

# 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

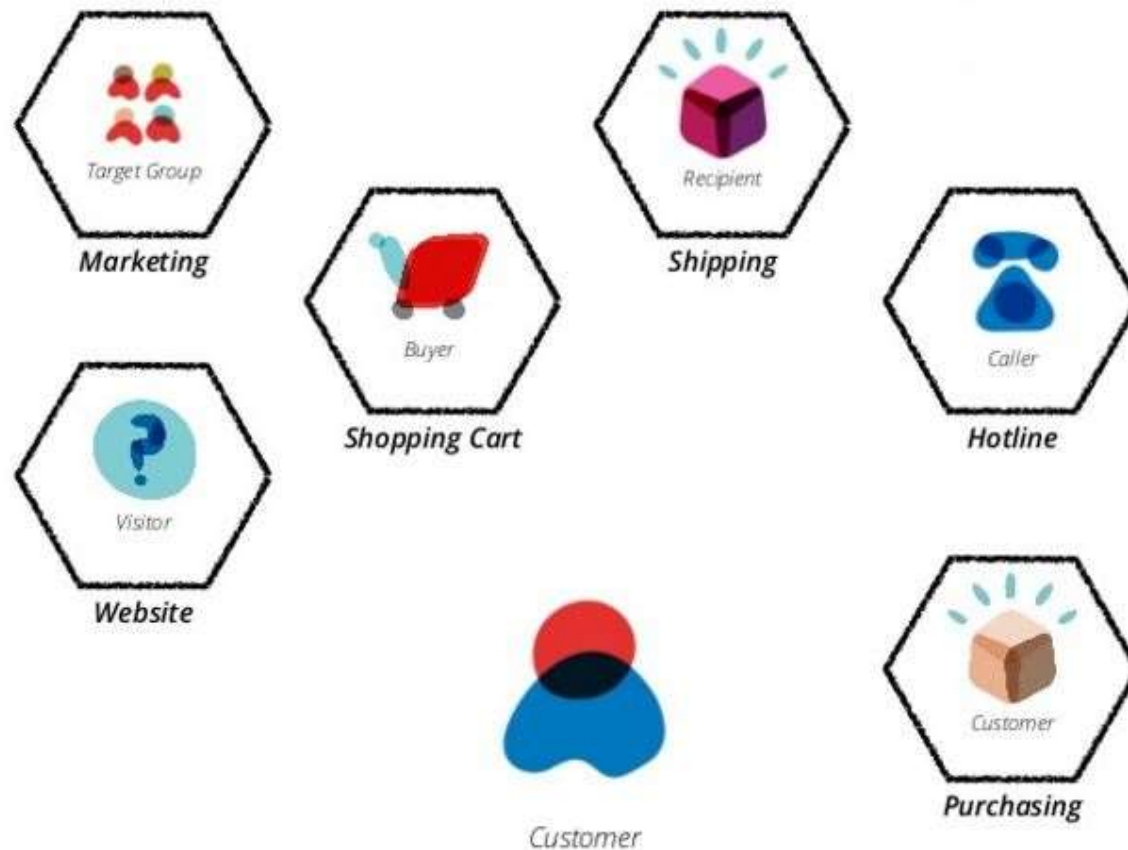
---

- 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.

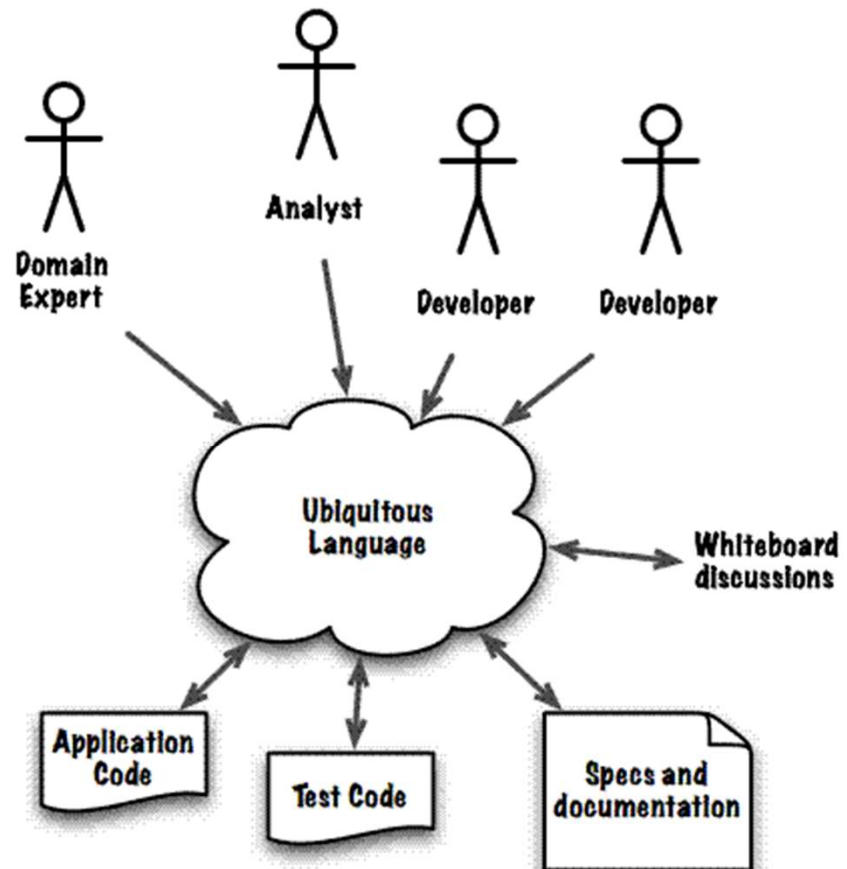


*We need a common language*



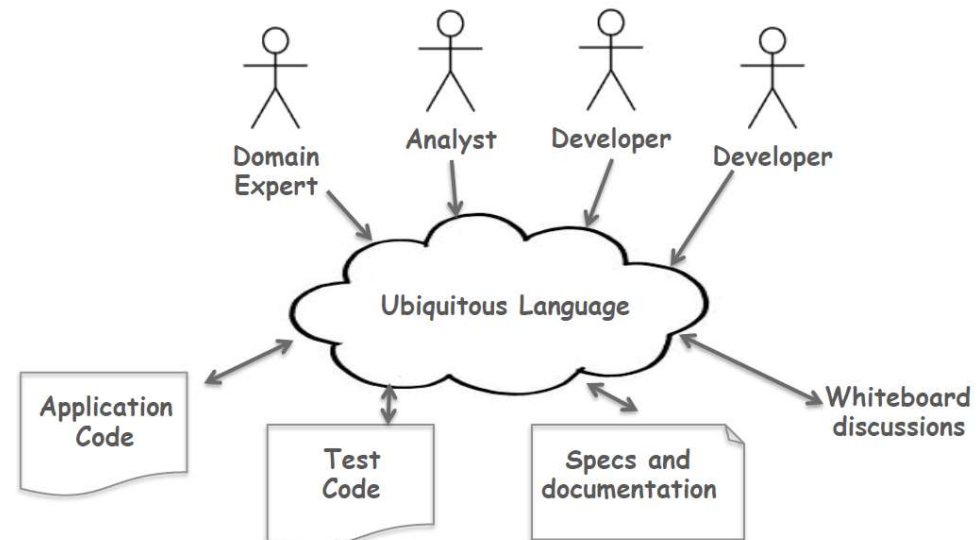
# Ubiquitous Language

- 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



# **EXAMPLE OF MODEL AND UBIQUITOUS LANGUAGE**



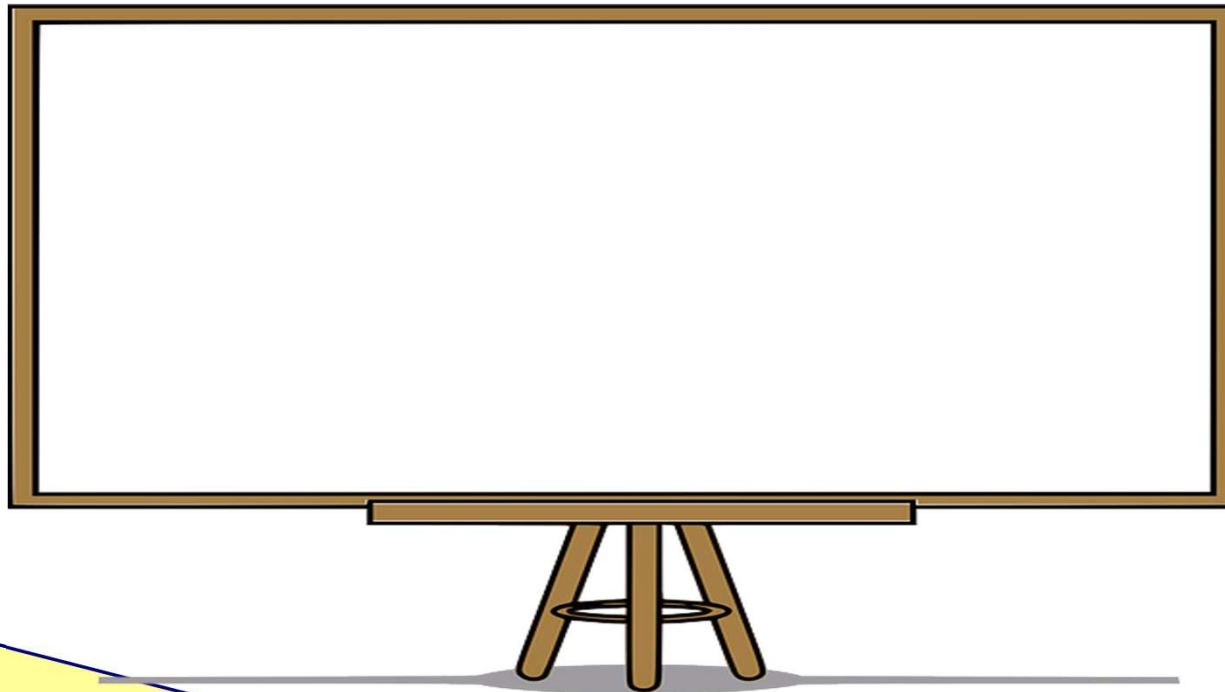


# Flight control system

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



aircraft  
controller



developer

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.

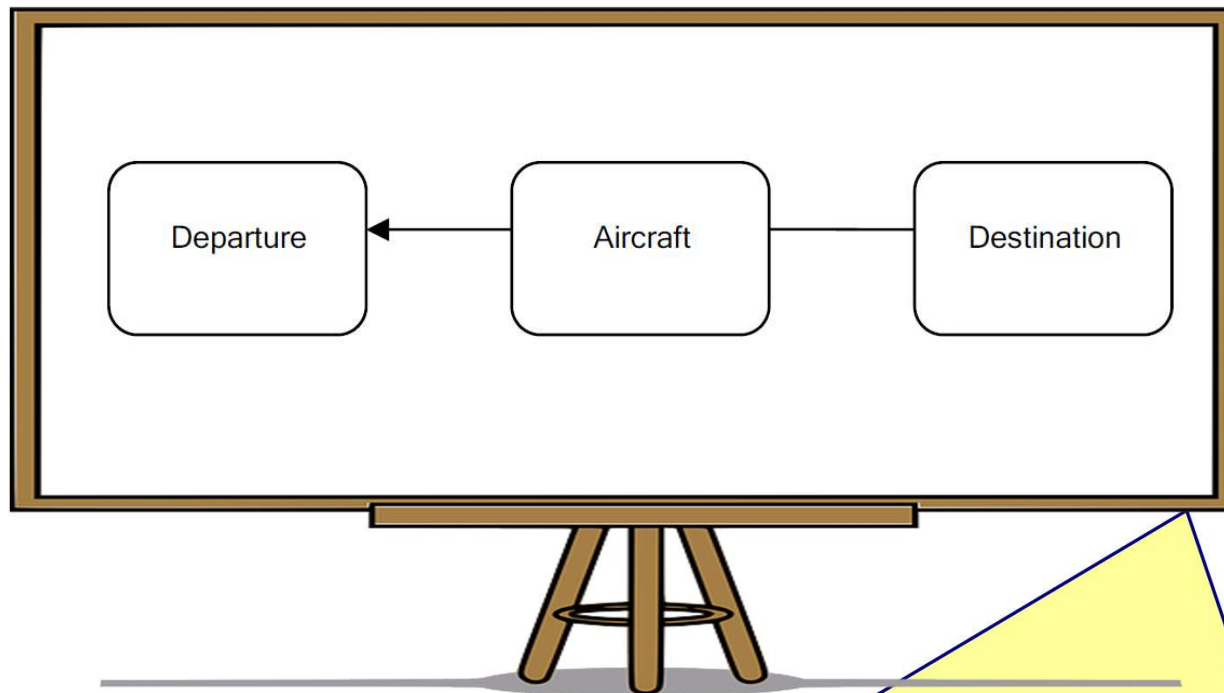


# Flight control system

OK, so we get something like this.



aircraft  
controller



developer

That's easy. When it flies, the plane can just choose any air path the pilots like? Is it up to them to decide which way they should go, as long as they reach the destination?

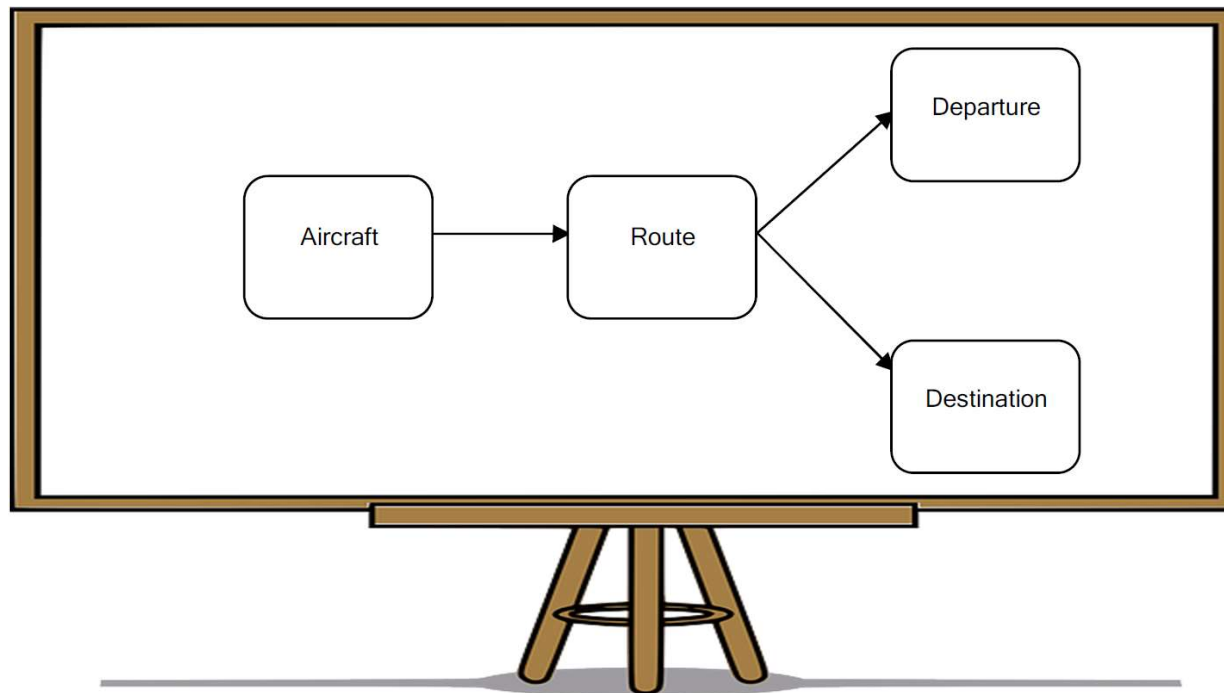


# Flight control system

Oh, no. The pilots receive a **route** they must follow.  
And they should stay on that route as close as possible.



aircraft  
controller



developer

Can you explain routes to me?

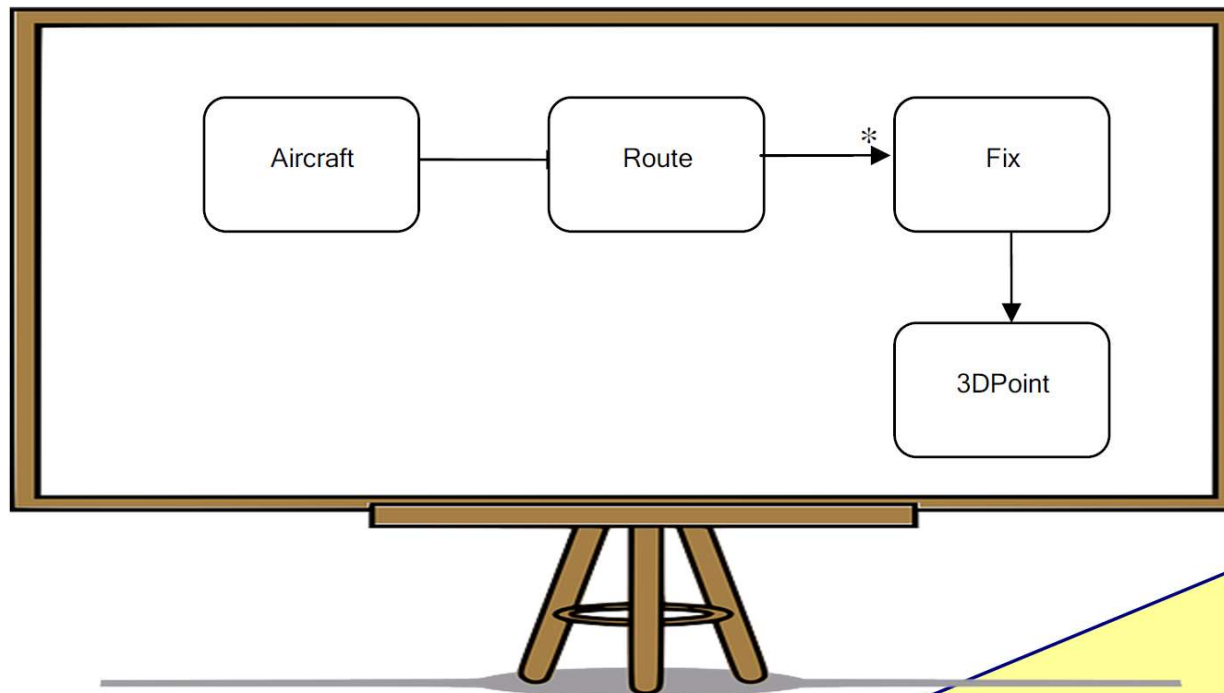


# Flight control system

Well, a route is made up of small segments, and each segment has predetermined fixed points



aircraft  
controller



developer

OK, then let's call each of those points a **fix**, because it's a fixed point. And, by the way, the **departure** and **destination** are just **fixes**. I'm thinking of this **route** as a 3D path in the air. If we use a Cartesian system of coordinates, then the **route** is simply a series of 3D points.

