ĐẠI HỌC FPT

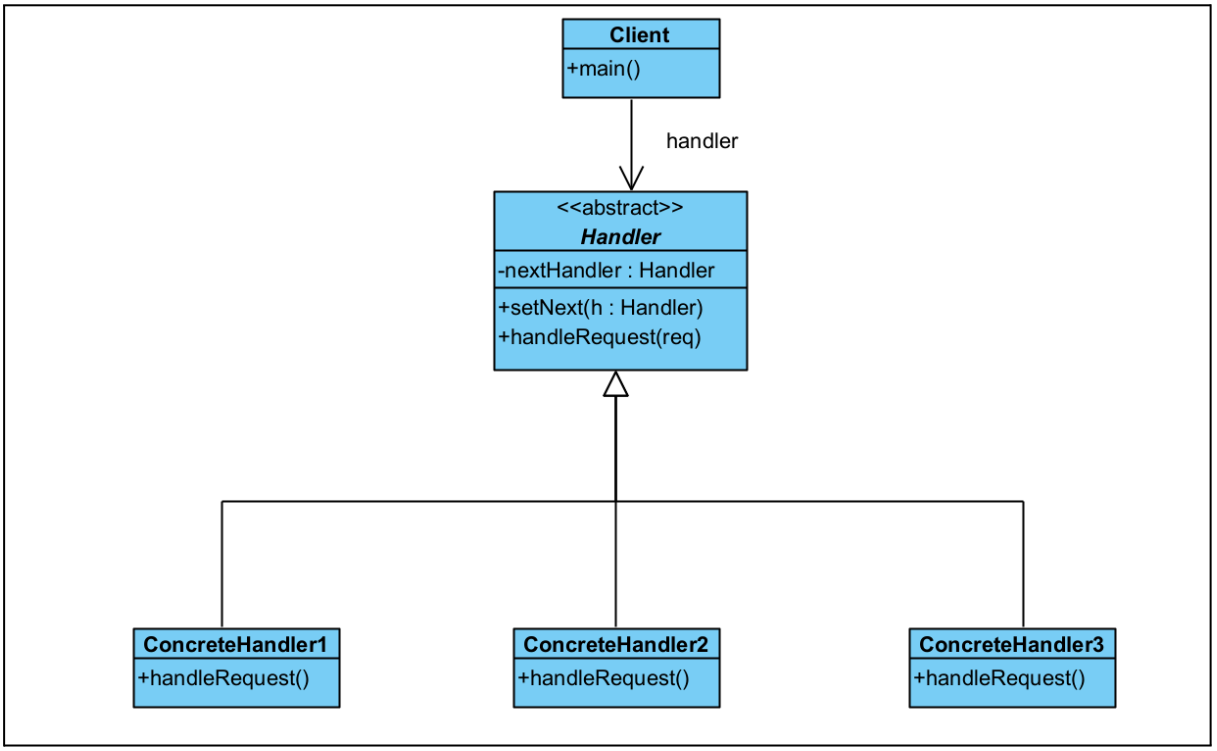
Behavioral Design Pattern

Ha Noi, October 3, 2025

I. Overall about patterns in Behavioral Design Pattern

Chain of Responsibility pattern

- Purpose: Allows a request to be passed along a chain of handlers, where each handler decides either to process the request or pass it to the next handler in the chain.
- Use case: When multiple objects can handle a request, but the specific handler is determined at runtime. This pattern decouples the sender of a request from its receiver, allowing flexible assignment and order of handling responsibilities.
- Structure (UML):

