Structural Design Patterns

Instructor: Mehroze Khan

Structural Design Patterns

- Structural design patterns explain how to **assemble objects and classes** into larger structures, while keeping these structures flexible and efficient.
- Structural class patterns use inheritance to compose interfaces or implementations.

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (class) (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (object) (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

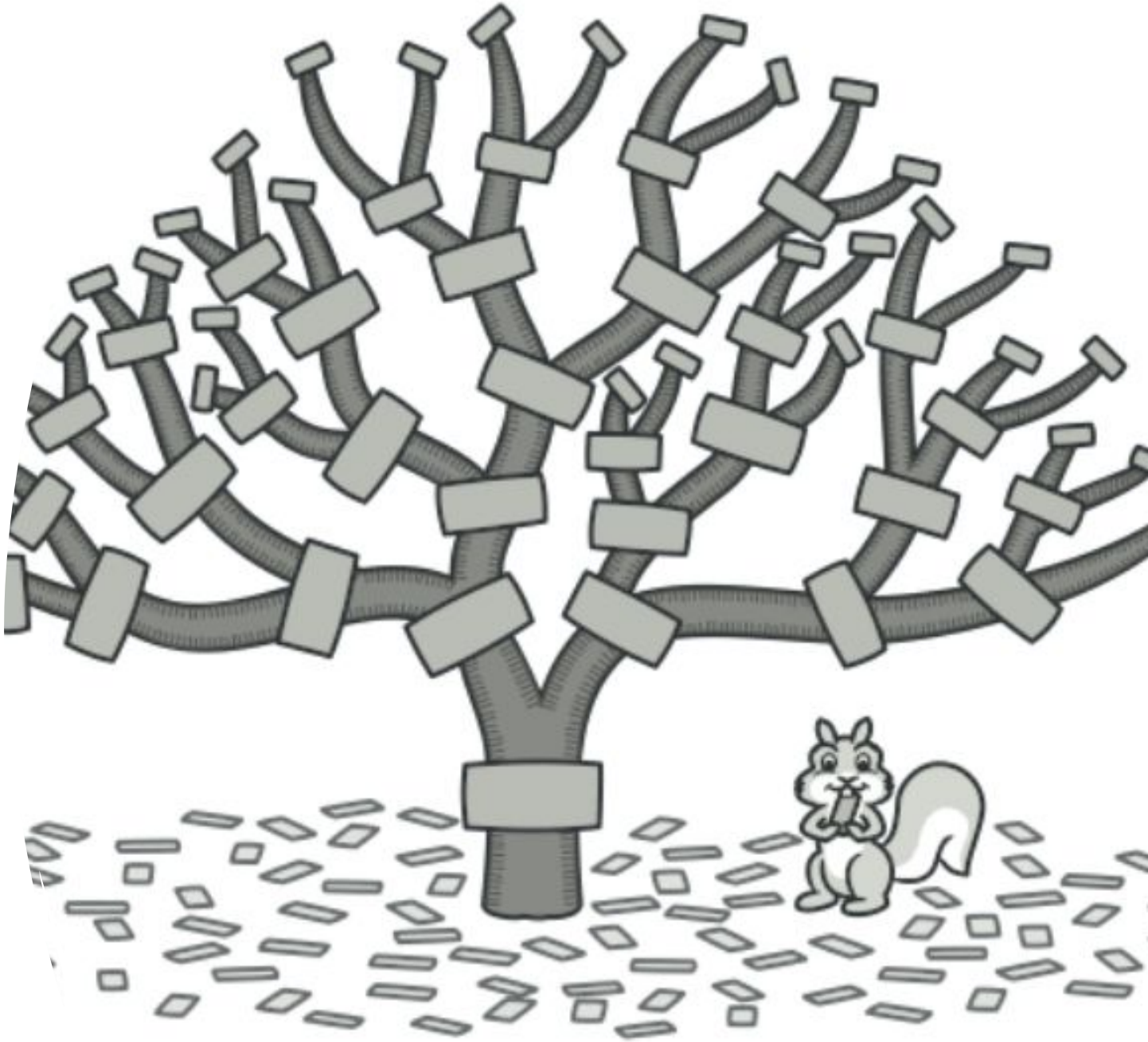
Composite Pattern

Composite Design Pattern

Also Known as: **Object Tree**

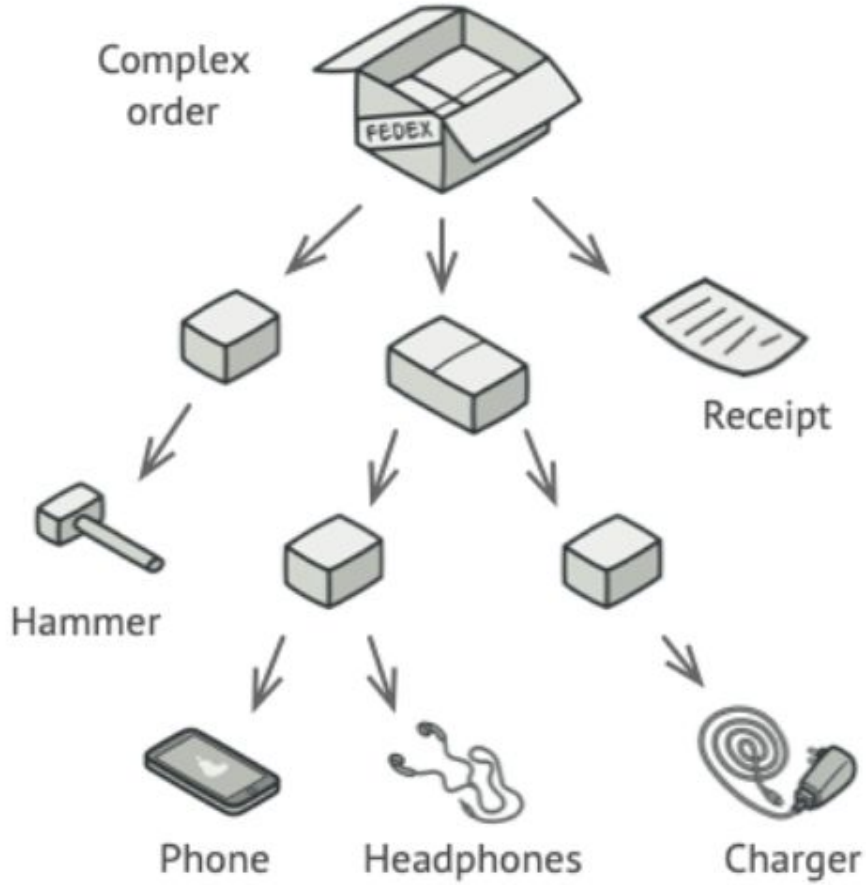
Intent:

- **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: **Products** and **Boxes**.
- A Box can contain several Products as well as several smaller Boxes. These little Boxes can also hold some Products or even smaller Boxes, 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?



