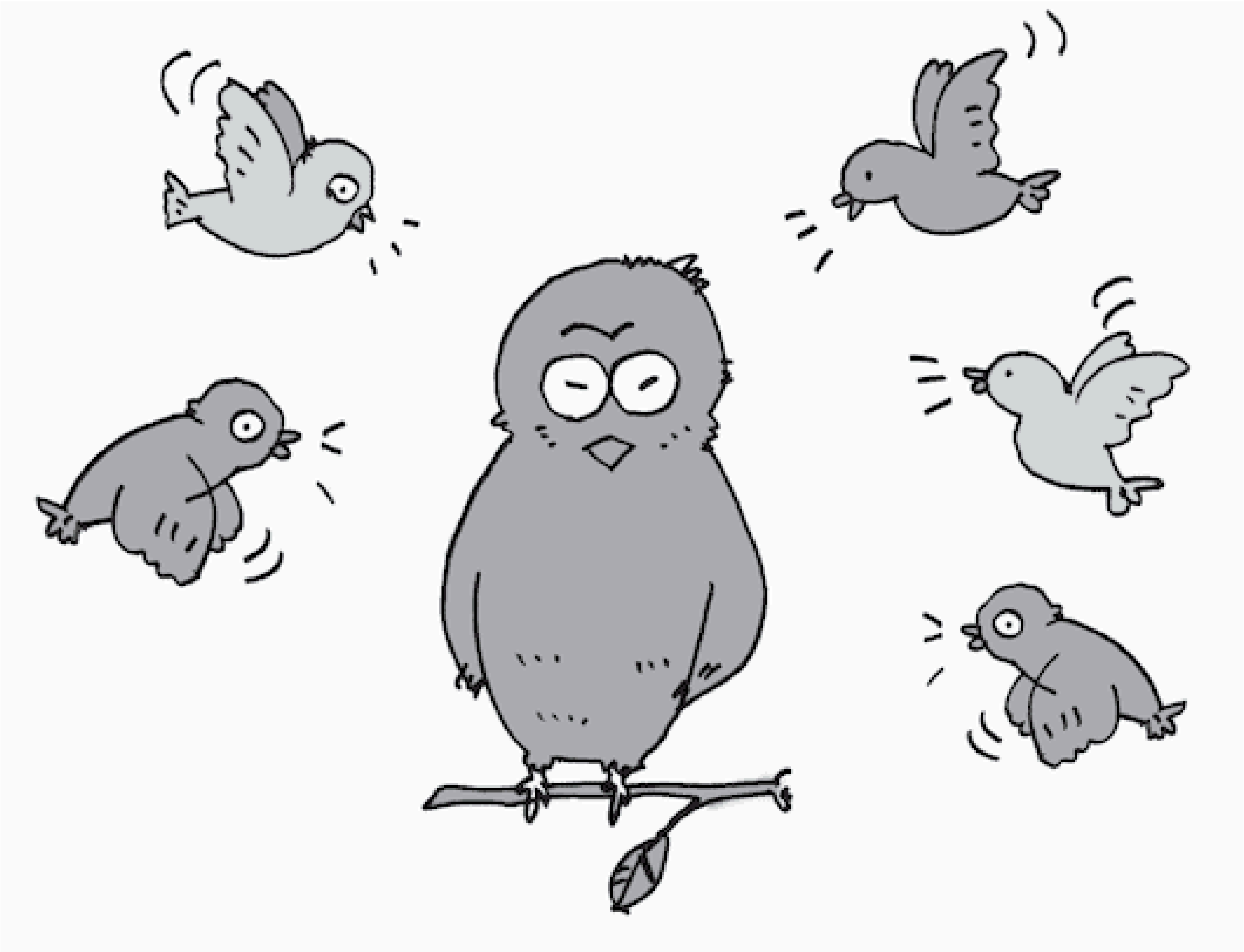
Mediator Pattern

Objects Communicate Through a Central Mediator



Mediator Pattern

Think of an **air traffic controller**:

- **Airplanes** don't talk to each other directly
- **All communication** goes through the **control tower**
- **Controller** coordinates all airplane interactions

The Problem

- We have **multiple objects** that need to **communicate** with each other.
- **Direct communication** creates **tight coupling** between objects.
- Adding new objects means **modifying existing objects** to know about them.

