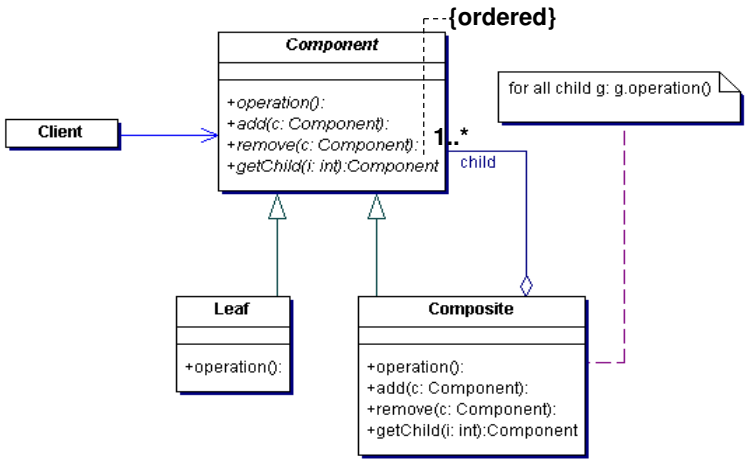
Tecniche di Progettazione: Design Patterns

GoF: Composite

Composite pattern

- Scopo
  - Comporre oggetti in una struttura ad albero per rappresentare gerarchie e lasciare che il cliente tratti in modo uniforme nodi e foglie
- Motivazioni
  - Molte applicazioni (per esempio editori di disegni e di circuiti) permettono agli utenti di costruire oggetti complessi a partire da oggetti semplici: se gli oggetti semplici sono trattati in modo diverso, l'applicazione diventa più complessa

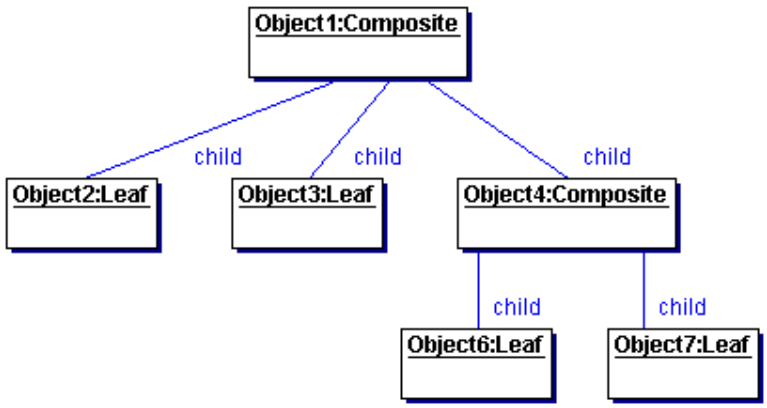
Composite: struttura



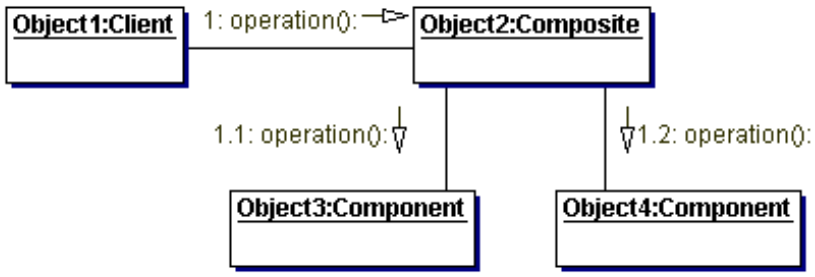
Composite: partecipanti

- Componente
  - Dichiara il tipo degli oggetti nella composizione
  - Realizza il comportamento standard di tutte le classi (se ne esiste uno)
  - Dichiara l'interfaccia per accedere ai figli
- Foglia
  - Definisce il comportamento degli oggetti primitivi nella composizione
- Composite
  - Definisce il comportamento degli oggetti con figli
  - Memorizza i figli
  - Realizza le operazioni per accedere ai figli

