Design patterns. Behavioural software design pattern

Mediator pattern

1. Design pattern description

Problems that the mediator pattern solves

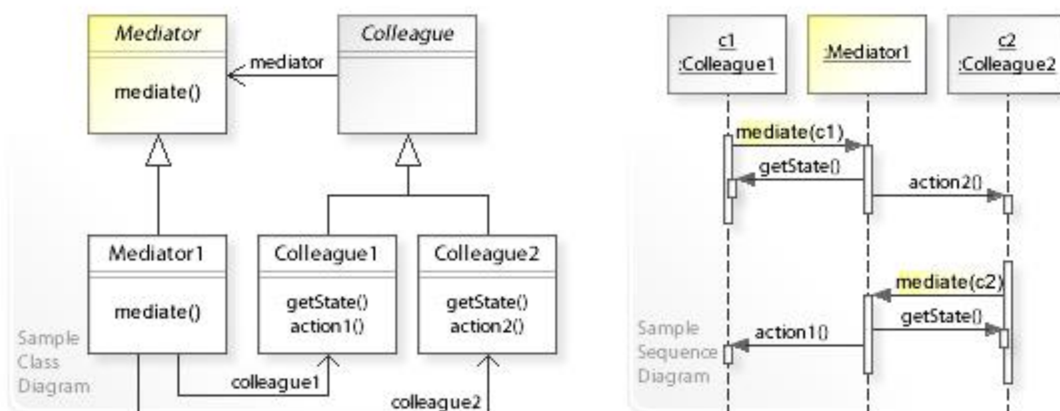
- Tight coupling between a set of interacting objects should be avoided.
- It should be possible to change the interaction between a set of objects independently

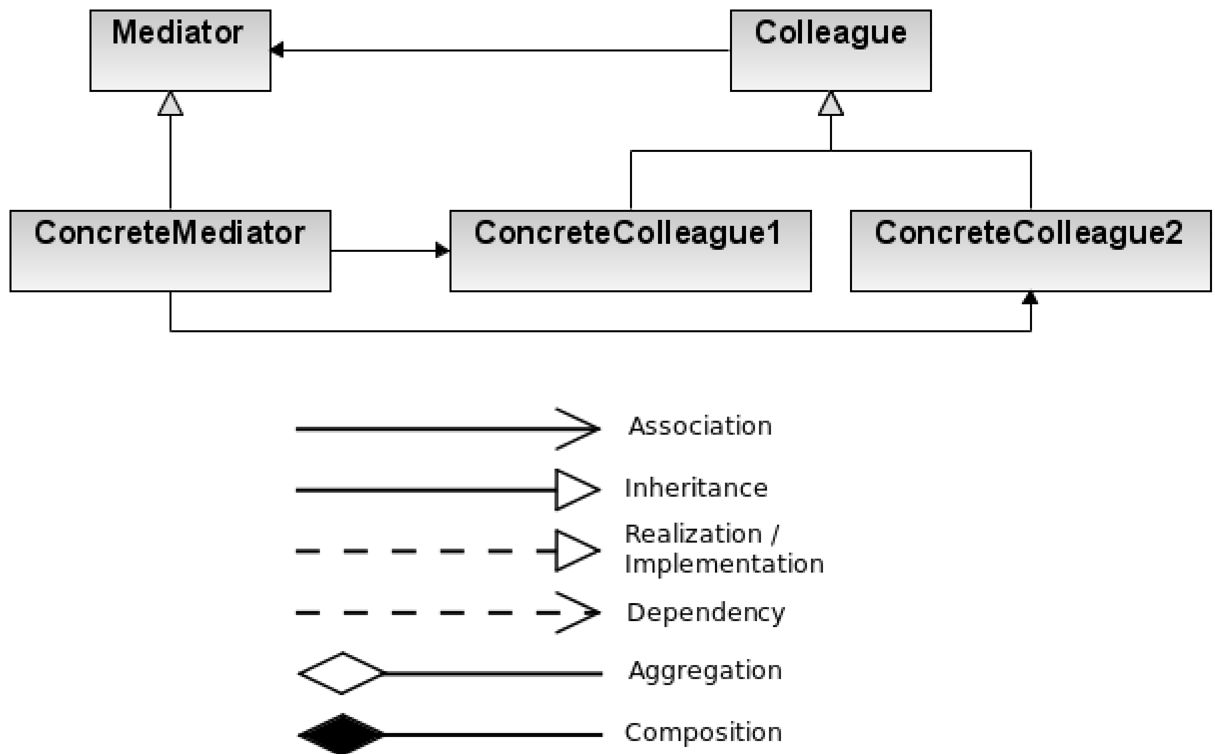
Defining a set of interacting objects by accessing and updating each other directly is inflexible because it tightly couples the objects to each other and makes it impossible to change the interaction independently from (without having to change) the objects. And it stops the objects from being reusable and makes them hard to test. *Tightly coupled objects* are hard to implement, change, test, and reuse because they refer to and know about many different objects.

