```
        self.names = [get_or_add(part) for part in
full_name.split(" ")]
```

Step 2: Create Helper Functions

We need random name generators to simulate large datasets.

```
import random
import string

# Generate random integer
def get_random_integer(max_value: int) -> int:
    return random.randint(0, max_value - 1)

# Generate random text string (10 characters)
def get_random_text() -> str:
    return ''.join(random.choice(string.ascii_uppercase)
for _ in range(10))
```

Step 3: Generate Random Names

We'll create **100 first names** and **100 last names**.

```
names = []
last_names = []

for _ in range(100):
    names.append(get_random_text())
    last_names.append(get_random_text())
```

Step 4: Create Users

We'll create **10,000 users** both with and without the Flyweight optimization.

```
users = []   # Without Flyweight
users2 = []  # With Flyweight

for name in names:
    for last_name in last_names:
        users.append(User(f"{name} {last_name}"))
```

```
        users2.append(User2(f"{name} {last_name}"))
```

Step 5: Compare Memory Usage

We'll compare sizes by serializing them into JSON-like strings.

```
import json

# Approximate memory usage without Flyweight
users_size = len(json.dumps([u.__dict__ for u in users]))
print(f"10,000 users occupy approximately {users_size}
characters")

# Approximate memory usage with Flyweight
users2_size = len(json.dumps([u.__dict__ for u in users2]))
strings_size = len(json.dumps(User2.strings))
total_size = users2_size + strings_size

print(f"10,000 users with Flyweight occupy ~{total_size}
characters")
```

Final Code:

```
import random
import string
import json


# Regular User class (no optimization)
class User:
    def __init__(self, full_name: str):
        self.full_name = full_name


# Optimized User class using Flyweight
class User2:
    strings = []  # shared pool of strings

    def __init__(self, full_name: str):
        def get_or_add(s: str) -> int:
            if s in User2.strings:
                return User2.strings.index(s)
            else:
                User2.strings.append(s)
                return len(User2.strings) - 1

        # Store references (indices) instead of full string
        self.names = [get_or_add(part) for part in
full_name.split(" ")]


# Helper functions
def get_random_integer(max_value: int) -> int:
    return random.randint(0, max_value - 1)


def get_random_text() -> str:
    return ''.join(random.choice(string.ascii_uppercase)
for _ in range(10))
```

```
# Step 1: Generate random names
names = []
last_names = []
for _ in range(100):
    names.append(get_random_text())
    last_names.append(get_random_text())

# Step 2: Create users (10,000 total)
users = []   # Without Flyweight
users2 = []  # With Flyweight

for name in names:
    for last_name in last_names:
        users.append(User(f"{name} {last_name}"))
        users2.append(User2(f"{name} {last_name}"))

# Step 3: Compare memory usage
users_size = len(json.dumps([u.__dict__ for u in users]))
print(f"10,000 users occupy approximately {users_size}
characters")

users2_size = len(json.dumps([u.__dict__ for u in users2]))
strings_size = len(json.dumps(User2.strings))
total_size = users2_size + strings_size
print(f"10,000 users with Flyweight occupy ~{total_size}
characters")

# 10,000 users occupy approximately 400000 characters
# 10,000 users with Flyweight occupy ~221800 characters
```

Proxy

The Proxy pattern allows one class to represent the functionality of another class, acting as an intermediary or "proxy" between the client and the real object. This can be useful for controlling access to the real object, performing additional operations before or after accessing the real object, or even replacing the real object entirely.

This is useful when:

- You want to **control access** to an object (e.g., authentication, permissions).
- You want to **add behavior** before or after interacting with the object (e.g., logging, caching).
- You want to **delay the creation** of expensive objects until they're needed (lazy initialization).
- You want to **represent a simplified version** of a complex object.

Example: Percentage Proxy in Python

We'll implement a simple proxy object called **`Percentage`** that represents percentage values.

