Conference Paper:
HP 3000 International Users Group Meeting

7. Oct. 1981

On the Use of "Prototyping" in Software Development

C. Floyd

Institute for Applied Informatics
Technical University Berlin

The phrase "rapid prototyping" is currently en vogue in certain software engineering circles. The basic idea is to aid communication between software producers and software users (customers), in particular during the early stages of software development, by furnishing experimental versions of the system, to be tried out as part of requirement analysis.

In what follows I will attempt to demonstrate the role that a software-"prototype" might assume in different production settings in a manner compatible with the main line of software engineering's strive for a methodology, as illustrated for example in Prof. Turskis lecture at this meeting on October 9th (/**TURSKI** 81/). To begin with, however, some comments about the phrase "rapid prototyping" are in order, since this promises to be yet another unfortunate misnomer, which may well lead to serious misunderstandings, if ever this technique should be adopted by the software industry. A prototype is a well established concept in the engineering disciplines where it refers to the first functioning version of a new kind of product. In this context, a prototype is intended to exhibit all essential features of the final product and thus becomes the basis for experiments before the beginning of large scale production. This analogy does not carry over easily to software production, where we are not faced with mass production at all. Surely, if we use the concept of a prototype in software production - as I will do from now onwards, though under protest - we shall have to give it a new meaning appropriate for our purposes.

The second unfortunate term in the phrase is the epithet "rapid", which misleads us into believing that speed is the essential aspect in building a prototype. Again, this is in conflict with the engineering tradition, where the prototype is the final result of careful design, extensive calculations and field tests. I fail to see how a software prototype produced rapidly, without the careful preparations mentioned above will yield reliable answers in determining actual requirements.

In order to judge the usefulness of prototyping in software development we must find answers to the following questions:

- Why is communication about software requirements based as it is on interviews, checklists and bulky documents not sufficiently reliable and how could a prototype be helpful in this context?

- How does the software prototype relate to the final product?

- Is there one, or are there several prototypes and how are they evaluated?

- Under what circumstances can we justify the additional investment brought about by producing a prototype in the early stages, i.e. what do we hope to gain later on?

- How does prototyping relate to an orderly approach to software development, based on deriving a program from a rigorous specification according to the rules of programming methodology?

As a starting point in answering these questions we should take a close look at the well known phase-oriented approach

to software development, its merits and shortcomings (see for example /LEHMANN 80/). The phase-oriented approach was devised as a means to find contractual bases in software development and to define intermediate results in terms of documents, which form the basis for subsequent work. The phase-oriented approach relies on some important assumptions, as there are:

- that requirements, at least in principle, can be fixed in advance,

- that documents, provided that their contents are described in a sufficiently rigorous manner, are adequate as a primary means of communication, i.e. that the customer knows what he will get when he signs the contract.

Both of these assumptions unfortunately are contradicted in the daily practice of software professionals who are faced with the difficult task to base their own work on existing base-line documents, while at the same time coping with constantly changing requirements from their customers. The phase-oriented approach does of course permit to go back to earlier phases when needed, but it does not encourage the planing of profound revisions.

The phase-oriented approach provides a sound basis to limit the liability of the software producer. The product is defined by its specification and the liability of the software producer ends when he has derived a program, which is correct with respect to its specification. As Prof. Turski will point out in two days, this is a highly nontrivial activity which is well supported by modern software engineering techniques. Yet, experience shows, that even a correct program may not at all be adequate to fit the user's needs, because of far reaching misconceptions about the actual requirements.

The situation is aggravated by the fact that mistakes made early in software development are the most costly to correct. Serious mistakes in requirement analysis may well be too costly to correct at all. The user organization will have to adapt to the software - not vice versa.

There are important reasons why it may prove very hard to find out detailed software requirements for the development of large programs:

1) It is extremely difficult for people to visualize how seemingly minor decisions about software will later on affect their work with the system.

2) It is often extremely difficult to locate all groups of people who will be directly or indirectly affected by the system. Different user groups often have conflicting views about an information system (which they perceive from their own perspective), or they simply ignore each others needs.

