Design Patterns (and some GUI Programming)

Horstmann Chapter 5 (sections 1-7)

CS 4354
Summer II 2016

Jill Seaman

Design Patterns

- In object-oriented development, **Design Patterns** are solutions that developers have refined over time to solve a range of recurring design problems.
- A design pattern has four elements
  - ✦A **name** that uniquely identifies the pattern from other patterns.
  - ✦A **problem description** that describes the situation in which the pattern can be used.
  - ✦A **solution** stated as a set of collaborating classes and interfaces.
  - ✦A **set of consequences** that describes the trade-offs and alternatives to be considered with respect to the design goals being addressed.

Design Patterns

- The following terms are often used to denote the classes that collaborate in a design pattern:
  - ✦The **client class** accesses the pattern classes.
  - ✦The **pattern interface** is the part of the pattern that is visible to the client class (might be an interface or abstract class).
  - ✦The **implementor class** provides low level behavior of the pattern. Often the pattern contains many of these.
  - ✦The **extender class** specializes an implementor class to provide different implementation of the pattern. These usually represent future classes anticipated by the developer.
- Common tradeoff: Simple architecture vs extensibility

Delegation

- **Delegation:** A special form of composition, commonly used in design patterns.
  - ✦One class (A) contains a reference to another (B) (via member variable)
  - ✦A implements its operations/methods by calling methods on B. (Methods may have different names)
  - ✦Makes explicit the dependencies between A and B.
- Advantages of delegation:
  - ✦B might be a pre-existing class, so we can reuse it without changing it.
  - ✦B is hidden from clients of A, B can easily be changed or even replaced with another class.

Encapsulating Traversals with The ITERATOR Pattern

- Recall using an Iterator to iterate through the elements of a linked list in Java:

```
LinkedList<String> countries = new LinkedList<String>();
countries.add("Belgium");
countries.add("Italy");
countries.add("Thailand");
Iterator<String> iterator = countries.iterator();
while (iterator.hasNext())  {
   String country = iterator.next();
   System.out.println(country);
}
```

- The hasNext method tests whether the iterator is at the end of the list.
- The next method returns the current element and advances the iterator to the next position.
- Why does the Java library use an iterator to traverse a linked list?

Compare to C++ linked list traversal

- Recall performing a traversal of a linked list in C++:

```
ListNode *p = head;
 while (p!=NULL) {
     cout << p->value << " ";
     p = p->next;
}
cout << endl;
```

- This exposes the internal structure of the list.
- And it's error-prone: "it is very easy to mess up links and corrupt the link structure of a linked list"

What interface (methods) to use instead?

- For a (java) Queue:

```
void add(E x)  //enqueue
E remove()     //dequeue
E peek()       //returns next elem (doesn't remove)
int size()     //number of elems in the queue
```

- For an Array List:

```
E get(int i)
void set(int i, E x)
void add(E x)
int size()
```

- For a Linked List, we want to be able to add and remove elements from the middle of the list, but it would be very inefficient to specify a position in a linked list with an integer index.

A Cursor-based Linked List

- A common linked-list implementation is a list with a cursor
- A list cursor marks a position similar to the cursor in a word processor.

```
E getCurrent()     // Get element at cursor
void set(E x)      // Set element at cursor to x
E remove()         // Remove element at cursor
void insert(E x)   // Insert x before cursor
void reset()       // Reset cursor to head
void next()        // Advance cursor
boolean hasNext()  // Check if cursor can be advanced
```

Note: next() does not return anything

- Requires adding a field to point to the current element.
- Here is how to traverse the list:

```
list.reset();
while (list.hasNext()) {
  //do something with list.getCurrent();
  list.next();
}
```

Problem with the Cursor-based Linked List

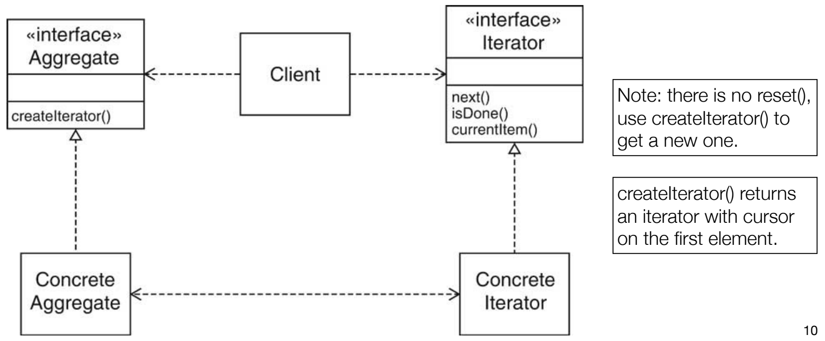
- There is only one cursor, you can’t implement algorithms that compare different list elements.
- You can’t even print the contents of the list for debugging purposes, because it moves the cursor to the end.

