Low Level Design (W/O Spring)

System Designs:

1. How to approach LLD interviews
2. Write unit tests for LLD classes (Cmd + Shift + T)
3. Class Diagrams
4. Sequence Diagrams
5. Design RateLimiter
6. Design Load Balancer
7. Design Logger System
8. Design **bit.ly**
9. Design Unique ID Generator Service
10. Design URL Shortener
11. Design Tic-Tac-Toe
12. Design Web Crawler
13. Design Search Engine
14. Design Parking Lot
    a. Singleton (for single instance of the Parking Lot)
    b. Factory (for selecting the vehicle instance)
    c. Strategy (for computation of vehicle feee)
    d. Observer (for notification)
    e. Command (for parking and unparking commands using a queue and invoker)
15. Design Splitwise
    a. Singleton (For balance manager to maintain consistency)
    b. Factory (For creating expense objects based on type)
    c. Strategy (For handling different types of splits (equal, exact, percentage)
    d. Observer (Notify users when their balances change)
16. Design Chess
17. Design Chess Validator

18. Design Distributed Queue | Kafka
19. Design Restaurant Management System: ClearFood
20. Design BookMyShow
21. Design Cab Booking (Ride Booking App)
22. Design a quick commerce grocery delivery system.
23. Design a system inspired by the internal app of Swiggy/Zomato to notify all drivers when the rating of any driver changes.
24. Design an leaderboard for an internal app used by Swiggy or Zomato to rate their delivery drivers after each delivery.
25. Design News Feed
26. Design Whatsapp
27. Design Instagram/Twitter News Feed
28. Design Search Autocomplete
29. Design Zomato Search
30. Car Rental System
31. Hotel Management
32. Library Management System
33. Movie Ticket Booking System
34. Restaurant Management System
35. Amazon - Online Shopping System
36. Design an Airline Management System
37. Design an ATM (Automated Teller Machine)
38. Design an Online Stock Brokerage System
39. Design Blackjack and a Deck of Cards
40. Design Cricinfo
41. Design Facebook - a social network
42. Design LinkedIn

---

1. How to approach LLD interviews

Approaching a **Low-Level Design (LLD)** round effectively requires structured thinking, clarity in communication, and hands-on knowledge of OOP principles, design patterns, and system behavior.

Here's a step-by-step guide to **ace your LLD round**:

✅ 1. Clarify the Requirements

- **Ask questions** to remove ambiguity.
  - Who are the users?
  - What are the core features?
  - Should it be extensible or just MVP?

> "Just to clarify, would you prefer a basic MVP-focused implementation for now, or should I keep the design flexible and extensible for future use cases?"

🧠 Example in Practice:

If asked to design a **Payment System**:

- For MVP: Hardcode `CreditCardPayment` and `UPIPayment`.
- For extensible: Use a `PaymentMethod` interface and implement each method separately, so you can add `WalletPayment`, `NetBanking`, etc. later.
- **Example**: If asked to design a Ride Booking App, clarify if it includes driver assignment, payment, real-time tracking, etc.

🧠 2. Identify the Core Entities (Classes)

- Think of the system as objects.
- List the **nouns** in the requirements: these become classes.
  - For a Ride Booking App: `User`, `Driver`, `Ride`, `Vehicle`, `Payment`.

🔗 3. Define Relationships and Associations

- Use **UML Class Diagrams** or note:
  - Inheritance (`Driver extends User`)
  - Composition (`Ride has-a Vehicle`)
  - Aggregation (`User has multiple Rides`)
- Identify **One-to-One**, **One-to-Many**, or **Many-to-Many** relationships.

🛠️ 4. Define Class Attributes & Methods

- Include core properties and behaviors.

- `Ride` → `startLocation`, `endLocation`, `fare`, `status`
- `startRide()`, `endRide()`, `calculateFare()`

🧩 5. Handle Edge Cases & Validations

- What happens when no driver is available?
- What if a payment fails?
- Can a user book multiple rides at once?

♻️ 6. Use Design Patterns Where Appropriate

- Mention **why** you're using one (not just name-dropping):
  - `Factory Pattern` for object creation (`VehicleFactory`)
  - `Strategy Pattern` for fare calculation
  - `Observer Pattern` for notifications
  - `Singleton` for managing config or DB connections

🚦 7. Keep Extensibility & Scalability in Mind

- Make your design **open for extension but closed for modification** (SOLID principles).
- For example, if more payment methods come in future, use interfaces like `PaymentMethod` → `UPI`, `CreditCard`, etc.

💬 8. Communicate Constantly

- Think **out loud**: interviewers assess your thought process.
- Ask: "Would you like me to go deeper into class diagram, or start with flow?"

📦 9. Discuss Data Flow & Persistence

- How is data stored or queried?
  - Use in-memory structures or basic DB schema.
  - You can mention using repositories or DAO pattern.

DAO = Data Access Object

It abstracts and encapsulates all access to the data source (like a DB, file, or external service), providing a simple API for the rest of the application to interact with.

```
// User Entity
public class User {
    private int id;
    private String name;
    // getters & setters
}

// DAO Interface
public interface UserDAO {
    User getUserById(int id);
    List<User> getAllUsers();
    void saveUser(User user);
    void deleteUser(int id);
}

// DAO Implementation
public class UserDAOImpl implements UserDAO {
    private Database db;

    public UserDAOImpl(Database db) {
        this.db = db;
    }

    public User getUserById(int id) {
        // logic to fetch from DB
    }

    public void saveUser(User user) {
        // logic to save to DB
    }
}
```

🔄 10. Time Permitting: Mention APIs / Interfaces

- Describe key API endpoints or interface methods if asked.

---

2. Write unit tests for LLD classes (Cmd + Shift + T)

🧪 What Can You Test in LLD?

Component	What to Test
Service Layer	Business logic, valid inputs/outputs
Strategy Pattern	Each strategy independently
Factory	Correct object creation
DAOs	Stub/mock DB calls
State Machines	Transitions, invalid states

```
// Notification interface
public interface Notification {
    void send(String to, String message);
}

// Email Notification
public class EmailNotification implements Notification {
    public void send(String to, String message) {
        System.out.println("Email sent to " + to + " with message: " + message);
    }
}

// Service using Notification
public class NotificationService {
    private final Notification notifier;

    public NotificationService(Notification notifier) {
        this.notifier = notifier;
    }

    public boolean notifyUser(String email, String message) {
        if (email == null || message == null) return false;
        notifier.send(email, message);
        return true;
    }
}

@Service
public class EmailNotificationSpringService implements Notification {
    public void send(String to, String message) {
        System.out.println("Email sent to " + to + " with message: " + message);
    }
}
```

```
@Service
public class NotificationSpringService {
//    @Autowired
//    private Notification notifier
    private final Notification notifier;

    @Autowired
    public NotificationSpringService(Notification notifier) {
        this.notifier = notifier;
    }

    public boolean notifyUser(String email, String message) {
        if (email == null || message == null) return false;
        notifier.send(email, message);
        return true;
    }
}
```

```
package com.sai.lld.UnitTestsExample;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.mockito.InjectMocks;
import org.mockito.Mock;

import static org.junit.jupiter.api.Assertions.assertFalse;
import static org.junit.jupiter.api.Assertions.assertTrue;
import static org.mockito.Mockito.*;

public class NotificationServiceTest {

    @Mock
    private Notification mockNotifier;

    @InjectMocks
    private NotificationService service;

    @BeforeEach
    void setUp() {
        mockNotifier = mock(Notification.class);
        service = new NotificationService(mockNotifier);
```

```
//        MockitoAnnotations.openMocks(this);
    }

    @Test
    void testNotifyUser_successful() {
        boolean result = service.notifyUser("user@example.com", "Hello");
        verify(mockNotifier, times(1)).send("user@example.com", "Hello");
        assertTrue(result);
    }

    @Test
    void testNotifyUser_nullMessage() {
        boolean result = service.notifyUser("user@example.com", null);
        verify(mockNotifier, never()).send(anyString(), any());
        assertFalse(result);
    }
}
```

```
package com.sai.lld.UnitTestsExample;

import org.junit.jupiter.api.Test;

public class EmailNotificationSpringServiceTest {

    @Test
    void testSend_shouldPrintToConsole() {
        EmailNotificationSpringService service = new EmailNotificationSpringService();
        service.send("user@example.com", "Hello");
        // No assertions — you would verify this manually by console output.
    }
}
```

```
package com.sai.lld.UnitTestsExample;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;
```

```
import static org.mockito.Mockito.*;

public class NotificationSpringServiceTest {

    @Mock
    private Notification notifier;

    @InjectMocks
    private NotificationSpringService service;

    @BeforeEach
    void setUp() {
        MockitoAnnotations.openMocks(this);
    }

    @Test
    void testNotifyUser_successful() {
        String email = "user@example.com";
        String message = "Hello";
        boolean result = service.notifyUser(email, message);
        verify(notifier, times(1)).send(email, message);
        assertTrue(result);
    }

    @Test
    void testNotifyUser_nullMessage() {
        boolean result = service.notifyUser("user@example.com", null);
        verifyNoInteractions(notifier);
        assertFalse(result);
    }

    @Test
    void testNotifyUser_nullEmail() {
        boolean result = service.notifyUser(null, "Hello");
        verifyNoInteractions(notifier);
        assertFalse(result);
    }

    @Test
    void testNotifyUser_nullEmailAndMessage() {
        boolean result = service.notifyUser(null, null);
        verifyNoInteractions(notifier);
        assertFalse(result);
```

```
    }
}
```

```
package com.sai.lld.UnitTestsExample;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

class UserEntityTest {


    @Test
    public void testNoArgsConstructorAndSetters() {
        UserEntity user = new UserEntity(); // No-args constructor
        user.setId(1L);
        user.setName("Sai");
        user.setEmail("sai@example.com");

        assertEquals(1L, user.getId());
        assertEquals("Sai", user.getName());
        assertEquals("sai@example.com", user.getEmail());
    }

    @Test
    public void testAllArgsConstructorAndGetters() {
        UserEntity user = new UserEntity(2L, "Ashish", "ashish@example.com");

        assertEquals(2L, user.getId());
        assertEquals("Ashish", user.getName());
        assertEquals("ashish@example.com", user.getEmail());
    }

    @Test
    public void testToString() {
        UserEntity user = new UserEntity(3L, "Test", "test@example.com");
        String str = user.toString();

        assertTrue(str.contains("3"));
        assertTrue(str.contains("Test"));
        assertTrue(str.contains("test@example.com"));
    }
```

```
@Test
public void testEqualsAndHashCode() {
    UserEntity user1 = new UserEntity(4L, "Equal", "equal@example.com");
    UserEntity user2 = new UserEntity(4L, "Equal", "equal@example.com");
    UserEntity user3 = new UserEntity(5L, "NotEqual", "notequal@example.com");

    assertEquals(user1, user2);
    assertNotEquals(user1, user3);
    assertEquals(user1.hashCode(), user2.hashCode());
    assertNotEquals(user1.hashCode(), user3.hashCode());
}

@Test
void getId() {
}

@Test
void getName() {
}

@Test
void getEmail() {
}

@Test
void setId() {
}

@Test
void setName() {
}

@Test
void setEmail() {
}

@Test
void testEquals() {
}

@Test
void canEqual() {
}
```

```
    @Test
    void testHashCode() {
    }

}
```

```
class NotificationTest {

    @Test
    void send() {
        Notification emailNotification = new EmailNotification();
        assertDoesNotThrow(() -> emailNotification.send("test@example.com", "Hello"));
    }
}
```

Directly testing an interface itself is not possible because an interface only declares method signatures without any implementation. Tests require executable code, so you need a concrete implementation of that interface to run tests on.

---

3. **UML Class Diagrams**

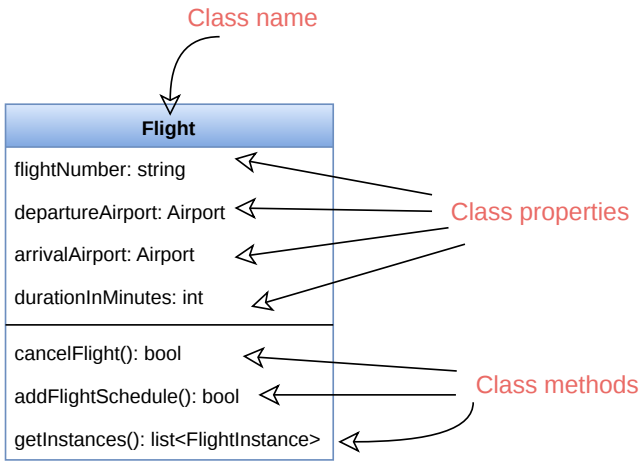
Class diagram is the backbone of object-oriented modeling - it shows how different entities (people, things, and data) relate to each other. In other words, it shows the static structures of the system.

A class diagram describes the attributes and operations of a class and also the constraints imposed on the system. Class diagrams are widely used in the modeling of object-oriented systems because they are the only UML diagrams that can be mapped directly to object-oriented languages.

The purpose of the class diagram can be summarized as:

1. Analysis and design of the static view of an application.
2. To describe the responsibilities of a system.
3. To provide a base for component and deployment diagrams.
4. Forward and reverse engineering.

A class is depicted in the class diagram as a rectangle with three horizontal sections, as shown in the figure below. The upper section shows the class's name (Flight), the middle section contains the properties of the class, and the lower section contains the class's operations (or "methods").



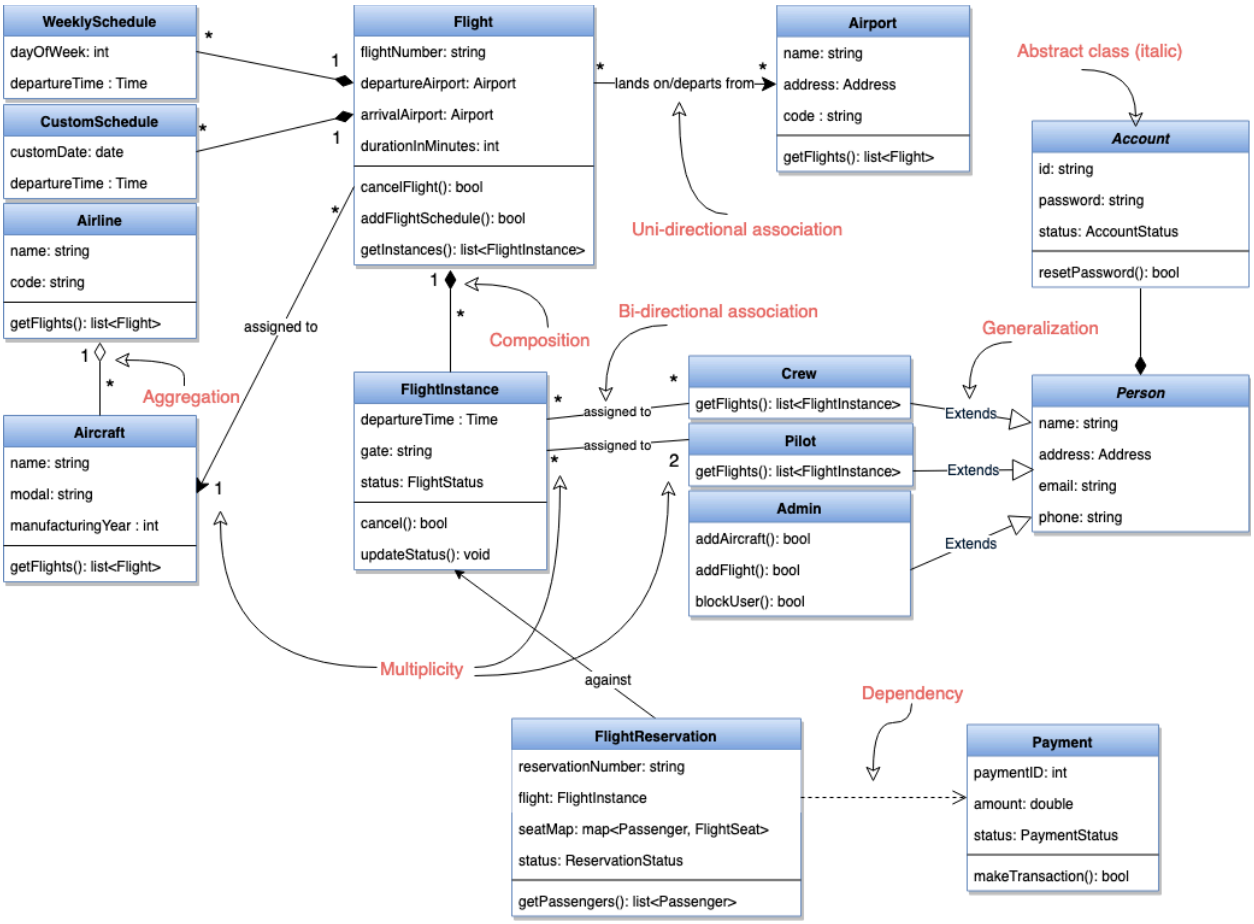
These are the different types of relationships between classes:

Association: If two classes in a model need to communicate with each other, there must be a link between them. This link can be represented by an association. Associations can be represented in a class diagram by a line between these classes with an arrow indicating the navigation direction.

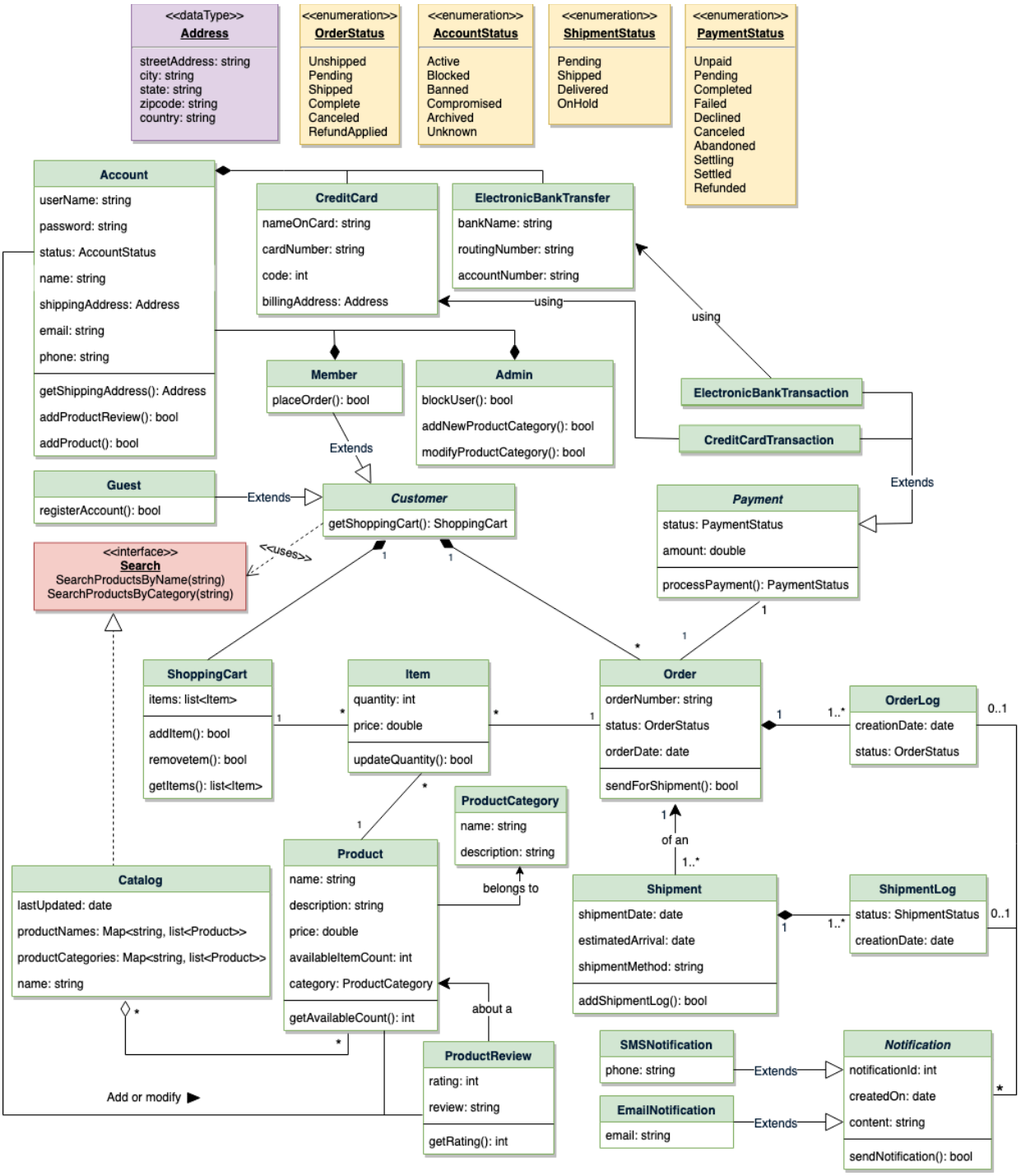
- By default, associations are always assumed to be bi-directional; this means that both classes are aware of each other and their relationship. In the diagram below, the association between Pilot and FlightInstance is bi-directional, as both classes know each other.
- By contrast, in a uni-directional association, two classes are related - but only one class knows that the relationship exists. In the below example, only Flight class knows about Aircraft; hence it is a uni-directional association

Multiplicity: Multiplicity indicates how many instances of a class participate in the relationship. It is a constraint that specifies the range of permitted cardinalities between two classes. For example, in the diagram below, one FlightInstance will have two Pilots, while a Pilot can have many FlightInstances. A ranged multiplicity can be expressed as “0...*” which means “zero to many" or as “2...4” which means “two to four”.

We can indicate the multiplicity of an association by adding multiplicity adornments to the line denoting the association. The below diagram, demonstrates that a FlightInstance has exactly two Pilots but a Pilot can have many FlightInstances.



Sample class diagram for flight reservation system



Class Diagram for Online Shoopping System

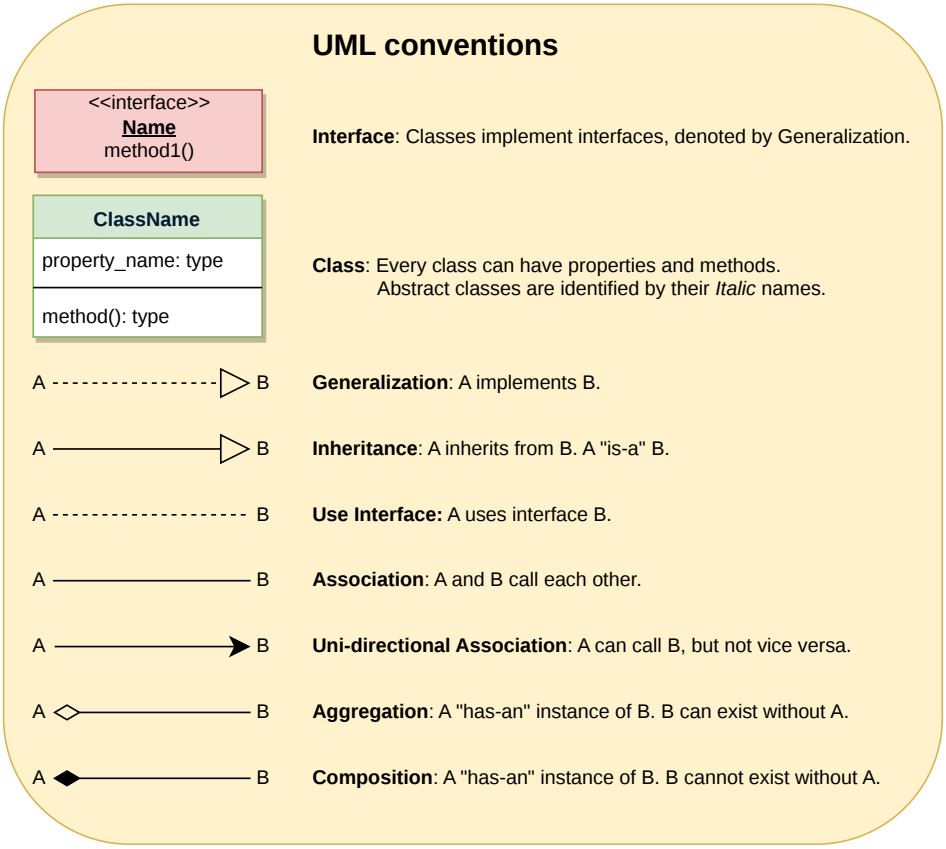
Aggregation: Aggregation is a special type of association used to model a “whole to its parts” relationship. In a basic aggregation relationship, the lifecycle of a PART class is independent of the WHOLE class’s lifecycle. In other words, aggregation implies a relationship where the child can exist independently of the parent. In the above diagram, Aircraft can exist without Airline.

Composition: The composition aggregation relationship is just another form of the aggregation relationship, but the child class's instance lifecycle is dependent on the parent class's instance lifecycle. In other words, Composition implies a relationship where the child cannot exist independent of the parent. In the above example, WeeklySchedule is composed in Flight which means when Flight lifecycle ends, WeeklySchedule automatically gets destroyed.

Generalization: Generalization is the mechanism for combining similar classes of objects into a single, more general class. Generalization identifies commonalities among a set of entities. In the above diagram, Crew, Pilot, and Admin, all are Person.

Dependency: A dependency relationship is a relationship in which one class, the client, uses or depends on another class, the supplier. In the above diagram, FlightReservation depends on Payment.

Abstract Class: An abstract class is identified by specifying its name in italics. In the above diagram, both Person and Account classes are abstract classes.



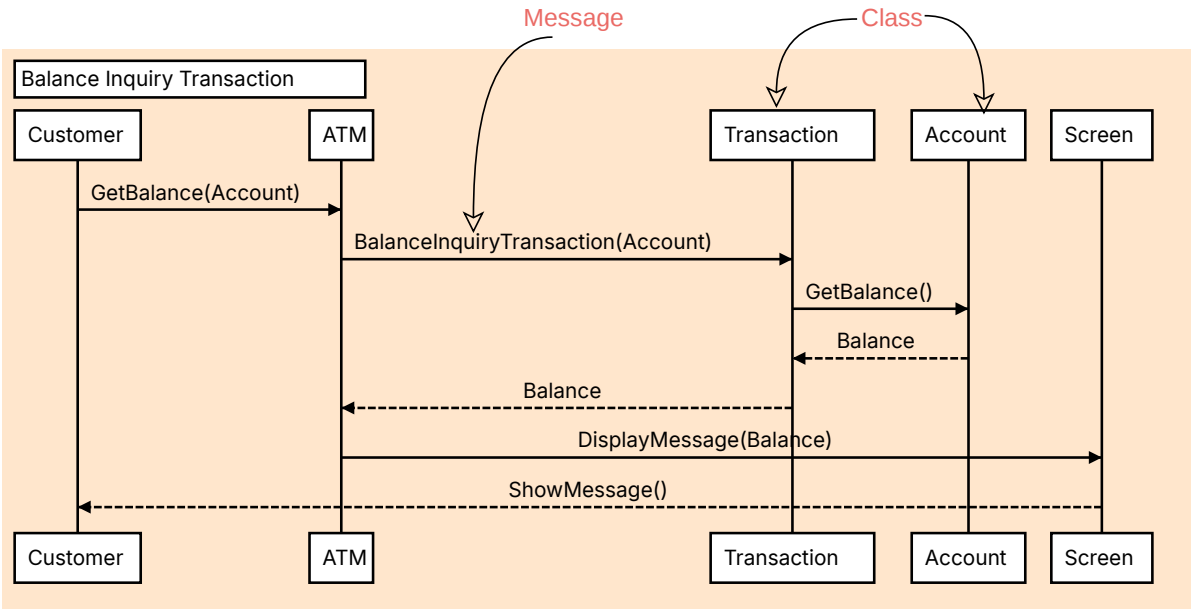
4.

UML Sequence Diagrams

Sequence diagrams describe interactions among classes in terms of an exchange of messages over time and are used to explore the logic of complex operations, functions or procedures. In this diagram, the sequence of interactions between the objects is represented in a step-by-step manner.

Sequence diagrams show a detailed flow for a specific use case or even just part of a particular use case. They are almost self-explanatory; they show the calls between the different objects in their sequence and can explain, at a detailed level, different calls to various objects.

A sequence diagram has two dimensions: The vertical dimension shows the sequence of messages in the chronological order that they occur; the horizontal dimension shows the object instances to which the messages are sent.



Sample sequence diagram for ATM balance inquiry

A sequence diagram is straightforward to draw. Across the top of your diagram, identify the class instances (objects) by putting each class instance inside a box (see above figure). If a class instance sends a message to another class instance, draw a line with an open arrowhead pointing to the receiving class instance and place the name of the message above the line. Optionally, for important messages, you can draw a dotted line with an arrowhead pointing back to the originating class instance; label the returned value above the dotted line.

---

5. Design RateLimiter

https://www.geeksforgeeks.org/how-to-design-a-rate-limiter-api-learn-system-design/

*A **Rate Limiter API** is a tool that developers can use to define rules that specify how many requests can be made in a given time period and what actions should be taken when these limits are exceeded.*

It helps to prevent a high volume of requests from overwhelming a server or API. Here is a basic design for a rate limiter API In this article, we will discuss the design of a rate limiter API, including its requirements, high-level design, and algorithms used for rate limiting.

Rate limiting errors typically occur when a client exceeds the number of allowed requests to a server within a specific time frame. Here are common HTTP status codes used for rate limiting:

🔢 Common Rate Limiting Error Codes

- `429 Too Many Requests` → Most widely used for rate limiting.

  Indicates the user has sent too many requests in a given amount of time.

📋 Optional Headers (often included with 429):

- `Retry-After`: Tells the client how long to wait before making a new request (in seconds or as a date/time).
- `X-RateLimit-Limit`: Max number of requests allowed.
- `X-RateLimit-Remaining`: Requests remaining in the current window.
- `X-RateLimit-Reset`: Time when the limit will reset.

Why is rate limiting used?

- Avoid resource starvation due to a Denial of Service (DoS) attack.
- Ensure that servers are not overburdened. Using rate restriction per user
- ensures fair and reasonable use without harming other users.
- Control the flow of information, for example, prevent a single worker from accumulating a backlog of unprocessed items while other workers are idle.

`System Requirements:`

Functional requirements to Design a Rate Limiter API:

- The API should allow the definition of multiple rate-limiting rules.
- The API should provide the ability to customize the response to clients when rate limits are exceeded.
- The API should allow for the storage and retrieval of rate-limit data.
- The API should be implemented with proper error handling as in when the threshold limit of requests are crossed for a single server or across different combinations, the client should get a proper error message.

Non-functional requirements to Design a Rate Limiter API:

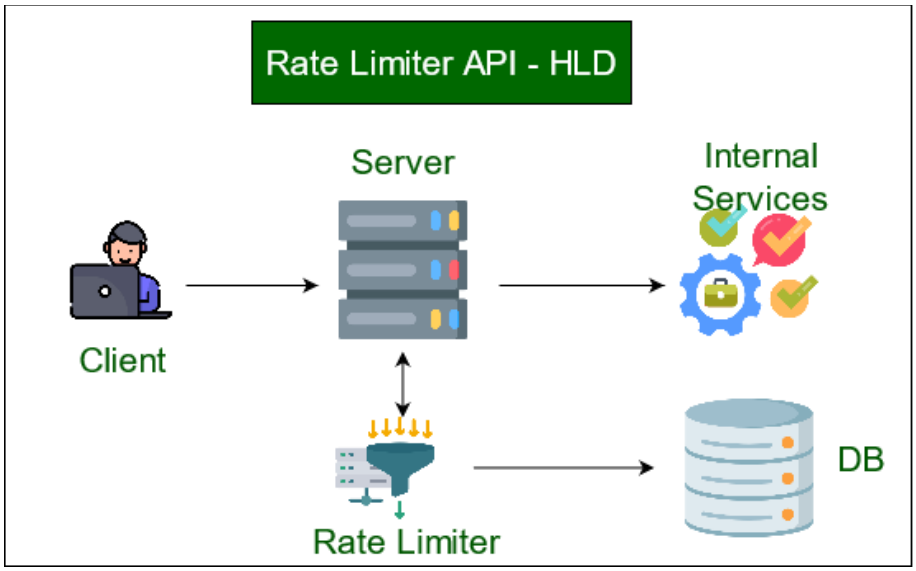
- The API should be highly available and scalable. Availability is the main pillar in the case of request fetching APIs.
- The API should be secure and protected against malicious attacks.
- The API should be easy to integrate with existing systems.
- There should be low latency provided by the rate limiter to the system, as performance is one of the key factors in the case of any system.

High Level Design (HLD) to Design a Rate Limiter API

Where to place the Rate Limiter - Client Side or Server Side?

> A rate limiter should generally be implemented on the server side rather than on the client side. This is because of the following points:

- Positional Advantage: The server is in a better position to enforce rate limits across all clients, whereas client-side rate limiting would require every client to implement their own rate limiter, which would be difficult to coordinate and enforce consistently.
- Security: Implementing rate limiting on the server side also provides better security, as it allows the server to prevent malicious clients from overwhelming the system with a large number of requests. If rate limiting were implemented on the client side, it would be easier for attackers to bypass the rate limit by just modifying or disabling the client-side code.
- Flexible: Server-side rate limiting allows more flexibility in adjusting the rate limits and managing resources. The server can dynamically adjust the rate limits based on traffic patterns and resource availability, and can also prioritize certain types of requests or clients over others. Thus, lends to better utilization of available resources, and also keeps performance good.



HLD of Rate Limiter API - rate limiter placed at server side

The overall basic structure of a rate limiter seems relatively simpler. We just need a counter associated with each user to track how many requests are being same submitted in a particular timeframe. The request is rejected if the counter value hits the limit.

Memory Structure/Approximation

> Thus, now let's think of the data structure which might help us. Since we need fast retrieval of the counter values associated with each user, we can use a hash-table. Considering we have a key-value pair. The key would contain hash value of each User Id, and the corresponding value would be the pair or structure of counter and the startTime, e.g.,UserId → {counter, startTime}Now, each UserId let's say takes 8 bytes(long long) and the counter takes 2 bytes(int), which for now can count to 50k(limit). Now for the time if we store only the minute and seconds, it will also take 2 bytes. So in total, we would need 12 bytes to store each user's data.Now considering the overhead of 10 bytes for each record in our hash-table, we would be needing to track at least 5 million users at any time(traffic), so the total memory in need would be:(12+10)bytes*5 million = 110 MB

Key Components in the Rate Limiter

- **Define the rate limiting policy** : The first step is to determine the policy for rate limiting. This policy should include the maximum number of requests allowed per unit of time, the time window for measuring requests, and the actions to be taken when a limit is exceeded (e.g., return an error code or delay the request).

- **Store request counts** : The rate limiter API should keep track of the number of requests made by each client. One way to do this is to use a database, such as Redis or Cassandra, to store the request counts.
- **Identify the client** : The API must identify each client that makes a request. This can be done using a unique identifier such as an IP address or an API key.
- **Handle incoming requests** : When a client makes a request, the API should first check if the client has exceeded their request limit within the specified time window. If the limit has been reached, the API can take the action specified in the rate-limiting policy (e.g., return an error code). If the limit has not been reached, the API should update the request count for the client and allow the request to proceed.
- **Set headers** : When a request is allowed, the API should set appropriate headers in the response to indicate the remaining number of requests that the client can make within the time window, as well as the time at which the limit will be reset.
- **Expose an endpoint** : Finally, the rate limiter API should expose an endpoint for clients to check their current rate limit status. This endpoint can return the number of requests remaining within the time window, as well as the time at which the limit will be reset.

Where should we keep the counters?

> Due to the slowness of Database operations, it is not a smart option for us. This problem can be handled by an in-memory cache such as Redis. It is quick and supports the already implemented time-based expiration technique.We can rely on two commands being used with in-memory storage,INCR: This is used for increasing the stored counter by 1.EXPIRE: This is used for setting the timeout on the stored counter. This counter is automatically deleted from the storage when the timeout expires.In this design, client requests pass through a rate limiter middleware, which checks against the configured rate limits. The rate limiter module stores and retrieves rate limit data from a backend storage system. If a client exceeds a rate limit, the rate limiter module returns an appropriate response to the client.

Algorithms to Design a Rate Limiter API

Several algorithms are used for rate limiting, including

- The Token bucket,
- Leaky bucket,
- Sliding window logs, and

- Sliding window counters.

Let's discuss each algorithm in detail:

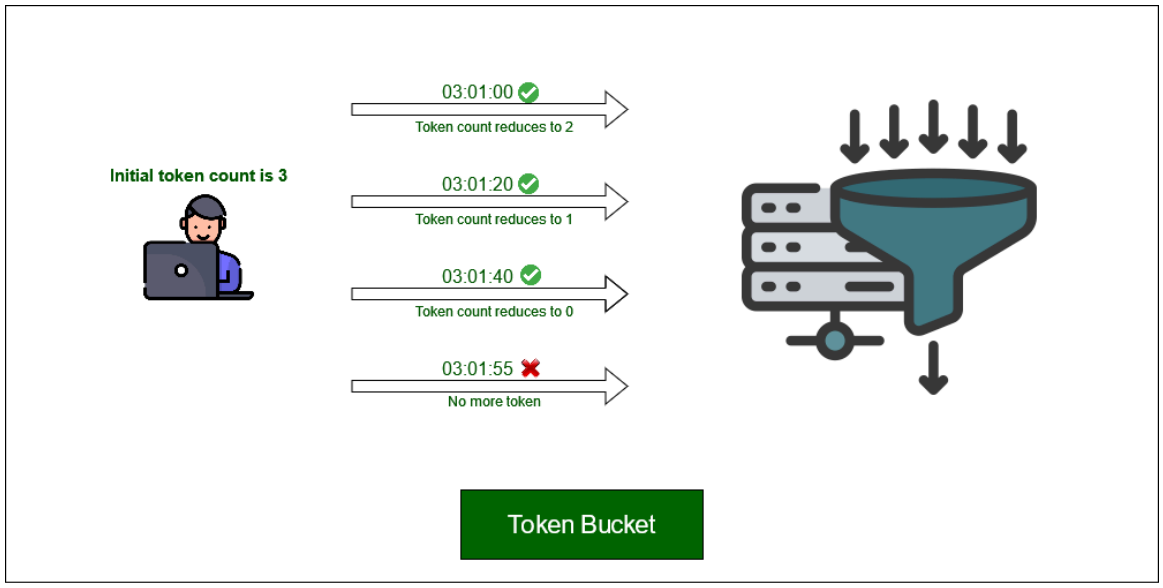
Token Bucket

The token bucket algorithm is a simple algorithm that uses a fixed-size token bucket to limit the rate of incoming requests. The token bucket is filled with tokens at a fixed rate, and each request requires a token to be processed. If the bucket is empty, the request is rejected.

The token bucket algorithm can be implemented using the following steps:

- Initialize the token bucket with a fixed number of tokens.
- For each request, remove a token from the bucket.
- If there are no tokens left in the bucket, reject the request.
- Add tokens to the bucket at a fixed rate.

Thus, by allocating a bucket with a predetermined number of tokens for each user, we are successfully limiting the number of requests per user per time unit. When the counter of tokens comes down to 0 for a certain user, we know that he or she has reached the maximum amount of requests in a particular timeframe. The bucket will be auto-refilled whenever the new timeframe starts.



Token bucket example with initial bucket token count of 3 for each user in one minute

Leaky Bucket

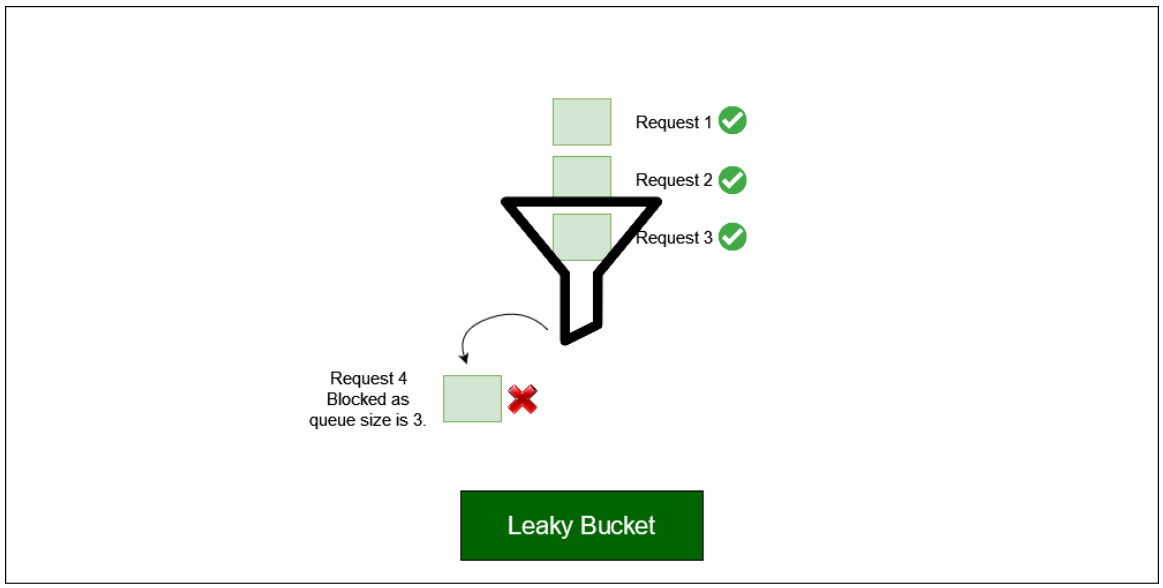
It is based on the idea that if the average rate at which water is poured exceeds the rate at which the bucket leaks, the bucket will overflow.

The leaky bucket algorithm is similar to the token bucket algorithm, but instead of using a fixed-size token bucket, it uses a leaky bucket that empties at a fixed rate. Each incoming request adds to the bucket's depth, and if the bucket overflows, the request is rejected.

One way to implement this is using a queue, which corresponds to the bucket that will contain the incoming requests. Whenever a new request is made, it is added to the queue's end. If the queue is full at any time, then the additional requests are discarded.

The leaky bucket algorithm can be separated into the following concepts:

- Initialize the leaky bucket with a fixed depth and a rate at which it leaks.
- For each request, add to the bucket's depth.
- If the bucket's depth exceeds its capacity, reject the request.
- Leak the bucket at a fixed rate.



