Coupling Cohesion notes

Cohesion

Coupling

Tight Coupling with Hard-Coded Dependencies

Tight Coupling:

```
// Tight Coupling
class EmailService {
    public void sendEmail(String to, String message) {
        // Email sending logic
    }
}

class NotificationService {
    private EmailService emailService;

    public NotificationService() {
        this.emailService = new EmailService(); // Tight coup
ling to the specific implementation
    }

    public void sendNotification(String user, String message)
{
        emailService.sendEmail(user, message);
        // Other notification logic
    }
}
```

Resolution:

```
// Resolution using Dependency Injection
class NotificationService {
    private EmailService emailService;

    // Dependency Injection
    public NotificationService(EmailService emailService) {
        this.emailService = emailService;
    }

    public void sendNotification(String user, String message)
{
        emailService.sendEmail(user, message);
        // Other notification logic
    }
}
```

Tight Coupling with Tight Method Dependency

Tight Coupling:

```
// Tight Coupling
class ShoppingCart {
    public double calculateTotalPrice(List<Item> items) {
        double total = 0;
        for (Item item : items) {
            total += item.getPrice(); // Tight coupling to th
e specific method
        }
        return total;
    }
}

class Item {
    private double price;

    public double getPrice() {
        return price;
    }
}
```

Resolution:

```
// Resolution using Dependency Injection
class ShoppingCart {
    public double calculateTotalPrice(List<Item> items, Price
Calculator priceCalculator) {
        double total = 0;
        for (Item item : items) {
            total += priceCalculator.calculatePrice(item);
        }
        return total;
    }
}

interface PriceCalculator {
    double calculatePrice(Item item);
}

class DefaultPriceCalculator implements PriceCalculator {
    public double calculatePrice(Item item) {
        return item.getPrice();
    }
}

class DiscountedPriceCalculator implements PriceCalculator {
    private double discountPercentage;

    public DiscountedPriceCalculator(double discountPercentag
e) {
        this.discountPercentage = discountPercentage;
    }

    public double calculatePrice(Item item) {
        double discountedPrice = item.getPrice() * (1 - disco
untPercentage / 100);
        return discountedPrice;
    }
}

class TaxIncludedPriceCalculator implements PriceCalculator {
    private double taxRate;

    public TaxIncludedPriceCalculator(double taxRate) {
        this.taxRate = taxRate;
    }

    public double calculatePrice(Item item) {
        double taxAmount = item.getPrice() * (taxRate / 100);
        double totalPrice = item.getPrice() + taxAmount;
        return totalPrice;
    }
}
```

In these resolutions, dependency injection, interfaces, and dependency inversion are used to reduce tight coupling and make the code more modular and flexible. These examples demonstrate how adhering to SOLID principles can lead to more maintainable and scalable software.

Low Cohesion in a Reporting System

```
// Low Cohesion
class ReportingSystem {
    private List<Employee> employees;
    private List<FinancialTransaction> transactions;

    public void loadEmployees() {
        // Load employee data
    }

    public void loadFinancialTransactions() {
        // Load financial transaction data
    }

    public void generateEmployeeReport() {
        // Generate a report based on employee data
    }

    public void generateFinancialReport() {
        // Generate a report based on financial transaction d
ata
    }

    public void sendReportsByEmail() {
        // Send generated reports via email
    }
}
```

In this example, the `ReportingSystem` class has low cohesion because it combines the responsibilities of loading data, generating reports, and sending reports via email. This results in a class that is trying to do too many different things.

Refactoring for High Cohesion:

```
// High Cohesion
class EmployeeDataProvider {
    private List<Employee> employees;

    public void loadEmployees() {
        // Load employee data
    }

    public List<Employee> getEmployees() {
        return employees;
    }
}

class TransactionDataProvider {
    private List<FinancialTransaction> transactions;

    public void loadFinancialTransactions() {
        // Load financial transaction data
    }

    public List<FinancialTransaction> getTransactions() {
        return transactions;
    }
}

class ReportGenerator {
    public void generateEmployeeReport(List<Employee> employe
es) {
        // Generate a report based on employee data
    }

    public void generateFinancialReport(List<FinancialTransac
tion> transactions) {
        // Generate a report based on financial transaction d
ata
    }
}

class EmailSender {
    public void sendReportByEmail(String reportContent, Strin
g recipient) {
        // Send a report via email
    }
}
```

Low Cohesion in a Utility Class

```
// Low Cohesion
class Utility {
    private List<Integer> numbers;
    private String data;

    public void processNumbers() {
        // Process the list of numbers
    }

    public void processData() {
        // Process the string data
    }

    public void displayResults() {
        // Display processed results
    }

    public void saveResultsToFile() {
        // Save processed results to a file
    }
}
```

