

Choice of Representation when Implementing a Data Abstraction

6.170 Staff

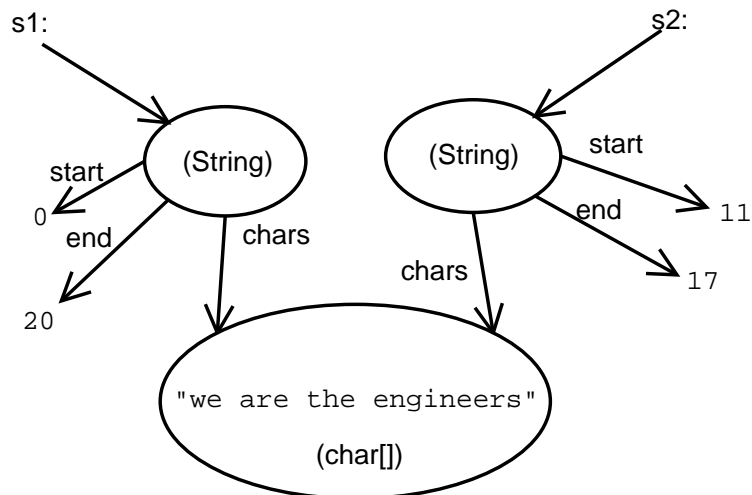
September 29, 2000

Many answers for PS1 took a very simple approach to implementing the specifications for the classes given:

1. look at each abstract field in the specification
2. find a simple way to map that abstract field to a class (for example, represent a sequence with a Vector)
3. put a concrete field in the implementation for that abstract field.

This can be a perfectly acceptable way of writing an implementation to satisfy a specification. However, you should know that it does not always result in the best performing implementations, and for some applications, the performance of the simplest implementation may not be acceptable.

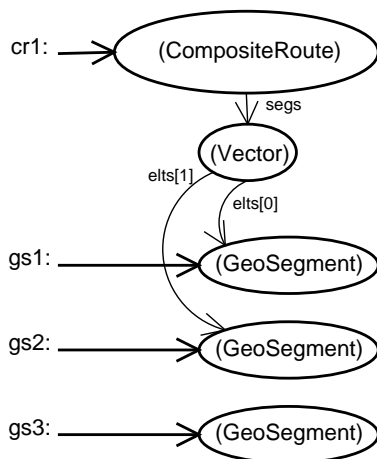
Immutable data types in particular give the implementor a lot of freedom in choice of representation, because it is often very efficient for instances of an immutable class to share state with other instances of the same class. For example, an instance of a String could share the same char array with the Strings produced by its substring method, as long as all of the String instances kept track of where in the char array they started and ended. This is much more efficient than making copies of the char array for every newly produced String. So, one example object diagram for a system that is implemented like this looks like so:



`s1` represents the string “we are the engineers”, while `s2` represents the string “engine”, but they share the same character array. If String was a mutable type, this sort of sharing would be impossible without destroying our ability to reason about each of the objects locally, because changes to `s2` would affect the state of `s1`.

Some might consider this an unimportant hack, but optimizations like this can make massive differences in the scalability of your code, as the following discussion will show.

You had such sharing of state as an option in your choice of representation for `CompositeRoute`, though you may not have seen it. `CompositeRoute`, in essence, is a sequence of `GeoSegments`.¹ Thus, many students chose a representation of a `Vector` of `GeoSegments` which resulted in object diagrams such as the following:

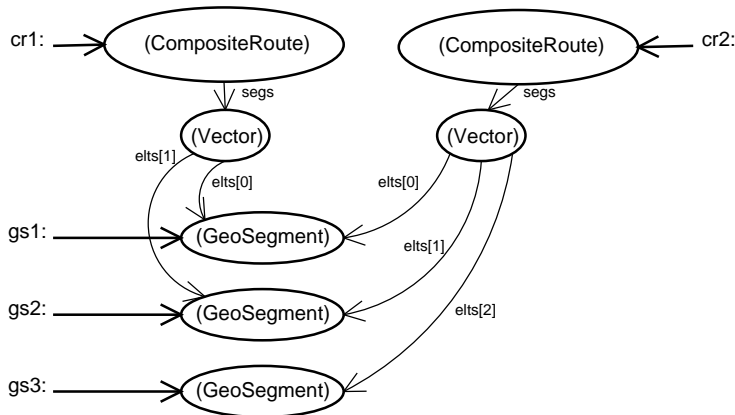


In this diagram, `cr1` is a `CompositeRoute` made up of the `Route` starting at the beginning of `gs1` and continuing along `gs2`. In particular, `gs3` is **not** a part of `cr1`; it is merely a lone `GeoSegment` which we can refer to by the name `gs3`. Now, let us assume that `gs3` is an acceptable `GeoSegment` to add to the end of `cr1` (that is, `gs3.p1 = cr1.end`). With that assumption, we are allowed to execute the code

```
CompositeRoute cr2 = cr1.extend(gs3);
```