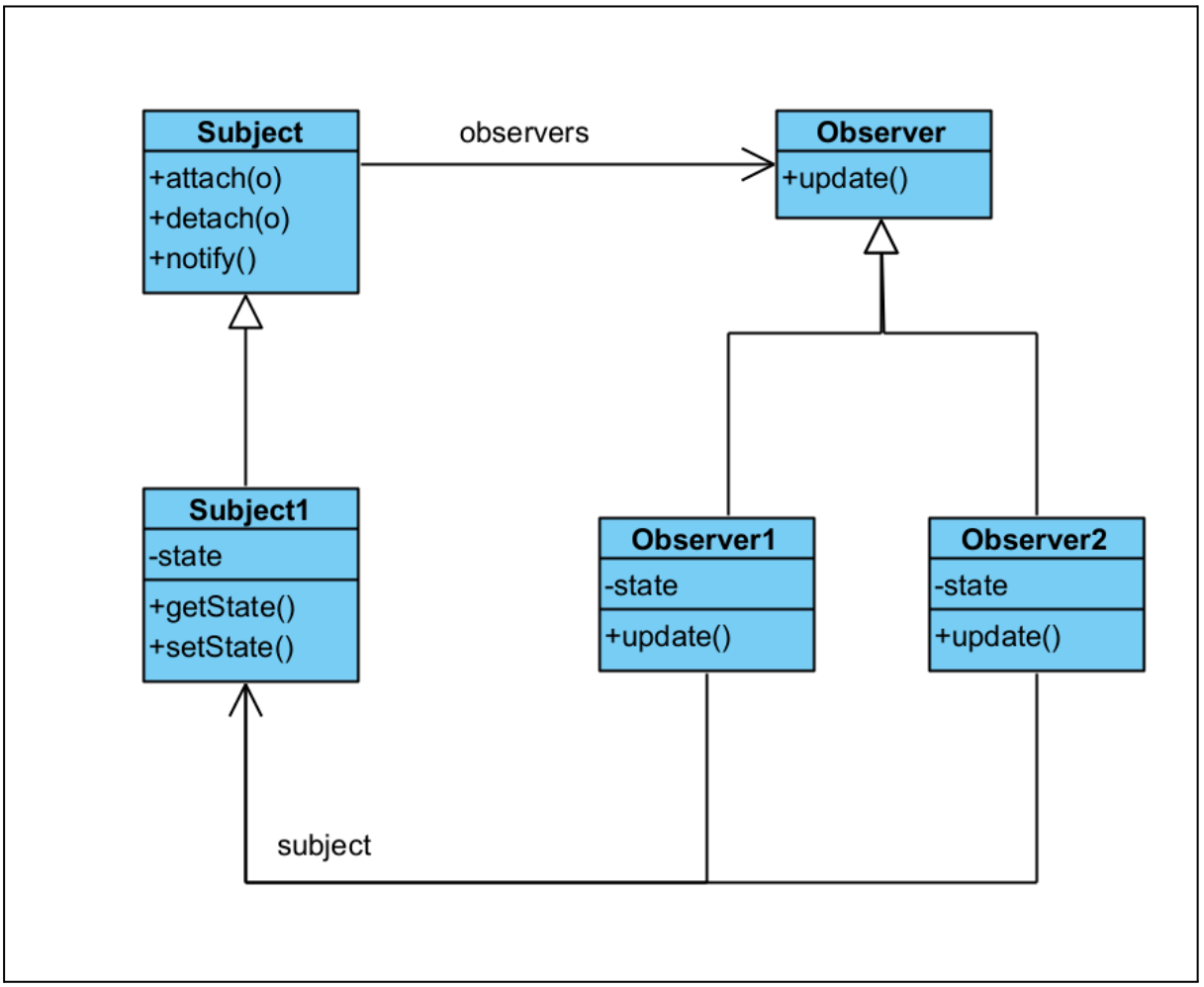


- Participating Components:
  - Client: Initiates the request and sends it to the first handler in the chain.
  - Handler (abstract class): Declares an interface for handling requests and holds a reference to the next handler in the chain.
    - nextHandler: reference to the next handler.
    - setNext(h: Handler): sets the next handler.
    - handleRequest(req): defines the method for handling the request
  - ConcreteHandler1 / ConcreteHandler2 / ConcreteHandler3: Implement the handleRequest() method. Each handler decides whether to process the request or pass it to the next handler.

Observer Pattern

- Purpose: Defines a one-to-many dependency between objects so that when one object (the subject) changes its state, all its dependent objects (observers) are automatically notified and updated.

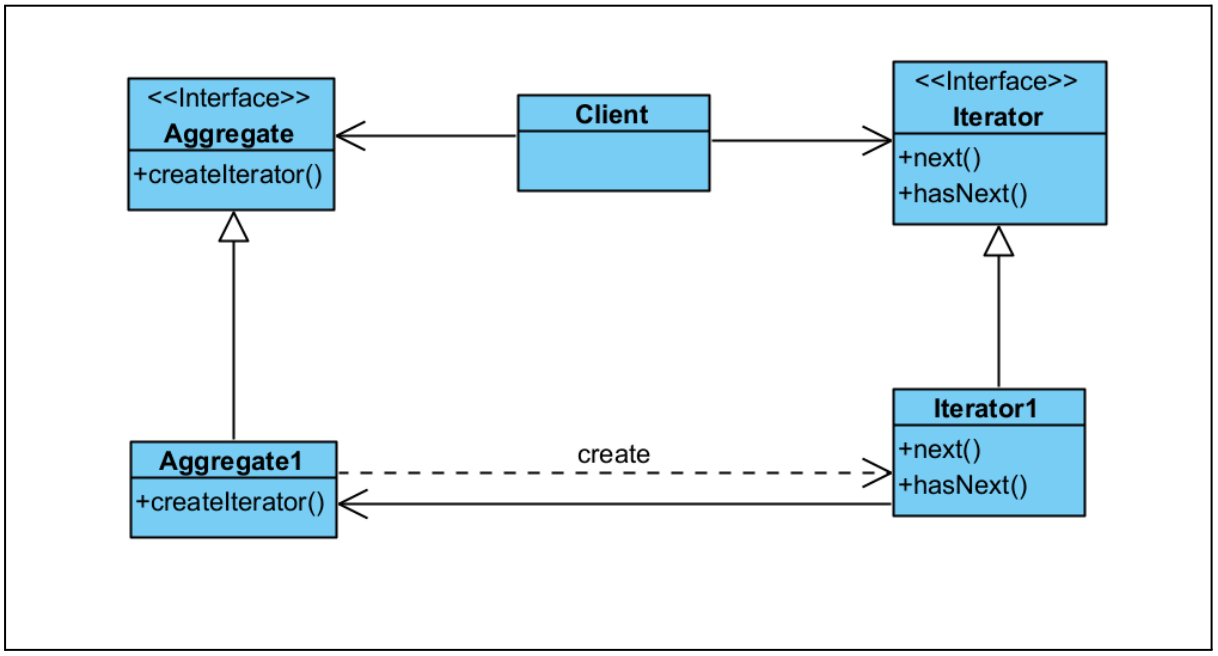
- Use case: When changes in one object need to be reflected automatically in other dependent objects, without tightly coupling them.
- Structure (UML):



- Participating Components:
  - Subject (abstract class): Maintains a list of observers and provides methods to attach, detach, and notify them.
    - attach(o): adds an observer to the list
    - detach(o): removes an observer from the list.
    - notify(): notifies all attached observers about a change.
  - ConcreteSubject (Subject1): Stores the state of interest to observers and sends notifications when the state changes.
    - state: the data that observers depend on.
    - getState(): returns the current state.
    - setState(): updates the state and triggers notify().
  - Observer (interface): Defines the update() method that subjects call to notify observers.
  - ConcreteObservers (Observer1, Observer2): Implement the update() method to synchronize their state with the subject's state. Each observer maintains its own copy of the subject's state and updates it when notified.

Iterator Pattern

- Purpose: Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- Use case: When you want to traverse elements of a collection without revealing how the collection is implemented
- Structure (UML):



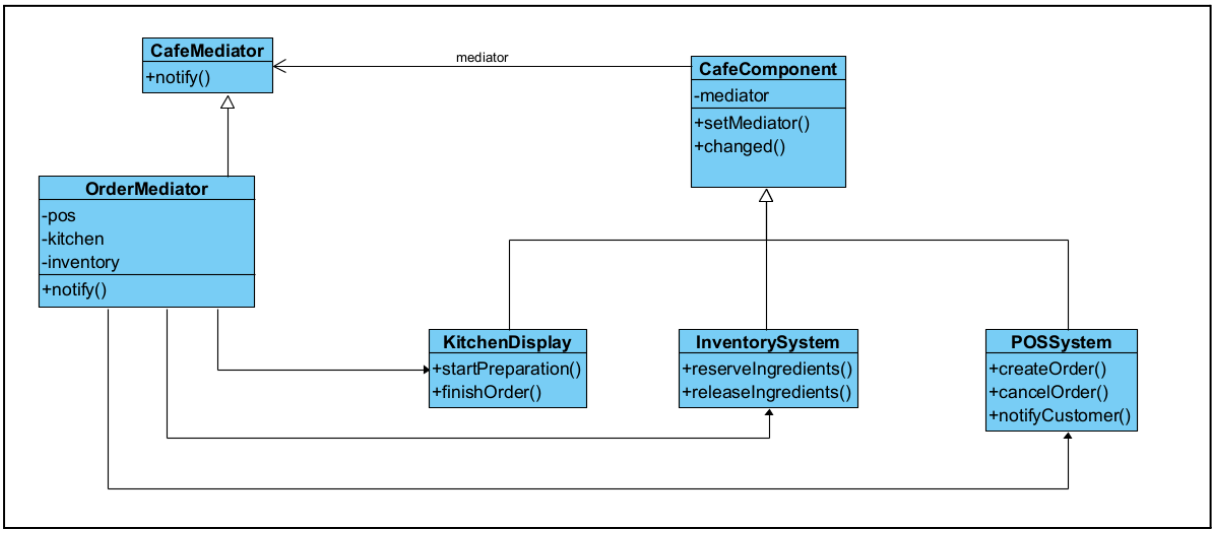
- Participating Components:
  - Client: Uses the iterator object to traverse elements of the aggregate without knowing its internal structure.
  - Iterator (interface): Declares the operations for accessing and traversing elements.
    - next(): returns the next element in the collection.
    - hasNext(): checks if there are more elements to iterate
  - ConcreteIterator (Iterator1): Implements the Iterator interface and keeps track of the current position in the traversal.
    - state: the data that observers depend on.
    - getState(): returns the current state.
    - setState(): updates the state and triggers notify().
  - Aggregate (interface): Declares the factory method createIterator() that returns an iterator for the collection.
  - ConcreteAggregate (Aggregate1): Implements the createIterator() method to return an instance of the corresponding ConcreteIterator. It stores the collection of items and delegates iteration logic to the iterator.

