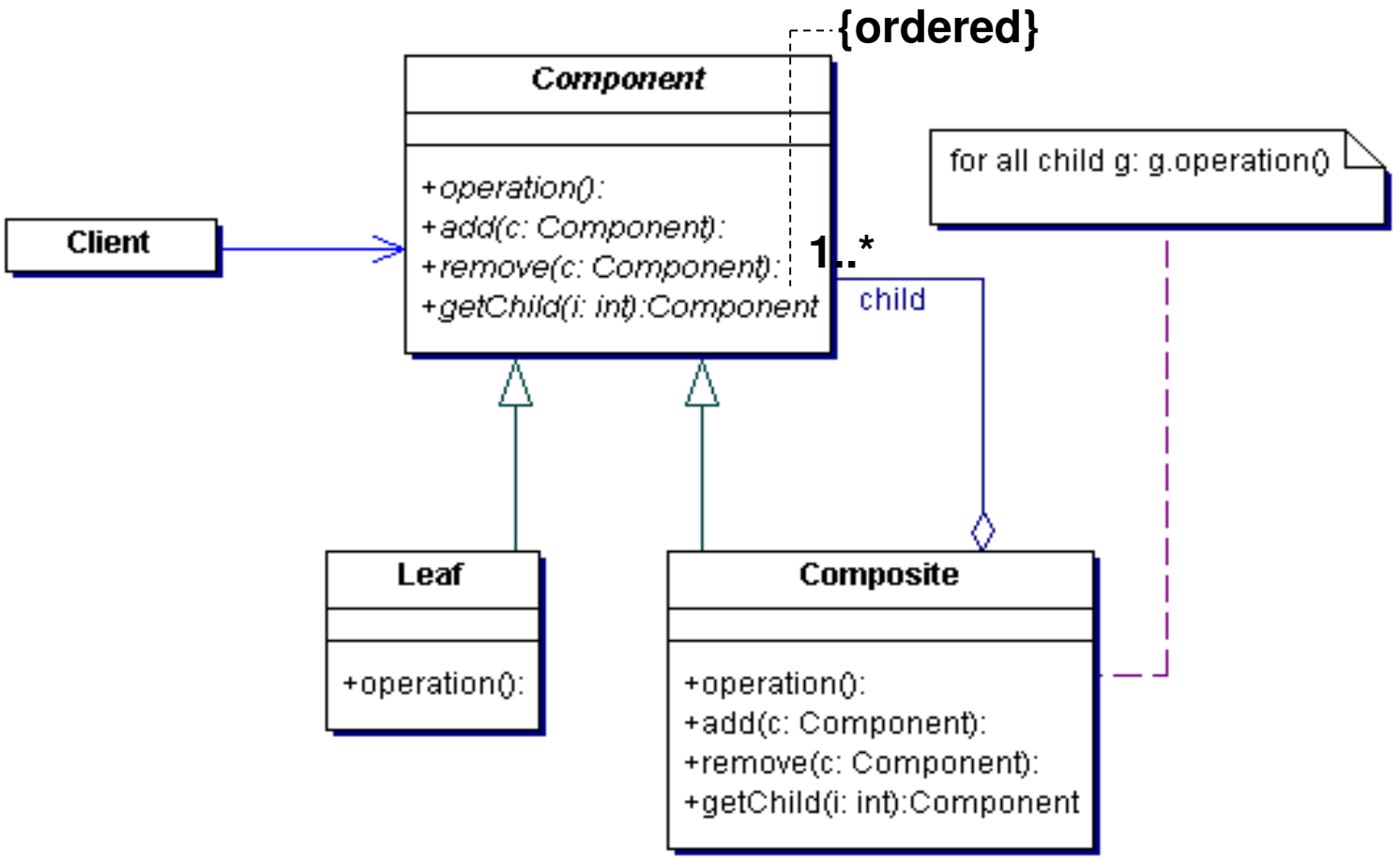
# Tecniche di Progettazione: Design Patterns

GoF: Composite

# Composite pattern

- Intent
  - Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. This is called recursive composition.
- Applicability
  - Use the Composite pattern when
    - You want to represent part-whole hierarchies of objects
    - You want clients to be able to ignore the difference between compositions of objects and individual objects.

