**Iterator**

The **Iterator Design Pattern** is a **behavioral pattern** that provides a standard way to **access elements of a collection sequentially without exposing its internal structure**.

When faced with this need, developers often write custom for loops or expose the underlying data structures (like ArrayList or LinkedList) directly.

But this approach makes the client tightly coupled to the collection’s internal structure, and it violates **encapsulation**. If the internal storage changes, the client code breaks. It also becomes difficult to add new traversal logic or support lazy iteration.

The **Iterator Pattern** solves this by abstracting the iteration logic into a dedicated object — the **iterator**. Collections provide an iterator via a method like createIterator(), and the client uses it to access elements one by one.

**1. The Problem: Traversing a Playlist**

Each playlist stores a list of songs and provides features like: Playing songs one by one , Skipping to the next or previous song , Shuffling songs and Displaying the current song queue.

class Playlist:

def \_\_init\_\_(self):

self.\_songs = [] *# Using underscore convention for "private" member*

def add\_song(self, song: str) -> None:

"""Add a song to the playlist"""

self.\_songs.append(song)

@property

def songs(self) -> list[str]:

"""Get a copy of the songs list (protects internal state)"""

return self.\_songs.copy()

def \_\_iter\_\_(self):

"""Make the playlist iterable"""

return iter(self.\_songs)

class MusicPlayer:

def play\_all(playlist: Playlist) -> None:

for song in playlist: *# Uses the playlist's iterator*

print(f"Playing: {song}")

def play\_with\_details(playlist: Playlist) -> None:

for i, song in enumerate(playlist.songs, 1):

print(f"{i}. Now playing: {song}")

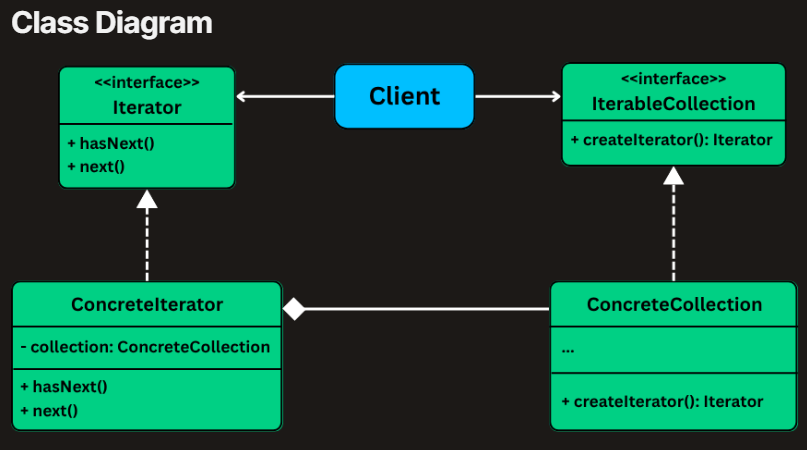
**Why This Is a Problem**

**1. Breaks Encapsulation**

By exposing the internal list of songs (getSongs()), you allow clients to directly modify the collection.

**2. What is the Iterator Pattern**

The **Iterator Pattern** provides a **standard way to traverse elements in a collection** without exposing its internal structure.



**1. Iterator (interface) :** Defines the **contract** for traversing a collection. Declares methods for traversing elements like: **hasNext()**— checks if there are more elements and **next()**— returns the next element in the sequence.

