Composite

Ondřej Hlava

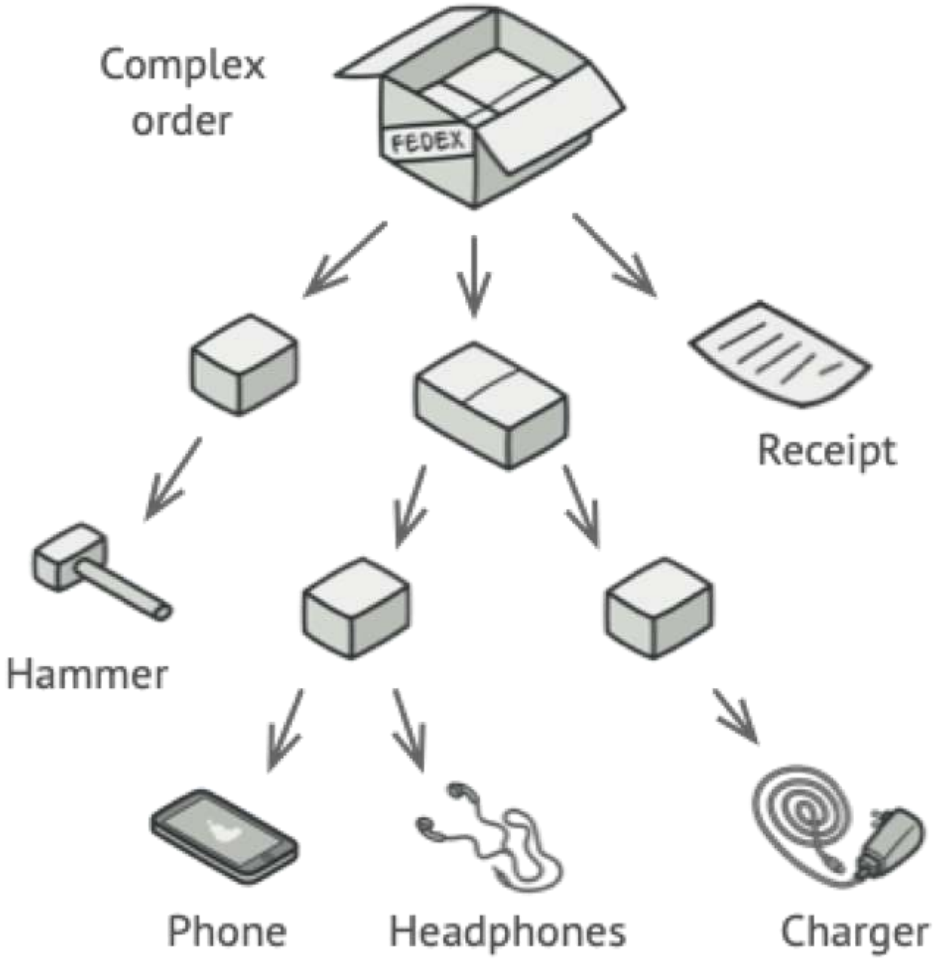
Základní vlastnosti

- Složený/kombinovaný
- Strukturální
- Velmi intuitivní

Motivace

Práce se stromovou strukturou

Vybalit zboží, spočítat cenu



Pojďme si to rozbalit a spočítat

První co by nás možná napadlo

```
public class Item {
	public int price;
	public string name;

	Public Item(int price, string name) {
		this.price = price;
		this.name = name;
	}
}
```

```
public class Box {
	public List<Item> itemsInBox;
	public List<Box> boxesInBox;

	public Box() {
		itemsInBox = new ArrayList<>();
		boxesInBox = new ArrayList<>();
	}
}
```

```
public static void Main(string[] args) {

        // Assuming getOrder() is implemented elsewhere
        Box order = GetOrder();
        List<Item> unpacked = new List<Item>();
        int total = 0;
        Queue<Box> packed = new Queue<Box>();
        packed.Enqueue(order);

        while (packed.Count > 0) {
                var current = packed.Dequeue();

                foreach (var box in current.BoxesInBox) {
                        packed.Enqueue(box);
                }

                foreach (var item in current.ItemsInBox) {
                        unpacked.Add(item);
                        total += item.Price;
                }
        }
}
```

