Composite Pattern

Isaac Griffith

CS 2263
Department of Informatics and Computer Science
Idaho State University

Outcomes

After today’s lecture you should be able to:

- Understand the use of the Composite Design Pattern
- Use and implement the Composite Pattern

Inspiration

“Smart data structures and dumb code works a lot better than the other way around.” – Eric S. Raymond

Moving On: Composite Pattern

- The Composite Pattern allows us to build structures of objects in the form of trees that contain both objects and other composites
  - Simple example: Grouping objects in a vector drawing tool
    - You can create an individual shape and apply operations to it: `move()`, `scale()`, `rotate()`, etc.
    - You can create a group of objects and apply the SAME operations to it: `move()`, `scale()`, `rotate()`, etc.
    - Client view: individual objects and groups behave in a similar fashion
- The composite pattern lets us take individual objects, group them into a composite and then deal with that group as if it was an individual object
- Using a composite structure, we can apply the same operations over both the composites and individual objects allowing us to ignore their differences
  - ... for the most part; there will still be a need for code that knows the diff.

Menu Example Extended

- To explore the composite pattern, we are going to add a requirement to our menu program such that it has to allow for menus to have sub-menus
  - In particular, the diner menu is now going to feature a dessert menu
- We can view our concepts in a hierarchical fashion

All Menus

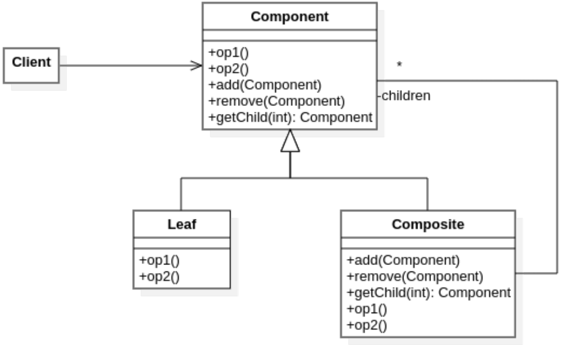
- Menu Objects
  - Menu Items and Sub-Menus
    - Menu Items

- Once we have this (new) composite structure, we still need to meet all our previous requirements, such as being able to iterate over all menu items.

Composite Pattern: Definition

- The Composite Pattern allows you to compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly
  - Items with children are called nodes (Menus)
  - Items with no children are called leaves (Menu Items)
- We can create arbitrarily complex trees (see pages 356 and 357 in textbook)
  - And treat them as groups or individuals (that is individual nodes within the tree are accessible, if needed)
  - And, we can apply an operation to the root of the tree and it will make sure that the operation is applied to all nodes within the tree
    - That is, if you apply `print()` to the root, an internal traversal makes sure that `print()` is applied to all child nodes

Composite Pattern: Structure

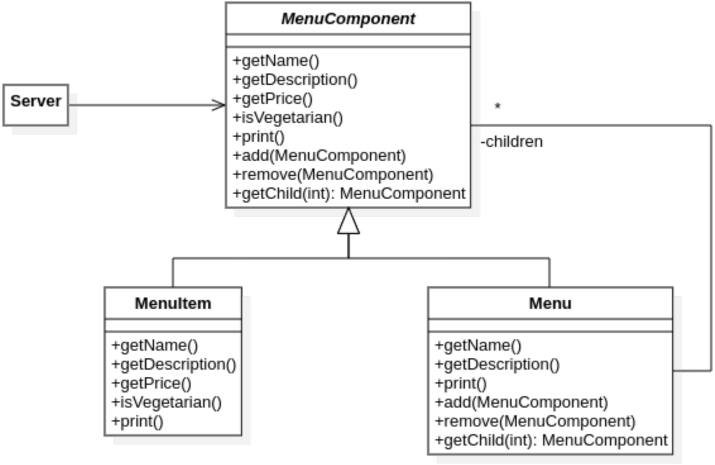


- Client has a reference to a node and, typically invokes shared operations on it (`op1()`, `op2()`).
- Component defines the **shared interface** between `Composite` and `Leaf` and adds signatures for **tree-related methods** (`add()`, `remove()`, `getChild()`).
- `Leaf` implements just the shared interface and ignores the tree-related methods.
- `Composite` implements the tree-related methods and implements the shared interface methods in a way that causes them

Implementing Menus as a Composite (I)

`MenuComponent` is abstract; will provide implementations for some of the methods

`MenuItem` is pretty much the same, it ignores the tree-based methods; `Menu` is different, it implements the tree-based methods and three of the shared operations.