The above mentioned difficulties do not pertain to all requirements alike, in fact the following classification of requirements helps to point out the areas where troubles most likely arise. We can distinguish:

- functional requirements describing the desired output to be produced for a given input (the relation between input and output may be highly nontrivial, but it is normally governed by a stringent set of rules; whether or not the program obeys the same set of rules can be proved - at least in principle).

- performance requirements stating the resources available to achieve these functions (it may be difficult to show the precise constraints on resources, whether or not the program meets these constraints can be measured - at least in principle).

- handling requirements characterizing the manner in which the system is to be embedded into the activities of all people affected by it.

Of these three, the handling requirements are the least well understood. Handling requirements pertain amongst others to the following areas of special concern:

- The design of man-machine interfaces in the widest sense (including conceptual models the user must have, in order to understand what the system does);

- The degree of system integration and as a consequence the possibility of interfering with or reshuffling the system's functions as needed ("conviviality" of the system according to Ivan Ilich /ILICH 79/);

- The interplay between formalized (i.e. computer-supported) and informalized work-steps permitted by the system (with the two extremes: the system enforces a working style akin to the assembly line or the system offers a tool-box to be used as needed).

This list does not claim to be complete. The examples are indicated in order to demonstrate that handling requirements will indeed lead to important decisions about software structure, that may well determine the adequacy or inadequacy of an otherwise correct program!

In the absence of a suitable theory of organizations and of sound user psychology, communication with the user, about software requirements, will continue to rely largely on experience and intuition. In this context, it is felt by many that communication is more reliable, if it is based on an already existing program which can be evaluated (albeit not systematically since there is no underlying theory how this might be done). A prototype therefore, is to be furnished in order to reduce the probability of misunderstanding requirements. The additional investment needed for its production is justified by the hope that this investment is significantly smaller than the costs that are likely to arise from the need to adapt an inadequate program later on.

It should be kept in mind, that the production of a prototype is justifiable only in the case of long-life systems, where a further expansion of the early phases will presumably lead to profits over a considerable period of time. Further, this technique is particularly relevant for programs which are embedded in technical or socio-technical environments, because such programs will have elaborate handling requirements associated with them.

How then, does a software prototype relate to the actual product in time, scope and quality? We can distinguish several feasible approaches here:

- the prototype may be intended to aid requirements analysis only or it may be intended to accompany the actual system throughout its lifetime.

- The prototype may be intended to cover essentially the same scope as the actual system or it may be intended to exhibit selected features only.

The intention of the propagators of "rapid prototyping" seems to be to produce throw-away prototypes - with the same functional scope as the actual system, but of lower quality - which precede the development of the actual system itself. This approach implies the call for new techniques, such as prototyping languages and interpreters, which reduce the effort of prototype production. I would like to point out that this approach is highly problematic:

- Since the specification does not yet exist at the time of prototype production, it is not clear what the functional scope of the prototype should be, and we find ourselves thrown back into the kind working style which was - with good reason - deplored ever since the 1960's.

- Should the specification already exist, it is not clear what is to be gained by quickly producing a system with the same functional scope, but of lesser quality than the final product. It should be remembered that the essential thing about the prototype is its evaluation, for which there is no systematic basis available as yet and which will prove to be a large effort if the prototype itself is complex. Therefore the feedback obtained by the evaluation of such a prototype will come late and will be unreliable. The production of the real system will be delayed, with no obvious gain to justify the delay.

- If the prototype is to precede, rather than to accompany, the actual system it will not be helpful in dealing with changing requirements, as will be argued below.

Requirements for software embedded in technical or socio-technical systems must be expected to change, because

- original requirements were misstated (the probability of this may perhaps be reduced with a "rapid prototype"),

- the environment evolves and develops new requirements,

- the system, once in use, transforms its environment and thus itself contributes to producing new requirements.

Because of the last two of these points, a "rapid" throw-away prototype cannot be expected to aid in reducing troubles with changing requirements in the long term.

