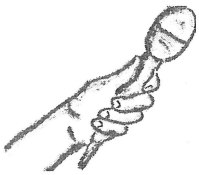

Patterns Exposed

This week's interview:
The Composite Pattern, on Implementation issues

HeadFirst: We're here tonight speaking with the Composite Pattern. Why don't you tell us a little about yourself, Composite?

Composite: Sure... I'm the pattern to use when you have collections of objects with whole-part relationships and you want to be able to treat those objects uniformly.

HeadFirst: Okay, let's dive right in here... what do you mean by whole-part relationships?

Composite: Imagine a graphical user interface; there you'll often find a top level component like a Frame or a Panel, containing other components, like menus, text panes, scrollbars and buttons. So your GUI consists of several parts, but when you display it, you generally think of it as a whole. You tell the top level component to display, and count on that component to display all its parts. We call the components that contain other components, composite objects, and components that don't contain other components, leaf objects.

HeadFirst: Is that what you mean by treating the objects uniformly? Having common methods you can call on composites and leaves?

Composite: Right. I can tell a composite object to display or a leaf object to display and they will do the right thing. The composite object will display by telling all its components to display.

HeadFirst: That implies that every object has the same interface. What if you have objects in your composite that do different things?

Composite: Well, in order for the composite to work transparently to the client, you must implement the same interface for all objects in the composite, otherwise, the client has to worry about which interface each object is implementing, which kind of defeats the purpose. Obviously that means that at times you'll have objects for which some of the method calls don't make sense.

HeadFirst: So how do you handle that?

Composite: Well there's a couple of ways to handle it; sometimes you can just do nothing, or return null or false – whatever makes sense in your application. Other times you'll want to be more proactive and throw an exception. Of course, then the client has to be willing to do a little work and make sure that the method call didn't do something unexpected.

HeadFirst: But if the client doesn't know which kind of object they're dealing with, how would they ever know which calls to make without checking the type?

Composite: If you're a little creative you can structure your methods so that the default implementations do something that does make sense. For instance, if the client is calling getChild(), on the composite this makes sense. And it makes sense on a leaf too, if you think of the leaf as an object with no children.

HeadFirst: Ah... smart. But, I've heard some clients are so worried about this issue, that they require separate interfaces for different objects so they aren't allowed to make nonsensical method calls. Is that still the Composite Pattern?

Composite: Yes. It's a much safer version of the Composite Pattern, but it requires the client to check the type of every object before making a call so the object can be cast correctly.

HeadFirst: Tell us a little more about how these composite and leaf objects are structured.

Composite: Usually it's a tree structure, some kind of hierarchy. The root is the top level composite, and all its children are either composites or leaf nodes.

HeadFirst: Do children ever point back up to their parents?

Composite: Yes, a component can have a pointer to a parent to make traversal of the structure easier. And, if you have a reference to a child, and you need to delete it, you'll need to get the parent to remove the child. Having the parent reference makes that easier too.

HeadFirst: There's really quite a lot to consider in your implementation. Are there other issues we should think about when implementing the Composite Pattern?

Composite: Actually there are... one is the ordering of children. What if you have a composite that needs to keep its children in a particular order? Then you'll need a more sophisticated management scheme for adding and removing children, and you'll have to be careful about how you traverse the hierarchy.

HeadFirst: A good point I hadn't thought of.

Composite: And did you think about caching?

HeadFirst: Caching?

Composite: Yeah, caching. Sometimes, if the composite structure is complex or expensive to traverse, it's helpful to implement caching of the composite nodes. For instance, if you are constantly traversing a composite and all its children to compute some result, you could implement a cache that stores the result temporarily to save traversals.

HeadFirst: Well, there's a lot more to the Composite Patterns than I ever would have guessed. Before we wrap this up, one more question: What do you consider your greatest strength?

Composite: I think I'd definitely have to say simplifying life for my clients. My clients don't have to worry about whether they're dealing with a composite object or a leaf object, so they don't have to write if statements everywhere to make sure they're calling the right methods on the right objects. Often, they can make one method call and execute an operation over an entire structure.

HeadFirst: That does sound like an important benefit. There's no doubt you're a useful pattern to have around for collecting and managing objects. And, with that, we're out of time... Thanks so much for joining us and come back soon for another Patterns Exposed.