Prototyping as Modeling: What is Being Modeled?

Jeff Rothenberg
The RAND Corporation*
1700 Main Street
P.O. Box 2138
Santa Monica, CA 90406-2138
U.S.A.
e-mail: jeff@rand.org

Abstract

It is commonly argued [16, 7] that an evolutionary approach to software system development based on prototyping can solve many of the problems of traditional development methods. In pursuing this approach, prototypes are often thought of as models, yet it is unclear exactly what they are modeling. This question is examined below, leading to an inquiry into the nature of prototypes, systems, conceptual models, specifications, designs, and the relationships among them. This in turn leads to a fundamental inquiry into the nature of modeling, which reveals that traditional views provide little insight into what it means to build a model for something that does not yet exist. This paper examines these issues in some detail and attempts to provide deeper insight into prototyping by means of a new understanding of the modeling relationship.

1. Introduction

The process of building successful computerized information systems is widely recognized as something of a "black art". While progress in computer hardware continues unabated, corresponding progress in software development has been agonizingly slow. Despite the undisputed power of the software medium, it has proven surprisingly difficult to design, implement and deliver any but the most trivial systems in a way that reliably satisfies their purchasers and users. Though individual, small-scale application programs often succeed, a strategy for building successful large-scale information systems remains an elusive goal [8, 9]. Modern software engineering methodology has made great strides in providing programming languages and environments that embody the hard-won wisdom of decades of experience, yet software projects continue to overrun, flounder, and fail. To set the context for the discussion of

*This research was sponsored by the Defense Advanced Research Projects Agency (DARPA) under the auspices of RAND's National Defense Research Institute, a Federally Funded Research and Development Center sponsored by the Office of the Secretary of Defense, under contract No. MDA903-85-C-0030. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official opinion of DARPA, the U.S. government, or any person or agency connected with them.

prototyping and modeling that follows, this section examines some of the reasons for the failure of software development.

The frequent failure of the software development process stems from failures in several subsidiary processes: the design process, the software engineering process *per se*, and the technology transfer process. These processes are fundamental to software development, regardless of which particular model of the development cycle is adopted. Though this paper primarily addresses the design process, the other two processes will be discussed first, since all three are closely interrelated.

1.1. *The failure of the software engineering process*

The failure of software engineering *per se* is often perceived as the crux of the problem. This perception may be due to the fact that the software engineering process is highly visible, time-consuming, and obviously expensive (though poor design may accentuate these problems and poor technology transfer may overshadow them by making the entire development process a waste of time). It may also be that software engineering problems receive disproportionate attention within the software engineering community simply because they fall most clearly within its jurisdiction: such problems are at least in theory amenable to solution by means of improved programming languages, environments, and tools.

Whether or not it is the core of the problem, the failure of software engineering is certainly a major hindrance to software development. Building large programs is still expensive, time-consuming, and fraught with risk, despite the fact that modern techniques such as structured and object-oriented design, dataflow analysis, and encapsulation can greatly improve the quality of software products. At their best, such methods facilitate verifying that a program correctly implements its specifications. However, it is generally acknowledged that even such verification cannot "validate" a program in the sense of proving that it does the right thing [6, 1, 12].

1.2. *The failure of the technology transfer process*

An equally significant cause of software development failure is the failure of the technology transfer process.* This process is often thought of as the process by which new technology is delivered to its end-users, where "end-users" may include those who use the new technology directly or who depend on its use or consume its results, whether directly or indirectly.

Yet waiting until software is ready for delivery before involving end-users is a good way to ensure the failure of the development process. Software development is far more likely to succeed if the end-user community is involved from the earliest stages onward. The technology transfer process should therefore be conceived as encompassing the involvement of proposed end-users during the specification and design process, as well as throughout software development. This is especially true for "targeted technology development" in which a new

*This is referred to as "implementation" in the social science literature [4], but in the context of software development, it is less confusing to reserve the term "implementation" to mean building a system and to use the terms "transfer" or "delivery" to denote the introduction of a system into an organization.

technological solution is designed to meet the specific needs and desires of an identified end-user group (as is the case for many information system development efforts).* In addition, the technology transfer process should include administrative efforts to ensure that a proposed system will be accepted and used when it is finally delivered.

Such concerns are often given only token emphasis or are recognized and addressed too late to be effective, despite mounting evidence that the successful introduction of targeted technology requires early and continued user involvement in the development process [3, 4].

1.3. *The failure of the design process*

Since the design and specification process is logically (if not always temporally) the starting point for any software development project, it warrants special consideration. Early software development efforts, which "plunged in" and built systems before understanding what those systems should be, conclusively showed that this was a recipe for disaster [8, 23]. Some kind of design effort must be undertaken, whether it is carried out prior to any implementation or in an iterative fashion (for example, using prototypes to evolve and refine a design).

The verb "design" can refer to many aspects of software development. In particular, it is often used to mean two rather different things: the design of what a system should *be* and the design of *how* the system should be implemented. In this paper, the latter is thought of as part of the software engineering (or "implementation") process, while the term "design" is reserved to denote the activity of designing what a system should *be*. The design activity also includes a "specification" activity that produces a concrete description of what a system should be; the term "specification" will be used here to emphasize the production of this concrete description as distinct from the more amorphous activity of designing what the system should be.