Refactoring for High Cohesion:

```
// High Cohesion
class NumberProcessor {
    private List<Integer> numbers;

    public void processNumbers() {
        // Process the list of numbers
    }
}
```

```
class DataProcessor {
    private String data;

    public void processData() {
        // Process the string data
    }
}

class ResultsDisplayer {
    public void displayResults() {
        // Display processed results
    }
}

class FileSaver {
    public void saveResultsToFile() {
        // Save processed results to a file
    }
```

LSP

It emphasizes that objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program. In other words, if a class S is a subclass of class T, an object of class T should be replaceable with an object of class S without affecting the desirable properties of the program.

In practical terms, this means that a subclass should extend the behavior of its superclass without changing the expected outcomes of the program. It ensures that the derived class honors the contract established by the base class.

This principle is crucial for maintaining the integrity of the codebase, enabling polymorphism, and facilitating code reuse. Adhering to Liskov's Substitution Principle helps create more flexible, scalable, and maintainable software systems.

Violation of Liskov Substitution Principle:

```
class Rectangle {
    protected int width;
    protected int height;

    public void setWidth(int width) {
        this.width = width;
    }

    public void setHeight(int height) {
        this.height = height;
    }

    public int calculateArea() {
        return width * height;
    }
}

class Square extends Rectangle {
    @Override
    public void setWidth(int width) {
        super.setWidth(width);
        super.setHeight(width);
    }

    @Override
    public void setHeight(int height) {
        super.setWidth(height);
        super.setHeight(height);
    }
}
```

In this example, the `Square` class inherits from `Rectangle`, and it overrides the `setWidth` and `setHeight` methods to ensure that the width and height are always equal. However, this violates the Liskov Substitution Principle.

Problems with the Violation:

1. **Inconsistent Behavior:** Modifying the width or height of a `Square` through the overridden methods leads to inconsistent behavior.
2. **Unexpected Results:** Setting the width and height separately for a `Square` can result in a shape that is no longer a square.

Fix with Liskov Substitution Principle:

To adhere to Liskov's Substitution Principle, you might reconsider the class hierarchy. One way to fix this is to separate the concerns and avoid the use of inheritance for this relationship:

```
interface Shape {
    int calculateArea();
}

class Rectangle implements Shape {
    protected int width;
    protected int height;

    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }

    @Override
    public int calculateArea() {
        return width * height;
    }
}

class Square implements Shape {
    private int side;

    public Square(int side) {
        this.side = side;
    }

    @Override
    public int calculateArea() {
        return side * side;
    }
}
```

In this fixed version, both `Rectangle` and `Square` implement a common `Shape` interface without using inheritance in a way that violates LSP.

The Interface Segregation Principle (ISP) is one of the SOLID principles of object-oriented design, introduced by Robert C. Martin. The SOLID principles are a set of guidelines that aim to make software design more maintainable, flexible, and scalable. The Interface Segregation Principle specifically addresses the design of interfaces.

The Interface Segregation Principle states that a class should not be forced to implement interfaces it does not use. In other words, a class should only be required to implement the methods that are relevant to its behavior, and it should not be burdened with unnecessary methods that it won't use.

Here's a more detailed explanation:

1. **Definition of Interfaces:**
    - Interfaces are used to define a contract or a set of methods that a class must implement.
    - Classes that implement an interface are expected to provide concrete implementations for all the methods declared in that interface.
2. **Problem Statement:**
    - In some cases, interfaces may become too large and include methods that are not relevant to all implementing classes.
    - When a class is forced to implement methods it doesn't need, it violates the principle of keeping classes focused and cohesive.

3. **Example:**
    - Consider an interface `Worker` that has methods like `work()` and `eat()`.
    - If you have a class representing a robot, it might not need the `eat()` method. However, because it must implement the `Worker` interface, it is forced to provide an unnecessary implementation for `eat()`.
4. **Solution - Segregate Interfaces:**
    - Instead of having a single large interface, break it down into smaller, more specific interfaces.
    - In the example above, you could have separate interfaces for `Workable` and `Eatable`, and classes can choose to implement only the interfaces that are relevant to them.
5. **Benefits:**
    - Reduces the likelihood of having classes with unnecessary or irrelevant methods.
    - Increases the flexibility of the codebase, as classes can implement only the interfaces they need.
6. **Example Implementation:**

```
// Instead of a single large interface
interface Worker {
    void work();
    void eat();
}

// Use segregated interfaces
interface Workable {
    void work();
}

interface Eatable {
    void eat();
}
```

```
class Robot implements Workable {
    @Override
    public void work() {
        // Implementation for work
    }
}
```

By following the Interface Segregation Principle, you create more modular and maintainable code, allowing classes to be more focused on their specific responsibilities.

