CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2004
Doc 15 Interpreter, Mediator, Type Object
Contents

Reference

Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides, 1995, pp. 243-256, 273-282

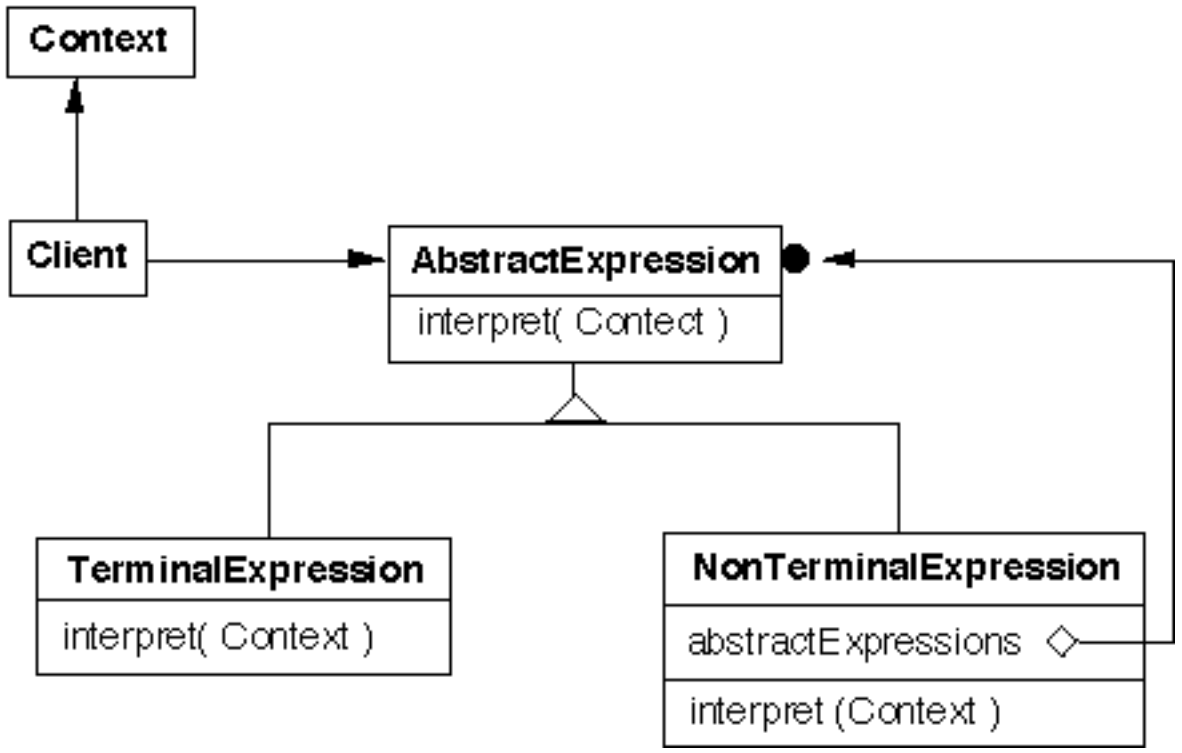
The Design Patterns Smalltalk Companion, Alpert, Brown, Woolf, 1998, pp. 261-272, 287-296

Type Object, Ralph Johnson & Bobby Woolf in Pattern Languages of Program Design 3, Edited by Martin, Riehle, Buschmann, 1998, pp. 47-65

Interpreter

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language

Structure



Given a language defined by a simple grammar with rules like:

R ::= R1 R2 ... Rn

you create a class for each rule

The classes can be used to construct a tree that represents elements of the language

Example - Boolean Expressions

```
BooleanExpression ::=
            Variable        |
            Constant        |
            Or              |
            And             |
            Not             |
            BooleanExpression

And   ::= BooleanExpression 'and' BooleanExpression
Or    ::= BooleanExpression 'or' BooleanExpression
Not      ::= 'not' BooleanExpression

Constant ::= 'true' | 'false'
Variable ::= String

public interface BooleanExpression{
   public boolean evaluate( Context values );
   public BooleanExpression replace( String varName,
                        BooleanExpression replacement );
   public Object clone();
   public String toString();
}
```

