Liceria & Co.

ARCHITECTURE PROBLEMS

Ferdous Gulzar
FA22-BSE-034

Please Read Till END

Architectural Problems and Solutions in Software Systems

Part 1: Identifying Major Architectural Problems and Solutions

1. Scalability Issues

Problem:
A monolithic architecture struggles to handle increasing user loads or data volumes, leading to performance degradation.

Impact:

- System crashes during peak traffic.
- Slow response times affecting user experience.
- Revenue loss due to unsatisfied users.

Solution:
Switch to **Microservices Architecture**:

- Break the system into smaller, independent services (e.g., user service, order service).
- Each service scales independently based on demand.
- Example: Netflix scaled its operations by adopting a microservices approach, improving system availability.

---

2. Performance Bottlenecks

Problem:
Certain components in the system (e.g., a single database or API) slow down the entire application.

Impact:

- Increased latency, especially during high traffic.
- Users face delays in loading pages or receiving data.
- Limits the system's growth potential.

Solution:
Use **Caching** and **Load Balancing**:

- Cache frequently accessed data (e.g., with Redis or Memcached).
- Distribute incoming requests across multiple servers using a load balancer.
- Example: Amazon uses caching to speed up product searches and load balancers to handle millions of requests seamlessly.

---

3. Tight Coupling

Problem:
Modules or components in the system are heavily dependent on each other, making it difficult to update or replace parts without affecting the entire system.

Impact:

- Hard to maintain or extend the system.
- Increases the cost and time for updates.
- Bugs in one module can cascade to others.

Solution:
Adopt **Dependency Injection (DI)** and **Modular Architecture**:

- Decouple modules by injecting dependencies at runtime (e.g., using Spring Framework in Java).
- Use a layered architecture to separate concerns.
- Example: A tightly coupled e-commerce system was modularized by decoupling the payment and notification modules using DI.

---

4. Data Handling Issues

Problem:
Centralized data storage cannot handle massive volumes of data efficiently, leading to slow query responses and system failures.

Impact:

- Latency in retrieving data.
- Difficulty in maintaining data integrity.
- Risk of data loss during crashes.

Solution:
Implement **Distributed Data Stores**:

- Use databases like MongoDB or Cassandra to store data across multiple nodes.
- Employ partitioning and replication for high availability and fast query responses.
- Example: Facebook's chat feature transitioned to a distributed data store, enabling faster message delivery.

---

5. Security Vulnerabilities

Problem:
Weak security measures in software architecture leave the system exposed to attacks like SQL injection, cross-site scripting (XSS), and data breaches.

Impact:

- Loss of sensitive user data.
- Legal penalties and damage to reputation.
- Financial losses from cyberattacks.

Solution:
Introduce a **Secure API Gateway and Encrypted Communication**:

- Use an API gateway to validate requests and manage access control.
- Employ HTTPS and data encryption for secure data transfer.
- Example: A payment processing system improved security by introducing OAuth-based authentication and encrypted API requests.

