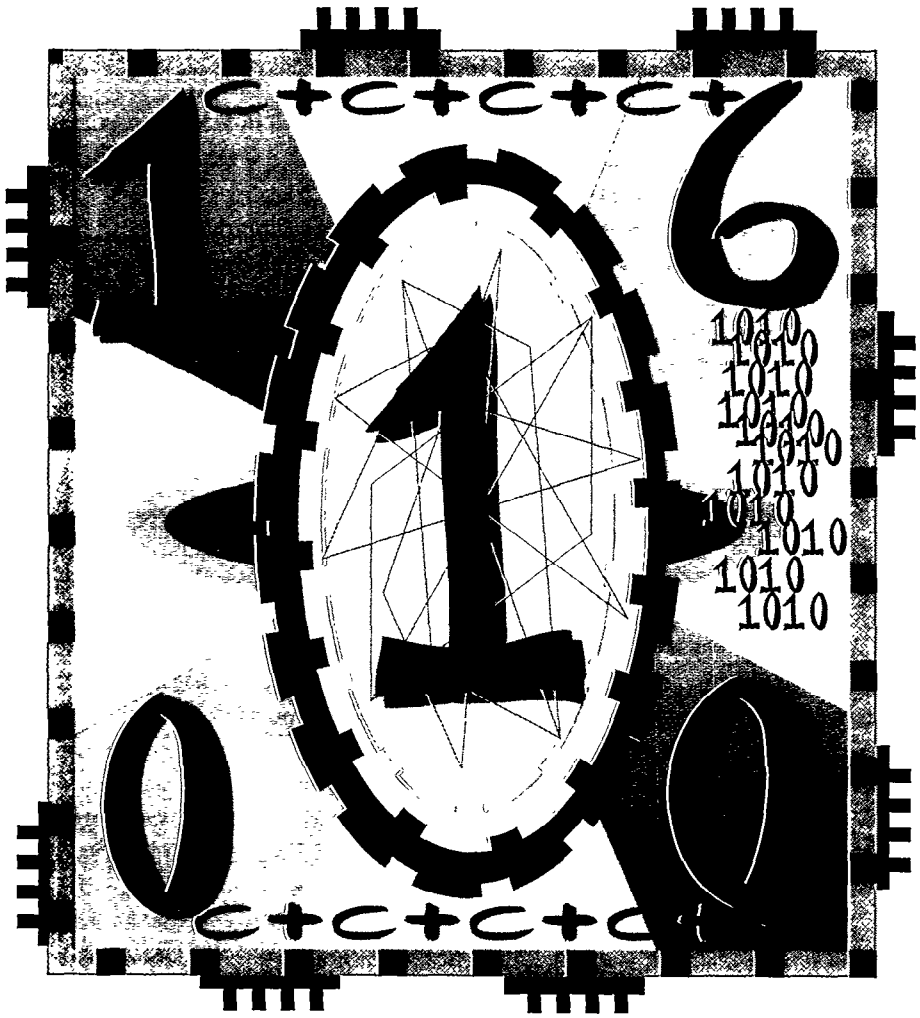
HAL BERGHEL

New Wave Prototyping: Use and Abuse of Vacuous Prototypes



Pamela Hobbs

In the inaugural issue of interactions, Rudd and Isensee [6] offer suggestions for happier, healthier prototypes. While many of these suggestions are useful, we take issue with the author's willingness to unquestioningly embrace the current trend toward "vacuous" prototyping. The increased ease with which modern interfaces can be created does not come without risk because it strains our reliance on prototypes as springboards for important new technology. While this is not world-shaking, for we can all become accustomed to looking at rapid prototypes in this new way, it will continue to cause discomfort. Vacuous prototypes are for the 1990's what "vaporware" was for the 1970's and early 1980's.

Prototyping has long been the neglected offspring of software engineering. It is largely an activity without rules or structure. Although software prototyping is as old as computing there is little orthodoxy concerning its use. This is in part a result of the disorganized and variegated state of its literature. It is more likely that an article on prototyping will appear as a filler chapter of a trade book on another subject, a privately circulated report, or in a "how-to" section of a trade magazine than in a refereed journal or conference proceedings.

Consider the following data. In their recent survey of the literature, Jenkins and Kennedy [4] identified 217 articles on software prototyping. Of these, the majority (119) are from such non-refereed sources as industry reports and working papers (48), magazine articles (55) and un-refereed newsletters and bulletins (16). Further, the subject areas spanned (in decreasing frequency of occurrence) such diverse topics as prototyping support tools, case studies, prototyping methodology, management issues, and prototype use, to name only the most frequent. What is more, the overwhelming majority of citations from all sources are non-critical reports of experiences rather than theoretical or philosophical discussions. Opinions abound. Carefully articulated methodology is hard to find.

