The Composite Pattern

Human Computer Interaction Research
University of Nevada, Reno

Iterator Pattern

★ Structural Patterns
  » adapter
  » façade
  » composite

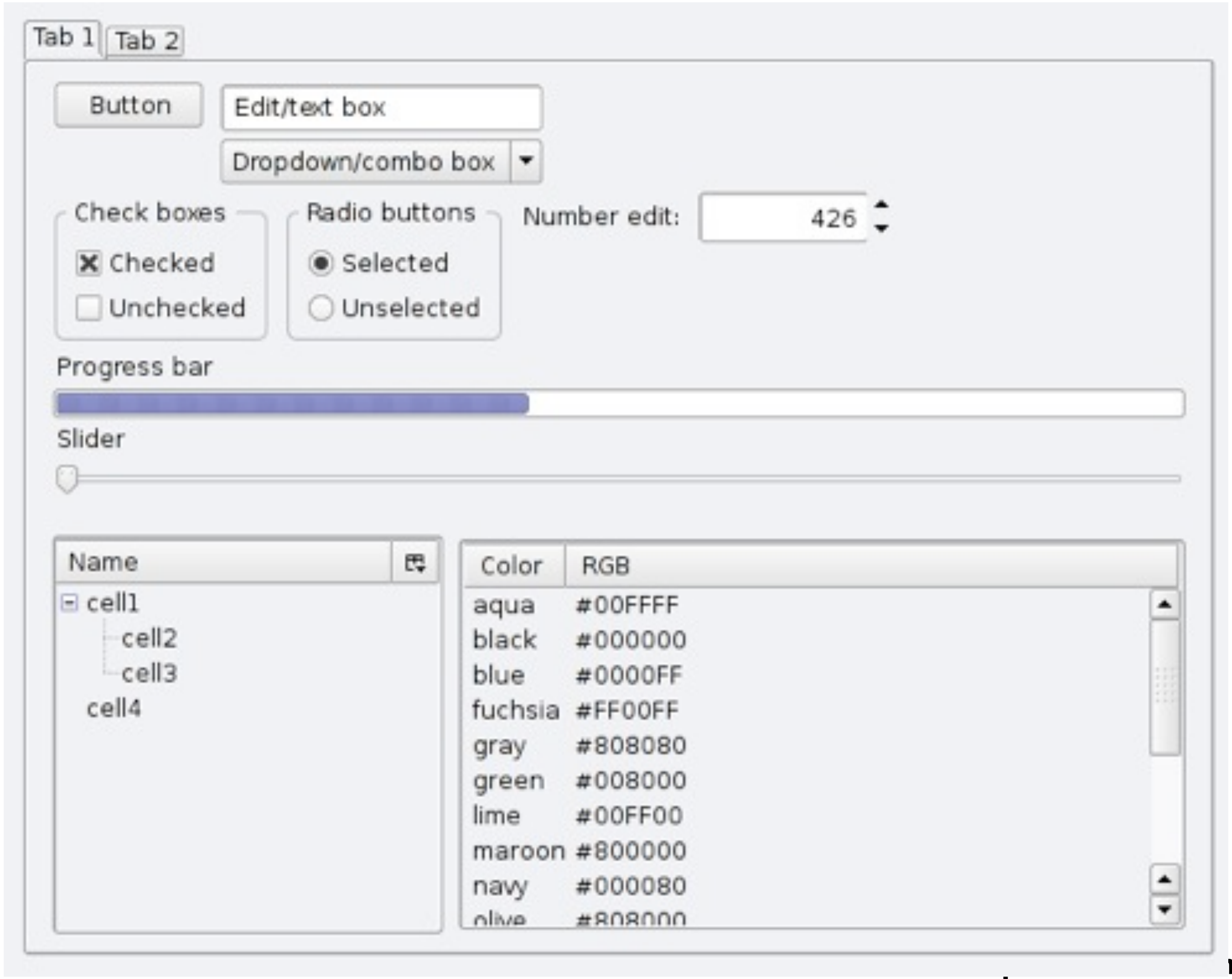
★ Creational Patterns
  » factory method
  » abstract factory
  » singleton

