**Template method design pattern in controllers**

**Understanding the Template Design Pattern:**

*The Template Design Pattern falls under the behavioral design patterns category, focusing on defining the skeleton of an algorithm in the superclass while allowing subclasses to override certain steps of the algorithm. This pattern promotes code reusability and provides a structure for algorithm execution.*

**Implementation in Spring Boot Application:**

Let us consider a scenario where we need to implement a product import feature in our Spring Boot application. We’ll define a template for importing products, with specific steps such as reading data from a file, parsing the data, and persisting it into the database. Subclasses will then override the parsing logic based on the file format (e.g., CSV, JSON).

***Detailed Code Example 1:***

Template abstract class —

public abstract class AbstractProductImportTemplate {  
  
 public void importProducts(String filePath) {  
 List<Product> products = readDataFromFile(filePath);  
 List<Product> parsedProducts = parseData(products);  
 saveProducts(parsedProducts);  
 }  
  
 protected abstract List<Product> readDataFromFile(String filePath);  
  
 protected abstract List<Product> parseData(List<Product> rawData);  
  
 protected void saveProducts(List<Product> products) {  
 // Logic to persist products into the database  
 }  
}

Concrete classes —

@Component  
public class CSVProductImport extends AbstractProductImportTemplate {  
  
 @Override  
 protected List<Product> readDataFromFile(String filePath) {  
 // Logic to read CSV file and convert to list of products  
 }  
  
 @Override  
 protected List<Product> parseData(List<Product> rawData) {  
 // Logic to parse CSV data (specific to CSV format)  
 }  
}

@Component  
public class JSONProductImport extends AbstractProductImportTemplate {  
  
 @Override  
 protected List<Product> readDataFromFile(String filePath) {  
 // Logic to read JSON file and convert to list of products  
 }  
  
 @Override  
 protected List<Product> parseData(List<Product> rawData) {  
 // Logic to parse JSON data (specific to JSON format)  
 }  
}

Service Class —

@Service  
public class ProductImportService {  
  
 @Autowired  
 private List<AbstractProductImportTemplate> importTemplates;  
  
 public void importProducts(String filePath, String fileFormat) {  
 AbstractProductImportTemplate importTemplate = importTemplates.stream()  
 .filter(template -> template.getClass().getSimpleName().startsWith(fileFormat.toUpperCase()))  
 .findFirst()  
 .orElseThrow(() -> new IllegalArgumentException("Unsupported file format"));  
  
 importTemplate.importProducts(filePath);  
 }  
}

In the above code:

* **AbstractProductImportTemplate** defines the template for importing products, with abstract methods for reading data from a file and parsing it.
* **CSVProductImport** and **JSONProductImport** are subclasses that implement specific logic for reading and parsing CSV and JSON files, respectively.
* **ProductImportService** orchestrates the import process, dynamically selecting the appropriate import template based on the file format.

By employing the Template Design Pattern in our Spring Boot application, we have achieved a modular and extensible solution for importing products from different file formats. This approach promotes code reuse, maintainability, and flexibility, making it easier to accommodate future changes and enhancements in the import functionality.

***Example 2 —***

***Let us consider a different scenario where we’re building an email notification system within our Spring Boot eCommerce application. We’ll use the Template Design Pattern to create a flexible structure for sending different types of email notifications, such as order confirmation, shipping updates, and promotional offers.***

Code Example:

Abstract Template class —

public abstract class AbstractEmailNotificationTemplate {  
  
 public void sendNotification(String recipient) {  
 String subject = constructSubject();  
 String body = constructBody();  
 sendEmail(recipient, subject, body);  
 }  
  
 protected abstract String constructSubject();  
  
 protected abstract String constructBody();  
  
 private void sendEmail(String recipient, String subject, String body) {  
 // Logic to send email  
 System.out.println("Email sent to " + recipient + " with subject: " + subject + " and body: " + body);  
 }  
}

Concrete classes —

@Component  
public class OrderConfirmationEmailTemplate extends AbstractEmailNotificationTemplate {  
  
 @Override  
 protected String constructSubject() {  
 return "Order Confirmation";  
 }  
  
 @Override  
 protected String constructBody() {  
 return "Dear Customer, your order has been confirmed.";  
 }  
}

@Component  
public class ShippingUpdateEmailTemplate extends AbstractEmailNotificationTemplate {  
  
 @Override  
 protected String constructSubject() {  
 return "Shipping Update";  
 }  
  