However, the lack of orthodoxy is even more due to the nature of prototyping itself. Prototyping is inherently experimental and is driven by the needs of the practitioners who routinely work with tight software development schedules and inflexible deadlines. Front-line applications areas like office automation, language translator development and information retrieval have their own well-defined body of foundational literature to rely on. There exist more-or-less standard publication venues, special interest groups, theme conferences, frequent birds-of-a-feather workshops, and an identifiable subculture of professionals who specialize in just those applications. These complement this foundational literature in defining in at least a semi-formal way the accepted standards and practices of the discipline. With prototyping the de facto standards and practices tend to be more closely associated with projects and development environments then with foundational resources.

This is what is wrong with the practice of software prototyping. What is right with it is that it works well for the most part and serves a very important function in the software development process. This apparent incongruity is a result of the fact that prototyping, as an experimental activity, suffers from the same problem as experimental computer science and engineering generally. Here, practice frequently leads theory. While in some sciences, experimentalists may confirm theoretical engineering, experimentalists often define the field as they go. This is a necessary by-product of the rapid evolution of the field and the increasing sophistication and complexity of its artifacts.

If this is not unusual in other areas of experimental computing, then what is the problem? Enter visual programming environments (Visual Basic, Visual C++, Objectvision, Powerbuilder, Access, etc.). These products allow a neophyte to create high-fidelity, though content-free, prototypes. This was not a problem in years past, for the skill-set required for developing front-end interfaces was the same skill-set required for the back-end application. Programming practice and experience tended to have an amorphous character to them and, as such, ported quite well to all aspects of program development. So if one could find a programmer/analyst who knew how to write the interfaces, one could find a programmer/analyst who was likely to know how to write the kernel routines. It's not that way anymore.

The problem is not with the visual programming environments. On the contrary, they represent a positive, even if incremental, step forward in applications development software. The problem lies within the practice of prototyping.

The problem arises in a round about way. It is now possible, with few technical skills, to create an interface at a higher conceptual level and with more sophistication than one can produce in the back-end application. One result has been the proliferation of hollow, or vacuous, prototypes. This is a downside to the increased ease of use of prototyping tools. This trend will continue and increasing numbers of vacuous prototypes will appear which purport to show the viability of some idea or other although the actual prototype has little or no explanatory and predictive power.

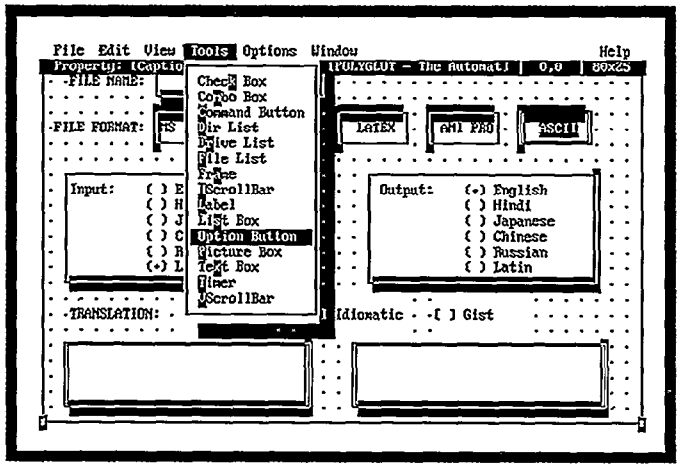
So that no misunderstanding results, let me state clearly that my remarks are NOT directed toward the fields of interface design and engineering. It is the misuse of interface design and interface engineering tools that we are concerned with. It is now possible to develop useful and interesting, hi-fi interfaces on top of smoke and mirrors. The modern venture-capital prototype is coming to resemble a spaghetti western: all theatrics and no substance. This can be a real problem if the user expects of the prototype that it is a platform upon which something important will be constructed.

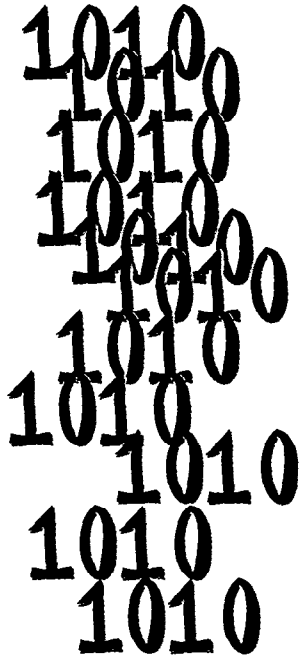
We are not opposed to placing more of the responsibility for program design and development in the hands of the end user either. Whether user-centered or the product of collaboration, the success of a prototype-in contrast with the final commercial grade application-requires a certain level of mutual understanding and trust between the user/viewer and developer. If this is not present, the credibility of the developer and confidence in the project is lost. In the "good old days" it was usually obvious from the behavior of the prototype what level of confidence was justified. Now it's getting harder to tell.

