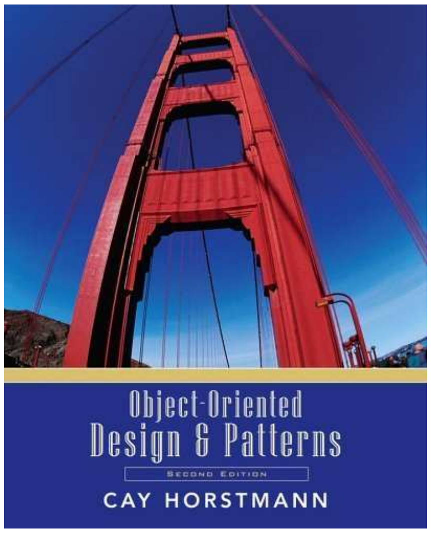
Object-Oriented Design & Patterns

Cay S. Horstmann

Chapter 10

More Design Patterns



Chapter Topics

- The ADAPTER Pattern
- Actions and the COMMAND Pattern
- The FACTORY METHOD Pattern
- The PROXY Pattern
- The SINGLETON Pattern
- The VISITOR Pattern
- Other Design Patterns

Adapters

- Cable adapter: adapts plug to foreign wall outlet
- OO Programming; Want to adapt class to foreign interface type
- Example: Add `CarIcon` to container
- Problem: Containers take components, not icons
- Solution: Create an adapter that adapts `Icon` to `Component`
- Ch10/adapter/IconAdapter.java
- Ch10/adapter/IconAdapterTester.java

```java
import java.awt.*;
import javax.swing.*;

/**
   An adapter that turns an icon into a JComponent.
*/
public class IconAdapter extends JComponent
{
   /**
      Constructs a JComponent that displays a given icon.
      @param icon the icon to display
   */
   public IconAdapter(Icon icon)
   {
      this.icon = icon;
   }

   public void paintComponent(Graphics g)
   {
      icon.paintIcon(this, g, 0, 0);
   }

   public Dimension getPreferredSize()
   {
      return new Dimension(icon.getIconWidth(),
            icon.getIconHeight());
   }

   private Icon icon;
}
```

```java
import java.awt.*;
import javax.swing.*;

/**
   This program demonstrates how an icon is adapted to
   a component. The component is added to a frame.
*/
public class IconAdapterTester
{
   public static void main(String[] args)
   {
      Icon icon = new CarIcon(300);
      JComponent component = new IconAdapter(icon);

      JFrame frame = new JFrame();
      frame.add(component, BorderLayout.CENTER);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.pack();
      frame.setVisible(true);
   }
}
```

previous | start | next

The ADAPTER Pattern

Context

1. You want to use an existing class (adaptee) without modifying it.
2. The context in which you want to use the class requires target interface that is different from that of the adaptee.
3. The target interface and the adaptee interface are conceptually related.

previous | start | next

previous | start | next

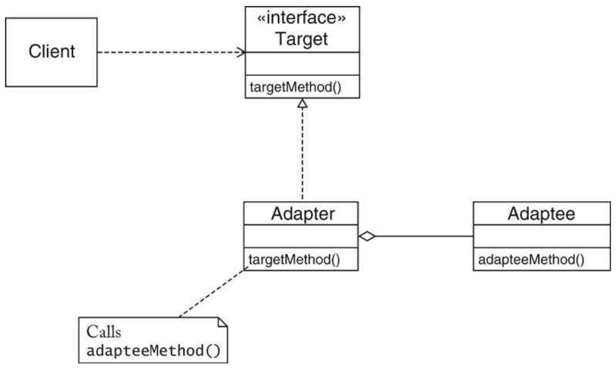
The ADAPTER Pattern

Solution

1. Define an adapter class that implements the target interface.
2. The adapter class holds a reference to the adaptee. It translates target methods to adaptee methods.
3. The client wraps the adaptee into an adapter class object.

previous | start | next

The ADAPTER Pattern



The ADAPTER Pattern

Name in Design Pattern	Actual Name (Icon->Component)
Adaptee	Icon
Target	JComponent
Adapter	IconAdapter
Client	The class that wants to add icons into a container
targetMethod()	paintComponent(), getPreferredSize()
adapteeMethod()	paintIcon(), getIconWidth(), getIconHeight()

