DOMAIN DRIVEN DESIGN 2

Principles of Domain Driven Design

- Use one common language to describe the concepts of a domain
    - Ubiquitous language
- Create a domain model that shows the important concepts of the domain
    - Rich domain model
- Let the software be a reflection of the real world domain
- Create small contexts in which a domain model is valid
    - Bounded context

# Principles of Domain Driven Design

- Use one common language to describe the concepts of a domain
    - Ubiquitous language
- Create a domain model that shows the important concepts of the domain
    - Rich domain model
- Let the software be a reflection of the real world domain
- Create small contexts in which a domain model is valid
    - Bounded context

Common language

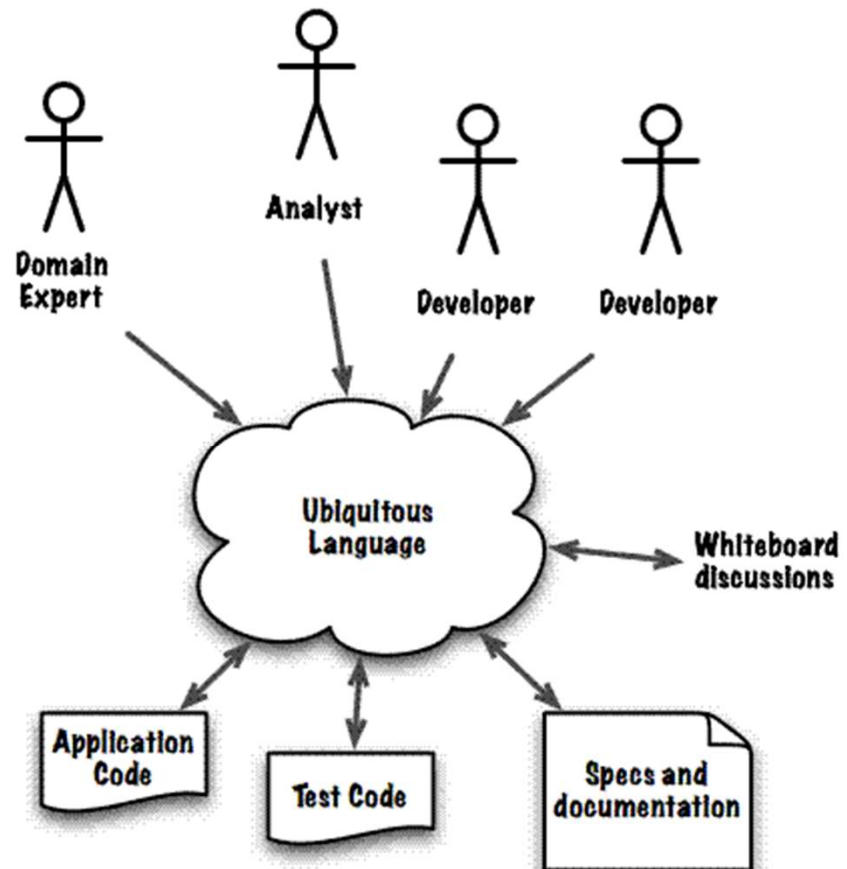- Different people from the business use different names for the same thing.
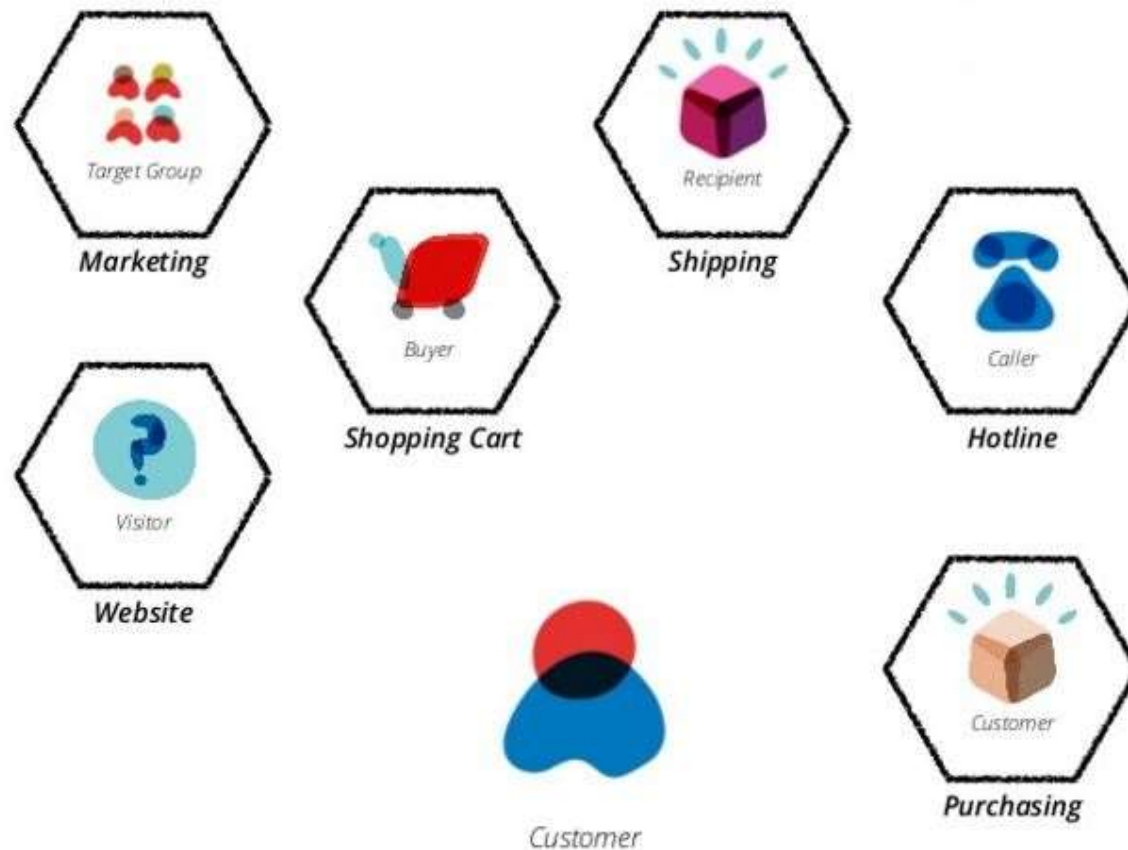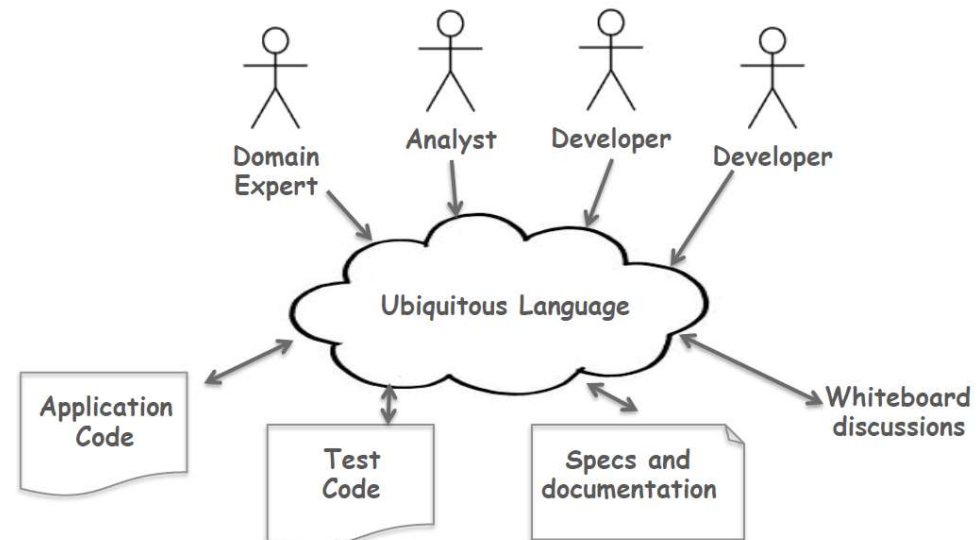

We need a common language

Ubiquitous Language

- Language used by the team to capture the concepts and terms of a specific core business domain.
    - Used by the people
    - Used in the code
    - Used everywhere

Ubiquitous Language

- Based on the domain model
- Takes time to create
- Can change in time
    - New names are discovered
- The whole team should use the common language

Characteristics of the Ubiquitous Language

- Not fixed

- Changes all the time

- Takes discipline to use, maintain and evolve

  - But improves communication and learning

knowledge crunching is to understand the domain

in devops the idea is the same team builds and mantain the project

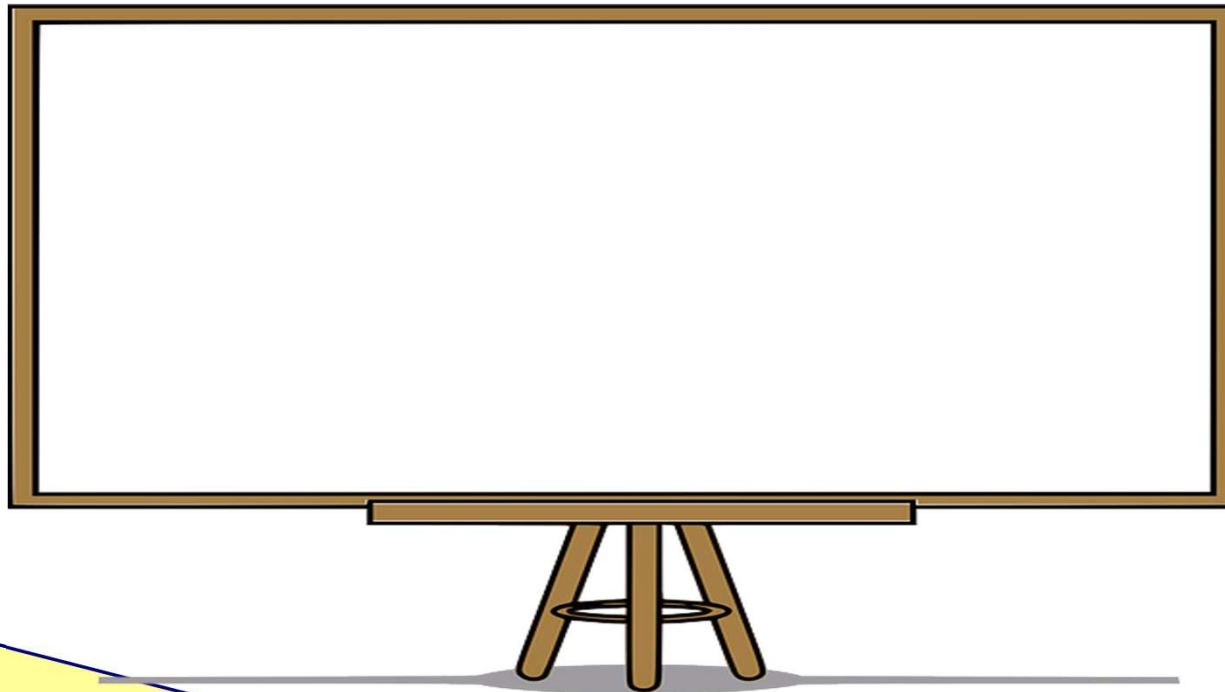# EXAMPLE OF MODEL AND UBIQUITOUS LANGUAGE

Flight control system

aircraft controller

developer

We want to monitor air traffic. Where do we start?

Let's start with the basics. All this traffic is made up of **planes**. Each plane takes off from a **departure** place, and lands at a **destination** place.
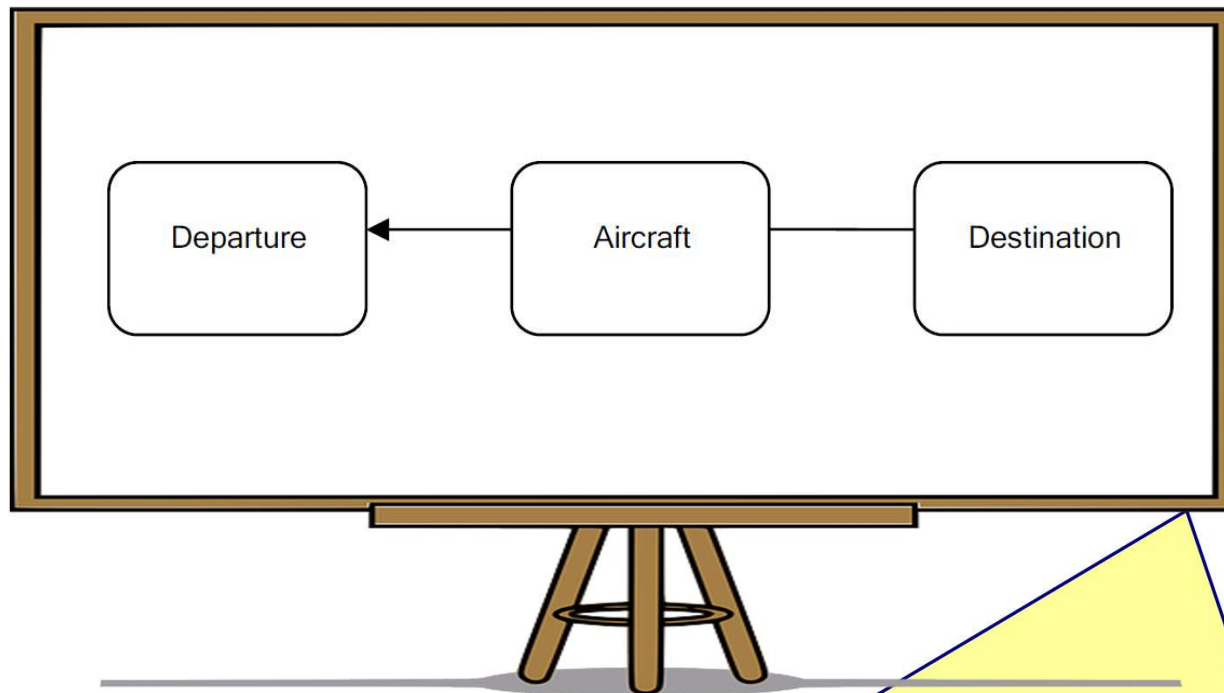
Flight control system

Flight control system

Flight control system

Flight control system

Flight control system

Before leaving the airport, the pilots receive a detailed **flight plan** which includes all sorts of information about the **flight**: the **route**, cruise **altitude**, the cruise **speed**, the type of **airplane**, even information about the crew members.

aircraft controller

Aircraft → Flight Plan → Route

Route → Fix → 2DPoint
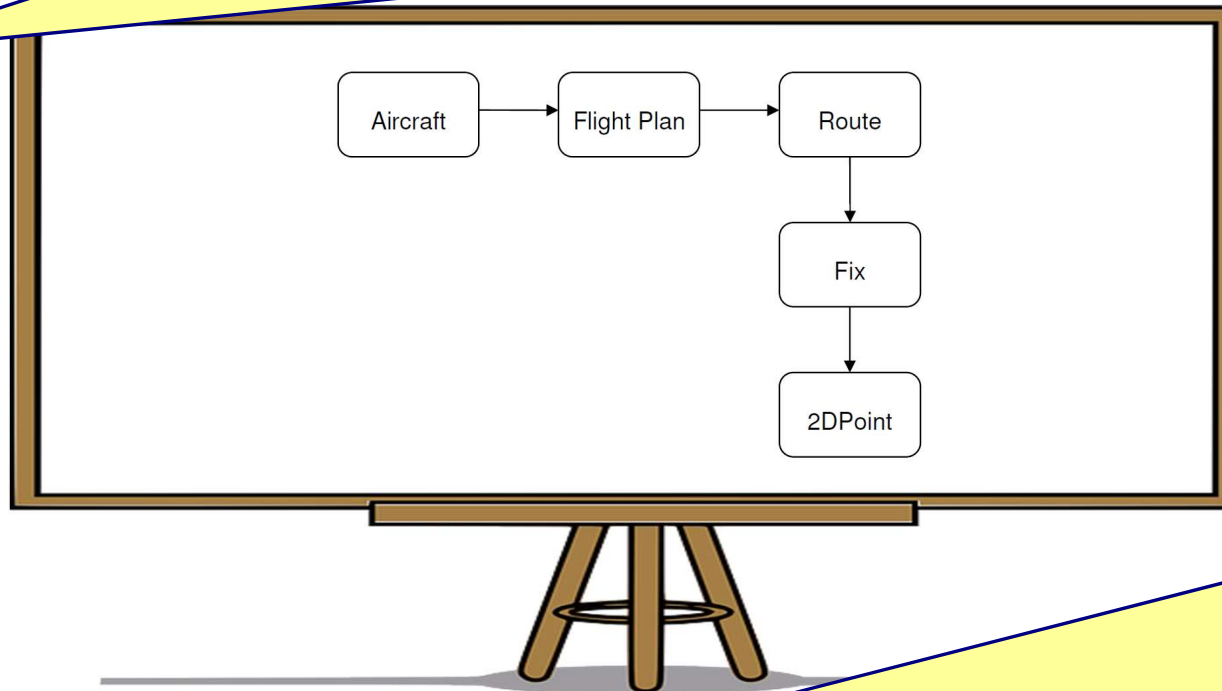
developer

Now that I'm looking at it, I realize something. When we are monitoring air traffic, we are not actually interested in the planes themselves, if they are white or blue, or if they are Boeing or Airbus. We are interested in their **flight**. That's what we are actually tracking and measuring. I think we should change the model a bit in order to be more accurate.

