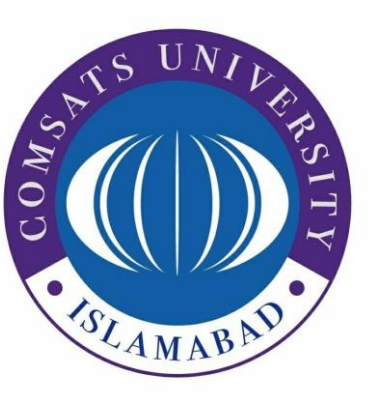
Comsats university Islamabad, Abbottabad campus



SDA

Lab Assignment 02

Name:

Waqas Ahmed (FA22-BSE-009)

Submitted To:

Mukhtiar Zamin

Report: Problems of software in architecture

Software Problems and Solutions Report

In the field of software development, several critical architectural challenges can arise, particularly when systems scale or when best practices are not followed. Below is a summary of five common architectural problems faced by developers and architects, along with the solutions implemented to address them, including the principles that were applied.

1. **Inadequate Modularization**
    - **Problem**: In monolithic systems, code becomes highly tangled, making maintenance difficult. Adding new features can unintentionally affect other parts of the system due to tight coupling.
    - **Who Proposed**: Software developers and system architects.
    - **Solution**: The system was divided into small, independent modules. Each module handled a specific function with minimal dependencies on other parts. This approach was implemented using **microservices architecture** or **modular design patterns**, enabling loose coupling between components and simplifying scalability and maintenance.
    - **Principle Used**: **Single Responsibility Principle (SRP)** — Each module is designed to have a single responsibility, improving maintainability and flexibility.

2. **Lack of Proper Layering (Separation of Concerns)**
    - **Problem**: In systems with poorly structured architecture, presentation logic, business logic, and data access are often intertwined. This leads to difficulties in maintaining and scaling the system.
    - **Who Proposed**: Architects.
    - **Solution**: The architecture was refactored into a layered **design** (e.g., **MVC (Model-View-Controller)**), where each layer had a distinct responsibility. The separation of concerns allowed for easier maintenance, improved scalability, and the ability to manage different parts of the system independently.
    - **Principle Used**:**Separation of Concerns (SoC)** — Dividing the system into distinct layers to reduce complexity.

3. **Insufficient Error Handling Mechanisms**
    - **Problem**: Inadequate error handling leads to system crashes, especially in distributed systems where failures can cascade through multiple components. Lack of centralized logging further complicates debugging.
    - **Who Proposed**: Software testing and quality assurance teams.
    - **Solution**: Exception handling was implemented across the system, and centralized **logging and monitoring** tools were introduced. Fault-tolerant design patterns such as the **Circuit Breaker** were employed to ensure the system could recover gracefully from failures without affecting the entire system.

- **Principle Used**: **Fail Fast —** Detect and handle errors early to prevent system-wide failures.

4. **Monolithic Systems with Slow Release Cycles**
   - **Problem**: Large monolithic systems result in slow release cycles, where adding new features or bug fixes takes considerable time. This delay affects product delivery and customer satisfaction.
   - **Who Proposed**: Business stakeholders and developers.
   - **Solution**: The system was migrated to **microservices architecture**, where the system was decomposed into small, independent services. This allowed for quicker release cycles, easier scaling, and the ability to update individual services without downtime or delays.
   - **Principle Used**: **Autonomy of Components —** Independent services that can be deployed and scaled separately for faster releases.

5. **Data Consistency Across Distributed Databases (Eventual Consistency Issue)**
   - **Problem**: In distributed systems using multiple databases, data consistency can become problematic. Changes made in one system might not immediately reflect in others, leading to potential data corruption or stale data.
   - **Who Proposed**: Developers and system architects.
   - **Solution**: Strong consistency was enforced by implementing the **CAP theorem** and using techniques like **event sourcing** and **CQRS (Command Query Responsibility Segregation)**. In addition, **two-phase commit** and atomic transactions were used to maintain consistency. For systems requiring high availability, hybrid approaches were applied, such as combining **ACID-compliant databases** for critical operations with eventually consistent databases for non-critical data.
   - **Principle Used**: **Eventual Consistency** — Allows temporary inconsistency for improved availability and partition tolerance.

