Initial Thoughts on Rapid Prototyping Techniques

Tamara Taylor

and

Thomas A. Standish

Irvine

Programming Environment Research Center

Technical Report 167

Computer Science Department

University of California

Irvine, California 92717

February 7, 1981

a b s t r a c t

This paper sets some context, raises issues, and provides our initial thinking on the characteristics of effective rapid prototyping techniques.

After discussing the role rapid prototyping techniques can play in the software lifecycle, the paper looks at possible technical approaches including: heavily parameterized models, reusable software, rapid prototyping languages, prefabrication techniques for system generation, and reconfigurable test harnesses.

The paper concludes that a multi-faceted approach to rapid prototyping techniques is needed if we are to address a broad range of applications successfully --- no single technical approach suffices for all potentially desirable applications.

I n t r o d u c t i o n

When we are given a new computer system to build, we may find ourselves facing one of several possible sets of circumstances:

(1) In the best of all worlds, the system requirements are precisely stated, they reflect the true needs of the users, and it is known how to implement the system using techniques in the current state-of-the-art.

However, it is not always the case that things are as optimal as they are in case (1). For instance:

(2) The requirements may be perfectly stated but it may not be known how to build a system with the required properties. For example, we may specify that we want to build a "world champion chess program." We can state the requirements with complete precision, and there can be no doubt about whether the true needs of the users have been correctly and completely captured in the requirements statement. The rules of chess and of chess tournaments are clear, complete, and unambiguous. However, there does not exist the knowledge of how to build such a system in the current state-of-the-art in computer science. Here, we have a case where the "ends" sought are perfectly well specified, but the "means" are unknown.

Yet another case occurs when the "means" are adequate but the "ends" are unclear. For instance:

(3) The user may not really know what he needs and has no idea of how his needs may change later. E.g.,"My office procedures are too ad hoc. I need an office information system that will organize my transactions and will allow me to make decisions more effectively." In this case, there are enough computer science techniques available in the current state-of-the-art to build office automation systems, but the "ends" to be served are too vague.

So we see, in general, that the user may either not know what he wants, or may describe what he wants in such vague unhelpful terms that the system is not really specified, or he may specify what he wants exactly, but computer science does not know how to build what he wants.

In some of these cases, having a precise specification language is of no help, since the user really doesn't know what statements to make in such a language --- that is, he can't articulate his needs if he doesn't know what they are regardless of whether or not there is a precise language for stating them.

Under some of the above circumstances, what may be needed is a learning process. System implementers may attempt to build a system they think meets the true user needs on an experimental basis. Then, they may attempt to expose the user to its behavior to allow the user to experience what it can do and to learn whether he thinks it satisfies his needs. Often, the user is able to articulate what he likes and dislikes about an actual working system that gives him concrete examples of behavior to judge, and often the chance to react

to actual system behavior helps him to articulate statements of his needs, especially if he was previously unable to do so.

Thus, exposure to working systems is often a helpful learning method. Looking at a system design on paper may not be as effective as direct exposure to the system behavior, since the user can often understand the latter without technical training, whereas it takes technical training to examine a design and to imagine what its behavioral implications are.

If it is the case, then, that exposure to working system behavior is a useful idea, we may wish to find ways of producing such results rapidly and cheaply. This gives birth to the concept of rapid prototyping. Rapid prototyping techniques are just techniques for constructing working models of systems rapidly and cheaply. The aim is to accelerate our learning process about whether a system design meets user needs, and to do so as cheaply and rapidly as possible.

Here we can adopt the philosophy that, "Programs are like waffles --- the first one should always be thrown away." We can attempt to find ways of exchanging the increased power and capacity of the new generation of computers to get compression and ease of expression, since the latter is at a premium and the former may soon be cheap to acquire.

Rapid prototyping may also have a role to play in helping to improve software quality progressively during the software lifecycle.

During the software lifecycle it is usual to find activities such as: (1) requirements analysis, (2) specification, (3) design, (4) coding, (5) testing and integration, and (6) maintenance and upgrade.

