PROTOTYPING

A prototype is a working, though abbreviated, version of a system. Just as a product engineer might create a working model of a new toaster for testing before actually sending plans to the assembly line, so might an analyst create a prototype of computer software before the final version is produced. A prototype performs the same basic tasks that the finished system will perform but ignores such features as efficiency, calculations, security, error handling, and program and user documentation. It is, in effect, a skeletal version of the system, often "dummying" the internal processing of the system while concentrating on portraying user interfaces such as screens and reports. Since the value of a prototype lies in the fact that it is built quickly and cheaply, it is usually constructed in a matter of days using a fourth-generation language, an easy-to-use database management system, a data dictionary, and various other integrated CASE tools. Even if the final version is to be run on a mainframe, the analyst often creates and tests the prototype on a microcomputer workstation. For purposes of illustration, let's assume that we are building a system to allow students to register for classes using data entry screens set up around campus. The first part that we would probably prototype would be the data entry screens that the students would use. The screens would be a mere facade for the system, since nothing "behind" the screens would actually work. However, to students, the screens would appear to be real. We would then put students at the screens and tell them to use the screens as if they were actually registering for classes. We would watch the students and ask them what they liked and did not like about the screens. Through this, we would be able to discover that perhaps we omitted important directions, or chose to put the data entry fields in a confusing order, or left off important fields completely. Each time we found problems, we would modify the prototype, then test it on the users again.

After prototyping and modifying the student data entry screens, we might move on to the screens and reports used by the employees in the department responsible for handling student registration. Once again, we would build the facade for the system, then let the employees try it out. This entire process would continue until we had prototyped every important screen and report for our system. Then, and only then, would we worry about building the processing that goes on behind the facade of screens and reports.

It is particularly appropriate to build prototypes for online systems because of their strong user interfaces. These interfaces require a multitude of input and output screens as well as reports on paper, all of which can easily be modeled using prototyping. On the other hand, a calculation-intensive system, one that involves only minimal input and output but complex algorithms for processing, is not a good candidate for prototyping. Building a prototype during the analysis and design phases can clarify the user's requirements and verify that the finished system will meet those requirements. After all, consumers do not buy cars on the basis of the advertisements they read; why should computer users be expected to accept a computer system design that exists only on paper? As Schrage says, "The real value of rapid prototyping, for example, isn't that we can quickly respond to user requests, it's that users can now explicitly see just what they've asked...The dirty secret of systems development is that the overwhelming majority of users don't really know what they want, or worse, have grossly unrealistic expectations." All this is not to say that paper models are useless, since the better part of this book has been concerned with creating them. Rather, it implies that the process of analysis and design calls for the application of all available tools, whether paper or prototype.

Prototyping is user-centered; the user and the analyst work closely together as the prototype evolves through a process of iteration, or incremental refinements. Using this approach of phased development, functions are added one by one as a deeper understanding of the system emerges, until the prototype is considered to be complete by both the user and the analyst. This process allows the users to "test-drive" the system; and as they do so, they can locate inadequacies and suggest enhancements that previously

were not imaginable. Just as a person who has never driven a car is unlikely to think about the need for cruise control, so a person unfamiliar with computers is not likely to be able to prespecify every need without first having some hands-on experience with a system. The prototype is a dynamic vehicle for providing that experience. In addition, being able to use the prototype helps to keep the users enthusiastic throughout the often long, drawn-out system development life cycle.

Prototyping is also useful for coping with users who can't explain what they want but are sure they will know it when they see it. It is a particularly effective tool when building systems with a high level of uncertainty; that is, neither the users nor the developers are exactly sure what is required of the new system. Research has shown that systems with a high level of uncertainty during the development phases are often more expensive to maintain throughout the system development life cycle because of excessive errors and omissions. During the prototyping process, uncertainty can be reduced, thereby also reducing errors and omissions. In addition to clarifying requirements and reducing uncertainty, prototyping is likely to foster a more positive attitude on the part of the user. The user helped to build the system, so it is precisely what she or he needs to do the work more effectively. The user is bound to feel a certain amount of proprietary pride in the final result.

