Creating Subclasses

Juanita J. Ewing

Instantiations, Inc.

Derived from Smalltalk Report

Class hierarchies are a way to capture variations and specializations. A subclass is generally a more specialized kind of entity than its superclass. For example, the class Sphere is a subclass of the class Solid. If we needed a representation for pyramids, we would create a new subclass, Pyramid, whose superclass is Solid.

In this example it is easy to decide how to how Pyramid fits into the hierarchy because there is an abstract superclass. This abstract superclass is a generalization representing different kinds of solids. Often the decision about where to insert a new class in the hierarchy is not straightforward. This column explores strategies for placing subclasses in a hierarchy and consequences of the placement.

Benefits

Well-formed class hierarchies are those in which functionality is factored into a number of classes. Subclasses are specializations, and superclasses are generalizations. When functionality is factored into hierarchies, classes are more reusable and maintainable. Highly factored hierarchies are also easier to extend.

Heuristics

A significant part of creating subclasses is choosing the most appropriate superclass. It is almost always better to inherit behavior rather than reimplement behavior, though not at the cost of inheriting inappropriate behavior. In order to inherit the greatest amount of appropriate behavior, we use two heuristics to select candidate superclasses.

Heuristic One: Look for a class that fits the is-kind-of or is-type-of relationship with your new subclass. Often it helps to make this heuristic into an English question. For example, we can ask the question, “Is a pyramid a-kind-of solid?”

Documentation describing a class often helps understand exactly what the class represents. Because of your understanding of classes that you implemented, it is much easier to insert new classes into hierarchies that you have developed. Personal knowledge of the class hierarchy can substitute for class documentation.

Heuristic Two: Look for a class with behavior that is similar to the desired behavior of the new subclass. In this heuristic you must look at the methods or good documentation for the methods. Often, just the message selectors will give you enough information to reject many inappropriate classes.

Behavioral Inheritance vs. Implementation Inheritance

The two heuristics we have presented are oriented towards class hierarchies based on behavior. This kind of inheritance is known as *behavioral inheritance*. In these hierarchies a subclass and its superclass have a subtype relationship. That is, the subclass supports all the behavior that the superclass supports, and the subclass can add new behavior. Any use of an instance of the superclass can be replaced by the use of an instance of the subclass. Some examples from Smalltalk class libraries are: RecordingPen is a subclass of Pen, Time is a subclass of Magnitude, Integer is a subclass of Number, and WildPattern is a subclass of Pattern.

Inheritance can also be used in a more pragmatic fashion, in which a class is placed in a hierarchy because of the desire to inherit code and implementation rather than behavior. Inheritance used in this fashion is called *implementation inheritance*. Most class libraries also have examples of this kind of inheritance: Process is a subclass of OrderedCollection, and Debugger is a subclass of Inspector.

BusRoute Example

An example involving bus routes will illustrate the different kinds of inheritance. In this example, we need to create a class to represent a bus route, which is used to inform the bus driver and passengers of the bus's path through the city. A bus route is a collection of bus stops, in a particular order. A bus route needs the ability to compose the route out of bus stops, to supply a summary report on the route's stops, to determine how many intermediate stops there are between two stops, and the fare from one stop to another. The fare computation may vary depending on which zones the stops are located in.

People who are familiar with Smalltalk class libraries will immediately start to think of the class OrderedCollection when they read the description of a bus route. OrderedCollection is a concrete collection class that holds elements in order, similar to a stack or queue. The elements can be of any type.

Implementation Inheritance Alternative

We need to make a new class, which we will call BusRoute. Should BusRoute be a subclass of OrderedCollection? As a subclass of OrderedCollection, it would inherit the implementation that maintains elements in order. It would also inherit the code for adding and removing elements which can be used to compose the bus route. This relationship is shown in Figure 1.

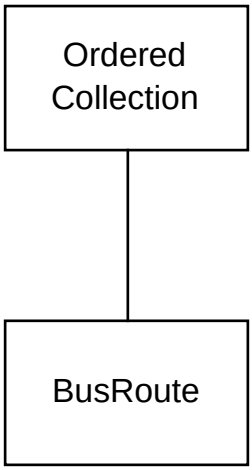

Figure 1. BusRoute as a subclass of OrderedCollection

It is useful to determine whether this placement of BusRoute uses behavioral inheritance or implementation inheritance. Is a bus route a-kind-of ordered collection? No. Instances of OrderedCollection have an implicit responsibility to hold objects of arbitrary type, and a bus route holds only bus stops. A BusRoute is not a generic data structure class.

Is all the behavior of OrderedCollection appropriate for BusRoute? No. According to the description, bus routes shouldn't respond to the do:, select: or reject: messages, or many of the other generic collection messages. Therefore BusRoute is not a subtype of OrderedCollection. Placing BusRoute as a subclass of OrderedCollection is an example of implementation inheritance, in which code and implementation are usefully inherited.

Behavioral Inheritance Alternative

Another alternative is to make BusRoute a subclass of some other class. A bus route is a-kind-of route. Are there any route classes in the Smalltalk library? If the answer is no, then make BusRoute a subclass of Object. The behavior of Object is appropriate for all objects, so Object is selected when there isn't any other appropriate superclass. This alternative is an example of behavioral inheritance because all the behavior in Object is appropriate for BusRoute. The inheritance relationship is shown in Figure 2.

