Composite Pattern

- Structural Patterns
  - strategy
  - adapter
  - façade
  - composite
- Creational Patterns
  - factory method
  - abstract factory
  - singleton
- Behavioral Patterns
  - observer
  - decorator
  - command
  - template method
  - iterator

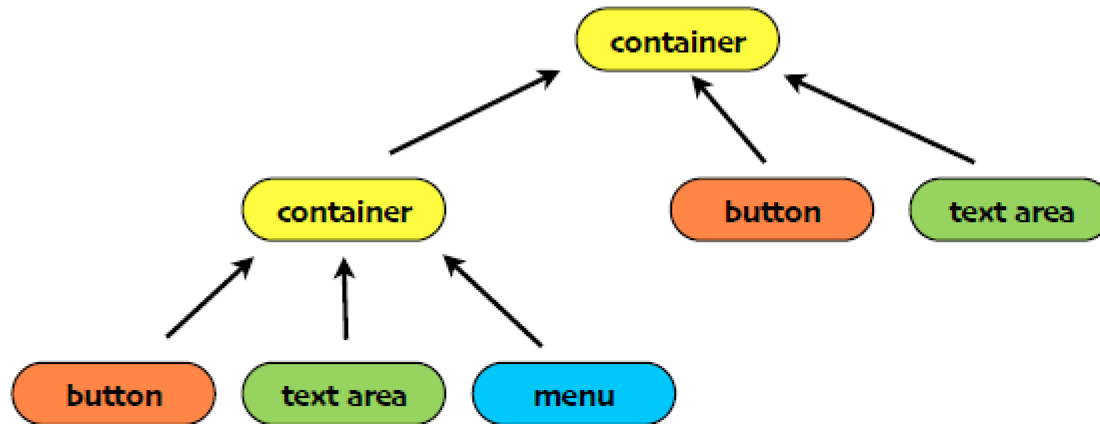
Composite Pattern Definition

Allows you to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and composition of objects uniformly.

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.

GUI Example



Possible Implementation

pretty ugly

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

```
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();
}
```

Refactor!