★ Behavioral Patterns
  » strategy
  » observer
  » decorator
  » command
  » template method
  » iterator

Problem

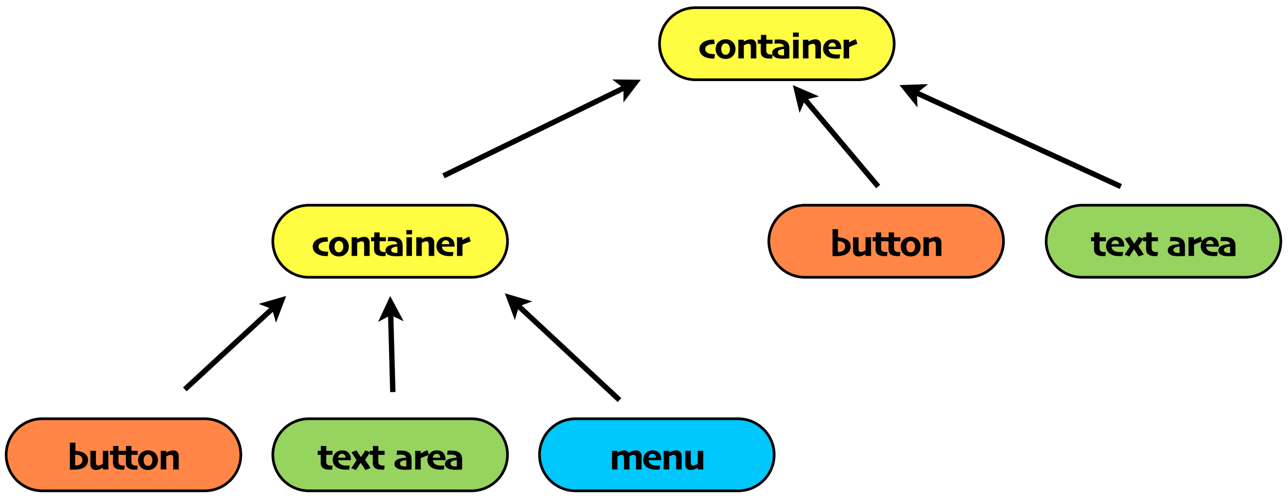
When dealing with tree-structured data, programmers often have to discriminate between a leaf-node and a branch. This makes code more complex, and therefore, error prone.

example GUI's



Color	RGB
aqua	#00FFFF
black	#000000
blue	#0000FF
fuchsia	#FF00FF
gray	#808080
green	#008000
lime	#00FF00
maroon	#800000
navy	#000080
olive	#808000

widget


container
container
button
text area
button
text area
menu

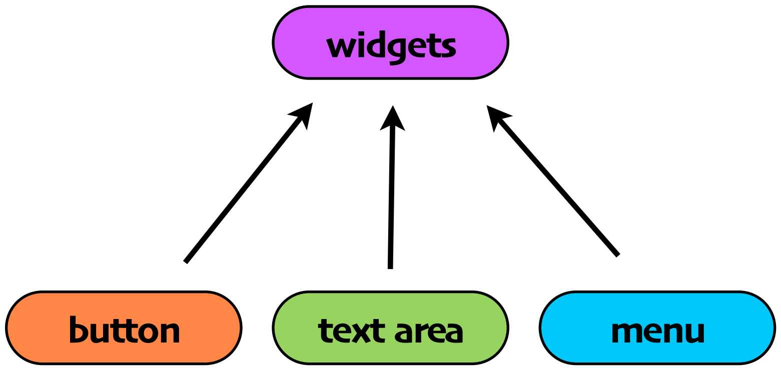
Possible implementation

pretty ugly

```
public class Window {
Button[] buttons;
Menu[] menus;
TextArea[] textAreas;
WidgetContainer[] containers;

public void update() {
if (buttons != null) {
         for (int k = 0; k < buttons.length; k++) buttons[k].draw();
     }
}
if (menus != null) for (int k = 0; k < menus.length; k++) {
         menus[k].refresh();
}
if (containers!=null) {
      for (int k = 0; k < containers.length; k++ ) {
        containers[k].updateWidgets();
}
```