Sample Use

```
public class Test {
  public static void main( String args[] ) throws Exception  {
    BooleanExpression left =
      new Or(  Constant.getTrue(), Variable.get( "x" ) );
    BooleanExpression right =
      new And(  Variable.get( "w" ), Variable.get( "x" ) );

    BooleanExpression all = new And(  left, right );

    System.out.println( all );
    Context values = new Context();
    values.setValue( "x", true );
    values.setValue( "w", false );

    System.out.println( all.evaluate( values ) );
    System.out.println( all.replace( "x", right ) );
    }
  }
```

Output

```
((true or x) and (w and x))
false
((true or (w and x)) and (w and (w and x)))
```

And

And ::= BooleanExpression ‘&&’ BooleanExpression

```
public class And implements BooleanExpression {
   private BooleanExpression leftOperand;
   private BooleanExpression rightOperand;

   public And( BooleanExpression leftOperand,
               BooleanExpression rightOperand) {
      this.leftOperand = leftOperand;
      this.rightOperand = rightOperand;
   }

   public boolean evaluate( Context values ) {
      return leftOperand.evaluate( values ) &&
            rightOperand.evaluate( values );
   }

   public BooleanExpression replace( String varName,
         BooleanExpression replacement ) {
      return new And( leftOperand.replace( varName, replacement),
               rightOperand.replace( varName, replacement) );
   }

   public Object clone() {
      return new And( (BooleanExpression) leftOperand.clone( ),
               (BooleanExpression)rightOperand.clone( ) );
   }

   public String toString(){
      return "(" + leftOperand.toString() + " and " +
            rightOperand.toString() + ")";
   }
}
```

Or

Or ::= BooleanExpression ‘or’ BooleanExpression

```
public class Or implements BooleanExpression {
   private BooleanExpression leftOperand;
   private BooleanExpression rightOperand;

   public Or( BooleanExpression leftOperand,
              BooleanExpression rightOperand) {
      this.leftOperand = leftOperand;
      this.rightOperand = rightOperand;
   }

   public boolean evaluate( Context values ) {
      return leftOperand.evaluate( values ) ||
            rightOperand.evaluate( values );
   }

   public BooleanExpression replace( String varName,
         BooleanExpression replacement ) {
      return new Or( leftOperand.replace( varName, replacement),
              rightOperand.replace( varName, replacement) );
   }

   public Object clone() {
      return new Or( (BooleanExpression) leftOperand.clone( ),
              (BooleanExpression)rightOperand.clone( ) );
   }

   public String toString() {
      return "(" + leftOperand.toString() + " or " +
            rightOperand.toString() + ")";
   }
}
```

Not

Not ::= ‘not’ BooleanExpression

```
public class Not implements BooleanExpression {
   private BooleanExpression operand;

   public Not( BooleanExpression operand) {
      this.operand = operand;
   }

   public boolean evaluate( Context values ) {
      return  ! operand.evaluate( values );
   }

   public BooleanExpression replace( String varName,
       BooleanExpression replacement ) {
      return new Not( operand.replace( varName, replacement) );
   }

   public Object clone() {
      return new Not( (BooleanExpression) operand.clone( ) );
   }

   public String toString() {
      return "( not " +  operand.toString() + ")";
   }
```

Constant

Constant ::= ‘true’ | ‘false’

```
public class Constant implements BooleanExpression {
   private boolean value;
   private static Constant True = new Constant( true );
   private static Constant False = new Constant( false );

   public static Constant getTrue() {
      return True;
   }

   public static Constant getFalse(){
      return False;
   }

   private Constant( boolean value) {
      this.value = value;
   }

   public boolean evaluate( Context values ) {
      return  value;
   }

   public BooleanExpression replace( String varName,
         BooleanExpression replacement ) {
      return this;
   }

   public Object clone() {
      return this;
      }

   public String toString() {
      return String.valueOf( value );
   }
}
```

