State Pattern

🧠 What is State Pattern?

State Pattern allows an object to alter its behavior when its internal state changes. It appears as if the object changed its class.

- Avoids large `if-else` or `switch` statements.
- Each state is encapsulated in a separate class.

State Scenario: ATM machine. Behavior changes based on state: hasCard, noCard, hasPin, outOfCash. in depth java code , jdk internel example spring boot

---

🏧 Real-life Scenario: ATM Machine

An ATM behaves differently based on its **state**:

State	Behavior
`NoCard`	Only accepts a card
`HasCard`	Allows PIN entry
`HasPin`	Allows withdrawal
`NoCash`	Displays "Out of service"

We'll model this using the **State Pattern**.

✅ Java Implementation of ATM using State Pattern

🔷 1. `ATMState` Interface (State)

```
public interface ATMState {
    void insertCard();
    void ejectCard();
    void enterPin(int pin);
    void withdrawCash(int amount);
}
```

🔷 2. Concrete States

`NoCardState`

```
public class NoCardState implements ATMState {
    private final ATMMachine atm;

    public NoCardState(ATMMachine atm) {
        this.atm = atm;
    }

    public void insertCard() {
        System.out.println("Card inserted.");
        atm.setState(atm.getHasCardState());
    }

    public void ejectCard() {
        System.out.println("No card to eject.");
    }

    public void enterPin(int pin) {
        System.out.println("Insert card first.");
    }

    public void withdrawCash(int amount) {
        System.out.println("Insert card first.");
    }
}
```

HasCardState

```
public class HasCardState implements ATMState {
    private final ATMMachine atm;

    public HasCardState(ATMMachine atm) {
        this.atm = atm;
    }

    public void insertCard() {
        System.out.println("Card already inserted.");
    }

    public void ejectCard() {
        System.out.println("Card ejected.");
        atm.setState(atm.getNoCardState());
    }

    public void enterPin(int pin) {
        if (pin == 1234) {
            System.out.println("Correct PIN.");
            atm.setState(atm.getHasPinState());
        } else {
            System.out.println("Incorrect PIN.");
        }
    }

    public void withdrawCash(int amount) {
        System.out.println("Enter PIN first.");
    }
}
```

HasPinState

```
public class HasPinState implements ATMState {
    private final ATMMachine atm;

    public HasPinState(ATMMachine atm) {
        this.atm = atm;
    }
    public void insertCard() {
        System.out.println("Card already inserted.");
    }

    public void ejectCard() {
        System.out.println("Card ejected.");
        atm.setState(atm.getNoCardState());
    }

    public void enterPin(int pin) {
        System.out.println("PIN already entered.");
    }

    public void withdrawCash(int amount) {
        if (atm.getCashInMachine() >= amount) {
            System.out.println("Dispensed ₹" + amount);
            atm.setCashInMachine(atm.getCashInMachine() - amount);
            if (atm.getCashInMachine() <= 0) {
                System.out.println("Machine is now out of cash.");
                atm.setState(atm.getNoCashState());
            }
        } else {
            System.out.println("Not enough cash.");
        }
    }
}
```

`NoCashState`

```
public class NoCashState implements ATMState {
    public NoCashState(ATMMachine atm) {}

    public void insertCard() {
        System.out.println("No cash. Cannot accept card.");
    }

    public void ejectCard() {
        System.out.println("No card inserted.");
    }

    public void enterPin(int pin) {
        System.out.println("No cash. Cannot process PIN.");
    }

    public void withdrawCash(int amount) {
        System.out.println("No cash available.");
    }
}
```

🔷 3. `ATMMachine` (Context)

```
public class ATMMachine {
    private final ATMState noCard;
    private final ATMState hasCard;
    private final ATMState hasPin;
    private final ATMState noCash;

    private ATMState currentState;
    private int cashInMachine;
    public ATMMachine(int initialCash) {
        this.noCard = new NoCardState(this);
        this.hasCard = new HasCardState(this);
        this.hasPin = new HasPinState(this);
        this.noCash = new NoCashState(this);

        this.cashInMachine = initialCash;
        this.currentState = initialCash > 0 ? noCard : noCash;
    }

    // Delegate to current state
    public void insertCard() { currentState.insertCard(); }
    public void ejectCard() { currentState.ejectCard(); }
    public void enterPin(int pin) { currentState.enterPin(pin); }
    public void withdrawCash(int amount) { currentState.withdrawCash(amount); }

    // State and cash management
    public void setState(ATMState state) { this.currentState = state; }
    public void setCashInMachine(int cash) { this.cashInMachine = cash; }

    public ATMState getNoCardState() { return noCard; }
    public ATMState getHasCardState() { return hasCard; }
    public ATMState getHasPinState() { return hasPin; }
    public ATMState getNoCashState() { return noCash; }

    public int getCashInMachine() { return cashInMachine; }
}
```