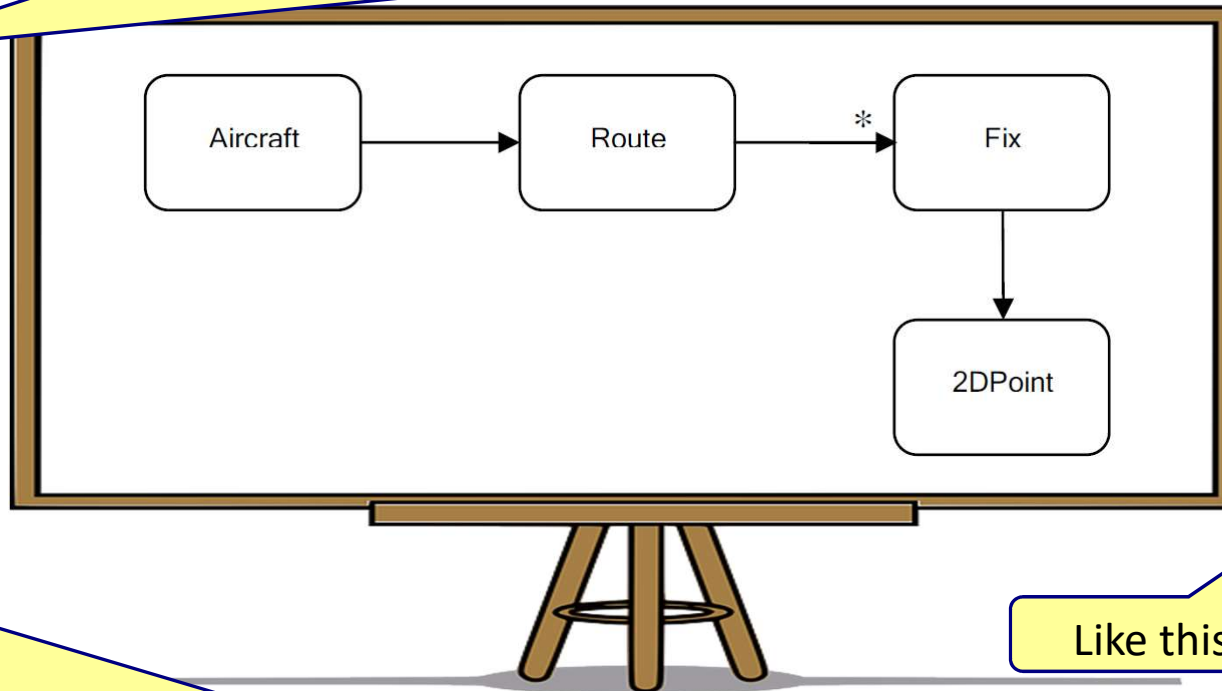


# Flight control system

I don't think so. We don't see **route** that way. The **route** is actually the projection on the ground of the expected air path of the airplane. The **route** goes through a series of points on the ground determined by their **latitude** and **longitude**.



aircraft  
controller



developer

Like this?

Yes, the **altitude** that an airplane is to have at a certain moment is also established in the **flight plan**.

**Flight plan?** What is that?

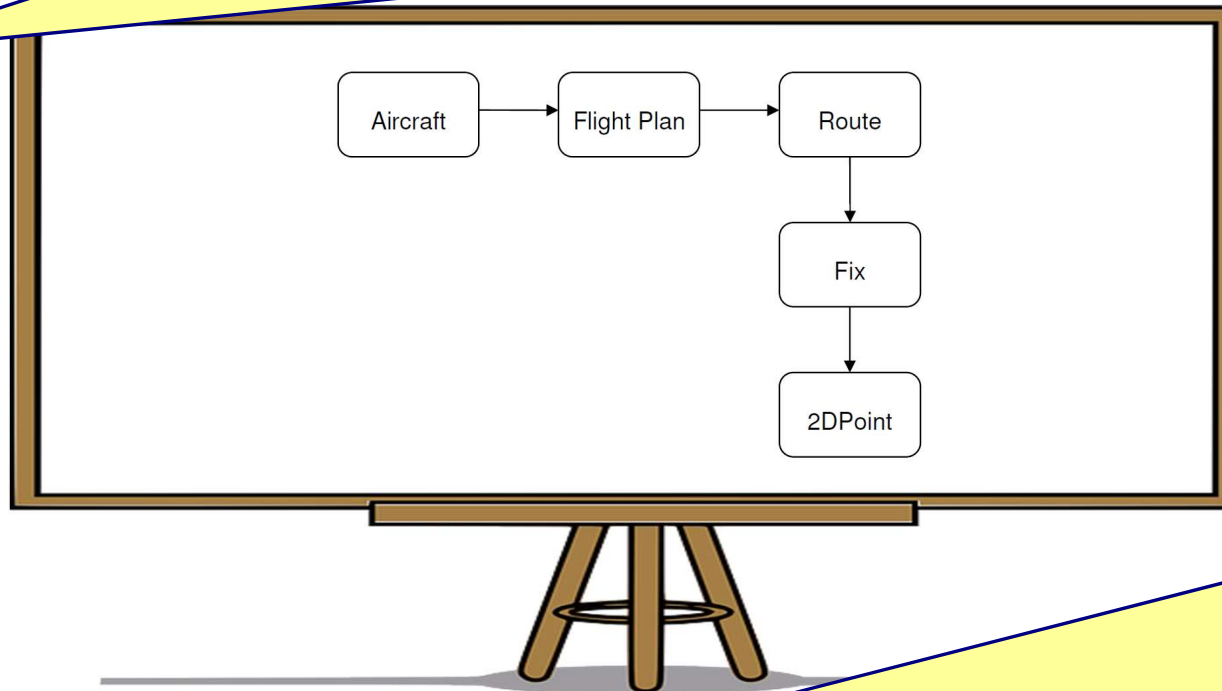


# Flight control system

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



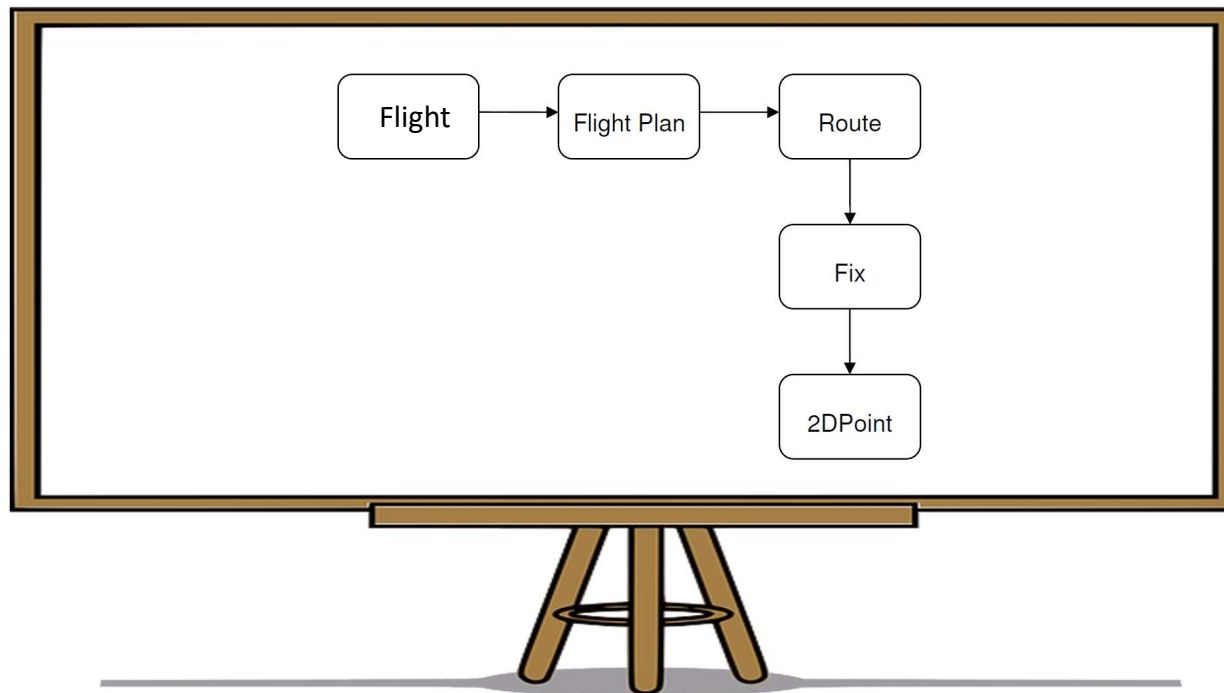
developer

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



# 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

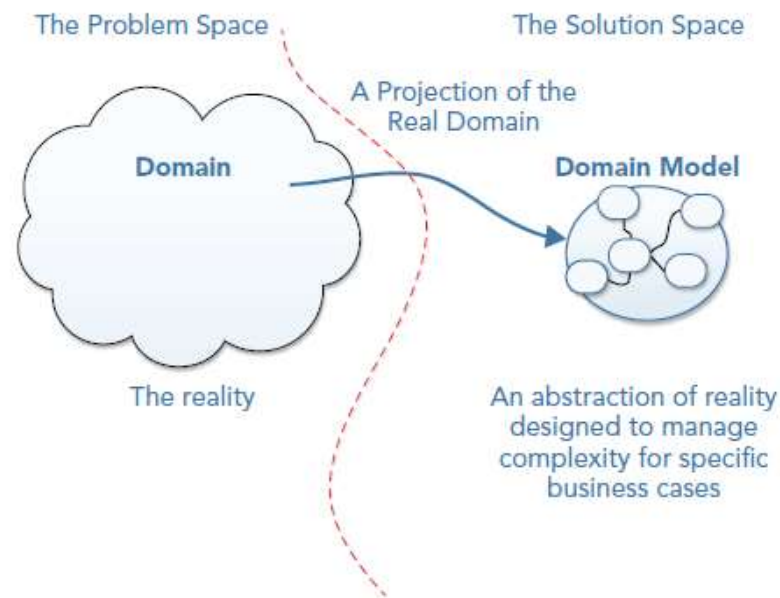


- More complexity -> More modeling
  - Higher level of abstraction
  - Allows for visualization
  - Vehicle of communication



# 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**



# Domain model

- Simplification of reality
- Area of interest

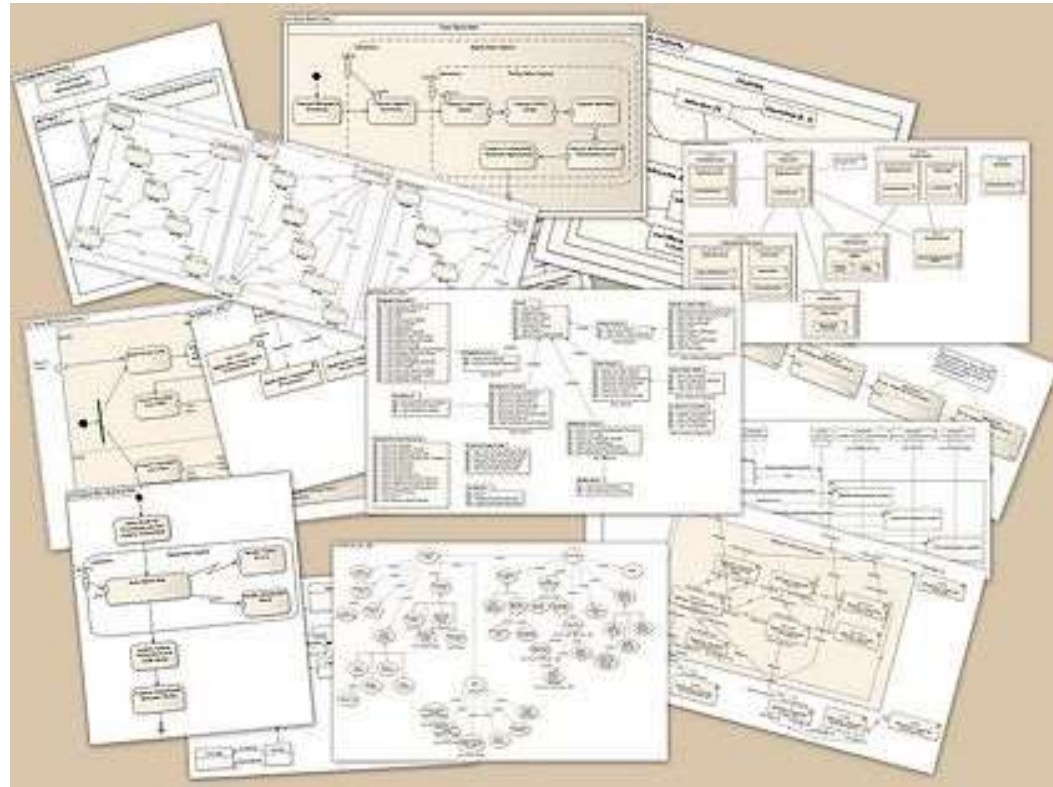




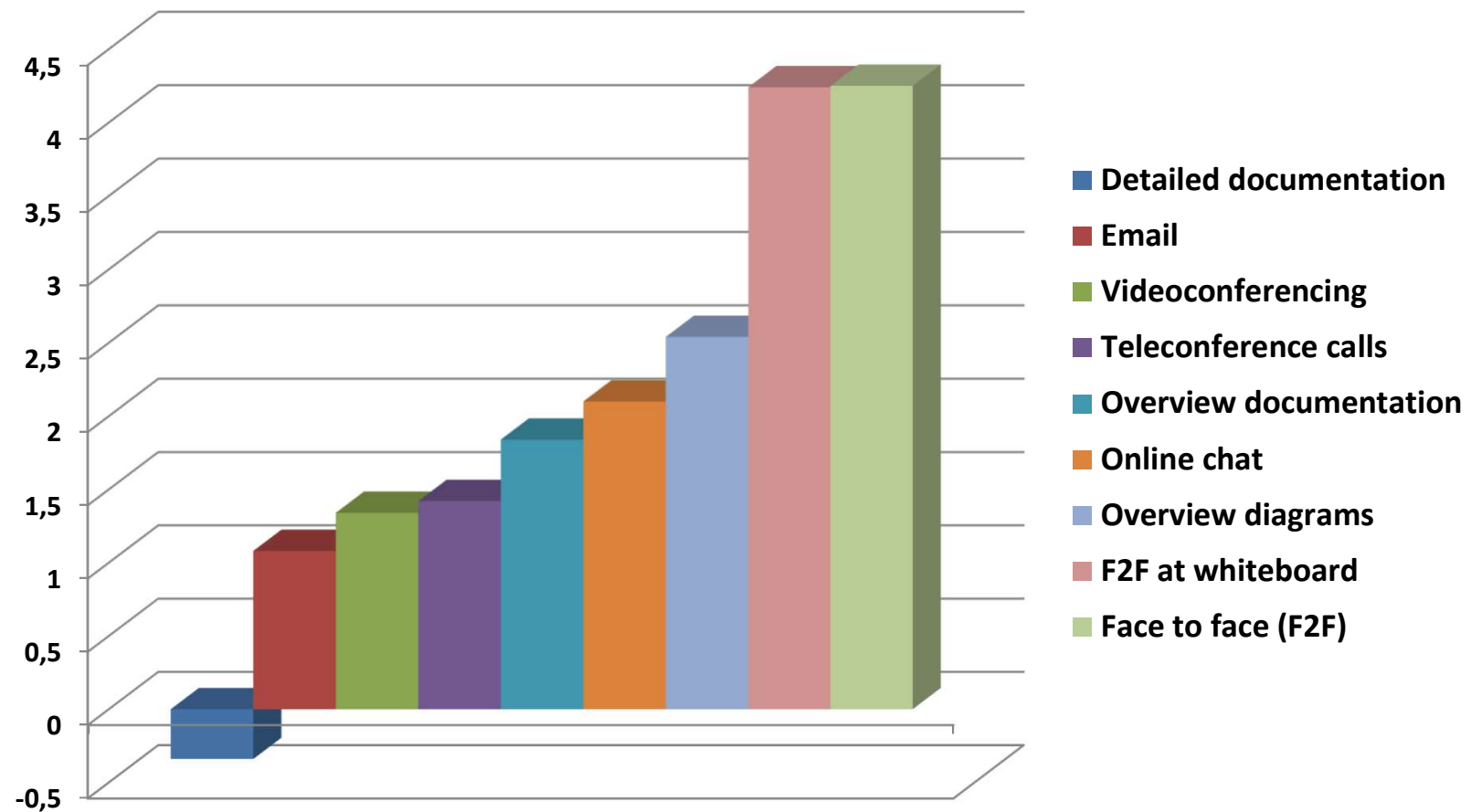
# Structure of the domain model

---

- A domain model is not a particular diagram
- Use the format that communicates the best
  - Diagram
  - Text
  - Code
  - Table
  - Formula



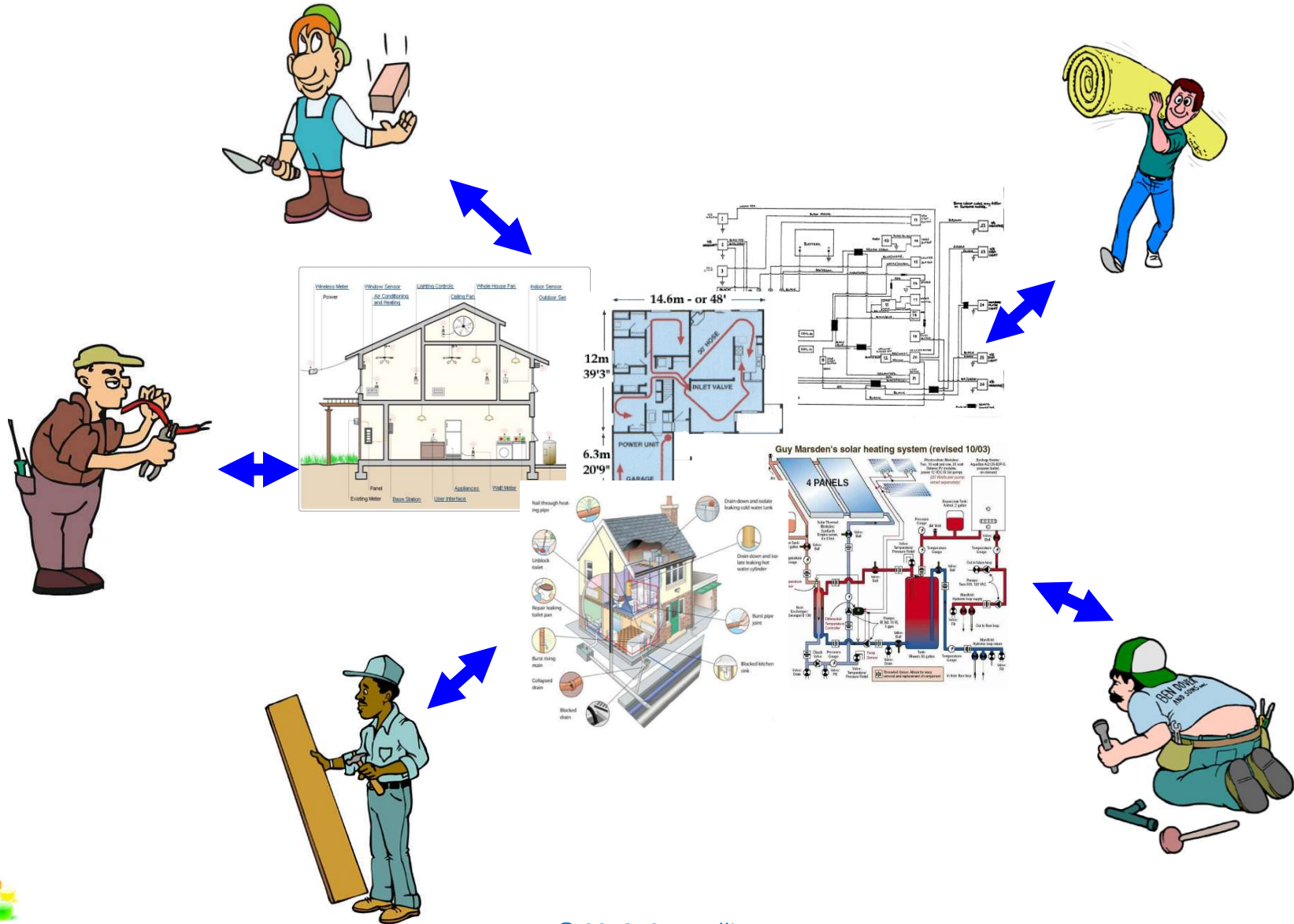
# Effectiveness of communication



# Effective communication



# Model and diagrams





# Advantages of a domain model

---

- Improves understanding
- Validates understanding
- Improves communication
- Shared glossary
- Improves discovery