It allows:

1. Displaying the value as `"5%"`.
2. Using it in mathematical expressions naturally, since it behaves like a number.

Step 1: Define the Proxy Class

```
class Percentage:
    def __init__(self, percentage: float):
        # Store the percentage as a whole number (e.g., 5
for 5%)
        self.percentage = percentage

    def __str__(self) -> str:
        """
        Convert the percentage into a string with a '%'
sign.
```

```
        Example: Percentage(5) -> "5%"
        """
        return f"{self.percentage}%"

    def __float__(self) -> float:
        """
        Convert the percentage into a usable numeric value.
        Example: Percentage(5) -> 0.05
        """
        return self.percentage / 100

    def __mul__(self, other):
        """
        Allow multiplication directly with numbers.
        Example: 50 * Percentage(5) -> 2.5
        """
        return float(self) * other

    def __rmul__(self, other):
        """
        Support multiplication in reverse order.
        Example: Percentage(5) * 50 -> 2.5
        """
        return self.__mul__(other)
```

Step 2: Use the Proxy Object

```
# Create a proxy for 5%
percentage_five = Percentage(5)

# See how it looks as text
print(str(percentage_five))
# Output: "5%"

# Use it in a math expression
result = 50 * percentage_five
print(f"5% of 50 is {result}")
# Output: "5% of 50 is 2.5"
```

Final Code:

```
class Percentage:
    def __init__(self, percentage: float):
        self.percentage = percentage

    def __str__(self) -> str:
        return f"{self.percentage}%"

    def __float__(self) -> float:
        return self.percentage / 100

    def __mul__(self, other):
        return float(self) * other

    def __rmul__(self, other):
        return self.__mul__(other)


# Example usage
percentage_five = Percentage(5)
print(str(percentage_five))  # "5%"

result = 50 * percentage_five
print(f"5% of 50 is {result}")  # "5% of 50 is 2.5"
```

Here, the `Percentage` class acts as a proxy:

- To the outside world, it looks like a percentage ("5%").
- Internally, it behaves like a number (0.05) in calculations.
- It adds **clarity and readability** without changing how math works.

Behavioural Design Patterns

Behavioral design patterns focus on the communication and interaction between objects in a software system. These patterns identify common patterns of communication and collaboration between objects, increasing flexibility and efficiency in designing and implementing the behavior of a system.

Below are the most common behavioral design patterns in Python:

Chain of Responsibility: Allows multiple objects to handle a request without the sender knowing which one will process it.

Command: Encapsulates a request as an object, allowing clients to parameterize objects with operations, queue requests, log requests, and support undoable operations.

Iterator: Provides a way to access elements of a collection sequentially without exposing its underlying representation.

Mediator: Defines an object that encapsulates how communication occurs between objects in a system, promoting loose coupling by preventing objects from explicitly referring to each other.

Memento: Allows capturing and externalizing an object's internal state so that the object can be restored to that state later.

Observer: Defines a one-to-many dependency between objects so that when one object changes its state, all its dependents are notified and updated automatically.

Visitor: Represents an operation to be performed on the elements of an object structure, allowing defining a new operation without changing the classes of the elements on which it operates.

Strategy: Defines a family of algorithms, encapsulates each one, and makes them interchangeable, allowing the algorithm to vary independently of clients that use it.

State: Allows an object to alter its behavior when its internal state changes, often implemented as a finite state machine.

Template Method: Defines the skeleton of an algorithm in an operation but allows subclasses to redefine parts of the algorithm without changing its structure.

Chain of Responsability

Create a chain of objects. Starting from a point, it stops until a certain condition is met.

In object-oriented design, the Chain of Responsibility pattern is a design pattern consisting of a source of command objects and a series of processing objects.

This is useful when:

- You want to **avoid coupling** the sender of a request to its receiver.
- You want **more than one object to handle a request**, but without knowing who.
- You want to **process things step by step**, where each handler can modify data or stop the chain.
- You want to **implement game mechanics, event pipelines, or middleware-like structures**.

