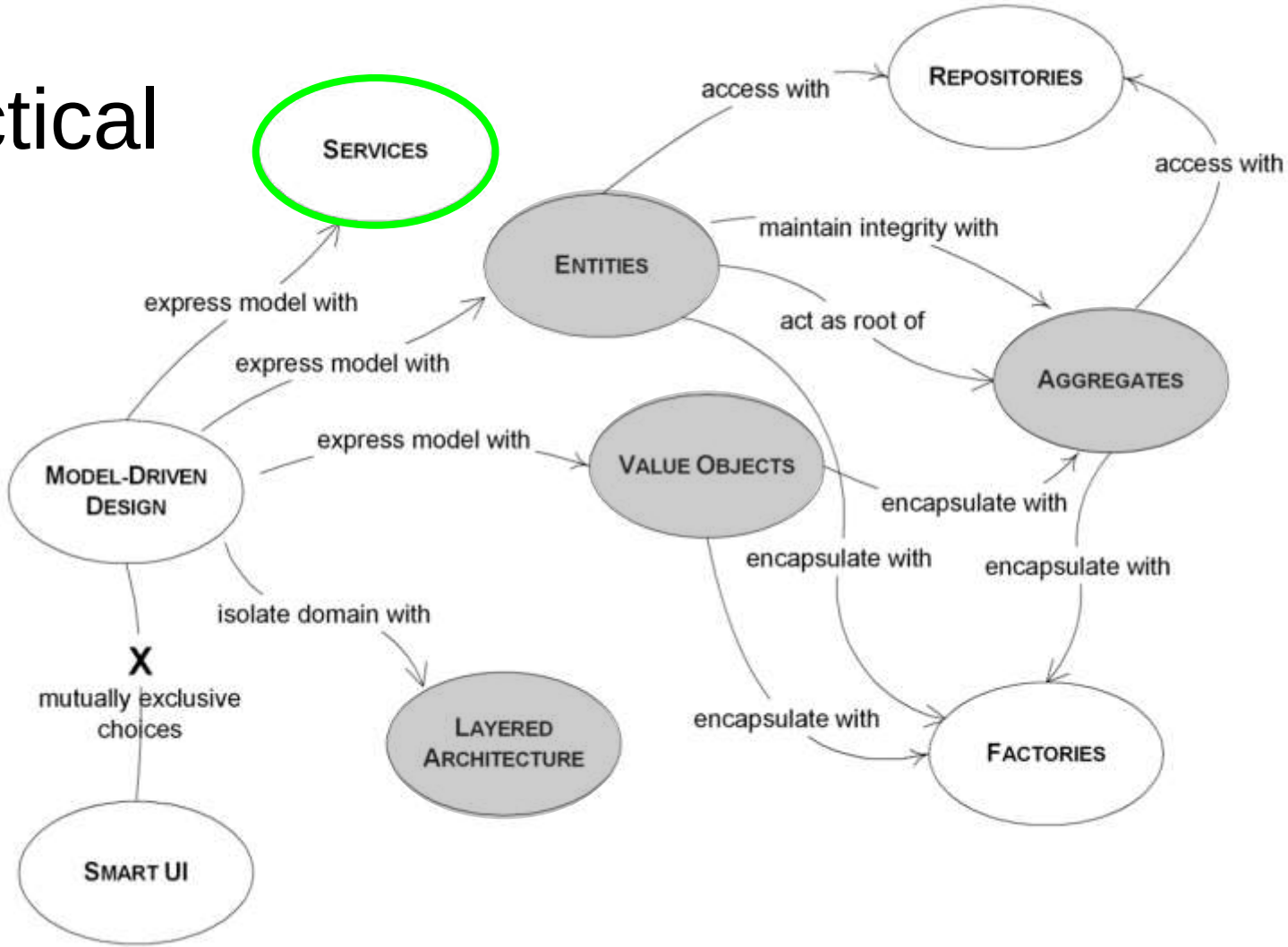

Customer Aggregate
Order Aggregate
Credit Card
Customer
Address
Quantity
Postcode
Address
Order
Line
Price
Product
Product Aggregate
Color
Price
Size

