Facade

Resources

- Video → 32. All Structural Design Patterns | Decorator, Proxy, Composite, Adapter, Bridge, Facade, FlyWeight
- Video → 25. Facade Design Pattern with Example | Facade Low Level Design Pattern | Facade Pattern LLD Java

Definition

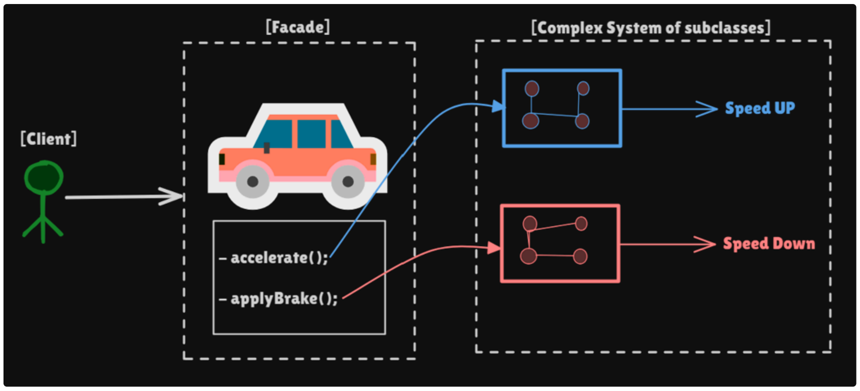
> ***The Facade Design Pattern provides a simplified interface to a complex subsystem for the client to interact with it seamlessly.***

Facade Pattern Use-cases

When and why do we use the Facade Design Pattern in the code?

Whenever we have to **hide system complexity** from the client, we use the Facade Design Pattern.

Real Life Example: Car

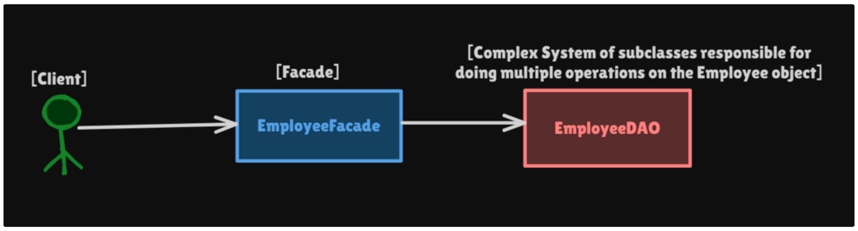


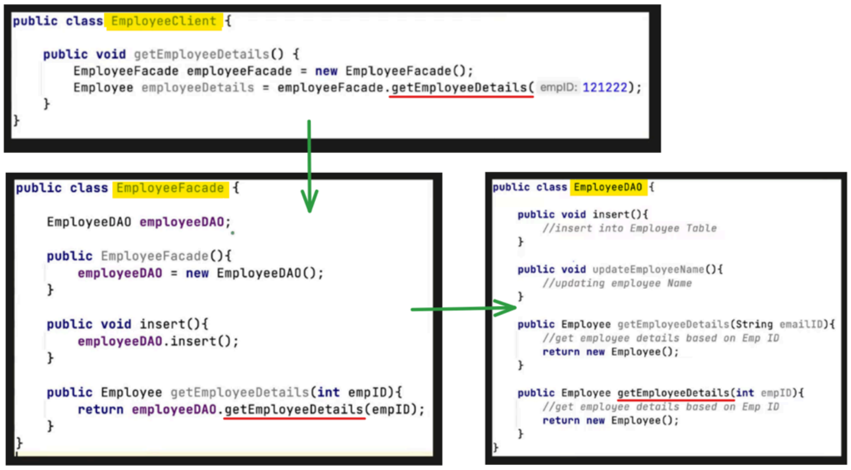
Think of it like using `accelerator()` and `brake()` features in a `Car`. The `Car` conceals the intricate complexities of smaller hardware processes (subsystems), which are interconnected to sequentially manage the actions (speeding or slowing down) once the driver presses the pedals while driving.

> It is also important to note that the **client doesn't need** to use a facade to communicate with the subsystem components. It can always choose to communicate with the components and carry out the execution by itself.

Scenario 1: Employee Operations

Expose only a few methods/features the client is interested in using.