Leaky bucket example with token count per user per minute is 3, which is the queue size.

Sliding Window Logs

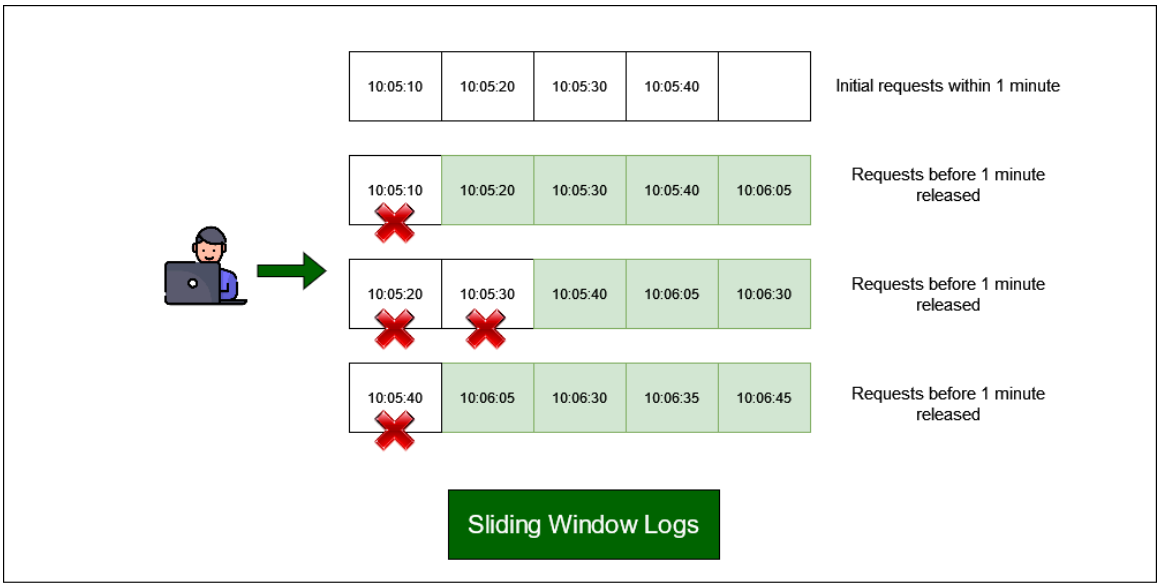
Another approach to rate limiting is to use sliding window logs. This data structure involves a "window" of fixed size that slides along a timeline of events, storing information about the events that fall within the window at any given time.

The window can be thought of as a buffer of limited size that holds the most recent events or changes that have occurred. As new events or changes occur, they are added to the buffer, and old events that fall outside of the window are removed. This ensures that the buffer stays within its fixed size, and only contains the most recent events.

This rate limitation keeps track of each client's request in a time-stamped log. These logs are normally stored in a time-sorted **hash set or table.**

The sliding window logs algorithm can be implemented using the following steps:

- A time-sorted queue or hash table of timestamps within the time range of the most recent window is maintained for each client making the requests.
- When a certain length of the queue is reached or after a certain number of minutes, whenever a new request comes, a check is done for any timestamps older than the current window time.
- The queue is updated with new timestamp of incoming request and if number of elements in queue does not exceed the authorised count, it is proceeded otherwise an exception is triggered.



Sliding window logs in a timeframe of 1 minute

Sliding Window Counters

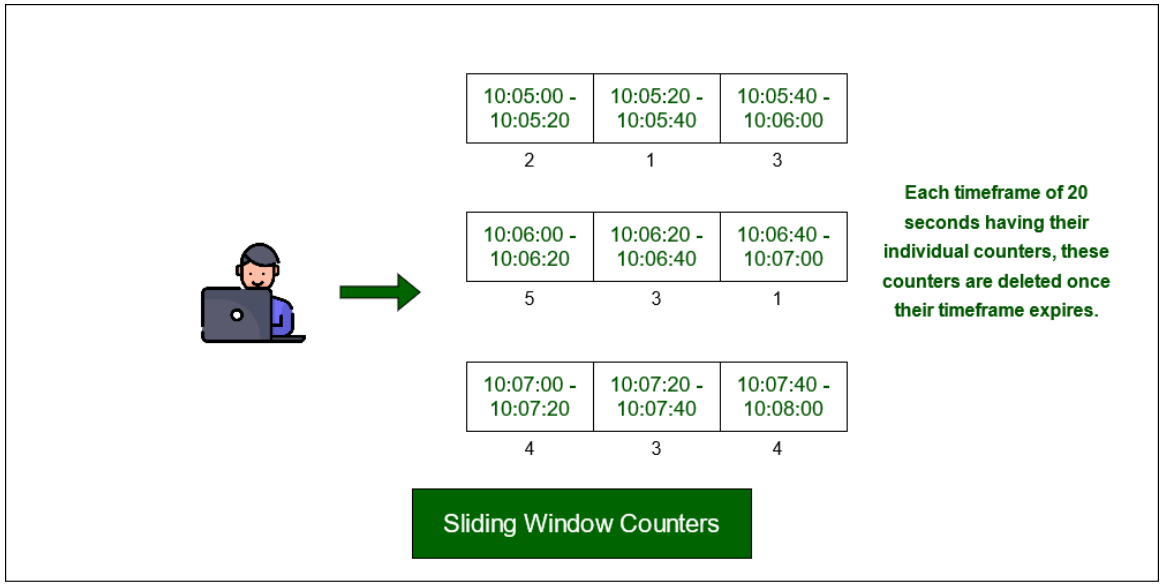
The sliding window counter algorithm is an optimization over sliding window logs. As we can see in the previous approach, memory usage is high. For example, to manage numerous users or huge window timeframes, all the request timestamps must be kept for a window time, which eventually uses a huge amount of memory. Also, removing numerous timestamps older than a particular timeframe means high complexity of time as well.

To reduce surges of traffic, this algorithm accounts for a weighted value of the previous window's request based on timeframe. If we have a one-minute rate limit, we can record the counter for each second and calculate the sum of all counters in the previous minute whenever we get a new request to determine the throttling limit.

The sliding window counters can be separated into the following concepts:

- Remove all counters which are more than 1 minute old.
- If a request comes which falls in the current bucket, the counter is increased.

- If a request comes when the current bucket has reached it's throat limit, the request is blocked.



sliding window counters with a timeframe of 20 seconds

Examples of Rate Limiting APIs used worldwide

- **Google Cloud Endpoints:** It is a platform for building APIs that includes a built-in rate limiter to help prevent excessive API usage.
- **AWS API Gateway:** Amazon Web Services (AWS) API Gateway includes a feature called Usage Plans that allows for rate limiting and throttling of API requests.
- **Akamai API Gateway:** Akamai API Gateway is a cloud-based platform that includes a rate limiter feature for controlling API requests.
- **Cloudflare Rate Limiting:** Cloudflare's Rate Limiting feature helps prevent DDoS attacks and other types of abusive traffic by limiting the number of requests that can be made to an API.
- **Redis:** It is an in-memory data structure store that can be used as a database, cache, and message broker. It includes several features that make it useful for implementing a rate limiter, such as its ability to store data in memory for fast access and its support for atomic operations.

Designing a **Rate Limiter using the Token Bucket algorithm** is a common system design interview question. Here's a complete breakdown with implementation logic and optional code in Java.

✅ Concept: Token Bucket Algorithm

The **Token Bucket** algorithm allows **burst traffic** while maintaining an average rate over time.

🪨 Core Concepts:

- A "bucket" holds **tokens**.
- Tokens are **added at a fixed rate** (e.g., 1 token per second).
- Every **request consumes 1 token**.
- If the bucket has tokens → **allow request** and remove a token.
- If no tokens → **reject the request (rate limit hit)**.

⚙️ Design Requirements

- `capacity`: max number of tokens bucket can hold.
- `refillRate`: tokens added per second.
- `lastRefillTimestamp`: last time the bucket was refilled.
- `tokens`: current number of tokens in the bucket.

📦 Java Implementation

```
import java.util.concurrent.locks.ReentrantLock;

public class TokenBucket {
    private final int capacity;
    private final double refillRatePerSecond;
    private double tokens;
    private long lastRefillTimestamp;

    private final ReentrantLock lock = new ReentrantLock();

    public TokenBucket(int capacity, double refillRatePerSecond) {
        this.capacity = capacity;
        this.tokens = capacity;
        this.refillRatePerSecond = refillRatePerSecond;
        this.lastRefillTimestamp = System.nanoTime();
    }

    public boolean allowRequest() {
        lock.lock();
        try {
            refill();
            if (tokens >= 1) {
```

```
            tokens -= 1;
            return true;
        }
        return false;
    } finally {
        lock.unlock();
    }
  }

  private void refill() {
    long now = System.nanoTime();
    double secondsElapsed = (now - lastRefillTimestamp)
    / 1_000_000_000.0;
    double tokensToAdd = secondsElapsed * refillRatePerSecond;

    if (tokensToAdd > 0) {
        tokens = Math.min(capacity, tokens + tokensToAdd);
        lastRefillTimestamp = now;
    }
  }
}
```

🧪 Sample Usage

```
public class Main {
  public static void main(String[] args) throws InterruptedException {
    TokenBucket limiter = new TokenBucket(5, 1); // capacity=5, 1 token/sec

    for (int i = 0; i < 10; i++) {
        boolean allowed = limiter.allowRequest();
        System.out.println("Request " + i + ": " + (allowed ? "✅ Allowed" : "❌ Denied"));
        Thread.sleep(200); // simulate 5 requests per second
    }
  }
}
```

💡 When to Use Token Bucket

- Allows bursty traffic (unlike Leaky Bucket).
- Ensures long-term rate limiting.
- Suitable for APIs, login attempts, messaging systems, etc.

🔄 Comparison with Leaky Bucket

Feature	Token Bucket	Leaky Bucket
Bursts allowed	✅ Yes	❌ No
Fairness	Medium	High (uniform output rate)
Typical use case	API Rate Limiting, Uploads	Traffic shaping

✅ Token Bucket Class (Reusable Component)

Assume you have this implementation already:

```
public class TokenBucket {

    private final int capacity;
    private final double refillRatePerSecond;
    private double tokens;
    private long lastRefillTimestamp;

    public TokenBucket(int capacity, double refillRatePerSecond) {
        this.capacity = capacity;
        this.refillRatePerSecond = refillRatePerSecond;
        this.tokens = capacity;
        this.lastRefillTimestamp = System.nanoTime();
    }

    public synchronized boolean allowRequest() {
        refill();
        if (tokens >= 1) {
            tokens--;
            return true;
        }
        return false;
    }

    private void refill() {
        long now = System.nanoTime();
        double tokensToAdd = ((now - lastRefillTimestamp) / 1e9) * refillRatePerSecond;
        tokens = Math.min(capacity, tokens + tokensToAdd);
        lastRefillTimestamp = now;
    }
}
```

🧪 Unit Tests (using JUnit)

```
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

class TokenBucketTest {

    @Test
    void testAllowRequest_whenTokensAvailable() {
        TokenBucket bucket = new TokenBucket(5, 1); // 5 tokens, 1 token/sec refill

        for (int i = 0; i < 5; i++) {
            assertTrue(bucket.allowRequest(), "Request should be allowed");
        }

        assertFalse(bucket.allowRequest(), "6th request should be denied");
    }

    @Test
    void testRefillAfterDelay() throws InterruptedException {
        TokenBucket bucket = new TokenBucket(1, 1); // 1 token/sec

        assertTrue(bucket.allowRequest()); // Consume 1 token
        assertFalse(bucket.allowRequest()); // No token left

        Thread.sleep(1100); // Wait for refill

        assertTrue(bucket.allowRequest(), "Token should be refilled after 1 sec");
    }
}
```

🚀 Spring Boot Integration

1. Register TokenBucket as a Bean

```
@Configuration
public class RateLimiterConfig {

    @Bean
    public TokenBucket tokenBucket() {
        return new TokenBucket(10, 2); // 10 capacity, 2 tokens/sec
```

```
    }
}
```

2. Inject in Controller

```
@RestController
@RequestMapping("/api")
public class RateLimitedController {

    private final TokenBucket tokenBucket;

    public RateLimitedController(TokenBucket tokenBucket) {
        this.tokenBucket = tokenBucket;
    }

    @GetMapping("/limited")
    public ResponseEntity<String> limitedEndpoint() {
        if (tokenBucket.allowRequest()) {
            return ResponseEntity.ok("✅ Request successful");
        } else {
            return ResponseEntity.status(HttpStatus.TOO_MANY_REQUESTS)
                                 .body("❌ Too many requests - rate limit exceeded");
        }
    }
}
```

3. Integration Test (Optional)

```
@SpringBootTest
@AutoConfigureMockMvc
class RateLimitedControllerIntegrationTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    void testRateLimiting() throws Exception {
        for (int i = 0; i < 12; i++) {
            MvcResult result = mockMvc.perform(get("/api/limited"))
                .andReturn();

            String content = result.getResponse().getContentAsString();
```

```
            int status = result.getResponse().getStatus();

            System.out.println("Status: " + status + ", Body: " + content);
        }
    }
}
```

🧠 Notes:

- This is an **in-memory** implementation, suitable for single-node apps or demos.
- For distributed environments, consider **Redis-backed buckets** or tools like **Bucket4J**, **resilience4j**, or **RateLimiter in Spring Cloud Gateway**.

💧 Leaky Bucket Rate Limiter

💡 Concept:

Imagine a bucket with a small hole in the bottom:

- Requests **enter** the bucket like water.
- Water **leaks** out at a **constant rate**.
- If the bucket overflows → **request is rejected** (rate limit hit).

✅ Properties:

- Fixed output rate: handles traffic evenly.
- No burst allowed (unlike Token Bucket).
- Smoother & fair for downstream systems.

📦 Java Implementation

```
public class LeakyBucketRateLimiter {

    private final int capacity;
    private final double leakRatePerSecond;
    private double currentWaterLevel;
    private long lastLeakTimestamp;

    public LeakyBucketRateLimiter(int capacity, double leakRatePerSecond) {
```

```
        this.capacity = capacity;
        this.leakRatePerSecond = leakRatePerSecond;
        this.currentWaterLevel = 0;
        this.lastLeakTimestamp = System.nanoTime();
    }

    public synchronized boolean allowRequest() {
        leak();  // leak before processing new request

        if (currentWaterLevel < capacity) {
            currentWaterLevel++;
            return true;
        }

        return false;
    }

    private void leak() {
        long now = System.nanoTime();
        double elapsedSeconds = (now - lastLeakTimestamp) / 1_000_000_000.0;

        double leaked = elapsedSeconds * leakRatePerSecond;

        if (leaked > 0) {
            currentWaterLevel = Math.max(0, currentWaterLevel - leaked);
            lastLeakTimestamp = now;
        }
    }
}
```

🧪 Example Usage

```
public class Main {
    public static void main(String[] args) throws InterruptedException {
        LeakyBucketRateLimiter limiter = new LeakyBucketRateLimiter(5, 1); // capacity=5, leak
Rate=1/sec

        for (int i = 0; i < 10; i++) {
            boolean allowed = limiter.allowRequest();
            System.out.println("Request " + i + ": " + (allowed ? "✅ Allowed" : "❌ Rejected"));
            Thread.sleep(200); // 5 requests per second
        }
```

```
    }
}
```

🔄 Comparison: Token Bucket vs Leaky Bucket

Feature	Token Bucket	Leaky Bucket
Allows bursts	✅ Yes	❌ No
Output rate	Variable	Fixed
Complexity	Medium	Simple
Use case	API gateways	Network shaping

🧠 Common Use Cases:

- Throttling user requests (prevent spamming)
- Rate-limiting login attempts
- Limiting API calls per service

📘 Sliding Window Logs Rate Limiter

💡 Concept:

- Maintain a **log of request timestamps** for each user or key.
- Only allow a request if the number of timestamps in the **last** `x` **seconds** is less than the allowed limit.
- Logs slide as time progresses — hence the name **Sliding Window**.

✅ Use Case:

Let's say:

- You want to allow **100 requests per 60 seconds** per user.
- You track each request timestamp.
- If a new request comes in, you remove timestamps older than 60 seconds.
- If the remaining timestamps are < 100 → **allow** the request.

📦 Java Implementation

```
import java.util.*;
```

```
public class SlidingWindowLogRateLimiter {

    private final int maxRequests;
    private final long windowSizeInMillis;
    private final Map<String, Deque<Long>> userRequestLogs;

    public SlidingWindowLogRateLimiter(int maxRequests, int windowSizeInSeconds) {
        this.maxRequests = maxRequests;
        this.windowSizeInMillis = windowSizeInSeconds * 1000L;
        this.userRequestLogs = new HashMap<>();
    }

    public synchronized boolean allowRequest(String userId) {
        long now = System.currentTimeMillis();
        userRequestLogs.putIfAbsent(userId, new ArrayDeque<>());

        Deque<Long> logs = userRequestLogs.get(userId);

        // Clean up old requests outside the window
        while (!logs.isEmpty() && (now - logs.peekFirst() > windowSizeInMillis)) {
            logs.pollFirst();
        }

        if (logs.size() < maxRequests) {
            logs.addLast(now);
            return true;
        }

        return false;
    }
}
```

🧪 Example Usage

```
public class Main {
    public static void main(String[] args) throws InterruptedException {
        SlidingWindowLogRateLimiter limiter = new SlidingWindowLogRateLimiter(3, 10); // 3 re
quests per 10 sec

        String userId = "user1";
```

```
        for (int i = 1; i <= 5; i++) {
            boolean allowed = limiter.allowRequest(userId);
            System.out.println("Request " + i + ": " + (allowed ? "✅ Allowed" : "❌ Rejected"));
            Thread.sleep(2000); // 2 seconds between requests
        }
    }
}
```

🔍 Time & Space Complexity

Operation	Complexity
`allowRequest`	O(n), where n = number of recent requests in the window
Space	O(n) per user (timestamps kept in memory)

⚖️ Pros & Cons

✅ Pros	❌ Cons
Very accurate	High memory usage (stores logs)
Fair rate limiting	Slower as log grows
Easy to implement & test	Not ideal for high-scale systems

🔄 When to use Sliding Window Log?

- **Low-to-medium traffic** applications
- **Exact control** over request count
- When accuracy > performance

Or implementation of **Sliding Window Counter** (optimized variant).

📘 Sliding Window Counter – Overview

💡 Concept:

- The time window is divided into smaller equal sub-windows (e.g., divide 60s into 6 × 10s).
- Maintain a **counter per sub-window**
- When a request arrives:
    - Drop expired sub-windows.
    - Sum the counters of the **current sliding window**.

- If total count < allowed limit, allow the request and increment the current sub-window's counter.

✅ Use Case Example:

- Allow 100 requests per 60 seconds.
- Divide into 6 sub-windows (10 seconds each).
- Track each sub-window's timestamp and count.
- Efficient and nearly accurate.

📦 Java Implementation

```
import java.util.*;

public class SlidingWindowCounterRateLimiter {

    private static class Window {
        long timestamp; // start time of this window
        int count;

        Window(long timestamp, int count) {
            this.timestamp = timestamp;
            this.count = count;
        }
    }

    private final int maxRequests;
    private final long windowSizeInMillis;
    private final int numberOfBuckets;
    private final long bucketSizeInMillis;

    private final Map<String, Deque<Window>> userWindows;

    public SlidingWindowCounterRateLimiter(int maxRequests, int windowSizeInSeconds, int
numberOfBuckets) {
        this.maxRequests = maxRequests;
        this.windowSizeInMillis = windowSizeInSeconds * 1000L;
        this.numberOfBuckets = numberOfBuckets;
        this.bucketSizeInMillis = windowSizeInMillis / numberOfBuckets;
        this.userWindows = new HashMap<>();
    }
```

```
    public synchronized boolean allowRequest(String userId) {
        long now = System.currentTimeMillis();
        long currentBucketTime = now - (now % bucketSizeInMillis);

        userWindows.putIfAbsent(userId, new ArrayDeque<>());
        Deque<Window> windows = userWindows.get(userId);

        // Remove expired buckets
        while (!windows.isEmpty() && now - windows.peekFirst().timestamp >= windowSizeIn
Millis) {
            windows.pollFirst();
        }

        // Sum the counts in the current sliding window
        int totalCount = windows.stream().mapToInt(w -> w.count).sum();
        if (totalCount >= maxRequests) {
            return false;
        }

        // Update or create the current bucket
        if (!windows.isEmpty() && windows.getLast().timestamp == currentBucketTime) {
            windows.getLast().count++;
        } else {
            windows.addLast(new Window(currentBucketTime, 1));
        }

        return true;
    }
}
```

🧪 Example Usage

```
public class Main {
    public static void main(String[] args) throws InterruptedException {
        SlidingWindowCounterRateLimiter limiter = new SlidingWindowCounterRateLimiter(5, 1
0, 5); // 5 req per 10s

        String userId = "user123";

        for (int i = 1; i <= 7; i++) {
            boolean allowed = limiter.allowRequest(userId);
```

```
            System.out.println("Request " + i + ": " + (allowed ? "✅ Allowed" : "❌ Rejected"));
            Thread.sleep(1500); // 1.5s between requests
        }
    }
}
```

📊 Time & Space Complexity

Operation	Complexity
`allowRequest`	O(k), k = number of buckets
Space	O(k × n) where `n` = number of users

⚖️ Pros & Cons

✅ Pros	❌ Cons
Memory-efficient (only k buckets)	Slightly less accurate than log
Predictable performance	Fixed granularity
Easier to scale/distribute	May need sync in multi-node apps

When to Use Sliding Window Counter?

- For **high-traffic systems**.
- When **memory optimization** is important.
- When **approximate accuracy** is acceptable.

6. `Design Load Balancer`

Designing a **load balancer using Java** involves building a component that accepts incoming requests and distributes them across multiple backend servers to balance the load. Here's a basic yet extendable design with a focus on modularity and clarity.

🧠 Core Concepts

- **Load Balancer** accepts requests from clients.
- **Routing Strategy** determines which backend server gets the request.
- **Health Check** (optional) ensures only healthy servers receive traffic.

🧱 Basic Architecture

```
Client ⟶ LoadBalancer (Java) ⟶ Backend Servers
```

✅ Features in This Simple Version:

- Round-robin load balancing
- Server health check (ping simulation)
- Concurrent handling using thread pool

📦 1. Server Class

```
public class Server {
    private final String url;
    private boolean isAlive;

    public Server(String url) {
        this.url = url;
        this.isAlive = true;
    }

    public String getUrl() {
        return url;
    }

    public boolean isAlive() {
        return isAlive;
    }

    public void setAlive(boolean status) {
        this.isAlive = status;
    }

    public void handleRequest(String request) {
        System.out.println("Handling request: " + request + " at " + url);
    }
}
```

🔄 2. Load Balancer Class (Round Robin)

```
import java.util.List;
import java.util.concurrent.atomic.AtomicInteger;

public class LoadBalancer {
    private final List<Server> servers;
    private final AtomicInteger currentIndex = new AtomicInteger(0);

    public LoadBalancer(List<Server> servers) {
        this.servers = servers;
    }

    public synchronized Server getNextAvailableServer() {
        int index = currentIndex.getAndIncrement() % servers.size();
        for (int i = 0; i < servers.size(); i++) {
            Server server = servers.get((index + i) % servers.size());
            if (server.isAlive()) {
                return server;
            }
        }
        throw new RuntimeException("No alive servers available.");
    }

    public void forwardRequest(String request) {
        Server server = getNextAvailableServer();
        server.handleRequest(request);
    }
}
```

🧪 3. Main App to Simulate Load

```
import java.util.Arrays;
import java.util.List;

public class LoadBalancerApp {
    public static void main(String[] args) {
        List<Server> servers = Arrays.asList(
            new Server("http://localhost:8081"),
            new Server("http://localhost:8082"),
            new Server("http://localhost:8083")
        );

        LoadBalancer lb = new LoadBalancer(servers);
```

```
        for (int i = 0; i < 10; i++) {
            String req = "Request_" + i;
            lb.forwardRequest(req);
        }
    }
}
```

🧩 Optional Enhancements

- ✅ **Health Check Thread** (ping each server every few seconds).
- 💡 **Different Routing Algorithms**: Least connections, IP hash, Weighted round robin.
- 🌐 **Real Networking**: Use sockets or HTTP clients to actually forward real requests.
- 🔒 **Thread Safety**: Use concurrent collections if expanding.
- 📊 **Metrics & Logging**

There are several **types of load balancers**, classified based on how they distribute traffic, what layer of the network stack they operate on, and how sophisticated their routing logic is.

🧱 By OSI Layer

1. Layer 4 (Transport Layer) Load Balancer

- Works at the TCP/UDP level.
- Routes traffic based on **IP address, port**, and **protocol**.
- **Example:** AWS Network Load Balancer, HAProxy (L4 mode).
- ⚡ Fast and lightweight.

2. Layer 7 (Application Layer) Load Balancer

- Works at the HTTP/HTTPS level.
- Routes traffic based on **URL path, headers, cookies, query params**.
- **Example:** NGINX, AWS Application Load Balancer, Traefik.
- 🧠 Supports smart routing, SSL termination, authentication.

🔁 By Routing Algorithm

1. Round Robin

- Each server gets traffic in turn.
- ⚖️ Good for evenly distributed loads.

2. Least Connections

- Chooses the server with the fewest active connections.
- 📉 Reduces the chance of overloading a busy server.

3. IP Hash

- Hashes the client's IP to determine which server to route to.
- 🔁 Ensures session stickiness.

4. Weighted Round Robin

- Servers are assigned weights based on capacity.
- 🏋️ More powerful servers get more requests.

5. Random

- Randomly selects a backend.
- ⚠️ Not ideal for production unless paired with health checks.

🌐 By Deployment Type

1. Hardware Load Balancer

- Physical device (e.g., F5, Cisco).
- 🏢 Often used in enterprise data centers.
- 💰 Expensive but high-performance.

2. Software Load Balancer

- Runs on standard servers (e.g., NGINX, HAProxy, Envoy).
- 🧩 Configurable, open source, cloud-native options.

3. Cloud Load Balancer

- Managed services (e.g., AWS ELB, GCP Load Balancer, Azure LB).
- ☁️ Scalable, integrated with cloud infra, pay-per-use.

🧠 Smart/Modern Load Balancers

Service Mesh Load Balancing (e.g., Istio, Linkerd)

- Uses **sidecar proxies** to handle inter-service communication.
- 🎯 Fine-grained traffic control, retries, circuit breaking.

Global Load Balancers (GSLB)

- Distributes traffic across regions/data centers.
- 🗺️ Uses DNS or Anycast routing.
- **Example:** Cloudflare Load Balancer, AWS Route 53 with health checks.

🎯 Weighted Round Robin (WRR) – Explained

Weighted Round Robin is an enhancement of the basic Round Robin algorithm that distributes traffic **proportionally** to server "weights." It allows **more powerful or capable servers to receive more requests** than less capable ones.

🔁 How It Works

Each server is assigned a **weight** based on its capacity (CPU, memory, bandwidth, etc.).

- A server with a higher weight gets **more requests per cycle**.
- After all weights are accounted for, the cycle repeats.

📌 Example

Let's say you have 3 servers:

Server	Weight
A	3
B	1
C	2

Then over 6 requests, WRR would distribute like this:

```
Request 1 → A
Request 2 → A
Request 3 → A
Request 4 → C
Request 5 → C
Request 6 → B
```

✅ Advantages

- ⚖️ Balances load **according to server capacity**.
- 🔁 Simple to implement.
- 📈 Can be **dynamically updated** based on real-time metrics.

🚫 Disadvantages

- Doesn't account for **connection time** or **server response time**.
- Needs tuning if server performance changes dynamically.

🧱 Java Implementation (Conceptual)

```
public class WeightedRoundRobin {
    private static class Server {
        String name;
        int weight;
        int currentWeight;

        public Server(String name, int weight) {
            this.name = name;
            this.weight = weight;
            this.currentWeight = 0;
        }
    }

    private final List<Server> servers;
    private int totalWeight;

    public WeightedRoundRobin(List<Server> servers) {
        this.servers = servers;
        this.totalWeight = servers.stream().mapToInt(s -> s.weight).sum();
    }

    public synchronized Server getNextServer() {
        Server best = null;
        for (Server server : servers) {
            server.currentWeight += server.weight;
            if (best == null || server.currentWeight > best.currentWeight) {
                best = server;
            }
        }
```

```
        if (best != null) {
            best.currentWeight -= totalWeight;
        }
        return best;
    }
}
```

🧪 Example Usage:

```
List<WeightedRoundRobin.Server> serverList = List.of(
    new WeightedRoundRobin.Server("A", 3),
    new WeightedRoundRobin.Server("B", 1),
    new WeightedRoundRobin.Server("C", 2)
);

WeightedRoundRobin wrr = new WeightedRoundRobin(serverList);

for (int i = 0; i < 12; i++) {
    System.out.println("Request " + i + " → " + wrr.getNextServer().name);
}
```

⚙️ Given:

```
List<Server> servers = List.of(
    new Server("A", 3),
    new Server("B", 1),
    new Server("C", 2)
);
```

📊 Initial State:

Each server has:

Server	Weight	Current Weight
A	3	0
B	1	0
C	2	0

 `totalWeight = 3 + 1 + 2 = 6`

🔁 Iterations (6 requests):

Request 1

- A: `0 + 3 = 3`
- B: `0 + 1 = 1`
- C: `0 + 2 = 2`
- **Pick A (max = 3)**

  A → `3 - 6 = -3`

  ✅ **Chosen: A**

Request 2

- A: `3 + 3 = 0`
- B: `1 + 1 = 2`
- C: `2 + 2 = 4`
- **Pick C (max = 4)**

  C → `4 - 6 = -2`

  ✅ **Chosen: C**

Request 3

- A: `0 + 3 = 3`
- B: `2 + 1 = 3`
- C: `2 + 2 = 0`
- **Pick A (tie, but appears first)**

  A → `3 - 6 = -3`

  ✅ **Chosen: A**

Request 4

- A: `3 + 3 = 0`
- B: `3 + 1 = 4`
- C: `0 + 2 = 2`
- **Pick B (max = 4)**

  B → `4 - 6 = -2`

  ✅ **Chosen: B**

Request 5

- A: `0 + 3 = 3`
- B: `2 + 1 = -1`
- C: `2 + 2 = 4`
- **Pick C (max = 4)**

  C → `4 - 6 = -2`

  ✅ **Chosen: C**

Request 6

- A: `3 + 3 = 6`
- B: `1 + 1 = 0`
- C: `2 + 2 = 0`
- **Pick A (max = 6)**

  A → `6 - 6 = 0`

  ✅ **Chosen: A**

✅ Final Output (Request-wise server)

```
Request 1 → A
Request 2 → C
Request 3 → A
Request 4 → B
Request 5 → C
Request 6 → A
```

💡 This pattern respects the weights:

- A (3 times)
- C (2 times)
- B (1 time)

🧠 Key Difference:

✅ Smooth Weighted Round Robin:

- It **balances the distribution over time**, ensuring that servers are not hit back-to-back too often, even with higher weight.

- It dynamically adjusts `currentWeight` to **smooth out request spikes**, which is why you saw this distribution in the dry run:

  ```
  A → C → A → B → C → A
  ```

❌ Naive Weighted Round Robin:

- Just unrolls a list according to weights (e.g. `[A, A, A, C, C, B]`) and rotates through it.
- Does **not adapt** to new weights or load during runtime.

🔁 Want That A-A-A-C-C-B Style?

If you *do* want the fixed-order round like:

```
[A, A, A, C, C, B] → rotate
```

Then you need a **static repeated list approach** like:

```
List<String> sequence = Arrays.asList("A", "A", "A", "C", "C", "B");
int pointer = 0;

String getNext() {
    String server = sequence.get(pointer);
    pointer = (pointer + 1) % sequence.size();
    return server;
}
```

This is **not ideal** if:

- Weights change dynamically
- You want fairness over time

```python
def load_balancer(requests: Stream[Request], servers: List[Server], weights: List[float]):
    # class Request, Server, Stream, List all predefined. You can assume reasonably any function exists in the class.
    # there are continuous requests coming into the "requests" stream. You need to distribute it fairly across the list of servers.
    # server.assign(request)
    # you can assume that all servers are exactly the same, and have inifinite capacity, so they will never get busy.
    # request.hash() -- assume that hash returns a random integer.

    cumulative_weights = []
    total_weight = 0

    # Compute cumulative weights
    for weight in weights:
        total_weight += weight
        cumulative_weights.append(total_weight)
        # prefix array

    # TASK 1: done
    requests.onNewRequest(lambda request: () => {
        hashCode = request.hash() % servers.length
        server = servers[hashCode]
        server.assign(request)
    })

    # TASK 2: done
    requests.onNewRequest(lambda request: () => {
        ratio = request.hash() % total_weight
        server_index = bisect_left(cumulative_weights, ratio)
        servers[server_index].assign(request)

        # over time, server1 should get roughly 2/37 of the total requests
        # over time, server2 should get roughly 5/37 of the total requests,
        # over time, server3 should get roughly 10/37 of the total requests.
        # over time, server4 should get roughly 20/37 of the total requests.
        # after every 37 or so requests (on average), 1 -2, 2- 5, 3- 10, 4- 20

# Example: If weights = [2, 5, 10, 20], then
# cumulative_weights = [2, 7, 17, 37]

# Suppose request.hash() % 37 = 6, the request falls into the range [2, 7], so it goes to Server 2.
# Suppose request.hash() % 37 = 15, the request falls into [7, 17], so it goes to Server 3.

def bisect_left(arr, target):
    left, right = 0, len(arr)
    while left < right:
        mid = (left + right) // 2
        if arr[mid] < target:
            left = mid + 1
        else:
            right = mid
    return left
```

7. Design Logger System

Design Pattern Used:

Chain Of Responsibility Structural Design Pattern

🧱 Step-by-step Structure

1. LogLevel Enum

```
public enum LogLevel {
    INFO,
    DEBUG,
```

```
    ERROR
}
```

2. Abstract Logger Handler

```
public abstract class Logger {
    protected LogLevel level;
    protected Logger next;

    public Logger(LogLevel level) {
        this.level = level;
    }

    public void setNext(Logger nextLogger) {
        this.next = nextLogger;
    }

    public void log(LogLevel logLevel, String message) {
        if (this.level == logLevel) {
            write(message);
        }
        if (next != null) {
            next.log(logLevel, message);
        }
    }

    protected abstract void write(String message);
}
```

3. Concrete Loggers

```
public class ConsoleLogger extends Logger {
    public ConsoleLogger(LogLevel level) {
        super(level);
    }

    @Override
    protected void write(String message) {
        System.out.println("[Console] " + message);
    }
}
```

```
public class FileLogger extends Logger {
    public FileLogger(LogLevel level) {
        super(level);
    }

    @Override
    protected void write(String message) {
        System.out.println("[File] " + message);
    }
}

public class ErrorLogger extends Logger {
    public ErrorLogger(LogLevel level) {
        super(level);
    }

    @Override
    protected void write(String message) {
        System.err.println("[Error] " + message);
    }
}
```

4. Chain Setup (Client Code)

```
public class LoggerChain {
    public static Logger getChainOfLoggers() {
        Logger errorLogger = new ErrorLogger(LogLevel.ERROR);
        Logger fileLogger = new FileLogger(LogLevel.DEBUG);
        Logger consoleLogger = new ConsoleLogger(LogLevel.INFO);
        errorLogger.setNext(fileLogger);
        fileLogger.setNext(consoleLogger);
        return errorLogger; // return head of the chain
    }

    public static void main(String[] args) {
        Logger loggerChain = getChainOfLoggers();

        loggerChain.log(LogLevel.INFO, "This is an info message");
        loggerChain.log(LogLevel.DEBUG, "This is a debug message");
        loggerChain.log(LogLevel.ERROR, "This is an error message");
    }
}
```

✅ Output:

```
[Console] This is an info message
[File] This is a debug message
[Error] This is an error message
```

💡 Summary:

Log Level	Logger
INFO	ConsoleLogger
DEBUG	FileLogger
ERROR	ErrorLogger

Each logger:

- Processes the log **only if its level matches**
- Always forwards the message to the **next handler**

8. Design bit.ly

Designing **Bit.ly** (a URL shortening service) involves system design principles to handle **short URL generation**, **redirection**, **analytics**, and **scalability**.

🔧 Requirements

1. Functional

- Shorten a given long URL.
- Redirect from short URL to original long URL.
- Track analytics: clicks, user agent, location, etc.
- Support custom aliases (optional).
- Expiry time (optional).

2. Non-functional

- High availability & low latency.
- Horizontal scalability.
- High throughput (read-heavy system).

- Consistency in redirection.

🧰 System Components

1. **Frontend**: Input long URL → Output short URL.
2. **API Gateway**: Routes shortening & redirection.
3. **Shortening Service**: Generates short URL.
4. **Redirection Service**: Resolves and redirects.
5. **Database**: Stores mappings & metadata.
6. **Analytics Service** (optional): Tracks usage.

🗃️ Database Schema

URLs Table:

Field	Type	Description
id	int (auto)	Primary key
long_url	TEXT	Full URL
short_code	VARCHAR(8)	Unique short alias
created_at	TIMESTAMP	Creation time
expiration	TIMESTAMP	Optional expiration
click_count	INT	Number of times accessed

🔐 Short URL Generation

✅ Options:

1. **Hashing (base62 of auto-increment id)**

   Example: ID 125 → base62(125) = `cb`
2. **UUID + base62 (for distributed)**
3. **Custom alias (if user provides one)**

🔄 Base62 Encoding

```
private static final String BASE62 =
"0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";

public String encode(int id) {
    StringBuilder sb = new StringBuilder();
```

```
    while (id > 0) {
        sb.append(BASE62.charAt(id % 62));
        id /= 62;
    }
    return sb.reverse().toString();
}
```

🔁 Redirection Flow

1. Request: `GET /cb`
2. Look up `cb` in DB → get `long_url`
3. Respond with HTTP 302 Redirect

💥 Analytics (Optional)

- Store logs in a distributed queue (Kafka).
- Async processing for geo/clicks/device.

🧱 System Design Diagram

```
User
 ↓
Load Balancer
 ↓
+--------------------+
| API Gateway        |
+--------------------+
   ↓         ↓
Shorten URL   Redirect URL
 Service       Service
   ↓         ↓
+-----------------------+
|    Database (URLs)    |
+-----------------------+
   ↓
[Optional] Analytics DB
```

⚖️ Scaling Strategy

- **Caching**: Use Redis/Memcached to cache hot `short_code → long_url` mappings.

- **DB Sharding**: Based on short_code prefix.
- **CDN**: For static frontend.
- **Rate Limiting**: To prevent abuse.

🛡️ Security

- Validate URLs.
- Prevent XSS/phishing.
- Expiry tokens (optional).
- Abuse detection.

⚙️ Tech Stack (example)

Layer	Tech
Frontend	React, HTML/CSS
Backend	Java/Spring Boot / Node.js
DB	PostgreSQL / MongoDB
Caching	Redis
Queue (optional)	Kafka
Load Balancer	NGINX / AWS ALB
Deployment	Docker + Kubernetes

✅ Tech Stack

- **Spring Boot**
- **PostgreSQL**
- **Base62 encoding**
- **Lombok** (for brevity)
- **JPA (Hibernate)**

🧱 1. Database Schema (PostgreSQL)

```
CREATE TABLE urls (
    id SERIAL PRIMARY KEY,
```

```
    long_url TEXT NOT NULL,
    short_code VARCHAR(10) UNIQUE NOT NULL,
    created_at TIMESTAMP DEFAULT NOW(),
    expiration TIMESTAMP,
    click_count INTEGER DEFAULT 0
);
```

📦 2. Entity Class

```
@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Url {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, columnDefinition = "TEXT")
    private String longUrl;

    @Column(unique = true)
    private String shortCode;

    private LocalDateTime createdAt = LocalDateTime.now();
    private LocalDateTime expiration;
    private Integer clickCount = 0;
}
```

🔠 3. Base62 Encoder Utility

```
@Component
public class Base62Encoder {
    private static final String BASE62 = "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJK
LMNOPQRSTUVWXYZ";

    public String encode(Long id) {
        StringBuilder sb = new StringBuilder();
        while (id > 0) {
            sb.append(BASE62.charAt((int) (id % 62)));
            id /= 62;
```

```
        }
        return sb.reverse().toString();
    }
}
```

💾 4. Repository

```
public interface UrlRepository extends JpaRepository<Url, Long> {
    Optional<Url> findByShortCode(String shortCode);
}
```

🧠 5. Service Layer

```
@Service
@RequiredArgsConstructor
public class UrlService {
    private final UrlRepository urlRepository;
    private final Base62Encoder encoder;

    public String shortenUrl(String longUrl) {
        Url url = new Url();
        url.setLongUrl(longUrl);
        url = urlRepository.save(url); // generates ID

        String shortCode = encoder.encode(url.getId());
        url.setShortCode(shortCode);
        urlRepository.save(url);

        return shortCode;
    }

    public String resolveUrl(String shortCode) {
        Url url = urlRepository.findByShortCode(shortCode)
            .orElseThrow(() -> new RuntimeException("Short URL not found"));

        url.setClickCount(url.getClickCount() + 1);
        urlRepository.save(url);

        return url.getLongUrl();
```

```
    }
}
```

🌐 6. Controller

```
@RestController
@RequiredArgsConstructor
public class UrlController {
    private final UrlService urlService;

    @PostMapping("/shorten")
    public String shorten(@RequestParam String longUrl) {
        return urlService.shortenUrl(longUrl);
    }

    @GetMapping("/{shortCode}")
    public ResponseEntity<?> redirect(@PathVariable String shortCode) {
        String longUrl = urlService.resolveUrl(shortCode);
        return ResponseEntity.status(HttpStatus.FOUND)
                .location(URI.create(longUrl))
                .build();
    }
}
```

🚀 7. Sample Usage

➤ Request

```
POST /shorten
Body: longUrl=https://example.com/very/long/path
```

➤ Response

```
shortCode: aZ3F
```

➤ Redirect

```
GET /aZ3F
```

Returns HTTP 302 redirect to `https://example.com/very/long/path`.