**2. ConcreteIterator :** Maintains the **current position** in the collection and iterates over it one item at a time.

**3. IterableCollection (interface) :** Defines a method for creating an iterator.

**4. ConcreteCollection :** Stores the actual data and implements the  Iterable Collection interface.

**3. Implementing Iterator**

**1. Define the Iterator Interface**

class Iterator(ABC, Generic[T]):

def has\_next(self) -> bool:

pass

def next(self) -> T:

pass

**2. Define the IterableCollection Interface**

class IterableCollection(ABC, Generic[T]):

def create\_iterator(self) -> Iterator[T]:

**3. Implement the Concrete Collection – Playlist**

class Playlist(IterableCollection[str]):

def \_\_init\_\_(self):

self.\_songs = []

def add\_song(self, song: str) -> None:

self.\_songs.append(song)

def get\_song\_at(self, index: int) -> str:

return self.\_songs[index]

def get\_size(self) -> int:

return len(self.\_songs)

def create\_iterator(self) -> Iterator[str]:

return PlaylistIterator(self)

**4. Implement the Concrete Iterator – PlaylistIterator**

class PlaylistIterator(Iterator[str]):

def \_\_init\_\_(self, playlist: Playlist):

self.\_playlist = playlist

self.\_index = 0

def has\_next(self) -> bool:

return self.\_index < self.\_playlist.get\_size()

def next(self) -> str:

if self.has\_next():

song = self.\_playlist.get\_song\_at(self.\_index)

self.\_index += 1

return song

raise StopIteration("No more songs in playlist")

**5. Client Code – Using the Iterator**

class MusicPlayer:

@staticmethod

def main() -> None:

playlist = Playlist()

playlist.add\_song("Shape of You")

playlist.add\_song("Bohemian Rhapsody")

playlist.add\_song("Blinding Lights")

iterator = playlist.create\_iterator()

print("Now Playing:")

while iterator.has\_next():

print(f" 🎵 {iterator.next()}")

**Observer**

The **Observer Design Pattern** is a **behavioral pattern** that defines a **one-to-many dependency** between objects so that when one object (the subject) changes its state, **all its dependents (observers) are automatically notified and updated**.

 For example, a **NewsPublisher** might call update() on a MobileApp, EmailService, and WebsiteFeed every time a new article is published.

**1. The Problem: Broadcasting Fitness Data**

Imagine you're developing a **Fitness Tracker App** that connects to a wearable device and receives **real-time fitness data**.

**LiveActivityDisplay**

class FitnessDataObserver(ABC):

def update(self, data: FitnessData) -> None:

pass

**ProgressLogger**

class ProgressLoggerNaive:

def log\_data\_point(self, steps:int, active\_minutes:int,calories:int)->None:

print(f"NAIVE Logger: Saving data - Steps: {steps}, "

f"Active Mins: {active\_minutes}, Calories: {calories}")

**NotificationService**

class NotificationServiceNaive:

def \_\_init\_\_(self):

self.step\_goal = 10000

self.daily\_step\_goal\_notified = False

def check\_and\_notify(self, current\_steps: int) -> None:

if current\_steps>=self.step\_goal and not self.daily\_step\_goal\_notified:

print(f"You've reached your {self.step\_goal} step goal!")

self.daily\_step\_goal\_notified = True

def reset\_daily\_notifications(self) -> None:

self.daily\_step\_goal\_notified = False

**FitnessDataNaive**

class FitnessDataNaive:

def \_\_init\_\_(self):

self.steps = 0

self.active\_minutes = 0

self.calories = 0

self.live\_display = LiveActivityDisplayNaive()

self.progress\_logger = ProgressLoggerNaive()

self.notification\_service = NotificationServiceNaive()

def new\_fitness\_data\_pushed(self, new\_steps: int, new\_active\_minutes: int, new\_calories: int) -> None:

self.steps = new\_steps

self.active\_minutes = new\_active\_minutes

self.calories = new\_calories

print(f"\nFitnessDataNaive: New data received - "

f"Steps: {self.steps}, ActiveMins: {self.active\_minutes}, Calories: {self.calories}")

*# Notify all dependent modules*

self.live\_display.show\_stats(self.steps, self.active\_minutes, self.calories)

self.progress\_logger.log\_data\_point(self.steps, self.active\_minutes, self.calories)

self.notification\_service.check\_and\_notify(self.steps)

def daily\_reset(self) -> None:

self.notification\_service.reset\_daily\_notifications()

print("FitnessDataNaive: Daily data reset.")