(Note that the software engineering process involves both *designing* an implementation and *building* that implementation. This terminology is especially confusing because the word "implementation" can be used to mean either a process or activity (of building something) or an artifact (i.e., the chosen approach or the thing built according to that approach). Similarly, the word "design" can be used to mean a process or activity (of designing something) or an artifact (the thing designed). In this paper, the term "design process" is used to denote the entire process of deciding what a system should be, whereas the verb "design" is used to refer to the activity of designing, and the noun phrase "a design" is used to mean the thing produced by the design process. Similarly, the term "implementation process" is used to denote the entire process of implementing a system, whereas "implementation" is used to mean the activity of implementing a system, and the noun phrase "an implementation" is used to mean the thing produced by the implementation process. The terms "specification process" and "specification" are used in an analogous way.)

The failure of the design and specification process continues to be a significant cause of software development failure. The basic problem is that it is difficult to know whether the right system is being built and whether it will do the right thing even if it is implemented perfectly. It is even

*In "untargeted technology development", where a new technology is developed in the absence of an identified user population, technology transfer is necessarily more problematic.

difficult to define what "doing the right thing" means. It will be argued below that this difficulty stems from a lack of understanding of the design process. In particular, although methods such as structured and object-oriented design [11, 19, 18] attempt to provide relatively formal ways of specifying and designing a system, they implicitly assume a conceptual model of the desired system in terms of its domain, its users, its functionality, its interfaces, and the organizational context of its target environment. This conceptual model determines (or at least constrains) all subsequent design decisions, and is therefore a crucial element in the design process. Errors or misconceptions in formulating the conceptual model are likely to have expensive or disastrous consequences. Yet it is difficult to validate the conceptual model for a system. The current trend toward prototyping can be viewed as an attempt to "try out" and refine the conceptual model in the early stages of the design process, to avoid dire consequences.

The remainder of this paper focuses on the roles of prototypes and conceptual models in the design process. The next section discusses prototyping as a technique for avoiding or ameliorating many of the problems discussed above. Section 3 discusses conceptual models and their relationship to prototyping. Section 4 proposes a new view of modeling that provides a framework for understanding the design process. Section 5 uses this framework to provide further insight into the role of prototyping in design.

2. Prototyping

Recent trends in software development have moved away from strictly sequential development strategies such as the well-known "waterfall" approach, in which various phases (such as specification, design, coding, testing, integration, etc.) are performed in sequence, with minimal backtracking [23]. It is now generally accepted that these sequential strategies incur too high a risk of encountering unforeseen problems in their later phases, requiring drastic backtracking (i.e., redesign), in which earlier phases must be redone at excessive cost. Except in cases where specifications can be formulated with great certainty in advance, these sequential strategies are rapidly losing favor. Taking their place are a number of new incremental, evolutionary, and cyclic strategies that attempt to reduce the risk (and therefore cost) of software development by identifying key uncertainties as early as possible and resolving them before proceeding. One of the best known and most general of these new strategies is Boehm's "spiral" model of software development and enhancement [5]. This focuses on identifying risks in all phases of development and is broad enough to allow the use of any appropriate technique for resolving uncertainties to reduce these risks.

One technique that has been widely proposed for use with these new strategies is prototyping, i.e., building partial, exploratory implementations of certain aspects of a proposed system prior to building the "real" system. In the context of risk reduction, prototyping is a way of trying out proposed approaches or comparing alternatives to reduce the uncertainty inherent in the development process. In this sense it is an experimental technique for evaluating known alternatives or testing specific hypotheses; for convenience, this will be referred to as **evaluative prototyping**. However, the potential of prototyping extends well beyond this: an equally important use of prototyping is to provide a surrogate system that can be investigated to improve the design of the eventual system; this will be referred to as **strawman prototyping**. In fact, it will be argued that prototyping should be thought of as a primary technique for validating and

refining the conceptual model for a system. Before elaborating this argument, it will be helpful to discuss prototyping in greater detail.

In general, the purpose of a prototype is to learn something about a proposed system during the *early* stages of development, to avoid the excessive cost of discovering important mistakes later on. A prototype will be said to be **cost-effective** to the extent that it delivers useful results of this kind sufficiently early in the development process.

2.1. *Evaluative prototyping to reduce risk*

One of the primary uses of prototyping is to reduce risk in the software development process by facilitating the early resolution of uncertainty. To do this, a prototyping effort must be focused appropriately. Though it may be tempting to begin by prototyping some technical aspect of a proposed design, this does not always (or even often) address the areas of greatest risk in an information system effort. Often the areas that most deserve prototyping effort will have to do with functionality, user interface, system integration, reliability, and performance. These aspects bear directly on the ultimate success of the system in its target environment. They are also all closely tied to the choice of an appropriate conceptual model, as discussed below (Section 3.2).

