Extending the Collection Hierarchy

Juanita J. Ewing

Instantiations, Inc.

Derived from Smalltalk Report

Last month, I discussed creating subclasses and two heuristics for selecting superclasses. This month I will continue the discussion about subclassing with a case study that extends the Collection hierarchy. In this case study, we will create a new Collection class that contains unique elements and also maintains the order of these elements.

Heuristics Review

A key step in creating a new subclass is to select a suitable superclass. The heuristics for selecting a superclass are:

Heuristic One: Look for a class that fits the is-kind-of or is-type-of relationship with your new subclass.

Heuristic Two: Look for a class with behavior that is similar to the desired behavior of the new subclass.

Case Study

We want create a new data structure class that holds elements in order and disallows duplicate elements. When sent a request to add a duplicate object, the request should be quietly ignored.

This new data structure class contains elements, so is similar to Arrays, Strings and other Collection subclasses. Because of these similarities, we will begin our search for candidate superclasses in the Collection hierarchy. Two classes immediately stand out.

- OrderedCollections keep elements in order.
- Sets store each element only once, disallowing duplicate elements.

The combination of these characteristics is what we are seeking for our new class. A good descriptive name for our new class is OrderedSet.

Apply Heuristics

Where should we insert our new class, OrderedSet, into the hierarchy? Our first heuristic is to look for potential superclasses that match the is-kind-of criteria. We use is-kind-of as a shorthand for categorization based on characteristics. The significant characteristics we use in this determination, and the classes that have them are:

- varying number of elements (Collection)
- store arbitrary objects (Collection)
- dynamically add and remove elements (Collection)
- enumeration (Collection)
- store elements in order (OrderedCollection)
- store unique elements (Set)

The desired characteristics of OrderedSet are closest to those of OrderedCollection and Set, so OrderedSet could be a-kind-of Set or a-kind-of OrderedCollection.

In a system that supports multiple inheritance, we might be tempted to have two superclasses, Set and OrderedCollection. In Smalltalk we must choose a single superclass, either Set or OrderedCollection.

Our second heuristic is to choose candidate superclasses with suitable public behavior. Let's compare the candidate classes we've selected, Set and OrderedCollection , in terms of behavior. Set and OrderedCollection have a common superclass, Collection, so we will ignore public behavior from the Collection on up.

If we were to make OrderedSet a subclass of Set, it would inherit these methods from Set.

```
add:
do:
includes:
occurencesOf:
remove:ifAbsent:
size:
```

All of these methods also have an implementation in the abstract superclass Collection, so Set doesn't add any new public behavior to the behavior from the common superclass.

If OrderedSet were a subclass of OrderedCollection, it would inherit behavior from OrderedCollection and IndexedCollection (or OrderedCollection and

SequencableCollection in Objectworks\Smalltalk™). OrderedCollection has adding and removing methods and many more methods related to its element ordering characteristic. The list of methods includes

add:
add:after:
add:afterIndex:
add:before:
add:beforeIndex:
addFirst:
addLast:
remove:ifAbsent:
removeFirst:
removeLast:

Many of these methods are extensions of the public behavior from the common superclass Collection.

The public behaviors for Sets and OrderedCollections have some similarities. In fact, the behavior of Set is a subset of the behavior of OrderedCollection, which makes Set the behavioral supertype of OrderedCollection. Set doesn't add any additional behavior, so we just need to determine whether the additional behavior in OrderedCollection is desirable.

Because instances of OrderedSet maintain elements in order, we will need public behavior to support the ordering characteristic. The behavior in OrderedCollection is a good set of behavior for supporting this characteristic. In addition, if the behavior of OrderedSet is the same as for OrderedCollection, the interchangeability of the classes is better and therefore the classes are easier to reuse. Based on behavioral analysis, the best superclass for OrderedSet is OrderedCollection.

Implementation

We can also look in more detail at what is required to implement OrderedSet. The implementation of OrderedCollection uses an indexable portion or indexable object, and instance variables to keep track of the indices that are valid. Set is implemented with hashing for efficiency in determining uniqueness of elements. If a Set already contains an element, it quietly ignores the request to add an element.

OrderedSet needs to support instances with a large number of elements. Hashing the elements is a good way to support large numbers. OrderedCollections would potentially have to examine every element before determining if the addition of an element would be a duplication. To maintain order and enforce uniqueness we will use two structures, one to implement the unique elements characteristic, and one to implement the ordering characteristic, as shown in Figure 1.