Aggregates

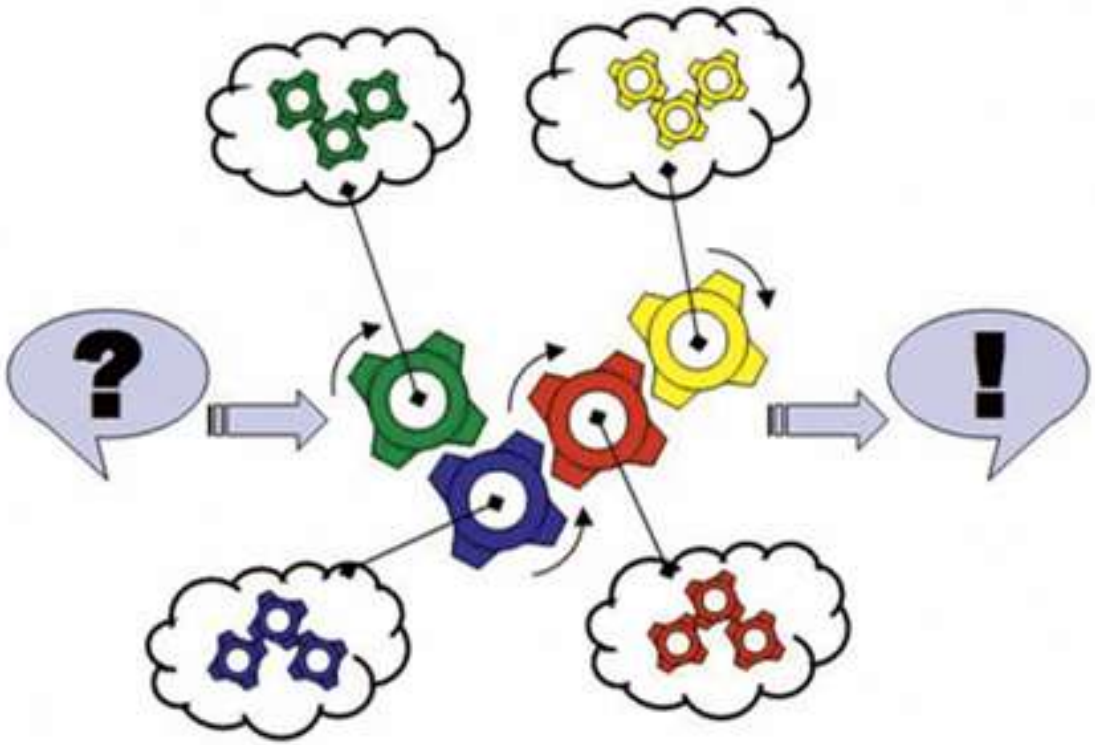

Customer Aggregate
Credit Card
Customer
Postcode
Address
Order Aggregate
Address
Quantity
Order
Line
Price
Product
Product Aggregate
Color
Price
Size
Only reference an aggregate by its ID and not the entire object graph

```ruby
class Order < Infra::Models::ApplicationRecord
  has_many :items, autosave: true
  attribute :customer, :string

  def add_product(product, quantity)
    raise Exceptions::BusinessException.new('Product already exists') if product_already_exists?(proc

    items << Item.new(quantity: quantity, product: product)
  end

  def change_product_quantity(product, quantity)
    validate_product_presence!(product)
    item = items.find { |i| i.product == product }
    item.change_quantity(quantity)
  end

  def remove_product(product)
    validate_product_presence!(product)
    items = self.items.reject { |item| item.product == product }
    self.items = items
  end

  private
  # ...
end
```

Tactical


SERVICES
access with
REPOSITORIES
access with
ENTITIES
maintain integrity with
express model with
act as root of
AGGREGATES
express model with
express model with
MODEL-DRIVEN DESIGN
VALUE OBJECTS
encapsulate with
encapsulate with
encapsulate with
isolate domain with
X
mutually exclusive choices
LAYERED ARCHITECTURE
encapsulate with
FACTORIES
SMART UI