- A list can have any number of iterators attached to it.
- The iterator concept is also useful outside of the collection classes (see the Scanner).

- It is a good solution to a common problem.

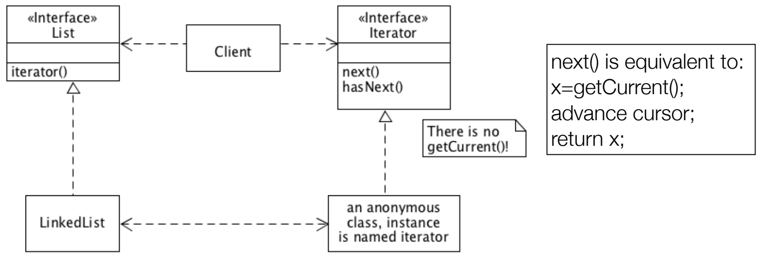
The Iterator as a Design Pattern

Name: Iterator Design Pattern
Problem Description: Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation, for multiple clients simultaneously.
Solution: ConcreteIterator class implements the **Iterator** interface for accessing and traversing elements. The **ConcreteAggregate** implements the **Aggregate** interface for creating an Iterator object.


«interface»
Aggregate
createIterator()
Client
«interface»
Iterator
next()
isDone()
currentItem()
Concrete
Aggregate
Concrete
Iterator
Note: there is no reset(), use createIterator() to get a new one.
createIterator() returns an iterator with cursor on the first element.

Example: Linked List Iterators

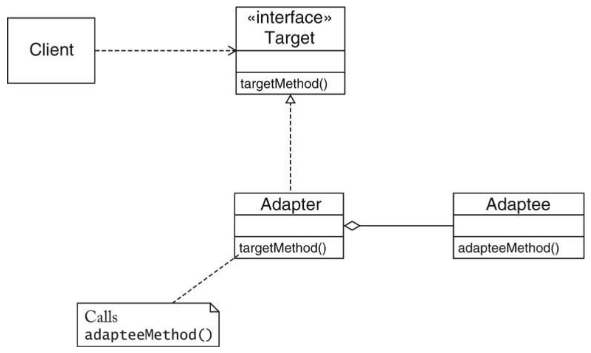
- I substituted the types from the Linked List example into the class diagram to show how it fits the pattern.



```
LinkedList<String> countries = new LinkedList<String>();
countries.add("Belgium");
countries.add("Italy");
countries.add("Thailand");
Iterator<String> iterator = countries.iterator();
while (iterator.hasNext())  {
   String country = iterator.next();
   System.out.println(country);
}
```

Encapsulating Legacy Components with the ADAPTER Pattern

Name: Adapter Design Pattern
Problem Description: Convert the interface of a legacy class into a different interface expected by the client, so they can work together without changes.
Solution: Adapter class implements the **Target** interface expected by the client. The **Adapter** delegates requests from the client to the **Adaptee** (the pre-existing legacy class) and performs any necessary conversion.

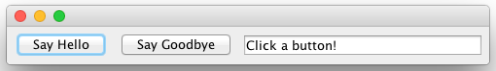


Example: Adding an Icon to a UI Container

- First, how to display a window and add UI Components to it:
- A frame window is a top-level window, usually decorated with borders and a title bar, displayed as follows:

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

- pack: sets the size of the frame to the smallest size needed to display its components.
- EXIT_ON_CLOSE: program exits when user closes the window.
- Let's build this:



13

Example: Adding an Icon to a UI Container

- First, construct some buttons:

```
JButton helloButton = new JButton("Say Hello");
JButton goodbyeButton = new JButton("Say Goodbye");
```

- Make a text field:

```
final int FIELD_WIDTH = 20;
JTextField textField = new JTextField(FIELD_WIDTH);
textField.setText("Click a button!");
```

- Set a layout manager (how to position components):

```
frame.setLayout(new FlowLayout());
```

- Finally, add the components to the frame and display.

```
frame.add(helloButton);
frame.add(goodbyeButton);
frame.add(textField);
```

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

14

Example: Adding an Icon to a UI Container

- NOW, let's say I want to add a MarsIcon to my JFrame.
- Problem: the JFrame.add() method takes a Component, not an Icon.
- I could just make my MarsIcon implement the Component interface, but that's a lot of work.
- There is a JComponent class that I could make MarsIcon inherit from, but let's assume MarsIcon already has another superclass.
- Solution: Make a new class (IconAdapter) that is a subclass of JComponent. It will hold a reference to an Icon. It translates JComponent methods to Icon methods.