Flight control system

aircraft controller

developer

Flight → Flight Plan → Route → Fix → 2DPoint
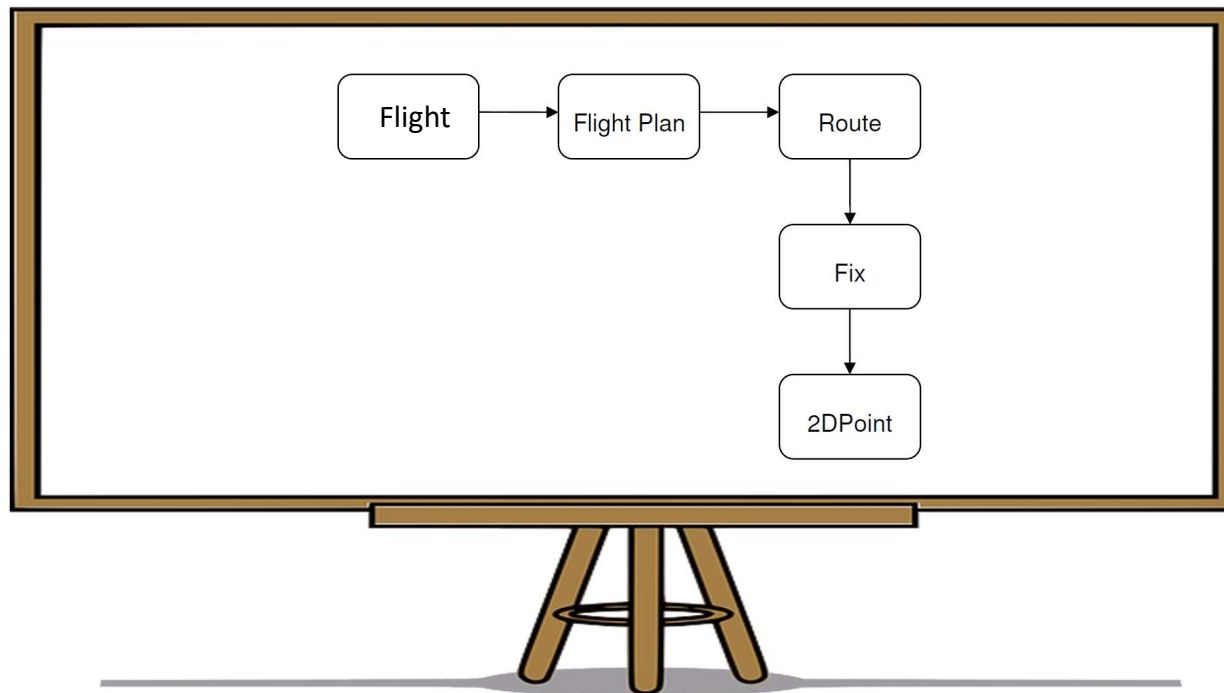
Effective knowledge crunching

- Conversations
  - Start with the areas of the problem domain that keep the business up at night, the areas that will make a difference to the business.
    - Which parts of the current system are hard to use?
    - Which manual processes stop them from doing more creative, value-adding work?
    - What changes would increase revenue or improve operational efficiencies and save money?
- Start from the requirements (use cases, user stories)
- Ask powerful questions
  - Where does the need of this system come from?
  - How will this system give value to the business?
  - What would happen if this system wasn't built?
  - What is the success criteria of this product?
- Sketching
  - Whiteboard or paper.
  - Quick and informal
  - Keep your diagrams at a consistent level of detail.
  - Use multiple diagrams

Principles of Domain Driven Design

- Use one common language to describe the concepts of a domain
    - Ubiquitous language
- Create a domain model that shows the important concepts of the domain
    - Rich domain model
- Let the software be a reflection of the real world domain
- Create small contexts in which a domain model is valid
    - Bounded context

Model

Cottonwoods -Maple House Model

Ground Floor   Second Floor

- **More complexity -> More modeling**
  - Higher level of abstraction
  - Allows for visualization
  - Vehicle of communication

Domain model

- Extracts domain essential elements
    - Relevant to a specific use
- Layers of abstractions representing selected aspects of the domain
- Contains concepts of importance and their relationships

The Problem Space

A Projection of the Real Domain

Domain

The reality

The Solution Space

Domain Model

An abstraction of reality designed to manage complexity for specific business cases
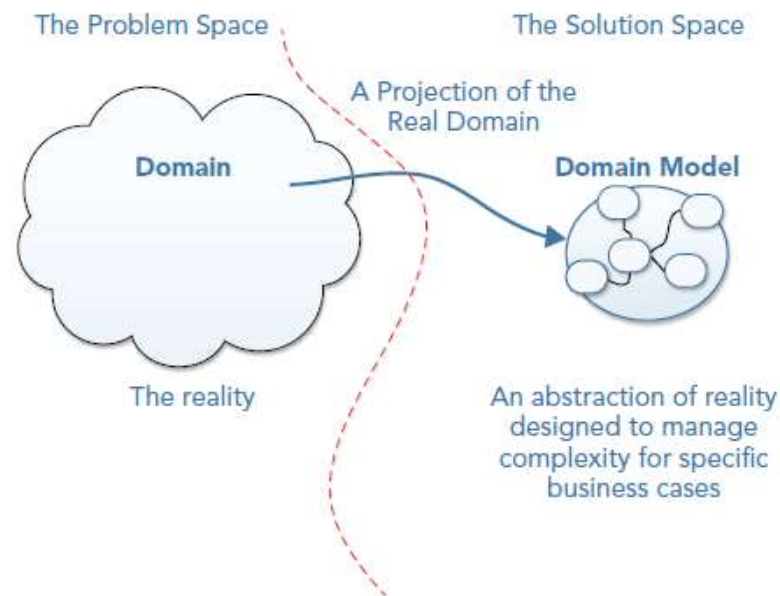
Domain model

- Simplification of reality
- Area of interest

Structure of the domain model

- A domain model is not a particular diagram

- Use the format that communicates the best
  - Diagram
  - Text
  - Code
  - Table
  - Formula
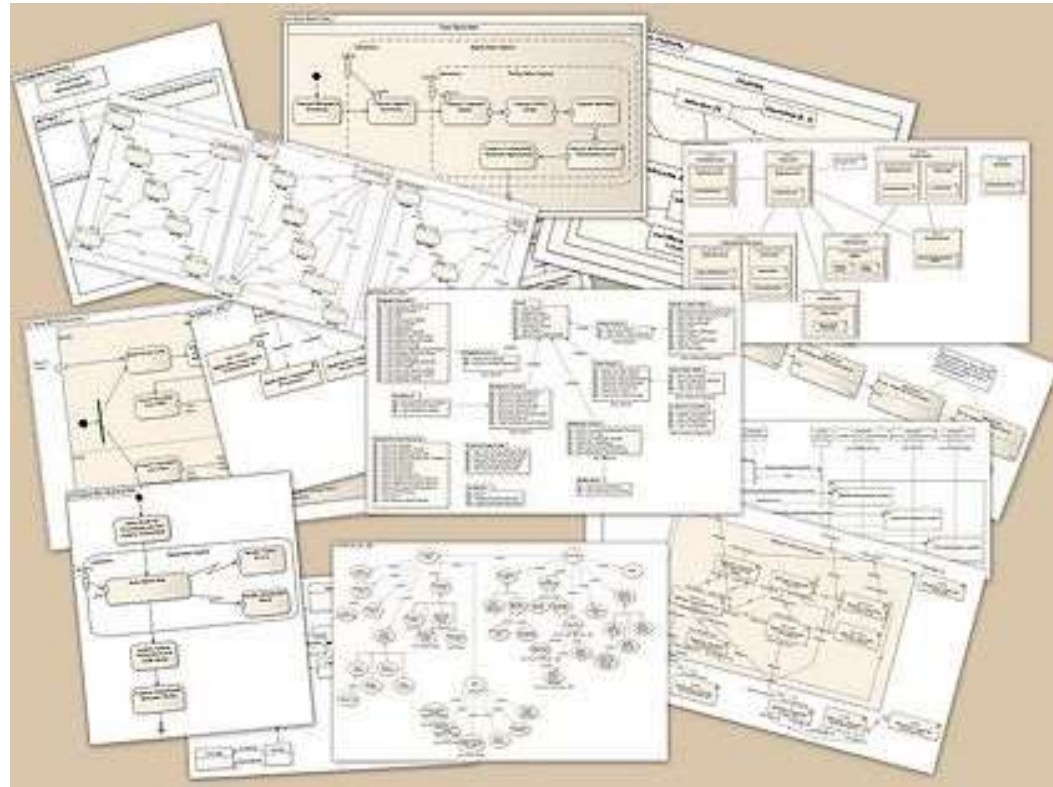
Effectiveness of communication

- Detailed documentation
- Email
- Videoconferencing
- Teleconference calls
- Overview documentation
- Online chat
- Overview diagrams
- F2F at whiteboard
- Face to face (F2F)
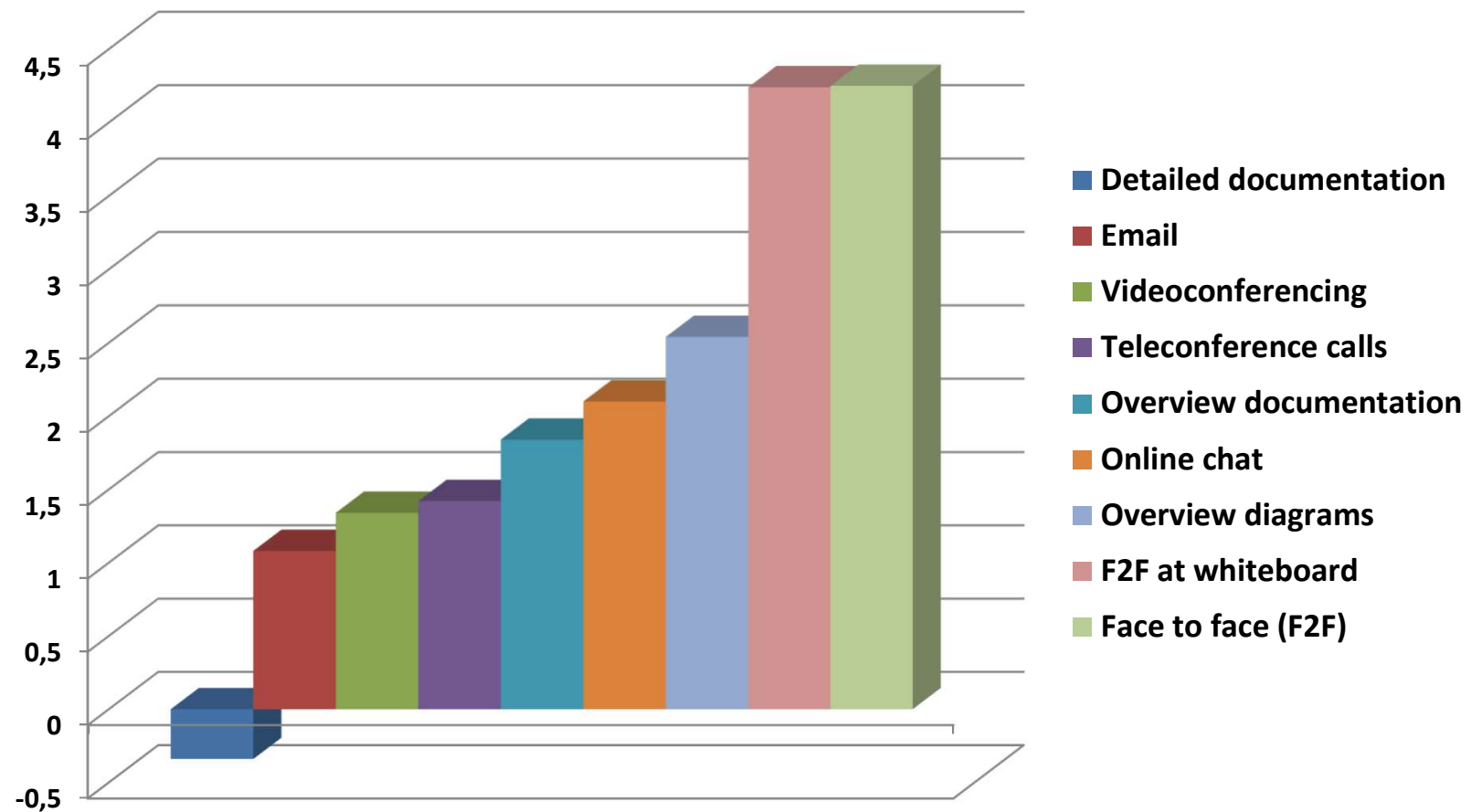
Effective communication

TOP 5 PETITION CATEGORIES

11,891
9,956
7,197
6,503
6,297

SIGNATURES

150,945
MOST SIGNATURES
ON A SINGLE PETITION

406
AVERAGE SIGNATURE
PER HOUR

15.2%

People remember 10% of what they hear, 20% of what they read,
but 80% of what they see.**

1,000,000
500,000
0
SEPTEMBER 2011   FEBRUARY 2012   JULY 201

Model and diagrams
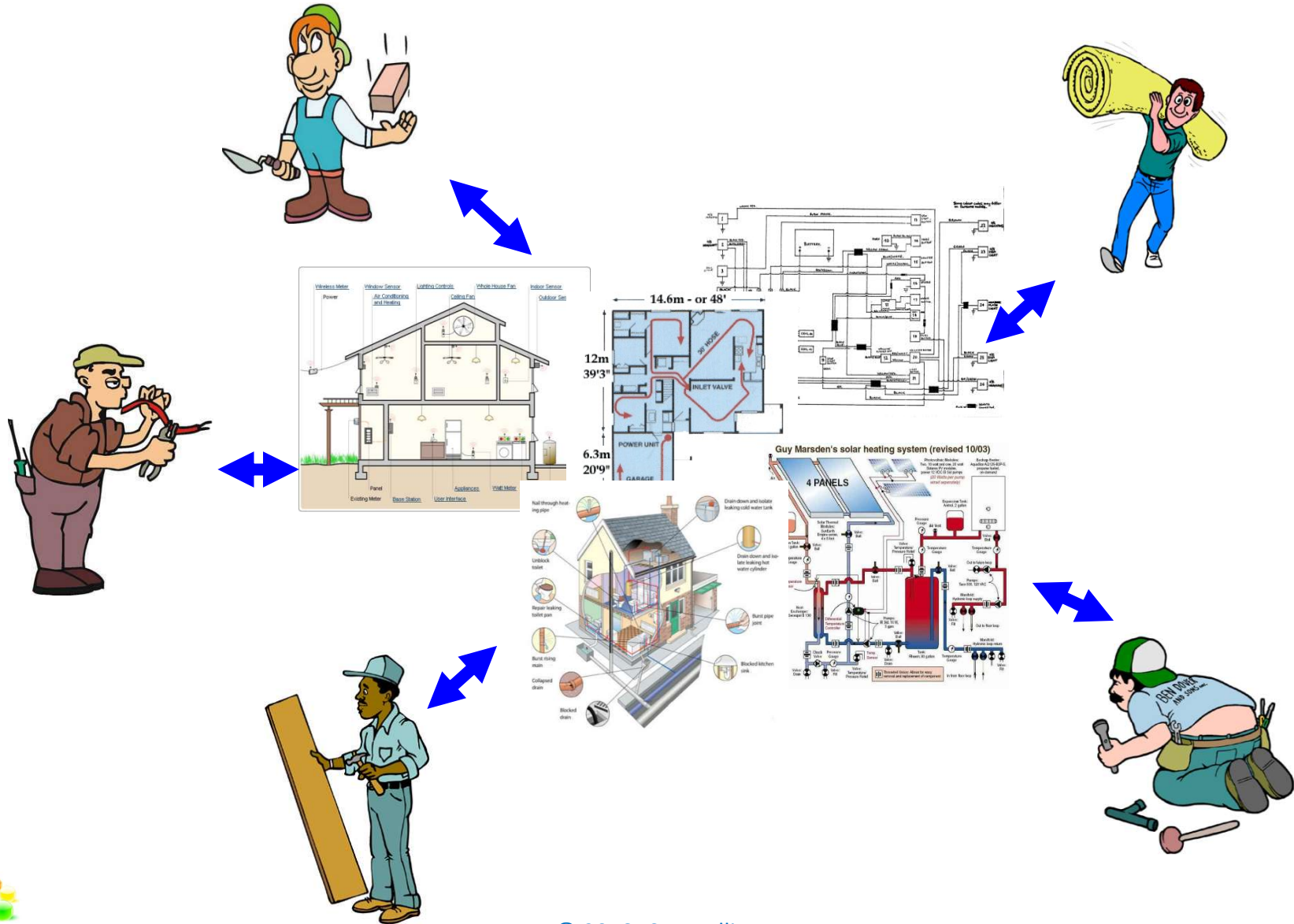
Advantages of a domain model

- Improves understanding

- Validates understanding

- Improves communication   no documentation

- Shared glossary

- Improves discovery

Effective modeling

- Cultivate a language based on the model
- Use the format that fits the best
- Model light
  - Simple sketches
  - Focus on communication, not completeness
- Update the model
  - Add new concepts, but also remove concepts that are not useful or central anymore
- Use the model for brainstorming and experimenting sessions
- Bind the model and the implementation

Isolate the domain model

encapsulate

Client layer

Application layer

The domain model is isolated in its own layer

Domain layer

Infrastructure layer
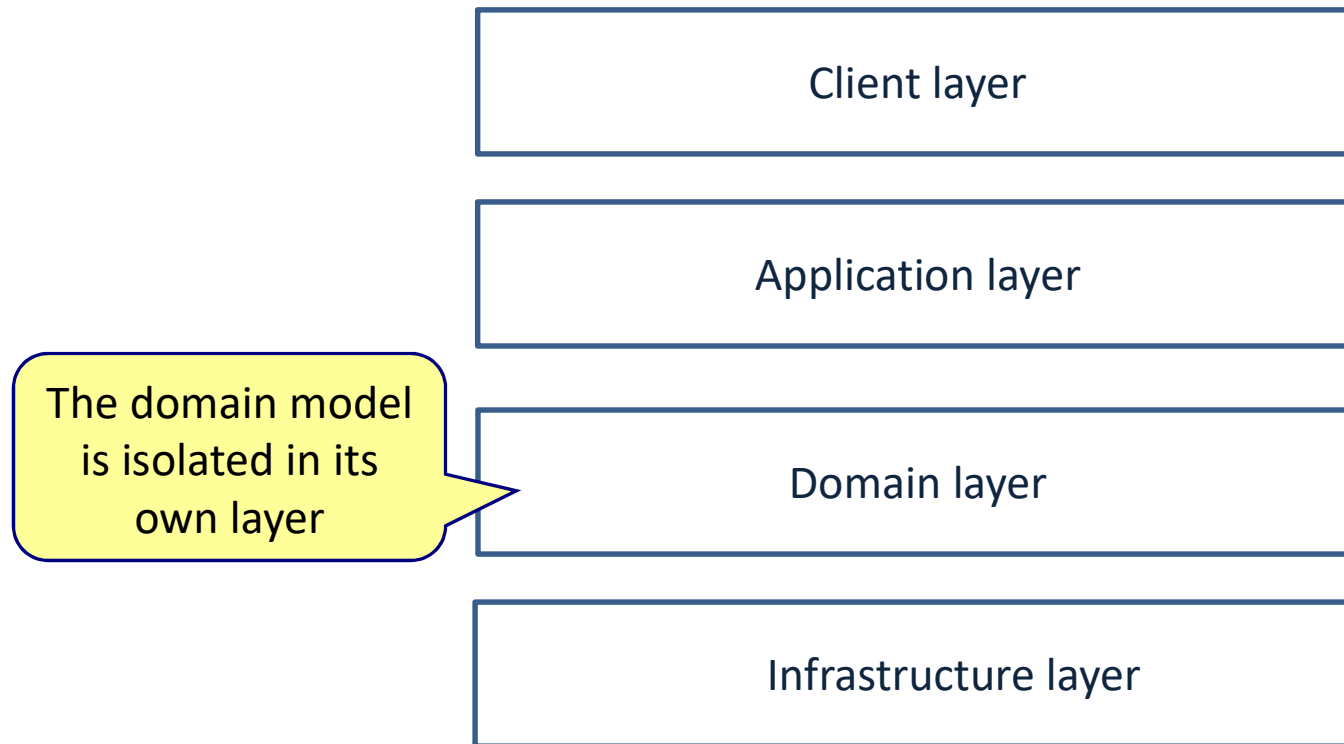
Keep the model and code in sync.

