**day26\_107856406\_dsdipt\_sudipto\_13august2025**

**Employee Code:** 107856406

**Login ID:** dsdipt

**Email :** dsdipt@amazon.com

**Name:** Sudipto Das

**Date:** 13 August 2025 (Day 26)

### ***Task 00: Architecture & Pattern Quick Notes***

**Strategy Pattern (Behavioral DP)**

We define a family of algorithms, encapsulate each in its own class, and make them interchangeable without changing client code.

**Key Points:**

* Uses *Strategy interface* + *Concrete Strategies*.
* Allows runtime switching of algorithms.
* Promotes open/closed principles (easy to add new strategies).

**When to Use:**

* When we have multiple ways of doing something and want to choose at runtime.
* Example: Choosing different sorting algorithms or payment methods.

**Command Pattern (Behavioral DP)**

We encapsulate a request in an object so we can queue, log, and undo operations without tightly coupling sender and receiver.

**Key Points:**

* Components: Command, ConcreteCommand, Receiver, Invoker.
* Supports undo/redo.
* Decouples request invoker from request executor.

**When to Use:**

* When we need to support history, undo/redo, or command queuing.
* Example: Text editor, task schedulers.

**Microservices**

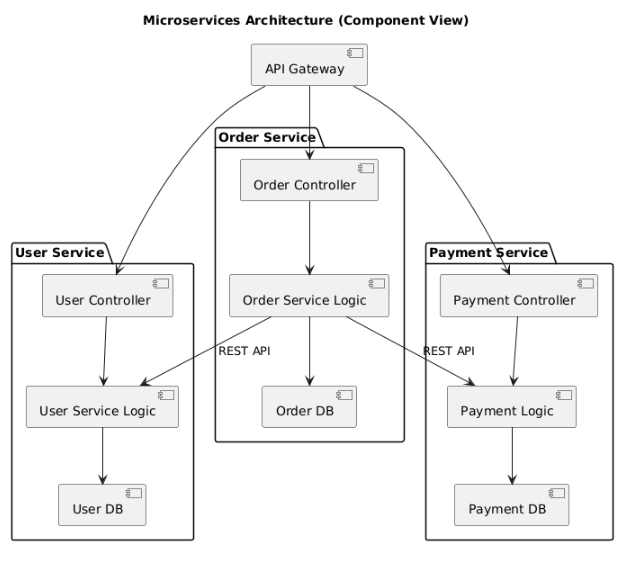
We build an app as a collection of small, independent services, each with its own database and API.

**Key Points:**

* Services communicate via REST/gRPC/messaging.
* Each service can be deployed and scaled independently.
* Fault isolation — one service failing doesn’t crash the whole app.

**When to Use:**

* Large-scale apps needing scalability and independent deployments.
* Example: E-commerce with separate Auth, Catalog, and Payment services.



**Event Sourcing Pattern**

We store state changes as a series of events instead of only storing the final state.

**Key Points:**

* Can replay events to rebuild the state.
* Provides complete audit history.
* Works well with CQRS (Command Query Responsibility Segregation).

**When to Use:**

* When we need audit logs, history, or “time travel” debugging.
* Example: Banking, trading systems.

**Circuit Breaker Pattern**

We prevent cascading failures by stopping calls to a failing service after repeated failures, and retrying after a cooldown.

**Key Points:**

* States: Closed, Open, Half-Open.
* Protects the system from overload.
* Improves resilience.

**When to Use:**

* When dependent services may fail or slow down.
* Example: Payment gateway failure handling.

**Pub-Sub Pattern**

We decouple senders and receivers by publishing messages to a topic that subscribers listen to.

**Key Points:**

* Publishers don’t know subscribers.
* Supports multiple subscribers for the same event.
* Often implemented with message brokers (Kafka, RabbitMQ).

**When to Use:**

* Event-driven systems where components must react to changes.
* Example: Order notifications, chat systems.

**Three-Tier Architecture**

We split an application into three layers: Presentation (UI), Application (business logic), and Data (database).

**Key Points:**

* Clear separation of concerns.
* Easier to maintain and scale.
* Layers communicate only with the layer directly below or above.

**When to Use:**

* Traditional web or enterprise applications.
* Example: React → Spring Boot → MySQL.

**Multi-Tier Architecture**

We extend the three-tier concept to more layers for specialized purposes like caching, APIs, analytics.

**Key Points:**

* Can include 4, 5, or more layers.
* Improves modularity and scalability.
* Each tier handles a specific responsibility.

**When to Use:**

* Large, complex systems requiring multiple specialized layers.
* Example: UI → API Gateway → Service Layer → Cache → Database.