Because we live in an imperfect world, each of these activities usually takes place in the context of imperfect predecessors. That is, we live in a world where requirements are never likely to be complete or accurate, designs are never likely to be correct, and implementations are never likely to satisfy the requirements and reflect the design intentions perfectly.

In such a world, we must resort to special measures in order to improve quality progressively. This yields various quality assurance disciplines such as design walkthroughs, independent validation, thorough testing at pre-release time and so forth. Even maintenance can be seen as an incremental activity that progressively improves software quality by, for example, removing bugs and upgrading the system to meet user needs better.

At a deeper level, we see that there are feedback loops between the activities in the lifecycle that help us incrementally to improve understanding and quality achieved at each stage. Thus, we may only really begin to understand the true system requirements when we are exposed to the behavior of an implementation. Cyclical exposure to the behavior of the artifacts we build may be necessary to achieve understanding of the true requirements, especially for a system we are trying to construct for the first time.

In this context, when we attempt to build systems with novel capabilities, we often imperfectly understand the true user needs. There is a learning process involved in articulating the true user needs, which involves exposing the user to a working initial version of the system and seeing if he is satisfied. Often, the user discovers that the requirements he originally stated need to be revised in light of the experience gained with a working model of the system. Here again, exposure to a working version of the system accelerates the learning process in which the user discovers and articulates his true needs.

We often see circumstances in which the requirements statements for a system get incrementally improved in just this fashion. However, if the true requirements are not discovered and articulated early enough in the software lifecycle there is often considerable wasted activity downstream. Designs and implementations are sometimes built to satisfy unstable requirements statements. As the requirements shift, the designs and implementations must be redesigned and reimplemented to track the changing requirements. This can be highly wasteful of resources as modules and portions of the system must be discarded and redone, and, perhaps worst of all, it is often sociologically disastrous to the morale of the designers and implementers who are forced to discard their previous work and are made to feel that their accomplishments may have little permanent value.

It would be highly useful, therefore, to find some methodology for learning about stable, accurate requirements as early as possible in the lifecycle in order to prevent as much downstream waste as possible and to prevent poor morale among project personnel due to shifting requirements.

If a methodology for requirements validation were available that involved static analysis of requirements statements and verification that requirements were complete, accurate, not over-constraining or under-constraining, and were truly reflective of user needs, we could apply such a methodology at great savings. However, no such satisfactory static analysis seems to have emerged and to have been successful.

Another approach is to consider the possibility that incremental learning via exposure to the behavior of working prototypes is a methodology for which we already have existence proofs, and to attempt to devise rapid prototyping techniques that enable rapid, cheap construction of working system prototypes (without much attention to efficiency or polish).

There are two more circumstances in which rapid prototyping techniques are of potential value.

First, rapid prototyping has to do with quick response to changing requirements after a system has been released as well as with initial articulation of correct, complete requirements. There are instances where the requirements for a system that we are perfectly happy with may change in a matter of hours and may need to be upgraded

in a matter of hours in response. For example, if we suddenly discover that an electronic countermeasures device fails to protect adequately against certain surface-to-air missiles, the viability of a nation's air defenses may depend on reprogramming these devices rapidly.

Second, in some branches of industry and government, it is not uncommon that three to five years are spent building a system that may be subsequently determined to be non-responsive to user needs, and the system requirements analysis is iterated in succeeding procurement cycles. In this case, exposure of the user to working versions of the system still happens --- only it happens with very long cycle times in the learning feedback loop. This is the second circumstance in which a speed-up of the response time is important.

Thus, rapid prototyping has to do with more rapid effective development of initial versions of the system as well as with quick response to changing requirements in released systems during the maintenance and upgrade portion of the lifecycle.

Often, if we relax the optimization constraints on a system, we can build models at less expense than the expense of building the real system. Thus, partial models of the system can function as mock-ups that yield samples of system behavior adequate to determine responsiveness to user needs at a fraction of the cost of real systems. In addition, in building a prototype, often one need not model everything. Instead, one need only model things relevant to the functionality of the system as viewed by the user.