"Classes should be open for extension, but closed for modification"

Abstraction



Refactor

```
public class Window {
Widget[] widgets;
WidgetContainer[] containers;


public void update() {

if (widgets != null) for (int k = 0; k < widgets.length; k++) {
                widgets[k].update();
}

if (containers != null) {
           for (int k = 0; k < containers.length; k++ ) {
                containers[k].updateWidgets();
           }
 }
```

"program to an interface"

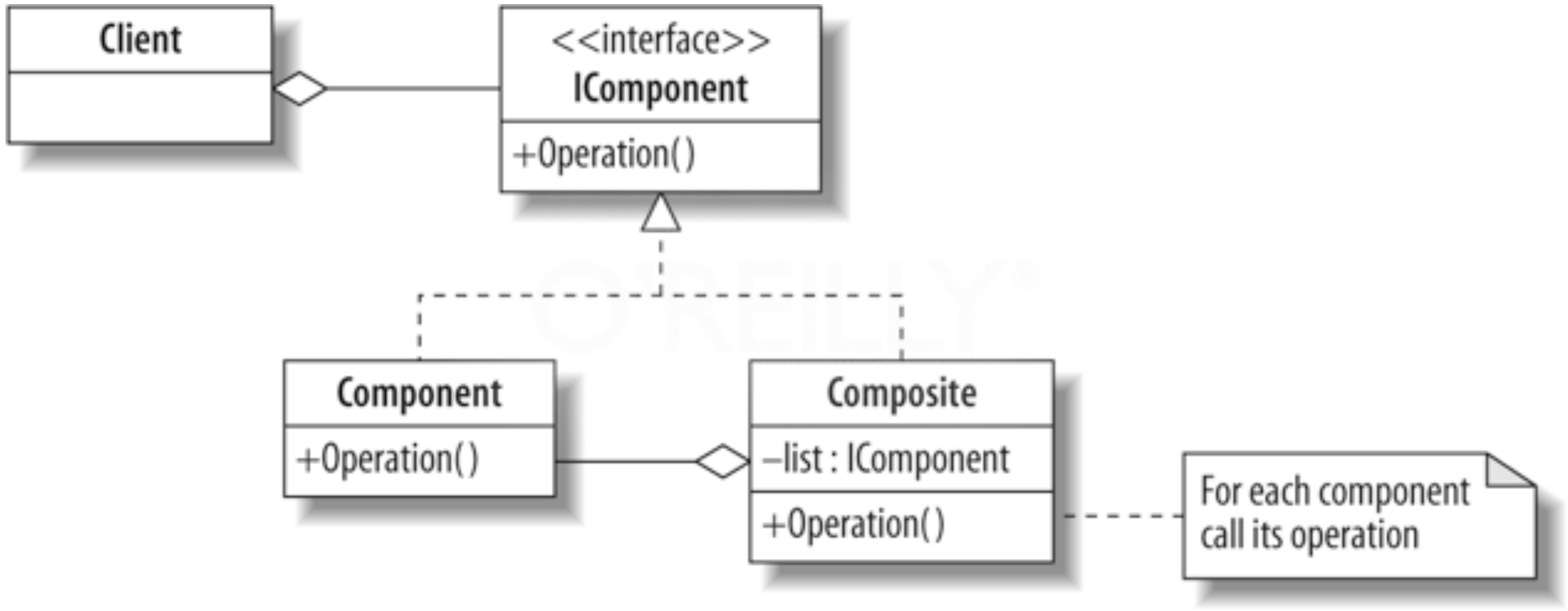
all widgets support update()