 @Override  
 protected String constructBody() {  
 return "Dear Customer, your order has been shipped.";  
 }  
}

@Component  
public class PromotionalOfferEmailTemplate extends AbstractEmailNotificationTemplate {  
  
 @Override  
 protected String constructSubject() {  
 return "Exclusive Offer";  
 }  
  
 @Override  
 protected String constructBody() {  
 return "Dear Customer, check out our latest promotional offers!";  
 }  
}

@Component  
public class OrderCancellationEmailTemplate extends AbstractEmailNotificationTemplate {  
  
 @Override  
 protected String constructSubject() {  
 return "Order Cancellation";  
 }  
  
 @Override  
 protected String constructBody() {  
 return "Dear Customer, your order has been canceled.";  
 }  
}

@Component  
public class PasswordResetEmailTemplate extends AbstractEmailNotificationTemplate {  
  
 @Override  
 protected String constructSubject() {  
 return "Password Reset";  
 }  
  
 @Override  
 protected String constructBody() {  
 return "Dear User, a password reset request has been initiated.";  
 }  
}

Service class —

@Service  
public class EmailNotificationService {  
  
 @Autowired  
 private List<AbstractEmailNotificationTemplate> emailTemplates;  
  
 public void sendNotification(String recipient, String notificationType) {  
 AbstractEmailNotificationTemplate emailTemplate = emailTemplates.stream()  
 .filter(template -> template.getClass().getSimpleName().startsWith(notificationType.toUpperCase()))  
 .findFirst()  
 .orElseThrow(() -> new IllegalArgumentException("Unsupported notification type"));  
  
 emailTemplate.sendNotification(recipient);  
 }  
}

In this example:

* **AbstractEmailNotificationTemplate** defines the template for sending email notifications, with abstract methods for constructing the subject and body of the email.
* **OrderConfirmationEmailTemplate**, **ShippingUpdateEmailTemplate**, and **PromotionalOfferEmailTemplate** are subclasses that implement specific logic for different types of email notifications.
* **OrderCancellationEmailTemplate**: Sends an email notification to inform the customer about the cancellation of their order.
* **PasswordResetEmailTemplate**: Sends an email notification to inform the user about a password reset request. EmailNotificationService orchestrates the sending of email notifications, dynamically selecting the appropriate email template based on the notification type.

By utilizing the Template Design Pattern, we have created a modular and extensible solution for sending various types of email notifications within our Spring Boot eCommerce application. This approach enables us to easily add new types of notifications in the future without modifying existing code, promoting code reuse and maintainability. As you continue to enhance your eCommerce platform, consider incorporating design patterns like the Template Pattern to improve the flexibility and scalability of your application.

***Example 3 —***

***In the context of a Spring Boot application, let us consider a scenario where we have different types of products (e.g., electronics, clothing, books) and we want to generate invoices for orders. We can use the Template design pattern to create a generic Invoice template and then let specific product types override certain methods to customize the invoice generation process.***

***Here’s an example implementation:***

Abstract Template class —

import java.util.List;  
  
// Abstract class defining the template for generating an invoice  
public abstract class InvoiceTemplate {  
   
 // Template method to generate an invoice  
 public final void generateInvoice(Order order) {  
 generateHeader();  
 generateOrderDetails(order);  
 generateFooter();  
 }  
   
 // Abstract methods to be implemented by subclasses  
 protected abstract void generateHeader();  
 protected abstract void generateOrderDetails(Order order);  
 protected abstract void generateFooter();  
}

Concrete subclass for generating invoice for **electronics products -**

public class ElectronicsInvoice extends InvoiceTemplate {  
   
 @Override  
 protected void generateHeader() {  
 // Generate header specific to electronics invoice  
 System.out.println("Electronics Invoice");  
 }  
   
 @Override  
 protected void generateOrderDetails(Order order) {  
 // Generate order details specific to electronics products  
 System.out.println("Ordered Electronics:");  
 for (Product product : order.getProducts()) {  
 if (product instanceof Electronics) {  
 System.out.println(product.getName() + " - $" + product.getPrice());  
 }  
 }  
 }  
   
 @Override  
 protected void generateFooter() {  
 // Generate footer specific to electronics invoice  
 System.out.println("Thank you for shopping electronics with us!");  
 }  
}