Composite: esempio

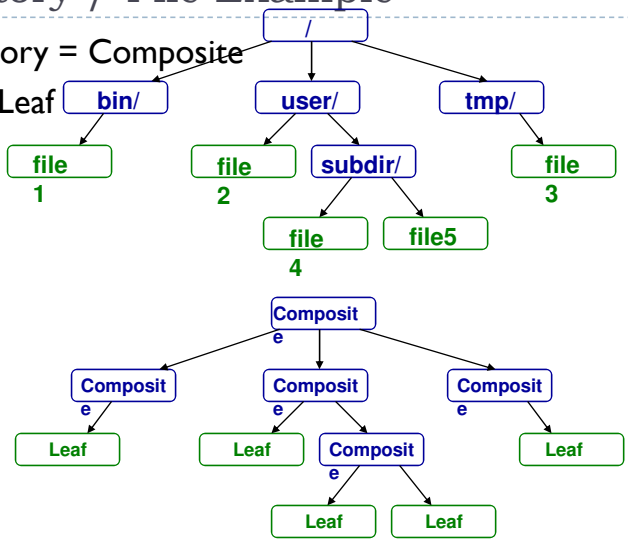


Composite: collaborazione



Directory / File Example

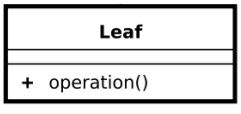
- Directory = Composite
- File = Leaf

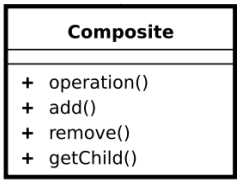


Directory / File Example – Classes

- One class for Files (Leaf nodes)
- One class for Directories (Composite nodes)
  - Collection of Directories and Files
- How do we make sure that Leaf nodes and Composite nodes can be handled uniformly?
  - Derive them from the same abstract base class

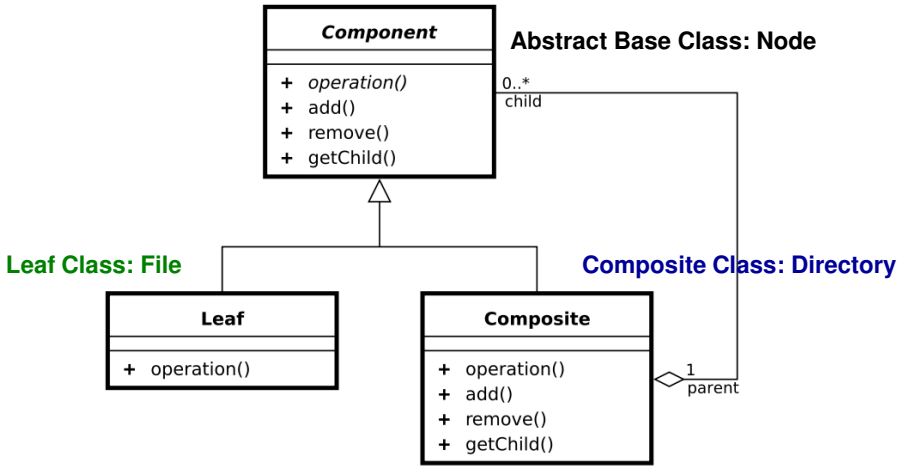
Leaf Class: File



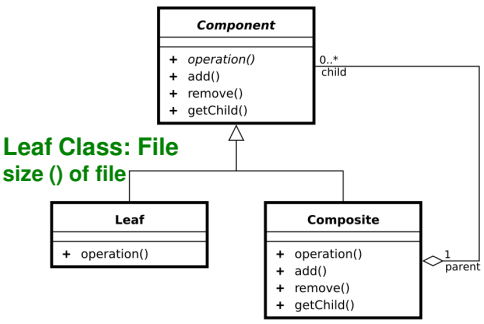


Composite Class: Directory

Directory / File Example – Structure



Directory / File Example – Operation



Abstract Base Class: Node
size() in bytes of entire directory and sub-directories

Composite Class: Directory
size () Sum of file sizes in this directory and its sub-directories