For example, the authors have built a model of an Automated Flight Service Station Information System incorporating aircraft weather and routing data to be used for pilots for preflight briefings. The prototype did not have on-line weather data, nor did it work for 4,000 terminals spread all over the continent, nor did it have all the airways and navigational aids in it. Rather, it had weather for one twelve hour period and airway and navigational aids only for the northeast corridor. Further, it modeled only what the user would do interacting at one terminal while getting weather, winds aloft, and navaid data, and while calculating and filing a flight plan. But it did model very accurately what the user could do at such a terminal, and was built at a very small fraction of the cost of building a real system (two man-weeks as opposed to who knows what?).

The database was resident in core rather than stored in large file structures on secondary memory, and so forth. The prototype was constructed in an extensible language and was used in a live demo at the Federal Aviation Administration in Washington, D.C.

During the demo, it was evident that potential users of the system could learn about whether the prototype satisfied their true user needs --- i.e. the prototype was a fully effective means of accelerating the learning process about the true system requirements at a fraction of the cost of experimentation with real systems. We cite this to illustrate our confidence that a basis already exists for a workable technology of rapid prototyping.

In the next section, we proceed to examine some possible technical approaches to the development of a set of useful rapid prototyping techniques.

Technical Approaches

What are some good techical approaches to rapid prototyping? Are there good general purpose techniques? In addition to such general techniques, are there situations where we must have well-adapted special purpose techniques in order to build prototypes rapidly and cheaply?

Are there ways to trade processing power for ease of construction? Can we devise good rapid prototyping languages that give us a promising means for accomplishing this?

What specific method shall we use to expose the user to the behavior of the prototype, and by what methodology can such exposure result in improvement of the requirement statements? Can we get traceability of the requirements and some specific methodology for completeness of enumeration or coverage? For each prototype can we automatically produce a test plan for running the prototype to check out the requirements systematically?

Here are some possible technical approaches:

Heavily Parameterized Models

Sometimes we can have a family of systems that differ from each other by variations in parameters or tables. For example, once we had a computer graphics system that modeled a radar air-traffic control system. The CRT displayed moving aircraft radar targets together with attached data blocks on a background map of the airspace. The system

incorporated laws for moving the airplane targets to simulate a real-time radar air-traffic control system.

Then one day, some people from the St. Lawrence Seaway came by and mentioned that they needed a system for controlling ship traffic on the St. Lawrence Seaway. Could we give a demonstration of a system concept for controlling ships?

We were able to substitute new display tables giving maps of the seaway, new symbols for ships, and new equations for ship motion starting with our air-traffic control system. In a matter of days, a live demonstration of the Seaway control system was working. This illustrates a technique for rapid-prototyping. It was only necessary to view the air-traffic control system as an instance of a more general system for moving "widgets". Once this view was adopted, it was trivial to respecialize the system to move ships instead of airplanes.

This yields the technique of rapid prototyping by generalization and respecialization. In general, we may wish to have heavily parameterized systems that can be specialized into particular prototypes by supplying appropriate parameters, tables, and subroutine packages.

Reusable Software

One way to gain leverage in constructing working systems rapidly is to make use of other people's work. The goal is to "Stand on other people's shoulders, instead of stepping on their toes." By this means, we may advance more rapidly.

We already use other people's work when we call subroutines from a general subroutine library, or when we import packages of utilities in some language that runs on our own machine. FORTRAN is a frequently used medium for the exchange of programs since FORTRAN runs on nearly every machine. We also make use of other people's work when we implement algorithms drawn from the general computer science literature. Why write your own binary search routine when, e.g. Knuth, has done all the good thinking to get it right and to make it efficient and when all you have to do is pay the cost of translation into your own programming language?

In a more powerful sense, if we can agree on interface and linkage conventions, it may be possible to have large libraries of modules that can be conveniently assembled. This requires assembly techniques and good languages, such as Ada, in which we can do information hiding, clean interface specification, and independent compilation. Perhaps Ada will give us the incentive to have large libraries of reusable software giving us reliable pieces we can assemble rapidly.

