Application Platforms & Architecture

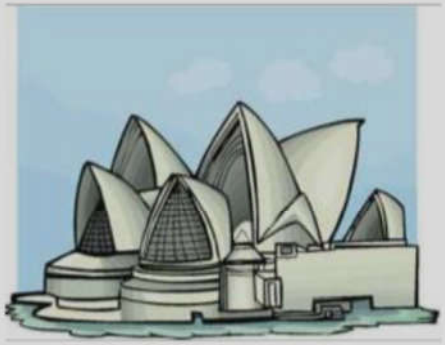
Day #4

ARCHITECTURE/DESIGN PRINCIPLES

Design principles

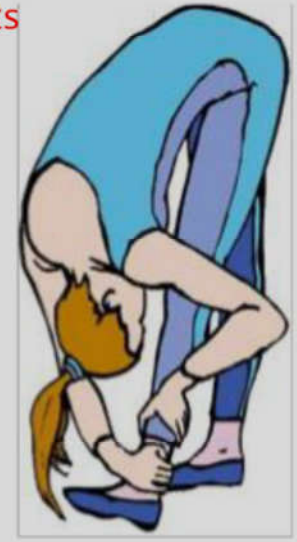
- Keep it simple
- Keep it flexible
- Loose coupling
- Separation of concern
- Information hiding
- Principle of modularity

Keep it simple



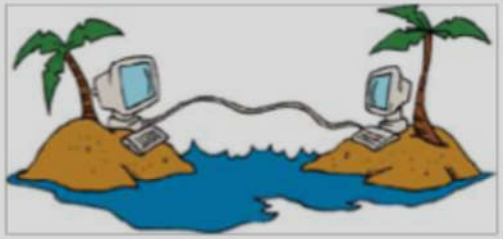
Keep it flexible

- Everthing changes
  - Business
  - Technical
- More flexibility leads to more complexity
  - Increased Decision Points
  - Management Overhead
  - Potential for Misuse
  - Testing Challenges



Loose coupling

- Different levels of coupling
  - Technology
  - Time
  - Location
  - Data structure
- You need coupling somewhere
  - Important is the level of coupling



Loose coupling

- Different levels of coupling
  - Technology
  - Time
  - Location
  - Data structure

Tightly Coupled (MySQL Specific)

Module A (Database.java):

```
public class Database {

    private static final String URL = "jdbc:mysql://localhost:3306/mydatabase";
    private static final String USERNAME = "myuser";
    private static final String PASSWORD = "mypassword";

    public static Connection getConnection() throws SQLException {
        return DriverManager.getConnection(URL, USERNAME, PASSWORD);
    }

    public static void main(String[] args) throws SQLException {
        Connection connection = getConnection();
        Statement statement = connection.createStatement();
        ResultSet resultSet = statement.executeQuery("SELECT * FROM users");

        // Process and display results (omitted for brevity)

        resultSet.close();
        statement.close();
        connection.close();
    }
}
```

Loosely Coupled (Database Interface)

```
public interface Database {

   public Connection getConnection() throws SQLException;

   public void closeConnection(Connection connection) throws SQLException;

   public ResultSet executeQuery(Connection connection, String sql) throws SQLException;
}
```

MySQL Implementation (MySqlDatabase.java)

```
public class MySqlDatabase implements Database {

    private static final String URL = "jdbc:mysql://localhost:3306/mydatabase";
    private static final String USERNAME = "myuser";
    private static final String PASSWORD = "mypassword";

    @Override
    public Connection getConnection() throws SQLException {
        return DriverManager.getConnection(URL, USERNAME, PASSWORD);
    }

    @Override
    public void closeConnection(Connection connection) throws SQLException {
        connection.close();
    }

    @Override
    public ResultSet executeQuery(Connection connection, String sql) throws SQLException {
        Statement statement = connection.createStatement();
        return statement.executeQuery(sql);
    }
}
```

ModuleA.java

```
public class ModuleA {

    public static void main(String[] args) throws SQLException {
        Database database = new MySqlDatabase(); // Choose implementation

        Connection connection = database.getConnection();
        ResultSet resultSet = database.executeQuery(connection, "SELECT * FROM users");

        // Process and display results

        database.closeConnection(connection);
    }
}
```

Loose coupling

- Different levels of coupling
  - Technology
  - Time
  - Location
  - Data structure

- Time Coupling refers to a situation where two or more parts of a system are dependent on each other executing in a specific time or sequence.
- This makes them tightly bound, meaning that one component needs the other to run or respond immediately (or within a small-time frame) to work properly.
- It is common in systems that involve synchronous interactions or rely on shared clock assumptions, such as real-time applications, distributed systems, or microservices.

Synchronous Method Call with Tight Timing Dependency

```java
public class OrderService {

    private PaymentProcessor paymentProcessor = new PaymentProcessor();

    // Time-coupled dependency: Payment must be processed immediately after order creation
    public void createOrder() {
        Order order = new Order(1, "New Order");
        System.out.println("Order created: " + order.getId());

        // Payment must be processed immediately after order creation
        paymentProcessor.processPayment(order);
    }

    public static void main(String[] args) {
        OrderService orderService = new OrderService();
        orderService.createOrder();
    }
}
```

```java
class Order {
    private int id;
    private String description;

    public Order(int id, String description) {
        this.id = id;
        this.description = description;
    }

    public int getId() {
        return id;
    }
}
```

Synchronous Method Call with Tight Timing Dependency

```java
public class PaymentProcessor {

    // Simulate payment service that is time-coupled with order creation
    public void processPayment(Order order) {
        if (order == null) {
            throw new IllegalArgumentException("Order cannot be null.");
        }
        System.out.println("Processing payment for order: " + order.getId());

        // Assume it takes time to complete the payment transaction
        try {
            Thread.sleep(2000); // Simulate network delay of 2 seconds
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Payment processed successfully for order: " + order.getId());
    }
}
```

There is a time-sensitive dependency: the payment must be processed immediately after the order is created. If this does not happen, the system might behave incorrectly (e.g., orders marked as unpaid).

Solution: Decouple with Asynchronous Processing

```java
import java.util.concurrent.CompletableFuture;
public class OrderService {
    private PaymentProcessor paymentProcessor = new PaymentProcessor();
    // Decoupled: Payment is now processed asynchronously
    public void createOrder() {
        Order order = new Order(1, "New Order");
        System.out.println("Order created: " + order.getId());
        // Process payment asynchronously to avoid blocking
        CompletableFuture.runAsync(() -> paymentProcessor.processPayment(order));
        System.out.println("Order creation finished, payment will process in background.");
    }

    public static void main(String[] args) {
        OrderService orderService = new OrderService();
        orderService.createOrder();

        // Give some time for async payment processing to complete before program ends
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

- Now, the payment processing runs in the background using CompletableFuture.runAsync().
- The order creation and payment processing are no longer time-coupled, making the system more resilient to delays.
- This design is more scalable since other orders can be processed without waiting for payment confirmation.

Loose coupling

- Different levels of coupling
  - Technology
  - Time
  - Location
  - Data structure

Location coupling in software engineering refers to the degree of interdependence between modules based on their physical location within the codebase.

Tight Location Coupling

```java
package com.shop.services;

public class OrderService {
    public void placeOrder(String item) {
        System.out.println("Order placed for: " + item);
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        OrderManager manager = new OrderManager();
        manager.manageOrder("Laptop");
    }
}
```

```java
package com.shop.managers;

import com.shop.services.OrderService;  // Hard location dependency

 class OrderManager {
    private final OrderService orderService;

    public OrderManager() {
        this.orderService = new OrderService();  // Direct coupling
    }

    public void manageOrder(String item) {
        System.out.println("Managing order...");
        orderService.placeOrder(item);
    }
}
```

The problem is that OrderManager is directly dependent on the exact location of OrderService.
If the package or structure of the OrderService changes, you'll need to update every place where it's used.

```java
package com.shop.api;

public interface OrderService {
    void placeOrder(String item);
}
```

```java
package com.shop.services;

import com.shop.api.OrderService;

public class OrderServiceImpl implements OrderService {
    @Override
    public void placeOrder(String item) {
        System.out.println("Order placed for: " + item);
    }
}
```

```java
package com.shop.managers;

import com.shop.api.OrderService;

public class OrderManager {
    private final OrderService orderService;
    // Injecting the dependency via the constructor
    public OrderManager(OrderService orderService) {
        this.orderService = orderService;
    }
    public void manageOrder(String item) {
        System.out.println("Managing order...");
        orderService.placeOrder(item);
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        OrderServiceImpl service = new OrderServiceImpl();
        OrderManager manager = new OrderManager(service);

        manager.manageOrder("Laptop");
    }
}
```

```java
package com.shop.api;

public interface OrderService {
    void placeOrder(String item);
}
```

```java
package com.shop.services;

import com.shop.api.OrderService;

public class OrderServiceImpl implements OrderService {
    @Override
    public void placeOrder(String item) {
        //...
    }
}
```

```java
package com.shop.managers;

import com.shop.api.OrderService;

public class OrderManager {
    private final OrderService orderService;
    // Injecting the dependency via the constructor
    public OrderManager(OrderService orderService) {
        this.orderService = orderService;
    }
    public void manageOrder(String item) {
        System.out.println("Managing order...");
        orderService.placeOrder(item);
    }
}
```

No Location Dependency in OrderManager:
OrderManager only knows about the OrderService interface, not the specific implementation. If the implementation moves to a different package, no changes are needed in OrderManager.

Easier to Change Implementations:
You can swap out OrderServiceImpl with a new implementation (e.g., MockOrderService) for testing or future upgrades without changing the OrderManager code.

More Modular Code:
The code is more modular and maintainable. You can introduce multiple implementations of OrderService for different contexts (like online vs offline orders) without touching the OrderManager.

Loose coupling

- Different levels of coupling
  - Technology
  - Time
  - Location
  - Data structure

- This refers to situations where classes or components are tightly bound to the specific internal data structures of each other.
- This type of coupling happens when one class directly relies on or manipulates the data fields or structures (such as arrays, lists, or maps) of another class, making changes to these structures disruptive and reducing flexibility in the design.
- When data structure coupling is high, changes to the internal representation of data in one class can break other classes that depend on it. Lowering this coupling can make code more modular, encapsulated, and maintainable.

```java
public class StudentManager {
    private final StudentRecords records;

    public StudentManager(StudentRecords records) {
        this.records = records;
    }

    public void addStudent(String name) {
        records.students.add(name); // Directly accessing the List
in StudentRecords
    }

    public void printStudents() {
        for (String student : records.students) {
            System.out.println("Student: " + student);
        }
    }
}
```

```java
public class StudentRecords {
    // List directly accessible
    public List<String> students = new ArrayList<>();

    public StudentRecords() {
        students.add("Alice");
        students.add("Bob");
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        StudentRecords records = new StudentRecords();
        StudentManager manager = new StudentManager(records);
        manager.addStudent("Charlie");
        manager.printStudents();
    }
}
```

```java
public class StudentManager {
    private final StudentRecords records;

    public StudentManager(StudentRecords records) {
        this.records = records;
    }

    public void addStudent(String name) {
        records.students.add(name);
        records.addStudent(name); // No direct access to List
    }

    public void printStudents() {
        for (String student : records.getStudents()) {
            System.out.println("Student: " + student);
        }
    }
}
```

```java
public class StudentRecords {
    private final List<String> students = new ArrayList<>();

    public StudentRecords() {
        students.add("Alice");
        students.add("Bob");
    }

    // Encapsulated method to add a student
    public void addStudent(String name) {
        students.add(name);
    }

    // Encapsulated method to get a read-only view of students
    public List<String> getStudents() {
        return Collections.unmodifiableList(students);
    }
}
```

Separation of concern

- Separate technology from business
- Separate stable things from changing things
- Separate things that need separate skills
- Separate business process from application logic
- Separate implementation from specification

Separate technology from business

SoC with Technology Layers

Scenario:
A user service interacts with a database to manage user data.