🚧 Next Enhancements

- Add **expiry support**
- Add **custom aliases**
- Add **rate limiting**
- Add **Redis cache** for resolving short codes

🧰 Steps to Add Redis Caching

✅ Overview

- Use **Spring Cache Abstraction**.
- Configure Redis in your project.
- Annotate methods to use cache for lookups.

📦 1. Add Redis Dependency

In `pom.xml`:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-cache</artifactId>
</dependency>
```

⚙️ 2. Redis Configuration (in `application.properties` or `application.yml`)

```
spring.cache.type=redis
spring.redis.host=localhost
spring.redis.port=6379
```

🧠 3. Enable Caching

In your main application class:

```
@SpringBootApplication
@EnableCaching
public class UrlShortenerApplication {
    public static void main(String[] args) {
        SpringApplication.run(UrlShortenerApplication.class, args);
    }
}
```

💾 4. Update Service with Caching

```
@Service
@RequiredArgsConstructor
public class UrlService {
    private final UrlRepository urlRepository;
    private final Base62Encoder encoder;

    public String shortenUrl(String longUrl) {
        Url url = new Url();
        url.setLongUrl(longUrl);
        url = urlRepository.save(url); // gets ID

        String shortCode = encoder.encode(url.getId());
        url.setShortCode(shortCode);
        urlRepository.save(url);

        return shortCode;
    }

    @Cacheable(value = "shortUrlCache", key = "#shortCode")
    public String resolveUrl(String shortCode) {
        Url url = urlRepository.findByShortCode(shortCode)
            .orElseThrow(() -> new RuntimeException("Short URL not found"));

        url.setClickCount(url.getClickCount() + 1);
        urlRepository.save(url);

        return url.getLongUrl();
    }
}
```

🧪 5. Run Redis Locally

If you don't have Redis:

- Install via Docker:

```
docker run --name redis -p 6379:6379 redis
```

🧹 Optional: Customize TTL (Time To Live)

To set expiry for cache entries, use:

```
@Cacheable(value = "shortUrlCache", key = "#shortCode", unless = "#result == null")
@CacheConfig(cacheNames = "shortUrlCache")
```

Or configure TTL via a `RedisCacheConfiguration` bean.

✅ Summary

- ✅ `@Cacheable` ensures fast redirection for repeat requests.
- ✅ Redis stores shortCode → longUrl mapping.
- ✅ DB is only hit once per new shortCode access.

✅ Goal:

> Cache each shortened URL with its own expiration (TTL) based on the expiration field in the database.

🔧 Steps Overview:

1. Store `shortCode → longUrl` in Redis with a **custom TTL**.
2. Fetch from Redis first.
3. On cache miss, read from DB, cache it with TTL.

🧰 Step-by-Step Implementation

1. Inject RedisTemplate

```
@Configuration
public class RedisConfig {
    @Bean
    public RedisTemplate<String, String> redisTemplate(RedisConnectionFactory factory) {
        RedisTemplate<String, String> template = new RedisTemplate<>();
        template.setConnectionFactory(factory);
        return template;
    }
}
```

2. Modify the `UrlService` to handle custom TTL:

```
@Service
@RequiredArgsConstructor
public class UrlService {
    private final UrlRepository urlRepository;
    private final Base62Encoder encoder;
    private final RedisTemplate<String, String> redisTemplate;

    private static final String PREFIX = "shortUrl:";

    public String shortenUrl(String longUrl, Duration ttl) {
        Url url = new Url();
        url.setLongUrl(longUrl);
        url.setExpiration(LocalDateTime.now().plus(ttl));
        url = urlRepository.save(url);

        String shortCode = encoder.encode(url.getId());
        url.setShortCode(shortCode);
        urlRepository.save(url);

        return shortCode;
    }

    public String resolveUrl(String shortCode) {
        String redisKey = PREFIX + shortCode;

        // Step 1: Check cache
        String cached = redisTemplate.opsForValue().get(redisKey);
        if (cached != null) {
            return cached;
        }
```

```
        // Step 2: Fallback to DB
        Url url = urlRepository.findByShortCode(shortCode)
            .orElseThrow(() -> new RuntimeException("Short URL not found"));

        if (url.getExpiration() != null && url.getExpiration().isBefore(LocalDateTime.now())) {
            throw new RuntimeException("URL has expired");
        }

        // Step 3: Cache with custom TTL
        Duration ttl = Duration.between(LocalDateTime.now(),
        url.getExpiration());
        redisTemplate.opsForValue().set(redisKey, url.getLongUrl(), ttl);

        return url.getLongUrl();
    }
}
```

3. Modify `shortenUrl` endpoint to accept TTL

```
@PostMapping("/shorten")
public String shorten(
    @RequestParam String longUrl,
    @RequestParam(defaultValue = "7") int days
) {
    return urlService.shortenUrl(longUrl, Duration.ofDays(days));
}
```

🧪 Example

```
POST /shorten?longUrl=https://example.com&days=3
```

➡️ Will store the mapping in Redis with a TTL of 3 days.

✅ Final Result:

- Each short URL is cached with **its own TTL**.
- After expiration, it's removed from Redis.
- DB access is only used on first request or after expiration.

9. Design Unique ID Generator Service

Designing a **Unique ID Generator Service** (like Twitter's Snowflake) involves several decisions about scale, uniqueness, time-ordering, and distributed reliability. Here's a complete design:

📌 Goal

Build a service that generates **unique**, **time-sortable** IDs with:

- High throughput (e.g., thousands/millions per second)
- Low latency
- Optional decentralization (works across services/data centers)

🧩 Use Cases

- URL shorteners (e.g., Bit.ly)
- Order IDs
- Tweet IDs
- Distributed DB row keys (e.g., Cassandra, DynamoDB)

🧱 Design Approaches

1. UUID (Universally Unique Identifier)

- ❌ Not sortable, longer (128 bits), less efficient
- ✅ No coordination needed

2. Centralized Counter

- ✅ Simple: Increment counter in DB
- ❌ Bottleneck, Single point of failure

3. Twitter Snowflake (Recommended)

- ✅ Time-based, sharded, distributed, sortable

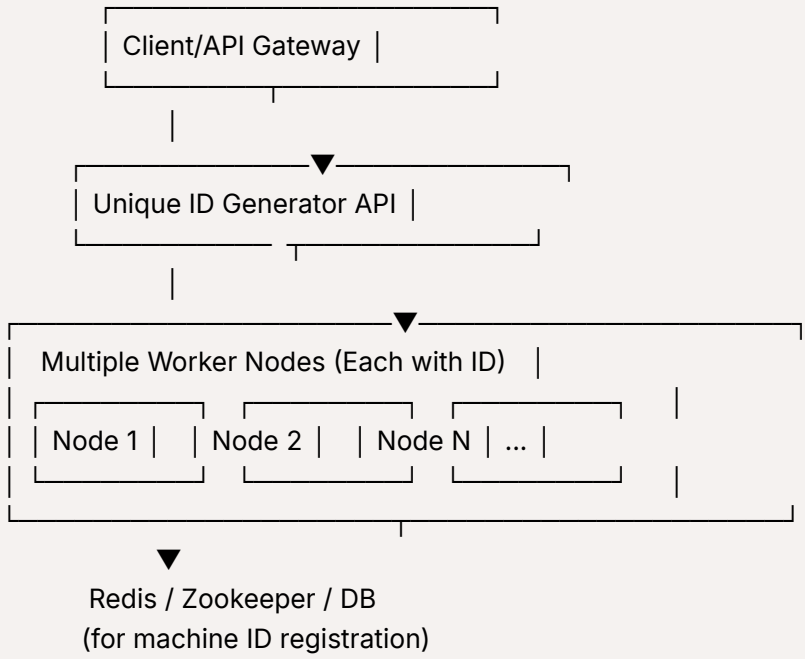
🧮 Snowflake-inspired 64-bit ID Format

```
[ 1 Bit ] [ 41 Bits ] [ 10 Bits ] [ 12 Bits ]
  sign     timestamp   machineID   sequence
```

Component	Description
Sign bit	Always 0
Timestamp	Milliseconds since custom epoch
Machine ID	Identifies the worker/machine
Sequence	Auto-increment within the same ms

- 41 bits = 69 years of time
- 10 bits = 1024 machines
- 12 bits = 4096 IDs per machine/ms

⚙️ System Design Diagram



🔐 Guarantees

Property	Ensured By
Uniqueness	Machine ID + sequence
Time-ordering	Timestamp component
High throughput	Parallel node generation
Fault tolerance	Retry with new timestamp

🧰 Tech Stack Suggestions

Component	Tool/Library
Language	Java / Go / Node / Python
Clock sync	NTP (Network Time Protocol)
Coordination	Redis / ZooKeeper / Etcd
Scaling	Kubernetes / Horizontal Scale

🔁 Edge Cases to Handle

- **Clock rollback** → Use a fallback wait or monotonic clock
- **Sequence overflow** in same millisecond → Wait for next millisecond
- **Machine ID conflict** → Coordination service to assign IDs

🧪 Sample Java Code (Simplified)

```
public class UniqueIdGenerator {
    private final long machineId;
    private final long epoch = 1700000000000L; // custom epoch
    private long lastTimestamp = -1L;
    private long sequence = 0L;

    private final int machineBits = 10;
    private final int sequenceBits = 12;

    private final long maxSequence = ~(-1L << sequenceBits);
    private final long machineShift = sequenceBits;
    private final long timeShift = sequenceBits + machineBits;

    public UniqueIdGenerator(long machineId) {
        this.machineId = machineId;
    }

    public synchronized long nextId() {
        long timestamp = System.currentTimeMillis();

        if (timestamp < lastTimestamp)
            throw new RuntimeException("Clock moved backwards!");

        if (timestamp == lastTimestamp) {
            sequence = (sequence + 1) & maxSequence;
            if (sequence == 0) {
                // Sequence exhausted in 1ms, wait for next ms
```

```
            while ((timestamp = System.currentTimeMillis()) <= lastTimestamp);
        }
    } else {
        sequence = 0;
    }

    lastTimestamp = timestamp;
    return ((timestamp - epoch) << timeShift)
          | (machineId << machineShift)
          | sequence;
  }
}
```

🔄 Optional Enhancements

- Custom TTL per ID (store in Redis)
- REST API to get/batch-generate IDs
- Metrics dashboard for ID generation rates

✅ Simplified Design Goals

- Unique ID generation
- Fast and reliable
- Easy to scale across services
- Optional time-based sorting

🧱 ID Format (Basic)

You can build a **64-bit ID** using:

```
[ timestamp (in ms) ] + [ machine ID ] + [ counter ]
```

For example:

```
< 13-digit time > + < 2-digit machine ID > + < 3-digit sequence >
= 18-digit ID like: 1716910012345001
```

🛠️ Simple Java Implementation

```
public class SimpleIdGenerator {
    private final int machineId;
    private long lastTimestamp = -1L;
    private int counter = 0;

    public SimpleIdGenerator(int machineId) {
        this.machineId = machineId;
    }

    public synchronized String generateId() {
        long currentTimestamp = System.currentTimeMillis();

        if (currentTimestamp == lastTimestamp) {
            counter++;
        } else {
            counter = 0;
            lastTimestamp = currentTimestamp;
        }

        return currentTimestamp + String.format("%02d%03d", machineId, counter);
    }
}
```

Example Output:

```
1716913456123001
1716913456123002
```

🧩 Key Components

Component	Purpose
`timestamp`	Ensures time-based uniqueness
`machineId`	Differentiates between instances
`counter`	Handles multiple requests per ms

⚡ Advantages

- Simple and thread-safe
- No database needed
- Time-sortable IDs

- Good enough for medium-scale apps

🛡️ Optional Add-ons

- Store generated IDs in Redis for deduplication
- Use a hash (e.g., base62 encoding) for shorter URLs
- Use in-memory counters (like AtomicInteger) per instance

✅ Given

```
SimpleIdGenerator generator = new SimpleIdGenerator(12);
```

🧪 1st Call: `generateId()`

- Assume `System.currentTimeMillis()` returns `1716912345678`
- `lastTimestamp = -1`, so `currentTimestamp != lastTimestamp`

Execution Flow:

```
counter = 0;
lastTimestamp = 1716912345678;
```

Return:

```
"1716912345678" + String.format("%02d%03d", 12, 0)
= "171691234567812000"
```

🧪 2nd Call: `generateId()` (within the same millisecond)

- `System.currentTimeMillis()` = `1716912345678` (same as last call)
- `currentTimestamp == lastTimestamp` → increment counter

Execution Flow:

```
counter = 1;
```

Return:

```
"1716912345678" + String.format("%02d%03d", 12, 1)
= "171691234567812001"
```

🧪 3rd Call: `generateId()` (in a new millisecond)

- `System.currentTimeMillis()` = `1716912345679`
- `currentTimestamp != lastTimestamp`

Execution Flow:

```
counter = 0;
lastTimestamp = 1716912345679;
```

Return:

```
"1716912345679" + String.format("%02d%03d", 12, 0)
= "171691234567912000"
```

✅ Summary

Call	Timestamp	Counter	Output ID
1	1716912345678	0	171691234567812000
2	1716912345678	1	171691234567812001
3	1716912345679	0	171691234567912000

---

10. `Design URL Shortener`

Designing a **URL Shortener** like bit.ly involves converting long URLs into short aliases while ensuring uniqueness, scalability, and optional analytics.

🎯 Goal

Convert a long URL like:

```
https://example.com/articles/some-very-long-name
```

into a short one like:

```
https://sho.rt/XyZ123
```

🧰 Core Components

Component	Responsibility

Frontend / API	Accepts long URL and returns short one
Hashing Service	Generates unique short codes
Database	Maps short codes to original URLs
Redirect Service	Redirects short URL to original long URL

🔐 Requirements

Functional:

- Accept a long URL and return a short URL.
- Redirect short URL to the original.
- (Optional) Expiration date.
- (Optional) Analytics (click count, geo info).

Non-Functional:

- High availability and low latency
- Unique code generation
- Scalable DB

🧩 ID Generation Approaches

1. Auto-Increment + Base62

- Store each URL with an auto-increment ID (`1, 2, 3...`)
- Encode ID using **Base62 (0-9a-zA-Z)** → shorter strings

```
// Example: ID 125 → Base62 = "cb"
```

2. Hashing (MD5/SHA1)

- Hash the URL and use first 6-8 chars
- ❌ May need collision handling

💾 Schema Design (SQL)

```
Table: urls
-----------------------------
| id  | short_code | long_url      | created_at | expiry |
```

```
|-----|------------|----------------|------------|--------|
| 1   | XyZ123     | https://...    | timestamp  | NULL   |
```

🛠️ Simple Java Flow (Base62 ID Approach)

1. Encode Long URL

```
@PostMapping("/shorten")
public String shorten(@RequestBody String longUrl) {
    long id = urlRepository.save(longUrl); // Auto-increment ID
    String shortCode = encodeBase62(id);
    return "https://sho.rt/" + shortCode;
}
```

2. Redirect Short URL

```
@GetMapping("/{shortCode}")
public ResponseEntity<?> redirect(@PathVariable String shortCode) {
    long id = decodeBase62(shortCode);
    String longUrl = urlRepository.findById(id);
    return ResponseEntity.status(302).header("Location", longUrl).build();
}
```

🔁 Base62 Conversion (Java)

```
private static final String BASE62 = "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKL
MNOPQRSTUVWXYZ";

public String encodeBase62(long num) {
    StringBuilder sb = new StringBuilder();
    while (num > 0) {
        sb.append(BASE62.charAt((int)(num % 62)));
        num /= 62;
    }
    return sb.reverse().toString();
}

public long decodeBase62(String str) {
    long num = 0;
    for (char c : str.toCharArray()) {
```

```
        num = num * 62 + BASE62.indexOf(c);
    }
    return num;
}
```

🚀 Scalability Add-ons

Feature	Solution
Caching	Redis (shortCode → longURL)
Expiry support	Add TTL per URL
Analytics	Async log to Kafka, store in DB/Clickhouse
Horizontal scaling	Use distributed ID gen (e.g. Snowflake)
Custom aliases	Let users choose custom shortCode

🧪 Example

```
POST /shorten
Body: https://openai.com/chat
→ Response: https://sho.rt/XyZ123

GET /XyZ123
→ 302 Redirect → https://openai.com/chat
```

11. `Design Tic-Tac-Toe`

🎯 Requirements

Functional:

- A `3×3` board
- Two players: `X` and `O`
- Players take turns
- Detect a win or a draw
- Reset or restart game

Non-functional:

- Easy to extend (e.g., 4×4, AI support)
- Clean, object-oriented design

🧱 Classes and Responsibilities

✅ 1. `TicTacToeGame`

- Core class controlling the game

✅ 2. `Board`

- Represents the `n x n` board

✅ 3. `Player`

- Encapsulates player info (symbol: `X` or `O`)

📦 Class Diagram (Simple)

```
TicTacToeGame
 ├── Board board
 ├── Player[] players
 ├── int currentPlayerIndex
 └── playMove(int row, int col)

Board
 ├── char[][] grid
 └── isValidMove(), makeMove(), checkWin(), checkDraw()

Player
 └── char symbol
```

🧠 Core Logic (Java)

```
class Player {
    char symbol;
    public Player(char symbol) {
        this.symbol = symbol;
    }
}

class Board {
    private final int size;
```

```
    private final char[][] grid;

    public Board(int size) {
        this.size = size;
        grid = new char[size][size];
        for (char[] row : grid) Arrays.fill(row, '-');
    }

    public boolean isValidMove(int row, int col) {
        return row >= 0 && col >= 0 && row < size && col < size && grid[row][col] == '-';
    }

    public void makeMove(int row, int col, char symbol) {
        grid[row][col] = symbol;
    }

    public boolean checkWin(int row, int col, char symbol) {
        boolean rowWin = true, colWin = true, diagWin = true, antiDiagWin = true;

        for (int i = 0; i < size; i++) {
            if (grid[row][i] != symbol) rowWin = false;
            if (grid[i][col] != symbol) colWin = false;
            if (grid[i][i] != symbol) diagWin = false;
            if (grid[i][size - 1 - i] != symbol) antiDiagWin = false;
        }
        return rowWin || colWin || diagWin || antiDiagWin;
    }

    public boolean checkDraw() {
        for (char[] row : grid)
            for (char cell : row)
                if (cell == '-') return false;
        return true;
    }

    public void printBoard() {
        for (char[] row : grid)
            System.out.println(Arrays.toString(row));
    }
}

class TicTacToeGame {
    private final Board board;
```

```
    private final Player[] players;
    private int currentPlayerIndex;

    public TicTacToeGame() {
        board = new Board(3);
        players = new Player[]{new Player('X'), new Player('O')};
        currentPlayerIndex = 0;
    }

    public void play(int row, int col) {
        Player currentPlayer = players[currentPlayerIndex];

        if (!board.isValidMove(row, col)) {
            System.out.println("Invalid move. Try again.");
            return;
        }

        board.makeMove(row, col, currentPlayer.symbol);
        board.printBoard();

        if (board.checkWin(row, col, currentPlayer.symbol)) {
            System.out.println("Player " + currentPlayer.symbol + " wins!");
            return;
        }

        if (board.checkDraw()) {
            System.out.println("Game is a draw!");
            return;
        }

        currentPlayerIndex = 1 - currentPlayerIndex; // Toggle between 0 and 1
    }
}
```

▶️ Example Usage

```
TicTacToeGame game = new TicTacToeGame();
game.play(0, 0); // X
game.play(0, 1); // O
game.play(1, 1); // X
```

```
game.play(0, 2); // O
game.play(2, 2); // X wins!
```

✅ Extensions

Feature	How to Extend
AI player	Add a `ComputerPlayer` with minimax
N x N board	Change `Board(size)` dynamically
Web or UI layer	Wrap logic in REST or UI components
Multiplayer game	Manage users/sockets on backend

```
class TicTacToe {
    private int[] rows;
    private int[] cols;
    private int diag;
    private int antiDiag;
    private int n;

    public TicTacToe(int n) {
        this.n = n;
        rows = new int[n];
        cols = new int[n];
        diag = 0;
        antiDiag = 0;
    }

    public int move(int row, int col, int player) {
        int mark = (player == 1) ? 1 : -1;

        rows[row] += mark;
        cols[col] += mark;

        if (row == col) {
            diag += mark;
        }

        if (row + col == n - 1) {
            antiDiag += mark;
        }
```

```
        if (Math.abs(rows[row]) == n ||
            Math.abs(cols[col]) == n ||
            Math.abs(diag) == n ||
            Math.abs(antiDiag) == n) {
            return player;
        }

        return 0; // No winner yet
    }
}
```

---

12. Design Web Crawler

Designing a **Web Crawler** involves building a system that visits web pages, extracts useful content or links, and recursively continues crawling through discovered links while respecting constraints like domains, depth, and robots.txt.

✅ Problem Statement

Build a **Web Crawler** that:

- Starts from a URL
- Fetches HTML content
- Extracts links from it
- Recursively crawls new URLs (up to depth limit or domain)
- Avoids duplicate visits
- Obeys polite crawling practices (optional for now)

💡 Real-World Constraints

- Parallel crawling
- Rate limiting / politeness
- robots.txt compliance
- Timeout, broken links
- Domain restrictions

🧱 Components

1. `WebCrawler`
   - Starts the crawl
   - Manages visited set
   - Can use BFS (iterative) or DFS
2. `HtmlParser`
   - Extracts hyperlinks from HTML content
3. `Fetcher`
   - Makes HTTP GET requests to fetch pages

🧩 Class Diagram

```
WebCrawler
 ├── Set<String> visited
 ├── Queue<UrlDepthPair> queue
 └── crawl(String startUrl)

Fetcher
 └── fetch(String url) → String html

HtmlParser
 └── parseLinks(String html) → List<String> links
```

🔁 Iterative BFS Java Implementation

```
import java.util.*;
import java.net.*;
import java.io.*;
import java.util.regex.*;

class WebCrawler {

    private Set<String> visited = new HashSet<>();
    private Queue<UrlDepthPair> queue = new LinkedList<>();

    public void crawl(String startUrl, int maxDepth) {
        queue.offer(new UrlDepthPair(startUrl, 0));
        visited.add(startUrl);
```

```
        while (!queue.isEmpty()) {
            UrlDepthPair current = queue.poll();
            if (current.depth > maxDepth) continue;

            System.out.println("Crawling: " + current.url + " at depth " + current.depth);

            String html = Fetcher.fetch(current.url);
            List<String> links = HtmlParser.extractLinks(html);

            for (String link : links) {
                if (!visited.contains(link)) {
                    visited.add(link);
                    queue.offer(new UrlDepthPair(link, current.depth + 1));
                }
            }
        }
    }

    static class UrlDepthPair {
        String url;
        int depth;
        UrlDepthPair(String url, int depth) {
            this.url = url;
            this.depth = depth;
        }
    }
}
```

🌐 Fetcher Class

```
class Fetcher {
    public static String fetch(String urlStr) {
        try {
            URL url = new URL(urlStr);
            BufferedReader in = new BufferedReader(new InputStreamReader(url.openStream()));

            StringBuilder content = new StringBuilder();
            String line;
            while ((line = in.readLine()) != null)
                content.append(line);
            in.close();
```

```
            return content.toString();
        } catch (Exception e) {
            System.out.println("Failed to fetch: " + urlStr);
            return "";
        }
    }
}
```

🔍 Simple HTML Link Extractor

```
class HtmlParser {
    private static final Pattern LINK_PATTERN = Pattern.compile("href=\"(http[s]?://[^\"\\s]+)\"");

    public static List<String> extractLinks(String html) {
        List<String> links = new ArrayList<>();
        Matcher matcher = LINK_PATTERN.matcher(html);
        while (matcher.find()) {
            links.add(matcher.group(1));
        }
        return links;
    }
}
```

📈 Example Usage

```
public class Main {
    public static void main(String[] args) {
        WebCrawler crawler = new WebCrawler();
        crawler.crawl("https://example.com", 2); // BFS up to depth 2
    }
}
```

✅ Extensions

Feature	Implementation Approach
Parallel crawling	Use `ExecutorService` for multi-threading
robots.txt support	Use `UserAgent` and parse `robots.txt`
Domain constraints	Only add links with matching host
Rate limiting	Thread sleep or Token Bucket Algorithm

Content scraping	Extract meta, title, paragraph tags

---

13. Design Search Engine

✅ Problem Statement

Design a system that:

- Crawls web pages
- Indexes content
- Supports fast keyword-based search
- Ranks results based on relevance

🔧 Key Components

1. Web Crawler

Fetches and stores raw web pages.

> Already designed in previous step.

2. Parser / Tokenizer

Extracts plain text and splits into keywords (tokens).

3. Indexer

Builds an **inverted index** mapping each word → list of document IDs (URLs).

4. Ranker

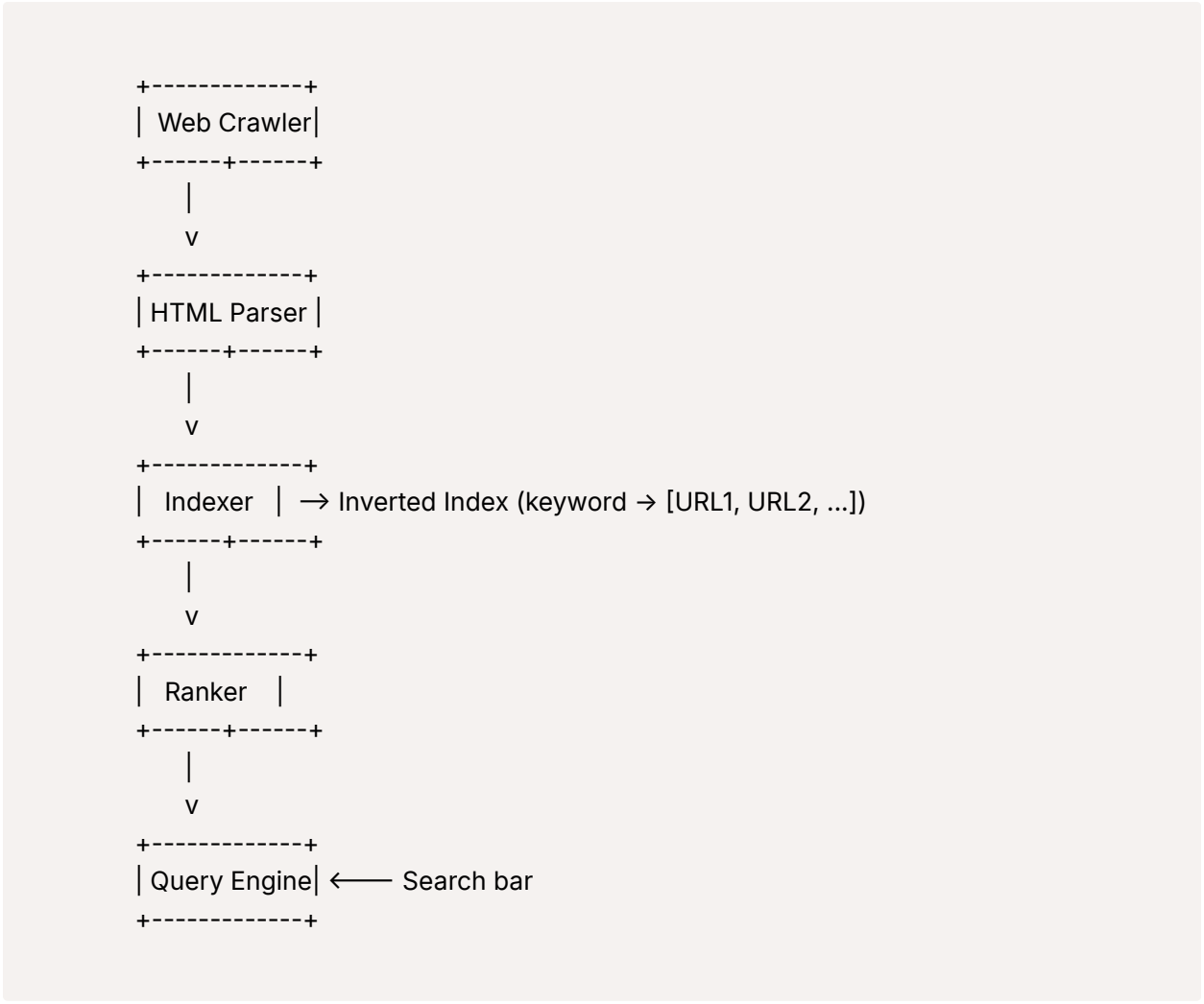
Ranks results using algorithms like:

- TF-IDF
- PageRank
- BM25

5. Query Engine

Handles search queries, matches keywords, and returns ranked results.

🧱 High-Level Architecture



🗂️ Inverted Index (Core Data Structure)

```
Map<String, List<String>> invertedIndex;
```

- `"java"` → `[https://site1.com, https://site2.com]`
- `"design"` → `[https://site1.com]`

🧠 Ranking with TF-IDF (Simplified)

- **TF (Term Frequency)** = # times word appears in doc
- **IDF (Inverse Document Frequency)** = log(N / docs_with_term)
- Score = TF × IDF

✅ Simplified Code: Indexing & Search

```
class SearchEngine {
    Map<String, List<String>> invertedIndex = new HashMap<>();

    // Index content of a page
    public void indexPage(String url, String content) {
        String[] words = content.toLowerCase().split("\\W+");

        for (String word : words) {
            invertedIndex.putIfAbsent(word, new ArrayList<>());
            if (!invertedIndex.get(word).contains(url)) {
                invertedIndex.get(word).add(url);
            }
        }
    }

    // Search for a keyword
    public List<String> search(String query) {
        return invertedIndex.getOrDefault(query.toLowerCase(), List.of());
    }
}
```

🧪 Example Usage

```
public class Main {
    public static void main(String[] args) {
        SearchEngine engine = new SearchEngine();

        engine.indexPage("https://example.com", "Java design patterns and principles");
        engine.indexPage("https://tutorial.com", "Java tutorials and design examples");

        System.out.println(engine.search("java"));     // Both URLs
        System.out.println(engine.search("patterns")); // First URL
    }
}
```

🚀 Scaling Up (Advanced)

Feature	Technology
Distributed Indexing	Apache Lucene / Elasticsearch

Distributed Crawling	Kafka + Worker Queue
Page Ranking	Graph algorithms / PageRank
Search Suggestions	Trie / Prefix Tree
Spell Correction	Levenshtein Distance
Caching Results	Redis / CDN

🧩 Bonus: Add Redis Caching for Search Results

```
Map<String, List<String>> cache = new HashMap<>();

public List<String> search(String query) {
    if (cache.containsKey(query)) return cache.get(query);

    List<String> result = invertedIndex.getOrDefault(query.toLowerCase(), List.of());
    cache.put(query, result);
    return result;
}
```

---

14. Design Parking Lot

✅ Problem Statement

Design a parking lot system in Java that can:

- Track entry/exit of vehicles
- Assign parking spots efficiently
- Calculate parking fees

🔧 Functional Requirements

1. **Parking operations**
    - Park a vehicle
    - Remove a vehicle

- Display available slots

2. **Vehicle types**
   - Car, Bike, Truck (varying slot sizes)
3. **Slot management**
   - Assign nearest available slot
   - Handle different floor levels
4. **Billing system**
   - Calculate parking fee based on duration and vehicle type

---

🚫 Non-Functional Requirements

- Thread-safe operations (concurrent entry/exit)
- High availability
- Scalable design (multiple floors/zones)
- Extendable (add more vehicle types, payment methods)

`Design Patterns Used:`

1. Factory
2. Singleton
3. Strategy
4. Observer
5. Command

🧱 Class Design

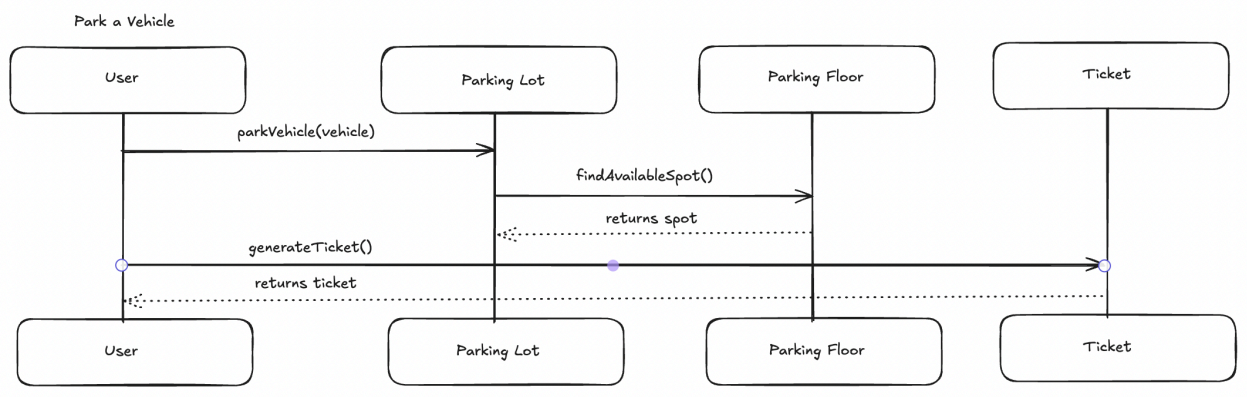
1. Entities:

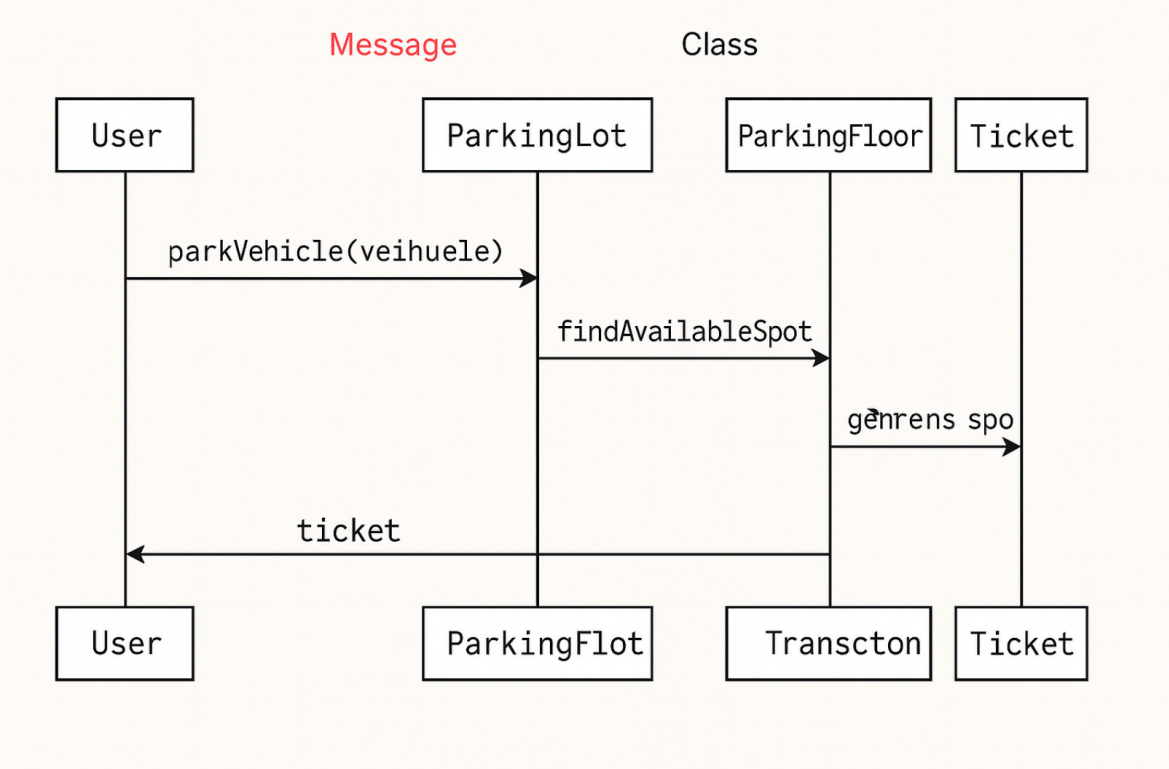
- `Vehicle` (abstract) → `Car`, `Bike`, `Truck`
- `ParkingSpot` → can be `Compact`, `Large`, etc.

- `ParkingFloor`
- `ParkingLot`
- `Ticket`
- `Payment`

plaintext Copy Edit

📊 Sequence Diagram (Park a Vehicle)





Source Code

1. Vehicle abstract class

```
public abstract class Vehicle {
    private final String licensePlate;
```

```
    public Vehicle(String licensePlate) {
        this.licensePlate = licensePlate;
    }

    public abstract VehicleType getType();
}
```

2. Car
3. Bus
4. Truck

```
public class Car extends Vehicle {
    public Car(String plate) {
        super(plate);
    }

    @Override
    public VehicleType getType() {
        return VehicleType.CAR;
    }
}
```

```
public class Bike extends Vehicle {
    public Bike(String plate) {
        super(plate);
    }

    @Override
    public VehicleType getType() {
        return VehicleType.BIKE;
    }
}
```

```
public class Truck extends Vehicle {
    public Truck(String plate) {
        super(plate);
    }
```

```
    @Override
    public VehicleType getType() {
        return VehicleType.TRUCK;
    }
}
```

4. VehicleFactory

```
public class VehicleFactory {
    public static Vehicle createVehicle(String type, String plate) {
        switch (type.toUpperCase()) {
            case "CAR": {
                return new Car(plate);
            }
            case "BIKE": {
                return new Bike(plate);
            }
            case "TRUCK": {
                return new Truck(plate);
            }
            default: {
                throw new IllegalArgumentException("Unknown vehicle type");
            }
        }
    }
}
```

5. ParkingLotObserver
6. ParkingLotSubject
7. Command
8. ParkCommand
9. Parking Command
10. UnparkCommand
11. ParkingCommandInvoker
12. ParkingLot

13. ParkingLotDisplay
14. Main

```
public interface ParkingLotObserver {
    void update(String message);
}
```

```
import java.util.ArrayList;
import java.util.List;

public class ParkingLotSubject {
    private List<ParkingLotObserver> observers = new ArrayList<>();

    public void addObserver(ParkingLotObserver observer) {
        observers.add(observer);
    }

    public void removeObserver(ParkingLotObserver observer) {
        observers.remove(observer);
    }

    protected void notifyObservers(String message) {
        for (ParkingLotObserver o : observers) {
            o.update(message);
        }
    }
}
```

```
public interface Command {
    void execute();
}
```

```
public class ParkCommand implements Command {
    private ParkingLot parkingLot;
    private Vehicle vehicle;
```

```
    public ParkCommand(ParkingLot parkingLot, Vehicle vehicle) {
        this.parkingLot = parkingLot;
        this.vehicle = vehicle;
    }

    @Override
    public void execute() {
        parkingLot.park(vehicle);
    }
}
```

```
public class UnparkCommand implements Command {
    private ParkingLot parkingLot;
    private Vehicle vehicle;

    public UnparkCommand(ParkingLot parkingLot, Vehicle vehicle) {
        this.parkingLot = parkingLot;
        this.vehicle = vehicle;
    }

    @Override
    public void execute() {
        parkingLot.unpark(vehicle);
    }
}
```

```
import java.util.LinkedList;
import java.util.Queue;

public class ParkingCommandInvoker {
    private Queue<Command> commandQueue = new LinkedList<>();

    public void addCommand(Command command) {
        commandQueue.offer(command);
    }

    public void executeCommands() {
        while (!commandQueue.isEmpty()) {
```

```
            Command cmd = commandQueue.poll();
            cmd.execute();
        }
    }
}
```

```
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Optional;

public class ParkingLot extends ParkingLotSubject {
    private static ParkingLot instance;
    private final List<ParkingFloor> floors;
    private final Map<Vehicle, Ticket> activeTickets = new HashMap<>();
    private int capacity;
    private int currentCount = 0;

    private ParkingLot(List<ParkingFloor> floors, int capacity) {
        this.floors = floors;
        this.capacity = capacity;
    }

    public static synchronized ParkingLot getInstance(List<ParkingFloor> floors, int capacity) {
        if (instance == null) {
            instance = new ParkingLot(floors, capacity);
        }
        return instance;
    }

    public synchronized Ticket park(Vehicle vehicle) {
        if (currentCount >= capacity) {
            notifyObservers("Parking Lot Full");
            throw new RuntimeException("Parking Lot Full");
        }
        for (ParkingFloor floor : floors) {
            Optional<ParkingSpot> spotOpt = floor.findAvailable(vehicle);
            if (spotOpt.isPresent()) {
                ParkingSpot spot = spotOpt.get();
                spot.park(vehicle);
                Ticket ticket = new Ticket(vehicle);
```

```
                activeTickets.put(vehicle, ticket);
                currentCount++;
                if (currentCount == capacity) {
                    notifyObservers("Parking Lot Full");
                }
                return ticket;
            }
        }
        throw new RuntimeException("No spot available for vehicle type");
    }

    public synchronized double unpark(Vehicle vehicle) {
        Ticket ticket = activeTickets.remove(vehicle);
        if (ticket == null) {
            throw new RuntimeException("Vehicle not found");
        }
        long hours = ticket.getDurationHours();
        FeeStrategy strategy = FeeStrategyFactory.getStrategy(vehicle.getType());
        double fee = strategy.calculateFee(hours);

        // Find spot and free it
        for (ParkingFloor floor : floors) {
            Optional<ParkingSpot> spotOpt = floor.getSpots().stream()
                    .filter(s -> !s.getIsFree() && s.getParkedVehicle().equals(vehicle))
                    .findFirst();
            if (spotOpt.isPresent()) {
                spotOpt.get().vacate();
                break;
            }
        }
        currentCount--;
        notifyObservers("Slot freed. Available slots: " + (capacity - currentCount));
        return fee;
    }
}
```

```
public class ParkingLotDisplay implements ParkingLotObserver {
    @Override
    public void update(String message) {
        System.out.println("Parking Lot Update: " + message);
```

```
    }
}
```

```
import java.util.ArrayList;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<ParkingFloor> floors = new ArrayList<>();
        ParkingLot parkingLot = ParkingLot.getInstance(floors, 100);

        ParkingLotDisplay display = new ParkingLotDisplay();
        parkingLot.addObserver(display);

        ParkingCommandInvoker invoker = new ParkingCommandInvoker();