🔷 4. Client Code

```
public class ATMClient {
    public static void main(String[] args) {
        ATMMachine atm = new ATMMachine(5000);

        atm.insertCard();
        atm.enterPin(1234);
        atm.withdrawCash(2000);

        atm.withdrawCash(3000);
        atm.insertCard(); // Should show "No cash..."
    }
}
```

📌 Output

```
Card inserted.
Correct PIN.
Dispensed ₹2000
Dispensed ₹3000
Machine is now out of cash.
No cash. Cannot accept card.
```

🧪 JDK Internal Example of State Pattern

🔹 `javax.swing.text.Document`

Swing's `Document` interface uses internal state machines to manage **editable vs non-editable**, **undo/redo**, etc.

🔹 `java.nio.channels.Selector`

A `Selector` manages channels in different states: ready-to-read, ready-to-write, etc.

🌱 Spring Boot Use Case: `State` Pattern

Spring Boot doesn't use State directly, but it fits in:

✅ 1. Workflow Engines (e.g., Camunda, Activiti)

- Each state in a process (Submitted, Approved, Rejected) can be implemented using State pattern.
- Spring Boot can inject the correct `State` object based on the current status.

✅ 2. Order Management System (OMS)

Example:

```
java
CopyEdit
public interface OrderState {
    void proceed(OrderContext context);
}
```

States: `NewOrderState`, `PackedState`, `ShippedState`, `DeliveredState`

Spring can:

- Autowire these states.
- Keep state in DB.
- Inject dynamically using a map or strategy.

✅ 3. Spring State Machine Project

Spring Statemachine: A framework built specifically to implement workflows and state-driven behavior.

✅ Summary

Feature	State Pattern
Problem Solved	Change behavior based on internal state
Key Benefit	Avoid `if-else`/`switch` based logic
JDK Use	`Selector`, `Document`, Thread lifecycle
Spring Boot Use	Workflow engines, order states, tasks
Real-World Analogy	ATM, traffic signal, order processing

Would you like me to create a **Spring Boot mini-project** that simulates this ATM State Machine or Order State Tracker?

You said:

explain this example with code in depth Order Management System (OMS) Example: java Copy Edit public interface OrderState { void proceed(OrderContext context); } States: NewOrderState, PackedState, ShippedState, DeliveredState Spring can: Autowire these states. Keep state in DB. Inject dynamically using a map or strategy.

ChatGPT said:

Updated saved memory

Let's dive into the **State Design Pattern** using a realistic **Order Management System (OMS)** scenario, explaining both plain Java and a Spring Boot version that uses autowiring, dynamic injection, and persistence.

🧩 What is the State Pattern?

> **State pattern** allows an object to change its behavior when its internal state changes.
> It appears as if the object changed its class.

🛒 Real-World Scenario: Order Lifecycle in OMS

States:

- `NewOrderState` → Order placed
- `PackedState` → Order packed
- `ShippedState` → Shipped to customer
- `DeliveredState` → Delivered

Each state controls what can or cannot happen next.

✅ Plain Java Implementation

🔹 Step 1: Define `OrderState` interface

```
java
CopyEdit
public interface OrderState {
    void proceed(OrderContext context);
    String getStatus();
}
```

🔹 Step 2: Implement each state

🟢 New Order

```
java
CopyEdit
public class NewOrderState implements OrderState {
    public void proceed(OrderContext context) {
        System.out.println("Order packed.");
        context.setState(new PackedState());
    }
    public String getStatus() {
        return "New Order";
    }
}
```

📦 Packed

```
java
CopyEdit
public class PackedState implements OrderState {
    public void proceed(OrderContext context) {
        System.out.println("Order shipped.");
        context.setState(new ShippedState());
    }
    public String getStatus() {
        return "Packed";
    }
}
```