self.new\_fitness\_data\_pushed(0, 0, 0)

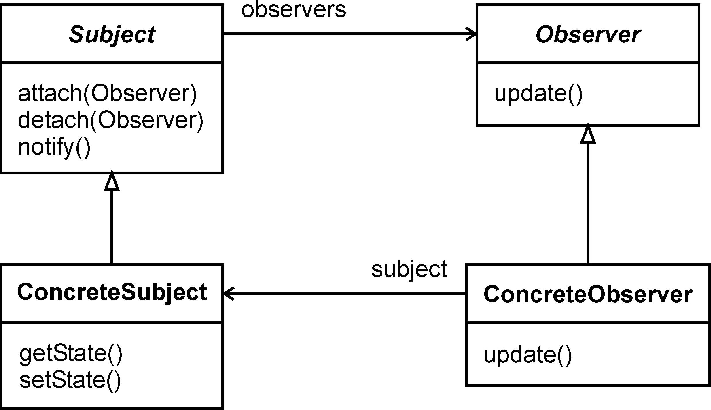
**Problems with This Approach**

**1. Tight Coupling**

The FitnessDataNaive class is tightly coupled to the specific implementations of all observers. It **must know about all dependent modules**, maintain direct references to them, and call their methods manually.

**2. The Observer Pattern**

The **Observer Design Pattern** provides broadcasting changes from one central object (the **Subject**) to many dependent objects (the **Observers**)



**1. Observer Interface (e.g., FitnessDataObserver) :** Declares an update() method**.**

**2. Subject Interface (e.g., FitnessDataSubject) :** Declares methods to : attach() detach() and notify()

**3. ConcreteSubject (e.g., FitnessData)**

**4. ConcreteObservers (e.g., LiveActivityDisplay)**

**3. Implementing Observer**

**1. Define the FitnessDataObserver Interface**

class FitnessDataObserver(ABC):

def update(self, data: 'FitnessData') -> None:

**2. Define the FitnessDataSubject Interface**

class FitnessDataSubject(ABC):

def register\_observer(self, observer: FitnessDataObserver) -> None:

def remove\_observer(self, observer: FitnessDataObserver) -> None:

def notify\_observers(self) -> None:

**3. Implement the FitnessData Class (ConcreteSubject)**

class FitnessData(FitnessDataSubject):

def \_\_init\_\_(self):

self.\_steps = 0

self.\_active\_minutes = 0

self.\_calories = 0

self.\_observers: list[FitnessDataObserver] = []

def register\_observer(self, observer: FitnessDataObserver) -> None:

self.\_observers.append(observer)

def remove\_observer(self, observer: FitnessDataObserver) -> None:

self.\_observers.remove(observer)

def notify\_observers(self) -> None:

for observer in self.\_observers:

observer.update(self)

def new\_fitness\_data\_pushed(self, steps: int, active\_minutes: int, calories: int) -> None:

self.\_steps = steps

self.\_active\_minutes = active\_minutes

self.\_calories = calories

self.notify\_observers()

def daily\_reset(self) -> None:

self.\_steps = 0

self.\_active\_minutes = 0

self.\_calories = 0

self.notify\_observers()

**4. Implement Observer Modules**

class LiveActivityDisplay(FitnessDataObserver):

def update(self, data: FitnessData) -> None:

print(f"Live Display → Steps: {data.steps} | "

f"Active Minutes: {data.active\_minutes} | "

f"Calories: {data.calories}")

class ProgressLogger(FitnessDataObserver):

def update(self, data: FitnessData) -> None:

print(f"Logger → Saving to DB: Steps={data.steps}, "

f"ActiveMinutes={data.active\_minutes}, "

f"Calories={data.calories}")

class GoalNotifier(FitnessDataObserver):

def \_\_init\_\_(self):

self.\_step\_goal = 10000

self.\_goal\_reached = False

def update(self, data: FitnessData) -> None:

if data.steps >= self.\_step\_goal and not self.\_goal\_reached:

print(f"Notifier → 🎉 Goal Reached! You've hit {self.\_step\_goal} steps!")

self.\_goal\_reached = True

def reset(self) -> None:

self.\_goal\_reached = False

**Strategy**

The **Strategy Design Pattern** is a **behavioral design pattern** that lets you define a family of algorithms, put each one into a separate class, and makes their objects interchangeable.