Part 2: Replicating and Solving a Problem

Scenario: Tight Coupling in a Notification System

Problem Definition

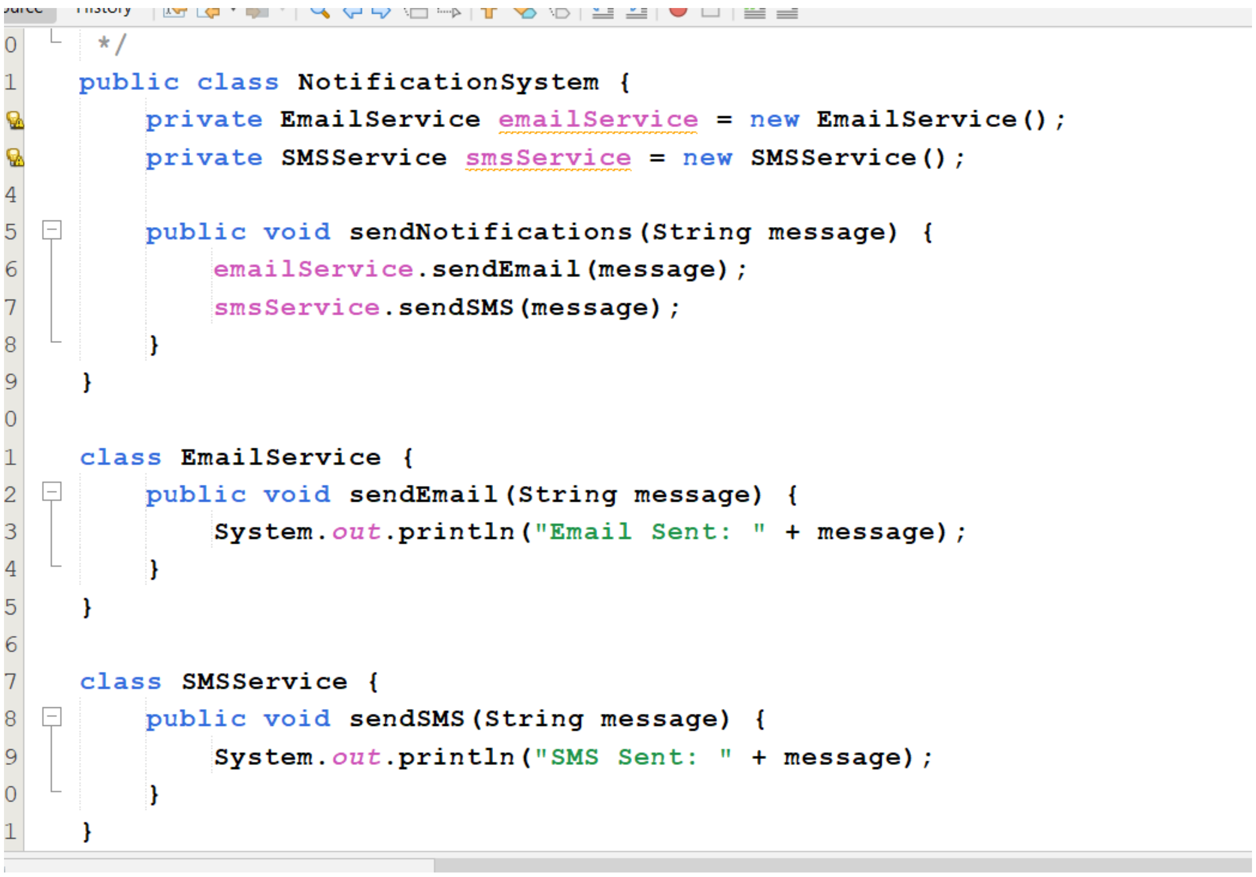
In tightly coupled systems, components depend directly on one another, making it challenging to update, extend, or test the system. This issue often arises when classes instantiate their dependencies rather than relying on abstraction or external configuration.

Example: A Notification System

Imagine a **Notification System** responsible for sending messages through different channels (e.g., Email and SMS). Initially, the system directly creates instances of `EmailService` and `SMSService` within its main logic. This approach hardwires the dependencies, causing tight coupling.

Impacts of Tight Coupling

1. **Hard to Extend**: Adding a new notification service (e.g., Push Notifications) requires modifying the `NotificationSystem` class, violating the **Open/Closed Principle**.
2. **Difficult Maintenance**: If an existing service needs changes (e.g., switching `SMSService` to a new provider), the core system logic must also be updated.
3. **Poor Testability**: Mocking or replacing the `EmailService` or `SMSService` for testing purposes becomes cumbersome.
4. **Decreased Flexibility**: The system cannot dynamically decide which services to use at runtime, limiting configurability.