Variable

Variable ::= String

```
public class Variable implements BooleanExpression {
   private static Hashtable flyWeights = new Hashtable();

   private String name;

   public static Variable get( String name ) {
      if ( ! flyWeights.contains( name ))
         flyWeights.put( name , new Variable( name ));

      return (Variable) flyWeights.get( name );
   }

   private Variable( String name ) {
      this.name = name;
   }

   public boolean evaluate( Context values ) {
      return values.getValue( name );
   }

   public BooleanExpression replace( String varName,
         BooleanExpression replacement ) {
      if ( varName.equals( name ) )
         return (BooleanExpression) replacement.clone();
      else
         return this;
   }

   public Object clone() {
      return this;
   }

   public String toString() { return name; }
}
```

Context

```
public class Context {
  Hashtable values = new Hashtable();

  public boolean getValue( String variableName ) {
    Boolean wrappedValue = (Boolean) values.get( variableName );
    return wrappedValue.booleanValue();
  }

  public void setValue( String variableName, boolean value ) {
    values.put( variableName, new Boolean( value ) );
  }
}
```

Consequences

It's easy to change and extend the grammar

Implementing the grammar is easy

Complex grammars are hard to maintain

Adding new ways to interpret expressions

The visitor pattern is useful here

Implementation

The pattern does not talk about parsing!

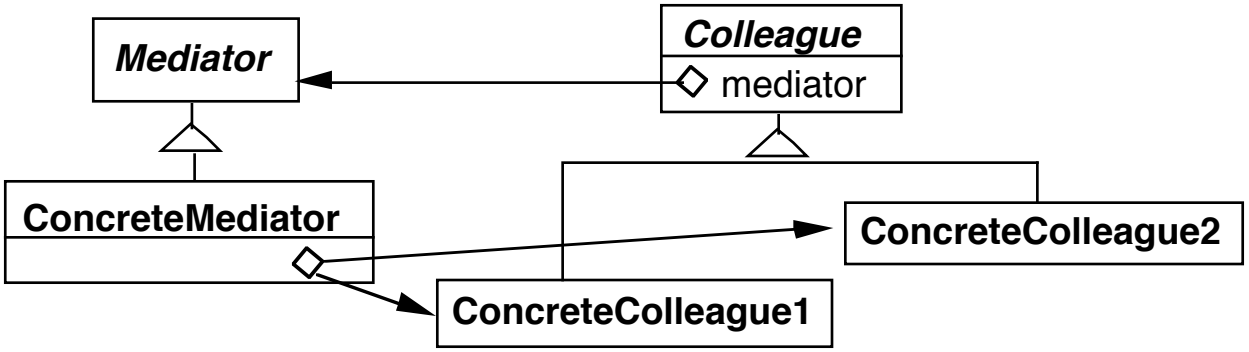
Flyweight

- If terminal symbols are repeated many times using the Flyweight pattern can reduce space usage
- The above example has each terminal class manage the flyweights for its objects, since Java does limited support for protecting constructors

