# SOLID Principles - Code & Scenario-Based Quiz with Answers

This document contains a quiz on SOLID principles with real-world coding scenarios. Each question includes multiple choices, and the correct answer is highlighted.

## Question 1

1. SRP: Which of the following violates the Single Responsibility Principle?

class ReportGenerator {  
 public function generateReport() {  
 // Logic to generate report  
 }  
  
 public function sendEmailReport() {  
 // Logic to send report via email  
 }  
}

A) The class only generates reports

**B) The class handles both report generation and sending emails**

C) The class only sends reports via email

D) The class follows SRP

## Question 2

2. SRP: How would you fix the above violation?

**A) Move the `sendEmailReport()` method to a separate `EmailService` class**

B) Keep the class as is, since it’s convenient to have all report logic in one place

C) Add more functions like `storeReportToDatabase()` to make it a utility class

D) Move everything to a controller

## Question 3

3. OCP: What is the best way to follow the Open/Closed Principle in the given case?

class PaymentProcessor {  
 public function process($type, $amount) {  
 if ($type === 'credit\_card') {  
 return "Processing credit card payment of $$amount";  
 } elseif ($type === 'paypal') {  
 return "Processing PayPal payment of $$amount";  
 }  
 return "Payment method not supported.";  
 }  
}

A) Keep adding more `if` conditions for new payment methods

**B) Create separate classes for each payment method and use polymorphism**

C) Modify the method to accept only `credit\_card` and `paypal` as valid types

D) Use a switch statement instead of `if` conditions

## Question 4

4. OCP: How should the PaymentProcessor be refactored to follow OCP?

**A) Use an interface for payment methods and let each method have its own class**

B) Keep all logic in the same class for easy debugging

C) Make `process()` accept only an `array` of payment types

D) Use multiple `if` conditions with logging

## Question 5

5. LSP: Which of the following breaks the Liskov Substitution Principle?

class Bird {  
 public function fly() {  
 return "Flying...";  
 }  
}  
  
class Penguin extends Bird {  
 public function fly() {  
 throw new Exception("Penguins cannot fly!");  
 }  
}

**A) The Penguin class throws an exception instead of overriding `fly()` correctly**

B) Penguins should extend Bird because they are birds

C) The base class should force all subclasses to implement `fly()`

D) The code correctly follows LSP

## Question 6

6. LSP: How should the above code be refactored to follow LSP?

A) Remove the `Penguin` class because it cannot fly

**B) Move `fly()` to a new `FlyingBird` interface, and only birds that can fly should implement it**

C) Make `Penguin` override `fly()` but return `'Can't fly'` instead of throwing an exception

D) Create an abstract class for birds and force all birds to implement `fly()`

## Question 7

7. ISP: Which of the following violates the Interface Segregation Principle?

interface Worker {  
 public function work();  
 public function eat();  
}  
class Developer implements Worker {  
 public function work() {  
 return "Writing code";  
 }  
 public function eat() {  
 throw new Exception("Developers don’t eat at work!");  
 }  
}

**A) The `Worker` interface has too many responsibilities**

B) The `Developer` class should implement `eat()` even if it's not needed

C) The `Developer` class should be renamed to `RobotDeveloper`

D) The `Worker` interface is fine, and the `Developer` class should not throw an exception

## Question 8

8. ISP: How should the above violation be fixed?

**A) Split `Worker` into `Workable` and `Eatable` interfaces**

B) Remove the `eat()` method from `Worker`, as eating is optional

C) Keep the `eat()` method but make it return `'Not applicable'` instead of throwing an exception

D) Make all developers override `eat()` with `'Eating...'`

## Question 9

9. DIP: Which of the following violates the Dependency Inversion Principle?

class OrderService {  
 private $emailService;  
  
 public function \_\_construct() {  
 $this->emailService = new EmailService();  
 }  
  
 public function sendOrderConfirmation($order) {  
 return $this->emailService->sendEmail("Order confirmed: " . $order);  
 }  
}

**A) `OrderService` directly instantiates `EmailService`, making it tightly coupled**

B) The code is fine because `EmailService` is reusable

C) The class should use a `static` method instead

D) It should pass `$order` as a parameter instead of concatenating

## Question 10

10. DIP: How should the `OrderService` be refactored to follow DIP?

**A) Inject `EmailService` through the constructor using an interface**

B) Make `EmailService` a `static` class so it doesn’t need to be instantiated

C) Create a new `NotificationService` that extends `OrderService`

D) Keep `OrderService` unchanged but rename the `sendEmail()` method