الجامعة السورية الخاصة
SYRIAN PRIVATE UNIVERSITY

Topic 2

Software Architecture

Software Architecture Style: Layered Architecture

Dr. Riad Sonbol

Course Roadmap

- General Principles in Software Design and Architecture: *The Context of Software Architecture, System Decomposition, What is Good Design?*
- Layered Architecture
- Practical Concerns 1: Communication between Layers, Façade Design Pattern, and Dependency Injection
- N-Tier, Client-server, Peer-to-Peer, MVC
- Microservices
- Practical Concerns 2: Message Broker, Related Design Patterns: CQRS, API Gateway, Strangler Pattern, Externalized Configuration, case study
- Clean Architecture: The Hexagonal Architecture, Ports and Adapters
- More Architectural Styles: Repository, Service-Oriented Architecture (SOA), Pipes and Filters

الجامعة السورية الخاصة
SYRIAN PRIVATE UNIVERSITY

Software Architecture Style

Layered Architecture

Software Architecture Style

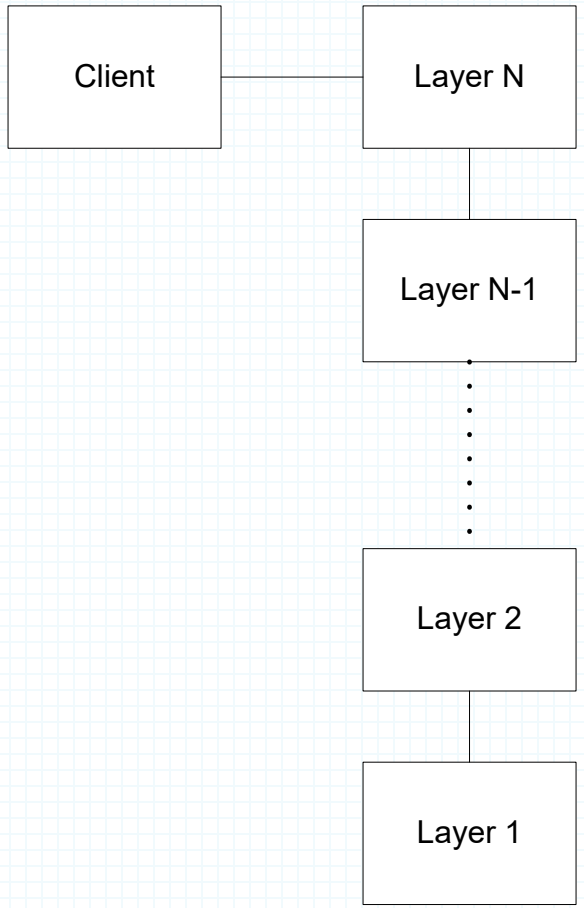
- A software architecture style is a solution to a **recurring problem** that is well understood, in a particular context.
- Each pattern consists of a **context, a problem, and a solution.**
- Using patterns simplifies design and allows us to gain the benefits of **using a solution that is proven to solve** a particular design problem.
- **More than one software architecture pattern** can be used in a single software system. These patterns can be combined to solve problems.

Example: Layered Architecture

- The Problem!
  - You are designing a system that needs to handle **a mix of low-level and high-level issues**
  - High-level operations rely on lower-level ones
  - The system is **large**, and a methodical approach to organizing it is needed to keep it understandable

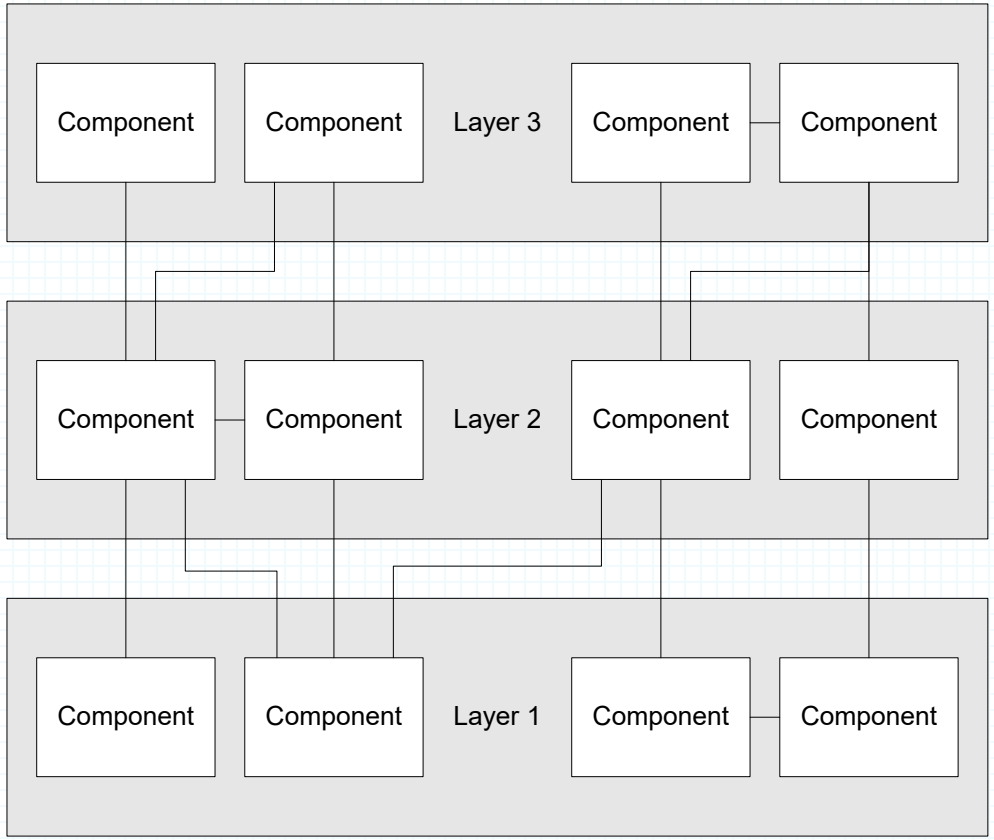
Example: Layered Architecture

- Structure the system into an appropriate **number of layers**, and place them on top of each other
- Each layer represents a **level of abstraction**
- Classes are placed in layers based on their levels of abstraction
- Layer 1 is at the bottom and contains classes **closest** to the hardware/OS
- Layer N is at the top and contains classes that **interact directly** with the system's clients



Example: Layered Architecture

- Layer J uses the services of Layer J-1 and provides services to Layer J+1
- Most of Layer J's services are implemented by composing Layer J-1's services in meaningful ways
- Layer J **raises the level of abstraction** by one level
- Classes in Layer J may also use each other



Relaxed (Open) vs. strict (Closed) Layers

RELAXED (OPEN)

