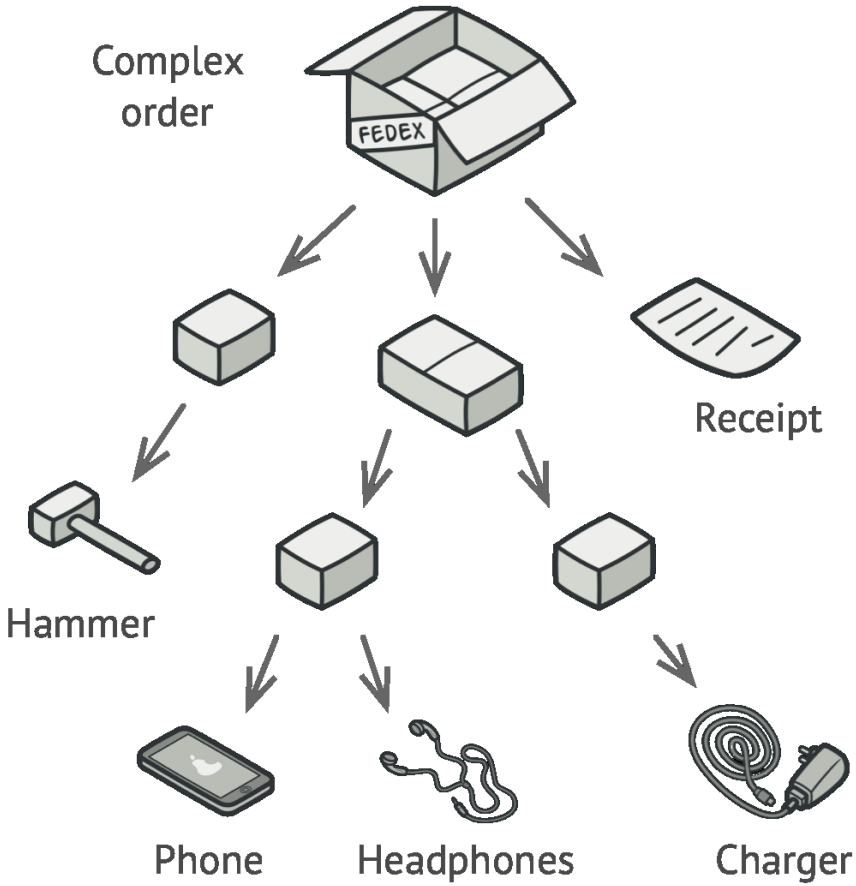
Composite

Also known as: Object Tree

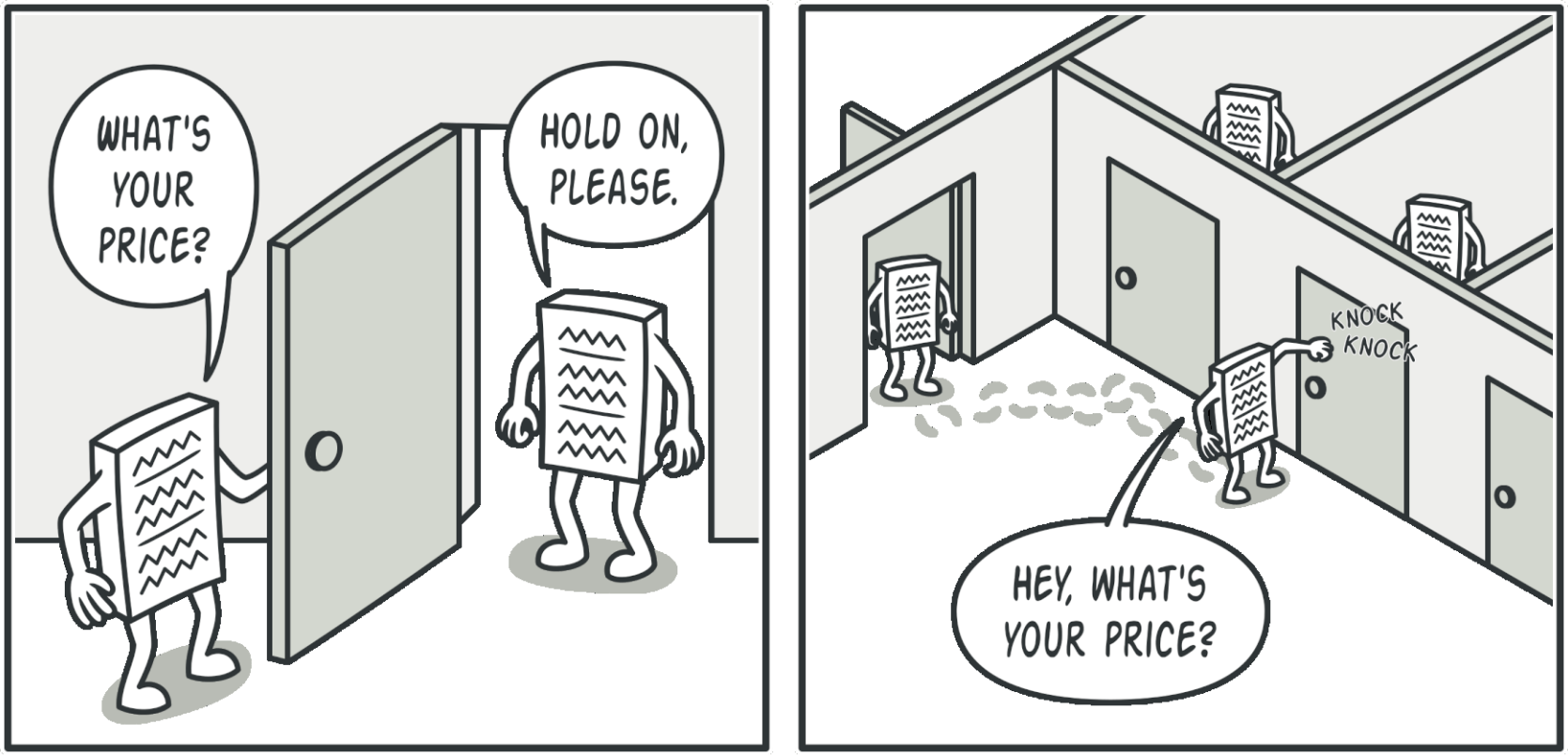
Composite is a structural design pattern which allows to compose objects into tree structures and then work with these structures