**Structural Design Pattern**

**Adapter Design Pattern**

The Adapter design pattern is a structural design pattern that allows two incompatible interfaces to work together. It acts as a bridge between two interfaces, making them compatible and enabling them to work seamlessly. This pattern is useful when you need to integrate existing code or libraries that have different interfaces with your codebase without making significant modifications.

**Understanding the Adapter Pattern**

The Adapter Pattern involves three key components:

1. **Target Interface:** This is the interface that the client code expects and understands.
2. **Adaptee:** This is the class or interface that you want to adapt to the Target Interface. It is the existing code or component with an incompatible interface.
3. **Adapter:**The Adapter is a class that implements the Target Interface and wraps an instance of the Adaptee, translating calls between them.

**Example Scenario: Payment Gateway Integration**

Imagine you’re building an e-commerce application that supports payments from different payment gateway providers, such as PayPal and Stripe. Each payment gateway has its own unique interface and methods for processing payments. However, you want to create a unified payment processing system in your application that can work with various payment gateway providers without changing your existing code.

**Implementing the Adapter Pattern**

**Step 1: Define a Common Interface**

The first step in implementing the Adapter Pattern is to define a common interface that your application understands. In our case, we’ll create a PaymentGateway interface:

public interface PaymentGateway {  
 void processPayment(double amount);  
}

This interface defines a single method, processPayment, which is responsible for processing payments.

**Step 2: Create Payment Gateway Adapters**

Next, we’ll create adapter classes for each specific payment gateway provider (PayPal and Stripe) to make them compatible with our PaymentGateway interface. Let's start with the PayPal adapter:

**PayPal Adapter**

public class PayPalAdapter implements PaymentGateway {  
 private PayPal paymentGateway;  
  
 public PayPalAdapter(PayPal paymentGateway) {  
 this.paymentGateway = paymentGateway;  
 }  
  
 @Override  
 public void processPayment(double amount) {  
 // Convert our application's method to PayPal's method  
 paymentGateway.makePayment(amount);  
 }  
}

**Stripe Adapter**

Similarly, we create an adapter for the Stripe payment gateway:

public class StripeAdapter implements PaymentGateway {  
 private StripePaymentGateway paymentGateway;  
  
 public StripeAdapter(StripePaymentGateway paymentGateway) {  
 this.paymentGateway = paymentGateway;  
 }  
  
 @Override  
 public void processPayment(double amount) {  
 // Convert our application's method to Stripe's method  
 paymentGateway.charge(amount);  
 }  
}

These adapter classes take the specific payment gateway instances as constructor parameters and implement the processPayment method by converting the method calls to the corresponding methods of the payment gateway providers.

**Step 3: Implement Concrete Payment Gateway Providers**

In our example, we’ll create hypothetical implementations of the PayPal and Stripe payment gateways.

**PayPal Implementation**

public class PayPal {  
 public void makePayment(double amount) {  
 // PayPal-specific payment processing logic  
 System.out.println("Paid $" + amount + " via PayPal.");  
 }  
}

**Stripe Implementation**

public class StripePaymentGateway {  
 public void charge(double amount) {  
 // Stripe-specific payment processing logic  
 System.out.println("Charged $" + amount + " using Stripe.");  
 }  
}

These classes represent the specific payment gateway providers and define their unique payment processing logic.

**Step 4: Client Code**

Now that we have defined the common interface (PaymentGateway), created adapter classes for payment gateways, and implemented concrete payment gateway providers, we can use these components in our client code:

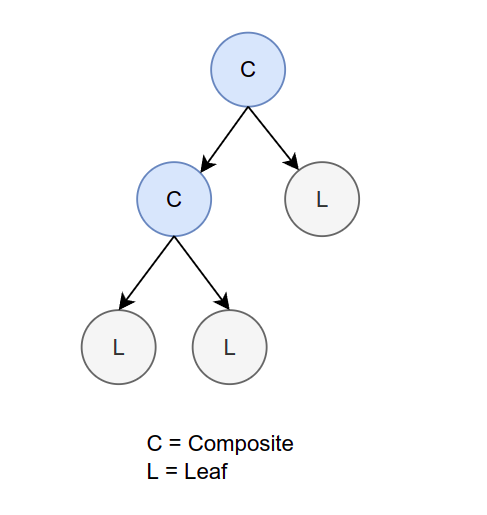
public class PaymentApp {  
 public static void main(String[] args) {  
 PaymentGateway paypalGateway = new PayPalAdapter(new PayPal());  
 PaymentGateway stripeGateway = new StripeAdapter(new StripePaymentGateway());  
  
 double amount = 100.0;  
  
 // Process payments using different payment gateways  
 paypalGateway.processPayment(amount);  
 stripeGateway.processPayment(amount);  
 }  
}

In the client code, we create instances of the PayPal and Stripe adapters, passing the corresponding payment gateway instances as parameters. Then, we can use the processPayment method to initiate payments through the unified PaymentGateway interface.

**Composite Design Pattern**

*It represents a group of objects through a defined structure/hierarchy.*

Actually it’s a tree structure. A composite can have a composite or a leaf.

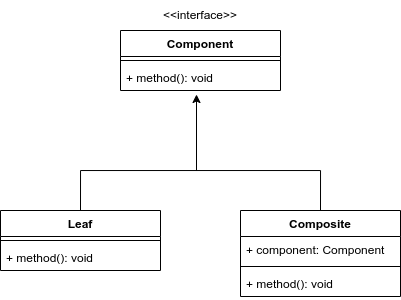


As an example, we can think of a company. We have several departments. Under each department we have set of employees. It’s kind of a hierarchy.

Here we have some key things to remember from the design pattern aspect.

**3 Key Concepts**

1. **Component**— defines the interface for objects in the composition and for accessing and managing its child components.
2. **Leaf**— defines behavior for primitive objects in the composition. It represents leaf objects in the composition.
3. **Composite**— stores child components and implements children derived from the component interface and manages them.



As you see, **Composite** class it holding **Component** objects. So, that is the place we see the composition!

Let’s take the same company example to demonstrate this.

Our component class will be *Employee* interface.

**Employee**

public interface Employee {  
 void showDetails();  
}

Leaves will be the below classes..Developer/QA/Manager/CEO

**CEO**

public class CEO implements Employee {  
 private final String name;  
 private final String dept;  
 private final String job;  
  
 public CEO(String name, String dept, String job) {  
 this.name = name;  
 this.dept = dept;  
 this.job = job;  
 }  
  
 @Override  
 public void showDetails() {  
 System.out.println(this);  
 }  
  
 @Override  
 public String toString() {  
 return "CEO{" +  
 "name='" + name + '\'' +  
 ", dept='" + dept + '\'' +  
 ", job='" + job + '\'' +  
 '}';  
 }  
}

**Developer**

import java.util.List;  
  
public class Developer implements Employee {  
 private final String name;  
 private final String dept;  
 private final String job;  
  
 public Developer(String name, String dept, String job) {  
 this.name = name;  
 this.dept = dept;  
 this.job = job;  
 }  
  
 @Override  
 public void showDetails() {  
 System.out.println(this);  
 }  
  
 @Override  
 public String toString() {  
 return "Developer{" +  
 "name='" + name + '\'' +  
 ", dept='" + dept + '\'' +  
 ", job='" + job + '\'' +  
 '}';  
 }  
}

**Manager**

public class Manager implements Employee {  
 private final String name;  
 private final String dept;  
 private final String job;  
  
 public Manager(String name, String dept, String job) {  
 this.name = name;  
 this.dept = dept;  
 this.job = job;  
 }  
  
 @Override  
 public void showDetails() {  
 System.out.println(this);  
 }  
  
 @Override  
 public String toString() {  
 return "Manager{" +  
 "name='" + name + '\'' +  
 ", dept='" + dept + '\'' +  
 ", job='" + job + '\'' +  
 '}';  
 }  
}

**QA**

public class QA implements Employee {  
  
 private final String name;  
 private final String dept;  
 private final String job;  
  
 public QA(String name, String dept, String job) {  
 this.name = name;  
 this.dept = dept;  
 this.job = job;  
 }  
  
 @Override  
 public void showDetails() {  
 System.out.println(this);  
 }  
  
 @Override  
 public String toString() {  
 return "QA{" +  
 "name='" + name + '\'' +  
 ", dept='" + dept + '\'' +  
 ", job='" + job + '\'' +  
 '}';  
 }  
}

Now we need a **Composite** **class** which holds our employees.