15

Example: IconAdapter

```
import java.awt.*;
import javax.swing.*;
/**
  An adapter that turns an icon into a JComponent.
 */
public class IconAdapter extends JComponent
{
   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;
}
```

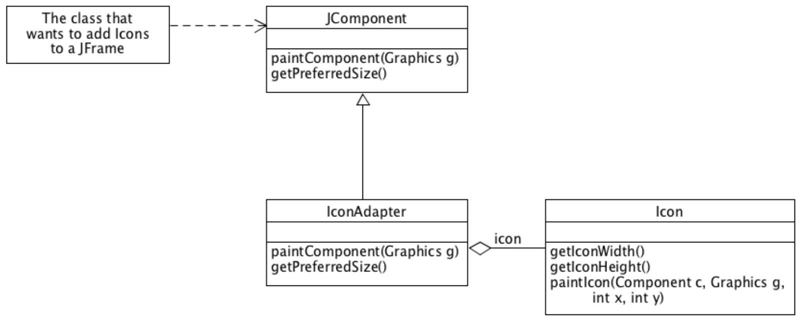
16

Example: IconAdapterTester

```
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 MarsIcon(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);
   }
}
```

Adapter Pattern: IconAdapter

- The IconAdapter as an instance of the ADAPTER Pattern.
- Note we use inheritance instead of implementing an interface.

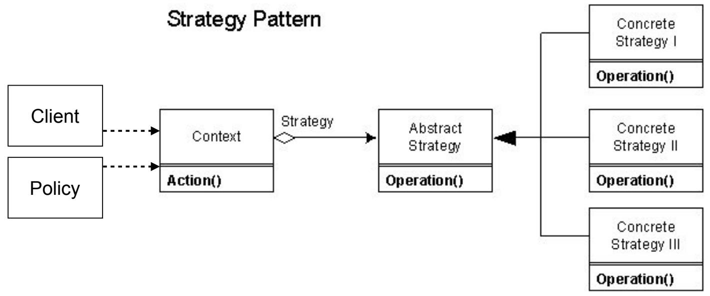

The class that wants to add Icons to a JFrame
JComponent
paintComponent(Graphics g)
getPreferredSize()
IconAdapter
paintComponent(Graphics g)
getPreferredSize()
icon
Icon
getIconWidth()
getIconHeight()
paintIcon(Component c, Graphics g, int x, int y)

Adapter Pattern: consequences

- Client and Adaptee work together without any modification to either.
- Adapter works with Adaptee and all of its sub classes
- A new Adapter needs to be written for each specialization (extension) of the Target interface.
- Question: Where does the Adapter Pattern use inheritance? Where does it use delegation?

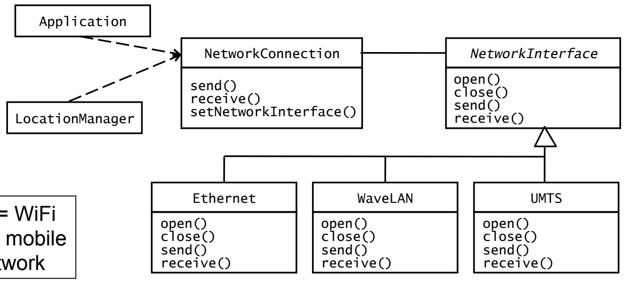
Encapsulating Context with the STRATEGY Pattern

Name: Strategy Design Pattern
Problem Description: Define a family of algorithms, encapsulate each one, and make them interchangeable. The algorithm is decoupled from the client.
Solution: A Client accesses services provided by a Context. The **Context** is configured to use one of the **ConcreteStrategy** objects (and maintains a reference to it) . The **AbstractStrategy** class describes the interface that is common to all the ConcreteStrategies.



Example: switching between network protocols

- Based on location (available network connections), switch between different types of network connections
  - ✦LocationManager configures NetworkConnection with a concrete NetworkInterface based on the current location
  - ✦Application uses the NetworkConnection independently of concrete NetworkInterfaces (NetworkConnection uses delegation).