Certainly! Let's explore two more complex examples to illustrate the issues that may arise violating the Interface Segregation Principle (ISP) and how to address them.

Document Processing System

Issue:
Consider a document processing system that defines an interface for different types of documents, such as `EditableDocument` and `PrintableDocument`. The interface may look like this:

```
interface Document {
    void open();
    void edit();
    void save();
    void print();
}
```

Now, if you have a class representing a read-only document (e.g., a PDF document), it is forced to implement the `edit()` and `save()` methods, which are irrelevant to its nature.

```
class PdfDocument implements Document {
    @Override
    public void open() {
        // Implementation for opening a PDF document
    }

    @Override
    public void edit() {
        // This method is not applicable to read-only documen
ts
        throw new UnsupportedOperationException("Cannot edit
a PDF document");
    }

    @Override
    public void save() {
        // This method is not applicable to read-only documen
ts
        throw new UnsupportedOperationException("Cannot save
changes to a PDF document");
    }

    @Override
    public void print() {
        // Implementation for printing a PDF document
    }
}
```

Fix:

To adhere to the Interface Segregation Principle, you can create more specific interfaces for different aspects of a document:

```
interface Openable {
    void open();
}

interface Editable {
    void edit();
    void save();
}

interface Printable {
    void print();
}

class PdfDocument implements Openable, Printable {
    @Override
    public void open() {
        // Implementation for opening a PDF document
    }

    @Override
    public void print() {
        // Implementation for printing a PDF document
    }
}
```

Now, the `PdfDocument` class only implements the interfaces that are relevant to its functionality, avoiding the unnecessary burden of implementing irrelevant methods.

Messaging System

Issue:
Imagine a messaging system with a generic
`Message` interface:

```
interface Message {
    void send();
    void receive();
}
```

Now, if you have a class representing a broadcast message that is only sent and not received, it is forced to implement the `receive()` method.

```
class BroadcastMessage implements Message {
    @Override
    public void send() {
        // Implementation for sending a broadcast message
    }

    @Override
    public void receive() {
        // This method is not applicable to broadcast message
s
        throw new UnsupportedOperationException("Broadcast me
ssages cannot be received");
    }
}
```

Fix:
To follow the Interface Segregation Principle, you can create separate interfaces for sending and receiving:

```
interface Sendable {
    void send();
}

interface Receivable {
    void receive();
}

class BroadcastMessage implements Sendable {
    @Override
    public void send() {
        // Implementation for sending a broadcast message
    }
}
```

Now, the `BroadcastMessage` class only implements the `Sendable` interface, reflecting the specific nature of broadcast messages.

In both examples, segregating interfaces results in more focused and cohesive designs, where classes implement only the methods relevant to their specific responsibilities.

The Dependency Inversion Principle (DIP) is one of the SOLID principles of object-oriented design, introduced by Robert C. Martin. The Dependency Inversion Principle consists of two key concepts: high-level modules should not depend on low-level modules, but both should depend on abstractions; and abstractions should not depend on details, but details should depend on abstractions.

In simpler terms, the principle suggests that the direction of dependency should be inverted. Instead of high-level modules (which contain the main business logic) depending on low-level modules (which contain the implementation details), both should depend on abstractions (interfaces or abstract classes). This helps to decouple the high-level modules from the specific details of the low-level modules, making the system more flexible and easier to maintain.

Here's an explanation of the Dependency Inversion Principle and how Dependency Injection (DI) is used to adhere to this principle:

1. **High-Level Modules and Low-Level Modules:**
    - High-level modules contain the business logic and policies of an application.
    - Low-level modules contain the implementation details and are responsible for specific tasks.
2. **Traditional Dependency:**
    - In traditional dependency structures, high-level modules depend directly on low-level modules.
    - This can make the system rigid and less adaptable to changes because any modification in the low-level modules might affect the high-level modules.
3. **Dependency Inversion Principle (DIP):**

- DIP suggests that both high-level and low-level modules should depend on abstractions (interfaces or abstract classes) rather than on concrete implementations.

