Hierarchically Managed Attributes

Jeff Patton

jpatton@acm.org

Abstract: Assigning and maintaining values for attributes of large numbers of business objects can be cumbersome. Hierarchically Managed Attributes offers a simple solution by arranging business objects into categorized hierarchies, then assigning and maintaining those attributes at a category level.

Hierarchically Managed Attributes

A Business Object [C2Wiki] is often used to represent a conceptual object such as a customer or inventory item found in a business domain. Each of these business objects may have many attributes with their respective values. For instance, a customer might have a credit limit, or an inventory item might have a price. Creating and assigning attributes to business objects one by one is relatively straightforward. But in many domains, where there may hundreds of thousands or even millions of a given business object type, assigning and managing these attributes can be very time consuming and complex.

> *How can I effectively assign and maintain attributes over a large number of Business Objects?*

Business objects will likely have attributes with values assigned to them. They may be as simple as those mentioned previously: a credit limit to a customer, or a price assigned to an inventory item. They may be complex like a computational *Strategy* [GoF p. 315] for a rate applied to an insurance policy. Domains such as retail or insurance will generally manage huge numbers of business objects such as inventory items or customers. Keeping item or customer information accurate can be a time consuming maintenance task. Maintaining attributes that change frequently, such as sale price on an item, must be done efficiently and accurately to support the needs of the business.

An obvious solution might be to design business objects with their necessary attributes, then allow users of software to maintain values for these attributes. If I'm a grocer who chooses to discount all my fresh fruit next week, this could be a time consuming task. I must locate all items that are fresh and create a discount for them. There may be a few dozen items that are fresh fruit so I'll need to make sure the discount attribute is accurately entered and duplicated for all of them.

Because managing large quantities of business objects can be cumbersome, it's useful to categorize or arrange those objects into a hierarchy. For example customers are often arranged into hierarchies based on geographical area. They might be grouped by country then by state or province within that country. Inventory items are usually arranged into a merchandise hierarchy. A grocery store might group items by department, family group, and commodity class: "Produce, fresh fruit, apples" for example. With all their items categorized this way, a grocer might easily be able to say "Next week, discount all fresh fruit by 25%."

Arranging business objects into such a hierarchy can help simplify attribute maintenance from the system user's perspective. Working at a grocer with a system built this way, I might simply locate all items in the fresh fruit category and indicate a scheduled discount of 25% on those items. The system could then locate each item that is in the fresh fruit category, and make the attribute adjustment for me. This saves me lots of time, but the system still stores and maintains essentially the same data as there would be without the categorization since it still carries the discount attribute for each item, and a value of 25%.

But there's a problem: I believe there's a new kind of apple from New Zealand coming in next week, and I definitely want that included, but it isn't yet entered into the database. If I make the change to all fresh fruit today, the system doesn't yet know about the new apple, and I know from experience the discount attribute won't get set on the new apple. Then come sale day, all fresh fruit except the new apple will be discounted. This makes my customers unhappy. I'll have to make a note to myself to make that entry when the new apple comes in and is entered into the database. This wouldn't be so bad if there weren't so many other similar discounts that needed to be created for ongoing promotions for next week and into the future.

And it gets worse: someone using the system inappropriately categorized Roma tomatoes as fresh fruit. Now I know that scientifically speaking a tomato might be a fruit, but most folks

consider it a vegetable. So, catching this mistake, I reassign the Roma Tomato to the "fresh vegetables" category. However, I don't notice that the 25% discount I'd indirectly assigned earlier to "fresh fruit" was kept with it. Since for our purposes it's not really a fruit, it shouldn't get that discount.

The maintenance time and risk of error is daunting. How can I make assigning temporary price reductions to large numbers of items more simple?

> *Arrange business objects into a hierarchical structure and use **Hierarchically Managed Attributes** to assign attributes to the elements of the hierarchy rather than directly to the item.*

Start by creating **Categories** to group items. A category may have child categories, or may have child business objects. From the example above I might create a category called *produce*, a child of that category called *fresh fruit*, and children of *fresh fruit* called *apples* and *oranges*.