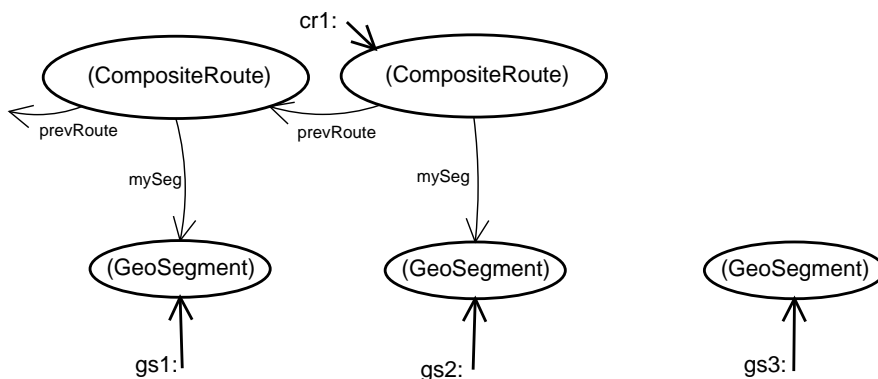
With this representation, executing this statement results in the following object diagram:

¹Some instead view `CompositeRoute` as a sequence of `ElementaryRoute`, in order to allow a different optimization in the implementation. That is also a valid view. However, the following discussion still applies in that case; it just complicates the implementation of some of the operations slightly.



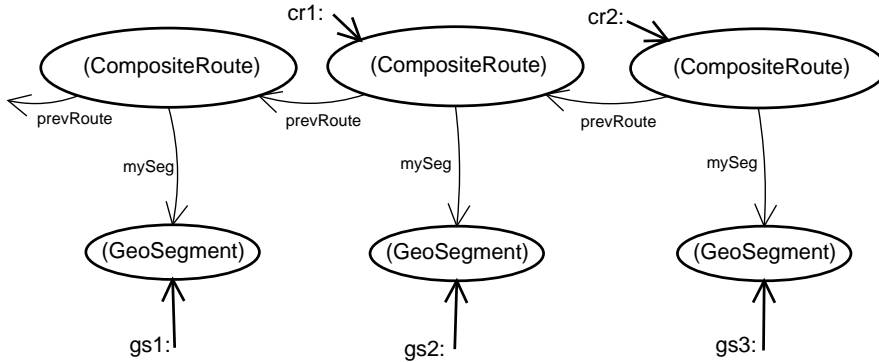
Thus, we have made a copy of the Vector `cr1.segs` and added a reference to `gs3` to the end of the copy, and set `cr2.segs` to refer to the new Vector. This process should be familiar to many of you, since it is exactly how many of you implemented CompositeRoute's `addSegment` method.

However, consider the alternative representation:



In this representation, each CompositeRoute produced from CompositeRoute's `addSegment` method has a reference, `prevRoute`, to the CompositeRoute that it was extended from and a reference, `mySeg`, to the GeoSegment it adds to the end of the previous route. The CompositeRoute constructed from a single GeoSegment refers to the GeoSegment it was constructed with through `mySeg`, and since it has no previous route, `prevRoute` points to null (notated here with an arrow missing a target circle). This is essentially a Linked-List of CompositeRoutes, where each cell in the list maintains a GeoSegment and a reference to the previous cell.

Thus, after we execute the same line of code,
`CompositeRoute cr2 = cr1.extend(gs3);`
 we get the following object diagram:



This representation supports all of the operations given by the specifications for ps1. So what's the difference between them? For small sets of GeoSegments, the difference between the two representations is negligible. However, as the data set scales up, this linked-list-rep is more efficient than the vector-rep given earlier for two reasons.

1. The linked-list-rep occupies less space when many CompositeRoutes produced by CompositeRoute's addSegment method are all kept alive at once. If you do not keep references to the intermediate Routes, then Java's garbage collector is free to throw away the Vectors being used by the intermediate routes. But if you do keep those Routes alive, such as by storing them in a priority queue while searching a graph, then the Vectors that the vector-rep uses internally will not be able to be garbage collected until the search is complete.
2. The execution of addSegment is much quicker in the linked-list-rep. In the vector-rep, the entire Vector must be copied; this operation takes time proportional to the length of the route. However, in the linked-list-rep, the CompositeRoute only needs to set up its two fields; thus it runs in constant time, independently of the length of the route that it was extended from.

Thus, it is easy to see that the choice of representation is not always an easy thing. We stress the point that the abstract fields given in a specification need not always map directly to the concrete fields in the implementation.