Code Model

Domain Experts and
the Development Team

Developers and
business experts
knowledge crunch

Code and Analysis model
are in synergy

A hidden insight is
revealed in the analysis
model and is updated in
the code model

UML

Analysis Model

Domain experts and
business analysts
understand the code model
and help to shape it

Constant feedback and
collaboration helps to
shape both models

The model should reflect the business

- If domain experts don't understand the model, something is wrong with the model.
  diagrams

  if developer explains the code, the domain expert should understand

Knowledge crunching
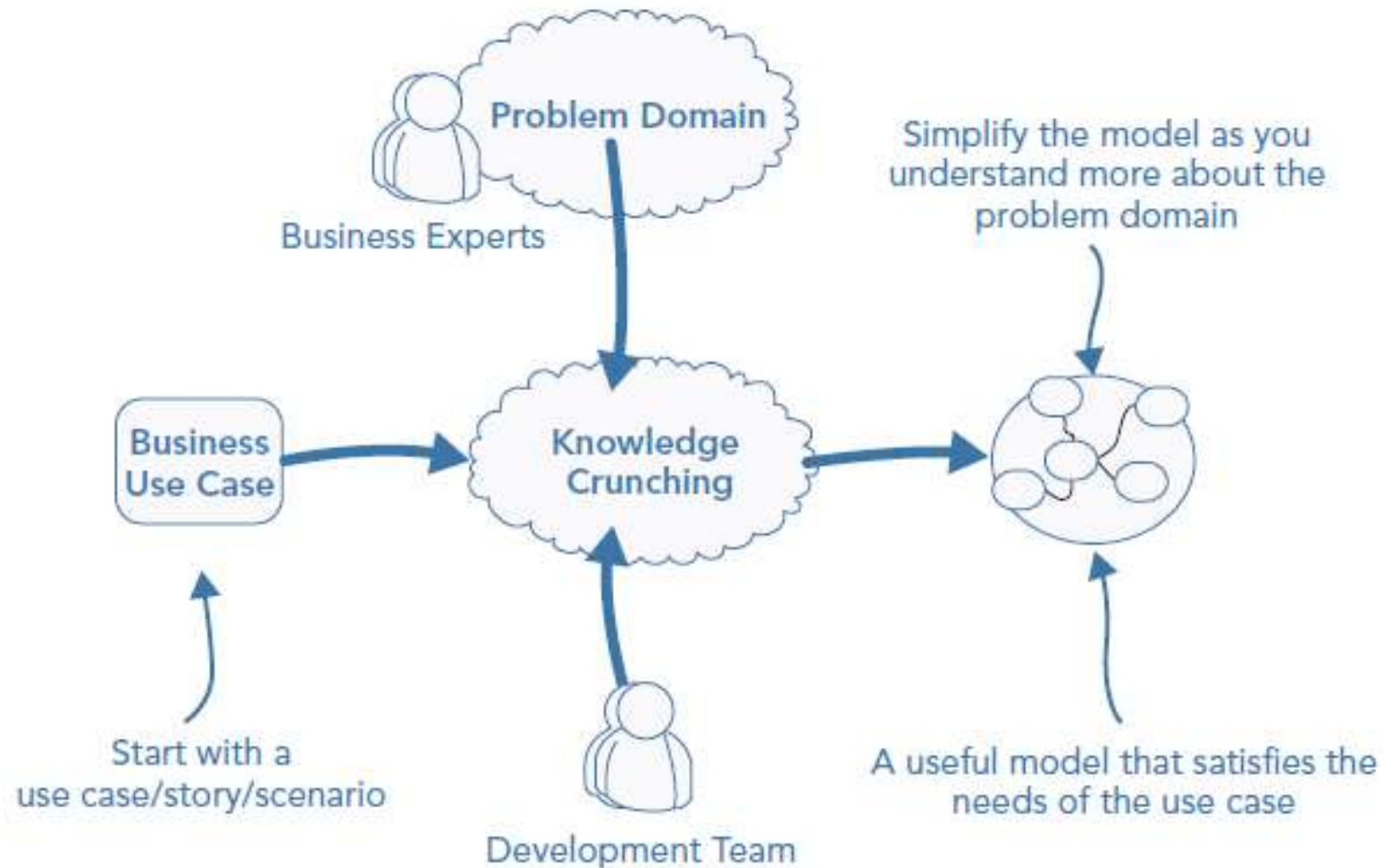
Principles of Domain Driven Design

- Use one common language to describe the concepts of a domain
  - Ubiquitous language
- Create a domain model that shows the important concepts of the domain
  - Rich domain model
- Let the software be a reflection of the real world domain
- Create small contexts in which a domain model is valid
  - Bounded context

Big ball of mud

Initial software
incarnation fast to
produce

Over time without care
and consideration,
software turns to ball
of mud

The software is a reflection of the real world

- It is easier to spot inconsistencies, errors, misconceptions.
- The software is easier to understand for
  - Existing developers
  - Testers
  - Business people (with guidance)
  - New developers and testers
- By looking at the code you can learn a lot of domain knowledge
- No translation necessary
- It is easier to write tests
- Easier to maintain the code
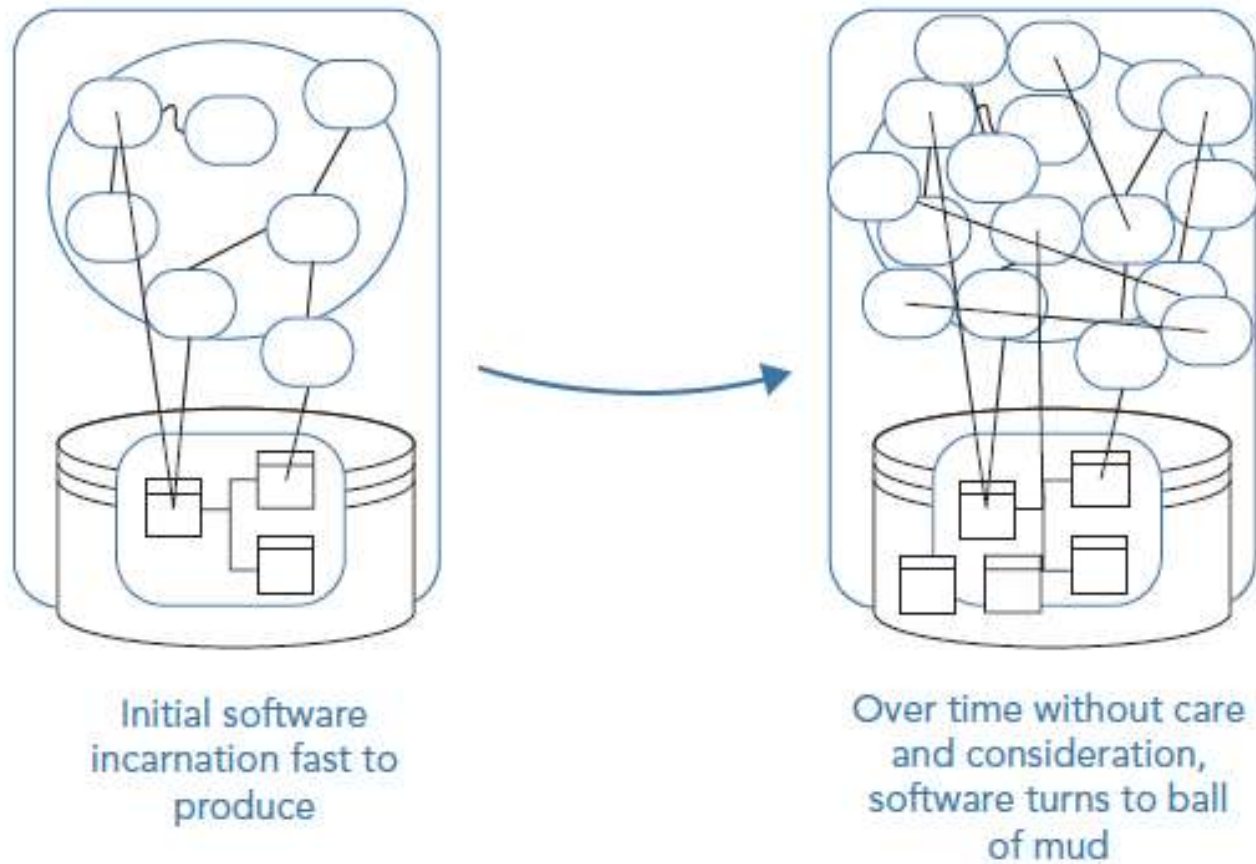
Principles of Domain Driven Design

- Use one common language to describe the concepts of a domain
    - Ubiquitous language
- Create a domain model that shows the important concepts of the domain
    - Rich domain model
- Let the software be a reflection of the real world domain
- Create small contexts in which a domain model is valid
    - Bounded context

Shared objects between subdomains

E-commerce Application

Application Logic

Promotion Subdomain | Allocation Subdomain | Loyalty Subdomain | Shipping Subdomain

Product — Business Objects/Logic
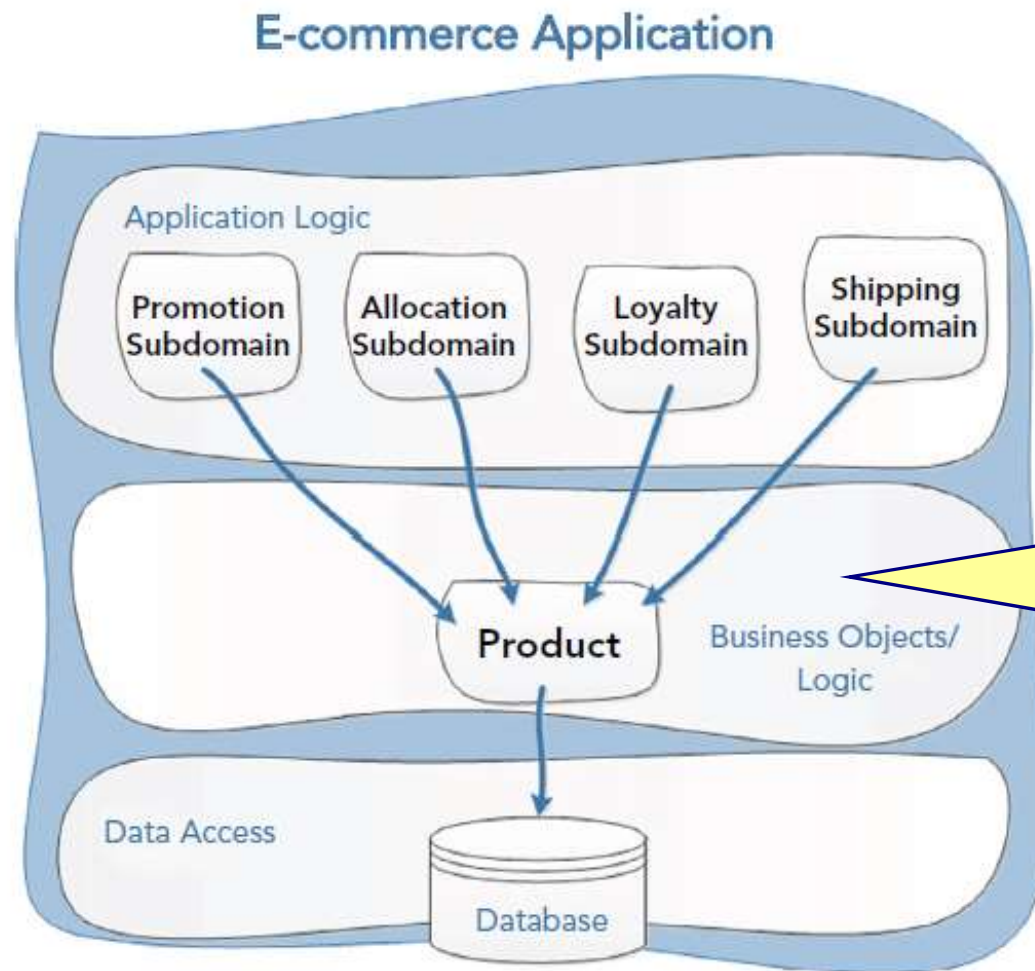
Single view of an entity for all subdomains can quickly become a problem

Data Access

Database

Big ball of mud

should have single responsability, it should have only one reason to change

Inventory Model

Fulfillment Model

Pricing Model

Public Class Product
{
    public void Allocate () { . . . }
    public Locations Stock () { . . . }
    public Recommendations Similar () { . . . }
    public Price PriceFor (CustomerType . .)
    Public PurchaseOrder BuyFrom (Supplier . .)
    Public bool CanShipTo (Country . . .)
    . . .
}

Personalization Model

Procurement Model

Shipping Model

A change in one subdomain can effect all other subdomains

Context

- A specific domain term may have a different definition in a different context
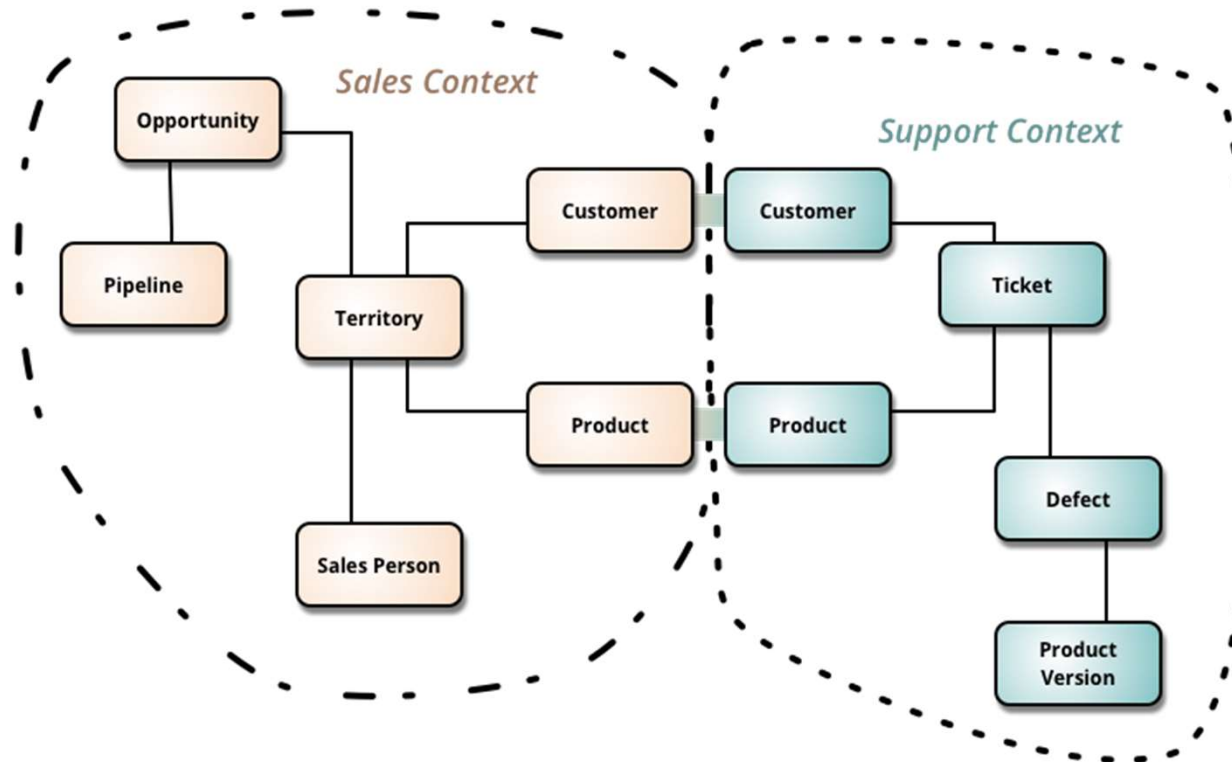
Bounded context

it is a context with its own language, with its own rules,

Bank context

Bankaccount context

Account
+accountNumber
+balance
+deposit()
+withdraw()
+getBalance()
+addInterest()

*

AccountEntry
+date
+amount
+description

**Online banking
web application context**

Account
+accountId
+userId
+password
+checkCredentials()