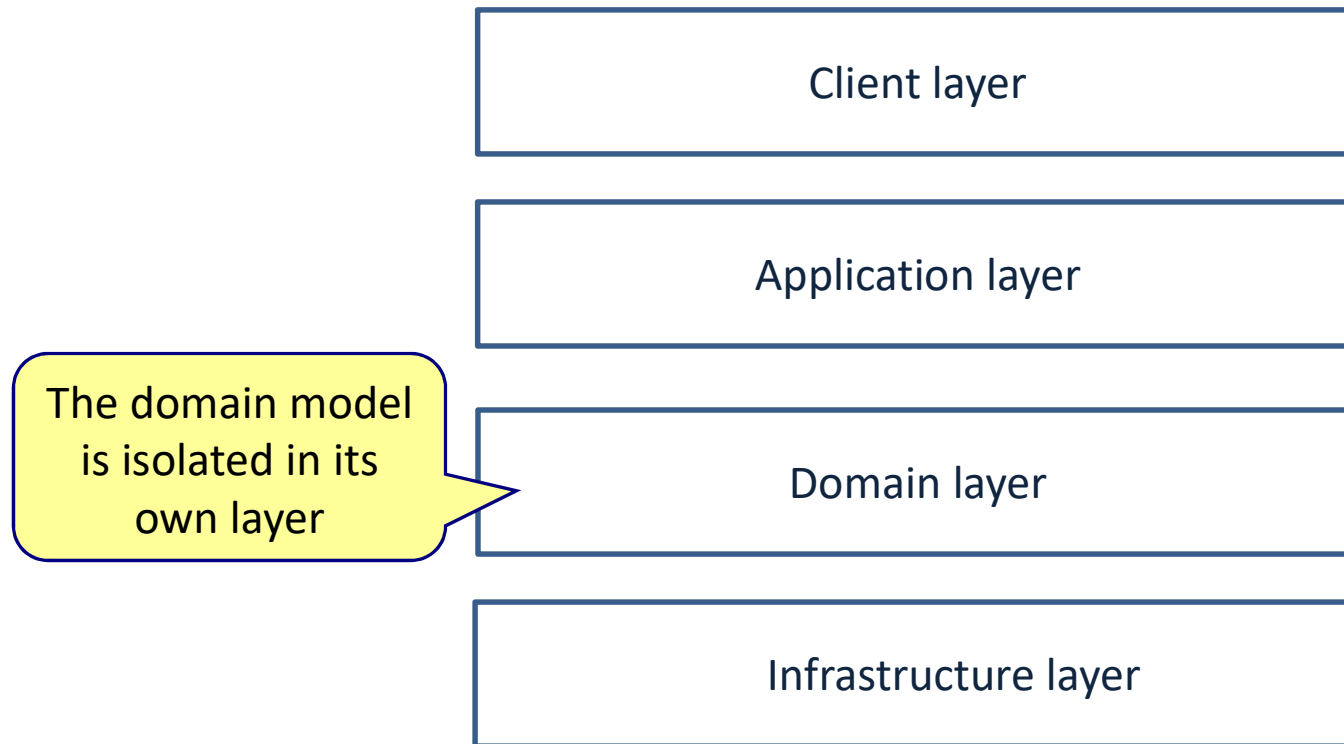
# Effective modeling

---

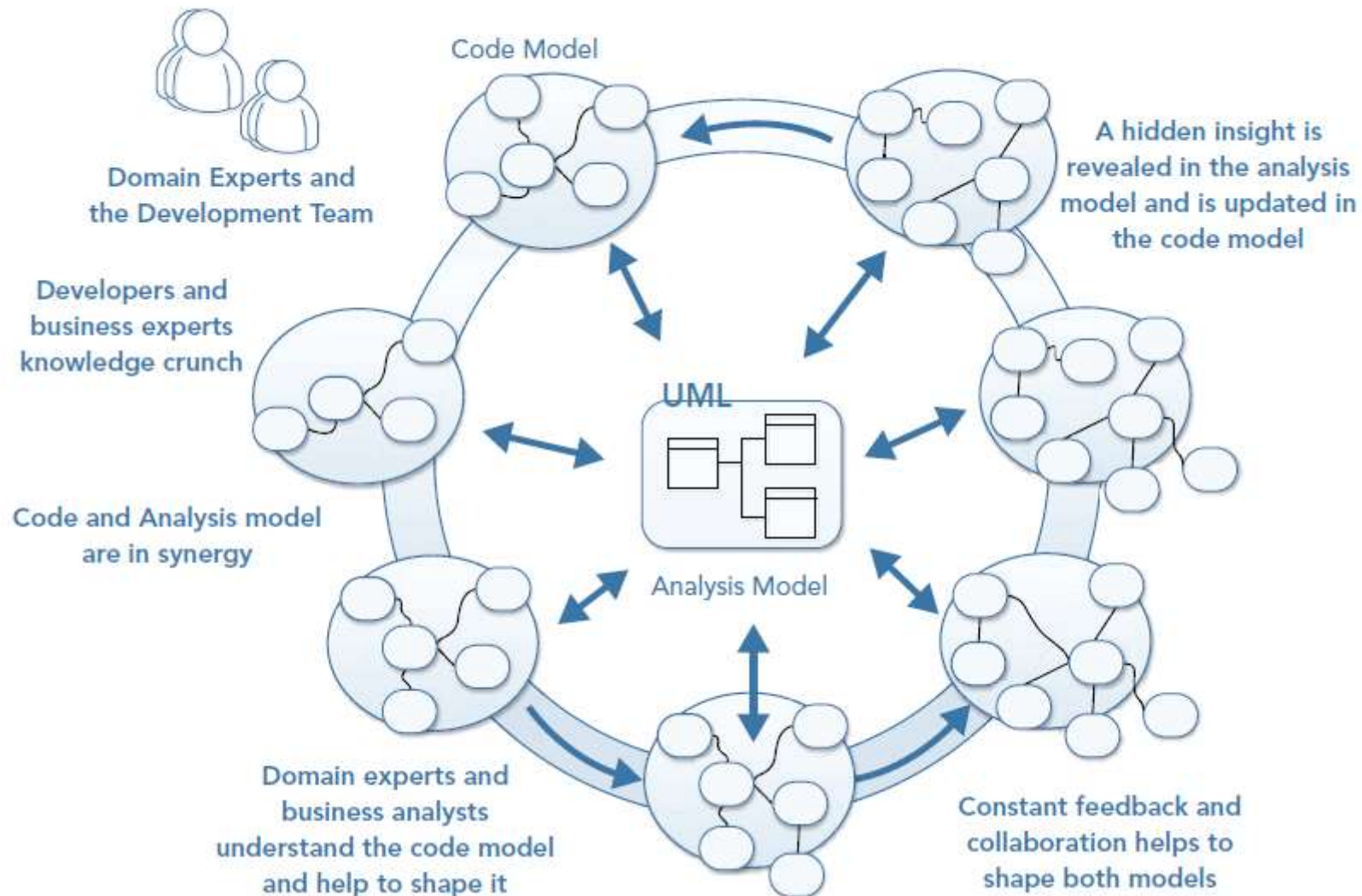
- Cultivate a language based on the model
- Use the format that fits the best
- Model light
  - Simple sketches
  - Focus on communication, not completeness
- Update the model
  - Add new concepts, but also remove concepts that are not useful or central anymore
- Use the model for brainstorming and experimenting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