as if they were individual objects.

Problem



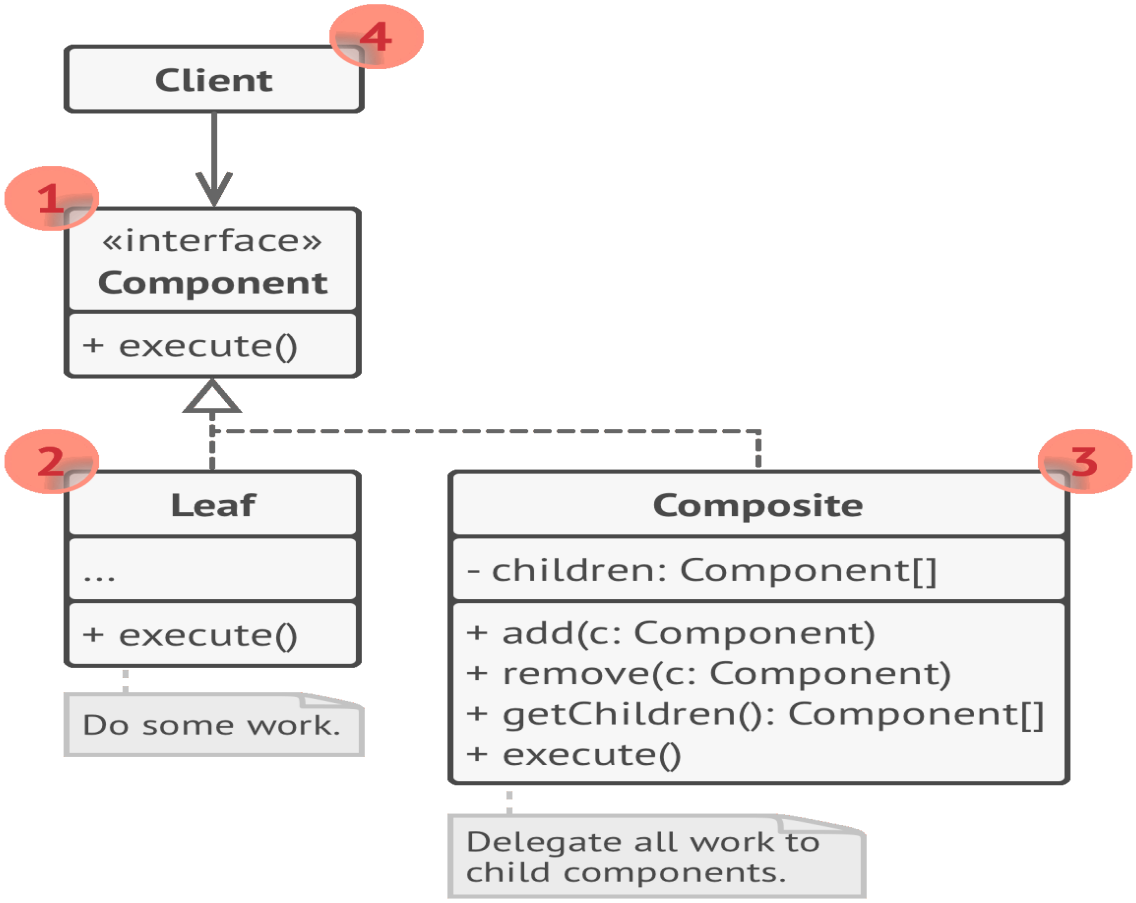
An order might comprise various products, packaged in boxes, which are packaged in bigger boxes and so on. The whole structure looks like an upside down tree.