WaveLAN = WiFi
UMTS = 3G mobile phone network

Strategy Pattern example: Network protocols

```
// Context Object: Network Connection
public class NetworkConnection {
   private String destination;
   private NetworkInterface intf;
   private StringBuffer queue;

   public NetworkConnect(String destination, NetworkInterface intf) {
      this.destination = destination;  this.intf = intf;
      this.intf.open(destination);
   }
   public void send(byte msg[]) {
      queue.concat(msg);
      if (intf.isReady()) {
         intf.send(queue);
         queue.setLength(0);
      }
   }
   public byte[] receive () {
      return intf.receive();
   }
   public void setNetworkInterface(NetworkInterface newIntf) {
      intf.close()
      newIntf.open(destination);
      intf = newIntf;
}  }
```

Strategy Pattern example: Network protocols

```
//Abstract Strategy,
//Implemented by EthernetNetwork, WaveLanNetwork, and UMTSNetwork (not shown)
interface NetworkInterface {
   void open(String destination);
   void close();
   byte[] receive();
   void send(StringBuffer queue);
   bool isReady();
}
//LocationManager: decides on which strategy to use
public class LocationManager {
   private NetworkConnection networkConn;

   // called by event handler when location has changed
   public void doLocation() {
      NetworkInterface networkIntf;
      if (isEthernetAvailable())
         networkIntf = new EthernetNetwork();
      else if (isWaveLANAvailable())
         networkIntf = new WaveLanNetwork();
      else if (isUMTSAvailable())
         networkIntf = new UMTSNetwork();
      networkConn.setNetworkInterface(networkIntf);
   }
}
```

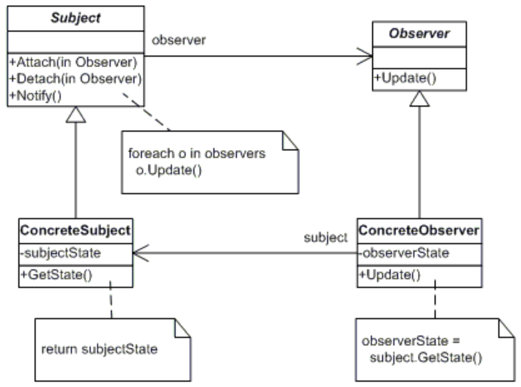
Strategy Pattern: consequences

- ConcreteStrategies can be substituted transparently from Context.
- Client (or Policy) decides which Strategy is best, given current circumstances
- New algorithms can be added without modifying Context or Client

Decoupling Entities from Views with the OBSERVER Pattern

Name: Observer Design Pattern
Problem Description: Maintain consistency across the states of one Subject and many Observers.

Solution: The **Subject** maintains some state. One or more **Observers** use the state maintained by the Subject. Observers invoke the attach() method to register with a Subject. Each **ConcreteObserver** defines an update() method to synchronize its state with the Subject. Whenever the state of the Subject changes, it invokes its notify method, which iteratively invokes each Observer.update() method.



Observer Pattern: Java support

- We could implement the Observer pattern "from scratch" in Java. But Java provides the Observable/Observer classes as built-in support for the Observer pattern.

- The java.util.Observer interface is the Observer **interface**. It must be implemented by any observer class. It has one method.
  - void **update** (Observable o, Object arg)
    This method is called whenever the observed object is changed.
    Observable o is the observed object.
    Object arg can be some value sent by the observed object.

Observer Pattern: Java support

- The java.util.Observable class is the base Subject **class**. Any class that wants to be observed extends this class.
  - public synchronized void **addObserver**(Observer o)
    Adds an observer to the set of observers of this object
  - boolean **hasChanged**() (see below)
  - protected void **setChanged**()
    Indicates this object has changed (hasChanged now returns true)
  - public void **notifyObservers**(Object arg)
  - public void **notifyObservers**()
    IF hasChanged(), THEN notify all of its observers. Each observer has its update() method called with this Observable object (and an argument). The argument can be used to indicate which attribute of this object has changed. (hasChanged now returns false).

Observer Pattern example:

```
import java.util.Observable;

/* A subject to observe! */
public class ConcreteSubject extends Observable {
   private String name;
   private float price;
   public ConcreteSubject(String name, float price) {
       this.name = name;
       this.price = price;
       System.out.println("ConcreteSubject created: " + name + " at " + price);
   }
   public String getName() {return name;}
   public float getPrice() {return price;}
   public void setName(String name) {
        this.name = name;
        setChanged();
        notifyObservers();
   }
   public void setPrice(float price) {
       this.price = price;
       setChanged();
       notifyObservers();
   }
}
```