# Isolate the domain model



# Keep the model and code in sync.



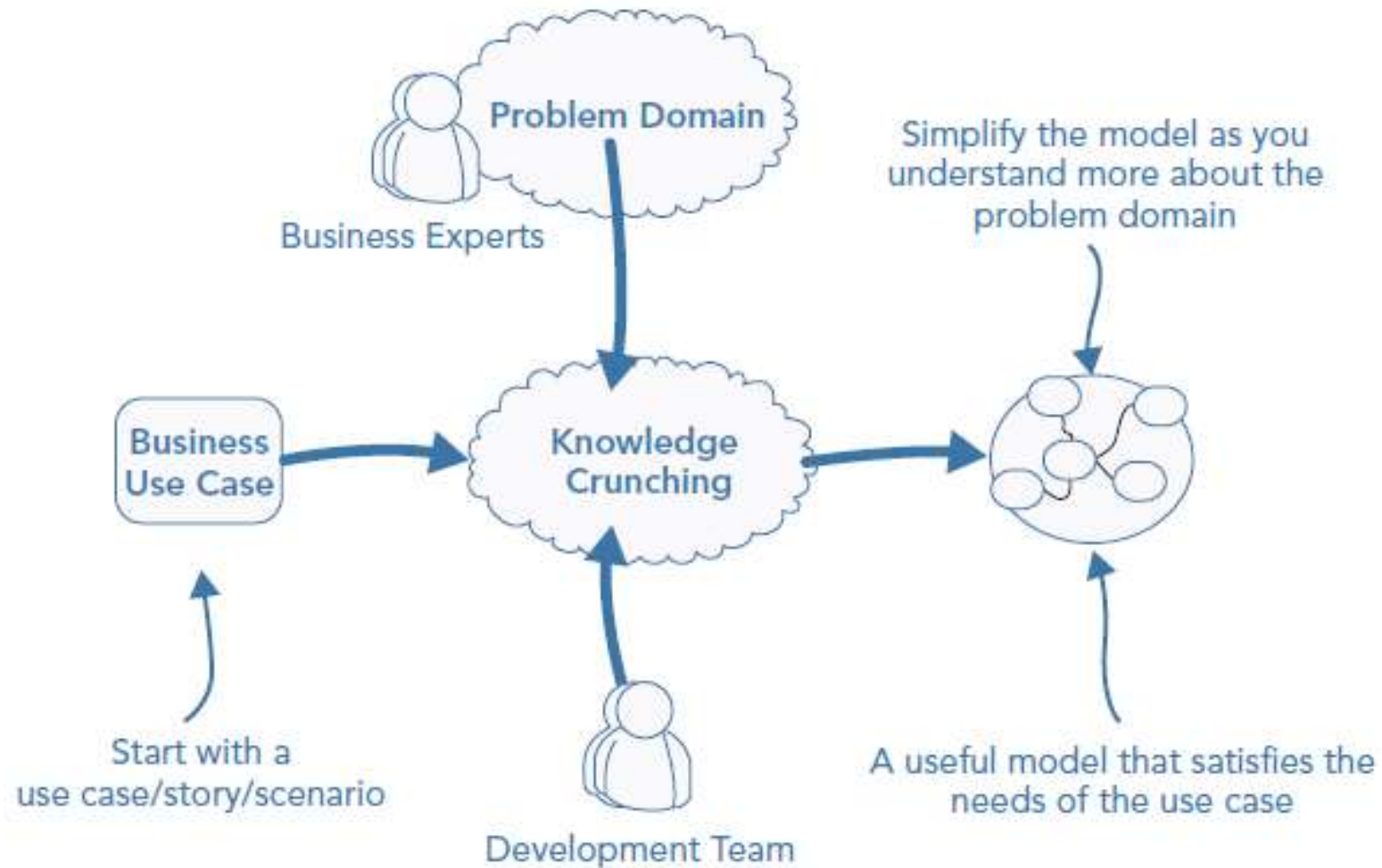
# The model should reflect the business

---

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



# Knowledge crunching



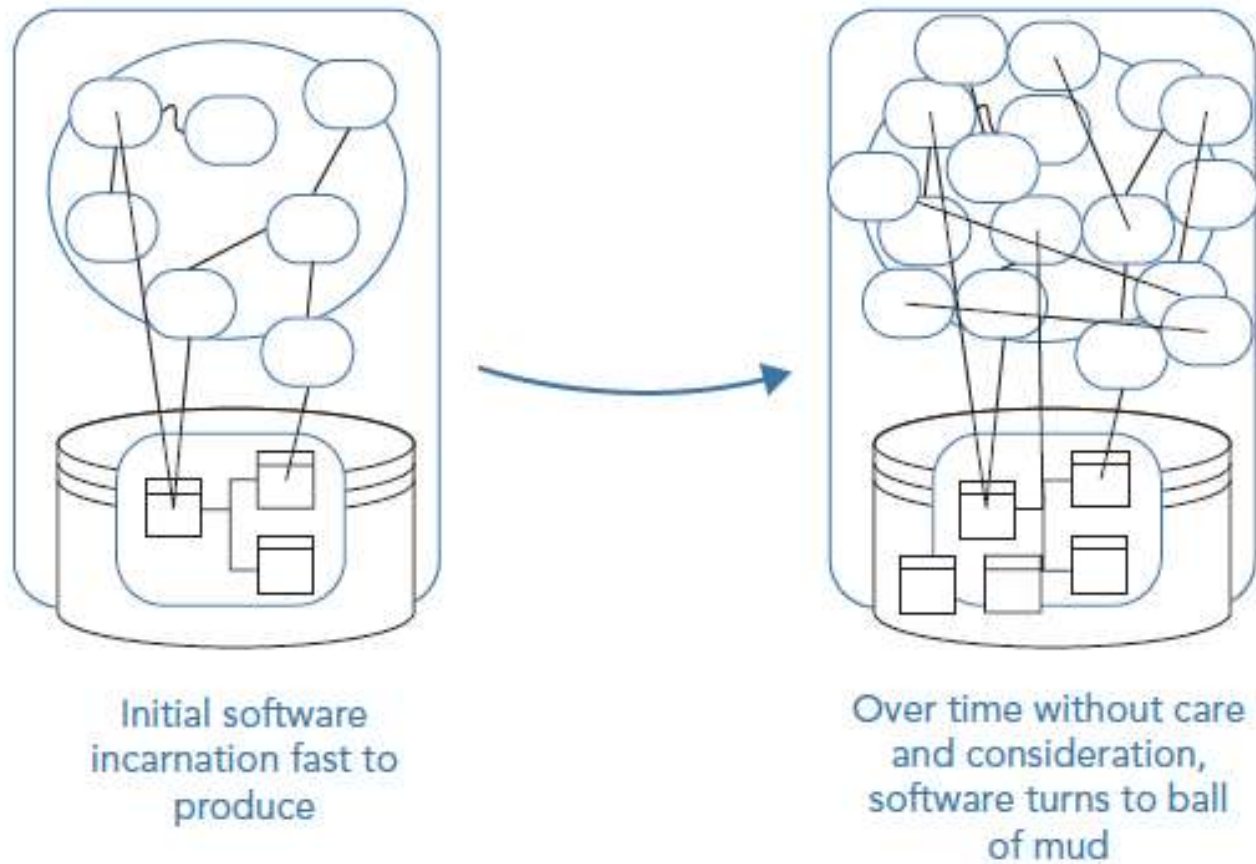
# Principles of Domain Driven Design

---

- 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





# 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



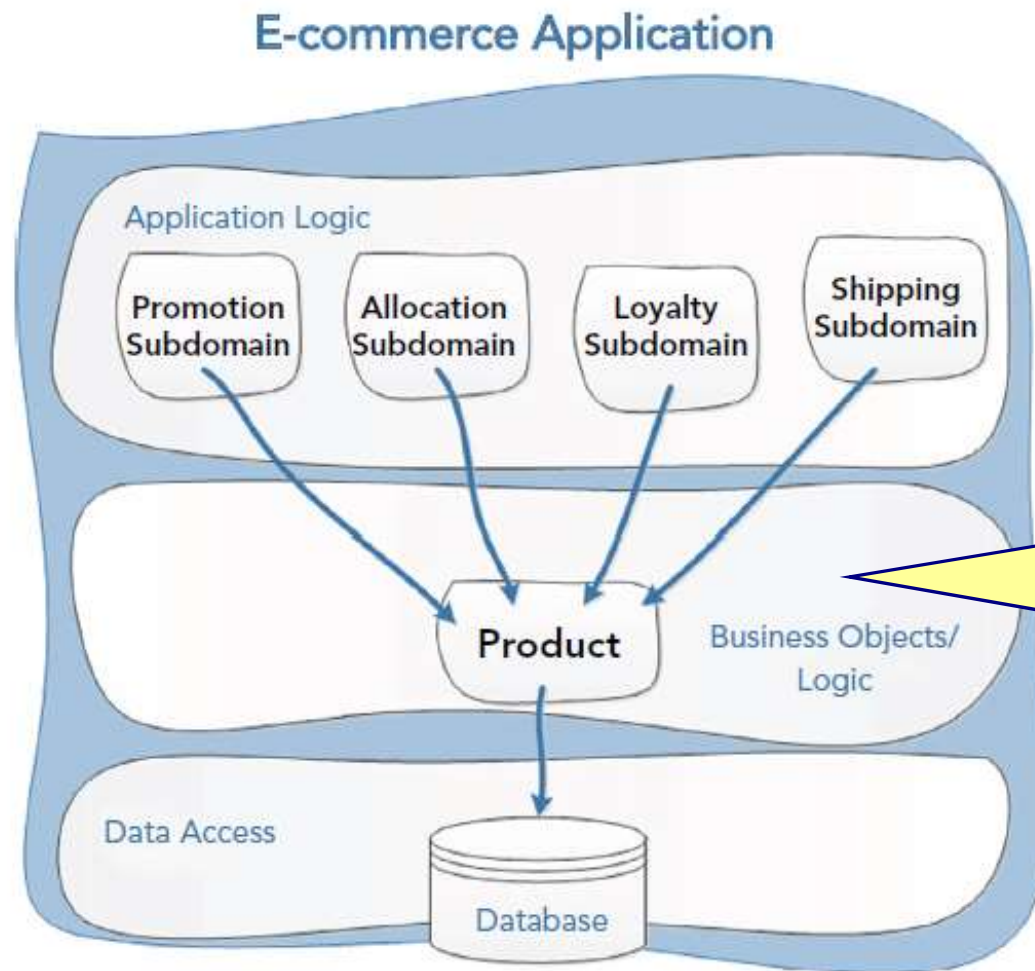
# Principles of Domain Driven Design

---

- 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



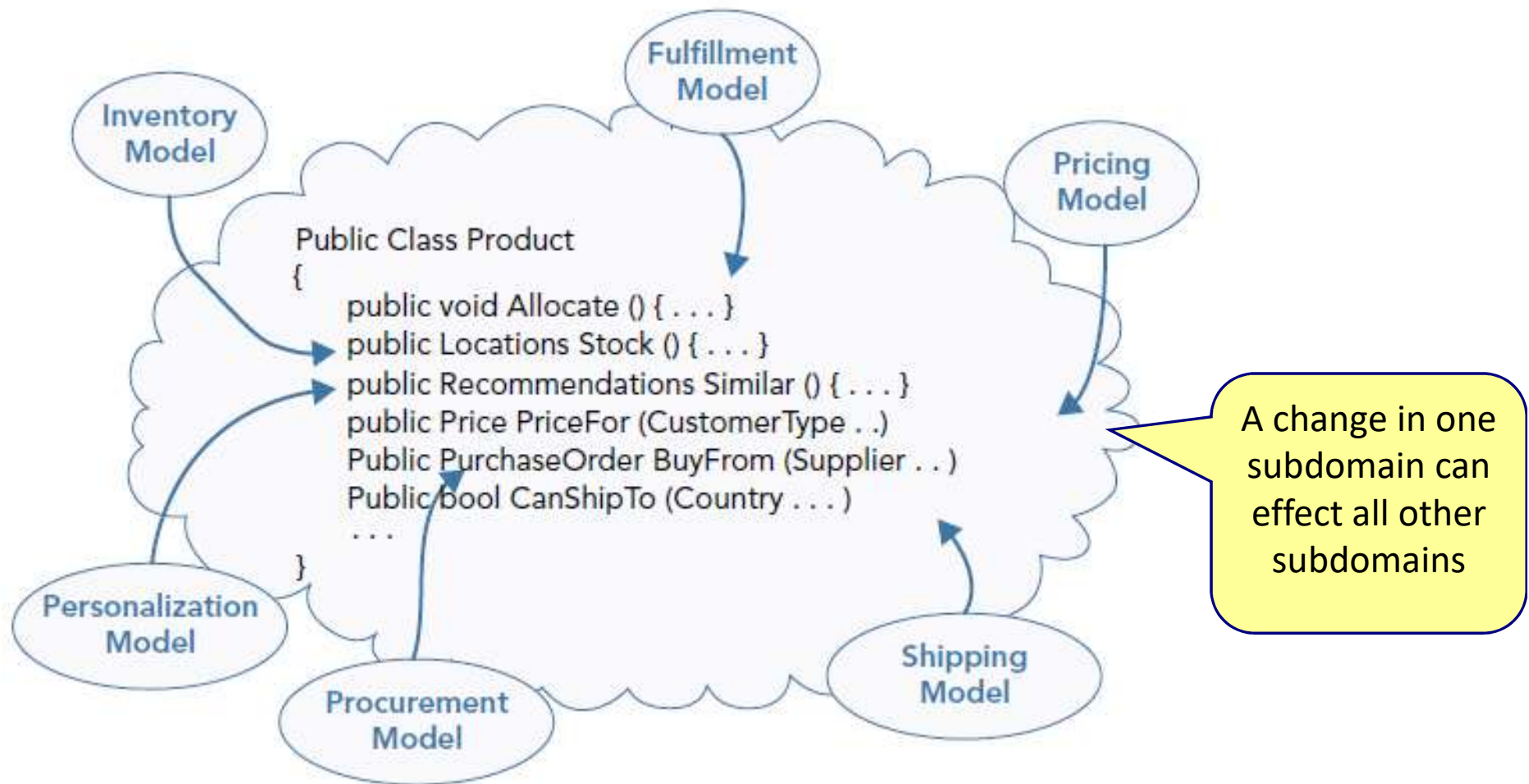
# Shared objects between subdomains



Single view of an entity for all subdomains can quickly become a problem

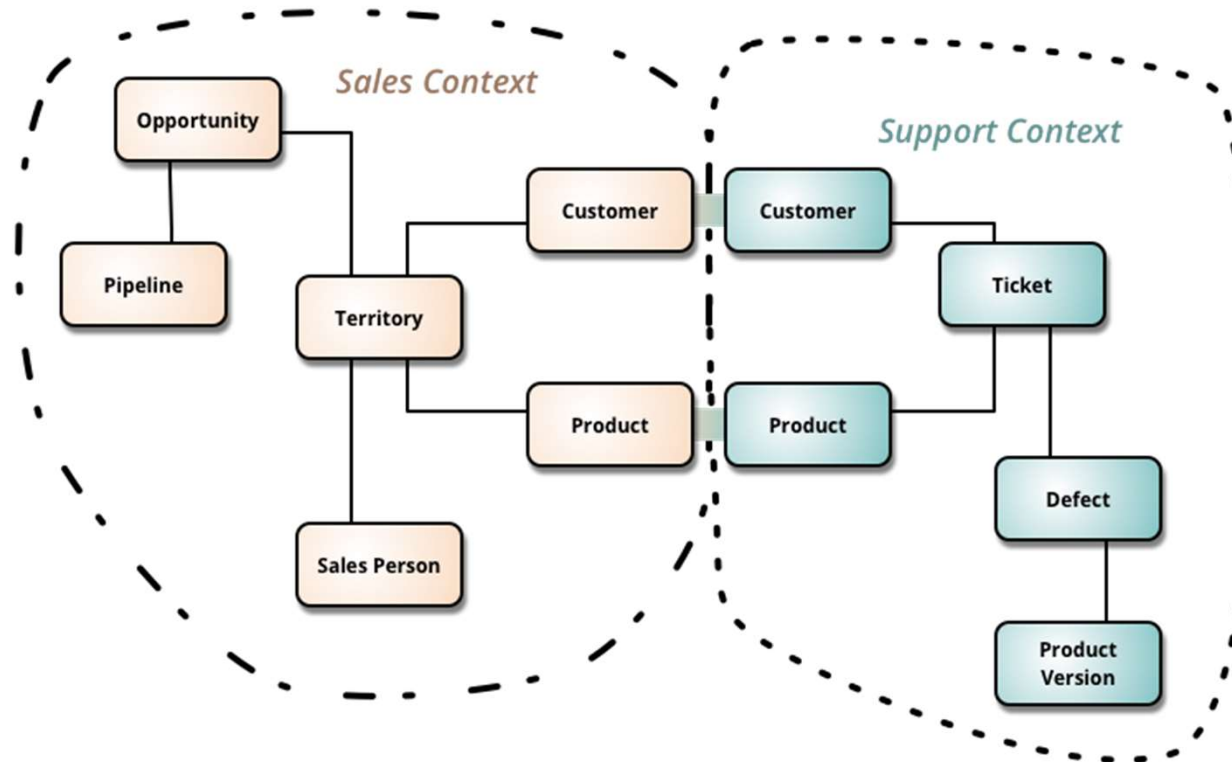


# Big ball of mud

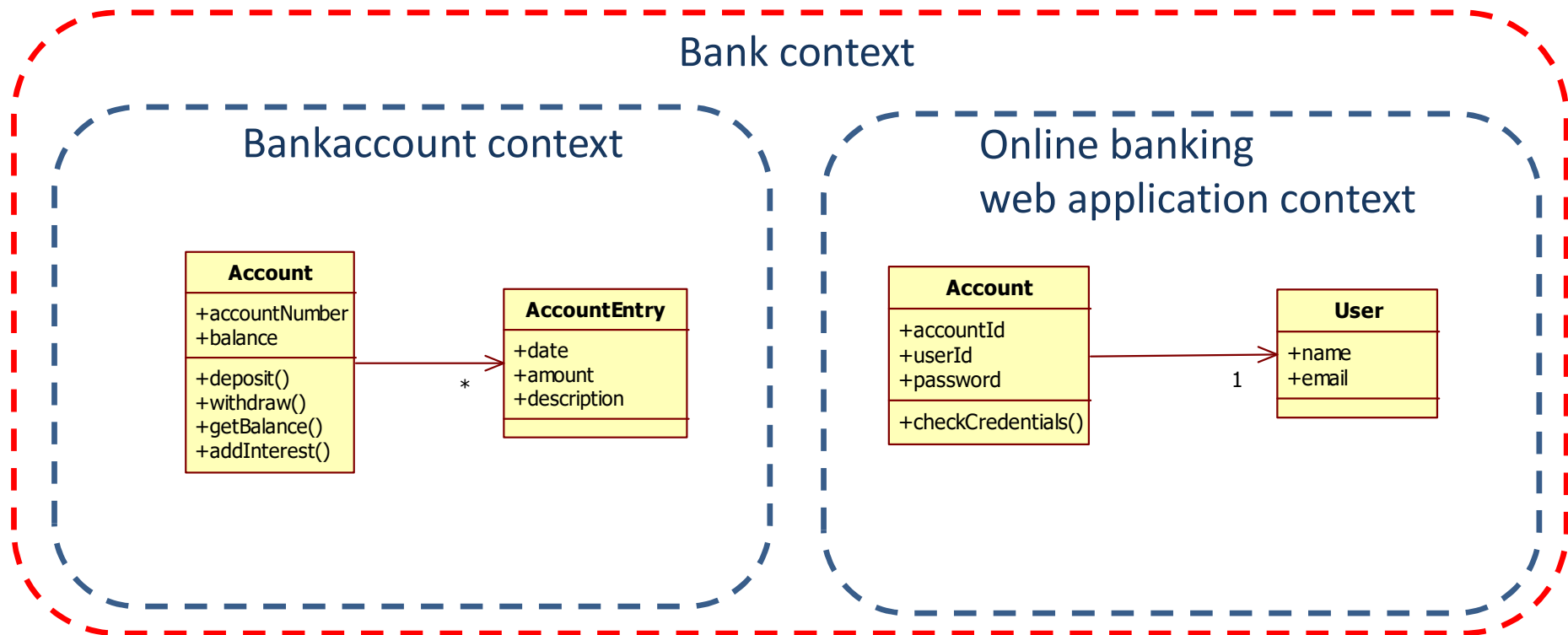


# Context

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



# Bounded context



# Bounded context

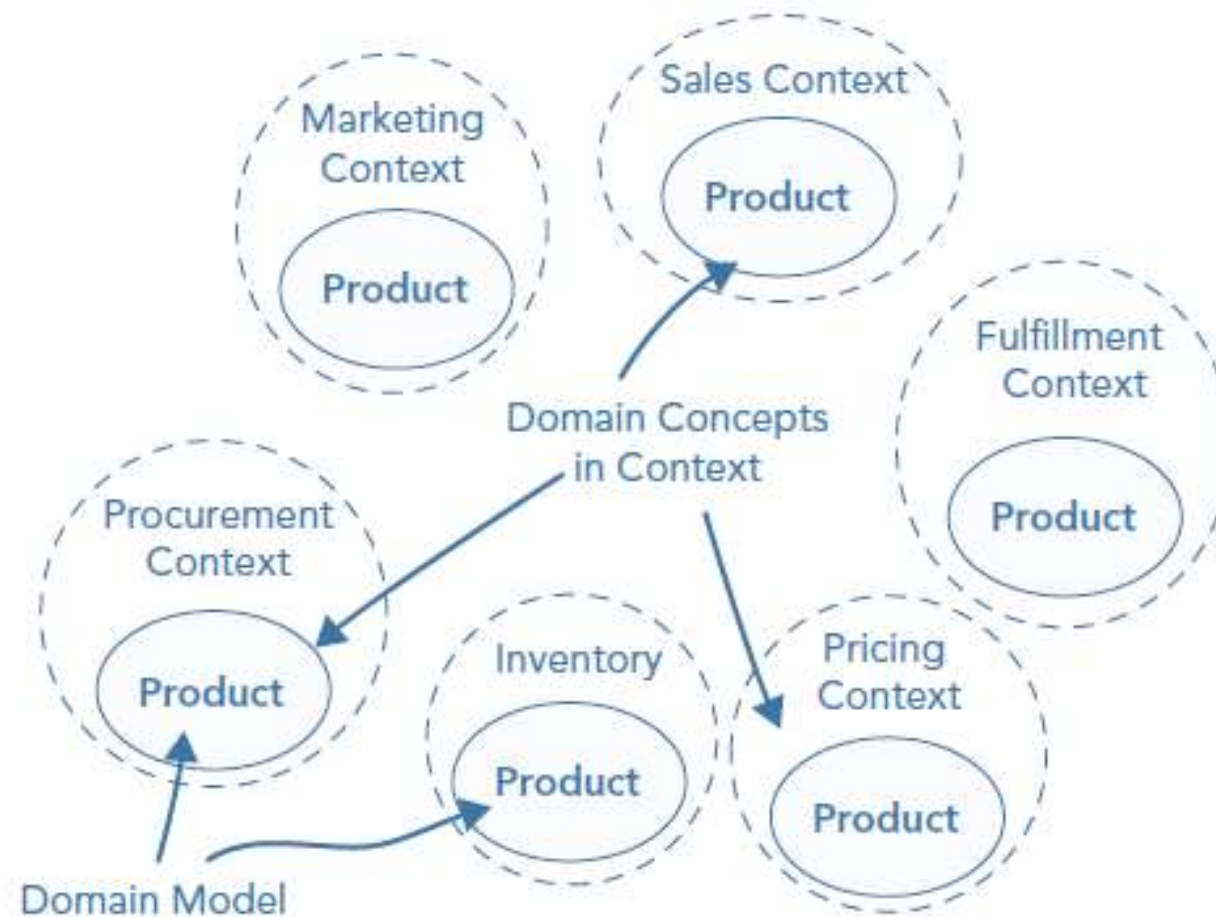
---

- 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



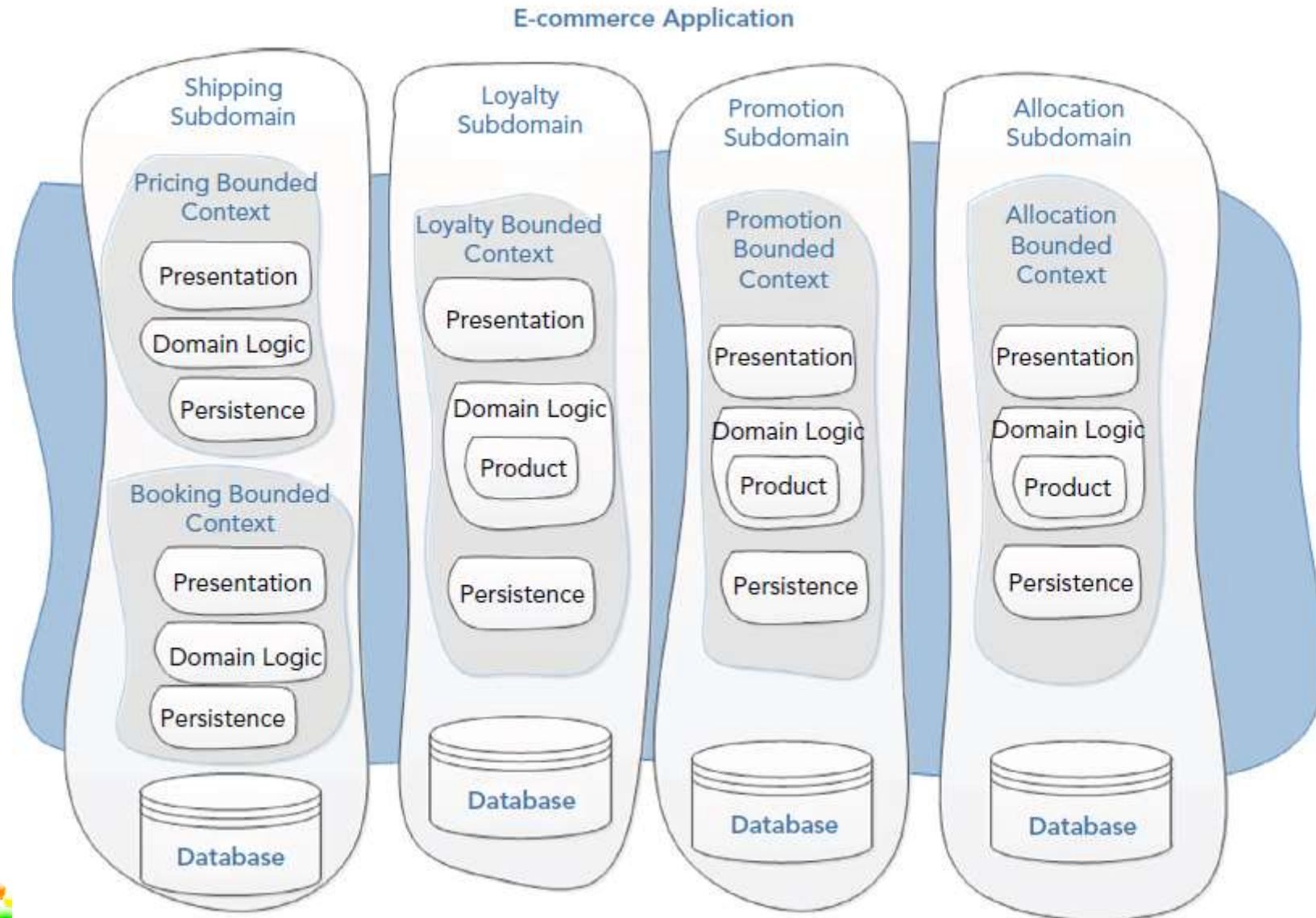
# Each bounded context has its own model