In view of all the problems cited above it seems appropriate to drop the analogy with engineering prototypes, to generalize the concept of a "software prototype" considerably and to combine its production with an orderly approach to software production.

In the following, a prototype will designate a preliminary version of the actual system which exhibits selected features of the final product.

There may be one or a series of such prototypes, depending on the needs of the specific project. The prototypes serve primarily to aid discussions about handling requirements, i.e. whereas their functional scope may be only a fraction of the actual product's; they are carefully designed, so as to illustrate how the system can be embedded into its working environment.

This way of using prototypes implies a different view of software development, which has been termed the process-oriented approach elsewhere (/**FLOYD** 81/). Rather than viewing software development as the production of one program, by going through several phases and ending up in "installation" and "maintenance", I prefer to view software development as a sequence of development cycles (re-)design, (re-)implementation and (re-)evaluation. It must be emphasized, that each development cycle is based on a specification from which the program version to be produced can be derived in an orderly fashion, thus there is no contradiction between this approach and software engineering methodology; instead the specification itself is viewed as an evolving document.

As opposed to the common phase-oriented approach, communication with the user is continuous and feedback from the evaluation of successive prototypes is incorporated into redesign at the end of each development cycle.

The specification serves as a common defining document for both software producer and user. In particular, the application model associated with the specification must be phrased so as to exhibit the embedment characteristics of the system in its working environment.

The responsebility of the user consists in providing, in each development cycle, an evaluation basis for the prototype which can be derived from the application model. In the absence of a theory we can still point to no systematic way of how to do this, but at least the new framework will allow to progress in small, meaningful steps.

In the context of the modified approach to software development described above, a prototype can assume one of the following roles:

1) A prototype may coexist with the actual product; it is, then, a program model of the same specification, less rigorously treated, and serves as basis of experimental changes before the program itself gets modified.

2) A prototype may coincide with the actual product: This is intended in version-oriented software production in development cycles, as described above. The specification is an evolving document; it may or may not change from one version to the next.

3) The product itself is a prototype: This arrangement is relevant to the production of standard software, which is designed to meet the functional requirements of a class of users, but where handling requirements can be decided by replugging existing components to fit individual needs.

4) Production starts from a prototype: Analysis and redesign of existing software can be viewed as a special case of the same approach.

Each of these arrangements may prove a valuable help to aid communication with the user in certain production settings and each of these can be combined with orderly programming methods. How much of a previous version of the program can be retained to be incorporated into a subsequent version, must be decided as part of the redesign effort following each development cycle.

In order not to create false hopes, however, we must remember that in this manner we have obtained a more flexible framework - no more. Modern software design and specification methods do not necessarily facilitate incremental partial changes, which makes the use of a specification as an evolving document awkward. We all know that in practice the discrepancy between programs and specifications increase with time, thus making the specification obsolete long before the program is shelved. We can hope that progress in specification research will help to remedy this situation.

On the other hand, we cannot hope that communication with the user - even based on carefully designed prototypes - will significantly improve, unless we find a theory of software embedment which is based on solid grounds in both psychology and the social sciences. Such a theory will help the software designer to make intelligent choices that can be justified to the user by rational arguments, rather than by individual tastes. It will also allow for the systematic evaluation of prototypes.

Because of the serious concern for the adequacy of software systems in their working environment, research efforts in these directions must be considered one important front of software engineering research.

References:

FLOYD, C.:
A Process-oriented Approach to Software Development.
in: Systems Architecture,
Proc. 6th ACM European Regional Conf. (1981),
Westbury House 1981, pp. 285-294.

ILICH, I.:
Tools for Conviviality.
FONTANA/COLLINS 1979.

LEHMANN, M.M.:
Programs, Life Cycles and Laws of Software Evolution.
in: Proc. **IEEE**, Special Issue of Software Engineering
SEPT. 1980, pp. 1060-1076.

TURSKI, W.:
Some Problems of Software Engineering.
in: **HP3000** International Users Group Meeting
OCT. 1981.