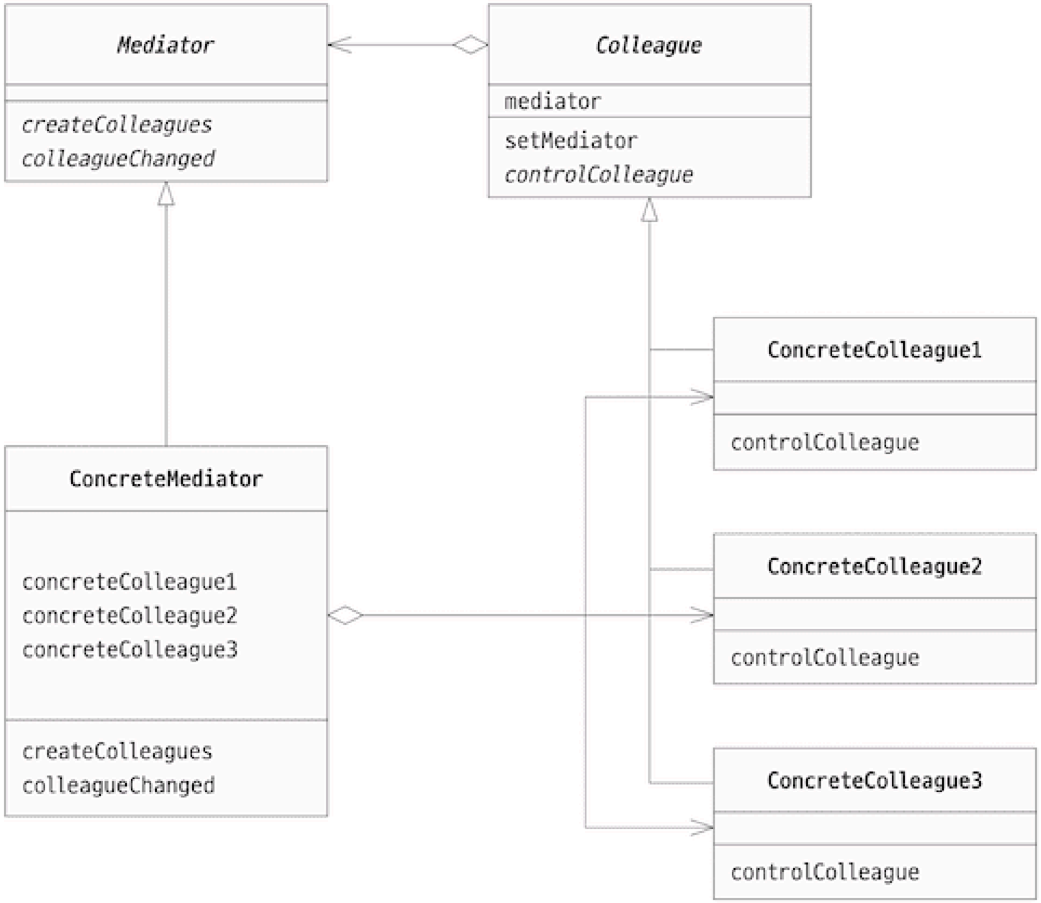
Challenge: How should objects interact without knowing about each other directly?

The *Facade* as the Solution

❌ **Direct**: Button has a reference to the TextBox

✅ **Mediated**: Button tells Mediator, Mediator clears TextBox

The Solution (Design)



Step 1: Understand the Players

- *Mediator* (abstract): Interface for coordination
  - **ConcreteMediator**: Implements coordination logic (SimpleDialog)
- *Colleague* (abstract): Interface for mediated objects
  - **ConcreteColleagues**: Objects that work through a mediator (Button, TextBox)

Key Relationship: Colleagues only know Mediator, Mediator knows all Colleagues

Step 2: Interface

Abstract classes define the pattern structure:

- *Mediator*: Defines `notify(sender, event)` method for colleagues to call
- *Colleague*: Defines `set_mediator()` and `notify_mediator()` methods

Purpose: These interfaces ensure all components follow the same protocol

Step 3: Understand abstractions

Concrete classes provide the actual functionality:

- **ConcreteMediator** (SimpleDialog): Contains all coordination logic
- **ConcreteColleagues** (Button, TextBox): Simple objects with basic operations

Purpose: Implements the specific behavior while following the abstract interfaces

Code

Main

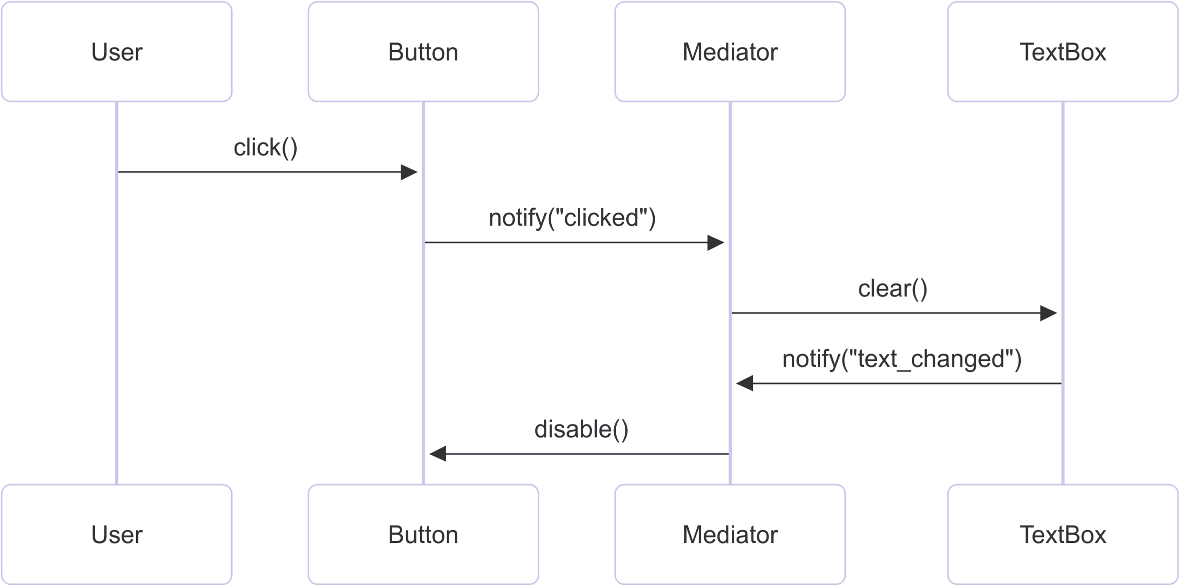
```python
def mediator():
    # Creating the mediator (which creates colleagues)
    dialog = SimpleDialog()

    # User interaction – Button click:
    dialog.button.click()
```

The Button (clear) & TextBox are created, and the Dialog (mediator) is ready.

```
Button 'Clear' created
TextBox created with text: 'Hello World'
Button 'Clear' enabled
Simple Dialog ready
-----------
```

The Button (clear) is clicked, and the notifier is notified to clear the button.

```
Button 'Clear' was clicked!
Mediator received: clicked from Button
TextBox cleared
Mediator received: text_changed from TextBox
Button 'Clear' is disabled
```



- User clicks the button, and mediator mediates all the actions with the notify() method.