---

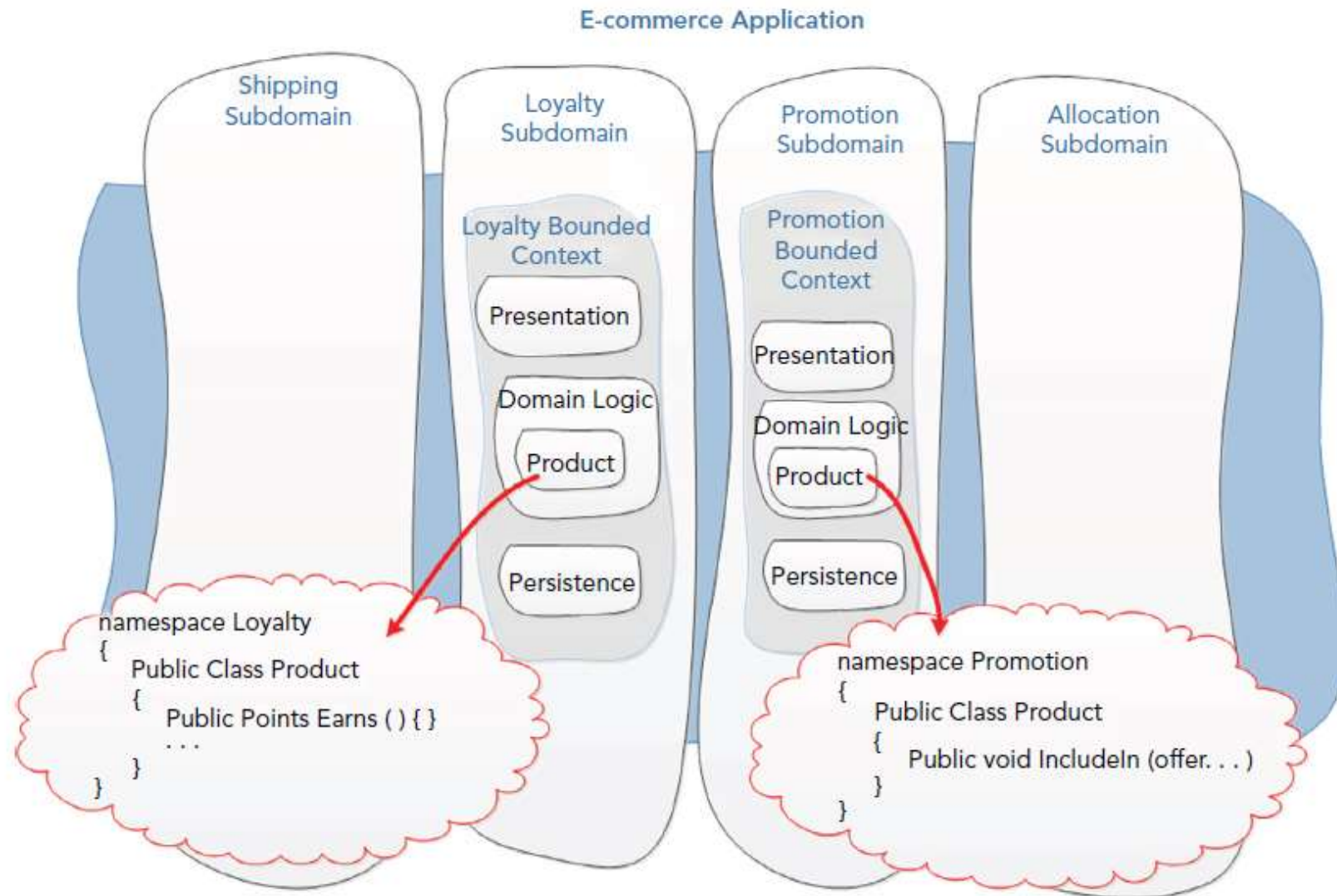




# Bounded context example



# 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

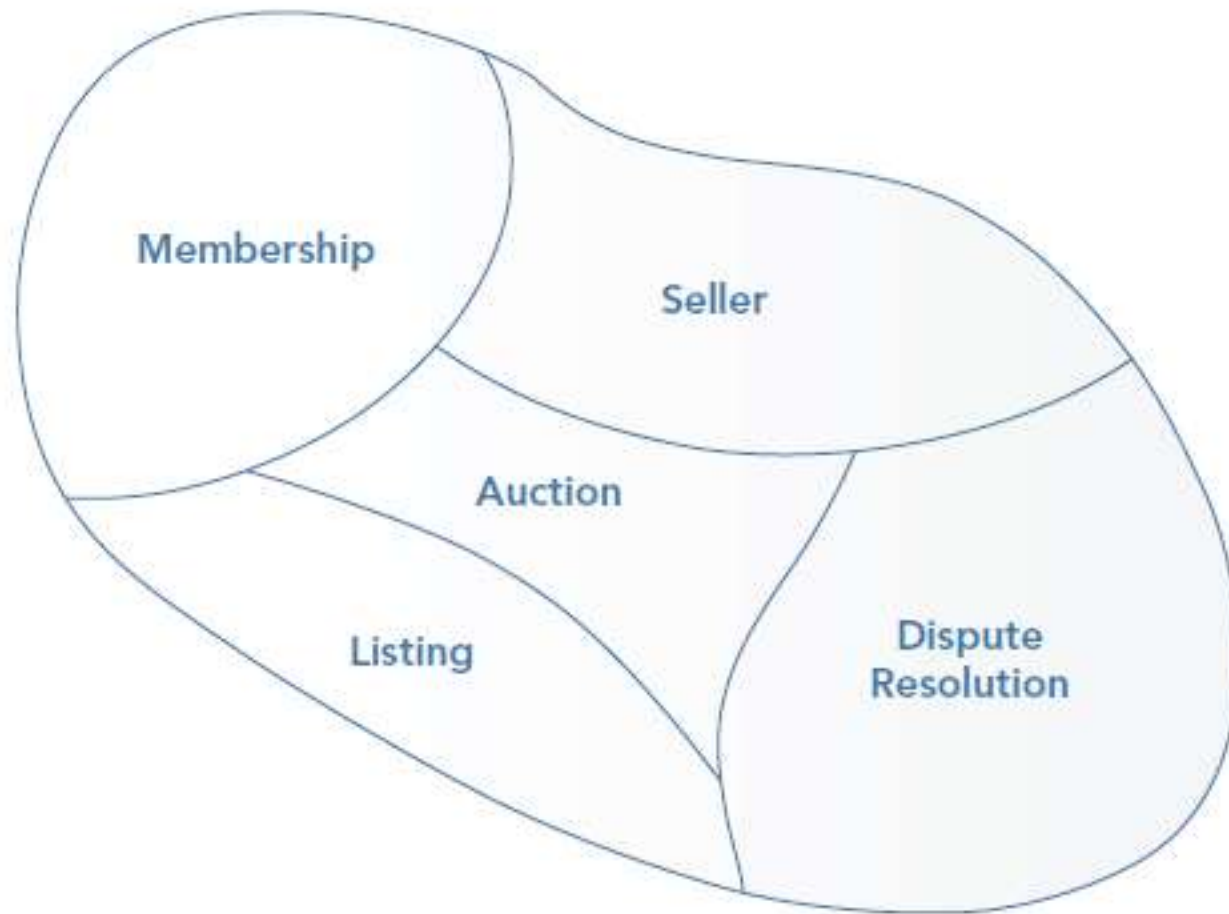


# Distilling the domain

- The large domain of online auction

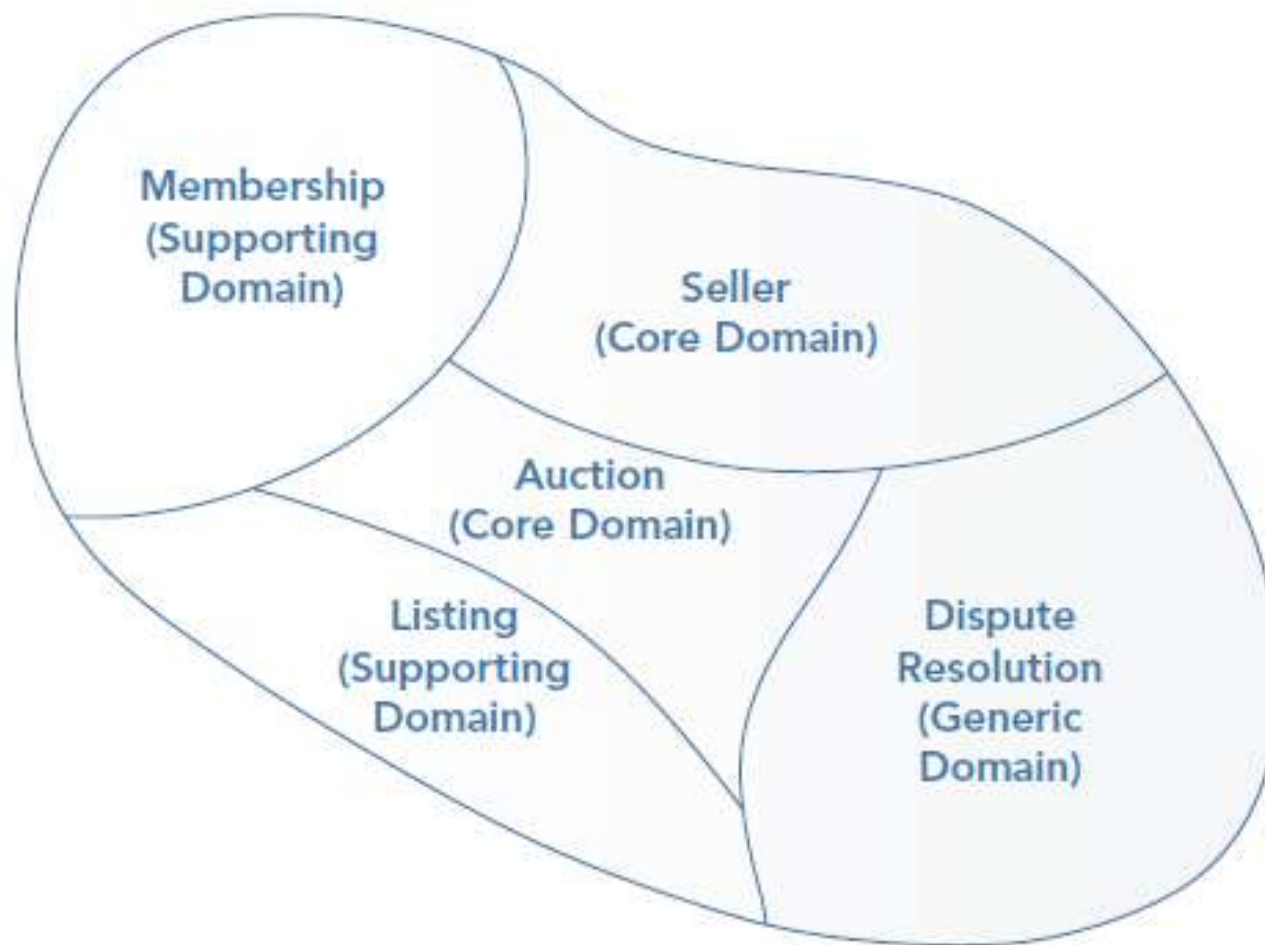


# Find the subdomains



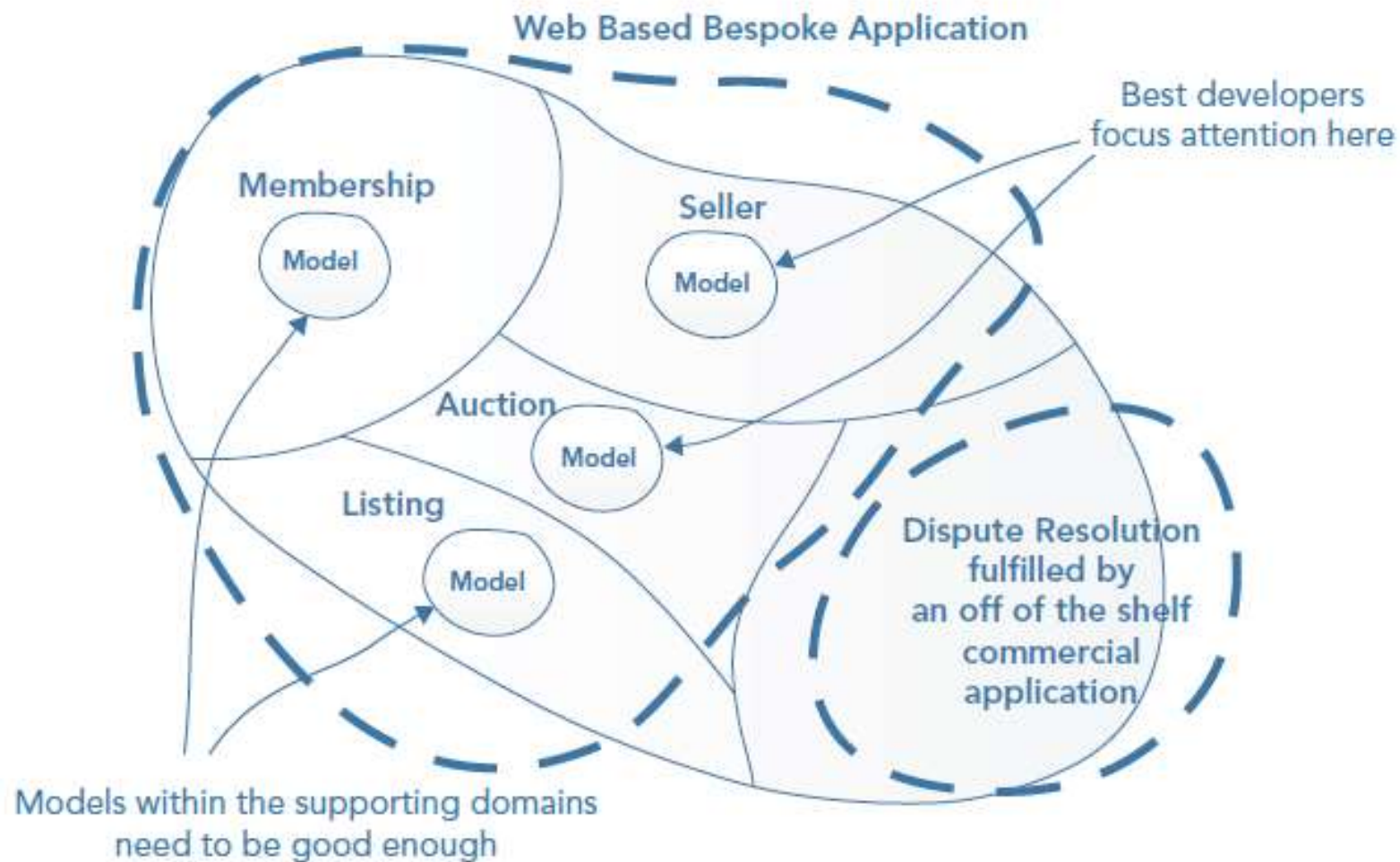
# Identify the core domain

---





# Subdomains shape the solution



# Why DDD?

---

- 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.”

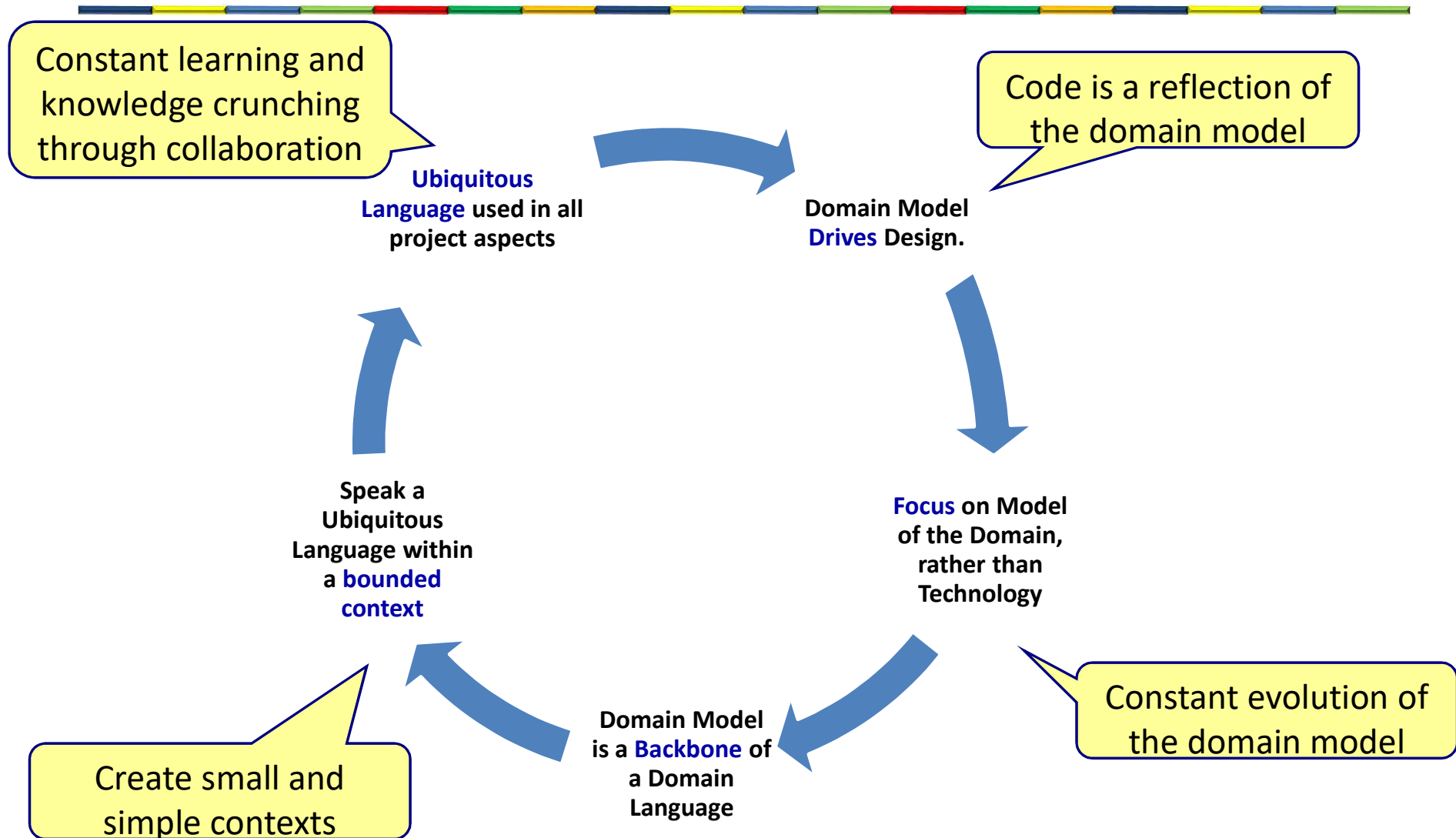
- Eric Evans

“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



# **WHEN TO USE DDD AND WHEN NOT?**



# Cost of DDD

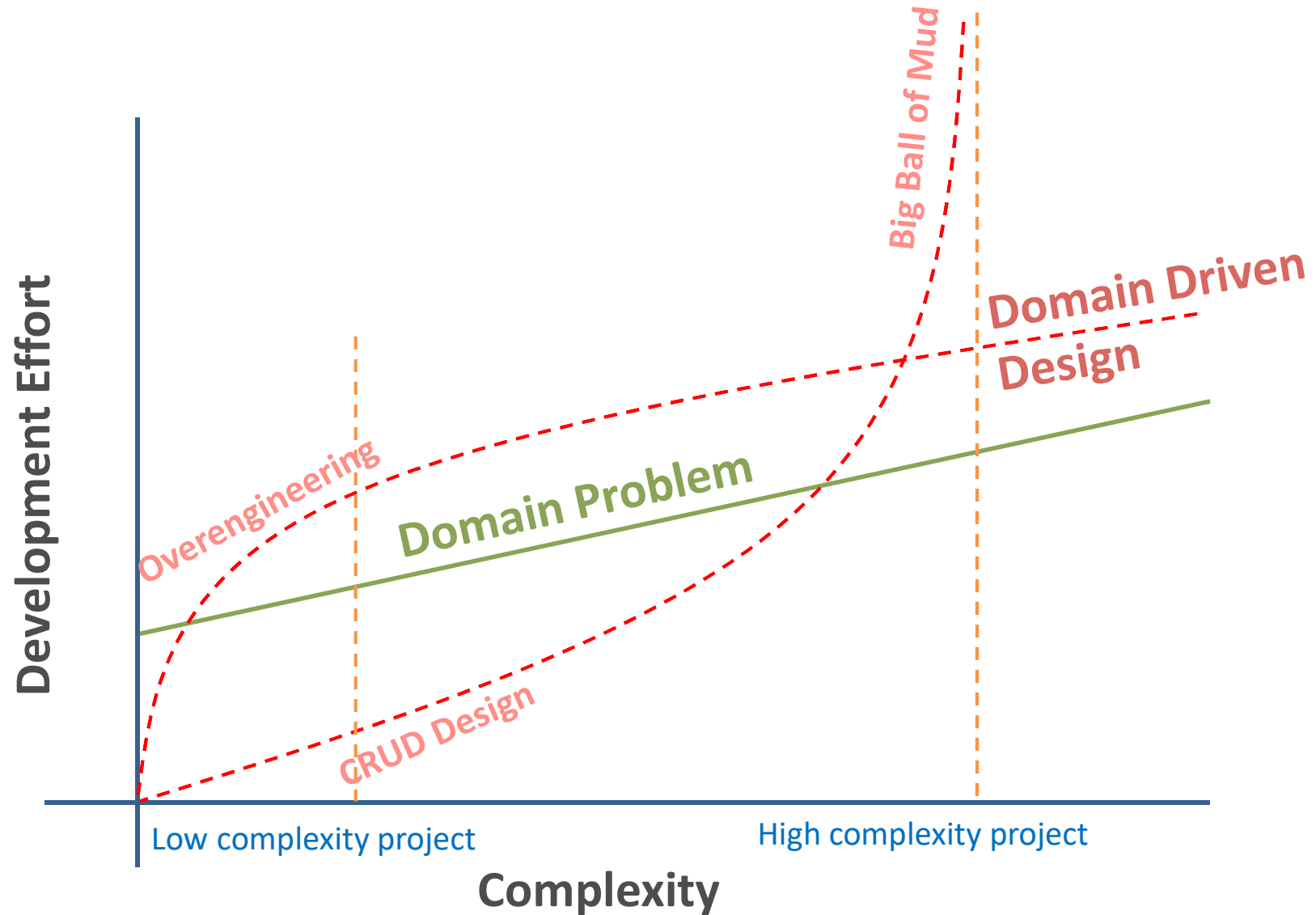
---

- 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?