### ***Task 01: What are the Advantages of Multi tier Architecture?***

**Separation of Concerns**

* Each tier handles a specific responsibility (UI, business logic, data, etc.).
* Makes the system easier to understand and maintain.

**Scalability**

* We can scale individual tiers independently based on load (e.g., scale the API layer without touching the database).

**Flexibility & Modularity**

* We can modify or replace one tier without affecting others as long as the interfaces remain consistent.

**Maintainability**

* Clear structure makes debugging, testing, and updating code easier.

**Reusability**

* Business logic or services in the middle tiers can be reused by multiple applications (e.g., web + mobile).

**Security**

* Sensitive operations (like DB queries) can be isolated in the backend tiers, reducing direct access risk.

**Technology Independence**

* Different tiers can use different technologies (e.g., React for UI, Java Spring Boot for business logic, MySQL for database).

### ***Task 02: What is a Persistent Object?***

A *persistent object* is an object whose state is stored in a database (or other persistent storage) so that it can be retrieved and used later, even after the application restarts.

**Key Points:**

* Lives beyond the lifetime of the program execution.
* Typically stored in a relational or NoSQL database.
* Managed using ORM (Object-Relational Mapping) frameworks like Hibernate, JPA, or Sequelize.
* Has a unique identifier (primary key) to retrieve it later.
* Transitions between states: transient → persistent → detached → removed (in ORM terms).

**Example:**

If we have a User object in Java and we save it using Hibernate:

User u = new User("John");

session.save(u); // Now it's a persistent object stored in DB

Even after the application shuts down, u can be loaded again from the database.

**When to Use:**

* When we need the application’s data to survive beyond a single runtime session.
* Common in any system where saving user, product, or transaction data is necessary.

### ***Task 03: Class Diagram Relationships in UML***

**Composition** (Solid diamond → Solid line)

* Strong "part-of" relationship.
* Part cannot exist without the whole.
* Example: A Car has an Engine; if the car is destroyed, the engine is too.

**Aggregation** (Hollow diamond → Solid line)

* Weak "part-of" relationship.
* Part can exist independently of the whole.
* Example: A Team has Players; players can exist without the team.

**Association** (Solid line)

* General relationship between two classes.
* Example: A Teacher teaches a Student.

**Directed Association** (Solid line with arrow)

* Association with a specific navigability direction.
* Example: Order → Customer (Order knows about Customer).

**Usage / Dependency** (Dashed arrow)

* One class depends on another temporarily (often through method parameters).
* Example: A ReportGenerator uses a Printer object.

**Generalization** (Solid line with hollow arrow)

* Inheritance (“is-a” relationship).
* Example: Dog → Animal.

**Realization** (Dashed line with hollow arrow)

* Implementation of an interface.
* Example: ArrayList implements List.

@startuml

title UML Class Diagram Relationships

' Classes for examples

class Car

class Engine

class Team

class Player

class Teacher

class Student

class Order

class Customer

class ReportGenerator

class Printer

class Animal

class Dog

interface List

class ArrayList

' Composition

Car \*-- Engine : Composition

' Aggregation

Team o-- Player : Aggregation

' Association

Teacher -- Student : Association

' Directed Association

Order --> Customer : Directed Association

' Dependency

ReportGenerator ..> Printer : Usage / Dependency

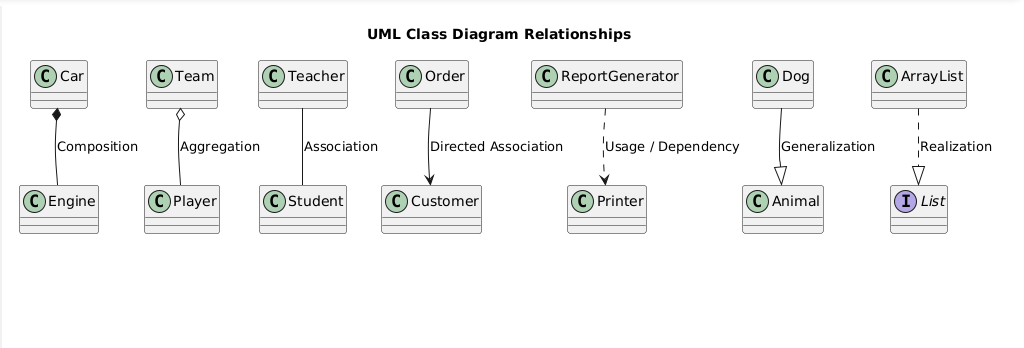
' Generalization

Dog --|> Animal : Generalization

' Realization

ArrayList ..|> List : Realization

@enduml



### ***Task 04: Package diagram representation of the Command Design Pattern***

@startuml

title Command Design Pattern - Package Diagram

package "Command Pattern" {

package "Command Interface" {

interface Command {

+execute()