The ADAPTER Pattern

- In stream library
- Input streams read bytes
- Readers read characters
- Non-ASCII encoding: multiple bytes per char
- System.in is a stream
- What if you want to read characters?
- Adapt stream to reader
- `InputStreamReader`

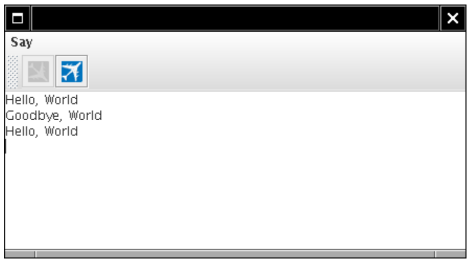
The ADAPTER Pattern

Name in Design Pattern	Actual Name (Stream->Reader)
Adaptee	InputStream
Target	Reader
Adapter	InputStreamReader
Client	The class that wants to read text from an input stream
targetMethod()	read (reading a character)
adapteeMethod()	read (reading a byte)

User Interface Actions

- Multiple routes to the same action
- Example: Cut a block of text
  - Select Edit->Cut from menu
  - Click toolbar button
  - Hit Ctrl+X
- Action can be disabled (if nothing is selected)
- Action has *state*
- Action should be an *object*

User Interface Actions

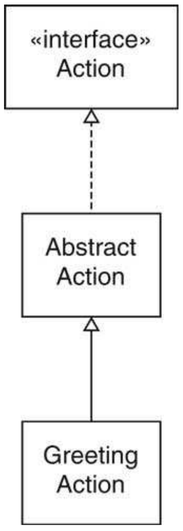


```
Hello, World
Goodbye, World
Hello, World
```

The `Action` Interface Type

- Extends `ActionListener`
- Can be enabled/disabled
- Additional state, including
  - Action name
  - Icon
- helloAction.putValue(Action.NAME, "Hello");
- `menu.add(helloAction);`
- Extend `AbstractAction` convenience class

The `Action` Interface Type



Action Example

- Ch10/command/CommandTester.java
- Ch10/command/GreetingAction.java

```java
import java.awt.*;
import javax.swing.*;

/**
   This program demonstrates action objects. Two actions
   insert greetings into a text area. Each action can be
   triggered by a menu item or toolbar button. When an
   action is carried out, the opposite action becomes enabled.
*/
public class CommandTester
{
   public static void main(String[] args)
   {
      JFrame frame = new JFrame();
      JMenuBar bar = new JMenuBar();
      frame.setJMenuBar(bar);
      JMenu menu = new JMenu("Say");
      bar.add(menu);
      JToolBar toolBar = new JToolBar();
      frame.add(toolBar, BorderLayout.NORTH);
      JTextArea textArea = new JTextArea(10, 40);
      frame.add(textArea, BorderLayout.CENTER);

      GreetingAction helloAction = new GreetingAction(
            "Hello, World", textArea);
      helloAction.putValue(Action.NAME, "Hello");
      helloAction.putValue(Action.SMALL_ICON,
         new ImageIcon("hello.png"));

      GreetingAction goodbyeAction = new GreetingAction(
            "Goodbye, World", textArea);
      goodbyeAction.putValue(Action.NAME, "Goodbye");
      goodbyeAction.putValue(Action.SMALL_ICON,
            new ImageIcon("goodbye.png"));

      helloAction.setOpposite(goodbyeAction);
      goodbyeAction.setOpposite(helloAction);
      goodbyeAction.setEnabled(false);

      menu.add(helloAction);
      menu.add(goodbyeAction);

      toolBar.add(helloAction);
      toolBar.add(goodbyeAction);
```

```java

      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.pack();
      frame.setVisible(true);
   }
}
```

```java
import java.awt.event.*;
import javax.swing.*;

/**
   This action places a greeting into a text field
   and afterwards disables itself and enables its
   opposite action.
*/
public class GreetingAction extends AbstractAction
{
   /**
      Constructs a greeting action.
      @param greeting the string to add to the text area
      @param textArea the text area to which to add the greeting
   */
   public GreetingAction(String greeting, JTextArea textArea)
   {
      this.greeting = greeting;
      this.textArea = textArea;
   }

   /**
      Sets the opposite action.
      @param action the action to be enabled after this action was
      carried out
   */
   public void setOpposite(Action action)
   {
      oppositeAction = action;
   }

   public void actionPerformed(ActionEvent event)
   {
      textArea.append(greeting);
      textArea.append("\n");
      if (oppositeAction != null)
      {
         setEnabled(false);
         oppositeAction.setEnabled(true);
      }
   }

   private String greeting;
   private JTextArea textArea;
   private Action oppositeAction;
}
```