Example: A Game Creature

We'll create a **creature** with attack and defense points. Then we'll apply a **chain of modifiers** to change its stats.

- Some modifiers increase stats.
- Some prevent bonuses at all.
- The order of the chain matters.

Step 1: Define Core Classes

```
class Creature:
    def __init__(self, name: str, attack: int, defense:
int):
        self.name = name
        self.attack = attack
        self.defense = defense

    def __str__(self):
        return f"{self.name} ({self.attack} /
{self.defense})"
```

```
class CreatureModifier:
    """
    Base class for modifiers in the chain.
    Each modifier knows the creature it modifies
    and the next modifier in the chain.
    """
    def __init__(self, creature: Creature):
        self.creature = creature
        self.next_modifier = None

    def add(self, modifier):
        """
        Add a modifier at the end of the chain.
        """
        if self.next_modifier:
            self.next_modifier.add(modifier)
        else:
            self.next_modifier = modifier

    def handle(self):
        """
        By default, simply forward to the next modifier.
        """
        if self.next_modifier:
            self.next_modifier.handle()
```

Step 2: Define Specific Modifiers

```
class NoBonusesModifier(CreatureModifier):
    """
    This modifier blocks the chain completely.
    No other modifiers will be applied.
    """
    def handle(self):
        print("No bonuses for you!")
        # Notice: we do NOT call super().handle()
        # This stops the chain here.
```

```
class DoubleAttackModifier(CreatureModifier):
    """
    This modifier doubles the creature's attack.
    """
    def handle(self):
        print(f"Doubling attack of {self.creature.name}")
        self.creature.attack *= 2
        super().handle()  # continue the chain


class IncreaseDefenseModifier(CreatureModifier):
    """
    This modifier increases defense,
    but only if the attack is small (<= 2).
    """
    def handle(self):
        if self.creature.attack <= 2:
            print(f"Increasing defense of
{self.creature.name}")
            self.creature.defense += 1
        super().handle()
```

Step 3: Use the Chain of Responsibility

```
if __name__ == "__main__":
    # Create a creature
    pikachu = Creature("Pikachu", 1, 1)
    print(pikachu)  # Pikachu (1 / 1)

    # Create a root of the chain
    root = CreatureModifier(pikachu)

    # Add modifiers in order
    root.add(NoBonusesModifier(pikachu))       # stops the
chain
```

```
    root.add(DoubleAttackModifier(pikachu))    # won't
apply
    root.add(IncreaseDefenseModifier(pikachu)) # won't
apply

    # Apply the chain
    root.handle()

    # Show final state
    print(pikachu)  # Pikachu (1 / 1) - unchanged because
NoBonuses stopped the chain
```

Alternate Scenario: Without `NoBonusesModifier`

```
if __name__ == "__main__":
    charmander = Creature("Charmander", 1, 1)
    print(charmander)  # Charmander (1 / 1)

    root = CreatureModifier(charmander)

    root.add(DoubleAttackModifier(charmander))    # doubles
attack -> 2
    root.add(IncreaseDefenseModifier(charmander)) # attack
<= 2, so defense +1

    root.handle()
    print(charmander)  # Charmander (2 / 2)
```

With this setup:

- You can dynamically **add, remove, and reorder modifiers**.
- You can **stop the chain** with NoBonusesModifier.
- This mirrors **middleware pipelines** and **game buff/debuff systems**.

Command

The Command pattern is a behavioral design pattern in which an object is used to encapsulate all the information needed to perform an action or trigger an event at a later time. This information includes the method's name, the object that owns the method, and the method's parameter values.

This is useful when:

- You need to **undo/redo** operations.
- You want to **queue or log** actions.
- You want to decouple an object that issues requests from the object that performs them.
- You need to **implement transactions** or **macro commands**.

Example: Bank Account Operations

We'll model a **BankAccount** class and then encapsulate its operations (`deposit` and `withdraw`) into **command objects**.

