Composite
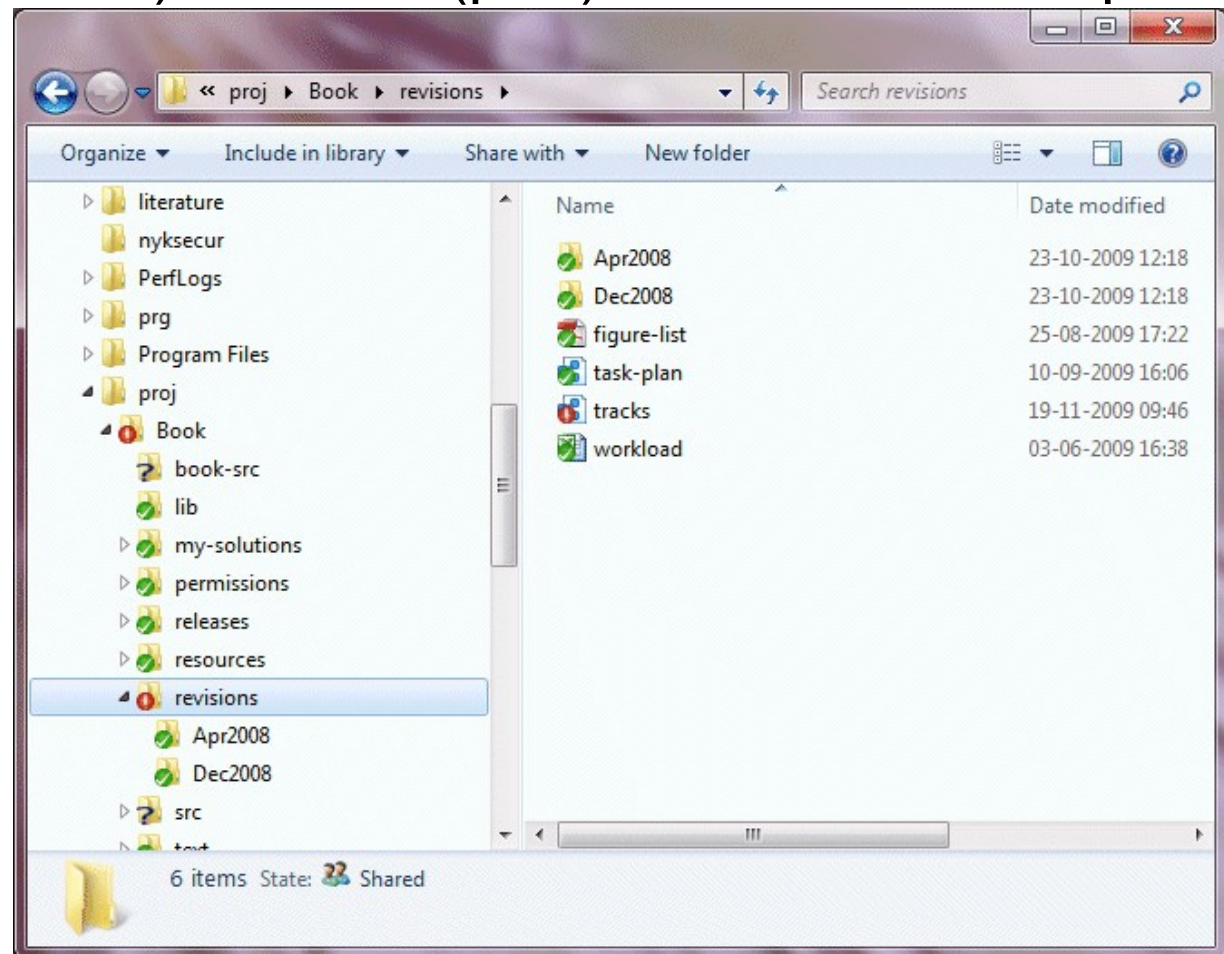
The Recursive Tree Structure in Design Pattern terms…

Hierarchical data structures pervade IT systems
– Folders (whole) and files (part) is a classic example

Using the **model perspective** (who/what) we focus on concepts in the domain:

- Who: Folder and File

- What: Very different things

  - Folder: addFile, addFolder, removeFile, etc.