**1. The Problem: Shipping Cost Calculation**

Common strategies you may need to support:

* **Flat Rate**: A fixed fee (e.g., $10 per shipment), regardless of weight or destination.
* **Weight-Based**: Cost is calculated as a fixed amount per kilogram.
* **Distance-Based**: Different rates depending on destination zones (e.g., Zone A = $5, Zone B = $12).
* **Third-Party API**: Fetch dynamic rates from providers like FedEx or UPS.

A quick and naive solution might be to implement all of this logic inside a single class, using a long chain of conditionals

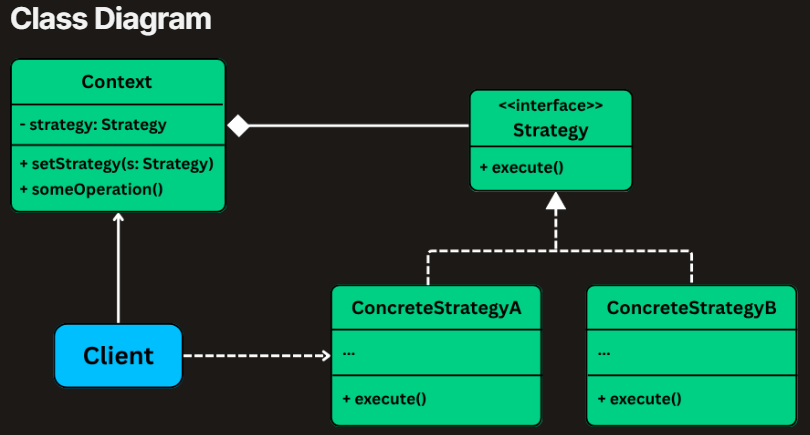
**2. The Strategy Design Pattern**

The Strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable.

**Strategy Interface (e.g., ShippingStrategy)**

**Concrete Strategies (e.g., FlatRateShipping, WeightBasedShipping)**

**Context Class (e.g., ShippingCostService)**



**3. Implementing Strategy Pattern**

**1. Define the Strategy Interface (ShippingStrategy)**

class Order:

total\_weight: float = 0.0

destination\_zone: str = "ZoneA"

order\_value: float = 0.0

class ShippingStrategy(ABC):

@abstractmethod

def calculate\_cost(self, order: Order) -> float:

pass

**2. Implement Concrete Strategy Classes**

class FlatRateShipping(ShippingStrategy):

def \_\_init\_\_(self, rate: float):

self.rate = rate

def calculate\_cost(self, order: Order) -> float:

print(f"Calculating with Flat Rate strategy (${self.rate})")

return self.rate

class WeightBasedShipping(ShippingStrategy):

def \_\_init\_\_(self, rate\_per\_kg: float):

self.rate\_per\_kg = rate\_per\_kg

def calculate\_cost(self, order: Order) -> float:

print(f"Calculating with Weight-Based strategy (${self.rate\_per\_kg}/kg)")

return order.total\_weight \* self.rate\_per\_kg

class DistanceBasedShipping(ShippingStrategy):

def \_\_init\_\_(self, rate\_per\_km: float):

self.rate\_per\_km = rate\_per\_km

def calculate\_cost(self, order: Order) -> float:

print(f"Calculating with Distance-Based strategy for zone: {order.destination\_zone}")

zone\_multipliers = {

"ZoneA": 5.0,

"ZoneB": 7.0

}

return self.rate\_per\_km \* zone\_multipliers.get(order.destination\_zone, 10.0)