What solution does the Mediator design pattern describe?

- Define a separate (mediator) object that encapsulates the interaction between a set of objects.
- Objects delegate their interaction to a mediator object instead of interacting with each other directly.

For example, defining a set of interacting objects (like buttons, menu items, and input/display fields) in a GUI/Web application. It should be possible (1) to change the interaction behavior independently from (without having to change) the objects and (2) to reuse the objects in different applications.





2. Design pattern example

```
(1) Changing state of Colleague1 ...
    Colleague1: My state changed to: Hello World1! Calling my mediator ...
    Mediator  : Mediating the interaction ...
    Colleague2: My state synchronized to: Hello World1!

(2) Changing state of Colleague2 ...
    Colleague2: My state changed to: Hello World2! Calling my mediator ...
    Mediator  : Mediating the interaction ...
    Colleague1: My state synchronized to: Hello World2!
```

```
package com.sample.mediator.basic;
public class MyApp {

    public static void main(String[] args) {

        Mediator1 mediator = new Mediator1();

        // Creating colleagues

        // and configuring them with a Mediator1 object.

        Colleague1 c1 = new Colleague1(mediator);

        Colleague2 c2 = new Colleague2(mediator);

        // Setting mediator's colleagues.
```

```java
        mediator.setColleagues(c1, c2);
        System.out.println("(1) Changing state of Colleague1 ...");
        c1.setState("Hello World1!");

         System.out.println("\n(2) Changing state of Colleague2 ...");
         c2.setState("Hello World2!");
     }
 }
```

```java
package com.sample.mediator.basic;
public abstract class Mediator {
    // Mediating the interaction between colleagues.
    public abstract void mediate(Colleague colleague);
}
```

```java
package com.sample.mediator.basic;
public class Mediator1 extends Mediator {
    private Colleague1 colleague1;
    private Colleague2 colleague2;
    void setColleagues(Colleague1 colleague1, Colleague2 colleague2) {
        this.colleague1 = colleague1;
        this.colleague2 = colleague2;
    }
public void mediate(Colleague colleague) {
    System.out.println("    Mediator  : Mediating the interaction ...");
    // Message from colleague1 that its state has changed.
    if (colleague == colleague1) {
        // Performing an action on colleague2.
        String state = colleague1.getState();
        colleague2.action2(state);
    }
    // Message from colleague2 that its state has changed.
    if (colleague == colleague2) {
        // Performing an action on colleague1.
        String state = colleague2.getState();
        colleague1.action1(state);
    }
    }
```

```java
}
package com.sample.mediator.basic;
public abstract class Colleague {
    Mediator mediator;
    public Colleague(Mediator mediator) {
        this.mediator = mediator;
    }
}

package com.sample.mediator.basic;
public class Colleague1 extends Colleague {
    private String state;
    public Colleague1(Mediator mediator) {
        super(mediator); // Calling the super class constructor
        }
    public String getState() {
        return state;
    }
    void setState(String state) {
        if (state != this.state) {
            this.state = state;
            System.out.println("    Colleague1: My state changed to: "
                    + this.state + " Calling my mediator ...");
            mediator.mediate(this);
        }
    }
    void action1 (String state) {
        // For example, synchronizing and displaying state.
        this.state = state;
        System.out.println("    Colleague1: My state synchronized to: "
                    + this.state);
    }
}

package com.sample.mediator.basic;
public class Colleague2 extends Colleague {
    private String state;
    public Colleague2(Mediator mediator) {
```

```java
            super(mediator);
        }
        public String getState() {
            return state;
        }
        void setState(String state) {
            if (state != this.state) {
                this.state = state;
                System.out.println("    Colleague2: My state changed to: "
                        + this.state + " Calling my mediator ...");
                mediator.mediate(this);
            }
        }
        void action2 (String state) {
            // For example, synchronizing and displaying state.
            this.state = state;
            System.out.println("    Colleague2: My state synchronized to: "
                        + this.state);
        }
    }
```