Implementing Menus as a Composite (II)

- Initial steps are easy

❶ `MenuComponent` is an abstract class that implements all methods with the same line of code

  - `throw new UnsupportedOperationException()`

- This is a run-time exception that indicates that the object doesn’t respond to this method
  - Since `Menu` and `MenuItem` are subclasses they need to override each method that they support
  - This means that both of these classes will behave the same when an unsupported method is invoked on them

❷ `MenuItem` is exactly the same as before, except now it includes the phrase `extends MenuComponent` in its declaration

```
public class Menu extends MenuComponent {

  ArrayList menuComponents = new ArrayList();
  ...

  public Menu(String name, String description) {
    this.name = name;
    this.description = description;
  }

  public void add(MenuComponent menuComponent) {
    menuComponents.add(menuComponent);
  }

  public void remove(MenuComponent menuComponent) {
    menuComponents.remove(menuComponent);
  }

  public MenuComponent getChild(int i) {
    return (MenuComponent)menuComponents.get(i);
  }
  ...
  public void print() {
    System.out.print("\n" + getName());
    System.out.println(", " + getDescription());
    System.out.println("---------------------");

    Iterator iterator = menuComponents.iterator();
    while (iterator.hasNext()) {
      MenuComponent menuComponent =
          (MenuComponent)iterator.next();
      menuComponent.print();
    }
  }
```

Menu uses an `ArrayList` to store its children; making the implementation of `add()`, `remove()`, and `get()` trivial.

Menus have names and descriptions. Getter methods for these attributes are not shown.

The `print()` operation displays the menu's title and description and then uses `ArrayList`'s iterator to loop over its children; it invokes `print()` on each of them, thus (eventually) displaying information for the entire menu.

Demonstration

Design Trade-Offs

- The Composite Pattern violates one of our design principles
  - The Single Responsibility Principle
- In particular, the design of Composite is handling two responsibilities, tree-related methods and component-related methods
  - `Menu` IS-A `Menu` and `Menu` IS-A `Node`
  - `MenuItem` IS-A `MenuItem` AND `MenuItem` IS-A `Leaf`
- Even worse, both `Menu` and `MenuItem` inherit methods that they don't use!
- BUT, we gain transparency! Our client code can treat nodes and leaves in the same way... it doesn't care which one its pointing at!
  - And sometimes that characteristic is worth violating other principles
    - As with all trade-offs, you have to evaluate the benefits you are receiving and decide if they are worth the cost

Adding Iterator to Composite

- Producing your own iterator for a composite is straightforward
  - Add a `createIterator()` to `MenuComponent`
  - Have `Leaf` return a `NullIterator`
    - a `NullIterator`'s `hasNext()` method always returns false
  - Implement the traversal semantics that you want for your Composite's iterator
    - The code will be different depending on whether you want an in-order, pre-order, or post-order traversal of the tree
    - The book shows code for a pre-order traversal of the Menu tree
- **Demonstration**

Wrapping Up

- Composite: allow individual objects and groups of objects to be treated uniformly. Side Note: Caching
  - if the purpose of a shared operation is to calculate some value based on information stored in the node's children
  - then a composite pattern can add a field to each node that ensures that the value is only calculated once.
  - the first time the operation is called, we traverse the children, compute the value, and store it in the root
  - thereafter, we return the cached value
  - this technique requires monitoring changes to the tree to clear out cached values that are stale.

Are there any questions?