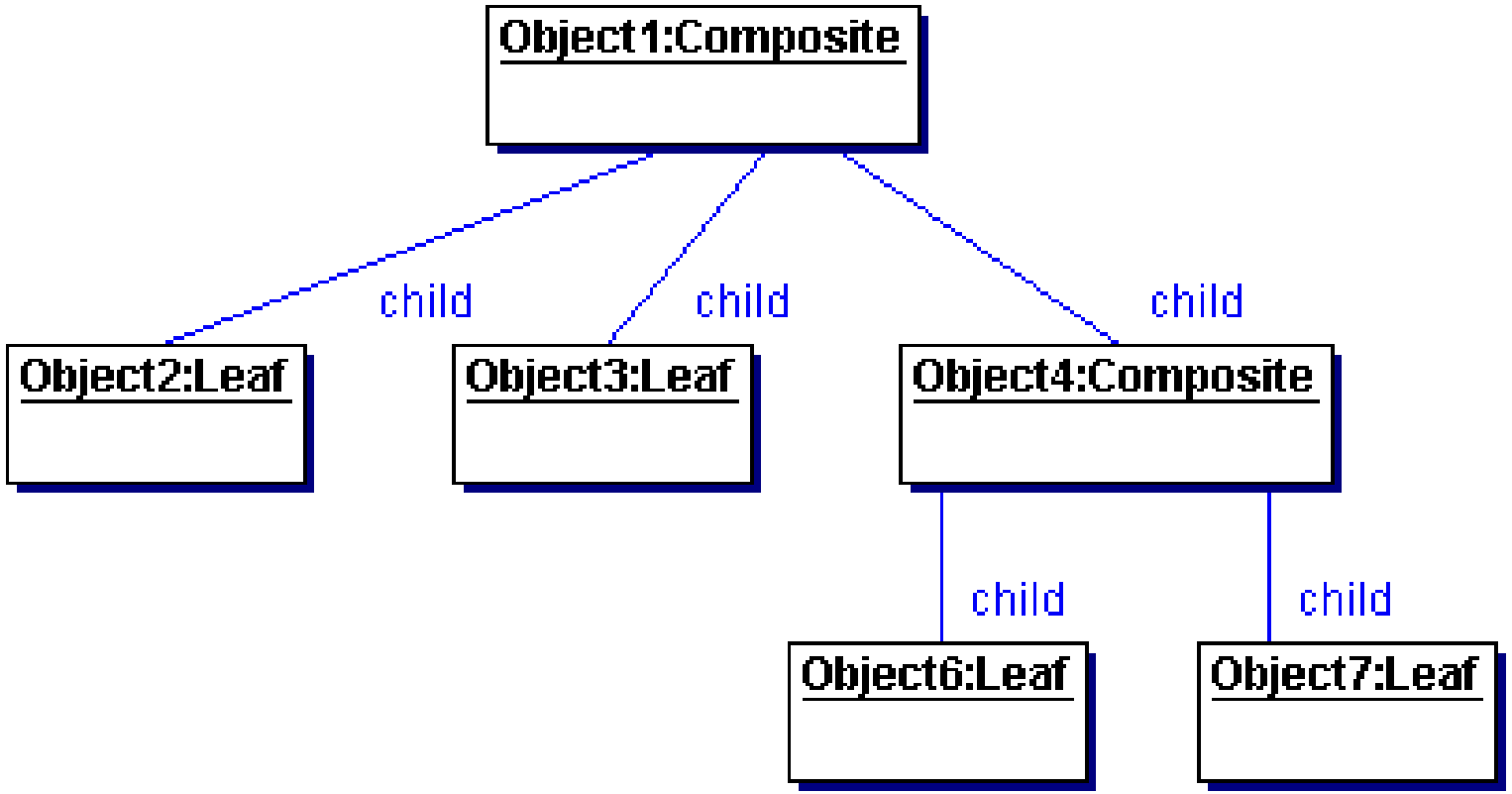
# Composite: structure



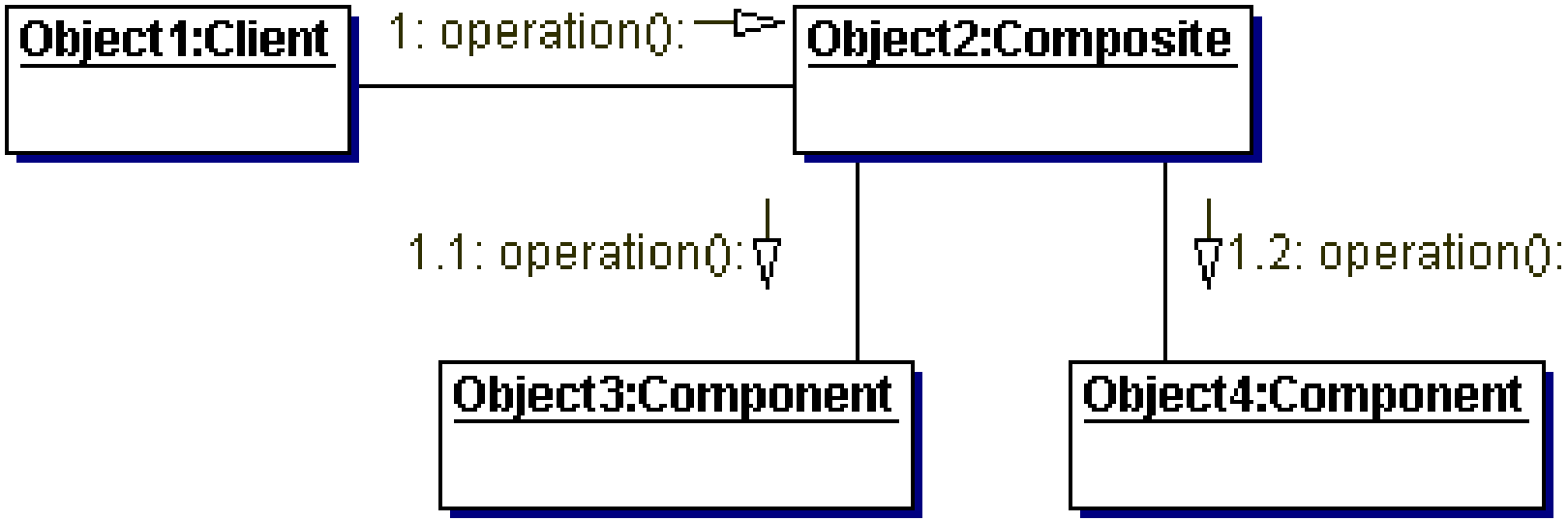
# Composite: participants

- Component
  - declares the interface for object composition
  - implements default behaviour (if any)
  - declares an interface for accessing and managing the child components
- Leaf
  - Defines the behaviour of the composition primitive objects
- Composite:
  - defines behaviour for components having children
  - stores child components
  - implements operations to access childs
- Client:
  - manipulates objects in the composition through the Composite interface