Observer Pattern example:

```
import java.util.Observable;
import java.util.Observer;

//An observer of name changes.
public class NameObserver implements Observer {
    private String name;

    public NameObserver(ConcreteSubject cs) {
        cs.addObserver(this);
        name = cs.getName();
        System.out.println("NameObserver created: Name is " + name);
    }

    public void update(Observable obj, Object arg) {
        ConcreteSubject cs = (ConcreteSubject)obj;

        if (!name.equals(cs.getName())) {
            name = cs.getName();
            System.out.println("NameObserver: Name changed to " + name);
        }
    }
}
```

Observer Pattern example:

```
import java.util.Observable;
import java.util.Observer;

//An observer of price changes.
public class PriceObserver implements Observer {
    private float price;

    public PriceObserver(ConcreteSubject cs) {
        cs.addObserver(this);
        price = cs.getPrice();
        System.out.println("PriceObserver created: Price is " + price);
    }

    public void update(Observable obj, Object arg) {
        ConcreteSubject cs = (ConcreteSubject)obj;
        if (cs.getPrice()!= price) {
            price = cs.getPrice();
            System.out.println("PriceObserver: Price changed to " + price);
        }
    }
}
```

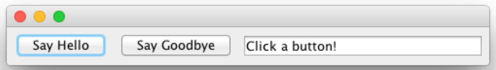
Observer Pattern example:

```
//Test program for ConcreteSubject, NameObserver and PriceObserver
public class TestObservers {
    public static void main(String args[]) {
        // Create the Subject.
        ConcreteSubject s = new ConcreteSubject("Corn Pops", 1.29f);
        // Create the Observers, who attach themselves to the subject
        NameObserver nameObs = new NameObserver(s);
        PriceObserver priceObs = new PriceObserver(s);
        // Make changes to the Subject.
        s.setName("Frosted Flakes");
        s.setPrice(4.57f);
        s.setPrice(9.22f);
        s.setName("Sugar Crispies");
    }
}
```

```
ConcreteSubject created: Corn Pops at 1.29
NameObserver created: Name is Corn Pops
PriceObserver created: Price is 1.29
NameObserver: Name changed to Frosted Flakes
PriceObserver: Price changed to 4.57
PriceObserver: Price changed to 9.22
NameObserver: Name changed to Sugar Crispies
```

Example: Making Buttons Work with UI Actions.

- Recall the JFrame window we made earlier:



- These buttons do nothing if you click on them.
- We need to add listener objects to the button.
- Listener objects implement the following Interface:

```
public interface ActionListener {
   int actionPerformed(ActionEvent event);
}
```

- The ActionEvent parameter contains information about the event, such as the event source. But we usually don't need that info.

Example: Making Buttons Work with UI Actions.

- ActionListeners attach themselves to a button, and when the button is clicked the code of the actionPerformed method is executed.
- To define the action of the helloButton, we use an *anonymous class* to implement the ActionListener interface type:

```
final JTextField textField = new JTextField(FIELD_WIDTH);
JButton helloButton = new JButton("Say Hello");

helloButton.addActionListener(new
  ActionListener() {
  public void actionPerformed(ActionEvent event) {
    textField.setText("Hello, World!");
  }
});
```

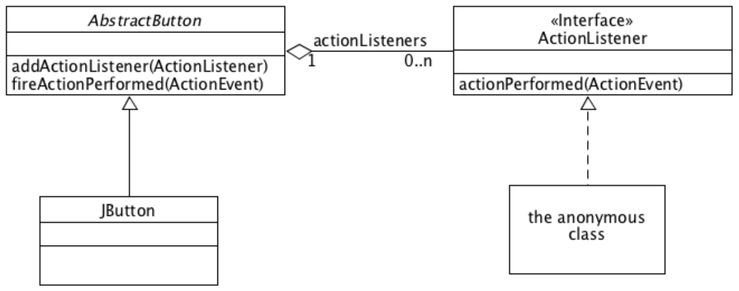
- When the button is clicked, the textField will be set.
- Note: the anonymous class can access fields from enclosing class (like textField) IF they are marked Final.

Example: Making Buttons Work with UI Actions.

```
public class ActionTester {
   public static void main(String[] args) {
      JFrame frame = new JFrame();
      final JTextField textField = new JTextField(20);
      textField.setText("Click a button!");
      JButton helloButton = new JButton("Say Hello");
      helloButton.addActionListener(new
         ActionListener() {
            public void actionPerformed(ActionEvent event) {
               textField.setText("Hello, World!");
            }
         });
      JButton goodbyeButton = new JButton("Say Goodbye");
      goodbyeButton.addActionListener(new
         ActionListener() {
            public void actionPerformed(ActionEvent event) {
               textField.setText("Goodbye, World!");
            }
         });
      frame.setLayout(new FlowLayout());
      frame.add(helloButton); frame.add(goodbyeButton); frame.add(textField);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.pack();
      frame.setVisible(true);
   }
}
```

Example: Making Buttons Work with UI Actions.