2. řešení

```
public abstract class Packable { }

public class Item : Packable {
      public int Price;
      public string Name;
      public Item(int price, string name) {
            Price = price;
            Name = name;
      }
}
```

```
public class Box : Packable {
      public List<Packable> ContentsOfBox;
      public Box() {
            ContentsOfBox = new
      List<Packable>();
      }

      public void Insert(Packable p) {
      ContentsOfBox.Add(p);
      }
}
```

```
public static void Main(string[] args) {
    // Assuming getOrder() is implemented elsewhere
    Box order = GetOrder();
    List<Item> unpacked = new List<Item>();
    int total = 0;
    Queue<Box> packed = new Queue<Box>();
    packed.Enqueue(order);

    while (packed.Count > 0) {
        var current = packed.Dequeue();

        foreach (var content in current.ContentsOfBox) {
            if (content is Item i) {
                unpacked.Add(i);
                total += i.Price;
            } else if (content is Box b) {
                packed.Enqueue(b);
            } else {
                throw new NotSupportedException("Unknown type in structure");
            }
        }
    }
}
```

3. řešení

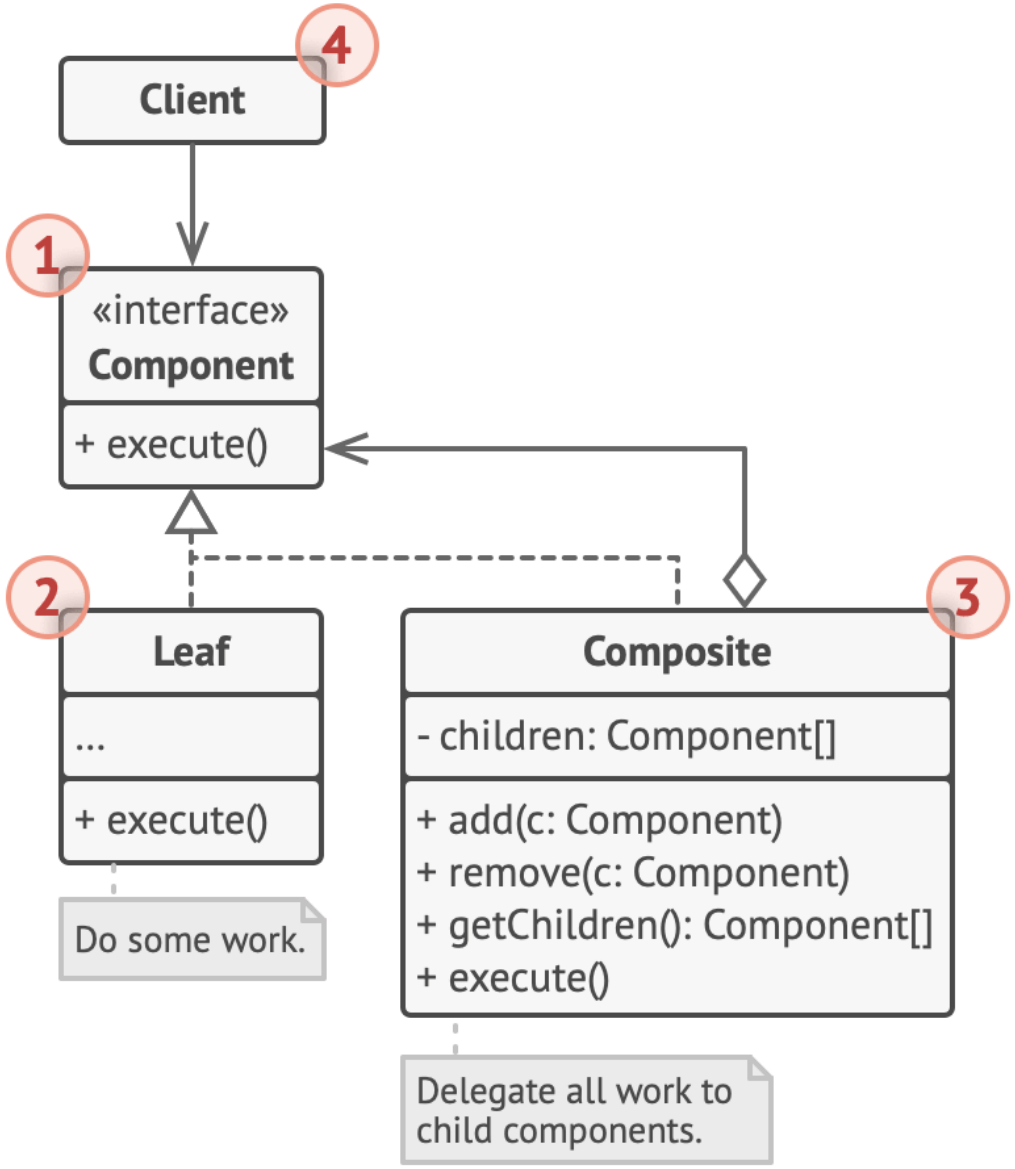
```
public abstract class Packable {
	public abstract int GetPrice();
	public abstract void UnpackTo(List<Item> list);
}


public class Item : Packable {
	public int Price;
	public string Name;
	public Item(int price, string name) {
		Price = price;
		Name = name;
	}

	public override int GetPrice() { return Price; }

	public override void UnpackTo(List<Item> list) {
		list.Add(this);
	}
}
```

```
public class Box : Packable {
	public List<Packable> ContentsOfBox;
	public Box() {
		ContentsOfBox = new List<Packable>();
	}

	public void Insert(Packable p) {
		ContentsOfBox.Add(p);
	}
	public override int GetPrice() {
		int result = 0;
		foreach (var content in ContentsOfBox) {
			result += content.GetPrice();
		}
		return result;
	}
	public override void UnpackTo(List<Item> list) {
		foreach (var content in ContentsOfBox) {
			content.UnpackTo(list);
		}
	}
}
```