}

}

package "Concrete Commands" {

class ConcreteCommandA {

-receiver: Receiver

+execute()

}

class ConcreteCommandB {

-receiver: Receiver

+execute()

}

}

package "Receiver" {

class Receiver {

+actionA()

+actionB()

}

}

package "Invoker" {

class Invoker {

-command: Command

+setCommand(cmd: Command)

+invoke()

}

}

package "Client" {

class Client

}

}

' Relationships

Command <|.. ConcreteCommandA

Command <|.. ConcreteCommandB

ConcreteCommandA --> Receiver

ConcreteCommandB --> Receiver

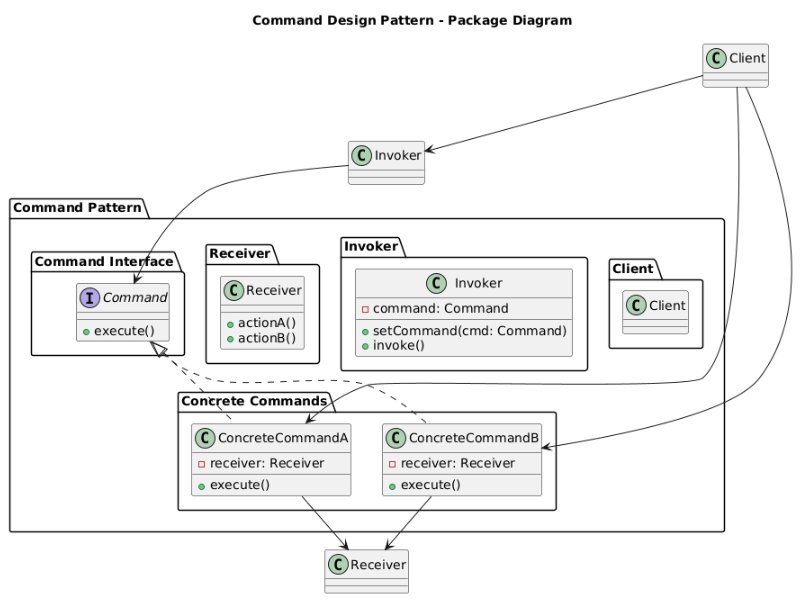
Invoker --> Command

Client --> Invoker

Client --> ConcreteCommandA

Client --> ConcreteCommandB

@enduml



### ***Task 05: Understand the difference between the following topics.***

**Aggregation vs Composition**

* **Aggregation**:
  + “Has-a” relationship (weak).
  + Part can exist without the whole.
  + UML: Hollow diamond.
  + Example: A *Team* has *Players*.
* **Composition**:
  + “Has-a” relationship (strong).
  + Part cannot exist without the whole.
  + UML: Solid diamond.
  + Example: A *Car* has an *Engine*.

**Association vs Dependency**

* **Association**:
  + Long-term relationship between objects.
  + Objects know about each other.
  + UML: Solid line.
  + Example: *Teacher* ↔ *Student*.
* **Dependency**:
  + Temporary relationship where one object uses another.
  + No permanent link.
  + UML: Dashed arrow.
  + Example: *ReportGenerator* → *Printer*.

**Three-Tier vs Multi-Tier Architecture**

* **Three-Tier**:
  + Fixed 3 layers: Presentation → Business Logic → Database.
  + Simpler and easier to manage.
  + Less scalable compared to multi-tier.
* **Multi-Tier**:
  + More than 3 layers (can include API gateway, caching, analytics).
  + Highly scalable and modular.
  + More complex to implement and manage.

### ***Task 06: MCQs***

**1. Which of the following components is not typically part of the Command pattern?** a) Invoker  
 b) Receiver  
 **c) Abstract Factory** d) Command (interface/abstract class)

**2. What role does the Invoker play in the Command pattern?** a) It knows how to perform the operations associated with a request.  
 b) It encapsulates the request as an object.  
 **c) It asks the command to carry out the request.** d) It defines the interface for executing an operation.

**3. A key benefit of using the Command pattern is its ability to support:** a) Lazy initialization  
 **b) Undo/Redo functionality** c) Singleton instance creation  
 d) Compile-time polymorphism

**4. In the Strategy pattern, what role does the "Context" play?** a) It defines the interface for the algorithms.  
 b) It implements a specific algorithm.  
 **c) It maintains a reference to a Strategy object and delegates the task to it.** d) It creates the Concrete Strategy objects.