        Vehicle car1 = new Car("ABC123");
        Vehicle bike1 = new Bike("XYZ789");

        invoker.addCommand(new ParkCommand(parkingLot, car1));
        invoker.addCommand(new ParkCommand(parkingLot, bike1));
        invoker.executeCommands();

        invoker.addCommand(new UnparkCommand(parkingLot, car1));
        invoker.executeCommands();
    }
}
```

```
import java.util.LinkedList;
import java.util.Queue;

public class ParkingCommandInvoker {
    private Queue<Command> commandQueue = new LinkedList<>();

    public void addCommand(Command command) {
        commandQueue.offer(command);
    }

    public void executeCommands() {
```

```
        while (!commandQueue.isEmpty()) {
            Command cmd = commandQueue.poll();
            cmd.execute();
        }
    }
}
```

15. FeeStrategy
16. CarFeeStrategy
17. BikeFeeStrategy
18. TruckFeeStrategy
19. FeeStrategyFactory

```
public interface FeeStrategy {
    double calculateFee(long durationInHours);
}
```

```
public class CarFeeStrategy implements FeeStrategy {
    public double calculateFee(long hours) {
        return 20 * hours;
    }
}
```

```
public class BikeFeeStrategy implements FeeStrategy {
    public double calculateFee(long hours) {
        return 10 * hours;
    }
}
```

```
public class TruckFeeStrategy implements FeeStrategy {
    public double calculateFee(long hours) {
        return 10 * hours;
```

```
    }
}
```

```
public class FeeStrategyFactory {
    public static FeeStrategy getStrategy(VehicleType type) {
        return switch (type) {
            case CAR -> new CarFeeStrategy();
            case BIKE -> new BikeFeeStrategy();
            case TRUCK -> new TruckFeeStrategy();
        };
    }
}
```

20. Ticket

```
public class Ticket {
    private final Vehicle vehicle;
    private final long entryTime;

    public Ticket(Vehicle vehicle) {
        this.vehicle = vehicle;
        this.entryTime = System.currentTimeMillis();
    }

    public long getDurationHours() {
        return (System.currentTimeMillis() - entryTime) / (1000 * 60 * 60);
    }
}
```

21. ParkingSpot
22. ParkingFloor

```
public class ParkingSpot {
    private final String id;
    private final VehicleType supportedType;
```

```
    private boolean isFree = true;
    private Vehicle parkedVehicle;

    public boolean canFit(Vehicle v) {
        return isFree && v.getType() == supportedType;
    }

    public void park(Vehicle v) {
        this.parkedVehicle = v;
        this.isFree = false;
    }

    public void vacate() {
        this.parkedVehicle = null;
        this.isFree = true;
    }
}
```

```
public class ParkingFloor {
    private final List<ParkingSpot> spots;

    public Optional<ParkingSpot> findAvailable(Vehicle v) {
        return spots.stream().filter(s -> s.canFit(v)).findFirst();
    }
}
```

23. VehicleType

```
public enum VehicleType {
    CAR,
    BIKE,
    TRUCK
}
```

24. Driver Code

```
public class ParkingLotDriver {
    public static void main(String[] args) {
        // Create parking lot
        ParkingLot lot = new ParkingLot(Arrays.asList(new ParkingFloor(Arrays.asList(new ParkingSp

        lot.addObserver(new ParkingLotObserverImpl("AdminConsole"));

        // Create vehicles
        Vehicle car1 = VehicleFactory.createVehicle(VehicleType.CAR, "KA-01-HH-1234");
        Vehicle car2 =  VehicleFactory.createVehicle(VehicleType.CAR, "KA-01-HH-9999");
        Vehicle car3 = VehicleFactory.createVehicle(VehicleType.CAR, "KA-01-BB-0001");

        // Park vehicles
        Ticket ticket1 = lot.park(car1);
        Ticket ticket2 = lot.park(car2);
        Ticket ticket3 = lot.park(car3); // Will notify full
        lot.unpark(car1);
        Ticket ticket4 = lot.park(car3); // Should work
    }
}
```

```
public class ParkingLotObserverImpl implements ParkingLotObserver {
    private final String name;

    public ParkingLotObserverImpl(String name) {
        this.name = name;
    }

    @Override
    public void update(String message) {
        System.out.println("🔔 [" + name + "] Notification: " + message);
    }
}
```

```
Parked
🔔 [AdminConsole] Notification: Parking Lot Full
Parked
🔔 [AdminConsole] Notification: Parking Lot Full
Parking Lot Full
No spot available for vehicle type
🔔 [AdminConsole] Notification: Slot freed. Available slots: 1
```

```
🔔 [AdminConsole] Notification: Parking Lot Full
Parked
```

---

25. Design Splitwise

✅ 1. Functional Requirements

- Add users
- Create expense (equal, exact, percentage)
- Split expenses among users
- Show balances for a user or all
- Settle expenses

✅ 2. Non-Functional Requirements

- Scalability: support many users/groups
- Extensibility: easily add new split types
- Thread-safety (multi-threaded updates)
- Maintainability and separation of concerns

✅ 3. Design Patterns Used

Pattern	Purpose
Strategy	For handling different types of splits (equal, exact, percentage)
Observer	Notify users when their balances change
Singleton	For balance manager to maintain consistency
Factory	For creating expense objects based on type

Class Diagram

Sequence Diagram

✅ 5. Sequence Diagram (Add Expense - Equal Split)

sql Copy Edit

```
User
 |
 |---> addExpense(amount, paidBy, List<users>, type)
 |
 |---> ExpenseManager.addExpense()
        |
        |---> SplitStrategy.validateSplit()
        |
        |---> Expense created and stored
        |
        |---> Balances updated
```

Source Code

👨‍💻 User.java

```
public class User {
    private final String id;
    private final String name;
    private final String email;

    public User(String id, String name, String email) {
        this.id = id;
        this.name = name;
        this.email = email;
    }
}
```

🧾 Split.java

```
public class Split {
    private final User user;
    private final double amount;
```

```
    public Split(User user, double amount) {
        this.user = user;
        this.amount = amount;
    }

    public User getUser() { return user; }
    public double getAmount() { return amount; }
}
```

💳 Expense.java

```
import java.util.List;

public class Expense {
    private final double amount;
    private final User paidBy;
    private final List<Split> splits;
    private final String description;

    public Expense(double amount, User paidBy, List<Split> splits,
        String description) {
        this.amount = amount;
        this.paidBy = paidBy;
        this.splits = splits;
        this.description = description;
    }

    public double getAmount() { return amount; }
    public User getPaidBy() { return paidBy; }
    public List<Split> getSplits() { return splits; }
}
```

🧠 SplitStrategy.java

```
import java.util.List;

public interface SplitStrategy {
    boolean validate(double amount, List<Split> splits);
}
```

🎯 EqualSplitStrategy.java

```
import java.util.List;

public class EqualSplitStrategy implements SplitStrategy {
    public boolean validate(double amount, List<Split> splits) {
        double splitAmount = amount / splits.size();
        for (Split split : splits) {
            if (Math.abs(split.getAmount() - splitAmount) > 0.01) return false;
        }
        return true;
    }
}
```

 ExactSplitStrategy.java

```
import java.util.List;

public class ExactSplitStrategy implements SplitStrategy {
    public boolean validate(double amount, List<Split> splits) {
        double total = 0;
        for (Split s : splits) total += s.getAmount();
        return Math.abs(total - amount) < 0.01;
    }
}
```

 PercentSplitStrategy.java

```
import java.util.List;

public class PercentSplitStrategy implements SplitStrategy {
    public boolean validate(double amount, List<Split> splits) {
        double total = 0;
        for (Split s : splits) total += s.getAmount();
        return Math.abs(total - amount) < 0.01;
    }
}
```

 ExpenseManager.java

```
import java.util.*;
```

```
public class ExpenseManager {
    private static ExpenseManager instance;
    private final Map<String, User> users = new HashMap<>();
    private final Map<String, Map<String, Double>> balances = new HashMap<>();

    private ExpenseManager() {}

    public static ExpenseManager getInstance() {
        if (instance == null) instance = new ExpenseManager();
        return instance;
    }

    public void addUser(User user) {
        users.put(user.getId(), user);
        balances.put(user.getId(), new HashMap<>());
    }

    public void addExpense(Expense expense) {
        User paidBy = expense.getPaidBy();
        double amount = expense.getAmount();

        for (Split split : expense.getSplits()) {
            String paidTo = split.getUser().getId();
            if (paidTo.equals(paidBy.getId())) continue;

            double splitAmt = split.getAmount();
            balances.get(paidTo).put(paidBy.getId(),
                balances.get(paidTo).getOrDefault(paidBy.getId(), 0.0)
            + splitAmt);
            balances.get(paidBy.getId()).put(paidTo,
                balances.get(paidBy.getId()).getOrDefault(paidTo, 0.0)
            - splitAmt);
        }
    }

    public void showBalance(String userId) {
        Map<String, Double> userBalance = balances.get(userId);
        for (Map.Entry<String, Double> entry : userBalance.entrySet()) {
            if (entry.getValue() < 0) {
                System.out.println(users.get(userId).getName() + " owes "
      + users.get(entry.getKey()).getName() + ": " + Math.abs(entry.getValue()));
            }
        }
```

```
    }

    public void showBalances() {
        for (String userId : balances.keySet()) {
            showBalance(userId);
        }
    }
}
```

✅ 7. Sample Driver Code

```
public class Main {
    public static void main(String[] args) {
        ExpenseManager manager = ExpenseManager.getInstance();

        User u1 = new User("u1", "Alice", "a@example.com");
        User u2 = new User("u2", "Bob", "b@example.com");
        User u3 = new User("u3", "Charlie", "c@example.com");

        manager.addUser(u1);
        manager.addUser(u2);
        manager.addUser(u3);

        List<Split> splits = List.of(
            new Split(u1, 100),
            new Split(u2, 100),
            new Split(u3, 100)
        );

        Expense expense = new Expense(300, u1, splits, "Dinner");
        manager.addExpense(expense);
        manager.showBalances();
    }
}
```

✅ Output

```
Bob owes Alice: 100.0
Charlie owes Alice: 100.0
```

```
// --- Observer Pattern Interfaces ---
interface ExpenseObserver {
    void notifyExpenseAdded(Expense expense);
}

class NotificationService implements ExpenseObserver {
    @Override
    public void notifyExpenseAdded(Expense expense) {
        System.out.println("[NOTIFY] Expense added: " + expense);
    }
}

// --- Enum for Split Strategies ---
enum SplitStrategyType {
    EQUAL,
    EXACT,
    PERCENT
}

// --- Factory Pattern ---
class SplitFactory {
    public static List<Split> createSplits(SplitStrategyType type, List<User> users, List<Double> val
        List<Split> splits = new ArrayList<>();
        switch (type) {
            case EQUAL:
                for (User user : users) {
                    splits.add(new EqualSplit(user));
                }
                break;
            case EXACT:
                for (int i = 0; i < users.size(); i++) {
                    splits.add(new ExactSplit(users.get(i), values.get(i)));
                }
                break;
            case PERCENT:
                for (int i = 0; i < users.size(); i++) {
                    splits.add(new PercentSplit(users.get(i), values.get(i)));
                }
                break;
            default:
                throw new IllegalArgumentException("Unknown split strategy type: " + type);
        }
        return splits;
```

```
    }
}

// --- Expense Manager with Observer Integration ---
class ExpenseManager {
    private Map<String, User> userMap = new HashMap<>();
    private Map<String, Map<String, Double>> balanceSheet = new HashMap<>();
    private List<ExpenseObserver> observers = new ArrayList<>();

    public void addObserver(ExpenseObserver observer) {
        observers.add(observer);
    }

    public void removeObserver(ExpenseObserver observer) {
        observers.remove(observer);
    }

    private void notifyObservers(Expense expense) {
        for (ExpenseObserver observer : observers) {
            observer.notifyExpenseAdded(expense);
        }
    }

    public void addUser(User user) {
        userMap.put(user.getId(), user);
        balanceSheet.put(user.getId(), new HashMap<>());
    }

    public void addExpense(ExpenseType type, double amount, String paidBy, List<Split> splits) {
        Expense expense = ExpenseService.createExpense(type, amount, userMap.get(paidBy), split
        if (expense == null) throw new IllegalArgumentException("Invalid Expense");

        notifyObservers(expense);

        for (Split split : splits) {
            String paidTo = split.getUser().getId();
            if (!paidBy.equals(paidTo)) {
                Map<String, Double> balances = balanceSheet.get(paidBy);
                balances.put(paidTo, balances.getOrDefault(paidTo, 0.0) + split.getAmount());
                Map<String, Double> balancesTo = balanceSheet.get(paidTo);
                balancesTo.put(paidBy, balancesTo.getOrDefault(paidBy, 0.0) - split.getAmount());
            }
        }
```

```
    }

    public void showBalances() {
        for (Map.Entry<String, Map<String, Double>> userBalance : balanceSheet.entrySet()) {
            String user1 = userBalance.getKey();
            for (Map.Entry<String, Double> balances : userBalance.getValue().entrySet()) {
                if (balances.getValue() > 0) {
                    System.out.println(userMap.get(balances.getKey()).getName() + " owes " + userMap.g
                }
            }
        }
    }
}

// Note: All other classes (User, Expense, Split types, etc.) remain unchanged
// Ensure to plug in the above ExpenseManager in your driver class and use SplitFactory for creatir

// --- Driver Code Example ---
public class SplitwiseApp {
    public static void main(String[] args) {
        ExpenseManager manager = new ExpenseManager();
        manager.addObserver(new NotificationService());
        User u1 = new User("u1", "Sai");
        User u2 = new User("u2", "Ashish");
        User u3 = new User("u3", "Alex");
        manager.addUser(u1);
        manager.addUser(u2);
        manager.addUser(u3);
        List<User> users = Arrays.asList(u1, u2, u3);
        List<Double> exactValues = Arrays.asList(100.0, 200.0, 100.0);
        List<Split> splits = SplitFactory.createSplits(SplitStrategyType.EXACT, users, exactValues);
        manager.addExpense(ExpenseType.EXACT, 400, "u1", splits);
        manager.showBalances();
    }
}
```

---

16. Design Chess

⚙️ Class Diagram (High-Level)

```
Game
 ├── Player whitePlayer
 ├── Player blackPlayer
 ├── Board board
 ├── Player currentTurn
 └── playMove(Cell from, Cell to)

Board
 ├── Cell[][] cells
 └── initialize()

Cell
 ├── int row, col
 └── Piece piece

Piece (abstract)
 ├── boolean isWhite
 └── abstract List<Cell> getValidMoves(Board board, Cell from)

(Subclasses): King, Queen, Bishop, Knight, Rook, Pawn

Move
 ├── Cell from
 ├── Cell to
 ├── Piece movedPiece
 └── Piece capturedPiece
```

✅ Key Code Skeleton

```
enum PieceType { KING, QUEEN, BISHOP, KNIGHT, ROOK, PAWN }

abstract class Piece {
    boolean isWhite;
    PieceType type;

    public Piece(boolean isWhite, PieceType type) {
        this.isWhite = isWhite;
        this.type = type;
    }

    public abstract boolean isValidMove(Board board, Cell from, Cell to);
    public boolean isWhite() { return isWhite; }
```

```
    public PieceType getType() { return type; }
}

class King extends Piece {
    public King(boolean isWhite) { super(isWhite, PieceType.KING); }

    @Override
    public boolean isValidMove(Board board, Cell from, Cell to) {
        int dx = Math.abs(from.getRow() - to.getRow());
        int dy = Math.abs(from.getCol() - to.getCol());
        return dx <= 1 && dy <= 1;
    }
}
```

🧩 Board, Cell, and Game

```
class Cell {
    private int row, col;
    private Piece piece;

    public Cell(int row, int col) {
        this.row = row;
        this.col = col;
    }

    public void setPiece(Piece piece) { this.piece = piece; }
    public Piece getPiece() { return piece; }
    public int getRow() { return row; }
    public int getCol() { return col; }
}

class Board {
    private final int SIZE = 8;
    private Cell[][] cells = new Cell[SIZE][SIZE];

    public Board() {
        for (int i = 0; i < SIZE; i++)
            for (int j = 0; j < SIZE; j++)
                cells[i][j] = new Cell(i, j);
        initialize();
    }
```

```
    private void initialize() {
        cells[0][4].setPiece(new King(false)); // Black King
        cells[7][4].setPiece(new King(true));  // White King
        // Add rest of the pieces...
    }

    public Cell getCell(int row, int col) { return cells[row][col]; }
}
```

🎮 Game Logic

```
class Player {
    private boolean isWhite;

    public Player(boolean isWhite) { this.isWhite = isWhite; }
    public boolean isWhite() { return isWhite; }
}

class Game {
    private Player whitePlayer = new Player(true);
    private Player blackPlayer = new Player(false);
    private Board board = new Board();
    private Player currentTurn = whitePlayer;

    public boolean playMove(int fromRow, int fromCol, int toRow, int toCol) {
        Cell from = board.getCell(fromRow, fromCol);
        Cell to = board.getCell(toRow, toCol);
        Piece piece = from.getPiece();

        if (piece == null || piece.isWhite() != currentTurn.isWhite()) return false;

        if (!piece.isValidMove(board, from, to)) return false;

        to.setPiece(piece);
        from.setPiece(null);

        currentTurn = (currentTurn == whitePlayer) ? blackPlayer : whitePlayer;
        return true;
    }
}
```

📝 Additional Considerations

- Castling, En Passant, Promotion logic
- Move history
- Check, Checkmate, Stalemate detection
- UI layer (CLI or GUI)
- Timer support (blitz/chess clock)
- Undo/Redo support
- Multiplayer or AI integration

```
// --- Enums ---

enum PieceType {
    KING, QUEEN, BISHOP, KNIGHT, ROOK, PAWN
}

// --- Core Classes ---

abstract class Piece {
    protected boolean isWhite;
    protected PieceType type;

    public Piece(boolean isWhite, PieceType type) {
        this.isWhite = isWhite;
        this.type = type;
    }

    public boolean isWhite() {
        return isWhite;
    }

    public PieceType getType() {
        return type;
    }

    public abstract boolean isValidMove(Board board, Cell from, Cell to);
}

class King extends Piece {
    public King(boolean isWhite) {
        super(isWhite, PieceType.KING);
```

```
    }

    @Override
    public boolean isValidMove(Board board, Cell from, Cell to) {
        int dx = Math.abs(from.getRow() - to.getRow());
        int dy = Math.abs(from.getCol() - to.getCol());
        return dx <= 1 && dy <= 1;
    }
}

class Rook extends Piece {
    public Rook(boolean isWhite) {
        super(isWhite, PieceType.ROOK);
    }

    @Override
    public boolean isValidMove(Board board, Cell from, Cell to) {
        return from.getRow() == to.getRow() || from.getCol() == to.getCol();
    }
}

class Cell {
    private int row, col;
    private Piece piece;

    public Cell(int row, int col) {
        this.row = row;
        this.col = col;
    }

    public int getRow() {
        return row;
    }

    public int getCol() {
        return col;
    }

    public Piece getPiece() {
        return piece;
    }

    public void setPiece(Piece piece) {
```

```
        this.piece = piece;
    }
}

class Board {
    private final int SIZE = 8;
    private Cell[][] cells = new Cell[SIZE][SIZE];

    public Board() {
        for (int i = 0; i < SIZE; i++) {
            for (int j = 0; j < SIZE; j++) {
                cells[i][j] = new Cell(i, j);
            }
        }
        initialize();
    }

    public void initialize() {
        cells[0][0].setPiece(new Rook(false));
        cells[0][4].setPiece(new King(false));

        cells[7][0].setPiece(new Rook(true));
        cells[7][4].setPiece(new King(true));
        // Add other pieces as needed
    }

    public Cell getCell(int row, int col) {
        return cells[row][col];
    }
}

class Player {
    private boolean isWhite;

    public Player(boolean isWhite) {
        this.isWhite = isWhite;
    }

    public boolean isWhite() {
        return isWhite;
    }
}
```

```
class Game {
    private Board board;
    private Player whitePlayer;
    private Player blackPlayer;
    private Player currentTurn;

    public Game() {
        board = new Board();
        whitePlayer = new Player(true);
        blackPlayer = new Player(false);
        currentTurn = whitePlayer;
    }

    public boolean playMove(int fromRow, int fromCol, int toRow, int toCol) {
        Cell from = board.getCell(fromRow, fromCol);
        Cell to = board.getCell(toRow, toCol);
        Piece piece = from.getPiece();

        if (piece == null || piece.isWhite() != currentTurn.isWhite()) {
            return false;
        }

        if (!piece.isValidMove(board, from, to)) {
            return false;
        }

        to.setPiece(piece);
        from.setPiece(null);
        currentTurn = (currentTurn == whitePlayer) ? blackPlayer : whitePlayer;
        return true;
    }

    public void printBoard() {
        for (int i = 0; i < 8; i++) {
            for (int j = 0; j < 8; j++) {
                Piece piece = board.getCell(i, j).getPiece();
                if (piece == null) {
                    System.out.print("-- ");
                } else {
                    String name = piece.getType().toString().substring(0, 1);
                    name = piece.isWhite() ? name.toUpperCase() : name.toLowerCase();
                    System.out.print(name + "  ");
                }
```

```
            }
            System.out.println();
        }
    }
}

public class ChessApp {
    public static void main(String[] args) {
        Game game = new Game();
        game.printBoard();

        System.out.println("\nMove White Rook from (7,0) to (5,0): " +
                game.playMove(7, 0, 5, 0));
        game.printBoard();

        System.out.println("\nMove Black Rook from (0,0) to (3,0): " +
                game.playMove(0, 0, 3, 0));
        game.printBoard();
    }
}
```

17. Design Chess Validator

Designing a **Chess Validator** means building a system that can validate if a move in a chess game is legal or not, based on:

- The piece being moved.
- The current state of the board.
- Turn-based rules.
- Special rules like castling, en passant, promotion.
- Check, checkmate, and stalemate detection.

✅ Core Components

1. Enums

```
enum PieceType {
    KING, QUEEN, ROOK, BISHOP, KNIGHT, PAWN
}
```

```
enum Color {
    WHITE, BLACK
}
```

2. Position

```
class Position {
    int row, col;

    public Position(int row, int col) {
        this.row = row;
        this.col = col;
    }

    public boolean isValid() {
        return row >= 0 && row < 8 && col >= 0 && col < 8;
    }

    public boolean equals(Position other) {
        return this.row == other.row && this.col == other.col;
    }
}
```

3. Piece (Abstract Base Class)

```
abstract class Piece {
    Color color;
    PieceType type;

    public Piece(Color color, PieceType type) {
        this.color = color;
        this.type = type;
    }

    public Color getColor() {
        return color;
    }

    public PieceType getType() {
        return type;
    }
```

```
    public abstract boolean isValidMove(Position from, Position to, Board board);
}
```

4. Concrete Piece Implementations (example: Rook & King)

```
class Rook extends Piece {
    public Rook(Color color) {
        super(color, PieceType.ROOK);
    }

    @Override
    public boolean isValidMove(Position from, Position to, Board board) {
        if (from.row != to.row && from.col != to.col) return false;

        int rowStep = Integer.compare(to.row, from.row);
        int colStep = Integer.compare(to.col, from.col);

        int currentRow = from.row + rowStep;
        int currentCol = from.col + colStep;

        while (currentRow != to.row || currentCol != to.col) {
            if (board.getPiece(new Position(currentRow, currentCol)) != null) {
                return false; // Path blocked
            }
            currentRow += rowStep;
            currentCol += colStep;
        }

        Piece destPiece = board.getPiece(to);
        return destPiece == null || destPiece.getColor() != this.getColor();
    }
}

class King extends Piece {
    public King(Color color) {
        super(color, PieceType.KING);
    }

    @Override
    public boolean isValidMove(Position from, Position to, Board board) {
        int dr = Math.abs(from.row - to.row);
```

```
        int dc = Math.abs(from.col - to.col);
        if (dr > 1 || dc > 1) return false;

        Piece destPiece = board.getPiece(to);
        return destPiece == null || destPiece.getColor() != this.getColor();
    }
}
```

5. Board

```
class Board {
    Piece[][] grid;

    public Board() {
        grid = new Piece[8][8];
        setup();
    }

    private void setup() {
        grid[0][0] = new Rook(Color.WHITE);
        grid[0][7] = new Rook(Color.WHITE);
        grid[7][0] = new Rook(Color.BLACK);
        grid[7][7] = new Rook(Color.BLACK);

        grid[0][4] = new King(Color.WHITE);
        grid[7][4] = new King(Color.BLACK);
    }

    public Piece getPiece(Position pos) {
        if (!pos.isValid()) return null;
        return grid[pos.row][pos.col];
    }

    public void movePiece(Position from, Position to) {
        Piece piece = getPiece(from);
        grid[to.row][to.col] = piece;
        grid[from.row][from.col] = null;
    }

    public void printBoard() {
        for (int r = 0; r < 8; r++) {
            for (int c = 0; c < 8; c++) {
```

```
            Piece p = grid[r][c];
            if (p == null) System.out.print(".");
            else System.out.print(p.getColor() == Color.WHITE ?
            p.getType().name().charAt(0) :
            Character.toLowerCase(p.getType().name().charAt(0)));
            System.out.print(" ");
        }
        System.out.println();
    }
  }
}
```

6. Validator

```
class MoveValidator {
  public static boolean validateMove(Board board, Position from, Position to, Color turn) {
    if (!from.isValid() || !to.isValid()) return false;

    Piece piece = board.getPiece(from);
    if (piece == null || piece.getColor() != turn) return false;

    return piece.isValidMove(from, to, board);
  }
}
```

7. Game Controller

```
class Game {
  Board board = new Board();
  Color turn = Color.WHITE;

  public void makeMove(Position from, Position to) {
    if (MoveValidator.validateMove(board, from, to, turn)) {
      board.movePiece(from, to);
      turn = (turn == Color.WHITE) ? Color.BLACK : Color.WHITE;
      board.printBoard();
    } else {
      System.out.println("Invalid move");
    }
  }
}
```

8. Driver Code

```
public class ChessValidatorApp {
    public static void main(String[] args) {
        Game game = new Game();
        game.board.printBoard();

        System.out.println("White moves rook:");
        game.makeMove(new Position(0, 0), new Position(0, 5)); // Valid

        System.out.println("Black tries illegal move:");
        game.makeMove(new Position(7, 0), new Position(6, 1)); // Invalid diagonal

        System.out.println("Black moves rook:");
        game.makeMove(new Position(7, 0), new Position(5, 0)); // Valid
    }
}
```

✅ Output Example

```
R . . . K . . R
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
r . . . k . . r
White moves rook:
. . . . K R . R
...
Black tries illegal move:
Invalid move
...
```

18. Design Distributed Queue | Kafka

Designing a **Distributed Queue like Kafka** involves addressing scalability, durability, fault tolerance, and high throughput. Here's a complete **system design overview** followed by a **code-level prototype** (simplified Kafka-like system).

📌 High-Level Requirements

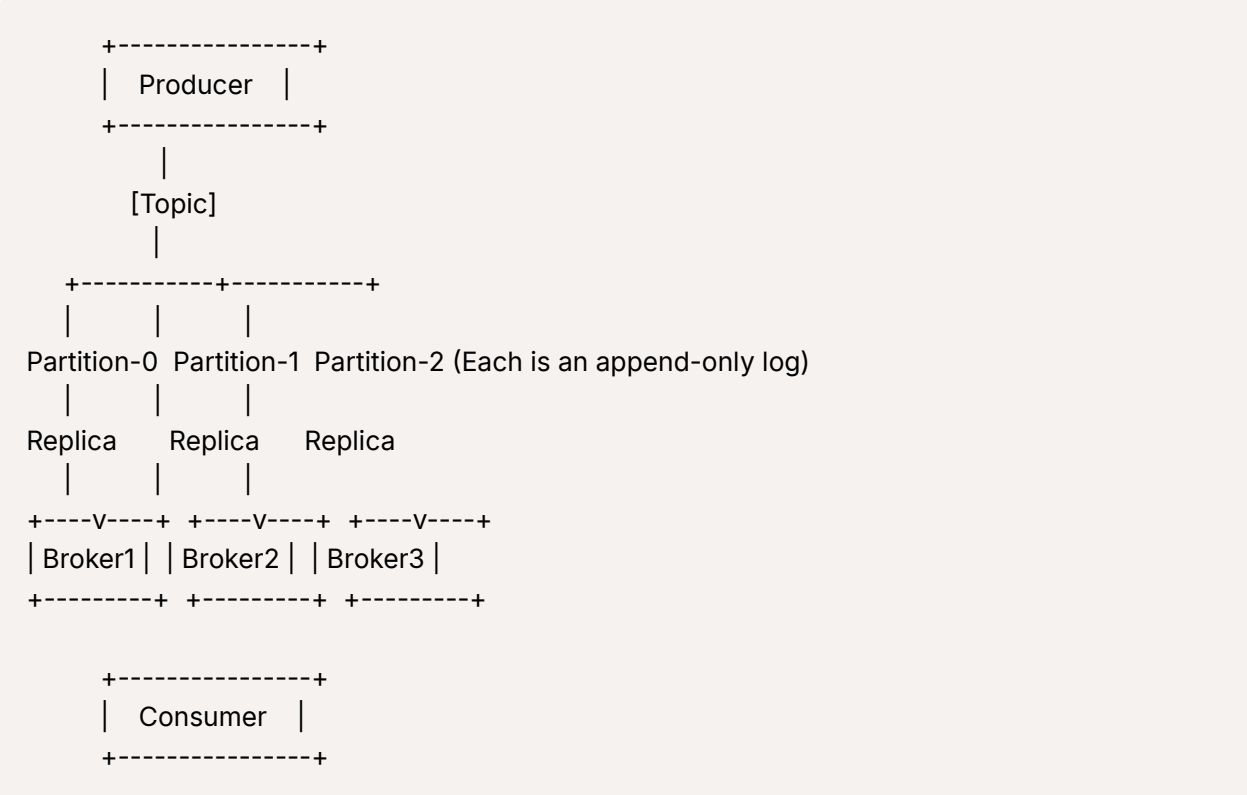
Functional:

- **Producers** can publish messages to topics.
- **Consumers** subscribe to topics and read messages in order.
- **Durable**: messages persist even if a consumer is down.
- **Scalable**: handles high throughput with many consumers/producers.
- **At-least-once delivery**.

Non-Functional:

- **Horizontal scalability** (partitioning).
- **High availability** (replication).
- **Fault-tolerant**.
- **Low latency and high throughput**.

📦 Kafka-Like Architecture Components



✅ Key Concepts

1. Topic

- Logical grouping of messages.
- Split into **partitions** for scalability.

2. Partition

- Each partition is an **append-only log**.
- Each message has a unique **offset**.
- Reads are sequential per partition (strong ordering).

3. Broker

- Stores partitions.
- Manages persistence and replication.

4. Producer

- Publishes messages to topics.
- Uses a **partitioning strategy** (e.g., round robin, hash of key).

5. Consumer

- Reads from one or more partitions.
- Maintains its **offset**.

6. Zookeeper (or Controller)

- Maintains metadata: brokers, topics, partition leaders.

✅ 1. Message.java

```
public class Message {
    private final int offset;
    private final String value;

    public Message(int offset, String value) {
        this.offset = offset;
        this.value = value;
    }

    public int getOffset() { return offset; }
    public String getValue() { return value; }
}
```

✅ 2. Partition.java

```
import java.util.*;
import java.util.concurrent.atomic.AtomicInteger;

public class Partition {
    private final List<Message> messages = new ArrayList<>();
    private final AtomicInteger offset = new AtomicInteger(0);

    public synchronized void append(String value) {
        messages.add(new Message(offset.getAndIncrement(), value));
    }

    public synchronized List<Message> readFromOffset(int startOffset) {
        List<Message> result = new ArrayList<>();
        for (int i = startOffset; i < messages.size(); i++) {
            result.add(messages.get(i));
        }
        return result;
    }
}
```

✅ 3. Topic.java

```
import java.util.*;

public class Topic {
    private final String name;
    private final List<Partition> partitions;

    public Topic(String name, int numPartitions) {
        this.name = name;
        this.partitions = new ArrayList<>();
        for (int i = 0; i < numPartitions; i++) {
            partitions.add(new Partition());
        }
    }

    public void publish(String message, int partitionId) {
        partitions.get(partitionId).append(message);
    }
```

```
    public Partition getPartition(int partitionId) {
        return partitions.get(partitionId);
    }

    public int getNumPartitions() {
        return partitions.size();
    }
}
```

✅ 4. Broker.java

```
import java.util.*;

public class Broker {
    private final Map<String, Topic> topics = new HashMap<>();

    public void createTopic(String topicName, int numPartitions) {
        topics.put(topicName, new Topic(topicName, numPartitions));
    }

    public Topic getTopic(String topicName) {
        return topics.get(topicName);
    }
}
```

✅ 5. Producer.java

```
public class Producer {
    private final Broker broker;

    public Producer(Broker broker) {
        this.broker = broker;
    }

    public void send(String topicName, String message) {
        Topic topic = broker.getTopic(topicName);
        int partitionId = Math.abs(message.hashCode())
        % topic.getNumPartitions(); // hash partitioning
        topic.publish(message, partitionId);
    }
}
```

✅ 6. Consumer.java

```
import java.util.*;

public class Consumer {
    private final Broker broker;
    private final String topicName;
    private final int partitionId;
    private int currentOffset = 0;

    public Consumer(Broker broker, String topicName, int partitionId) {
        this.broker = broker;
        this.topicName = topicName;
        this.partitionId = partitionId;
    }

    public void poll() {
        Partition partition =
        broker.getTopic(topicName).getPartition(partitionId);
        List<Message> messages = partition.readFromOffset(currentOffset);
        for (Message message : messages) {
            System.out.println("Consumed: " + message.getValue());
            currentOffset = message.getOffset() + 1;
        }
    }
}
```

✅ 7. Main.java

```
public class Main {
    public static void main(String[] args) {
        Broker broker = new Broker();
        broker.createTopic("logs", 3);

        Producer producer = new Producer(broker);
        producer.send("logs", "Event 1");
        producer.send("logs", "Event 2");
        producer.send("logs", "Event 3");

        Consumer consumer0 = new Consumer(broker, "logs", 0);
        Consumer consumer1 = new Consumer(broker, "logs", 1);
        Consumer consumer2 = new Consumer(broker, "logs", 2);
```

```
        System.out.println("Polling from partition 0");
        consumer0.poll();

        System.out.println("Polling from partition 1");
        consumer1.poll();

        System.out.println("Polling from partition 2");
        consumer2.poll();
    }
}
```

✅ Output

```
Polling from partition 0
Consumed: Event 3
Polling from partition 1
Consumed: Event 1
Polling from partition 2
Consumed: Event 2
```

19. `Design Restaurant Management System: ClearFood`

- `Singleton Pattern` : Ensures a single instance of the `RestaurantService` class, centralizing user sessions and restaurant data management.
- `Strategy Pattern` : Facilitates dynamic sorting of restaurants based on different criteria (e.g., price, rating) by encapsulating sorting algorithms.
- `Factory Pattern` : Streamlines the creation of `User`, `Restaurant`, and `Order` objects, promoting scalability and maintainability.
- `Observer Pattern` : Allows the system to notify users about restaurant updates, such as new menu items or promotions.

```
import java.util.*;

class User {
    String name, gender, phoneNumber, location;
    List<Order> orders = new ArrayList<>();
```

```
    User(String name, String gender, String phoneNumber, String location) {
        this.name = name;
        this.gender = gender;
        this.phoneNumber = phoneNumber;
        this.location = location;
    }
}

class FoodItem {
    String name;
    int price;
    int quantity;

    FoodItem(String name, int price, int quantity) {
        this.name = name;
        this.price = price;
        this.quantity = quantity;
    }
}

class Review {
    int rating;
    String comment;

    Review(int rating, String comment) {
        this.rating = rating;
        this.comment = comment;
    }
}

class Restaurant {
    String name;
    Set<String> pincodes;
    FoodItem foodItem;
    List<Review> reviews = new ArrayList<>();

    Restaurant(String name, Set<String> pincodes, FoodItem foodItem) {
        this.name = name;
        this.pincodes = pincodes;
        this.foodItem = foodItem;
    }

    double getAverageRating() {
```

```
        return reviews.stream().mapToInt(r -> r.rating).average().orElse(0);
    }
}

class Order {
    String restaurantName;
    String foodName;
    int quantity;
    int totalCost;

    Order(String restaurantName, String foodName, int quantity, int totalCost) {
        this.restaurantName = restaurantName;
        this.foodName = foodName;
        this.quantity = quantity;
        this.totalCost = totalCost;
    }
}

class RestaurantService {
    Map<String, User> users = new HashMap<>();
    Map<String, Restaurant> restaurants = new HashMap<>();
    User currentUser = null;

    void registerUser(String name, String gender, String phone, String location) {
        users.put(phone, new User(name, gender, phone, location));
    }

    void loginUser(String phone) {
        if (users.containsKey(phone)) {
            currentUser = users.get(phone);
        }
    }

    void registerRestaurant(String name, String pincodeStr, String foodName, int price, int qty) {
        Set<String> pincodes = new HashSet<>(Arrays.asList(pincodeStr.split("/")));
        FoodItem item = new FoodItem(foodName, price, qty);
        restaurants.put(name, new Restaurant(name, pincodes, item));
    }

    void updateQuantity(String restName, int qtyToAdd) {
        Restaurant r = restaurants.get(restName);
        if (r != null) r.foodItem.quantity += qtyToAdd;
    }
```

```
void updateLocation(String restName, String newPincodes) {
    Restaurant r = restaurants.get(restName);
    if (r != null) r.pincodes = new HashSet<>(Arrays.asList(newPincodes.split("/")));
}

void createReview(String restName, int rating, String comment) {
    Restaurant r = restaurants.get(restName);
    if (r != null) r.reviews.add(new Review(rating, comment));
}

void showRestaurants(String sortBy) {
    List<Restaurant> serviceable = new ArrayList<>();
    for (Restaurant r : restaurants.values()) {
        if (r.pincodes.contains(currentUser.location)) serviceable.add(r);
    }

    if (sortBy.equalsIgnoreCase("price")) {
        serviceable.sort(Comparator.comparingInt(r -> r.foodItem.price));
    } else if (sortBy.equalsIgnoreCase("rating")) {
        serviceable.sort((a, b) -> Double.compare(b.getAverageRating(), a.getAverageRating()));
    }

    for (Restaurant r : serviceable) {
        System.out.println(r.name + ", " + r.foodItem.name);
    }
}

void placeOrder(String restName, int qty) {
    Restaurant r = restaurants.get(restName);
    if (r == null || !r.pincodes.contains(currentUser.location)) {
        System.out.println("Cannot place order");
        return;
    }
    if (r.foodItem.quantity < qty) {
        System.out.println("Cannot place order");
        return;
    }
    r.foodItem.quantity -= qty;
    int total = qty * r.foodItem.price;
    Order order = new Order(r.name, r.foodItem.name, qty, total);
    currentUser.orders.add(order);
    System.out.println("Order Placed Successfully.");
```

```
    }

    void showOrderHistory() {
        for (Order o : currentUser.orders) {
            System.out.println(o.restaurantName + " " + o.foodName + " " + o.quantity + " " + o.totalCo
        }
    }
}

public class ClearFoodApp {
    public static void main(String[] args) {
        RestaurantService app = new RestaurantService();

        app.registerUser("Pralove", "M", "phoneNumber-1", "HSR");
        app.registerUser("Nitesh", "M", "phoneNumber-2", "BTM");
        app.registerUser("Vatsal", "M", "phoneNumber-3", "BTM");
        app.loginUser("phoneNumber-1");

        app.registerRestaurant("Food Court-1", "BTM/HSR", "NI Thali", 100, 5);
        app.registerRestaurant("Food Court-2", "BTM/pincode-2", "Burger", 120, 3);
        app.loginUser("phoneNumber-2");
        app.registerRestaurant("Food Court-3", "HSR", "SI Thali", 150, 1);
        app.loginUser("phoneNumber-1");

        app.showRestaurants("Price");
        app.placeOrder("Food Court-1", 2);
        app.placeOrder("Food Court-2", 7);

        app.createReview("Food Court-2", 3, "Good Food");
        app.createReview("Food Court-1", 5, "Nice Food");
        app.showRestaurants("rating");

        app.loginUser("phoneNumber-1");
        app.updateQuantity("Food Court-2", 5);
        app.updateLocation("Food Court-2", "BTM/HSR");
    }
}
```

1. Observer Interface

```
public interface Observer {
    void update(String message);
}
```

2. Concrete Observer (User)

```
public class User implements Observer {
    private String name;
    private String phoneNumber;
    private String location;

    public User(String name, String phoneNumber, String location) {
        this.name = name;
        this.phoneNumber = phoneNumber;
        this.location = location;
    }

    @Override
    public void update(String message) {
        System.out.println("Notification for " + name + ": " + message);
    }

    public String getLocation() {
        return location;
    }

    public String getPhoneNumber() {
        return phoneNumber;
    }

    public String getName() {
        return name;
    }
}
```

3. Subject Interface

```
public interface Subject {
    void addObserver(Observer observer);
    void removeObserver(Observer observer);
```

```
    void notifyObservers(String message);
}
```

4. Concrete Subject (Restaurant)

```
import java.util.*;

public class Restaurant implements Subject {
    private String name;
    private Set<String> pincodes;
    private String foodItem;
    private int price;
    private int quantity;
    private final List<Observer> observers = new ArrayList<>();

    public Restaurant(String name, Set<String> pincodes, String foodItem, int price, int quantity)
{
        this.name = name;
        this.pincodes = pincodes;
        this.foodItem = foodItem;
        this.price = price;
        this.quantity = quantity;
    }

    @Override
    public void addObserver(Observer observer) {
        observers.add(observer);
    }

    @Override
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers(String message) {
        for (Observer observer : observers) {
            observer.update(message);
        }
    }

    public void updateQuantity(int addQty) {
```

```
        this.quantity += addQty;
        notifyObservers("Quantity updated for " + name + ": " + quantity);
    }

    public void updateLocation(Set<String> newPincodes) {
        this.pincodes = newPincodes;
        notifyObservers("Location updated for " + name + ": " + newPincodes);
    }

    public String getName() {
        return name;
    }

    public int getQuantity() {
        return quantity;
    }

    public String getFoodItem() {
        return foodItem;
    }

    public int getPrice() {
        return price;
    }

    public Set<String> getPincodes() {
        return pincodes;
    }
}
```

5. Usage in Service Layer

