**List of Potential Silver Bullets*:***

1. **Ada and other high-level language advances.**

Ada is a general-purpose, high-level language of the 1980s. Ada philosophy is more of na advance than the Ada language, the philosophy of modularization, of abstract data types, of hierarchical structuring. Ada will not be the silver bullet that slays the software productivity because it is just another high-level language. Ada’s greatest contribution will be that switching to it occasioned training programmers in modern software design techniques.

1. **Object-oriented programming.**

For both abstract types and hierarchical types, the result is to remove a higher-order sort of accidental difficulty and allow a higher-order expression of design. An order-of-magnitude gain can be made by object-oriented programming only if the unnecessary underbrush of type specification remaining today in our programming language is itself responsible for nine-tenths of the work involved in designing a program product.

1. **Artificial intelligence.**

The techniques used for speech recognition seem to have little in common with those used for image recognition, and both are different from those used in expert systems. I have a hard time seeing how image recognition, for example, will make any appreciable difference in programming practice.

1. **Expert systems.**

I believe the most important advance offered by the technology is the separation of the application complexity from the program itself. Suggesting interface rules, advising on testing strategies, remembering but-type frequencies, offering optimization hints, etc. Even more difficult and important is the twofold task of knowledge acquisition: finding articulate, self-analytical experts who know why they do things; and developing efficient techniques for extracting what they know and distilling it into rule bases.

1. **“Automatic” programming.**

It is the generation of a program for solving a problem from a statement of the problem specifications. The system assessed the parameters, chose from a library of methods of solution, and generated the programs. It is hard to imagine how this breakthrough in generalization could conceivably occur.

1. **Graphical programming.**

In the pitiful, multipage, connection-boxed form to which the flow chart has today been elaborated, it has proved to be essentially useless as a design-tool programmers draw flow charts after, not before, writing the programs they describe. Also, the screens of today are too small, in pixels, to show both the scope and the resolution of any serious detailed software diagram. Whether we diagram control flow, variable scope nesting, variable cross-references, data blow, hierarchical data structures, or whatever, we feel only one dimension of the intricately interlocked software elephant.

1. **Program verification.**

Verifications are so much work that only a few substantial programs have ever been verified. Program verification does not mean error-proof programs. There is no magic here, either. Mathematical proofs also can be faulty. So whereas verification might reduce the program-testing load, it cannot eliminate it.

1. **Environments and tools.**

Big-payoff problems were the first attacked, and have been solved: hierarchical file systems, uniform file formats so as to have uniform program interfaces, and generalized tools. Language-specific smart editors are developments not yet widely used in practice,

but the most they promise is freedom from syntactic errors and simple semantic errors.

1. **Workstations.**

The composition and editing of programs and documents is fully supported by today’s speeds. Compiling could stand a boost, but a factor of 10 in machine speed would surely leave think-time the dominant activity in the programmer’s day.