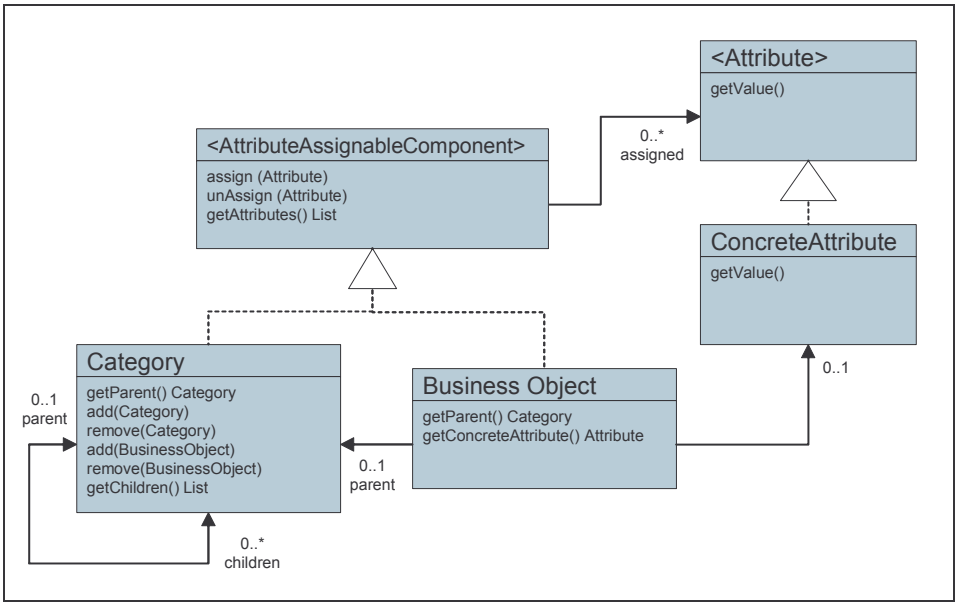
Assign each **Business Object** to a parent category. In the example above, I'd assign each specific item business object that's an apple to the *apples* category, all the oranges to the *oranges* category.

Create an **Attribute** for each type of attribute you'd like to manage. In my system I'll create an attribute for *temporary price discount*. It will have a start date, a stop date, and a discount percentage.

Consider both **Categories** and **Business Objects** to be **Attribute-Assignable** then assign attribute values to appropriate levels of the hierarchy. To take care of my *fresh fruit* discount I'll create a new *discount* attribute for next week at 25% and assign it to the *fresh fruit* category. I can be sure that all child categories, such as *apples* and *oranges*, and their child items will get the discount. If I add or remove item business objects from the *fresh fruit*, *apples*, or *oranges* categories they they'll get the appropriate discount attribute.

Structure

In its simplest form, *Hierarchically Managed Attributes* looks and behaves much like the *Composite [GoF p163]* pattern where all composite components support the operation of assigning and retrieving attributes.



AttributeAssignableComponent: the declared interface for a component that supports the assignment of attributes.

Category: an implementer of AttributeAssignableComponent representing a generic name for a group of Business Objects or Categories. The Category is a composite component that holds a collection of child Business Objects and/or Categories.

BusinessObject: an implementer of AttributeAssignableComponent representing an object encapsulating the attributes and behavior of a real world object. The type name of the business object and its attributes, including inherited attributes, are usually readily identifiable in the domain language of the business problem being solved.

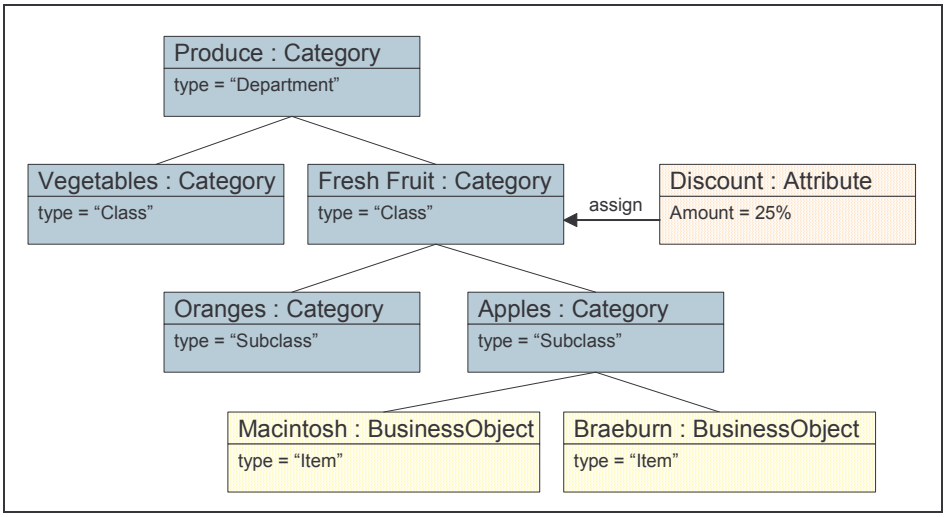
Attribute: the declared interface for and assignable attribute. The Attribute allows access to the value of that attribute.