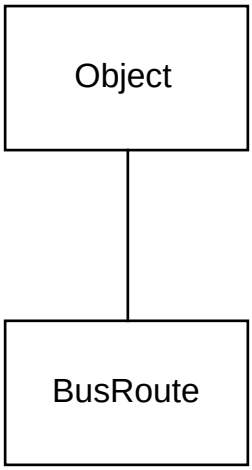

Figure 2. BusRoute as a subclass of Object.

In this alternative, BusRoute would collaborate with OrderedCollection to store bus stops in order. Figure 3 illustrates the collaboration between the two objects. An instance variable, busStops, references an instance of OrderedCollection that stores bus stops. Instances of BusRoute can relay messages to the instance of OrderedCollection referenced by the busStops instance variable.

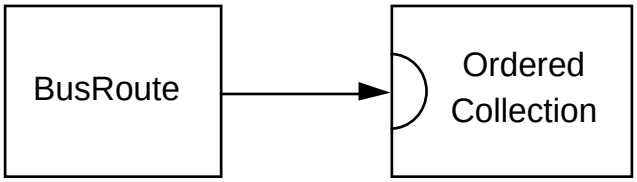

Figure 3. BusRoute collaborates with OrderedCollection.

Override Inappropriate Methods

In the first alternative, in which BusRoute was a subclass of OrderedCollection, we proposed using inherited public methods such as add: and remove: to compose the bus route. But this is not a very good way to compose bus routes because bus routes would be subject to accidental and inappropriate modifications. Further, if a bus stop is added to a route, then what results is a new and different route. It should not be the same object.

Many methods must be overridden to disallow in-place modifications. For example, the add: method is public and should be overridden to prevent changes.

BusRoute subclass of OrderedCollection
instance methods

add: aBusStop
"Override inherited public method to produce an error.
Bus stops cannot be added to a route."

^self error: 'Bus routes cannot be modified.'

In the second alternative, in which BusRoute is a subclass of Object, we don't need extra methods to override inappropriate behavior.

Create New BusRoutes

A more appropriate way to compose bus routes disallows in-place modifications. We need to make an instance creation method that creates a an initialized bus route. To support the instance creation method, a private instance method is needed to set the collection of bus stops.

BusRoute subclass of Object
class methods

withAll: collectionOfBusStops
"Create a new instance of the receiver initialized
from <collectionOfBusStops>."
^self new busStops: collectionOfBusStops

i*nstance methods*

busStops: collectionOfBusStops
"Private - Set the collection of bus stops."
busStops := collectionOfBusStops

Classes that collaborate with BusRoute need to access the bus stops to select stops based on some criteria. The class BusRoute needs to provide access to the bus stops and yet protect the private collection of bus stops from modification. The instance method busStops returns a copy of the collection referenced by the instance variable. This way collaborators can modify the returned collection of bus stops without any side effects on the bus route.

BusRoute subclass of Object
instance methods

busStops
"Return a copy of the collection of bus stops."
^busStops copy

New bus routes can be created using these methods. The following code illustrates the creation of a new route based on the bus stops from another route:

```
shoppingStops := downtownRoute busStops.
shoppingStops  removeFirst.
derivedRoute := BusRoute withAll: shoppingStops
```

Reuse Impacts

One of the benefits of the behavioral inheritance alternative is that it easy to change the collection characteristics. It is easier to modify the initialization code that allocates an object for an instance variable than to rearrange the hierarchy in order to get different collection characteristics. If you are forced to rearrange the hierarchy, then collaborators of BusRoute must change also. This is because different collection classes respond to different messages, and the inherited messages are directly accessed by the collaborators of BusRoute.

Also, new subclasses of BusRoute can be created based on collection characteristics. One subclass could support set semantics, in which no duplicates are allowed. Another could support sorted collection semantics. Each subclass can be implemented by simply overriding the initialization method. This is much more awkward with implementation inheritance.

In the behavioral inheritance alternative, the exact collaboration between BusRoute and OrderedCollection is clear because messages are relayed to the instance of OrderedCollection from BusRoute. In the implementation inheritance alternative, there is no collaboration. An examination of BusRoute does not determine which methods from OrderedCollection are used by BusRoute. Instead, collaborators of BusRoute must be examined. Furthermore, because not all inherited messages are appropriate, other developers will not know which messages they can send to BusRoute. This makes it much more difficult to extend and maintain the application containing BusRoute, and to reuse BusRoute in related applications.

Summary

Use behavioral inheritance whenever possible, because the resulting subclasses will be more reusable and easier to maintain. In locating subclasses in a hierarchy, use the is-kind-of criteria and similar behaviors to guide your selection. Only after locating a subclass based on behavior should you examine implementation details.

It is okay to change the superclass. After some implementation and testing, it is quite common to revisit class placement in the hierarchy. Re-examining placement can occur in conjunction with reorganizing the entire hierarchy and with the addition of new classes.