In cases where the uncertainties in a software development effort involve relatively objective questions, such as whether a given technique will work or not, prototyping can often provide an answer by direct experimentation. For example, technical or performance uncertainties may be resolved by building a prototype that demonstrates that a proposed implementation approach produces a feasible solution to a technical problem. Similarly, system integration uncertainties may be addressed by building a prototype that demonstrates that a proposed approach actually works with existing systems, databases, or networks. These are examples of “demonstration proofs” where the prototype proves that something is possible by actually doing it (though differences in scale or other constraints between the prototype and the proposed system may still leave some residual uncertainty as to whether this “proof” is valid).

Even when uncertainties involve subjective factors, prototyping may still reduce uncertainty by producing a “preview” of some aspect of a system that can be evaluated by subjective means. For example, functional uncertainties may be addressed by implementing skeletal functionality to see if all parties agree that the appropriate functions have been identified and targeted. Similarly, user interface uncertainties may be addressed (up to a point) by having end-users try out a proposed approach and observing their reactions.

An evaluative prototype of this kind can address combinations of such questions, but to fulfill its promise of resolving uncertainty *early*, it must focus on a relatively small number of questions to avoid becoming so complex that it requires undue development time. It may be more effective to undertake several independent evaluative prototyping efforts, each focused on a different question about the proposed system,[*] rather than trying to build a single, comprehensive prototype that runs the risk of becoming a complete initial system implementation. Furthermore, prototypes should generally be thought of as “throw-away” implementations, so that they can remain free from all constraints except those directly related to the questions being addressed;

[*]This corresponds to “mini-spirals” in the spiral model [5].

otherwise, the speed of prototype development will be compromised by the need to satisfy these constraints. For example, unless reliability is the focus of the prototyping effort, it is typically de-emphasized (or even ignored) to expedite prototype development.

When used to answer questions like those above, whether by objective or subjective means, evaluative prototypes typically produce bounded answers: either a technique works or it doesn't; either an interface is acceptable or it isn't. Yet beyond answering such "go/no-go" questions, prototypes may be used as a more intimate part of the design process, extending their use beyond mere risk reduction.

2.2. *Strawman prototyping to improve design*

In addition to serving as an experimental technique for evaluating known alternatives or testing specific hypotheses, prototyping can be used to gain preliminary experience with some aspect of a system, thereby refining and improving the system design itself. That is, beyond merely testing and validating the feasibility of design decisions, a prototype can be used to investigate the utility or limitations of a proposed system, leading to new design insights.

This requires thinking of a prototype as a "strawman" (or "pilot") implementation that can be tried out to make sure that an appropriate system is being built. Unlike evaluative prototyping, strawman prototyping answers open-ended questions about what the proposed system should do and how it should behave. The common thread across these two kinds of prototyping is that they both deliver their results *early* in the development process. In general, the purpose of any prototype is to learn something important about the eventual system or its implementation *before* actually implementing it.

By having end-users (or even developers) use a strawman prototype, it is often possible to discover things about a proposed system that would otherwise not be revealed until the eventual system was delivered. For example, missing, unnecessary, confusing, or inappropriate functionality may become apparent, or subtle interactions across seemingly independent functions may emerge. Similarly, integration or interface requirements may surface which could only have been anticipated by the most omniscient systems analysts. Without the use of strawman prototyping, such insights are most elusive for systems that provide truly novel capabilities that have no direct analog among existing systems or even in the manual world; however, such insights may be revealed even for targeted technology that is designed to satisfy a well-defined, expressed need.

The use of a strawman prototype to refine the design of a system can be thought of as simulating the eventual system [23] before building it, so as to understand it better. The motivation for this, of course, is to improve the design, so as to improve the eventual system. But this is circular: a prototype is asked to simulate a system that does not yet exist, in order to design that system, on which the prototype must be based. This apparent "chicken-and-egg" problem leads to an investigation of what it means for a prototype to be a model.

2.3. *The relationships of prototypes to their eventual systems*

It is natural to think of a prototype as a model. At first glance it appears to serve as a model of the eventual system. However, although a prototype *is* a model, it is misleading to think of it as a model of the eventual system. Before facing this issue directly, it will be useful to discuss the relationship of a prototype to the eventual system without assuming that this is a modeling relationship. It is convenient to characterize the relationship of any prototype to its eventual system along four dimensions, which will be referred to (somewhat arbitrarily) as focus, scope, depth, and scale.

The **focus** of a prototype refers to the aspect or aspects of the eventual system that are of concern for the prototype; for example, as discussed above, a prototype may focus on the functionality, user interface, system integration, reliability, performance, or similar aspects of an eventual system. By hypothesis, any prototype must focus on only one or a very small number of such aspects, in order to be cost-effective.

The **scope** of a prototype is a measure of how much of the eventual system the prototype represents, i.e., how big a subset of the eventual system's functionality (or overall behavior) is represented in the prototype. The intent of the distinction between focus and scope is to recognize that many aspects of a system are orthogonal to its functionality. For example, the user interface of a system can be examined within a small subset of the system's functional capability or across its full range. Similarly, the reliability of a system can be investigated over a narrow scope (e.g., the reliability of its network communication) or a broad scope (e.g., the reliability of all of its communication, file storage, I/O, etc.).

The **depth** of a prototype is a measure of how deeply it represents the behavior of the eventual system. For example, a shallow prototype of a message system might display "canned" messages, whereas a deeper prototype might actually perform communication to provide a more realistic surrogate for the eventual system. Similarly, a shallow prototype may only pretend to perform certain computations, providing random or pre-computed results of roughly the right form. In order to be cost-effective, a prototype must generally be fairly shallow.

