**Promising attacks on the conceptual Essence**

All of the technological attacks on the accidents of the software process are fundamentally limited by the productivity equation:

Time of task = ∑ (Frequency) i x (Time) i

The conceptual components of the task are now taking most of the time.

**1- Buy versus build:** The most radical possible solution for constructing software is not to construct it at all, attending the mass market. More specialized sources offer very powerful products for the workstation and other Unix markets. Even software tolls and environments can be bought off-the-shelf. Such product not only is cheaper to buy than to build afresh but also tend to be much better documented and somewhat better maintained than homegrown software. The cost of software has always been development cost. There some ways to reduce it, such as sharing that cost among a few users (just this can radically cut the per-user cost). Another way of looking at it is that the use of n copies of a software system effectively multiplies the productivity of its developers by n. That is an enhancement of the productivity of the discipline and of the nation. The key issue, of course, is applicability. The big change has been in the hardware/software cost ratio. There are dramatic exceptions to my argument that the generalization of the software packages has changed little over the years: electronic spreadsheets and simple database systems.

**2- Requirements refinement and rapid prototyping:** The hardest single part of building a software system is deciding precisely what to build, therefore the most important function that software builders do for their clients is the iterative extraction and refinement of the product requirements. This problem exists, specially because the clients do not know what they want making it hard to know what questions must be answered. So in planning any software activity, it is necessary to allow for an extensive iteration between the client and the designer as part of the system definition.

**3-Incremental development−grow, not build, software:** The secret is that it is grown, not built. Software systems should be grown by incremental development. The approach necessitates top-down design, for it is a top-down growing of the software. It allows easy backtracking.

**4-Great designers** The central question of how to improve the software art centers, as it always, on people. Great designs come from great designers and software construction is a creative process. A little retrospection shows that although many fine, useful software systems have been designed by committees and built by multipart projects, those software systems that have excited passionate fans are those that are the products of one or a few designing minds, great designers. Consider Unix, APL, Pascal, Modula, the Smalltalk interface, even Fortran; and contrast with Cobol, PL/I, Algol, MVS/370, and MS-DOS I think the most important single effort we can mount is to develop ways to grow great designers. My first proposal is that each software organization must determine and proclaim that great designers are as important to its success as great managers are, and that they can be expected to be similarly nurtured and rewarded.

**How to grow great designers?**

Systematically identify top designers as early as possible. The best are often not the most experienced. Assign a career mentor to be responsible for the development of the prospect, and keep a careful career file. Devise and maintain a career development plan for each prospect, including carefully selected apprenticeships with top designers, episodes of advanced formal education, and short courses, all interspersed with solo design and technical leadership assignments. Provide opportunities for growing designers to interact with and stimulate each other.