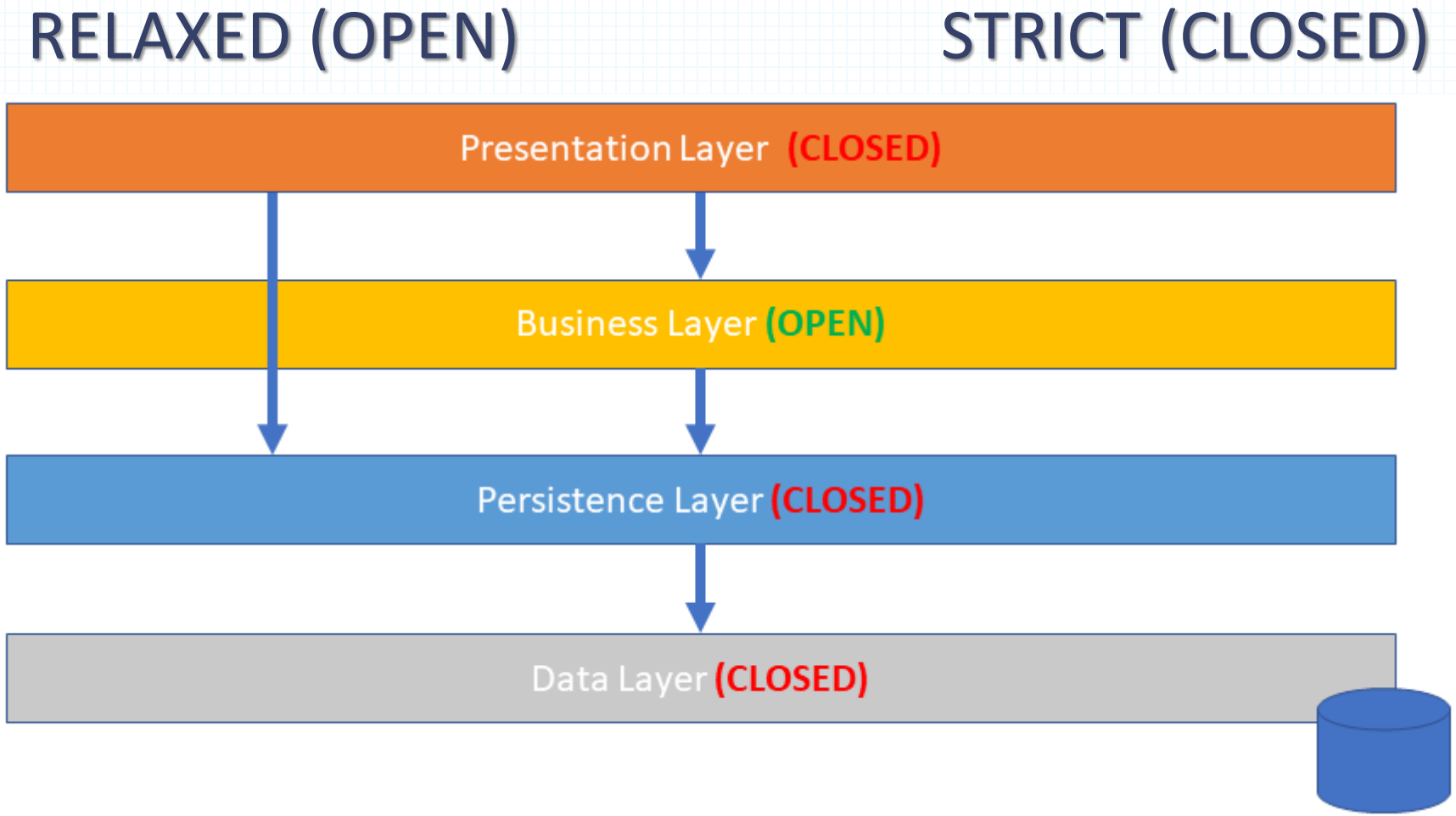
- Layer J can call directly into any layer below it, not just Layer J-1.
- **Pros:** More flexible and efficient than strict layers:
- **Cons:** Increases complexity, less understandable and maintainable than strict layers

STRICT (CLOSED)

- Layer J can ONLY call Layer J-1
- **Pros:** makes code easier to change, write, and understand.
- **Cons:** unnecessary traffic can result

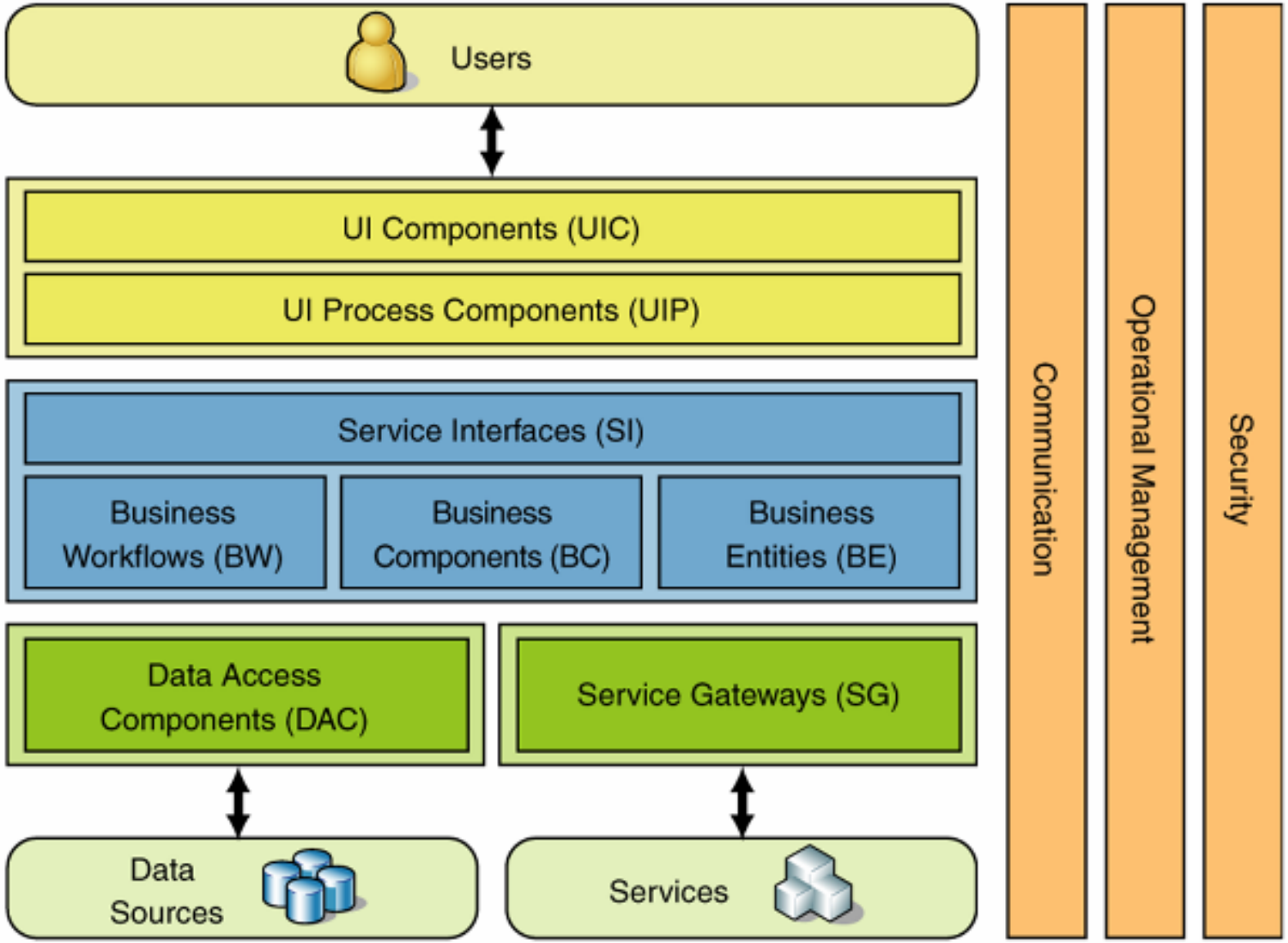
It is not necessary to make all of the layers open or closed.
You may selectively choose which layers, if any, are open.

