CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2009
Doc 21 Memento, Facade, Mediator
23 Apr 2009

References

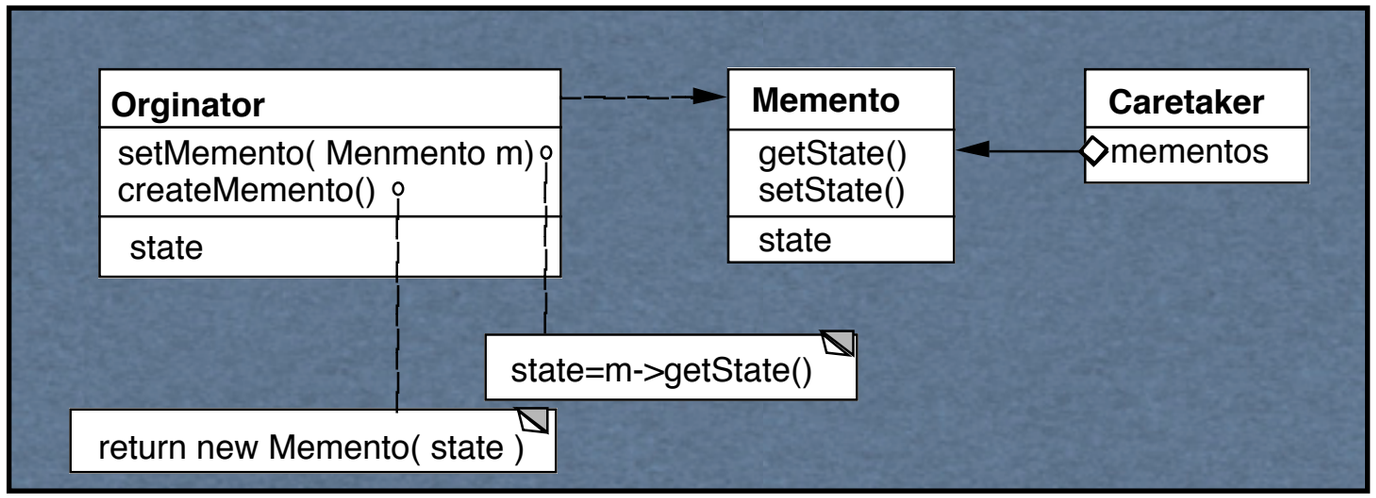
Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides, 1995, pp. 273-282, 283-292, 185-206

The Design Patterns Smalltalk Companion, Alpert, Brown, Woolf, 1998, pp. 287-304, 179-212

Memento

Store an object's internal state, so the object can be restored to this state later without violating encapsulation

undo, rollbacks



Only originator:

Can access Memento’s get/set state methods

Create Memento

Example

```
package Examples;
class Memento{
	private Hashtable savedState = new Hashtable();

	protected Memento() {}; //Give some protection

	protected void setState( String stateName, Object stateValue ) {
		savedState.put( stateName, stateValue );
	}

	protected Object getState( String stateName) {
		return savedState.get( stateName);
	}

	protected Object getState(String stateName, Object defaultValue ) {
		if ( savedState.containsKey( stateName ) )
			return savedState.get( stateName);
		else
			return defaultValue;
	}
}
```

Sample Orginator

```
package Examples;
class ComplexObject {
        private String name;
        private int someData;
        private Vector objectAsState = new Vector();

        public Memento createMemento() {
                Memento currentState = new Memento();
                currentState.setState( "name", name );
                currentState.setState( "someData", new Integer(someData) );
                currentState.setState( "objectAsState", objectAsState.clone() );
                return currentState;
        }

        public void restoreState( Memento oldState) {
                name = (String) oldState.getState( "name", name );
                objectAsState = (Vector) oldState.getState( "objectAsState" );
                Integer data = (Integer) oldState.getState( "someData");
                someData = data.intValue();
        }
```

Why not let the Originator save its old state?

```
class ComplexObject {
        private String name;
        private int someData;
        private Vector objectAsState = new Vector();
        private Stack history;

        public createMemento() {
                Memento currentState = new Memento();
                currentState.setState( "name", name );
                currentState.setState( "someData", new Integer(someData) );
                currentState.setState( "objectAsState", objectAsState.clone() );
                history.push(currentState);
        }

        public void restoreState() {
                Memento oldState = history.pop();
                name = (String) oldState.getState( "name", name );
                objectAsState = (Vector) oldState.getState( "objectAsState" );
                Integer data = (Integer) oldState.getState( "someData");
                someData = data.intValue();
        }
```