Abstract Mediator

```python
from abc import ABC, abstractmethod

class Mediator(ABC):
    @abstractmethod
    def notify(self, sender, event):
        """Called when a colleague needs coordination"""
        pass
```

Abstract Colleague

```python
class Colleague(ABC):
    def __init__(self, mediator=None):
        self._mediator = mediator

    def notify_mediator(self, event):
        if self._mediator:
            self._mediator.notify(self, event)
```

Concrete Colleagues

Each Colleague uses `notify_mediator` to communicate.

```python
class Button(Colleague):
    def click(self):
        print("Button clicked!")
        self.notify_mediator("clicked")  # Tell mediator

class TextBox(Colleague):
    def __init__(self, mediator=None):
        super().__init__(mediator)
        self.text = "Hello World"

    def clear(self):
        self.text = ""
        self.notify_mediator("text_changed")  # Tell mediator
```

Concrete Mediator

The mediator analyzes the sender and event to process the event accordingly.

```python
class SimpleDialog(Mediator):
    def __init__(self):
        # Create colleagues and set their mediator
        self.button = Button(self)
        self.textbox = TextBox(self)

    def notify(self, sender, event):
        if sender == self.button and event == "clicked":
            self.textbox.clear()  # Coordinate!

        elif sender == self.textbox and event == "text_changed":
            if self.textbox.text:
                self.button.enable()   # Has text → enable
            else:
                self.button.disable()  # No text → disable
```

Discussion

Misunderstanding of Mediator

Prevents adding new communication rules — Mediator makes it *hard* to add or change rules among elements

Wrong! mediator itself can become complex and hard to maintain, mediator makes it it *easier* to add or change rules.

Key Pattern Benefits

Loose Coupling

- Button doesn't know about TextBox
- TextBox doesn't know about Button
- They only know about the Mediator interface

Centralized Control

- All interaction logic in SimpleDialog
- Easy to change rules (modify only mediator)
- Clear separation of concerns

Extensible

- Add new colleagues easily
- Create different mediators
- Change interaction rules

When to Use Mediator

✅ **Use Mediator when:**

- Objects need to interact, but shouldn't know about each other
- You want to avoid tight coupling between objects
- You have complex interactions between multiple objects
- You want centralized control of object interactions

❌ **Don't use when:**

- Only two objects with effortless interaction
- Performance is critical (mediator adds indirection)
- Objects naturally belong together

Pattern Variations

Simple Mediator (Our example)

- Basic notify/coordinate approach
- Good for simple UI interactions

Event-Based Mediator

- Uses event objects instead of simple strings
- Good for complex event systems

Observer + Mediator

- Combines with the Observer pattern
- Good for publish/subscribe scenarios

Related Patterns

- **Observer**: Mediator can use Observer to notify colleagues
- **Facade**: Both provide a unified interface, Mediator coordinates
- **Chain of Responsibility**: Both avoid direct coupling

Mediator vs Observer

Observer Pattern:

```
Subject → Observer1, Observer2, Observer3  (One-to-many)
```

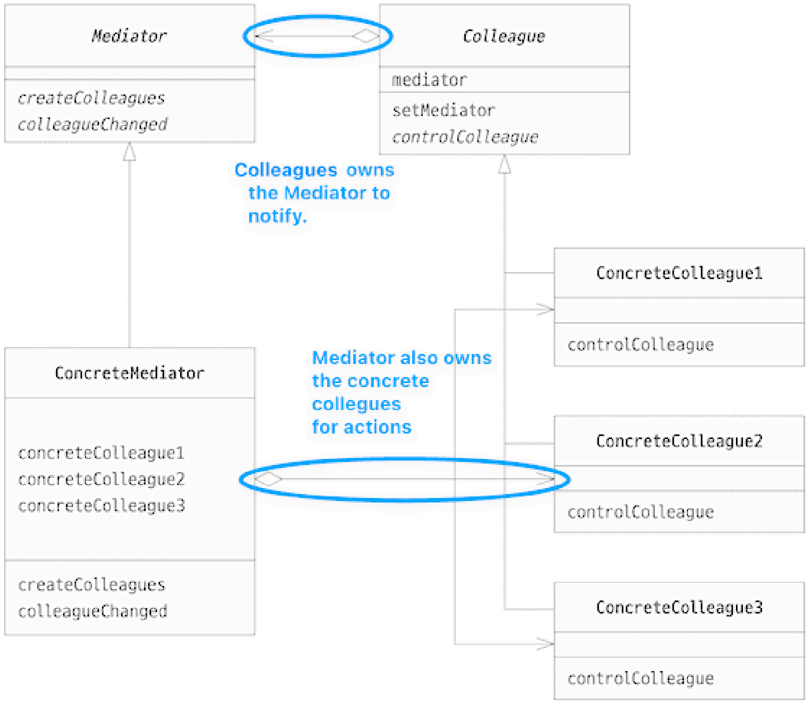
Mediator Pattern:

```
Colleague1 ↔️ Mediator ↔️ Colleague2  (Many-to-many coordination)
```