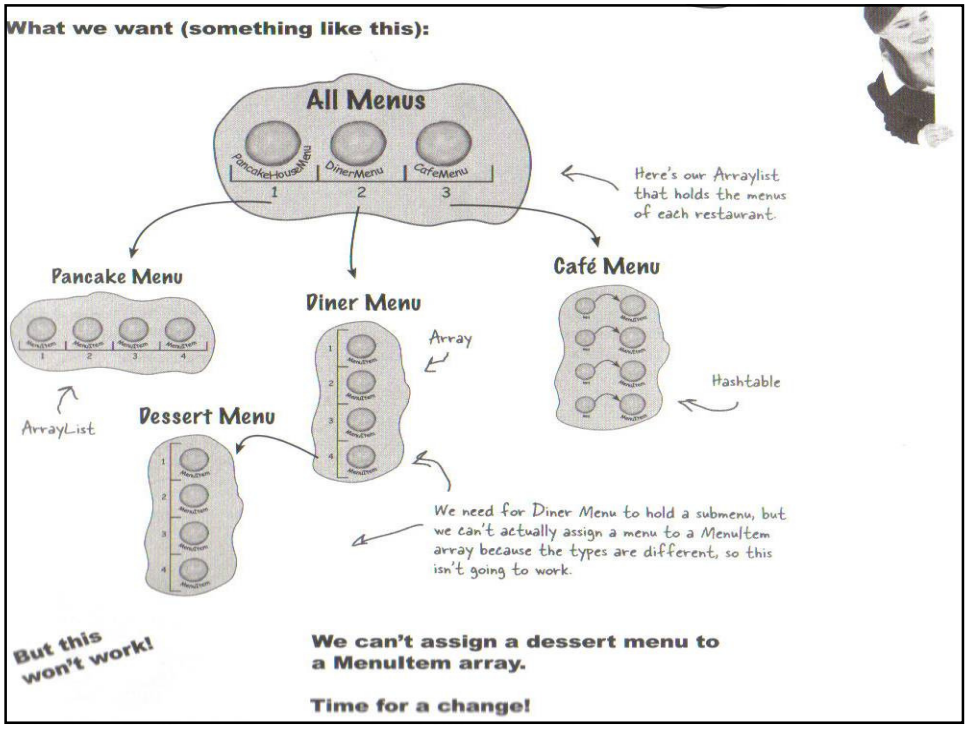
```
long Directory::size () {
  long total = 0;
  Node* child;
  for (int i = 0; child = getChild(); ++i; {
    total += child->size();
  }
  return total;
}
```

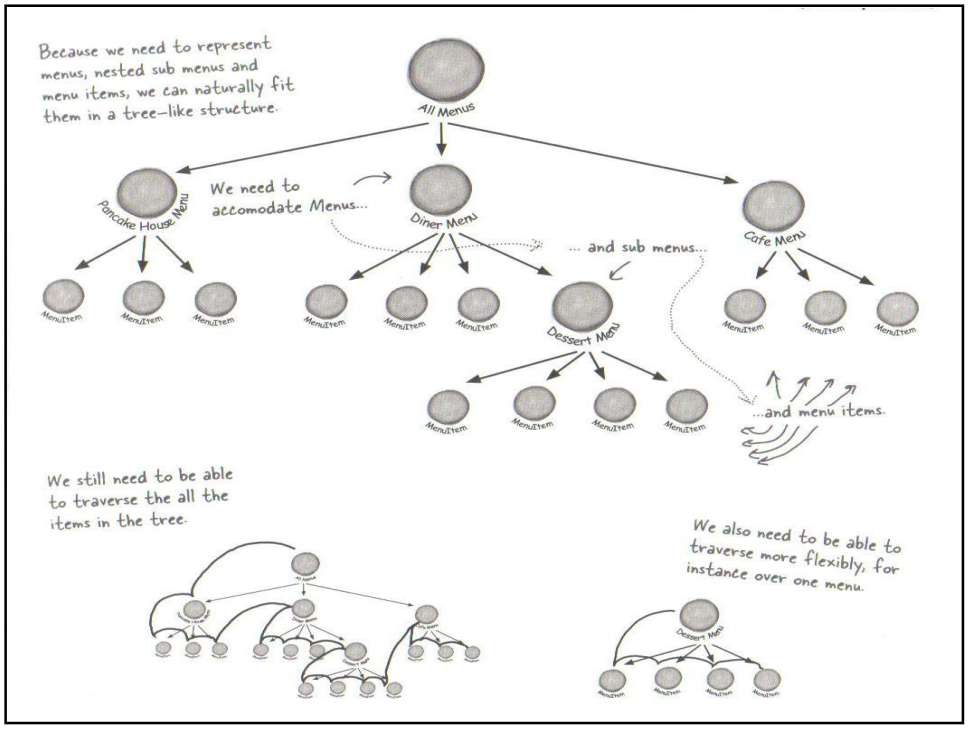
Consequences

- Solves problem of how to code recursive hierarchical part-whole relationships.
- Client code is simplified.
  - Client code can treat primitive objects and composite objects uniformly.
  - Existing client code does not need changes if a new leaf or composite class is added (because client code deals with the abstract base class).
- Can make design overly general.
  - Can't rely on type system to restrict the components of a composite. Need to use run-time checks.

