8. Behavioral Patterns – 3 of 3

***SW Design Patterns*,**
by Boyan Bontchev,
FMI - Sofia University
2006/2016

Annotation

- Definitions
- Properties
- Intent, motivation, structure, participants, collaborations, consequences, implementation issues about:
  - Template Method
  - Strategy
  - Visitor
  - Memento
- Examples in Java

References

- Gamma, Helm, Johnson, Vlissides **("Gang of Four" - GoF)** *Design Patterns: Elements of Reusable Object-Oriented Software*, 1995
- *Design Patterns Explained*, by Allan Shalloway and James Trott, Prentice Hall, 2001
- THE DESIGN PATTERNS JAVA COMPANION, by JAMES W. COOPER, Adison-Wesley, October 2, 1998

Design pattern catalog - GoF

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	• Factory Method	• Adapter	• Interperter
	Object	• Abstract Factory • Builder • Prototype • Singleton	• Bridge • Composite • Decorator • Facade • Flyweight • Proxy	• Chain of Responsibility • Command • Iterator • Mediator • Template Method • Memento • Observer • State • Strategy • Visitor

Behavioral Design Pattern

- Behavioral patterns are *concerned with algorithms and the assignment of responsibilities* between objects.

- Behavioral patterns describe not just patterns of objects or classes but also the *patterns of communication* between them. These patterns characterize *complex control* flow that's difficult to follow at run-time. They shift your focus away from flow of control to let you concentrate just on the way objects are interconnected.

Let follow a paper…

- **Non-Software Examples of Software Design Patterns,** by Michael Duell, in AG Communication Systems e-zine:

  http://www2.ing.puc.cl/~jnavon/IIC2142/patexamples.htm

The 11 Behavioral Design Pattern 1/3

Behavioral class patterns use inheritance to distribute behavior between classes:

- **Template Method** (GoF325) - concerns an abstract definition of an algorithm; defines the algorithm step by step, where each step invokes either an abstract operation or a primitive operation. A subclass fleshes out the algorithm by defining the abstract operations.
- **Interpreter** (GoF243) - represents a grammar as a class hierarchy and implements an interpreter as an operation on instances of these classes.
- **Memento** (GoF381) - without violating encapsulation, captures and externalizes an object's internal state so that the object can be restored to this state later

11 Behavioral Design Pattern 2/3

Behavioral object patterns use object composition rather than inheritance. Some describe how a group of peer objects cooperate to perform a task that no single object can carry out by itself.

An important issue here is ***how peer objects know about each other***. Peers could maintain explicit references to each other, but that would increase their coupling. In the extreme, every object would know about every other.

- **Mediator** (GoF273) - avoids this by introducing a mediator object between peers; provides the indirection needed for loose coupling.
- **Chain of Responsibility** (GoF223) - provides even looser coupling. It lets you send requests to an object implicitly through a chain of candidate objects. Any candidate may fulfill the request depending on runtime conditions. The number of candidates is open-ended, and you can select which candidates participate in the chain at run-time.
- **Observer** (GoF293) - pattern defines and maintains a dependency between objects. The classic example of *Observer is in Smalltalk Model/View/Controller*, where all views of the model are notified whenever the model's state changes.

11 Behavioral Design Pattern 3/3

Other behavioral object patterns are concerned with ***encapsulating behavior in an object and delegating requests*** to it:

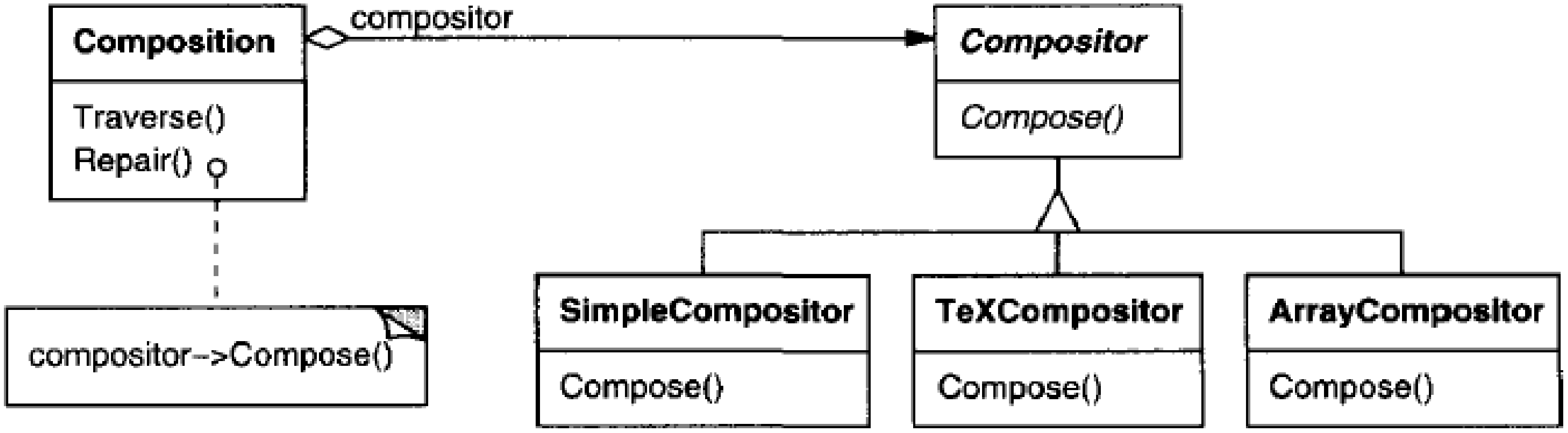
- **Strategy** (GoF315) - encapsulates an algorithm in an object. Strategy makes it easy to specify and change the algorithm an object uses;
- **Command** (GoF233) - encapsulates a request in an object so that it can be passed as a parameter, stored on a history list, or manipulated in other ways.
- **State** (GoF305) - encapsulates the states of an object so that the object can change its behavior when its state object changes;
- **Visitor** (GoF331) encapsulates behavior that would otherwise be distributed across classes;
- **Iterator** (GoF257) - abstracts the way you access and traverse objects in an aggregate.

The Strategy Pattern [1]

- **Intent :**
  - defines a *family of algorithms*,
  - encapsulate each one, and
  - make them *interchangeable*.
    Strategy lets us to vary the algorithm independently from clients that use it.
- **Also Known As -** Policy
- **Motivation -** many algorithms exist <u>*for breaking a stream of text into lines*</u>. Hard-wiring all such algorithms into the classes that require them isn't desirable for several reasons:
  - **Clients that need line breaking get more complex if they include this code. That makes clients *bigger* and <u>harder to maintain</u>, especially if they support multiple line breaking algorithms.**
  - **Different algorithms will be appropriate at different times. We don't want to support multiple line-breaking algorithms if we don't use them all.**
  - **It's difficult to add new algorithms and vary existing ones when line-breaking is an integral part of a client.**

The Solution

- We can avoid these problems by defining classes that encapsulate different line-breaking algorithms. An algorithm that's encapsulated in this way is called a **strategy.**
- The Composition class will be responsible for maintaining and updating the line-breaks of text displayed in a text viewer. Line-breaking strategies are implemented separately by subclasses of an ***abstract Compositor*** class, for implementing different strategies:
  - **SimpleCompositor** implements a simple strategy that determines line-breaks one at a time.
  - **TeXCompositor** implements the TeX algorithm for finding line-breaks. This strategy tries to optimize line-breaks globally, that is, one paragraph at a time.
  - **ArrayCompositor** implements a strategy that selects breaks so that each row has a fixed number of items. It's useful for breaking a collection of icons into rows, for example.