4. **Abstractions:**
    - Abstractions define the contract or interface that high-level and low-level modules should adhere to.
    - Abstractions should be designed based on the needs of high-level modules, providing a level of indirection.
5. **Dependency Injection (DI):**
    - Dependency Injection is a technique to implement the Dependency Inversion Principle.
    - Instead of high-level modules creating instances of low-level modules directly, dependencies are "injected" into the high-level module from outside.
    - This injection is often achieved through constructor injection, method injection, or property injection.
6. **Example Implementation:**
    - Consider a high-level module `UserService` that requires a low-level module `UserRepository` to perform data storage operations.

```
// Before applying DIP and DI
class UserService {
    private UserRepository userRepository;

    public UserService() {
        // Direct instantiation of a low-level module
        this.userRepository = new UserRepository();
    }

    // Business logic using userRepository
}
```

```
// Abstraction (interface) representing UserRepository
interface UserRepository {
    void save(User user);
    User getById(String userId);
}

// High-level module using dependency injection
class UserService {
    private final UserRepository userRepository;

    // Constructor injection
    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    // Business logic using userRepository
}
```

- After applying DIP and DI:

In this example, the `UserService` no longer directly depends on a concrete implementation of `UserRepository`. Instead, it depends on the abstraction `UserRepository`, and the actual implementation is injected at runtime through the constructor.

Applying the Dependency Inversion Principle through Dependency Injection makes the system more modular, testable, and adaptable to changes, as it promotes loose coupling between high-level and low-level modules.

```
class OrderProcessor {
    private PaymentProcessor payProcessor;

    public OrderProcessor() {
                payProcessor = new CreditCardPaymentProcessor()
        }
```

```
}


// how to fix this

interface PaymentProcessor {

}

class CreditCardPaymentProcessor implements PaymentProcessor{}
class CashPaymentProcessor implements PaymentProcessor{}
class UpiPaymentProcessor implements PaymentProcessor{}

class OrderProcessor {
    private PaymentProcessor payProcessor;

    public OrderProcessor(PaymentProcessor payProcessor;) {
                this.payProcessor = payProcessor;
        }
}



// Example 2

class EmailService {
     void sendEmail(String to, String message) {

     }
}

class NotificationService {
        private EmailService emailService;
     public NotificationService() {
       emailService = new EmailService();// tight coupling
     }
     public sendNotification(String user, String message) {
        emailService.sendEmail(user, message);
     }
}



class Item {
     private double price;
     //....
     public getPrice(){return price;};
}

class ShoppingCart {
     public double calcTotalPrice(List<Item> items) {
         double total = 0;
         for(Item item: items) {
            total += item.getPrice(); // tight coupling
         }
         return total;
     }
}

class ShoppingCart {
     public double calcTotalPrice(List<Item> items, PriceCalcula
         double total = 0;
         for(Item item: items) {
            total += priceClac.calculatePrice(item);
         }
         return total;
     }
}

interface PriceCalculator{
   double calculatePrice(Item item);
```

```
}

class DefaltPriceCalculator implements PriceCalculator{

}

class TaxIncludePriceCalculator implements PriceCalculator{

}

class SummerSalePriceCalculator implements PriceCalculator{

}
```

Cohesion

```
class Reporting {
    private List<Employee> empList;
    private List<PaymentTransaction> tansactions;

    loadEmployee(){}
    generateEmployeeReport(){}
    generateFinalcialReport(){}
    sendReportByEmail(){}
}
```

LSP

Liskov's substitution principle

```
class Rectangle {
    int width;
    int height;
    setWidth(int w) { this.width = w;}
    setHeight(int h) { this.height = h;}

   int area() { return width*height;}
}

class Square extends Rectangle {
    @Override
    setWidth(int w) { this.height=this.width=w;};

    @Override
    setHeight(int h) { this.height=this.width=h;};
}

int main() {
    Square sq = new Rectangle();
    sq.setWidth(10);
    sq.setHeight(20);
}



// fix
interface Shape {
  int area();
}

Rectangle implements Shape {
      int width;
    int height;
    Rectangle(int h, int w) {};

   @Override
   int area() { return width*height;}
}

Square implements Shape {
    int side;
    Square(int s) {this.side = s;}

    @Override
   int area() { return side*side;}
}
```

ISP

interface segregation principle

```
interface Worker {. // violates ISP
    void work();
    void eat();
}

// fix
interface Workable {
   void work();
}
interface Eatable{
   void eat();
}

class Human implements Workable, Eatable {
   void work(){};
   void eat(){};
}

class Robot implements Workable {
   void work(){};
}
```

DIP dependency inversion principle

```
class UserService {
    private UserRepo userRepo;

    public UserService() { this.userRepo = new UserRepo();}
  // ...
}


fix
interface UserRepo {
    save();
    getById();
}

class PremiumUserRepo implements UserRepo {}
class DefaultUserRepo implements UserRepo {}

class UserService {
    private UserRepo userRepo;

    // constructor injection
    public UserService(UserRepo userRepo) { this.userRepo = user
}
```