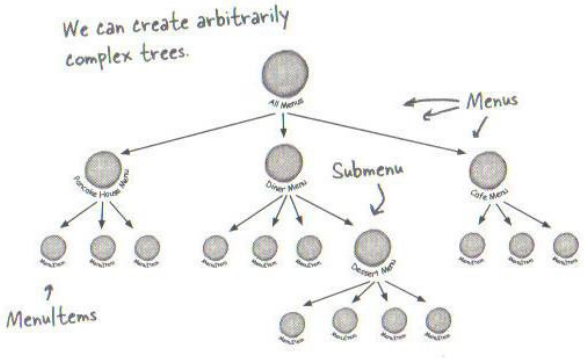
Implementation Issues

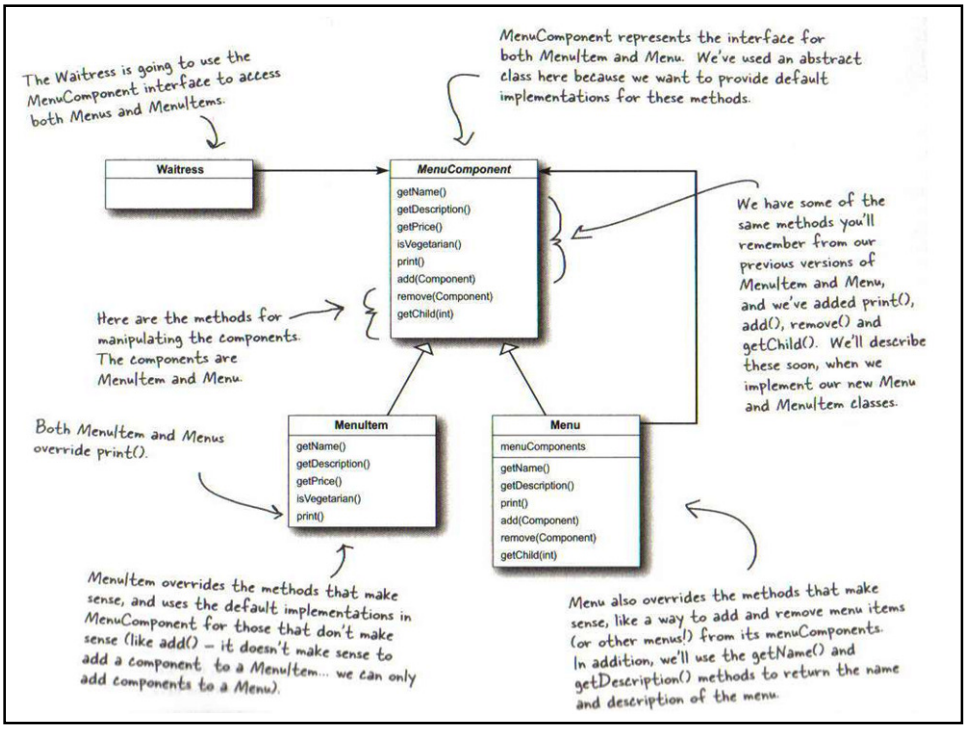
- Should Component maintain the list of components that will be used by a composite object? That is, should this list be an instance variable of Component rather than Composite?
  - Better to keep this part of Composite and avoid wasting the space in every leaf object
- Is child ordering important?
  - Depends on application
- Who should delete components?
  - Not a problem in Java! The garbage collector will come to the rescue!
- What's the best data structure to store components?
  - Depends on application


What we want (something like this):
All Menus
PancakeHouseMenu
DinerMenu
CafeMenu
1
2
3
Here's our Arraylist that holds the menus of each restaurant.
Pancake Menu
Diner Menu
Café Menu
Array
Hashtable
ArrayList
Dessert Menu
We need for Diner Menu to hold a submenu, but we can't actually assign a menu to a MenuItem array because the types are different, so this isn't going to work.
But this won't work!
We can't assign a dessert menu to a MenuItem array.
Time for a change!