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

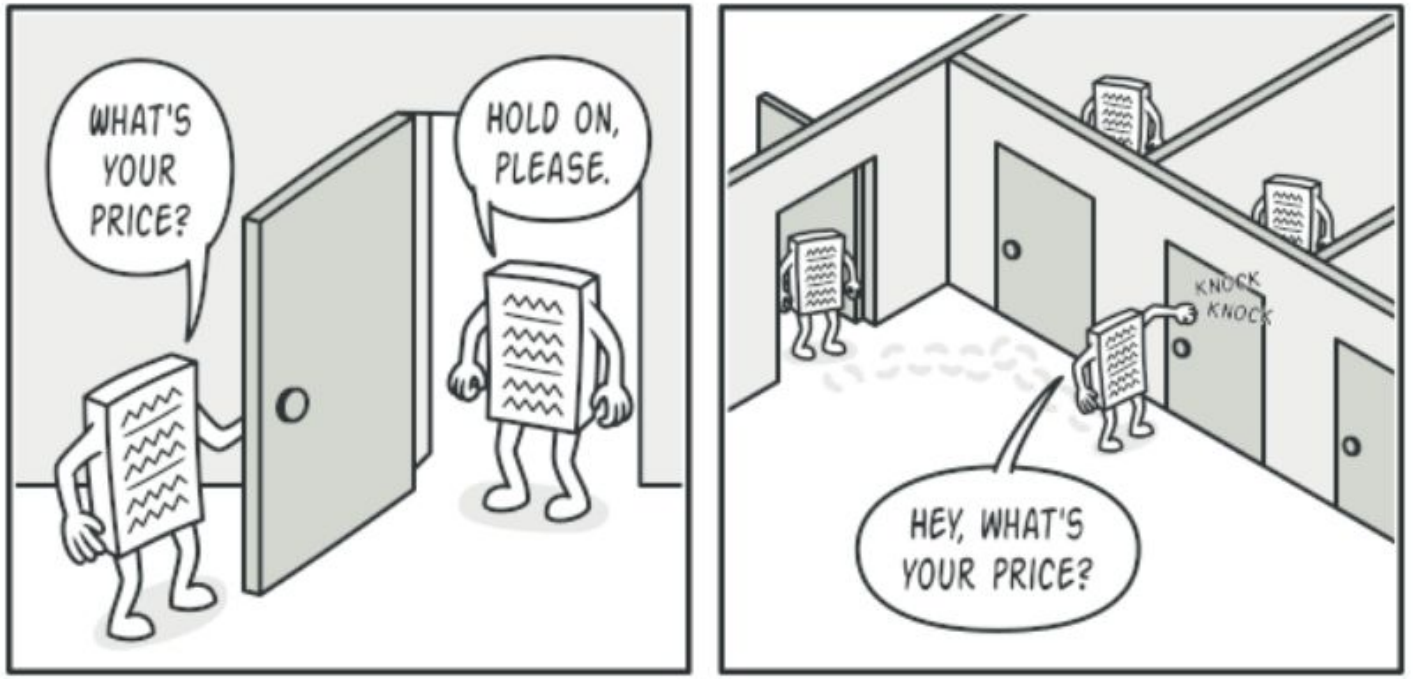
Problem

- 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 must know the classes of Products and Boxes you’re going through, the nesting level 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 Products and Boxes through a **common interface** which declares a method for calculating the total price.
- **How would this method work?**

- For a product, it'd simply return the product's price.
- For a box, it'd go over each item the box contains, ask its price and then return a total for this box.
- If one of these items were a smaller box, that box would also start going over its contents 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.

Solution

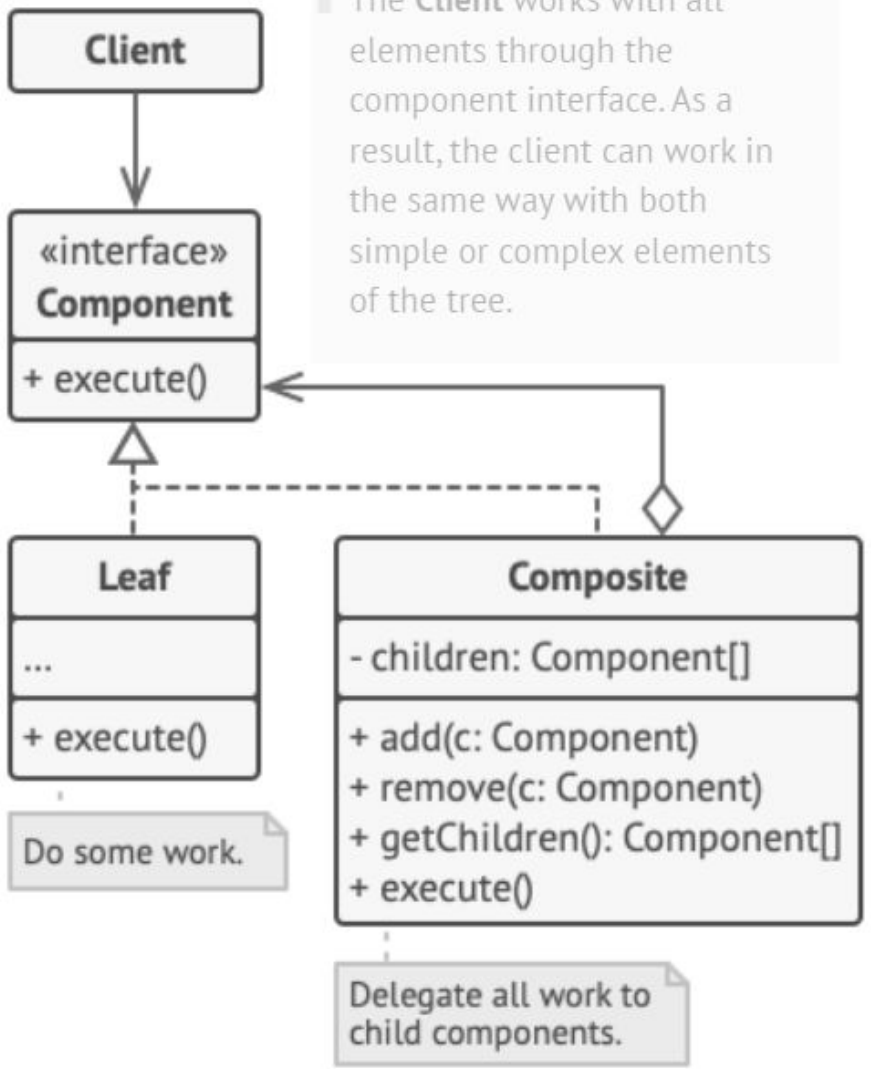
- The greatest benefit of this approach is that you don’t need to care about the concrete classes of objects that compose the tree.
- You don’t need to know whether an object is a simple product or a sophisticated box.
- You can treat them all the same via the common interface. When you call a method,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

