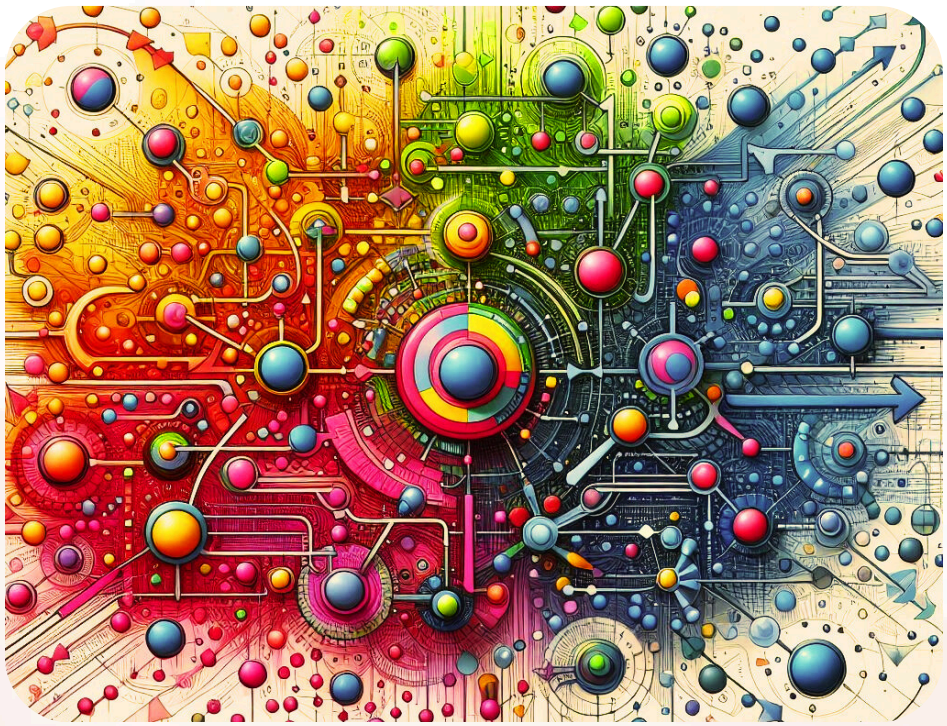
The State Design Pattern



@Ankitz007

Definition

*The **State** pattern is a behavioral design pattern that allows an object to alter its behavior when its internal state changes. The object appears to change its class by delegating behavior to state-specific classes.*

Imagine you have a TV remote 📺🕹️

When you press the power button, the TV can be in three different states:

1. *Off* ❌ → *The TV is off. If you press the power button, it turns on.*
2. *On* ✅ → *The TV is on. If you press the power button, it turns off.*
3. *Mute* 🔇 → *The TV is on but silent. If you press the mute button* 🔊*, sound comes back.*

Now let's relate this to the State pattern

- *The TV changes behavior based on what state it's in.*
- *If it's off* ❌*, pressing volume buttons* 🔼🔽 *does nothing.*
- *If it's on* ✅*, pressing volume buttons* 🔼🔽 *adjusts sound.*
- *If it's muted* 🔇*, pressing volume buttons* 🔊 *brings sound back.*
- *Instead of checking "if the TV is on, do this; if off, do that", the TV just knows what to do* 🤖 *in each state.*