1

User
+name
+email
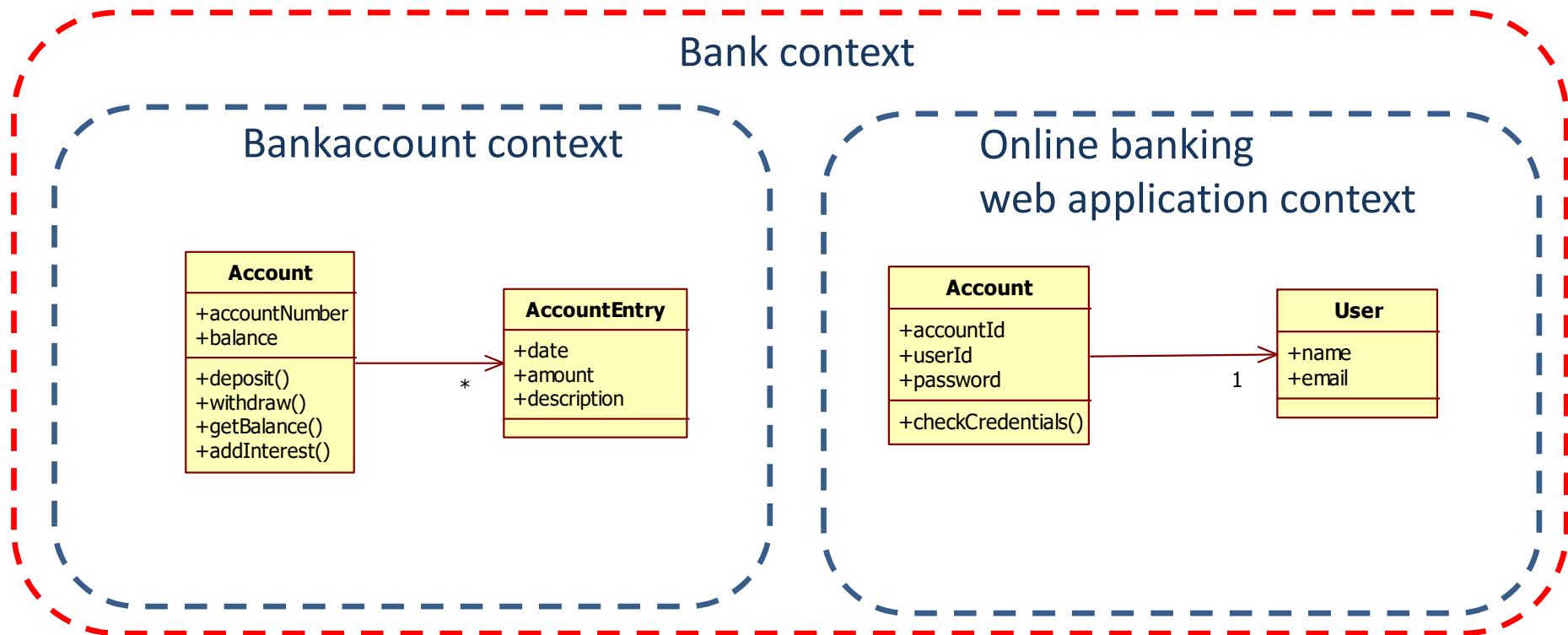
Bounded context

it is a context with its own language, with its own rules, it is also called subdomain or sub systems

- Create explicit boundaries in terms of
  - Team organization
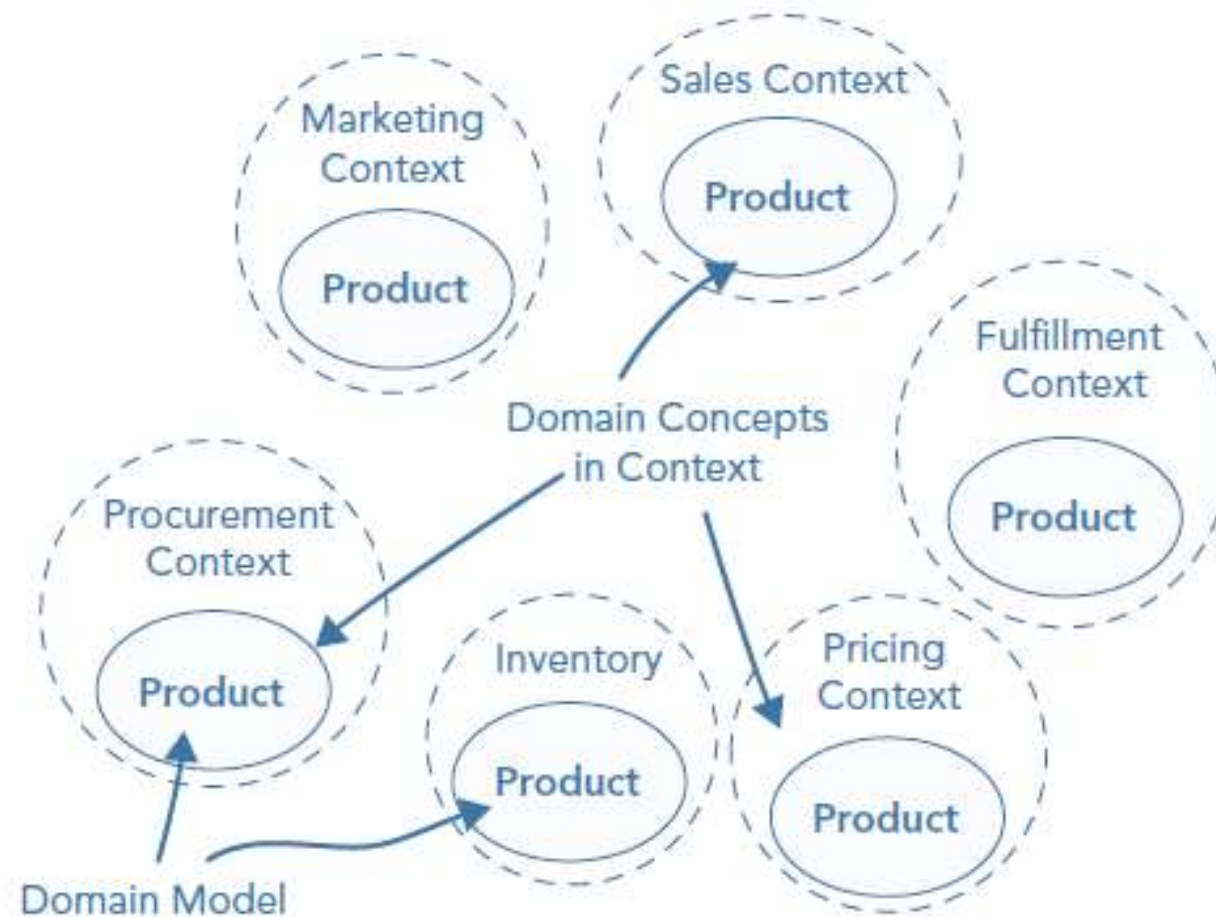 - Usage of the system
  - Physical manifestation (code, database)
- Create a different domain model per bounded context
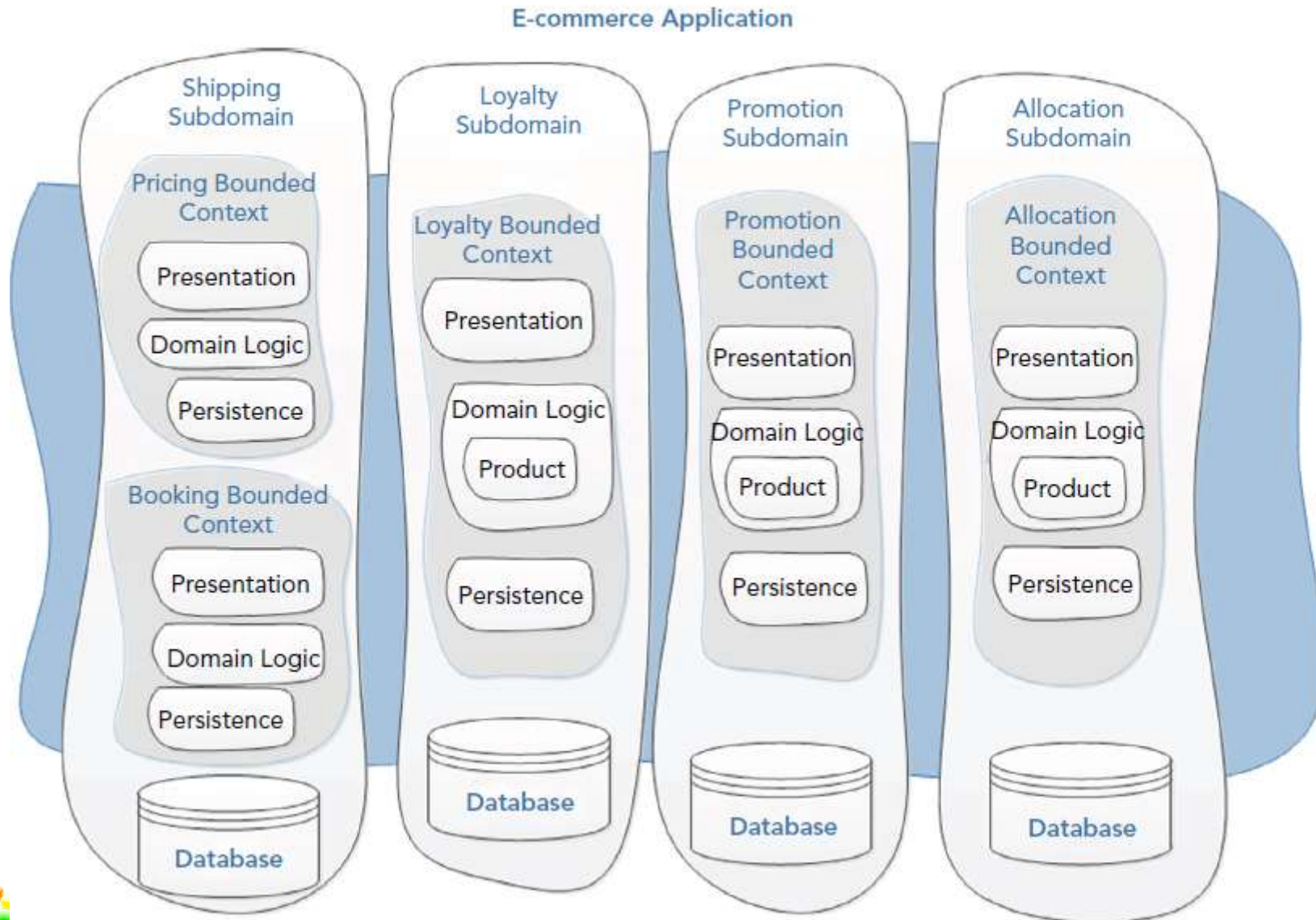 - A model is only valid within the scope of the bounded context

Each bounded context has its own model

Bounded context example

reuse becomes more important if that what you want to reuse is big

E-commerce Application

Shipping Subdomain

Pricing Bounded Context
- Presentation
- Domain Logic
- Persistence

Booking Bounded Context
- Presentation
- Domain Logic
- Persistence

Database

Loyalty Subdomain

Loyalty Bounded Context
- Presentation
- Domain Logic
  - Product
- Persistence

Database

Promotion Subdomain

Promotion Bounded Context
- Presentation
- Domain Logic
  - Product
- Persistence

Database

Allocation Subdomain

Allocation Bounded Context
- Presentation
- Domain Logic
  - Product
- Persistence

Database

Bounded context example

Domains

- **Core subdomain**
    - This is the reason you are writing the software.

- **Supporting subdomain**
    - Supports the core domain

- **Generic subdomain**
    - Very generic functionality
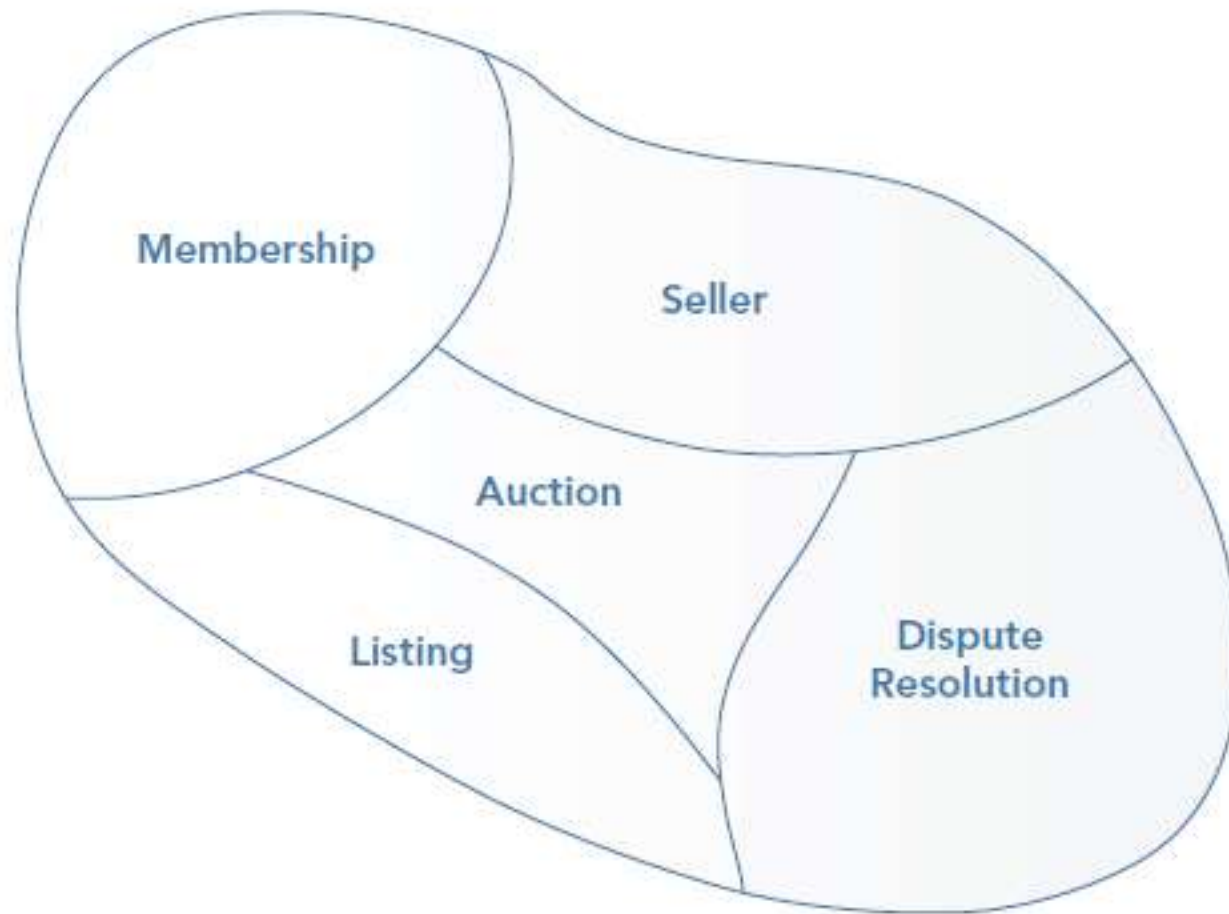   - Email sending service
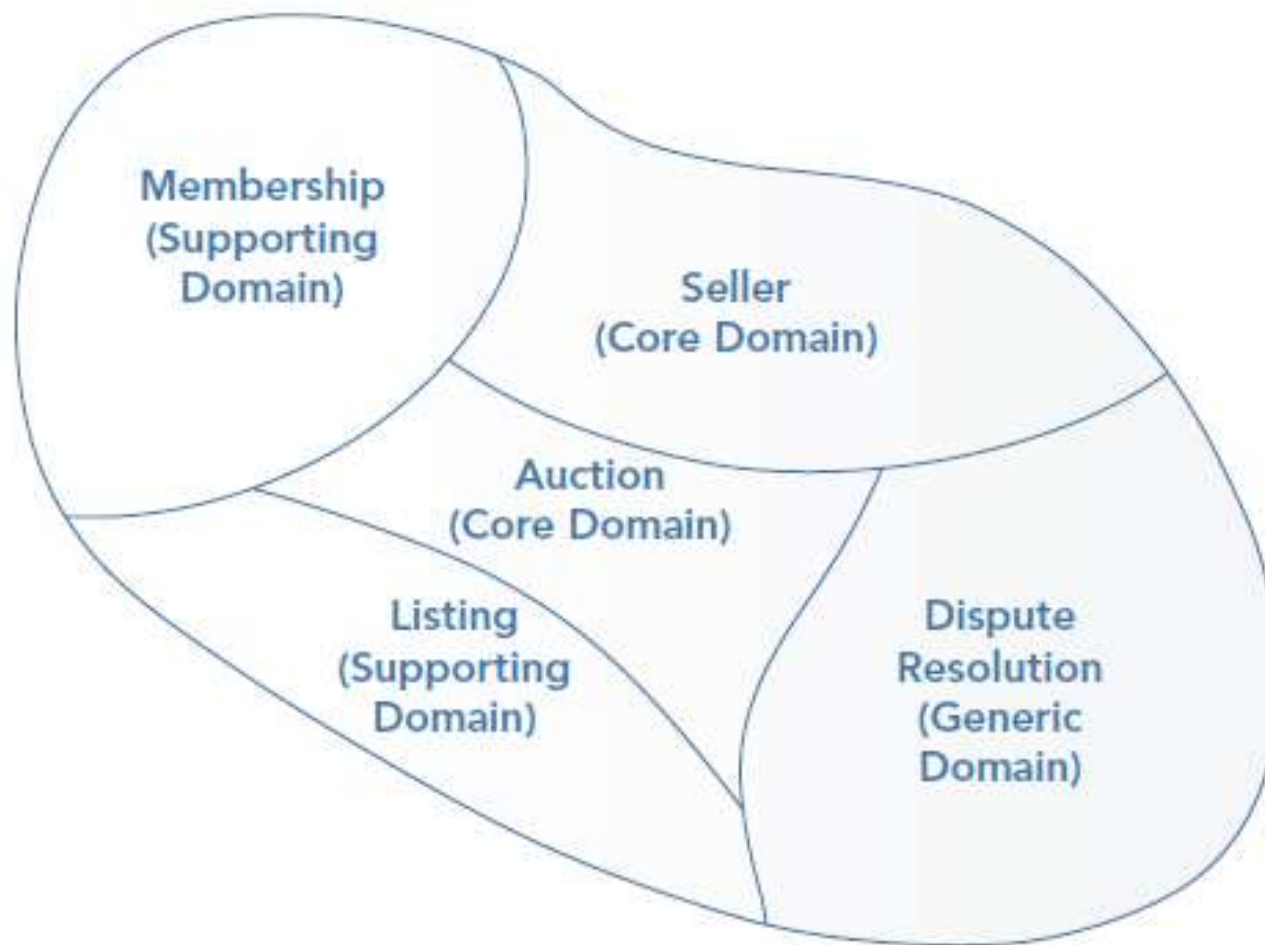   - Creating reports service