Some Consequences

Expensive

Narrow & Wide interfaces - Keep data hidden

```
Class Memento {
public:
    virtual ~Memento();
private:
    friend class Originator;
    Memento();
    void setState(State*);
    State* GetState();
```

```
class Originator {
    private String state;

 private class Memento {
    private String state;
    public Memento(String stateToSave)
        { state = stateToSave; }
public String getState() { return state; }
    }

    public Object memento()
        { return new Memento(state);}
```

Using Clone to Save State

```
interface Memento extends Cloneable { }

class ComplexObject implements Memento {
        private String name;
        private int someData;

        public Memento createMemento() {
                Memento myState = null;
                try {
                        myState =  (Memento) this.clone();
                }
                catch (CloneNotSupportedException notReachable) {
                }
                return myState;
        }

        public void  restoreState( Memento savedState) {
                ComplexObject myNewState = (ComplexObject)savedState;
                name = myNewState.name;
                someData = myNewState.someData;
        }
}
```

What if Protocol

When there are complex validations or
performing operations that make it difficult to restore later

Make a copy of the Originator

Perform operations on the copy

Check if operations invalidate the internal state of copy

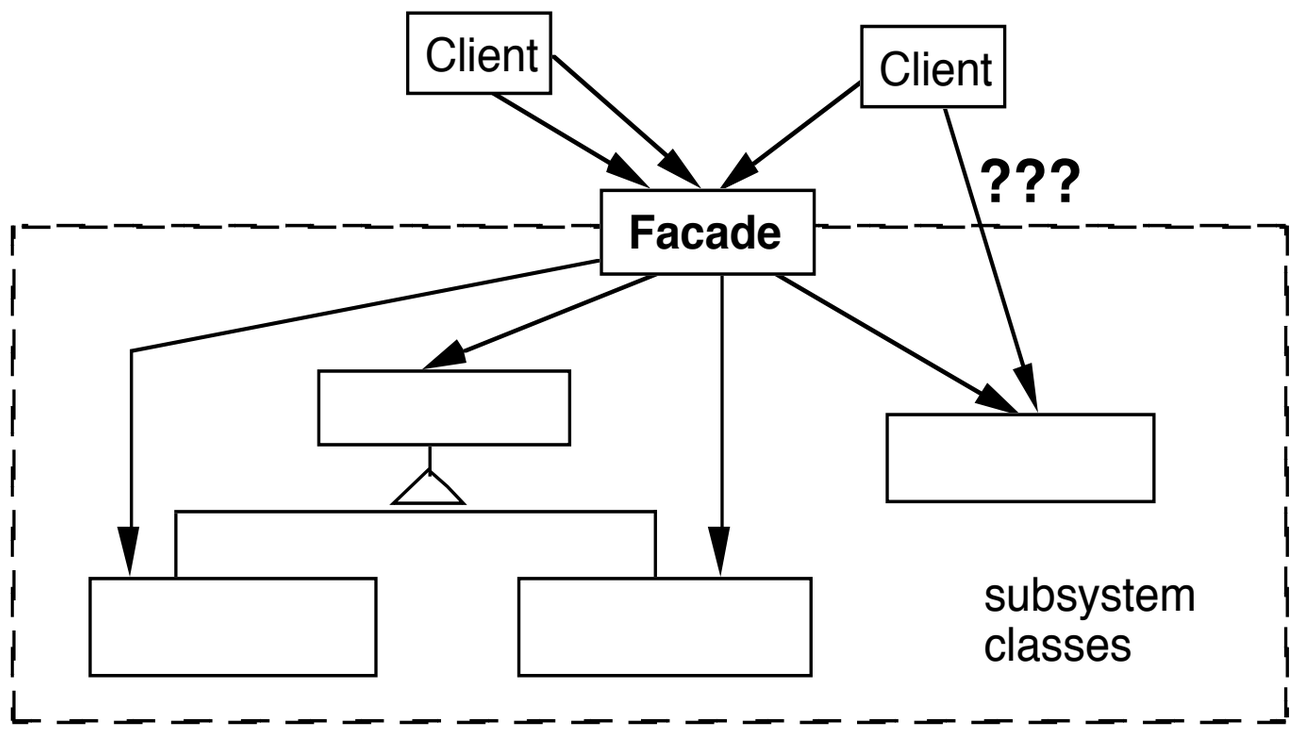
If so discard the copy & raise an exception