# Composite: Example

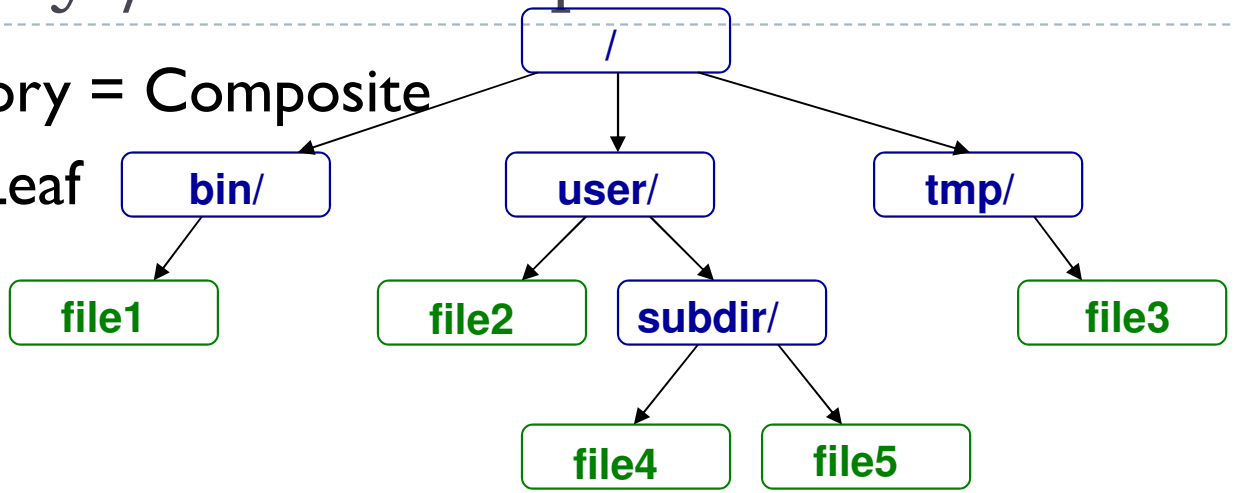


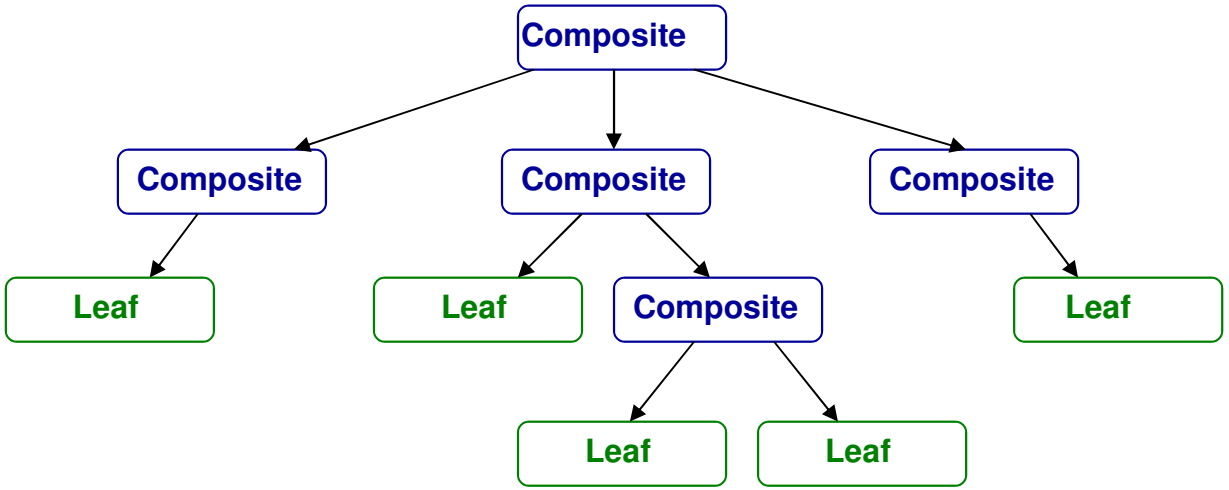
# Composite: Collaboration



# Directory / File Example

- Directory = Composite
- File = Leaf

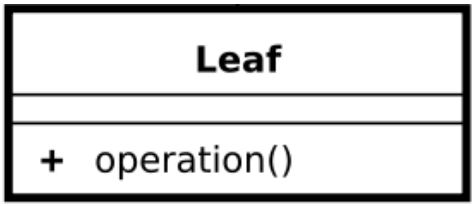


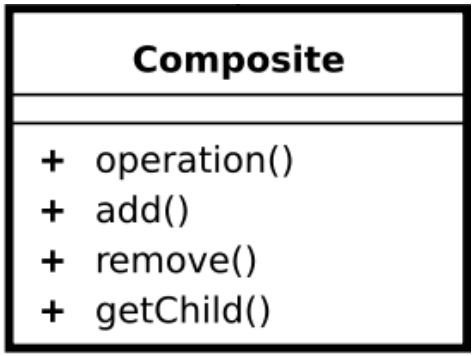



# Directory / File Example – Classes

- 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

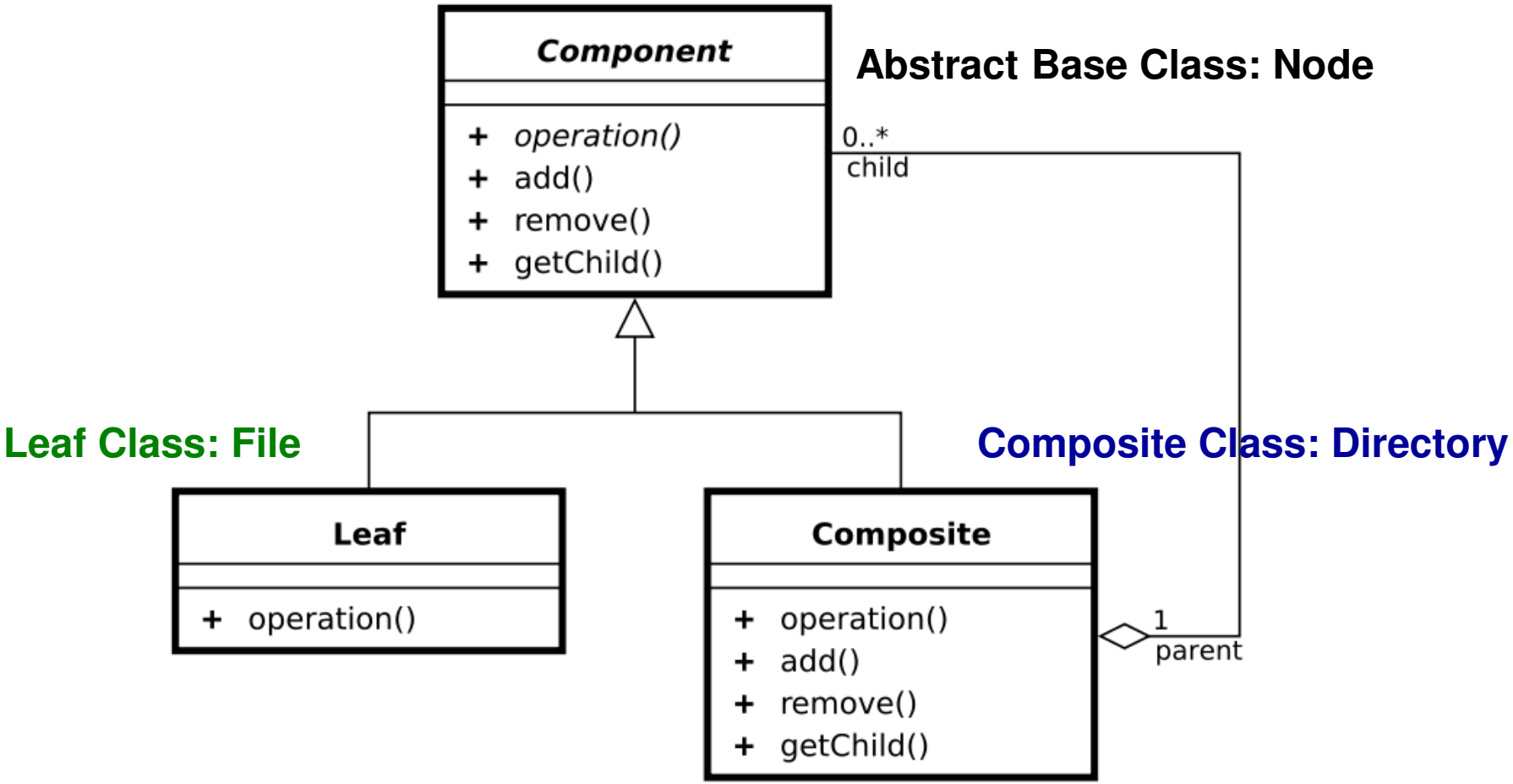
**Leaf Class: File**



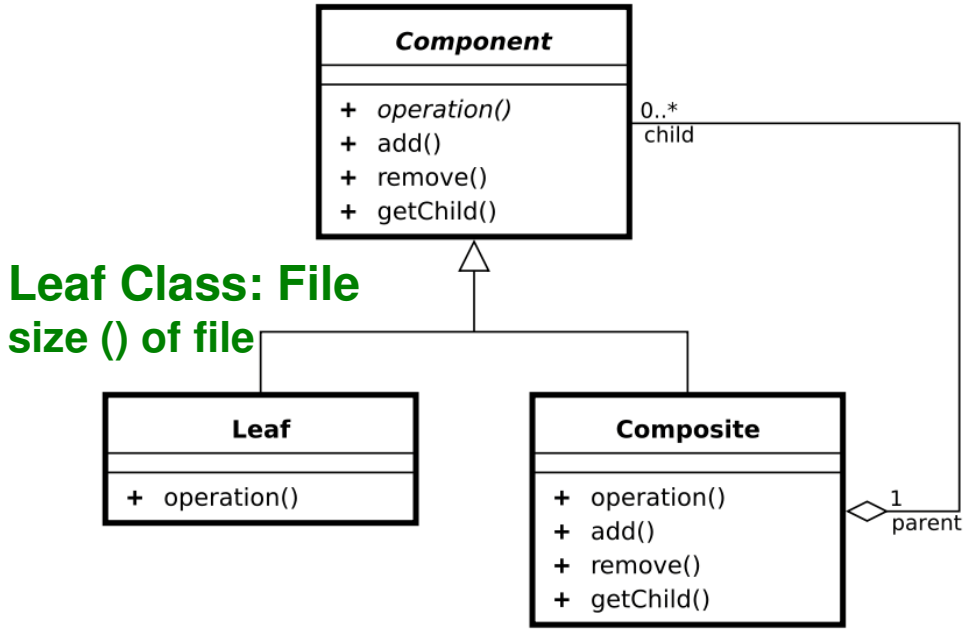




**Composite Class: Directory**

# Directory / File Example – Structure



# Directory / File Example – Operation




**Leaf Class: File**
**size () of file**

**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.
- Where should the child management methods (add(), remove(), getChild()) be declared?
  - In the Component class: Gives transparency, since all components can be treated the same. But it's not safe, since clients can try to do meaningless things to leaf components at run-time.
  - In the Composite class: Gives safety, since any attempt to perform a child operation on a leaf component will be caught at compile-time. But we lose transparency, since now leaf and composite components have different interfaces.

# Implementation Issues cont'd

- Is child ordering important?
  - Depends on application
- What's the best data structure to store components?
  - Depends on application
- A composite object knows its contained components, that is, its children. Should components maintain a reference to their parent component?
  - Depends on application, but having these references supports the Chain of Responsibility pattern