Finally the **scale** of a prototype is a measure of how its size or performance compares with that of the eventual system. For example, a "small scale" prototype may demonstrate a proposed database technique without implementing a large database; on the other hand, if the focus of the prototyping effort is on the size requirements or performance of the database, then the scale of the prototype may have to be closer to that of the eventual system to yield meaningful results. That is, the scale of a prototype may depend on its focus. The scale of a prototype can often be equated with the degree to which it reflects the expected constraints on the eventual system.

The relationships of evaluative and strawman prototypes to their eventual systems are somewhat different. Since an evaluative prototype is intended to verify some particular question about the feasibility of its eventual system (or its proposed implementation), it generally represents only a small part of the eventual system with respect to only one aspect. Its value as an early eliminator of uncertainty depends on its representing as small a fraction of the overall eventual system as possible. For example, it may implement a processing technique that may be crucial for achieving the desired performance of the proposed system but which by itself provides only a trivial subset of the eventual system's functionality. Furthermore, unless they are themselves

the focus of an evaluative prototype, the reliability and performance constraints of the eventual system are often greatly relaxed in the prototype, making it unlikely to be usable for anything beyond answering the questions that constitute its focus. The relationship of an evaluative prototype to its eventual system is therefore a very limited one: it generally has narrow focus, narrow scope, and the minimum depth and scale consistent with its focus.

The relationship of a strawman prototype to its eventual system, on the other hand, is relatively broad. Since the purpose in this case is to gain experience with a surrogate system in order to improve the design of the eventual system, a strawman prototype must provide a usable subset of the functionality of the eventual system, at least with respect to the aspect or aspects of interest. For example, if a strawman prototype is built to gain a better understanding of how the user interface of its eventual system should be designed, it must provide at least the appearance of a reasonable subset of the functionality of the eventual system to allow exercising the interface in a realistic way. Therefore whereas a strawman prototype may still focus on a single aspect of its eventual system, it may require relatively broad scope to provide a reasonable subset of the eventual system's behavior.

This point is worth emphasizing. An evaluative prototype may be able to answer a specific question about its eventual system without implementing even a trivial recognizable part of that system: for example, it may prove that a proposed algorithm works in the abstract, without producing anything that resembles the eventual system. But a strawman prototype must produce something similar enough to the eventual system to serve as a surrogate for learning about the eventual system in order to improve its design. Still, in order to be cost-effective, a strawman prototype must avoid becoming an initial system implementation by limiting its focus, scope, depth, and scale. It must limit its focus to a relatively narrow aspect of the eventual system, and it must limit its scope, or coverage, of the eventual system's behavior to the smallest subset that can serve as a useful surrogate for investigating the aspect under consideration. Similarly, it must limit the depth and scale of its implementation along both of these other dimensions. In general, a strawman prototype will be focused, relatively broad, fairly shallow, and fairly small scale.

Note that this implies that strawman prototyping is more appropriate for investigating some aspects than others. For example, it may be difficult to build a strawman prototype to focus on issues of integration or reliability, since it may be difficult to implement these aspects of a system in a broad yet shallow way.

2.4. *Prototyping to improve software development*

As previously stated, prototyping is often recommended as a way of improving the software development process. In particular, risk reduction strategies such as the spiral model recommend using prototypes to focus on specific questions that arise during design. This usually involves evaluative prototyping to reduce uncertainty, typically as part of the software engineering process.

As discussed above, the failure of the software engineering process has tended to overshadow the failures of the design and technology transfer processes. It is fairly obvious how evaluative prototyping can answer specific software engineering questions, and little more will be said about this. Yet evaluative prototyping is too narrow to be of much use in solving problems of

system design or technology transfer. Fortunately, strawman prototyping can be of great value in these areas.

Although the technology transfer process may appear at first glance to gain little from prototyping, this is only true if technology transfer is viewed as occurring after a system has been designed and implemented, which is a misconception. In fact, to be successful, technology transfer should begin during or before system design and continue throughout development. Prototyping can therefore improve technology transfer in two ways, one direct, the other indirect. Of these, the indirect effect is the more obvious: prototyping can improve the likelihood of successful technology transfer by improving the design of the eventual system. In particular, strawman prototyping can be used to give end-users a chance to generate feedback on proposed design decisions early in the development process. Though this improves the design process, it does so specifically by making the design more appropriate for the intended end-users: this has the important indirect effect of improving the technology transfer process by ensuring that the eventual system will serve the needs and desires of its end-users.

Perhaps less obvious is the fact that strawman prototypes can be used at all stages of development to give end-users direct (albeit simulated) experience with the eventual system by serving as its surrogate. Whether or not this results in useful design feedback, it is guaranteed to make end-users feel more involved in the development process. This in turn will give them greater appreciation of the eventual system's potential, greater understanding of its limits, and more realistic expectations about its utility.* Above all, this involvement will give end-users an invaluable sense of participation in the creation of the eventual system. This may be as important as any other factor in making the system successful: to the extent that end-users feel they have participated in its creation, they will accept psychological ownership of the system well before it is completed. When it is eventually delivered to them, they will be far more likely to accept it as their own offspring, rather than rejecting it as an intruder in their work environment.