Domain Services

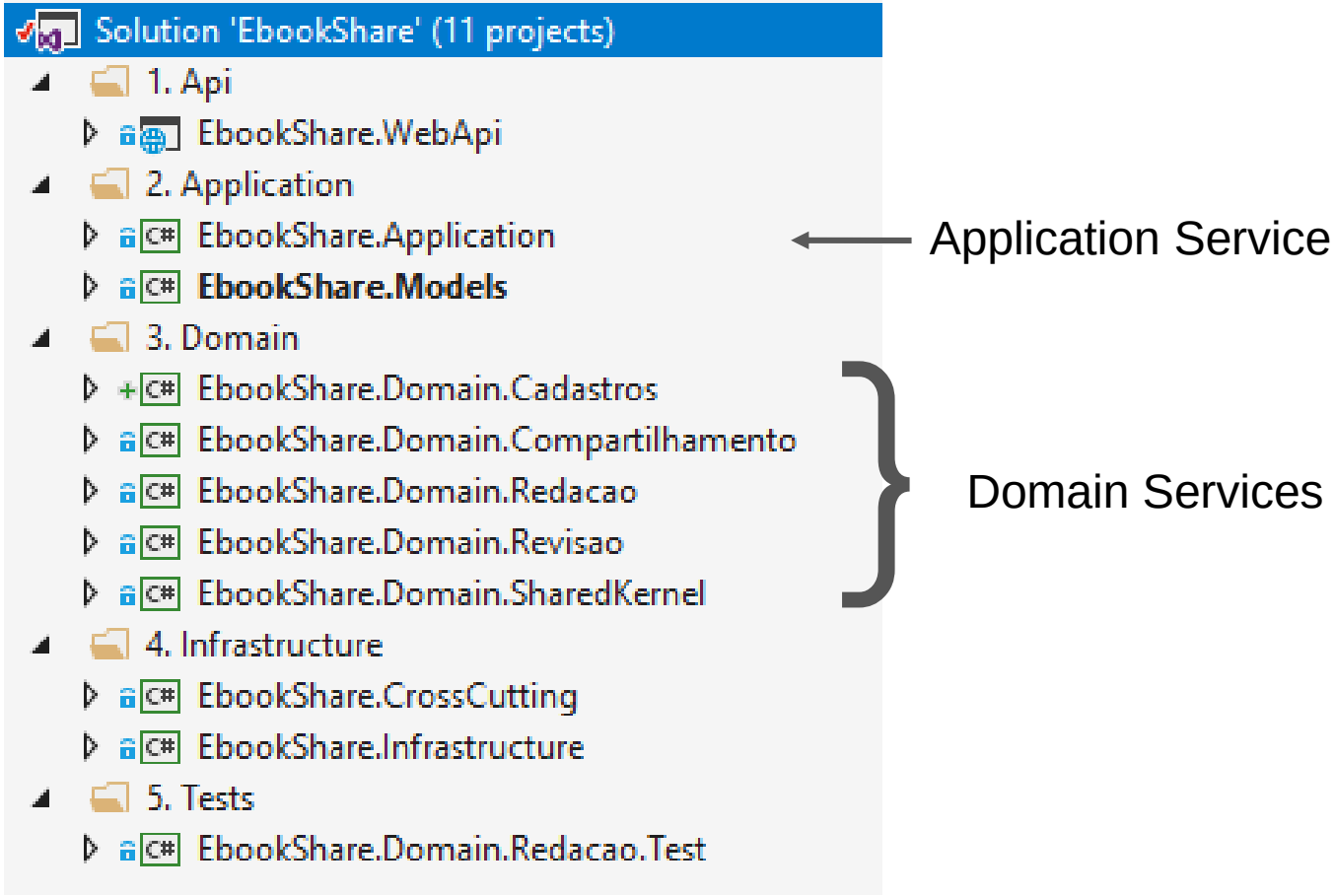


Domain Services

VS

Application Services

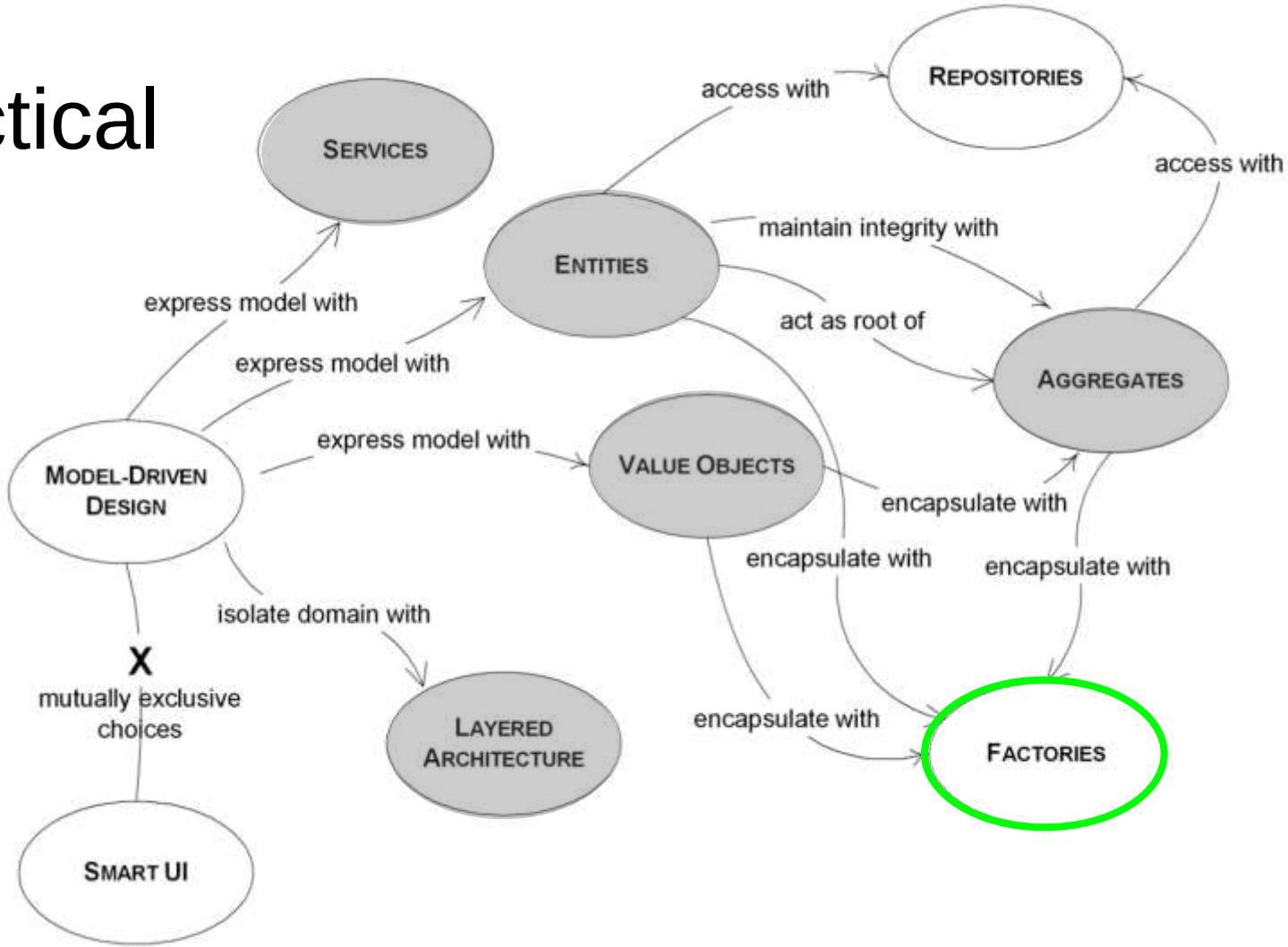
Domain Service vs Application Service



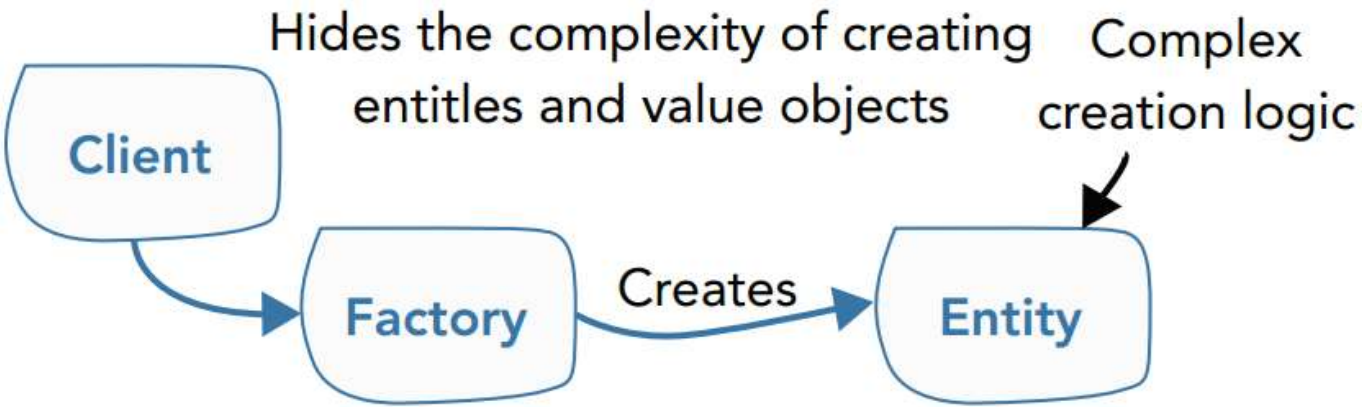
Application Service

```ruby
class OrderApplication
  def initialize(repositories = {})
    @order_repository = repositories.fetch(:order) { Infra::Repositories::OrderRepository.new }
    @product_repository = repositories.fetch(:product) { Infra::Repositories::ProductRepository.new }
  end

  def create_order(create_order_command)
    order = Domain::Order::Order.new(customer: create_order_command.customer)

    ActiveRecord::Base.transaction do
      @order_repository.save(order)

      order.id
    end
  end

  def add_product(add_product_command)
    order = @order_repository.find_by_id(add_product_command.order_id)
    product = @product_repository.find_by_id(add_product_command.product_id)

    ActiveRecord::Base.transaction do
      order.add_product(product, add_product_command.quantity)
```