Mediator Pattern

Purpose: Coordinate communication between multiple café subsystems (POS, Kitchen, and Inventory) through a central mediator, reducing direct dependencies between them.

Use Case: When the café's order, kitchen, and inventory modules must collaborate — for example, creating an order triggers ingredient reservation and kitchen preparation — without components calling each other directly.

Structure (UML):



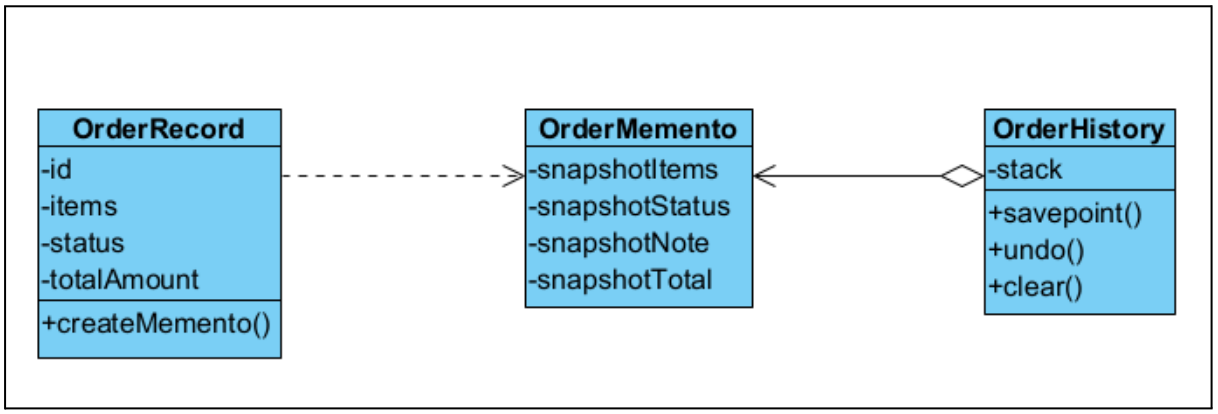
- Participating Components:
  - CafeMediator (abstract): Defines the communication interface between components.
  - OrderMediator (concrete): Implements coordination logic for orders, kitchen, and inventory.
  - CafeComponent (abstract): Base class for all components that communicate via the mediator.
  - POSSystem: Creates or cancels orders and notifies the mediator.
  - KitchenDisplay: Starts and finishes drink preparation when notified.
  - InventorySystem: Reserves or releases ingredients based on order events.

Memento Pattern

Purpose: Save and restore an object's previous state (e.g., an order or report) without exposing its internal details.

Use Case: When the café system needs to undo or restore an order to a previous version after changes or cancellations.

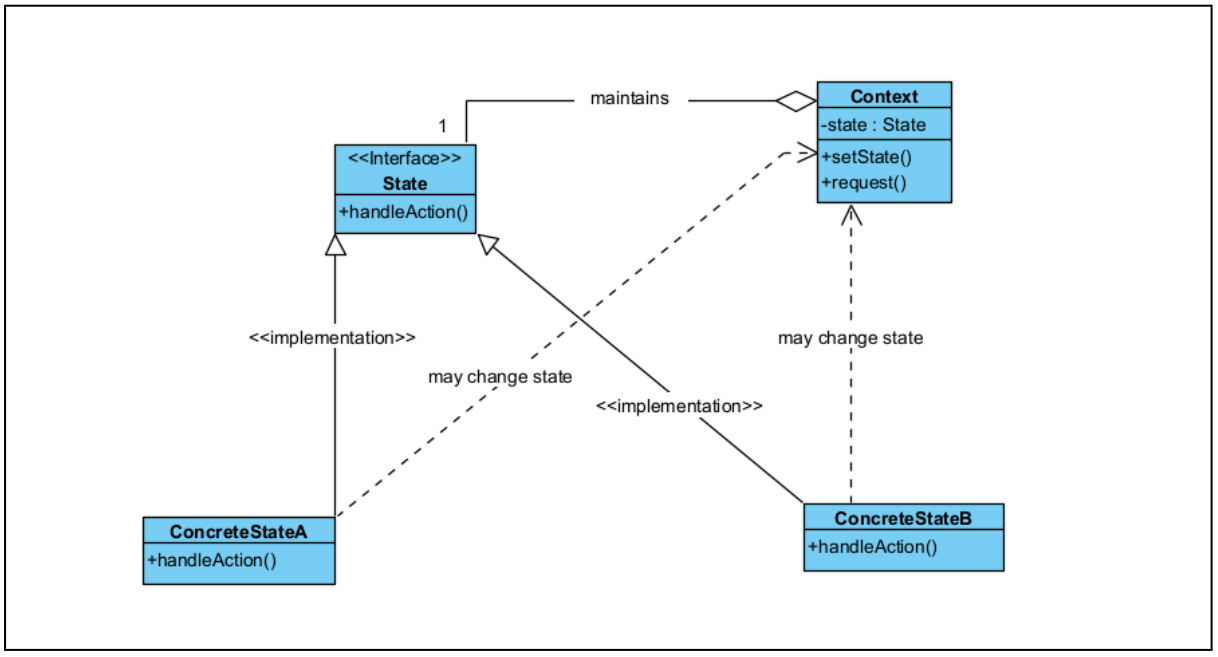
Structure (UML):



- Participating Components:
  - OrderRecord (Originator): Creates and restores saved states.
  - OrderMemento (Memento): Stores order snapshot data.
  - OrderHistory (Caretaker): Keeps mementos and triggers undo or restore.