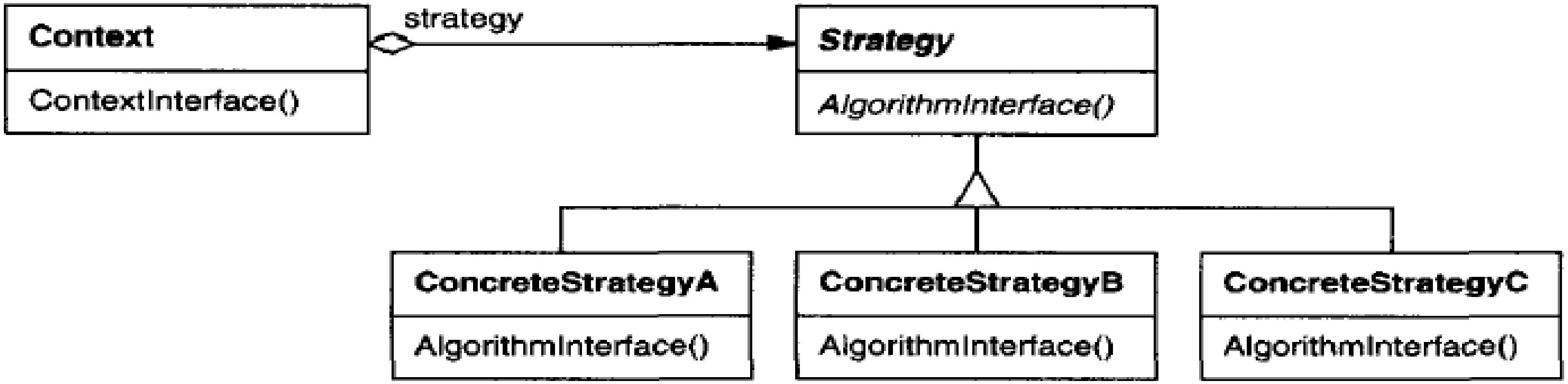
- A Composition maintains a reference to a Compositor object.
- Whenever a Composition reformats its text, it forwards this responsibility to its Compositor object.
- *The client of Composition specifies which Compositor should be used by installing the Compositor* it desires into the Composition.

Applicability

Use the Strategy pattern when:

- many related classes differ only in their behavior. Strategies provide *a way to configure a class with one of many behaviors*.
- you need *different variants of an algorithm*. For example, you might define algorithms reflecting different space/time trade-offs (i.e., DOM vs. SAX/StaX). Strategies can be used when these variants are implemented as a class hierarchy of algorithms.
- an algorithm uses data that clients shouldn't know about. Use the Strategy pattern *to avoid exposing complex, algorithm-specific data structures*.
- a class defines many behaviors, and these appear as multiple conditional statements in its operations. *Instead of many conditionals*, move related conditional branches into their own Strategy class.

Structure and Participants



- **Strategy** (Compositor)
  - - declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.
- **ConcreteStrategy** (SimpleCompositor, etc.)
  - - implements the algorithm using the Strategy interface.
- **Context** (Composition)
  - - is configured with a ConcreteStrategy object.
  - - maintains a reference to a Strategy object.
  - - ***may define an interface that lets Strategy access its data***.

Collaborations

- Strategy and Context interact to implement the chosen algorithm.
  - A context may pass all data required by the algorithm to the strategy when the algorithm is called.
  - Alternatively, *the context can pass itself as an argument to Strategy operations*. That lets the strategy call back on the context as required.

- A context forwards requests from its clients to its strategy.
  - Clients usually create and pass a ConcreteStrategy object to the context; thereafter, clients interact with the context exclusively.
  - There is often a family of ConcreteStrategy classes for a client to choose from.

Consequences ++

- *Families of related algorithms.* Hierarchies of Strategy classes define a family of algorithms or <u>*behaviors for contexts to reuse*</u>. Inheritance can help factor out common functionality of the algorithms.
- *An alternative to subclassing the Context (<u>it is another abstraction</u>).* Inheritance offers another way to support a variety of algorithms or behaviors. Instead of subclassing a Context class (<u>*which mixes the algorithm implementation with Context's, making Context harder to understand, maintain, and extend*</u>) we encapsulate the algorithm in separate Strategy classes lets you vary the algorithm independently of its context.
- *Strategies eliminate conditional statements.* The Strategy pattern offers an alternative to conditional statements for selecting desired behavior. Otherwise, when different behaviors are lumped into one class, it's hard to avoid conditionals.
- *A different choice of implementations.* Strategies can provide <u>*different implementations of the same behavior*</u>. The client can choose among strategies with <u>*different time and space trade-offs*</u>.

Consequences --

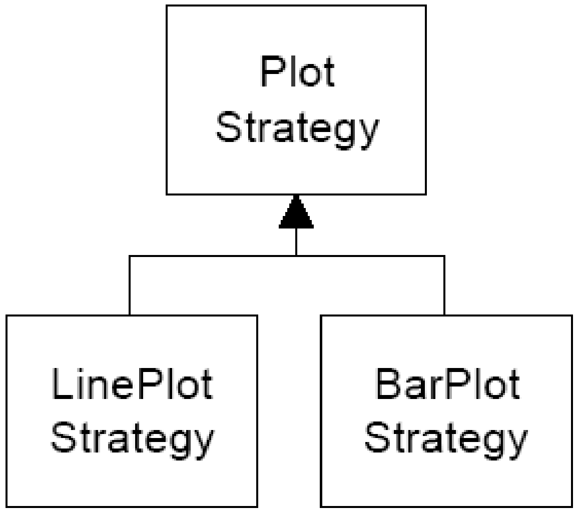
- *Clients must be aware of different Strategies.* The pattern has a potential drawback in that *a client must understand how Strategies differ before it can select the appropriate one*. Clients might be exposed to implementation issues.
- *Communication overhead between Strategy and Context.* The *Strategy interface is shared by all ConcreteStrategy* classes whether the algorithms they implement are trivial or complex. Hence it's likely that some ConcreteStrategies won't use all the information passed to them through this interface; *simple ConcreteStrategies may use none of it*!
- *Increased number of objects.* Strategies increase the number of objects in an application. Sometimes you can *reduce this overhead by implementing strategies as stateless objects* that contexts can share. Any residual state is maintained by the context, which passes it in each request to the Strategy object.

Implementation

- *Defining the Strategy and Context interfaces.* The Strategy and Context interfaces must give a ConcreteStrategy efficient access to any data it needs from a context, and vice versa.
  - One approach is to have *Context pass data in parameters to Strategy operations* - in other words, bring the data to the strategy. This keeps Strategy and Context *decoupled*. Or, a context passes itself as an argument, and the strategy requests data from the context explicitly.
  - Alternatively, the strategy can store a reference to its context, eliminating the need to pass anything at all. *Strategies as template parameters.* In C++ templates can be used to configure a class with a strategy.
- *Making Strategy objects optional.* The Context class may be simplified if it's meaningful *not* to have a Strategy object. Context checks to see if it has a Strategy object before accessing it. If there is one, then Context uses it normally. *If there isn't a strategy, then Context carries out default behavior*. The benefit of this approach is that clients don't have to deal with Strategy objects at all *unless they don't like the default behavior*.

Sample Java Code [3]

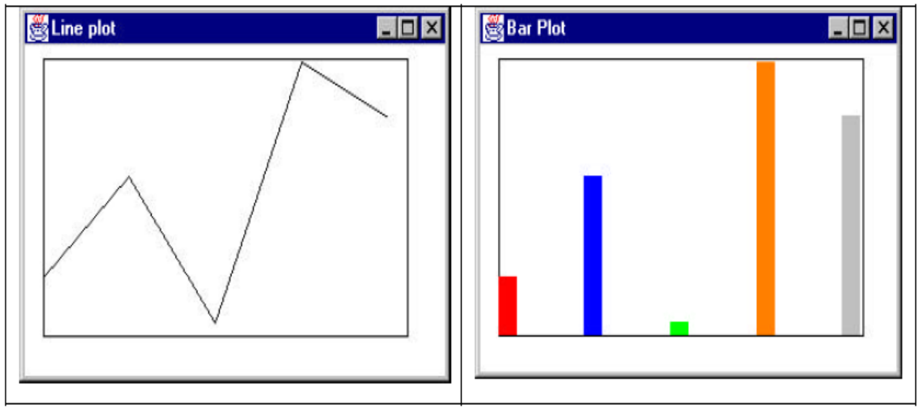
- Let’s consider a simplified graphing program that can present data as a line graph or a bar chart. We’ll start with an abstract PlotStrategy class and derive the two plotting classes from it
- Since each plot will appear in its own frame, our base PlotStrategy class will be derived from JFrame



```
public abstract class PlotStrategy extends JFrame {
        protected float[] x, y;
        protected Color color;
        protected int width, height;
        public PlotStrategy(String title) {
                super(title);
                width = 300; height =200;
                color = Color.black;
                addWindowListener(new WindAp(this));
        }
        public abstract void plot(float xp[], float yp[]);
        public void setPenColor(Color c)   {color = c;}

// we add a WindowAdapter class that just hides the window if it is closed.
class WindAp extends WindowAdapter {
        JFrame fr;
        public WindAp(JFrame f) {
                fr = f; //copy Jframe instance
        }
        public void WindowClosing(WindowEvent e) {
                fr.setVisible(false); //hide window
        }
}
```

```
//the Context needs to set a variable to refer to one concrete strategy or another.
public class Context {
    //this object selects one of the strategies
    //to be used for plotting
    private PlotStrategy plotStrategy; //points to selected strategy
    float x[], y[]; //data stored here

    //---------------------------------
    public Context() {
        setLinePlot(); //make sure it is not null
    }

    //make current strategy the Bar Plot
    public void setBarPlot()        { plotStrategy = new BarPlotStrategy(); }
    //---------------------------------
    //make current strategy the Line Plot
    public void setLinePlot()       { plotStrategy = new LinePlotStrategy(); }
    //---------------------------------
    //call plot method of current strategy
    public void plot()              {plotStrategy.plot(x, y);}
    public void setPenColor(Color c)     {plotStrategy.setPenColor(c);}
    public void readData(String filename) { //read data from datafile
    } }
```

```
public class LinePlotStrategy extends PlotStrategy {
    LinePlotPanel lp;
    public LinePlotStrategy() {
        super("Line plot");
        lp = new LinePlotPanel();
        getContentPane().add(lp);
    }
    //----------------------------------------
    public void plot(float[] xp, float[] yp) {
        x = xp; y = yp; //copy in data
        findBounds(); //sets maxes and mins
        setSize(width, height);
        setVisible(true);
        setBackground(Color.white);
        lp.setBounds(minX, minY, maxX, maxY);
        lp.plot(xp, yp, color); //set up plot data
        repaint(); //call paint to plot
    }
}
```

