property of the approach, which does not require it, but it is pervasive in these and many other publications.

Still, notation conciseness is a virtue even at the design and architecture level, and functional programming languages may have some lessons here for other design notations.

The second advantage (emphasized by Simon Peyton Jones and Diomidis Spinellis in comments on an earlier version of this chapter), also involving notation, is the elegance of combinator expressions for defining objects. In an imperative object-oriented language, the equivalent of a combinator expression, such as:

```
on_top_of topping main_part
```

would be a creation instruction:

```
create pudding .make_top (topping, main_part)
```

with a creation procedure (constructor) `make_top` that initializes attributes `base` and `top` from the given arguments. The combinator form is descriptive rather than imperative. In practice, however, it is easy and indeed common to use a variant of the combinator form in object-oriented programming, using "factory methods" rather than explicit creation instructions.

The other two advantages are of a more fundamental nature. One is the ability to manipulate operations as "first-order citizens"—the conventional phrase, although we can simply say "as objects of the program" or just "as data." Lisp first showed that this could be done effectively; a number of mainstream languages offered a way to pass routines as arguments to other routines, but this was not considered a fundamental design technique, and was in fact sometimes viewed with suspicion as reminiscent of self-modifying code with all the associated uncertainties. Modern functional languages showed the benefit of accepting higher-order functionals as regular program objects, and developed the associated type systems. This is the part of functional programming that has had the most direct effect on the development of mainstream approaches to programming; as will be seen below, the notion of `agent`, directly derived from these functional programming concepts, is a welcome addition to the original object-oriented framework.

The fourth significant attraction of functional programming is lazy evaluation: the ability, in some functional languages such as Haskell, to describe a computation that is potentially infinite, with the understanding that any concrete execution of that computation will be finite. The earlier definition of `within` assumes laziness; this is even more clear in the definition of `repeat`:

```
repeat f a  = [a : repeat f (f a)]
```

which produces (in ordinary function application notation) the infinite sequence `a`, `f (a)`, `f (f (a))`.... With `next N x` defined as `(x + N / x) / 2`, the definition of `within` as used by `sqrt` will stop evaluating that sequence after a finite number of elements.

This is an elegant idea. Its general application in software design calls for two observations.