Reusability

Is the solution general, or better yet, can we turn it into a *component* to be plugged in directly, off-the-shelf, into a new application?

The success of object technology has largely followed from the marked improvements it brings—if applied properly as a method, not just through the use of an object-oriented programming language—to the reliability, extendibility, and reusability of the resulting programs.

The *functional programming* approach predates object-oriented thinking, going back to the Lisp language available for almost 50 years. To those fortunate enough to have learned it early, functional programming will always remain like the memory of a first kiss: sweet, and the foretaste of even better experiences. Functional programming has made a comeback in recent years, with the introduction of new languages such as Scheme, Haskell, OCaml and F#, sophisticated type systems, and advanced language mechanisms such as monads. Functional programming even seems at times to be presented as an improvement over object-oriented techniques. The present discussion compares the two approaches, using the cited software architecture criteria. It finds that the relationship is the other way around: object-oriented architecture, particularly if enriched with recent developments such as *agents* in Eiffel terminology ("closures" or "delegates" in other languages), subsumes functional programming, retaining its architectural advantages while correcting its limitations.

To qualify this finding, it is important to note both the study's limitations and arguments to mitigate some of them. The limitations include:

Few data points

The analysis is primarily based on two examples of functional design. This could cast doubts on the generality of the lessons drawn.

Lack of detail

The source of the examples consists of an article (Peyton Jones et al. 2000) and a PowerPoint presentation (Eber et al. 2001)—referred to from now on as "the article" and "the presentation"—complemented in the section "Assessing the Modularity of Functional Solutions," later in this chapter, by ideas from a classic functional programming paper (Hughes 1989). Uses of the presentation may miss some details and nuances that would be present in a more discursive document.

Specific focus

We only consider the issue of modularity. The case for functional programming also relies on other criteria, such as the elegance of a declarative approach.

Experimenter bias

The author of the present chapter is a long-time contributor to and exponent of object technology.