```
// Business Layer
public class UserService {

    private UserRepository userRepository;

    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public void registerUser(User user) {
        // Business logic for user registration
        userRepository.saveUser(user);
    }

    public User findUser(int id) {
        // Additional business logic if needed
        return userRepository.getUserById(id);
    }
}
```

```
// Data Access Layer
public interface UserRepository {
    void saveUser(User user);
    User getUserById(int id);
    // Other CRUD operations as needed
}

public class Database implements UserRepository {
    @Override
    public void saveUser(User user) {
        // Database connection and save logic
    }
    @Override
    public User getUserById(int id) {
        // Database connection and retrieve logic
        return new User();
    }
}
```

Loose Coupling: UserService depends on the UserRepository interface, not on the specific Database class. This enables flexible substitutions of the data layer (switching databases).
Clear Responsibilities: Business logic is centralized in UserService, while data access details are handled by Database.

Separate things that need separate skills

SoC with Skillsets

Scenario:
A UI developer focuses on building the user interface, while a backend developer works on the server-side logic.

```
//Code (UI Layer - MyWindow.java):

public class MyWindow extends JFrame {

   // UI components and event handling code

}
```

```
// Code (Backend Layer - MyService.java):

public class MyService {

   public String processData(String data) {
      // Business logic to process data (replace with actual implementation)
      return "Processed data";
   }
}
```

Separate business process from application logic

SoC with Process vs. Logic

Scenario:
An order processing system

- ***business workflow***
  - placing orders
- ***application logic***
  - calculating discounts
  - sending emails
  - generating invoices

SoC with Process vs. Logic

```
// Service Layer (service/OrderService.java)
@Service
public class OrderService {

    private final OrderDao orderDao;
    private final DiscountCalculator discountCalculator;  // Optional (Application Logic)
    private final EmailSender emailSender;  // Optional (Application Logic)
    private final InvoiceGenerator invoiceGenerator;  // Optional (Application Logic)

    public Order createOrder(CreateOrderRequest request) {
        Order order = new Order(request);

        // Business Process: Order Creation Workflow
        if (discountCalculator != null) {  // Optional (Application Logic)
            order.setDiscount(discountCalculator.calculateDiscount(order));
        }
        order.setTotalPrice(calculateTotalPrice(order));
        orderDao.saveOrder(order);

        // Business Process: Optional Invoice Generation and Email Sending
        if (invoiceGenerator != null && emailSender != null) {  // Optional (Application Logic)
            String invoice = invoiceGenerator.generateInvoice(order);
            emailSender.sendInvoice(order.getCustomerEmail(), invoice);
        }

        return order;
    }

    private double calculateTotalPrice(Order order) {
        // Logic to calculate total price based on order items and discount
    }
}
```

// Application Logic (util - Optional)

DiscountCalculator.java,
EmailSender.java
InvoiceGenerator.java

Separate implementation from specification

SoC with Interfaces

com.yourcompany.yourproject
- domain
  - product (Product.java)
- service
  - ProductService.java
- **service.impl**
  - PriceRangeSpecification.java (implementation detail)
- repository
  - ProductRepository.java
- controller
  - ProductController.java
- web (optional for web-specific configurations)
- ... (other web-related classes)

```
public interface ProductService {
   List<Product> findProducts(ProductSpecification<Product> spec);
}
```

```
public class PriceRangeSpecification implements
ProductSpecification<Product> {
   //...
}
```

Information hiding

- Black box principle
- Hide implementation behind an interface
- Hide the data structure behind stored procedures
- Hide the data structure behind business logic

Black box principle

```
public class Order {

    private long id;
    private List<OrderItem> items;
    private double totalPrice;

    public Order(long id, List<OrderItem> items) {
        //...
        calculateTotalPrice();
    }

    private void calculateTotalPrice() {
        // Logic to calculate total price based on order items (hidden)
        this.totalPrice = calculatePriceFromItems(items); // helper method
    }

    private double calculatePriceFromItems(List<OrderItem> items) {
        // Implement logic to calculate total price based on item details
}
```

```
    public long getId() {
        return id;
    }

    public List<OrderItem> getItems() {
        return
Collections.unmodifiableList(items);
    }

    public double getTotalPrice() {
        return totalPrice;
    }
}
```

Hide implementation behind an interface

```
interface Planet {
   double getGravity();
   double getDiameter();
}
```

```
class Earth implements Planet {
   @Override
   public double getGravity() {
      return 9.81; // in m/s^2
   }

   @Override
   public double getDiameter() {
      return 12756; // in kilometers
   }
}
```

```
public class Main {
   public static void main(String[] args) {
      Planet earth = new Earth();
      Planet mars = new Mars();

      System.out.println(earth.getGravity() + " m/s^2");
      System.out.println(earth.getDiameter() + " kilometers");

      System.out.println(mars.getGravity() + " m/s^2");
      System.out.println(mars.getDiameter() + " kilometers");
   }
}
```

```
class Mars implements Planet {
   @Override
   public double getGravity() {
      return 3.71; // in m/s^2
   }

   @Override
   public double getDiameter() {
      return 6792; // in kilometers
   }
}
```

Hide the data structure behind stored procedures

Suppose you have a database table named Employees with columns *id*, *name*, and *salary*.
You want to hide the data structure of this table behind stored procedures for CRUD (Create, Read, Update, Delete) operations.

```
1 •  SELECT * FROM mydb.Employees;
```

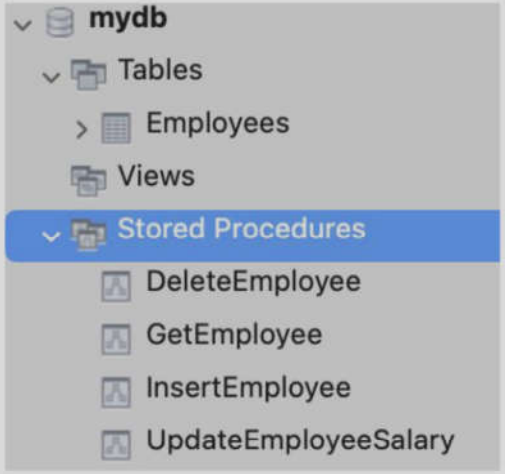
100% 30:1

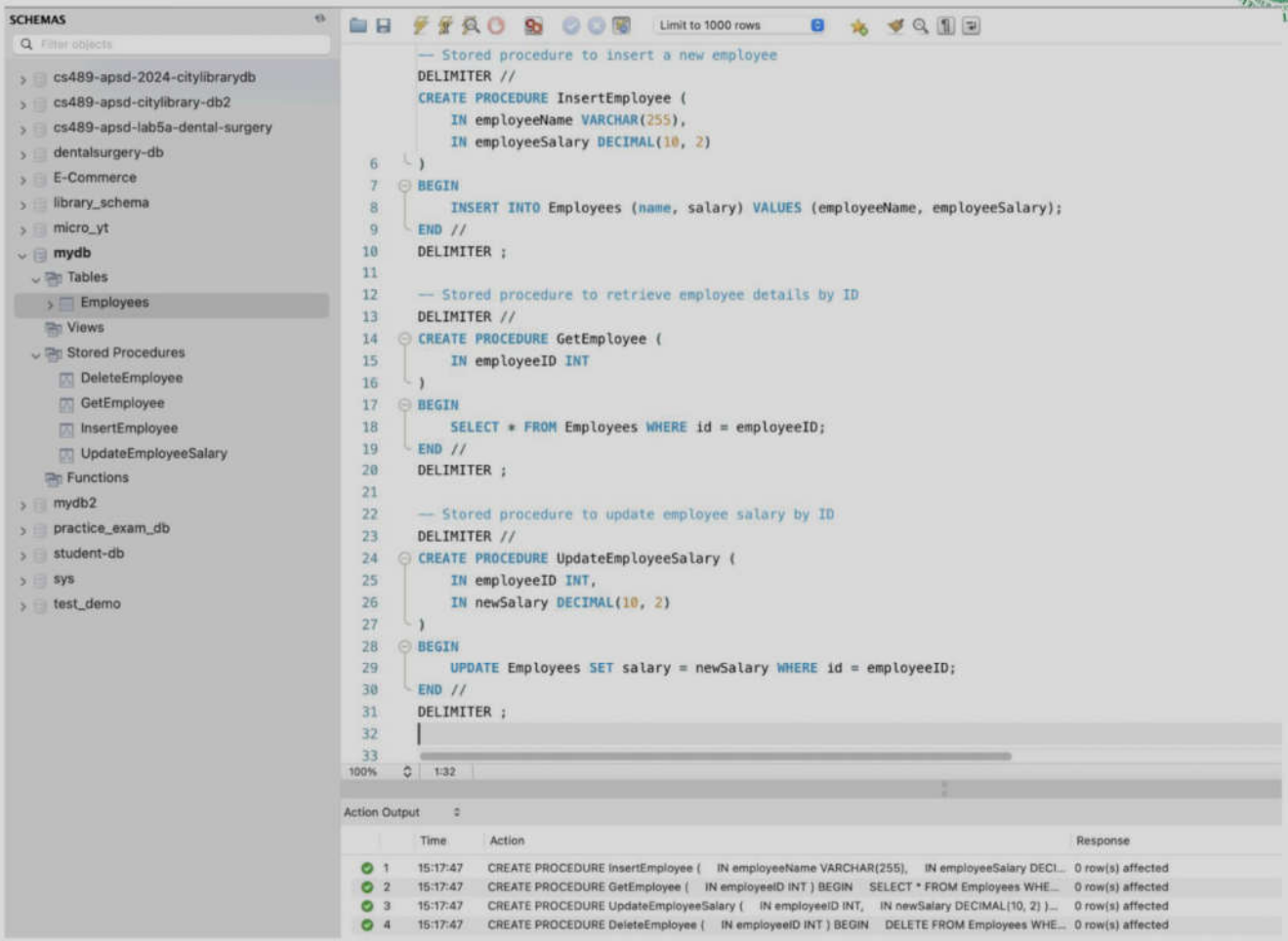
Result Grid Filter Rows: Search

id	name	salary
1	John Doe	50000.00
NULL	NULL	NULL

Hide the data structure behind stored procedures

Hide the data structure of this table behind stored procedures for CRUD (Create, Read, Update, Delete) operations.