# 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



# 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

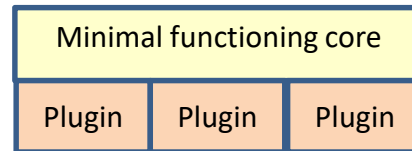
# ARCHITECTURAL STYLES



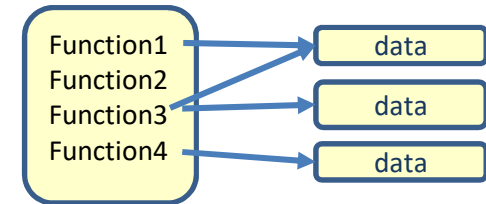
# Architecture styles



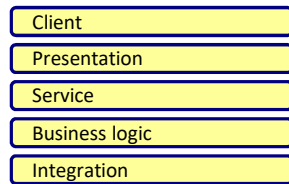
Client-server



Microkernel



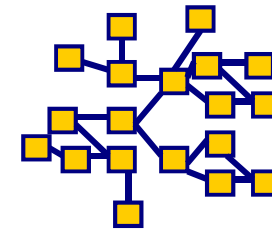
Procedural



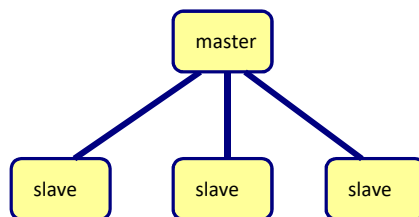
Layering



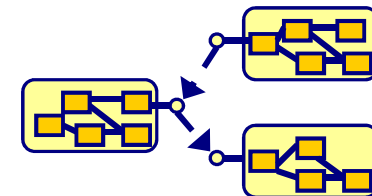
Pipe-and-Filter



Object oriented



Master-Slave



Component

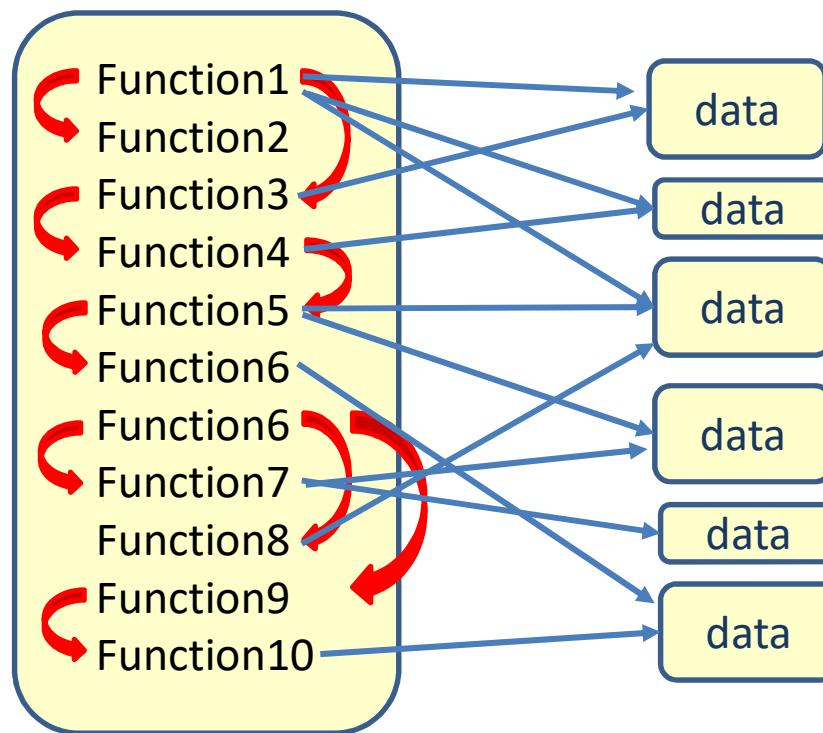


# PROCEDURAL PROGRAMMING



# 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

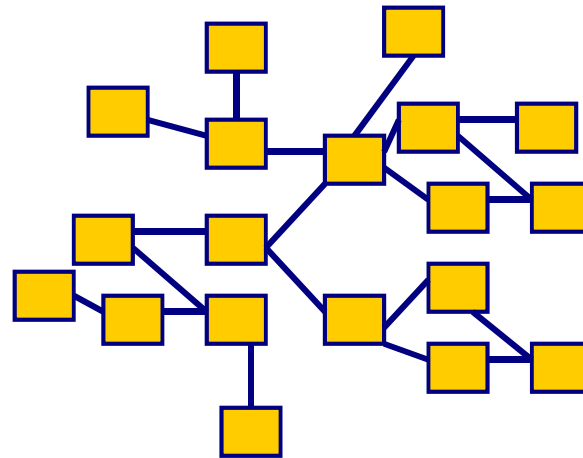


# OBJECT ORIENTED



# 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)



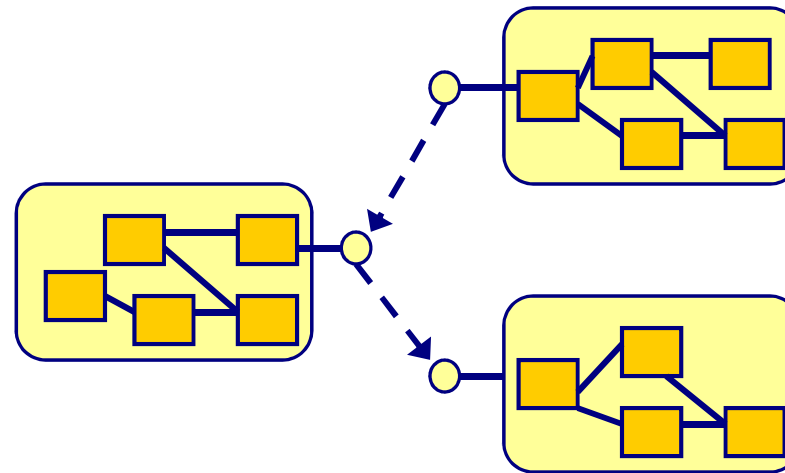


# COMPONENT BASED DEVELOPMENT



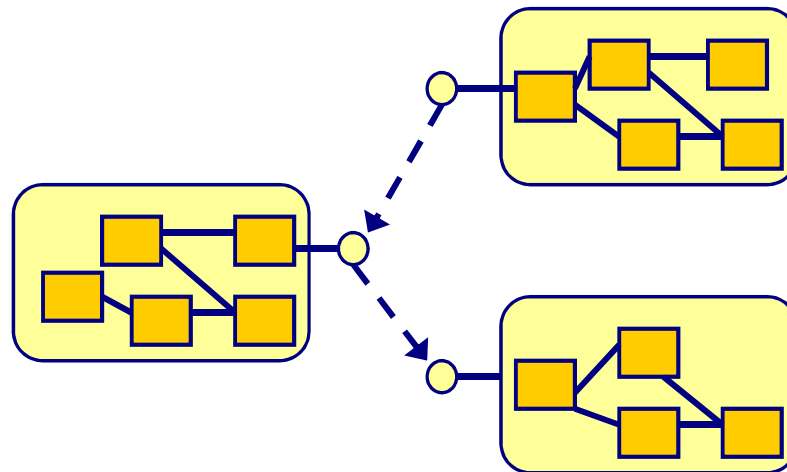
# Component Based Development

- Decompose the domain in functional components



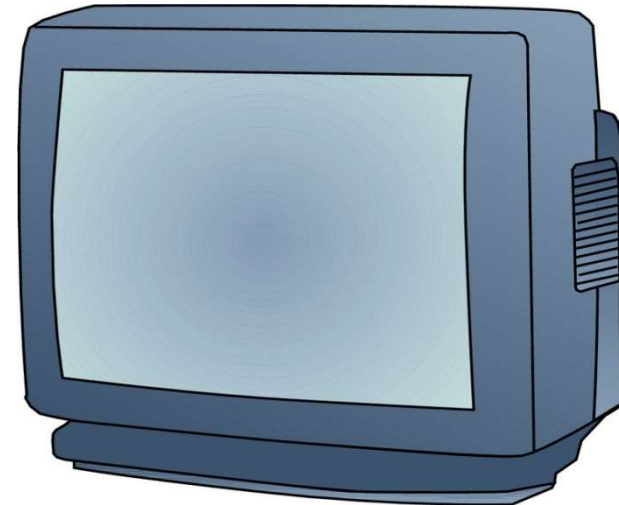
# What is a component?

- There is no definition
- What we agree upon:
  1. A component has an **interface**
  2. A component is **encapsulated**
- Plug-and-play
- A component can be a single unit of deployment



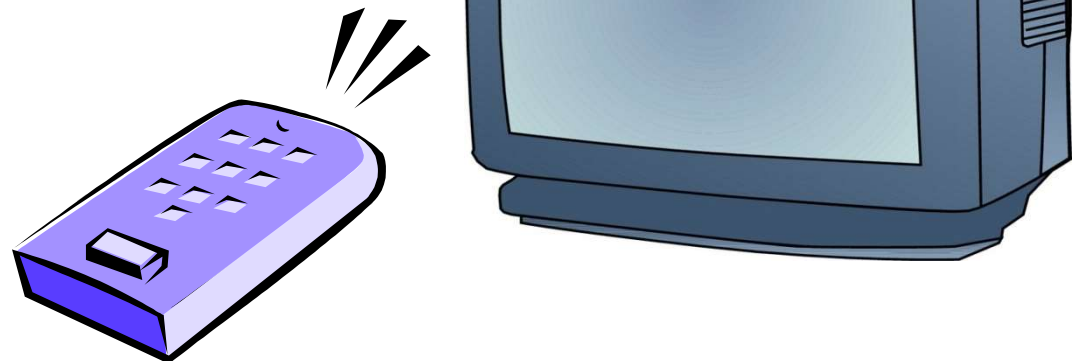
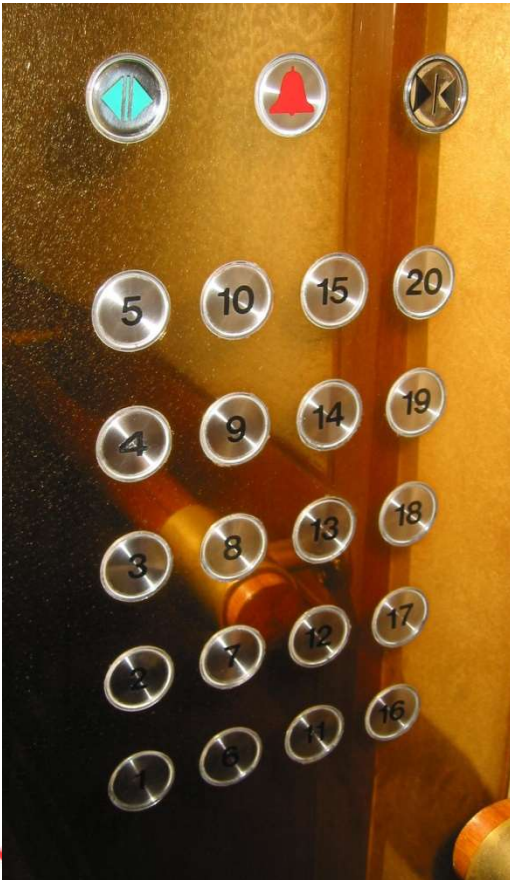
# Encapsulation

- The implementation details are hidden



# Interface

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





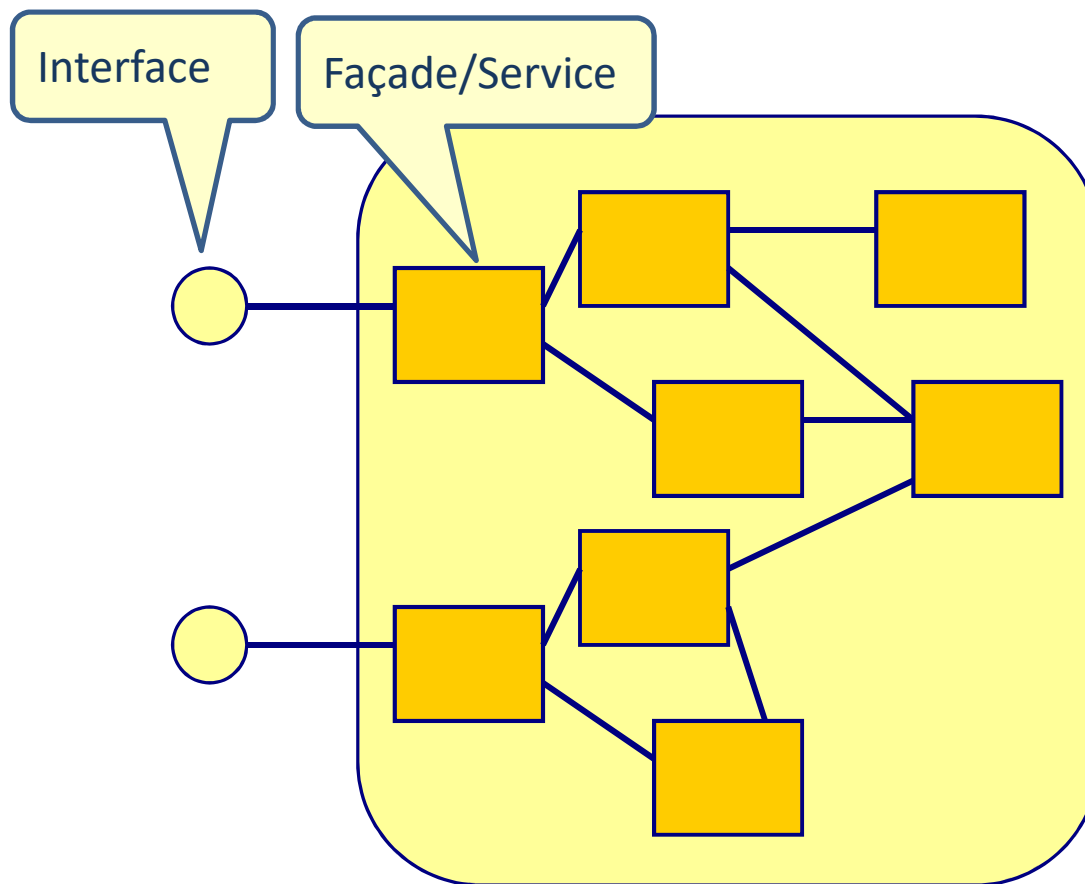
# Advantages of components

---

- High cohesion, low coupling
- Flexibility
- Reuse ?



# Internals of a component



# 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

```
public void update(Employee employee) {
    Connection conn = null;
    PreparedStatement prepareUpdateEmployee = null;
    try {
        conn = getConnection();
        conn.setAutoCommit(false);
        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();
        conn.commit();
    } catch (SQLException e) {
        conn.rollback();
        System.out.println("SQLException in EmployeeDAO update() :" + e);
    } finally {
        try {
            prepareUpdateEmployee.close();
            closeConnection(conn);
        } catch (SQLException e1) {
            System.out.println("Exception in closing jdbc connection in EmployeeDAO" + e);
        }
    }
}
```