**CompanyDirectory**

public class CompanyDirectory implements Employee {  
  
 private final List<Employee> employees;  
  
 public CompanyDirectory() {  
 this.employees = new ArrayList<>();  
 }  
  
 @Override  
 public void showDetails() {  
 for (Employee employee : employees) {  
 employee.showDetails();  
 }  
 }  
  
 public void addEmployee(Employee employee) {  
 employees.add(employee);  
 }  
  
 public void removeEmployee(Employee employee) {  
 employees.remove(employee);  
 }  
  
}

Then I’m going to create a **Company** class which is again a **Composite** type and having a **name** property additionally.

public class Company extends CompanyDirectory {  
 private final String name;  
  
 public Company(String name) {  
 this.name = name;  
 }  
  
 public void showCompanyName() {  
 System.out.println("Company: " + name);  
 }  
}

Classes setup is done now! It’s time to test!!! Let’s create test class…

**CompositeDemo**

public class CompositeDemo {  
  
 public static void main(String[] args) {  
  
 CEO ceo = new CEO("Andrew", "COM", "Chief Executive Officer");  
  
 Developer developer1 = new Developer("salitha", "ENG", "Software Engineer");  
 Developer developer2 = new Developer("Jhon", "ENG", "Senior Software Engineer");  
 QA qe1 = new QA("Tom", "ENG", "Quality Assurance Engineer");  
 QA qe2 = new QA("Jimmy", "ENG", "Quality Assurance Lead");  
  
 Manager manager = new Manager("Peter", "HR", "HR Manager");  
 Manager executive = new Manager("Derik", "HR", "Senior HR Executive");  
  
 CompanyDirectory engDepartment = new CompanyDirectory();  
 engDepartment.addEmployee(developer1);  
 engDepartment.addEmployee(developer2);  
 engDepartment.addEmployee(qe1);  
 engDepartment.addEmployee(qe2);  
  
 CompanyDirectory hrDepartment = new CompanyDirectory();  
 hrDepartment.addEmployee(manager);  
 hrDepartment.addEmployee(executive);  
  
 Company company = new Company("XYZ");  
 company.addEmployee(ceo);  
 company.addEmployee(engDepartment);  
 company.addEmployee(hrDepartment);  
 company.showCompanyName();  
 company.showDetails();  
  
 }  
  
}

**Facade Design Pattern**

**Understanding the Facade Pattern**

The Facade pattern is classified under the structural design patterns category because it simplifies the way we interact with complex systems. It doesn’t change the system itself but encapsulates it behind a Facade, thus making it easier to use. This is achieved by designing a higher-level interface that encapsulates and hides the system’s complexities.

Imagine a scenario where you have a complex subsystem with numerous interdependent classes. Rather than interacting with each class directly, you can introduce a Facade class to encapsulate the whole subsystem. This Facade becomes the entry point to the subsystem and provides a simple interface, thus abstracting away the complexities.

**Implementing the Facade Pattern in Java**

Let's delve into how we can implement the Facade pattern in Java using a real-world example.