```
-- Stored procedure to insert a new employee
DELIMITER //
CREATE PROCEDURE InsertEmployee (
    IN employeeName VARCHAR(255),
    IN employeeSalary DECIMAL(10, 2)
)
BEGIN
    INSERT INTO Employees (name, salary) VALUES (employeeName,
employeeSalary);
END //
DELIMITER ;

-- Stored procedure to retrieve employee details by ID
DELIMITER //
CREATE PROCEDURE GetEmployee (
    IN employeeID INT
)
BEGIN
    SELECT * FROM Employees WHERE id = employeeID;
END //
DELIMITER ;
```

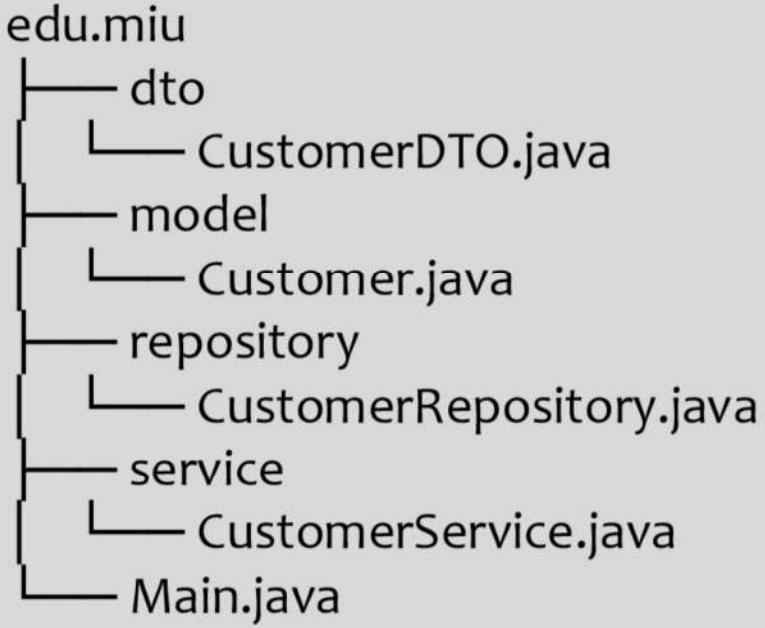
```
-- Stored procedure to update employee salary by ID
DELIMITER //
CREATE PROCEDURE UpdateEmployeeSalary (
    IN employeeID INT,
    IN newSalary DECIMAL(10, 2)
)
BEGIN
    UPDATE Employees SET salary = newSalary WHERE id = employeeID;
END //
DELIMITER ;

-- Stored procedure to delete employee by ID
DELIMITER //
CREATE PROCEDURE DeleteEmployee (
    IN employeeID INT
)
BEGIN
    DELETE FROM Employees WHERE id = employeeID;
END //
DELIMITER ;
```

```
String url = "jdbc:mysql://localhost:3306/mydb";
String user = ”user";
String password = ”***";

try (Connection conn = DriverManager.getConnection(url, user, password)) {
   // Call the InsertEmployee stored procedure
   CallableStatement insertStmt = conn.prepareCall("{call InsertEmployee(?, ?)}");
   insertStmt.setString(1, "John Doe");
   insertStmt.setDouble(2, 50000);
   insertStmt.execute();

   // Call the GetEmployee stored procedure
   CallableStatement getStmt = conn.prepareCall("{call GetEmployee(?)}");
   getStmt.setInt(1, 1); // Assuming employee ID 1 exists
   ResultSet rs = getStmt.executeQuery();
   if (rs.next()) {
      System.out.println("Employee ID: " + rs.getInt("id"));
      System.out.println("Employee Name: " + rs.getString("name"));
      System.out.println("Employee Salary: " + rs.getDouble("salary"));
   }
```

Hide the data structure behind business logic

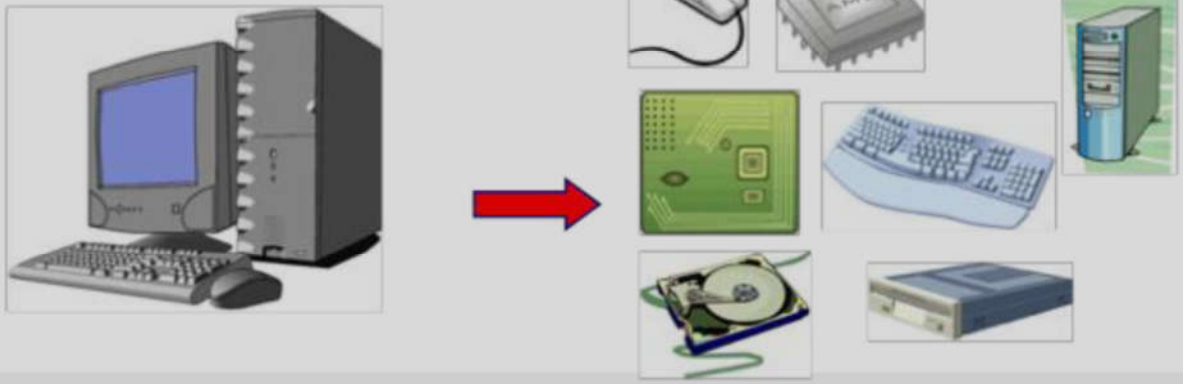
Why Hide Data Structure?

- **Loose Coupling:** Separates business logic from data access concerns. Changes to data storage or retrieval don't affect business logic as long as the interface remains the same.
- **Maintainability:** Makes code easier to understand and modify. Developers focus on what data needs to be accessed and manipulated, not on the specifics of how it's stored.
- **Flexibility:** Allows switching data access implementations (e.g., database vs. in-memory) without affecting the business logic.

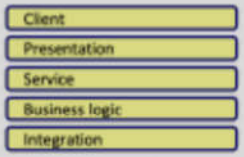


Principle of modularity

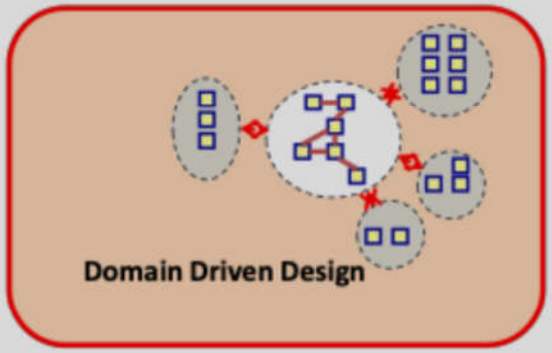
- Decomposition
- Devide a big complex problem is smaller parts
- Use components that are
  - Better understandable
  - Independent
  - Reusable
- Leads to more flexibility
- Makes finding and solvings bugs easier

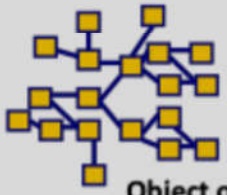


Architecture styles

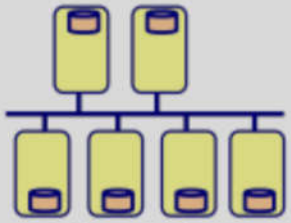


Layering

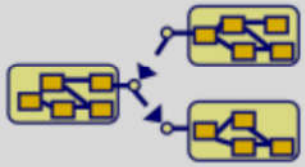




Object oriented



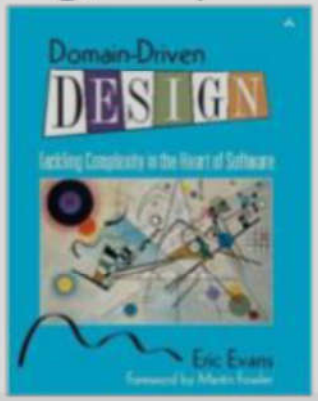
Microservices



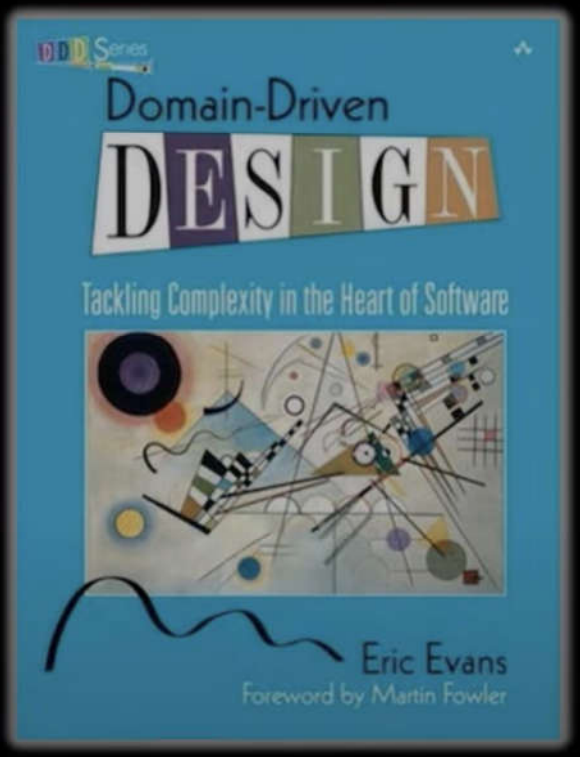
Component based

What is Domain Driven Design?

- An approach to software development where the focus is on the core Domain.
  - We create a domain model to communicate the domain
  - Everything we do (discussions, design, coding, testing, documenting, etc.) is based on the domain model.



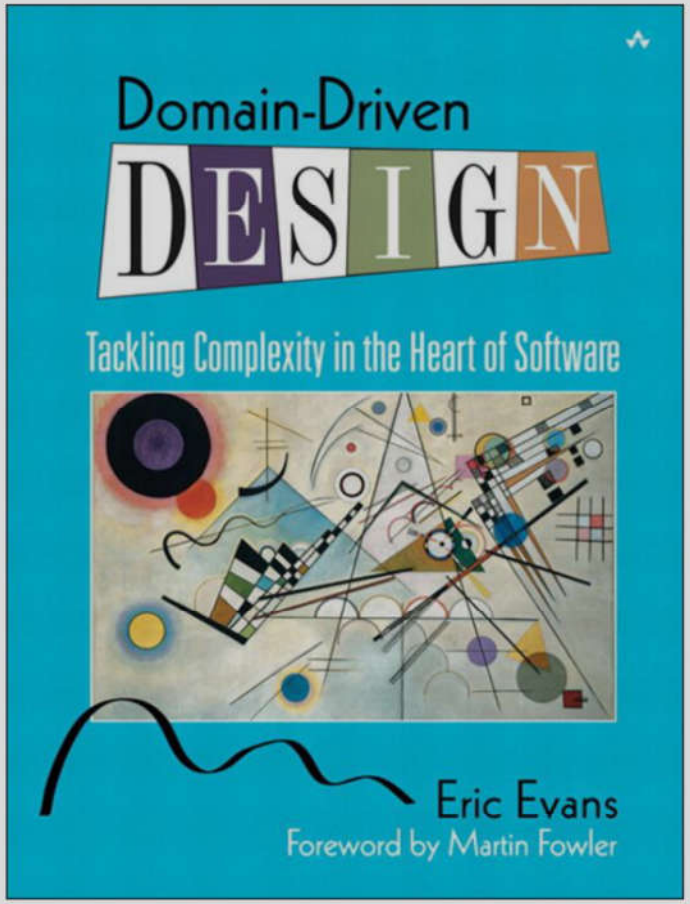
What's a Domain Model?



*it is **not a** particular **diagram**; it is **the idea** that the diagram **is intended to convey***

*it is **not just the knowledge of the domain** expert; it is a **rigorously organized and selective abstraction** of that knowledge*

the model is the internal representation of the target domain



- The leaders within a team who understand the centrality of the domain can put their software project back on course when enthusiastic developers get caught up in develop elaborate technical frameworks that do not serve, or actually get in the way of domain development, while development of a model that reflects deep understanding of the domain is lost in the shuffle.

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

Rich Domain Model

A rich domain model is a design approach in object-oriented programming where domain objects (or entities) encapsulate both data and behavior.