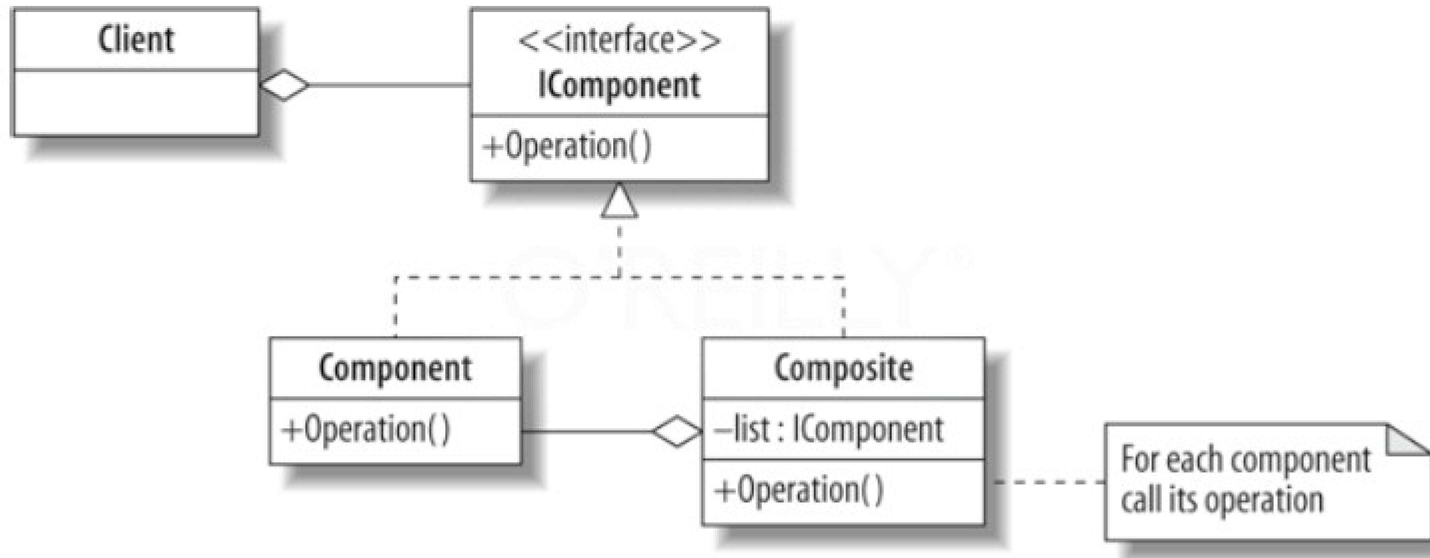
```
public class Window {
Widget[] widgets;
WidgetContainer[] containers;


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"

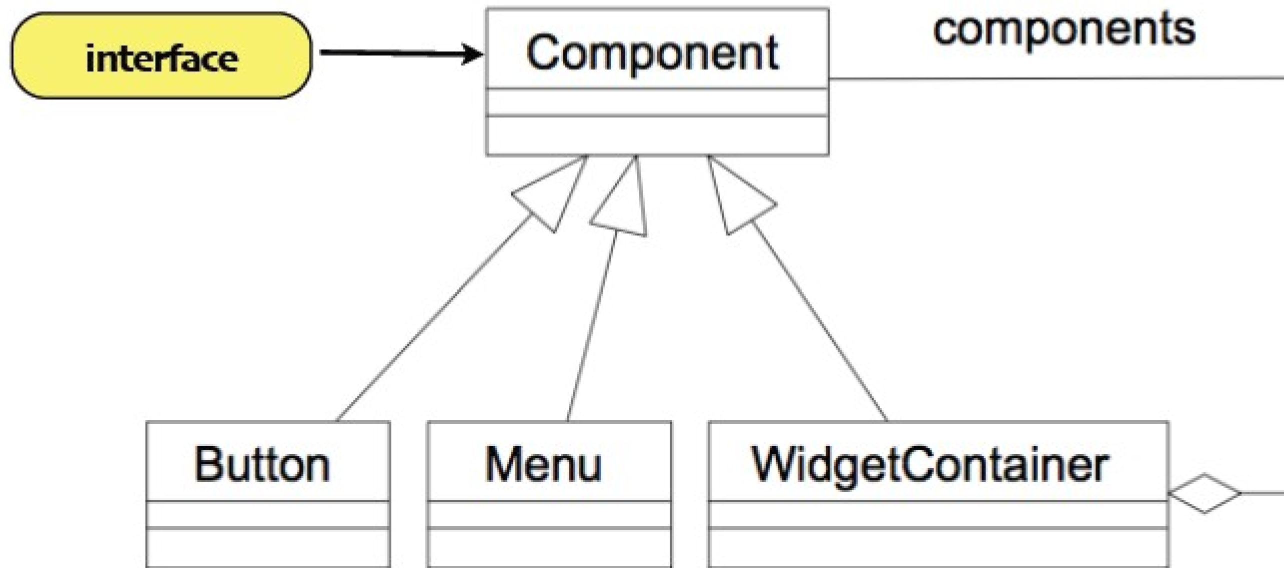
all widgets support update()

we still distinguish between containers and widgets

Class Diagram



For the GUI Example



Associated Code

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

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

Component Interface Features

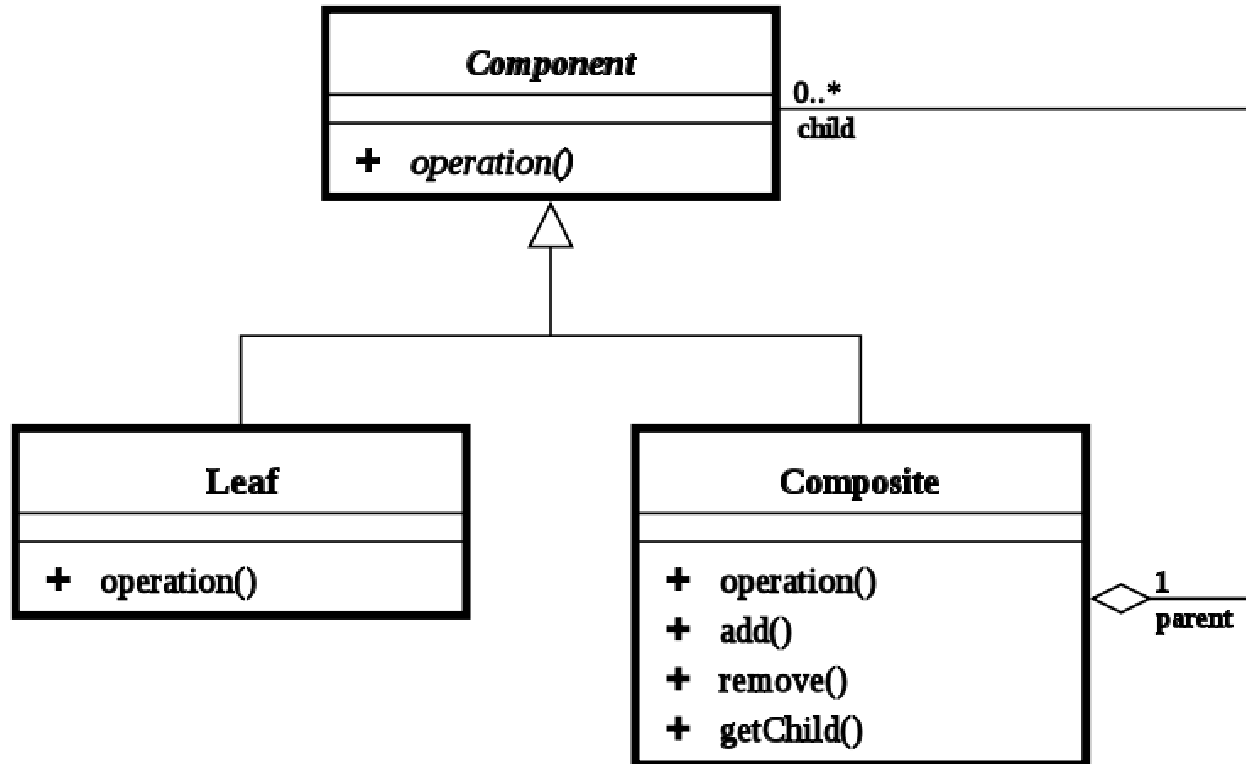
- Operation(): what each component needs to do
- Add(Component)
- Remove(Component)
- getChild(int)

>default behavior can be provided for Add, Remove, and getChild

Composite

- Defines the behavior of the components having children
- Stores child components
- Implements leaf related operations – if this does not make sense on the Composite, and exception may be generated (e.g. UnsupportedOperationException)

Another class diagram (from Wikipedia)



Thoughts on Previous Diagram

- Can place add, remove, getChild only in Composite if you want to limit access to those behaviors
- <u>Component</u> is the abstraction for all components; declares an interface for objects in the composition
- <u>Leaf</u> represents the leaf objects in the composition; implements all component methods
- <u>Composite</u> represents a component having children; implements methods to manipulate children; implements all component methods typically by delegating to children

Java Example: Graphic Class

Algebraic form:

Graphic = ellipse | GraphicList
GraphicList = empty | Graphic GraphicList

'Graphic' Example