Distilling the domain

- The large domain of online auction

Online Auction

Find the subdomains

Identify the core domain

Subdomains shape the solution

Web Based Bespoke Application

Best developers focus attention here

Membership

Model

Seller

Model

Auction

Model

Listing

Model

Dispute Resolution fulfilled by an off of the shelf commercial application

Models within the supporting domains need to be good enough
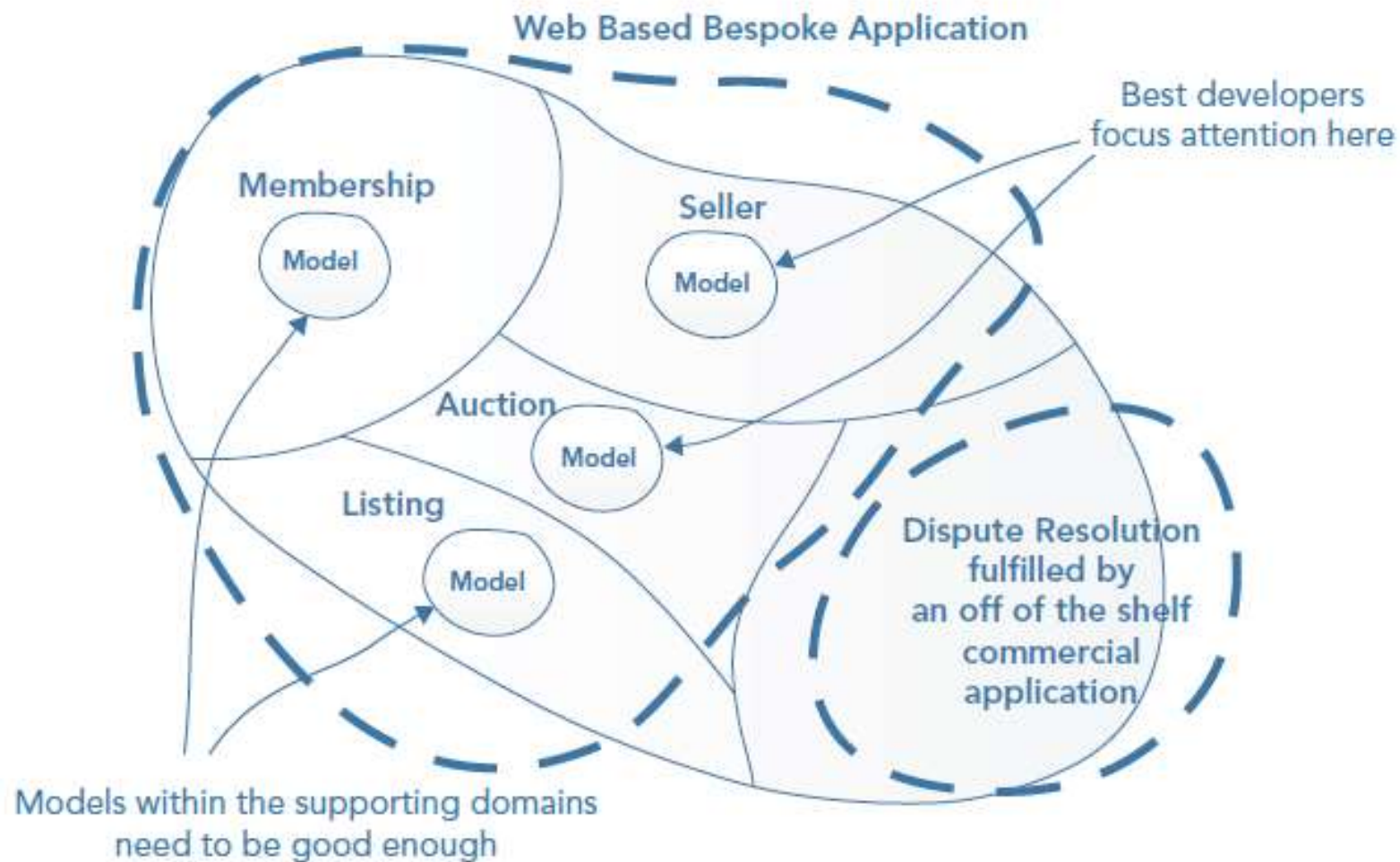
Why DDD?

the main reason for DDD is dealing with complex systems

- Writing code is easy.

- Learning an new technology is fairly easy

- The hardest part of software development is to understand what the business wants and how the business works

  - Focus on those areas of the application that deliver the most value to the business

  - This greatly helps to

    - Understand the application

    - To evolve the application

    - To test the application

Why DDD?

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."

- Martin Fowler

"The critical complexity of most software projects is in understanding the domain itself."
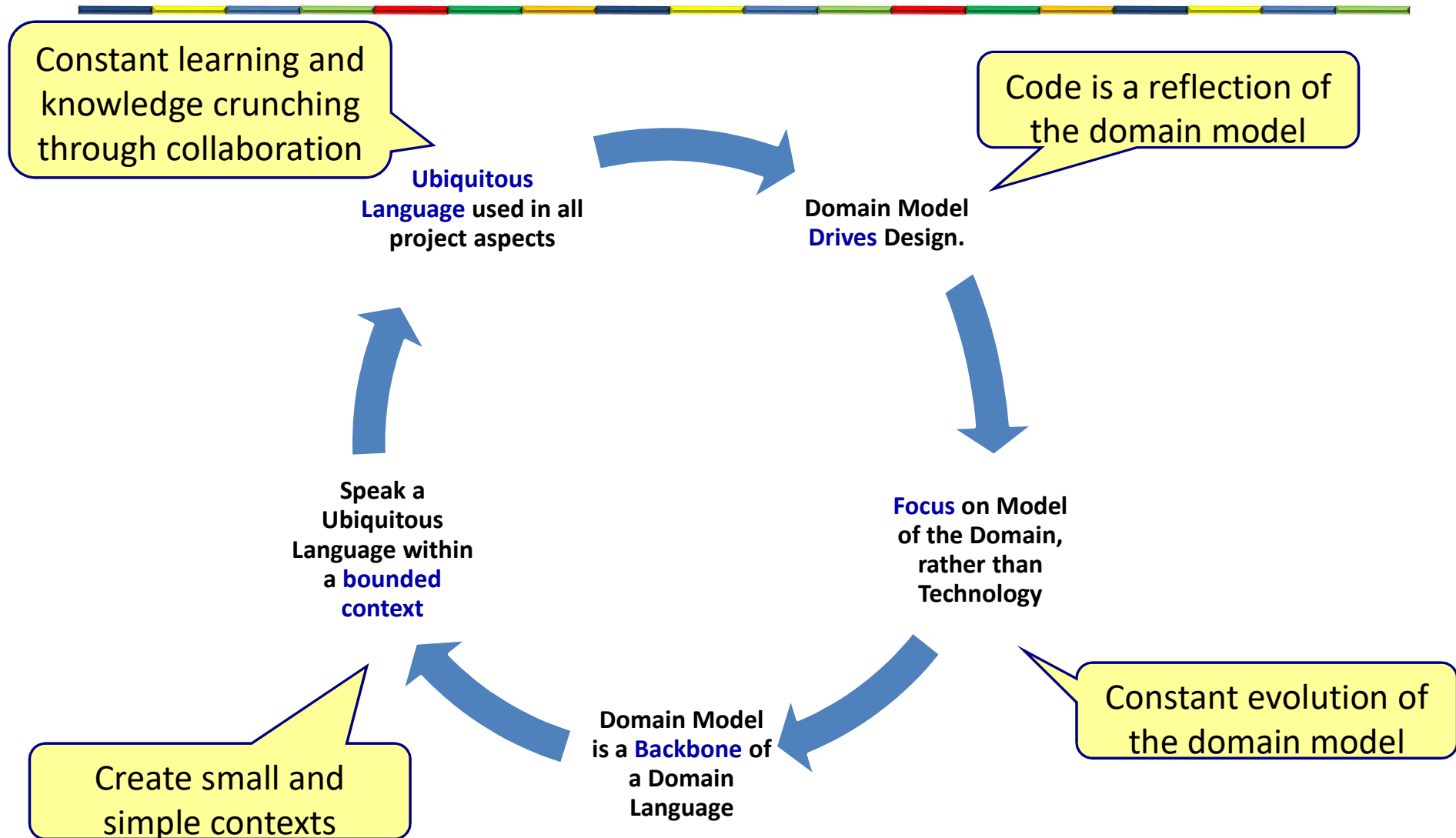
- Eric Evans

"One way or another, creating distinctive software comes back to a stable team accumulating specialized knowledge and crunching it into a rich model. No shortcuts. No magic bullets.

- Eric Evans

DDD summary

Constant learning and knowledge crunching through collaboration

Ubiquitous Language used in all project aspects

Code is a reflection of the domain model

Domain Model **Drives** Design.

Speak a Ubiquitous Language within a **bounded context**

Focus on Model of the Domain, rather than Technology

Create small and simple contexts

Domain Model is a **Backbone** of a Domain Language

Constant evolution of the domain model
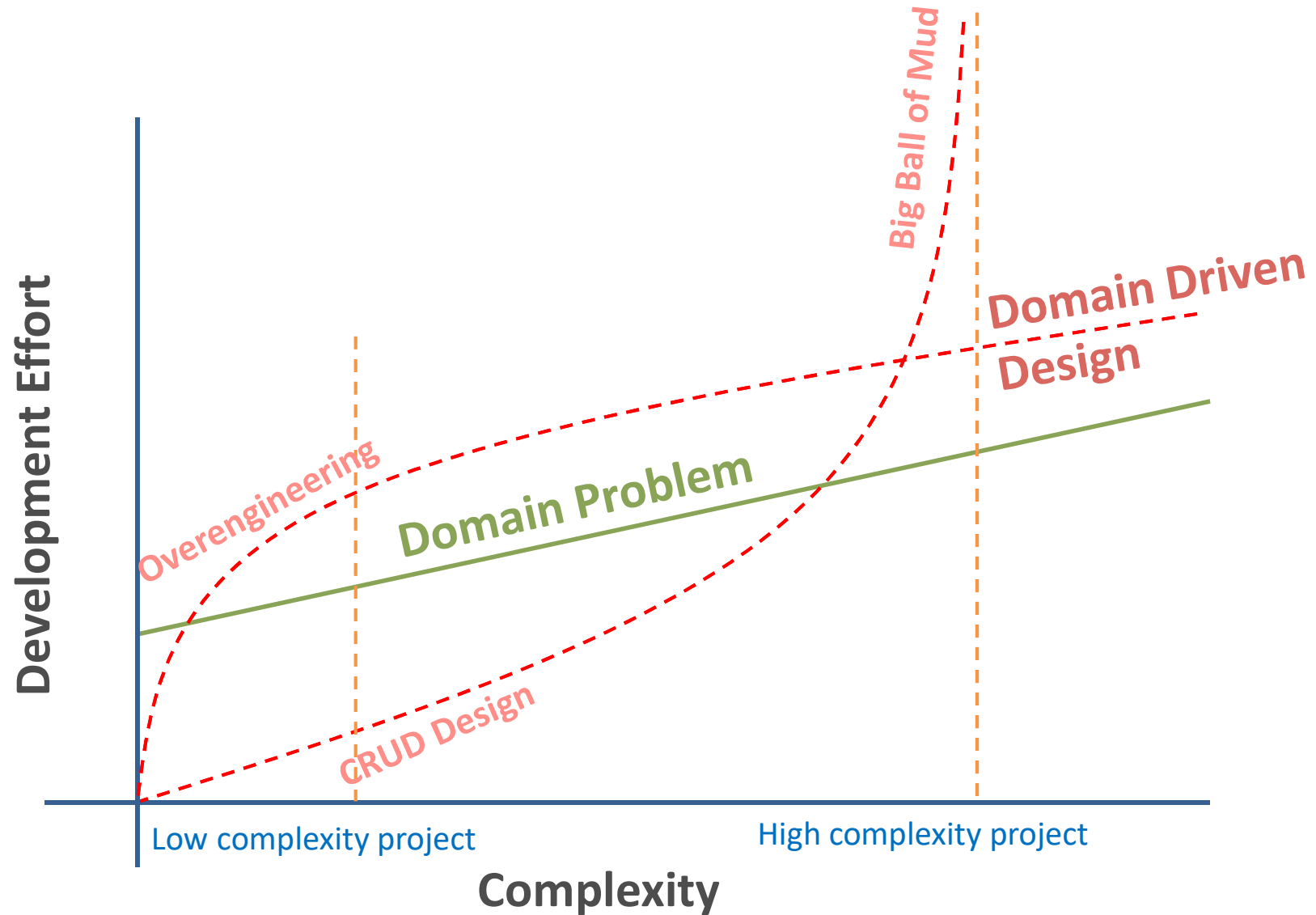
WHEN TO USE DDD AND WHEN NOT?

Cost of DDD

- A lot of time is spend on

- Understanding the business

  - Brainstorming

- Creating a ubiquitous language

- Modeling the business

  - Validation

- Bind the model and the implementation

*This works great in complex for complex **business** domains*

When and why DDD?

Development Effort vs. Complexity chart showing: Big Ball of Mud, Domain Driven Design, Domain Problem, Overengineering, CRUD Design, with markers for "Low complexity project" and "High complexity project".

When to use DDD?

- When the business domain is complex
  - Has initially nothing to do with technical complexity
- When the scope is medium to large
  - Team of > 4 developers
  - Project of > 4 months
- When the application needs to be maintained/evolved for a longer time
- Multiple teams working on the same application

SILVER BULLET™

Requirements for DDD

- You have a skilled, motivated, and passionate team that is eager to learn.

- You have a nontrivial problem domain that is important to your business.

- You have access to domain experts who are aligned to the vision of the project.

- You are following an iterative development methodology.

When not to use DDD

- **Simple CRUD systems**
  - Data centric
  - No or simple business rules
- **Simple small utilities**
  - 1 or 2 developers
  - Build in a few days or weeks

Main point

- Domain Driven Design contains many patterns and best practices for building complex systems

- The daily experience of absolute Being removes all stresses and strains from the human nervous system such that life can be lived in perfection.
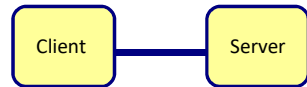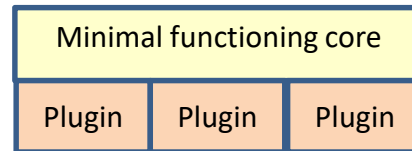
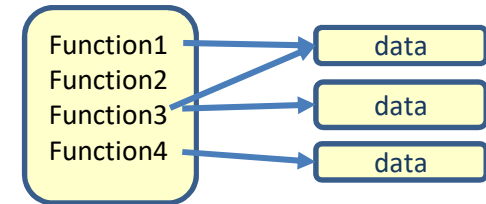Application Architecture
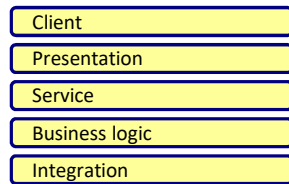
ARCHITECTURAL STYLES

Architecture styles

Client	Server

Client-server

Client
Presentation
Service
Business logic
Integration

Layering

Pipe-and-Filter

Master-Slave

Minimal functioning core		
Plugin	Plugin	Plugin

Microkernel

Function1 Function2 Function3 Function4	data
	data
	data
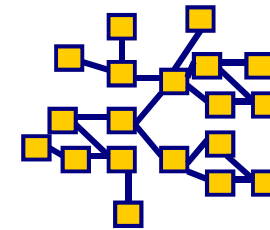
Procedural

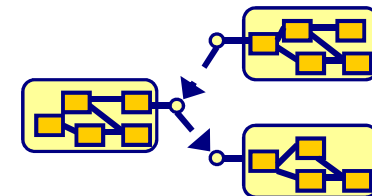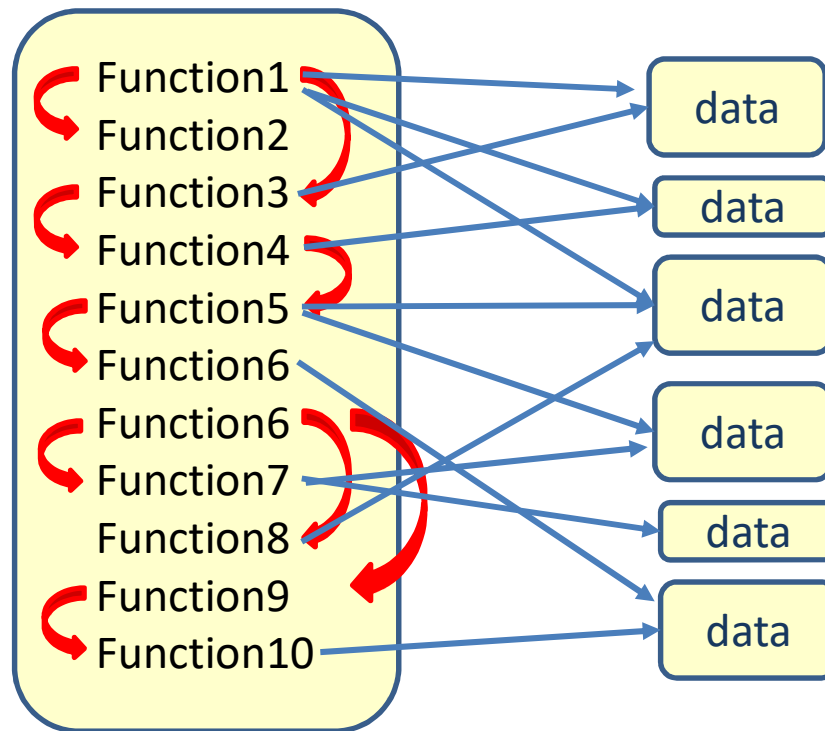**Object oriented**
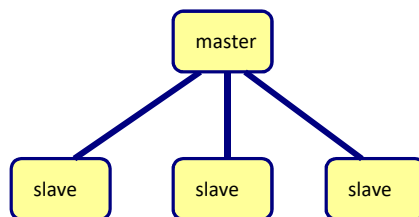
Component