previous | start | next

The COMMAND Pattern

Context

1. You want to implement commands that behave like objects
   - because you need to store additional information with commands
   - because you want to collect commands.

previous | start | next

previous | start | next

The COMMAND Pattern

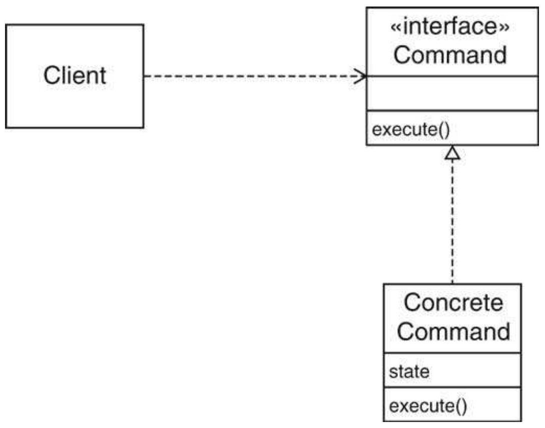
Solution

1. Define a command interface type with a method to execute the command.
2. Supply methods in the command interface type to manipulate the state of command objects.
3. Each concrete command class implements the command interface type.
4. To invoke the command, call the execute method.

previous | start | next

previous | start | next

The COMMAND Pattern



previous | start | next

The COMMAND Pattern

Name in Design Pattern	Actual Name (Swing actions)
Command	`Action`
ConcreteCommand	subclass of `AbstractAction`
execute()	`actionPerformed()`
state	`name` and `icon`

Factory Methods

- Every collection can produce an iterator
  `Iterator iter = list.iterator()`
- Why not use constructors?
  `Iterator iter = new LinkedListIterator(list);`
- Drawback: not generic
  ```
  Collection coll = ...;
  Iterator iter = new ???(coll);
  ```
- Factory method works for all collections
  `Iterator iter = coll.iterator();`
- Polymorphism!

The FACTORY METHOD Pattern

Context

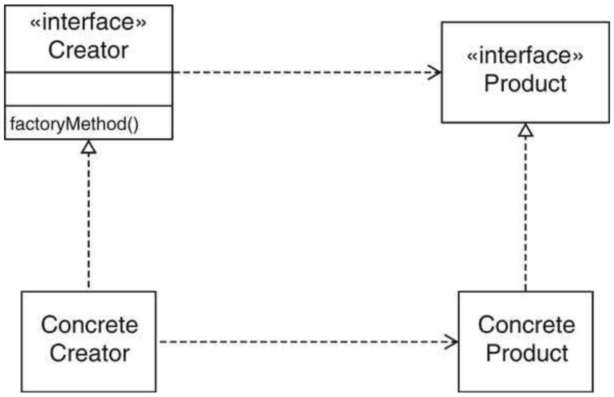
1. A type (the creator) creates objects of another type (the product).
2. Subclasses of the creator type need to create different kinds of product objects.
3. Clients do not need to know the exact type of product objects.

The FACTORY METHOD Pattern

Solution

1. Define a creator type that expresses the commonality of all creators.
2. Define a product type that expresses the commonality of all products.
3. Define a method, called the factory method, in the creator type.
   The factory method yields a product object.
4. Each concrete creator class implements the factory method so that it returns an object of a concrete product class.