```java
// Value Object
public class OrderItem {
    private final Product product;
    private final int quantity;

    public OrderItem(Product product, int quantity) {
        if (quantity <= 0) {
            throw new
IllegalArgumentException("Quantity must be
greater than zero.");
        }
        this.product = product;
        this.quantity = quantity;
    }

    public double getTotalPrice() {
        return product.getPrice() * quantity;
    }
}
```

```java
// Entity
public class Order {
    private final String orderId;
    private List<OrderItem> items = new ArrayList<>();
    private OrderStatus status;

    public Order(String orderId) {
        this.orderId = orderId;
        this.status = OrderStatus.NEW;
    }

    public void addItem(Product product, int quantity) {
        OrderItem orderItem = new OrderItem(product, quantity);
        items.add(orderItem);
        // You can include additional logic to update status or totals here
    }

    public double calculateTotal() {
        return items.stream().mapToDouble(OrderItem::getTotalPrice).sum();
    }

    public void placeOrder() {
        if (items.isEmpty()) {
            throw new IllegalStateException("Cannot place an empty order.");
        }
        this.status = OrderStatus.PLACED;
        // Raise domain event, etc.
    }
```

Use one common language to describe the concepts of a domain

Ubiquitous language

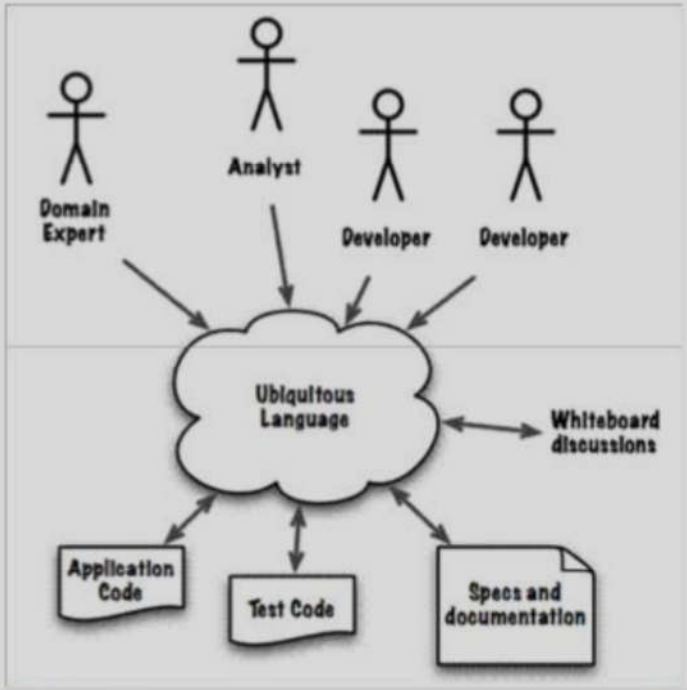
It's a common language that bridges the gap between:

Business Experts: People who understand the real-world problem the software is trying to solve (e.g., salespeople, managers, analysts).

Technical Team: Developers, designers, testers who build and maintain the software.

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



Model

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

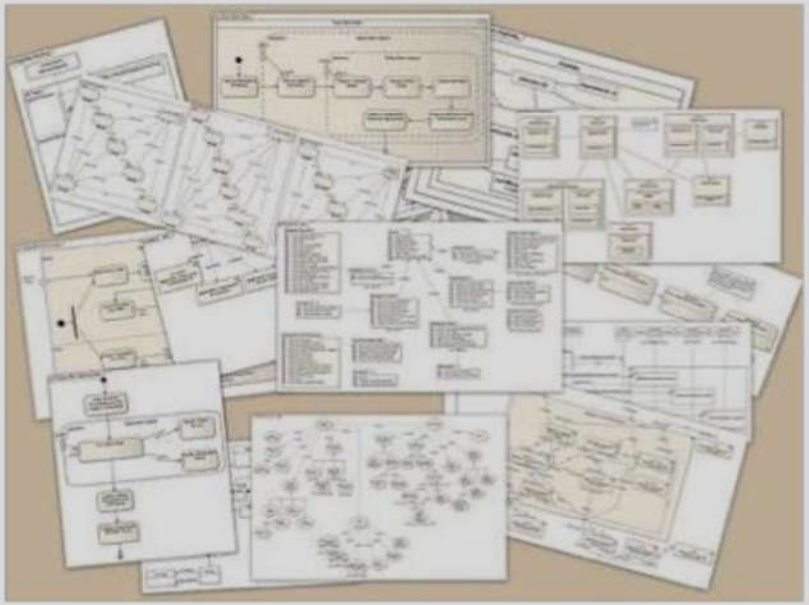
Domain model

- **Simplification of Reality**
  - Imagine a model train set.
  - It captures the essence of a real train system with tracks, locomotives, and carriages.
  - Still, it ignores all the complexities of a real train network, like weather, passenger flow, or intricate signaling systems.

- **Area of Interest**
  - The model train set focuses on the physical movement of trains on tracks.