Step 1: Define the BankAccount Class

```
class BankAccount:
    overdraft_limit = -500  # static overdraft limit

    def __init__(self, balance=0):
        self.balance = balance

    def deposit(self, amount: int):
        self.balance += amount
        print(f"{amount} deposited. New balance:
{self.balance}")

    def withdraw(self, amount: int):
        if self.balance - amount >=
BankAccount.overdraft_limit:
            self.balance -= amount
            print(f"{amount} withdrawn. New balance:
{self.balance}")
        else:
```

```
            print(f"Insufficient funds to withdraw
{amount}")

    def __str__(self):
        return f"Balance: {self.balance}"
```

Step 2: Define Actions and the Command Class

```
from enum import Enum, auto

class Action(Enum):
    DEPOSIT = auto()
    WITHDRAW = auto()


class BankAccountCommand:
    """
    Encapsulates an action (deposit or withdraw) and an amount.
    Provides execute() to perform the action
    and undo() to reverse it.
    """
    def __init__(self, account: BankAccount, action: Action, amount:
int):
        self.account = account
        self.action = action
        self.amount = amount

    def execute(self):
        if self.action == Action.DEPOSIT:
            self.account.deposit(self.amount)
        elif self.action == Action.WITHDRAW:
            self.account.withdraw(self.amount)

    def undo(self):
        if self.action == Action.DEPOSIT:
            self.account.withdraw(self.amount)  # undo deposit
        elif self.action == Action.WITHDRAW:
            self.account.deposit(self.amount)  # undo withdrawal
```

Step 3: Use the Command Pattern

```
if __name__ == "__main__":
    # Create a bank account with an initial balance
    account = BankAccount(100)
    print(account)  # Balance: 100

    # Create a deposit command
    cmd = BankAccountCommand(account, Action.DEPOSIT, 50)

    # Execute the command
    cmd.execute()   # 50 deposited. New balance: 150
    print(account)  # Balance: 150

    # Undo the command
    cmd.undo()      # 50 withdrawn. New balance: 100
    print(account)  # Balance: 100
```

With this setup, you can:

- Easily **add new commands** (like transfer).
- **Log commands** and replay them later.
- Support **undo/redo functionality**.
- Decouple **the invoker (client code)** from **the receiver (BankAccount)**.

Iterator

Iterator accesses the elements of an object without exposing its underlying representation.

In object-oriented programming, the iterator pattern is a design pattern where an iterator is used to traverse a container and access the elements of the container.

Definition of the 'Thing' Class: First, we define a class called 'Thing' that has two properties 'a' and 'b.'

This is useful when:

- You want to loop through a collection of objects without caring about how they are stored internally.
- You need both **forward** and **reverse iteration**.
- You want to **hide implementation details** and provide a simple interface for traversal.
- You're working with custom objects that behave like lists or sequences.

Step 1: Define a class with properties

We'll create a class called `Thing` that has two attributes `a` and `b`.

```
class Thing:
    def __init__(self):
        self.a = 11
        self.b = 22
```

Here, `Thing` is our custom object that holds two values.

Step 2: Make it iterable (forward iteration)

To make the class iterable in Python, we need to define:

- __iter__(): returns the iterator object itself.
- __next__(): returns the next element until iteration is finished.

```
class Thing:
    def __init__(self):
        self.a = 11
```

```
        self.b = 22
        self._index = 0  # helper for iteration

    def __iter__(self):
        self._index = 0  # reset index each time iteration
starts
        return self

    def __next__(self):
        if self._index == 0:
            self._index += 1
            return self.a
        elif self._index == 1:
            self._index += 1
            return self.b
        else:
            raise StopIteration
```

Now we can loop over `Thing` just like a list.