Else perform the operations on the Originator

The Facade Pattern

Create a class that is the interface to the subsystem

Clients interface with the Facade class to deal with the subsystem



Consequences of Facade Pattern

It hides the implementation of the subsystem from clients

It promotes weak coupling between the subsystems and its clients

It does not prevent clients from using subsystem classes directly, should it?

Facade does not add new functionality to the subsystem

12

Compiler Example

The VisualWorks Smalltalk compiler system has 75 classes

Programmers only use Compiler, which uses the other classes

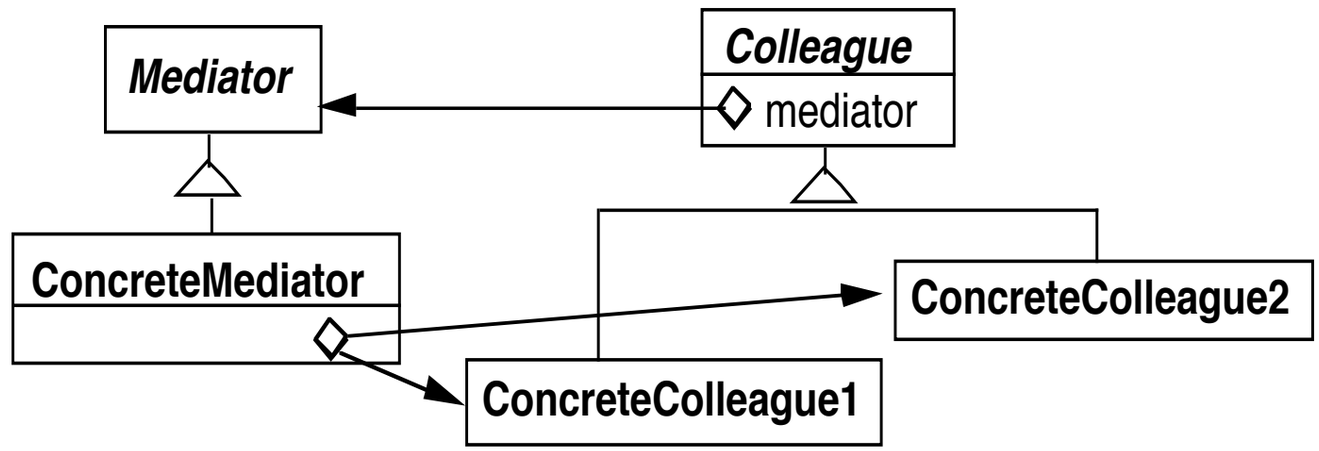
Compiler evaluate: '100 factorial'