Relaxed (Open) vs. strict (Closed) Layers

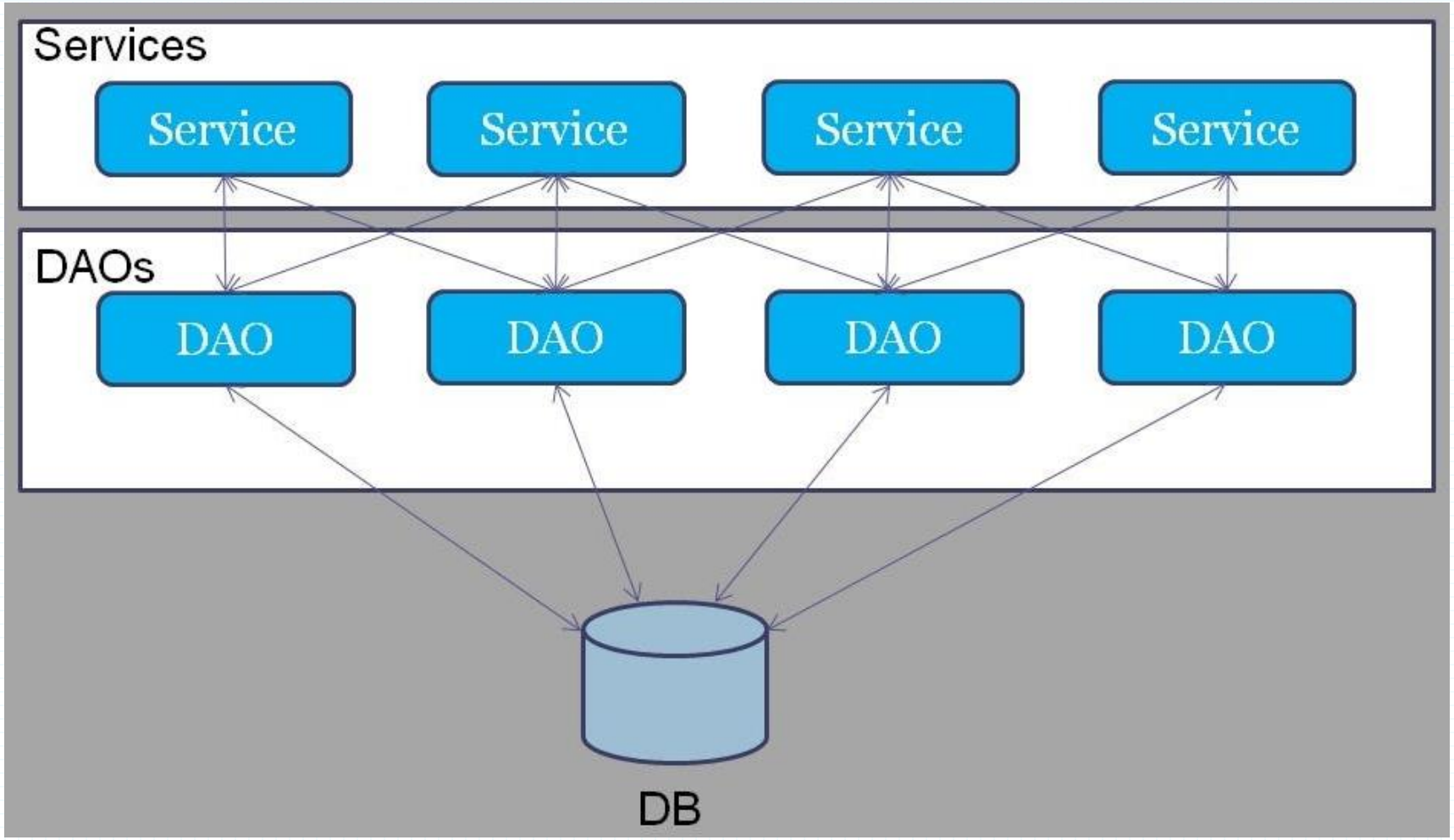


Layered Architecture

- **Presentation Layer:** Handles user interface (UI) and user experience (UX).
- **Business Layer:** Manages core application rules and logic.
- **Persistence Layer (or Data access layer):** Handles data access and storage operations.
- **Database Layer:** The underlying database storing data.

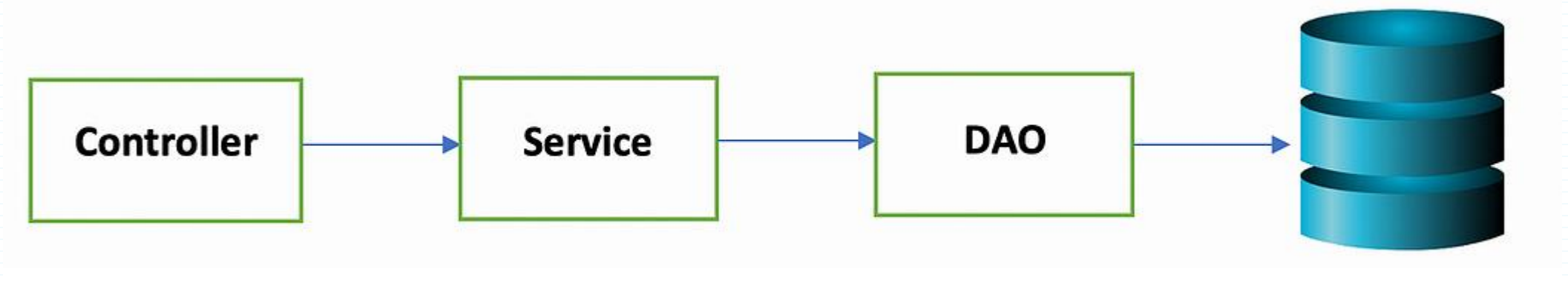


Layered Architecture: DAL



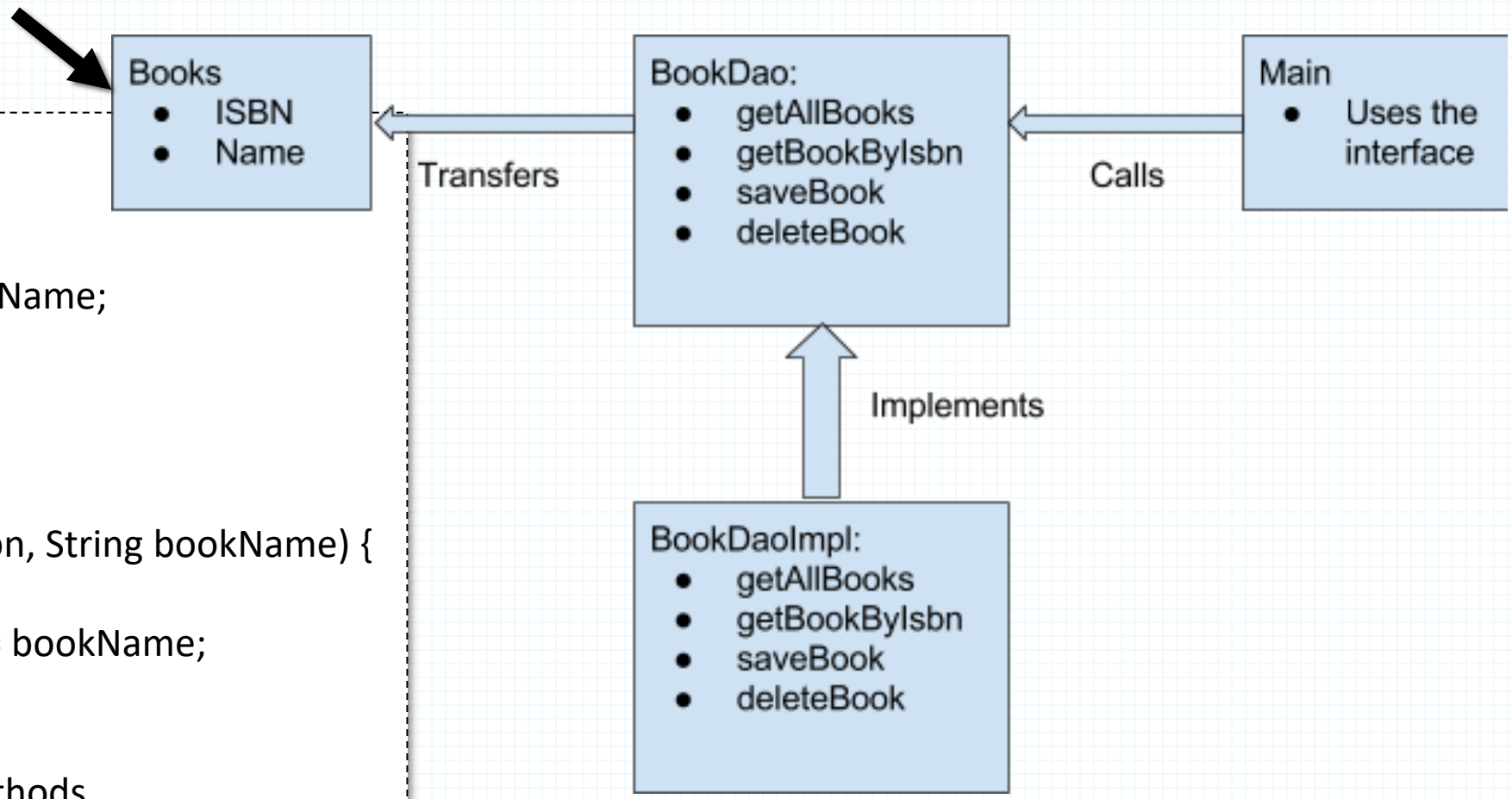
Layered Architecture: DAL

- The DAO (Data Access Object) layer is crucial in separating the database access logic from the business logic in the application.
- This means that the DAO layer hides the details of how data is stored and retrieved.