```
public class RestaurantService {
    private final Map<String, Restaurant> restaurantMap = new HashMap<>();
    private final Map<String, User> userMap = new HashMap<>();

    public void registerUser(User user) {
        userMap.put(user.getPhoneNumber(), user);
    }

    public void registerRestaurant(Restaurant restaurant) {
        restaurantMap.put(restaurant.getName(), restaurant);
```

```
    }

    public void subscribeUserToRestaurant(String phoneNumber, String restaurantName) {
        User user = userMap.get(phoneNumber);
        Restaurant restaurant = restaurantMap.get(restaurantName);
        if (user != null && restaurant != null) {
            restaurant.addObserver(user);
        }
    }

    public void updateRestaurantQuantity(String restaurantName, int quantity) {
        Restaurant restaurant = restaurantMap.get(restaurantName);
        if (restaurant != null) {
            restaurant.updateQuantity(quantity);
        }
    }

    public void updateRestaurantLocation(String restaurantName, Set<String> pincodes) {
        Restaurant restaurant = restaurantMap.get(restaurantName);
        if (restaurant != null) {
            restaurant.updateLocation(pincodes);
        }
    }
}
```

✅ Example Flow

```
RestaurantService service = new RestaurantService();

User user1 = new User("Sai", "123456", "BTM");
User user2 = new User("Ashish", "987654", "HSR");

service.registerUser(user1);
service.registerUser(user2);

Set<String> pincodes = new HashSet<>(Arrays.asList("BTM", "HSR"));
Restaurant rest1 = new Restaurant("Food Court-1", pincodes, "NI Thali", 100, 5);

service.registerRestaurant(rest1);
service.subscribeUserToRestaurant("123456", "Food Court-1");
service.subscribeUserToRestaurant("987654", "Food Court-1");
```

```
service.updateRestaurantQuantity("Food Court-1", 10);
service.updateRestaurantLocation("Food Court-1", new HashSet<>(Arrays.asList("BTM", "Indira nagar")));
```

🧠 Output

```
Notification for Sai: Quantity updated for Food Court-1: 15
Notification for Ashish: Quantity updated for Food Court-1: 15
Notification for Sai: Location updated for Food Court-1: [BTM, Indiranagar]
Notification for Ashish: Location updated for Food Court-1: [BTM, Indiranagar]
```

20. Design BookMyShow

🎯 Functional Requirements

1. **User Operations**: Search for movies/events, view showtimes, select seats, and book tickets.
2. **Theater Management**: Manage theaters, screens, shows, and seating arrangements.
3. **Booking System**: Handle seat selection, booking, and cancellations with real-time updates.
4. **Payment Processing**: Integrate with payment gateways to handle transactions securely.
5. **Notifications**: Send booking confirmations and alerts to users.Wikipedia+1Wikipedia+1

🧱 High-Level Architecture

- **Frontend**: User interface for browsing and booking.
- **Backend Services**:
  - **User Service**: Handles user authentication and profiles.
  - **Movie Service**: Manages movie and event information.
  - **Theater Service**: Manages theaters, screens, and shows.
  - **Booking Service**: Handles seat selection and booking logic.
  - **Payment Service**: Processes payments and refunds.
  - **Notification Service**: Sends emails/SMS to users.
- **Database**: Stores user data, bookings, movie details, etc.
- **Cache**: For quick access to frequently used data like seat availability.MediumMediumGitHub+4Wikipedia+4Wikipedia+4

🧩 Design Patterns Applied

1. **Singleton Pattern**: Ensure a single instance of services like `PaymentGateway` or `NotificationService`.
2. **Factory Pattern**: Create different types of notifications (Email, SMS) without specifying the exact class.
3. **Observer Pattern**: Notify users about booking confirmations or changes in showtimes.
4. **Strategy Pattern**: Implement different pricing strategies or seat selection algorithms.
5. **Builder Pattern**: Construct complex objects like `Booking` with multiple optional parameters.

📘 Class Diagram Overview

- **User**
  - Attributes: userId, name, email, phoneNumber
  - Methods: searchMovies(), bookTicket()
- **Movie**
  - Attributes: movieId, title, genre, duration
- **Theater**
  - Attributes: theaterId, name, location
  - Methods: addScreen(), getShows()
- **Screen**
  - Attributes: screenId, name, seatingLayout
- **Show**
  - Attributes: showId, movie, screen, showTime
  - Methods: getAvailableSeats()
- **Seat**
  - Attributes: seatId, row, number, type, status
- **Booking**
  - Attributes: bookingId, user, show, seats, paymentStatus
  - Methods: confirmBooking(), cancelBooking()
- **Payment**
  - Attributes: paymentId, amount, paymentMethod, status
  - Methods: processPayment()
- **Notification**

- Attributes: notificationId, user, message
- Methods: send()GitHubYouTube+5GeeksforGeeks+5Astik Anand+5Wikipedia+4Wikipedia+4Medium+4Astik Anand+13Medium+13GitHub+13

🔄 Sequence Diagram: Booking a Ticket

1. **User** searches for a movie.
2. **System** displays available shows.
3. **User** selects a show and chooses seats.
4. **System** locks selected seats temporarily.
5. **User** proceeds to payment.
6. **Payment Service** processes the payment.
7. **Upon success**, booking is confirmed, and seats are marked as booked.
8. **Notification Service** sends confirmation to the user.Astik Anand

---

🛠️ Sample Code Snippet: Factory Pattern for Notifications

```
// Notification.java
public interface Notification {
    void send(String message, User user);
}

// EmailNotification.java
public class EmailNotification implements Notification {
    public void send(String message, User user) {
        // Logic to send email
    }
}

// SMSNotification.java
public class SMSNotification implements Notification {
    public void send(String message, User user) {
        // Logic to send SMS
    }
}

// NotificationFactory.java
public class NotificationFactory {
    public static Notification createNotification(String type) {
```

```
        if (type.equalsIgnoreCase("EMAIL")) {
            return new EmailNotification();
        } else if (type.equalsIgnoreCase("SMS")) {
            return new SMSNotification();
        }
        throw new IllegalArgumentException("Unknown notification type");
    }
}
```

🎬 Overview

The system allows users to:GeeksforGeeks

- Search for movies
- View available shows
- Select seats
- Book tickets
- Receive notifications upon successful bookingMediumAstik Anand+5Educative+5GeeksforGeeks+5

🧩 Design Patterns Utilized

- **Singleton Pattern**: Ensures a single instance of the `BookingService` throughout the application.
- **Factory Pattern**: Creates different types of notifications (e.g., Email, SMS) without specifying the exact class.
- **Observer Pattern**: Notifies users about booking confirmations.
- **Strategy Pattern**: Allows for different pricing strategies based on seat types.GitHub

---

📦 Project Structure

```
BookMyShow/
├── Main.java
├── models/
│   ├── User.java
│   ├── Movie.java
│   ├── Theater.java
│   ├── Show.java
│   ├── Seat.java
│   └── Booking.java
├── services/
```

```
│   ├── BookingService.java
│   └── NotificationService.java
├── notifications/
│   ├── Notification.java
│   ├── EmailNotification.java
│   ├── SMSNotification.java
│   └── NotificationFactory.java
├── pricing/
│   ├── PricingStrategy.java
│   ├── RegularPricingStrategy.java
│   └── PremiumPricingStrategy.java
└── observers/
    ├── Observer.java
    └── Subject.java
```

👨‍💻 Sample Code Snippets

User Model

1. **User Model**

models/User.java

```
package models;

public class User {
    private String name;
    private String email;
    private String phoneNumber;

    // Constructors, Getters, Setters
}
```

2. Movie Model

models/Movie.java

```
package models;

public class Movie {
    private String title;
    private String genre;
    private int duration; // in minutes
```

```
    // Constructors, Getters, Setters
}
```

3. Seat Model

models/Seat.java

```
package models;

public class Seat {
    private int seatNumber;
    private boolean isBooked;
    private String seatType; // e.g., Regular, Premium

    // Constructors, Getters, Setters
}
```

4. Notification Interface and Implementations

notifications/Notification.java

```
package notifications;

import models.User;

public interface Notification {
    void send(User user, String message);
}
```

notifications/EmailNotification.java

```
package notifications;

import models.User;

public class EmailNotification implements Notification {
    @Override
    public void send(User user, String message) {
        // Logic to send email
        System.out.println("Email sent to " + user.getEmail() + ": " + message);
    }
}
```

notifications/SMSNotification.java

```
package notifications;

import models.User;

public class SMSNotification implements Notification {
    @Override
    public void send(User user, String message) {
    // Logic to send SMS
    System.out.println("SMS sent to " + user.getPhoneNumber() + ": " + message);
    }
}
```

notifications/NotificationFactory.java

```
package notifications;

public class NotificationFactory {
    public static Notification createNotification(String type) {
        if (type.equalsIgnoreCase("EMAIL")) {
            return new EmailNotification();
        } else if (type.equalsIgnoreCase("SMS")) {
            return new SMSNotification();
        }
        throw new IllegalArgumentException("Unknown notification type");
    }
}
```

5. Pricing Strategy Interface and Implementations

pricing/PricingStrategy.java

```
package pricing;

public interface PricingStrategy {
    double calculatePrice(double basePrice);
}
```

pricing/RegularPricingStrategy.java

```
package pricing;

public class RegularPricingStrategy implements PricingStrategy {
    @Override
    public double calculatePrice(double basePrice) {
```

```
        return basePrice;
    }
}
```

pricing/PremiumPricingStrategy.java

```
package pricing;

public class PremiumPricingStrategy implements PricingStrategy {
    @Override
    public double calculatePrice(double basePrice) {
        return basePrice * 1.5; // 50% increase for premium seats
    }
}
```

6. Booking Service (Singleton Pattern)

services/BookingService.java

```
package services;

import models.*;
import notifications.Notification;
import notifications.NotificationFactory;
import pricing.PricingStrategy;

public class BookingService {
    private static BookingService instance;

    private BookingService() {}

    public static BookingService getInstance() {
        if (instance == null) {
            instance = new BookingService();
        }
        return instance;
    }

    public void bookSeat(User user, Show show, Seat seat, String notificationType, PricingStrateg
y pricingStrategy) {
        if (!seat.isBooked()) {
            seat.setBooked(true);
            double price = pricingStrategy.calculatePrice(show.getBasePrice());
            // Process payment logic here
```

```
            Notification notification = NotificationFactory.createNotification(notificationType);
            notification.send(user, "Booking confirmed for " + show.getMovie().getTitle() + ". Seat: "
+ seat.getSeatNumber() + ". Price: $" + price);
        } else {
            System.out.println("Seat already booked.");
        }
    }
}
```

7. Main Class to Simulate Booking

Main.java

```
import models.*;
import pricing.*;
import services.BookingService;

public class Main {
    public static void main(String[] args) {
        User user = new User("John Doe", "john@example.com", "1234567890");
        Movie movie = new Movie("Inception", "Sci-Fi", 148);
        Seat seat = new Seat(1, false, "Premium");
        Show show = new Show(movie, "7:00 PM", 10.0);

        BookingService bookingService = BookingService.getInstance();
        bookingService.bookSeat(user, show, seat, "EMAIL", new PremiumPricingStrategy());
    }
}
```

🧩 Observer Pattern in BookMyShow Context

Use Case:

Users subscribe to notifications for a movie release or show availability. When a new movie is added, all subscribed users are notified.

✅ Step-by-step Implementation

1. `Observer` Interface

```
public interface Observer {
    void update(String message);
```

```
}
```

2. `User` Class (Concrete Observer)

```
public class User implements Observer {
    private String name;
    private String email;

    public User(String name, String email) {
        this.name = name;
        this.email = email;
    }

    @Override
    public void update(String message) {
        System.out.println("Notification to " + name + ": " + message);
    }

    public String getName() {
        return name;
    }
}
```

3. `Subject` Interface

```
public interface Subject {
    void registerObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers(String message);
}
```

4. `MovieNotifier` Class (Concrete Subject)

```
import java.util.ArrayList;
import java.util.List;

public class MovieNotifier implements Subject {
    private List<Observer> observers;

    public MovieNotifier() {
        observers = new ArrayList<>();
```

```
    }

    @Override
    public void registerObserver(Observer observer) {
        observers.add(observer);
    }

    @Override
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers(String message) {
        for (Observer observer : observers) {
            observer.update(message);
        }
    }

    public void addNewMovie(String movieName) {
        String message = "New Movie Released: " + movieName;
        notifyObservers(message);
    }
}
```

5. `Main` Class to Run the Program

```
public class Main {
    public static void main(String[] args) {
        // Create notifier (subject)
        MovieNotifier notifier = new MovieNotifier();

        // Create users (observers)
        User user1 = new User("Sai", "sai@example.com");
        User user2 = new User("Ashish", "ashish@example.com");

        // Register observers
        notifier.registerObserver(user1);
        notifier.registerObserver(user2);

        // Notify all observers about new movie
        notifier.addNewMovie("Dune: Part Two");
```

```
        // One user unsubscribes
        notifier.removeObserver(user2);

        // Notify remaining observers
        notifier.addNewMovie("Oppenheimer");
    }
}
```

🖥️ Output

```
Notification to Sai: New Movie Released: Dune: Part Two
Notification to Ashish: New Movie Released: Dune: Part Two
Notification to Sai: New Movie Released: Oppenheimer
```

21. `Design Cab Booking (Ride Booking App)`

🚖 Cab Booking System Overview

Core Functionalities:

- **User Management**: Registration and authentication of riders and drivers.
- **Driver Management**: Tracking driver availability and locations.
- **Ride Booking**: Allowing users to request rides and matching them with nearby available drivers.
- **Trip Management**: Monitoring the status of rides from initiation to completion.
- **Notifications**: Informing users and drivers about ride statuses.YouTube+4GeeksforGeeks+4lldcoding.com+4GitHub+1GeeksforGeeks+1lldcoding.com

🧩 Applied Design Patterns

- **Singleton Pattern**: Ensures a single instance of services like `BookingService` throughout the application.
- **Factory Pattern**: Creates different types of notifications (e.g., Email, SMS) without specifying the exact class.
- **Observer Pattern**: Notifies users and drivers about ride updates.
- **Strategy Pattern**: Allows for different ride matching strategies (e.g., nearest driver, highest-rated driver).

📦 Project Structure

```
CabBookingSystem/
├── Main.java
├── models/
│   ├── User.java
│   ├── Driver.java
│   ├── Ride.java
│   └── Location.java
├── services/
│   ├── BookingService.java
│   └── NotificationService.java
├── notifications/
│   ├── Notification.java
│   ├── EmailNotification.java
│   ├── SMSNotification.java
│   └── NotificationFactory.java
├── strategies/
│   ├── RideMatchingStrategy.java
│   ├── NearestDriverStrategy.java
│   └── HighestRatedDriverStrategy.java
└── observers/
    ├── Observer.java
    └── Subject.java
```

👨‍💻 Sample Code Snippets

User Model

1. **User Model**

models/User.java

```
package models;

public class User {
    private String id;
    private String name;
    private Location location;

    // Constructors, Getters, Setters
}
```

2. Driver Model

models/Driver.java

```
package models;

public class Driver {
    private String id;
    private String name;
    private Location location;
    private boolean isAvailable;

    // Constructors, Getters, Setters
}
```

3. Ride Model

models/Ride.java GeeksforGeeks

```
package models;

public class Ride {
    private String id;
    private User user;
    private Driver driver;
    private Location pickupLocation;
    private Location dropoffLocation;
    private String status; // e.g., REQUESTED, IN_PROGRESS, COMPLETED

    // Constructors, Getters, Setters
}
```

4. Location Model

models/Location.java

```
package models;

public class Location {
    private double latitude;
    private double longitude;

    // Constructors, Getters, Setters
}
```

5. Notification Interface and Implementations

notifications/Notification.java arXiv+4Medium+4arXiv+4

```
package notifications;

public interface Notification {
    void send(String message);
}
```

notifications/EmailNotification.java

```
package notifications;

public class EmailNotification implements Notification {
    @Override
    public void send(String message) {
        // Logic to send email
        System.out.println("Email: " + message);
    }
}
```

notifications/SMSNotification.java

```
package notifications;

public class SMSNotification implements Notification {
    @Override
    public void send(String message) {
        // Logic to send SMS
        System.out.println("SMS: " + message);
    }
}
```

notifications/NotificationFactory.java

```
package notifications;

public class NotificationFactory {
    public static Notification createNotification(String type) {
        if (type.equalsIgnoreCase("EMAIL")) {
            return new EmailNotification();
        } else if (type.equalsIgnoreCase("SMS")) {
            return new SMSNotification();
        }
```

```
            throw new IllegalArgumentException("Unknown notification type");
        }
    }
```

6. Observer Pattern Implementation

observers/Observer.java

```
package observers;

public interface Observer {
    void update(String message);
}
```

observers/Subject.java

```
package observers;

import java.util.ArrayList;
import java.util.List;

public class Subject {
    private List<Observer> observers = new ArrayList<>();

    public void registerObserver(Observer observer) {
        observers.add(observer);
    }

    public void notifyObservers(String message) {
        for (Observer observer : observers) {
            observer.update(message);
        }
    }
}
```

7. Ride Matching Strategy Interface and Implementations

strategies/RideMatchingStrategy.java

```
package strategies;

import models.Driver;
import models.Location;

import java.util.List;
```

```
public interface RideMatchingStrategy {
    Driver matchDriver(List<Driver> drivers, Location userLocation);
}
```

strategies/NearestDriverStrategy.java

```
package strategies;

import models.Driver;
import models.Location;

import java.util.List;

public class NearestDriverStrategy implements RideMatchingStrategy {
    @Override
    public Driver matchDriver(List<Driver> drivers, Location userLocation) {
        // Logic to find the nearest driver
        return drivers.get(0); // Simplified for example
    }
}
```

8. Booking Service (Singleton Pattern)

services/BookingService.java

```
package services;

import models.*;
import notifications.Notification;
import notifications.NotificationFactory;
import observers.Subject;
import strategies.RideMatchingStrategy;

import java.util.List;

public class BookingService extends Subject {
    private static BookingService instance;
    private List<Driver> drivers;
    private RideMatchingStrategy matchingStrategy;

    private BookingService() {
        // Initialize drivers and strategy
    }
```

```
    public static BookingService getInstance() {
        if (instance == null) {
            instance = new BookingService();
        }
        return instance;
    }

    public void setMatchingStrategy(RideMatchingStrategy strategy) {
        this.matchingStrategy = strategy;
    }

    public void bookRide(User user, Location pickup, Location dropoff) {
        Driver driver = matchingStrategy.matchDriver(drivers, pickup);
        if (driver != null) {
            Ride ride = new Ride(/* initialize with user, driver, locations */);
            notifyObservers("Ride booked with driver: " + driver.getName());
            Notification notification = NotificationFactory.createNotification("SMS");
            notification.send("Your ride is confirmed with driver: " + driver.getName());
        } else {
            System.out.println("No drivers available");
        }
    }
}
```

9. Main Class to Simulate Booking

Main.java Wikipedia+2GeeksforGeeks+2Wikipedia+2

```
import models.*;
import observers.Observer;
import services.BookingService;
import strategies.NearestDriverStrategy;

public class Main {
    public static void main(String[] args) {
        BookingService bookingService = BookingService.getInstance();
        bookingService.setMatchingStrategy(new NearestDriverStrategy());

        Observer userObserver = message -> System.out.println("User Notification: " + message);
        bookingService.registerObserver(userObserver);

        User user = new User(/* initialize user */);
```

```
        Location pickup = new Location(/* initialize location */);
        Location dropoff = new Location(/* initialize location */);

        bookingService.bookRide(user, pickup, dropoff);
    }
}
```

22. Design a quick commerce grocery delivery system.

Requirements:

a. How would you manage inventory updates in real-time across multiple store locations?

b. How would you handle order prioritization for perishable and non-perishable goods?

c. What considerations would you make for handling large order volumes during peak demand times, such as festivals or weekends?

d. How would you ensure that delivery routes are optimized based on the proximity of multiple stores, traffic conditions, and delivery time slots?

Proposed Solution:

1. Managing Real-time Inventory Updates Across Multiple Store Locations

- **Approach**: Each store maintains its own inventory database, and updates (e.g., stock levels) are broadcast in real-time to a central system and related microservices. This can be achieved via event-driven architecture.

Design Patterns:

- **Observer Pattern**: Used for notifying the central system whenever there's an inventory change in any store. Each store's inventory can act as a subject, and the central system as an observer.
- **Event-Driven Architecture (EDA)**: Each store can publish inventory events (REST API or message queue), which are processed by a central service to update the global inventory.

Tools & Technologies:

- **Kafka** or **RabbitMQ** for asynchronous event processing.
- **Redis** or **Memcached** for fast access to inventory data across stores.

```
public interface InventoryObserver {
    void update(String storeId, Product product, int quantity);
}
```

```
public class CentralInventoryService implements InventoryObserver {
    @Override
    public void update(String storeId, Product product, int quantity) {
        // Update central inventory or notify relevant microservices.
       //  Let's assume we can notify all the relevant stores within a location.
        System.out.println("Central System Updated: " + storeId + " updated " + product.getName
() + " stock to " + quantity);
    }
}
```

```
public class StoreInventory {
    private List<InventoryObserver> observers = new ArrayList<>();

    public void addObserver(InventoryObserver observer) {
        observers.add(observer);
    }

    public void updateProductStock(Product product, int quantity) {
        notifyObservers(product, quantity);
    }

    private void notifyObservers(Product product, int quantity) {
        for (InventoryObserver observer : observers) {
            observer.update(this.storeId, product, quantity);
        }
    }
}
```

```
public class Product {
    private String name;
    private boolean isPerishable;

    public Product(String name, boolean isPerishable) {
        this.name = name;
        this.isPerishable = isPerishable;
    }

    public boolean isPerishable() {
        return isPerishable;
    }
```

```
    public String getName() {
        return name;
    }
}
```

```
 StoreInventory storeInventory = new StoreInventory();
  storeInventory.addObserver(centralInventoryService);
  Product product = new Product("Apple", true);
  storeInventory.updateProductStock(product1 100); // Notify observers
```

```
OUTPUT: Central System Updated: Store123 updated Apple stock to 100
```

---

2. Order Prioritization for Perishable and Non-perishable Goods

- **Approach**: Orders with perishable goods should have higher priority over non-perishable ones, especially in terms of delivery time.

Design Patterns:

- **Strategy Pattern**: For dynamically applying different prioritization strategies based on factors like delivery time, perishability, and proximity to stores.
- **Chain of Responsibility Pattern**: Use this to allow prioritization based on product types. Different handlers (perishable, non-perishable, default or mixed) will process the orders.

```
class Order {
    private List<Product> products;

    public Order(List<Product> products) {
        this.products = products;
    }

    public boolean hasPerishableItems() {
        return products.stream().anyMatch(Product::isPerishable);
    }

    public boolean hasNonPerishableItems() {
        return products.stream().anyMatch(product -> !product.isPerishable());
    }
}
```

```
@Data
@AllArgsConstructor
public class OrderPriorityConfig {
    private int highPriority;
    private int mediumPriority;
    private int lowPriority;
    private int lowestPriority = 5;


    public OrderPriorityConfig() {
        this.highPriority = 1;
        this.mediumPriority = 2;
        this.lowPriority = 3;
    }

}
```

```
public interface OrderPriorityStrategy {
    int calculatePriority(Order order, OrderPriorityConfig config);
}
```

```
public class PerishablePriorityStrategy implements OrderPriorityStrategy {
    @Override
    public int calculatePriority(Order order, OrderPriorityConfig config) {
        return order.hasPerishableItems() ? config.getHighPriority() : config.getLowPriority();
    }
}
```

```
public class NonPerishablePriorityStrategy implements OrderPriorityStrategy {
    @Override
    public int calculatePriority(Order order, OrderPriorityConfig config) {
        return order.hasNonPerishableItems() ? config.getMediumPriority() : config.getLowPriority
();
    }
}
```

```
interface OrderPriorityHandler {
    void setNextHandler(OrderPriorityHandler nextHandler);
```

```
    Integer calculatePriority(Order order);
}
```

```
abstract class BasePriorityHandler implements OrderPriorityHandler {
    protected OrderPriorityHandler nextHandler;
    protected OrderPriorityStrategy strategy;

    public BasePriorityHandler(OrderPriorityStrategy strategy) {
        this.strategy = strategy;
    }

    @Override
    public void setNextHandler(OrderPriorityHandler nextHandler) {
        this.nextHandler = nextHandler;
    }

    protected Integer passToNext(Order order) {
        if (nextHandler != null) {
            return nextHandler.calculatePriority(order);
        }
        return null;
    }
}
```

```
class PerishablePriorityHandler extends BasePriorityHandler {
    public PerishablePriorityHandler(OrderPriorityStrategy strategy) {
        super(strategy);
    }

    @Override
    public Integer calculatePriority(Order order) {
        if (order.hasPerishableItems()) {
            return strategy.calculatePriority(order, new OrderPriorityConfig());
        } else {
            return passToNext(order);
        }
    }
}
```

```
class NonPerishablePriorityHandler extends BasePriorityHandler {
    public NonPerishablePriorityHandler(OrderPriorityStrategy strategy) {
```

```
        super(strategy);
    }

    @Override
    public Integer calculatePriority(Order order) {
        if (order.hasNonPerishableItems()) {
            return strategy.calculatePriority(order, new OrderPriorityConfig());
        } else {
            return passToNext(order);
        }
    }
}
```

```
class DefaultPriorityHandler extends BasePriorityHandler {
    public DefaultPriorityHandler() {
        super(null);
    }

    @Override
    public Integer calculatePriority(Order order) {
        return new OrderPriorityConfig().getLowestPriority();
    }
}
```

```
@Service
class OrderPriorityService {
    private OrderPriorityHandler chain;

    public OrderPriorityService() {
        OrderPriorityHandler perishableHandler = new PerishablePriorityHandler(new PerishablePr
iorityStrategy());

        OrderPriorityHandler nonPerishableHandler = new NonPerishablePriorityHandler(new Non
PerishablePriorityStrategy());

        OrderPriorityHandler defaultHandler = new DefaultPriorityHandler();

        perishableHandler.setNextHandler(nonPerishableHandler);
        nonPerishableHandler.setNextHandler(defaultHandler);

        this.chain = perishableHandler;
    }
```

```
    public int calculatePriority(Order order) {
        return chain.calculatePriority(order);
    }
}
```

```
Order order1 = new Order(Arrays.asList(new Product("A",true)));
System.out.println("Order 1 Priority: " + orderPriorityService.calculatePriority(order1)); // Output:
1

Order order2 = new Order(Arrays.asList(new Product("B", false)));
System.out.println("Order 2 Priority: " + orderPriorityService.calculatePriority(order2)); // Outpu
t: 2

Order order3 = new Order(Arrays.asList());
System.out.println("Order 3 Priority: " + orderPriorityService.calculatePriority(order3)); // Outpu
t: 5
```

3. Handling Large Order Volumes During Peak Demand (Festivals or Weekends)

- **Approach**: During peak demand times, the system must balance load across multiple microservices and ensure that large order volumes don't overwhelm any one part of the system.

Design Patterns:

- **Circuit Breaker Pattern**: To handle failures gracefully during high demand (e.g., one microservice is overwhelmed and slow, so it triggers fallback logic).
- **Bulkhead Pattern**: Isolate system resources to avoid cascading failures. For instance, isolate different parts of the system such as payment processing or inventory updates.
- **Queue-Based Load Leveling**: Use message queues (Kafka, SQS) to handle asynchronous order processing and spark for distributed processing.
- **SNS with SQS:** Use this combination for message polling across various receivers in notification systems.

```
public class OrderService {
    @CircuitBreaker(fallbackMethod = "fallbackProcessOrder")
    public void processOrder(Order order) {
        // Order processing logic
    }

    public void fallbackProcessOrder(Order order) {
        // Fallback logic if the circuit breaker is triggered
```

```
    }
}
```

Handling large order volumes during peak demand times, such as festivals or weekends, requires careful planning and optimization of resources to ensure smooth operations. Here are the key considerations for managing high traffic during such periods:

1. Scalability of Infrastructure

- **Cloud-Based Auto Scaling**: Use cloud infrastructure (e.g., AWS, Azure, GCP) with auto-scaling capabilities to dynamically adjust resources (e.g., servers, databases) based on traffic. This ensures you handle increased demand without manual intervention.
- **Horizontal Scaling**: Add more servers to distribute the load across multiple nodes rather than relying on a single powerful machine.
- **Load Balancers**: Implement load balancing to evenly distribute traffic across servers, preventing any one server from becoming overwhelmed.

2. Caching Mechanisms

- **Use Distributed Caching**: Implement caching solutions like Redis, Memcached, or CDN to reduce load on your database. Frequently accessed data, such as product catalogs or user profiles, should be cached to speed up response times.
- **Inventory Caching**: Cache inventory status locally for each store location, with regular syncs to the central inventory system. This prevents constant database hits for inventory checks during peak times.

3. Optimized Inventory Management

- **Real-Time Inventory Updates**: Use real-time inventory management with strong consistency to avoid stockouts or overselling. Systems should be capable of quick updates as inventory is consumed.
- **Distributed Inventory**: Spread inventory across multiple warehouses and stores to ensure proximity to delivery locations and faster fulfillment. This also helps balance order volumes across regions.
- **Buffer Stock**: Maintain a buffer stock of fast-moving items, especially perishable goods, during high-demand periods to prevent stockouts.

4. Order Prioritization and Batch Processing

- **Order Priority Based on Type and Urgency**: Implement a priority system for perishable vs. non-perishable goods. Perishable items should be fulfilled first to avoid spoilage, while high-priority customers (e.g., express deliveries) should also be handled sooner.
- **Batch Processing**: Group orders based on location or product category to optimize picking, packing, and delivery. This can reduce the load on the logistics team and improve efficiency

during peak times.

5. Efficient Delivery Logistics

- **Route Optimization**: Use route optimization algorithms (potentially enhanced by AI) to select the best delivery paths based on traffic, delivery location proximity, and available drivers. Solutions like Google Maps API, or custom-built tools with Spark and machine learning, can help optimize routes.
- **Dynamic Assignment of Drivers**: During high demand, use dynamic driver assignment by considering proximity to stores and current traffic conditions. This ensures faster delivery and reduces idle time for drivers.
- **Partnerships with 3rd Party Delivery Services**: If your fleet can't handle the volume, partner with third-party delivery services like Uber, DoorDash, or local logistics firms to expand capacity on demand.

6. Queue and Throttling Mechanisms

- **Queue-Based Order Processing**: Use message queues (e.g., Kafka, RabbitMQ) to decouple services and handle order bursts smoothly. Orders are placed in a queue and processed asynchronously, allowing for smoother operation during peak times.
- **Throttling**: Implement API request throttling to prevent system overload. This ensures that services don’t crash under sudden spikes in traffic, and important transactions can still be processed.

7. Peak-Time Capacity Planning

- **Predictive Analytics**: Use historical data to forecast order volumes during peak times. This allows you to prepare well in advance with adequate stock, delivery capacity, and customer support.
- **Elastic Workforce**: Hire temporary workers or have on-call staff for critical operations like packing, sorting, and customer service during peak times. Use workforce scheduling tools to ensure enough manpower is available.
- **Stock Pre-Allocation**: Pre-allocate stock to stores based on demand predictions. Use AI/ML models to predict product demand and ensure that popular items are stocked up.

8. System Resilience and Failover

- **Database Sharding**: Implement database sharding to distribute the load on the database across multiple instances. Each shard handles a subset of data, reducing the impact of heavy read/write operations.
- **Backup and Disaster Recovery**: Ensure that backup systems and disaster recovery plans are in place. If one system fails, another should take over without downtime.
- **Failover Systems**: If your main infrastructure experiences a bottleneck or failure, have redundant systems that can take over operations seamlessly to avoid disruption.

9. Customer Communication

- **Real-Time Updates**: Provide real-time status updates to customers about their order status, expected delivery times, and delays (if any). This can reduce customer inquiries and frustration during busy periods.
- **Surge Pricing and Expected Delays**: During peak demand, communicate the possibility of delays upfront or implement surge pricing for express services, much like Uber does, to manage customer expectations and load.

10. Optimized Checkout and Payment Systems

- **Quick and Efficient Checkout**: Optimize the checkout process by using cached user profiles, one-click payments, and seamless integration with multiple payment gateways.
- **Payment Load Balancing**: Use multiple payment gateways to handle large volumes of transactions without causing delays or downtime.

11. Monitoring and Alerting

- **Real-Time System Monitoring**: Implement robust monitoring tools (e.g., Datadog, New Relic, Grafana) to track system performance, database health, and order fulfillment metrics in real-time. Set up alerts to be notified of potential bottlenecks early.
- **Order Fulfillment Tracking**: Monitor order processing times, warehouse bottlenecks, and delivery delays during peak times. Implement contingency plans if certain thresholds are exceeded.

```
class CacheService {
    public void updateInventory(String storeId, String productId, int quantity) {
        System.out.println("Updated inventory in Redis cache for Store: " + storeId + ", Product: " +
productId + ", Quantity: " + quantity);
    }
}
```

```
public class CentralInventoryService implements InventoryObserver {
    private KafkaService kafkaService;
    private CacheService cacheService; // redis

    public CentralInventoryService(KafkaService kafkaService, CacheService cacheService) {
        this.kafkaService = kafkaService;
        this.cacheService = cacheService;
    }

    @Override
    public void update(String storeId, Product product, int quantity) {
```

```
        cacheService.updateInventory(storeId, product.getName(), quantity);
        kafkaService.publish("inventory_updates", new InventoryUpdateEvent(storeId, product.get
Name(), quantity));
        // other microservices will consume
        System.out.println("Central inventory updated and notification sent via Kafka.");
    }
}
```

```
class InventoryUpdateEvent {
    private String storeId;
    private String productId;
    private int quantity;

    public InventoryUpdateEvent(String storeId, String productId, int quantity) {
        this.storeId = storeId;
        this.productId = productId;
        this.quantity = quantity;
    }

    @Override
    public String toString() {
        return "InventoryUpdateEvent{" +
                "storeId='" + storeId + '\'' +
                ", productId='" + productId + '\'' +
                ", quantity=" + quantity +
                '}';
    }
}
```

```
class KafkaService<T> {
    public void publish(String topic, T event) {
        System.out.println("Published event to topic: " + topic + " → " + event);
    }
}
```

```
public class StoreInventory {
    private String storeId;
    private List<InventoryObserver> observers = new ArrayList<>();
    private CacheService cacheService;

    public StoreInventory(String storeId, CacheService cacheService) {
```

```
        this.storeId = storeId;
        this.cacheService = cacheService;
    }

    public void addObserver(InventoryObserver observer) {
        observers.add(observer);
    }

    public void updateProductStock(Product product, int quantity) {
        cacheService.updateInventory(storeId, product.getName(), quantity);
        System.out.println("Stock updated in local cache for store: " + storeId);
        notifyObservers(product, quantity);
    }

    private void notifyObservers(Product product, int quantity) {
        for (InventoryObserver observer : observers) {
            observer.update(this.storeId, product, quantity);
        }
    }
}
```

```
public class Driver {

    public static void main(String[] args) {
        KafkaService<InventoryUpdateEvent> kafkaService = new KafkaService<>();
        CacheService cacheService = new CacheService();
        CentralInventoryService centralInventoryService = new CentralInventoryService(kafkaServ
ice, cacheService);
        StoreInventory storeInventory = new StoreInventory("Store1", cacheService);
        storeInventory.addObserver(centralInventoryService);

        Product milk = new Product( "Milk", true);
        Product cereal = new Product( "Cereal", false);
        System.out.println("Peak demand updates:");
        storeInventory.updateProductStock(milk, 100);
        storeInventory.updateProductStock(cereal, 200);
        System.out.println("\nRegular updates:");
        storeInventory.updateProductStock(milk, 50);
    }

}
```

```
OUTPUT:

Peak demand updates:
Updated inventory in Redis cache for Store: Store1, Product: Milk, Quantity: 100
Stock updated in local cache for store: Store1
Updated inventory in Redis cache for Store: Store1, Product: Milk, Quantity: 100
Published event to topic: inventory_updates → InventoryUpdateEvent{storeId='Store1', productId='Milk', quantity=100}
Central inventory updated and notification sent via Kafka.
Updated inventory in Redis cache for Store: Store1, Product: Cereal, Quantity: 200
Stock updated in local cache for store: Store1
Updated inventory in Redis cache for Store: Store1, Product: Cereal, Quantity: 200
Published event to topic: inventory_updates → InventoryUpdateEvent{storeId='Store1', productId='Cereal', quantity=200}
Central inventory updated and notification sent via Kafka.

Regular updates:
Updated inventory in Redis cache for Store: Store1, Product: Milk, Quantity: 50
Stock updated in local cache for store: Store1
Updated inventory in Redis cache for Store: Store1, Product: Milk, Quantity: 50
Published event to topic: inventory_updates → InventoryUpdateEvent{storeId='Store1', productId='Milk', quantity=50}
Central inventory updated and notification sent via Kafka.
```

Refer to my other articles mentioned below to learn more about how to use kafka along with redis and spark for handling large scale:

https://www.linkedin.com/pulse/design-system-based-internal-app-swiggyzomato-notify-all-ashish-uv2fc

https://www.linkedin.com/pulse/designing-leaderboard-internal-app-used-swiggy-zomato-sai-ashish-kep4c

4. Optimizing Delivery Routes Based on Proximity, Traffic, and Delivery Time Slots

- **Approach**: A dynamic routing algorithm that optimizes delivery based on multiple factors such as store proximity, traffic data, and time slots using spark, kafka, min heap, batch processing and redis cache and influence factor computation based on related driver filter data from the database based on route, shift, location, and performance metrics such as delivery times and customer ratings.

```
@Service
public class CacheService {
    private Map<String, Driver> driverCache;
```

```
    private TrafficData trafficDataCache;

    public CacheService() {
        driverCache = new HashMap<>();
        trafficDataCache = new TrafficData();
    }

    public Map<String, Driver> getAvailableDrivers() {
        return driverCache;
    }

    public TrafficData getTrafficData() {
        return trafficDataCache;
    }

    public void updateDriverLocation(String driverId, Location location) {
        Driver driver = driverCache.get(driverId);
        if (driver != null) {
            driver.setLocation(location);
        }
    }
}
```

```
@Data
public class Delivery {
    private String customerId;
    private String storeId;
    private String deliveryId;
    private Location deliveryLocation;


    public Delivery(String customerId, String storeId, String deliveryId, Location deliveryLocation)
{
        this.customerId = customerId;
        this.storeId = storeId;
        this.deliveryId = deliveryId;
        this.deliveryLocation = deliveryLocation;
    }

}
```

```
public class DeliveryRequestEvent extends KafkaEvent {
    private List<Store> stores;
    private Delivery delivery;

    public DeliveryRequestEvent(List<Store> stores, Delivery delivery) {
        this.stores = stores;
        this.delivery = delivery;
    }

    public List<Store> getStores() {
        return stores;
    }

    public Delivery getDelivery() {
        return delivery;
    }
}
```

```
@NoArgsConstructor
@AllArgsConstructor
@Data
public class Driver {
    private String id = "1234";
    private Location location;
    private int estimatedTime = 23456789;
}
```

```
public class DeliveryRequestEvent extends KafkaEvent {
    private List<Store> stores;
    private Delivery delivery;

    public DeliveryRequestEvent(List<Store> stores, Delivery delivery) {
        this.stores = stores;
        this.delivery = delivery;
    }

    public List<Store> getStores() {
        return stores;
    }

    public Delivery getDelivery() {
```

```
        return delivery;
    }
}
```

```
public class DriverLocationUpdateEvent extends KafkaEvent {
    private String driverId;
    private Location location;

    public DriverLocationUpdateEvent(String driverId, Location location) {
        this.driverId = driverId;
        this.location = location;
    }

    public String getDriverId() {
        return driverId;
    }

    public Location getLocation() {
        return location;
    }
}
```

```
import java.util.UUID;
import java.time.Instant;

public abstract class KafkaEvent {
    private final String eventId;
    private final Instant timestamp;

    public KafkaEvent() {
        this.eventId = UUID.randomUUID().toString();
        this.timestamp = Instant.now();
    }

    public String getEventId() {
        return eventId;
    }

    public Instant getTimestamp() {
        return timestamp;
    }
```

```
    @Override
    public String toString() {
        return "KafkaEvent{" +
                "eventId='" + eventId + '\'' +
                ", timestamp=" + timestamp +
                '}';
    }
}
```

```
@Service
public class KafkaService<T extends KafkaEvent> {
    public void subscribe(String topic, Consumer<T> consumer) {
        System.out.println("Subscribed to topic: " + topic);
    }

    public void publish(String topic, KafkaEvent event) {
        System.out.println("Published event to topic: " + topic + ", event: " + event);
    }
}
```

```
@Data
@AllArgsConstructor
public class Location {
    private double latitude;
    private double longitude;
    private String address;
    private String city;

    public Location(double latitude, double longitude) {
        this.latitude = latitude;
        this.longitude = longitude;
    }

    @Override
    public String toString() {
        return "Location{" +
                "latitude=" + latitude +
                ", longitude=" + longitude +
                (address != null ? ", address='" + address + '\'' : "") +
                (city != null ? ", city='" + city + '\'' : "") +
                '}';
```

```
    }
}
```

```
public class OptimizedRouteEvent extends KafkaEvent {
    private Route route;

    public OptimizedRouteEvent(Route route) {
        this.route = route;
    }
}
```

```
@Data
@AllArgsConstructor
public class Route {
    private String id;
    private Location deliveryLocation;
    private int estimatedTime;
}
```

```
@Service
@RequiredArgsConstructor
public class RouteOptimizationService {
    private final CacheService cacheService;
    private final KafkaService<DeliveryRequestEvent> deliveryRequestService;
    private final KafkaService<DriverLocationUpdateEvent> driverLocationService;
    private final SparkService sparkService;

    public Route optimizeRoute(List<Store> stores, Delivery delivery) {
        Map<String, Driver> availableDrivers = cacheService.getAvailableDrivers();
        TrafficData trafficData = cacheService.getTrafficData();
        Location deliveryLocation = delivery.getDeliveryLocation();
        List<Location> storeLocations = stores.stream().map(Store::getLocation).toList();
        List<Driver> optimizedDrivers = sparkService.optimizeDrivers(availableDrivers, storeLocati
ons, deliveryLocation, trafficData);
        Driver bestDriver = selectBestDriver(optimizedDrivers, delivery);
        return new Route(bestDriver.getId(), deliveryLocation, bestDriver.getEstimatedTime());
    }

    private Driver selectBestDriver(List<Driver> drivers, Delivery delivery) {
        PriorityQueue<Driver> driversPQ = new PriorityQueue<>(drivers);
        // use min-heap to find the best driver and handle corner cases
```

```
        return driversPQ.poll();
    }

    public void listenForEvents() {
        deliveryRequestService.subscribe("delivery_requests", this::onDeliveryRequest);
        driverLocationService.subscribe("driver_locations", this::onDriverLocationUpdate);
    }

    private void onDeliveryRequest(DeliveryRequestEvent event) {
        List<Store> stores = event.getStores();
        Delivery delivery = event.getDelivery();
        Route route = optimizeRoute(stores, delivery);
        deliveryRequestService.publish("optimized_routes", new OptimizedRouteEvent(route));
    }

    private void onDriverLocationUpdate(DriverLocationUpdateEvent event) {
        cacheService.updateDriverLocation(event.getDriverId(), event.getLocation());
    }
}
```

```
@Service
public class SparkService {

    public List<Driver> optimizeDrivers(Map<String, Driver> availableDrivers, List<Location> stor
eLocations, Location deliveryLocation, TrafficData trafficData) {
        // Spark job to analyze historical data, traffic conditions, and optimize drivers
        // Mock optimization logic

        // Implement batch processing with a Kafka executor to
        // filter data from the database based on route, shift, location,
        // and performance metrics such as delivery times and customer ratings.
        // Utilize an influence factor computation to rank drivers.
        // Cache the relevant drivers and use a priority queue or
        // min-heap to identify the optimal driver for each delivery.