- How is this an example of the OBSERVER pattern?


AbstractButton
addActionListener(ActionListener)
fireActionPerformed(ActionEvent)
actionListeners
1
0..n
«Interface»
ActionListener
actionPerformed(ActionEvent)
JButton
the anonymous class

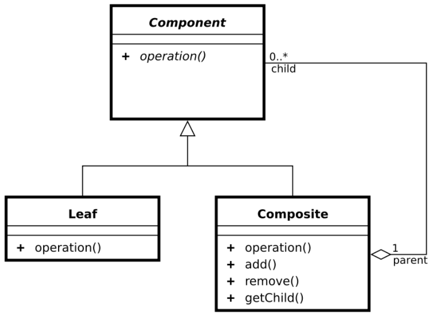
- Note:
  - ✦Does not use the Java Observer/Observable classes.
  - ✦The JButton has no state that the observer cares about.

Observer Pattern: consequences

- Decouples a Subject from the Observers. Subject knows only that it contains a list of Observers, each with an update() method. (The subject and observers can belong to different layers.)
- Observers can change or be added without changing Subject.
- Observers can ignore notifications (decision is not made by Subject).
- Can result in many spurious broadcasts (and calls to getState()) when the state of a Subject changes.

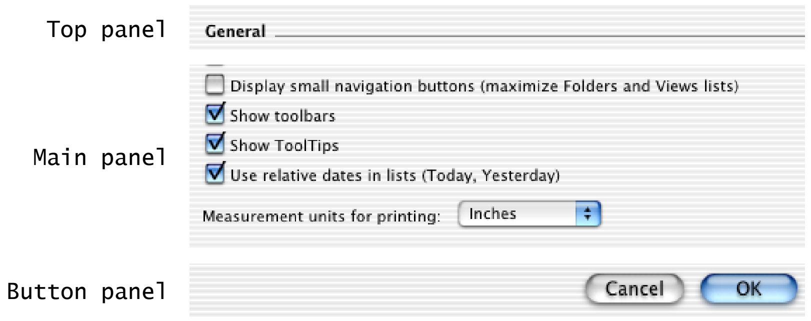
Encapsulating Hierarchies with the COMPOSITE Pattern

Name: Composite Design Pattern
Problem Description: Represent a hierarchy of variable width and depth so that leaves and composites can be treated uniformly through a common interface.
Solution: The **Component** interface specifies the services that are shared among Leaf and Composite (operation()). A **Composite** has an aggregation association with Components and implements each service by iterating over each contained Component. The **Leaf** services do most of the actual work.



Example: A hierarchy of user interface objects

- Anatomy of a preference dialog. Aggregates, called Panels, are used for grouping user interface objects that need to be resized and moved together.



General

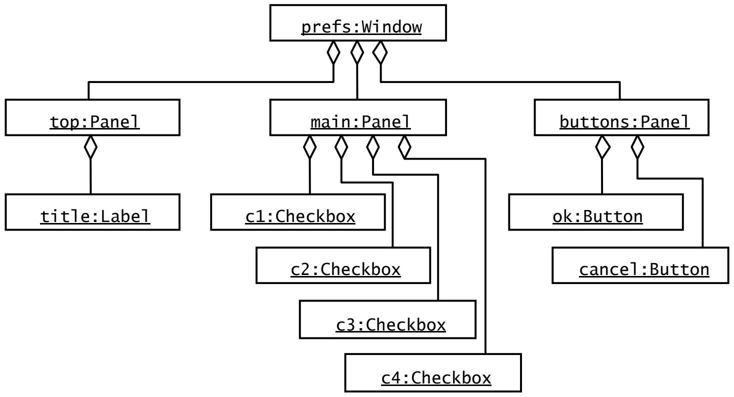
- [ ] Display small navigation buttons (maximize Folders and Views lists)
- [✓] Show toolbars
- [✓] Show ToolTips
- [✓] Use relative dates in lists (Today, Yesterday)