Acknowledging this potential for improving technology transfer, the greatest value of prototyping lies in improving the design process. Since design is so crucial to every other aspect of software development, its failure dooms development, whereas anything that improves the design process improves all aspects of development. In the absence of prototyping, the design process must either be overly cautious (basing all decisions on proven experience), or it must rely on intuition (or inspiration) to generate new ideas that remain untested until the eventual system is delivered.† The former approach (being overly cautious) produces pedestrian systems that inherit the limitations of their ancestors; the latter approach (relying on intuition or inspiration) is fraught with risk. Prototyping allows intuition and inspiration to be tested and informed, thereby eliminating—or at least drastically reducing—their risk. In particular, strawman prototyping allows developers and end-users to try new ideas before committing to them, thereby greatly improving the design process.

Since evaluative prototyping is primarily useful for software engineering, whereas strawman prototyping holds great potential for improving both the technology transfer and design

*This can backfire, of course, if prototypes give users a bad impression of the eventual system, for example by being excessively unreliable or annoying to use.

†If implementation were cost-free, then design could proceed by producing, testing, and discarding full-blown systems at will.

processes, the remainder of this paper will focus on strawman prototyping. In particular, the design process will be examined in more detail to see how it can be improved by prototyping.

3. Conceptual Models

Whether or not prototyping is used to explore design choices prior to building an information system, the design process always begins with a conceptual model, i.e., a concept of why the system is needed, what it is for, who will use it, what it should do, etc. This conceptual model serves as the departure point for all that follows. It must exist prior to any specifications, since it is the basis on which the most preliminary specifications must be founded. In short, it is difficult to conceive of a design approach (except perhaps one based on Zen) that could proceed without a conceptual model. This section investigates the nature and role of the conceptual model in the design process.

Because a conceptual model is usually vague and implicit, it is rarely subjected to much scrutiny or analysis; yet an inappropriate conceptual model can quickly condemn a software development effort to failure. To substantiate this claim, it is necessary to ask what a conceptual model really is and how it determines the course of the development process. In this regard, it is enlightening to ask whether and in what sense it is really a model. Though this is discussed in further depth below, it is clear that any model must have an appropriate relationship to the reality which it purports to model. In the case of a conceptual model, what is this reality?

3.1. *The "target reality" of a conceptual model*

The current discussion is concerned with what has been described above as "targeted technology development" in which a system is developed to satisfy an identified need or desire. Before the need itself has been manifested or identified, there can be no conceptual model for a system: there is as yet no reason to consider building a system (or any other sort of solution) since no problem has yet been formulated. However, once a need is recognized within a particular environment, this need implies a problem; the desire for a solution to this problem provides the motivation for formulating a conceptual model of a system to solve the identified problem.[*] Of course, there may be other solutions to the problem that do not involve an information system at all; similarly, the mere existence of a problem does not guarantee that it will be recognized by anyone capable of or inclined toward conceiving of a software solution. On the other hand, several such people may become aware of the problem and form independent conceptual models of different systems to solve the same problem. Each such conceptual model is one of many possible models of a reality that includes the identified "target" environment, the relevant organization and end-user population within this environment, all associated constraints, and the needs and desires that suggest the creation of a new information system. This "target reality" can be conceptualized in various ways, each producing a different conceptual model.

[*] Untargeted technology development differs from this only in that the conceptual model is based on an imaginary "reality" instead of a real one; to the extent that it is *purely* imaginary, this makes validating the conceptual model against its reality a purely subjective (or possibly inter-subjective) matter.

It is important to recognize that this hypothesized “target reality” is a single, real thing, though it may be arbitrarily complex. Different conceptual models of this reality may view it in different or conflicting ways, some of which may be more valid or useful than others for certain purposes. Some conceptual models may embody important misconceptions about this reality, whereas others may model only incomplete subsets of it.

The assertion that the design process proceeds from a conceptual model can now be made more concrete. All initial design assumptions and decisions about the target reality for a proposed system—including the need and desire for the system itself—come directly from a conceptual model. If this conceptual model is incorrect or inappropriate, then the design process will be misconceived from its inception. What does it mean for a conceptual model to be incorrect or inappropriate? If it is to serve as the departure point for designing a system, a conceptual model must correctly model the appropriate aspects and constraints of the target environment, organization, and end-user population for a system, and must correctly identify and characterize the needs and desires that have led to the decision to build a system. To do this it must be a “valid” model of the target reality. This is discussed in greater detail in Section 4.

The conceptual model bears a heavy burden. Can a vague, intuitive concept satisfy these requirements? Won’t the conceptual model be discarded once the design process is begun and formal specifications are developed? How can the conceptual model be refined as the design process unfolds? How can it be validated in the first place?