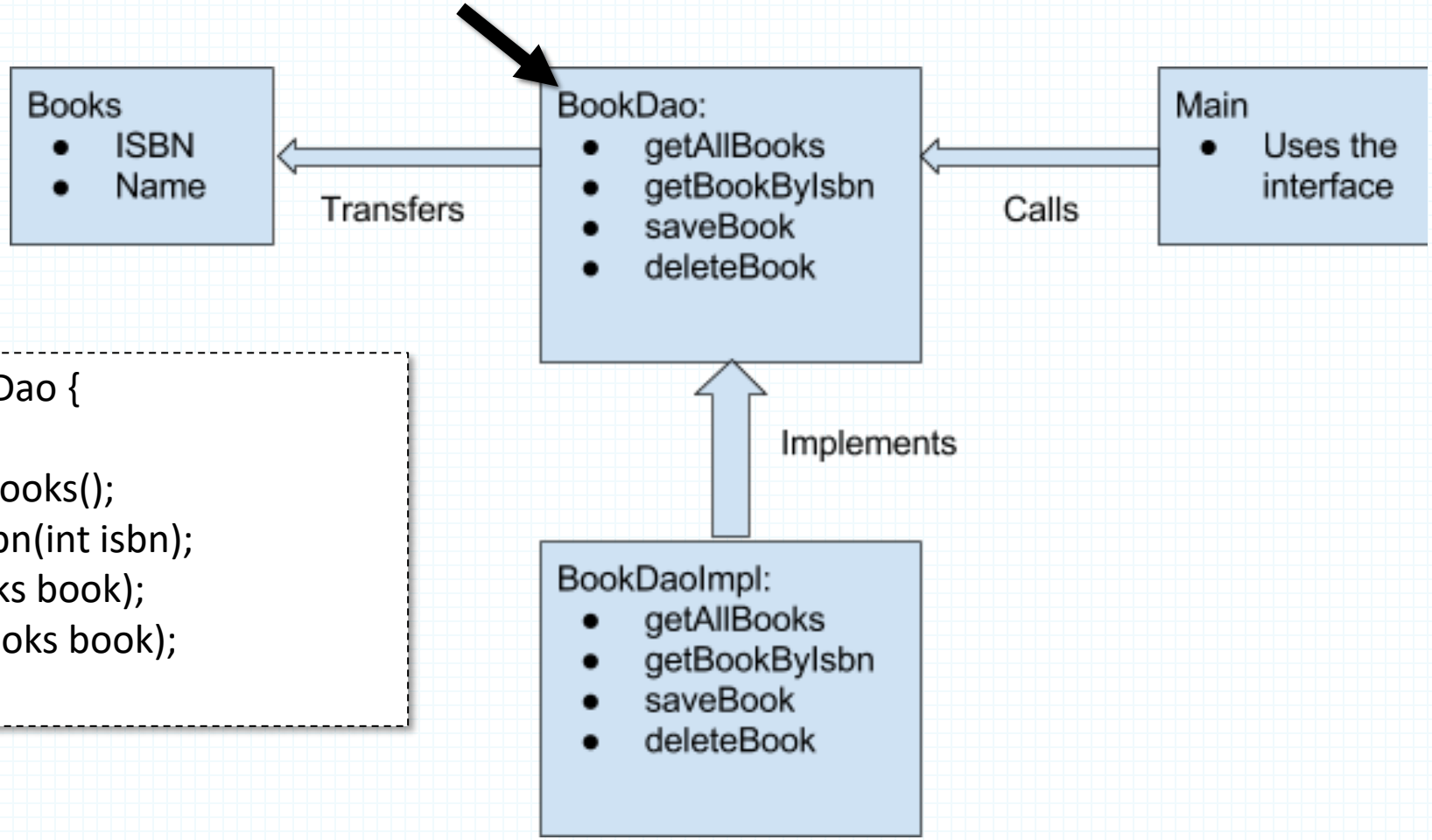


Layered Architecture: DAL

```
public class Books {

    private int isbn;
    private String bookName;

    public Books() {
    }

    public Books(int isbn, String bookName) {
        this.isbn = isbn;
        this.bookName = bookName;
    }

    // getter setter methods
}
```