Measurement units for printing: Inches

Cancel OK

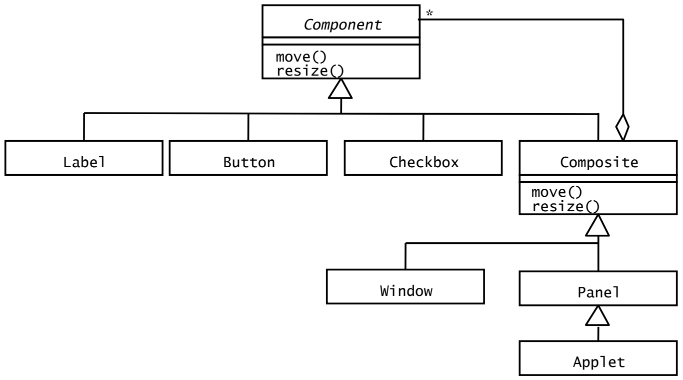
Example: A hierarchy of user interface objects

- An object diagram (it contains instances, not classes) of the previous example:



Example: A hierarchy of user interface objects

- A class diagram, for user interface widgets



Composite Pattern example: File system

```
//Component Node, common interface
interface AbstractFile {
    public void ls();
}

// File implements the common interface, a Leaf
class File implements AbstractFile {
    private String m_name;
    public File(String name) {
        m_name = name;
    }
    public void ls() {
        System.out.println(CompositeDemo.g_indent + m_name);
    }
}
```

Composite Pattern example: File system

```
// Directory implements the common interface, a composite
class Directory implements AbstractFile {
    private String m_name;
    private ArrayList<AbstractFile> m_files = new ArrayList<AbstractFile>();
    public Directory(String name) {
        m_name = name;
    }
    public void add(AbstractFile obj) {
        m_files.add(obj);
    }
    public void ls() {
        System.out.println(CompositeDemo.g_indent + m_name);
        CompositeDemo.g_indent.append("   ");  // add 3 spaces
        for (int i = 0; i < m_files.size(); ++i) {
            AbstractFile obj = m_files.get(i);
            obj.ls();
        }
         //remove the 3 spaces:
        CompositeDemo.g_indent.setLength(CompositeDemo.g_indent.length() - 3);
    }
}
```

Composite Pattern example: File system

```
public class CompositeDemo {
    public static StringBuffer g_indent = new StringBuffer();

    public static void main(String[] args) {
        Directory one = new Directory("dir111"),
                  two = new Directory("dir222"),
                  thr = new Directory("dir333");
        File a = new File("a"), b = new File("b"),
             c = new File("c"), d = new File("d"), e = new File("e");
        one.add(a);
        one.add(two);
        one.add(b);
        two.add(c);
        two.add(d);
        two.add(thr);
        thr.add(e);
        one.ls();
    }
}
```

Output:

```
dir111
   a
   dir222
      c
      d
      dir333
         e
   b
```

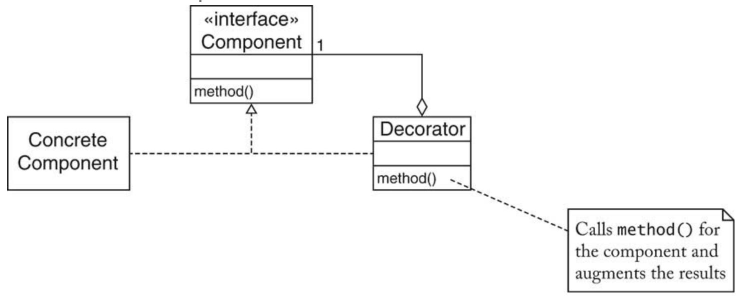
Composite Pattern: consequences

- Client uses the same code for dealing with Leaves or Composites
- Leaf-specific behavior can be modified without changing the hierarchy
- New classes of leaves (and composites) can be added without changing the hierarchy

- Could make your design too general. Sometimes you want composites to have only certain components. May have to add your own run-time checks.

Adding Behavior Dynamically with the DECORATOR Pattern

Name: Decorator Design Pattern
Problem Description: Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
Solution: The **Component** interface specifies the services for objects that can have responsibilities added to them. A **ConcreteComponent** is an independent object to which additional responsibilities can be attached. The **Decorator** conforms to the Component interface, and maintains a reference to a Component to which it adds responsibilities.



Example: Adding scrollbars to a UI Component

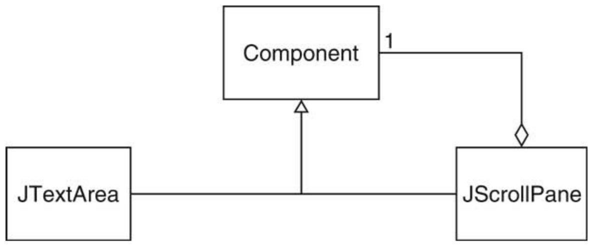
- When a component contains more information than can be shown on the screen, it becomes necessary to add scroll bars.
- This code adds scroll bars to a text area (a text area is a box where you can enter multiple lines of text):

```
JTextArea area = new JTextArea(20, 40); // 20 rows, 40 columns
JScrollPane scroller = new JScrollPane(area);
frame.add(scroller, BorderLayout.CENTER);
```

JScrollPane(Component view)
Creates a JScrollPane that displays the contents of the specified component, where both horizontal and vertical scrollbars appear whenever the component's contents are larger than the view.