The recent Irvine Ph.D. thesis of Jim Neighbors, Software Construction Using Components, gives an approach to building systems out of reusable software components [Neighbors 1980, UCI-ICS, TR160]. In this approach, reuse of software results only from reuse of analysis, design, and code --- not just reuse of code. Sophisticated program transformation techniques are part of the approach as are careful specification of interfacing techniques for software components.

Prefabrication and System Generation

If we have to build prototypes with special device types included, such as special types of displays, we may need to have prefabrication methods for programming the device types easily. For example, if we have a two-dimensional incrementally updatable display, we may want ways of programming the usual sort of graphical user interface package that has capabilities such as windowing, clipping, menuing, inking, latching, cut-and-paste editing, hand written character recognition, and the like. It should be possible to define tables giving the menuing choices, and to have lots of these capabilities come in prefabricated form. There should be system generators that take parameters and tables as inputs and which generate a display interface according to the paradigm for the particular device. This would hasten the job of generating a display interface and would reduce the cost.

Restricting Functionality

To expose a user to a sample of working behavior of a system, often we do not have to model everything. Instead, we need model only the functionality that the user will see.

In a previously mentioned example (that of the pilot's flight service terminal), we needed only to model a single terminal (not 4000), a small portion of the airspace (the northeast corridor, not all of North America), and a single twelve hour period of weather (not real-time, on-line weather). This enabled potential users to get the feeling for how to use the system to request weather and winds aloft data, how to file flight plans, and how to get "airline captain quality" flight logs printed, without having to model everything.

Reconfigurable Test Harnesses

In some situations, such as testing satellites out on the ground, or testing any sort of "embedded" computer system that has to respond to sensor data in real-time, and that has to control devices --- we may have to simulate the environment of operation to see how a prototype behaves. This requires consideration of the properties of the "test harness" and the simulation of events that the prototype must respond to. It is not enough to have just a rapid prototyping language. Here we need mature consideration of reconfigurable test harnesses complete with event simulators and data collection capabilities.

Different embedded systems may need to be hooked up to different real or simulated devices in such a test harness. For example, we may want to attach real or simulated clocks, gyroscopes, and accelerometers to the test harness in which the embedded system prototype is being checked out. This requires us to have a technical approach to being able to reconfigure the test harness rapidly --- dropping and adding new peripherals using some cleanly specified interfacing techniques. Simulation, data collection, and data analysis capabilities clearly need to be included in order for such a a system to be adequate to its task. (We are indebted to Dr. Stewart I. Schlesinger of the Aerospace Corporation and to Dr. Larry Druffel of the Defense Advanced Research Projects Agency for the origins of these ideas).

Rapid Prototyping Languages

Rapid prototyping languages may give us a general technique we can employ if our goal is to have a well-rounded set of rapid prototyping techniques.

The following list of possible characteristics and features of rapid prototyping languages represents an initial cut at our thinking on desirable features: (We are indebted to Dr. David A. Fisher of the Western Digital Corporation for some of these ideas.)

(1) Strongly extensible: (almost all of the following suggestions and characteristics address and expand upon the meaning of the phrase "strongly extensible").

(2) Program text is data: can have program writing programs and can execute programs that have been constructed as values.

(3) Has interpreter, and is highly interactive. Evaluator is extensible and incrementally reprogrammable. Can overload evaluator functions and can incrementally extend standard system functions for printing, selection, assignment, equality, and the like. Explicit control over the read-eval-print loop.

(4) Run-time environment accessible as data structure in the language.

(5) Extended calling forms: self-replacing calls as well as value returning calls. Command completion (or prompting with automatic fill-in) of calling forms (e.g., hit "escape" button and calling form fills in up to next point of ambiguity or next parameter position). Postponed definition of meaning. Use of syntax macros and program transformations to supply meaning and to show how to exchange the new for the known.

(6) Remove explicit representational dependencies: When we went from assembly language to high level languages we submerged things critical to the implementation such as register allocation and mappings between names and locations and we introduced application oriented things such as arithmetic expressions. Can we do more of this?