State pattern

- Purpose: Allows an object to change its behavior when its internal state changes.
- Use case: When an object's behavior depends on its state, and it must change its behavior dynamically at runtime.
- Structure (UML):

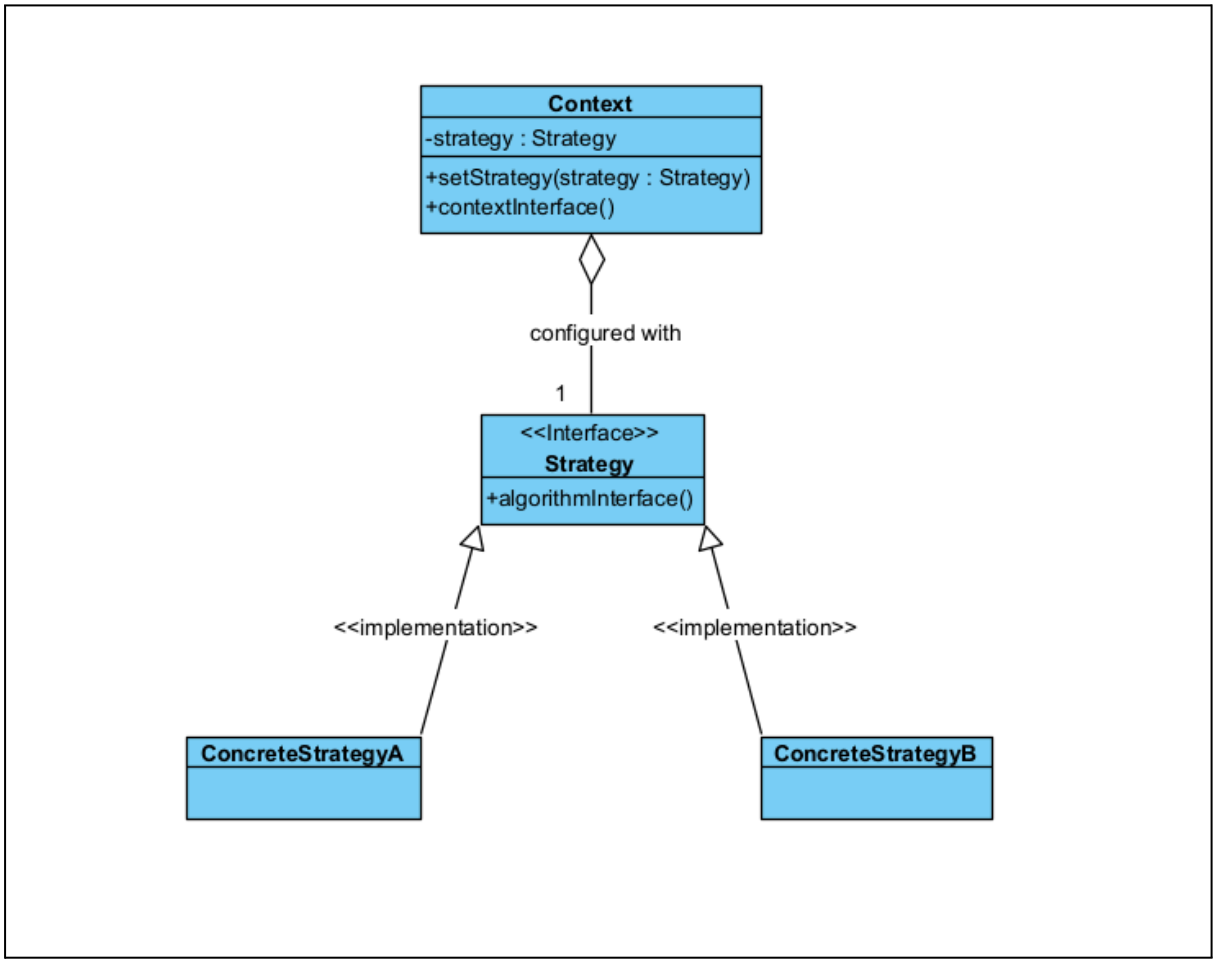


- Participating Components:

- Context (TCPConnection): Object with changing state, containing a reference to the current State object.
- State (TCPState): Interface defining common behaviors for all concrete states.
- ConcreteState (TCPEstablished, TCPListen): Concrete classes, implementing behaviors for each individual state of Context.

Strategy pattern

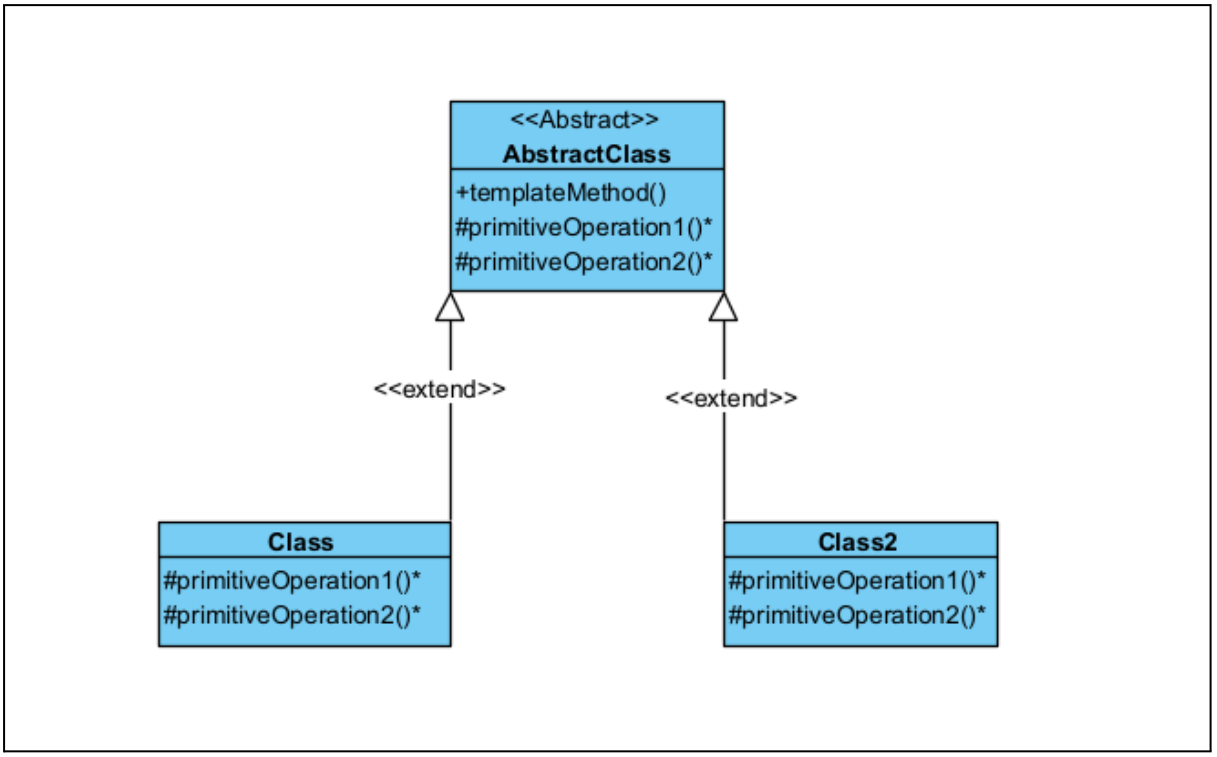
- Purpose: Define a family of algorithms, encapsulate each algorithm and make them interchangeable.
- Use case: When you have multiple algorithms/methods for a task and want the client to be able to choose which algorithm to use flexibly without modifying the client's code.
- Structure (UML):