The FACTORY METHOD Pattern



The FACTORY METHOD Pattern

Name in Design Pattern	Actual Name (iterator)
Creator	Collection
ConcreteCreator	A subclass of `Collection`
factoryMethod()	iterator()
Product	Iterator
ConcreteProduct	A subclass of `Iterator` (which is often anonymous)

Not a FACTORY METHOD

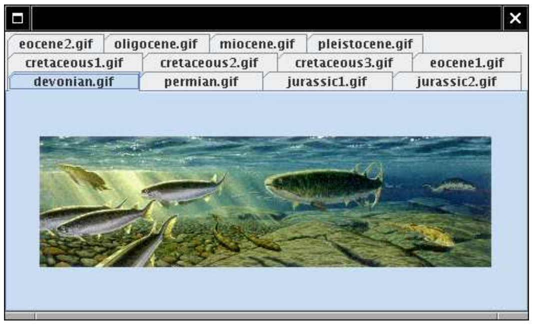
- Not all "factory-like" methods are instances of this pattern
- Create `DateFormat` instances
```
DateFormat formatter =
DateFormat.getDateInstance();
Date now = new Date();
String formattedDate = formatter.format(now);
```
- `getDateInstance` is a *static* method
- No polymorphic creation

Proxies

- Proxy: a person who is authorized to act on another person s behalf
- Example: Delay instantiation of object
- Expensive to load image
- Not necessary to load image that user doesn’t look at
- Proxy defers loading until user clicks on tab

previous | start | next

Deferred Image Loading



previous | start | next

previous | start | next

Deferred Image Loading

- Normally, programmer uses image for label:
  `JLabel label = new JLabel(new ImageIcon(imageName));`
- Use proxy instead:
  `JLabel label = new JLabel(new ImageProxy(imageName));`
- `paintIcon` loads image if not previously loaded

```
public void paintIcon(Component c, Graphics
g, int x, int y)
{
   if (image == null) image = new
ImageIcon(name);
   image.paintIcon(c, g, x, y);
}
```

previous | start | next

previous | start | next

Proxies

- Ch10/proxy/ImageProxy.java
- Ch10/proxy/ProxyTester.java
- "Every problem in computer science can be solved by an additional level of indirection"
- Another use for proxies: remote method invocation

previous | start | next

```
import java.awt.*;
import javax.swing.*;

/**
   A proxy for delayed loading of image icons.
*/
public class ImageProxy implements Icon
{
   /**
      Constructs a proxy for delayed loading of an image file.
      @param name the file name
   */
   public ImageProxy(String name)
   {
      this.name = name;
      image = null;
   }

   public void paintIcon(Component c, Graphics g, int x, int y)
   {
      ensureImageLoaded();
      image.paintIcon(c, g, x, y);
   }

   public int getIconWidth()
   {
      ensureImageLoaded();
      return image.getIconWidth();
   }

   public int getIconHeight()
   {
      ensureImageLoaded();
      return image.getIconHeight();
   }

   /**
      Loads the image if it hasn't been loaded yet. Prints
      a message when the image is loaded.
   */
   private void ensureImageLoaded()
   {
      if (image == null)
      {
```

```java
         System.out.println("Loading " + name);
         image = new ImageIcon(name);
      }
   }

   private String name;
   private ImageIcon image;
}
```

```java
import java.awt.*;
import javax.swing.*;

/**
   This program demonstrates the use of the image proxy.
   Images are only loaded when you press on a tab.
*/
public class ProxyTester
{
   public static void main(String[] args)
   {
      JTabbedPane tabbedPane = new JTabbedPane();
      for (String name : imageNames)
      {
         JLabel label = new JLabel(new ImageProxy(name));
         tabbedPane.add(name, label);
      }

      JFrame frame = new JFrame();
      frame.add(tabbedPane);

      frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }

   private static final String[] imageNames =
   {
      "devonian.gif",
      "permian.gif",
      "jurassic1.gif",
      "jurassic2.gif",
      "cretaceous1.gif",
      "cretaceous2.gif",
      "cretaceous3.gif",
      "eocene1.gif",
      "eocene2.gif",
      "oligocene.gif",
      "miocene.gif",
      "pleistocene.gif"
   };
```

```java

   private static final int FRAME_WIDTH = 500;
   private static final int FRAME_HEIGHT = 300;
}
```

previous | start | next

The PROXY Pattern

Context

1. A class (the real subject) provides a service that is specified by an interface type (the subject type)
2. There is a need to modify the service in order to make it more versatile.
3. Neither the client nor the real subject should be affected by the modification.