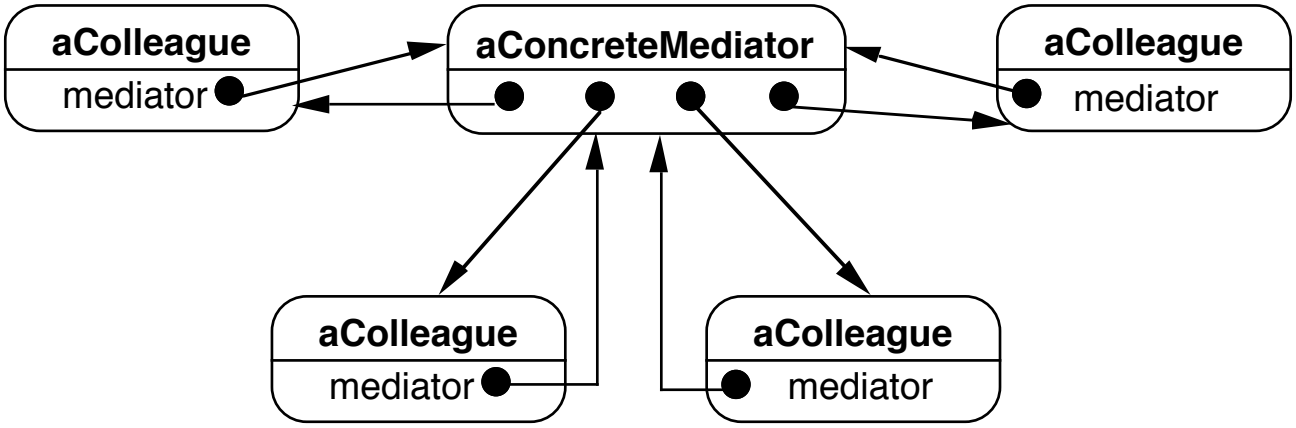
Mediator

A mediator is responsible for controlling and coordinating the interactions of a group of objects (not data structures)

Structure
Classes



Objects



Participants

Mediator

- Defines an interface for communicating with Colleague objects

ConcreteMediator

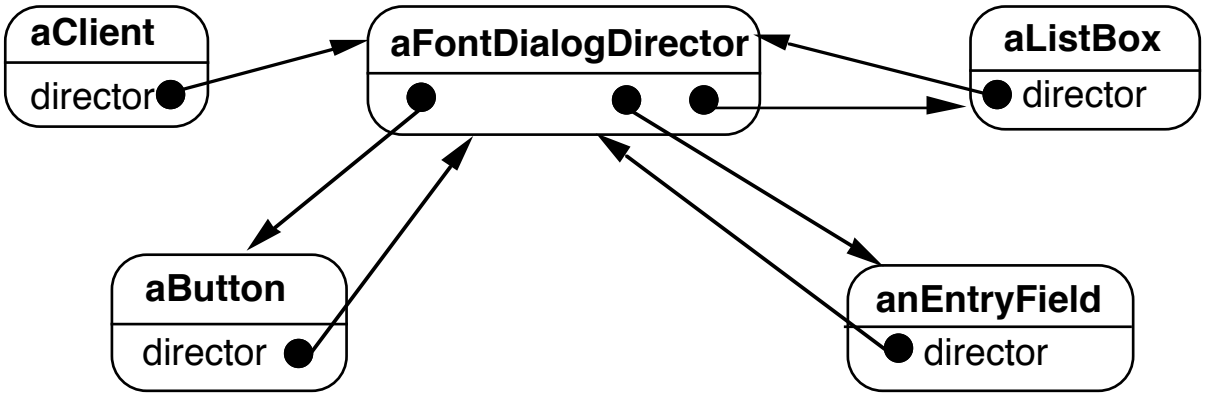
- Implements cooperative behavior by coordinating Colleague objects
- Knows and maintains its colleagues