Step 3: Add reverse iteration

Sometimes you need to traverse elements in **reverse order**. For that, we can create a separate method `backwards()` that returns a new iterator.

```
class Thing:
    def __init__(self):
        self.a = 11
        self.b = 22

    def __iter__(self):
        yield self.a
        yield self.b

    def backwards(self):
        yield self.b
        yield self.a
```

Here, we replaced `__next__` with Python's `yield`, which makes the code cleaner.

Step 4: Using the Iterator

```
thing = Thing()

print("Iterating over 'a' and 'b':")
for element in thing:
    print(element)

print("Iterating in reverse order:")
for element in thing.backwards():
    print(element)
```

Output:

```
Iterating over 'a' and 'b':
11
22
Iterating in reverse order:
22
11
```

Step 5: Iterating over collections

The pattern also applies naturally to lists, dictionaries, and other objects:

```
values = [100, 200, 300]

print("Iterating with index and value:")
for i, val in enumerate(values):
    print(f"Element at position {i} is {val}")

print("Iterating directly over values:")
for val in values:
    print(f"The value is {val}")
```

Now you have a **complete Python implementation** of the Iterator Pattern, with forward and backward iteration, and explained step by step.

Mediator

The mediator pattern adds a third-party object to control the interaction between two objects. It allows for flexible coupling between classes since it is the only class with detailed knowledge of its methods.

The mediator pattern defines an object that encapsulates how a set of objects interact. This pattern is considered a behavioral pattern due to the way it can alter the program's execution behavior. In object-oriented programming, programs often consist of many classes.

This is useful when:

- You have many objects that need to interact, and you want to avoid them being **tightly coupled**.
- You want to centralize complex communication logic in **one place**.
- You want to make it easier to add new participants without changing all existing classes.
- You're modeling systems like **chat rooms, air traffic control, or messaging hubs**.
- Example: creation of a chat room

Step 1: Define the `Person` class

This represents a person in a chat room. Each person has a name and a reference to the mediator (chat room).

```
class Person:
    def __init__(self, name, chat_room=None):
        self.name = name          # Person's name
        self.chat_room = chat_room  # Chat room they belong to

    # Method to send a message to the chat room
    def say(self, message):
        if self.chat_room:
            self.chat_room.send_message(self, message)

    # Method to receive a message
    def receive(self, message):
        print(f"{self.name} receives: {message}")
```

Step 2: Define the `ChatRoom` class (Mediator)

This acts as the **mediator** — it manages all the people in the chat room and controls how messages are distributed.

```
class ChatRoom:
    def __init__(self):
        self.people = []  # Stores all participants

    # Method to add a person to the chat room
    def join_person(self, person):
        self.people.append(person)
        person.chat_room = self  # Connect person to this
chat room

    # Method to send a message from one person to all
others
    def send_message(self, sender, message):
        for person in self.people:
            if person != sender:
                person.receive(f"{sender.name}: {message}")
```

Step 3: Create the chat room

```
chat_room = ChatRoom()
```

Step 4: Create people and add them to the room

```
john = Person("John")
tony = Person("Tony")

chat_room.join_person(john)
chat_room.join_person(tony)
```

Step 5: John sends a message

```
john.say("Hello, everyone!")
```

Step 6: Add a new person

```
doe = Person("Doe")
chat_room.join_person(doe)
```

Step 7: Doe sends a message

doe.say("Hi, everyone!")

Final Code:

```
class Person:
    def __init__(self, name, chat_room=None):
        self.name = name
        self.chat_room = chat_room

    def say(self, message):
        if self.chat_room:
            self.chat_room.send_message(self, message)

    def receive(self, message):
        print(f"{self.name} receives: {message}")


class ChatRoom:
    def __init__(self):
        self.people = []

    def join_person(self, person):
        self.people.append(person)
        person.chat_room = self

    def send_message(self, sender, message):
        for person in self.people:
            if person != sender:
                person.receive(f"{sender.name}: {message}")


# Usage
chat_room = ChatRoom()
```

```
john = Person("John")
tony = Person("Tony")

chat_room.join_person(john)
chat_room.join_person(tony)

john.say("Hello, everyone!")

doe = Person("Doe")
chat_room.join_person(doe)

doe.say("Hi, everyone!")
```

Memento

The Memento pattern allows restoring an object to its previous state.