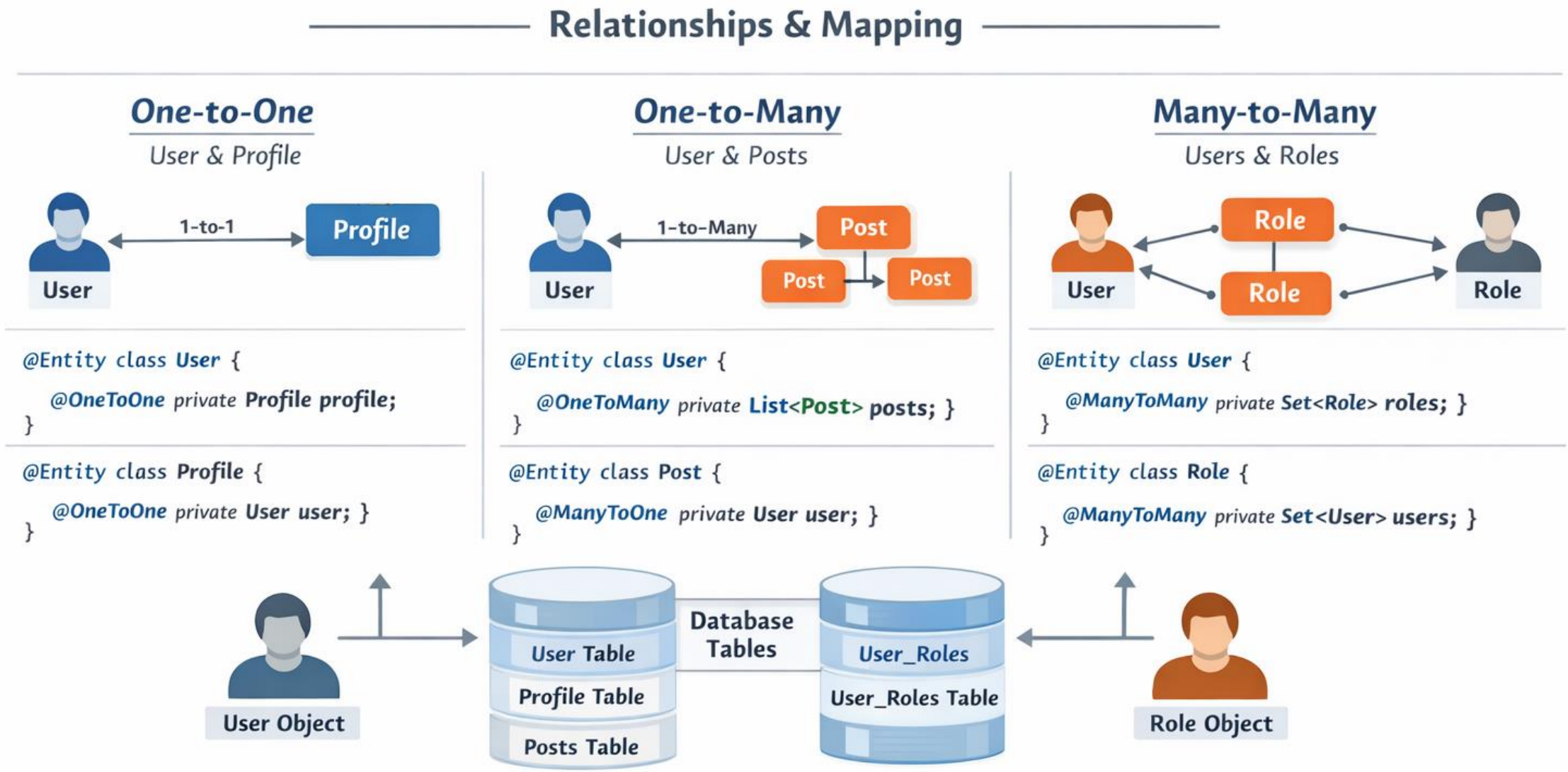


Layered Architecture: DAL



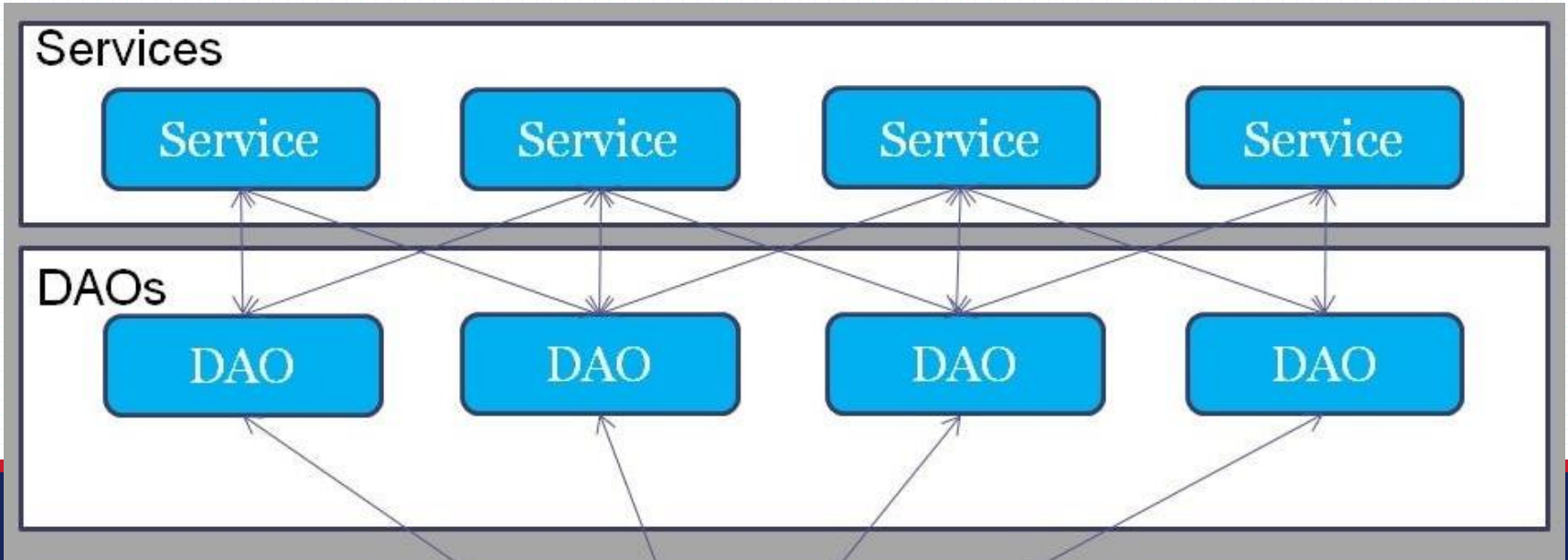

```
public interface BookDao {

    List<Books> getAllBooks();
    Books getBookByIsbn(int isbn);
    void saveBook(Books book);
    void deleteBook(Books book);
}
```

Layered Architecture: DAL



Layered Architecture: BL

- Business Logic (BL) is the part of the application that implements the rules, calculations, and workflows of your business or domain.
- It sits between the data layer (DAO/Entities) and the presentation layer (UI/API).
- Sometimes called “service layer” in modern architectures.



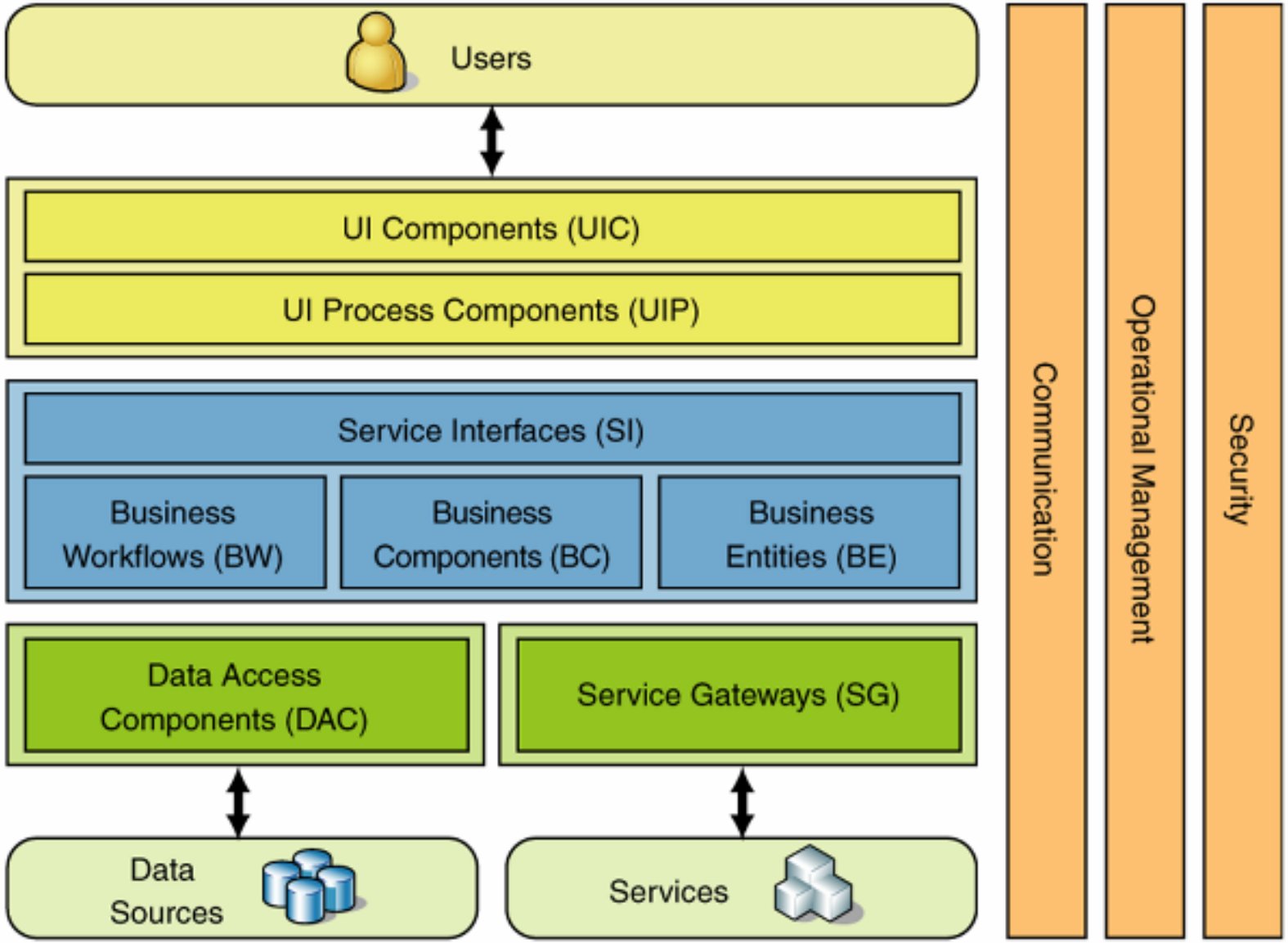
Layered Architecture: BL

BL is the brain of the application – it decides what can happen and what can't.

```java
public class UserService {

    private UserDAO userDAO = new UserDAO();

    // Business Logic: Register a new user
    public void registerUser(User user) throws Exception {
        // BL Rule: Email must be unique
        if(userDAO.findByEmail(user.getEmail()) != null) {
            throw new Exception("Email already exists!");
        }

        // BL Rule: Name must not be empty
        if(user.getName() == null || user.getName().isEmpty()) {
            throw new Exception("Name cannot be empty!");
        }

        // Save user if all BL rules pass
        userDAO.save(user);
    }

    // Business Logic: Add Post to User
    public void addPost(User user, Post post) {
        // Example: Only allow posts if user is active
        if(user.isActive()) {
            post.setUser(user);
            user.getPosts().add(post);
            userDAO.update(user);
        }
    }
}
```

Layered Architecture: PL

- **Presentation Layer:** Handles user interface (UI) and user experience (UX).