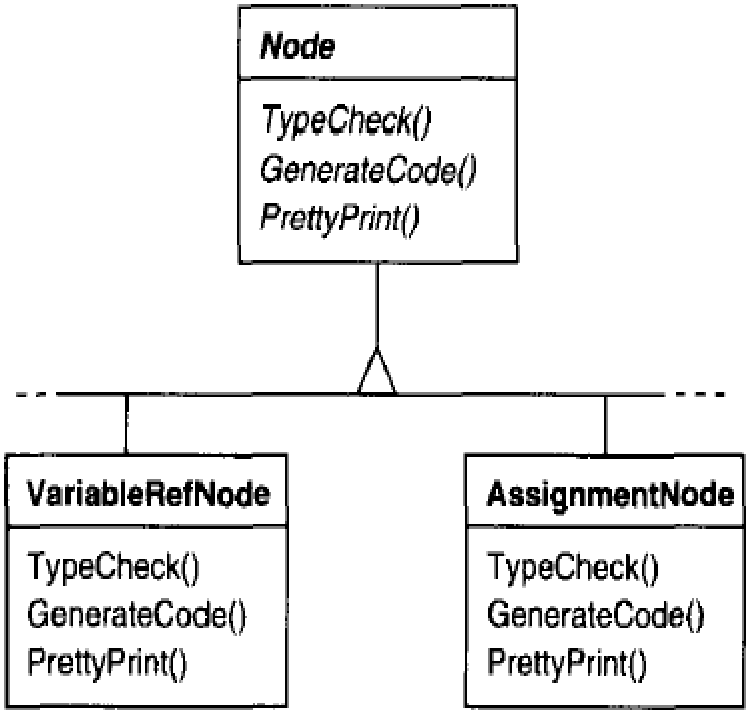


Visitor

- **Intent:**
  - to represent ***an operation to be performed on the elements of an object structure***.
  - Visitor lets you ***define a new operation without changing the classes of the elements on which it operates***.
- **Motivation** - consider a compiler that represents programs as abstract syntax trees. It will need to perform operations on abstract syntax trees for "static semantic" analyses like checking that all variables are defined. It will also need to generate code. So it might define operations for type-checking, code optimization, flow analysis, etc. Moreover, we could use the abstract syntax trees for pretty-printing, program restructuring, code instrumentation, and computing various metrics of a program.
- Most of these operations will need to treat nodes that represent assignment statements differently from nodes that represent variables or arithmetic expressions. *Hence there will be one class for assignment statements, another for variable accesses, another for arithmetic expressions, and so on*. The set of node classes depends on the language being compiled, of course, but it doesn't change much for a given language.

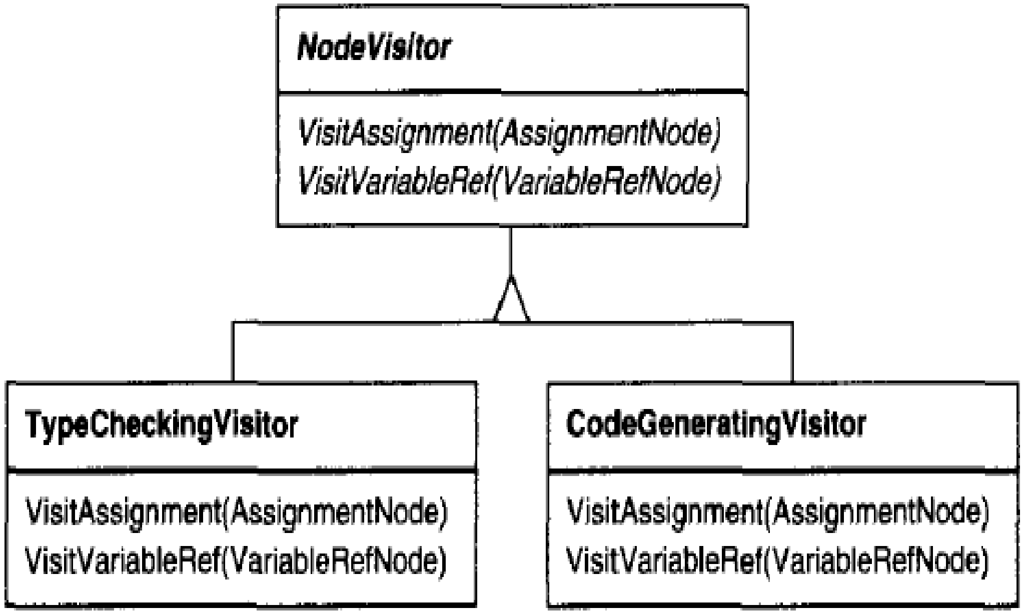
The Problem

- The problem here is that distributing all these operations across the various node classes leads to a system that's ***hard to understand, maintain, and change***. It will be <u>*confusing to have type-checking code mixed with pretty-printing code or flow analysis code*</u>.
- Moreover, adding a new operation usually requires recompiling all of these classes. It would *be* better if ***each new operation could be added separately, and the node classes were independent of the operations that apply to them***.

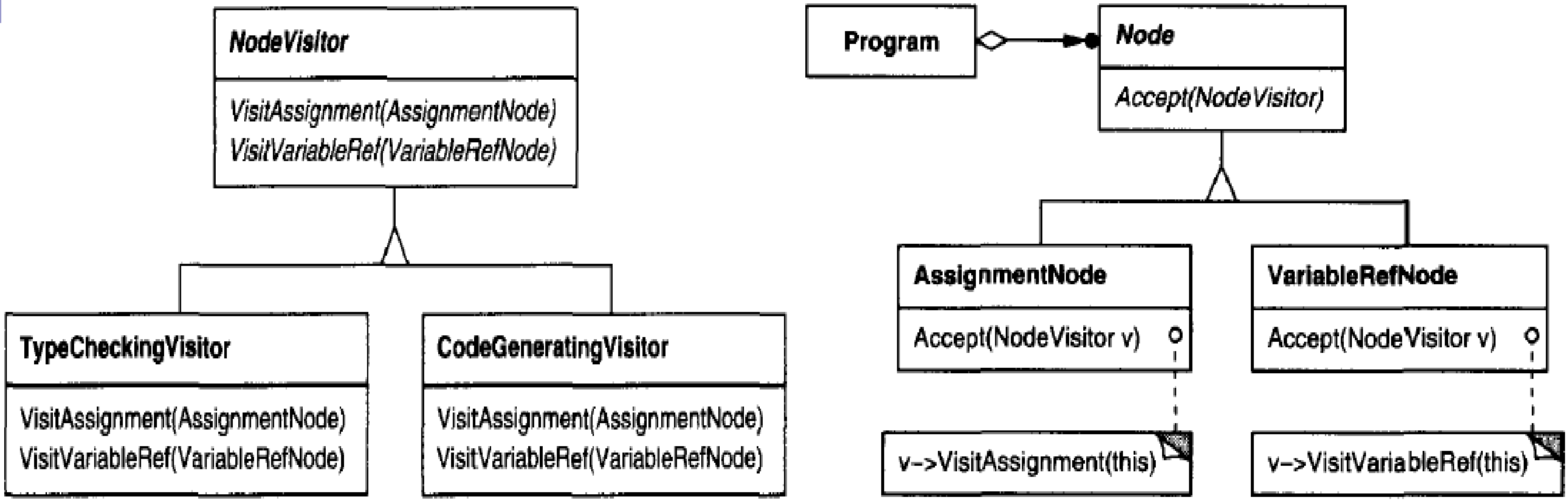


Solution

- We can have *package related operations from each class in a separate object*, called a **visitor,** and passing it to elements of the abstract syntax tree as it's traversed.
- When an element "accepts" the visitor, it sends a request to the visitor that encodes the element's class. It also includes the element as an argument. The visitor will then execute the operation for that element—the operation that used to be in the class of the element.


NodeVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)
TypeCheckingVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)
CodeGeneratingVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)

If the compiler type-checked a procedure using visitors, then it would create a TypeChecking-Visitor object and call the Accept operation on the abstract syntax tree with that object as an argument. *Each of the nodes would implement Accept by calling back on the visitor: an assignment node calls VisitAssignment operation on the visitor, while a variable reference calls VisitVariableReference*. What used to be the TypeCheck operation in class AssignmentNode is now the VisitAssignment operation on TypeCheckingVisitor.


NodeVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)
TypeCheckingVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)
CodeGeneratingVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)
Program
Node
Accept(NodeVisitor)
AssignmentNode
Accept(NodeVisitor v)
VariableRefNode
Accept(NodeVisitor v)
v->VisitAssignment(this)
v->VisitVariableRef(this)

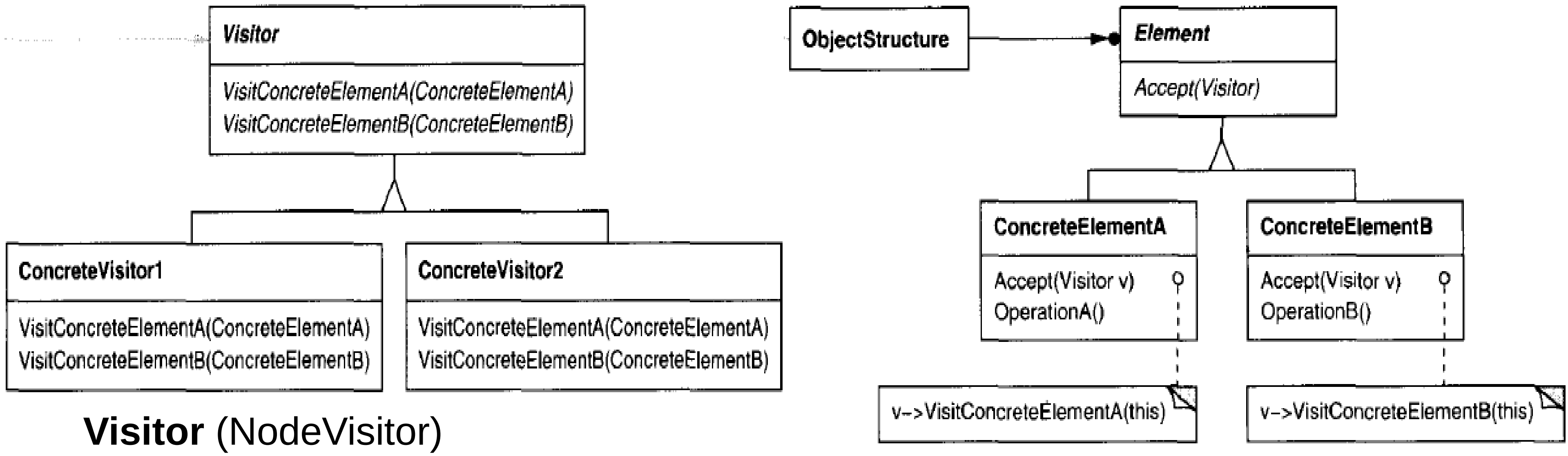
With the Visitor pattern, you define two class hierarchies: one for the elements being operated on (the Node hierarchy) and one for the visitors that define operations on the elements (the NodeVisitor hierarchy). *You create a new operation by adding a new subclass to the visitor class hierarchy*. As long as the grammar that the compiler accepts doesn't change (that is, we don't have to add new Node subclasses), we can add new functionality simply by defining new Node Visitor subclasses.

Applicability

Use the Visitor pattern when:

- an *<u>object structure contains many classes of objects with different interfaces</u>*, and you want *<u>to perform operations on these objects that depend on their concrete classes</u>*.
- many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" their classes with these operations. Visitor lets you *<u>keep related operations together by defining them in one class</u>*. When the object structure is shared by many applications, use Visitor to put operations in just those applications that need them.
- the classes defining the object structure *<u>rarely change</u>*, but you often want *<u>to define new operations over the structure</u>*. Changing the object structure classes requires redefining the interface to all visitors, which is potentially *<u>costly</u>*. If the object structure classes change often, then it's probably better to define the operations in those classes.

Structure and Participants

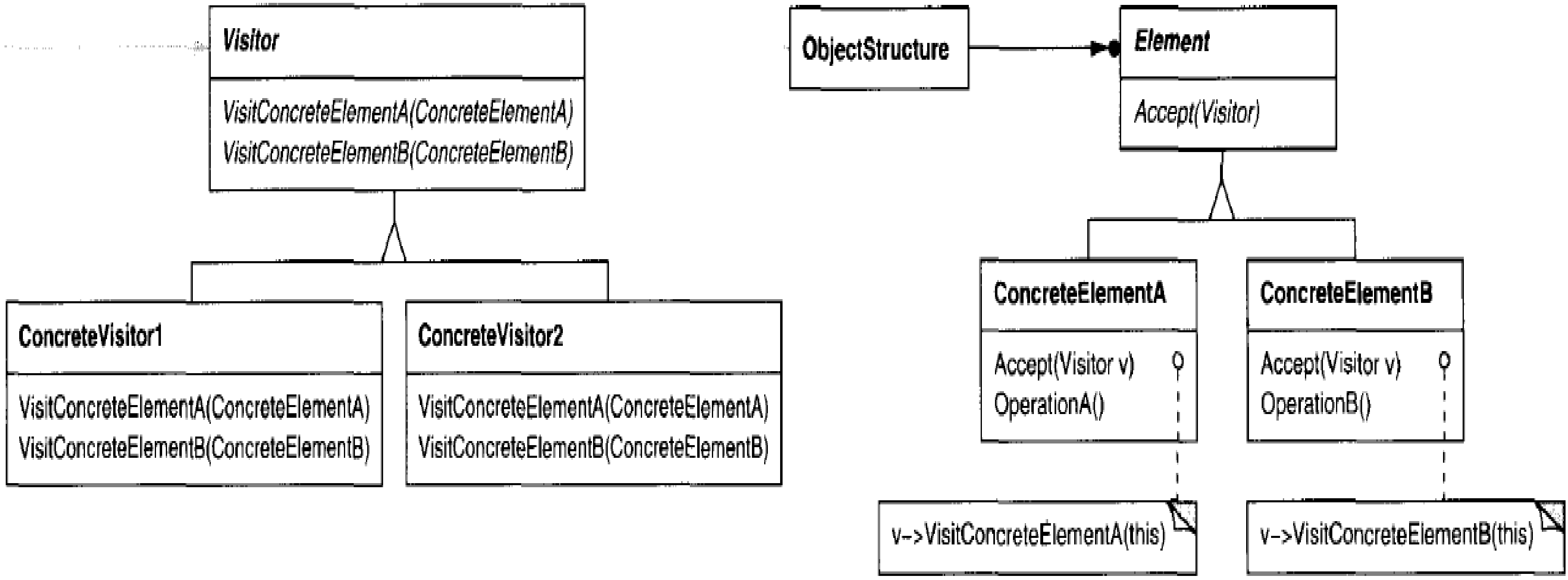

Visitor
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor1
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor2
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ObjectStructure
Element
Accept(Visitor)
ConcreteElementA
Accept(Visitor v)
OperationA()
ConcreteElementB
Accept(Visitor v)
OperationB()
v->VisitConcreteElementA(this)
v->VisitConcreteElementB(this)

Visitor (NodeVisitor)
- declares a Visit operation for each class of ConcreteElement in the object structure. *The operation's name and signature identifies the class that sends the Visit request to the visitor*. That lets the visitor determine the concrete class of the element being visited. Then the visitor can access the element directly through its particular interface.

• **ConcreteVisitor** (TypeCheckingVisitor)
- implements each operation declared by Visitor. *Each operation implements a fragment of the algorithm defined for the corresponding class*, of object in the structure. ConcreteVisitor provides the context for the algorithm and stores its local state. This *state often accumulates results during the traversal* of the structure.

• ***Element*** (*Node*)
- defines an Accept operation that takes a visitor as an argument.

Structure and Participants (cont.)