4. řešení

```
public abstract class Packable {
	public abstract int GetPrice();
	public abstract void UnpackTo(List<Item> list);
	public virtual void Insert(Packable p) {
		throw new NotSupportedException("Object is a leaf.");
} }


public class Item : Packable {
	public int Price;
	public string Name;
	public Item(int price, string name) {
		Price = price;
		Name = name;
	}

	public override int GetPrice() { return Price; }

	public override void UnpackTo(List<Item> list) {
		list.Add(this);
	}
}
```

```
public class Box : Packable {
	public List<Packable> ContentsOfBox;
	public Box() {
		ContentsOfBox = new List<Packable>();
	}

	public override void Insert(Packable p) {
		ContentsOfBox.Add(p);
	}
	public override int GetPrice() {
		int result = 0;
		foreach (var content in ContentsOfBox) {
			result += content.GetPrice();
		}
		return result;
	}
	public override void UnpackTo(List<Item> list) {
		foreach (var content in ContentsOfBox) {
			content.UnpackTo(list);
		}
	}
}
```



Component :
Cílová „komponenta“ kompozitní struktury

Leaf - List :
Koncový uzel s „bázovou“ verzí operace

Composite :
Vnitřní uzel s agregační verzí operace

+

- lepší práce se složitými stromovými strukturami
- dodržování Open/Closed principu
- rovnoměrné zozložení výpočtů
- budování datové struktury
- rozšiřitelnost
- najdeme spoustu uplatnění
- zjednodušení kódu

-

- složitost společného rozhraní
- vytváření příliš obecných struktur
- porušení Interface Segregation principu

Rozhodli jsme se pro composite…

```
public abstract class Graphics {
	public abstract void Draw();
	public abstract void Add(Graphics g);
	public abstract void Remove(Graphics g);
	public abstract Graphics GetChild(int index);
}

public class Canvas : Graphics {
	private List<Graphics> children = new List<Graphics>();
	public override void Draw() {
		foreach (var child in children) {
			child.Draw();
		}
	}
	public override void Add(Graphics g) => children.Add(g);
	public override void Remove(Graphics g) => children.Remove(g);
	public override Graphics GetChild(int index) => children[index];
}
```

```
public abstract class Graphics {
      public abstract void Draw();
      public abstract void Add(Graphics g);
      public abstract void Remove(Graphics g);
      public abstract Graphics GetChild(int index);
}

public class Primitive : Graphics {
      public override void Draw() { /* Implementation specific to primitive */ }
      public override void Add(Graphics g) { throw new NotImplementedException(); }
      public override void Remove(Graphics g) { throw new NotImplementedException();}
      public override Graphics GetChild(int index) { throw new NotImplementedException();}
}

public class Line : Primitive { /* Specific drawing implementation */ }
public class Circle : Primitive { /* Specific drawing implementation */ }
```