- Participating Components:
  - Context (Composition): Object that uses an algorithm. It contains a reference to the Strategy object.
  - Strategy (Compositor): Common interface for all algorithms.
  - ConcreteStrategy (SimpleCompositor, TeXCompositor): Concrete classes that implement a specific algorithm according to the Strategy interface.

TEMPLATE METHOD Pattern

- Purpose: Define the skeleton of an algorithm in a method, so that subclasses can redefine some specific steps of that algorithm.
- Use case: When you want subclasses to be able to customize some steps of an algorithm without changing the overall structure of the algorithm.
- Structure (UML):

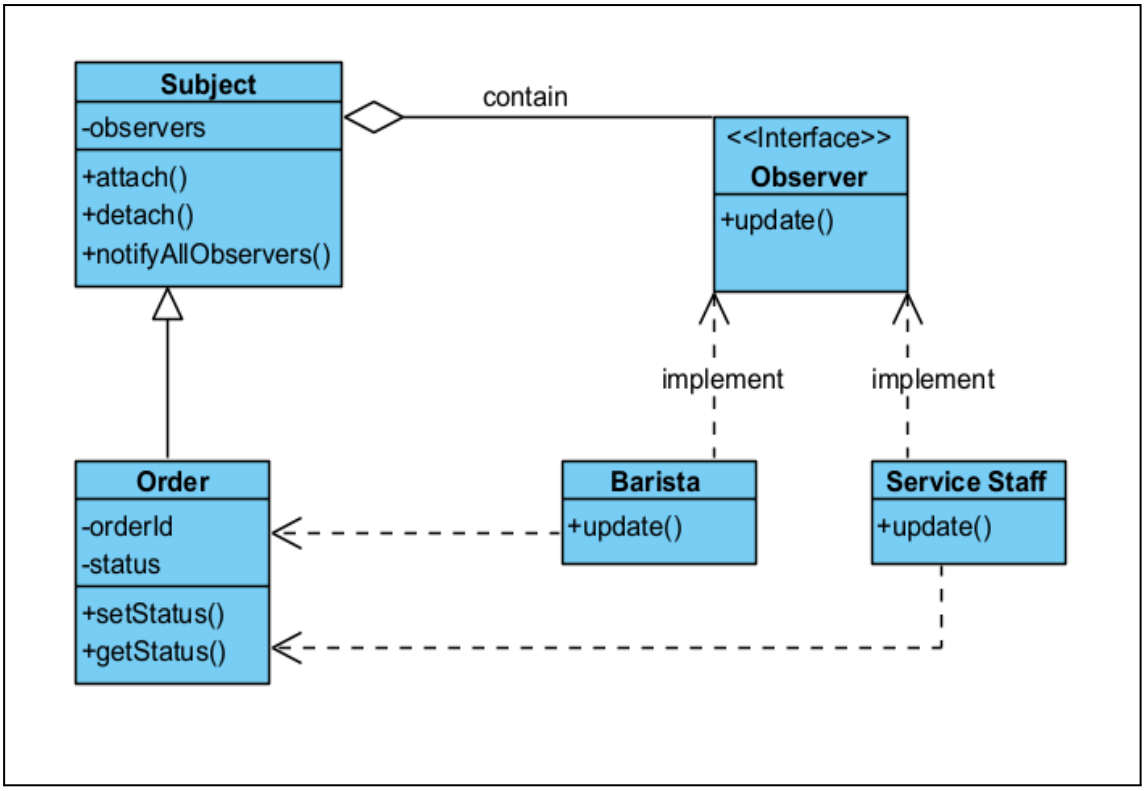


- Note: Methods marked with * or italics are abstract methods.
- Participating components:
  - AbstractClass (Application): Abstract parent class, defines "template method" (algorithm skeleton) and abstract steps that child class must implement.
  - ConcreteClass (MyApplication): Child class, inherits from AbstractClass and implements abstract steps to complete the algorithm.

II. Example for A Sample of Behavioral Design Pattern - Observer

Scenario 1:
In the coffee shop management system, When there is a new order, Both the Barista and Service Staff must be notified automatically. The Order is the Subject,The Barista and Service Staff are the Observers.