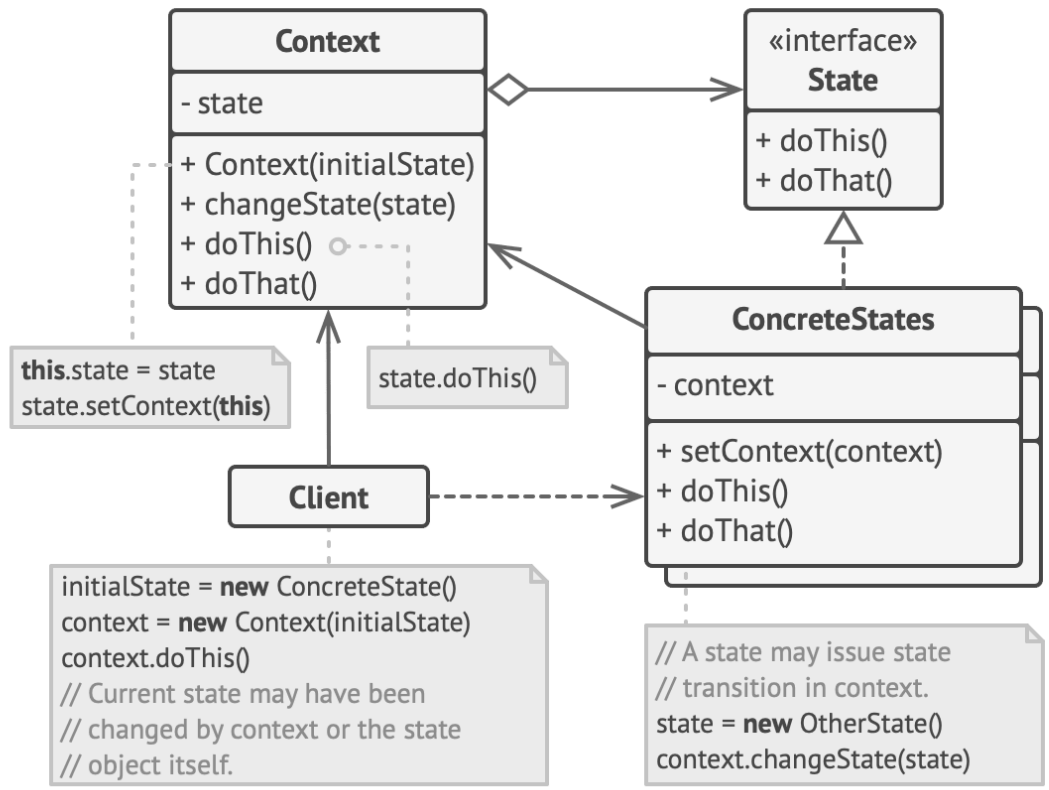
This is exactly how the State Pattern works—it makes the object act differently based on its current state, without needing a pile of "if-else" checks.

@Ankitz007

Structure

1 **Context** stores a reference to one of the concrete state objects and delegates to it all state-specific work. The context communicates with the state object via the state interface. The context exposes a setter for passing it a new state object.

2 The **State** interface declares the state-specific methods. These methods should make sense for all concrete states because you don't want some of your states to have useless methods that will never be called.



3 **Concrete States** provide their own implementations for the state-specific methods. To avoid duplication of similar code across multiple states, you may provide intermediate abstract classes that encapsulate some common behavior.

State objects may store a backreference to the context object. Through this reference, the state can fetch any required info from the context object, as well as initiate state transitions.

4 Both context and concrete states can set the next state of the context and perform the actual state transition by replacing the state object linked to the context.

Source: *Refactoring.Guru*

@Ankitz007

Key Characteristics

Encapsulates State-Specific Behavior *- Each state is represented by a separate class that encapsulates all behavior specific to that state.*

- ***Benefit:*** *Each state has its own class, making the code more organized and easier to maintain.*

State Transitions Are Handled by the State Objects *- Reduces coupling by keeping transition logic with the state that best understands when a transition should occur, rather than centralizing it in the context.*

- ***Benefit:*** *The object itself doesn't need to worry about when to switch states—the states handle transitions smoothly.*

Allows Dynamic Behavior Changes *- The pattern allows objects to change their behavior at runtime by changing their internal state object.*

- ***Benefit:*** *The object can change behavior at runtime by simply switching its current state.*

Follows the Open/Closed Principle *- New states can be added without modifying existing state classes or the context.*

- ***Benefit:*** *Client code doesn't need to change when new states or transitions are added, improving maintainability and reducing the impact of changes.*

@Ankitz007

When to use?