Observer: Broadcasts changes to many observers

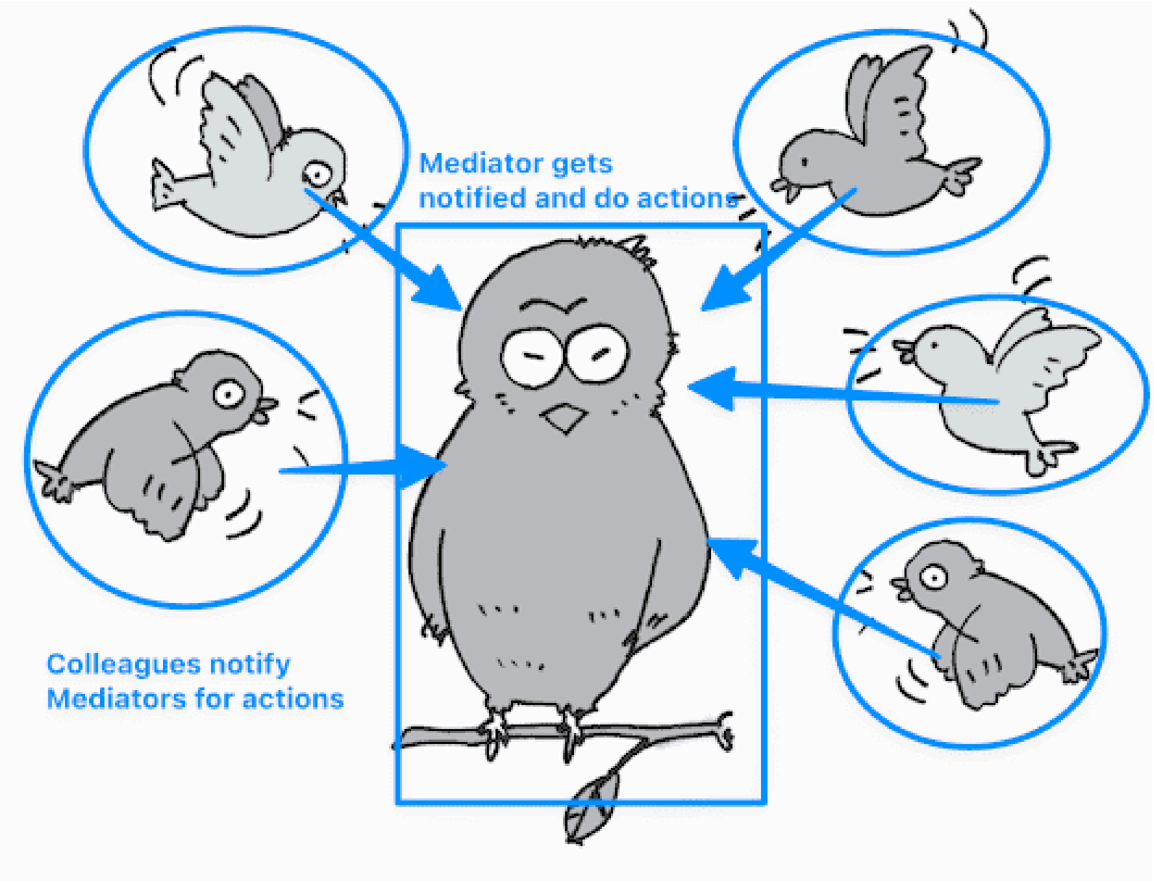
Mediator: Coordinates specific interactions between colleagues

UML Diagram



Colleagues owns the Mediator to notify.

Mediator also owns the concrete collegues for actions


Mediator gets
notified and do actions
Colleagues notify
Mediators for actions