Template Method Pattern

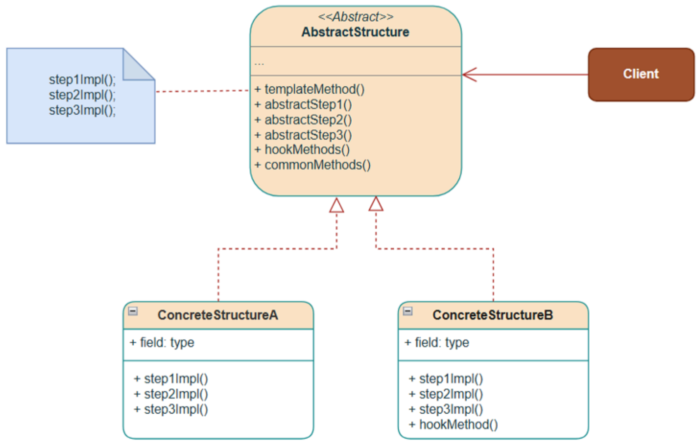
Resources

- 41. All Behavioral Design Patterns | Strategy, Observer, State, Template, Command, Visitor, Memento
- 39. Template Method Design Pattern Explanation in Java | Concept and Coding LLD | Low Level Design

Definition

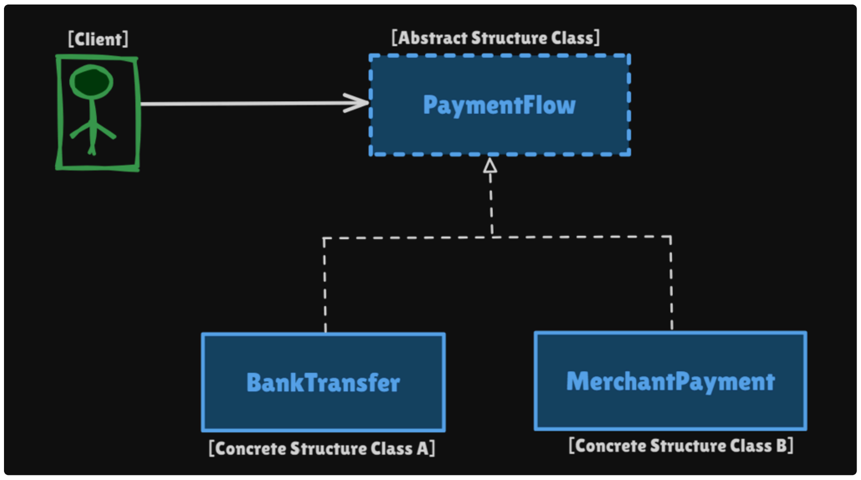
> **The Template Method pattern is a behavioral design pattern that defines the skeleton/structure of an algorithm(common workflow) in a base class and allows subclasses to override specific steps and provide custom implementation without changing the algorithm's core workflow.**

Class Diagram



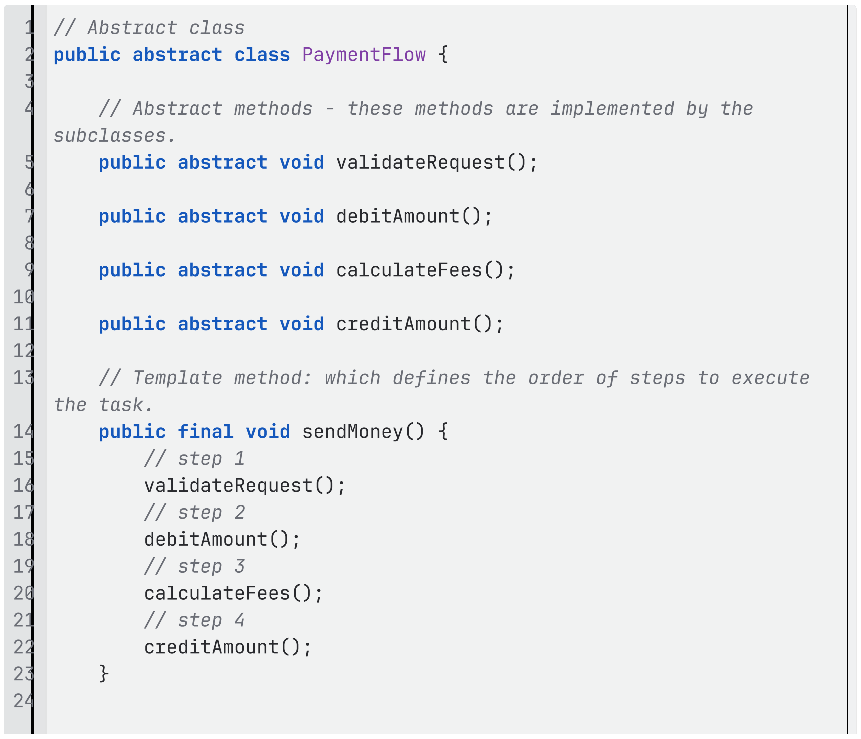
Structure of the Template Method Pattern

Let’s understand the Structure of the Template Method Pattern using the Payment Workflows example:


[Client]
[Abstract Structure Class]
PaymentFlow
BankTransfer
[Concrete Structure Class A]
MerchantPayment
[Concrete Structure Class B]

1. **Abstract Structure Class (e.g., `PaymentFlow`):**
   - Has a template method: Defines the sequence of steps for the algorithm (the skeleton of the payment processing workflow). Calls both abstract methods and hook methods in a specific sequence
   - Declares one or more abstract methods that must be implemented by each subclass. These define the varying parts(specific implementation) of the algorithm.
   - May contain hook methods: have default implementations. Subclasses can choose to override these for further customization if needed.
   - May also contain common methods: They are implemented once in the base class, and all subclasses share this common functionality.
2. **Concrete Structure Classes** (e.g., `BankTransfer`, `MerchantPayment`)
   - These classes extend the abstract base class and provide specific implementations for the abstract methods defined in the template.
   - They also optionally override hook methods to provide customization of business workflows. Improves flexibility.