PROCEDURAL PROGRAMMING

Procedural programming

- Functional decomposition
- Functions that work on data structures

Problems with procedural programming

- Tight coupling between functions and data structures

- Many functions
  - Often with duplicated code
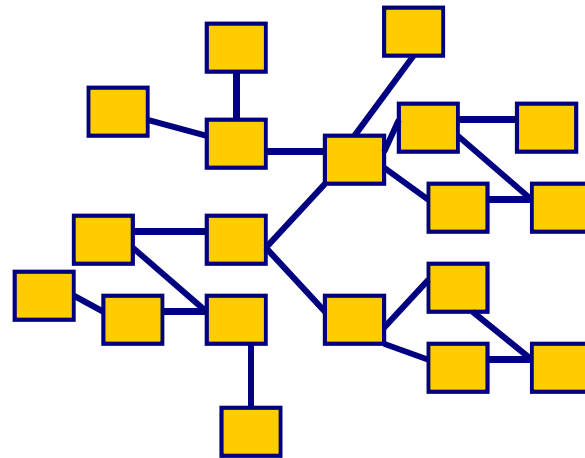
- Hard to maintain

OBJECT ORIENTED

Object Oriented

- Decompose the domain in real world objects
- Data and functionality together

Advantages of object orientation

- High cohesion, low coupling

- Encapsulation
    - Data hiding

- Flexibility
    - Polymorphism

- Reuse ?
    - Inheritance

Problems with OO

- Bigger in size

- Performance
  - OK for near real time and non real time
  - Often not OK for real time

- Difficult to reuse in isolation
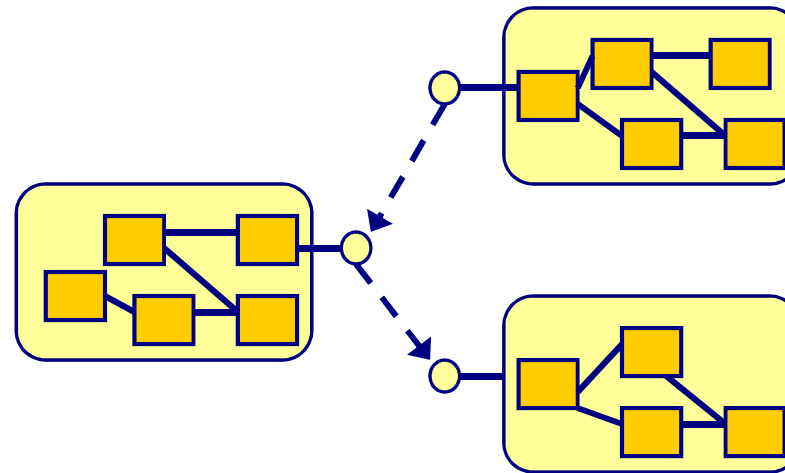 - If you want to reuse functionality, you have to reuse a cluster of objects (Components)

Big ball of mud

COMPONENT BASED DEVELOPMENT

Component Based Development

- Decompose the domain in functional components

  component is some classes together, an aggregate part of the system

What is a component?

- There is no definition

- What we agree upon:

  1. A component has an interface

  2. A component is encapsulated

- Plug-and-play   if we remove a component and replace with another, it will work if it uses the same interface

- A component can be a single unit of deployment
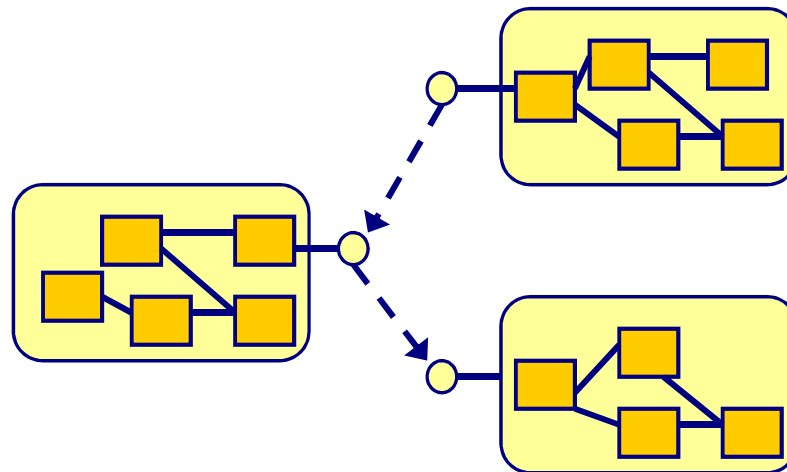
Encapsulation

- The implementation details are hidden
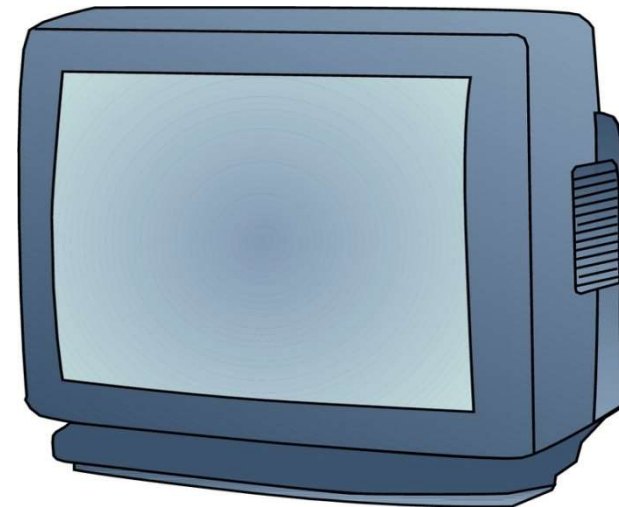
Interface

- The interface tells what you can do (but not how)

Advantages of components

- High cohesion, low coupling

- Flexibility

- Reuse ?   most of the time only technical components we can reuse

Internals of a component

encapsulate and has one or more interfaces
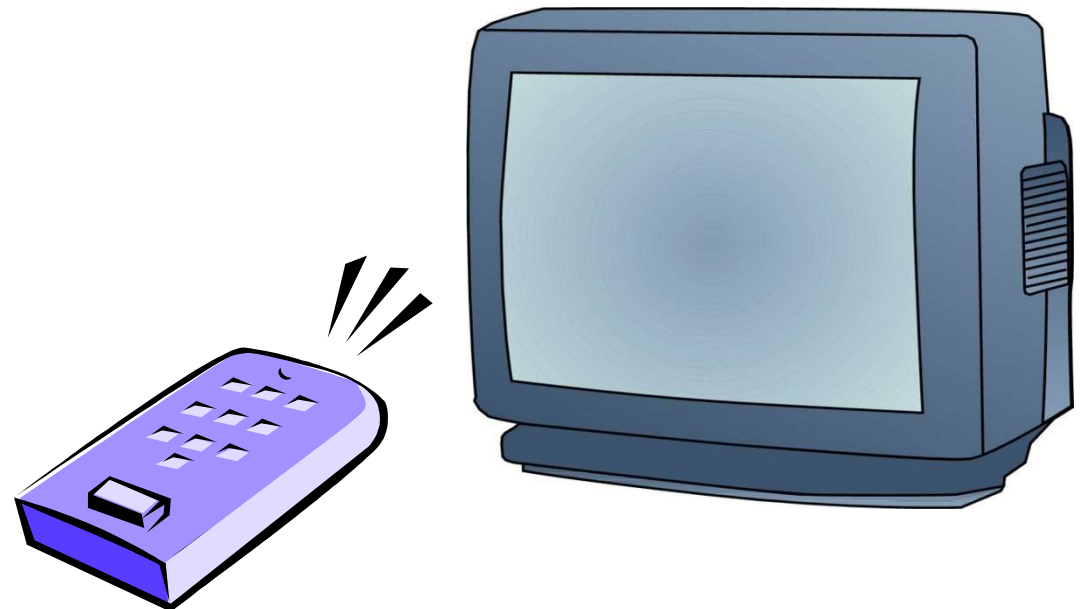
output depends on interface

Interface

Façade/Service

Importance of good interfaces

- Good interfaces increase the quality of our applications
    - Bad interfaces design infects our complete application
- The interfaces we use shapes our code
    - Coherent client
- Public interfaces are forever
    - One chance to get it right
- Usually write-once, read/learn many times by many different people

JDBC API client

```java
public void update(Employee employee) {
    Connection conn = null;
    PreparedStatement prepareUpdateEmployee = null;
    try {
        conn = getConnection();                    // Open connection
        conn.setAutoCommit(false);                 // Start transaction
        prepareUpdateEmployee = conn.prepareStatement("UPDATE Employee SET firstname= ?,
                                        lastname= ? WHERE employeenumber=?");
        prepareUpdateEmployee.setString(1, employee.getFirstName());
        prepareUpdateEmployee.setString(2, employee.getLastName());
        prepareUpdateEmployee.setLong(3, employee.getEmployeeNumber());

        int updateresult = prepareUpdateEmployee.executeUpdate();   // Send the SQL
        conn.commit();                             // Commit transaction
    } catch (SQLException e) {
        conn.rollback();                           // Rollback transaction
        System.out.println("SQLException in EmployeeDAO update() :" + e);
    } finally {                                    // Exception handling
        try {
            prepareUpdateEmployee.close();
            closeConnection(conn);                 // Close connaction
        } catch (SQLException e1) {
            System.out.println("Exception in closing jdbc connection in EmployeeDAO" + e);
        }
    }
}
```

JDBC Template client

```java
public void save(Product product) {
  NamedParameterJdbcTemplate  jdbcTempl = new NamedParameterJdbcTemplate (dataSource);
  Map<String,Object> namedParameters = new HashMap<String,Object>();
  namedParameters.put("productnumber", product.getProductnumber());
  namedParameters.put("name", product.getName());
  namedParameters.put("price", product.getPrice());
  int updateresult = jdbcTempl.update("INSERT INTO product VALUES ( :productnumber,
                                       :name, :price)",namedParameters);

}
```

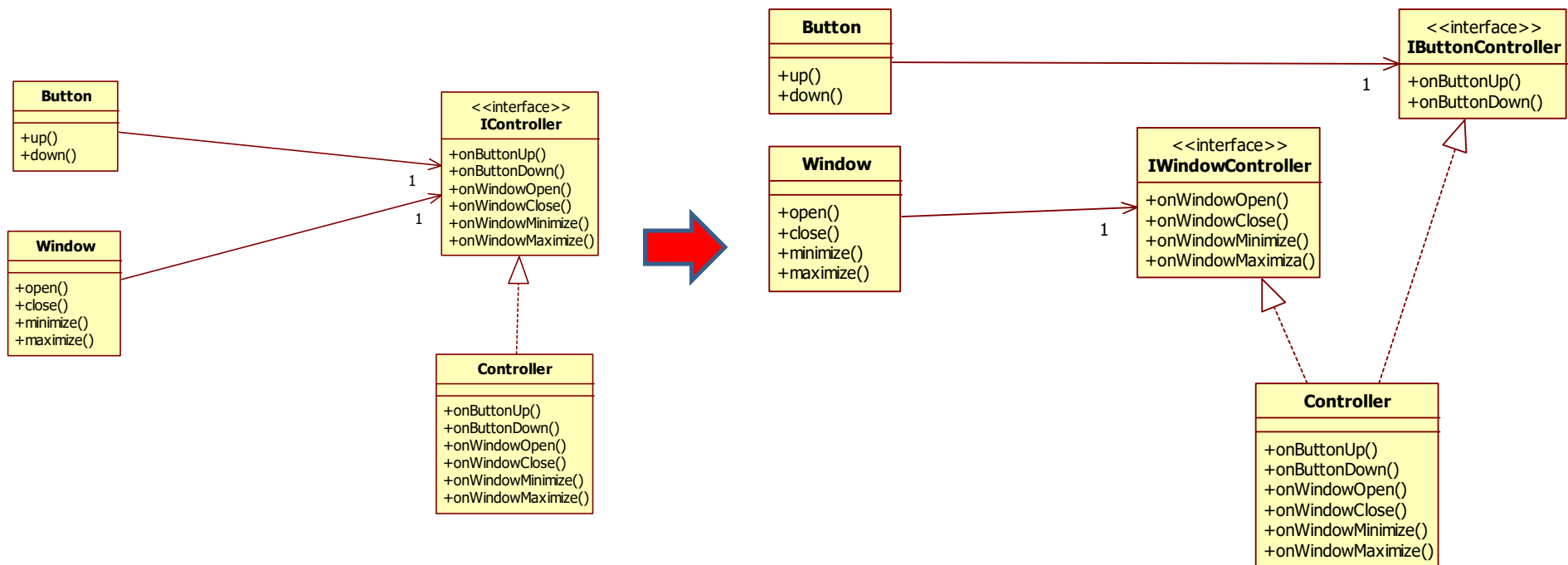The template takes care of connection, transaction and exception handling

Interface design

- Start with the <mark>client first</mark>

- <mark>Single responsibility</mark> principle

- Interface segregation principle
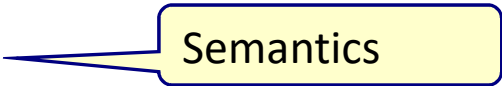
- Easy to use

- Easy to learn

Interface segregation principle

■ Clients should not be forced to depend on methods (and data) they do not use

The interface should be easy to use

- And hard to misuse.
    - Easy to do simple things
    - Possible to do complex things
    - Impossible (or at least difficult) to do wrong things
- The interface should be simple
- No surprising behavior — Semantics
    - Don't do anything that is relevant to the client and that you cannot derive from the names of the functions and parameters (or comments)

The interface should be easy to learn

- Well documented
- Names matter (classes, functions, variables)
  - Clear code

```
bucket.empty()
```

👎 NOT OK

```
getCurrValue()
getCurValue()
getCurVal()
```

```
bucket.isEmpty()

bucket.makeEmpty()
```
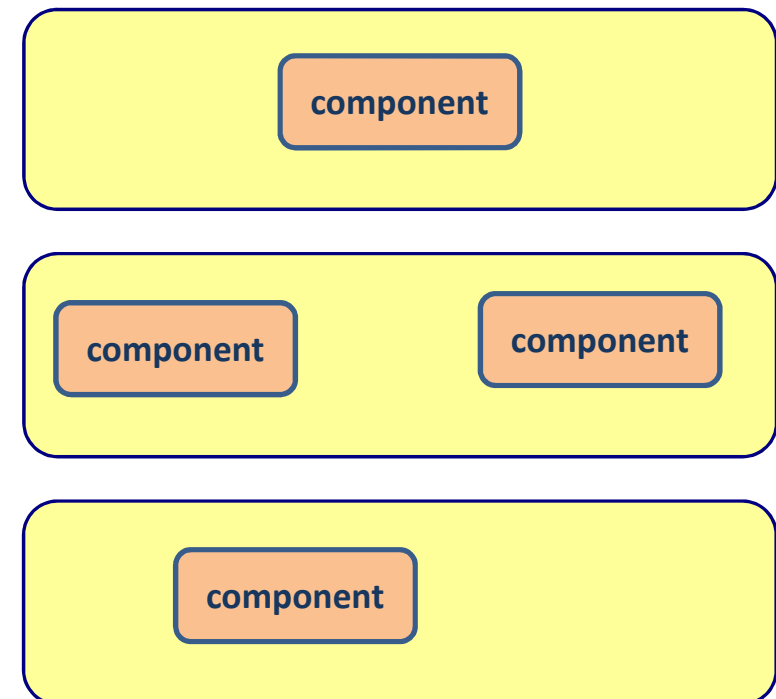
👍 OK
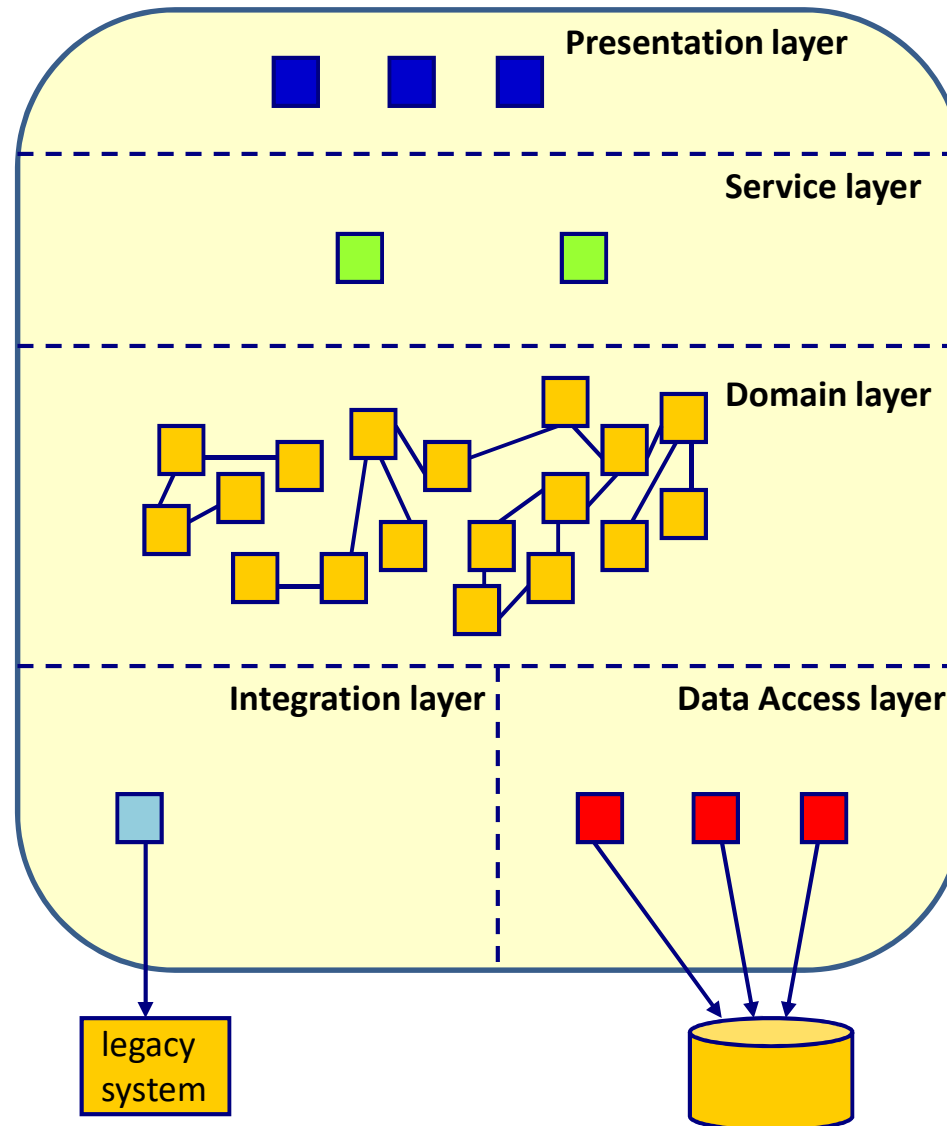
```
getCurrentValue()
```

Components in practice

- Components most often do not cross layers
- Business components are often nog reused
- Components are mostly project assets, not company assets.