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.



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.

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.

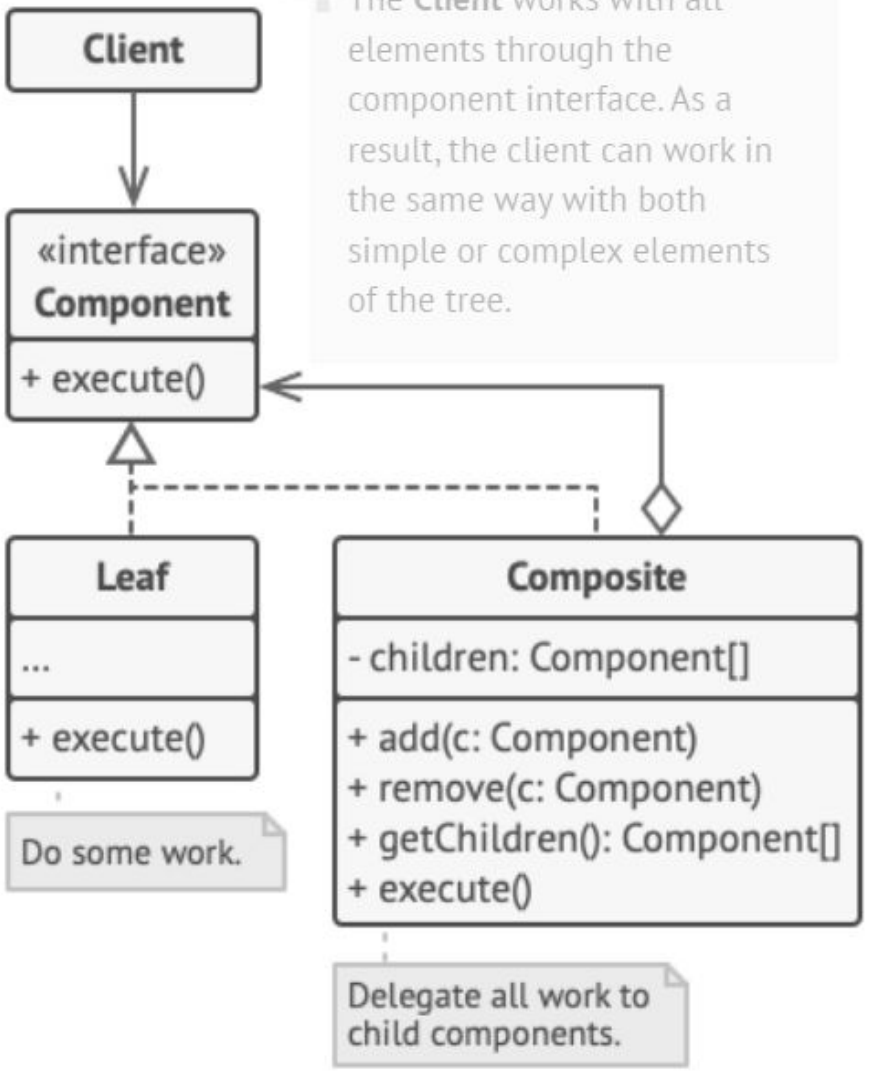
Upon receiving a request, a container delegates the work to its sub-elements, processes intermediate results and then returns the final result to the client.