It is true that as the design process develops formal specifications, these become a concrete embodiment of the conceptual model. Nevertheless, the very process of developing specifications must be guided by some idea of the purpose of the system and some understanding of the target reality. These must ultimately come from the conceptual model. As specifications for a system are evolved, they must constantly be validated against the target reality, and this must almost always be done by means of the conceptual model. Although it might in principle be better to validate specifications against the target reality directly, this is rarely practical, since the target reality is generally too complex and inaccessible to permit this. The conceptual model is therefore normally used in place of the target reality when designing a system. It may be argued that as the specifications for a system evolve, they will gradually take over the role of the conceptual model, and this is no doubt true to some extent. However, no matter how concrete specifications are, they will rarely be as complete as a conceptual model, even though it may be vague. The strength of a conceptual model lies in its ability to grasp much of the target reality in an abstract—though necessarily vague—way.

It must be admitted that one of the problems with a conceptual model is that it is abstract and therefore difficult to share. The convenient fiction adopted here is that a *single* conceptual model is used by a design team, though in practice each member of the team will have a different conceptual model. Short of telepathy, the best way to keep these conceptual models consistent is continuous communication among team members, ideally with automated support [2]. Alternatively, a single system “architect” may be given sole responsibility for maintaining *the* conceptual model for a design [8].

Since a correct and appropriate conceptual model is so important throughout the design process, it is vital to refine and evolve the conceptual model throughout this process. Furthermore, there

must be some way to validate the conceptual model, both initially and as it evolves. Prototyping offers a solution to these problems, providing direct feedback from the target reality.

3.2. *Prototyping to validate and refine a conceptual model*

As suggested above, the aspects of a system that are often the most deserving of prototyping effort involve functionality, user interface, system integration, reliability, and performance. These are all closely tied to the choice of an appropriate conceptual model.

For example, prototyping the functionality of a system requires a conceptual model of what the system should do; without such a conceptual model, a prototype of this sort cannot even be conceived. Conversely, the lessons learned from prototyping the functionality of the system can be used to validate or refine the conceptual model with respect to functionality. To the extent that users accept the prototype's functionality, this can be taken as validation of the conceptual model; additional feedback from users of the prototype (whether positive or negative) can be used to refine the conceptual model. Similarly, prototyping the user interface of a system requires a conceptual model of who the users will be and how they will expect to use the system. Feedback from such a prototype can validate and refine the conceptual model with respect to these aspects. Prototyping can even address technology transfer concerns by uncovering administrative, organizational, or user acceptance problems that might not otherwise become apparent until after delivery. The conceptual model of the end-user, target environment, and organizational aspects of the system can focus a prototyping effort to try to uncover such problems, and feedback from this prototype can validate and refine these aspects of the conceptual model.

In order for a prototype to validate and refine a conceptual model of the target reality, it must encounter that target reality directly. In most cases, this means that it must be used by target end-users, which implies that it must be a strawman prototype. In exceptional cases, evaluative prototypes may be designed for end-user interaction; for example, a specific hypothesis about a functional capability or an interface technique may be tried out on end-users by means of an evaluative prototype that has the minimum scope and depth needed to test the proposed behavior. In addition (as discussed above), certain aspects of a system, such as integration and reliability, may only be amenable to evaluative prototyping, since they may be difficult to implement in a broad yet shallow way. In such cases, an evaluative prototype may encounter the target reality without interacting with end-users; for example an evaluative prototype of system integration might encounter the target reality in the form of system interfaces, network protocols, etc. Nevertheless, strawman prototyping generally provides the greatest opportunity for end-users to produce useful feedback by experiencing a "preview" of the eventual system.

Although it may be preferable to prototype separate aspects of a system separately to keep each prototype cost-effective, there must be a single, unified conceptual model for a system, which is applied consistently to derive and evaluate all aspects of the proposed design. If prototyping one aspect of a design yields insight that refines or modifies the conceptual model with respect to that aspect, the resulting changes to the conceptual model must be carefully analyzed to understand their impact on all other aspects of the design. For example, suppose that prototyping the functionality of a document retrieval system reveals that users need to be able to add their own annotations to retrieved documents; this may require revising the user interface aspects of the

conceptual model to allow editing in addition to querying, as well as revising the system integration aspects of the conceptual model to provide access to the users' own files when composing annotations.

The discussion so far has focused on the relationships between the conceptual model, target reality, and prototypes. In order to show how these elements and relationships interact in the design process to produce an eventual system, the next section presents an expanded view of modeling.

4. Modeling

Most discussions of modeling distinguish two kinds of models, called **descriptive** and **prescriptive** [13, 17, 20]. Beyond this, models are often further categorized in many different ways, as I have discussed elsewhere [22]. However, models tend to be characterized from the perspective of the application areas in which they are used [14, 20, 24, 25], and no such categorization is generally accepted. Most of these categorizations provide little insight into the nature of modeling.

4.1. *The RPC-E interpretation of modeling*

I have suggested elsewhere [21, 22] that there are three criteria that define any model:

A **model** must

* *Refer* to some real-world **referent**,
* Have some intended (cognitive) *purpose* with respect to that referent, and
* Be more *cost-effective* to use for this purpose than the referent itself.

To **model**, then, is to represent a particular referent cost-effectively for a particular cognitive purpose. These three criteria will be referred to as **reference**, **purpose**, and **cost-effectiveness** and will collectively be called "the RPC-E criteria". Anything that meets these criteria can be considered a model of its referent[*] and will be said to obey the "RPC-E laws" and to be an "RPC-E model" (that is, a model in the RPC-E sense).