        System.out.println("Running Spark job to optimize drivers...");

        availableDrivers.put("1234", new Driver());

        return availableDrivers.values().stream().toList();
    }
}
```

```
public class TrafficData {
    private Map<String, Integer> trafficMap;  // Map of location to traffic delay in minutes

    public TrafficData() {
        trafficMap = new HashMap<>();
    }

    public int getTrafficDelay(String location) {
        return trafficMap.getOrDefault(location, 0);
    }

    public void updateTraffic(String location, int delay) {
        trafficMap.put(location, delay);
    }
}
```

```
KafkaService<DeliveryRequestEvent> deliveryRequestService = new KafkaService<>();

KafkaService<DriverLocationUpdateEvent> driverLocationService = new KafkaService<>();

RouteOptimizationService routeService = new RouteOptimizationService(cacheService, deliver
yRequestService, driverLocationService, sparkService);

Delivery delivery = new Delivery("123", "456", "789", new Location(12.9716, 77.5946));
List<Store> stores = Arrays.asList(
        new Store("Store1", new Location(12.9716, 77.5946)),
        new Store("Store2", new Location(12.9721, 77.5955))
);

Route route = routeService.optimizeRoute(stores, delivery);
System.out.println("Optimized Route: " + route);
```

```
OUTPUT:

Running Spark job to optimize drivers...
Optimized Route: Route(id=1234, deliveryLocation=Location{latitude=12.9716, longitude=77.594
6}, estimatedTime=23456789)
```

Related articles to understand how to compute related drivers and rate them at scale:

https://www.linkedin.com/pulse/design-system-based-internal-app-swiggyzomato-notify-all-ashish-uv2fc

https://www.linkedin.com/pulse/designing-leaderboard-internal-app-used-swiggy-zomato-sai-ashish-kep4c

#softwaredevelopment #softwareengineering #applicationdevelopment #systemdesign

---

23. Design a system inspired by the internal app of Swiggy/Zomato to notify all drivers when the rating of any driver changes.

Note: Millions of drivers can receive notification.

This article is extension to my previous article:

https://www.linkedin.com/pulse/designing-leaderboard-internal-app-used-swiggy-zomato-sai-ashish-kep4c/?trackingId=57m%2FOdVkQoqKNkEKYAh3%2Fg%3D%3D

System Components:

1. Order Service: Manages orders and publishes fulfillment events.

2. Rating Service: Updates driver ratings and influences related drivers.

3. Leaderboard Service: Maintains and serves the leaderboard.

4. Message Broker: Facilitates event-driven communication (e.g., Kafka).

5. Distributed Processing: Uses Spark for parallel processing of events.

6. Database: Stores driver ratings, related driver information, and leaderboard data. SQL aggregations can be used effectively to optimise various operations.

Learn more about how SQL aggregations work internally:

https://www.linkedin.com/feed/update/urn:li:activity:7220646194107011073/

7. Cache: Provides fast access to frequently requested data.

8. Worker Pool: Manages concurrent tasks efficiently (e.g., Java ExecutorService).

9. Fault Tolerance: Handles failures gracefully with retries and monitoring

10. **Notification Service**: Sends notifications to drivers using AWS SNS and SQS.

To handle notifications for millions of users efficiently, we can use AWS services like Amazon SNS (Simple Notification Service) and Amazon SQS (Simple Queue Service) in combination. Amazon SNS can be used to publish messages to a large number of subscribers, and Amazon SQS can handle the message queuing for individual notifications. This approach ensures scalability, reliability, and decoupling of services.

We'll use the **Observer** design (**Behavioral**) pattern to notify drivers about their ratings. The Observer pattern allows an object, called the subject, to maintain a list of dependents, called observers, and notify them automatically of any state changes. Here's how we can integrate this pattern into the existing system:

Overview

1. **Subject**: The RatingService acts as the subject that maintains a list of observers (drivers).
2. **Observers**: Each driver who wants to be notified about their rating updates will be an observer.
3. **Notification**: When the rating of a driver is updated, the RatingService notifies all registered observers.

Detailed Design:

1. Configure SNS and SQS

- Set up an SNS topic for broadcasting messages.
- Set up SQS queues for individual driver notifications.

2. Order Service:

- Manages orders and publishes Kafka events with driver ID and rating upon order fulfillment.

```
public class OrderService {
    private KafkaProducer<String, String> kafkaProducer;

    public void orderFulfilled(Order order) {
        String event = new Gson().toJson(new OrderFulfilledEvent(order.getId(), order.getDriverId
(), order.getRating()));
        kafkaProducer.send(new ProducerRecord<>("order-fulfilled", event));
    }
}
```

Spark Job:

```
object RatingUpdateJob { ... }
```

3. Message Broker:

- Ensures reliable event delivery and supports horizontal scaling using Apache Kafka. It allows for horizontal scaling by distributing events across multiple partitions.

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
KafkaProducer<String, String> producer = new KafkaProducer<>(props);
```

4. Observer Interface

```
public interface DriverObserver {
    void updateRating(int driverId, double newRating);
}
```

5. Concrete Observer

```
public class Driver implements DriverObserver {
    private int driverId;
    private double rating;

    public Driver(int driverId) {
        this.driverId = driverId;
    }

    @Override
    public void updateRating(int driverId, double newRating) {
        if (this.driverId == driverId) {
            this.rating = newRating;
            sendNotification(driverId, newRating);
        }
    }

    public int getDriverId() {
        return driverId;
    }

    private void sendNotification(int driverId, double newRating) {
        String message = "Driver " + driverId + " notified. New Rating: " + newRating;
        // Implement notification logic using AWS SNS or SQS
        NotificationService.sendNotification(driverId, message);
    }
}
```

6. Concrete Observer

```
public class NotificationService {
    private static final AmazonSNS snsClient = AmazonSNSClientBuilder.defaultClient();
    private static final AmazonSQS sqsClient = AmazonSQSClientBuilder.defaultClient();
    private static final String SNS_TOPIC_ARN = "";
    private static final String SQS_QUEUE_URL = "";

    public static void sendNotification(int driverId, String message) {
        // Publish to SNS topic
```

```
        PublishRequest publishRequest = new PublishRequest(SNS_TOPIC_ARN, message);
        PublishResult publishResult = snsClient.publish(publishRequest);
        System.out.println("Message sent to SNS topic. Message ID: " + publishResult.getMessage
Id());

        // Send to SQS queue
        SendMessageRequest sendMessageRequest = new SendMessageRequest(SQS_QUEUE_U
RL, message);
        sqsClient.sendMessage(sendMessageRequest);
        System.out.println("Message sent to SQS queue. Message: " + message);
    }
}
```

7. Rating Service

```
import java.util.*;
import java.util.concurrent.*;
import java.util.stream.Collectors;

public class RatingService {
    private MessageBroker messageBroker;
    private Database database;
    private ExecutorService executorService = Executors.newFixedThreadPool(20);
    private List<DriverObserver> observers = new ArrayList<>();
    private static final int BATCH_SIZE = 1000;

    public void start() {
        messageBroker.subscribe(event -> executorService.submit(() -> processEvent(event)));
    }

    public void registerObserver(DriverObserver observer) {
        observers.add(observer);
    }

    public void removeObserver(DriverObserver observer) {
        observers.remove(observer);
    }

    private void notifyObservers(int driverId, double newRating) {
        List<DriverObserver> relevantObservers = observers.stream()
                .filter(observer -> ((Driver) observer).getDriverId() == driverId)
                .collect(Collectors.toList());
```

```
        List<List<DriverObserver>> batches = createBatches(relevantObservers, BATCH_SIZE);
        for (List<DriverObserver> batch : batches) {
            executorService.submit(() -> processBatch(batch, driverId, newRating));
        }
    }

    private void processBatch(List<DriverObserver> batch, int driverId, double newRating) {
        for (DriverObserver observer : batch) {
            observer.updateRating(driverId, newRating);
        }
    }

    private List<List<DriverObserver>> createBatches(List<DriverObserver> observers, int batch
Size) {
        List<List<DriverObserver>> batches = new ArrayList<>();
        for (int i = 0; i < observers.size(); i += batchSize) {
            int end = Math.min(i + batchSize, observers.size());
            batches.add(observers.subList(i, end));
        }
        return batches;
    }

    private void processEvent(OrderFulfilledEvent event) {
        int driverId = event.getDriverId();
        int rating = event.getRating();
        updateDriverRating(driverId, rating);
        updateRelatedDriversRating(driverId, rating);
        updateLeaderboard(driverId);
    }

    private void updateDriverRating(int driverId, int rating) {
        DriverRating driverRating = database.getDriverRating(driverId);
        double newRating = (driverRating.getRating() * driverRating.getTotalRatings() + rating) / (dr
iverRating.getTotalRatings() + 1);
        driverRating.setRating(newRating);
        driverRating.incrementTotalRatings();
        database.updateDriverRating(driverRating);
        notifyObservers(driverId, newRating);
    }

    private void updateRelatedDriversRating(int driverId, int rating) {
        List<Integer> relatedDrivers = findRelatedDrivers(driverId);
        List<List<Integer>> batches = createBatches(relatedDrivers, BATCH_SIZE);
```

```
        for (List<Integer> batch : batches) {
            executorService.submit(() -> processBatch(batch, rating));
        }
    }

    private List<Integer> findRelatedDrivers(int driverId) {
        // Implement filter logic based on route, shift, location, etc.
        return null;
    }

    private List<List<Integer>> createBatches(List<Integer> drivers, int batchSize) {
        List<List<Integer>> batches = new ArrayList<>();
        for (int i = 0; i < drivers.size(); i += batchSize) {
            batches.add(drivers.subList(i, Math.min(i + batchSize, drivers.size())));
        }
        return batches;
    }

    private void processBatch(List<Integer> batch, int rating) {
        for (int relatedDriverId : batch) {
            DriverRating relatedDriverRating = database.getDriverRating(relatedDriverId);
            double newRating = relatedDriverRating.getRating() + calculateInfluenceFactor(relatedDr
iverId) * (rating - relatedDriverRating.getRating());
            relatedDriverRating.setRating(newRating);
            database.updateDriverRating(relatedDriverRating);
        }
    }

    private double calculateInfluenceFactor(int driverId) {
        return 0.1; // 10% influence
    }

    private void updateLeaderboard(int driverId) {
        LeaderboardService.updateDriverInLeaderboard(driverId);
    }
}
```

8. Leaderboard Service

```
import java.util.*;
import java.util.concurrent.*;

public class LeaderboardService {
```

```
    private Database database;
    private Cache cache;
    private static final int LEADERBOARD_SIZE = 100;

    public static void updateDriverInLeaderboard(int driverId) {
        DriverRating driverRating = database.getDriverRating(driverId);
        double rating = driverRating.getRating();
        cache.updateLeaderboard(driverId, rating);
        updateLeaderboardInDatabase();
    }

    private static void updateLeaderboardInDatabase() {
        List<DriverRating> topDrivers = cache.getTopDrivers(LEADERBOARD_SIZE);
        database.updateLeaderboard(topDrivers);
    }
}
```

9. **Database**

```
import java.util.*;
import java.util.concurrent.*;

public class Database {
    private Map<Integer, DriverRating> driverRatings = new ConcurrentHashMap<>();
    private List<DriverRating> leaderboard = new ArrayList<>();

    public DriverRating getDriverRating(int driverId) {
        return driverRatings.get(driverId);
    }

    public void updateDriverRating(DriverRating driverRating) {
        driverRatings.put(driverRating.getDriverId(), driverRating);
    }

    public void updateLeaderboard(List<DriverRating> topDrivers) {
        leaderboard = topDrivers;
    }
}
```

10. Cache

```
import java.util.*;
import java.util.concurrent.*;
```

```
import java.util.stream.Collectors;

public class Cache {
    private Map<Integer, Double> leaderboardMap = new ConcurrentHashMap<>();
    private PriorityQueue<DriverRating> leaderboardQueue = new PriorityQueue<>(Comparator.
comparingDouble(DriverRating::getRating).reversed());

    public void updateLeaderboard(int driverId, double rating) {
        leaderboardMap.put(driverId, rating);
        leaderboardQueue.add(new DriverRating(driverId, rating));
        if (leaderboardQueue.size() > LEADERBOARD_SIZE) {
            leaderboardQueue.poll();
        }
    }

    public List<DriverRating> getTopDrivers(int size) {
        return leaderboardQueue.stream().limit(size).collect(Collectors.toList());
    }
}
```

Optimization Strategies

1. **Batch Processing**: Batch updates to the database and leaderboard reduce the frequency of writes and improve performance.
2. **Asynchronous Processing**: Use asynchronous methods and a worker pool to handle updates concurrently.
3. **Distributed Systems**: Leverage distributed systems like Kafka for message brokering and Spark for large-scale processing.
4. **Caching**: Implement caching mechanisms to reduce database load and provide fast access to frequently requested data.
5. **Efficient Data Structures**: Use priority queues or balanced trees to maintain the leaderboard efficiently.
6. **Fault Tolerance**: Implement retry mechanisms, dead-letter queues, and monitoring to handle failures gracefully.

This implementation leverages AWS SNS and SQS to handle notifications for millions of drivers efficiently, ensuring scalability and reliability.

#softwaredevelopment #softwareengineering #applicationdevelopment #systemdesign

---

24. Design an leaderboard for an internal app used by Swiggy or Zomato to rate their delivery drivers after each delivery.

System Components:

1. Order Service: Manages orders and publishes fulfillment events.

2. Rating Service: Updates driver ratings and influences related drivers.

3. Leaderboard Service: Maintains and serves the leaderboard.

4. Message Broker: Facilitates event-driven communication (e.g., Kafka).

5. Distributed Processing: Uses Spark for parallel processing of events.

6. Database: Stores driver ratings, related driver information, and leaderboard data. SQL aggregations can be used effectively to optimise various operations.

Learn more about how SQL aggregations work internally:

https://www.linkedin.com/feed/update/urn:li:activity:7220646194107011073/

7. Cache: Provides fast access to frequently requested data.

8. Worker Pool: Manages concurrent tasks efficiently (e.g., Java ExecutorService).

9. Fault Tolerance: Handles failures gracefully with retries and monitoring.

Detailed Design:

1. Order Service:

- Manages orders and publishes Kafka events with driver ID and rating upon order fulfillment.

```
public class OrderService {
    private KafkaProducer<String, String> kafkaProducer;

    public void orderFulfilled(Order order) {
        String event = new Gson().toJson(new OrderFulfilledEvent(order.getId(), order.getDriverId
(), order.getRating()));
        kafkaProducer.send(new ProducerRecord<>("order-fulfilled", event));
    }
}
```

Spark Job:

```
object RatingUpdateJob { ... }
```

2. Message Broker:

- Ensures reliable event delivery and supports horizontal scaling using Apache Kafka. It allows for horizontal scaling by distributing events across multiple partitions.

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
KafkaProducer<String, String> producer = new KafkaProducer<>(props);
```

3. Rating Service:

```
public class RatingService {
    private MessageBroker messageBroker;
    private Database database;
    private ExecutorService executorService = Executors.newFixedThreadPool(10);

    public void start() {
      messageBroker.subscribe(event -> executorService.submit(() -> processEvent(event)));
    }

    private void processEvent(OrderFulfilledEvent event) {
      int driverId = event.getDriverId();
      int rating = event.getRating();
      updateDriverRating(driverId, rating);
      updateRelatedDriversRating(driverId, rating);
      updateLeaderboard(driverId);
    }

    private void updateDriverRating(int driverId, int rating) {
      DriverRating driverRating = database.getDriverRating(driverId);
      int totalRatings = driverRating.getTotalRatings();
      double currentRating = driverRating.getRating();
      double newRating = (currentRating * totalRatings + rating) / (totalRatings + 1);
      driverRating.setRating(newRating);
      driverRating.incrementTotalRatings();
      database.updateDriverRating(driverRating);
    }

    private void updateRelatedDriversRating(int driverId, int rating) {
      List<Integer> relatedDrivers = findRelatedDrivers(driverId);
      int batchSize = 1000;
      List<List<Integer>> batches = createBatches(relatedDrivers, batchSize);
      for (List<Integer> batch : batches) {
        executorService.submit(() -> processBatch(batch, rating));
      }
    }
```

```
    private List<Integer> findRelatedDrivers(int driverId) {
      /* Filter from Database based on route, shift, location, performance metrics such as delivery
times, customer ratings, etc. */
      return null;
    }

    private List<List<Integer>> createBatches(List<Integer> relatedDrivers, int batchSize) {
      List<List<Integer>> batches = new ArrayList<>();
      for (int i = 0; i < relatedDrivers.size(); i += batchSize) {
        int end = Math.min(i + batchSize, relatedDrivers.size());
        batches.add(relatedDrivers.subList(i, end));
      }
      return batches;
    }

    private void processBatch(List<Integer> batch, int rating) {
      for (int relatedDriverId : batch) {
        DriverRating relatedDriverRating = database.getDriverRating(relatedDriverId);
        double influenceFactor = calculateInfluenceFactor(relatedDriverId);
        double newRating = relatedDriverRating.getRating() + influenceFactor * (rating - relatedDri
verRating.getRating());
        relatedDriverRating.setRating(newRating);
        database.updateDriverRating(relatedDriverRating);
      }
    }

    private double calculateInfluenceFactor(int relatedDriverId) {
      // Compute influence factor
      return 0.1; // Example: 10% influence
    }

    private void updateLeaderboard(int driverId) {
      LeaderboardService.updateDriverInLeaderboard(driverId);
    }
}
```

5. Database:

Note: We can use SQL aggregations for counting relevant ratings.

```
public class Database {
    private Map <Integer, DriverRating> driverRatings = new ConcurrentHashMap<>();
    private List <DriverRating> leaderboard = new ArrayList<>();
```

```
    private Map <Integer, List<Integer>> relatedDriversMap = new ConcurrentHashMap<>();
    public DriverRating getDriverRating(int driverId) {
        return driverRatings.get(driverId);
    }

    public void updateDriverRating(DriverRating driverRating) {
        driverRatings.put(driverRating.getDriverId(), driverRating);
    }

    public void updateLeaderboard(List<DriverRating> topDrivers) {
        leaderboard = topDrivers;
    }

    public List<Integer> getRelatedDrivers(int driverId) {
        return relatedDriversMap.get(driverId);
    }
}
```

6. Cache: (Redis): Note: DSU is not suitable for leaderboards. Instead let's use a max heap.

```
public class Cache {
    private Map<Integer, Double> leaderboardMap = new ConcurrentHashMap<>();
    private PriorityQueue<DriverRating> leaderboardQueue = new PriorityQueue<>(Comparator.
comparingDouble(DriverRating::getRating).reversed());
    private static final int LEADERBOARD_SIZE = 100;

    public void updateLeaderboard(int driverId, double rating) {
        leaderboardMap.put(driverId, rating);
        leaderboardQueue.add(new DriverRating(driverId, rating));
        if (leaderboardQueue.size() > LEADERBOARD_SIZE) {
            leaderboardQueue.poll();
        }
    }

    public List<DriverRating> getTopDrivers(int size) {
        return leaderboardQueue.stream().limit(size).collect(Collectors.toList());
    }
}
```

The system includes the Order Service for event generation, the Rating Service for processing and updating ratings, and the Leaderboard Service for managing and serving the leaderboard. By using a Message Broker, SQL Database Aggregations, Cache, and efficient data structures like Priority Queues, the system can scale to manage millions of drivers and maintain high performance.

Optimization strategies such as batch processing, asynchronous handling, and distributed systems further enhance the system's efficiency and reliability. This approach ensures that the leaderboard remains up-to-date and performs well even under heavy load.

#softwaredevelopment #softwareengineering #applicationdevelopment #systemdesign

---

25. `Design News Feed`

🧩 Applied Design Patterns

- **Observer Pattern**: Notifies users when new posts are published in topics they follow.
- **Singleton Pattern**: Ensures a single instance of the `NewsFeedService` throughout the application.
- **Factory Pattern**: Creates different types of posts (e.g., text, image, video) without specifying the exact class.

📦 Project Structure

```
NewsFeedSystem/
├── Main.java
├── models/
│   ├── User.java
│   ├── Post.java
│   ├── TextPost.java
│   ├── ImagePost.java
│   ├── VideoPost.java
│   └── Topic.java
├── services/
│   └── NewsFeedService.java
├── factories/
│   ├── PostFactory.java
│   └── PostType.java
└── observers/
    ├── Observer.java
    └── Subject.java
```

👨‍💻 Sample Code Snippets

Observer Interface

1. **Observer Interface**

`observers/Observer.java` Medium+2Wikipedia+2Wikipedia+2

```
package observers;

public interface Observer {
    void update(String topicName, String postTitle);
}
```

2. Subject Interface

observers/Subject.java

```
package observers;

public interface Subject {
    void registerObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers(String postTitle);
}
```

3. User Class (Observer Implementation)

models/User.java Wikipedia+2Hello Interview+2Medium+2

```
package models;

import observers.Observer;

public class User implements Observer {
    private String name;

    public User(String name) {
        this.name = name;
    }

    public void subscribeToTopic(Topic topic) {
        topic.registerObserver(this);
    }

    public void unsubscribeFromTopic(Topic topic) {
        topic.removeObserver(this);
    }

    @Override
    public void update(String topicName, String postTitle) {
```

```
        System.out.println(name + " received update from " + topicName + ": " + postTitle);
    }

    public String getName() {
        return name;
    }
}
```

4. Post Abstract Class and Subclasses

models/Post.java

```
package models;

public abstract class Post {
    protected String title;
    protected String content;

    public Post(String title, String content) {
        this.title = title;
        this.content = content;
    }

    public abstract void display();

    public String getTitle() {
        return title;
    }
}
```

models/TextPost.java

```
package models;

public class TextPost extends Post {
    public TextPost(String title, String content) {
        super(title, content);
    }

    @Override
    public void display() {
        System.out.println("Text Post: " + title + "\n" + content);
```

```
    }
}
```

models/ImagePost.java

```
package models;

public class ImagePost extends Post {
    public ImagePost(String title, String content) {
        super(title, content);
    }

    @Override
    public void display() {
        System.out.println("Image Post: " + title + "\n[Image]: " + content);
    }
}
```

models/VideoPost.java

```
package models;

public class VideoPost extends Post {
    public VideoPost(String title, String content) {
        super(title, content);
    }

    @Override
    public void display() {
        System.out.println("Video Post: " + title + "\n[Video]: " + content);
    }
}
```

5. PostType Enum

factories/PostType.java

```
package factories;

public enum PostType {
    TEXT,
    IMAGE,
    VIDEO
}
```

6. PostFactory Class

factories/PostFactory.java

```
package factories;

import models.*;

public class PostFactory {
    public static Post createPost(PostType type, String title, String content) {
        switch (type) {
            case TEXT:
                return new TextPost(title, content);
            case IMAGE:
                return new ImagePost(title, content);
            case VIDEO:
                return new VideoPost(title, content);
            default:
                throw new IllegalArgumentException("Unknown PostType");
        }
    }
}
```

7. Topic Class (Subject Implementation)

models/Topic.java

```
package models;

import observers.Observer;
import observers.Subject;

import java.util.ArrayList;
import java.util.List;

public class Topic implements Subject {
    private String name;
    private List<Observer> observers;
    private List<Post> posts;

    public Topic(String name) {
        this.name = name;
        this.observers = new ArrayList<>();
        this.posts = new ArrayList<>();
    }
```

```
    public void addPost(Post post) {
        posts.add(post);
        notifyObservers(post.getTitle());
    }

    @Override
    public void registerObserver(Observer observer) {
        observers.add(observer);
    }

    @Override
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers(String postTitle) {
        for (Observer observer : observers) {
            observer.update(name, postTitle);
        }
    }

    public String getName() {
        return name;
    }

    public List<Post> getPosts() {
        return posts;
    }
}
```

8. NewsFeedService Class (Singleton Pattern)

services/NewsFeedService.ja

```
package services;

import models.Topic;

import java.util.HashMap;
import java.util.Map;

public class NewsFeedService {
```

```
    private static NewsFeedService instance;
    private Map<String, Topic> topics;

    private NewsFeedService() {
        topics = new HashMap<>();
    }

    public static NewsFeedService getInstance() {
        if (instance == null) {
            instance = new NewsFeedService();
        }
        return instance;
    }

    public Topic createTopic(String name) {
        Topic topic = new Topic(name);
        topics.put(name, topic);
        return topic;
    }

    public Topic getTopic(String name) {
        return topics.get(name);
    }
}
```

9. Main Class to Simulate News Feed

Main.java

```
import factories.PostFactory;
import factories.PostType;
import models.*;
import services.NewsFeedService;

public class Main {
    public static void main(String[] args) {
        NewsFeedService service = NewsFeedService.getInstance();

        // Create topics
        Topic tech = service.createTopic("Technology");
        Topic sports = service.createTopic("Sports");

        // Create users
        User alice = new User("Alice");
```

```
        User bob = new User("Bob");

        // Users subscribe to topics
        alice.subscribeToTopic(tech);
        bob.subscribeToTopic(tech);
        bob.subscribeToTopic(sports);

        // Create posts
        Post post1 = PostFactory.createPost(PostType.TEXT,
        "New Java Release", "Java 17 has been released with new features.");
        Post post2 = PostFactory.createPost(PostType.IMAGE,
        "Championship Highlights", "Image content here.");
        Post post3 = PostFactory.createPost(PostType.VIDEO,
        "Tech Talk", "Video content here.");

        // Add posts to topics
        tech.addPost(post1);
        sports.addPost(post2);
        tech.addPost(post3);

        // Display posts in Technology topic
        System.out.println("\nPosts in Technology:");
        for (Post post : tech.getPosts()) {
            post.display();
            System.out.println();
        }

        // Display posts in Sports topic
        System.out.println("Posts in Sports:");
        for (Post post : sports.getPosts()) {
            post.display();
            System.out.println();
        }
    }
}
```

🖥️ Sample Output

```
Alice received update from Technology: New Java Release
Bob received update from Technology: New Java Release
Bob received update from Sports: Championship Highlights
Alice received update from Technology: Tech Talk
```

```
Bob received update from Technology: Tech Talk

Posts in Technology:
Text Post: New Java Release
Java 17 has been released with new features.

Video Post: Tech Talk
[Video]: Video content here.

Posts in Sports:
Image Post: Championship Highlights
[Image]: Image content here.
```

---

26. Design Whatsapp

🧩 Applied Design Patterns

- **Observer Pattern**: Notifies users when new messages are received.
- **Singleton Pattern**: Ensures a single instance of the `ChatService` throughout the application.
- **Factory Pattern**: Creates different types of messages (e.g., text, image) without specifying the exact class.
- **Decorator Pattern**: Adds additional functionalities to messages, such as encryption.

📦 Project Structure

```
WhatsAppClone/
├── Main.java
├── models/
│   ├── User.java
│   ├── Message.java
│   ├── TextMessage.java
│   ├── ImageMessage.java
│   └── EncryptedMessage.java
├── services/
│   └── ChatService.java
├── factories/
│   ├── MessageFactory.java
```

```
│   └── MessageType.java
├── observers/
│   ├── Observer.java
│   └── Subject.java
└── decorators/
    └── MessageDecorator.java
```

👨‍💻 Sample Code Snippets

Observer Interface

1. **Observer Interface**

observers/Observer.java

```
package observers;

public interface Observer {
    void update(String message);
}
```

2. Subject Interface

observers/Subject.java

```
package observers;

public interface Subject {
    void registerObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers(String message);
}
```

3. User Class (Observer Implementation)

models/User.java

```
package models;

import observers.Observer;

public class User implements Observer {
    private String name;

    public User(String name) {
```

```
        this.name = name;
    }

    public void sendMessage(User receiver, Message message) {
        receiver.receiveMessage(message);
    }

    public void receiveMessage(Message message) {
        System.out.println(name + " received: " + message.getContent());
    }

    @Override
    public void update(String message) {
        System.out.println(name + " notification: " + message);
    }

    public String getName() {
        return name;
    }
}
```

4. Message Abstract Class and Subclasses

models/Message.java

```
package models;

public abstract class Message {
    protected String content;

    public Message(String content) {
        this.content = content;
    }

    public abstract String getContent();
}
```

models/TextMessage.java

```
package models;

public class TextMessage extends Message {
    public TextMessage(String content) {
        super(content);
```

```
    }

    @Override
    public String getContent() {
        return content;
    }
}
```

models/ImageMessage.java

```
package models;

public class ImageMessage extends Message {
    public ImageMessage(String content) {
        super(content);
    }

    @Override
    public String getContent() {
        return "[Image]: " + content;
    }
}
```

5. MessageType Enum

factories/MessageType.java ByteByteGo+2Hello Interview+2Wikipedia+2

```
package factories;

public enum MessageType {
    TEXT,
    IMAGE
}
```

6. MessageFactory Class

factories/MessageFactory.java

```
package factories;

import models.*;

public class MessageFactory {
    public static Message createMessage(MessageType type, String content) {
```

```
        switch (type) {
            case TEXT:
                return new TextMessage(content);
            case IMAGE:
                return new ImageMessage(content);
            default:
                throw new IllegalArgumentException("Unknown MessageType");
        }
    }
}
```

7. MessageDecorator Class (Decorator Pattern)

decorators/MessageDecorator.java

```
package decorators;

import models.Message;

public abstract class MessageDecorator extends Message {
    protected Message message;

    public MessageDecorator(Message message) {
        super(message.getContent());
        this.message = message;
    }

    public abstract String getContent();
}
```

decorators/EncryptedMessage.java

```
package decorators;

import models.Message;

public class EncryptedMessage extends MessageDecorator {
    public EncryptedMessage(Message message) {
        super(message);
    }

    @Override
    public String getContent() {
        return encrypt(message.getContent());
```

```
    }

    private String encrypt(String content) {
        // Simple encryption logic (for illustration)
        return new StringBuilder(content).reverse().toString();
    }
}
```

8. ChatService Class (Singleton Pattern)

services/ChatService.java

```
package services;

import models.User;

public class ChatService {
    private static ChatService instance;

    private ChatService() {}

    public static ChatService getInstance() {
        if (instance == null) {
            instance = new ChatService();
        }
        return instance;
    }

    public void sendMessage(User sender, User receiver, String content) {
        sender.sendMessage(receiver, new models.TextMessage(content));
    }
}
```

9. Main Class to Simulate Chat

Main.java Medium+10DEV Community+10Wikipedia+10

```
import factories.MessageFactory;
import factories.MessageType;
import models.*;
import services.ChatService;
import decorators.EncryptedMessage;

public class Main {
```

```
    public static void main(String[] args) {
        User alice = new User("Alice");
        User bob = new User("Bob");

        Message textMessage = MessageFactory.createMessage(MessageType.TEXT, "Hello, Bo
b!");
        Message imageMessage = MessageFactory.createMessage(MessageType.IMAGE, "photo.j
pg");

        // Encrypt the text message
        Message encryptedText = new EncryptedMessage(textMessage);

        // Send messages
        alice.sendMessage(bob, encryptedText);
        alice.sendMessage(bob, imageMessage);
    }
}
```

🖥️ Sample Output

```
Bob received: !boB ,olleH
Bob received: [Image]: photo.jpg
```

27. `Design Instagram/Twitter News Feed`

🧰 Applied Design Patterns

1. **Observer Pattern**: To notify followers when a user posts new content.
2. **Factory Pattern**: To create different types of posts (text, image, video) without specifying the exact class.
3. **Decorator Pattern**: To add additional functionalities to posts, such as tagging or filtering.
4. **Strategy Pattern**: To implement various feed ranking algorithms (e.g., chronological, relevance-based).
5. **Singleton Pattern**: To ensure a single instance of services like `FeedService` throughout the application.

📦 Project Structure

```
NewsFeedApp/
├── Main.java
├── models/
│   ├── User.java
│   ├── Post.java
│   ├── TextPost.java
│   ├── ImagePost.java
│   └── VideoPost.java
├── services/
│   ├── FeedService.java
│   └── NotificationService.java
├── factories/
│   ├── PostFactory.java
│   └── PostType.java
├── decorators/
│   ├── PostDecorator.java
│   └── TaggedPost.java
├── strategies/
│   ├── FeedStrategy.java
│   ├── ChronologicalStrategy.java
│   └── RelevanceStrategy.java
└── observers/
    ├── Observer.java
    └── Subject.java
```

👨‍💻 Sample Code Snippets

1. Observer Interface

`observers/Observer.java` The New Yorker

```
package observers;

public interface Observer {
    void update(Post post);
}
```

2. Subject Interface

`observers/Subject.java` Wikipedia+1Hello Interview+1

```
package observers;

public interface Subject {
    void registerObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers(Post post);
}
```

3. User Class (Subject Implementation)

models/User.java The New Yorker

```
package models;

import observers.Observer;
import observers.Subject;
import java.util.ArrayList;
import java.util.List;

public class User implements Subject, Observer {
    private String name;
    private List<Observer> followers;
    private List<Post> feed;

    public User(String name) {
        this.name = name;
        this.followers = new ArrayList<>();
        this.feed = new ArrayList<>();
    }

    public void postContent(Post post) {
        System.out.println(name + " posted: " + post.getContent());
        notifyObservers(post);
    }

    @Override
    public void registerObserver(Observer observer) {
        followers.add(observer);
    }

    @Override
    public void removeObserver(Observer observer) {
        followers.remove(observer);
```

```
    }

    @Override
    public void notifyObservers(Post post) {
        for (Observer follower : followers) {
            follower.update(post);
        }
    }

    @Override
    public void update(Post post) {
        feed.add(post);
        System.out.println(name + " received a new post: " + post.getContent());
    }

    public String getName() {
        return name;
    }
}
```

4. Post Abstract Class and Subclasses

models/Post.java

```
package models;

public abstract class Post {
    protected String content;

    public Post(String content) {
        this.content = content;
    }

    public abstract String getContent();
}
```

models/TextPost.java

```
package models;

public class TextPost extends Post {
    public TextPost(String content) {
        super(content);
    }
```

```
    @Override
    public String getContent() {
        return content;
    }
}
```

models/ImagePost.java

```
package models;

public class ImagePost extends Post {
    public ImagePost(String content) {
        super(content);
    }

    @Override
    public String getContent() {
        return "[Image]: " + content;
    }
}
```

5. PostType Enum

factories/PostType.java

```
package factories;

public enum PostType {
    TEXT,
    IMAGE,
    VIDEO
}
```

6. PostFactory Class

factories/PostFactory.java

```
package factories;

import models.*;

public class PostFactory {
    public static Post createPost(PostType type, String content) {
        switch (type) {
```

```
            case TEXT:
                return new TextPost(content);
            case IMAGE:
                return new ImagePost(content);
            case VIDEO:
                return new VideoPost(content);
            default:
                throw new IllegalArgumentException("Unknown PostType");
        }
    }
}
```

7. FeedStrategy Interface and Implementations

strategies/FeedStrategy.java

```
package strategies;

import models.Post;
import java.util.List;

public interface FeedStrategy {
    List<Post> sortFeed(List<Post> posts);
}
```

strategies/ChronologicalStrategy.java

```
package strategies;

import models.Post;
import java.util.List;
import java.util.Collections;

public class ChronologicalStrategy implements FeedStrategy {
    @Override
    public List<Post> sortFeed(List<Post> posts) {
        // Assuming posts have timestamps, sort accordingly
        Collections.sort(posts, (a, b) -> a.getTimestamp().compareTo(b.getTimestamp()));
        return posts;
    }
}
```

8. FeedService Class (Singleton Pattern)

services/FeedService.java

```
package services;

import models.Post;
import models.User;
import strategies.FeedStrategy;
import java.util.List;

public class FeedService {
    private static FeedService instance;
    private FeedStrategy strategy;

    private FeedService() {}

    public static FeedService getInstance() {
        if (instance == null) {
            instance = new FeedService();
        }
        return instance;
    }

    public void setStrategy(FeedStrategy strategy) {
        this.strategy = strategy;
    }

    public List<Post> getFeed(User user) {
        List<Post> feed = user.getFeed();
        return strategy.sortFeed(feed);
    }
}
```

9. Main Class to Simulate News Feed

Main.java Wikipedia+12System Design Notes+12Medium+12

```
import models.*;
import factories.*;
import strategies.*;
import services.*;

public class Main {
    public static void main(String[] args) {
        User alice = new User("Alice");
        User bob = new User("Bob");
```

```
        // Bob follows Alice
        alice.registerObserver(bob);

        // Alice posts content
        Post post1 = PostFactory.createPost(PostType.TEXT, "Hello World!");
        alice.postContent(post1);

        // Set feed strategy
        FeedService feedService = FeedService.getInstance();
        feedService.setStrategy(new ChronologicalStrategy());

        // Get Bob's feed
        feedService.getFeed(bob).forEach(post -> System.out.println("Bob's feed: " + post.getCon
tent()));
    }
}
```

🖥️ Sample Output

```
Alice posted: Hello World!
Bob received a new post: Hello World!
Bob's feed: Hello World!
```

28. Design Search Autocomplete

✅ Problem Statement

Build a system that can:

- Store a list of searchable terms.
- Return autocomplete suggestions based on a given prefix.
- Easily allow extending the algorithm (e.g., switch from trie-based to DB-based search).

✅ Design Patterns Used

Pattern	Purpose
Strategy	Choose between different autocomplete algorithms (e.g., Trie, DB, Fuzzy).

Singleton	Maintain a single instance of the autocomplete engine.
Factory	Instantiate the right autocomplete strategy.
Observer	(Optional) Notify listeners on new word addition or search query.

✅ Key Classes and Interfaces

1. `AutocompleteStrategy` – Interface for different search algorithms.
2. `TrieAutocomplete` – Concrete strategy using a Trie.
3. `AutocompleteFactory` – Chooses strategy implementation.
4. `AutocompleteEngine` – Singleton service that exposes the main API.
5. `Main` – Driver to demonstrate working.

✅ UML Class Diagram (Textual)

```
+------+
| Main |
+------+
```

✅ Java Implementation

1. AutocompleteStrategy.java

```
import java.util.List;

public interface AutocompleteStrategy {
    void addWord(String word);
    List<String> suggest(String prefix);
}
```

2. TrieAutocomplete.java

```
import java.util.*;

public class TrieAutocomplete implements AutocompleteStrategy {

    private static class TrieNode {
        Map<Character, TrieNode> children = new HashMap<>();
        boolean isWord = false;
    }

    private final TrieNode root = new TrieNode();

    @Override
    public void addWord(String word) {
        TrieNode curr = root;
        for (char ch : word.toLowerCase().toCharArray()) {
            curr = curr.children.computeIfAbsent(ch, c -> new TrieNode());
        }
        curr.isWord = true;
    }

    @Override
    public List<String> suggest(String prefix) {
        List<String> results = new ArrayList<>();
```

```
        TrieNode curr = root;

        for (char ch : prefix.toLowerCase().toCharArray()) {
            if (!curr.children.containsKey(ch)) {
                return results;
            }
            curr = curr.children.get(ch);
        }

        dfs(prefix, curr, results);
        return results;
    }

    private void dfs(String prefix, TrieNode node, List<String> results) {
        if (node.isWord) {
            results.add(prefix);
        }
        for (Map.Entry<Character, TrieNode> entry : node.children.entrySet()) {
            dfs(prefix + entry.getKey(), entry.getValue(), results);
        }
    }
}
```

3. AutocompleteFactory.java

```
public class AutocompleteFactory {
    public static AutocompleteStrategy getStrategy(String type) {
        switch (type.toLowerCase()) {
            case "trie":
            default:
                return new TrieAutocomplete();
        }
    }
}
```

4. AutocompleteEngine.java

```
import java.util.List;

public class AutocompleteEngine {
    private static AutocompleteEngine instance;
    private final AutocompleteStrategy strategy;
```

```
    private AutocompleteEngine(AutocompleteStrategy strategy) {
        this.strategy = strategy;
    }

    public static AutocompleteEngine getInstance(String type) {
        if (instance == null) {
            instance = new AutocompleteEngine(AutocompleteFactory.getStrategy(type));
        }
        return instance;
    }

    public void addWord(String word) {
        strategy.addWord(word);
    }

    public List<String> suggest(String prefix) {
        return strategy.suggest(prefix);
    }
}
```

5. Main.java