Concrete subclass for generating invoice for **clothing products -**

public class ClothingInvoice extends InvoiceTemplate {  
   
 @Override  
 protected void generateHeader() {  
 // Generate header specific to clothing invoice  
 System.out.println("Clothing Invoice");  
 }  
   
 @Override  
 protected void generateOrderDetails(Order order) {  
 // Generate order details specific to clothing products  
 System.out.println("Ordered Clothing:");  
 for (Product product : order.getProducts()) {  
 if (product instanceof Clothing) {  
 System.out.println(product.getName() + " - $" + product.getPrice());  
 }  
 }  
 }  
   
 @Override  
 protected void generateFooter() {  
 // Generate footer specific to clothing invoice  
 System.out.println("Thank you for shopping clothing with us!");  
 }  
}

Concrete subclass for generating invoice for **furniture products -**

public class FurnitureInvoice extends InvoiceTemplate {  
   
 @Override  
 protected void generateHeader() {  
 // Generate header specific to furniture invoice  
 System.out.println("Furniture Invoice");  
 }  
   
 @Override  
 protected void generateOrderDetails(Order order) {  
 // Generate order details specific to furniture products  
 System.out.println("Ordered Furniture:");  
 for (Product product : order.getProducts()) {  
 if (product instanceof Furniture) {  
 System.out.println(product.getName() + " - $" + product.getPrice());  
 }  
 }  
 }  
   
 @Override  
 protected void generateFooter() {  
 // Generate footer specific to furniture invoice  
 System.out.println("Thank you for shopping furniture with us!");  
 }  
}

Concrete subclass for generating invoice for **food products -**

public class FoodInvoice extends InvoiceTemplate {  
   
 @Override  
 protected void generateHeader() {  
 // Generate header specific to food invoice  
 System.out.println("Food Invoice");  
 }  
   
 @Override  
 protected void generateOrderDetails(Order order) {  
 // Generate order details specific to food products  
 System.out.println("Ordered Food:");  
 for (Product product : order.getProducts()) {  
 if (product instanceof Food) {  
 System.out.println(product.getName() + " - $" + product.getPrice());  
 }  
 }  
 }  
   
 @Override  
 protected void generateFooter() {  
 // Generate footer specific to food invoice  
 System.out.println("Thank you for shopping food with us!");  
 }  
}

**Order class representing an order containing multiple products —**

@Data  
public class Order {  
   
 private List<Product> products;  
   
 public Order(List<Product> products) {  
 this.products = products;  
 }  
   
 public List<Product> getProducts() {  
 return products;  
 }  
}

**Product class representing a generic product —**

@Data  
public class Product {  
   
 private String name;  
 private double price;  
   
}

**Concrete classes —**

public class Electronics extends Product {  
 // Additional properties and methods specific to electronics  
}  
  
  
  
public class Clothing extends Product {  
 // Additional properties and methods specific to clothing  
}  
  
  
  
public class Furniture extends Product {  
 // Additional properties and methods specific to furniture  
}  
  
  
  
public class Food extends Product {  
 // Additional properties and methods specific to food  
}

**Main Application class —**

public class ApplicationMain {  
 public static void main(String[] args) {  
 List<Product> products = List.of(  
 new Electronics("Laptop", 1000),  
 new Clothing("T-shirt", 20),  
 new Furniture("Sofa", 500),  
 new Food("Bread", 5)  
 );  
   
 Order order = new Order(products);  
   
 InvoiceTemplate electronicsInvoice = new ElectronicsInvoice();  
 electronicsInvoice.generateInvoice(order);  
   
 InvoiceTemplate clothingInvoice = new ClothingInvoice();  
 clothingInvoice.generateInvoice(order);  
   
 InvoiceTemplate furnitureInvoice = new FurnitureInvoice();  
 furnitureInvoice.generateInvoice(order);  
   
 InvoiceTemplate foodInvoice = new FoodInvoice();  
 foodInvoice.generateInvoice(order);  
 }  
}

Each invoice type has its own implementation of the **generateHeader**(), **generateOrderDetails**(), and **generateFooter**() methods tailored to the respective product categories. This approach simulates a real-life eCommerce application where invoices may vary based on the type of products being purchased.

As demonstrated in the above example, the Template Design Pattern enables the creation of diverse invoice types tailored to different product categories while maintaining a consistent structure. This approach ensures that as the application evolves, new functionalities and variations can be easily integrated, contributing to a more scalable and adaptable codebase. The Template Design Pattern, with its elegant simplicity, stands as a robust solution for designing modular and customizable systems in Java.