There are, however, several disadvantages to creating prototypes. Most notably, users must be heavily involved with creating and evaluating the prototype. Some users shun such responsibility and begrudge the investment of their time. Also, as mentioned earlier, prototyping is often inappropriate for systems that require complex algorithms for processing.

There can be three stages to the prototyping process. The first stage involves only the user interfaces such as screens and reports. At this point, any later processing such as calculations and data manipulation are simulated with program stubs. In the second stage, the omitted program functions are introduced, thereby replacing the program stubs with truly functional programs. In the third stage, the prototype is expanded to become the finished system. Features initially left out such as user documentation and help screens, security and backup procedures, and the ability to handle a large volume of transactions are added to the existing prototype to form the finished system.

Depending upon the situation, creation of the prototype may be complete after any one of these three stages. If the prototyping process ends after the first or second stage, the prototype is used only as a means of clarifying requirements and communicating with the users during the analysis and design phases. If the prototype is expanded to become the finished system, as it is in stage three, the system development life cycle is altered dramatically, since many of the normal activities of analysis, design, and construction are instead carried out throughout the ongoing process of refining the prototype. This approach is called rapid application development (RAD). RAD, systems are "grown," not built. That is, systems design and construction is an evolutionary refinement rather than a step-by-step process. With this approach, the finished version is inexpensive to develop, its flexibility ensures that lifetime maintenance will also be inexpensive, and it can be implemented in a fraction of the time required for a more conventional system.

Still, limitations of many fourth-generation languages make large information systems difficult to prototype to the third stage, these languages sometimes execute slowly, consume memory voraciously, and prevent other parts of the system from executing in a timely fashion. These problems become magnified as the size of the prototype is expanded by adding omitted functions and increasing the size of both the database and of transaction volume. Therefore it is a mistake to assume that the third stage, finished system will operate at the same processing speed as versions one and two. As fourth-generation

languages mature over the next few years, many of their inherent flaws should be corrected, making third stage prototyping more practical.

If the finished system is to be used only rarely or is very small, flaws such as poor response time might be tolerated. Otherwise, we should consider a prototype whose performance is unacceptable to be just a demonstration of the new system, and we might choose to recode all or part of it in a standard procedural, or third-generation, language.

Regardless of whether the prototype goes on to become the finished system or is later discarded, nothing is lost in its creation. We can construct a prototype cheaply and quickly, thereby providing an effective vehicle for purposes of testing and user evaluation. Also, the prototype itself serves as a design specification for the work to follow. According to Bernard Boar (1986), a leading expert on prototyping, "The creation of an accurate and vivid requirements document is sufficient justification for building models [prototypes] without the need to have implementation potential.

ADVANTAGES OF PROTOTYPING

- Heavy user involvement means a more complete, accurate, and friendly user interface.
- Prototyping may discover user needs that even the user was not previously aware of.
- Users feel more confident approving a system that they can try out than they do approving one that exists only on paper.
- Users have a more positive attitude toward any system that they have helped to create.
- Prototyping often results in a development project that costs less and takes less time to build, simply because prototyping is so effective in minimizing development requirements and scope changes.
- A system that has been prototyped during information gathering is cheaper to maintain because of reduced uncertainty; there are often fewer errors and omissions.
- A prototype that evolves into the finished system is often inexpensive to maintain because a system coded in a fourth-generation language is easier to modify than one coded in a third-generation language.

DISADVANTAGES OF PROTOTYPING

- Users may not want to invest the time and effort that prototyping requires of them.
- A prototype that evolves into the finished system may execute slowly, consume memory voraciously, and prevent other parts of the system from executing in a timely fashion.
- Often, the fourth-generation prototype must be recoded in a third-generation language in order to improve performance.
- Prototyping is often inappropriate for systems that require complex algorithms for processing.