ConcreteAttribute: an implementer of Attribute representing a specific type of attribute and its value.

Collaborations

- Business Objects are created to represent any number of domain specific objects.
- Categories are created to represent generalized categories of business objects.
- Categories are arranged into a hierarchical scheme by adding child categories to parent categories.
- Business Objects are associated with a parent Category by adding them to that Category.
- Attributes are created to hold a value that is correct for many types of Business Objects.
- An instance of an attribute is assigned to a Category or BusinessObject.
- A BusinessObject may return any or all Attributes assigned to it or its parent Categories.

Using this structure, the example we've been using might look a bit like this.



Resulting Consequences

Specific hierarchically maintainable attributes may be created and added at any time to adapt to the needs of the business.

Using very few simple attribute assignments, a specific attribute instance such as a sale price might be assigned to a large number of business objects by assigning it to a parent category of those business objects.

An attribute may be maintained independently of its assignment. For example I might create a sale price of 25% off and assign it to a category containing large numbers of items. I might later decide to increase that discount to 40%. I need only modify the attribute value. All items that have that attribute assigned directly, or indirectly through a parent category, will reflect that new attribute value.

Business Objects may be added or removed from categories. This may result in adding or removing attributes and values. For example I might add an item to a category where items in that category have an attribute for their tax rate. The new item will automatically receive the tax rate attribute shared by other items in the category. Moving that item to another category will result in it receiving the attribute value for the new category. No item level maintenance is necessary.

Attributes may be unassigned or reassigned to a category. I may create an attribute with a specific value such as a sale for 25% off. I could assign it to multiple categories. I might later decide that this sale attribute isn't appropriate for a particular category. I need only remove the assignment to that category. All items contained within that category will reflect the change in sale status.

Considerations

When implementing hierarchically assigned attributes there are several issues to consider. There are also variations that add flexibility while potentially increasing complexity.

You must decide if attribute assignments accumulate or replace each other. For example I might have a sale price attribute. I might create an instance of a sale price with a value of 25% off and assign that to a category – like *fresh fruit* from our example. I might also create an instance of a sale price with a value of 40% off and assign it to a child category – like *apples* from our example. For business objects that are members of the *apples* category, you must decide if they receive the 40% discount (replaced assignment) or both the 25% and 40% attributes (accumulated assignments.)

You must determine if it's possible to remove assignments at lower category levels. For example I may create a sale price of 25% off and assign it to all my *fresh fruit*. However, the *berries* have a lower profit margin, so I'd rather not extend the sale to those *fresh fruits*. To accommodate this situation, I must support removing the sale price attribute for *berries* to *break* the sale price assignment inherited from the parent *fresh fruit* category.

You must determine if your business objects are arranged in a true hierarchy or simply categorized. To use a different retail example, you might have web store that allows customers to browse your merchandise by category. If you sell books, you might have a category for *best-sellers*, and a category for *mystery-suspense*. A book like The da Vinci Code might appear in both categories since it is both. This makes your categorization scheme not a true hierarchical structure since this item belongs to multiple categories. You must first decide if this is allowed in your categorization structure. Assuming it is, you must then determine if this particular book title inherits attributes from one or both categories. If you choose to inherit attributes from both, then you must determine if these attributes accumulate or replace each other. If you choose to inherit

from one category over another, you must determine which category is most appropriate to inherit from.

When using an imperfect hierarchy it can be helpful to attach precedence to each category. In the example above, *best-sellers* may have precedence over other categories. In situations where an item inherited an attribute from the *best-sellers* category as well as another category, the *best-sellers* category would take precedence displacing other attributes. For example if the *mystery-suspense* category carried a discount attribute of 20%, but the *bestseller* category carried a discount of 30%, since the *bestseller* category has higher precedence, a book in both categories would receive the 30% discount attribute.