Open connection

Start transaction

Send the SQL

Commit transaction

Rollback transaction

Exception handling

Close connection



# JDBC Template client

---

```
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

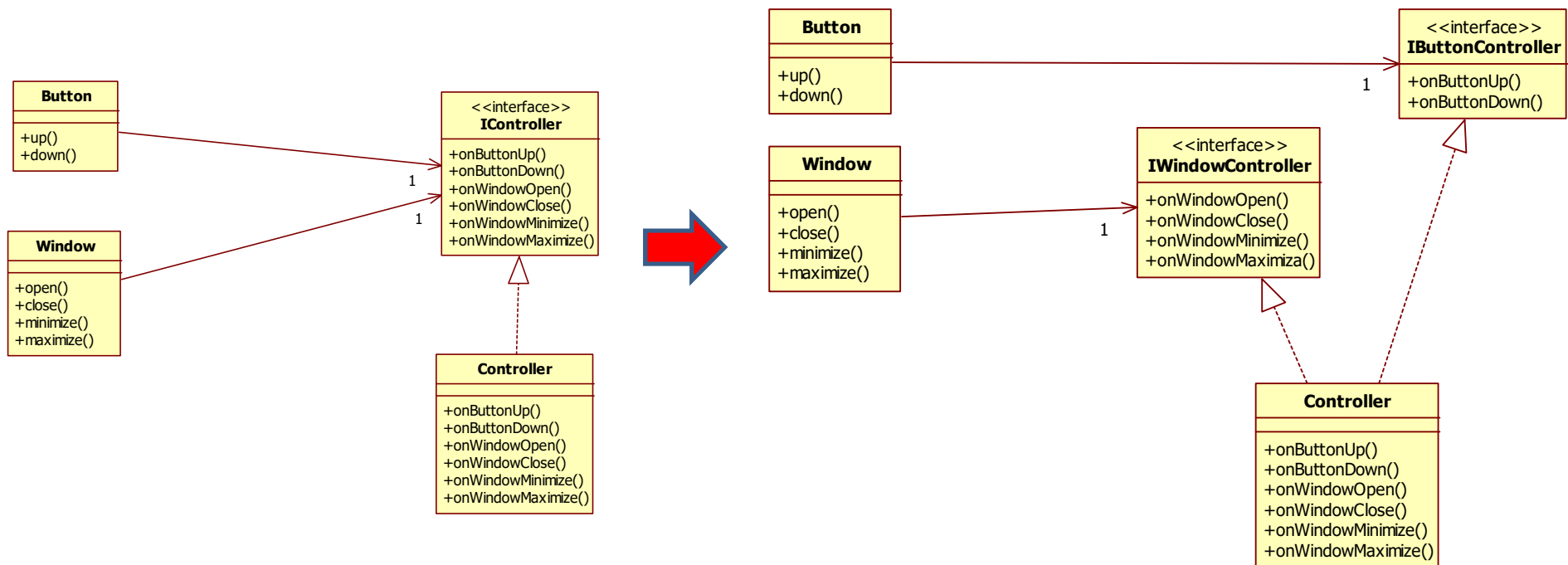
---

- Start with the client first
- Single responsibility principle
- Interface segregation principle
- Easy to use
- Easy to learn



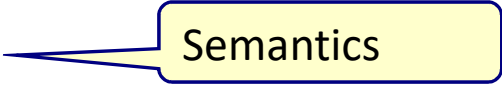
# Interface segregation principle

- 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 
  - 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

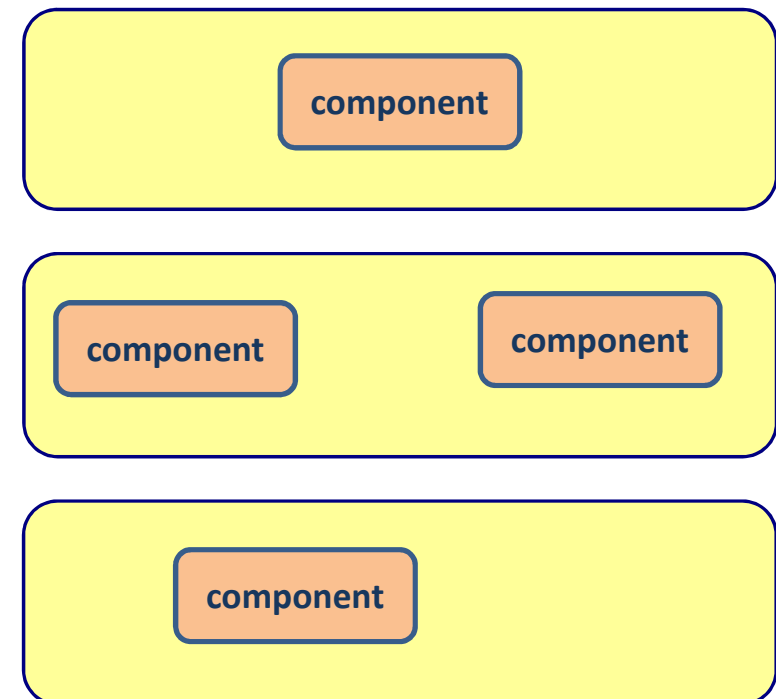
`getCurrentValue()`



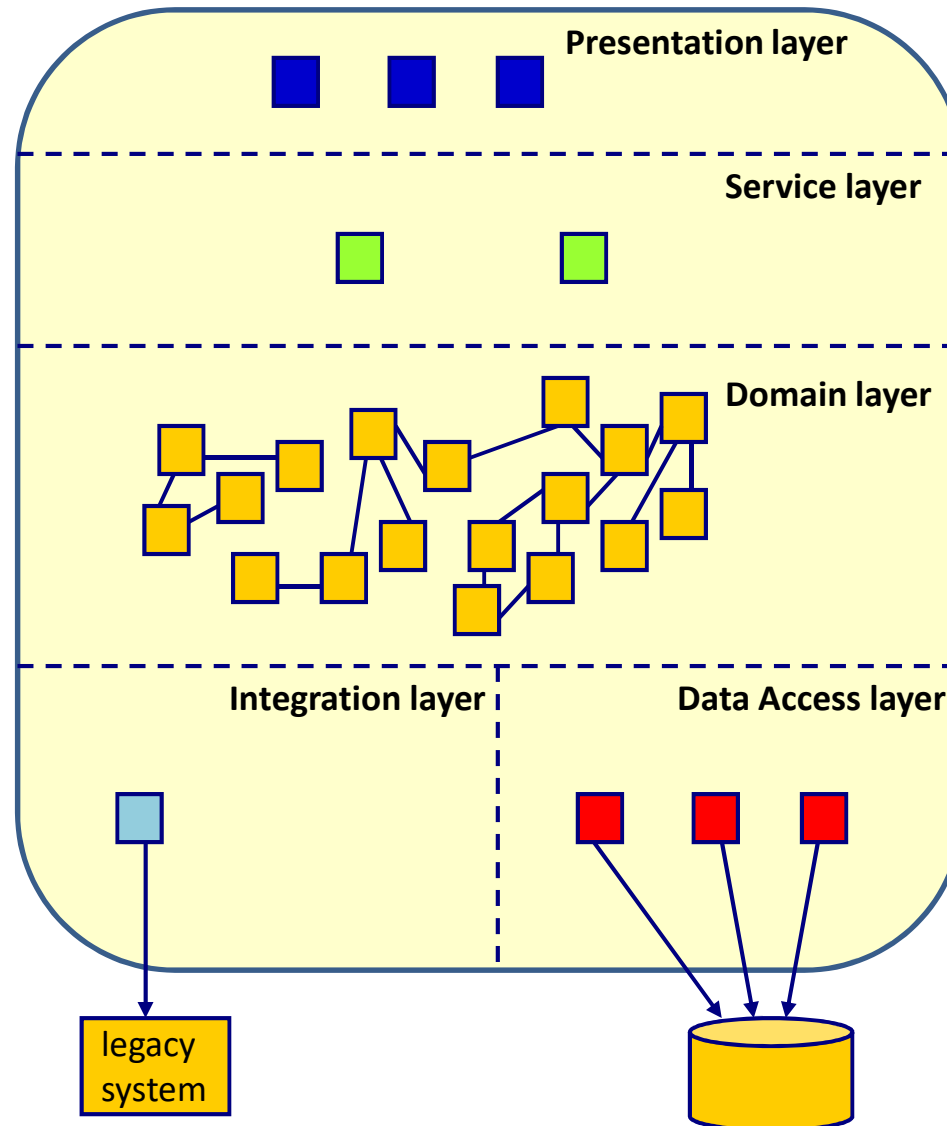
# Components in practice

---

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



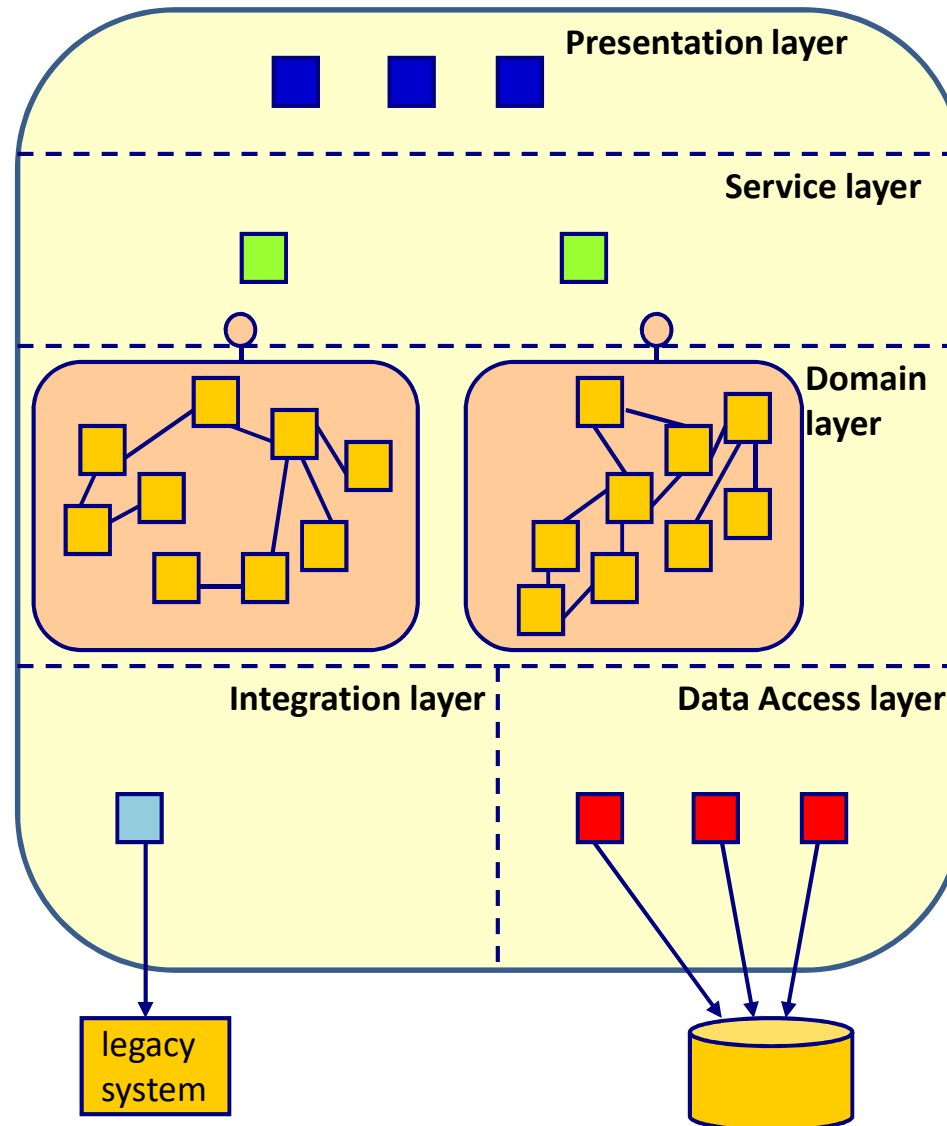
# Object-Oriented Design



- Everything is connected with everything else



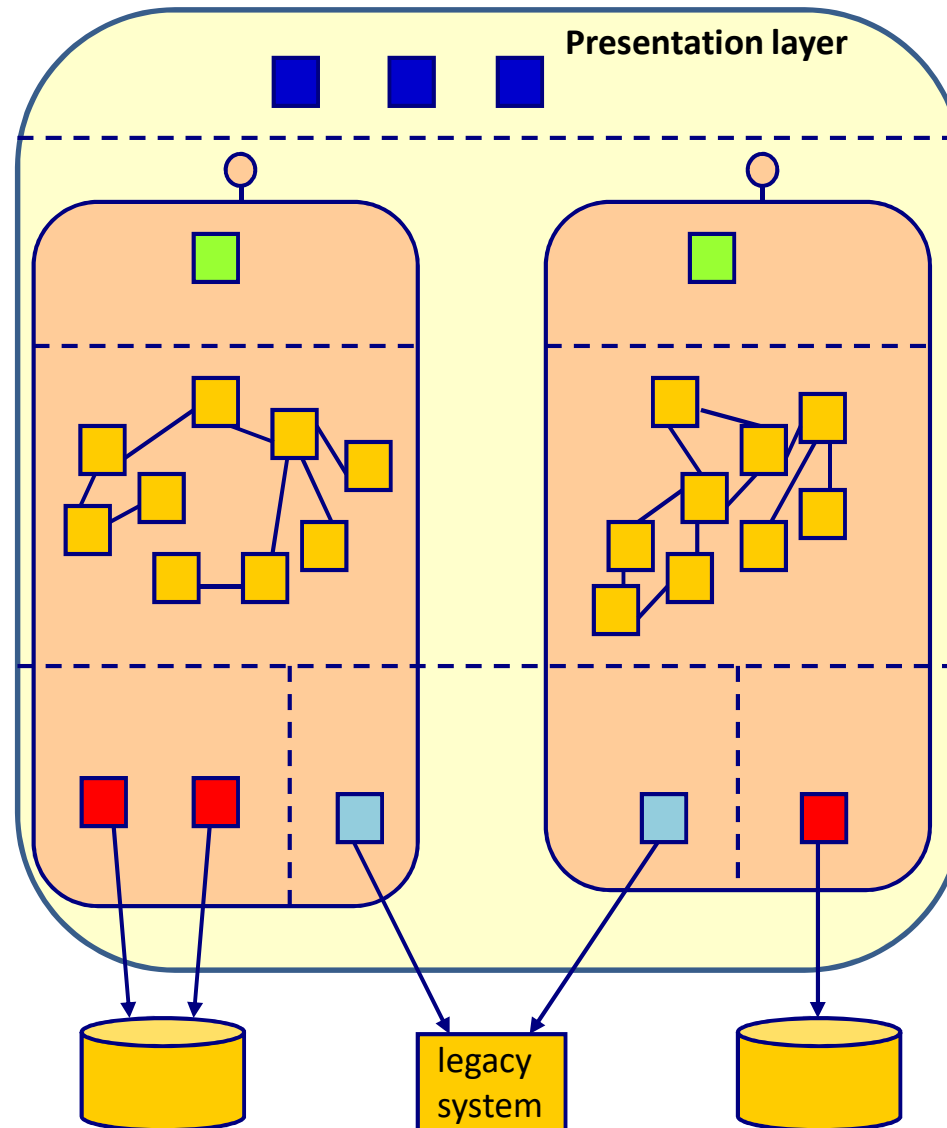
# Component Based Design (CBD)



- Better separation of concern
- More flexibility
- Possibility for reuse



# Service/Component Based Design



- Better separation of autonomous services
- Easier to build the services in isolation by different projects
- More flexibility
- Possibility for reuse



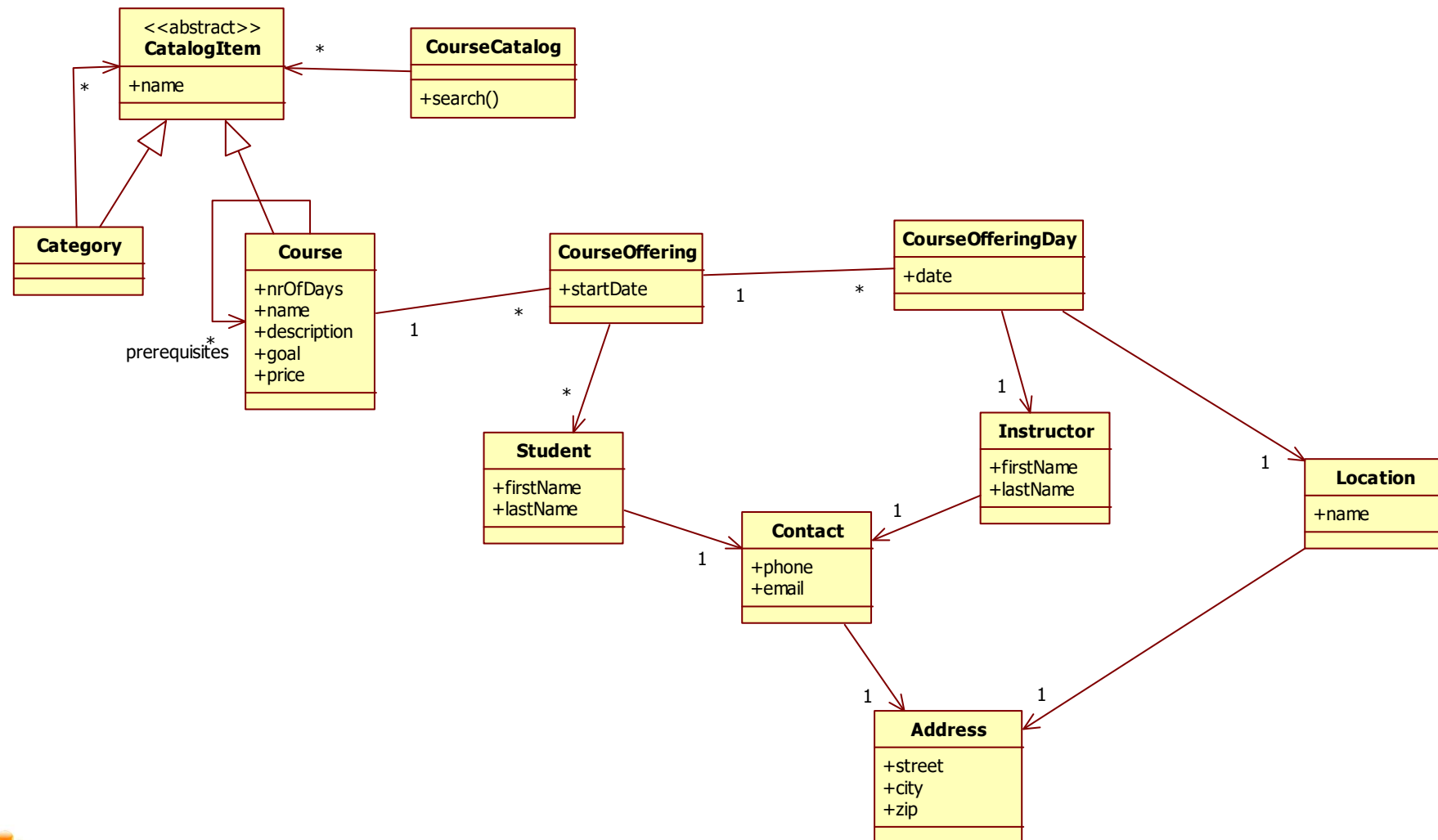
# Main point

---

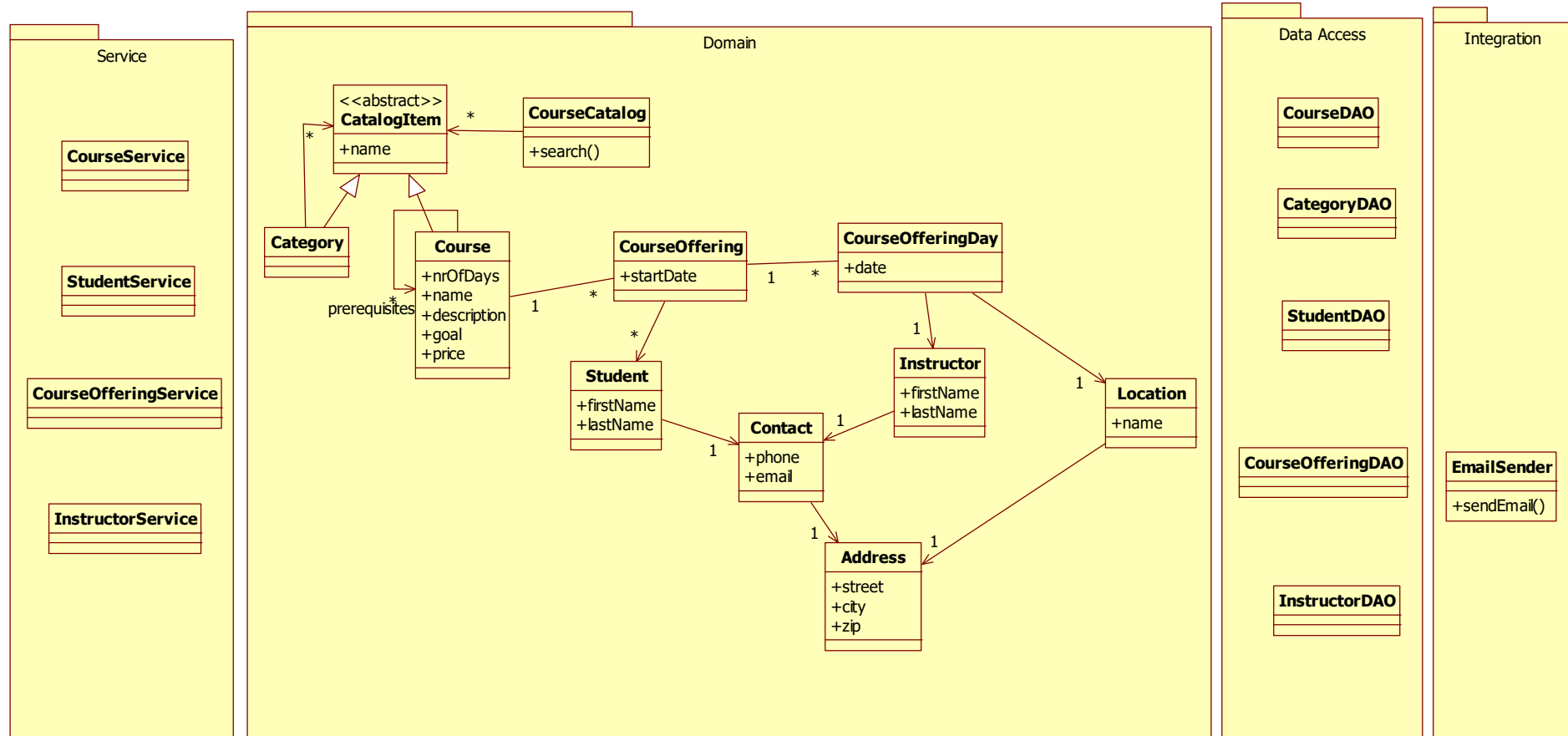
- Components are encapsulated and completely autonomous plug-and-play elements.
- The human nervous system is capable to transcend to that abstract field of pure consciousness which lies at the basis of the whole creation.



# Course Registration Domain Model

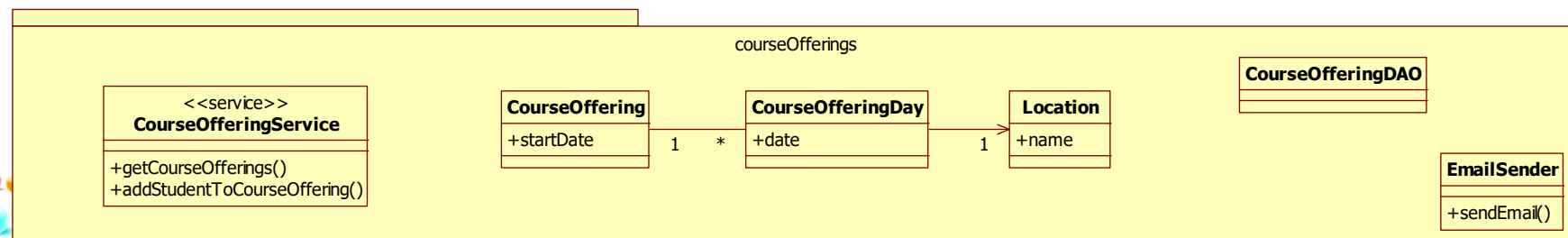
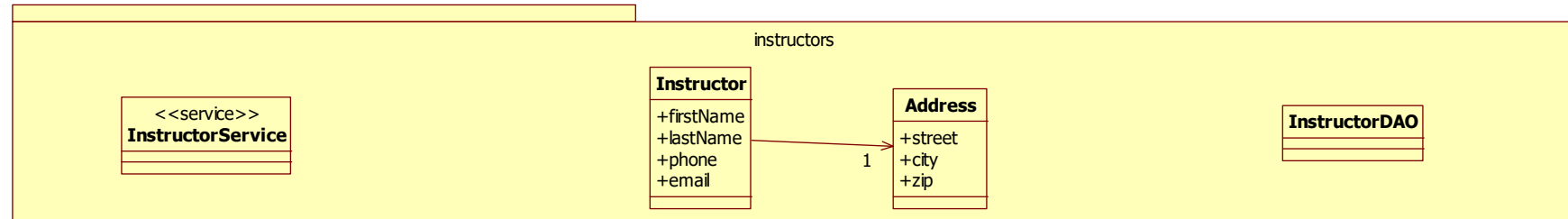