we still distinguish between containers and widgets

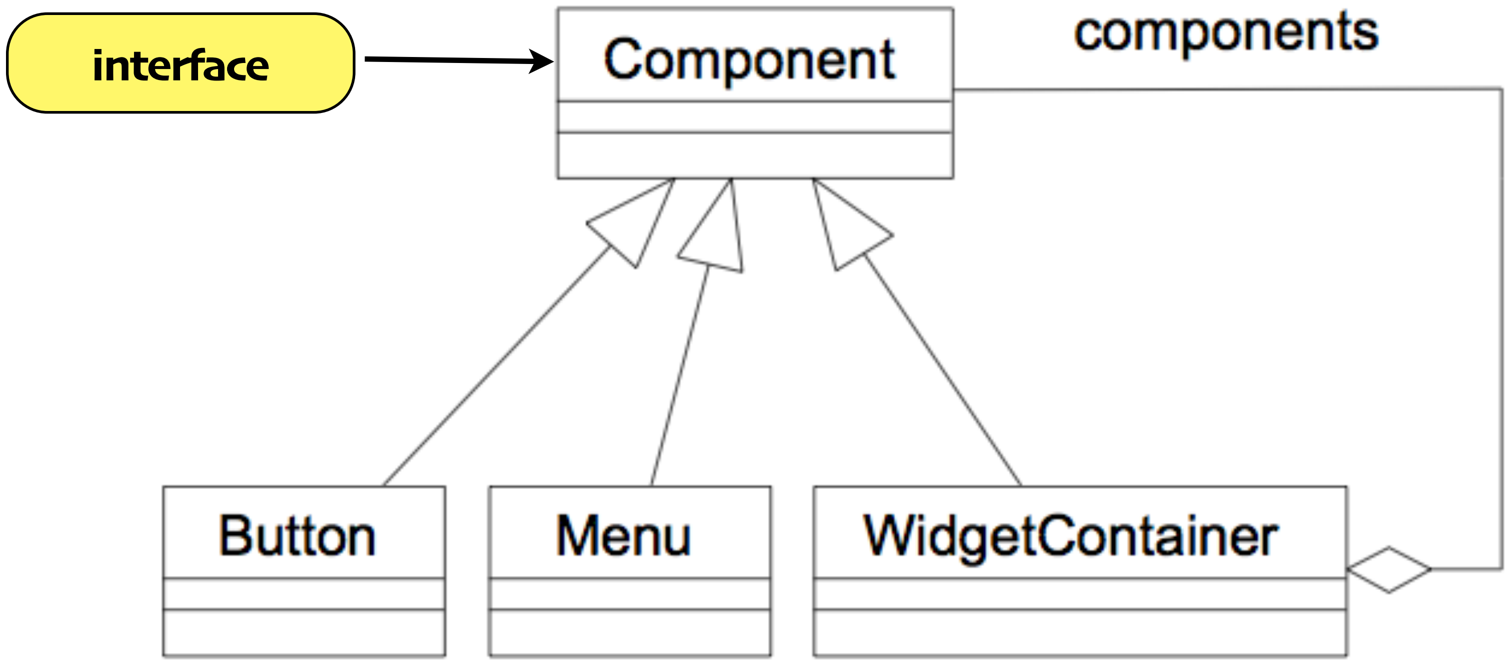
Composite Pattern

The Composite Pattern allows you to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly

Class Diagram



for our GUI


interface
Component
components
Button
Menu
WidgetContainer

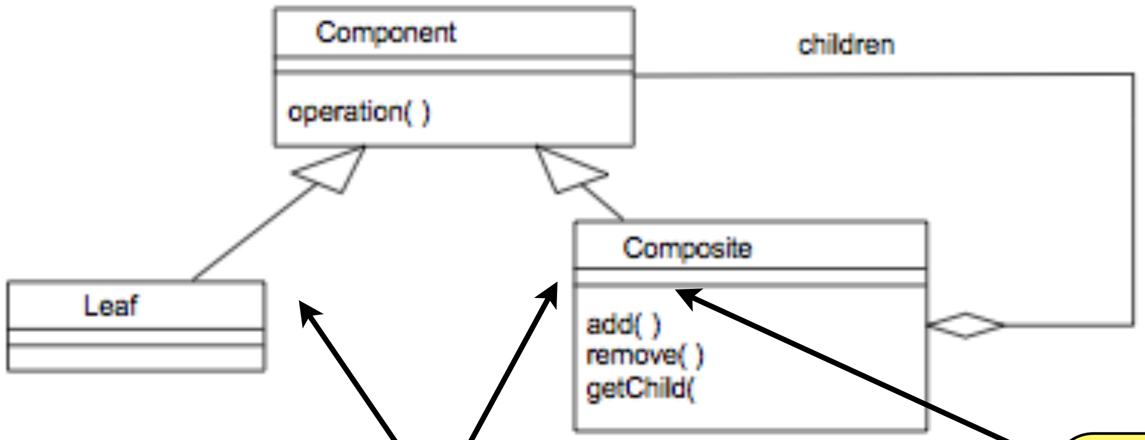
```
public class Window {
    Component[] components;

public void update() {
    if (components != null) {
            for (int k = 0; k < components.length; k++) {
             components[k].update();
        }
    }
}
```