Object-Oriented Design

Presentation layer

Service layer

Domain layer

Integration layer

Data Access layer

•Everything is connected with everything else

legacy system

Component Based Design (CBD)

Presentation layer

Service layer

Domain layer

Integration layer

Data Access layer

legacy system
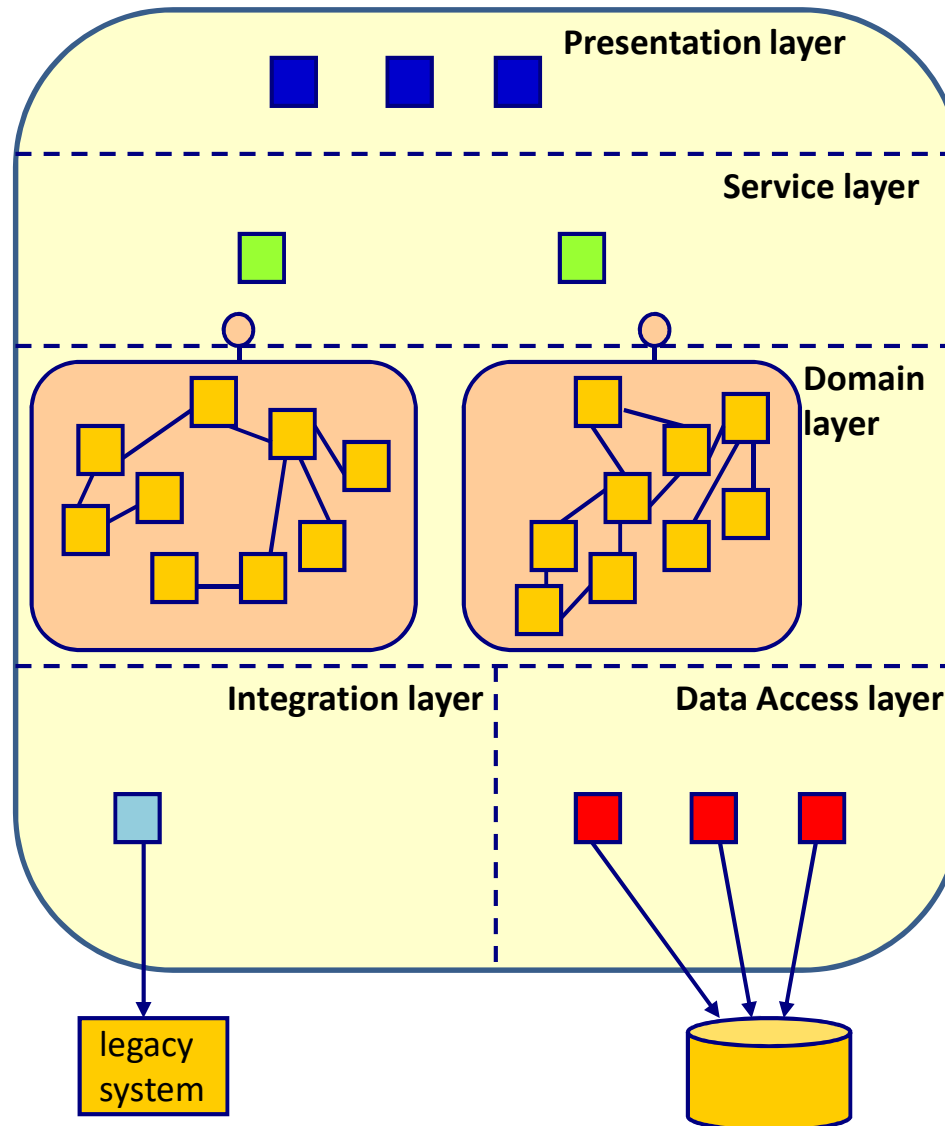
•Better separation of concern

•More flexibility

•Possibility for reuse

Service/Component Based Design

Presentation layer
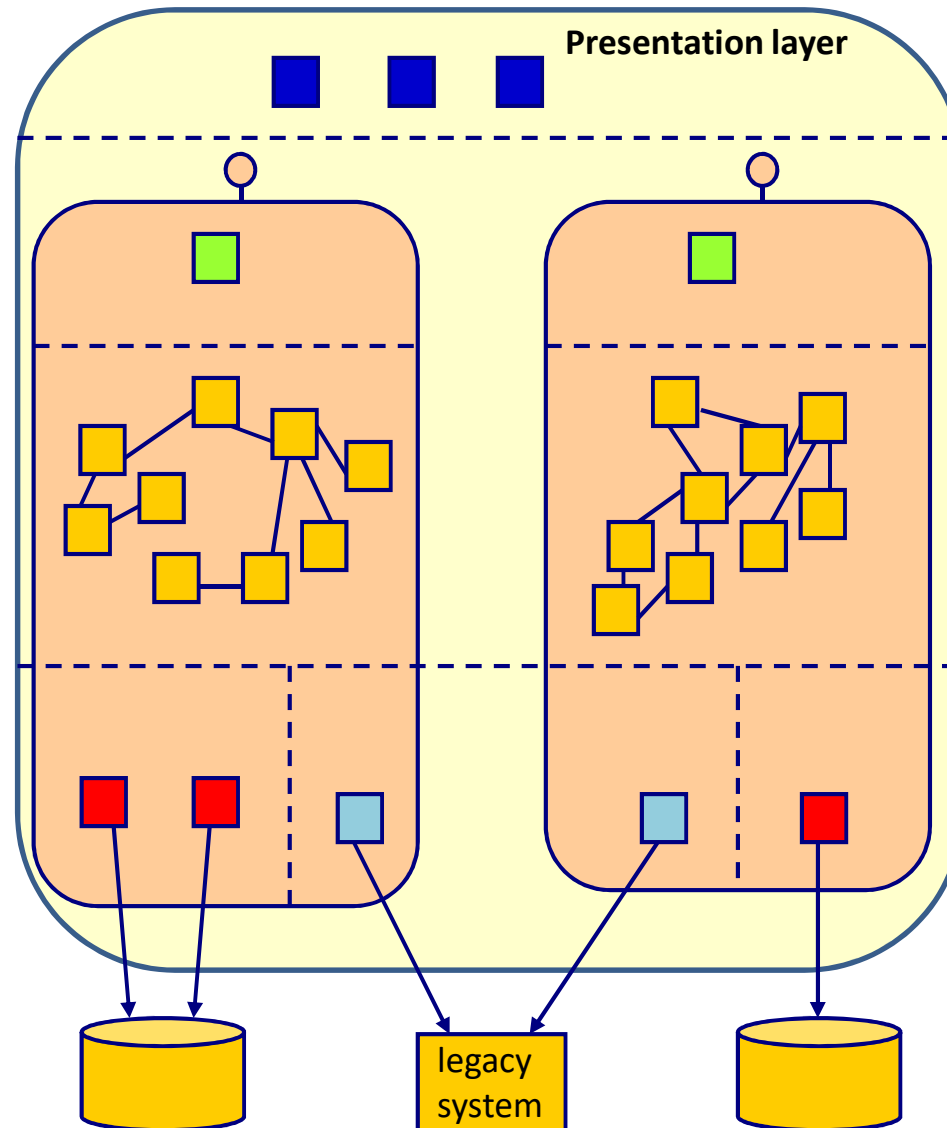
- Better separation of autonomous services

- Easier to build the services in isolation by different projects

- More flexibility

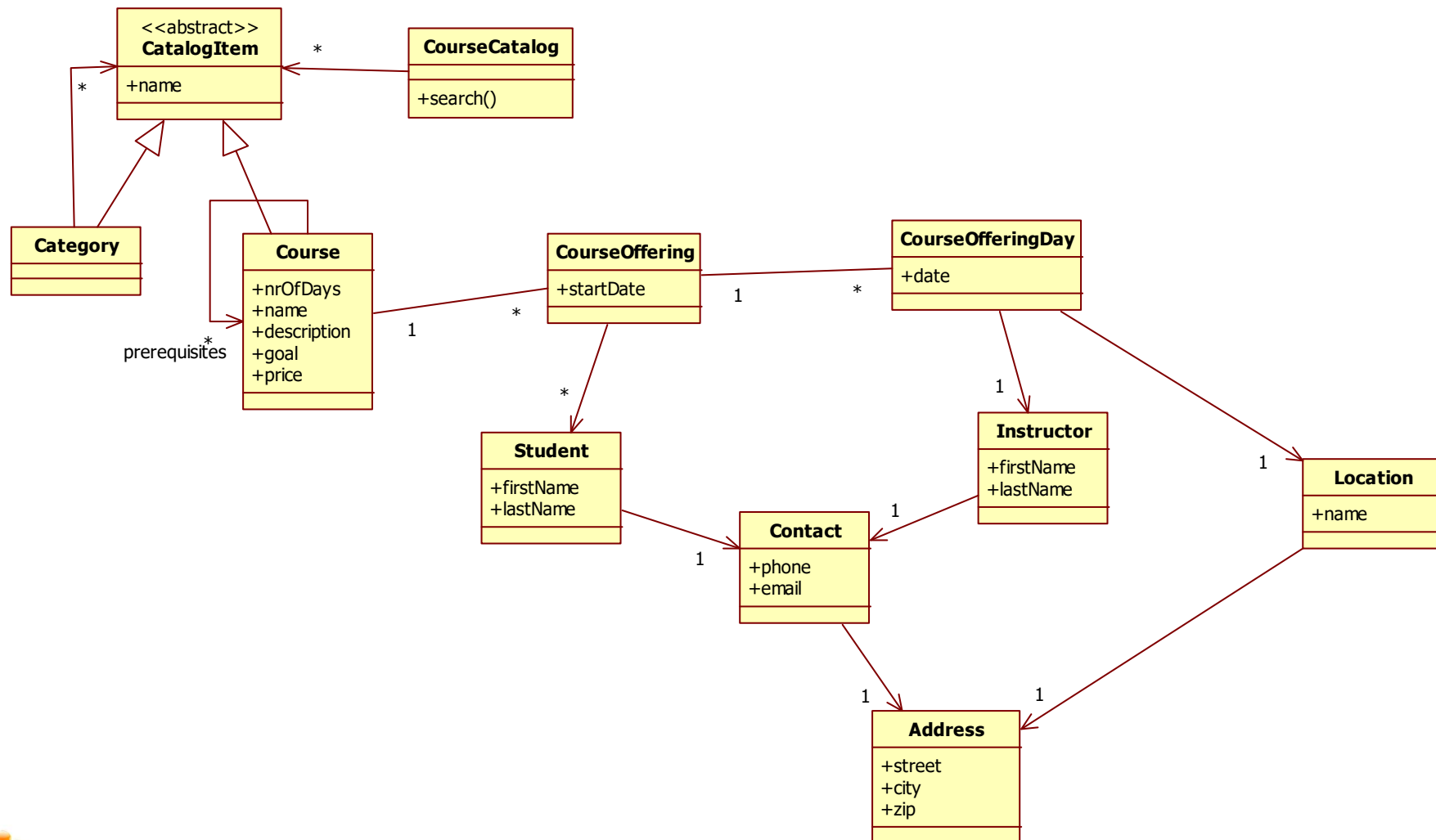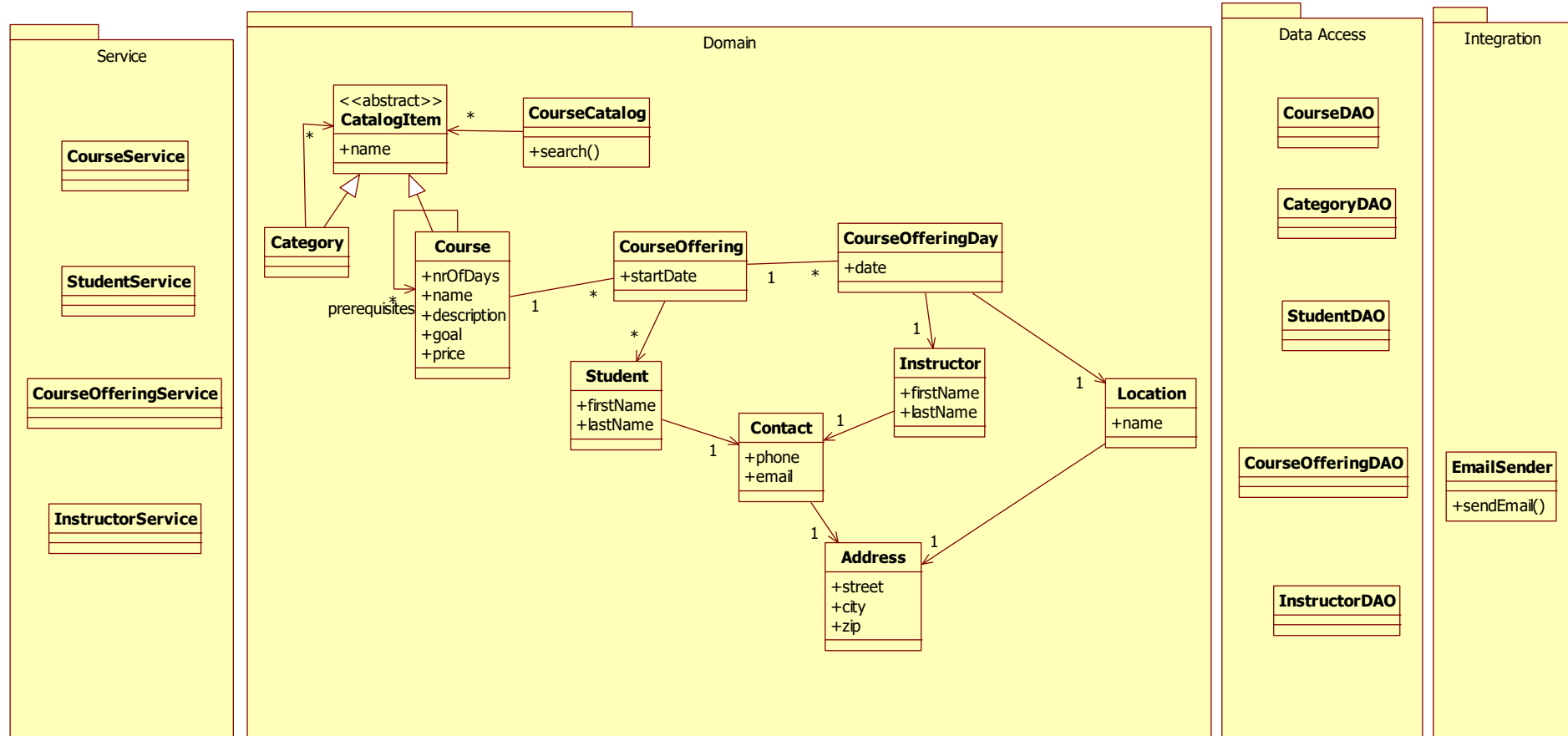- Possibility for reuse

legacy system

Main point

- Components are encapsulated and completely autonomous plug-and-play elements.