Tactical


SERVICES
access with
REPOSITORIES
access with
ENTITIES
maintain integrity with
express model with
act as root of
AGGREGATES
express model with
express model with
MODEL-DRIVEN DESIGN
VALUE OBJECTS
encapsulate with
encapsulate with
encapsulate with
isolate domain with
X
mutually exclusive choices
LAYERED ARCHITECTURE
encapsulate with
FACTORIES
SMART UI

Factories


Hides the complexity of creating
entitles and value objects
Complex
creation logic
Client
Factory
Creates
Entity

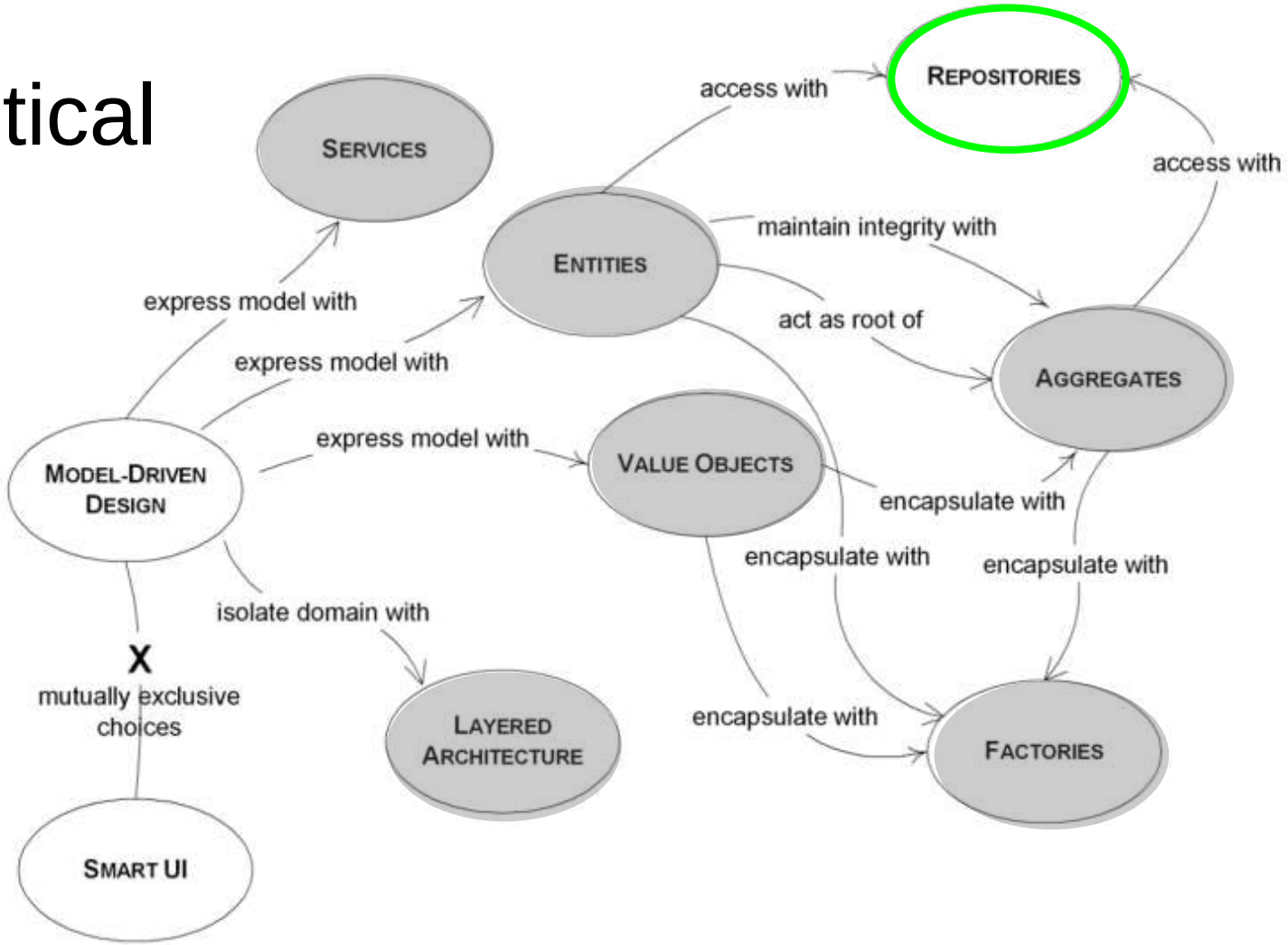
Factory Method

```
public class Sexo
{
    private char sexo;

    private Sexo(char sexo)
    {
        this.sexo = char.ToUpper(sexo);
    }

    public static Sexo Masculino()
    {
        return new Sexo('M');
    }

    public static Sexo Feminino()
    {
        return new Sexo('F');
    }
}
```