🚚 Shipped

```
java
CopyEdit
public class ShippedState implements OrderState {
    public void proceed(OrderContext context) {
        System.out.println("Order delivered.");
        context.setState(new DeliveredState());
    }
    public String getStatus() {
        return "Shipped";
    }
}
```

📬 Delivered

```
java
CopyEdit
public class DeliveredState implements OrderState {
    public void proceed(OrderContext context) {
        System.out.println("Order already delivered.");
    }
    public String getStatus() {
        return "Delivered";
    }
}
```

🔹 Step 3: OrderContext (State Manager)

java
CopyEdit

```
public class OrderContext {
    private OrderState state;

    public OrderContext() {
        this.state = new NewOrderState(); // default
    }

    public void setState(OrderState state) {
        this.state = state;
    }

    public void proceed() {
        state.proceed(this);
    }

    public String getStatus() {
        return state.getStatus();
    }
}
```

🔹 Step 4: Client

java
CopyEdit

```
public class OMSClient {
    public static void main(String[] args) {
        OrderContext order = new OrderContext();

        System.out.println("Status: " + order.getStatus());
        order.proceed();

        System.out.println("Status: " + order.getStatus());
        order.proceed();

        System.out.println("Status: " + order.getStatus());
        order.proceed();

        System.out.println("Status: " + order.getStatus());
        order.proceed();
    }
}
```

🖨️ Output

vbnet
CopyEdit

```
Status: New Order
Order packed.
Status: Packed
Order shipped.
Status: Shipped
Order delivered.
Status: Delivered
Order already delivered.
```

💡 Spring Boot Version (Dynamic Autowiring)

Let’s assume:

- States are Spring Beans.
- State transitions stored in DB.
- Spring injects the right state dynamically using `@Qualifier` or a map.

◆ 1. Define State Beans

```
java
CopyEdit
@Component("new")
public class NewOrderState implements OrderState {
    public void proceed(OrderContext context) {
        context.setState(context.getStateByKey("packed"));
    }
    public String getStatus() { return "New"; }
}

@Component("packed")
public class PackedState implements OrderState {
    public void proceed(OrderContext context) {
        context.setState(context.getStateByKey("shipped"));
    }
    public String getStatus() { return "Packed"; }
}
```

◆ 2. OrderContext with `Map<String, OrderState>`

```
java
CopyEdit
@Component
public class OrderContext {
    private OrderState currentState;

    @Autowired
    private Map<String, OrderState> stateMap;  // Inject all @Component states

    @PostConstruct
    public void init() {
        this.currentState = stateMap.get("new");  // Initial state
    }

    public void setState(OrderState state) {
        this.currentState = state;
    }
    public void proceed() {
        currentState.proceed(this);
    }

    public String getStatus() {
        return currentState.getStatus();
    }

    public OrderState getStateByKey(String key) {
        return stateMap.get(key);
    }
}
```

🗃️ Persistence of State

In a real app, you can save the order state in DB:

```
java
CopyEdit
@Entity
public class Order {
    @Id
    private Long id;

    private String state;  // "new", "packed", etc.
}
```

During retrieval:

```
java
CopyEdit
Order order = orderRepo.findById(1L);
orderContext.setState(stateMap.get(order.getState()));
```

✅ JDK Internal Example

🔹 `Thread.State` enum is a good example

Java threads go through these states: `NEW`, `RUNNABLE`, `WAITING`, `TIMED_WAITING`, `BLOCKED`, `TERMINATED`.

Each state defines **what behavior is valid** — this follows the **state pattern**.

✅ Where to Apply in Spring Boot

Use Case	Role of State Pattern
Order Processing	Change behavior as order progresses
Workflow Engines (Camunda, Flowable)	Each stage is a state
Subscription Management	Free → Trial → Paid → Suspended states
Approval Flows	Draft → Submitted → Approved → Rejected

🔚 Summary

Concept	Detail
Intent	Vary behavior of an object based on internal state
JDK Example	`Thread.State`
Spring Boot Use	Use Spring beans to model states, inject dynamically
Persistence	Store state key in DB and hydrate using Spring beans
Benefit	Cleaner transitions, no long if-else chains
