Structural Design Pattern

Hung Tran

Fpt software

November 1, 2021

Outline

1. Structural Pattern Overview

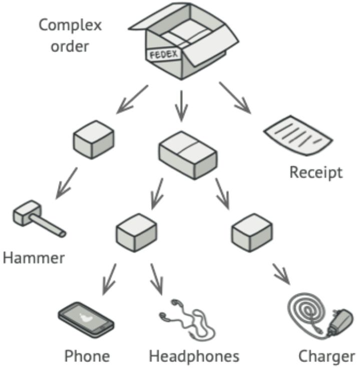
2. Composite design pattern

Structural Pattern Overview

How classes and objects are composed to form larger structure.

- **Adapter**: Convert the interface of a class into another interface.
- **Bridge**: Decouple an abstraction from its implementation.
- **Composite**: Compose objects into tree structure.
- **Decorator**: Attach additional responsibilities to an object dynamically.
- **Facade**: Provide a unified interface to a set of interfaces.
- **Flyweight**: Use sharing to support large numbers of fine-grained objects efficiently.
- **Proxy**: Provide a surrogate or placeholder for another object to control access to it.

Problem Statement



- Imagine that you have two types of objects: **Products** and **Boxes**
- A Box can contain several Products as well as a number of smaller Boxes.
- These little Boxes can also hold some Products or even smaller Boxes, and so on.

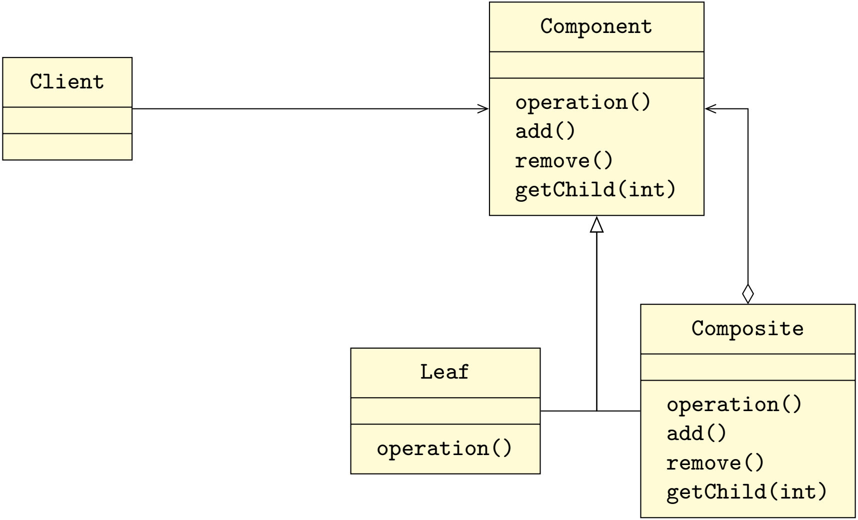
Problem Statement

- 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?
- 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 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.

The Intent of Composite Design Pattern

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Structure of Bridge Pattern: Object adapter



Tree implementation

component.h

```cpp
#ifndef _COMPONENT_H_
#define _COMPONENT_H_
#include <algorithm>
#include <string>
class Component {
protected:
  Component* _parent;
public:
  virtual ~Component();
  void setParent(Component* parent);
  Component* getParent() const;
  virtual void add(Component* component);
  virtual void remove(Component* component);
  virtual bool isComposite() const;
  virtual std::string operation() const = 0;
};
#endif // _COMPONENT_H_
```

component.cpp

```cpp
#include "component.h"

Component::~Component() {
}

void Component::setParent(Component* parent) {
  this->_parent = parent;
}

void Component::add(Component* component) {
}

void Component::remove(Component* component) {
}

Component* Component::getParent() const {
  return this->_parent;
}

bool Component::isComposite() const {
  return false;
}
```

Tree implementation

leaf.h

```cpp
#ifndef _LEAF_H_
#define _LEAF_H_
#include "component.h"

class Leaf : public Component {
public:
  std::string operation() const override;
};
#endif
```

leaf.cpp

```cpp
#include "leaf.h"

std::string Leaf::operation() const {
  return "Leaf";
}
```

Tree implementation

composite.h

```cpp
#ifndef _COMPOSITE_H_
#define _COMPOSITE_H_
#include "component.h"
#include <list>

class Composite : public Component {
protected:
  std::list<Component*> _children;
public:
  void add(Component* component) override;
  void remove(Component* component) override;
  bool isComposite() const override;
  std::string operation() const override;
};
#endif // _COMPOSITE_H_
```

composite.cpp

```cpp
#include "composite.h"

void Composite::add(Component* component) {
  this->_children.push_back(component);
  component->setParent(this);
}

void Composite::remove(Component* component) {
  _children.remove(component);
  component->setParent(nullptr);
}

bool Composite::isComposite() const {
  return true;
}

std::string Composite::operation() const {
  std::string result;
  for(const Component* c : _children) {
    if(c == _children.back()) {
      result += c->operation();
    }
    else {
      result += c->operation() + "+";
    }
```

Applicability

- 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.
- Declare the component interface with a list of methods that make sense for both simple and complex components.
- Create a leaf class to represent simple elements. A program may have multiple different leaf classes.
- 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 it's declared with the component interface type.
- Finally, define the methods for adding and removal of child elements in the container.

How to implement

- 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.

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 trees 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 Iterators 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 Flyweights 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.
- Designs that make heavy use of Composite and Decorator can often benefit from using Prototype.

Thank You!