Decorator Pattern: Scroll Bars

- The JScrollPane is an instance of the DECORATOR Pattern.
- This is not a hierarchy, the JScrollPane contains only 1 Component.
- The JScrollPane is itself a Component.
  - ✦It has the same methods as JTextArea, implemented by calling the method on JTextArea and modifying the result.
  - ✦It can be decorated by another JScrollPane

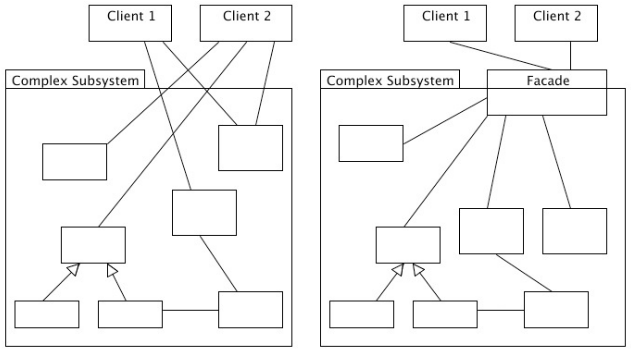


Decorator Pattern: consequences

- Component objects can be decorated (visually or behaviorally enhanced)
- The decorated object can be used in the same way as the undecorated object
- The component class does not want to take on the responsibility of the decoration (loose coupling!)
- There may be an open-ended set of different kinds of decorations.
- A decorated object can itself be decorated. And so on. And so on.

Encapsulating Subsystems with the FACADE Pattern

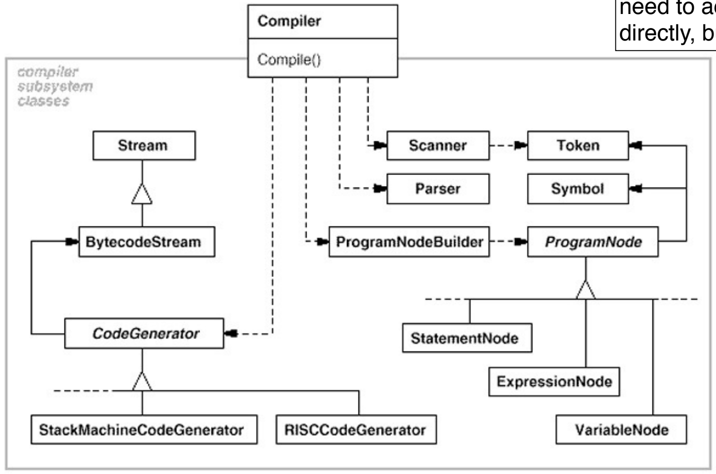
Name: Facade Design Pattern
Problem Description: Reduce coupling between a set of related classes and the rest of the system. Provide a simple interface to a complex subsystem.
Solution: A single **Facade** class implements a high-level interface for a subsystem by invoking the methods of lower-level classes. A Facade is opaque in the sense that a caller does not access the lower-level classes directly. The use of Facade patterns recursively yields a layered system.



Example: Compiler subsystem

- Compiler class is a facade hiding the Scanner, Parser, ProgramNodeBuilder and CodeGenerator.

Some specialized apps might need to access the classes directly, but most don't.



Facade Pattern: consequences

- Shields a client from the low-level classes of a subsystem.
- Simplifies the use of a subsystem by providing higher-level methods.
- Promotes "looser" coupling between subsystems.

- Note the use of delegation to reduce coupling.
- Note the similarity to the Controller GRASP pattern.

Heuristics for Selecting Design Patterns

- Use key phrases from design goals to help choose pattern

Phrase	**Design Pattern**
"Must support aggregate structures" "Must allow for hierarchies of variable depth and width"	Composite
"Must have mechanism to process aggregate structure" "Must support multiple traversals at same time"	Iterator
"Must comply with existing interface" "Must reuse existing legacy component"	Adapter
"Must be notified of changes"	Observer
"Must allow functionality to be added during runtime" "Client programs must be able to add functionality"	Decorator
"Policy and mechanisms should be decoupled" "Must allow different algorithms to be interchanged at runtime"	Strategy