```
public class Main {
    public static void main(String[] args) {
        AutocompleteEngine engine = AutocompleteEngine.getInstance("trie");

        engine.addWord("dog");
        engine.addWord("doll");
        engine.addWord("dolphin");
        engine.addWord("cat");
        engine.addWord("camel");
        engine.addWord("cap");

        System.out.println("Suggestions for 'do': " + engine.suggest("do"));
        System.out.println("Suggestions for 'ca': " + engine.suggest("ca"));
        System.out.println("Suggestions for 'z': " + engine.suggest("z"));
    }
}
```

✅ Sample Output

```
Suggestions for 'do': [dog, doll, dolphin]
Suggestions for 'ca': [cat, camel, cap]
Suggestions for 'z': []
```

29. `Design Zomato Search`

✅ Features to Support

1. **Search Restaurants by Name / Cuisine / Dish**
2. **Filters**:
   - Rating (min threshold)
   - Price (min-max)
   - Location (Pincode, Area)
3. **Sorting**:
   - By Rating (desc)
   - By Price (asc or desc)

✅ Key Design Patterns Used

Pattern	Use Case
Strategy	Choose different search/filter strategies
Builder	Construct complex `SearchRequest` objects
Factory	Create search engine (Trie-based, DB-based, etc.)
Singleton	Maintain a single instance of the search engine
Observer (opt)	For logging/analytics of user searches

✅ Core Components

Domain Models:

- `Restaurant`
- `SearchRequest`
- `SearchResult`

Search Engine Layer:

- `SearchStrategy` interface

- `InMemorySearchEngine` – uses filters and search logic
- `SearchEngineFactory` – creates appropriate strategy

API Layer:

- `ZomatoSearchService` – Singleton class exposing APIs

✅ Class Diagram (Textual UML)

```
+--------------------+
| Restaurant         |
+--------------------+
| name               |
| cuisines           |
| menuItems          |
| rating             |
| location (pincode) |
| priceForTwo        |
+--------------------+

+--------------------+
| SearchRequest      |
+--------------------+
| keyword            |
| pincode            |
| minRating          |
| priceRange         |
| sortBy             |
+--------------------+

+--------------------+
| SearchResult       |
+--------------------+
| List<Restaurant>   |
+--------------------+

+---------------------+
| SearchStrategy      | (Interface)
+---------------------+
| +search(request)    |
+---------------------+

+--------------------------+
```

```
| InMemorySearchStrategy    |
+---------------------------+
| +search(request)          |
+---------------------------+

+---------------------+
| SearchEngineFactory |
+---------------------+
| +getEngine(type)    |
+---------------------+

+-----------------------+
| ZomatoSearchService   | ← Singleton
+-----------------------+
| +search(SearchRequest) |
| +addRestaurant(...)   |
+-----------------------+
```

✅ Code Implementation

1. Restaurant.java

```
import java.util.*;

public class Restaurant {
    private String name;
    private Set<String> cuisines;
    private List<String> menuItems;
    private double rating;
    private String locationPincode;
    private double priceForTwo;

    public Restaurant(String name, Set<String> cuisines, List<String> menuItems, double rating,
String locationPincode, double priceForTwo) {
        this.name = name;
        this.cuisines = cuisines;
        this.menuItems = menuItems;
        this.rating = rating;
        this.locationPincode = locationPincode;
        this.priceForTwo = priceForTwo;
    }
```

```
    public String getName() { return name; }
    public Set<String> getCuisines() { return cuisines; }
    public List<String> getMenuItems() { return menuItems; }
    public double getRating() { return rating; }
    public String getLocationPincode() { return locationPincode; }
    public double getPriceForTwo() { return priceForTwo; }

    @Override
    public String toString() {
        return name + " (" + rating + ") - ₹" + priceForTwo + " - " + locationPincode;
    }
}
```

2. SearchRequest.java

```
public class SearchRequest {
    String keyword;
    String pincode;
    double minRating;
    double minPrice;
    double maxPrice;
    String sortBy; // "rating", "price"

    // Use builder pattern
    public static class Builder {
        private final SearchRequest request = new SearchRequest();

        public Builder keyword(String keyword) {
            request.keyword = keyword.toLowerCase();
            return this;
        }

        public Builder pincode(String pincode) {
            request.pincode = pincode;
            return this;
        }

        public Builder minRating(double rating) {
            request.minRating = rating;
            return this;
        }
```

```
        public Builder priceRange(double min, double max) {
            request.minPrice = min;
            request.maxPrice = max;
            return this;
        }

        public Builder sortBy(String sortBy) {
            request.sortBy = sortBy;
            return this;
        }

        public SearchRequest build() {
            return request;
        }
    }
}
```

3. SearchStrategy.java

```
import java.util.List;

public interface SearchStrategy {
    List<Restaurant> search(SearchRequest request);
}
```

4. InMemorySearchStrategy.java

```
import java.util.*;
import java.util.stream.Collectors;

public class InMemorySearchStrategy implements SearchStrategy {

    private final List<Restaurant> database;

    public InMemorySearchStrategy(List<Restaurant> database) {
        this.database = database;
    }

    @Override
    public List<Restaurant> search(SearchRequest req) {
        return database.stream()
```

```
            .filter(r -> r.getLocationPincode().equals(req.pincode))
            .filter(r -> r.getRating() >= req.minRating)
            .filter(r -> r.getPriceForTwo() >= req.minPrice && r.getPriceForTwo() <= req.maxPrice)
            .filter(r -> {
                String keyword = req.keyword.toLowerCase();
                return r.getName().toLowerCase().contains(keyword)
                    || r.getMenuItems().stream().anyMatch(item -> item.toLowerCase().contains(keywor
d))
                    || r.getCuisines().stream().anyMatch(c -> c.toLowerCase().contains(keyword));
            })
            .sorted((r1, r2) -> {
                if (req.sortBy.equals("rating")) {
                    return Double.compare(r2.getRating(), r1.getRating());
                } else {
                    return Double.compare(r1.getPriceForTwo(), r2.getPriceForTwo());
                }
            })
            .collect(Collectors.toList());
    }
}
```

5. SearchEngineFactory.java

```
import java.util.List;

public class SearchEngineFactory {
    public static SearchStrategy getEngine(String type, List<Restaurant> data) {
        switch (type.toLowerCase()) {
            case "inmemory":
            default:
                return new InMemorySearchStrategy(data);
        }
    }
}
```

6. ZomatoSearchService.java

```
import java.util.*;

public class ZomatoSearchService {
    private static ZomatoSearchService instance;
    private final List<Restaurant> data = new ArrayList<>();
```

```
    private final SearchStrategy strategy;

    private ZomatoSearchService() {
        this.strategy = SearchEngineFactory.getEngine("inmemory", data);
    }

    public static synchronized ZomatoSearchService getInstance() {
        if (instance == null) {
            instance = new ZomatoSearchService();
        }
        return instance;
    }

    public void addRestaurant(Restaurant r) {
        data.add(r);
    }

    public List<Restaurant> search(SearchRequest request) {
        return strategy.search(request);
    }
}
```

7. Main.java

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        ZomatoSearchService service = ZomatoSearchService.getInstance();

        service.addRestaurant(new Restaurant("Tandoori Palace", Set.of("Indian", "North Indian"),
                List.of("Paneer Tikka", "Butter Chicken"), 4.5, "560102", 500));

        service.addRestaurant(new Restaurant("Burger Town", Set.of("American", "Fast Food"),
                List.of("Cheeseburger", "Fries"), 4.2, "560102", 300));

        service.addRestaurant(new Restaurant("Sushi Express", Set.of("Japanese"),
                List.of("Sushi", "Ramen"), 4.8, "560102", 800));

        SearchRequest req = new SearchRequest.Builder()
                .keyword("sushi")
                .pincode("560102")
```

```
            .minRating(4.0)
            .priceRange(200, 1000)
            .sortBy("rating")
            .build();

        List<Restaurant> results = service.search(req);

        System.out.println("Search Results:");
        for (Restaurant r : results) {
            System.out.println(r);
        }
    }
}
```

✅ Output (Example)

```
Search Results:
Sushi Express (4.8) - ₹800.0 - 560102
```

30. `Car Rental System`

🚗 Car Rental System Design in Java

✅ Key Functional Requirements

- **Vehicle Management**: Add, update, and remove vehicles.
- **User Management**: Register and manage customers and staff.
- **Booking System**: Search for available vehicles, make reservations, and handle cancellations.
- **Rental Process**: Manage vehicle pickups, returns, and track ongoing rentals.
- **Billing**: Calculate rental costs, apply discounts, and process payments.
- **Reporting**: Generate reports on rentals, revenues, and vehicle utilization.

🧱 Core Components and Design Patterns

1. Vehicle Management

- **Classes**: `Vehicle`, `Car`, `Truck`, `SUV`, etc.

- **Design Pattern**: *Factory Pattern* to create different vehicle types.

```
public abstract class Vehicle {
    private String id;
    private String model;
    private String licensePlate;
    private VehicleStatus status;
    // Getters and setters
}

public class Car extends Vehicle {
    // Specific attributes for Car
}

public class VehicleFactory {
    public static Vehicle createVehicle(String type, String model, String licensePlate) {
        switch (type) {
            case "Car":
                return new Car(model, licensePlate);
            // Add cases for other vehicle types
            default:
                throw new IllegalArgumentException("Invalid vehicle type");
        }
    }
}
```

2. User Management

- **Classes**: `User`, `Customer`, `Staff`.
- **Design Pattern**: *Builder Pattern* for constructing complex user objects.

```
public class User {
    private String id;
    private String name;
    private String email;
    // Getters and setters
}

public class UserBuilder {
    private String id;
    private String name;
    private String email;
```

```
    public UserBuilder setId(String id) {
        this.id = id;
        return this;
    }

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

    public UserBuilder setEmail(String email) {
        this.email = email;
        return this;
    }

    public User build() {
        return new User(id, name, email);
    }
}
```

3. Booking System

- **Classes**: `Booking`, `BookingService`.
- **Design Pattern**: *Strategy Pattern* to apply different pricing strategies.

```
public interface PricingStrategy {
    double calculatePrice(Vehicle vehicle, int days);
}

public class StandardPricing implements PricingStrategy {
    public double calculatePrice(Vehicle vehicle, int days) {
        return vehicle.getDailyRate() * days;
    }
}

public class BookingService {
    private PricingStrategy pricingStrategy;

    public BookingService(PricingStrategy pricingStrategy) {
        this.pricingStrategy = pricingStrategy;
    }

    public Booking createBooking(User user, Vehicle vehicle, int days) {
```

```
        double price = pricingStrategy.calculatePrice(vehicle, days);
        // Create and return booking
    }
}
```

4. Rental Process

- **Classes**: `Rental`, `RentalService`.
- **Design Pattern**: *State Pattern* to manage rental states (e.g., Booked, InProgress, Completed).

```
public interface RentalState {
    void proceed(Rental rental);
}

public class BookedState implements RentalState {
    public void proceed(Rental rental) {
        rental.setState(new InProgressState());
    }
}

public class Rental {
    private RentalState state;

    public void proceed() {
        state.proceed(this);
    }

    public void setState(RentalState state) {
        this.state = state;
    }
}
```

5. Billing

- **Classes**: `Invoice`, `PaymentService`.
- **Design Pattern**: *Decorator Pattern* to apply additional charges or discounts.

```
public interface Invoice {
    double getTotal();
}

public class BasicInvoice implements Invoice {
    private double amount;
```

```
    public BasicInvoice(double amount) {
        this.amount = amount;
    }

    public double getTotal() {
        return amount;
    }
}

public class DiscountedInvoice implements Invoice {
    private Invoice invoice;
    private double discount;

    public DiscountedInvoice(Invoice invoice, double discount) {
        this.invoice = invoice;
        this.discount = discount;
    }

    public double getTotal() {
        return invoice.getTotal() - discount;
    }
}
```

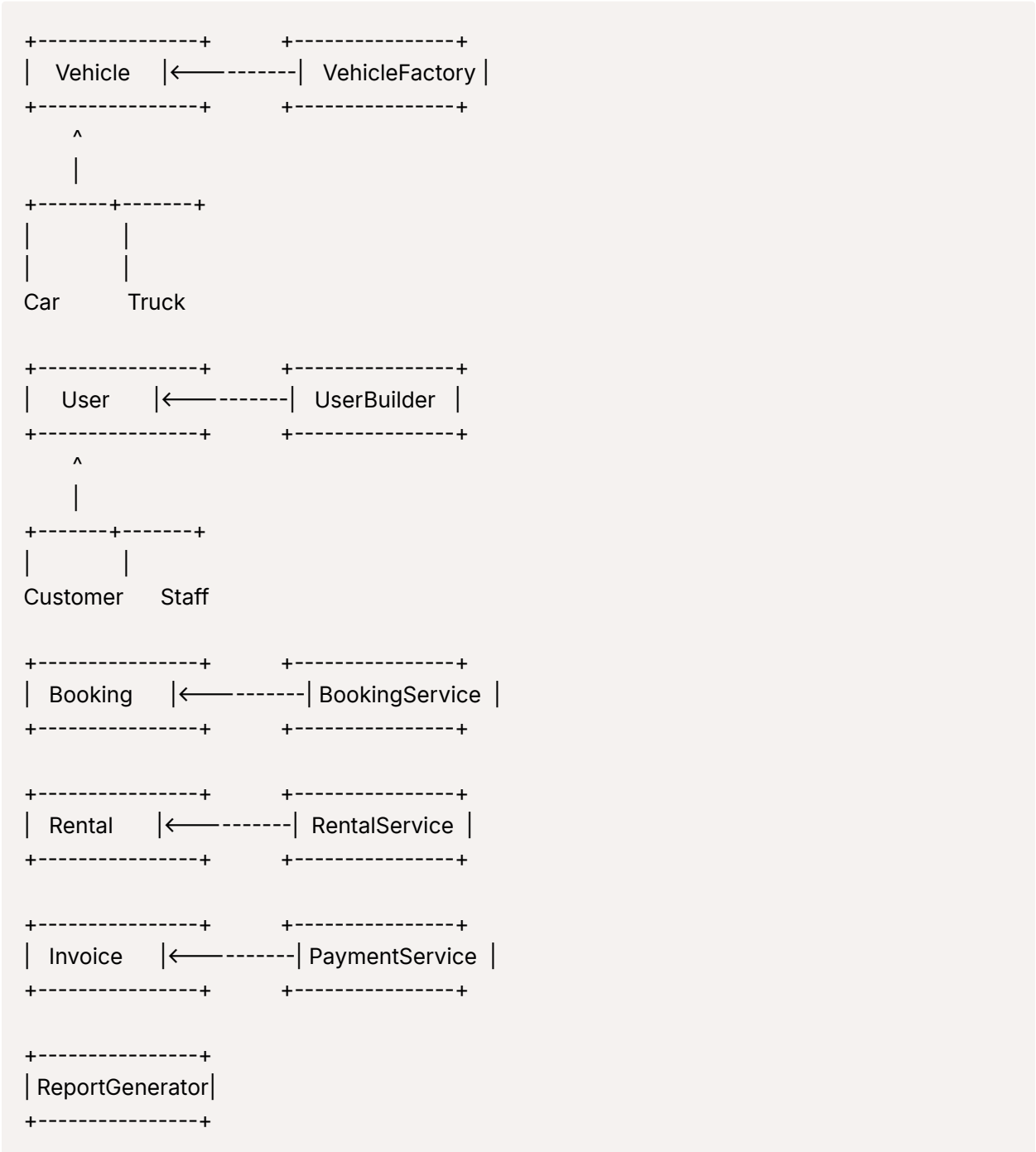
6. Reporting

- **Classes**: `ReportGenerator`.
- **Design Pattern**: *Singleton Pattern* to ensure a single instance of the report generator.

```
public class ReportGenerator {
    private static ReportGenerator instance;

    private ReportGenerator() {}

    public static ReportGenerator getInstance() {
        if (instance == null) {
            instance = new ReportGenerator();
        }
        return instance;
    }

    public void generateReport() {
        // Generate report logic
```

```
    }
}
```

🗂️ Class Diagram Overview



🧪 Sample Usage

```
public class Main {
    public static void main(String[] args) {
        Vehicle car = VehicleFactory.createVehicle("Car", "Toyota Camry", "ABC123");

        User customer = new UserBuilder()
                .setId("U001")
                .setName("John Doe")
                .setEmail("john@example.com")
                .build();

        PricingStrategy pricingStrategy = new StandardPricing();
        BookingService bookingService = new BookingService(pricingStrategy);
        Booking booking = bookingService.createBooking(customer, car, 5);

        Rental rental = new Rental();
        rental.setState(new BookedState());
        rental.proceed(); // Moves to InProgressState

        Invoice invoice = new BasicInvoice(500);
        invoice = new DiscountedInvoice(invoice, 50); // Apply discount

        ReportGenerator reportGenerator = ReportGenerator.getInstance();
        reportGenerator.generateReport();
    }
}
```

---

31. Hotel Management

🏨 Functional Requirements

1. **Room Management**: Add, update, and remove rooms; track room availability.
2. **Guest Management**: Register guests, manage check-ins and check-outs.
3. **Booking System**: Search for available rooms, make reservations, handle cancellations.
4. **Billing**: Calculate charges, apply discounts, generate invoices.
5. **Reporting**: Generate reports on occupancy, revenue, and guest history.

🧰 Core Components and Design Patterns

1. Room Management

- **Classes**: `Room`, `StandardRoom`, `DeluxeRoom`, `SuiteRoom`.
- **Design Pattern**: *Factory Pattern* to create different room types.

```
public abstract class Room {
    private String roomNumber;
    private double rate;
    private boolean isAvailable;

    // Getters and setters
}

public class StandardRoom extends Room {
    // Specific attributes for StandardRoom
}

public class RoomFactory {
    public static Room createRoom(String type, String roomNumber) {
        switch (type) {
            case "Standard":
                return new StandardRoom(roomNumber);
            // Add cases for other room types
            default:
                throw new IllegalArgumentException("Invalid room type");
        }
    }
}
```

2. Guest Management

- **Classes**: `Guest`, `GuestBuilder`.
- **Design Pattern**: *Builder Pattern* for constructing complex guest objects.

```
public class Guest {
    private String id;
    private String name;
    private String email;
    // Getters and setters
}
```

```
public class GuestBuilder {
    private String id;
    private String name;
    private String email;

    public GuestBuilder setId(String id) {
        this.id = id;
        return this;
    }

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

    public GuestBuilder setEmail(String email) {
        this.email = email;
        return this;
    }

    public Guest build() {
        return new Guest(id, name, email);
    }
}
```

3. Booking System

- **Classes**: `Booking`, `BookingService`.
- **Design Pattern**: *Strategy Pattern* to apply different pricing strategies.

```
public interface PricingStrategy {
    double calculatePrice(Room room, int nights);
}

public class StandardPricing implements PricingStrategy {
    public double calculatePrice(Room room, int nights) {
        return room.getRate() * nights;
    }
}

public class BookingService {
    private PricingStrategy pricingStrategy;
```

```
    public BookingService(PricingStrategy pricingStrategy) {
        this.pricingStrategy = pricingStrategy;
    }

    public Booking createBooking(Guest guest, Room room, int nights) {
        double price = pricingStrategy.calculatePrice(room, nights);
        // Create and return booking
    }
}
```

4. Billing

- **Classes**: `Invoice`, `PaymentService`.
- **Design Pattern**: *Decorator Pattern* to apply additional charges or discounts.

```
public interface Invoice {
    double getTotal();
}

public class BasicInvoice implements Invoice {
    private double amount;

    public BasicInvoice(double amount) {
        this.amount = amount;
    }

    public double getTotal() {
        return amount;
    }
}

public class DiscountedInvoice implements Invoice {
    private Invoice invoice;
    private double discount;

    public DiscountedInvoice(Invoice invoice, double discount) {
        this.invoice = invoice;
        this.discount = discount;
    }

    public double getTotal() {
        return invoice.getTotal() - discount;
    }
```

```
}
```

5. Reporting

- **Classes**: `ReportGenerator`.
- **Design Pattern**: *Singleton Pattern* to ensure a single instance of the report generator.

```
public class ReportGenerator {
    private static ReportGenerator instance;

    private ReportGenerator() {}

    public static ReportGenerator getInstance() {
        if (instance == null) {
            instance = new ReportGenerator();
        }
        return instance;
    }

    public void generateReport() {
        // Generate report logic
    }
}
```

🗂️ Class Diagram Overview

```
+----------------+       +----------------+
|    Room      |<-------|  RoomFactory   |
+----------------+       +----------------+
     ^
     |
+-------+-------+
|               |
StandardRoom  DeluxeRoom

+----------------+       +----------------+
|    Guest     |<-------| GuestBuilder   |
+----------------+       +----------------+

+----------------+       +----------------+
|  Booking     |<-------| BookingService  |
```

```
+----------------+       +----------------+

+----------------+       +----------------+
|  Invoice      |<-------| PaymentService  |
+----------------+       +----------------+

+----------------+
| ReportGenerator|
+----------------+
```

🧪 Sample Usage

```
public class Main {
    public static void main(String[] args) {
        Room room = RoomFactory.createRoom("Standard", "101");

        Guest guest = new GuestBuilder()
                .setId("G001")
                .setName("Alice Smith")
                .setEmail("alice@example.com")
                .build();

        PricingStrategy pricingStrategy = new StandardPricing();
        BookingService bookingService = new BookingService(pricingStrategy);
        Booking booking = bookingService.createBooking(guest, room, 3);

        Invoice invoice = new BasicInvoice(300);
        invoice = new DiscountedInvoice(invoice, 30); // Apply discount

        ReportGenerator reportGenerator = ReportGenerator.getInstance();
        reportGenerator.generateReport();
    }
}
```

32. `Library Management System`

Here is a complete **Library Management System** implemented in **Java** using **Object-Oriented Design Principles** and key **Design Patterns** such as:

- **Singleton** – for `Library` instance

- **Factory** – for creating different book types
- **Strategy** – for implementing fine calculation strategies
- **Observer** – to notify users of overdue books or new arrivals

---

✅ Functional Requirements

1. **Book Management**:
   - Add/update/remove/search books
   - Maintain availability status
2. **User Management**:
   - Register users (students, faculty)
3. **Borrowing & Returning**:
   - Borrow a book (if available)
   - Return a book (calculate fine if late)
4. **Fine Calculation**:
   - Apply different fine rules for different user types
5. **Notifications**:
   - Notify users for overdue or new arrivals

🔧 Core Classes and Design Patterns

1. `Book` and `BookFactory` – Factory Pattern

```
abstract class Book {
   protected String id, title, author;
   protected boolean isAvailable = true;

   public Book(String id, String title, String author) {
      this.id = id; this.title = title; this.author = author;
   }

   public void borrow() { isAvailable = false; }
   public void returned() { isAvailable = true; }
   public boolean isAvailable() { return isAvailable; }

   public String getTitle() { return title; }
}
```

```
class TextBook extends Book {
    public TextBook(String id, String title, String author) {
        super(id, title, author);
    }
}

class BookFactory {
    public static Book createBook(String type, String id, String title, String author) {
        return switch (type.toLowerCase()) {
            case "textbook" -> new TextBook(id, title, author);
            default -> throw new IllegalArgumentException("Invalid type");
        };
    }
}
```

2. `User` and `FineStrategy` – Strategy Pattern

```
abstract class User {
    protected String id, name;
    protected List<Book> borrowedBooks = new ArrayList<>();

    public User(String id, String name) {
        this.id = id; this.name = name;
    }

    public abstract double calculateFine(int daysLate);

    public void borrowBook(Book book) {
        if (book.isAvailable()) {
            borrowedBooks.add(book);
            book.borrow();
        } else {
            System.out.println("Book unavailable");
        }
    }

    public void returnBook(Book book, int daysLate) {
        if (borrowedBooks.remove(book)) {
            book.returned();
            double fine = calculateFine(daysLate);
            System.out.println("Returned with fine: " + fine);
        }
```

```
    }
}

class Student extends User {
    public Student(String id, String name) { super(id, name); }
    public double calculateFine(int daysLate) {
        return daysLate > 7 ? (daysLate - 7) * 1.0 : 0.0;
    }
}

class Faculty extends User {
    public Faculty(String id, String name) { super(id, name); }
    public double calculateFine(int daysLate) {
        return daysLate > 14 ? (daysLate - 14) * 0.5 : 0.0;
    }
}
```

3. Library – Singleton + Observer Pattern

```
interface Observer {
    void update(String message);
}

class Library {
    private static Library instance;
    private List<Book> books = new ArrayList<>();
    private Map<String, User> users = new HashMap<>();
    private List<Observer> observers = new ArrayList<>();

    private Library() {}

    public static Library getInstance() {
        if (instance == null) instance = new Library();
        return instance;
    }

    public void registerObserver(Observer observer) {
        observers.add(observer);
    }

    public void notifyObservers(String message) {
        for (Observer o : observers) o.update(message);
```

```
    }

    public void addBook(Book book) {
        books.add(book);
        notifyObservers("New Book Added: " + book.getTitle());
    }

    public void registerUser(User user) {
        users.put(user.id, user);
    }

    public Book searchBook(String title) {
        return books.stream()
                .filter(b -> b.getTitle().equalsIgnoreCase(title) && b.isAvailable())
                .findFirst()
                .orElse(null);
    }

    public User getUser(String id) {
        return users.get(id);
    }
}
```

4. Notification Subscriber Example

```
class UserNotifier implements Observer {
    private String name;
    public UserNotifier(String name) { this.name = name; }

    public void update(String message) {
        System.out.println("[" + name + "] Notification: " + message);
    }
}
```

🧪 Sample Driver

```
public class Main {
    public static void main(String[] args) {
        Library library = Library.getInstance();

        // Add observer
```

```
        UserNotifier aliceNotifier = new UserNotifier("Alice");
        library.registerObserver(aliceNotifier);

        // Register Users
        User student = new Student("U1", "Alice");
        User faculty = new Faculty("U2", "Bob");

        library.registerUser(student);
        library.registerUser(faculty);

        // Add books
        Book book1 = BookFactory.createBook("textbook", "B1", "DSA", "Kunal");
        Book book2 = BookFactory.createBook("textbook", "B2", "Java", "James");
        library.addBook(book1);
        library.addBook(book2);

        // Borrow & return
        student.borrowBook(book1);
        student.returnBook(book1, 10); // Exceeds 7 day limit

        faculty.borrowBook(book2);
        faculty.returnBook(book2, 15); // Exceeds 14 day limit
    }
}
```

✅ Output

```
[Alice] Notification: New Book Added: DSA
[Alice] Notification: New Book Added: Java
Returned with fine: 3.0
Returned with fine: 0.5
```

33. Movie Ticket Booking System

Here's a **complete Java-based Movie Ticket Booking System** designed using **Object-Oriented Programming** principles and key **Design Patterns** like:

- **Singleton** – for centralized system access
- **Factory Pattern** – for screen and show creation

- **Strategy Pattern** – for dynamic pricing
- **Observer Pattern** – for user notifications on show updates

---

✅ Functional Requirements

- User registration/login
- Add theaters with screens and seats
- Add movies and schedule shows
- Book tickets for specific shows and seats
- View show listings
- Notify users about show cancellations or new releases

🔧 Core Classes and Design

1. User

```
class User {
    private final String id;
    private final String name;

    public User(String id, String name) {
        this.id = id;
        this.name = name;
    }

    public String getId() { return id; }
    public String getName() { return name; }
}
```

---

2. Movie

```
class Movie {
    private final String id;
    private final String title;
    private final int duration; // in minutes

    public Movie(String id, String title, int duration) {
        this.id = id;
        this.title = title;
```

```
        this.duration = duration;
    }

    public String getTitle() { return title; }
}
```

3. `Seat`

```
enum SeatType { REGULAR, PREMIUM }

class Seat {
    private final int seatNumber;
    private final SeatType type;
    private boolean isBooked;

    public Seat(int seatNumber, SeatType type) {
        this.seatNumber = seatNumber;
        this.type = type;
        this.isBooked = false;
    }

    public boolean isBooked() { return isBooked; }
    public void book() { this.isBooked = true; }

    public int getSeatNumber() { return seatNumber; }
    public SeatType getType() { return type; }
}
```

4. `Screen` and `Theater` – (Factory Pattern can be used for building screens)

```
class Screen {
    private final String id;
    private final List<Seat> seats;

    public Screen(String id, int numSeats) {
        this.id = id;
        this.seats = new ArrayList<>();
        for (int i = 1; i <= numSeats; i++) {
            seats.add(new Seat(i, i <= 5 ? SeatType.PREMIUM : SeatType.REGULAR));
        }
    }
```

```
    public List<Seat> getSeats() { return seats; }
    public String getId() { return id; }
}
```

```
class Theater {
    private final String id;
    private final String name;
    private final List<Screen> screens;

    public Theater(String id, String name) {
        this.id = id;
        this.name = name;
        this.screens = new ArrayList<>();
    }

    public void addScreen(Screen screen) { screens.add(screen); }
    public List<Screen> getScreens() { return screens; }
    public String getName() { return name; }
}
```

5. `Show` and `Booking`

```
class Show {
    private final String id;
    private final Movie movie;
    private final Screen screen;
    private final LocalDateTime startTime;

    public Show(String id, Movie movie, Screen screen, LocalDateTime startTime) {
        this.id = id;
        this.movie = movie;
        this.screen = screen;
        this.startTime = startTime;
    }

    public Movie getMovie() { return movie; }
    public Screen getScreen() { return screen; }
    public LocalDateTime getStartTime() { return startTime; }
}
```

```
class Booking {
    private final String id;
    private final User user;
    private final Show show;
    private final List<Seat> bookedSeats;

    public Booking(String id, User user, Show show, List<Seat> bookedSeats) {
        this.id = id;
        this.user = user;
        this.show = show;
        this.bookedSeats = bookedSeats;
    }

    public void printReceipt() {
        System.out.println("Booking for " + user.getName() + " - Movie: " +
                show.getMovie().getTitle() + " Seats: " + bookedSeats.stream()
                .map(seat -> String.valueOf(seat.getSeatNumber()))
                .collect(Collectors.joining(", ")));
    }
}
```

---

6. `PricingStrategy` – Strategy Pattern

```
interface PricingStrategy {
    double calculatePrice(Seat seat);
}

class RegularPricing implements PricingStrategy {
    public double calculatePrice(Seat seat) {
        return seat.getType() == SeatType.PREMIUM ? 250 : 150;
    }
}
```

---

7. `BookingService` (Singleton)

```
class BookingService {
    private static BookingService instance;
    private final List<Booking> bookings = new ArrayList<>();
    private final PricingStrategy pricingStrategy = new RegularPricing();

    private BookingService() {}
```

```
    public static BookingService getInstance() {
        if (instance == null) instance = new BookingService();
        return instance;
    }

    public Booking bookSeats(User user, Show show, List<Integer> seatNumbers) {
        List<Seat> selectedSeats = new ArrayList<>();
        for (int number : seatNumbers) {
            Seat seat = show.getScreen().getSeats().get(number - 1);
            if (seat.isBooked()) {
                System.out.println("Seat already booked: " + number);
                return null;
            }
            seat.book();
            selectedSeats.add(seat);
        }

        Booking booking = new Booking(UUID.randomUUID().toString(), user, show, selectedSeat
s);
        bookings.add(booking);
        return booking;
    }
}
```

🧪 Sample Flow (Driver)

```
public class Main {
    public static void main(String[] args) {
        // Create Users
        User user1 = new User("U1", "Alice");

        // Create Theater, Screen, Movie
        Theater inox = new Theater("T1", "INOX");
        Screen screen1 = new Screen("S1", 10);
        inox.addScreen(screen1);

        Movie movie = new Movie("M1", "Avengers", 120);
        Show show = new Show("SH1", movie, screen1, LocalDateTime.now().plusHours(1));

        // Book Tickets
        BookingService bookingService = BookingService.getInstance();
```

```
        Booking booking = bookingService.bookSeats(user1, show, List.of(1, 2, 3));
        if (booking != null) booking.printReceipt();
    }
}
```

✅ Output Example

```
Booking for Alice - Movie: Avengers Seats: 1, 2, 3
```

🧠 Possible Extensions

- Add **seat hold** & timeout feature
- Integrate with **payment gateway**
- Add **discount coupons**
- Integrate with Spring Boot for APIs
- Add Observer pattern for **user notifications** on cancellations or new shows

---

34. `Restaurant Management System`

Here's a complete **Restaurant Management System** implemented in **Java**, applying **object-oriented design principles** and **design patterns** such as:

- **Singleton** – for managing the central system state
- **Strategy** – for sorting restaurants by price or rating
- **Observer** – for notifying users of updates
- **Builder Pattern** – for creating complex `Restaurant` objects

✅ Features:

- Register user
- Register restaurant with multiple pincodes
- Add/update quantity
- Place orders
- Create reviews
- Show restaurants by price/rating

- View user order history

🔧 Core Models

1. User.java

```
class User {
    private final String name;
    private final String phone;
    private final String location;
    private final List<Order> orderHistory = new ArrayList<>();

    public User(String name, String phone, String location) {
        this.name = name;
        this.phone = phone;
        this.location = location;
    }

    public String getPhone() { return phone; }
    public String getLocation() { return location; }
    public List<Order> getOrderHistory() { return orderHistory; }

    public void addOrder(Order order) {
        orderHistory.add(order);
    }
}
```

2. FoodItem.java

```
class FoodItem {
    private final String name;
    private final int price;
    private int quantity;

    public FoodItem(String name, int price, int quantity) {
        this.name = name;
        this.price = price;
        this.quantity = quantity;
    }

    public void addQuantity(int q) { this.quantity += q; }
    public boolean isAvailable(int q) { return quantity >= q; }
```

```
    public void reduceQuantity(int q) { this.quantity -= q; }

    public int getPrice() { return price; }
    public String getName() { return name; }
    public int getQuantity() { return quantity; }
}
```

3. Restaurant.java

```
class Restaurant {
    private final String name;
    private final Set<String> pincodes;
    private final FoodItem foodItem;
    private final List<Integer> ratings = new ArrayList<>();
    private final List<String> comments = new ArrayList<>();

    public Restaurant(String name, Set<String> pincodes, FoodItem foodItem) {
        this.name = name;
        this.pincodes = pincodes;
        this.foodItem = foodItem;
    }

    public boolean servesPincode(String pincode) {
        return pincodes.contains(pincode);
    }

    public void addRating(int rating, String comment) {
        ratings.add(rating);
        if (comment != null && !comment.isEmpty()) comments.add(comment);
    }

    public double getAverageRating() {
        return ratings.isEmpty() ? 0 : ratings.stream().mapToInt(i -> i).average().orElse(0);
    }

    public String getName() { return name; }
    public FoodItem getFoodItem() { return foodItem; }

    public void addQuantity(int q) {
        foodItem.addQuantity(q);
    }
```

```
    public boolean canServe(int quantity) {
        return foodItem.isAvailable(quantity);
    }

    public void reduceQuantity(int quantity) {
        foodItem.reduceQuantity(quantity);
    }

    public String getDescription() {
        return name + ", " + foodItem.getName() + " - " + foodItem.getPrice() + " (" + getAverageR
ating() + ")";
    }
}
```

4. Order.java

```
class Order {
    private final Restaurant restaurant;
    private final int quantity;

    public Order(Restaurant restaurant, int quantity) {
        this.restaurant = restaurant;
        this.quantity = quantity;
    }

    public String getSummary() {
        return restaurant.getName() + ", " + restaurant.getFoodItem().getName() + " x" + quantity;
    }
}
```

🧠 Strategy Pattern for Sorting

```
interface RestaurantSorter {
    List<Restaurant> sort(List<Restaurant> list);
}

class PriceSorter implements RestaurantSorter {
    public List<Restaurant> sort(List<Restaurant> list) {
        return list.stream()
                .sorted(Comparator.comparingInt(r -> r.getFoodItem().getPrice()))
                .collect(Collectors.toList());
```

```
    }
}

class RatingSorter implements RestaurantSorter {
    public List<Restaurant> sort(List<Restaurant> list) {
        return list.stream()
                .sorted(Comparator.comparingDouble(Restaurant::getAverageRating).reversed())
                .collect(Collectors.toList());
    }
}
```

5. RestaurantSystem.java (Singleton)

```
class RestaurantSystem {
    private static final RestaurantSystem instance = new RestaurantSystem();
    private final Map<String, User> users = new HashMap<>();
    private final List<Restaurant> restaurants = new ArrayList<>();
    private User loggedInUser;

    public static RestaurantSystem getInstance() {
        return instance;
    }

    public void registerUser(String name, String phone, String location) {
        users.put(phone, new User(name, phone, location));
    }

    public void loginUser(String phone) {
        if (!users.containsKey(phone)) {
            System.out.println("User not found");
            return;
        }
        loggedInUser = users.get(phone);
    }

    public void registerRestaurant(String name, String pincodesStr, String foodName, int price, int
qty) {
        Set<String> pincodes = new HashSet<>(Arrays.asList(pincodesStr.split("/")));
        restaurants.add(new Restaurant(name, pincodes, new FoodItem(foodName, price, qty)));
    }

    public void updateQuantity(String name, int qty) {
```

```
        for (Restaurant r : restaurants) {
            if (r.getName().equals(name)) {
                r.addQuantity(qty);
                System.out.println("Updated: " + r.getDescription());
            }
        }
    }

    public void placeOrder(String name, int qty) {
        for (Restaurant r : restaurants) {
            if (r.getName().equals(name) && r.servesPincode(loggedInUser.getLocation())) {
                if (r.canServe(qty)) {
                    r.reduceQuantity(qty);
                    Order order = new Order(r, qty);
                    loggedInUser.addOrder(order);
                    System.out.println("Order Placed Successfully.");
                    return;
                } else {
                    System.out.println("Cannot place order. Not enough quantity.");
                    return;
                }
            }
        }
        System.out.println("Restaurant not serviceable in your area.");
    }

    public void createReview(String name, int rating, String comment) {
        for (Restaurant r : restaurants) {
            if (r.getName().equals(name)) {
                r.addRating(rating, comment);
            }
        }
    }

    public void showRestaurants(String filter) {
        List<Restaurant> serviceable = restaurants.stream()
                .filter(r -> r.servesPincode(loggedInUser.getLocation()))
                .collect(Collectors.toList());

        RestaurantSorter sorter = "rating".equalsIgnoreCase(filter) ? new RatingSorter() : new Price
Sorter();
        List<Restaurant> sorted = sorter.sort(serviceable);
```

```
        for (Restaurant r : sorted) {
            System.out.println(r.getDescription());
        }
    }

    public void showOrderHistory() {
        for (Order o : loggedInUser.getOrderHistory()) {
            System.out.println(o.getSummary());
        }
    }
}
```

🚀 Driver Program

```
public class Main {
    public static void main(String[] args) {
        RestaurantSystem system = RestaurantSystem.getInstance();

        system.registerUser("Pralove", "phoneNumber-1", "HSR");
        system.registerUser("Nitesh", "phoneNumber-2", "BTM");

        system.loginUser("phoneNumber-1");
        system.registerRestaurant("Food Court-1", "BTM/HSR", "NI Thali", 100, 5);
        system.registerRestaurant("Food Court-2", "BTM/Indiranagar", "Burger", 120, 3);

        system.loginUser("phoneNumber-2");
        system.registerRestaurant("Food Court-3", "HSR", "SI Thali", 150, 1);

        system.loginUser("phoneNumber-1");
        system.showRestaurants("price"); // sorted by price

        system.placeOrder("Food Court-1", 2);
        system.placeOrder("Food Court-2", 7); // not enough quantity

        system.createReview("Food Court-1", 5, "Nice Food");
        system.createReview("Food Court-2", 3, "Good Food");

        system.showRestaurants("rating");

        system.updateQuantity("Food Court-2", 5);

        system.showOrderHistory();
```

```
    }
}
```

✅ Sample Output

```
Food Court-1, NI Thali - 100 (0.0)
Food Court-2, Burger - 120 (0.0)
Order Placed Successfully.
Cannot place order. Not enough quantity.
Food Court-1, NI Thali - 100 (5.0)
Food Court-2, Burger - 120 (3.0)
Updated: Food Court-2, Burger - 8 (3.0)
Food Court-1, NI Thali x2
```

---

35. Amazon - Online Shopping System

🛒 Functional Requirements

- **User Management**: Registration, login, and profile management for customers and administrators.
- **Product Catalog**: Browsing, searching, and filtering products by categories, brands, and other attributes.
- **Shopping Cart**: Adding, updating, and removing items; viewing cart contents.
- **Order Processing**: Placing orders, order history, and order status tracking.
- **Payment Integration**: Supporting multiple payment methods.
- **Shipping and Delivery**: Managing shipping addresses, delivery options, and tracking shipments.
- **Product Reviews and Ratings**: Allowing customers to review and rate products.
- **Notifications**: Sending order confirmations, shipping updates, and promotional messages.