Note that whereas it is possible to *define* the term "model" in a completely arbitrary way, that is not the intent of this discussion; instead, the intent is to illuminate the term (and concept) of a model *as it is used*. In other words, this discussion is an attempt to provide a *model* of the concept of a "model". That is, since the concept already exists and is widely used, it can be considered a real-world referent to be modeled by the current discussion.[†] Since this discussion attempts to model the existing concept of a model, rather than presenting its own arbitrary definition, the discussion itself obeys the RPC-E laws stated above and can be viewed as an

[*]Verbal forms like "modeling" and "to model" are used here to denote the process of building and using models. These forms are sometimes reserved for the process of developing a model as distinguished from using one [15]; however, this precludes being able to say that a model "models" its referent, which seems too natural to be given up lightly.

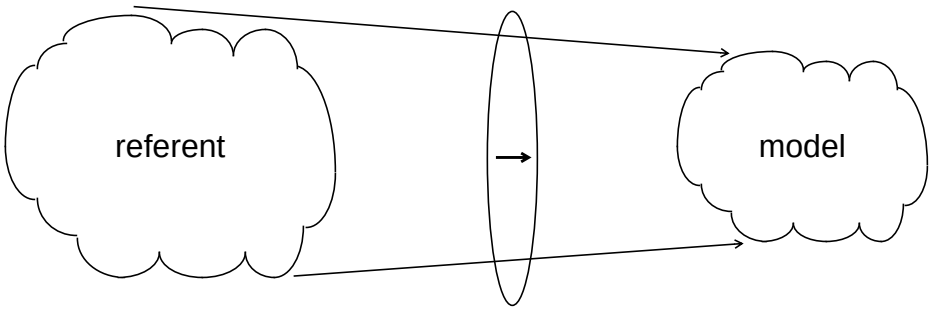

Figure 1: Modeling as projection

attempt to provide an RPC-E model of modeling. Since the phrase “model of modeling” may be confusing, this will be referred to as “the RPC-E interpretation” of modeling, emphasizing the fact that this is as much a description as a definition.

A model is **valid** if it cost-effectively fulfills its purpose with respect to its referent (according to its cost-effectiveness criterion). This is a very strong condition. It effectively requires that—to be a model at all—a model must be valid or at least include within its RPC-E criteria the conditions for its validation. The RPC-E criteria provide “falsifiability” for a model, allowing its validity to be tested and rejected; without this, the validity of a model would be purely a matter of conjecture. Given this possibility of rejection, a “bad” model is one which does not satisfy its RPC-E criteria very well, whereas a “non-model” (or “pseudo-model”) is something which does not even state its criteria or which states them but violates them flagrantly. A “potential” or “putative” model is something which states its criteria and *claims* to fulfill them without yet being able to demonstrate that it does so.

This conception of a model emphasizes its descriptive nature: models are conceived as describing some specific “real-world” referent. Though this referent need not be a concrete, existing thing, it must be objective and accessible enough to permit the model to be validated against it.

A model that has no referent is a contradiction in terms, or at best a meaningless abstraction. The notion of a referent and the relationship between a model and its referent is fundamental to the concept of a model. The RPC-E criteria define necessary and sufficient conditions for this relationship to be called a modeling relationship. The relationship of a model to its referent can be visualized as **projecting** the referent onto the model, as shown in Figure 1. A model without a referent cannot be visualized in this way: since there is nothing being projected in this case, the so-called model is completely undetermined and un-falsifiable—it can be anything at all, but it can serve no purpose.

[†]In this sense, the concept of a model is similar to the conceptual model of an information system discussed above, i.e., it is a vague, abstract idea about what a model is. However, the concept of a model is far more familiar than the conceptual model of any particular system.

The purpose of a model can be almost anything [10, 22]. Models are typically used for such purposes as understanding, appreciating, communicating, or predicting something about some real-world entity in situations where using the entity itself would be inconvenient, impractical, unsafe, or too expensive. The qualification that the purpose of a model must be *cognitive* is intended to distinguish modeling from the mere utilitarian (or "instrumental") use of something in place of something else. For example, using a rock in place of a hammer to drive nails should *not* be thought of as modeling the hammer, whereas using the rock to *understand* some aspect of the hammer (for example, how its mass or hardness allow it to drive nails) *is* an example of modeling. Note that the purpose of a model may be to manipulate reality, but this always denotes the cognitive aspect of manipulation. That is, the model itself does not manipulate reality, it simply provides the understanding of *how* to manipulate it. This is true even if the model is directly connected to an "effector" mechanism that manipulates reality with no human intervention: in this case, the model serves the cognitive purpose of deciding what should be done, and this decision is used by the effector to perform the actual manipulation. This is consistent with common usage. For example, an intelligent tutor program might decide how to respond to a student's query based on a model of what the student knows; here, the model would be used to decide how to respond, but the response itself would be generated by some other part of the program (a natural language generator or dialogue component) rather than by the model itself.