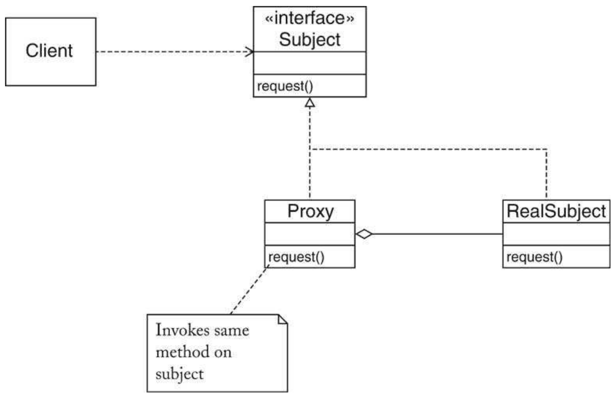
previous | start | next

The PROXY Pattern

Solution

1. Define a proxy class that implements the subject interface type. The proxy holds a reference to the real subject, or otherwise knows how to locate it.
2. The client uses a proxy object.
3. Each proxy method invokes the same method on the real subject and provides the necessary modifications.

The PROXY Pattern



The PROXY Pattern

Name in Design Pattern	Actual Name (image proxy)
Subject	Icon
RealSubject	ImageIcon
Proxy	ImageProxy
request()	The methods of the `Icon` interface type
Client	JLabel

Singletons

- "Random" number generator generates predictable stream of numbers
- Example: seed = (seed * 25214903917 + 11) % 2^{48}
- Convenient for debugging: can reproduce number sequence
- Only if all clients use *the same random number generator*
- Singleton class = class with one instance

Random Number Generator Singleton

```
public class SingleRandom
{
   private SingleRandom() { generator = new
Random(); }
   public void setSeed(int seed) {
generator.setSeed(seed); }
   public int nextInt() { return
generator.nextInt(); }
   public static SingleRandom getInstance() {
return instance; }
   private Random generator;
   private static SingleRandom instance = new
SingleRandom();
}
```

The SINGLETON Pattern

Context

1. All clients need to access a single shared instance of a class.
2. You want to ensure that no additional instances can be created accidentally.

The SINGLETON Pattern

Solution

1. Define a class with a private constructor.
2. The class constructs a single instance of itself.
3. Supply a static method that returns a reference to the single instance.

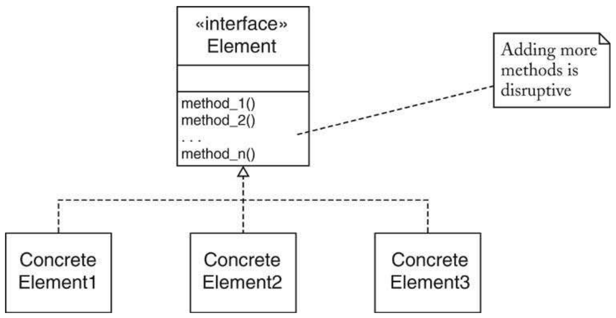
Not a SINGLETON

- Toolkit used for determining screen size, other window system parameters
- `Toolkit` class returns default toolkit
  `Toolkit kit = Toolkit.getDefaultToolkit();`
- Not a singleton--can get other instances of `Toolkit`
- `Math` class not example of singleton pattern
- *No* objects of class `Math` are created

Inflexible Hierarchies

- How can one add operations to compound hierarchies?
- Example: AWT `Component, Container,` etc. form hierarchy
- Lots of operations: `getPreferredSize`,`repaint`
- Can't add new methods without modifying `Component` class
- VISITOR pattern solves this problem
- Each class must support one method
  `void accept(Visitor v)`

Inflexible Hierarchies



Visitors

- `Visitor` is an interface type
- Supply a separate class for each new operation
- Most basic form of `accept` method:
  ```
  public void accept(Visitor v) {
  v.visit(this); }
  ```
- Programmer must implement `visit`

Visitors

- Problem: Operation may be different for different element types
- Can't rely on polymorphism
- Polymorphism assumes fixed set of methods, defined in superclass
- Trick: Can use variable set of methods *if set of classes is fixed*
- Supply separate visitor methods:
  ```
  public interface Visitor
  {
     void visitElementType1(ElementType1
  element);
     void visitElementType2(ElementType2
  element);
     ...
     void visitElementTypen(ElementTypen
  element);
  }
  ```