Solution



The Composite pattern allows to run a behavior recursively over all components of an object tree.

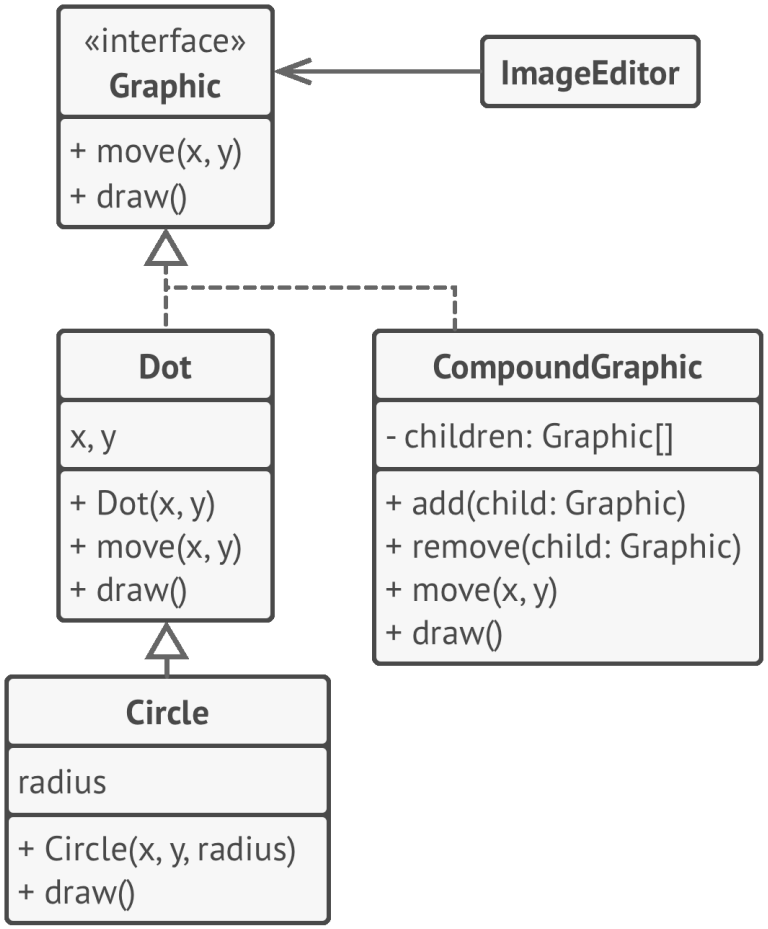
Structure


4
Client
1
«interface»
Component
+ execute()
2
Leaf
...
+ execute()
Do some work.
3
Composite
- children: Component[]
+ add(c: Component)
+ remove(c: Component)
+ getChildren(): Component[]
+ execute()
Delegate all work to
child components.

Contd.

- The **Component** interface describes operations that are common to both simple and complex elements of the tree.
- 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.
- 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.
- 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.

Pseudo code



The geometric shapes editor example.

Applicability

- Use the Composite pattern when we have to implement a tree-like object structure.
- Use the pattern when we want the client code to treat both simple and complex elements uniformly.

Pros and Cons

- We can work with complex tree structures more conveniently: use polymorphism and recursion as advantage.
- *Open/Closed Principle*. We 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, there is a need of overgeneralization of the component interface, making it harder to comprehend.