Q1: Why is adherence to design principles like SOLID critical in modern software development, especially in Agile and large-team environments?

In Agile Environments:

- Rapid Iteration: SOLID principles enable faster feature development and modifications by creating modular code that can be changed without breaking existing functionality
- Continuous Integration: Well-designed code following SOLID principles reduces integration conflicts and makes automated testing more effective
- Adaptability: Agile requires frequent requirement changes, and SOLID principles ensure the codebase can accommodate these changes without major refactoring

Large-Team Environments:

- Parallel Development: When multiple developers work on the same codebase, SOLID principles reduce dependencies between components, allowing teams to work independently
- Code Maintainability: Clear responsibilities and interfaces make it easier for different team members to understand and modify code written by others
- Reduced Technical Debt: Following SOLID principles prevents the accumulation of tightly-coupled, hard-to-maintain code that becomes increasingly expensive to modify

Q2: Define "Design Rot" and explain how symptoms like Rigidity, Fragility, and Viscosity affect the maintainability of a codebase.

Design Rot is the gradual degradation of software design quality over time, typically caused by repeated modifications that violate good design principles.

Symptoms of Design Rot:

Rigidity:

- Definition: The tendency for software to be difficult to change, where even small modifications require changes to many dependent modules
- Impact on Maintainability:
    - Simple feature requests become major undertakings
    - Developers avoid making necessary changes due to fear of breaking existing functionality

    - Example: Changing a database field type requires modifications in 20+ different classes

Fragility:

- Definition: The tendency for software to break in unexpected places when changes are made
- Impact on Maintainability:
    - Bug fixes introduce new bugs in seemingly unrelated areas
    - Testing becomes exponentially more complex as the system grows
    - Example: Fixing a user interface bug causes the payment system to malfunction

Viscosity:

- Definition: The resistance to doing the right thing, where it's easier to implement hacks than proper solutions
- Impact on Maintainability:
    - Shortcuts accumulate over time, creating technical debt
    - The "easy" path diverges from the correct architectural path
    - Example: Adding a feature by copying and modifying existing code rather than creating a reusable component

Q3: Can SOLID principles fully prevent design rot? Justify your answer with examples.

No, SOLID principles cannot fully prevent design rot, they significantly reduce its likelihood and severity.

Why SOLID Principles Help:

- They provide clear guidelines for maintaining clean architecture
- They promote loose coupling and high cohesion
- They make code more testable and modular

Limitations and Examples:

1. Human Factors:

- Even with knowledge of SOLID principles, developers may cut corners under tight deadlines, Not all team members may fully understand or consistently apply these principles
- Example: A junior developer might create a God class that violates SRP because they don't recognize the multiple responsibilities

2. Evolving Requirements:

- Scope Creep: Requirements that fundamentally change the system's purpose can make even well-designed code obsolete, Business Model Changes: Architectural decisions that were correct initially may become impediments
- Example: An e-commerce system designed for single-vendor sales struggles when pivoting to a marketplace model

3. External Dependencies:

- Third-party Library Changes: External APIs or libraries can force architectural compromises.Technology Obsolescence: Framework updates may require significant refactoring
- Example: A payment system designed around a specific API must be restructured when that service is discontinued

Single Responsibility Principle (SRP)

Scenario 1:

Violated Principle: Single Responsibility Principle (SRP)

How it's violated: The StudentService class has multiple responsibilities:

1. Student enrollment (business logic)
2. Certificate printing (presentation/formatting)
3. Email communication (messaging/notification)

This violates SRP because the class has multiple reasons to change: enrollment process changes, certificate format changes, or email system changes.

```
class StudentService {
    void enrollStudent(Student s) { ... }
}
class CertificateService {
    void printCertificate(Student s) { ... }
}
class EmailService {
    void emailCertificate(Student s) { ... }
}
// Coordinator class if needed
class StudentEnrollmentFacade {
    private StudentService studentService;
    private CertificateService certificateService;
    private EmailService emailService;

    void completeEnrollment(Student s) {
        studentService.enrollStudent(s);
        certificateService.printCertificate(s);
        emailService.emailCertificate(s);
    }
```