Visitor
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor1
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor2
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ObjectStructure
Element
Accept(Visitor)
ConcreteElementA
Accept(Visitor v)
OperationA()
ConcreteElementB
Accept(Visitor v)
OperationB()
v->VisitConcreteElementA(this)
v->VisitConcreteElementB(this)

- **ConcreteElement** (AssignmentNode,VariableRefNode)

- implements an Accept operation that takes a visitor as an argument.

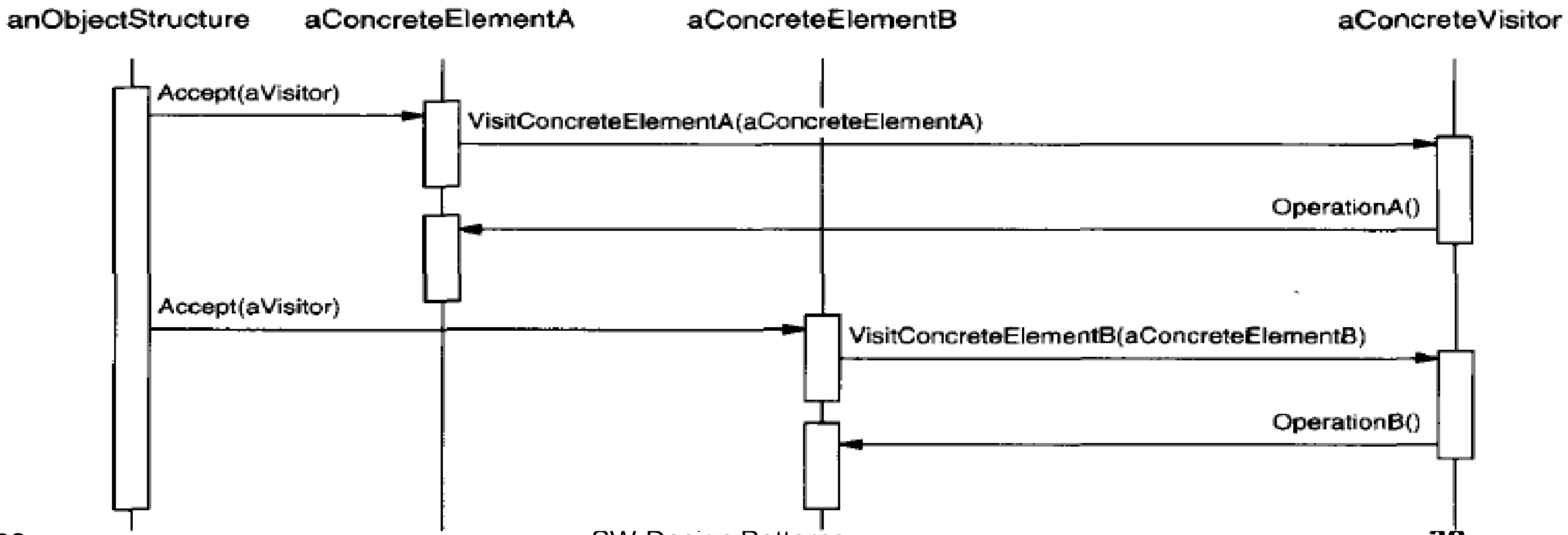
- **ObjectStructure** (Program)

- can enumerate its elements.
- may provide a high-level interface to allow the visitor to visit its elements.
- may either be a composite (GoF163) or a collection such as a list or a set.

Collaborations

1. A client that uses the Visitor pattern must create a Concrete Visitor object and then traverse the object structure, visiting each element with the visitor.
2. When an element is visited, it calls the Visitor operation that corresponds to its class.
3. The element supplies itself as an argument to this operation to let the visitor access its state, if necessary.

The interaction diagram below illustrates the collaborations between an object structure, a visitor, and two elements



Consequences

- ++*Visitor makes adding new operations easy* - to add operations that depend on the components of complex objects. You can define a new operation over an object structure simply by adding a new visitor.
- ++ *A visitor gathers related operations and separates unrelated ones.* Related behavior isn't spread over the classes defining the object structure; it's localized in a visitor.
- -- *Adding new ConcreteElement classes is hard* - adding new subclasses of Element gives rise to a new abstract operation on Visitor and a corresponding implementation in every ConcreteVisitor class. Sometimes a default implementation can be provided in Visitor that can be inherited by most of the Concrete Visitors.
- *Visiting across class hierarchies.* An iterator (GoF257) can visit the objects in a structure as it traverses them by calling their operations.
- ++ *Accumulating state.* Visitors can accumulate state as they visit each element in the object structure.

Implementation

- *Each object structure has an associated Visitor class*. This abstract visitor class declares a VisitConcreteElement operation for each class of ConcreteElement defining the object structure.
- Each Visit operation on the Visitor declares its argument to be a particular ConcreteElement, *allowing the Visitor to access the interface of the ConcreteElement directly*.
- *Concrete Visitor classes override each Visit operation to implement visitor-specific behavior for the corresponding ConcreteElement class*.

- *Who is responsible for traversing the object structure? A* visitor must visit each element of the object structure. The question is, how does it get there? We can put responsibility for traversal in any of three places: **1)** in the object structure, **2)** in the visitor, or, **3)** in a separate iterator object (see Iterator (GoF257)).

Sample Java Example [1]

```
//let have a simple Employee object which maintains a record of the employee's –
//name, salary, vacation taken and number of sick days taken.

public class Employee {
    int sickDays, vacDays;
    float Salary;
    String Name;
    public Employee(String name, float salary, int vacdays, int
    sickdays) {
        vacDays = vacdays; sickDays = sickdays;
        Salary = salary; Name = name;
    }
    public String getName() { return Name; }
    public int getSickdays() { return sickDays; }
    public int getVacDays() { return vacDays; }
    public float getSalary() { return Salary; }
    public void accept(Visitor v) { v.visit(this); }
}
```

```
//we want to prepare a report of the number of vacation days that
//all employees have taken so far this year. We could just write some code in
//the client to sum the results of calls to each Employee's getVacDays function,
//or we could put this function into a Visitor.
//the base Visitor class needs to have a suitable abstract visit method for
//each kind of class in our program.
public abstract class Visitor {
            public abstract void visit(Employee emp);
}
public class VacationVisitor extends Visitor {
      protected int total_days;
      public VacationVisitor() { total_days = 0; }
      //----------------------------
      public void visit(Employee emp) {
            total_days += emp.getVacDays();
      }
      //----------------------------
      public int getTotalDays() {
            return total_days;
      }
}
```

Now, all we have to do to compute the total vacation taken is to go through a list of the employees and visit each of them, and then ask the Visitor for the total.

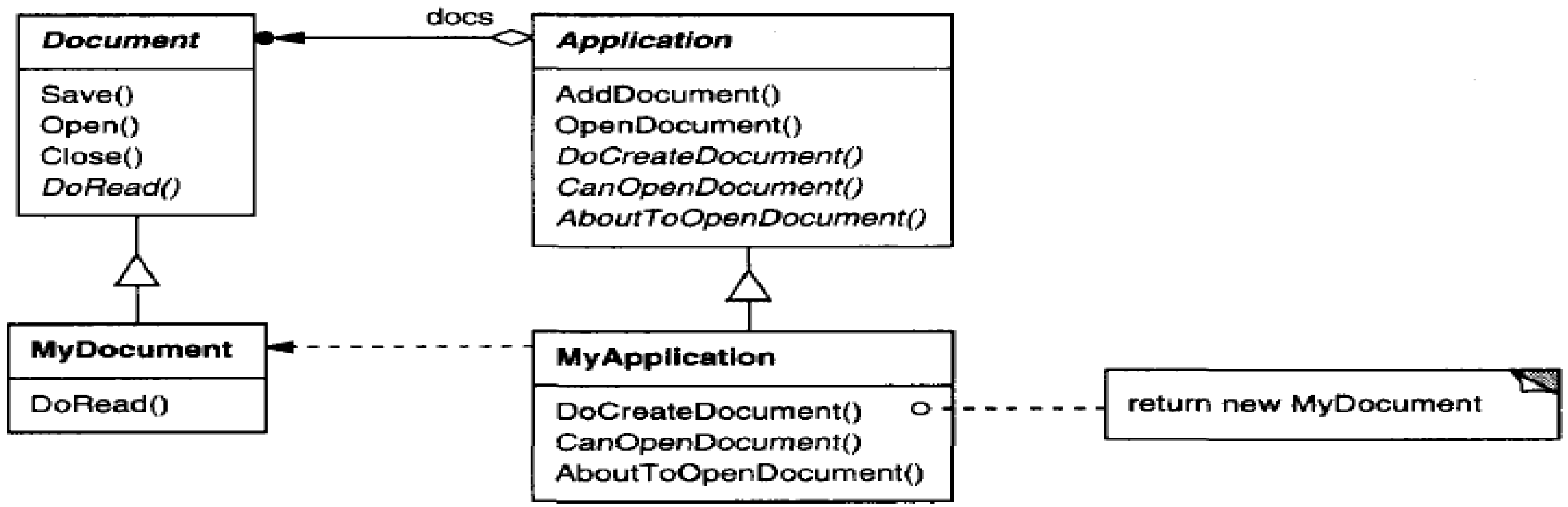
```
VacationVisitor vac = new VacationVisitor();
for (int i = 0; i < employees.length; i++) {
        employees[i].accept(vac);
}
System.out.println(vac.getTotalDays());
```

Let's re-iterate what happens for each visit:

1. We move through a loop of all the Employees.
2. The Visitor calls each Employee's *accept* method.
3. That instance of Employee calls the Visitor's *visit* method.
4. The Visitor fetches the vacation days and adds them into the total.
5. The main program prints out the total when the loop is complete.