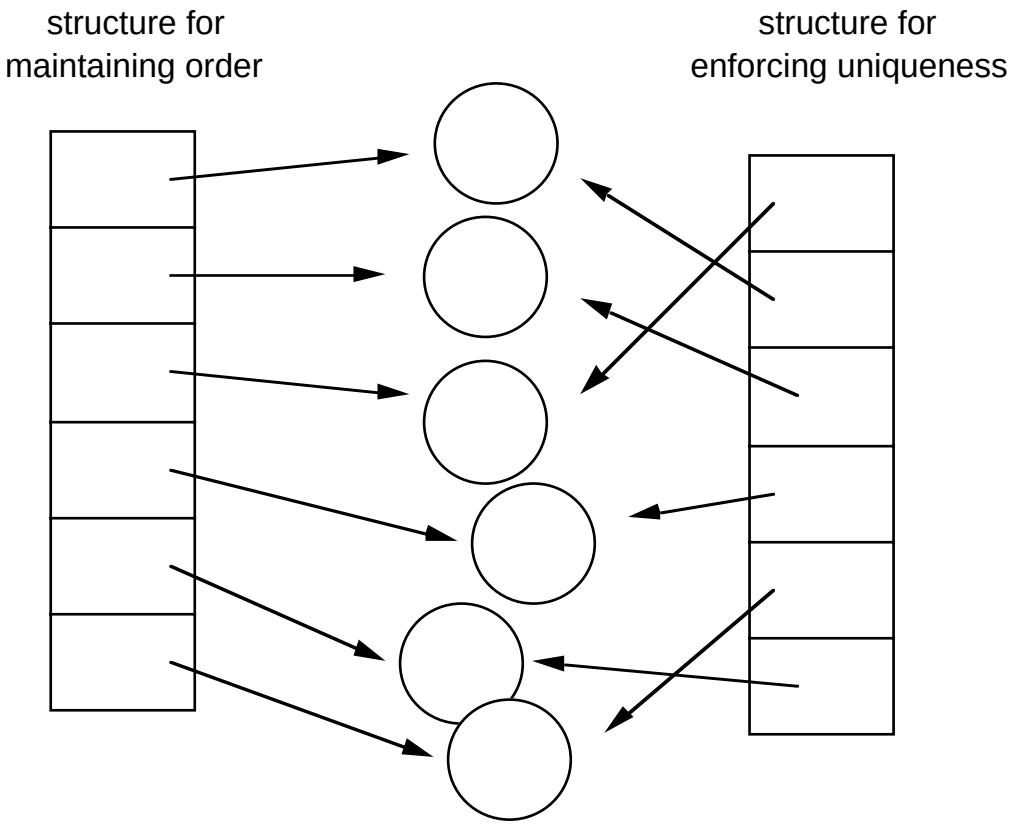

Figure 1. Using multiple structures.

Now we will examine the implementations with each of our candidate superclasses. If OrderedSet is a subclass of OrderedCollection, we inherit the portion that stores elements in order and need to implement the portion that does the hashing and enforces uniqueness. The structure and behavior for maintaining order is inherited from OrderedCollection, and the structure for enforcing uniqueness can be stored in an instance variable. This structure could be an instance of Set.

With this alternative, some inherited methods would need to be overridden. All the add and remove methods must potentially be altered to maintain both structures. As we saw in the list of public behavior, there are a number of these methods, such as add:, add:after:, add:afterIndex:, addFirst:, removeFirst and removeLast. Fortunately, not all of these methods have to be overridden because some of these methods call each other. We would also want to override includes: because the hashing used in the uniqueness structure gives us a quick look up of elements. We wouldn't override do: because it operates on the inherited structure that maintains order.

If OrderedSet were a subclass of Set, the inherited structure is the one that enforces uniqueness and an auxiliary structure for maintaining order is referenced from an instance variable. Presumably, the order maintaining structure would be an instance of OrderedCollection .

We would also need to override adding and removing methods, and there is just one of each. The majority of the coding is in implementing behavior that implements the element ordering characteristic. We wouldn't need to override includes: because we inherit the version that makes use of hashing, but we would need to override do: so that we process elements in the ordered defined by the order maintaining structure.

Naming

The other criteria that might bias our judgment is implications of a class name. If the class hierarchy is part of the public interface for a library, it might be easier for users to locate a class if it is located in a logical place in the hierarchy. If we have a class called OrderedSet, people are more likely to look for this class as a specialization of Set. They might not find the class as easily if it is a subclass of OrderedCollection.

Conclusion

We choose to make OrderedSet a subclass of OrderedCollection because

- the behavior of OrderedCollection is more suitable than the behavior of Set
- it is more likely that the behavior will be interchangeable if the relationship between the two classes is explicit
- there are fewer methods, overridden and new, that must be implemented in OrderedSet.

Furthermore, we argue that in browsing the Collection hierarchy, developers will generally examine several Collection classes at a time, and will probably notice OrderedSet as a subclass of OrderedCollection.

The is-kind-of heuristic is useful for generating candidate superclasses. It's intuitive nature can be an advantage. However, analysis of public behavior usually yields a better selection. If we used just the is-kind-of heuristic in our case study, we would be most likely to make OrderedSet a subclass of Set. Yet when we use the public behavior heuristic, we conclude that OrderedCollection is a better choice.