الجامعة السورية الخاصة
SYRIAN PRIVATE UNIVERSITY

Case Study 1

Example

- Design a layered architecture for a system that includes the following features:

1. **Course Management**: The ability to manage courses (add, remove, update courses).
2. **Student Management**: The ability to manage students (add, remove, update student information).
3. **Enroll Students in Courses**: Implement business rules for enrollment:
    1. A course can have no more than 50 students.
    2. A student can be enrolled in no more than 5 courses.
4. **Print Course Enrollments**: Display a list of students for any given course.

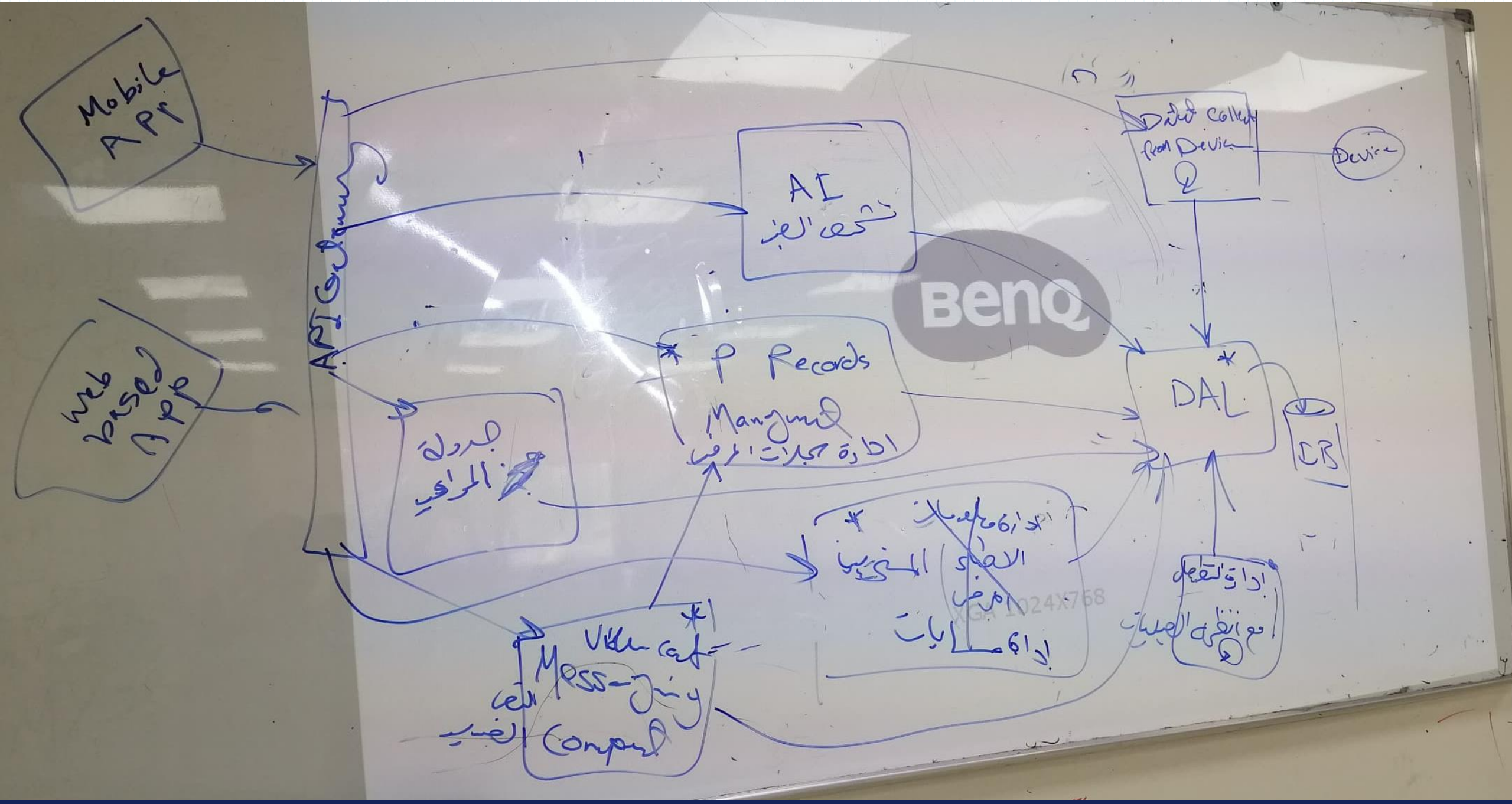
الجامعة السورية الخاصة
SYRIAN PRIVATE UNIVERSITY

Case Study 2

حالة عملية للنقاش

شركة تطوير تعمل على بناء نظام برمجي لإدارة خدمات الرعاية الصحية عن بُعد، ويتميز النظام المراد بالخصائص التالية:

- يتكامل النظام مع أجهزة قياس العلامات الحيوية للمرضى لتحصيل البيانات والقياسات الحيوية مباشرة لتحليلها.
- يوفر واجهة مستخدم تفاعلية للأطباء والممرضين تتيح لهم مراقبة الحالة الصحية للمرضى بشكل لحظي عند الحاجة وتقديم الاستشارات عبر الإنترنت.
- يتيح النظام للمرضى حجز المواعيد والتواصل مع الأطباء عبر المحادثات النصية أو مكالمات الفيديو.
- يقدم النظام توصيات طبية مخصصة بناءً على بيانات المرضى وسجلهم الصحي باستخدام تقنيات الذكاء الصنعي.
- يوفر النظام خدمة إدارة الوصفات الطبية الإلكترونية، حيث يمكن للأطباء إصدار الوصفات وتحويلها مباشرة إلى الصيدليات.
- يدعم النظام تخزين الملفات الطبية بشكل آمن مع إمكانية مشاركتها مع مقدمي الرعاية الصحية بناءً على موافقة المريض.


Mobile APP
Web based APP
API Gateway
AI
Data collect from Device
Device
P Records Manager
DAL
DB
Messaging Component