Template Method [1]

- **Intent** - to define the *skeleton of an algorithm in an operation*, deferring some steps to subclasses. Template Method lets *subclasses redefine certain steps of an algorithm without changing the algorithm's structure*.
- **Motivation** - consider an application framework that provides Application and Document classes. The Application class is responsible for opening existing documents stored in an external format, such as a file. A Document object represents the information in a document once it's read from the file.
- Applications built with the framework can subclass Application and Document to suit specific needs. For example, a drawing application defines DrawApplication and DrawDocument subclasses; a spreadsheet application defines Spreadsheet-Application and SpreadsheetDocument subclasses.



The abstract Application class defines the algorithm for opening and reading a document in its OpenDocument operation:

```
void Application::OpenDocument (const char* name) {
    if(! CanOpenDocument(name)) {
            // cannot handle this document
            return;
    }
    Document* doc = DoCreateDocument();
    if (doc) {
            _docs->AddDocument(doc) ;
            AboutToOpenDocument(doc) ;
            doc->Open() ;
            doc->DoRead() ;
    }
}
```

By defining some of the steps of an algorithm using abstract operations, the ***template method fixes their ordering****, but it* ***lets Application and Document subclasses vary those steps to suit their needs****.*

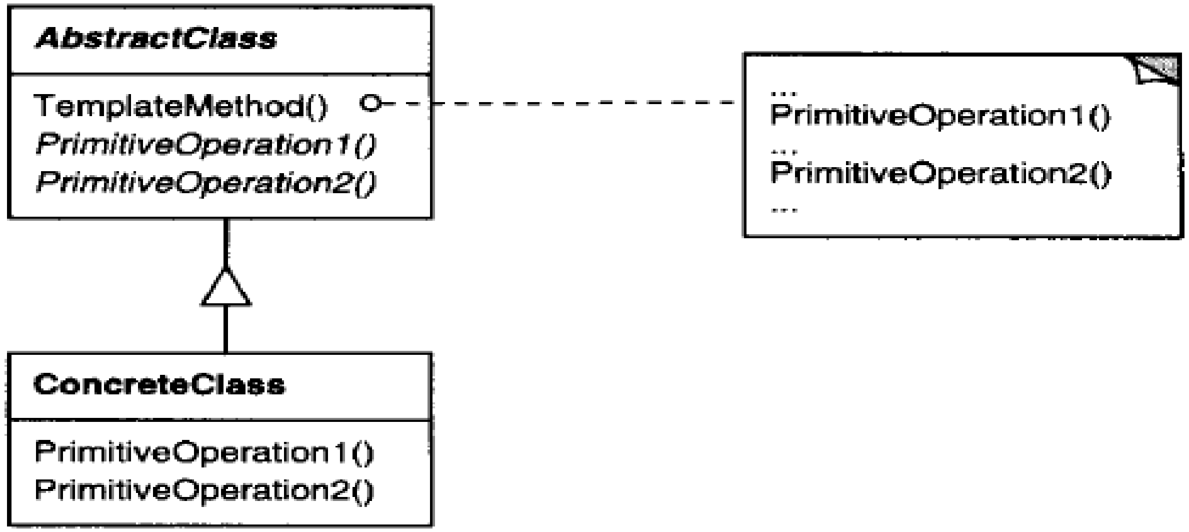
We call OpenDocument a **template method.** A template method defines an algorithm in terms of *abstract operations that subclasses override to provide concrete behavior*. Application *subclasses define the steps of the algorithm* that check if the document can be opened (CanOpenDocument) etc. Document classes define the step that reads the document (DoRead).

Applicability

The Template Method pattern should be used:

- to implement the *invariant parts of an algorithm* once and leave it up to subclasses to implement the behavior that can vary.
- when *common behavior among subclasses should be factored and localized in a common class to avoid code duplication*. This is a good example of "***refactoring to generalize***“. You should:
  - **first** identify the differences in the existing code;
  - **then** separate the differences into new operations;
  - **finally**, you replace the differing code with a template method that calls one of these new operations.
- to *control subclasses extensions*. You can define a template method that calls "hook“ operations (see Consequences) at specific points, thereby permitting extensions only at those points.

Structure and Participants



- **AbstractClass** (Application)
  - - defines abstract **primitive operations** that concrete subclasses define to implement steps of an algorithm.
  - - implements a **template method** defining the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in AbstractClass or those of other objects.
- **ConcreteClass** (MyApplication)
  - - implements the primitive operations to carry out subclass-specific steps of the algorithm.
  - - relies on AbstractClass to implement invariant steps of the algorithm.

Consequences

- Template methods are a fundamental technique for code reuse. They are particularly important in class libraries, because they are the *means for factoring out common behavior in library classes*.
- Template methods lead to an *inverted control structure*. This refers to how a parent class calls the operations of a subclass and not the other way around.
- Template methods call the following kinds of operations:
  - concrete operations (either on the ConcreteClass or on client classes);
  - concrete AbstractClass operations (i.e., operations that are generally useful to subclasses);
  - primitive operations (i.e., abstract operations);
  - factory methods (see Factory Method (GoF107)); and
  - **hook operations,** which provide default behavior that subclasses can extend if necessary. A hook operation often does nothing by default.

Implementation

- *Using C++/Java access control -* the primitive operations that a template method calls can be declared as **protected**. This ensures that they are only called by the template method.
- *Minimizing primitive operations.* An important goal in designing template methods is to minimize the number of primitive operations that a subclass must override to flesh out the algorithm. The more operations that need overriding, the more tedious things get for clients.
- *Naming conventions.* You can identify the operations that should be overridden by adding a prefix to their names. Use prefixes template method names with *"Do-":* "DoCreateDocument", "DoRead", etc.

Sample Java Example [3]

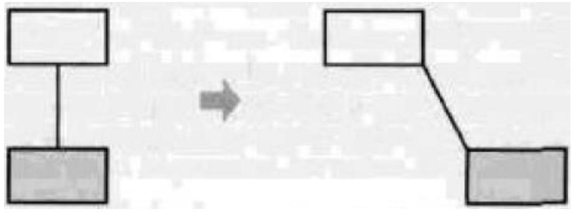
Let’s consider a simple program for drawing triangles on a screen. We’ll start with an abstract Triangle class, and then derive some special triangle types from it.

```
public abstract class Triangle {
        Point p1, p2, p3;
        public Triangle(Point a, Point b, Point c) { //save
                p1 = a; p2 = b; p3 = c;
        }
        public void draw(Graphics g) { //This routine draws a general triangle
                drawLine(g, p1, p2);
                Point current = draw2ndLine(g, p2, p3);
                closeTriangle(g, current);
        }
        public void drawLine(Graphics g, Point a, Point b) {
                g.drawLine(a.x, a.y, b.x, b.y);
        }
        //this routine has to be implemented for each triangle type.
        abstract public Point draw2ndLine(Graphics g, Point a, Point b);
        public void closeTriangle(Graphics g, Point c) { //draw back to first point
                g.drawLine(c.x, c.y, p1.x, p1.y);
        }
}
```

```
//To draw a standard, general triangle with no restrictions on its shape, we
//simple implement the draw2ndLine method in a derived stdTriangle class:
public class stdTriangle extends Triangle {
      public stdTriangle(Point a, Point b, Point c) {
            super(a, b, c);
      }
      public Point draw2ndLine(Graphics g, Point a, Point b) {
            g.drawLine(a.x, a.y, b.x, b.y);
            return b;
      }
}
//Drawing an Isosceles Triangle
public class IsoscelesTriangle extends Triangle {
      Point newc; int newcx, newcy; int incr;
      public IsoscelesTriangle(Point a, Point b, Point c) {
            super(a, b, c);
            … //calculate newc
      }
                        //draws 2nd line using saved new point
                        public Point draw2ndLine(Graphics g, Point b, Point c){
                              g.drawLine(b.x, b.y, newc.x, newc.y);
                              return newc;
                        }
```