Complex business objects may sometimes inherit attributes assigned via multiple categories. For example, I might assign a discount of 30% to *bestseller* books. All books in the *bestseller* category would get that discount. I might also create a special discount of 10% for *preferred customers*. All customers who've purchased more than a certain number of books from our web store receive this discount. As customers purchase more books, they may be assigned to the *preferred customer* category. If a *preferred customer* comes up to the website with a shopping cart of books to purchase, the software may create a *Transaction* for her. That *Transaction* might be a business object holding a *Customer* business object along with business object for each *Book* our customer intends to purchase. As books are added to the transaction, a discount attribute may be looked up from two categorizations: the customer's categorization, and the book's categorization. Since the customer is a preferred customer all items will receive a 10% discount. If some items are bestsellers, those items will receive the additional 30% discount – the discount assigned to bestsellers.

Summary

This pattern discusses a simple implementation and some considerations. You'll find that Hierarchically Managed Attributes can be applied easily to many domains other than the retail domain examples given here. For instance:

- File folders in a directory structure may have attributes assigned that individual files inherit
- Help desk incidents may be categorized by product, and escalation levels may be assigned to products then overridden by specific incident
- Employees might be categorized by location and role, and network security might be maintained by location and role

Hierarchically Managed Attributes are a simple solution for maintaining attributes and their values over large numbers of categorized Business Objects.

References:

[GoF] Gamma, Helm, Johnson, & Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, MA, 1995

[C2Wiki] Cunningham & Cunningham Wiki, http://c2.com/cgi/wiki?BusinessObject

Appendix A: Sample Implementation

The following code sample demonstrates a simple Java implementation of the *Hierarchically Managed Attributes* pattern.

<AttributeAssignableComponent>

```
public interface AttributeAssignableComponent {
	public void assign(Attribute attribute);
	public void unAssignAttribute(Attribute attribute);
	public List getAttributes();

}
```

<Attribute>

```
public interface Attribute {
	public Object getValue();
}
```

Category

```
public class Category implements AttributeAssignableComponent {
	final String name;
	Category parentCategory;
	List children = new ArrayList();
	List attributes = new ArrayList();

	public Category(String name) { this.name = name; }

	public String toString() { return name; }

	private final void setParentCategory(Category parent) {
		this.parentCategory = parent;
	}

	public Category getParentCategory() {
		return parentCategory;
	}

	public void add(Category category) {
		if (!children.contains(category)) {
			children.add(category);
			category.setParentCategory(this);
		}
	}

	public void add(BusinessObject obj) {
		if (!children.contains(obj)) {
			children.add(obj);
			obj.setParentCategory(this);
		}
	}

	public void remove(Category category) {
```

```
            if (category.getParentCategory() == this) {
                  category.setParentCategory(null);
                  children.remove(category);
            }
      }

      public void remove(BusinessObject obj) {
            if (obj.getParentCategory() == this) {
                  obj.setParentCategory(null);
                  children.remove(obj);
            }
      }

      public List getChildren() { return children; }

      public void assign(Attribute attribute) {
            if (!attributes.contains(attribute))
                  attributes.add(attribute);
      }

      public void unAssignAttribute(Attribute attribute) {
            attributes.remove(attribute);
      }

      public List getAttributes() {
            List allAssignedAttributes =
                  new ArrayList(attributes);
            if (parentCategory != null) {
                  allAssignedAttributes.addAll(
                              parentCategory.getAttributes());
            }
            return allAssignedAttributes;
      }
}
```

BusinessObject

```
public abstract class BusinessObject implements
AttributeAssignableComponent {

      final String name;
      Category parentCategory;
      List attributes = new ArrayList();

      public BusinessObject(String name) { this.name = name; }

      public String toString() { return name; }

      public String getName() {
            return name;
      }

      public void setParentCategory(Category parentCategory) {
            this.parentCategory = parentCategory;
      }
```

```
	public Category getParentCategory() {
		return parentCategory;
	}

	public void assign(Attribute attribute) {
		if (!attributes.contains(attribute))
			attributes.add(attribute);
	}

	public void unAssignAttribute(Attribute attribute) {
		attributes.remove(attribute);
	}

	public List getAttributes() {
		List allAssignedAttributes =
			new ArrayList(attributes);
		if (parentCategory != null) {
			allAssignedAttributes.addAll(
					parentCategory.getAttributes());
		}
		return allAssignedAttributes;
	}
}
```