Zlepšíme:

```
public interface IGraphic { void Draw(); }

public interface ICompositeGraphic : IGraphic {
      void Add(IGraphic graphic);
      void Remove(IGraphic graphic);
      IGraphic GetChild(int index);
}
public class Canvas : ICompositeGraphic {
      private List<IGraphic> children = new List<IGraphic>();
      public void Draw() { foreach (var child in children) { child.Draw(); } }
      public void Add(IGraphic graphic) => children.Add(graphic);
      public void Remove(IGraphic graphic) => children.Remove(graphic);
      public IGraphic GetChild(int index) => children[index];
}
public class Line : IGraphic {
      public void Draw() { /* Implementation for drawing a line */ }
}
public class Circle : IGraphic {
      public void Draw() { /* Implementation for drawing a circle */ }
}
```

Problém, transparence

1. řešení:

```
public interface IGraphic {
      void Draw();
      bool IsComposite();
}

class Canvas : IGraphic {
      ...
      public bool IsComposite() => true;
}

public class Line : IGraphic {
      public void Draw() { /* Implementation for drawing a line */ }
      public bool IsComposite() => false;
}

public class Circle : IGraphic {
      public void Draw() { /* Implementation for drawing a circle */ }
      public bool IsComposite() => false;
}
```

Problém, transparence

2. řešení:

```
public interface IGraphic {
    void Draw();
    // Default implementation of GetComposite that returns null
    IGraphic GetComposite() => null;
}

public class Canvas : IGraphic {
    ...
    public IGraphic GetComposite() => this;
}

public class Line : IGraphic {
    public void Draw() { /* Implementation for drawing a line */ }
}

public class Circle : IGraphic {
    public void Draw() { /* Implementation for drawing a circle */ }
}
```

Composite + Other Patterns

Builder:

- při vytváření složitých kompozitních stromů
- jeho konstrukční kroky můžete naprogramovat tak, aby pracovaly rekurzivně

Iterator:

- pro průchod composite stromem bez znalosti vnitřní reprezentace

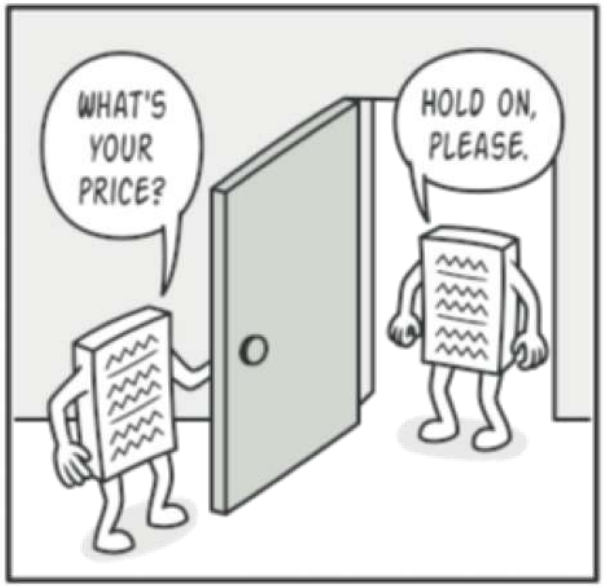
Visitor:

- pro provedení operací nad composite stromem

Decorator:

- také spoleh na rekurzivní kompozici pro uspořádání neomezeného počtu objektů
- pouze jedna podřízená komponenta
- Decorator přidává zabalenému objektu další povinnosti, zatímco Composite pouze "sčítá" výsledky svých dětí.
- Spolupráce - pomocí Decoratoru můžete rozšířit chování konkrétního objektu ve stromu Composite.

Díky za pozornost


WHAT'S YOUR PRICE?
HOLD ON, PLEASE.


KNOCK KNOCK
HEY, WHAT'S YOUR PRICE?