Class Diagram for Order Notification:



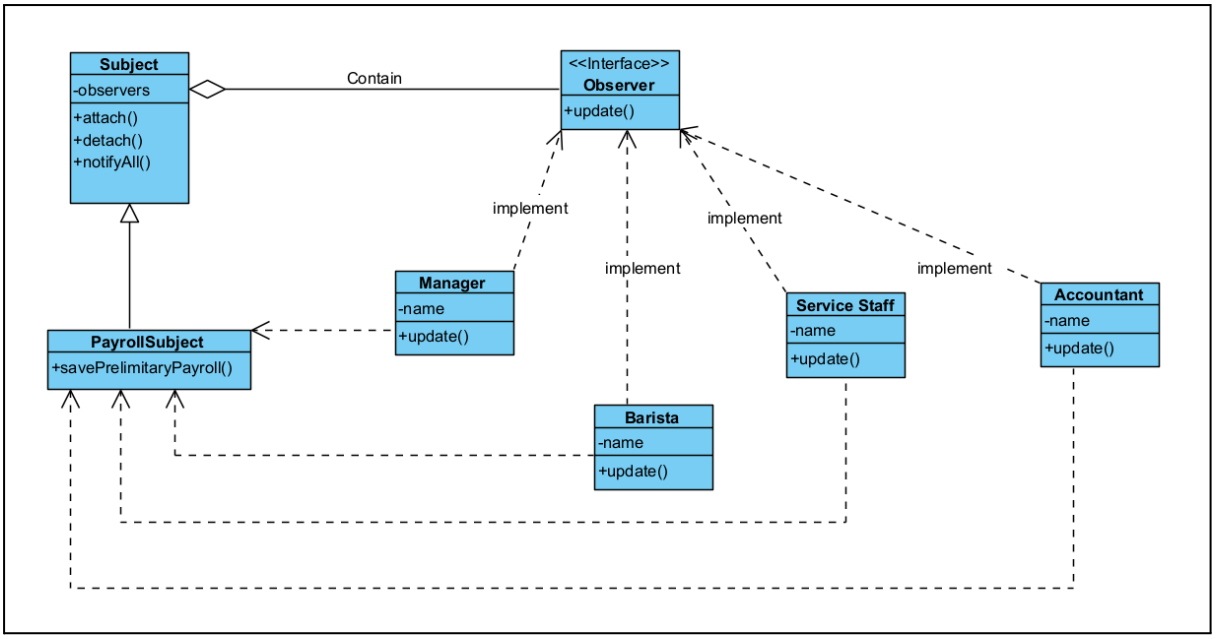
In the above diagram, Subject will manage the list of observers. This is the order notification manager in the system. Order will inherit the Subject class, it creates orders and notifies when the state changes. Observer is the Interface for the listeners, where the update() method is defined. Barista, the waiter who implements the Observer interface will depend and receive notifications when the Order updates its state.

Code demo:

```java
import java.util.ArrayList;
import java.util.List;

// Observer interface
interface Observer {
    void update(String orderStatus);
}

// Subject (Publisher)
class Subject {
    private List<Observer> observers = new ArrayList<>();

    public void attach(Observer observer) {
        observers.add(observer);
    }

    public void detach(Observer observer) {
        observers.remove(observer);
    }

    public void notifyAllObservers(String status) {
        for (Observer observer : observers) {
            observer.update(status);
        }
    }
}

// ConcreteSubject
class Order extends Subject {
    private String orderId;
    private String status;

    public Order(String orderId) {
        this.orderId = orderId;
    }

    public void setStatus(String status) {
        this.status = status;
        System.out.println("Đơn hàng " + orderId + " cập nhật trạng thái: " + status);
        notifyAllObservers(status);
    }

    public String getStatus() {
        return status;
    }
}
```

```java
// ConcreteObservers
class KitchenObserver implements Observer {
    @Override
    public void update(String orderStatus) {
        System.out.println("👨‍🍳 Bếp nhận thông báo: Đơn hàng hiện trạng là '" + orderStatus + "'")
    }
}

class WaiterObserver implements Observer {
    @Override
    public void update(String orderStatus) {
        System.out.println("🧾 Phục vụ nhận thông báo: Đơn hàng hiện trạng là '" + orderStatus +
    }
}

// Test
public class CafeObserverDemo {
    public static void main(String[] args) {
        Order order = new Order("ORD123");

        Observer kitchen = new KitchenObserver();
        Observer waiter = new WaiterObserver();

        order.attach(kitchen);
        order.attach(waiter);

        order.setStatus("Đang chuẩn bị");
        order.setStatus("Đã hoàn thành");
    }
}
```

Scenario 2: update: When the accountant creates a preliminary payroll, the payroll will be automatically notified to the Barista, Service Staff, and manager to review and respond if any.



In the above diagram, Subject will manage the list of observers. This is the PayrollSubject notification manager in the system. PayrollSubject will inherit the Subject class, it creates payroll and notifies when the accountant creates payroll. Observer is the Interface for the listeners, where the update() method is defined. Manager, Barista, Service Staff, Accountant who implements the Observer interface will depend and receive notifications when the payroll is created

Code demo:

```
public interface Observer {
    void update(String message);
}
```

```java
import java.util.ArrayList;
import java.util.List;

public class PayrollSubject {
    private List<Observer> observers = new ArrayList<>();

    public void attach(Observer observer) {
        observers.add(observer);
    }

    public void detach(Observer observer) {
        observers.remove(observer);
    }

    public void notifyAllObservers(String message) {
        for (Observer observer : observers) {
            observer.update(message);
        }
    }

    // Khi kế toán viên tạo bảng lương
    public void createPreliminaryPayroll() {
        System.out.println("📑 Kế toán đã tạo bảng lương sơ bộ!");
        notifyAllObservers("Bảng lương sơ bộ đã được tạo. Vui lòng kiểm tra và phản hồi nếu có.");
    }
}
```

```java
public class CafePayrollSystem {
    public static void main(String[] args) {
        PayrollSubject payroll = new PayrollSubject();

        Barista barista = new Barista("Lan");
        ServiceStaff staff = new ServiceStaff("Minh");
        Manager manager = new Manager("Huy");

        payroll.attach(barista);
        payroll.attach(staff);
        payroll.attach(manager);

        // Kế toán viên tạo bảng lương sơ bộ
        payroll.createPreliminaryPayroll();
    }
}
```