Visitors

- Example: Directory tree
- Two kinds of elements: `DirectoryNode`,`FileNode`
- Two methods in visitor interface type:
```
void visitDirectoryNode(DirectoryNode node)
void visitFileNode(FileNode node)
```

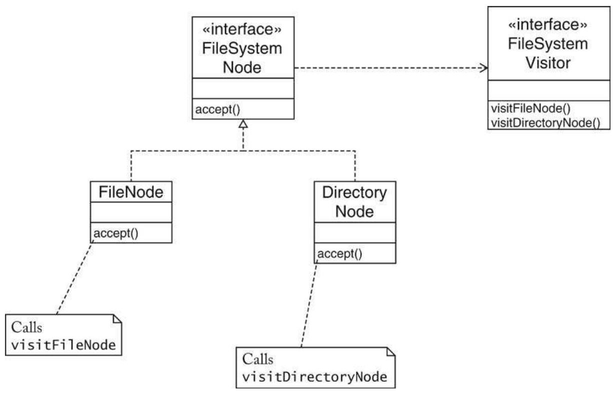
Double Dispatch

- Each element type provides methods:
```
public class ElementType i
{
   public void accept(Visitor v) {
v.visitElementType i (this); }
   ...
}
```
- Completely mechanical
- Example:
```
public class DirectoryNode
{
   public void accept(Visitor v) {
v.visitDirectoryNode(this); }
   ...
}
```

Visitor Example

- Standard `File` class denotes both files and directories
- Improved design: `FileNode`,`DirectoryNode`
- Common interface type: `FileSystemNode`
- Accepts `FileSystemVisitor`
- Visitor methods:
```
visitFileNode
visitDirectoryNode
```

Visitor Example



Visitor Example

- Actual visitor: `PrintVisitor`
- Prints names of files (in `visitFileNode`)
- Lists contents of directories (in `visitDirectoryNode`)
- Maintains indentation level

```
..
   command
     CommandTester.java
     GreetingAction.java
   visitor
     FileNode.java
      DirectoryNode.java
```

Visitor Example

- ch10/visitor/FileSystemNode.java
- ch10/visitor/FileNode.java
- ch10/visitor/DirectoryNode.java
- ch10/visitor/FileSystemVisitor.java
- ch10/visitor/PrintVisitor.java
- ch10/visitor/VisitorTester.java

```java
/**
   The common interface for file and directory nodes.
*/
public interface FileSystemNode
{
   void accept(FileSystemVisitor v);
}
```

```java
import java.io.*;

public class FileNode implements FileSystemNode
{
   public FileNode(File file)
   {
      this.file = file;
   }

   public File getFile() { return file; }

   public void accept(FileSystemVisitor v)
   {
      v.visitFileNode(this);
   }

   private File file;
}
```

```java
import java.io.*;

public class DirectoryNode implements FileSystemNode
{
   public DirectoryNode(File directory)
   {
      this.directory = directory;
   }

   public void accept(FileSystemVisitor v)
   {
      v.visitDirectoryNode(this);
   }

   public File getDirectory() { return directory; }

   public FileSystemNode[] getChildren()
   {
      File[] files = directory.listFiles();
      FileSystemNode[] children = new FileSystemNode[files.length];
      for (int i = 0; i < files.length; i++)
      {
         File f = files[i];
         if (f.isDirectory())
            children[i] = new DirectoryNode(f);
         else
            children[i] = new FileNode(f);
      }
      return children;
   }

   private File directory;
}
```

```java
/**
   The visitor interface type for visiting file system nodes.
*/
public interface FileSystemVisitor
{
   /**
      Visits a file node.
      @param node the file node
   */
   void visitFileNode(FileNode node);

   /**
      Visits a directory node.
      @param node the directory node
   */
   void visitDirectoryNode(DirectoryNode node);
}
```

```java
import java.io.*;

public class PrintVisitor implements FileSystemVisitor
{
   public void visitFileNode(FileNode node)
   {
      for (int i = 0; i < level; i++) System.out.print(" ");
      System.out.println(node.getFile().getName());
   }

   public void visitDirectoryNode(DirectoryNode node)
   {
      for (int i = 0; i < level; i++) System.out.print(" ");
      System.out.println(node.getDirectory().getName());
      level++;
      for (FileSystemNode c : node.getChildren())
         c.accept(this);
      level--;
   }

   private int level = 0;
}
```