Implementation(Example: Payment Workflows)



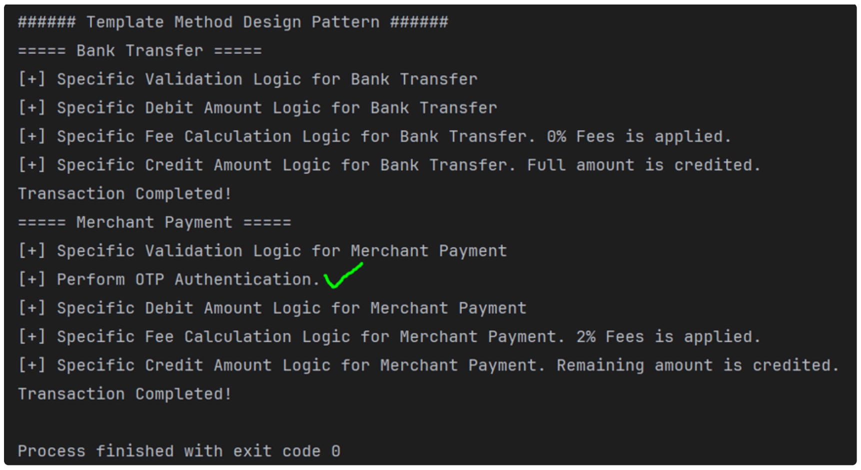
```java
// Abstract class
public abstract class PaymentFlow {

    // Abstract methods - these methods are implemented by the subclasses.
    public abstract void validateRequest();

    public abstract void debitAmount();

    public abstract void calculateFees();

    public abstract void creditAmount();

    // Template method: which defines the order of steps to execute the task.
    public final void sendMoney() {
        // step 1
        validateRequest();
        // step 2
        debitAmount();
        // step 3
        calculateFees();
        // step 4
        creditAmount();
    }

    // Hook method: which can be overridden by the subclasses.
    protected boolean requiresOTPAuthentication() {
        return false; // Default: authentication not required
    }

    // Common method: All subclasses share this common functionality.
    public void logTransaction() {
        System.out.println("Transaction Completed!");
    }
}
```

```java
// Concrete class
public class BankTransfer extends PaymentFlow {
    @Override
    public void validateRequest() {
        System.out.println("[+] Specific Validation Logic for Bank 
Transfer");
    }

    @Override
    public void debitAmount() {
        System.out.println("[+] Specific Debit Amount Logic for Bank 
Transfer");
    }

    @Override
    public void calculateFees() {
        System.out.println("[+] Specific Fee Calculation Logic for 
Bank Transfer. 0% Fees is applied.");
    }

    @Override
    public void creditAmount() {
        System.out.println("[+] Specific Credit Amount Logic for Bank 
Transfer. Full amount is credited.");
    }

}
```

```java
// Concrete class
public class MerchantPayment extends PaymentFlow {
    @Override
    public void validateRequest() {
        System.out.println("[+] Specific Validation Logic for Merchant 
Payment");
    }

    @Override
    public void debitAmount() {
        if (requiresOTPAuthentication()) {
            System.out.println("[+] Perform OTP Authentication.");
        }
        System.out.println("[+] Specific Debit Amount Logic for 
Merchant Payment");
    }

    @Override
    public void calculateFees() {
        System.out.println("[+] Specific Fee Calculation Logic for 
Merchant Payment. 2% Fees is applied.");
    }

    @Override
    public void creditAmount() {
        System.out.println("[+] Specific Credit Amount Logic for 
Merchant Payment. Remaining amount is credited.");
    }

    // Hook method - overridden by subclass
    @Override
    protected boolean requiresOTPAuthentication() {
        return true;
    }
}
```

```java
// Client class
public class TemplateDemo {
    public static void main(String[] args) {
        System.out.println("###### Template Method Design Pattern ######");

        // Bank Transfer
        System.out.println("===== Bank Transfer =====");
        PaymentFlow bankTransfer = new BankTransfer();
        bankTransfer.sendMoney(); // Template method
        bankTransfer.logTransaction(); // Common method

        // Merchant Payment
        System.out.println("===== Merchant Payment =====");
        PaymentFlow merchantPayment = new MerchantPayment();
        merchantPayment.sendMoney(); // Template method
        merchantPayment.logTransaction(); // Common method
    }
}
```

Output



```
###### Template Method Design Pattern ######
===== Bank Transfer =====
[+] Specific Validation Logic for Bank Transfer
[+] Specific Debit Amount Logic for Bank Transfer
[+] Specific Fee Calculation Logic for Bank Transfer. 0% Fees is applied.
[+] Specific Credit Amount Logic for Bank Transfer. Full amount is credited.
Transaction Completed!
===== Merchant Payment =====
[+] Specific Validation Logic for Merchant Payment
[+] Perform OTP Authentication. ✓
[+] Specific Debit Amount Logic for Merchant Payment
[+] Specific Fee Calculation Logic for Merchant Payment. 2% Fees is applied.
[+] Specific Credit Amount Logic for Merchant Payment. Remaining amount is credited.
Transaction Completed!

Process finished with exit code 0
```