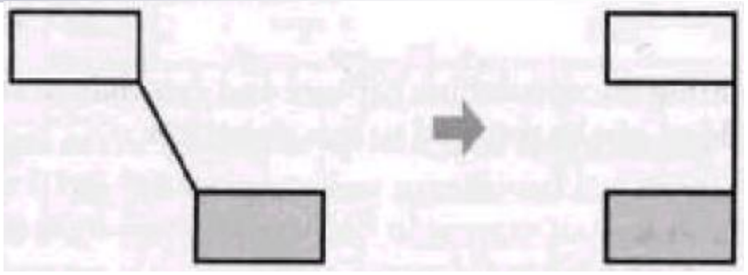
The Memento Pattern [1]

- **Intent -** without violating encapsulation, captures and externalizes an object's internal state so that the object can be restored to this state later.
- **Also Known As -** Token
- **Motivation –** sometimes it's necessary *to record the internal state of an object*. This is required when *implementing checkpoints and undo mechanisms* that let users back out of tentative operations or recover from errors. You must save state information somewhere so that you can restore objects to their previous states. But *objects normally encapsulate some or all of their state*, making it inaccessible to other objects and impossible to save externally. Exposing this state would *violate encapsulation*, which can compromise the application's reliability and extensibility.



Consider for example a graphical editor that supports connectivity between objects. A user can connect two rectangles with a line, and the rectangles stay connected when the user moves either of them.

Motivating Problem



- Usually, a ConstraintSolver object records connections as they are made and generates mathematical equations that describe them. It solves these equations whenever the user makes a connection or otherwise modifies the diagram. ConstraintSolver uses the results of its calculations to rearrange the graphics so that they maintain the proper connections.
- Supporting undo in this application isn't as easy as it may seem. In general, the ConstraintSolver's public interface might be insufficient to allow precise reversal of its effects on other objects. <u>*The undo mechanism must work more closely with ConstraintSolver to reestablish previous state, but we should also avoid exposing the ConstraintSolver's internals to the undo mechanism*</u>.

Solution 1/2

- We can solve this problem with the Memento pattern. A **memento** is an object that stores a snapshot of the internal state of another object—the memento's **originator.**

- The undo mechanism will request a memento from the originator when it needs to checkpoint the originator's state. The originator initializes the memento with information that characterizes its current state.

- Only the originator can store and retrieve information from the memento - the memento is "opaque" to other objects.

Solution 2/2

- In the graphical editor example just discussed, the ConstraintSolver can act as an originator. The following sequence of events characterizes the undo process:
  - **The editor requests a memento from the ConstraintSolver as a side-effect of the move operation. The ConstraintSolver creates and returns a memento, an instance of a class SolverState in this case.**
  - **A SolverState memento contains data structures that describe the current state of the ConstraintSolver's internal equations and variables. Later when the user undoes the move operation, the editor gives the SolverState back to the ConstraintSolver.**
  - **Based on the information in the SolverState, the ConstraintSolver changes its internal structures to return its equations and variables to their exact previous state.**

Applicability

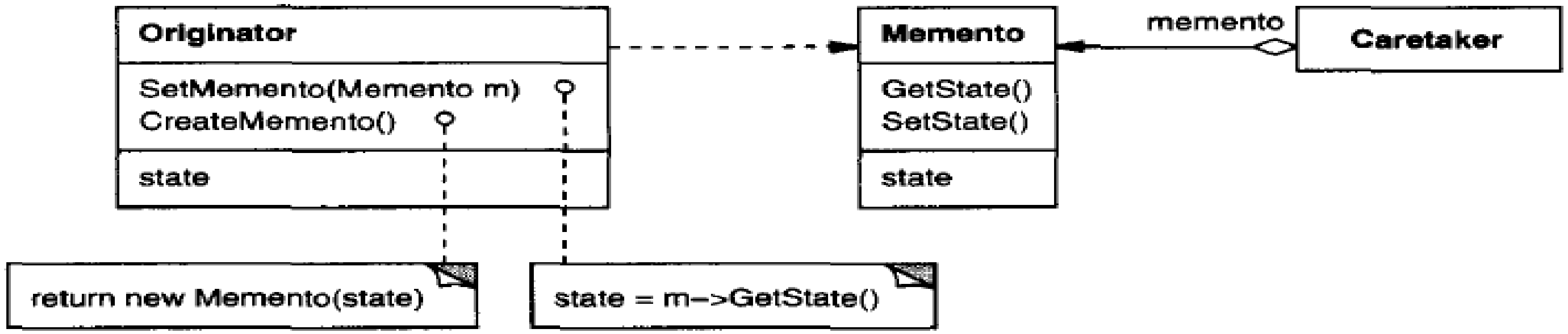
Use the Memento pattern when

- a snapshot of (some portion of) an object's state must be saved so that it can be restored to that state later,

    and

- usage of a direct interface to obtaining the state would expose implementation details and would break the object's encapsulation.

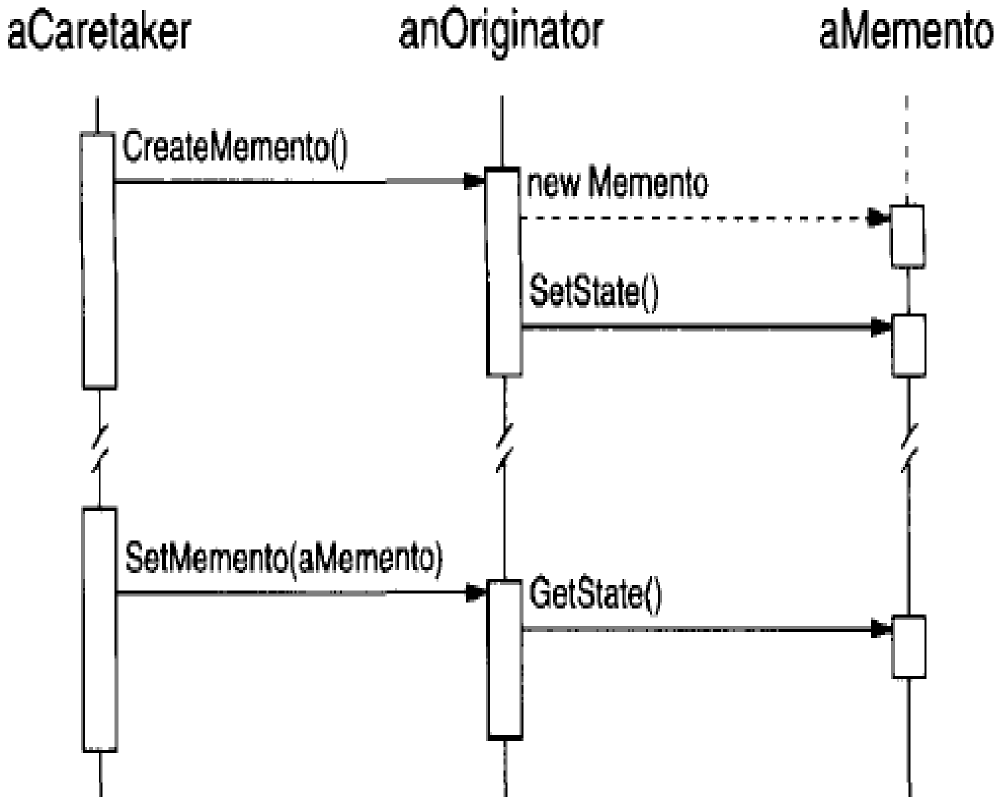
Structure and Participants



- **Memento** (SolverState)
  - - stores internal state of the Originator object. The memento may store as much or as little of the originator's internal state as necessary for its originator.
  - - protects against access by objects other than the originator. Mementos have effectively two interfaces. *Caretaker sees a narrow interface to the Memento*—it can *only pass the memento to other objects*. *Originator sees a wide interface*, one that lets it *access all the data necessary to restore itself to its previous state*. Ideally, only the originator can access the memento's internal state.
- **Originator** (ConstraintSolver)
  - - creates a memento containing a snapshot of its current internal state.
  - - uses the memento to restore its internal state.
- **Caretaker** (undo mechanism)
  - - is responsible for the memento's safekeeping.
  - - never operates on *or* examines the contents of a memento.

Collaborations

- A caretaker requests a memento from an originator, holds it for a time, and passes it back to the originator, as the interaction diagram illustrates ->
- Sometimes the caretaker won't pass the memento back to the originator, because *the originator might never need to revert to an earlier state*.
- Mementos are *passive*. Only the originator that created a memento will *assign or retrieve its state*.