  - It does not represent the ticketing system, the train stations' architecture, or the surrounding landscape – all of which might be relevant in a different context (e.g., city planning simulation).

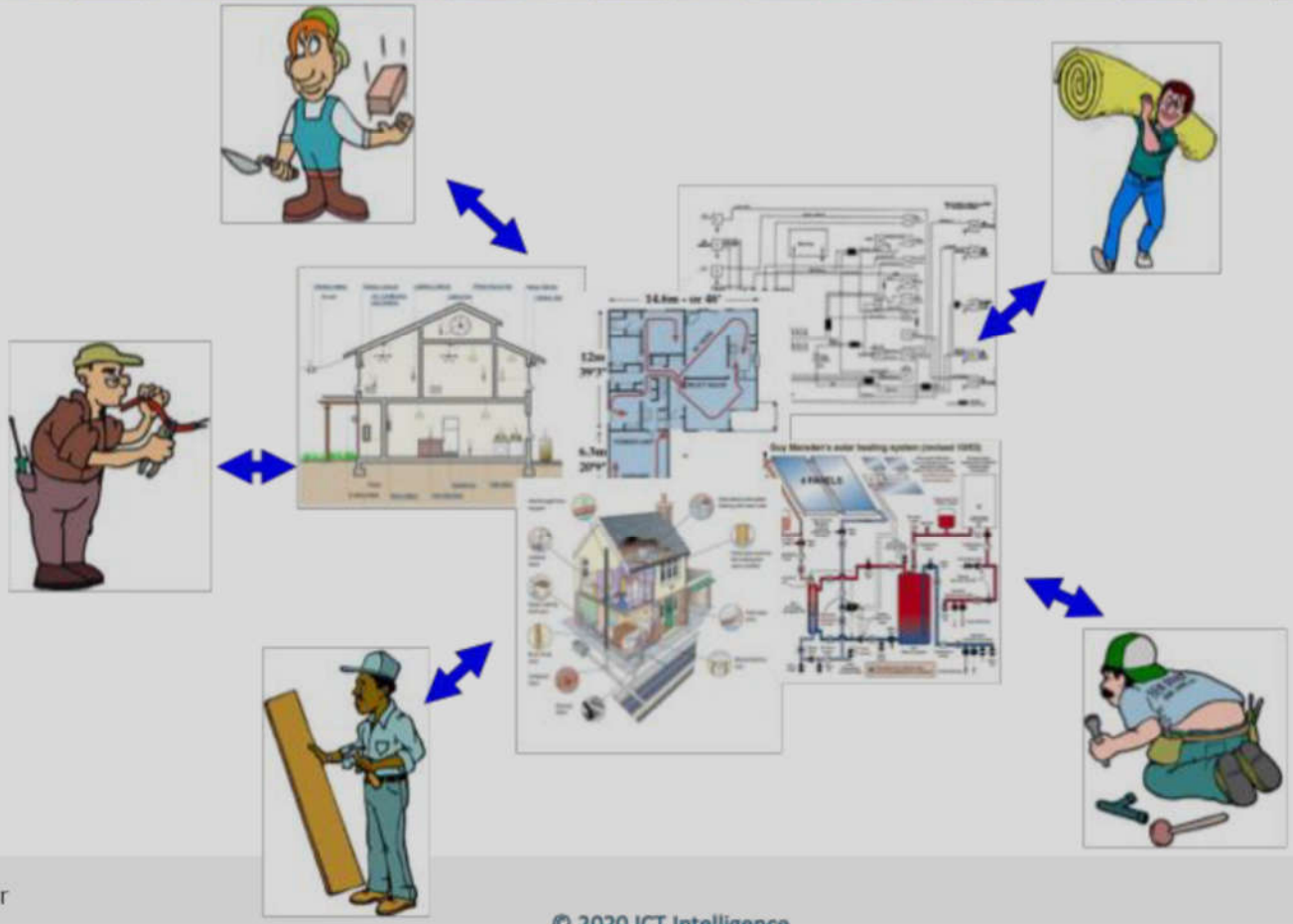
Structure of the domain model

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



Effective communication

Model and diagrams



Advantages of a domain model

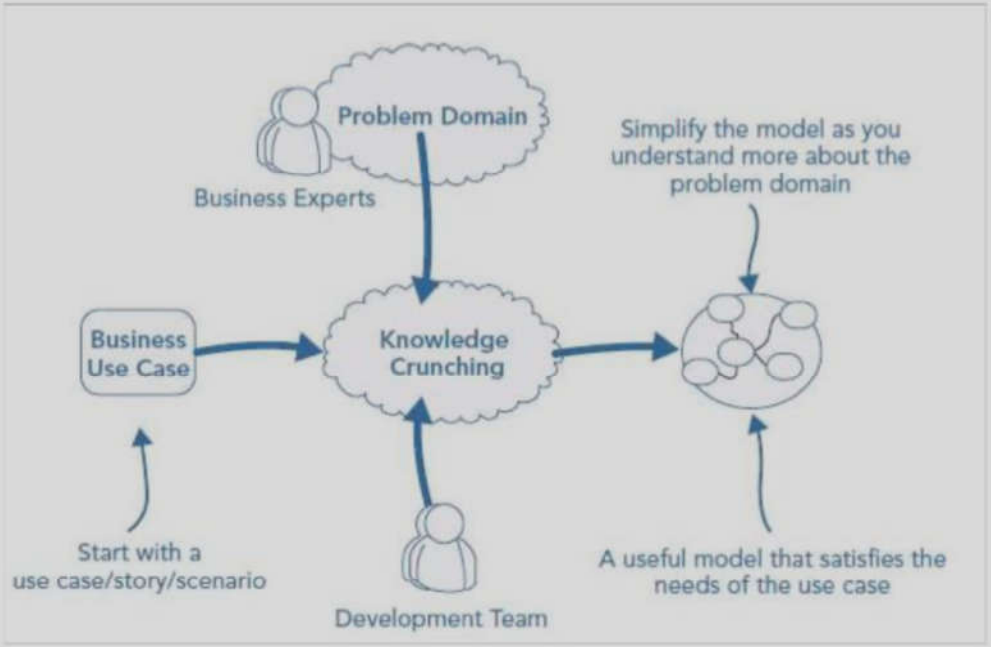
- Improves understanding
- Validates understanding
- Improves communication
- Shared glossary
- Improves discovery

Knowledge Crunching

Knowledge crunching is the process of collaboratively extracting key concepts and information from a domain (the specific area of business that the software is being built for).

It's like sifting through a large amount of data to find the most relevant bits for building an accurate and effective software model.

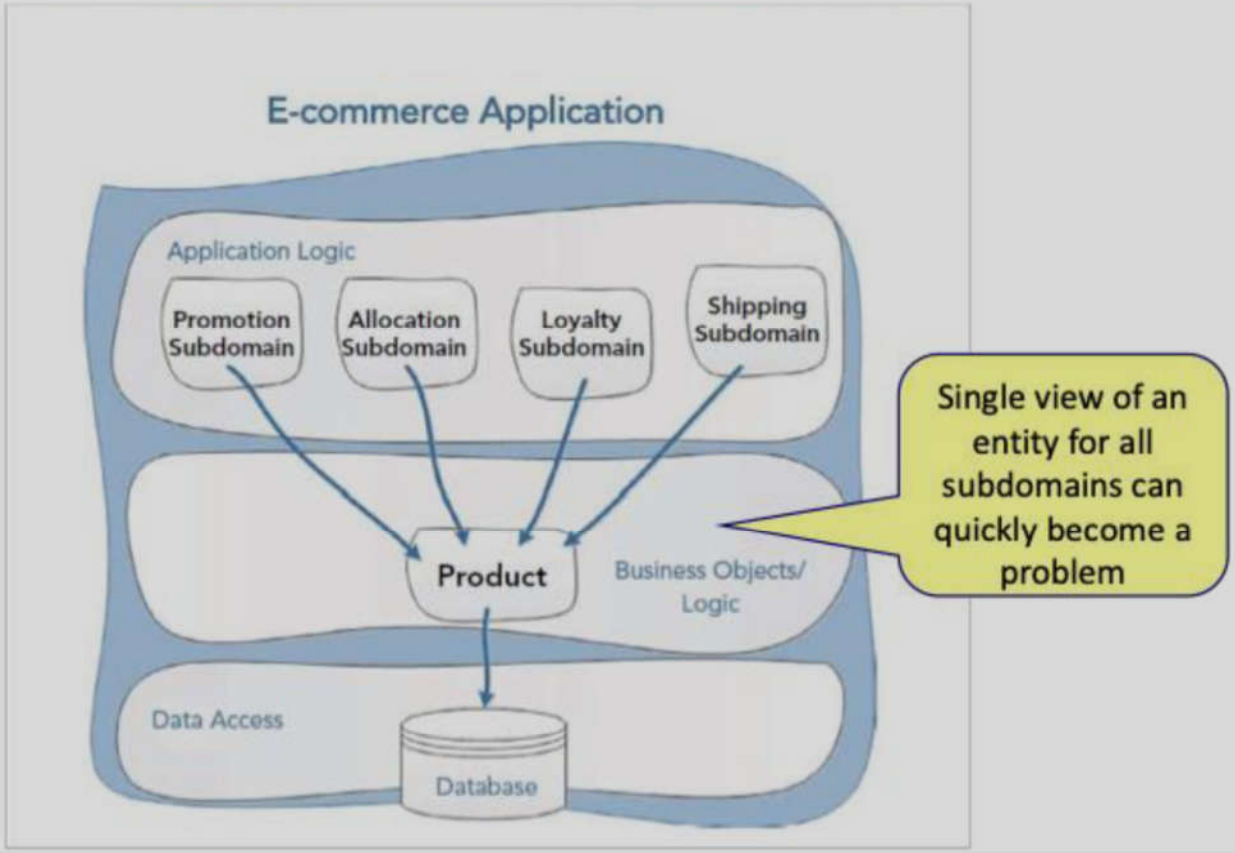
Business experts are also referred to as domain experts



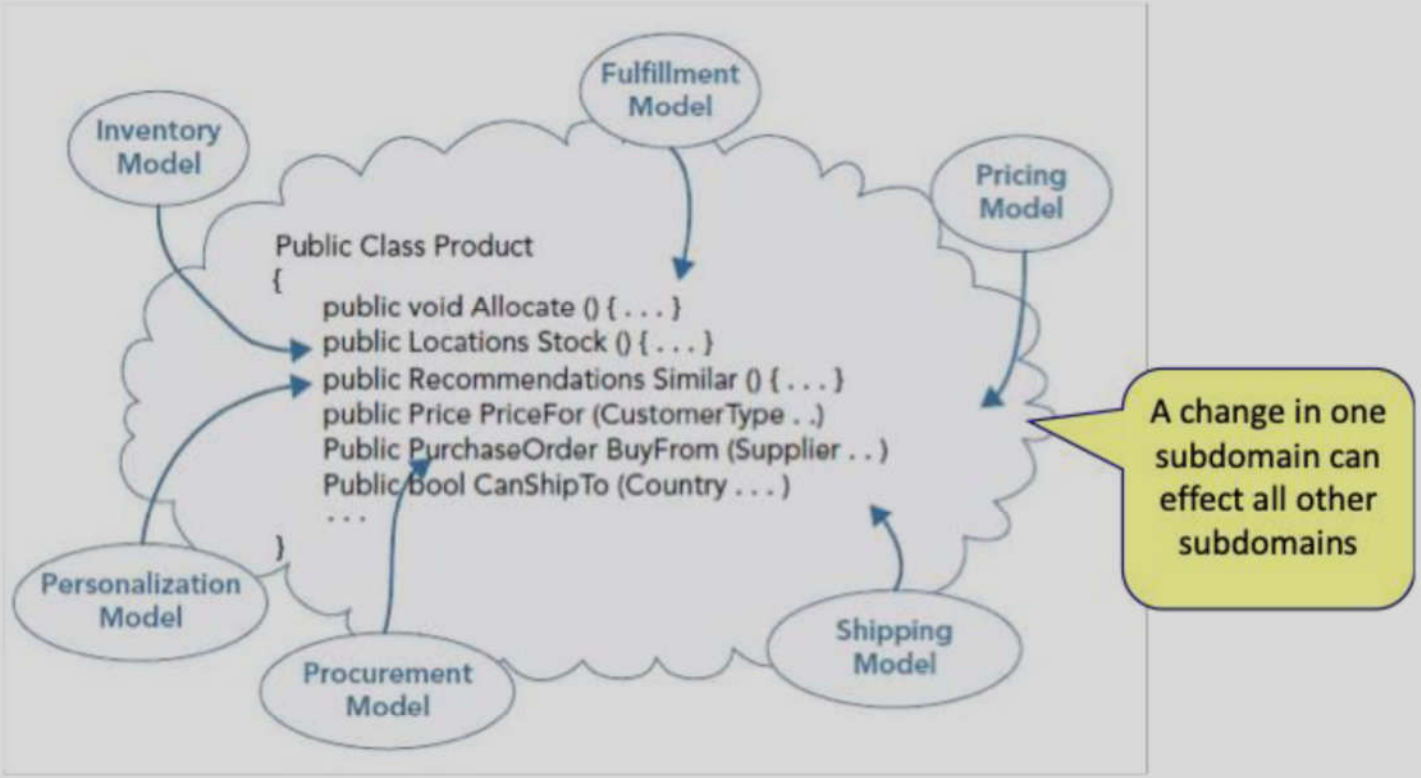
The software is a reflection of the real world

- 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

Shared objects between subdomains

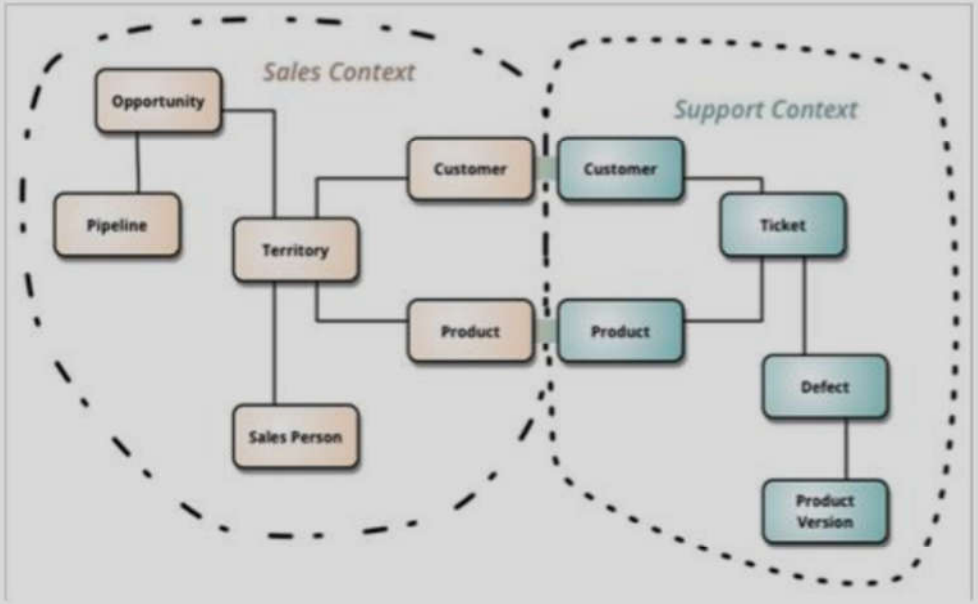


Big ball of mud

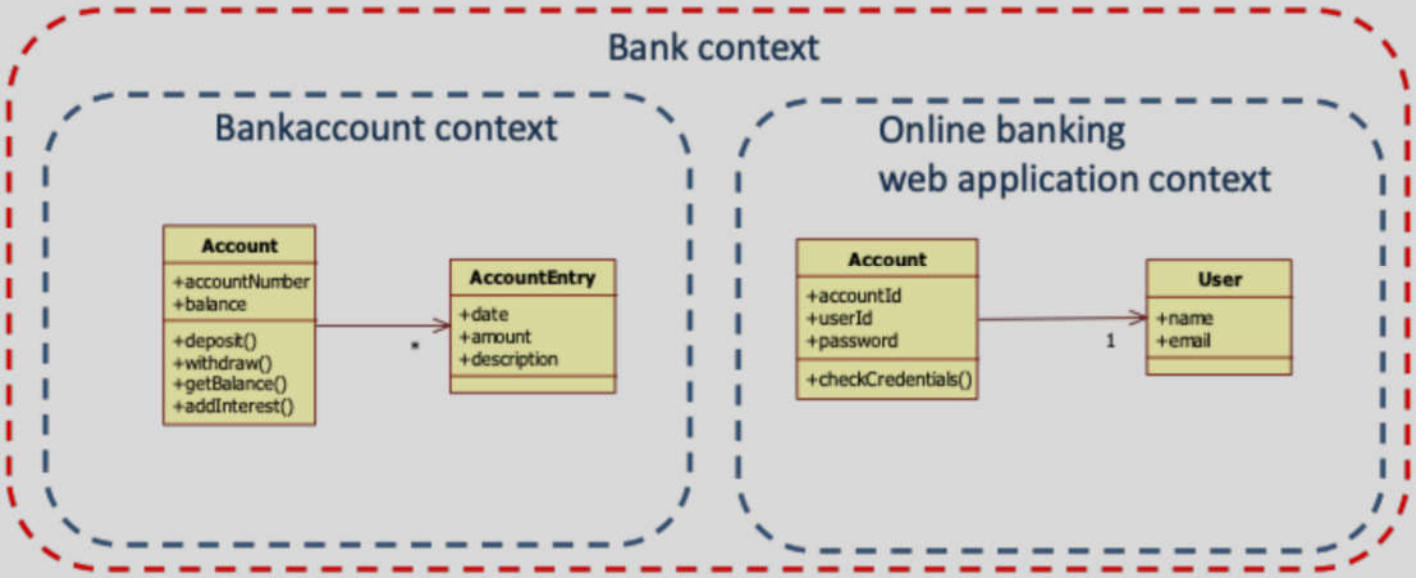


Context

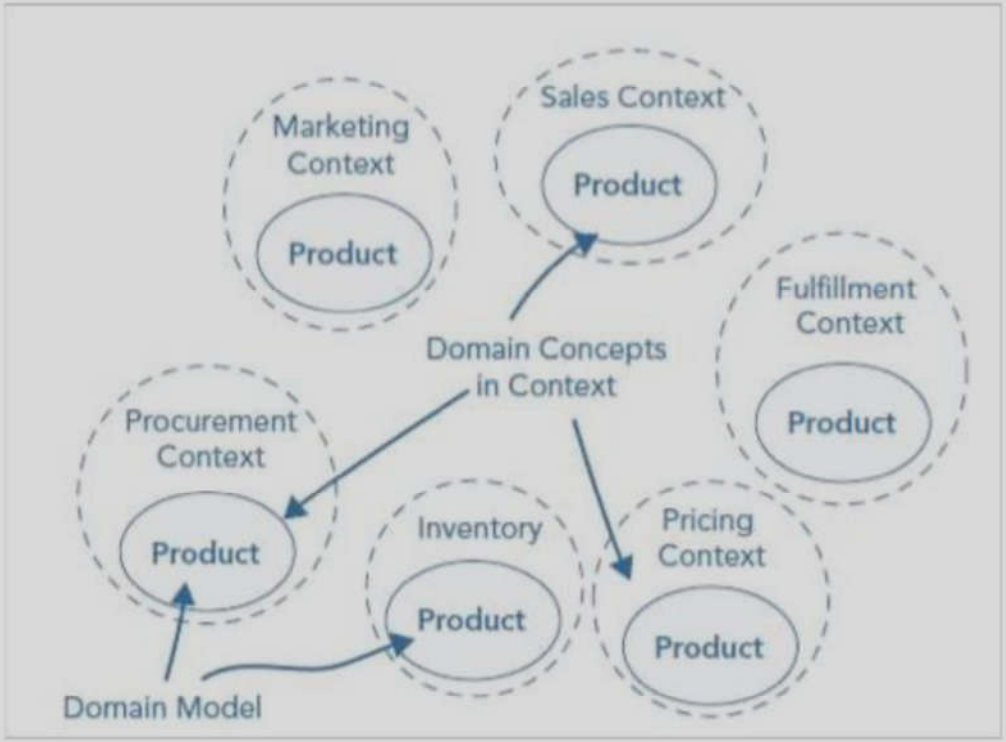
- A specific domain term may have a different definition in a different context



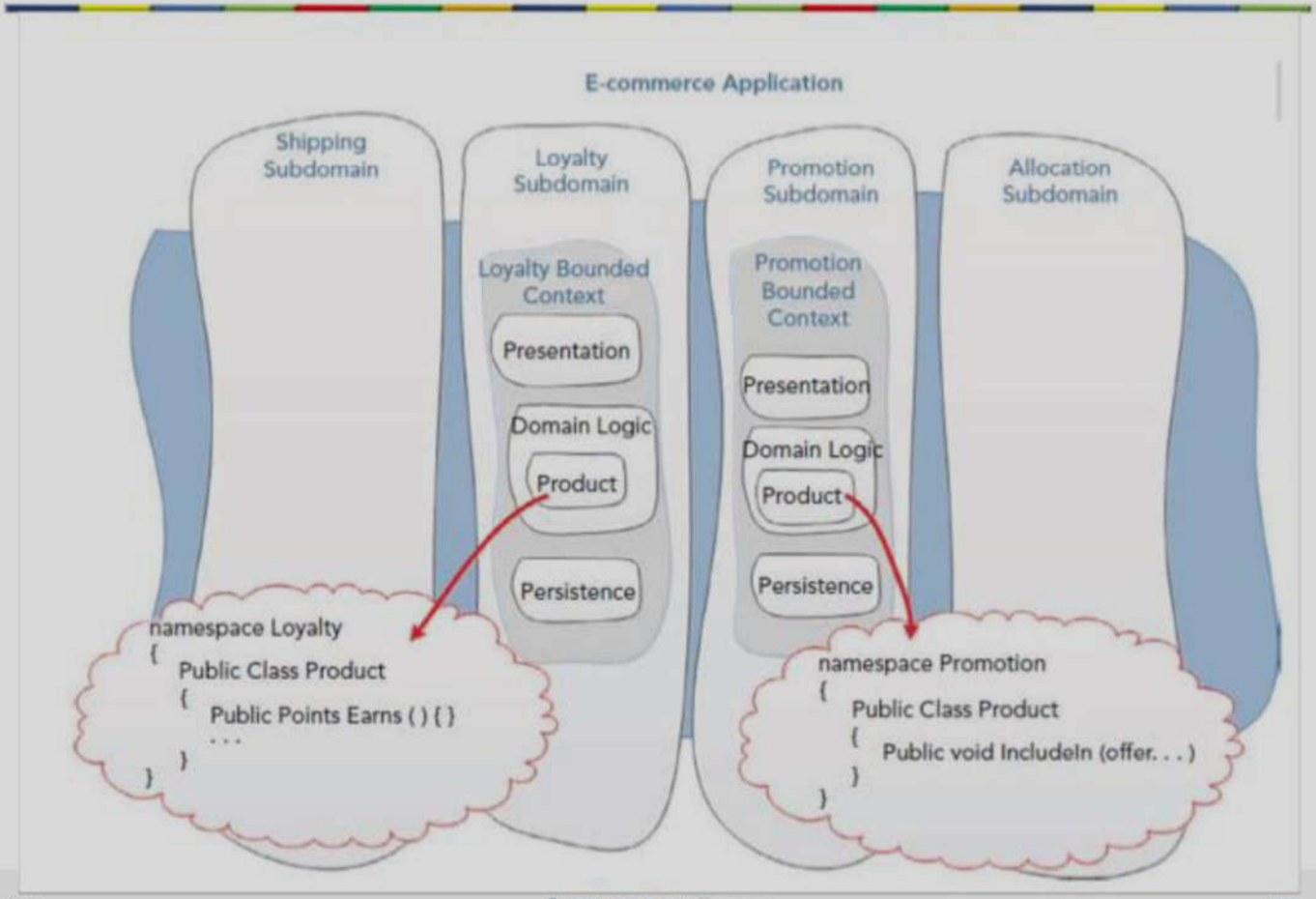
Bounded context



Each bounded context has its own model



Bounded context example

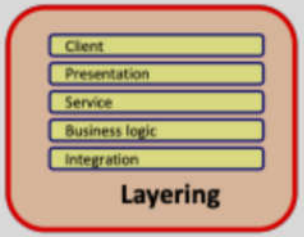


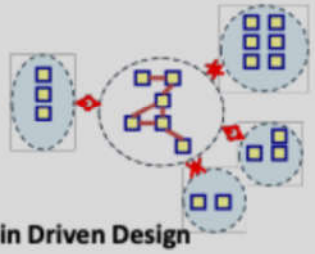
Why DDD?

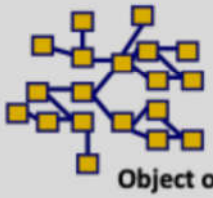
- Writing code is easy.
- Learning an new technology is fairly easy
- The hardest part of software development is to understand what the business wants and how the business works
  - Focus on those areas of the application that deliver the most value to the business
  - This greatly helps to
    - Understand the application
    - To evolve the application
    - To test the application

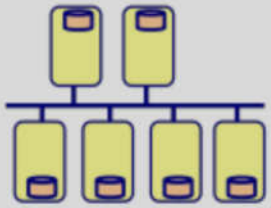
> Any fool can write code that a computer can understand. Good programmers write code that humans can understand.
> — Refactoring: Improving the Design of Existing Code, 1999
> By *Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts, and Erich Gamma*

