SOLID

Single Responsibily Principle (SRP)

•A class/module/service should have one reason to change
• Another way to define loose coupling and high cohesion
1 reason to change → 1 job per class. Long methods = smell
• Benefits:

- Easier Maintenance
- Enhanced Flexibility
- Simpler Testing

Good Example:

- Order Class: Manages order details.
- PaymentService Class: Handles payments.
- NotificationService Class: Sends notifications.

Bad Example:

- Order Class: Manages order details, processes payments, handles inventory, and sends notifications

Open-Closed Principle

• Software modules should be open for extension but closed for modification.
• Purpose: Allow the behavior of a module to be extended without modifying its source code.
Open to add, closed to edit → Strategy pattern
Benefits:
• Modification increases coupling, extension reduces coupling
• Promotes Reusability: Extending functionality without altering existing code.
• Improves Maintainability: Reduces risk of introducing bugs when adding features.
• Facilitates Testing: Easier to test new functionality without affecting existing code.

Good OCP:
• Notifier Interface/Class: Extended by Email, Push, SMS, etc.
• NotificationService: Sends notifications using the Notifier without knowing specific notification types.
Bad OCP:
• Notifier Class - Sends notification via all channels and decides which channel to use

Liskov Substitution Principle (LSP)

• Subtypes must be substitutable for their base types without altering the correctness of the program.
Subclass replaces parent w/o breaking. Don't override to throw

Interface Segregation Principle

• Don't force the client to depend on things they don't use.
Don't force unused methods. Avoid "fat" interfaces

Dependency Injection

• Give a software module/class what it needs as arguments instead of having the module create it.
Depend on abstractions. Use interfaces, not concrete classes

```
//myObject can't be injected
public SomeClass(){
myObject = Factory getObject();}
//myObject can be passed in
public SomeClass (MyClass myObject){
this.myObject = myObject;}
```

Patterns

Creational Patterns

• Deal with object creation mechanisms, optimizing flexibility and reuse.
• Singleton: Ensures a class has only one instance (beware threading issues).
• Factory Method: Creates objects without specifying the exact class (Central place to create subclasses).
• Builder: Constructs (creates) complex objects step by step.

Structural Patterns

• Help organize classes and objects to form larger structures.
• Adapter: Converts one interface to another (Converts APIs/interfaces).
• Decorator: Adds behavior to objects dynamically (Adds runtime behavior).
• Facade: Simplifies complex systems with a unified interface.

Behavioral Patterns

• Concerned with object interaction and responsibility delegation.
• Observer: Defines a subscription mechanism for event changes (Listeners update on change).
• Strategy: Enables selecting an algorithm at runtime (Swappable behavior; OCP-friendly).
• Command: Encapsulates requests as objects to parameterize actions (Encapsulate action as object (undo/redo)).

Domain Driven Design (DDD)

Core Concepts

Domain
• The subject area of the software.
• Crucial to understand the business context (e.g., healthcare, e-commerce, education).

Example: In a healthcare domain, you have subdomains like patient management, appointment scheduling, billing, etc.

Bounded Context
• A semantic boundary where a domain model applies.
• Defines the meaning of terms within that context.
• Reduces ambiguity.

Example: The term "Product" means different things in "Sales" (features, marketing) vs. "Inventory" (stock levels, storage).

Ubiquitous Language: Shared dev + domain terms in code

Building Blocks

Aggregate
• A cluster of related objects treated as a single unit (Root + controlled children (e.g., Order w/ OrderItems)).
• Has an aggregate root that controls access.
• Ensures data consistency within the boundary.
Example: Order (includes Orderltems), Course (includes Assignments).
• Key takeaway: Aggregates control data changes to maintain integrity

• Entity: An object with a unique identity. Identity persists over time. Mutable. (Patient, Course, Order, StudentAccount)

• Value Object: Replaced, not modified. Immutable. No unique identity. An object defined by its attributes. (Address, Money, DateRange, Grade)

• Domain Service: Operation that doesn't belong to an entity or value object. Look for operations, not data. Stateless. Performs actions. Identify stateless behavior. (ShippingRateCalculator, CourseRegistrationService, GradePostingService)
Key takeaway: Domain Services perform actions.

• Domain Event: A significant occurrence in the domain. Triggers actions. Used for communication between Bounded Contexts. (OrderPlaced, PaymentReceived, CourseCompleted)

• Repository: Save/load aggregate from persistence. Mechanism to persist and retrieve aggregates. Abstracts away database details.

Runtime Architecture

Concurrency & Asynchronicity

• Concurrency: Running multiple tasks seemingly at once (e.g., threads, coroutines). - Async/await: Great for I/O - Threads/pools: Best for CPU-bound - Coroutines: Lightweight concurrency
• Asynchronicity: Start a task, move on — get notified when it finishes. - Improves UI responsiveness - Boosts scalability (e.g., handle multiple users)
UI Benefits:
  • UI stays responsive (no blocking main thread) Asynchronous.
Scalability Benefits:
  • System scales better under load (More users/requests handled in parallel) Concurrency.

Applying Concurrency in Transactional Domains:
• Break transactions into clear steps: Identify the distinct stages involved in a complete operatio (e.g., check → reserve → confirm).
• Identify which steps can run independently: (do in parallel).
For example, in reservation:
 -Checking availability for different users can often happen concurrently.
 -But reserving a specific resource can't be done concurrently
• Keep UI smooth by using background threads for heavy tasks.
 Offload long-running or potentially blocking steps to background threads or asynchronous tasks to keep the user interface interactive.
• Optimizing Asynchronous Operations: Async tasks = faster execution (parallel > sequential).

Concurrency models:
Event-loop based concurrency (with async/await): Efficient for handling a large number of concurrent, mostly I/O-bound connections with a single thread. (Threaded, Coroutines)

Ensuring Data Integrity: The Role of Locking
• The Problem of Concurrent Access: Simultaneous modification of shared data can lead to inconsistencies. (avoids bugs or bad data). Simultaneous access to shared data = inconsistencies.
• Locks as a Solution: Locks = Safety: Ensure only 1 thread modifies a resource at a time.
• Granularity of Locks: The scope of the lock impacts concurrency
  • Fine-grained lock: Only lock specific resource (Maximizes concurrency)
  • Coarse-grained lock: Lock big areas of data → safer but slower (Limits concurrency).
• Transactional Consistency: Locks help keep transactions consistent by supporting ACID rules (Atomicity, Consistency, Isolation, Durability).

Coupling in Distributed Systems
• Common Coupling: Two services share the same data or resource (e.g., same database).
• Temporal Coupling: One service depends on another being available right away—if one fails or is slow, both break.
• Domain Coupling: A necessary link because of the business logic (e.g., placing an order must involve payment).
• Pass-through Coupling: A service just forwards data it doesn't use or understand—adds complexity without value.

Bottlenecks in Scalable Systems
Databases often become bottlenecks because:
  • Limited Connections: They can only handle a limited number of connections at once.
  • Transactional Overhead: Maintaining consistency (like using locks) slows things down when traffic is high.

Runtime Architecture vs. Code Structure
• Runtime Architecture = Concerns the system's behavior and interactions during execution (e.g., concurrency, data flow).
• Code Structure = Relates to how the codebase is organized and how different modules depend on each other at the source code level (compile time). These are distinct concepts.