Suppose we are developing a home automation system that controls different aspects like lighting, music, and climate. Each of these subsystems is complex and consists of various classes.

public class Lighting {  
 public void on() {  
 System.out.println("Lights are on");  
 }  
 public void off() {  
 System.out.println("Lights are off");  
 }  
}  
  
public class MusicSystem {  
 public void playMusic() {  
 System.out.println("Music is playing");  
 }  
 public void stopMusic() {  
 System.out.println("Music is stopped");  
 }  
}  
  
public class ClimateControl {  
 public void setTemperature(int temp) {  
 System.out.println("Temperature set to " + temp + " degrees");  
 }  
}

Each subsystem is complex in its own right. However, we can introduce a Facade class, "SmartHomeFacade," to simplify interaction with these subsystems.

public class SmartHomeFacade {  
 private Lighting lighting;  
 private MusicSystem musicSystem;  
 private ClimateControl climateControl;  
  
 public SmartHomeFacade(Lighting lighting, MusicSystem musicSystem, ClimateControl climateControl) {  
 this.lighting = lighting;  
 this.musicSystem = musicSystem;  
 this.climateControl = climateControl;  
 }  
  
 public void startEveningRoutine() {  
 lighting.on();  
 musicSystem.playMusic();  
 climateControl.setTemperature(22);  
 }  
  
 public void endEveningRoutine() {  
 lighting.off();  
 musicSystem.stopMusic();  
 }  
}

This facade class wraps the complexity of the subsystem and exposes a simple-to-use interface to the client.

**Benefits of the Facade Pattern**

Using the Facade design pattern offers several advantages:

* **Simplification:** Facade simplifies the interaction with complex systems by providing a single simplified interface.
* **Decoupling:** It decouples the subsystems from the clients and other subsystems, promoting subsystem independence and portability.
* **Manageability:** Facade improves the readability and manageability of the code, enhancing the overall software maintainability.

**When to Use the Facade Pattern**

The Facade pattern is beneficial in scenarios where:

* A system is very complex or difficult to understand.
* An entry point is needed for each subsystem.
* There is a need to layer your subsystems.

Remember, it’s not necessary to encapsulate every class with a Facade. Overuse can lead to an overly complicated design, which defeats the pattern’s purpose.

**The Nitty-Gritty of Facade Pattern**

While we have established a solid understanding of the Facade design pattern and its usage in Java, let's further delve into some intricacies.

**Implementing Facade in Larger Code Bases**

Working with larger codebases can get confusing and intimidating, especially with numerous subsystems. However, with the use of the Facade design pattern, the interaction becomes more manageable.

Let's consider a larger system – an eCommerce platform. This platform includes numerous subsystems such as user authentication, payment processing, inventory management, and order fulfillment, among others. Let's look at how a Facade pattern can simplify the interactions.

public class UserAuthentication {  
 public void login(String username, String password) {  
 System.out.println("User logged in");  
 }  
 public void logout() {  
 System.out.println("User logged out");  
 }  
}  
  
public class PaymentProcessing {  
 public void processPayment(String paymentMethod) {  
 System.out.println("Payment processed");  
 }  
}  
  
public class InventoryManagement {  
 public void updateInventory(String productId, int quantity) {  
 System.out.println("Inventory updated");  
 }  
}  
  
public class OrderFulfillment {  
 public void fulfillOrder(String orderId) {  
 System.out.println("Order fulfilled");  
 }  
}

We could interact directly with these subsystems. However, using a Facade class makes it more manageable. We could create an ECommerceFacade class that wraps around these subsystems and provides simpler methods to interact with them.

public class ECommerceFacade {  
 private UserAuthentication userAuthentication;  
 private PaymentProcessing paymentProcessing;  
 private InventoryManagement inventoryManagement;  
 private OrderFulfillment orderFulfillment;  
  
 public ECommerceFacade(UserAuthentication userAuthentication, PaymentProcessing paymentProcessing,  
 InventoryManagement inventoryManagement, OrderFulfillment orderFulfillment) {  
 this.userAuthentication = userAuthentication;  
 this.paymentProcessing = paymentProcessing;  
 this.inventoryManagement = inventoryManagement;  
 this.orderFulfillment = orderFulfillment;  
 }  
  
 public void purchaseProduct(String username, String password, String paymentMethod, String productId, int quantity) {  
 userAuthentication.login(username, password);  
 paymentProcessing.processPayment(paymentMethod);  
 inventoryManagement.updateInventory(productId, quantity);  
 orderFulfillment.fulfillOrder(productId);  
 userAuthentication.logout();  
 }  
}

In the example above, the Facade simplifies the interaction by abstracting the underlying complexities, making the code more readable and maintainable.