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.



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.

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.

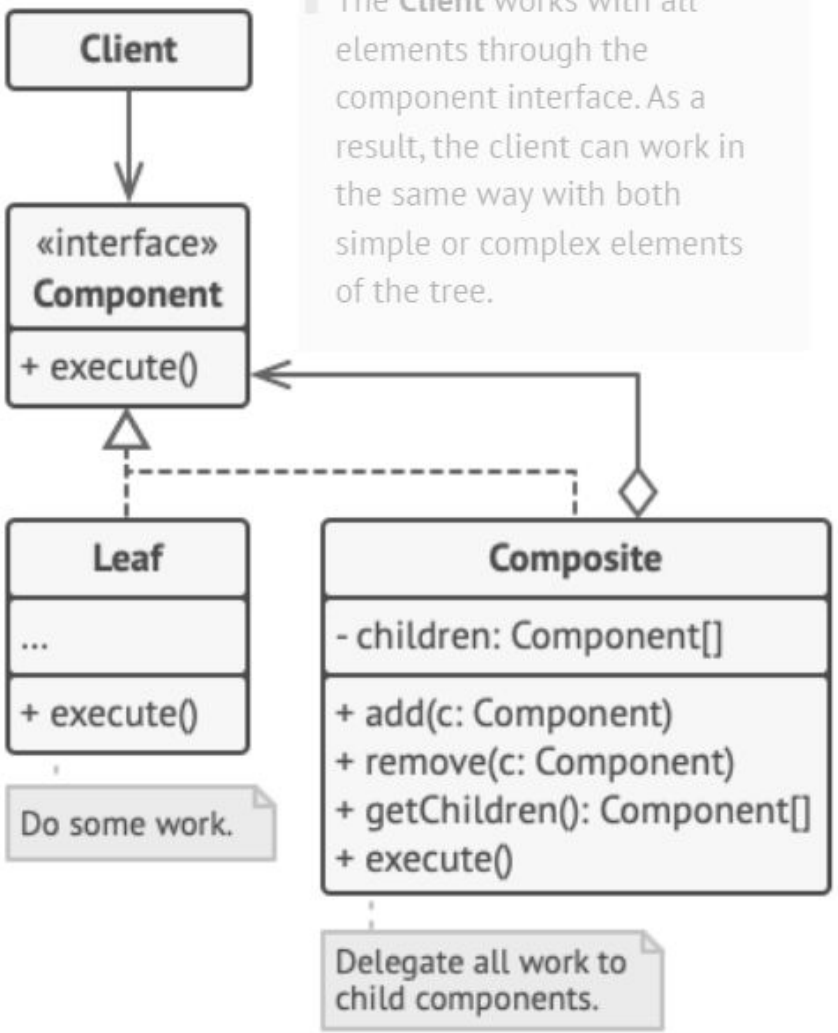
Upon receiving a request, a container delegates the work to its sub-elements, processes intermediate results and then returns the final result to the client.

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.



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.

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.