The Memento pattern provides the capability to restore an object to a previous state. It is implemented using three objects: the originator, the caretaker, and the memento.

- **Originator**: the object whose state we want to save/restore (e.g., BankAccount).
- **Memento**: a simple object that stores the state.
- **Caretaker**: manages the history of states (saves and retrieves Memento objects).

This is useful when:

- You want to **implement undo/redo functionality**.
- You want to maintain **snapshots of an object's state** over time.
- You need to separate **state storage** from **business logic**.
- Example use cases: text editors (undo), games (save checkpoints), transactions (rollback).

Step 1: Define the `Memento` class

This class holds a snapshot of the bank account's state.

```
class Memento:
    def __init__(self, balance):
        self.balance = balance
```

Step 2: Define the `BankAccount` class (Originator)

This is the main class where deposits happen. It can save and restore its state using `Memento`.

```
class BankAccount:
    def __init__(self, balance=0):
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount
```

```
        return Memento(self.balance)  # return snapshot

    def restore(self, memento):
        self.balance = memento.balance

    def __str__(self):
        return f"Balance: {self.balance}"
```

Step 3: Define the `Caretaker` class

The caretaker holds a **history of mementos** so we can go back to earlier states.

```
class Caretaker:
    def __init__(self):
        self.mementos = []

    def add_memento(self, memento):
        self.mementos.append(memento)

    def get_memento(self, index):
        return self.mementos[index]
```

Step 4: Using the Memento Pattern

```
# Create an instance of BankAccount and Caretaker
bank_account = BankAccount(100)
caretaker = Caretaker()

# Make deposits and save states
caretaker.add_memento(bank_account.deposit(50))
print(bank_account)  # Balance: 150

caretaker.add_memento(bank_account.deposit(25))
print(bank_account)  # Balance: 175

# Restore to previous states
bank_account.restore(caretaker.get_memento(0))
```

```
print(bank_account)  # Balance: 150

bank_account.restore(caretaker.get_memento(1))
print(bank_account)  # Balance: 175
```

Final Code:

```
class Memento:
    def __init__(self, balance):
        self.balance = balance


class BankAccount:
    def __init__(self, balance=0):
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount
        return Memento(self.balance)

    def restore(self, memento):
        self.balance = memento.balance

    def __str__(self):
        return f"Balance: {self.balance}"


class Caretaker:
    def __init__(self):
        self.mementos = []

    def add_memento(self, memento):
        self.mementos.append(memento)

    def get_memento(self, index):
        return self.mementos[index]


# Usage Example
```

```
if __name__ == "__main__":
    bank_account = BankAccount(100)
    caretaker = Caretaker()

    caretaker.add_memento(bank_account.deposit(50))
    print(bank_account)  # Balance: 150

    caretaker.add_memento(bank_account.deposit(25))
    print(bank_account)  # Balance: 175

    bank_account.restore(caretaker.get_memento(0))
    print(bank_account)  # Balance: 150

    bank_account.restore(caretaker.get_memento(1))
    print(bank_account)  # Balance: 175
```

Observer

Allows multiple observer objects to watch an event.

The Observer pattern is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and automatically notifies them of any state changes, usually by calling one of their methods.

This is useful when:

- You want **multiple objects** to react to a single event (e.g., multiple systems listening for notifications).
- The subject should **not need to know the details** of the observers (who they are or how they react).
- You want to implement **event-driven programming**, logging, or UI update mechanisms.