**Delving Deeper: Advanced Scenarios with the Facade Pattern**

While we have covered the Facade design pattern’s basics and applied it to some common scenarios, let’s explore its usage in more complex, real-world situations.

**Facade and Legacy Systems**

The Facade pattern shines in dealing with legacy systems — older systems that may be difficult to work with due to outdated technologies, tangled codebases, or lack of documentation. By introducing a Facade, we can encapsulate the complexities of the legacy system and make it easier to interact with.

Consider a banking system built using older technologies. This system might have multiple classes to handle customer accounts, transactions, audits, etc. Directly interacting with these classes could be challenging, but introducing a BankingFacade class simplifies the process.

public class BankingFacade {  
 private CustomerAccount customerAccount;  
 private Transactions transactions;  
 private Audit audit;  
  
 public BankingFacade(CustomerAccount customerAccount, Transactions transactions, Audit audit) {  
 this.customerAccount = customerAccount;  
 this.transactions = transactions;  
 this.audit = audit;  
 }  
  
 public void makeTransaction(String fromAccount, String toAccount, double amount) {  
 customerAccount.debitAccount(fromAccount, amount);  
 customerAccount.creditAccount(toAccount, amount);  
 transactions.recordTransaction(fromAccount, toAccount, amount);  
 audit.logTransaction(fromAccount, toAccount, amount);  
 }  
}

The BankingFacade class in the example above simplifies the process of making a transaction, wrapping the interaction with multiple classes into a single method.

**Facade Pattern and Microservices**

The Facade pattern is also useful in a microservices architecture. Each microservice is a separate, self-contained module, but they often need to communicate with each other. A Facade can serve as a gateway, managing and simplifying these inter-microservice communications.

For instance, an eCommerce platform might have separate microservices for user management, product management, and order management. A ECommerceFacade can simplify the interaction between these microservices.

public class ECommerceFacade {  
 private UserManagementMicroservice userManagement;  
 private ProductManagementMicroservice productManagement;  
 private OrderManagementMicroservice orderManagement;  
  
 public ECommerceFacade(UserManagementMicroservice userManagement, ProductManagementMicroservice productManagement,   
 OrderManagementMicroservice orderManagement) {  
 this.userManagement = userManagement;  
 this.productManagement = productManagement;  
 this.orderManagement = orderManagement;  
 }  
  
 public void placeOrder(String username, String password, String productId, int quantity) {  
 userManagement.authenticateUser(username, password);  
 productManagement.checkProductAvailability(productId, quantity);  
 orderManagement.createOrder(username, productId, quantity);  
 }  
}

**Pitfalls and Best Practices**

While the Facade pattern offers several benefits, it’s crucial to be aware of potential pitfalls and follow best practices for the most effective use.

**Pitfall: Overusing Facades**

Overusing facades can lead to a tangled mess of facade classes that obfuscate the system’s structure rather than simplifying it. Facades should only be used to simplify complex systems or provide a specific, simplified view of a subsystem.

**Pitfall: Facade Pattern is not a Silver Bullet**

The Facade pattern is not a one-size-fits-all solution. It’s an excellent tool for certain situations, but it might not be the best choice for all scenarios. Using the Facade pattern where it’s not needed can lead to unnecessary abstraction and complexity.

**Proxy Pattern**

**Lazy Loading**

**The Problem**

In certain operations, such as fetching customer account details, may involve expensive database queries. What if you don’t need the full details immediately but still want to maintain flexibility to load them later?

**The Solution: Lazy Loading with a Proxy Class**

Lazy loading defers the expensive operation until it’s actually needed. Here’s how it works using a proxy:

**Step 1: Define an Interface**

public interface Account {  
 void getAccountDetails();  
}

**Why an interface?**

* It defines a common contract, allowing the client to interact with both the proxy and the real object seamlessly.
* Ensures loose coupling, making it easier to switch implementations.