Command Pattern

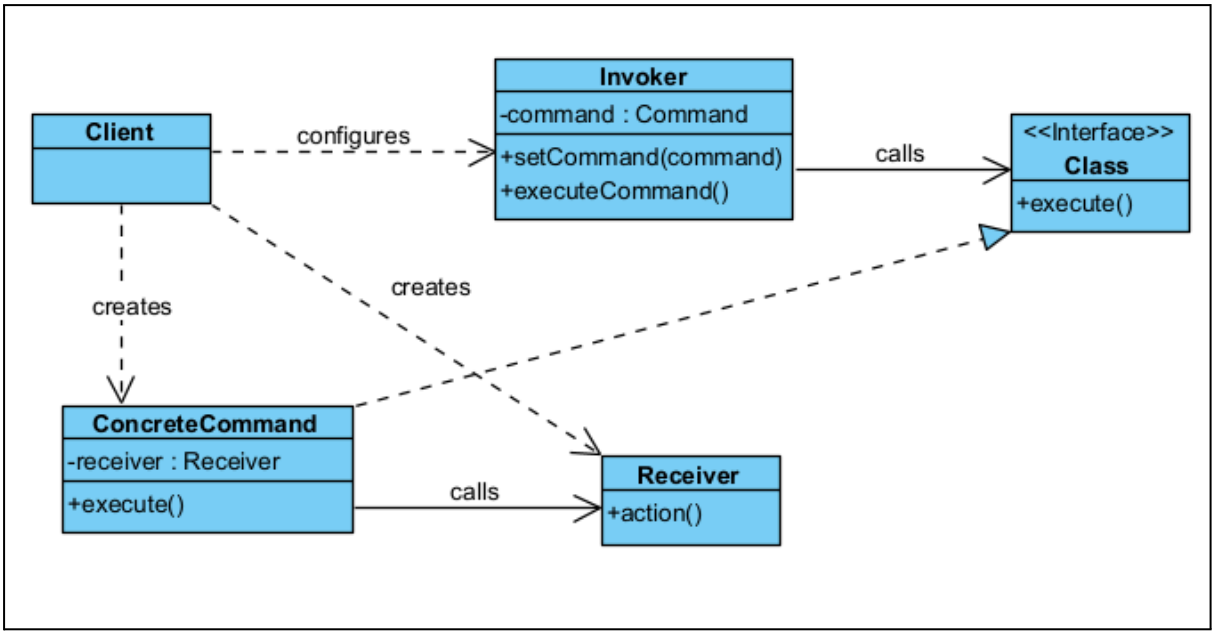


Figure 6: Generic Structure for Command Pattern

Intent: To encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

- When you want to decouple the object that issues a request from the object that performs the action.
- When you need to support operations like Undo/Redo or transactions.
- When you want to queue requests for sequential or asynchronous execution.

Participants:

- Command: Declares a common interface for executing an operation (typically the execute() method).
- ConcreteCommand: Implements the Command interface, defining a binding between a Receiver and an action.
- Client: Creates a ConcreteCommand object, sets its Receiver, and then configures the Invoker with that Command.
- Invoker: Asks the Command to carry out the request. The Invoker does not need to know the specific command; it only interacts through the Command interface.
- Receiver: The object that contains the business logic and knows how to perform the actual work.

Applicable Use Case: "Import Ingredient" :

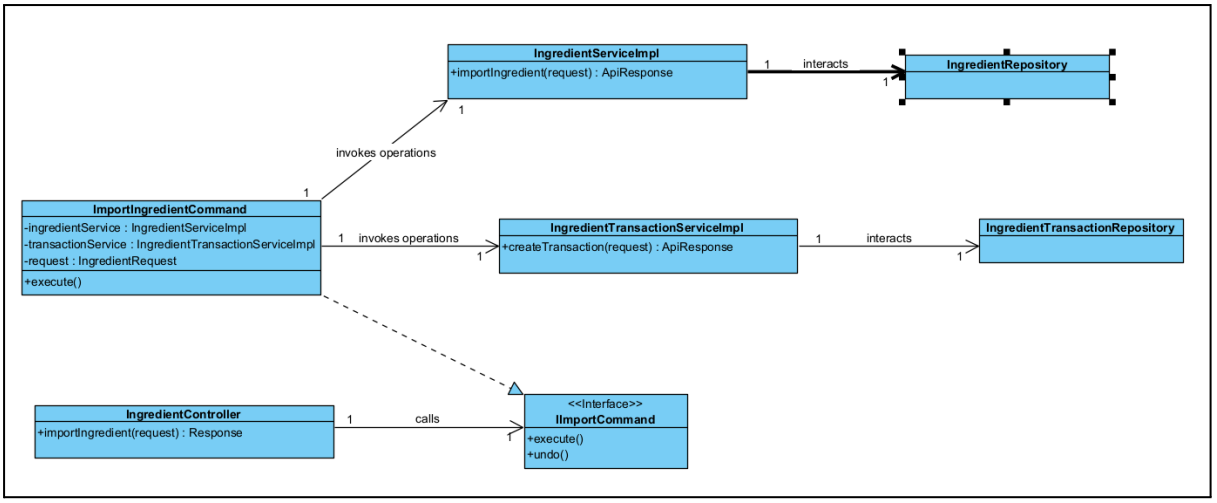


Figure Y: Command Pattern Applied to the 'Import Ingredient' Use Case

Command Pattern Role	**CSMS Class/Object (As per the Diagram)**
Invoker	IngredientController
Command (Interface)	IImportCommand
ConcreteCommand	ImportIngredientCommand
Receiver	IngredientServiceImpl, IngredientTransactionServiceImpl

1. Setup & Configuration: A Client (typically a Spring IoC container) constructs the ImportIngredientCommand object. It injects the necessary Receivers (IngredientServiceImpl and IngredientTransactionServiceImpl) into the command. This fully configured command object is then made available to the IngredientController.
2. Invocation: When an HTTP request arrives at the importIngredient endpoint, the IngredientController (the Invoker) does not know the details of the business logic. It simply executes the command by calling the execute() method on the IImportCommand interface.
3. Execution & Delegation: The execute() method within the ImportIngredientCommand (the ConcreteCommand) proceeds to orchestrate the receivers. It sequentially calls ingredientService.importIngredient() to update the stock, and then transactionService.createTransaction() to log the event. All the complex steps are hidden from the Invoker.

