The Composite Pattern

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

---

**Composite Pattern: Motivation**

- A Composite is an object designed as a composition of one-or-more similar objects (components) all exhibiting similar functionality.
- The key concept is that you can manipulate a single instance of the object just as you would a group of them.
- The Composite pattern is useful in designing a common interface for both individual and composite components so that client programs can view both the individual components and groups of components uniformly.
- Use to:
  - build part-whole hierarchies;
  - construct data representation of trees;
  - when you want your clients to ignore the difference between compositions of objects and individual objects.
The Composite Pattern Defined

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

1. We can create arbitrarily complex trees.
2. We can treat them as a whole or as parts.
3. Operations can be applied to the whole or the part.

Composite Pattern Structure

The client uses the Component interface to manipulate the objects in the composition. The Component defines an interface for all objects in the composition: both the composite and the leaf nodes. The Component may implement a default behavior for add(), remove(), getChild() and its operations.

Note that the leaf also inherits methods like add(), remove(), and getChild(), which don't make a lot of sense for a leaf node. We will come back to this issue.

A Leaf defines the behavior for the elements in the composition. It does this by implementing the operations the Composite supports.

The Composite's role is to define the behavior of the components having children and to store child components.

The Composite also implements the Leaf-related operations. Note that some of these may not make sense on a Composite so in that case an exception might be generated.
Composite Pattern Participants

- **Component:**
  - declares the interface for object composition
  - implements default behaviour
  - declares an interface for accessing and managing the child components

- **Leaf:**
  - represents leaf objects in the composition

- **Composite:**
  - defines behaviour for components having children
  - stores child components
  - implements child-related operations to the Component interface

- **Client:**
  - manipulates objects in the composition through the Composite interface

Composite Pattern: Structural Code

```java
import java.util.*;

interface Component {
    public String defaultMethod();
    public ArrayList<Component> getChildren();
    public boolean addComponent(Component c);
    public boolean removeComponent(Component c);
}
```
class Composite implements Component {
    private String id;

    private ArrayList<Component> components = new ArrayList<Component>();

    public Composite(String identification) {
        id = identification;
    }

    public String defaultMethod() {
        String s = "(" + id + ":";
        for (Component child : getChildren())
            s = s + " " + child.defaultMethod();
        return s + ");";
    }

    public ArrayList<Component> getChildren(){
        return components;
    }

    public boolean addComponent(Component c){
        return components.add(c); }

    public boolean removeComponent(Component c){
        return components.remove(c); }
}

class Leaf implements Component {
    private String id;
    public Leaf(String identification) { id = identification; }
    public String defaultMethod() { return id; }
    public ArrayList<Component> getChildren() { return null; }
    public boolean addComponent(Component c) { return false; }
    public boolean removeComponent(Component c) { return false; }
}

class CompositePattern {
    public static void main(String[] args) {
        Composite england = new Composite("England");
        Leaf york = new Leaf("York");
        Leaf london = new Leaf("London");
        england.addComponent(york);
        england.addComponent(london);
        england.removeComponent(york);
        Composite france = new Composite("France");
        france.addComponent(new Leaf("Paris"));
        Composite europe = new Composite("Europe");
        europe.addComponent(england);
        europe.addComponent(france);

        System.out.println( europe.defaultMethod() );
    }
}
Composite Pattern: Reasons/Consequences

- Defines class hierarchies

- Simple Clients
  - Doesn’t know/care whether dealing with leaf or composite class
  - Uniform treatment of composite & individual object

- Easier to add new components
  - New composites or leaves work automatically

- Can make design over general
  - Sometimes want a composite to have only certain components
    - Need run-time checks

- Sharing components to reduce space when there is more than one parent is difficult

Composite Pattern: Implementation issues

A composite object knows its contained components.

- Should components maintain a reference to their parent component?
  Depends on application

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

- In the Component class: gives transparency, since all components can be treated the same. But it's not safe, since clients can try to do meaningless things to leaf components at run-time.

- In the Composite class: gives safety, since any attempt to perform a child operation on a leaf component will be caught at compile-time. But we lose transparency, since now leaf and composite components have different interfaces.
Composite Pattern: Transparent vs. Safe

Transparent

Safe

Composite Pattern: Implementation issues

Should Component maintain the list of components that will be used by a composite object?

That is, should this list be an instance variable of Component rather than Composite?

- Better to keep this part of Composite and avoid wasting the space in every leaf object

Is child ordering important?

- Depends on application

Who should delete components?

- Not a problem in Java! The garbage collector will come to the rescue!

What's the best data structure to store components?

- Depends on application
Composite Pattern: Example

A GUI system has window objects which can contain various GUI components (widgets) such as, buttons and text areas. A window can also contain widget container objects which can hold other widgets.

Solution 1: We designed all the widgets with different interfaces for "updating" the screen. We would then have to write a Window update() method as follows:

```java
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();
        // Other widgets handled similarly.
        if (containers != null)
            for (int k = 0; k < containers.length; k++)
                containers[k].updateWidgets();
    }
}
```

It violates the Open-Closed Principle: if we want to add a new kind of widget, we have to modify the update() method of Window to handle it.

Solution 2: We should always try to program to an interface, right? So, let's make all widgets support the Widget interface, either by being subclasses of a Widget class or implementing a Java Widget interface.

Now our update() method becomes:

```java
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();
    }
}
```

That looks better, but we are still distinguishing between widgets and widget containers.
Solution 3: The Composite Pattern!

Now the update method looks like:

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

Summary

- The Composite Pattern provides a structure to hold both the individual objects and composites.

- The Composite Pattern allows clients to treat composites and individual objects uniformly.

- A Component is any object in a composite structure. Components may be other composites or leaf nodes.

- There are many design tradeoffs in implementing the Composite. You need to balance transparency and safety with your needs.