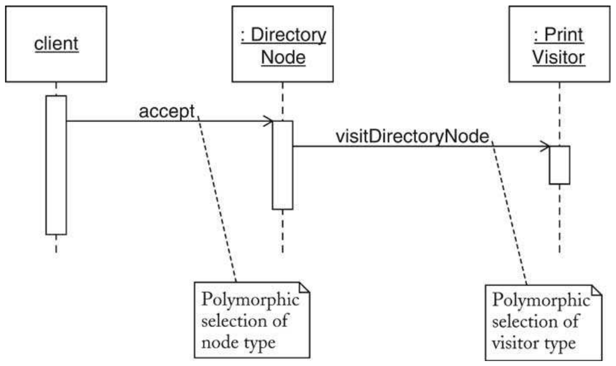
```java
import java.io.*;

public class VisitorTester
{
   public static void main(String[] args)
   {
      DirectoryNode node = new DirectoryNode(new File(".."));
      node.accept(new PrintVisitor());
   }
}
```

Double Dispatch Example

- `DirectoryNode node = new DirectoryNode(new File(".."));`
  `node.accept(new PrintVisitor());`
- `node` is a `DirectoryNode`
- Polymorphism: `node.accept` calls `DirectoryNode.accept`
- That method calls `v.visitDirectoryNode`
- `v` is a `PrintVisitor`
- Polymorphism: calls `PrintVisitor.visitDirectoryNode`
- Two polymorphic calls determine
-
  - node type
  - visitor type

Double Dispatch Example


client
: Directory Node
: Print Visitor
accept
visitDirectoryNode
Polymorphic selection of node type
Polymorphic selection of visitor type

The VISITOR Pattern

Context

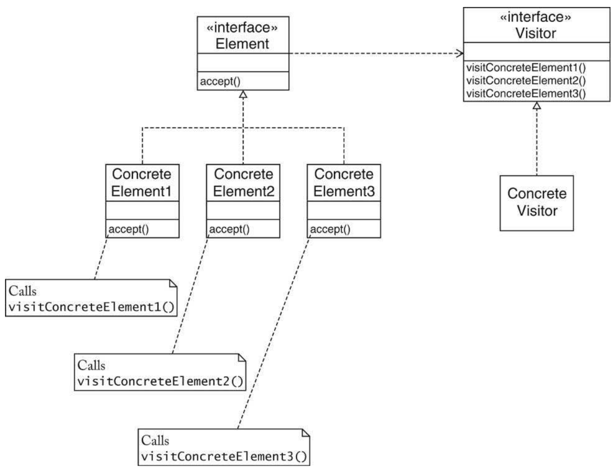
1. An object structure contains element classes of multiple types, and you want to carry out operations that depend on the object types.
2. The set of operations should be extensible over time.
3. The set of element classes is fixed.

The VISITOR Pattern

Solution

1. Define a visitor interface type that has methods for visiting elements of each of the given types.
2. Each element class defines an accept method that invokes the matching element visitation method on the visitor parameter.
3. To implement an operation, define a class that implements the visitor interface type and supplies the operation s action for each element type.

The VISITOR Pattern



The VISITOR Pattern

Name in Design Pattern	**Actual Name (file system visitor)**
Element	FileSystemNode
ConcreteElement	FileNode, DirectoryNode
Visitor	FileSystemVisitor
ConcreteVisitor	`PrintVisitor`

Other Design Patterns

- Abstract Factory
- Bridge
- Builder
- Chain of Responsibility
- Flyweight
- Interpreter
- Mediator
- Memento
- State

previous | start

Conclusion: What You Learned

1. Object-oriented design
   The design methodology
   CRC cards and UML diagrams
   Design patterns
2. Advanced Java
   Interface types, polymorphism, and inheritance
   Inner classes
   Reflection
   Multithreading
   Collections
3. User interface programming
   Building Swing applications
   Event handling
   Graphics programming

previous | start