الجامعة السورية الخاصة
SYRIAN PRIVATE UNIVERSITY

Practical Concerns

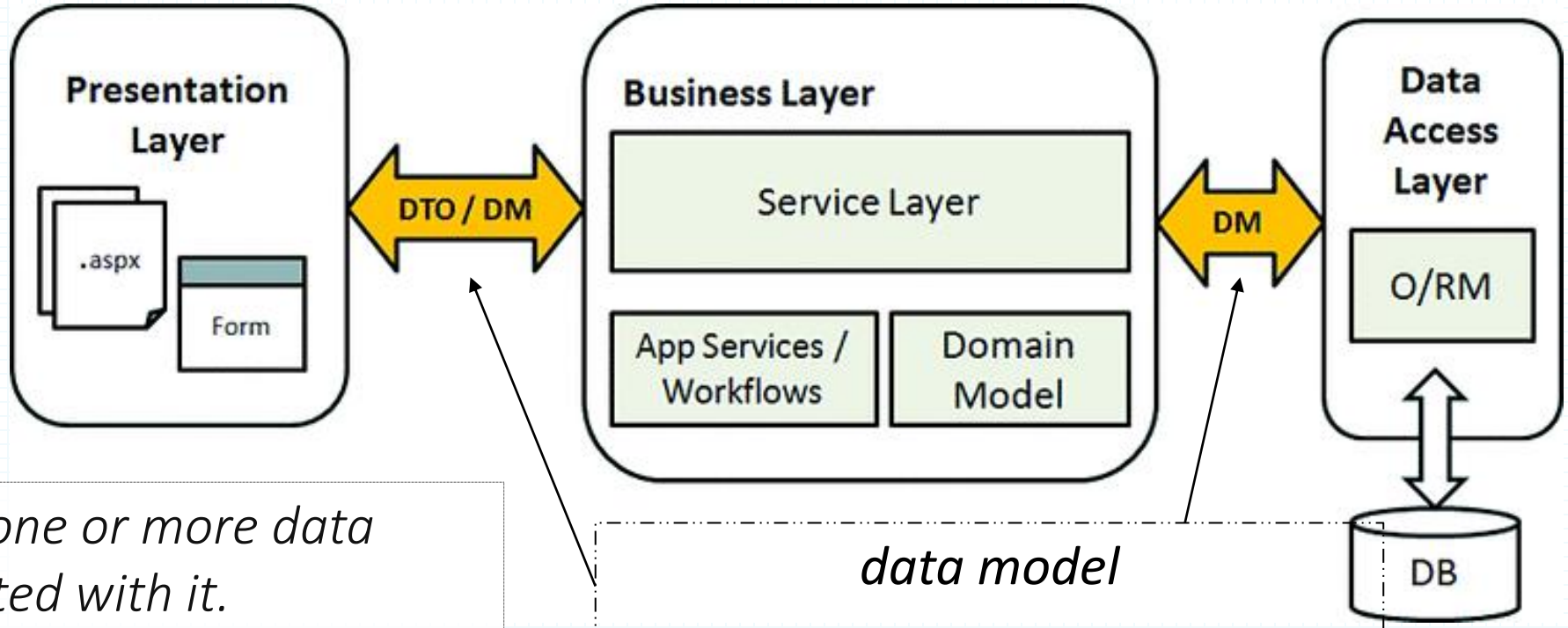
1st Practical Concern

Detailed Structure

HOW TO HANDLE THE COMMUNICATION BETWEEN LAYERS (HOW TO TRANSFER DATA?)

Application data is located in data models

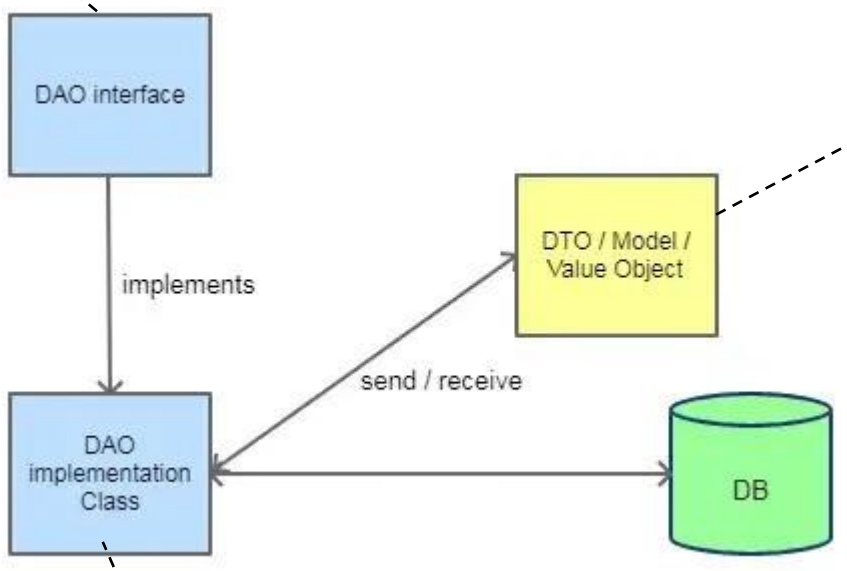
Data Transfer Object (DTO) is a pattern used to transfer data between software components



- *Each layer has one or more data models associated with it.*
- *The layer functionality can be implemented both in separate specialized classes and as methods in data model classes.*

data model
=
a collection of data structures
+
operations for processing them.

```java
interface DeveloperDao {
    public List<Developer> getAllDevelopers();
    public Developer getDeveloper(int DeveloperId);
    public void updateDeveloper(Developer Developer);
    public void deleteDeveloper(Developer Developer);
}
```




```java
// Implementing above defined interface
class DeveloperDaoImpl implements DeveloperDao {

….

}
```

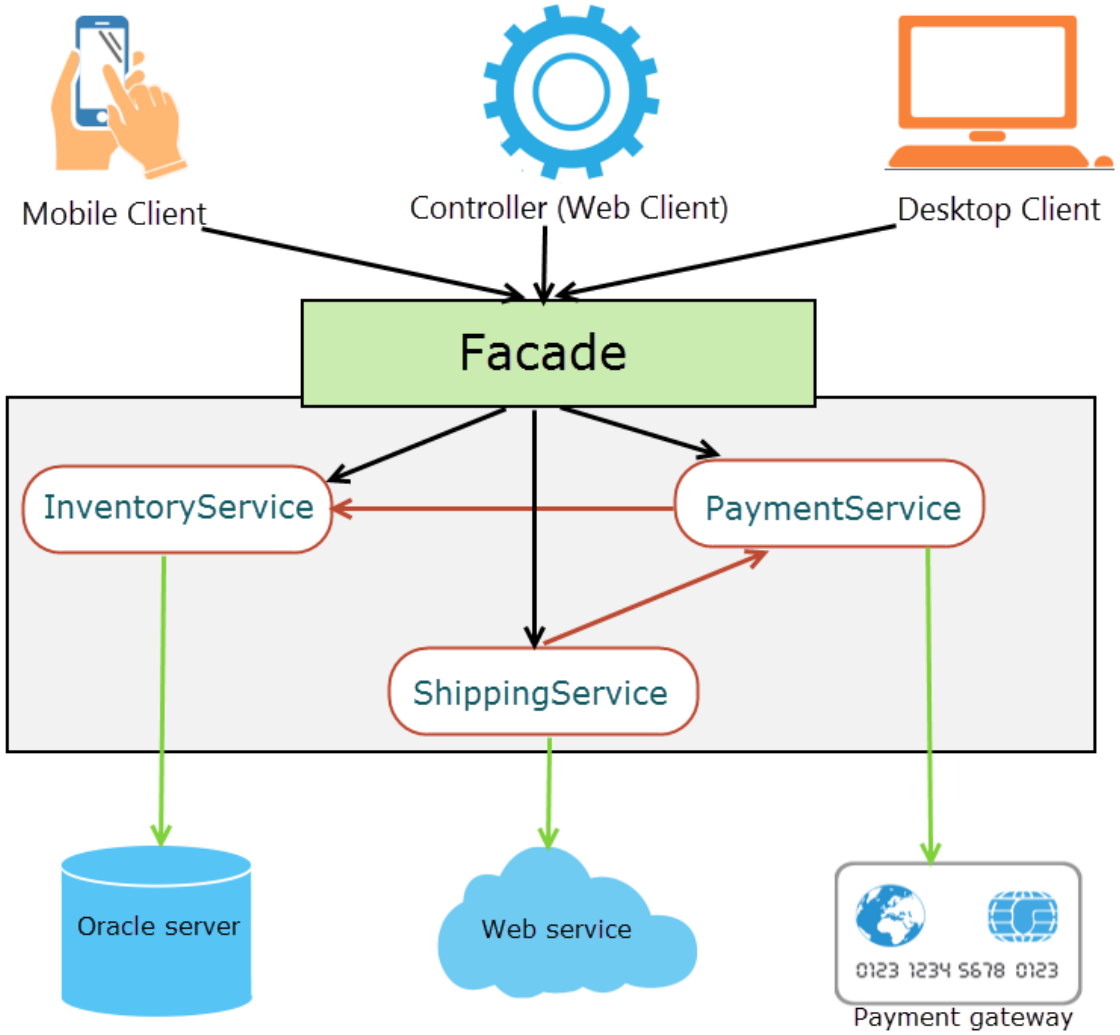
```java
class Developer {
    private String name;
    private int DeveloperId;
    // Constructor of Developer class
    Developer(String name, int DeveloperId)
    {
        this.name = name;
        this.DeveloperId = DeveloperId;
    }

    public String getName() { return name; }

    public void setName(String name) { this.name = name; }

    public int getDeveloperId() { return DeveloperId; }

    public void setDeveloperId(int DeveloperId)
    {
        this.DeveloperId = DeveloperId;
    }
}
```