Implementation issues

- ★ **Where should the child management methods (add(), remove(), getChild()) be declared?**

In Composite?

safer



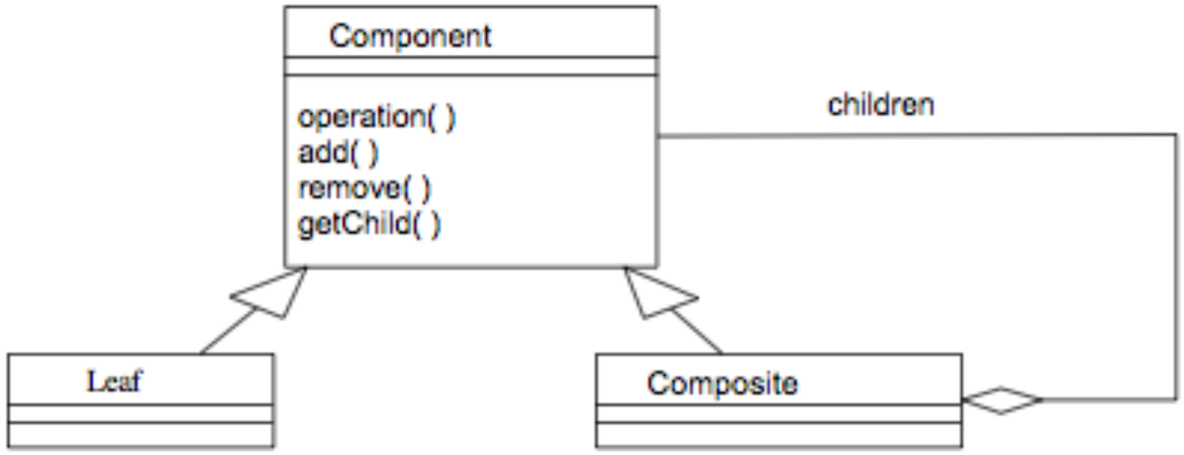
different interfaces

keep the list here

A matter of tradeoffs

In Component?

Transparent



unsafe

"A Class should have only one reason to change"

Internal Iterator

```
public void doSomething() {
 throw new UnsupportedOperationException();
}
```