  - File: open, close, getType, getSize, setReadOnly

Using a **responsibility perspective** (what/who) we instead focus on behavior:

- What: calculate size, move in structure, delete, set to read only

- Who: actually both folders and files…

Design 1:

– Make disjoint classes as they are disjoint concepts
  • class Folder {…} and class File {…}

But – will require a lot of casting…

```
private static void displaySize(Object item) {
  if (item instanceof File) {
    File file = (File) item;
    System.out.println( "File size is "+file.size() );
  } else if (item instanceof Folder) {
    Folder folder = (Folder) item;
    System.out.println( "Folder size is "+folder.size() );
  }
}
```

This *if* section will appear in every shared operation!

Design 2: ① *Program to an Interface*

Fragment: chapter/composite/CompositeDemo.java

```java
/** Define the Component interface
 * (partial for a folder hierarchy) */
interface Component {
  public void addComponent(Component sibling);
  public int size();
}
```

Fragment: chapter/composite/CompositeDemo.java

```java
/** Define a (partial) folder abstraction */
class Folder implements Component {
  private List<Component> components = new ArrayList<Component>();
  public void addComponent(Component sibling) {
    components.add(sibling);
  }
  public int size() {
    int size = 0;
    for ( Component c: components ) {
      size += c.size();
    }
    return size;
  }
}
```

Notice that this is a recursive depth-first descent into the tree…

```java
Fragment: chapter/composite/CompositeDemo.java
/** Define a (partial) folder abstraction */
class Folder implements Component {
  private List<Component> components = new ArrayList<Component>();
  public void addComponent(Component sibling) {
    components.add(sibling);
  }
  public int size() {
    int size = 0;
    for ( Component c: components ) {
      size += c.size();
    }
    return size;
  }
}
```

The Pattern

[26.1] Design Pattern: Composite

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

Problem   Handling of tree data structures.

Solution   Define a common interface for composite and atomic components alike. Define composites in terms of a set of children, each either a composite or atomic component. Define composite behaviour in terms of aggregating or composing behaviour of each child.

Structure:
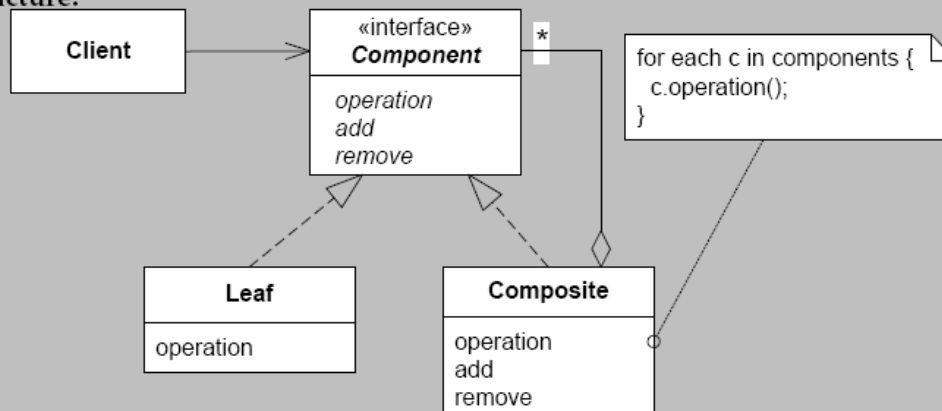
Roles   **Component** defines a common interface. **Composite** defines a component by means of aggregating other components. **Leaf** defines an primitive, atomic, component i.e. one that has no substructure.

Cost - Benefit   It defines a *hierarchy of primitive and composite objects*. It makes the *client interface uniform* as it does not need to know if it is a simple or composite component. It is *easy to add new kinds of components* as they will automatically work with the existing components. A liability is that the *design can become overly general* as it is difficult to constrain the types of leafs a composite may contain. The *interfaces may method bloat* with methods that are irrelevant; for instance an add method in a leaf.

Whole and part objects are treated identically

– Makes the client code much easier, avoiding a lot of testing on component types

Easy to add new types of components

– The Linux/Windows explorer can browse and manipulate any file, even those not known at deploy time.

Nonsense methods

– addComponent(Component c) is nonsense for Leaf/File

– i.e. Cohesion is low for Leaf ☹