Consequences ++

- ++ *Preserving encapsulation boundaries.* Memento avoids exposing information that only an originator should manage but that must be stored nevertheless outside the originator. The pattern *shields other objects from potentially complex Originator internals*, thereby preserving encapsulation boundaries.

- ++ *It simplifies Originator.* In other encapsulation-preserving designs, Originator keeps the versions of internal state that clients have requested. That puts all the storage management burden on Originator. Having clients managing the state they ask for:
  - *simplifies Originator and*
  - *keeps clients from having to notify originators* when they're done.

Consequences --

- *-- Using mementos might be expensive.* Mementos might incur considerable overhead if Originator must copy large amounts of information to store in the memento or if clients create and return mementos to the originator often enough. <u>*Unless encapsulating and restoring Originator state is cheap, the pattern might not be appropriate*</u>.

- *-- Defining narrow and wide interfaces.* It may be <u>*difficult in some lang's to ensure that only the originator can access the memento's state*</u>.

- *-- Hidden costs in caring for mementos.* A caretaker is responsible for deleting the mementos it cares for. However, the caretaker has no idea how much state is in the memento. Hence an otherwise lightweight caretaker might incur <u>*large storage costs*</u> when it stores mementos.

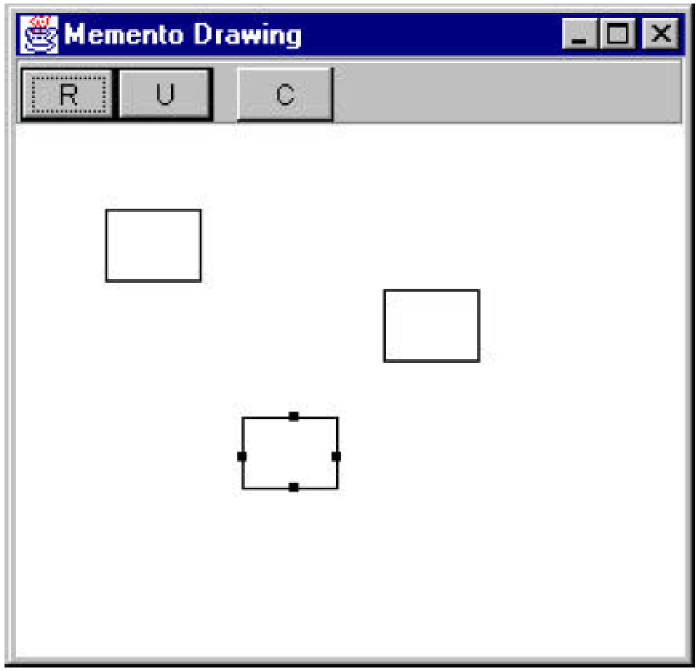
Implementation

- *Language support.* Mementos have two interfaces:
  - a wide one for originators, and
  - a narrow one for other objects.

  Ideally, the implementation language will support two levels of static protection. C++ let you do this by making the Originator a friend of Memento and making Memento's wide interface private. Only the narrow interface should be declared public.
- *Storing ONLY incremental changes.* When mementos get created and passed back to their originator *<u>in a predictable sequence</u>*, then Memento can save just the *incremental change* to the originator's internal state. For example, undoable commands in a history list can use mementos to ensure that commands are restored to their exact state when they're undone (see Command - GoF233). The history list defines a specific order in which commands can be undone and redone. That means *<u>mementos can store just the incremental change that a command makes rather than the full state</u>* of every object they affect.

Sample Java Example [3]

- Let’s consider a simple prototype of a graphics drawing program that creates rectangles, and allows you to select them and move them around by dragging them with the mouse. This program has a toolbar containing three buttons: **R**ectangle, **U**ndo and **C**lear
- The Rectangle button is a JToggleButton which stays selected until you click the mouse to draw a new rectangle. Once you have drawn the rectangle, you can click in any rectangle to select it;



There are 5 actions we need to respond to in this program:

1. **Rectangle button click**
2. **Undo button click**
3. **Clear button click**
4. **Mouse click**
5. **Mouse drag.**

- The _three buttons can be constructed as Command objects_ and the mouse click and drag can be treated as commands as well. This suggests an opportunity to use the Mediator pattern, and that is, in fact, the way this program is constructed.
- Moreover, our Mediator is an _ideal place to manage the Undo action list_; it can keep a list of the last *n* operations so that they can be undone. Thus, the Mediator also functions as the Caretaker object we described above. In fact, since there could be any number of actions to save and undo in such a program, a Mediator is virtually required so that there is a single place where these commands can be stored for undoing later.
- In this program we save and undo only two actions: _creating new rectangles_ and _changing the position of rectangles_. Let’s start with our visRectangle class which actually draws each instance of the rectangles:

```
public class visRectangle {
    int x, y, w, h; Rectangle rect; boolean selected;
    public visRectangle(int xpt, int ypt) {
        x = xpt; y = ypt; //save location
        w = 40; h = 30; //use default size
        saveAsRect();
    }
    public void setSelected(boolean b) { selected = b; }
    //-----------------------------------------
    private void saveAsRect() { //convert to rectangle so we can use “contains”
        rect = new Rectangle(x-w/2, y-h/2, w, h);
    }
    public void draw(Graphics g) {
        g.drawRect(x, y, w, h);
        if (selected) { //draw “handles”
            g.fillRect(x+w/2, y-2, 4, 4); g.fillRect(x-2, y+h/2, 4, 4);
            g.fillRect(x+w/2, y+h-2, 4, 4); g.fillRect(x+w-2, y+h/2, 4, 4);
        }
    }
    public boolean contains(int x, int y) {return rect.contains(x, y);}
    public void move(int xpt, int ypt) {
        x = xpt; y = ypt;saveAsRect();
    }
}
```

```
//our simple Memento class is contained in the same file, visRectangle.java,
//and thus has access to the position and size variables:
class Memento {
    visRectangle rect;
    //saved fields- remember internal fields
    //of the specified visual rectangle
    int x, y, w, h;
    public Memento(visRectangle r) {
        rect = r; //Save copy of instance
        x = rect.x; y = rect.y; //save position
        w = rect.w; h = rect.h; //and size
    }
    //-------------------------------------------
    public void restore() {
        //restore the internal state of
        //the specified rectangle
        rect.x = x; rect.y = y; //restore position
        rect.h = h; rect.w = w; //restore size
    }
}
```

When we create an instance of the Memento class, we pass it the visRectangle instance we want to save. It copies the size and position parameters and saves a copy of the instance of the visRectangle itself. Later, when we want to restore these parameters, the Memento knows which instance it has to restore them to and can do it directly, as we see in the *restore()* method.
The rest of the activity takes place in the Mediator class, where we save the previous state of the list of drawings as an Integer on the undo list:

```
public void createRect(int x, int y) {
    unpick(); //make sure no rectangle is selected
    if(startRect) { //if rect button is depressed
        Integer count = new Integer(drawings.size());
        undoList.addElement(count); //Save previous list size
        visRectangle v = new visRectangle(x, y);
        drawings.addElement(v); //add new element to list
        startRect = false; //done with this rectangle
        rect.setSelected(false); //unclick button
        canvas.repaint();
    } else pickRect(x, y); //if not pressed look for rect to select
}
```

```
          //save the previous position of a rectangle before moving it in a Memento
          public void rememberPosition() {
                    if(rectSelected){
                              Memento m = new Memento(selectedRectangle);
                              undoList.addElement(m);
                    } }
//the undo method simply decides whether to reduce the drawing list
//by one or to invoke the restore method of a Memento:
public void undo() {
          if(undoList.size()>0) { //get last element in undo list
                    Object obj = undoList.lastElement();
                    undoList.removeElement(obj); //and remove it
                    //if this is an Integer, the last action was a new rectangle
                    if (obj instanceof Integer) { //remove last created rectangle
                              Object drawObj = drawings.lastElement();
                              drawings.removeElement(drawObj);
                    }
                    //if this is a Memento, the last action was a move
                    if(obj instanceof Memento) { //get the Memento
                              Memento m = (Memento)obj;
                              m.restore(); //and restore the old position
                    }
                    repaint();
          }
}
```

We have covered all the ***GoF design patterns***

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	• Factory Method	• Adapter	• Interperter
	Object	• Abstract Factory • Builder • Prototype • Singleton	• Bridge • Composite • Decorator • Facade • Flyweight • Proxy	• Chain of Responsibility • Command • Iterator • Mediator • Template Method • Observer • State • Strategy • Visitor • Memento