Colleague classes

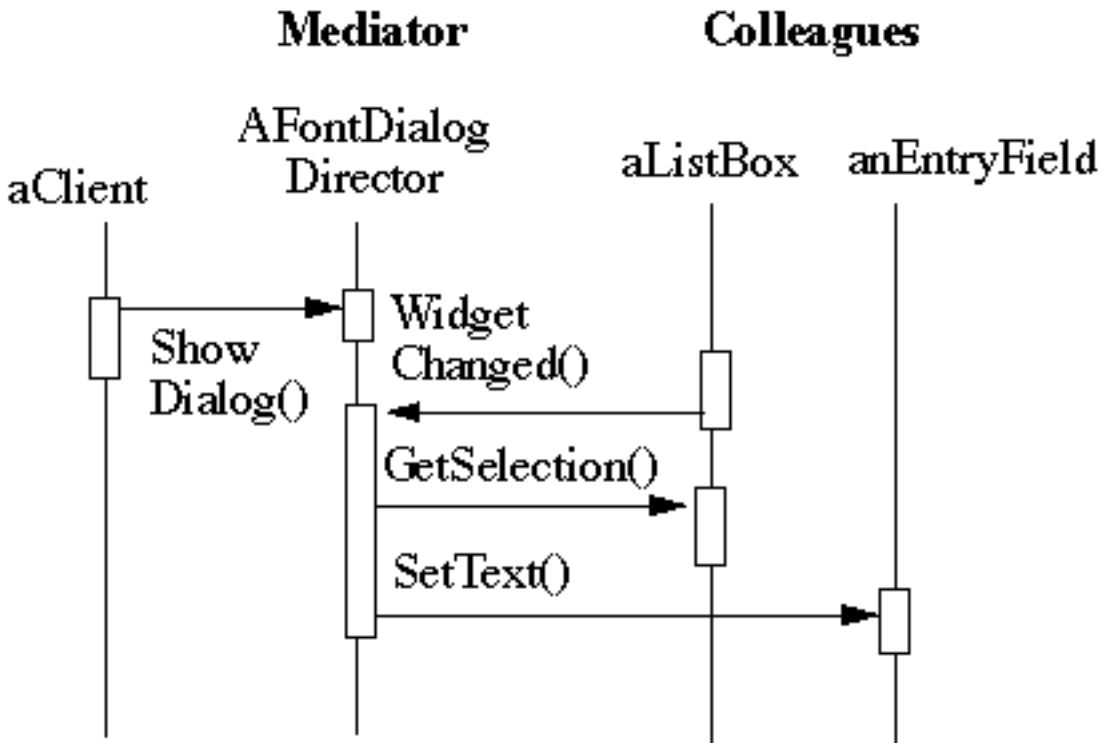
- Each Colleague class knows its Mediator object
- Each colleague communicates with its mediator whenever it would have otherwise communicated with another colleague

Motivating Example
Dialog Boxes

Objects



Interaction



How does this differ from a God Class?

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

Issues
How do Colleagues and Mediators Communicate?

1) Explicit methods in Mediator

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

  public void ListBoxItemSelected() { blah}

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

2) Generic change method

```
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
  }
}
```

3) Generic change method overloaded

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

  public void widgetChanged( Button changedWidget)
    {
    if ( changedWidget == ok )
      blah
    else if (  changedWidget == cancel )
      more blah
    }