composite

ugly but safe

```
public void doSomething() {
// do something
}
```

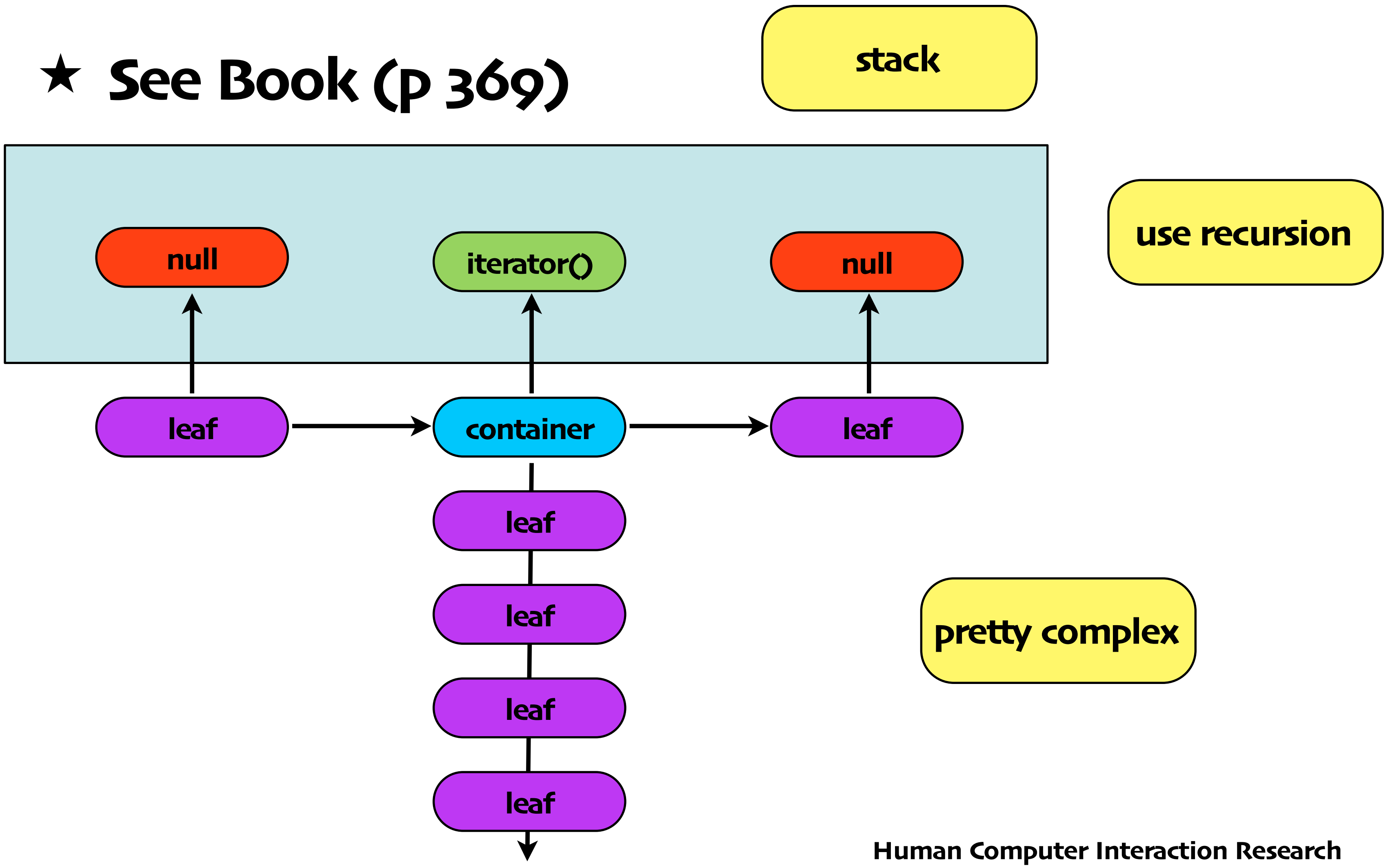
leaf

```
public void doSomething() {
// do something
Iterator iterator = menuComponents.iterator();
while (iterator.hasNext()) {
  Component component = (Component) iterator.next();
  component.doSomething();
  }
}
```