2st Practical Concern

Detailed Structure

FAÇADE DESIGN PATTERN

Façade Design Pattern



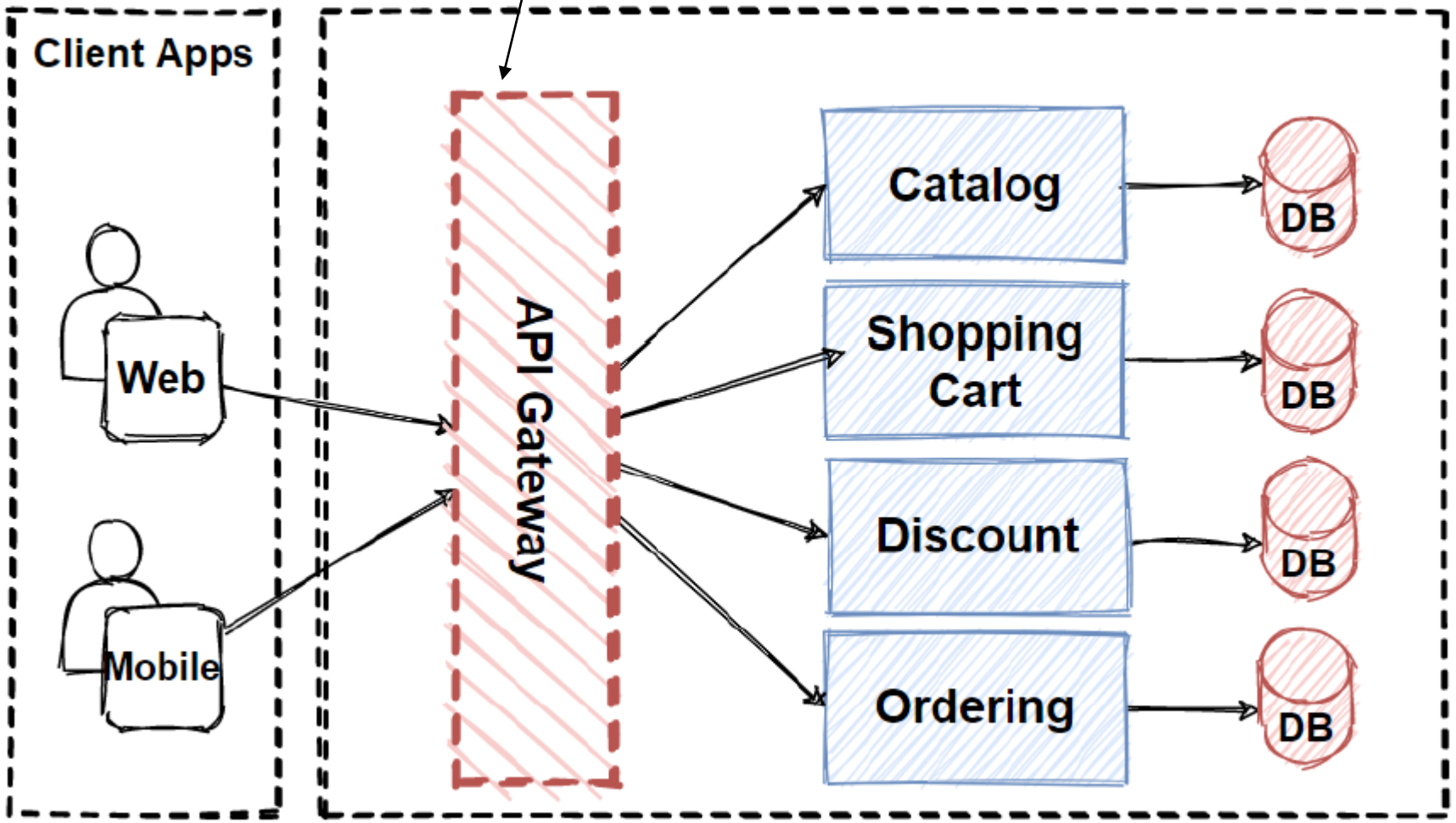
Façade sublayer encapsulates the functionality of the layer's functionality and provides high-level methods or operations that abstract away the complexity

Services implemented by subsystem classes

External Resources

Example

it provides a single entry point to the APIs with encapsulating the underlying system architecture.



3rd Practical Concern

Dependency Injection

Changing a service would imply changing a lot of the codebase, especially if the service has been used in multiple parts of the project.

Ex: if the email service is replaced with a new one (OutlookEmailServie, etc)

```
public class UserLogic
{
    private GoogleOAuthService _authService;
    private GoogleEmailService _emailService;

    public UserLogic()
    {
        _authService = new GoogleOAuthService();
        _emailService = new GoogleEmailService();
    }

    public void Register(string emailAddress, string password)
    {
        var authResult = _authService.RegisterUser(emailAddress,password);
        _emailService.SendMail(emailAddress, authResult.ConfirmationMessage);
    }
}
```

```
public class GoogleOAuthService
{
    public GoogleOAuthResult RegisterUser(string emailAddress, string password)
    {
        //Register a new user
    }
}
```

```
public class GoogleEmailService
{
    public SendMail(string emailAddress, string message)
    {
        //Send an email using google
    }
}
```

High-level modules should depend on abstractions rather than concrete implementations

```
public interface IEmailService
{
    void SendMail(string emailAddress, string message)
}
```

As a Contract

BOTH depends on abstraction!

```
public class GoogleEmailService: IEmailService
{
    public SendMail(string emailAddress, string message)
    {
        //Send an email using google
    }
}

public class OutlookEmailService: IEmailService
{
    public void SendMail(string emailAddress, string message)
    {
        //Send an email using outlook
    }
}
```

```
public class UserLogic
{
    private GoogleOAuthService _authService;
    private IEmailService _emailService;

    public UserLogic()
    {
        _authService = new GoogleOAuthService();
        _emailService = new OutlookEmailService() // or Google;
    }

    public void Register(string emailAddress, string password)
    {
        var authResult = _authService.RegisterUser(emailAddress,password
        _emailService.SendMail(emailAddress, authResult.ConfirmationMess
```

Better.. But Still Tightly Coupled

- Classes request dependencies instead of referencing directly
- Dependent object instances are injected

1. Constructor Injection

```
public class UserLogic
{
    private GoogleOAuthService _authService;
    private IEmailService _emailService;

    public UserLogic(IEmailSevice emailService)
    {
        _authService = new GoogleOAuthService();
        _emailService = emailService;
    }
    ...
}
```

```
...
    GoogleEmailService googleEmailService = new GoogleEmailService();
    UserLogic userLogic = new UserLogic(googleEmailService);
...
```

- Classes request dependencies instead of referencing directly
- Dependent object instances are injected

2. Setter Injection

```
public class UserLogic
{
    private GoogleOAuthService _authService;
    private IEmailService _emailService;

    public IEmailService EmailService
    {
        get
        {
            return _emailService;
        }
        set
        {
            _emailService = value;
        }
    }
```

- Classes request dependencies instead of referencing directly
- Dependent object instances are injected

2. Method Injection

```
public class UserLogic
{
    private GoogleOAuthService _authService;

    public UserLogic()
    {
        _authService = new GoogleOAuthService();
        _emailService = new OutlookEmailService() // or Google;
    }

    public void Register(string emailAddress, string password, IEmailService emailService)
    {
        var authResult = _authService.RegisterUser(emailAddress,password);
        emailService.SendMail(emailAddress, authResult.ConfirmationMessage);
    }
}
```

Course Roadmap

- General Principles in Software Design and Architecture: *The Context of Software Architecture, System Decomposition, What is Good Design?*
- Layered Architecture
- Practical Concerns 1: Communication between Layers, Façade Design Pattern, and Dependency Injection
- N-Tier, Client-server, Peer-to-Peer, MVC
- Microservices
- Practical Concerns 2: Message Broker, Related Design Patterns: CQRS, API Gateway, Strangler Pattern, Externalized Configuration, case study
- Clean Architecture: The Hexagonal Architecture, Ports and Adapters
- More Architectural Styles: Repository, Service-Oriented Architecture (SOA), Pipes and Filters