3. Existing pattern in SWAN?

Not explicitly , TargetParameters would require one but we use the *handle* functionality.

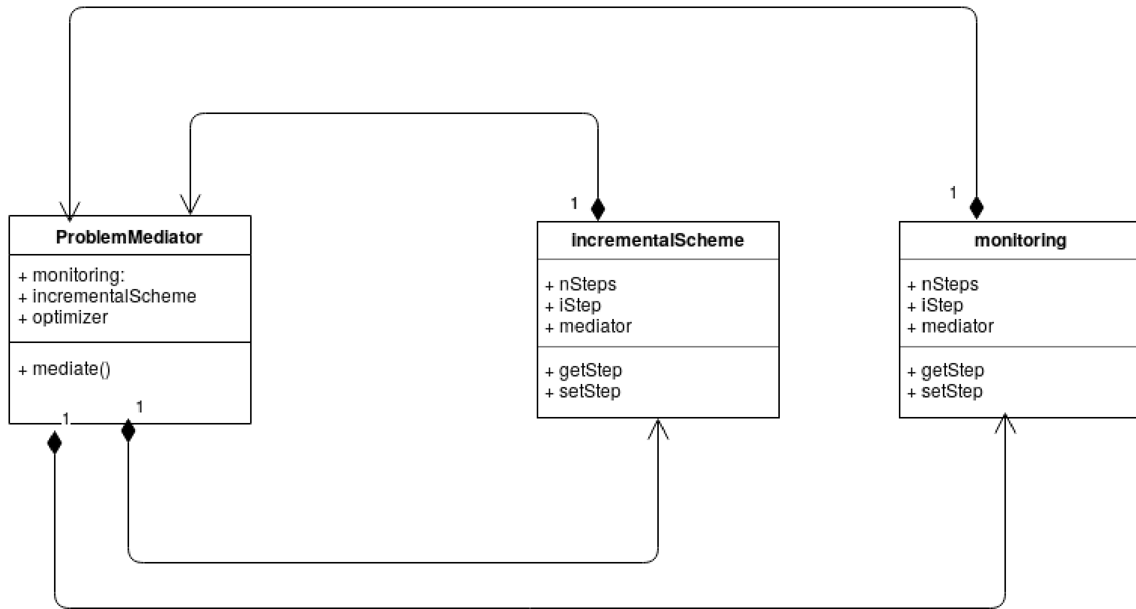
4. Design proposal in SWAN

Current design in Optimizer requires to have the IncrementalScheme as a property

```matlab
function itHas = hasExceededStepIterations(obj)
    iStep = obj.incrementalScheme.iStep;
    nStep = obj.incrementalScheme.nSteps;
    itHas = obj.nIter >= obj.maxIter*(iStep/nStep);
end

function refreshMonitoring(obj)
    iStep = obj.incrementalScheme.iStep;
    nStep = obj.incrementalScheme.nSteps;
    obj.monitor.refresh(obj.nIter,obj.hasFinished,iStep,nStep);
end

function printHistory(obj)
    iStep = obj.incrementalScheme.iStep;
    obj.historyPrinter.print(obj.nIter,iStep);
end
```


ProblemMediator
+ monitoring:
+ incrementalScheme
+ optimizer
+ mediate()
1
1
incrementalScheme
+ nSteps
+ iStep
+ mediator
+ getStep
+ setStep
1
monitoring
+ nSteps
+ iStep
+ mediator
+ getStep
+ setStep
1