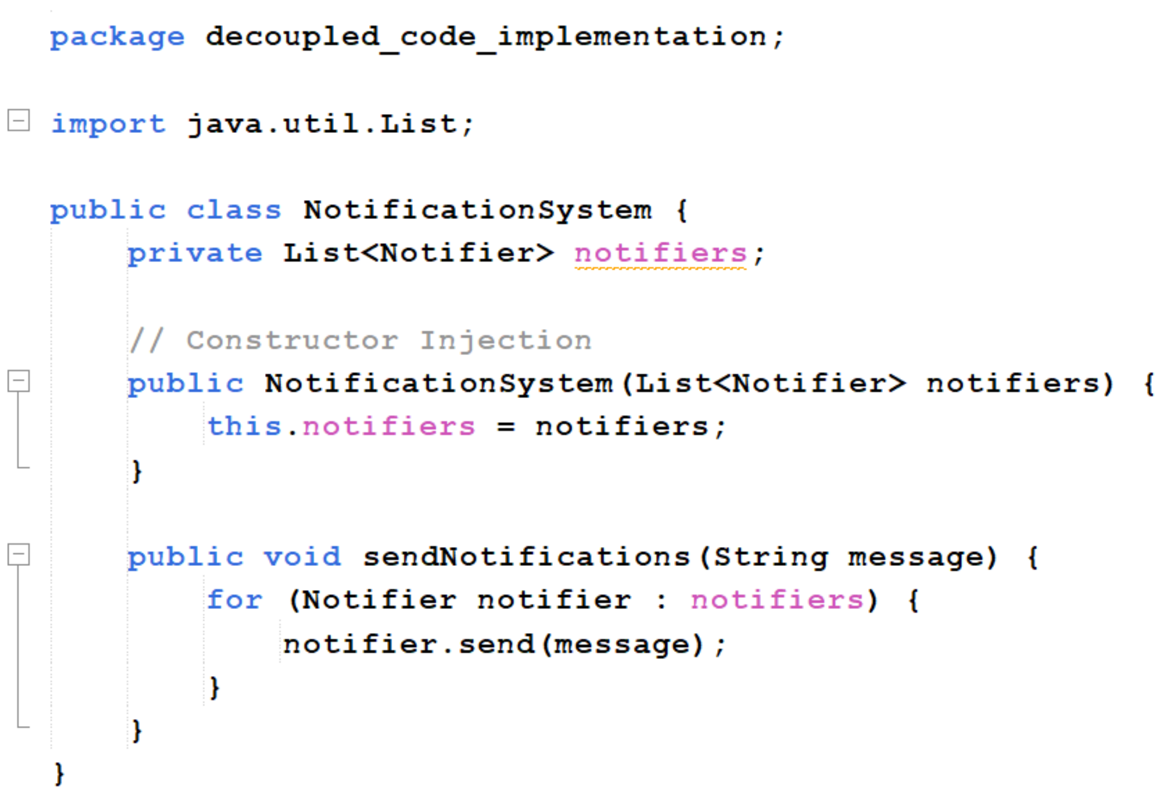
```
    */
public class NotificationSystem {
    private EmailService emailService = new EmailService();
    private SMSService smsService = new SMSService();

    public void sendNotifications(String message) {
        emailService.sendEmail(message);
        smsService.sendSMS(message);
    }
}

class EmailService {
    public void sendEmail(String message) {
        System.out.println("Email Sent: " + message);
    }
}

class SMSService {
    public void sendSMS(String message) {
        System.out.println("SMS Sent: " + message);
    }
}
```

Solution: Decoupled Notification System

To solve this problem, we use the **Dependency Injection (DI)** design pattern. The key idea is to decouple the `NotificationSystem` from its dependencies by using an interface (`Notifier`) and injecting dependencies at runtime.

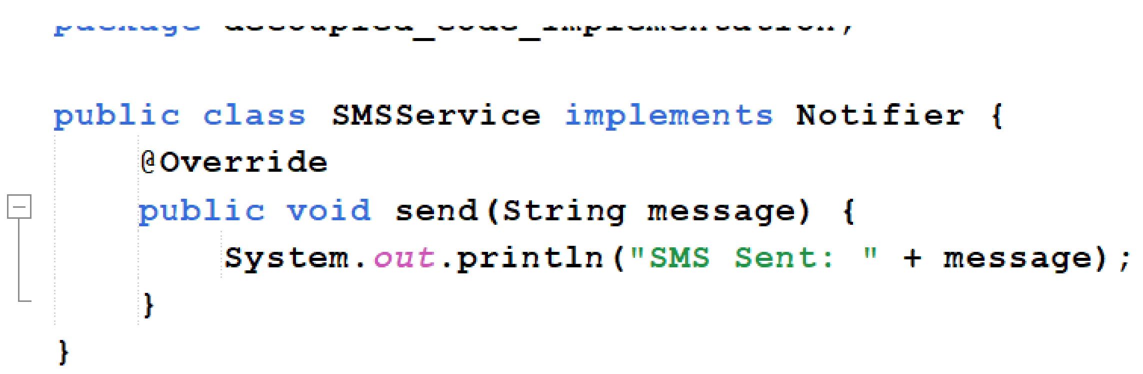
```
package decoupled_code_implementation;

public interface Notifier {
    void send(String message);
}
```