[Client]
[Facade]
EmployeeFacade
[Complex System of subclasses responsible for doing multiple operations on the Employee object]
EmployeeDAO

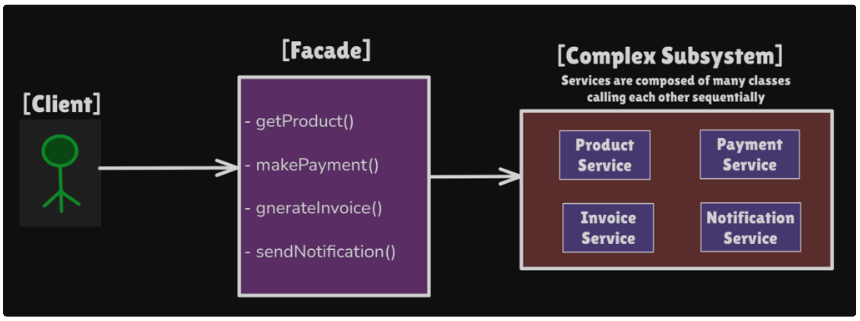


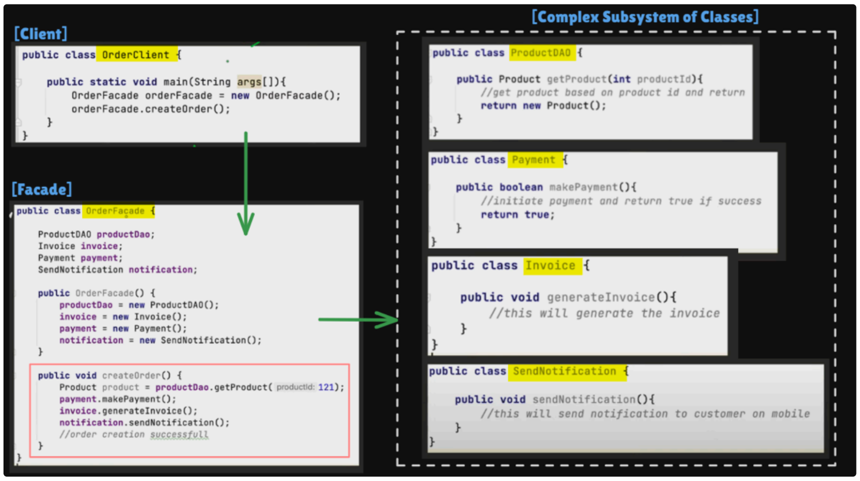
```java
public class EmployeeClient {

    public void getEmployeeDetails() {
        EmployeeFacade employeeFacade = new EmployeeFacade();
        Employee employeeDetails = employeeFacade.getEmployeeDetails( empID: 121222);
    }
}
```

```java
public class EmployeeFacade {

    EmployeeDAO employeeDAO;

    public EmployeeFacade(){
        employeeDAO = new EmployeeDAO();
    }

    public void insert(){
        employeeDAO.insert();
    }

    public Employee getEmployeeDetails(int empID){
        return employeeDAO.getEmployeeDetails(empID);
    }
}
```

```java
public class EmployeeDAO {

    public void insert(){
        //insert into Employee Table
    }

    public void updateEmployeeName(){
        //updating employee Name
    }

    public Employee getEmployeeDetails(String emailID){
        //get employee details based on Emp ID
        return new Employee();
    }

    public Employee getEmployeeDetails(int empID){
        //get employee details based on Emp ID
        return new Employee();
    }
}
```

Scenario 2: Order Processing





```java
public class OrderClient {

    public static void main(String args[]){
        OrderFacade orderFacade = new OrderFacade();
        orderFacade.createOrder();
    }
}
```

```java
public class OrderFacade {

    ProductDAO productDao;
    Invoice invoice;
    Payment payment;
    SendNotification notification;

    public OrderFacade() {
        productDao = new ProductDAO();
        invoice = new Invoice();
        payment = new Payment();
        notification = new SendNotification();
    }

    public void createOrder() {
        Product product = productDao.getProduct( productId: 121);
        payment.makePayment();
        invoice.generateInvoice();
        notification.sendNotification();
        //order creation successfull
    }
}
```

```java
public class ProductDAO {

    public Product getProduct(int productId){
        //get product based on product id and return
        return new Product();
    }
}
```

```java
public class Payment {

    public boolean makePayment(){
        //initiate payment and return true if success
        return true;
    }
}
```

```java
public class Invoice {

    public void generateInvoice(){
        //this will generate the invoice
    }
}
```