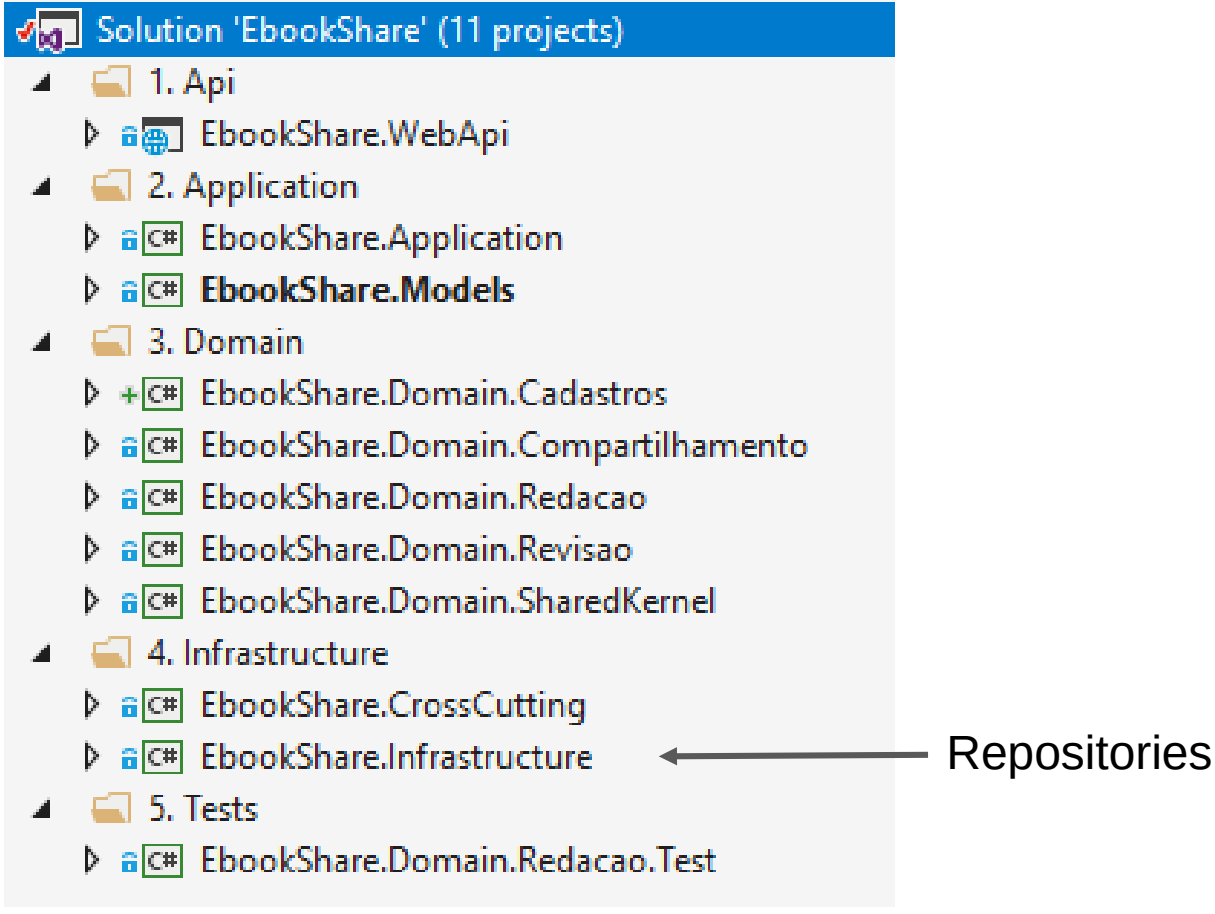
Factory Object

```
public class DocumentFactory
{
    public static Document CreateBibliography(string name, int age)...

    public static Document CreateResume(string name, int age, string profession)...

    public static Document CreateReport(string title)...
}
```

Tactical


REPOSITORIES
access with
SERVICES
access with
ENTITIES
maintain integrity with
express model with
act as root of
express model with
AGGREGATES
express model with
VALUE OBJECTS
MODEL-DRIVEN DESIGN
encapsulate with
encapsulate with
encapsulate with
isolate domain with
X
mutually exclusive choices
LAYERED ARCHITECTURE
encapsulate with
FACTORIES
SMART UI