✅ ***Complex State-Dependent Behavior:*** *When an object's behavior changes dramatically based on its internal state.*
Use Case: *A document editor that changes editing capabilities based on whether the document is in draft, review, or published state.*

✅ ***Eliminating Large Conditionals:*** *To replace large conditional statements where each branch represents a different state.*
Use Case: *A vending machine that handles user interactions (inserting coins, selecting products, dispensing items) without complex if/else statements for each operational state.*

✅ ***State-Specific Operations:*** *When different operations need to be performed depending on the object's state.*
Use Case: *A media player that provides different controls and UI elements when in playing, paused, or stopped states.*

✅ ***Runtime State Changes:*** *When objects need to change their behavior at runtime as their state changes.*
Use Case: *A phone can switch between Normal Mode, Battery Saver Mode, etc. Depending on the state, different features are enabled or disabled dynamically.*

@Ankitz007

When NOT to use?

❌ ***Simple State Transitions/Few state-dependent behaviors***

- *When an object's behavior doesn't vary significantly based on its state or has only a few states with straightforward transitions.*

❌ ***Resource Constraints***

- *The State pattern introduces additional classes and objects, which are not ideal in environments with strict memory or performance limitations.*

❌ ***Unpredictable State Changes***

- *If state transitions are not clearly defined or follow unpredictable patterns, the benefits of the State pattern might diminish.*

❌ ***When state transitions are centralized***

- *If a single component (like a controller) makes all decisions about state transitions rather than distributing that logic, a traditional state machine implementation might be cleaner.*

❌ ***Short-Lived Objects***

- *If the objects have very short lifespans and won't have time to transition through multiple states.*

@Ankitz007

Code Example

```python
# State Interface
class OrderState:
    def proceed(self, order):
        raise NotImplementedError("Subclasses must implement this!")

# Concrete States
class ReceivedState(OrderState):
    def proceed(self, order):
        print("Order received. Now processing the order.")
        order.state = ProcessingState()
class ProcessingState(OrderState):
    def proceed(self, order):
        print("Order is being processed. Now dispatching the order.")
        order.state = DispatchedState()
class DispatchedState(OrderState):
    def proceed(self, order):
        print("Order has been dispatched. Delivery in progress.")
        # Final state; no transition here.

# Context
class Order:
    def __init__(self, state: OrderState):
        self.state = state
    def next(self):
        self.state.proceed(self)

# Demonstration:
if __name__ == '__main__':
    order = Order(ReceivedState())
    order.next()  # Transitions from Received to Processing.
    order.next()  # Transitions from Processing to Dispatched.
    order.next()  # No further transition
```

CodeImage

@Ankitz007

Real World Examples

Vending Machine

- *Interface: VendingMachineState*
- *Concrete States: IdleState, HasMoneyState, DispensingState, OutOfStockState*
- *Context: VendingMachine*

Video Game Character States

- *Interface: CharacterState*
- *Concrete States: IdleState, RunningState, AttackingState, DeadState*
- *Context: GameCharacter*

Chat Application User Presence

- *Interface: PresenceState*
- *Concrete States: OnlineState, OfflineState, AwayState, BusyState*
- *Context: UserPresence*

ATM Machine

- *Interface: ATMState*
- *Concrete States: NoCardState, HasCardState, AuthorizedState, NoCashState*
- *Context: ATMMachine*

@Ankitz007

Real World Examples

Elevator System

- *Interface: ElevatorState*
- *Concrete States: IdleState, MovingUpState, MovingDownState, DoorOpenState*
- *Context: ElevatorController*

Document Workflow Management System

- *Interface: DocumentState*
- *Concrete States: DraftState, ReviewState, PublishedState, ArchivedState*
- *Context: Document*

E-commerce Product Lifecycle

- *Interface: ProductState*
- *Concrete States: NewProductState, ActiveState, DiscontinuedState*
- *Context: Product*

Printer Job Management

- *Interface: PrintJobState*
- *Concrete States: QueuedState, PrintingState, CompletedState, ErrorState*
- *Context: PrintJob*

@Ankitz007