Visitor Pattern

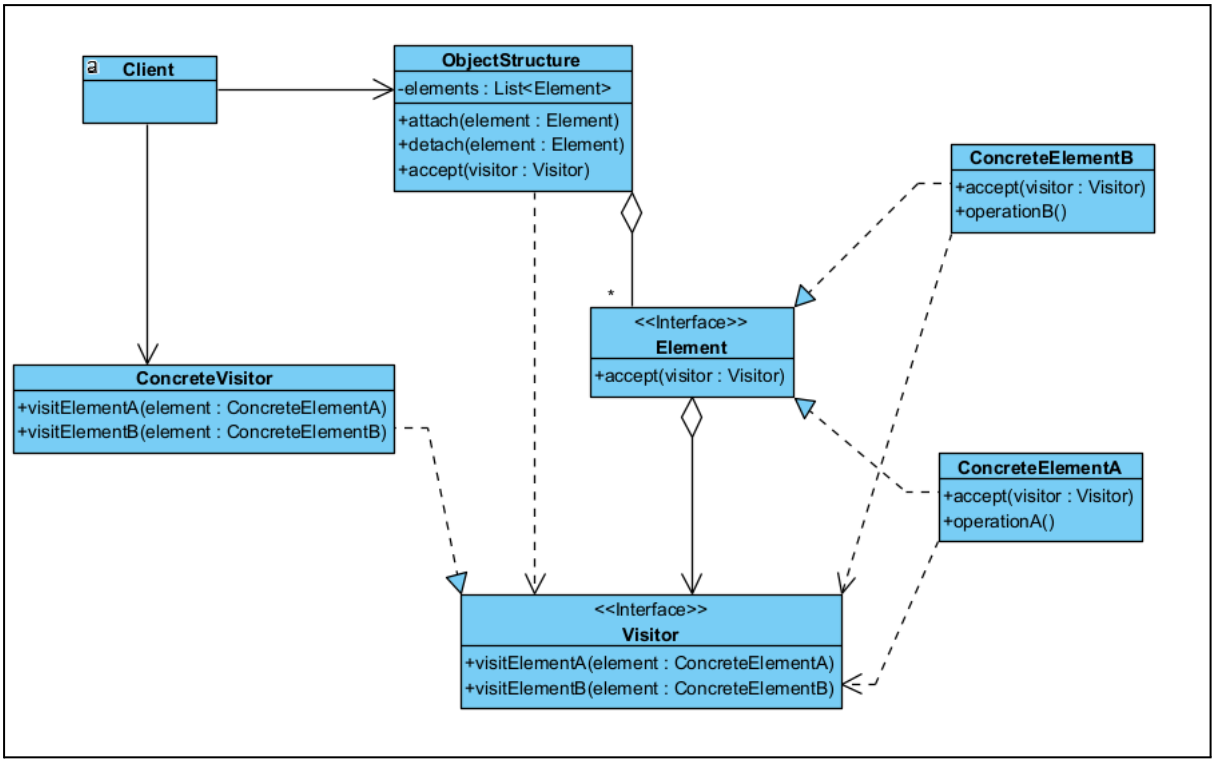


Figure A: Generic Structure of the Visitor Pattern

To represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

- When an object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes.
- When you need to perform many distinct and unrelated operations on an object structure, but you want to avoid "polluting" their classes with these operations.
- When the classes defining the object structure rarely change, but you often need to define new operations over the structure.

Participants:

- **Visitor:** An interface that declares a visit method for each class of ConcreteElement in the object structure.
- **ConcreteVisitor:** Implements the operations defined by the Visitor interface. Each method implements a fragment of the algorithm defined for the corresponding class of object in the structure.
- **Element:** An interface that declares an accept method which takes a visitor as an argument.
- **ConcreteElement:** A class that implements the Element interface. Its accept method typically calls the visitor's visit method, passing itself as the argument (visitor.visit(this)).

- **ObjectStructure:** A collection of Element objects (e.g., a list, a tree). It provides a way to iterate through its elements and allow a visitor to "visit" them.
- **Client:** Creates the ObjectStructure and configures it with Element objects. When an operation is needed, it creates a ConcreteVisitor and passes it to the ObjectStructure.

Applicable Use Case: Processing an Order with Multiple Item Types:

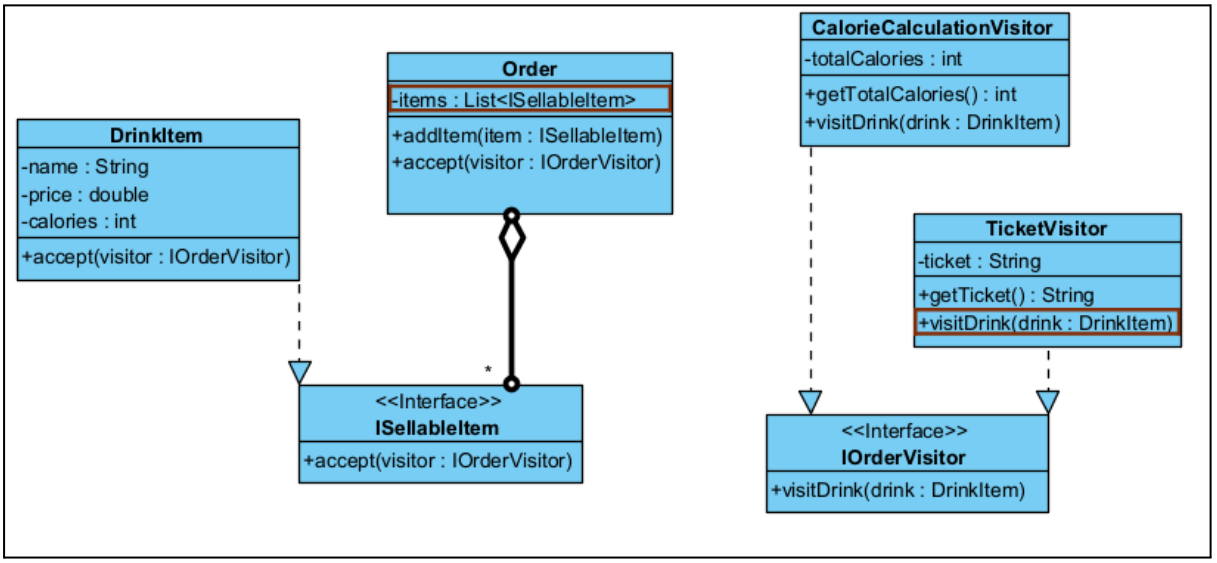


Figure B: Visitor Pattern Applied to Order Processing

Visitor Pattern Role	CSMS Class/Object
ObjectStructure	Order (contains a list of items)
Element (Interface)	ISellableItem (a new interface for all order items)
ConcreteElement	DrinkItem (classes that implement ISellableItem)
Visitor (Interface)	IOrderVisitor (a new interface for operations on orders)
ConcreteVisitor	CalorieCalculationVisitor, TicketVisitor

1. **Object Structure Creation:** An Order object is created and populated with instances of DrinkItem.
2. **Applying an Operation (e.g., Calculate Calories):**
    - An OrderProcessingService (the **Client**) creates an instance of CalorieCalculationVisitor.
    - It then calls order.accept(calorieVisitor).
    - The Order object iterates through its list of ISellableItems and calls item.accept(calorieVisitor) on each one.

3. **Double Dispatch Mechanism:**
    - When accept is called on a DrinkItem, it calls back to the visitor: visitor.visitDrink(this).
    - This "double dispatch" mechanism ensures that the correct method on the visitor is called for the correct item type.
4. **Result:** The CalorieCalculationVisitor accumulates the calories from each item. After the traversal is complete, the Client can retrieve the final result by calling calorieVisitor.getTotalCalories().

To generate a ticket, the Client would simply create a TicketVisitor and pass it to the same Order object, demonstrating how a new operation can be added without modifying any of the Order or item classes.