Hey, I hope you enjoyed the great free pull products. Let's prepare top programming skills for the post-COVID life. Refactoring UI

Composite

Also known as: Object tree

Intent

Composite is a structural design pattern that lets you compose objects into tree structures, and then work with these structures as if they were individual objects.

Problem

Using the Composite pattern makes sense only when the core model of your app can be represented as a tree.

For example, imagine that you have two types of objects: `Product`s and `Box`es. A `Box` can contain several `Product`s as well as a number of smaller `Box`es. These little `Box`es can also hold some `Product`s or even smaller `Box`es, and so on.

Say you decide to create an ordering system that uses these classes. Orders could contain simple products without any wrapping, as well as boxes stuffed with products, and other boxes. How would you determine the total price of such an order?

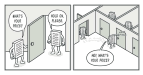
An order might comprise various products, packaged in boxes, which in turn might be placed in bigger boxes and so on. The whole structure looks like an upside-down tree.

You could try the direct approach: unwrap all the boxes, go over all the products and then calculate the total. That would be doable in the real world but in a program, it's not as simple as running a loop. You have to know the classes of `Product`s and `Box`es, you're going through, the nesting levels of the boxes and other nasty details beforehand. All of this makes the direct approach either too awkward or even impossible.

Solution

The Composite pattern suggests that you work with `Product`s and `Box`es through a common interface which declares a method for calculating the total price.

How would this method work? For a product, it's easy to return the product's price. For a box, it's go over each item the box contains, ask its price and then return a total price for this box. If one of these items is a smaller box, that box also knows that it contains some items and so on, until the prices of all inner components were calculated. A box could even add some extra cost to the final price, such as packaging cost.



The Composite pattern lets you run a behavior recursively over all components of an object tree.

The greatest benefit of this approach is that you don't need to care about the concrete classes of objects that compose the tree. You don't need to know whether an object is a simple product or a sophisticated box. You can treat them all the same via the common interface. When you call a method, the objects themselves pass the request down the tree.

Real-World Analogy

An example of a military division.

Armies of most countries are structured as hierarchies. An army consists of several divisions, a division is a unit of brigades, and a brigade consists of platoons, which can be broken down into squads. Finally, a squad is a small group of real soldiers. Orders are given at the top of the hierarchy and passed down onto each level until every soldier knows what needs to be done.

Structure

1. The **Component** interface describes operations that are common to both simple and complex elements of the tree.
2. The **Leaf** is a basic element of a tree that doesn't have sub-elements.

   Usually, leaf components end up doing most of the real work, since they don't have anyone to delegate the work to.
3. The **Container** (aka composite) is an element that has sub-elements: leaves or other containers. A container doesn't know the concrete classes of its children. It works with all sub-elements only via the component interface.

   Upon receiving a request, a container delegates the work to its sub-elements, processes intermediate results and then returns the final result to the client.
4. The **Client** works with all elements through the component interface. As a result, the client can work in the same way with both simple or complex elements of the tree.

Pseudocode

In this example, the **Composite** pattern lets you implement stacking of geometric shapes in a graphical editor.

The pattern's class diagram.

The `CompositeGraphic` class is a container that can comprise any number of sub-shapes, including other compound shapes. A compound shape has the same methods as a simple shape. However, instead of doing something on its own, a compound shape passes the request recursively to all its children and "sums up" the result.

The client code works with all shapes through the single interface common to all shape classes. Thus, the client doesn't know whether it's working with a simple shape or a compound one. The client can work with very complex object structures without being coupled to concrete classes that form that structure.

```
// The component interface declares common operations for both
// simple and complex objects of a composition.
interface Graphic {
    method move(x, y)
    method draw()
}

// The leaf class represents end objects of a composition. A
// primitive shape can't have any sub-shapes. Usually, primitive
// objects do the actual work, while composite objects only
// delegate to their sub-components.
class Dot implements Graphic {
    field x, y

    constructor Dot(x, y) { ... }

    method move(x, y) {
        this.x = x + this.x; this.y = y
    }

    method draw() {
        // Draw a dot at x and y.
    }
}

// All components can have other child components.
class Circle implements Graphic {
    field radius

    constructor Circle(x, y, radius) { ... }

    method draw() {
        // Draw a circle at x and y with radius r.
    }

// The composite class represents complex components that may
// have children. Usually, composites delegate all of their actual
// work to their children and then "sum up" the result.
class CompositeGraphic implements Graphic {
    field children: array of Graphic

    // A composite object can add or remove other components
    // (both simple or complex) to or from its child list.
    method add(child: Graphic) {
        // Add a child to the array of children.
    }

    method remove(child: Graphic) {
        // Remove a child from the array of children.
    }

    method move(x, y) {
        foreach (child in children) {
            child.move(x, y)
        }

    // A composite executes its primary logic in a particular
    // way. It traverses recursively through all its children
    // collecting and summing up their results. Since the
    // composite's children also these calls to their sub-
    // children, this whole hierarchy objects has to traverse
    // as a result.
    method draw() {
        // For each child component:
        // - Draw the component.
        // - Update the bounding rectangle.
        // - Pass the current rectangle along the remaining
        // siblingsitem.
    }

// The client code works with all the components via their base
// interface. This way the client code can support simple leaf
// components as well as complex composites.
class ImageEditor {
    field all: CompositeGraphic

    method load() {
        all = new CompositeGraphic()
        all.add(new Dot(1, 2))
        all.add(new Circle(5, 3, 10))
        // ...

        // Combine selected components into one complex composite
        // component.
        method groupSelected(components: array of Graphic) {
            group = new CompositeGraphic()
            foreach (component in components) {
                all.remove(component)
                group.add(component)
                // Add a group to all composites.
                all.add(group)
                // All components will be shown.
                all.draw()
```

