Mediator Pattern

Scenario:

Imagine you're developing a document editor with various components, such as a toolbar, a text editor, a status bar, and a sidebar. Each of these components may need to communicate with one another. For example, when a user clicks a button on the toolbar, the text editor may need to update, and the status bar may need to reflect the change. Without a structured way to manage these interactions, the components could become tightly coupled, leading to a tangled web of dependencies and making the system hard to maintain.

Purpose:

The Mediator Pattern defines an object that encapsulates how a set of objects interact. Instead of having the objects communicate directly with one another, they communicate through the mediator. This pattern promotes loose coupling by keeping objects from referring to each other explicitly, making their interactions easier to manage and maintain.

When to Use:

When you have complex interactions between multiple objects, and you want to avoid tight coupling between them.

When you want to centralize control of interactions between components in a single place.

When you want to make it easier to add, remove, or change the interactions between components without modifying the components themselves.

Key Concepts:

Mediator: The interface that defines methods for communication between the Colleagues. It coordinates the interactions between the different components.

Concrete Mediator: Implements the Mediator interface and contains the logic to manage and coordinate the interactions between the different components.

Colleagues: The components that interact with each other through the mediator. Each colleague communicates with the mediator instead of with other colleagues directly.

How It Works:

Mediator Interface: Defines methods that allow colleagues to communicate indirectly via the mediator.

Concrete Mediator: Implements the Mediator interface and contains the logic for coordinating communication between the colleagues.

Colleague Interface: Represents a component that interacts with the mediator.

Concrete Colleague: Implements the Colleague interface and communicates with other colleagues through the mediator.

Client: Configures the mediator and the colleagues, establishing the communication network.

Implementation Example:

Here's how the implementation might look in Java:

```
// Mediator Interface

interface Mediator {

    void notify(Component sender, String event);

}


// Concrete Mediator

class DocumentEditorMediator implements Mediator {

    private Toolbar toolbar;

    private TextEditor textEditor;

    private StatusBar statusBar;


    public void setToolbar(Toolbar toolbar) {

        this.toolbar = toolbar;
```

```java
    }

    public void setTextEditor(TextEditor textEditor) {
        this.textEditor = textEditor;
    }

    public void setStatusBar(StatusBar statusBar) {
        this.statusBar = statusBar;
    }

    @Override
    public void notify(Component sender, String event) {
        if (sender == toolbar && event.equals("bold")) {
            textEditor.applyBold();
            statusBar.updateStatus("Bold applied");
        } else if (sender == toolbar && event.equals("italic")) {
            textEditor.applyItalic();
            statusBar.updateStatus("Italic applied");
        } else if (sender == textEditor && event.equals("textChanged")) {
            statusBar.updateStatus("Text updated");
        }
    }
}

// Colleague Interface
abstract class Component {
    protected Mediator mediator;


    public Component(Mediator mediator) {
        this.mediator = mediator;
    }


    public void setMediator(Mediator mediator) {
        this.mediator = mediator;
    }
}


// Concrete Colleague - Toolbar
class Toolbar extends Component {
    public Toolbar(Mediator mediator) {
        super(mediator);
    }


    public void clickBold() {
        System.out.println("Toolbar: Bold clicked");
        mediator.notify(this, "bold");
    }


    public void clickItalic() {
        System.out.println("Toolbar: Italic clicked");
        mediator.notify(this, "italic");
    }
```

```
}


// Concrete Colleague - TextEditor

class TextEditor extends Component {

    public TextEditor(Mediator mediator) {

        super(mediator);

    }


    public void applyBold() {

        System.out.println("TextEditor: Applying bold");

        mediator.notify(this, "textChanged");

    }


    public void applyItalic() {

        System.out.println("TextEditor: Applying italic");

        mediator.notify(this, "textChanged");

    }

}


// Concrete Colleague - StatusBar

class StatusBar extends Component {

    public StatusBar(Mediator mediator) {

        super(mediator);

    }


    public void updateStatus(String message) {
        System.out.println("StatusBar: " + message);
    }
}


// Client Code
public class DocumentEditor {
    public static void main(String[] args) {
        DocumentEditorMediator mediator = new DocumentEditorMediator();


        Toolbar toolbar = new Toolbar(mediator);
        TextEditor textEditor = new TextEditor(mediator);
        StatusBar statusBar = new StatusBar(mediator);


        mediator.setToolbar(toolbar);
        mediator.setTextEditor(textEditor);
        mediator.setStatusBar(statusBar);


        toolbar.clickBold();
        toolbar.clickItalic();
    }
}
```

Key Points to Remember:

Mediator Pattern promotes loose coupling by centralizing the communication between components in a mediator. Instead of having components interact directly with each other, they communicate through the mediator, which coordinates their interactions.

Single Responsibility Principle: The pattern helps adhere to the Single Responsibility Principle by allowing you to encapsulate the communication logic in the mediator rather than spreading it across the components.

Advantages:

Reduces Complexity: By centralizing communication, the pattern reduces the complexity of component interactions, making the system easier to manage and maintain.

Encapsulation of Interaction: The mediator encapsulates the interaction logic, making it easier to modify or extend the communication without changing the components themselves.

Improves Reusability: Components become more reusable because they no longer need to know about the interactions with other components.

Disadvantages:

Mediator Overhead: The mediator can become overly complex as more components and interactions are added, leading to a "god object" that handles too much logic.

Single Point of Failure: The mediator becomes a central point of communication, so if it's not designed carefully, it can become a bottleneck or single point of failure in the system.