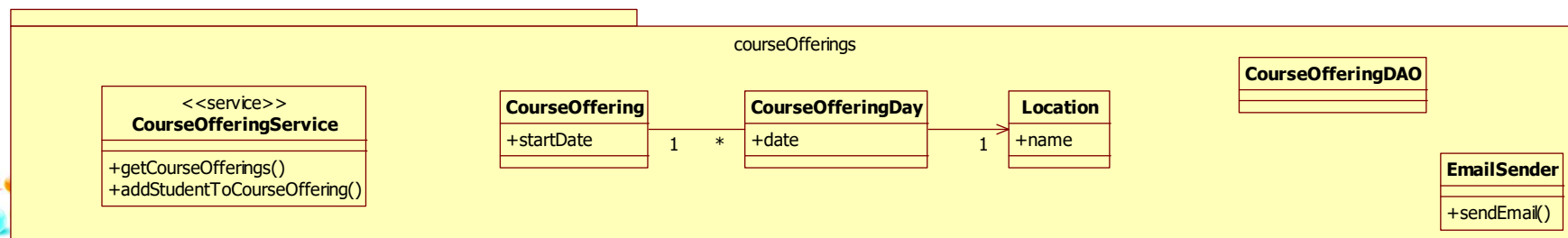- The human nervous system is capable to transcend to that abstract field of pure consciousness which lies at the basis of the whole creation.
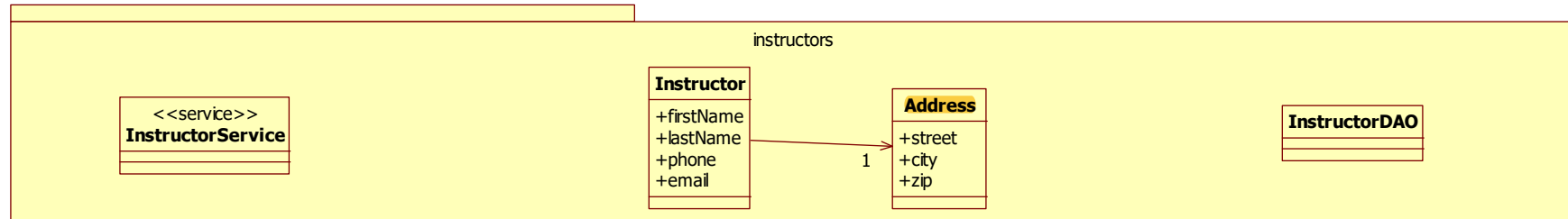
Course Registration Domain Model

Course Registration Design

Course Registration with components

courses

<>
CatalogItem

+name

<<service>>
CourseService

+findCourse()
+addCourse()
+updateCourse()
+addCategory()
+getCategories()

Category

Course

+nrOfDays
+name
+description
+goal
+price

prerequisites *

*

CourseDAO

+find()
+save()
+update()
+delete()

CategoryDAO

---

students

<<service>>
StudentService

+addStudent()

Student

+firstName
+lastName
+phone
+email

1

Address

+street
+city
+zip

StudentDAO

---

instructors

<<service>>
InstructorService

Instructor

+firstName
+lastName
+phone
+email

1

Address

+street
+city
+zip

InstructorDAO

---

courseOfferings

<<service>>
CourseOfferingService

+getCourseOfferings()
+addStudentToCourseOffering()

CourseOffering

+startDate

1   *

CourseOfferingDay

+date

1

Location

+name

CourseOfferingDAO

EmailSender

+sendEmail()

Course Registration with components

Shared data between components

ProductService

+getProduct(productNumber)
+findProduct()
+addProduct()

ShoppingService

+addProduct(Product)

Product

+name
+productNumber
+price
+description
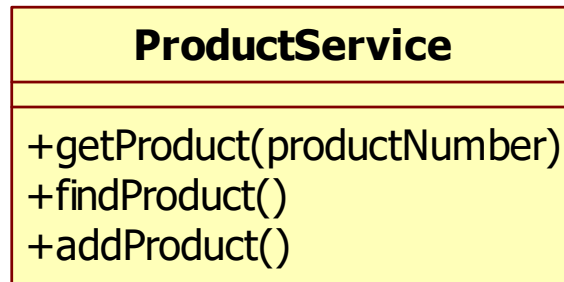
Both the ProductService and the ShoppingService depend on the same Product implementation
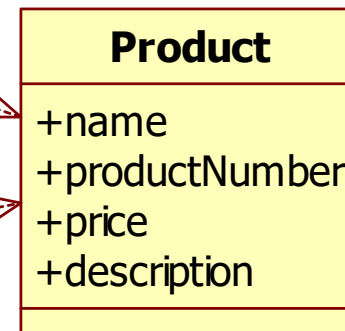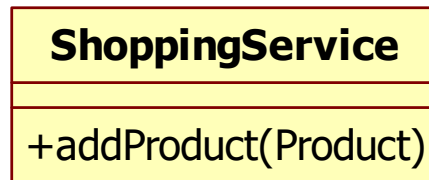
If the ProductService needs to change the Product, then the ShoppingService also needs to change

No shared data between components
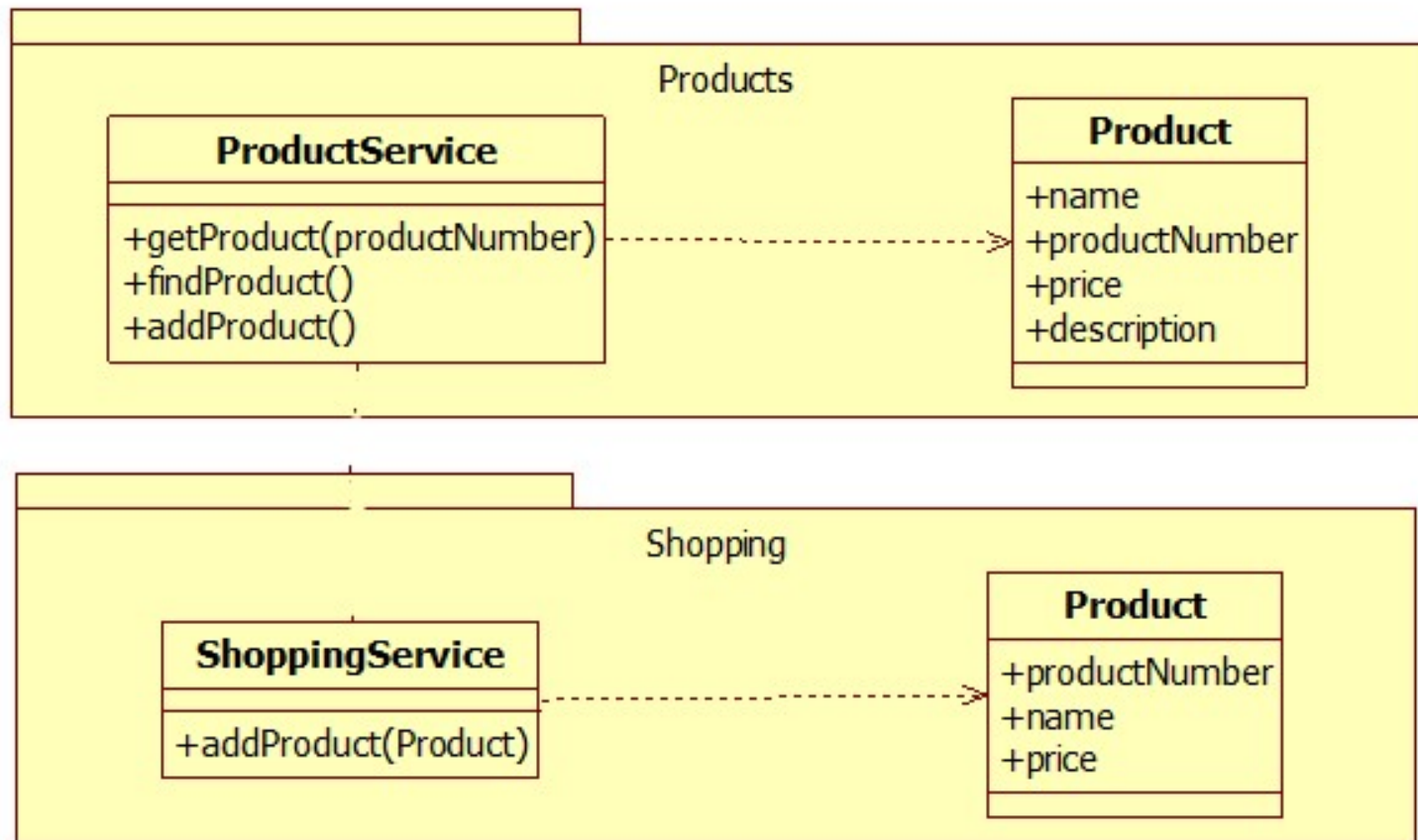
Data Transfer Objects (DTO)

- Object that contains only attributes and getters and setters

```
┌─────────────┐        ┌─────────┐        ┌─────────────┐
│             │        │   DTO   │        │             │
│  Component  │───────────────────────────▶│  Component  │
│             │        │         │        │             │
└─────────────┘        └─────────┘        └─────────────┘
```

Data Transfer Objects (DTO)

Products

ProductService
+getProduct(productNumber)
+findProduct()
+addProduct()

Product
+name
+productNumber
+price
+description

ProductDTO
+productNumber
+name
+price

Both the ProductService and the ShoppingService depend only on the ProductDTO

Shopping

ShoppingService
+addProduct(Product)

Product
+productNumber
+name
+price
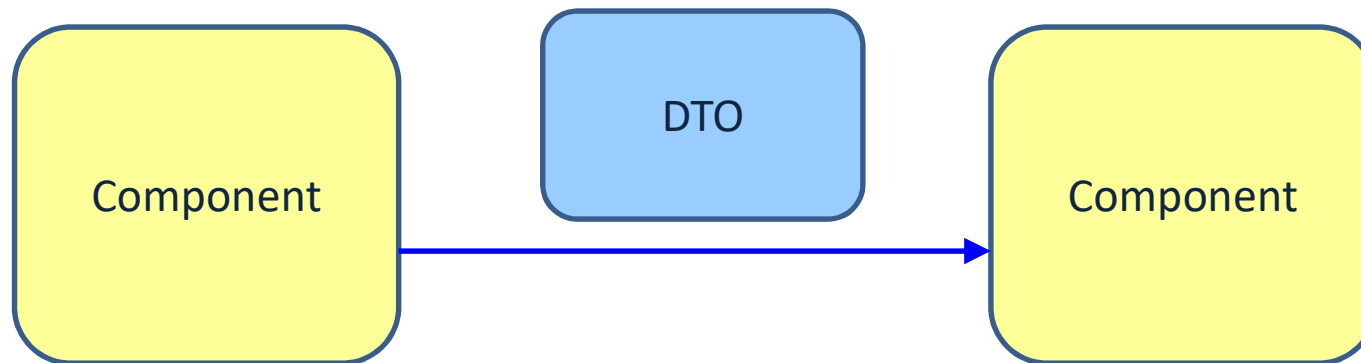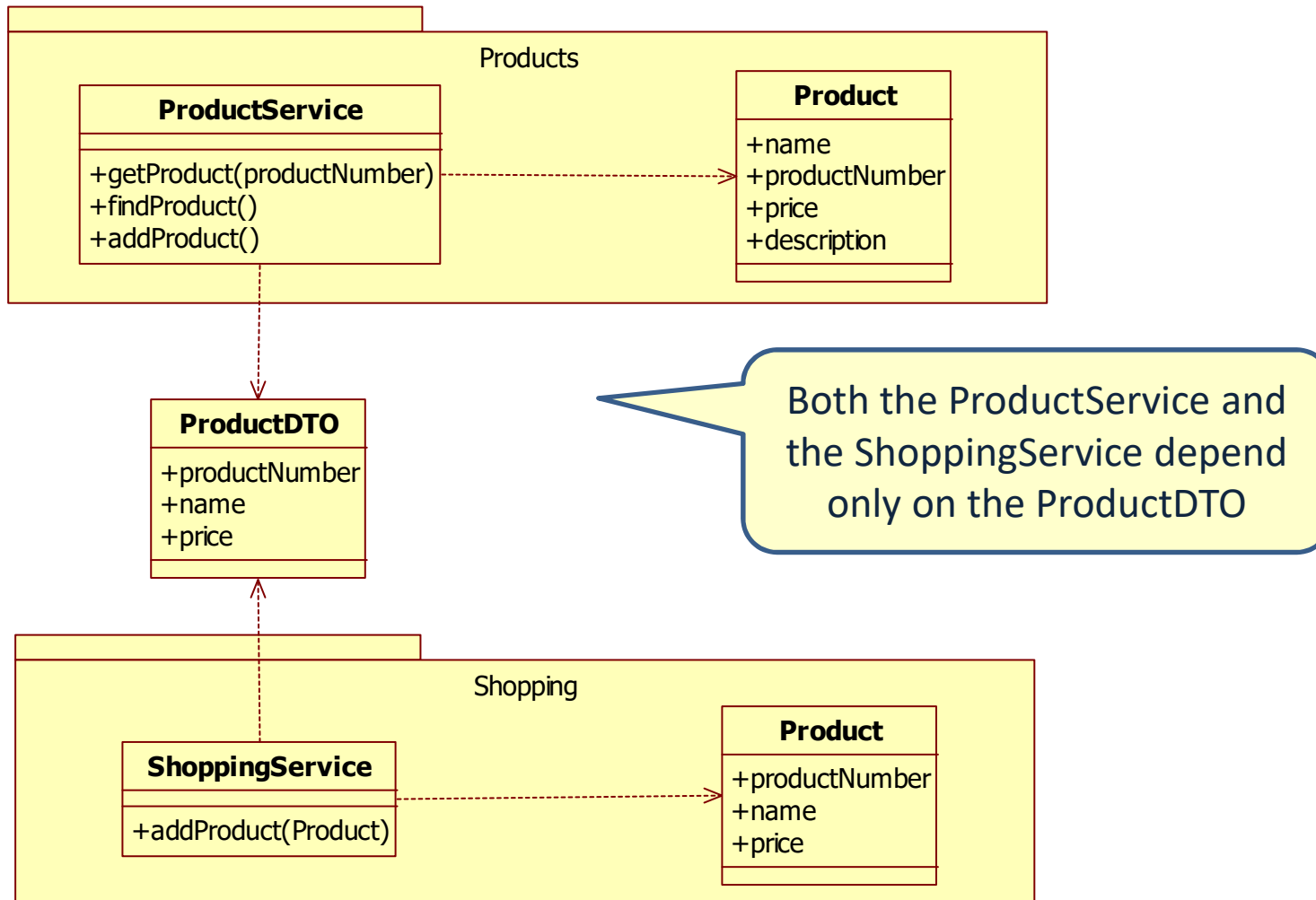
How does it all fits together?

- **Bounded context**
- **Component**
- **Service**
- **Module**

DTO lives in service layer, the service only return DTOs

Entities, value objects, domain services, events

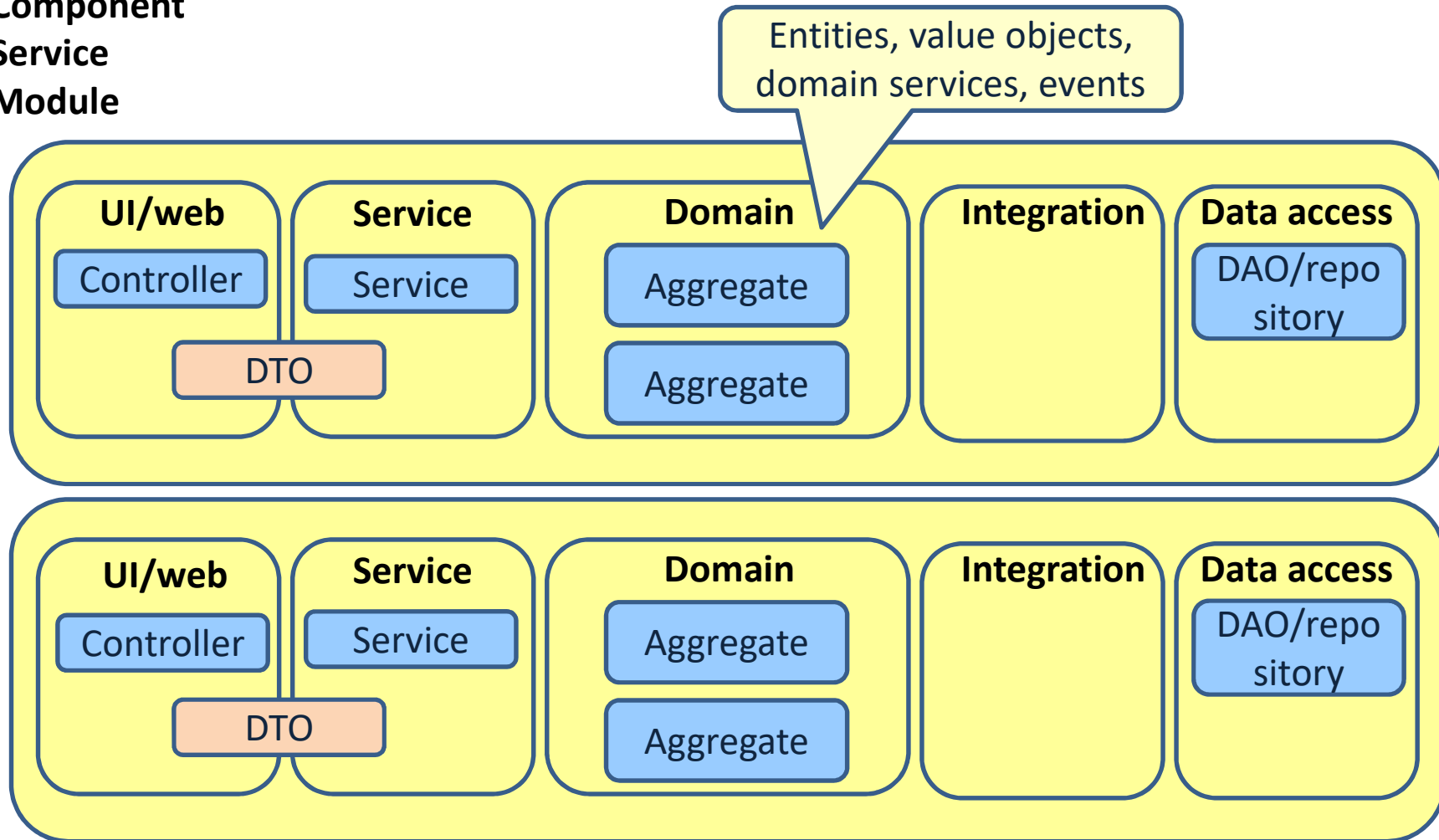| UI/web	Service	Domain	Integration	Data access
Controller	Service	Aggregate		DAO/repository
DTO		Aggregate		

UI/web	Service	Domain	Integration	Data access
Controller	Service	Aggregate		DAO/repository
DTO		Aggregate		

SPRING BOOT EVENTS

Events

```java
public class AddCustomerEvent {
  private String message;

  public AddCustomerEvent(String message) {
    this.message = message;
  }

  public String getMessage() {
    return message;
  }
}
```

A simple event class

Immutable

data should be in the event to sent

Event publisher and listener

```java
@Service
public class CustomerServiceImpl implements CustomerService {
    @Autowired
    private ApplicationEventPublisher publisher;

    public void addCustomer() {
        publisher.publishEvent(new AddCustomerEvent("New customer is added"));
    }
}
```

Inject a publisher

```java
@Service
public class Listener {

    @EventListener
    public void onEvent(AddCustomerEvent event) {
        System.out.println("received event :" + event.getMessage());;
    }
}
```

Listen to AddCustomer events

we do not put the whole data in event, only some data in the DTO

Asynchronous events

```
@Service
@EnableAsync
public class Listener {

  @Async          on its own thread          events by default is sync
  @EventListener
  public void onEvent(AddCustomerEvent event) {
    System.out.println("received event :" + event.getMessage());;
  }
}
```

Asynchronous means that you can execute multiple things at a time and you don't have to finish executing the current thing

Connecting the parts of knowledge with the wholeness of knowledge

1. A ubiquitous language is a common language we use for all aspects within a software project

2. One large domain needs to be split-up into many smaller domain contexts who have their own domain model.

3. **Transcendental consciousness** is the source of all contexts.

4. **Wholeness moving within itself:** In Unity Consciousness, one realizes that everything in creation are just expressions of the field of Pure Intelligence.