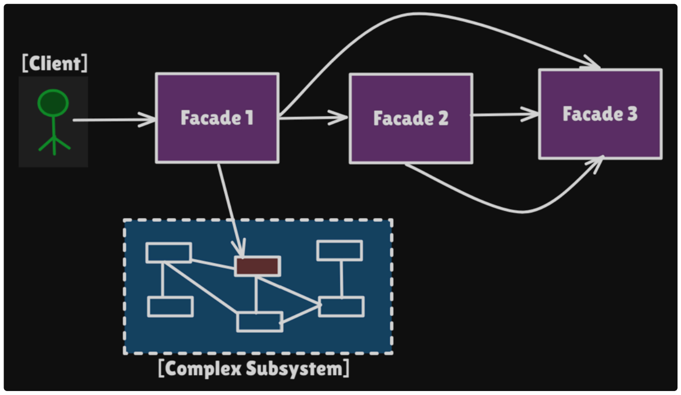
```java
public class SendNotification {

    public void sendNotification(){
        //this will send notification to customer on mobile
    }
}
```

The Problem(Without the Facade Pattern)

- The client has to talk to every subsystem directly and implement the proper sequence of execution order.
- No data hiding or encapsulation. The client has too much exposure to subsystems, making it tightly coupled with the internal structure.
- Hard to change/introduce new steps. If the order process changes (e.g., new `DiscountService` or different payment flow), you must update every client that uses it.
- It would be more error-prone, e.g., the Client might forget a step (e.g., `sendConfirmation()`).

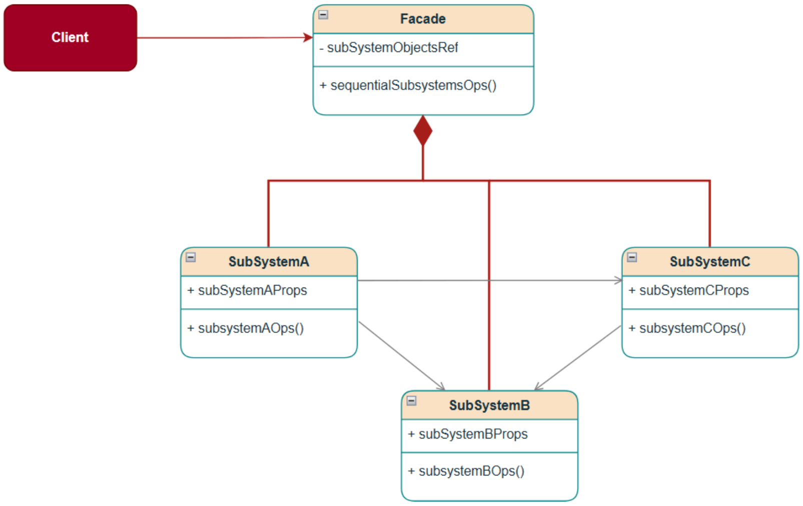
  Or call it in the wrong order (payment before stock availability check).

Scenario 3: Facade Using another Facade



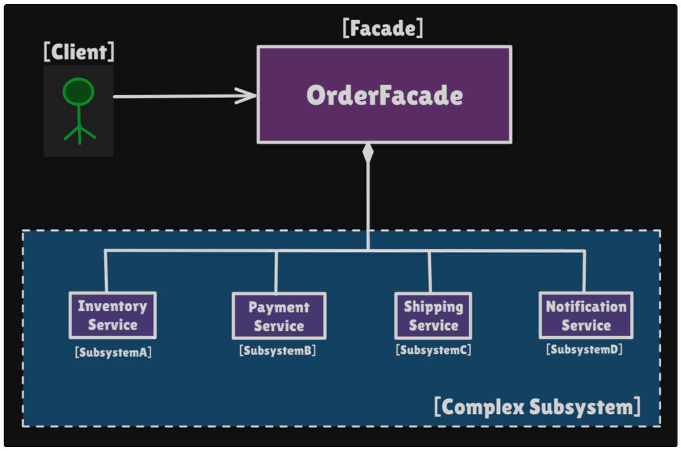
A facade can use another facade or a subsystem directly, depending on the requirement and complexity. The facade pattern provides a single entry point to the client for interaction.