Insufficient Error Handling Mechanisms(through coding implementation)

Introduction: In modern software systems, particularly distributed systems, ensuring reliability and resilience is essential. Effective error handling and fault tolerance mechanisms like the **Circuit Breaker Pattern** are critical for preventing system failure, ensuring smooth user experiences, and maintaining system stability. This report outlines the solutions for improving error handling and implementing a circuit breaker to enhance the robustness of a system.

1. Error Handling Solution:

Before implementing error handling, the system failed silently or showed generic error messages without providing any useful context, making it difficult for developers to pinpoint the issues. This caused delays in identifying and resolving problems, and led to an inconsistent user experience.

The solution involved implementing specific exception handling to capture and log errors in a meaningful way, along with providing appropriate feedback to users.

- **Specific Exceptions**: Errors were categorized by type (e.g., `PaymentGatewayException`, `UserNotFoundException`, `PaymentProcessingException`) to identify and handle specific failure scenarios effectively.
- **Meaningful Logging**: Logging mechanisms were integrated to capture detailed error information, allowing developers to trace issues more efficiently. The logs provide insights into the nature of the errors and their impact on the system.
- **User-Friendly Error Messages**: Instead of vague or generic error messages, users were now provided with clear and actionable feedback. For instance, "Unable to process payment, please try again later" or "An unexpected error occurred. Please contact support."

The main principle used here is **Robustness Principle** (also known as the "Fail-Fast" principle), which encourages systems to fail gracefully and provide enough information to facilitate easier debugging and recovery.

Before: The system might have failed silently or just thrown a generic error message without context, making it hard to determine what went wrong.
For example:

Before Error Handling (Java)

```
public class PaymentProcessor {

   public void processPayment(PaymentInfo paymentInfo) {

      paymentGateway.process(paymentInfo);  // No error handling

   }

}
```

After Error Handling (Java)

```
import java.util.logging.Logger;
```

```
public class PaymentProcessor {
    private static final Logger logger = Logger.getLogger(PaymentProcessor.class.getName());

    public void processPayment(PaymentInfo paymentInfo) {
        try {
            paymentGateway.process(paymentInfo);  // Attempt to process payment
        } catch (PaymentGatewayException e) {
            logger.severe("Payment failed for user " + paymentInfo.getUserId() + ": " +
e.getMessage());
            throw new PaymentProcessingException("Unable to process payment.");
        } catch (Exception e) {
            logger.severe("Unexpected error: " + e.getMessage());
            throw new SystemException("Unexpected error occurred.");
        }
    }
}
```

Circuit Breaker Pattern (Java)

For Java, we can use a library like **Resilience4j** to implement the Circuit Breaker pattern. Here's an example using Resilience4j:

1. First, add Resilience4j dependencies to your pom.xml:

```
<dependency>
    <groupId>io.github.resilience4j</groupId>
    <artifactId>resilience4j-circuitbreaker</artifactId>
    <version>1.7.0</version>
```

```
</dependency>
```

2)Then, implement the Circuit Breaker in your PaymentProcessor class:

```
import io.github.resilience4j.circuitbreaker.CircuitBreaker;
import io.github.resilience4j.circuitbreaker.CircuitBreakerConfig;
import java.util.function.Supplier;

public class PaymentProcessor {
    private final CircuitBreaker circuitBreaker;

    public PaymentProcessor() {
        CircuitBreakerConfig config = CircuitBreakerConfig.custom()
                .failureRateThreshold(50)
                .waitDurationInOpenState(java.time.Duration.ofMillis(5000))
                .build();
        this.circuitBreaker = CircuitBreaker.of("paymentProcessorCircuitBreaker", config);
    }

    public void processPayment(PaymentInfo paymentInfo) {
        try {
            Supplier<Void> paymentSupplier = () -> {
                paymentGateway.process(paymentInfo);
                return null;
```

```
            };
            circuitBreaker.executeRunnable(paymentSupplier);  // Execute with circuit breaker
        } catch (Exception e) {
            logger.severe("Payment failed: " + e.getMessage());
            throw new SystemException("Service temporarily unavailable.");
        }
    }
}
```