Applicability

Use the Composite pattern when you have to implement a tree-like object structure.

The Composite pattern provides you with two basic element types that share a common interface: simple leaves and complex containers. A container can be composed of both leaves and other containers. This lets you construct a nested recursive object structure that resembles a tree.

Use the pattern when you want the client code to treat both simple and complex elements uniformly.

All elements defined by the Composite pattern share a common interface. Using this interface, the client doesn't have to worry about the concrete class of the objects it works with.

How to Implement

1. Make sure that the core model of your app can be represented as a tree structure. Try to break it down into simple elements and containers. Remember that containers must be able to contain both simple elements and other containers.
2. Declare the component interface with a list of methods that make sense for both simple and complex components.
3. Create a leaf class to represent simple elements. A program may have multiple different leaf classes.
4. Create a container class to represent complex elements. In this class, provide an array field for storing references to sub-elements. The array must be able to store both leaves and containers, so make sure its declared with the component interface type.

   While implementing the methods of the component interface, remember that a container is supposed to be delegating most of the work to sub-elements.
5. Finally, define the methods for adding and removal of child elements in the container.

   Keep in mind that these operations can be declared in the component interface. This would violate the *Interface Segregation Principle* because the methods will be empty in the leaf class. However, the client will be able to treat all the elements equally, even when composing the tree.

Pros and Cons

- ✓ You can work with complex tree structures more conveniently: use polymorphism and recursion to your advantage.
- ✓ *Open/Closed Principle*. You can introduce new element types into the app without breaking the existing code, which now works with the object tree.
- ✗ It might be difficult to provide a common interface for classes whose functionality differs too much. In certain scenarios, you'd need to overgeneralize the component interface, making it harder to comprehend.

Relations with Other Patterns

- You can use **Builder** when creating complex **Composite**s because you can program its construction steps to work recursively.
- **Chain of Responsibility** is often used in conjunction with **Composite**. In this case, when a *leaf* component gets a request, it may pass it through the chain of all of the parent components down to the root of the object tree.
- You can use **Iterator**s to traverse **Composite** trees.
- You can use **Visitor** to execute an operation over an entire **Composite** tree.
- You can implement shared leaf nodes of the **Composite** tree as **Flyweight**s to save some RAM.
- **Composite** and **Decorator** have similar structure diagrams since both rely on recursive composition to organize an open-ended number of objects.

  A **Decorator** is like a **Composite** but only has one child component. There's another significant difference: **Decorator** adds additional responsibilities to the wrapped object, while **Composite** just "sums up" its children's results.

  However, the patterns can also cooperate: you can use **Decorator** to extend the behavior of a specific object in the **Composite** tree.
- Designs that make heavy use of **Composite** and **Decorator** can often benefit from using **Prototype**. Applying the pattern lets you clone complex structures instead of reconstructing them from scratch.

Code Examples

Support our free website and own the eBook!
- 22 design patterns and principles explained in detail
- 550+ illustrations, easy-to-read, jargon-free pages
- 120+ clear and helpful illustrations and diagrams
- on actual real-life code examples in 7 languages
- on devices supported: PDF/EPUB/MOBI/K-Kindle

Learn more

© 2014-2022 Refactoring.Guru. All rights reserved.
Illustrations by Dmitry Zhart
Terms & Conditions
Privacy Policy
Content Usage Policy

Kreditkartenbüro Ukraine JSC (ТД Кредитний бюро), Ukraine, 12345
Email: support@refactoring.guru