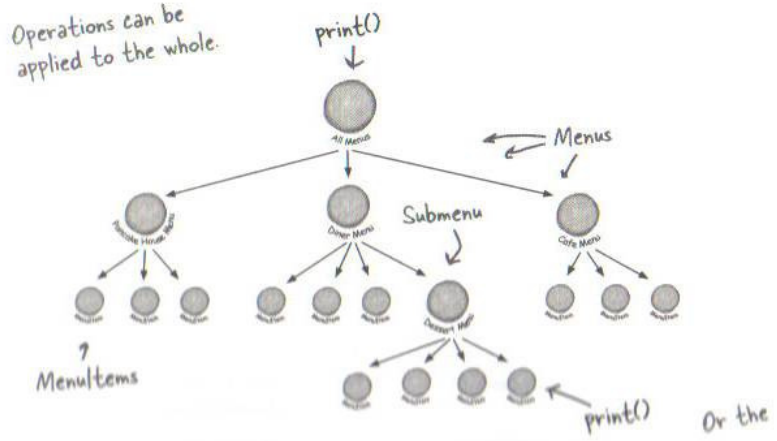

Because we need to represent menus, nested sub menus and menu items, we can naturally fit them in a tree-like structure.
All Menus
Pancake House Menu
We need to accomodate Menus...
Diner Menu
... and sub menus...
Cafe Menu
Dessert Menu
MenuItem
...and menu items.
We still need to be able to traverse the all the items in the tree.
We also need to be able to traverse more flexibly, for instance over one menu.

Complex hierarchy of menu items





Operations applied to whole or parts



Some observations

- The "print menu" method in the MenuComponent class is recursive.
- Now lets look at an alternative implementation which uses an iterator to iterate through composite classes ➔the composite iterator

MenuComponent

```
public abstract class MenuComponent {
public void add(MenuComponent menuComponent) {
throw new UnsupportedOperationException();
}
public void remove(MenuComponent menuComponent) {
throw new UnsupportedOperationException();
}
public MenuComponent getChild(int i) {
The composite methods
throw new UnsupportedOperationException();
}
public String getName() {
throw new UnsupportedOperationException();
}
```


MenuComponent

```
…
public String getDescription() {
throw new UnsupportedOperationException();
}
public double getPrice() {
throw new UnsupportedOperationException();
}
public boolean isVegetarian() {
throw new UnsupportedOperationException();}
public void print() {
throw new UnsupportedOperationException();
}
}
```

MenuItem

```
public class MenuItem extends MenuComponent {
String name;
String description;
boolean vegetarian;
double price;
public MenuItem(String name, String description, boolean vegetarian, double price)
{this.name = name;
this.description = description;
this.vegetarian = vegetarian;
this.price = price;
}
```


MenuItem

```
…
public String getName() { return name; }
public String getDescription() { return description; }
public double getPrice() { return price; }
public boolean isVegetarian() { return vegetarian; }
public void print() System.out.print(" " + getName());
if (isVegetarian()) { System.out.print("(v)");}
System.out.println(", " + getPrice());
System.out.println(" -- " + getDescription());
```

Menu

```
public class Menu extends MenuComponent {
ArrayList menuComponents = new ArrayList();
String name;
String description;
public Menu(String name, String description) {
this.name = name;
this.description = description;
}
public void add(MenuComponent menuComponent) {
menuComponents.add(menuComponent);
}
```


Menu

```
public void remove(MenuComponent menuComponent) {
menuComponents.remove(menuComponent);
}
public MenuComponent getChild(int i) { return
    (MenuComponent)menuComponents.get(i);}
public String getName() { return name;}
public String getDescription() { return description;}
public void print() {
System.out.print("\n" + getName());
System.out.println(", " + getDescription());
Iterator iterator = menuComponents.iterator();
while (iterator.hasNext()) {
MenuComponent menuComponent = (MenuComponent)iterator.next();
menuComponent.print();
}
```

Related Patterns

- Chain of Responsibility – Component-Parent Link
- Decorator – When used with composite will usually have a common parent class. So decorators will need to support the component interface with operations like: Add, Remove, GetChild.
- Flyweight – Lets you share components, but they can no longer reference their parents.
- Iterator – Can be used to traverse composites.
- Visitor – Localizes operations and behavior that would otherwise be distributed across composite and leaf classes.