**Step 2: Implement the Real Object**

public class RealAccount implements Account {  
 private final String accountId;  
  
 public RealAccount(String accountId) {  
 this.accountId = accountId;  
 loadAccountDetails(); // Expensive operation  
 }  
  
 private void loadAccountDetails() {  
 System.out.println("Loading account details for ID: " + accountId);  
 }  
  
 @Override  
 public void getAccountDetails() {  
 System.out.println("Displaying account details for ID: " + accountId);  
 }  
}

**What does the real object do?**

* It performs the actual operation, such as querying the database.
* However, this operation is resource-intensive and may not be needed immediately.

**Step 3: Implement the Proxy**

public class ProxyAccount implements Account {  
 private RealAccount realAccount;  
 private final String accountId;  
  
 public ProxyAccount(String accountId) {  
 this.accountId = accountId;  
 }  
  
 @Override  
 public void getAccountDetails() {  
 if (realAccount == null) {  
 realAccount = new RealAccount(accountId); // Object created only when needed  
 }  
 realAccount.getAccountDetails();  
 }  
}

**What problem does the proxy solve?**

* The proxy ensures that the real account is loaded only when the getAccountDetails method is called.
* This optimizes resource usage by avoiding unnecessary operations.

**Step 4: Use the Proxy**

public class Main {  
 public static void main(String[] args) {  
 Account account = new ProxyAccount("123456");  
 System.out.println("Proxy object created, but account details not loaded yet.");  
  
 // Real account is loaded only now  
 account.getAccountDetails();  
 }  
}

**Logging with Proxies**

**The Problem**

What if we want to log every method call in our application? A naive approach would be to write logging code inside every method, which would:

* Violate separation of concerns.
* Add repetitive and hard-to-maintain code.

**The Solution: Logging with AOP and Proxies**

Spring Boot uses proxies to implement logging dynamically using **Aspect-Oriented Programming (AOP)**. This eliminates the need to modify method bodies.

Example: Logging with AOP

@Aspect  
@Component  
public class LoggingAspect {  
  
 @Before("execution(\* com.example.banking.service.\*.\*(..))")  
 public void logBefore(JoinPoint joinPoint) {  
 System.out.println("Method invoked: " + joinPoint.getSignature().getName());  
 }  
}

**Key Points:**

1. The @Aspect annotation marks this as an aspect class.
2. The @Before annotation defines advice to run before methods in the com.example.banking.service package.
3. The JoinPoint object provides access to:

* Method name (joinPoint.getSignature().getName())
* Arguments passed to the method (joinPoint.getArgs()).

**Why use a proxy here?**

* The proxy intercepts method calls and applies the logging behavior before executing the real method.

**Transaction Management with**@Transactional

**The Problem**

In a banking system, transferring money between accounts must adhere to the **ACID** principles:

* **Atomicity:** Ensure the entire transaction succeeds or fails.
* **Consistency:** Maintain the database in a consistent state.
* **Isolation:** Ensure transactions don’t interfere with each other.
* **Durability:** Persist changes even in case of failures.

Manually managing transactions can be error-prone and cluttered.

**The Solution:**@Transactional

Spring Boot simplifies transaction management by using proxies with the @Transactional annotation. For example:

@Service  
public class TransferService {  
  
 @Transactional  
 public void transferMoney(String fromAccount, String toAccount, double amount) {  
 System.out.println("Transferring " + amount + " from " + fromAccount + " to " + toAccount);  
 // Logic for debit and credit operations  
 }  
}

**What happens under the hood?**

* Spring creates a proxy for the TransferService.
* The proxy starts a transaction before transferMoney is executed.
* If any exception occurs, the proxy rolls back the transaction; otherwise, it commits the transaction.

**The Issue: Where is the Interface?**

So far, we’ve defined a **common interface** (Account) that both the **proxy and the real object** implement. However, in the @Transactional example, we haven’t explicitly defined an interface for TransferService. Why?

