Mediator

**1. Conceptual Understanding**

**What is the Mediator design pattern, and why is it used?**  
The **Mediator** pattern defines an object (the mediator) that encapsulates how a set of objects interact. It promotes loose coupling by preventing objects from referring to each other explicitly and instead communicating through the mediator.

**2. Implementation Questions**

**How would you implement the Mediator pattern?**  
A basic implementation in Python:

python

Copy code

# Mediator Interface

class Mediator:

def notify(self, sender, event):

pass

# Concrete Mediator

class ChatRoom(Mediator):

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

self.users = []

def add\_user(self, user):

self.users.append(user)

user.mediator = self

def notify(self, sender, message):

for user in self.users:

if user != sender:

user.receive(message)

# Colleague

class User:

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

self.name = name

self.mediator = None

def send(self, message):

print(f"{self.name} sends: {message}")

self.mediator.notify(self, message)

def receive(self, message):

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

# Usage

chat = ChatRoom()

alice = User("Alice")

bob = User("Bob")

chat.add\_user(alice)

chat.add\_user(bob)

alice.send("Hello!")

bob.send("Hi!")

**What are the key components of the Mediator pattern?**

1. **Mediator Interface**: Defines the communication interface between colleagues.
2. **Concrete Mediator**: Implements the coordination between colleagues.
3. **Colleagues**: Objects that communicate via the mediator instead of directly with each other.

**Can you implement a system where multiple types of events are handled by a Mediator?**  
Yes, extend the notify method to handle different event types or implement a publish-subscribe mechanism.

**3. Real-World Scenarios**

**What are common use cases for the Mediator pattern?**

1. **Chat Applications**: Mediator manages message exchange between users (e.g., group chats).
2. **UI Frameworks**: Widgets communicate with each other through a central mediator (e.g., a dialog box managing buttons and fields).
3. **Air Traffic Control**: Planes (colleagues) communicate with the tower (mediator) to manage landing and takeoff.
4. **E-commerce Platforms**: Mediators manage communication between inventory, orders, and payment services.

**How would you use the Mediator pattern in a UI framework?**  
In a dialog box, text fields, buttons, and checkboxes notify the mediator of changes, and the mediator updates other widgets accordingly.

**4. Behavioral Questions**

**What are the advantages and disadvantages of the Mediator pattern?**

* **Advantages**:
  1. Promotes loose coupling between components.
  2. Centralizes complex communication logic, simplifying maintenance.
  3. Makes components reusable since they don't depend on each other.
* **Disadvantages**:
  1. The mediator can become a bottleneck or overly complex ("God object").
  2. Debugging can be challenging due to indirect communication.

**How does the Mediator pattern differ from the Observer pattern?**

* **Mediator**: Centralizes communication between multiple objects.
* **Observer**: Allows one-to-many notification where observers react to changes in the subject.  
  Example:
* **Mediator**: A chatroom manages messages between users.
* **Observer**: A notification system alerts multiple observers when a user posts a message.

**How does the Mediator pattern support the Single Responsibility Principle?**  
It centralizes communication logic, allowing colleagues to focus solely on their own responsibilities.

**5. Code Debugging Questions**

**Given a Mediator implementation, identify issues or suggest improvements.**

* **Problem**: Colleagues depend directly on each other.
  + **Solution**: Decouple them by routing all communication through the mediator.
* **Problem**: Mediator handles too many responsibilities.
  + **Solution**: Split into multiple mediators or refactor using a chain of responsibility.

**How would you test a Mediator pattern implementation?**

1. Test communication between two colleagues to ensure the mediator routes messages correctly.
2. Test scalability by adding more colleagues and verifying all interactions.
3. Test edge cases, like when no mediator is present or when a colleague is removed.

**6. Advanced Topics**

**How would you implement a broadcast vs. direct communication model in the Mediator pattern?**

* **Broadcast**: Notify all colleagues except the sender.
* **Direct**: Add methods for specifying the target colleague in the mediator.

python

Copy code

def notify(self, sender, message, target=None):

if target:

target.receive(message)

else:

for user in self.users:

if user != sender:

user.receive(message)

**Can the Mediator pattern be combined with other patterns?**

* **Observer**: Mediator notifies observers when events occur.
* **Command**: Mediator executes commands on behalf of colleagues.
* **State**: Mediator manages the state transitions of its colleagues.

**7. Comparison with Other Patterns**

**How does the Mediator pattern differ from the Facade pattern?**

* **Mediator**: Manages communication between multiple objects, acting as an intermediary.
* **Facade**: Simplifies interaction with a subsystem by providing a unified interface.

**How does the Mediator pattern differ from the Chain of Responsibility pattern?**

* **Mediator**: Directs communication between multiple objects.
* **Chain of Responsibility**: Passes requests along a chain until one handler processes it.

**8. Real-Life Questions**

**Describe a real-life situation where you used the Mediator pattern.**  
Example: In an **e-commerce system**, a Mediator was used to handle communication between the **inventory service**, **order service**, and **payment service**. When an order was placed, the Mediator coordinated inventory checks and payment processing.

**Can the Mediator pattern be used in a multithreaded environment? How?**  
Yes, but synchronization mechanisms (e.g., locks or thread-safe data structures) must be used to avoid race conditions.