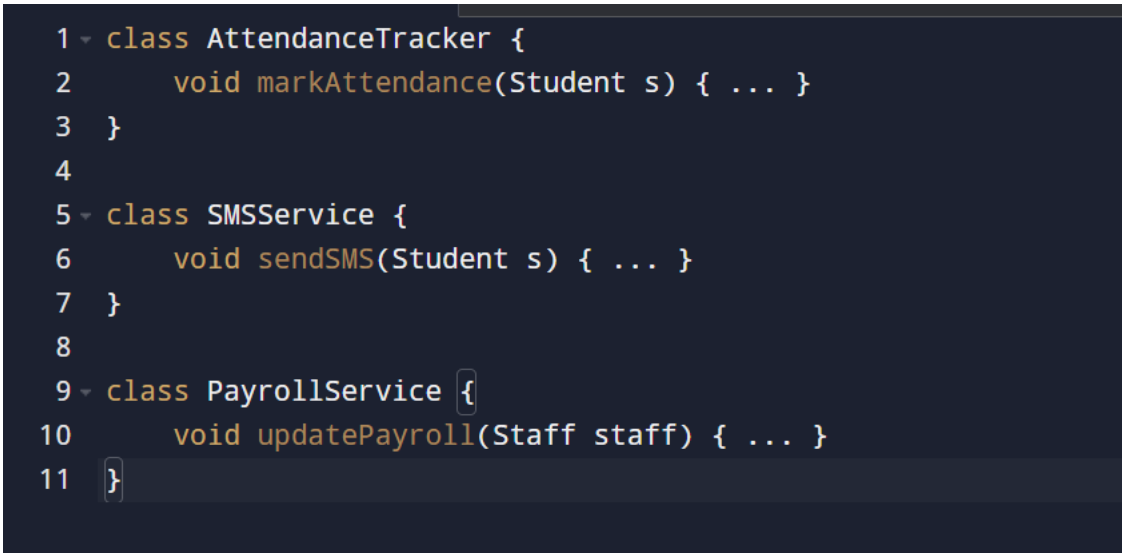
Scenario 2:

Violated Principle: Single Responsibility Principle (SRP)

How it's violated: The AttendanceTracker class has three distinct responsibilities:

1. Tracking student attendance
2. SMS communication
3. Staff payroll management

These responsibilities belong to different domains and would change for different reasons.



```
class AttendanceTracker {
    void markAttendance(Student s) { ... }
}

class SMSService {
    void sendSMS(Student s) { ... }
}

class PayrollService {
    void updatePayroll(Staff staff) { ... }
}
```

Open/Closed Principle (OCP)

Scenario 1:

Violated Principle: Open/Closed Principle (OCP)

How it's violated: The class is not open for extension but requires modification to add new customer types. Every new customer type requires changing the existing code, violating the "closed for modification" aspect.

```
interface DiscountStrategy {
    double getDiscount();
}

class RegularCustomerDiscount implements DiscountStrategy {
    public double getDiscount() { return 0.1; }
}

class VIPCustomerDiscount implements DiscountStrategy {
    public double getDiscount() { return 0.2; }
}

class PremiumCustomerDiscount implements DiscountStrategy {
    public double getDiscount() { return 0.3; }
}

class DiscountCalculator {
    double calculateDiscount(DiscountStrategy strategy) {
        return strategy.getDiscount();
    }
}
```

Scenario 2

Violated Principle: Open/Closed Principle (OCP)

How it's violated: Adding new print formats requires modifying the existing InvoicePrinter class, making it not closed for modification.

```
interface InvoiceFormatter {
    void print(Invoice invoice);
}
class PDFInvoiceFormatter implements InvoiceFormatter {
    public void print(Invoice invoice) {
        // PDF formatting logic
    }
}
class HTMLInvoiceFormatter implements InvoiceFormatter {
    public void print(Invoice invoice) {
        // HTML formatting logic
    }
}
class XMLInvoiceFormatter implements InvoiceFormatter {
    public void print(Invoice invoice) {
        // XML formatting logic
    }
}
class InvoicePrinter {
    void print(Invoice invoice, InvoiceFormatter formatter) {
        formatter.print(invoice);
    }
}
```

Section 3: Complex Refactoring Challenges

Scenario 1: LibraryManager

Violated Principles: Multiple violations

- SRP: The class handles book management, student interactions, fee calculations, reporting, and email communications
- OCP: Adding new fee calculation methods or report types would require modifying the class

How it's violated: The LibraryManager class has at least 6 different responsibilities:

1. Book inventory management
2. Book lending operations

3. Fee calculation
4. Report generation
5. Email notifications
6. Main application coordination

```java
// SRP: Each class has a single responsibility
class BookInventoryService {
    void addBook(Book b) {
        System.out.println("Book added: " + b.title);
    }

    void removeBook(Book b) {
        System.out.println("Book removed: " + b.title);
    }
}

class LendingService {
    void issueBook(Student s, Book b) {
        System.out.println("Book '" + b.title + "' issued to student: "
            + s.name);
    }
}

class FeeCalculationService {
    void calculateLateFees(Student s) {
        System.out.println("Late fees calculated for: " + s.name);
    }
}

class ReportService {
    void generateReport() {
        System.out.println("Library report generated.");
    }
}
```

```java
class NotificationService {
    void sendOverdueEmail(Student s) {
        System.out.println("Overdue email sent to: " + s.name);
    }
}
// Facade pattern to coordinate services
class LibraryFacade {
    private BookInventoryService inventoryService;
    private LendingService lendingService;
    private FeeCalculationService feeService;
    private ReportService reportService;
    private NotificationService notificationService;

    public LibraryFacade() {
        this.inventoryService = new BookInventoryService();
        this.lendingService = new LendingService();
        this.feeService = new FeeCalculationService();
        this.reportService = new ReportService();
        this.notificationService = new NotificationService();
    }

    // Delegate operations to appropriate services
    void addBook(Book b) {
        inventoryService.addBook(b);
    }

    void issueBook(Student s, Book b) {
        lendingService.issueBook(s, b);
```

```
    void issueBook(Student s, Book b) {
        lendingService.issueBook(s, b);
    }

    void processOverdueBooks(Student s) {
        feeService.calculateLateFees(s);
        notificationService.sendOverdueEmail(s);
    }

    void generateLibraryReport() {
        reportService.generateReport();
    }

    void removeBook(Book b) {
        inventoryService.removeBook(b);
    }
}

// Updated main class
class LibraryApplication {
    public static void main(String[] args) {
        LibraryFacade library = new LibraryFacade();

        Student student = new Student("Alice");
        Book book = new Book("Design Patterns");

        library.addBook(book);
        library.issueBook(student, book);
        library.processOverdueBooks(student);
```

Scenario 2: PaymentProcessor

Violated Principles:

- OCP: Adding new payment methods requires modifying existing code
- SRP: The class handles multiple payment method implementations

How it's violated: The PaymentProcessor class must be modified every time a new payment method is added. The if-else chain violates OCP because the class is not closed for modification.

```
class GooglePayPayment implements PaymentMethod {
    public void processPayment(double amount) {
        System.out.println("Processing Google Pay payment: " + amount);
    }
}

// Factory pattern for creating payment methods
class PaymentMethodFactory {
    static PaymentMethod createPaymentMethod(String method) {
        switch (method) {
            case "CreditCard": return new CreditCardPayment();
            case "PayPal": return new PayPalPayment();
            case "Crypto": return new CryptoPayment();
            case "ApplePay": return new ApplePayPayment();
            case "GooglePay": return new GooglePayPayment();
            default: throw new UnsupportedOperationException
                ("Unsupported payment method: " + method);
        }
    }
}

// Context class that uses the strategy
class PaymentProcessor {
    void processPayment(double amount, String methodType) {
        try {
            PaymentMethod method = PaymentMethodFactory
                .createPaymentMethod(methodType);
            method.processPayment(amount);
```

```
// Strategy pattern implementing OCP
interface PaymentMethod {
    void processPayment(double amount);
}

class CreditCardPayment implements PaymentMethod {
    public void processPayment(double amount) {
        System.out.println("Processing credit card payment: " + amount);
    }
}

class PayPalPayment implements PaymentMethod {
    public void processPayment(double amount) {
        System.out.println("Processing PayPal payment: " + amount);
    }
}

class CryptoPayment implements PaymentMethod {
    public void processPayment(double amount) {
        System.out.println("Processing crypto payment: " + amount);
    }
}

// New payment methods can be added without modifying existing code
class ApplePayPayment implements PaymentMethod {
    public void processPayment(double amount) {
        System.out.println("Processing Apple Pay payment: " + amount);
    }
}
```

```
            method.processPayment(amount);
        } catch (UnsupportedOperationException e) {
            System.out.println(e.getMessage());
        }
    }

    // Overloaded method for direct strategy usage
    void processPayment(double amount, PaymentMethod method) {
        method.processPayment(amount);
    }
}

class PaymentApplication {
    public static void main(String[] args) {
        PaymentProcessor processor = new PaymentProcessor();

        // Using string-based method selection
        processor.processPayment(100.0, "CreditCard");
        processor.processPayment(250.0, "PayPal");
        processor.processPayment(500.0, "Crypto");
        processor.processPayment(300.0, "ApplePay");
        processor.processPayment(400.0, "GooglePay");

        // Using direct strategy objects
        processor.processPayment(150.0, new CreditCardPayment());
        processor.processPayment(350.0, new ApplePayPayment());
    }
}
```