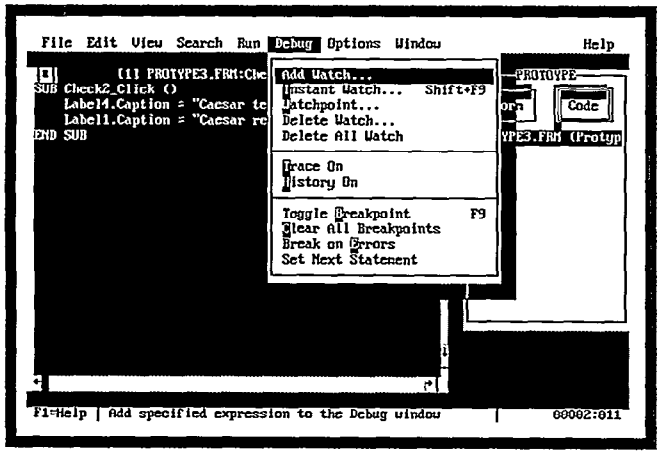
"SOFTWARE SURPRISE" esign

A vacuous prototype begins with some idea or other-not necessarily a good one. In this case, we prototype machine translation to and from English, Hindi, Japanese, Chinese, Russian and Latin. Development proceeds this way:

Step 1: Create Display. This GUI is generic DOS written in Visual Basic. ETH (elapsed time for hacker) = 30 minutes.

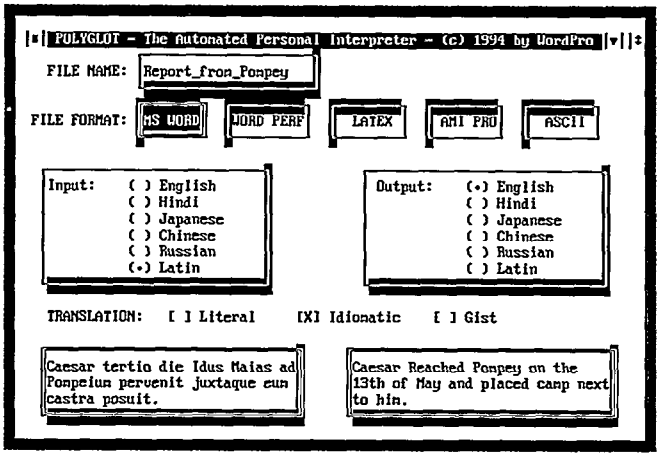






Step 2: Throw together a few lines of code for each screen object in a corresponding code window. These code fragments can be amateurish and yet still produce an effective demonstration. ETH = 1 hour.

Step 3: Compile, link and voila–"POLYGLOT: The Automated Personal Interpreter" appears up and running. Another useless, content-free prototype is unleashed upon the unsuspecting public. Good taste, if not professional ethics, dictates that "caveat emptor" appear on the sign-on screen. ETH = 15 minutes.



At first glance, one might think that this problem will go away as more energy is directed toward the support of end-user programming. We doubt that this be the case because end-user programming support tools (e.g., application or scripting languages and macro recorders) will tend to require greater levels of technical prowess than visual prototyping tools. This is because they will only give the appearance of utility if the ideas that they embody are correct. A vacuous prototype, on the other hand, can give the appearance of importance even if the underlying problem it represents is intractable, for no sound understanding of algorithms and data structures is required.

The new field of "programming- by-demonstration" [2] is also immune to this criticism. PBD seems a very useful extension to conventional programming. The underlying rationale is that if an end user can figure out how to do something once, the computer ought to be able to figure out how to do it after that. However, this assumes that the user can figure out how to do it the first time. It's one thing to develop software that can infer end users' intentions from their interactive behavior, it is quite another to develop software that will infer an algorithm from a nicely structured interface. There is hope for the former. We're not sure that it even makes sense to look for the latter.

In this regard, we should also say something about Visual Programming Languages (e.g., Prograph) where the focus is on visual programming of the entire application and not just the interfaces. This emphasis sets these languages apart from visual programming environments. It also makes them less suitable for vacuous prototyping! They are so different that they should be treated separately.