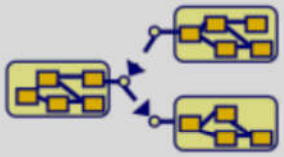
Architecture styles







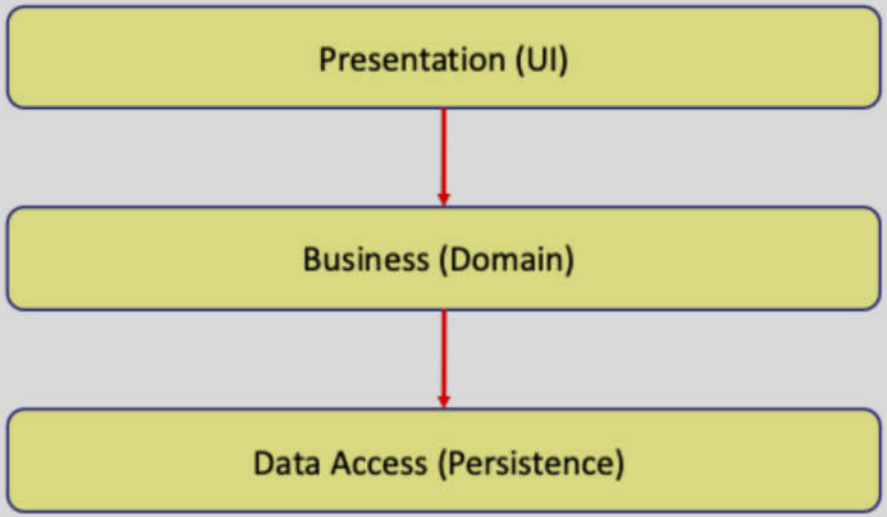




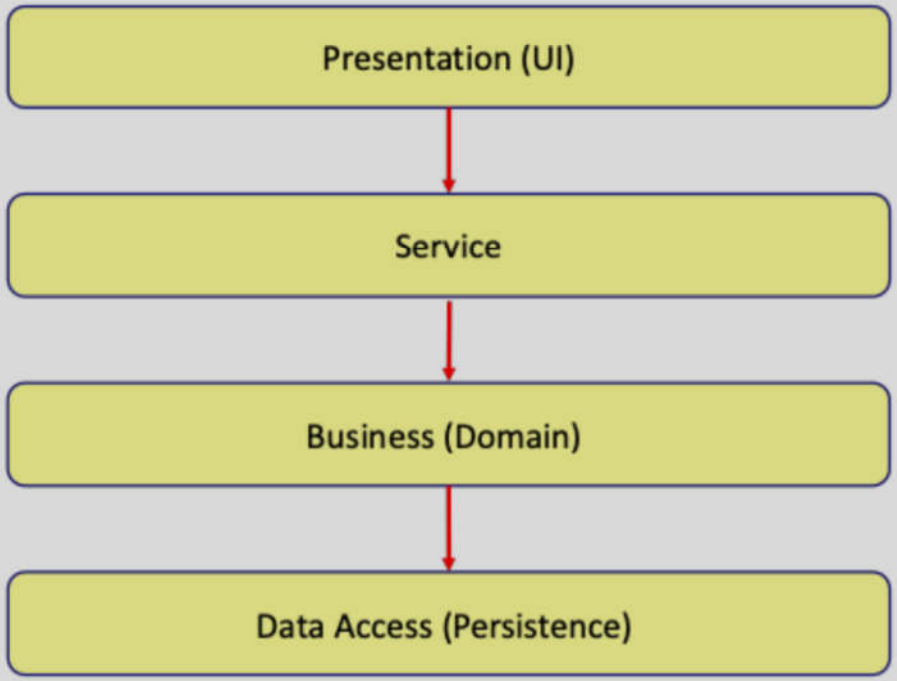
Layering

- Separation of concern
- Layers are independent
- Layers can be distributed
- Layers use different techniques

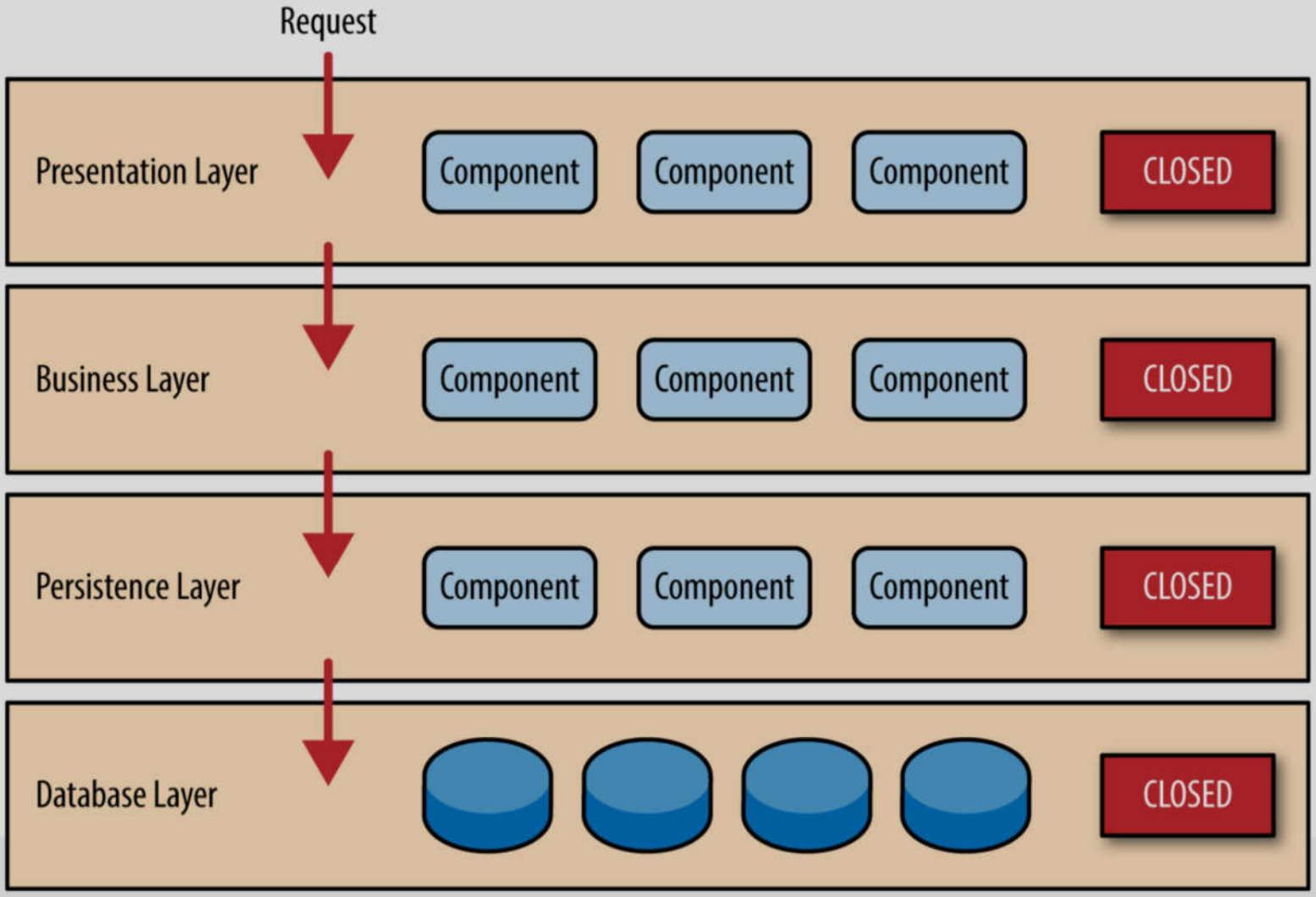
3 layered architecture



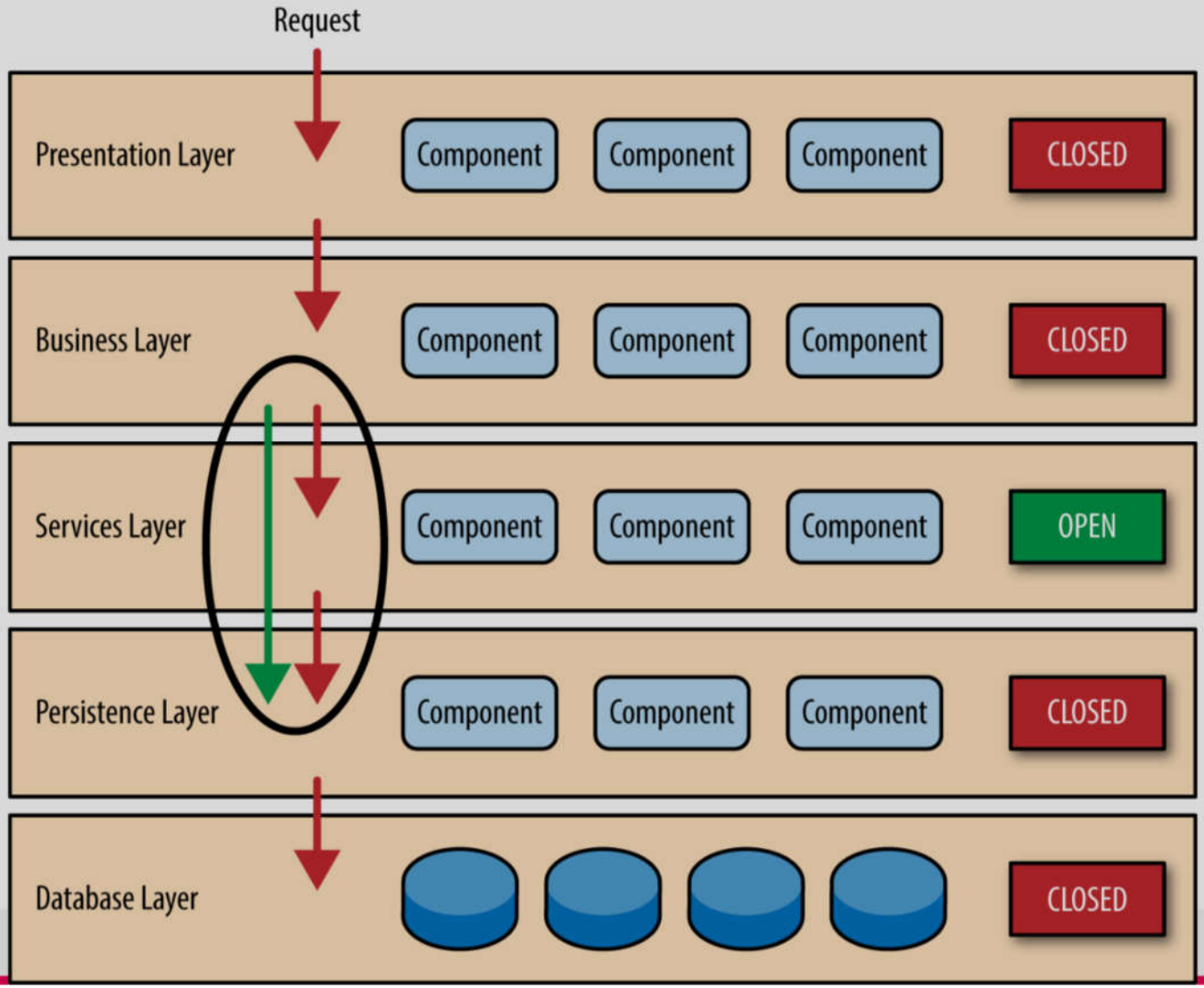
4 layered architecture



Source: https://www.oreilly.com/content/software-architecture-patterns/



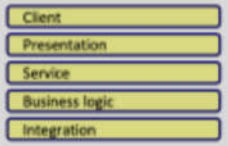
https://www.oreilly.com/content/software-architecture-patterns/



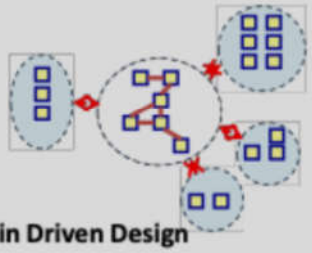
Must READ : https://www.oreilly.com/content/software-architecture-patterns/

Event-Driven Architecture

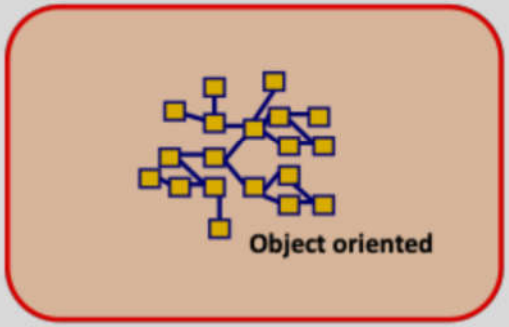
Architecture styles

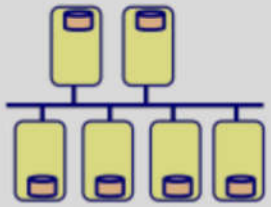


Layering

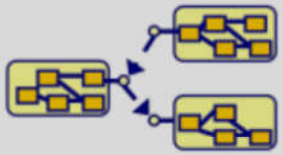


Domain Driven Design





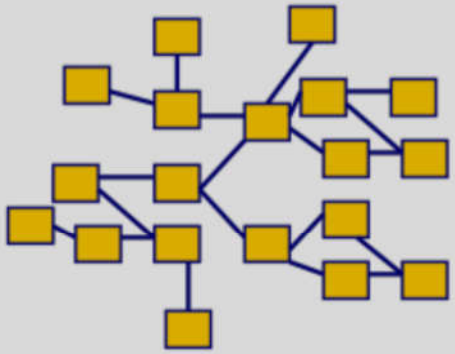
Microservices



Component based

Object Oriented

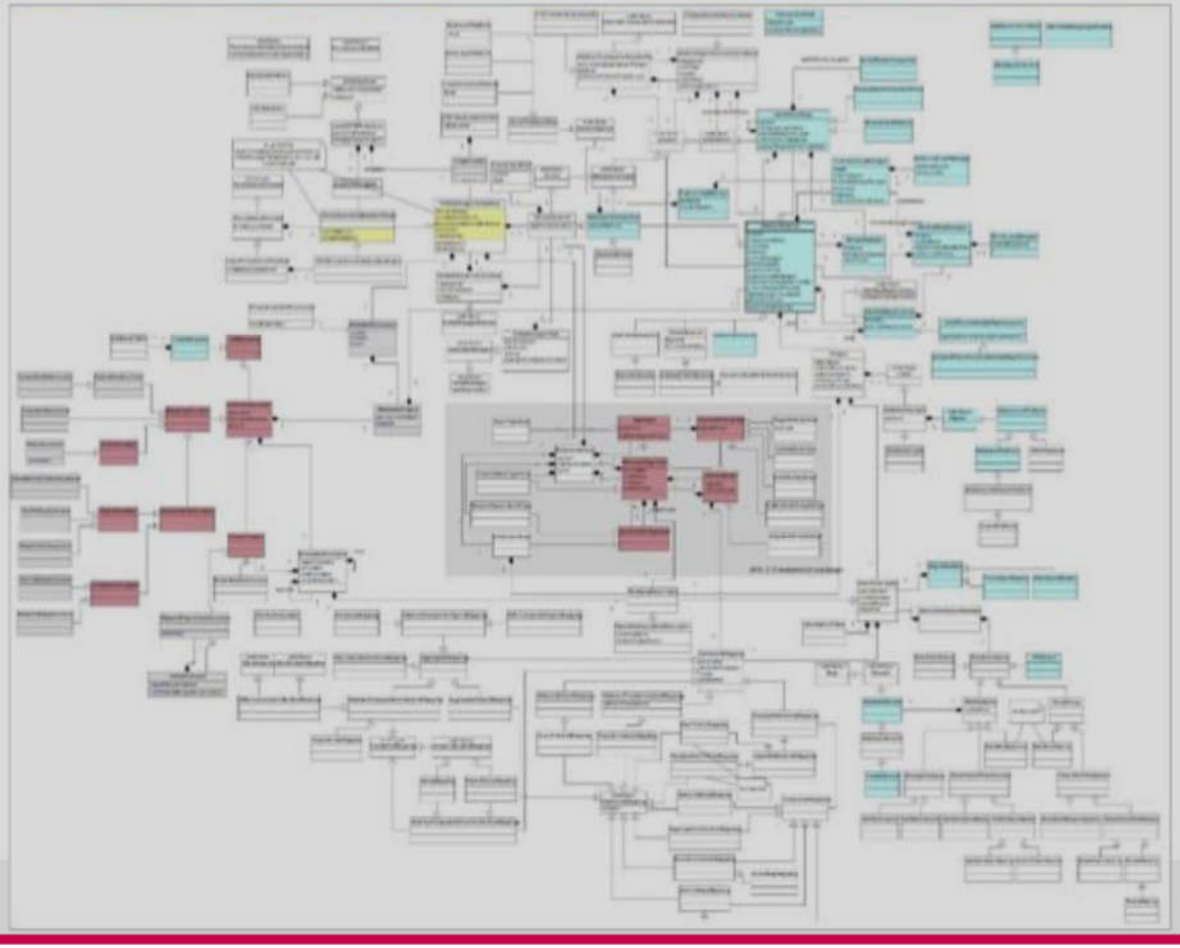
- Decompose the domain in real world objects
- Data and functionality together



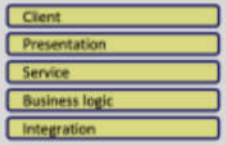
Advantages of object orientation

- High cohesion, low coupling
- Encapsulation
  - Data hiding
- Flexibility
  - Polymorphism
- Reuse ?
  - Inheritance

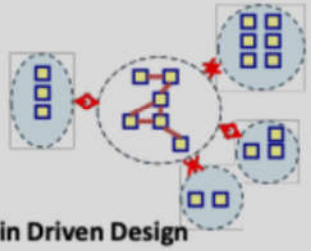
Big ball of mud



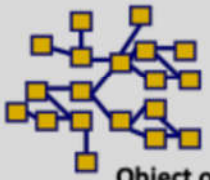
Architecture styles



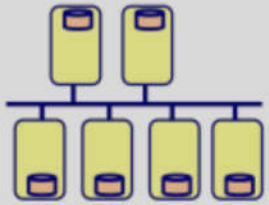
Layering



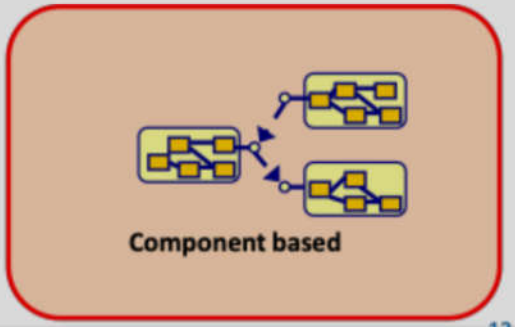
Domain Driven Design



Object oriented

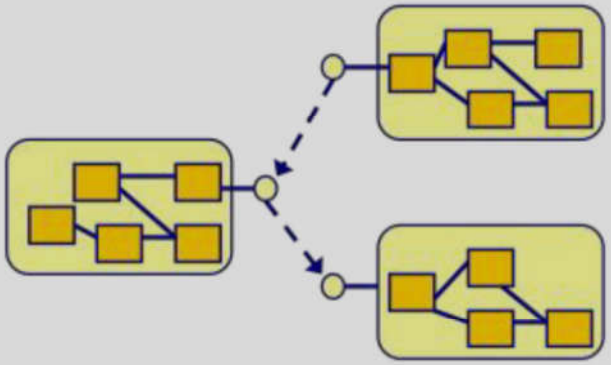


Microservices



Component Based Development

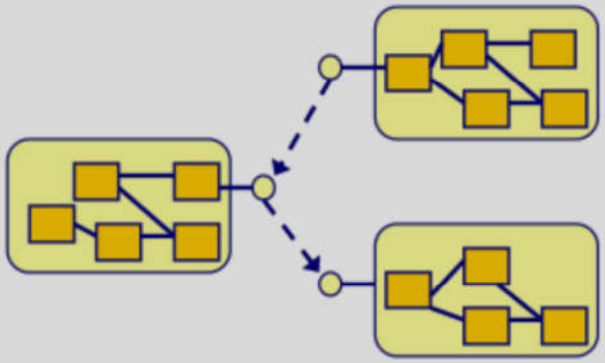
- Decompose the domain in functional components



Ref: https://www.simform.com/blog/component-based-development/

What is a component?

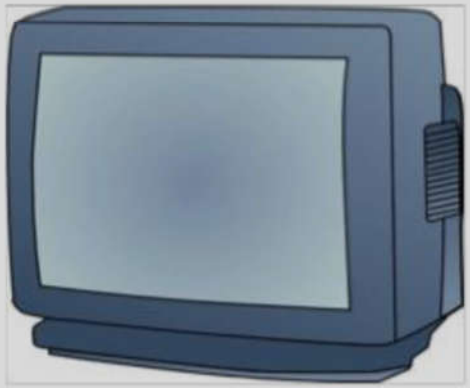
- There is no definition
- What we agree upon:
  1. A component has an interface
  2. A component is encapsulated
- Plug-and-play
- A component can be a single unit of deployment



A component is a reusable object that accelerates the development and deployment of an application while providing additional functionalities.

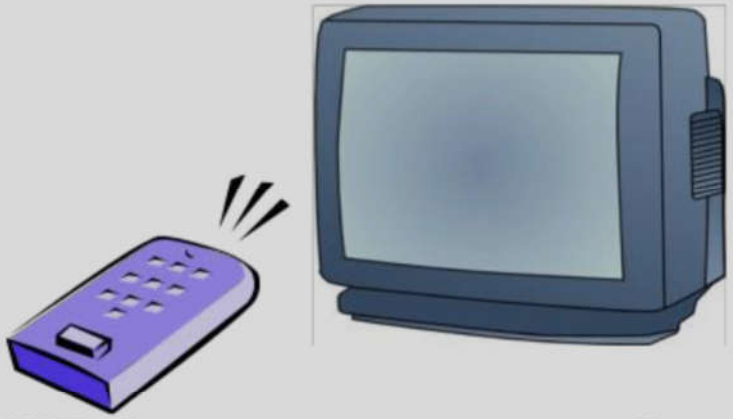
Encapsulation

- The implementation details are hidden