**Spring’s Approach to Proxying Without an Interface**

In Spring Boot, **there are two types of proxy mechanisms**:

1. **JDK Dynamic Proxies** (Requires an Interface)

* If the target class implements an interface, Spring will create a **dynamic proxy** that implements the same interface.

**2. CGLIB Proxies** (No Interface Required)

* If there is no interface, Spring generates a **subclass proxy** using **CGLIB (Code Generation Library)**.
* This allows Spring to wrap the class and modify its behavior at runtime.

Since TransferService **does not implement an interface**, Spring automatically chooses **CGLIB** to create a subclass proxy for transaction management.

Example: Proxy for Transaction Management

public class TransactionalProxy implements TransferService {  
 private final TransferService realService;  
  
 public TransactionalProxy(TransferService realService) {  
 this.realService = realService;  
 }  
  
 @Override  
 public void transferMoney(String fromAccount, String toAccount, double amount) {  
 System.out.println("[Proxy] Starting transaction...");  
 try {  
 realService.transferMoney(fromAccount, toAccount, amount);  
 System.out.println("[Proxy] Committing transaction...");  
 } catch (Exception e) {  
 System.out.println("[Proxy] Rolling back transaction...");  
 throw e;  
 }  
 }  
}

**Why is this important?**

* It ensures ACID compliance without manual transaction management.
* Simplifies code by abstracting transaction logic.

**How Proxying Works Here**

* When Spring detects the @Transactional annotation, it **does not modify the original class**.
* Instead, it **creates a proxy** that **intercepts** method calls and applies transaction management logic.
* This proxy ensures that:
* A **transaction is started** before calling transferMoney.
* If the method executes successfully, the **transaction is committed**.
* If an exception occurs, the **transaction is rolled back**.

**Example: Manually Creating a Proxy for Transactions**

If we were to manually implement transaction control using the **Proxy Pattern**, it would look like this:

public interface TransferService {  
 void transferMoney(String fromAccount, String toAccount, double amount);  
}  
  
public class RealTransferService implements TransferService {  
 @Override  
 public void transferMoney(String fromAccount, String toAccount, double amount) {  
 System.out.println("Transferring " + amount + " from " + fromAccount + " to " + toAccount);  
 }  
}  
  
public class TransactionalProxy implements TransferService {  
 private final TransferService realService;  
  
 public TransactionalProxy(TransferService realService) {  
 this.realService = realService;  
 }  
  
 @Override  
 public void transferMoney(String fromAccount, String toAccount, double amount) {  
 System.out.println("[Proxy] Starting transaction...");  
 try {  
 realService.transferMoney(fromAccount, toAccount, amount);  
 System.out.println("[Proxy] Committing transaction...");  
 } catch (Exception e) {  
 System.out.println("[Proxy] Rolling back transaction...");  
 throw e;  
 }  
 }  
}

**Why Spring Uses CGLIB Instead of Requiring an Interface?**

* If there’s an interface, Spring will **prefer JDK proxies**.
* If there’s **no interface**, Spring automatically **uses CGLIB to create a proxy subclass**.
* This flexibility allows Spring to **proxy both interfaces and concrete classes**.

**Should We Inject the Real Object into the Proxy or Use**new**?**

**The Issue: Instantiating the Real Object in the Proxy**

In our previous **manual proxy implementation**, we injected the real service (realService) into the proxy via **constructor injection**:

public class TransactionalProxy implements TransferService {  
 private final TransferService realService;  
  
 public TransactionalProxy(TransferService realService) { // Injected via constructor  
 this.realService = realService;  
 }  
  
 @Override  
 public void transferMoney(String fromAccount, String toAccount, double amount) {  
 System.out.println("[Proxy] Starting transaction...");  
 try {  
 realService.transferMoney(fromAccount, toAccount, amount); // Object is already created  
 System.out.println("[Proxy] Committing transaction...");  
 } catch (Exception e) {  
 System.out.println("[Proxy] Rolling back transaction...");  
 throw e;  
 }  
 }  
}

**The Problem**

By injecting the real object into the proxy **at the time of proxy creation**, we are **immediately creating the expensive object**, even if transferMoney() is never called. This contradicts one of the primary benefits of the proxy pattern—**delaying the cost of expensive object creation until it’s actually needed**.