Item

```
public class Item extends BusinessObject {

	public Item(String name) { super(name); }

	public Price getPrice() {
		for (ListIterator it = getAttributes().listIterator();
				it.hasNext(); ) {

			Attribute attribute = (Attribute) it.next();
			if (attribute instanceof Price)
			    return (Price) attribute;
		}
		return null;
	}

	public Discount getDiscount() {
		for (ListIterator it = getAttributes().listIterator();
				it.hasNext(); ) {

			Attribute attribute = (Attribute) it.next();
			if (attribute instanceof Discount)
			    return (Discount) attribute;
		}
		return null;
	}
}
```

Concrete Attributes

```
public class Price implements Attribute {
       final Double price;
       public Price(double price) { this.price = new Double(price); }
       public Object getValue() { return price; }
       public String toString() {return "Price: "+price; }
}

public class Discount implements Attribute {
       final Double discount;
       public Discount(double discount) {
           this.discount = new Double(discount);
       }
       public Object getValue() { return discount; }
       public String toString() {return "Discount: "+discount; }
}
```

Unit Test Code:

When writing code, it's helpful to drive design using unit tests. Unit tests often act as a "client" that demonstrates usage of the code. Consequently, reading unit test code can often say more about the design of the software than read the software code directly.

```
public class HeirarchicallyManagedAttributesTest extends TestCase {

       public void testHierarchyCanBeBuilt() {
              Category department =
                     new Category("produce");
              Category familyGroup =
                     new Category("fresh fruit");
              department.add(familyGroup);
              assertEquals("category had the correct number of subcategories",
                           1, department.getChildren().size());
              assertSame("category has correct subcategory",
                           familyGroup, department.getChildren().get(0));
              assertSame("subcategory has correct parent category",
                           department, familyGroup.parentCategory);

              Category commodityClass1 =
                     new Category("apples");
              Category commodityClass2 =
                     new Category("oranges");
              familyGroup.add(commodityClass1);
              familyGroup.add(commodityClass2);
              assertEquals("category has correnct number of subcategories",
                           2, familyGroup.getChildren().size());
              assertTrue("category has correct subcategories",
                     commodityClass1 == familyGroup.getChildren().get(0) &&
                     commodityClass2 == familyGroup.getChildren().get(1));
              assertTrue("categories have correct parents",
                     familyGroup == commodityClass1.parentCategory &&
                     familyGroup == commodityClass2.parentCategory);

              for (int i = 1; i <= 10; i++) {
                     Item item = new Item("item "+i);
                     if (i % 2 == 1) {
                            commodityClass1.add(item);
                     } else {
                            commodityClass2.add(item);
                     }
              }
```

```
                assertEquals("category has correct number of leaf nodes",
                             5, commodityClass1.getChildren().size());
                assertEquals("category has correct number of leaf nodes",
                             5, commodityClass2.getChildren().size());
        }

        public void testAttributesCanBeAssignedAndLookedUp() {
                Category produce =
                        new Category("produce");
                Category fruit =
                        new Category("fresh fruit");
                produce.add(fruit);
                Category apples =
                        new Category("apples");
                Category oranges =
                        new Category("oranges");
                fruit.add(apples);
                fruit.add(oranges);
                for (int i = 1; i <= 10; i++) {
                        Item item = new Item("item "+i);
                        if (i % 2 == 1) {
                                apples.add(item);
                        } else {
                                oranges.add(item);
                        }
                }

                Item anApple = (Item) apples.getChildren().get(0);
                Item anOrange = (Item) oranges.getChildren().get(0);

                apples.assign(new Price(1.59));
                assertEquals("apples is price correctly",
                             new Double(1.59), anApple.getPrice().getValue());

                oranges.assign(new Price(0.99));
                assertEquals("orange is price correctly",
                             new Double(0.99), anOrange.getPrice().getValue());

                fruit.assign(new Discount(25.0));
                assertEquals("apple is discounted correctly",
                             new Double(25.0), anApple.getDiscount().getValue());
                assertEquals("orange is discounted correctly",
                             new Double(25.0), anOrange.getDiscount().getValue());
                oranges.assign(new Discount(50.0));
                assertEquals(
                    "discounts assigned at lower category level take precidence",
                     new Double(50.0), anOrange.getDiscount().getValue());
        }
}
```