Interface

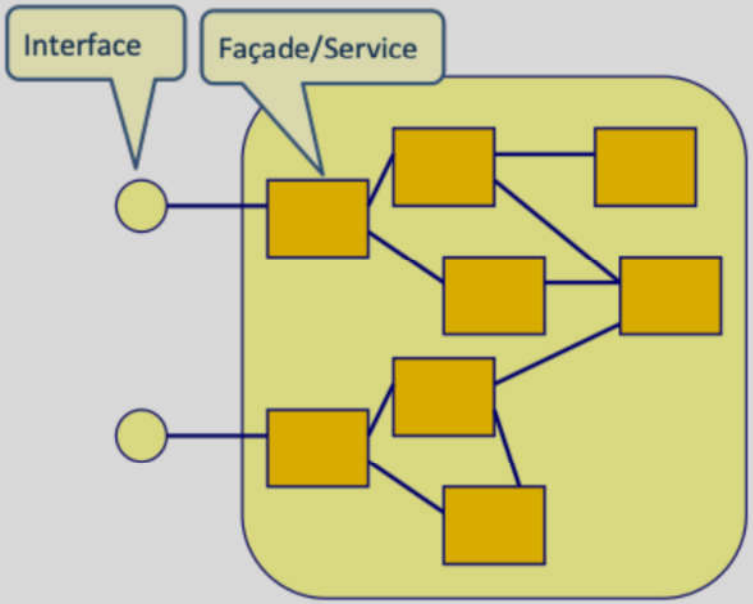
- The interface tells what you can do (but not how)



Advantages of components

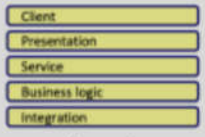
- High cohesion, low coupling
- Flexibility
- Reuse ?

Internals of a component

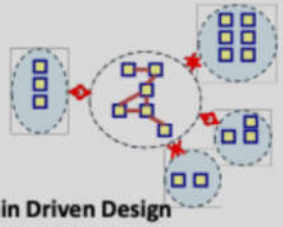


- Facade simplify interactions with complex systems by providing a unified interface.
- Facade encapsulate the complexities of the subsystem, making it easier for clients to use it without needing to understand all its details.
- Facades provide a single point of access to multiple components, reducing the number of interactions that a client needs to manage.
- By using a facade, clients can avoid direct dependencies on the underlying components.

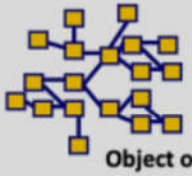
Architecture styles



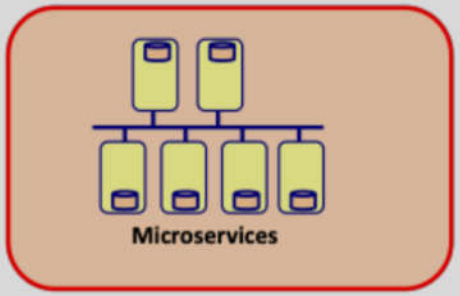
Layering



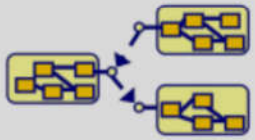
Domain Driven Design



Object oriented

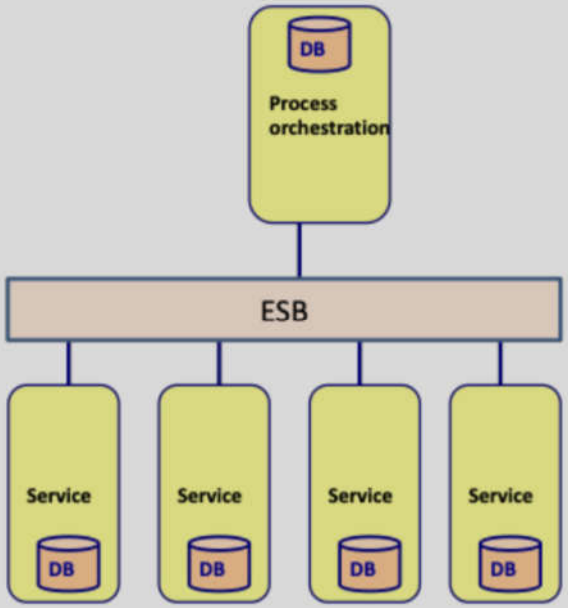


Microservices



Component based

Service Oriented Architecture

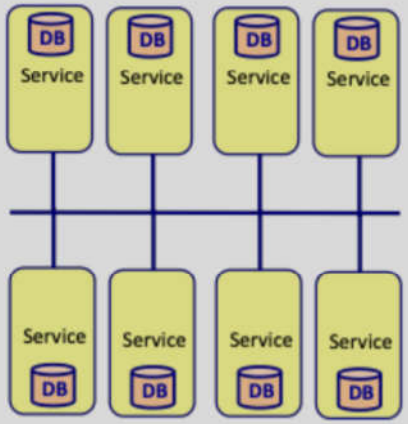


Enterprise Service Bus

- An ESB acts as a central hub for all communication between services.
- All messages and data flow through the ESB, making it a critical point in the system.
- If the ESB fails, all communication between services gets disrupted, potentially bringing down the entire system.

Microservices

- Small independent services
  - Simple and lightweight
  - Runs in an independent process
  - Technology agnostic
  - Decoupled



Summary of differences: SOA vs. microservices

	SOA	Microservices
Implementation	Different services with shared resources.	Independent and purpose-specific smaller services.
Communication	ESB uses multiple messaging protocols like SOAP, AMQP, and MSMQ.	APIs, Java Message Service, Pub/Sub
Data storage	Shared data storage.	Independent data storage.
Deployment	Challenging. A full rebuild is required for small changes.	Easy to deploy. Each microservice can be containerized.
Reusability	Reusable services through shared common resources.	Every service has its own independent resources. You can reuse microservices through their APIs.
Speed	Slows down as more services are added on.	Consistent speed as traffic grows.