**A Better Approach: Lazy Instantiation**

To fully take advantage of the proxy pattern, we should **only create the real object when it is actually needed**. Instead of injecting the real service upfront, we can initialize it inside the proxy **only when the method is called**:

public class TransactionalProxy implements TransferService {  
 private TransferService realService; // No instantiation at construction  
 private final String serviceType; // Just an identifier  
  
 public TransactionalProxy(String serviceType) {  
 this.serviceType = serviceType;  
 }  
  
 @Override  
 public void transferMoney(String fromAccount, String toAccount, double amount) {  
 if (realService == null) { // Create only when needed  
 System.out.println("[Proxy] Creating the real transfer service...");  
 realService = new RealTransferService();  
 }  
  
 System.out.println("[Proxy] Starting transaction...");  
 try {  
 realService.transferMoney(fromAccount, toAccount, amount);  
 System.out.println("[Proxy] Committing transaction...");  
 } catch (Exception e) {  
 System.out.println("[Proxy] Rolling back transaction...");  
 throw e;  
 }  
 }  
}

**Why is This a Better Approach?**

**Defers Object Creation** — The real service is only created when transferMoney() is actually called.  
 **Optimized Resource Usage** – If the method is never called, we don’t waste resources creating the object.  
**Better Scalability** – Helps in large-scale applications where creating multiple objects upfront would be costly.

**Tips:**

If the real object is **expensive to create** and may **not always be used**, then **lazy instantiation using new inside the proxy is the better choice**. It ensures we only "pay the price" when the object is actually needed.

However, **dependency injection** is still beneficial in cases where:

* The object is **lightweight** and frequently used.
* The dependency is required **across multiple method calls**.
* The service benefits from **singleton scope** instead of creating a new instance every time.

**Caching with Proxies**

**The Problem**

Fetching frequently accessed data, such as exchange rates or account balances, can strain the database and reduce performance. How can we minimize repetitive database calls?

**The Solution: Caching**

Spring Boot provides caching annotations like @Cacheable, which are proxy-based. These annotations store method results in a cache and return the cached data for subsequent calls.

Example: Caching Exchange Rates

@Service  
public class ExchangeRateService {  
  
 @Cacheable("exchangeRates")  
 public double getExchangeRate(String currency) {  
 System.out.println("Fetching exchange rate for " + currency);  
 // Simulate fetching from a database or API  
 return Math.random();  
 }  
}

**How it works:**

* The proxy checks if the result for the given currency exists in the cache.
* If cached data exists, it returns the cached value; otherwise, it fetches the data, stores it in the cache, and returns it.

Example: Proxy for Caching

public class CachingProxy implements ExchangeRateService {  
 private final ExchangeRateService realService;  
 private final Map<String, Double> cache = new HashMap<>();  
  
 public CachingProxy(ExchangeRateService realService) {  
 this.realService = realService;  
 }  
  
 @Override  
 public double getExchangeRate(String currency) {  
 if (cache.containsKey(currency)) {  
 System.out.println("[Proxy] Returning cached value for " + currency);  
 return cache.get(currency);  
 }  
 double rate = realService.getExchangeRate(currency);  
 cache.put(currency, rate);  
 return rate;  
 }  
}

**Where is the cache stored?**

* By default, Spring uses an in-memory cache, but you can configure external caches like Redis or Ehcache.

**What Are Cross-Cutting Concerns?**

**Definition**

Cross-cutting concerns are functionalities that affect multiple parts of an application, such as:

* Logging
* Security
* Caching
* Transaction Management

**Why proxies?** Proxies allow you to implement cross-cutting concerns without modifying the core business logic, ensuring clean, maintainable code.