**3. Create the Context Class (ShippingCostService)**

class ShippingCostService:

def \_\_init\_\_(self, strategy: ShippingStrategy):

self.\_strategy = strategy

def strategy(self) -> ShippingStrategy:

return self.\_strategy

@strategy.setter

def strategy(self, strategy: ShippingStrategy) -> None:

print(f"ShippingCostService: Strategy changed to {strategy.\_\_class\_\_.\_\_name\_\_}")

self.\_strategy = strategy

def calculate\_shipping\_cost(self, order: Order) -> float:

if not self.\_strategy:

raise ValueError("Shipping strategy not set")

cost = self.\_strategy.calculate\_cost(order)

print(f"ShippingCostService: Final Calculated Shipping Cost: ${cost} "

f"(using {self.\_strategy.\_\_class\_\_.\_\_name\_\_})")

return cost

**4. Client Code**

class ECommerceAppV2:

flat\_rate = FlatRateShipping(10.0)

weight\_based = WeightBasedShipping(2.5)

distance\_based = DistanceBasedShipping(5.0)

third\_party = ThirdPartyApiShipping(7.5, 0.02)

*# Create service with initial strategy*

shipping\_service = ShippingCostService(flat\_rate)

print("--- Order 1: Using Flat Rate (initial) ---")

shipping\_service.calculate\_shipping\_cost(order1)

**Command**

The **Command Design Pattern** is a **behavioral pattern** that turns a request into a **standalone object**, allowing you to **parameterize actions**, queue them, log them, or support undoable operations.

**1. The Problem: The Tightly Coupled Smart Home Controller**

**Naive Implementation: One Controller to Rule Them All**

Class Light: void on() void off()

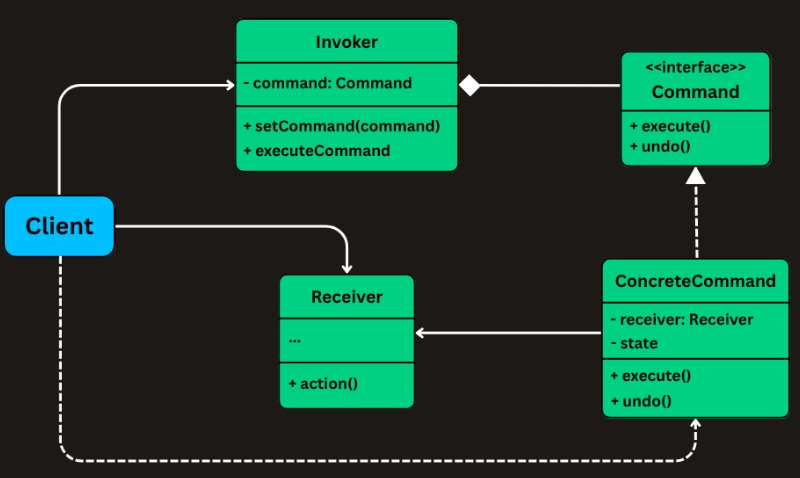
Class Thermostat : void settemp()

**Why This Design Fails as the System Grows**

**1. Tight Coupling :** You can’t reuse or generalize actions — every new device requires a new method in the controller.

**2. No Scheduling or Queuing**

**2. What is the Command Pattern**



**1. Command (Interface):** execute() method | undo() method

**2. ConcreteCommand:** Maintains a reference to the Receiver

**3. Receiver:** object that performs the actual work.

**4. Invoker:** Responsible for initiating command execution.

**5. Client**

**3. Implementing Command Pattern**

**1. Define the Command Interface**

class Command(ABC):

def execute(self) -> None:

pass

def undo(self) -> None:

pass

**2. Define the Receivers (Devices)**

class Light:

def on(self) -> None:

print("Light turned ON")

def off(self) -> None:

print("Light turned OFF")

class Thermostat:

def \_\_init\_\_(self):

self.\_current\_temperature = 20 *# default*

def set\_temperature(self, temp: int) -> None:

print(f"Thermostat set to {temp}°C")

self.\_current\_temperature = temp

@property

def current\_temperature(self) -> int:

return self.\_current\_temperature

**3. Implement Concrete Command Classes**

class LightOnCommand(Command):

def \_\_init\_\_(self, light: Light):

self.\_light = light

def execute(self) -> None:

self.\_light.on()

def undo(self) -> None:

self.\_light.off()

class LightOffCommand(Command):

def \_\_init\_\_(self, light: Light):

self.\_light = light

def execute(self) -> None:

self.\_light.off()

def undo(self) -> None:

self.\_light.on()

class SetTemperatureCommand(Command):

def \_\_init\_\_(self, thermostat: Thermostat, temperature: int):

self.\_thermostat = thermostat

self.\_new\_temperature = temperature

self.\_previous\_temperature: int = 0

def execute(self) -> None:

self.\_previous\_temperature = self.\_thermostat.current\_temperature

self.\_thermostat.set\_temperature(self.\_new\_temperature)

def undo(self) -> None:

self.\_thermostat.set\_temperature(self.\_previous\_temperature)

**4. Create the Invoker (SmartButton) with Undo Support**

class SmartButton:

def \_\_init\_\_(self):

self.\_current\_command: Command | None = None

self.\_history: list[Command] = []

def set\_command(self, command: Command) -> None:

self.\_current\_command = command

def press(self) -> None:

if self.\_current\_command:

self.\_current\_command.execute()

self.\_history.append(self.\_current\_command)

else:

print("No command assigned.")

def undo\_last(self) -> None:

if self.\_history:

last\_command = self.\_history.pop()

last\_command.undo()

else:

print("Nothing to undo.")

**5. Client Code – Using the Command System**

class SmartHomeApp:

@staticmethod

def main() -> None:

light = Light()

thermostat = Thermostat()

light\_on = LightOnCommand(light)

light\_off = LightOffCommand(light)

set\_temp\_22 = SetTemperatureCommand(thermostat, 22)

button = SmartButton()

print("→ Pressing Light ON")

button.set\_command(light\_on)

button.press()

print("→ Pressing Set Temp to 22°C")

button.set\_command(set\_temp\_22)

button.press()

print("→ Pressing Light OFF")

button.set\_command(light\_off)

button.press()

*# Undo sequence*

print("\n↶ Undo Last Action")

button.undo\_last() *# undo Light OFF*

print("↶ Undo Previous Action")

button.undo\_last() *# undo Set Temp*

print("↶ Undo Again")

button.undo\_last() *# undo Light ON*

**State**

The **State Design Pattern** is a **behavioral design pattern** that lets an object **change its behavior when its internal state changes**.

Imagine you're building a simple **vending machine system**.

At any given time, the vending machine can only be in **one state**, such as:

* **IdleState** – Waiting for user input (nothing selected, no money inserted).
* **ItemSelectedState** – An item has been selected, waiting for payment.
* **HasMoneyState** – Money has been inserted, waiting to dispense the selected item.
* **DispensingState** – The machine is actively dispensing the item.

The machine supports a few user-facing operations:

* **selectItem(String itemCode)** – Select an item to purchase
* **insertCoin(double amount)** – Insert payment for the selected item
* **dispenseItem()** – Trigger the item dispensing process

A common but flawed approach is to manage state transitions manually inside a monolithic VendingMachine class using if-else or switch statements:

class VendingMachine:

class State(Enum):

def select\_item(self, item\_code: str) -> None:

match self.current\_state:

case self.State.IDLE:

self.selected\_item = item\_code

self.current\_state = self.State.ITEM\_SELECTED

print(f"Selected item: {item\_code}")

case self.State.ITEM\_SELECTED:

print("Item already selected")

case self.State.HAS\_MONEY:

print("Payment already received for item")

case self.State.DISPENSING:

print("Currently dispensing")

def insert\_coin

“””

def dispense\_item

“””

def \_reset\_machine

”””

**What's Wrong with This Approach?**

While using an enum with switch statements can work for small, predictable systems, this approach **doesn't scale well**.

