

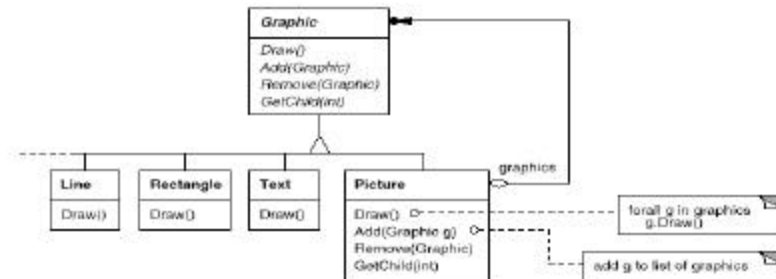
# The Composite Pattern

Design Patterns In Java

Bob Tarr

## The Composite Pattern

- Intent
  - ⇒ Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. This is called *recursive composition*.
- Motivation



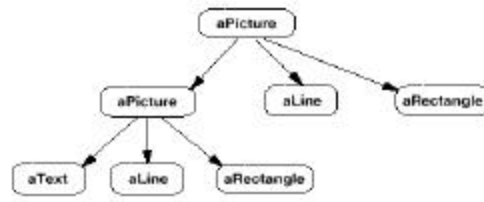
Design Patterns In Java

The Composite Pattern  
2

Bob Tarr

## The Composite Pattern

- Motivation



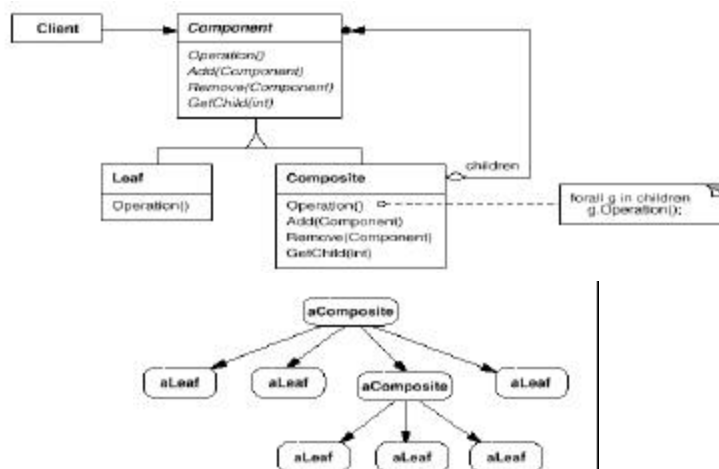
- Applicability

Use the Composite pattern when

- ⇒ You want to represent part-whole hierarchies of objects
- ⇒ You want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

## The Composite Pattern

- Structure



## The Composite Pattern

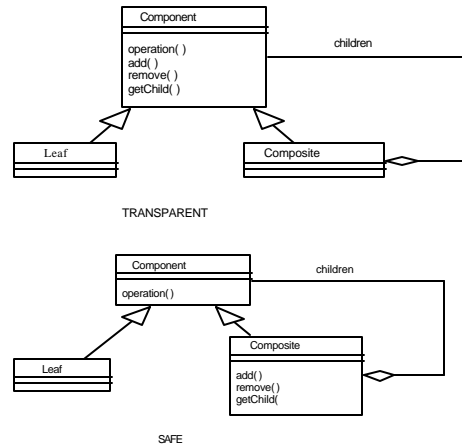
- Consequences
  - ⇒ Benefits
    - It makes it easy to add new kinds of components
    - It makes clients simpler, since they do not have to know if they are dealing with a leaf or a composite component
  - ⇒ Liabilities
    - It makes it harder to restrict the type of components of a composite

## The Composite Pattern

- Implementation Issues
  - ⇒ A composite object knows its contained components, that is, its children. Should components maintain a reference to their parent component?
    - Depends on application, but having these references supports the Chain of Responsibility pattern
  - ⇒ 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: Give 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.

## The Composite Pattern

- Transparent vs. Safe



Design Patterns In Java

The Composite Pattern

7

Bob Tarr

## The 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

Design Patterns In Java

The Composite Pattern

8

Bob Tarr

## Composite Pattern Example 1

- Situation: 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: What if we designed all the widgets with different interfaces for "updating" the screen? We would then have to write a Window update() method as follows:

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

## Composite Pattern Example 1 (Continued)

```
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();  
}  
...  
}
```

- Well, that looks particularly bad. 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.

### Composite Pattern Example 1 (Continued)

- 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:

```
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();  
    }  
}
```

Design Patterns In Java

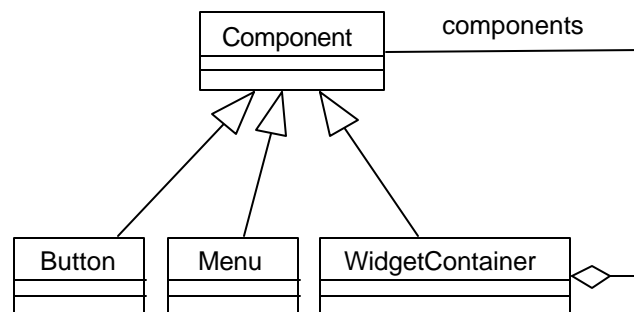
The Composite Pattern

11

Bob Tarr

### Composite Pattern Example 1 (Continued)

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



Design Patterns In Java

The Composite Pattern

12

Bob Tarr

## Composite Pattern Example 1 (Continued)

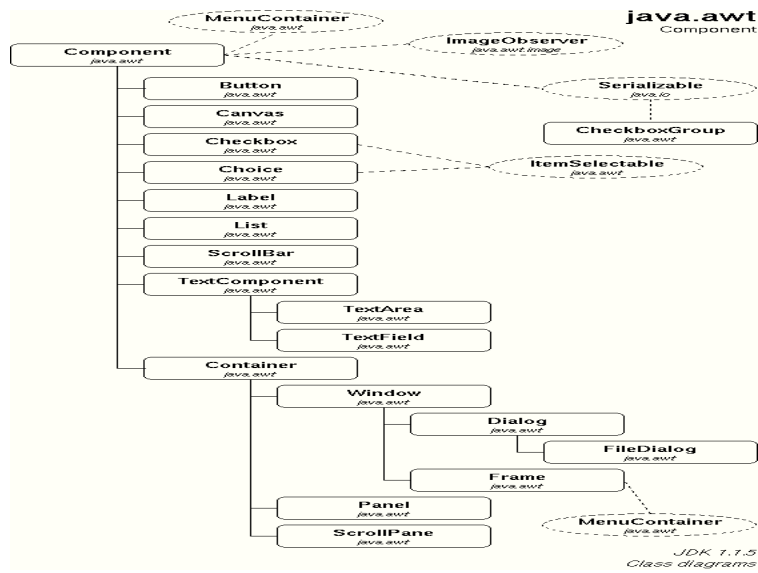
- Now the update method looks like:

```

public class Window {
    Component[] components;

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

## Composite Pattern Example 2 - The Java AWT



### Composite Pattern Example 3

- Situation: Many types of manufactured systems, such as computer systems and stereo systems, are composed of individual components and sub-systems that contain components. For example, a computer system can have various chassis that contain components (hard-drive chassis, power-supply chassis) and busses that contain cards. The entire system is composed of individual components (floppy drives, cd-rom drives), busses and chassis.

### Composite Pattern Example 3 (Continued)

- Solution: Use the Composite pattern!