There is a natural hierarchy in the evolution of modern applications development tools used for prototyping. Although a bit over-simplified, one may characterize this evolution in three stages. First came the translator-cum-libraries milieu in which the developer complemented the high-level code with code from run-time libraries for the time-intensive, though mundane, support routines. Next, more sophisticated third-party libraries evolved. Robust, specialized libraries for windowing, file management and database, memory management, communications and the like became inexpen-

Origins of the Vacuous Prototyping Problem

A Response to Hal Berghel

James Rudd and Scott Isensee

We agree with Berghel's concern about vacuous prototypes. Customer wants and needs embodied in a user-interface prototype that cannot be realized in the product's implementation often negate the benefit of prototyping. We also agree with Berghel that prototyping must be done in the framework of a well-defined, methodical prototyping process.

We disagree, however, that improved tools are the cause of the vacuous prototyping problem. Vacuous prototypes were around long before the latest generation of prototyping tools. They may be the offspring of user interface designers who understand customer requirements, but are not knowledgeable about implementation. Just as frequently, they are the offspring of experienced programmers who know implementation quite well but are not particularly skilled at collecting user-interface requirements of designing usable user interfaces. This was a common problem before user interface design became an established profession and prototypes were nearly always done by programmers.

The software industry clamors for designers who can competently walk both sides of the fence. Educational programs are usually compartmentalized into specialties like Psychology and Computer Science. There are few programs that provide an adequate combination of both. To make matters worse, the industry is pervaded with user-interface designers whose entire user-interface design training consists of attendance at a couple of satellite broadcasts from National Technological University, the perusal of the latest design guidelines for Windows or Motif, and years of experience developing marginally usable UIs.

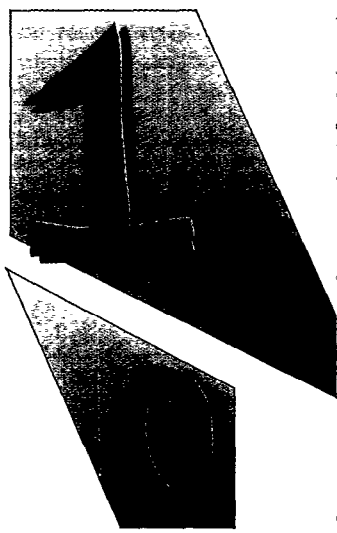
We have led user-interface prototyping and development efforts both at IBM and as user-interface consultants for other companies. Our experience tells us that user-interface prototyping is most successful when it is conducted by skilled user-interface designers as an integrated part of of a well-planned

sive and standard fare in the developers' toolkit. It was sophisticated third-party libraries that set the stage for the next stage in the evolution-visual programming environments.

Visual programming environments support modern graphical and visual techniques in the development of the interface, while requiring that the core of the program be created through "code windows" in conventional ways. In other words, what is visual about them is the development of the interfaces. While this is important from a practical point of view, it is not that interesting from a theoretical point of view.

The earlier run-time library approach could in principle accomplish much the same thing as the visual programming environment. The difference is that the run-time approach was procedural and not functional or structural. The fact that the visual programming environment approach might be done in an object-oriented way doesn't change the fact that it is still little more than a GUI-development tool, just as its run-time library ancestor. In both cases, the backplane code is still conventional and text-based.

Visual programming languages on the other hand use visual techniques for the programming itself. This is a significant difference, and one that represents an entirely new programming and software development paradigm. Here programming, and not just interface design, is raised to the conceptual level. There are no code windows to program. The kernel code is as visual as the interface. Visual programming languages compare to visual programming environments much as visual animations of algorithms stand to pseudo code. Unlike visual programming environments, visual programming languages represent a quantum leap forward in programming technology, not merely an incremental extension. In this sense it is more akin to programming by demonstration than visual programming environments. It is ironic that visual programming languages tend to require a higher skill level for vacuous prototyping than the simpler visual programming environments. This has the effect of discouraging their use for that purpose. We see this as a positive feature.



This is the background against which we placed Rudd and Isensee's remarks like "Add as many features to the prototype as you can.

development process. This process keeps the prototype on track and helps to compensate for the imperfect knowledge and skills of many prototypers. From the beginning of the prototyping process, Laura Lead Architect, for example, should work with customers in identifying and defining customer wants and needs and translating them into user-interface requirements through the prototyping effort. Conversely, Yolanda UserRep should consult with the architecture and programming team on a regular basis to ensure that user-interface requirements as embodied in the prototype are indeed implementable by the development team. Our forthcoming book, "The Art of Rapid Prototyping", from Van Nostrand Reinhold describes a detailed process for successful rapid user-interface prototyping and relates some of the pitfalls encountered when the designer/developer fails to adhere to such a process. Berghel's seamless prototyping proposal to design prototypes that reflect only what will be in the final application defeats one of the major purposes of prototyping-to determine what could and should be in the product. If the product is already defined, we could skip the prototype and start coding immediately. It is rare indeed, that we have such perfect knowledge in a software development project. Prototyping is by its very nature experimental and iterative. It is a way of setting requirements rather than just reflecting them.