Repositories



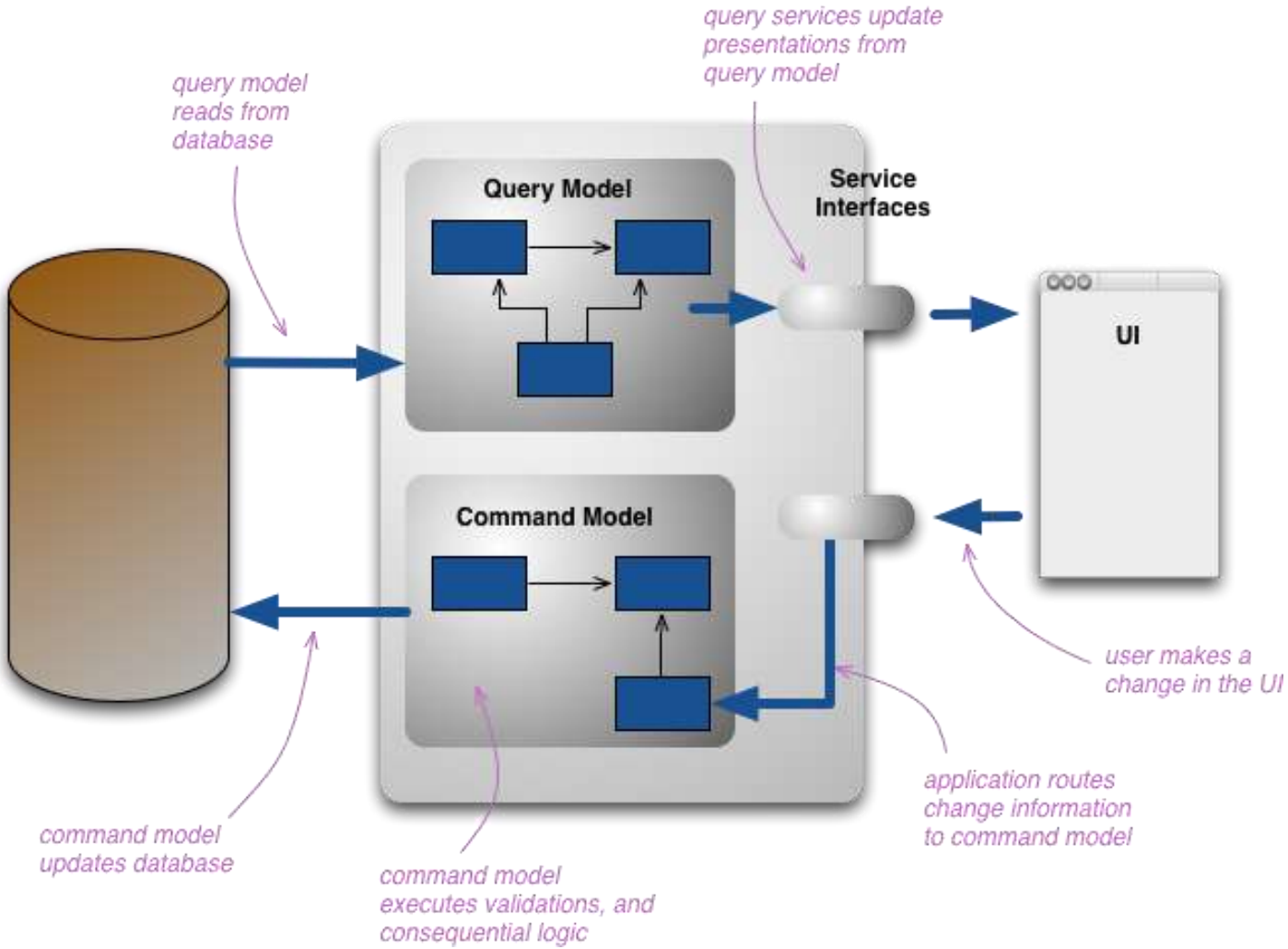
```ruby
module Infra
  module Repositories
    class OrderRepository < Domain::Order::OrderRepository
      def initialize(model = {})
        @order = model.fetch(:order) { Domain::Order::Order }
      end

      def save(order)
        order.save
      end

      def find_by_id(id)
        @order.find_by(id: id)
      end
    end
  end
end
```

Schedule

- ~~Basic Concepts~~
- ~~Ubiquitous Language~~
- ~~Domain Expert~~
- ~~Domain Model~~
- ~~Architectures types~~
  - ~~DDD~~,
  - ~~Smart UI~~,
  - ~~...~~
- ~~**Strategic Design**~~
  - ~~Bounded Context~~
  - ~~Context Maps~~
  - ~~Domain Events~~
  - ~~Event Storming~~
- ~~**Tactical Design** (building blocks~~)
  - ~~Layered Architecture~~
  - ~~Entities~~
  - ~~Value Objects~~
  - ~~Aggregates~~
  - ~~Domain Services~~
  - ~~Factories~~
  - ~~Repositories~~
- CQRS
- SOA
  - Event Driven Architecture
- Event Sourcing
- Final Thoughts

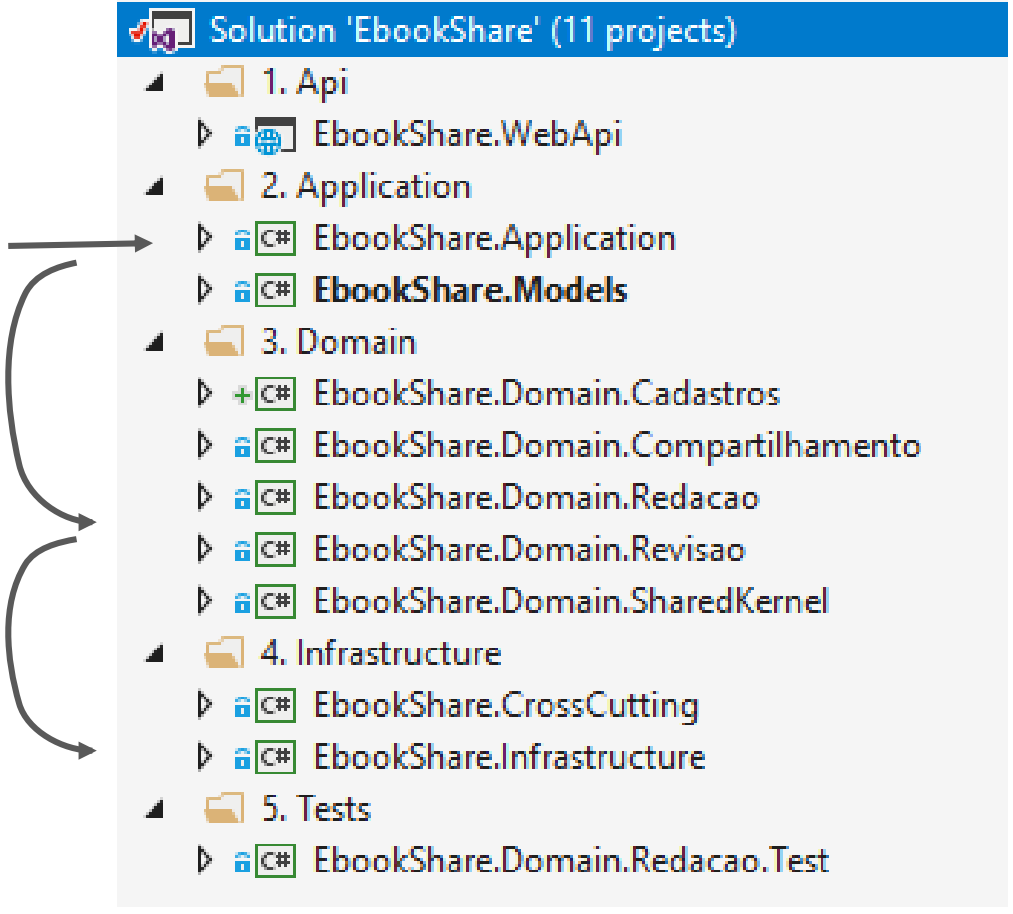
CAMADINHAS

CQRS


query services update presentations from query model
query model reads from database
Query Model
Service Interfaces
UI
Command Model
user makes a change in the UI
application routes change information to command model
command model updates database
command model executes validations, and consequential logic