**1. Cluttered Code**

All state-related logic is stuffed into a single class

**2. Hard to Extend**

Suppose you want to introduce new states like:

* OutOfStockState – when the selected item is sold out
* MaintenanceState – when the machine is undergoing service

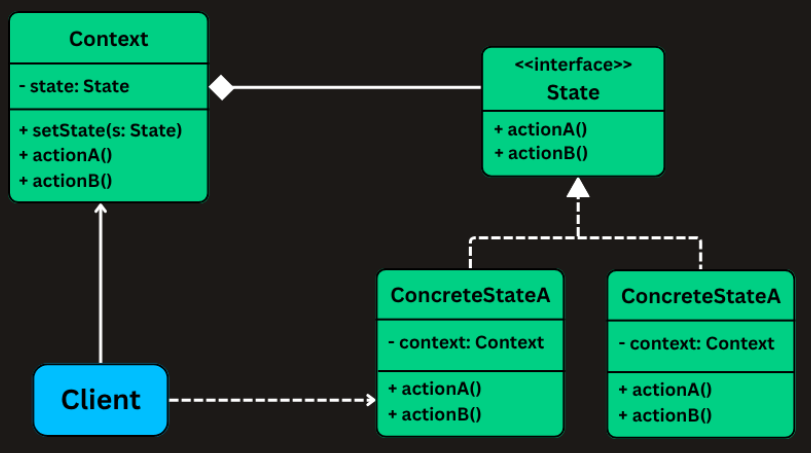
**The State Pattern**

The State pattern allows an object (the Context) to alter its behavior when its internal state changes.

**1. State Interface (e.g., MachineState)**

**2. Concrete States (e.g., IdleState, ItemSelectedState)**

**3. Context (e.g., VendingMachine)**



**1. Define the State Interface**

class MachineState(ABC):

def select\_item(self, context: 'VendingMachine', item\_code: str) -> None:

def insert\_coin(self, context: 'VendingMachine', amount: float) -> None:

def dispense\_item(self, context: 'VendingMachine') -> None:

**2. Implement Concrete State Classes**

class IdleState(MachineState):

def select\_item(self, context: 'VendingMachine', item\_code: str) -> None:

def insert\_coin(self, context: 'VendingMachine', amount: float) -> None:

def dispense\_item(self, context: 'VendingMachine') -> None:

class ItemSelectedState(MachineState):

def select\_item(self, context: 'VendingMachine', item\_code: str) -> None:

def insert\_coin(self, context: 'VendingMachine', amount: float) -> None:

def dispense\_item(self, context: 'VendingMachine') -> None:

class HasMoneyState(MachineState):

def select\_item(self, context: 'VendingMachine', item\_code: str) -> None:

def insert\_coin(self, context: 'VendingMachine', amount: float) -> None:

def dispense\_item(self, context: 'VendingMachine') -> None:

class DispensingState(MachineState):

def select\_item(self, context: 'VendingMachine', item\_code: str) -> None:

def insert\_coin(self, context: 'VendingMachine', amount: float) -> None:

def dispense\_item(self, context: 'VendingMachine') -> None:

**3. Implement the Context (VendingMachine)**

class VendingMachine:

def \_\_init\_\_(self):

self.state: MachineState = IdleState()

self.selected\_item: str = ""

self.inserted\_amount: float = 0.0

def select\_item(self, item\_code: str) -> None:

self.state.select\_item(self, item\_code)

def insert\_coin(self, amount: float) -> None:

self.state.insert\_coin(self, amount)

def dispense\_item(self) -> None:

self.state.dispense\_item(self)

def reset(self) -> None:

self.selected\_item = ""

self.inserted\_amount = 0.0

self.state = IdleState()

print("Machine has been reset to idle state.")