Governance flexibility	Consistent data governance across all services.	Different data governance policies for each storage.

Ref: https://aws.amazon.com/compare/the-difference-between-soa-microservices/#:~:text=Service%2Doriented%20architecture%20(SOA)%20encompasses%20a%20broader%20enterprise%20scope,service%20of%20an%20ecommerce%20system.

Orchestration vs. choreography

- Orchestration
  - One central brain

- Choreography
  - No central brain

Microservices

Orchestration

- It refers to a centralized approach where a single component (often called an orchestrator) controls the interaction between multiple microservices.
- The orchestrator is responsible for managing the sequence of service calls, coordinating data flow, and ensuring that all services work together to complete a task.

Choreography

- It refers to a decentralized approach where each microservice knows how to communicate with other services and reacts to events or messages.
- In this model, services interact with each other directly based on predefined protocols or event-driven messages without a central coordinator.

Orchestration

Consider an e-commerce order processing system:

An orchestrator could handle the order process by:

- Invoking the *Inventory Service* to check stock.
- Calling the *Payment Service* if the product is available.
- Triggering the *Shipping Service* once payment is confirmed.
- Sending notifications via the *Notification Service* at the end.

Tools for Orchestration:

- Workflow engines like Camunda, Apache Airflow, or AWS Step Functions.

Choreography

Tools for Choreography:
Messaging platforms like Apache Kafka, RabbitMQ, or cloud-based services like AWS SNS/SQS.

Workflow Steps

- Order Placement:
  - The Order Service publishes an *OrderPlaced* event when a customer places an order.
- Inventory Check:
  - The Inventory Service listens for the *OrderPlaced* event, checks stock, and publishes *InventoryConfirmed* if the product is available.
- Payment Processing:
  - The Payment Service listens for *InventoryConfirmed*, processes the payment, and publishes *PaymentProcessed*.
- Shipping:
  - The Shipping Service listens for *PaymentProcessed*, initiates shipping, and publishes *OrderShipped*.
- Customer Notification:
  - The Notification Service listens for *OrderShipped* and sends a notification to the customer with tracking details.