Class Diagram



Structure Of Facade Pattern

Let's look at the Structure of the Facade Pattern using the Order Processing example as discussed above.



- **Facade (`OrderFacade`):** Holds references to all downstream subsystems(that are working together to complete an action) and defines a method to perform the required operation, i.e., `OrderFacade.placeOrder()` .
- **SubsystemA (`InventoryService`):** This service checks if items are in stock.
- **SubsystemB (`PaymentService`):** This service processes payments.
- **SubsystemC (`ShippingService`):** This service arranges delivery.
- **SubsystemD (`NotificationService`):** This service sends confirmation.
- **Client(`ECommerceApp`):** Client places an order with just one call to the facade class `OrderFacade.placeOrder()` .

Implementation

Let's look at the Implementation of the Facade Pattern using the **Order Processing example** as discussed above.

```java
// Complex Subsytems

// Subsystem A: Inventory
public class InventoryService {
    public boolean checkStock(String productId) {
        System.out.println("Checking stock for product: " + productId);
        return true; // assume it's always in stock
    }
}
// Subsystem B: Payment
public class PaymentService {
    public boolean makePayment(String paymentMethod) {
        System.out.println("Processing payment using: " + paymentMethod);
        return true; // assume payment always succeeds
    }
}
// Subsystem C: Shipping
public class ShippingService {
    public void shipProduct(String productId) {
        System.out.println("Shipping product: " + productId);
    }
}
// Subsystem D: Notification
public class NotificationService {
    public void sendConfirmation(String productId) {
        System.out.println("Sending order confirmation for product: " + productId);
    }
}
```

```java
// Facade hides complexity and provides a simple unified interface
public class OrderFacade {
    private final InventoryService inventory;
    private final PaymentService payment;
    private final ShippingService shipping;
    private final NotificationService notification;

    public OrderFacade() {
        this.inventory = new InventoryService();
        this.payment = new PaymentService();
        this.shipping = new ShippingService();
        this.notification = new NotificationService();
    }

    // Simplified method for clients
    public void placeOrder(String productId, String paymentMethod) {

        // The following steps are hidden from the client and need to be executed in a specific order
        System.out.println("Placing order for product: " + productId);

        // Step 1: Check stock
        if (!inventory.checkStock(productId)) {
            System.out.println("Product out of stock!");
            return;
        }

        // Step 2: Make payment
        if (!payment.makePayment(paymentMethod)) {
            System.out.println("Payment failed!");
            return;
        }

        // Step 3: Ship product
        shipping.shipProduct(productId);

        // Step 4: Send confirmation
        notification.sendConfirmation(productId);

        System.out.println("Order placed successfully!");
    }
}
```

```java
// Client Usage
public class ECommerceApp {
    public static void main(String[] args) {
        System.out.println("====== Facade Design Pattern Demo ======");
        // Client interacts with a simple Facade, not with all subsystems.
        OrderFacade orderFacade = new OrderFacade();

        // Place order with one call to Facade
        orderFacade.placeOrder("MacBook Pro", "Credit Card");

        // Place another order with one call to Facade
        orderFacade.placeOrder("Cricket Bat", "UPI");
    }
}
```

Facade vs Proxy Pattern

Facade and Proxy are both structural patterns, but they solve different problems.

Intent

- **Facade Pattern:** Provides a simplified interface to a complex subsystem to facilitate clients' interaction. The main purpose is to reduce complexity for clients. It is associated with references to many objects involved in the specific task.
- **Proxy Pattern:** Acts as a surrogate/placeholder/substitute for another object to control access to it. It is used to add more control (security, logging, caching, lazy loading) and is associated with only ONE object directly.

Facade vs Adapter Pattern

Facade and Adapter often look alike because both give a different interface to clients, but differ in the following:

Intent

- **Facade Pattern:** Provides a simplified interface to a complex subsystem to facilitate clients' interaction and hide the complexity.
- **Adapter Pattern:** Provides a way to use an existing incompatible interface that the client expects. Improves compatibility, making two interfaces work together.