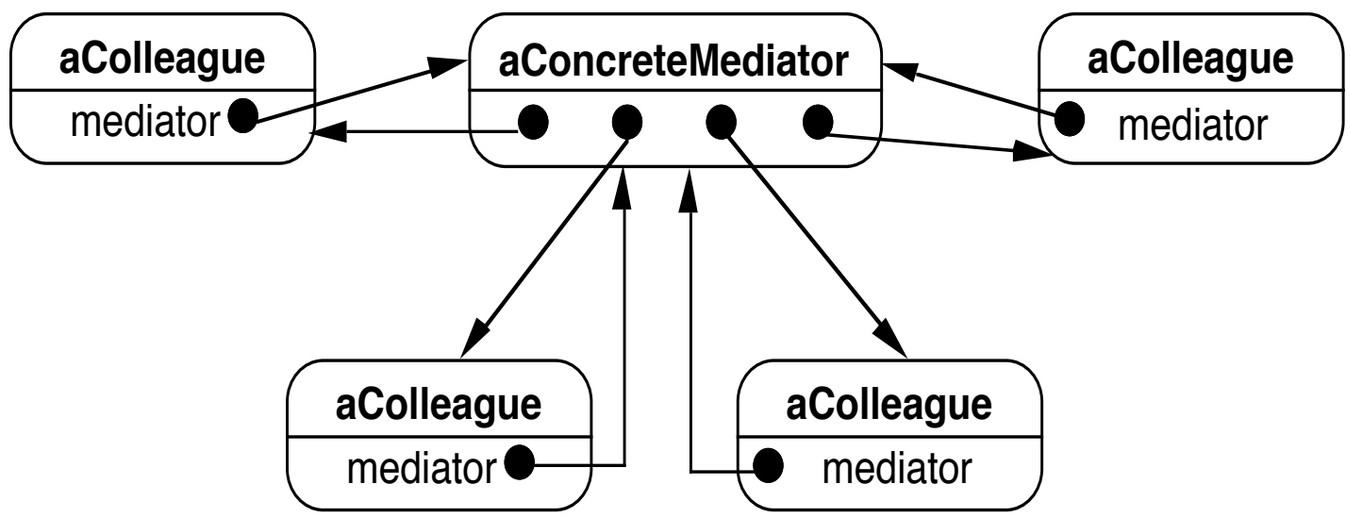
```
| method compiler |
method := 'reset
        "Resets the counter to zero"
        count := 0.'.

compiler := Compiler new.
compiler
        parse:method
        in: Counter
        notifying: nil
```


13

Mediator

A mediator is responsible for controlling and coordinating the interactions of a group of objects





