```
class REPETITION create
      make
feature
      base: FOOD
      quantity: REAL
      make (b: FOOD; q: REAL)
        -- Produce this food element from quantity units of base.
           ensure
                base = b
                quantity = q
           end
  ... Other features ...
end
```

This makes it possible to obtain an object of this type through `create apple_salad.make (6.0, apple)`, equivalent to an expression using the combinator. It is possible, as mentioned, to bring the notation closer to combinators by using factory methods.

Using Software Contracts and Genericity

Since we are concentrating on design, the effect of `make` has been expressed in the form of a postcondition, but it really would not be a problem to include the implementation clause (`do base := b; quantity := q`). It is one of the consequences of well-understood OO design to abate the distance between implementation and design (and specification). In all this we are freely using state-changing assignment instructions and still have (we thank the reader for inquiring) most of our teeth and hair.

Unlike the combinator, however, the class is not limited to these features. For example, it may have other creation procedures. One can usually mix two repetitions of the same thing:

```
make (r1, r2: REPETITION)
      -- Produce this food element by combining r1 and r2.
     require
          r1.base = r2.base
     ensure
          base = r1.base
          quantity = q
```

The precondition expresses that the quantities being mixed are from the same basic food types. This requirement can also be made static through the type system; genericity (also available in typed functional languages, under the curious if impressive-sounding name of "parametric polymorphism") leads to defining the class as:

```
class REPETITION  [FOOD] create
       ... As before ...
feature
      make (r1, r2: REPETITION [FOOD])
        ... No precondition necessary here ...
        ... The rest as before ...
end
```