  public void widgetChanged( ListBox changedWidget)
    {
    now find out how it changed and
    respond properly
    }
  }
```

Differences from Facade

Facade does not add any functionality, Mediator does

Subsystem components are not aware of Facade

Mediator's colleagues are aware of Mediator and interact with it

Type Object

Intent

Decouples instances from their classes so those classes can be implemented as instances of a class

- Allows new classes to be created dynamically at runtime
- Lets a system provide its own type-checking rules

Also Known As

- Power Type
- Item Descriptor
- Metaobject
- Data Normalization

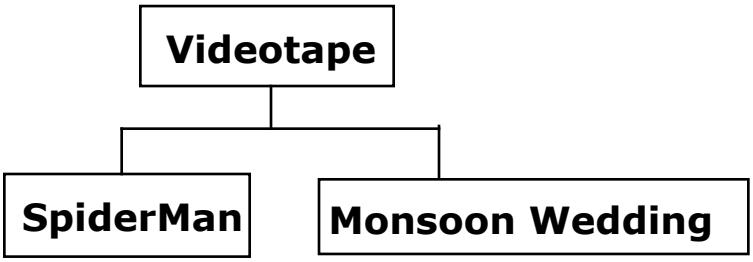
Motivation

Video Rental Store Inventory

Need to keep track of all the movies in the inventory

What
- About individual movies
- Multiple copies of a movie

Subclassing does not Work



What happens when new movies come out?

Instances of Videotape do not Work

Using one instance of Videotape class per movie

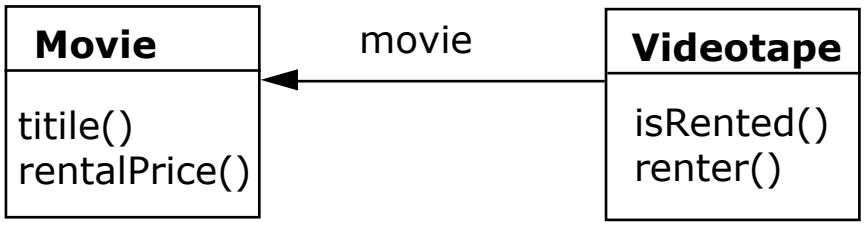
- Need to track multiple copies of a movie

Using one instance of Videotape for each copy of a movie

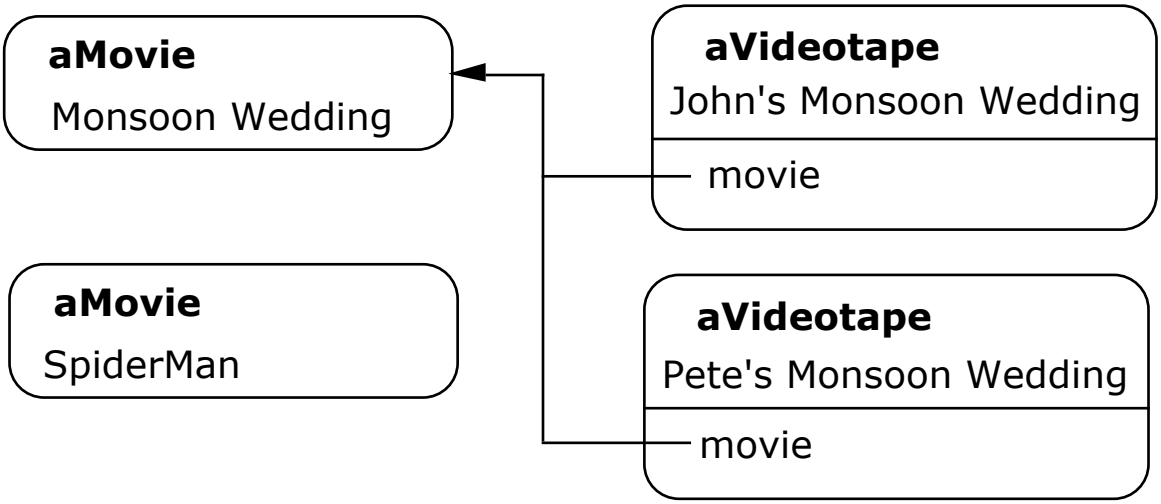
- Each copy contains a lot of duplicate information

Type Object Solution

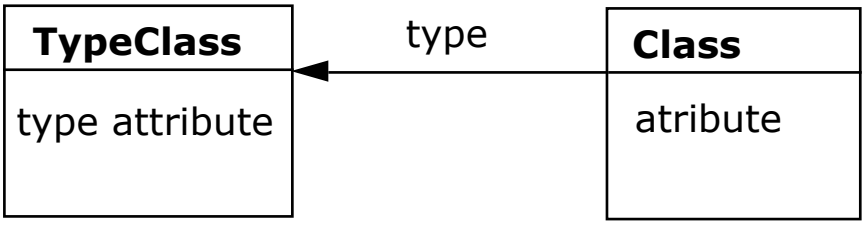
Class Structure



Object Structure



Type Object Structure



TypeClass (Movie)

- Is the class of TypeObject
- Has a separate instance for each type of Object

TypeObject (SpiderMan, Monsoon Wedding)

- Is instance of TypeClass
- Represents a type of Object
- Implements some of the behavior for TypeClass