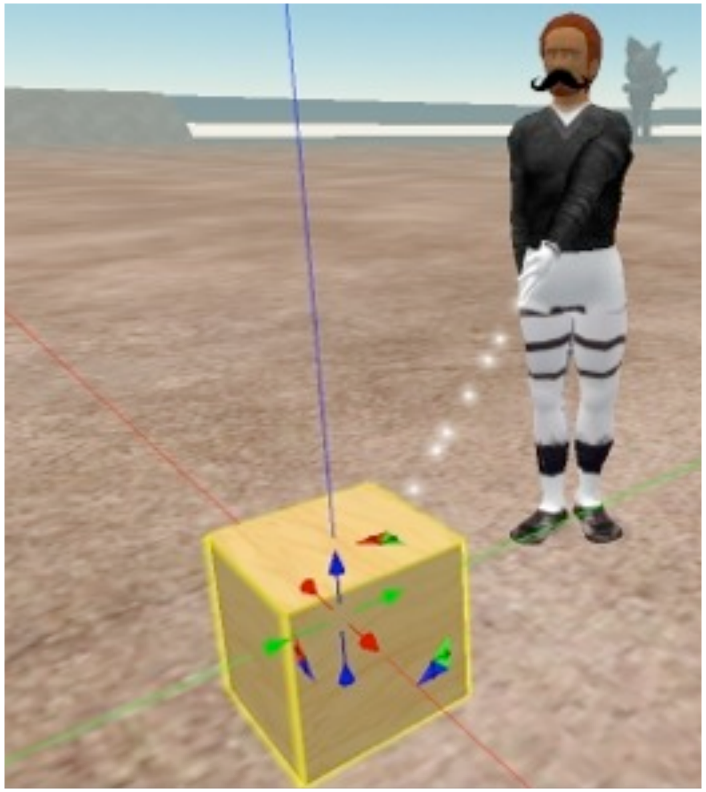
Composite

External Iterator

- See Book (p 369)



exercise



```java
import java.util.Iterator;

public interface Prim {
   public void render();
   public float volume();
   public Iterator createIterator();
}
```

Composite

```
import java.util.ArrayList;
import java.util.Iterator;

public class Prim_composite implements Prim {
                Iterator iterator=null;

                ArrayList<Prim> child_components = new ArrayList<Prim>();

                public void render() {
        for (Prim prim : child_components) {
            prim.render();
        }
                }

                public float volume() {
                                float total = 0;
                                 for (Prim prim : child_components) {
                            total+=prim.volume();
                        }
                                return total;
                }
```

Composite II

```
 //Adds the graphic to the composition.
    public void add(Prim graphic) {
    child_components.add(graphic);
    }

    //Removes the graphic from the composition.
    public void remove(Prim graphic) {
    child_components.remove(graphic);
    }

    public Iterator createIterator() {
        if (iterator==null) {
            iterator = new CompositeIterator(child_components.iterator());
        }
        return iterator;
    }

}
```

Sphere

```
import java.util.Iterator;

public class Sphere implements Prim {

     private float radius;

     public Sphere(){
       radius=1.0f;
     }

     public void render() {
          System.out.println("Sphere R:"+ radius);
     }

     public float volume() {
          return (float) (4/3 * Math.PI*radius*radius*radius);
     }

     public Iterator createIterator() {
          return new NullIterator();
     }

}
```

Cube

```
import java.util.Iterator;

public class Cube implements Prim {

    private float width;
    private float height;
    private float depth;

    public Cube(){
      width=height=depth=1.0f;
    }

    public void render() {
        System.out.println("Cube W:"+ width + " H:" + height + " D:" + depth);
    }

    public float volume() {
        return width*height*depth;
    }

    public Iterator createIterator() {
        return new NullIterator();
    }

}
```

Iterator

```
import java.util.Iterator;
import java.util.Stack;

public class CompositeIterator implements Iterator {
        Stack stack = new Stack();

        public CompositeIterator(Iterator iterator) {
                stack.push(iterator);
        }
        public boolean hasNext() {
                if (stack.empty()) {
                        return false;
                }
                else {
                        Iterator iterator = (Iterator) stack.peek();
                        if (!iterator.hasNext()) {
                                stack.pop();
                                return hasNext();
                        }
                        else {
                                return true;
                        }
                }
        }
```

Iterator 2

```
public Prim next() {
        if (hasNext()) {
                Iterator iterator = (Iterator) stack.peek();
                Prim prim = (Prim) iterator.next();
                if (prim instanceof Prim_composite) {
                        stack.push(prim.createIterator());
                }
                return prim;
        }
        else {
                return null;
        }
}

public void remove() {
        throw new UnsupportedOperationException();

}
```

null iterator

```
import java.util.Iterator;


public class NullIterator implements Iterator {

    public boolean hasNext() {
        return false;
    }

    public Object next() {
        return null;
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }

}
```