```
import java.util.List;
import java.util.ArrayList;

/** "Component" */
interface Graphic {

    //Prints the graphic.
    public void print();
}
```

Graphic Example

```
/** "Composite" */
class CompositeGraphic implements Graphic {

    //Collection of child graphics.
    private List<Graphic> mChildGraphics = new ArrayList<Graphic>();

    //Prints the graphic.
    public void print() {
        for (Graphic graphic : mChildGraphics) {
            graphic.print();        }    }

    //Adds the graphic to the composition.
    public void add(Graphic graphic) {
        mChildGraphics.add(graphic);    }

    //Removes the graphic from the composition.
    public void remove(Graphic graphic) {
        mChildGraphics.remove(graphic);    }}
```

Graphic Example

```
/** "Leaf" */
class Ellipse implements Graphic {

    //Prints the graphic.
    public void print() {
        System.out.println("Ellipse");
    }
}
```

Graphic Example

```
/** Client */
public class Program {

    public static void main(String[] args) {
        //Initialize four ellipses
        Ellipse ellipse1 = new Ellipse();
        Ellipse ellipse2 = new Ellipse();
        Ellipse ellipse3 = new Ellipse();
        Ellipse ellipse4 = new Ellipse();

        //Initialize three composite graphics
        CompositeGraphic graphic = new CompositeGraphic();
        CompositeGraphic graphic1 = new CompositeGraphic();
        CompositeGraphic graphic2 = new CompositeGraphic();

        //Composes the graphics
        graphic1.add(ellipse1);
        graphic1.add(ellipse2);
        graphic1.add(ellipse3);

        graphic2.add(ellipse4);

        graphic.add(graphic1);
        graphic.add(graphic2);

        //Prints the complete graphic (four times the string "Ellipse").
        graphic.print();    }}
```

Composite Iterator

- Compound Pattern
- Add a createIterator in every Component
- It makes sure you can iterate over all items in your component – including children
- Recursive in nature
- CompositeIterator will need to implement Iterator interface
- Can incorporate a NullIterator for items that have nothing to iterate over (this is an example of the Null Object pattern, which gives us an object so we don't have to worry about watching for null in our code)

Another Example – Composite Iterator

```
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

```
//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;
}
}//end class
```

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();
	}
}//end class
```

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();
     }
}//end class
```

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;
				}
		}
	}//end hasNext
```

Iterator

```
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();
    }
}//end class
```

Tester

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

public class test_prims {

   public static void main(String[] args) {
        Cube cube1 = new Cube(1.0f,1.0f,1.0f);
        Cube cube2 = new Cube(1.0f,1.0f,1.0f);
        Sphere sphere1 = new Sphere(4.0f);
       // Cylinder cylinder1 = new Cylinder();

        //Initialize three composite prims
        Object pcom1 = new Object();
        Object pcom2 = new Object();

         pcom1.add(cube1);
         pcom1.add(cube2);

         pcom2.add(pcom1);
         pcom2.add(sphere1);


         pcom2.render();
         System.out.println(pcom2.volume());

         Iterator iterator = pcom2.createIterator();
         while (iterator.hasNext()) {
            ((Object_Component) iterator.next()).render();
          }
        }
}
```

Composite Summary

- Provides a structure to hold both individual objects and composites (think tree data structure)
- Allows clients to treat composites and individual objects uniformly (decoupling)
- A Component is any object in a Composite structures that may be other composites or leaf nodes