This leads us to prototyping tip number 23: "Use prototyping to collect rather than just reflect requirements." Be sure to update your official prototyping wallet card. Until next time, Happy Prototyping!

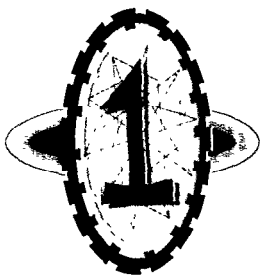
Scott Isensee
isensee@austin.ibm.com

James Rudd
jimrudd@vnet.ibm.com

Even though you know some may not be feasible for the release." Viewed from the perspective that we have just provided, this might be considered alarming rhetoric. We were not assuaged by the observation that "...successful prototyping effort requires more that a prototyping tool and a background in user interface design..."-an understatement.

Perhaps we expect too much of prototypes. Perhaps the denotation "prototype" should not require a non-coincidental functional and operational resemblance to a final product. Many of us still endorse the "minimal guarantee" approach to prototyping where the user/consumer is entitled to expect that a prototype will contribute something significant to "...reliable, economical, efficient software systems that meet their specifications..." [3]. On this view, the prototype is a real snapshot-in-time along a product's development path and not an abstract representation of what might be.

Hal Berghel is Professor of Computer Science at the University of Arkansas and past Director of the Center for Artificial Intelligence and Expert Systems (CAIES). email: hlb@acm.org



In terms of showing the viability or effectiveness of a complete program, vacuous prototypes offer little more than basic presentation software (e.g., Persuasion, Authorware). They actually compare unfavorably when such software is enhanced with multimedia support. So what is the real value of the look-and-feel without some level of assurance that the intended back-end application is structurally sound?

What we oppose is the unquestioned endorsement of vacuous prototyping as a universal software development strategy. Whether one refers to it as rapid, quick-and-dirty, quick-and-clean, or vacuous, it is a technique whose importance to the total software development exercise is proportional to the level of expertise of its consumers and the integrity and ability of the developers. In those cases when the consumer is looking to the prototype for proof-of-concept or proof-of-performance, it can be either uninformative at best or misleading at worst.

As an alternative strategy, we recommend consideration of more robust prototyping methods. While they have longer gestation periods, they are likely to be more "seamless" with the end products. While they are more expensive, they are more functional and can be relied on more heavily. In many, if not most, situations the consumers and end-users can be conditioned to accept this strategy (venture capital solicitation aside).

Seamless prototypes proceed from a philosophy that unlike back-end applications programming, the front-end programming is important IN software development but not central TO software development. The underlying premise of this philosophy is that whatever can be done with direct manipulation interfaces can be done (perhaps with greater difficulty and less finesse) with non-GUI interfaces augmented with keyboard overloading for interrupts, windowing control, and so forth. As a consequence, the overriding concern of the seamless prototyper is that the range of functionality of the prototype as closely resembles that of the final product as the situation allows.

We view seamless prototyping [1] as an extension of progressive prototyping [5] with the difference that seamless prototypes invest more effort in the front end of the development cycle toward the end of maximizing functional completeness, rigor and durability. Seamless prototyping is no less immune to the problems caused by fluid specifications, incongruities, false starts, and administrative distractions and interrupts than any other model. While not appropriate in all situations, seamless prototyping is especially amenable to those situations where software development is basically linear and where feedback loops do not proliferate beyond necessity. ■

References:

[1] Berghel, H., On Seamless Prototyping, *ACM SIG ICE Quarterly* [1994: in press].

[2] Cypher, A. *Watch What I Do: Programming by Demonstration.* M.I.T. Press, Cambridge, MA., 1993.

[3] Denning, P. and Dargan, P. A Discipline of Software Architecture, *interactions*, 1(1) (1994).

[4] Jenkins, A., and Kennedy, R.: An Annotated Bibliography on Prototyping. IRMIS Working Paper W811, Institute for Research on the Management of Information Systems, University of Indiana, 1991.

[5] King, D. *Current Practices in Software Development.* Yourdon Press, New York, 1984.

[6] Rudd, J. and Isensee, S. Twenty-Two Tips for a Happier, Healthier Prototype, *interactions* 1(1) (1994).