Participants

Mediator

Defines an interface for communicating with Colleague objects

ConcreteMediator

Implements cooperative behavior by coordinating Colleague objects

Knows and maintains its colleagues

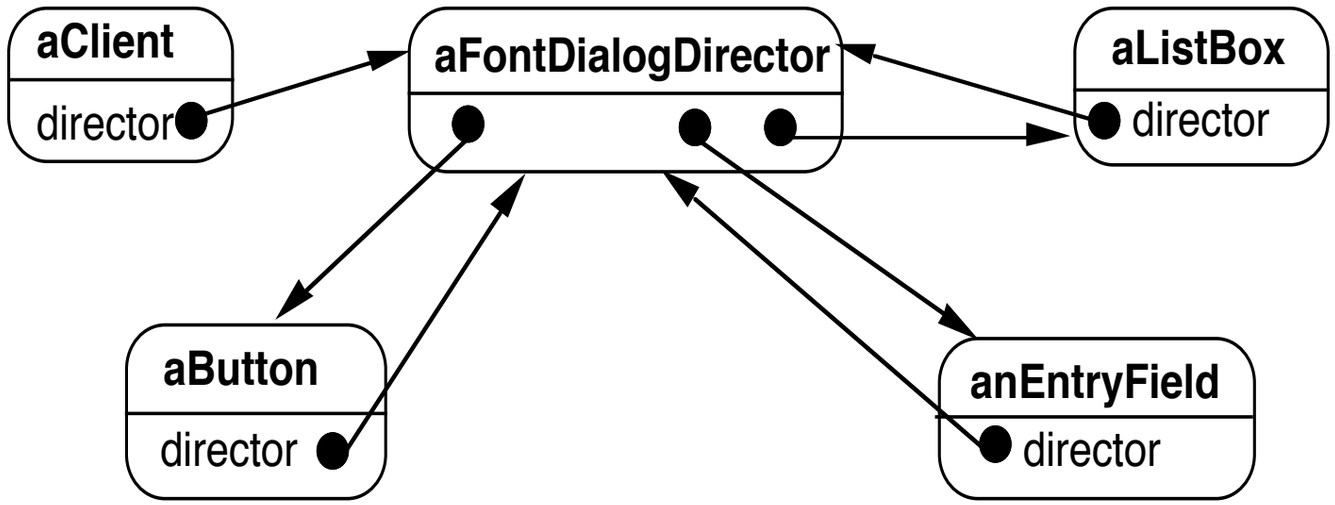
Colleague classes

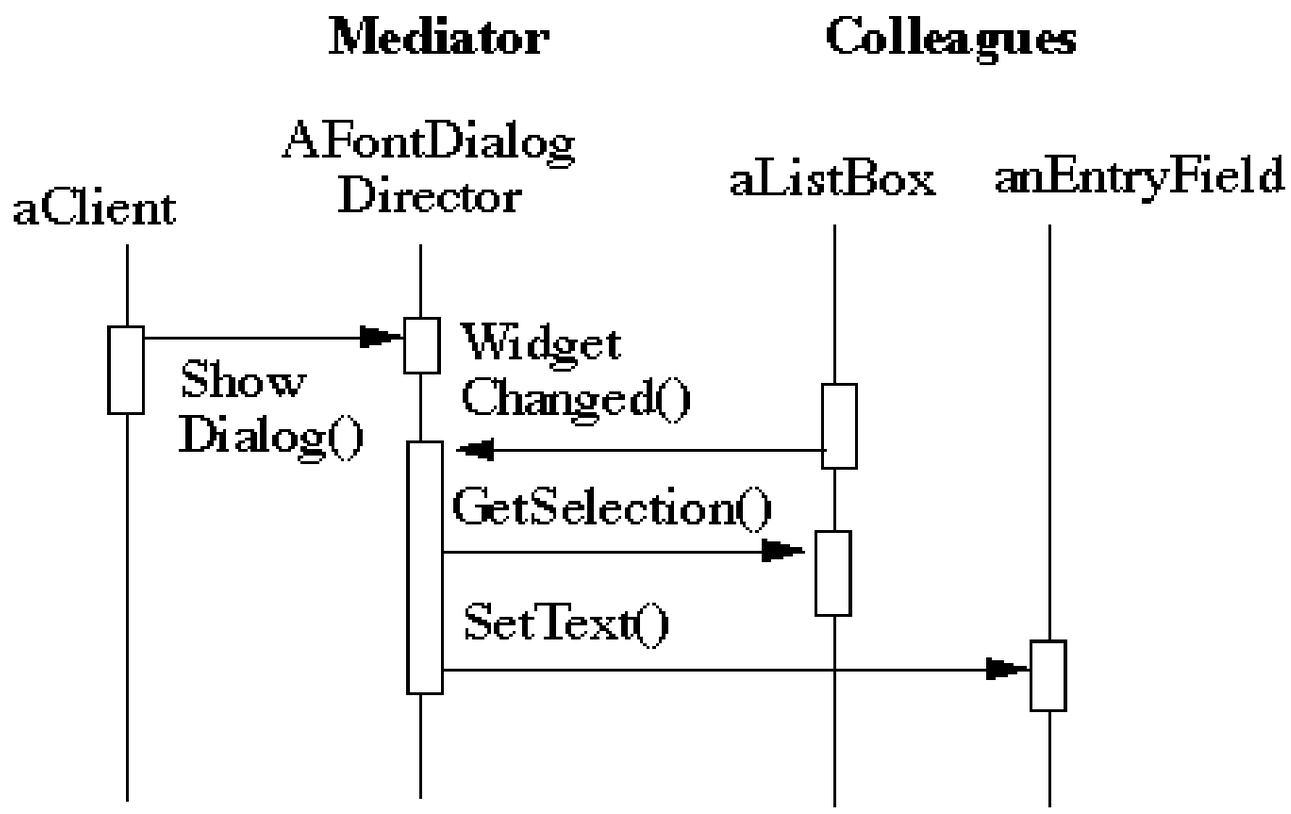
Each Colleague class knows its Mediator object

Each colleague communicates with its mediator whenever it would have otherwise communicated with another colleague

15

Motivating Example - Dialog Boxes





How does this differ from a God Class?

17

When to use the Mediator Pattern

When a set of objects communicate in a well-defined but complex ways

When reusing an object is difficult because it refers to and communicates with many other objects

When a behavior that's distributed between several classes should be customizable without a lot of subclassing

How do Colleagues & Mediators Communicate?

Explicit methods in Mediator

```
class DialogDirector
    {
    private Button ok;
    private Button cancel;
    private ListBox courses;

    public void listBoxItemSelected() { blah}

    public void listBoxScrolled() { blah }
    etc.
    }
```


19

How do Colleagues & Mediators Communicate?

Generic change notification

```
class DialogDirector {
    private Button ok;
    private Button cancel;
    private ListBox courses;

    public void widgetChanged( Object changedWidget) {
        if ( changedWidget == ok )                blah
        else if (  changedWidget == cancel )      more blah
        else if (  changedWidget == courses )     even more blah
```