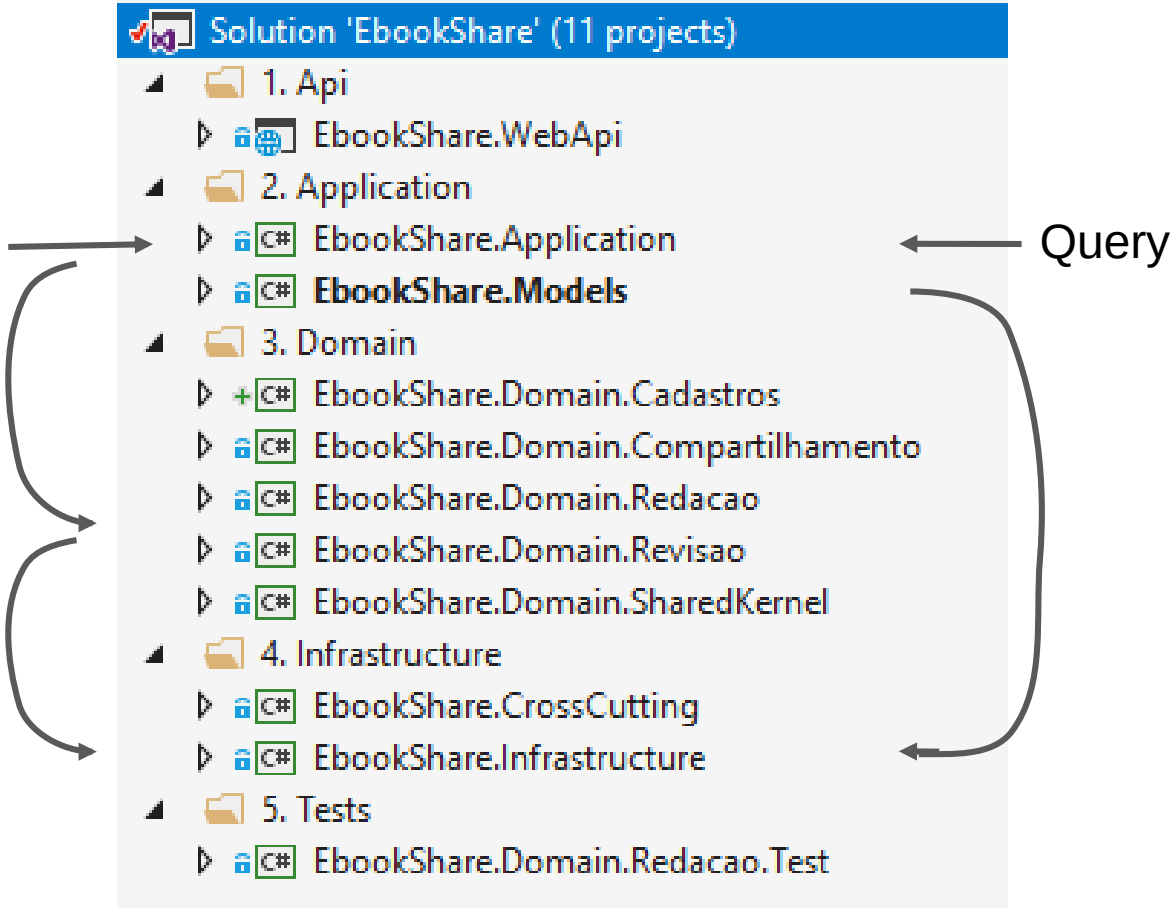
CQRS

Command



CQRS

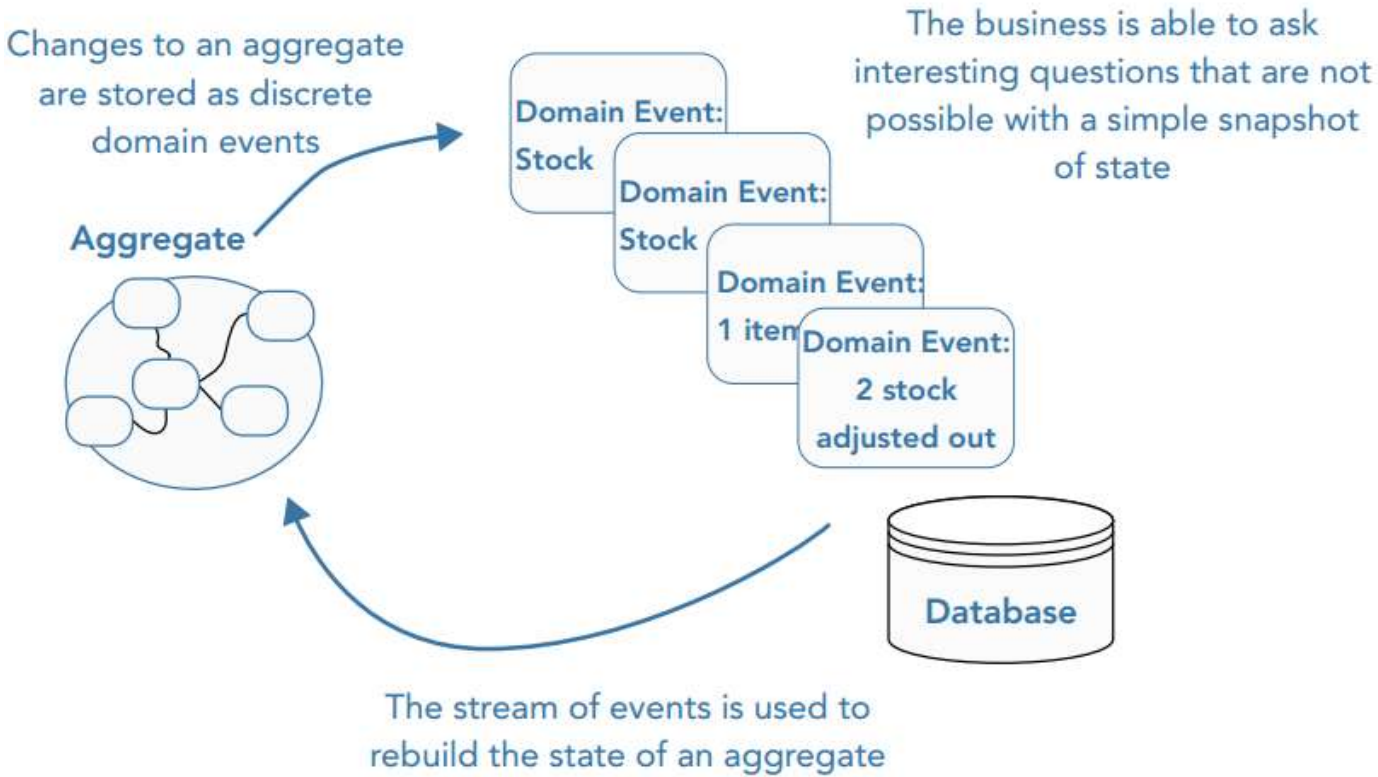
Command



Query

Integrating
between
external contexts

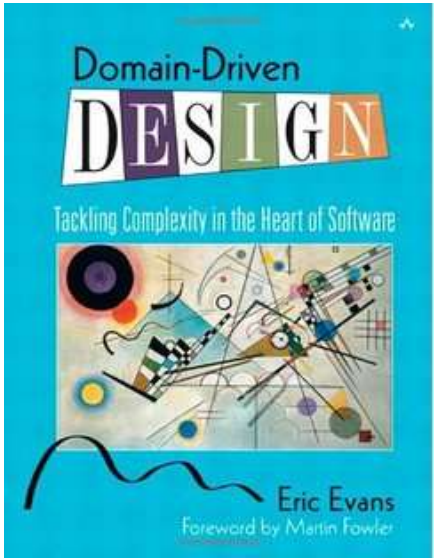
EVENT SOURCING

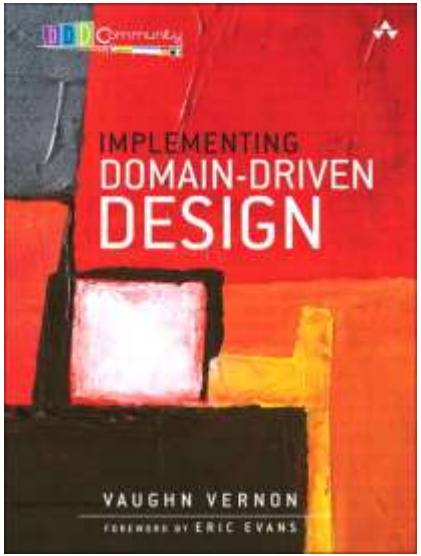


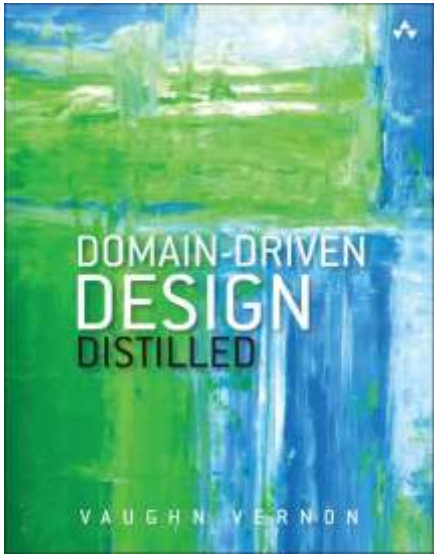
Schedule

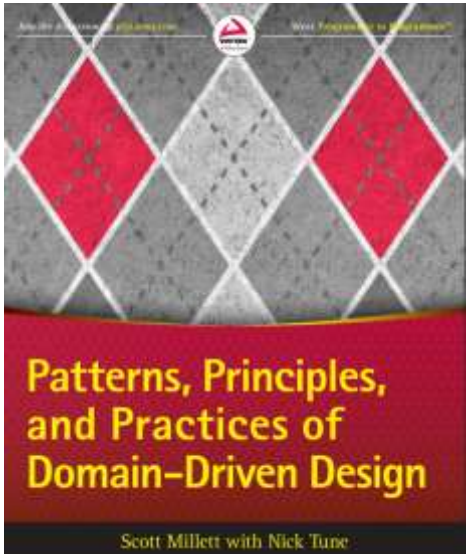
- ~~Basic Concepts~~
- ~~Ubiquitous Language~~
- ~~Domain Expert~~
- ~~Domain Model~~
- ~~Architectures types~~
  - ~~DDD~~,
  - ~~Smart UI~~,
  - ~~...~~
- ~~**Strategic Design**~~
  - ~~Bounded Context~~
  - ~~Context Maps~~
  - ~~Domain Events~~
  - ~~Event Storming~~
- ~~**Tactical Design** (building blocks~~)
  - ~~Layered Architecture~~
  - ~~Entities~~
  - ~~Value Objects~~
  - ~~Aggregates~~
  - ~~Domain Services~~
  - ~~Factories~~
  - ~~Repositories~~
- ~~CQRS~~
- ~~SOA~~
  - ~~Event Driven Architecture~~
- ~~Event Sourcing~~
- ~~Final Thoughts~~

Referências









https://github.com/VaughnVernon/IDDD_Samples_NET
https://github.com/heynickc/awesome-ddd

Obrigado!

fabriciorissetto@gmail.com

fabriciorissetto