(7) Concept minimization: remove different ways of saying the same thing. E.g., T'FIRST, Array(Index), Function(Arg), and Record.Component are all different ways of saying "the X of Y" in Ada.

(8) Boundary removal: example --- use conceptually unbounded objects such as Iota(infinity), infinite sets, or unbounded arrays. Automatically allocate and use only that finite portion needed to compute results.

(9) More abstract primitives: non-determinism, backtracking, use of predicates in program forms (e.g., Gcd(x,y)= Max{d: d|x & d|y}).

(1Ø) More powerful ways of defining things: In definitions, we always show how to exchange the new for the known. Already in extensible and ordinary programming languages, we have numerous ways of doing this. Function definition and calling, introducing new data definitions, introducing new operator definitions, and defining new notations each illustrate this principle. If we can use new calling forms and if we can introduce new ways of exchanging them for text with assigned meaning, we can have a very powerful handle on introducing compressed forms of expression of use in rapid prototyping. E.g., we may replace Gcd(x,y)=Max{d: d|x & d|y} with appropriate text in a programming language for computing the Gcd. The capability of manipulating programs as data and of having program writing programs opens up for us the possibility of powerful paradigms of exchanging new forms of expression (using predicates, sets, and other microworlds, for instance) with known executable program text.

(11) Data Extension Features: (a) can add new type and new operations on the type, (b) can give it all privileges of any initially supplied type including, (c) extend printing routines to print new type, (d) lexical recognition of literals of the new type, (e) assignment of values of new type, (f) equality defined on new type, (g) selection notation can perform selection on components of new type, (h) information hiding of its internal representation details, (i) extended appropriate new notation for operations on the new type.

(12) Use of expressions that compute locations which can be assigned values: E.g., (if x>Ø then y else z end if) := 9.

(13) Extended control structures: tasking/rendevous, real-time, exceptions, continuously evaluating expressions, monitors and traps, interrupts and priorities, back-tracking, side-tracking.

(14) More user services: diagnostics, type checking, information hiding and encapsulation, increased number of safe transformations because there is more information in the language.

(15) More Powerful Concept of Types: Can we strengthen the type system with a more powerful form of definition invocation and recognition by attaching more advanced properties to objects, such as "type T is the set of all even integers between 2 and 256 except 56." Attribute attachment and textual substitution switched on attached types. E.g., (x) is (y) ==> if (y) is a (Boolean procedure) then Y(x); elsif (y) is a (constant) then x=y; elsif (y) is a (set) then (x) in (y); end if;, and so on.

(16) Artificially Intelligent Transformations: Script based programming, transforming plans of purposeful agents into programs. Calculus of program derivation and synthesis. Use of special micro-worlds such as sets, sequences, bags, heaps, trees, geometry, relations, total orders, etc.

(17) Strong Program Transformations: program transformation catalogue and semi-automatic system for chaining transformations (as in Dennis Kibler's Ph.D. Thesis, UC Irvine).

Coupling Rapid Prototyping into the Software Lifecycle

We have already mentioned that we need to have an explicit feedback methodology for taking the results of a user's exposure to the behavior of a working prototype and creating from them an incremental update to the requirements statements. Thus, rapid prototyping must feed back on the requirements. But can it also feed forward into downstream lifecycle activities?

One possibility is to have a strategy for reworking the prototype into a polished, production-engineered version of the system.

Incremental Redevelopment

If we have identified a working prototype that provides a core of functionality certified by the user to meet his perceived requirements, we may wish to extend the core into a complete system that displays the same core functionality.

Generally, there may be two sorts of incremental activities we need to perform to transform an initial core system into a complete, production-engineered final system:

(1) extending it functionally to a complete system by adding functionality that the user didn't see, which wasn't in the prototype, and which is needed to have a full operational capability, and

(2) altering or replacing inefficient pieces of the prototype to yield required performance efficiency.

Activity (2) may involve rewriting the system in an efficient systems programming language, using the program written in a rapid prototyping language as a "design".