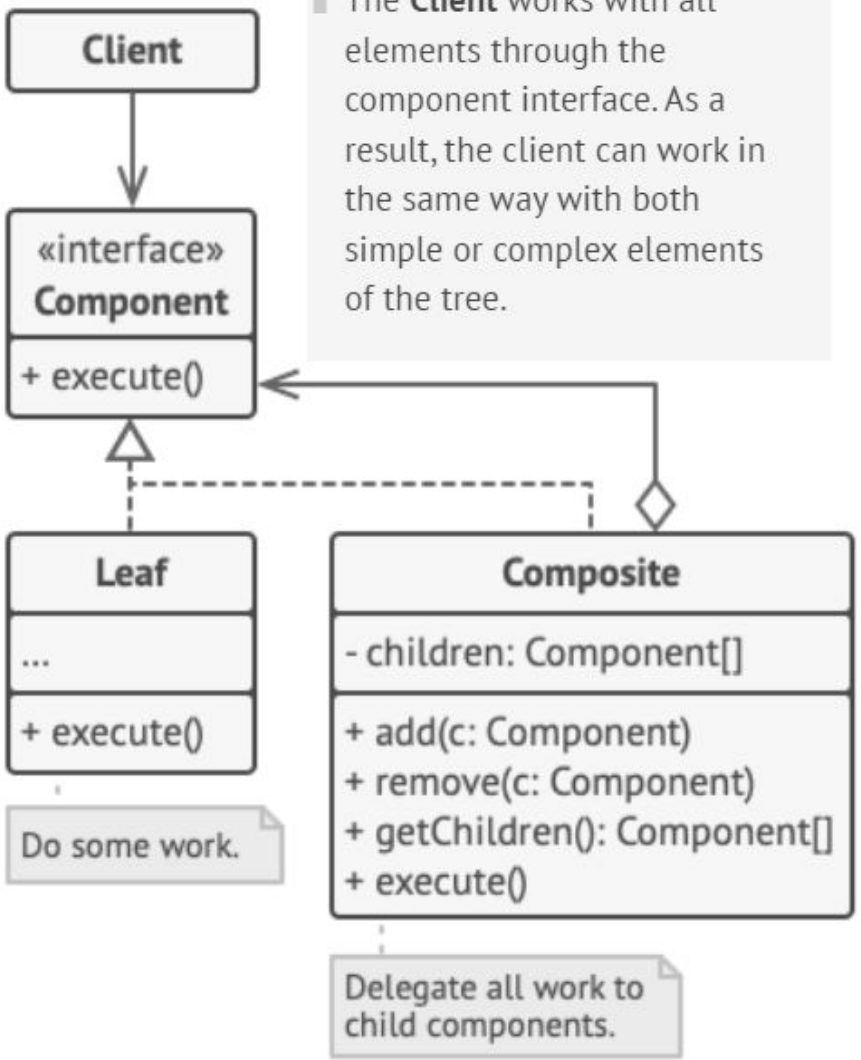
Upon receiving a request, a container delegates the work to its sub-elements, processes intermediate results and then returns the final result to the client.

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.



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.

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.

Example

```
// Component interface
class Item {
public:
    virtual double getPrice() const = 0;
};
// Leaf class (Product)
class Product : public Item {
private:
    double price;
public:
    Product(double p) {
        price = p;
    }
    double getPrice() const override {
        return price;
    }
};
```

```
// Composite class (Box)
class Box : public Item {
private:
    vector<Item*> items;
public:
    void add(Item* item) {
        items.push_back(item);
    }
    double getPrice() const override {
        double totalPrice = 0.0;
        for (const auto& item : items) {
            totalPrice += item->getPrice();
        }
        return totalPrice;
    }
};
```

Example

```
int main() {
 // Creating products
   Item* product1 = new Product(20.0);
   Item* product2 = new Product(30.0);

// Creating boxes
   Box box1;
   Box box2;

// Adding products to boxes
   box1.add(product1);
   box2.add(product2);

// Adding boxes to another box
   Box mainBox;
   mainBox.add(&box1);
   mainBox.add(&box2);

 // Calculating total price of the main box
   double totalPrice = mainBox.getPrice();
   cout << "Total Price of the main box: " << totalPrice << endl;
}
```

Output:

Total Price of the main box: 50

References

- Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Pearson, 1995.
- Helping Links:
- https://refactoring.guru/design-patterns/
- https://sourcemaking.com/design_patterns/
- https://www.youtube.com/watch?v=Q1jZ4TI6MF4&t=297s&ab_channel=Telusko