NotificationSystem class



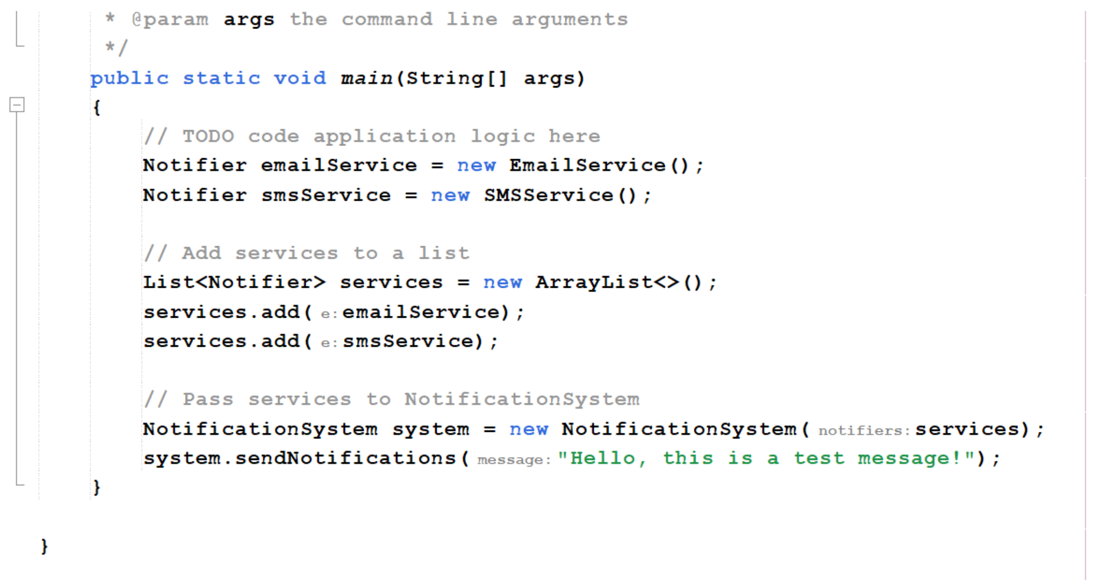
```java
package decoupled_code_implementation;

import java.util.List;

public class NotificationSystem {
    private List<Notifier> notifiers;

    // Constructor Injection
    public NotificationSystem(List<Notifier> notifiers) {
        this.notifiers = notifiers;
    }

    public void sendNotifications(String message) {
        for (Notifier notifier : notifiers) {
            notifier.send(message);
        }
    }
}
```

SSM SERVICSE CLASS:



```java
package decoupled_code_implementation;

public class SMSService implements Notifier {
    @Override
    public void send(String message) {
        System.out.println("SMS Sent: " + message);
    }
}
```

Main:



```
     * @param args the command line arguments
     */
    public static void main(String[] args)
    {
        // TODO code application logic here
        Notifier emailService = new EmailService();
        Notifier smsService = new SMSService();

        // Add services to a list
        List<Notifier> services = new ArrayList<>();
        services.add( e: emailService);
        services.add( e: smsService);

        // Pass services to NotificationSystem
        NotificationSystem system = new NotificationSystem( notifiers: services);
        system.sendNotifications( message: "Hello, this is a test message!");
    }

}
```

Benefits of the Solution

1. **Scalability**:
   - Adding new services (e.g., `PushNotificationService`) requires no change to the core `NotificationSystem`.
   - Example: Simply create a new class implementing `Notifier`, add it to the list in `Main`, and it will automatically work with the system.
2. **Flexibility**:
   - Dependencies (`EmailService`, `SMSService`) are passed to the `NotificationSystem` dynamically, allowing runtime configuration.
   - Example: You could send only emails or only SMS depending on the use case.
3. **Testability**:
   - Mocking `Notifier` becomes easy for unit tests, enabling independent testing of the `NotificationSystem`.
4. **Maintainability**:
   - Changes in one service don’t impact others, reducing the risk of bugs and simplifying updates.

Architectural Problems