Activity (1) is inherently a system design activity that requires knowledge of state-of-the-art system implementation techniques drawn from parts of software engineering independent of rapid prototyping and of known effectiveness in current practice.

We need to get some experience with some techniques for incremental redevelopment of a prototype to see what sorts of additional effort are required to rework prototypes into final systems. What are the ratios of effort involved to develop the prototype versus the effort involved to transform the prototype into a final system? What sorts of incremental activities are required during reworking, and how can they be scheduled and managed?

Feeding Design

Prototypes may perhaps best be used in some settings by not attempting to rework them into final systems, but rather by having them serve as an additional source of precise behavioral specification used as an input to a system design. In effect, they may serve as programs written in a program design language that are core designs for the larger system design. In this case, we do not care so much

about running efficiency as we do about clarity of conceptual structure and extensibility to complete system designs.

Are there any incremental techniques for expanding a core design written in a program design language into a complete system design? Can the features of a good rapid prototyping language serve double duty by providing an attractive basis for program design languages as well? Could programs written in a rapid prototyping language be used directly as core designs for downstream design completion? Could we devise a new "inside-out" software design methodology based on such an approach?

Coupling with Requirements Review and Testing

It would be useful to have some systematic method for conducting a review of the system requirements that is closely coordinated with systematic examination of the behavior of a prototype. The reason this is important is that we are trying to use prototypes to check out whether the requirements are adequate, and we need some systematic way of performing this task, especially in cases where the requirements are lengthy or complex.

To get samples of the behavior of the prototype, we need to conduct a series of tests. Then we need to examine the behavior of the prototype revealed by the tests to see if that behavior meets the items of the requirements that it is supposed to satisfy.

Thus, we have a case where testing (which is used to extract behaviors) needs to be coupled with requirements review (which checks whether the elicited behaviors satisfy the relevant pieces of the requirements). Systematic traceability of the requirements to the tests that elicit the behaviors that are supposed to satisfy them seems to be called for.

While we have no particular ideas or approach to offer on this subject, we want to flag the issue and to suggest that it is important that it be addressed in the future in connection with rapid prototyping methodologies.

Limits of Prototyping

Up until this point in the discussion we have extolled prototyping, but what are the limits on what we should expect?

What do we not easily learn from prototypes? Here are some possibilties (kindly contributed by R. Kling):

(1) What it is like to live with the system for a while.

(2) How easy a real system will be to alter.

(3) How a system will behave when it is pushed to the extremes of performance (e.g., heavily loaded, various buffers nearly exhausted, displays saturated with data, etc.).

(4) How a system will interact with other elements in the software environment or related systems with which it should easily share data.

In general, there can be many different prototypes of a given target system. Each is like a selective shadow, highlighting some features and losing details of others. To the extent that rapidly developed prototypes systematically distort since (a) they're designed to be plastic, (b) they're designed to be small, and (c) their interactions with other software will differ (by being more flexible and less easily interfaced, in some cases) rapid prototyping can lead users to misperceive what the target system may actually be like.

Thus, rapid prototyping may well be like democracy --- flawed, but far better than the available alternatives.

C o n c l u s i o n s

It is reasonable to conclude from our discussion that no single technical approach to rapid prototyping techniques can serve as a panacea --- universally applicable in all settings and fully sufficient to make prototyping cheap and rapid.

Rather, in some settings, such as the "test harness" setting for real-time, embedded systems, and the advanced two-dimensional, incrementally updatable display setting, we may need to take advantage of specially adapted rapid prototyping techniques, such as strongly parameterized system generation, reusable software, or easily reconfigurable test apparatuses coupled with event simulators and data collectors.

Thus, the existence of rapid prototyping languages alone as a general purpose technique won't provide rapid, cheap construction of prototypes in all settings, even though they may considerably enhance prototyping in many general settings and even though they may feed downstream software lifecycle activities effectively.

This points to the conclusion that we must have a multi-faceted technical approach to rapid prototyping if we are to address a broad range of prototyping applications successfully.