Step 1. Define the Event system and Observer mechanism

In this step, we build the foundation:

- **Event**: Manages a list of subscribed observers, allows adding/removing them, and notifies them when triggered.
- **GetSick**: Represents the event details (in this case, location).
- **Person**: Acts as the observable subject that can “get sick” and notify all observers.

```
# Event class representing an observable event
class Event:
    def __init__(self):
        self.handlers = {}   # Dictionary to store
observers
        self.counter = 0     # Unique ID for each
subscription

    def subscribe(self, handler):
        """Subscribe an observer function and return its
ID."""
        self.counter += 1
```

```
        self.handlers[self.counter] = handler
        return self.counter

    def unsubscribe(self, handler_id):
        """Remove an observer by its subscription ID."""
        if handler_id in self.handlers:
            del self.handlers[handler_id]

    def fire(self, sender, args):
        """Notify all observers about the event."""
        for handler in self.handlers.values():
            handler(sender, args)


# GetSick class representing a specific event
class GetSick:
    def __init__(self, location):
        self.location = location


# Person class that will be the observable subject
class Person:
    def __init__(self, location):
        self.location = location
        self.get_sick = Event()  # "get sick" event system

    def catch_cold(self):
        """Simulate person getting sick and notify
observers."""
        event_info = GetSick(self.location)
        self.get_sick.fire(self, event_info)
```

Step 2. Using the Observer Pattern

Now let's simulate:

- Create a **Person**.
- Subscribe an **observer** (a doctor reacting to sickness).

- Fire the event multiple times.
- Unsubscribe the observer, and show that no notification is received anymore.

```
# Create an instance of Person
person = Person("Route ABC")

# Subscribe an observer (doctor) to the "get sick" event
subscription_id = person.get_sick.subscribe(
    lambda subject, event: print(f"A doctor has been called
to {event.location}")
)

# The person gets sick twice
person.catch_cold()
person.catch_cold()

# Unsubscribe the observer
person.get_sick.unsubscribe(subscription_id)

# The person gets sick again, but no notification is
received
person.catch_cold()
```

Step 3. Explanation of the Flow

1. **Subscription**:
   The observer (`lambda`) is registered to listen for the `get_sick` event.
   Every time `catch_cold()` is called, all subscribed handlers are notified.

2. **Event Trigger**:
   When `catch_cold()` is executed, the `fire()` method of `Event` is called.
   This notifies **all observers** and provides the event details (location).

3. **Unsubscription**:
   When the observer is unsubscribed, it no longer receives notifications.

The subject (`Person`) does not care who subscribed/unsubscribed — it only triggers the event.

Final Code:

```
# Event class representing an observable event
class Event:
    def __init__(self):
        self.handlers = {}   # Dictionary to store
observers
        self.counter = 0     # Unique ID for each
subscription

    def subscribe(self, handler):
        """Subscribe an observer function and return its
ID."""
        self.counter += 1
        self.handlers[self.counter] = handler
        return self.counter

    def unsubscribe(self, handler_id):
        """Remove an observer by its subscription ID."""
        if handler_id in self.handlers:
            del self.handlers[handler_id]

    def fire(self, sender, args):
        """Notify all observers about the event."""
        for handler in self.handlers.values():
            handler(sender, args)


# GetSick class representing a specific event
class GetSick:
    def __init__(self, location):
        self.location = location
```

```
# Person class that will be the observable subject
class Person:
    def __init__(self, location):
        self.location = location
        self.get_sick = Event()  # "get sick" event system

    def catch_cold(self):
        """Simulate person getting sick and notify
observers."""
        event_info = GetSick(self.location)
        self.get_sick.fire(self, event_info)


# Example usage
if __name__ == "__main__":
    # Create an instance of Person
    person = Person("Route ABC")

    # Subscribe an observer (doctor) to the "get sick"
event
    subscription_id = person.get_sick.subscribe(
        lambda subject, event: print(f"A doctor has been
called to {event.location}")
    )

    # The person gets sick twice
    person.catch_cold()
    person.catch_cold()

    # Unsubscribe the observer
    person.get_sick.unsubscribe(subscription_id)

    # The person gets sick again, but no notification is
received
    person.catch_cold()
```