🧱 Core Classes and Design Patterns

1. User and Account Management

```
class User {
    private String userId;
```

```
    private String name;
    private String email;
    private String password;
    private Address shippingAddress;
    // Getters and setters
}
```

2. Product and Catalog

```
class Product {
    private String productId;
    private String name;
    private String description;
    private double price;
    private int availableQuantity;
    private Category category;
    private List<Review> reviews;
    // Getters and setters
}
```

3. Shopping Cart

```
class Cart {
    private String cartId;
    private User user;
    private List<CartItem> items;
    // Methods to add, remove, update items
}
```

4. Order and Payment

```
class Order {
    private String orderId;
    private User user;
    private List<OrderItem> orderItems;
    private Payment payment;
    private OrderStatus status;
    private LocalDateTime orderDate;
    // Getters and setters
}
```

5. Design Patterns Applied

- **Singleton Pattern**: To ensure a single instance of the shopping cart per user session.Wikipedia

```
class CartManager {
    private static CartManager instance;
    private Map<String, Cart> userCarts;
    private CartManager() {
        userCarts = new HashMap<>();
    }
    public static CartManager getInstance() {
        if (instance == null) {
            instance = new CartManager();
        }
        return instance;
    }
    // Methods to manage carts
}
```

- **Factory Pattern**: To create different types of payment methods.

```
interface PaymentMethod {
    void pay(double amount);
}

class CreditCardPayment implements PaymentMethod {
    public void pay(double amount) {
        // Process credit card payment
    }
}

class PayPalPayment implements PaymentMethod {
    public void pay(double amount) {
        // Process PayPal payment
    }
}

class PaymentFactory {
    public static PaymentMethod getPaymentMethod(String type) {
        if (type.equalsIgnoreCase("CreditCard")) {
            return new CreditCardPayment();
        } else if (type.equalsIgnoreCase("PayPal")) {
            return new PayPalPayment();
        }
```

```
        return null;
    }
}
```

- **Observer Pattern**: To notify users about order status changes.

```
interface Observer {
    void update(String message);
}

class User implements Observer {
    public void update(String message) {
        // Notify user
    }
}

class OrderStatusNotifier {
    private List<Observer> observers = new ArrayList<>();
    public void addObserver(Observer observer) {
        observers.add(observer);
    }
    public void notifyObservers(String message) {
        for (Observer observer : observers) {
            observer.update(message);
        }
    }
}
```

- **Strategy Pattern**: To apply different discount strategies.

```
interface DiscountStrategy {
    double applyDiscount(double price);
}

class NoDiscount implements DiscountStrategy {
    public double applyDiscount(double price) {
        return price;
    }
}

class SeasonalDiscount implements DiscountStrategy {
    public double applyDiscount(double price) {
        return price * 0.9; // 10% discount
```

```
    }
}

class Product {
    private DiscountStrategy discountStrategy;
    public void setDiscountStrategy(DiscountStrategy discountStrategy) {
        this.discountStrategy = discountStrategy;
    }
    public double getPrice() {
        return discountStrategy.applyDiscount(price);
    }
}
```

📦 Additional Considerations

- **Search Functionality**: Implementing search with filters and sorting options.
- **Inventory Management**: Updating product quantities post-purchase.
- **Security**: Implementing authentication, authorization, and secure payment processing.
- **Scalability**: Designing the system to handle high traffic and large data volumes.

---

36. Design an Airline Management System

🛫 Functional Requirements

1. **User Management**
    - Register/Login
    - Role-based access (Admin/User)
2. **Flight Management (Admin)**
    - Add/Update/Delete flights
    - Define source, destination, price, capacity, and schedule
3. **Search Flights (User)**
    - Search by source, destination, date
    - Filter by price/time
4. **Booking System**
    - Book seats, cancel tickets

- Check seat availability

5. **View Bookings**
   - See user's booking history
   - Admin can view all bookings
6. **Notifications**
   - Booking confirmation, flight updates

🧰 Core Classes and Design Patterns

1. `User`, `Admin`, and `Passenger`

```
abstract class User {
    String userId;
    String name;
    String email;
    String password;
    abstract void viewBookings();
}

class Admin extends User {
    void addFlight(Flight flight) {}
    void updateFlight(String flightId, Flight updatedFlight) {}
    void deleteFlight(String flightId) {}
    void viewAllBookings() {}
}

class Passenger extends User {
    void bookFlight(Flight flight, int seats) {}
    void cancelBooking(String bookingId) {}
}
```

---

2. `Flight` and `Airport`

```
class Flight {
    String flightId;
    String airline;
    Airport source;
    Airport destination;
    LocalDateTime departure;
    LocalDateTime arrival;
```

```
    double price;
    int capacity;
    int availableSeats;
}

class Airport {
    String code;  // e.g., JFK
    String city;
    String country;
}
```

3. Booking

```
class Booking {
    String bookingId;
    Passenger passenger;
    Flight flight;
    int seatsBooked;
    LocalDateTime bookingTime;
    BookingStatus status;
}

enum BookingStatus {
    CONFIRMED, CANCELLED
}
```

✅ Design Patterns

1. Factory Pattern — For creating users

```
class UserFactory {
    public static User createUser(String type, String name, String email, String password) {
        if ("admin".equalsIgnoreCase(type)) {
            return new Admin(); // set properties
        } else {
            return new Passenger(); // set properties
        }
    }
}
```

2. Strategy Pattern — For flight search strategies

```
interface SearchStrategy {
    List<Flight> search(List<Flight> allFlights, String source, String destination);
}

class TimeBasedSearch implements SearchStrategy {
    public List<Flight> search(List<Flight> flights, String src, String dest) {
        // Filter by time
    }
}

class PriceBasedSearch implements SearchStrategy {
    public List<Flight> search(List<Flight> flights, String src, String dest) {
        // Filter by price
    }
}
```

3. Observer Pattern — For notifications

```
interface Observer {
    void notify(String message);
}

class EmailService implements Observer {
    public void notify(String message) {
        System.out.println("Email: " + message);
    }
}

class NotificationManager {
    List<Observer> observers = new ArrayList<>();

    void addObserver(Observer o) {
        observers.add(o);
    }

    void notifyAll(String message) {
        for (Observer o : observers) {
            o.notify(message);
        }
```

```
    }
}
```

📊 Class Diagram (Text Representation)

```
User
 ├── Admin
 └── Passenger

Flight
Booking
Airport

SearchStrategy
 ├── PriceBasedSearch
 └── TimeBasedSearch

Observer
 └── EmailService
```

⛓️ Sequence Diagram (Book Flight)

Passenger → System

1. Search for flights
2. Select flight
3. Book flight (check seat availability, create booking, update flight availability)
4. Notify passenger via Observer (Email)

✅ Sample Flow

```
User user = UserFactory.createUser("passenger", "John", "john@mail.com", "pass123");

Airport src = new Airport("DEL", "Delhi", "India");
Airport dest = new Airport("BLR", "Bangalore", "India");

Flight flight = new Flight("AI101", "Air India", src, dest, ...);
Admin admin = (Admin) UserFactory.createUser("admin", ...);
admin.addFlight(flight);
```

```
SearchStrategy search = new PriceBasedSearch();
List<Flight> flights = search.search(flightRepo.getAll(), "Delhi", "Bangalore");

Passenger passenger = (Passenger) user;
passenger.bookFlight(flight, 2);
```

🧪 Extendability Ideas

- Add loyalty programs or discount strategies (Strategy Pattern)
- Use Command Pattern for undo booking
- Add persistence via JDBC/Hibernate
- Wrap APIs using Spring Boot

---

37. `Design an ATM (Automated Teller Machine)`

🧾 Functional Requirements

1. Insert card and validate PIN
2. Select account (savings/current)
3. View balance
4. Withdraw cash
5. Deposit cash
6. Print mini-statement
7. Eject card

🧱 Core Classes

1. `ATM`, `Bank`, and `ATMState` (State Pattern)

```
class ATM {
    private ATMState currentState;
    private Card insertedCard;
    private Bank bank;

    public ATM(Bank bank) {
        this.bank = bank;
```

```
        this.currentState = new IdleState(this);
    }

    public void setState(ATMState state) {
        this.currentState = state;
    }

    public void insertCard(Card card) {
        this.currentState.insertCard(card);
    }

    public void enterPin(String pin) {
        this.currentState.enterPin(pin);
    }

    public void withdraw(double amount) {
        this.currentState.withdraw(amount);
    }

    public void deposit(double amount) {
        this.currentState.deposit(amount);
    }

    public void ejectCard() {
        this.currentState.ejectCard();
    }

    public Bank getBank() {
        return this.bank;
    }

    public void setCard(Card card) {
        this.insertedCard = card;
    }

    public Card getCard() {
        return this.insertedCard;
    }
}
```

2. `ATMState` Interface and Concrete States

```
interface ATMState {
    void insertCard(Card card);
    void enterPin(String pin);
    void withdraw(double amount);
    void deposit(double amount);
    void ejectCard();
}

class IdleState implements ATMState {
    private ATM atm;

    public IdleState(ATM atm) {
        this.atm = atm;
    }

    public void insertCard(Card card) {
        System.out.println("Card Inserted.");
        atm.setCard(card);
        atm.setState(new CardInsertedState(atm));
    }

    public void enterPin(String pin) {
        System.out.println("No card inserted.");
    }

    public void withdraw(double amount) {
        System.out.println("No card inserted.");
    }

    public void deposit(double amount) {
        System.out.println("No card inserted.");
    }

    public void ejectCard() {
        System.out.println("No card to eject.");
    }
}
```

```
class CardInsertedState implements ATMState {
    private ATM atm;

    public CardInsertedState(ATM atm) {
```

```
        this.atm = atm;
    }

    public void insertCard(Card card) {
        System.out.println("Card already inserted.");
    }

    public void enterPin(String pin) {
        Card card = atm.getCard();
        if (atm.getBank().validatePin(card, pin)) {
            System.out.println("PIN verified.");
            atm.setState(new AuthenticatedState(atm));
        } else {
            System.out.println("Invalid PIN.");
        }
    }

    public void withdraw(double amount) {
        System.out.println("Enter PIN first.");
    }

    public void deposit(double amount) {
        System.out.println("Enter PIN first.");
    }

    public void ejectCard() {
        System.out.println("Card ejected.");
        atm.setCard(null);
        atm.setState(new IdleState(atm));
    }
}
```

3. `AuthenticatedState` and Transaction Handling

```
class AuthenticatedState implements ATMState {
    private ATM atm;

    public AuthenticatedState(ATM atm) {
        this.atm = atm;
    }

    public void insertCard(Card card) {
```

```
        System.out.println("Card already inserted.");
    }

    public void enterPin(String pin) {
        System.out.println("Already authenticated.");
    }

    public void withdraw(double amount) {
        Card card = atm.getCard();
        if (atm.getBank().withdraw(card, amount)) {
            System.out.println("Withdrawn: " + amount);
        } else {
            System.out.println("Insufficient balance.");
        }
    }

    public void deposit(double amount) {
        Card card = atm.getCard();
        atm.getBank().deposit(card, amount);
        System.out.println("Deposited: " + amount);
    }

    public void ejectCard() {
        System.out.println("Card ejected.");
        atm.setCard(null);
        atm.setState(new IdleState(atm));
    }
}
```

4. `Card`, `Account`, and `Bank`

```
class Card {
    private String cardNumber;
    private String pin;
    private Account account;

    public Card(String cardNumber, String pin, Account account) {
        this.cardNumber = cardNumber;
        this.pin = pin;
        this.account = account;
    }
```

```
    public boolean verifyPin(String inputPin) {
        return this.pin.equals(inputPin);
    }

    public Account getAccount() {
        return account;
    }
}

class Account {
    private String accountNumber;
    private double balance;

    public Account(String accountNumber, double balance) {
        this.accountNumber = accountNumber;
        this.balance = balance;
    }

    public double getBalance() {
        return balance;
    }

    public boolean withdraw(double amount) {
        if (amount > balance) return false;
        balance -= amount;
        return true;
    }

    public void deposit(double amount) {
        balance += amount;
    }
}
```

```
class Bank {
    private Map<String, Card> cardMap = new HashMap<>();

    public void addCard(Card card) {
        cardMap.put(card.cardNumber, card);
    }

    public boolean validatePin(Card card, String pin) {
        return card.verifyPin(pin);
    }
```

```
    public boolean withdraw(Card card, double amount) {
        return card.getAccount().withdraw(amount);
    }

    public void deposit(Card card, double amount) {
        card.getAccount().deposit(amount);
    }
}
```

✅ Sample Flow (Driver)

```
public class ATMApp {
    public static void main(String[] args) {
        Bank bank = new Bank();
        Account acc = new Account("123456", 1000.0);
        Card card = new Card("1111-2222", "1234", acc);
        bank.addCard(card);

        ATM atm = new ATM(bank);
        atm.insertCard(card);
        atm.enterPin("1234");
        atm.withdraw(300);
        atm.deposit(200);
        atm.ejectCard();
    }
}
```

📐 Design Patterns Used

Pattern	Usage
State Pattern	Manage ATM behavior based on state
Strategy	(optional for multi-account types)
Factory	Could be used for card/account generation
Singleton	(optional for ATM or Bank instance)

🧪 Extendability

- Add support for multiple ATMs

- Support for mini-statement printing
- Multi-bank support with interfaces
- Spring Boot REST API integration

---

38. Design an Online Stock Brokerage System

📌 Key Features

1. **User Management**: Registration, login, KYC.
2. **Account Management**: Track balances, holdings.
3. **Trading**: Place Buy/Sell orders.
4. **Portfolio**: View stock holdings and performance.
5. **Market Data**: Real-time price updates.
6. **Order Matching Engine**: Match buy and sell orders (simplified).
7. **Observer Pattern**: Notify users of stock price changes.

🧰 Key Classes and Components

1. User and Account

```
class User {
    private String id;
    private String name;
    private Account account;

    public User(String id, String name) {
        this.id = id;
        this.name = name;
        this.account = new Account();
    }

    public Account getAccount() {
        return account;
    }

    public String getName() {
        return name;
    }
}
```

```
class Account {
    private double balance;
    private Map<String, Integer> holdings = new HashMap<>(); // stockSymbol → quantity

    public boolean debit(double amount) {
        if (balance < amount) return false;
        balance -= amount;
        return true;
    }

    public void credit(double amount) {
        balance += amount;
    }

    public void addStock(String symbol, int qty) {
        holdings.put(symbol, holdings.getOrDefault(symbol, 0) + qty);
    }

    public boolean removeStock(String symbol, int qty) {
        int current = holdings.getOrDefault(symbol, 0);
        if (current < qty) return false;
        holdings.put(symbol, current - qty);
        return true;
    }

    public Map<String, Integer> getHoldings() {
        return holdings;
    }

    public double getBalance() {
        return balance;
    }
}
```

2. Stock and Observer Pattern

```
interface StockObserver {
    void onPriceUpdate(String symbol, double price);
}

class Stock {
```

```
    private String symbol;
    private double price;
    private List<StockObserver> observers = new ArrayList<>();

    public Stock(String symbol, double price) {
        this.symbol = symbol;
        this.price = price;
    }

    public void addObserver(StockObserver observer) {
        observers.add(observer);
    }

    public void updatePrice(double newPrice) {
        this.price = newPrice;
        notifyObservers();
    }

    private void notifyObservers() {
        for (StockObserver observer : observers) {
            observer.onPriceUpdate(symbol, price);
        }
    }

    public double getPrice() {
        return price;
    }

    public String getSymbol() {
        return symbol;
    }
}
```

3. Order and Trading Engine

```
enum OrderType {
    BUY, SELL
}

class Order {
    private User user;
    private String stockSymbol;
```

```
    private int quantity;
    private double price;
    private OrderType type;

    public Order(User user, String stockSymbol, int quantity, double price, OrderType type) {
        this.user = user;
        this.stockSymbol = stockSymbol;
        this.quantity = quantity;
        this.price = price;
        this.type = type;
    }

    // Getters...
    public User getUser() { return user; }
    public String getStockSymbol() { return stockSymbol; }
    public int getQuantity() { return quantity; }
    public double getPrice() { return price; }
    public OrderType getType() { return type; }
}
```

```
class OrderBook {
    private List<Order> buyOrders = new ArrayList<>();
    private List<Order> sellOrders = new ArrayList<>();

    public void placeOrder(Order order) {
        if (order.getType() == OrderType.BUY) {
            buyOrders.add(order);
            matchOrders();
        } else {
            sellOrders.add(order);
            matchOrders();
        }
    }

    private void matchOrders() {
        Iterator<Order> buys = buyOrders.iterator();
        while (buys.hasNext()) {
            Order buy = buys.next();
            Iterator<Order> sells = sellOrders.iterator();

            while (sells.hasNext()) {
                Order sell = sells.next();
```

```
            if (buy.getStockSymbol().equals(sell.getStockSymbol()) &&
                buy.getPrice() >= sell.getPrice() &&
                buy.getQuantity() == sell.getQuantity()) {

                // Execute trade
                executeTrade(buy, sell);
                buys.remove();
                sells.remove();
                break;
            }
        }
    }
  }

  private void executeTrade(Order buy, Order sell) {
    double totalPrice = sell.getPrice() * sell.getQuantity();

    if (!buy.getUser().getAccount().debit(totalPrice)) {
      System.out.println("Buyer has insufficient balance.");
      return;
    }

    if (!sell.getUser().getAccount().removeStock(sell.getStockSymbol(), sell.getQuantity())) {
      System.out.println("Seller has insufficient stocks.");
      return;
    }

    sell.getUser().getAccount().credit(totalPrice);
    buy.getUser().getAccount().addStock(sell.getStockSymbol(), sell.getQuantity());

    System.out.println("Trade Executed: " + sell.getQuantity() + " shares of " +
        sell.getStockSymbol() + " at " + sell.getPrice());
  }
}
```

4. Driver Code

```
public class StockBrokerageApp {
  public static void main(String[] args) {
    // Setup
    User alice = new User("1", "Alice");
    User bob = new User("2", "Bob");
```

```
        alice.getAccount().credit(10000);
        bob.getAccount().addStock("TSLA", 10);

        Stock tsla = new Stock("TSLA", 900);
        tsla.addObserver((symbol, price) -> System.out.println("Price update: " + symbol + " -> " +
price));

        OrderBook orderBook = new OrderBook();

        // Place Orders
        Order buyOrder = new Order(alice, "TSLA", 5, 910, OrderType.BUY);
        Order sellOrder = new Order(bob, "TSLA", 5, 905, OrderType.SELL);

        orderBook.placeOrder(buyOrder);
        orderBook.placeOrder(sellOrder);

        tsla.updatePrice(905);
    }
}
```

✅ Design Patterns Used

Pattern	Purpose
Observer	Stock price alerts to users
Strategy	Can be added for trading strategies
Factory	For generating orders/accounts
Singleton	For central market/order book manager

39. Design Blackjack and a Deck of Cards

Here's a **complete Java implementation of Blackjack and a Deck of Cards**, designed using **OOP principles** and **design patterns** like:

- **Strategy Pattern** (for different player behaviors)
- **Factory Pattern** (for deck/card creation)
- **Singleton Pattern** (for the game instance, optional)
- **Observer Pattern** (optional – notify player status or UI updates)

✅ Key Classes Overview

1. `Card` – represents a playing card
2. `Deck` – manages the deck of 52 cards
3. `Player` – represents a game player
4. `BlackjackGame` – core game logic
5. `GameStrategy` – strategy pattern for player decisions
6. `Main` – entry point

✅ Implementation

1. Card.java

```
public class Card {
    public enum Suit { HEARTS, DIAMONDS, CLUBS, SPADES }
    public enum Rank {
        TWO(2), THREE(3), FOUR(4), FIVE(5), SIX(6),
        SEVEN(7), EIGHT(8), NINE(9), TEN(10),
        JACK(10), QUEEN(10), KING(10), ACE(11);

        private final int value;
        Rank(int value) { this.value = value; }
        public int getValue() { return value; }
    }

    private final Suit suit;
    private final Rank rank;

    public Card(Suit suit, Rank rank) {
        this.suit = suit;
        this.rank = rank;
    }

    public int getValue() {
        return rank.getValue();
    }

    public boolean isAce() {
        return rank == Rank.ACE;
```

```
    }

    @Override
    public String toString() {
        return rank + " of " + suit;
    }
}
```

2. Deck.java

```
import java.util.*;

public class Deck {
    private final Stack<Card> cards = new Stack<>();

    public Deck() {
        for (Card.Suit suit : Card.Suit.values()) {
            for (Card.Rank rank : Card.Rank.values()) {
                cards.push(new Card(suit, rank));
            }
        }
        shuffle();
    }

    public void shuffle() {
        Collections.shuffle(cards);
    }

    public Card drawCard() {
        if (cards.isEmpty()) throw new IllegalStateException("Deck is empty!");
        return cards.pop();
    }
}
```

3. Player.java

```
import java.util.*;

public class Player {
    private final String name;
    private final List<Card> hand = new ArrayList<>();
    private boolean isDealer = false;
```

```
public Player(String name, boolean isDealer) {
    this.name = name;
    this.isDealer = isDealer;
}

public void addCard(Card card) {
    hand.add(card);
}

public int getHandValue() {
    int total = 0;
    int aceCount = 0;
    for (Card card : hand) {
        total += card.getValue();
        if (card.isAce()) aceCount++;
    }

    while (total > 21 && aceCount > 0) {
        total -= 10;  // Count Ace as 1 instead of 11
        aceCount--;
    }
    return total;
}

public boolean isBusted() {
    return getHandValue() > 21;
}

public List<Card> getHand() {
    return hand;
}

public String getName() {
    return name;
}

public boolean isDealer() {
    return isDealer;
}

public void showHand(boolean showAll) {
    System.out.println(name + "'s hand:");
```

```
        for (int i = 0; i < hand.size(); i++) {
            if (!showAll && isDealer && i == 0)
                System.out.println("Hidden Card");
            else
                System.out.println(hand.get(i));
        }
        if (!isDealer || showAll)
            System.out.println("Total Value: " + getHandValue());
    }
}
```

4. GameStrategy.java (Strategy Pattern)

```
public interface GameStrategy {
    boolean shouldHit(Player player);
}

public class BasicStrategy implements GameStrategy {
    @Override
    public boolean shouldHit(Player player) {
        return player.getHandValue() < 17;
    }
}
```

5. BlackjackGame.java

```
public class BlackjackGame {
    private final Deck deck;
    private final Player dealer;
    private final Player player;
    private final GameStrategy strategy;

    public BlackjackGame(String playerName) {
        this.deck = new Deck();
        this.player = new Player(playerName, false);
        this.dealer = new Player("Dealer", true);
        this.strategy = new BasicStrategy(); // can swap strategy here
    }

    public void start() {
        System.out.println("Starting Blackjack...\n");
```

```
        // Initial deal
        player.addCard(deck.drawCard());
        dealer.addCard(deck.drawCard());
        player.addCard(deck.drawCard());
        dealer.addCard(deck.drawCard());

        player.showHand(true);
        dealer.showHand(false);

        // Player's turn
        while (!player.isBusted() && strategy.shouldHit(player)) {
            System.out.println("\n" + player.getName() + " hits!");
            player.addCard(deck.drawCard());
            player.showHand(true);
        }

        if (player.isBusted()) {
            System.out.println("\n" + player.getName() + " busted! Dealer wins.");
            return;
        }

        // Dealer's turn
        System.out.println("\nDealer's turn:");
        dealer.showHand(true);
        while (strategy.shouldHit(dealer)) {
            System.out.println("Dealer hits.");
            dealer.addCard(deck.drawCard());
            dealer.showHand(true);
        }

        if (dealer.isBusted()) {
            System.out.println("Dealer busted! " + player.getName() + " wins!");
        } else {
            // Compare hands
            int playerScore = player.getHandValue();
            int dealerScore = dealer.getHandValue();
            System.out.println("\nFinal Scores ⇒ " + player.getName() + ": " + playerScore + ", Deal
er: " + dealerScore);

            if (playerScore > dealerScore) {
                System.out.println(player.getName() + " wins!");
            } else if (dealerScore > playerScore) {
                System.out.println("Dealer wins!");
```

```
        } else {
            System.out.println("It's a tie!");
        }
    }
  }
}
```

6. Main.java

```
public static void main(String[] args) {
        BlackjackGame game = new BlackjackGame("Alice");
        game.start();
    }
}
```

✅ Design Patterns Used

Pattern	Purpose
Strategy	Switch player/dealer behavior (e.g. AI vs manual)
Factory	Deck/card creation abstraction
Observer	(Optional) Push updates to UI/logger
Singleton	(Optional) Game manager instance

📌 Extensions

- Add user input for hit/stand decisions.
- Add betting and chips system.
- Add multiple players.
- Add UI using Swing/JavaFX.
- Connect to web frontend (Spring Boot REST).

40. Design Cricinfo

Cricinfo Simplified Design Outline

Key Features to Model

- Teams and Players
- Matches and Live Scores
- Score Updates and Notifications
- Display of Match Info (e.g., summary, scorecards)
- User subscriptions to match updates (optional)

Step 1: Core Classes & Interfaces

1.1 Entities

- **Player**: player details (name, role, stats)
- **Team**: team details (name, list of players)
- **Match**: match info (teams, status, scores)
- **Score**: track runs, wickets, overs

Step 2: Applying Design Patterns

Patterns to use:

- **Observer Pattern**: For live score updates — Users (or UI components) subscribe to Match to receive updates.
- **Factory Pattern**: To create different types of Matches (Test, ODI, T20).
- **Singleton Pattern**: For central MatchManager or ScoreManager to manage ongoing matches.
- **Strategy Pattern**: For different scoring calculation rules or display formats.
- **Builder Pattern**: For building complex objects like Match with multiple attributes.
- **Facade Pattern**: Simplify access to the system for clients/users.

Step 3: Code Sketch (Java)

3.1 Player and Team

```
class Player {
    private String name;
    private String role; // e.g., Batsman, Bowler, All-rounder

    public Player(String name, String role) {
        this.name = name;
```

```
        this.role = role;
    }

    // Getters and setters
}

class Team {
    private String name;
    private List<Player> players;

    public Team(String name) {
        this.name = name;
        this.players = new ArrayList<>();
    }

    public void addPlayer(Player player) {
        players.add(player);
    }

    // Getters and other methods
}
```

3.2 Score & Match

```
class Score {
    private int runs;
    private int wickets;
    private float overs;

    public Score() {
        this.runs = 0;
        this.wickets = 0;
        this.overs = 0.0f;
    }

    public void updateScore(int runs, int wickets, float overs) {
        this.runs = runs;
        this.wickets = wickets;
        this.overs = overs;
    }

    @Override
```

```
    public String toString() {
        return runs + "/" + wickets + " (" + overs + " overs)";
    }
}
```

3.3 Observer Pattern for Live Updates

```
// Observer interface
interface MatchObserver {
    void update(Score score);
}

// Subject interface
interface MatchSubject {
    void registerObserver(MatchObserver observer);
    void removeObserver(MatchObserver observer);
    void notifyObservers();
}
```

3.4 Match Class Implements Subject

```
class Match implements MatchSubject {
    private Team teamA;
    private Team teamB;
    private Score scoreTeamA;
    private Score scoreTeamB;
    private List<MatchObserver> observers;

    public Match(Team teamA, Team teamB) {
        this.teamA = teamA;
        this.teamB = teamB;
        this.scoreTeamA = new Score();
        this.scoreTeamB = new Score();
        this.observers = new ArrayList<>();
    }

    public void updateScore(Score newScoreTeamA, Score newScoreTeamB) {
        this.scoreTeamA = newScoreTeamA;
        this.scoreTeamB = newScoreTeamB;
        notifyObservers();
    }
```

```
    @Override
    public void registerObserver(MatchObserver observer) {
        observers.add(observer);
    }

    @Override
    public void removeObserver(MatchObserver observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers() {
        for (MatchObserver observer : observers) {
            // Notify observers with combined score info (simplified)
            observer.update(scoreTeamA); // For simplicity, just send one score
        }
    }

    // Additional methods: getMatchInfo(), getTeams(), etc.
}
```

3.5 Concrete Observer Example: User Interface Component

```
class LiveScoreDisplay implements MatchObserver {
    private Score currentScore;

    @Override
    public void update(Score score) {
        this.currentScore = score;
        display();
    }

    public void display() {
        System.out.println("Live Score Update: " + currentScore);
    }
}
```

3.6 Match Factory (Factory Pattern)

```
abstract class Match {
    // common attributes and methods
    abstract void startMatch();
```

```
}

class TestMatch extends Match {
    void startMatch() {
        System.out.println("Starting Test Match");
    }
}

class ODIMatch extends Match {
    void startMatch() {
        System.out.println("Starting ODI Match");
    }
}

class T20Match extends Match {
    void startMatch() {
        System.out.println("Starting T20 Match");
    }
}

class MatchFactory {
    public static Match createMatch(String type) {
        switch(type) {
            case "Test": return new TestMatch();
            case "ODI": return new ODIMatch();
            case "T20": return new T20Match();
            default: throw new IllegalArgumentException("Invalid match type");
        }
    }
}
```

3.7 Singleton for Match Manager

```
class MatchManager {
    private static MatchManager instance;
    private List<Match> ongoingMatches;

    private MatchManager() {
        ongoingMatches = new ArrayList<>();
    }

    public static synchronized MatchManager getInstance() {
```

```
        if (instance == null) {
            instance = new MatchManager();
        }
        return instance;
    }

    public void addMatch(Match match) {
        ongoingMatches.add(match);
    }

    public List<Match> getMatches() {
        return ongoingMatches;
    }
}
```

Step 4: Usage Example

```
public class CricinfoApp {
    public static void main(String[] args) {
        Team india = new Team("India");
        india.addPlayer(new Player("Virat Kohli", "Batsman"));
        Team australia = new Team("Australia");
        australia.addPlayer(new Player("Steve Smith", "Batsman"));

        Match match = new Match(india, australia);

        LiveScoreDisplay liveDisplay = new LiveScoreDisplay();
        match.registerObserver(liveDisplay);

        Score scoreIndia = new Score();
        scoreIndia.updateScore(150, 3, 20.0f);

        Score scoreAustralia = new Score();
        scoreAustralia.updateScore(140, 5, 20.0f);

        match.updateScore(scoreIndia, scoreAustralia);
    }
}
```

Summary of Design Patterns Used

Pattern	Role in Design
Observer	Notify live score updates to display components
Factory	Create different types of matches (Test, ODI)
Singleton	Manage ongoing matches centrally
Strategy	Could be added for scoring rules or display formats (extendable)
Builder	Could be used for complex match object construction
Facade	Could be added to provide a simple API for clients

---

41. `Design Facebook - a social network`

Designing **Facebook (a social network)** using Java and design patterns involves modeling core components like users, posts, friendships, likes, comments, newsfeed, notifications, and scalability patterns.

Below is a **modular, extensible, and object-oriented design** using Java along with appropriate **design patterns** to solve common problems in such systems.

✅ Step-by-Step Approach

⚙️ 1. Core Functionalities to Model

- User system (sign up, log in, friend requests)
- Posts (text, images, videos)
- Comments, Likes
- News Feed (show relevant posts)
- Notifications (when someone likes/comments)
- Messaging (optional)

🧱 2. Entities & Relationships

Entity	Description
`User`	Represents a Facebook user
`Post`	Can contain text/images/videos
`Comment`	Associated with a Post
`Like`	Tracks users who liked a post

`NewsFeed`	Shows list of posts
`Notification`	For actions like likes/comments

🧠 3. Design Patterns Used

Pattern	Used For
Observer	Notifications on likes/comments
Factory	Creating different types of posts
Strategy	Newsfeed sorting algorithms
Singleton	UserManager, FeedManager, NotificationManager
Builder	Building complex `Post` objects
Prototype	Cloning user settings/profile templates

📦 4. Java Code Modules (Simplified)

4.1 `User.java`

```
class User {
    private final String userId;
    private String name;
    private List<User> friends = new ArrayList<>();
    private List<Post> posts = new ArrayList<>();

    public User(String userId, String name) {
        this.userId = userId;
        this.name = name;
    }

    public void addFriend(User friend) {
        friends.add(friend);
        friend.friends.add(this); // mutual
    }

    public void createPost(Post post) {
        posts.add(post);
        FeedManager.getInstance().addPost(post);
    }

    public List<Post> getNewsFeed() {
        return FeedManager.getInstance().getFeedForUser(this);
```

```
    }

    public void notify(Notification notification) {
        System.out.println("Notification for " + name + ": " + notification.getMessage());
    }

    public String getName() {
        return name;
    }
}
```

4.2 Post.java (with Builder Pattern)

```
private final String postId;
    private final User author;
    private final String content;
    private final List<Like> likes = new ArrayList<>();
    private final List<Comment> comments = new ArrayList<>();

    private Post(PostBuilder builder) {
        this.postId = builder.postId;
        this.author = builder.author;
        this.content = builder.content;
    }

    public void addLike(User user) {
        likes.add(new Like(user));
        NotificationManager.getInstance().notifyUser(author,
            new Notification(user.getName() + " liked your post."));
    }

    public void addComment(Comment comment) {
        comments.add(comment);
        NotificationManager.getInstance().notifyUser(author,
            new Notification(comment.getAuthor().getName() + " commented: " +
            comment.getText()));
    }

    public String getContent() {
        return content;
    }
```

```
    public static class PostBuilder {
        private final String postId;
        private final User author;
        private String content;

        public PostBuilder(String postId, User author) {
            this.postId = postId;
            this.author = author;
        }

        public PostBuilder withContent(String content) {
            this.content = content;
            return this;
        }

        public Post build() {
            return new Post(this);
        }
    }
}
```

4.3 `Like.java` and `Comment.java`

```
class Like {
    private final User user;

    public Like(User user) {
        this.user = user;
    }
}

class Comment {
    private final User author;
    private final String text;

    public Comment(User author, String text) {
        this.author = author;
        this.text = text;
    }

    public User getAuthor() {
        return author;
```

```
    }

    public String getText() {
        return text;
    }
}
```

4.4 Observer Pattern – NotificationManager

```
class Notification {
    private final String message;

    public Notification(String message) {
        this.message = message;
    }

    public String getMessage() {
        return message;
    }
}

class NotificationManager {
    private static NotificationManager instance;

    private NotificationManager() {}

    public static synchronized NotificationManager getInstance() {
        if (instance == null) {
            instance = new NotificationManager();
        }
        return instance;
    }

    public void notifyUser(User user, Notification notification) {
        user.notify(notification);
    }
}
```

4.5 Singleton + Strategy Pattern – FeedManager

```
interface FeedStrategy {
    List<Post> generateFeed(User user, List<Post> allPosts);
```

```
}

class RecentPostsStrategy implements FeedStrategy {
    public List<Post> generateFeed(User user, List<Post> allPosts) {
        return allPosts.stream()
                .filter(p -> !p.equals(user))
                .sorted((a, b) -> -1) // mock for recent first
                .limit(10)
                .collect(Collectors.toList());
    }
}

class FeedManager {
    private static FeedManager instance;
    private List<Post> allPosts = new ArrayList<>();
    private FeedStrategy strategy = new RecentPostsStrategy();

    private FeedManager() {}

    public static FeedManager getInstance() {
        if (instance == null) {
            instance = new FeedManager();
        }
        return instance;
    }

    public void addPost(Post post) {
        allPosts.add(post);
    }

    public List<Post> getFeedForUser(User user) {
        return strategy.generateFeed(user, allPosts);
    }
}
```

🔍 Sample Usage

```
public class FacebookApp {
    public static void main(String[] args) {
        User alice = new User("u1", "Alice");
        User bob = new User("u2", "Bob");
```

```
        alice.addFriend(bob);

        Post post = new Post.PostBuilder("p1", alice)
                    .withContent("Hello Facebook!")
                    .build();
        alice.createPost(post);

        post.addLike(bob);
        post.addComment(new Comment(bob, "Welcome!"));

        System.out.println("Alice's News Feed:");
        for (Post p : alice.getNewsFeed()) {
            System.out.println(p.getContent());
        }
    }
}
```

✅ Summary of Components and Patterns

Component	Pattern(s) Used
`User`	Simple model, Observer (indirect)
`Post`	Builder for object creation
`FeedManager`	Singleton + Strategy for feed algo
`NotificationManager`	Singleton + Observer
`PostFactory`	(Optional) Factory for text/video
`Comment/Like`	Plain value objects

🧩 Possible Extensions

- **Prototype**: Clone user templates (e.g., default settings)
- **Command Pattern**: Undo/redo actions on posts
- **Decorator Pattern**: Enhance post with metadata (sponsored, shared)

12. `Design LinkedIn`

Designing **LinkedIn** using Java and design patterns involves building a **modular, extensible** backend system that supports users, professional profiles, connections, job postings, endorsements, notifications, and a newsfeed.

🧠 Core Features of LinkedIn

1. User registration/login
2. Professional profiles (skills, experience, education)
3. Connect with users (request/accept)
4. Posts, likes, comments
5. Job postings & applications
6. Notifications
7. News feed (based on network activity)

🧱 Key Components (Entity Overview)

Component	Description
`User`	Registered LinkedIn user
`Profile`	Skills, experience, education
`Connection`	Connection between users
`Post`	Shared content (text, image, article)
`Comment` , `Like`	Engagements
`JobPosting`	Recruiter-created job offers
`Application`	User-applied jobs
`Notification`	System alerts (like accepted request)
`Feed`	Curated content

🧩 Design Patterns Used

Pattern	Usage
Builder	Building complex objects like `Profile`
Singleton	Services like `FeedManager` , `JobPortal`
Observer	For notifications (connect, job posted)
Strategy	Feed generation algorithm
Factory	Post creation (text, article, image post)
Prototype	Cloning profiles (optional)

✅ Class Design and Java Code

---

1. `User` + Connection Logic

```
class User {
    private final String userId;
    private final String name;
    private final Profile profile;
    private final Set<User> connections = new HashSet<>();
    private final List<Post> posts = new ArrayList<>();

    public User(String userId, String name, Profile profile) {
        this.userId = userId;
        this.name = name;
        this.profile = profile;
    }

    public void sendConnectionRequest(User other) {
        other.receiveConnectionRequest(this);
    }

    public void receiveConnectionRequest(User fromUser) {
        // Accept all for simplicity
        connections.add(fromUser);
        fromUser.connections.add(this);
        NotificationService.getInstance().notifyUser(this,
            new Notification(fromUser.getName() + " connected with you."));
    }

    public void createPost(Post post) {
        posts.add(post);
        FeedManager.getInstance().addPost(post);
    }

    public List<Post> getFeed() {
        return FeedManager.getInstance().generateFeed(this);
    }

    public Profile getProfile() {
        return profile;
    }

    public String getName() {
        return name;
    }

    public Set<User> getConnections() {
```

```
        return connections;
    }
}
```

2. `Profile` with Builder Pattern

```
class Profile {
    private final List<String> skills;
    private final List<String> experience;
    private final List<String> education;

    private Profile(ProfileBuilder builder) {
        this.skills = builder.skills;
        this.experience = builder.experience;
        this.education = builder.education;
    }

    public static class ProfileBuilder {
        private List<String> skills = new ArrayList<>();
        private List<String> experience = new ArrayList<>();
        private List<String> education = new ArrayList<>();

        public ProfileBuilder addSkill(String skill) {
            skills.add(skill);
            return this;
        }

        public ProfileBuilder addExperience(String exp) {
            experience.add(exp);
            return this;
        }

        public ProfileBuilder addEducation(String edu) {
            education.add(edu);
            return this;
        }

        public Profile build() {
            return new Profile(this);
        }
    }
}
```

3. `Post`, `Comment`, `Like` using Factory

```
interface Post {
    String getContent();
    User getAuthor();
}

class TextPost implements Post {
    private final String content;
    private final User author;

    public TextPost(String content, User author) {
        this.content = content;
        this.author = author;
    }

    public String getContent() {
        return content;
    }

    public User getAuthor() {
        return author;
    }
}

class PostFactory {
    public static Post createTextPost(String content, User author) {
        return new TextPost(content, author);
    }
}
```

4. Notification via Observer Pattern

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

class NotificationService {
    private static NotificationService instance;
    private NotificationService() {}
```

```
    public static NotificationService getInstance() {
        if (instance == null) instance = new NotificationService();
        return instance;
    }

    public void notifyUser(User user, Notification notification) {
        System.out.println("To " + user.getName() + ": " + notification.getMessage());
    }
}
```

5. `FeedManager` using Strategy Pattern

```
interface FeedStrategy {
    List<Post> getFeed(User user, List<Post> allPosts);
}

class RecentActivityStrategy implements FeedStrategy {
    public List<Post> getFeed(User user, List<Post> allPosts) {
        return allPosts.stream()
            .filter(p -> user.getConnections().contains(p.getAuthor()))
            .limit(10)
            .collect(Collectors.toList());
    }
}

class FeedManager {
    private static FeedManager instance;
    private final List<Post> allPosts = new ArrayList<>();
    private FeedStrategy strategy = new RecentActivityStrategy();

    private FeedManager() {}

    public static FeedManager getInstance() {
        if (instance == null) instance = new FeedManager();
        return instance;
    }

    public void addPost(Post post) {
        allPosts.add(post);
    }

    public List<Post> generateFeed(User user) {
```

```
        return strategy.getFeed(user, allPosts);
    }

    public void setStrategy(FeedStrategy strategy) {
        this.strategy = strategy;
    }
}
```

6. Job Posting System (Recruiter-side)

```
class JobPosting {
    private final String jobId;
    private final String title;
    private final String company;
    private final User recruiter;

    public JobPosting(String jobId, String title, String company, User recruiter) {
        this.jobId = jobId;
        this.title = title;
        this.company = company;
        this.recruiter = recruiter;
    }

    public String getTitle() {
        return title;
    }

    public User getRecruiter() {
        return recruiter;
    }
}

class JobPortal {
    private static JobPortal instance;
    private final List<JobPosting> postings = new ArrayList<>();

    private JobPortal() {}

    public static JobPortal getInstance() {
        if (instance == null) instance = new JobPortal();
        return instance;
```

```
    }

    public void postJob(JobPosting job) {
        postings.add(job);
        NotificationService.getInstance().notifyUser(job.getRecruiter(),
            new Notification("Job posted: " + job.getTitle()));
    }

    public List<JobPosting> getJobs() {
        return postings;
    }
}
```

📌 Sample Main Function

```
public class LinkedInApp {
    public static void main(String[] args) {
        Profile aliceProfile = new Profile.ProfileBuilder()
            .addSkill("Java")
            .addExperience("Google")
            .addEducation("IIT Bombay")
            .build();

        User alice = new User("u1", "Alice", aliceProfile);

        Profile bobProfile = new Profile.ProfileBuilder()
            .addSkill("Spring")
            .addExperience("Amazon")
            .addEducation("IIT Madras")
            .build();

        User bob = new User("u2", "Bob", bobProfile);

        alice.sendConnectionRequest(bob);

        Post post = PostFactory.createTextPost("Excited to join Google!", alice);
        alice.createPost(post);

        JobPosting job = new JobPosting("j1", "Backend Engineer", "Amazon", bob);
        JobPortal.getInstance().postJob(job);
```

```
        System.out.println("Alice's feed:");
        for (Post p : alice.getFeed()) {
            System.out.println(p.getAuthor().getName() + ": " + p.getContent());
        }
    }
}
```

✅ Summary of Patterns Used

Use Case	Pattern
Build Profile	Builder
Unique Managers	Singleton
Notify users	Observer
Custom feed sort	Strategy
Post creation	Factory
Clone profile	Prototype (optional)

🚀 Optional Extensions

- Add **JPA layer** for persistence
- Use **Spring Boot** + REST controllers
- Add **ElasticSearch** for profile search
- Integrate **WebSockets** for real-time updates