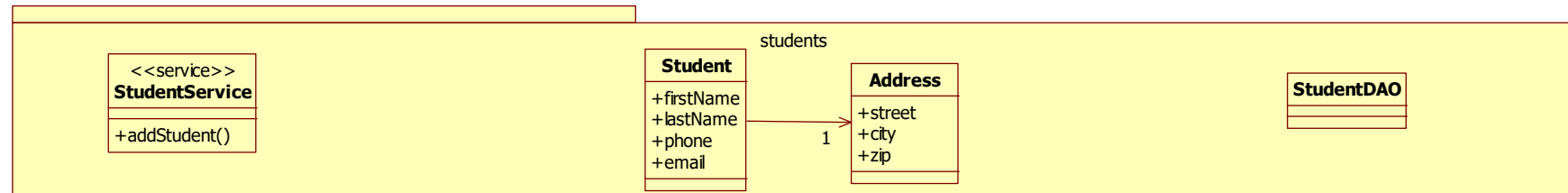
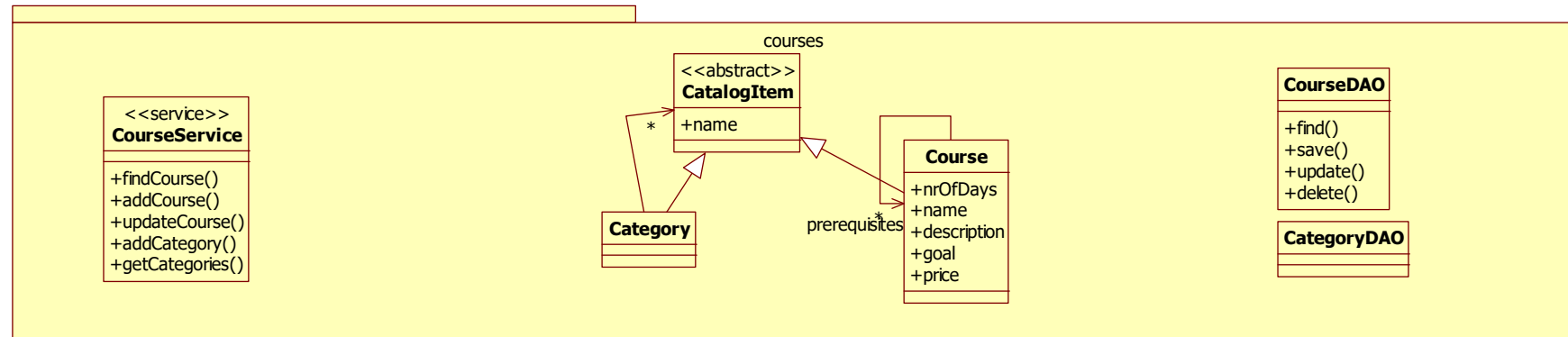


# Course Registration Design

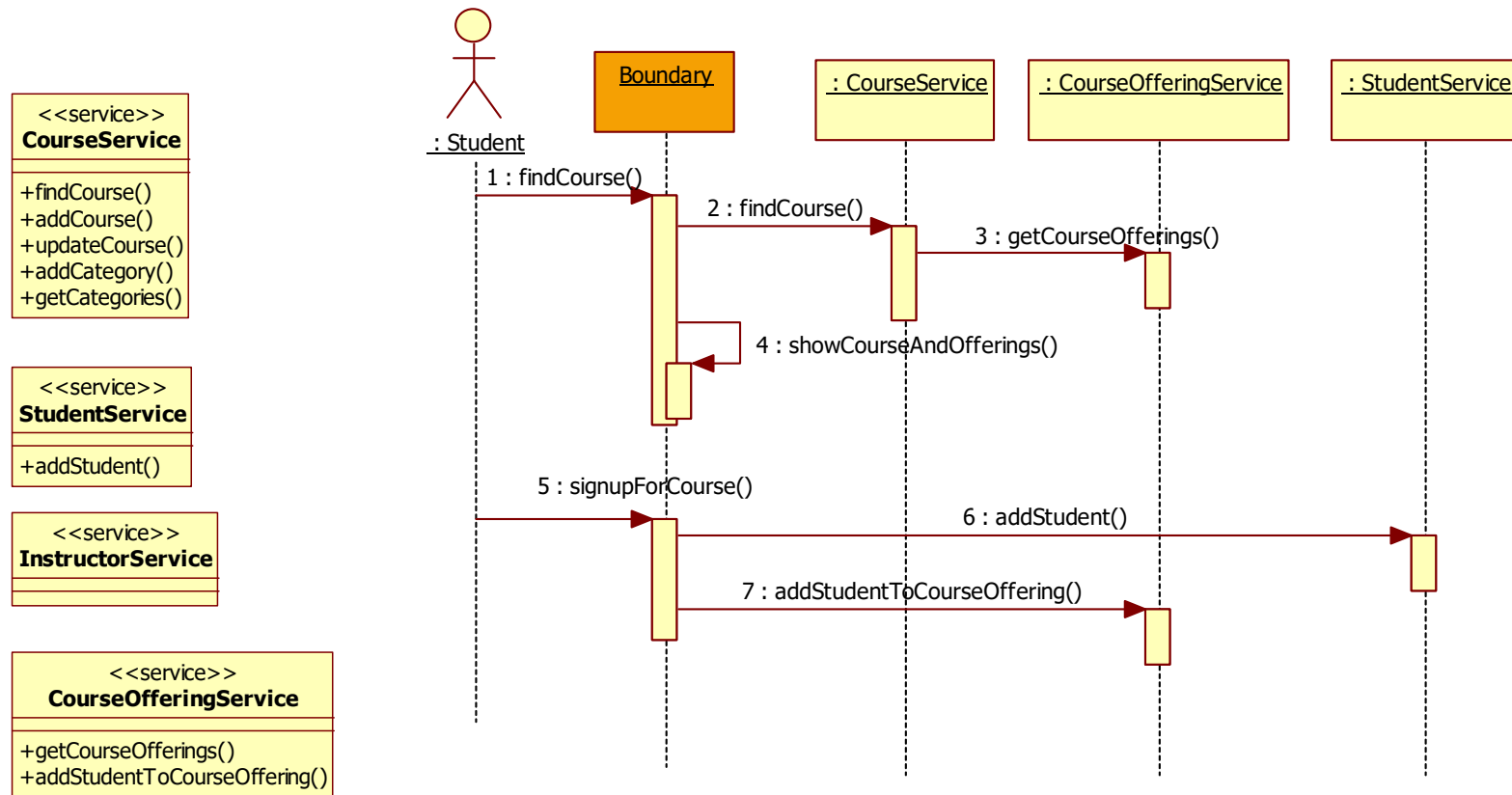




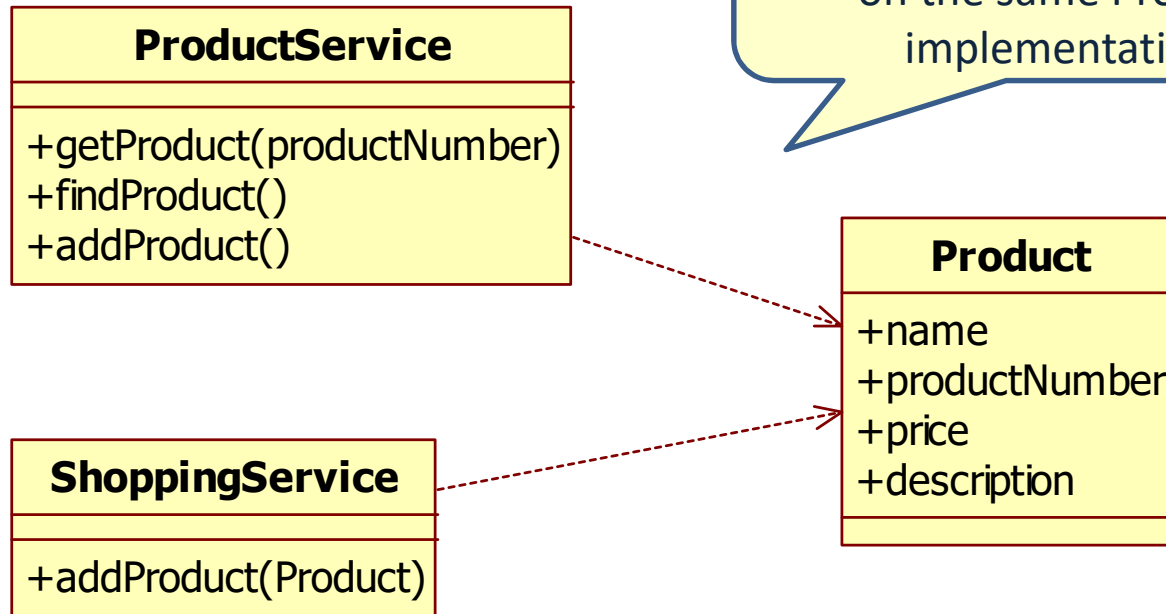
# Course Registration with components



# Course Registration with components



# Shared data between components

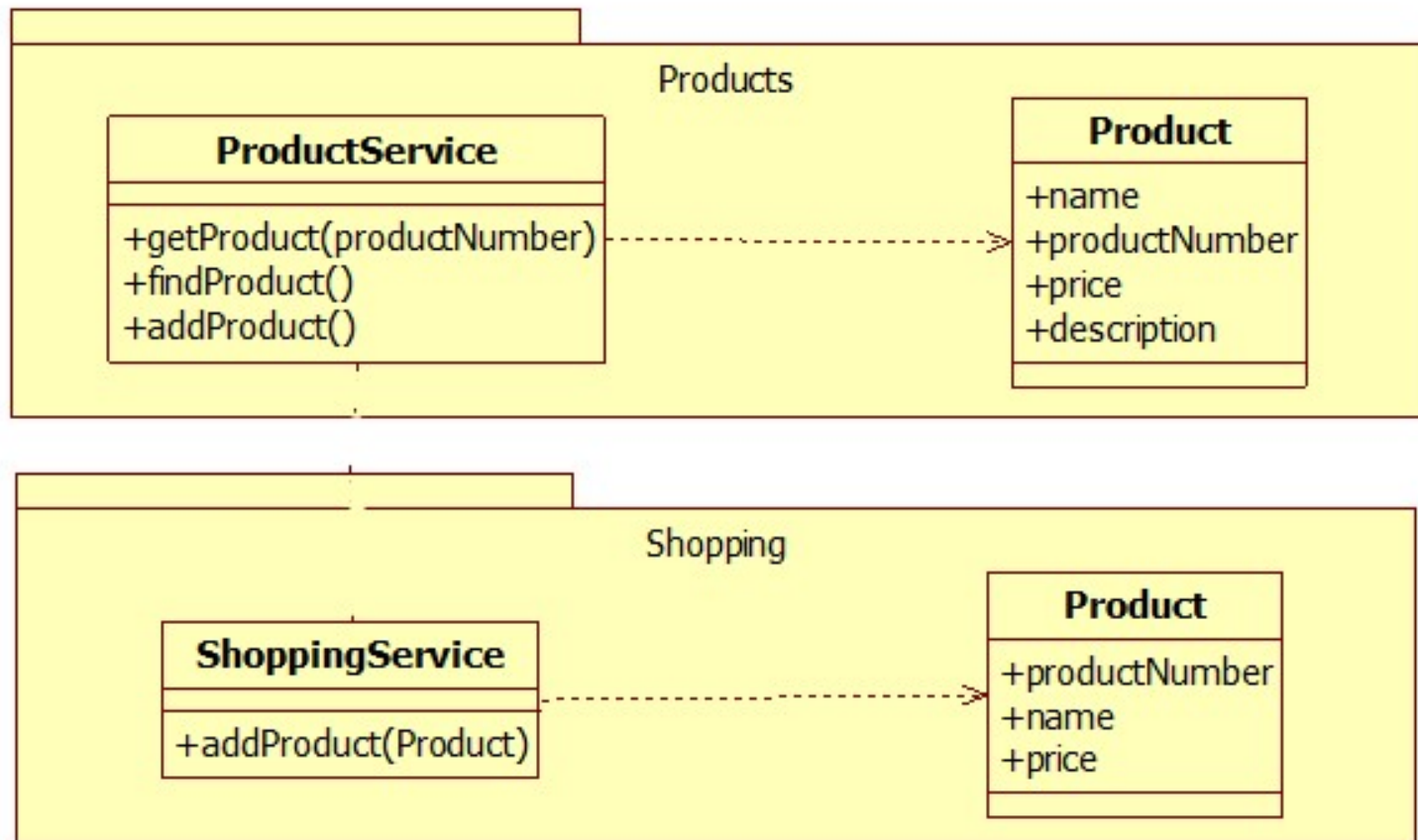


Both the ProductService and the ShoppingService depend on the same Product implementation

If the ProductService needs to change the Product, then the ShoppingService also needs to change



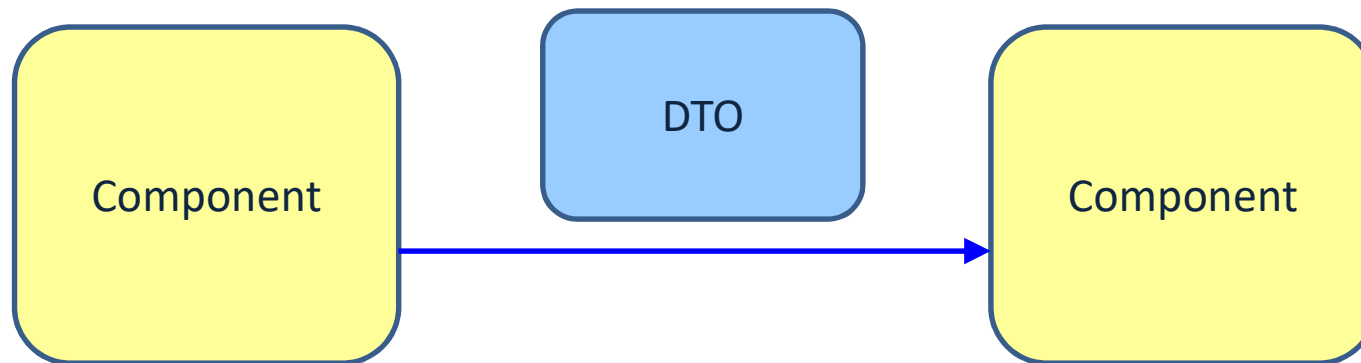
# No shared data between components



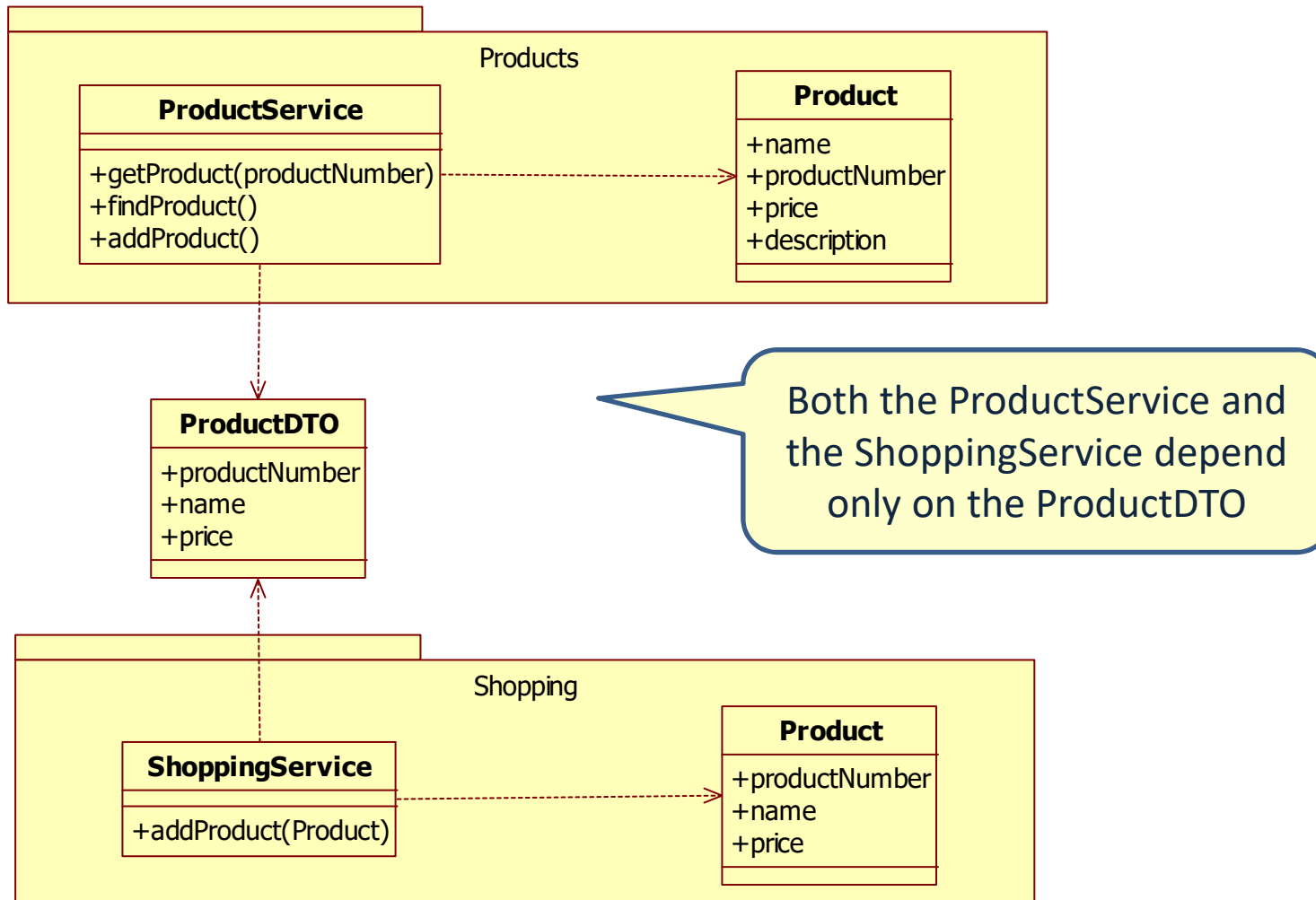
# Data Transfer Objects (DTO)

---

- Object that contains only attributes and getters and setters

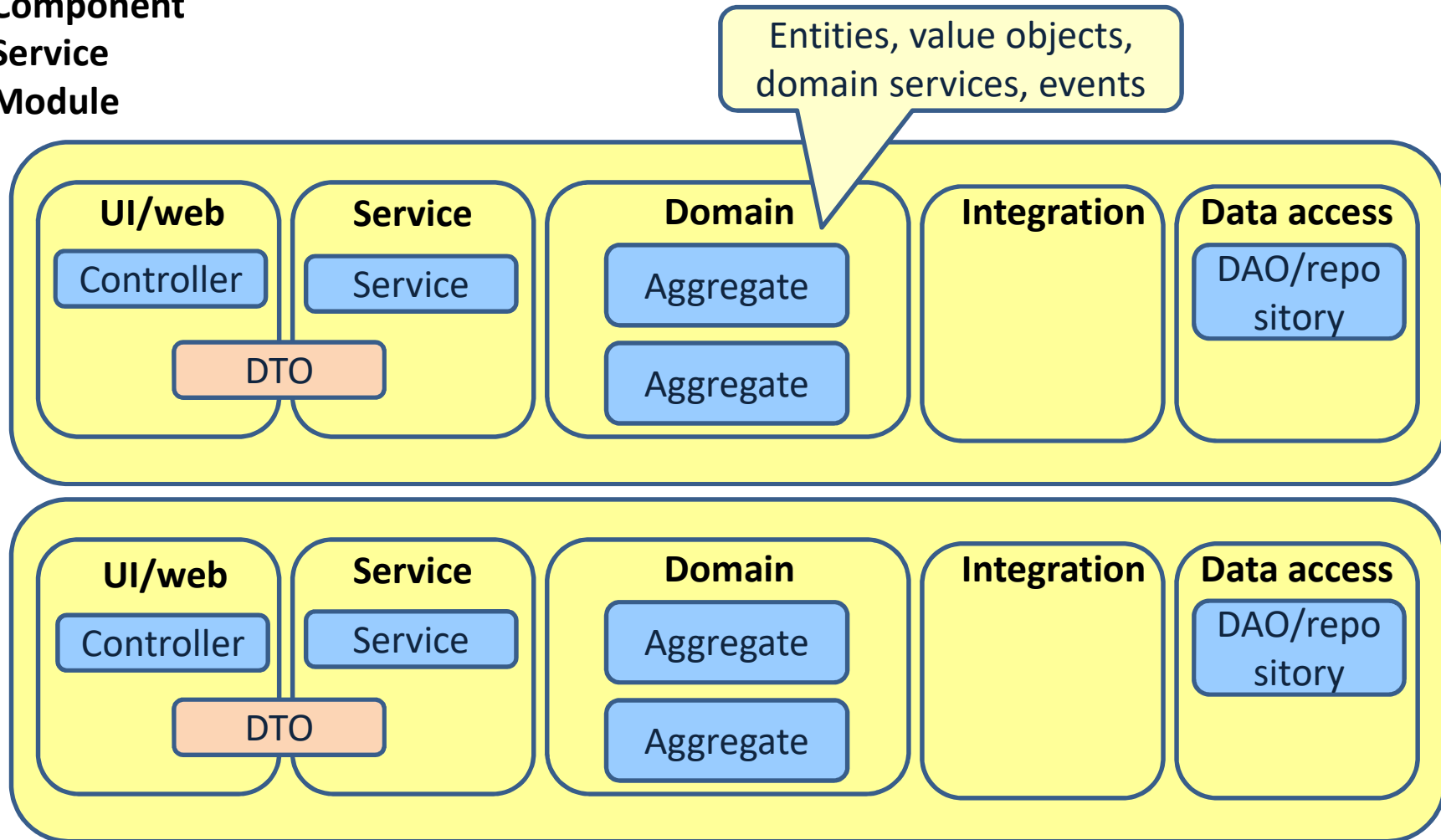


# Data Transfer Objects (DTO)



# How does it all fits together?

- Bounded context
- Component
- Service
- Module



# SPRING BOOT EVENTS





# Events

```
public class AddCustomerEvent {  
    private String message;  
  
    public AddCustomerEvent(String message) {  
        this.message = message;  
    }  
  
    public String getMessage() {  
        return message;  
    }  
}
```

A simple event class

Immutable



# Event publisher and listener

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

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

Inject a publisher

```
@Service
public class Listener {

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

Listen to AddCustomer events



# Asynchronous events

---

```
@Service
@EnableAsync
public class Listener {

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



# 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.