The cost-effectiveness criterion is rarely stated explicitly. The whole point of creating a model is to use it as a substitute for its referent for the given purpose. The motivation for this is that it must be more cost-effective to use the model for this purpose than it would be to use the referent directly (either because it is impossible, inconvenient, dangerous, or "expensive" to use the referent itself). In order for something to make sense as a model, it must satisfy some cost-effectiveness criterion with respect to its referent and its purpose.

4.2. *Prescriptive models*

The discussion so far has focused on *descriptive* models. What is its relevance to so-called *prescriptive* models? A prescriptive model is one whose purpose is to prescribe some desired (often optimal) state of the world in order to achieve some desired goal. This may appear to be quite different from a descriptive model, but the RPC-E criteria still apply: it is simply a model whose purpose is to find an optimum state or plan for achieving the desired goal.

Since a prescriptive model obeys the RPC-E laws, it must correspond to a real-world referent in order for its prescription to be meaningful and useful. For example, it cannot prescribe an impossible state or action, since this would not fulfill its purpose. A prescriptive model always begins as a descriptive model of some reality (its referent); it then uses this model to predict (and prescribe) a desired or optimum state for its referent. It might be argued that this prescriptive process extends beyond the RPC-E interpretation by arriving at a conclusion which is not a known consequence of the referent itself. However this is true of any prediction made by a model. One of the main purposes of modeling is to allow predicting unknown things about known referents. A prescriptive model is simply a descriptive model used to predict a desired or optimum state of the referent and/or to show how to achieve that state. Prescriptive models are therefore no different in kind from descriptive models: prescription is simply one possible purpose to which a descriptive model may be put.

4.3. *"Generative" models*

Now consider what appears to be a rather different use of the term "model". Prior to building a ship, for example, a small-scale model may be built to evaluate various aspects of the proposed ship, such as its hydrodynamics (using tank testing) or its aesthetics, or to help visualize, understand, and decide how to build the real ship. Similarly, the literal meaning of the word "prototype" is a "first example" or "archetype" of something; this implies a variation of the above example, in which a model serves as a first instance (which may or may not be small-scale). This might be called an "inventor's prototype", since this kind of model is often created to demonstrate an invention and to allow a production version to be built from the model.

A similar example is a blueprint for a house. As with the model ship, a blueprint can be used to understand and decide various things about a house before it is built, and it also serves as a plan for how to build the house. Models such as these are not just prescriptive: they are **generative**, in the sense that they are used in the process of building something (the ship, house, etc.).

What is the referent of a generative model? It may at first appear that the referent of a blueprint is the house it describes. If the blueprint is drawn from an *existing* house, then the blueprint is simply a descriptive model obeying the RPC-E laws; its purpose may be to understand how the house was built, to predict where modifications can be made, etc. In this case, the referent of the blueprint is clearly the existing house. However, in the usual—and more interesting—case, the house does not yet exist at the time the blueprint is created. The proposed house cannot serve as a referent for the blueprint, since the house does not yet exist. Yet if it is to serve the generative purpose of facilitating the building of the house, the blueprint cannot be drawn in arbitrary ways—it must conform to the constraints of building materials, civic codes, and construction practices (which might collectively be called "construction constraints"). Furthermore, it must represent some conception of the house that is to be built. Its referent, therefore, must be the *concept* of the house, embedded in the context of its construction constraints. This implies that the blueprint is a descriptive model whose referent is the concept of the house (including its construction constraints).

Note that a blueprint might be based solely on the concept of the house, ignoring construction constraints. In this case it would still be a model of the concept of the house, but it might fail in its generative purpose of facilitating the building of the house by failing to recognize real-world constraints. Such an unconstrained blueprint might be appropriate for some purposes: for example, it might appear in a fictional account of a fantasy world filled with impossible houses. The point here is that the blueprint is first of all a descriptive model whose referent is a concept of a house, whether or not that concept is realizable in the real world.

Nevertheless, the normal purpose of a blueprint is to facilitate building a house. The house is said to be built "from" or "according to" the blueprint. To satisfy this generative purpose, the blueprint must faithfully model not only the concept of the house itself, but also the construction constraints that apply to building it. These constraints may be considered part of the concept of the house, or the referent of the model may be viewed as consisting of two parts: the concept of the house plus the constraints. Either way, the referent of the blueprint as a generative model must include the combination of all these factors. It is useful to think of the "conceptual model" as including all these factors, where the concept of the house *per se* is only one part of the conceptual model.

A generative model can therefore be viewed as a descriptive model whose referent is itself a conceptual model. Its purpose is to facilitate generating a realization of some part of its referent in the real world. (Conceived of cognitively, this purpose means understanding and predicting how to build the house, *not* actually building it.) The cost-effectiveness criterion for a generative model is that it *facilitates* the realization of its referent; that is, it must be easier to realize the referent *with* this generative model than it would be *without* it. In most cases, this relies on the fact that a generative model is more concrete than the conceptual model that is its referent.

The RPC-E interpretation already encompasses generative models, but it is additionally useful to define the **objective** of a generative model as that which the model is intended to facilitate building. The objective is part of the *purpose* of a generative model, not part of the definition of the model itself, and it is important to avoid confusing the objective of a generative model with its referent. The objective is typically represented in some way in the referent; for example, in the case of the blueprint model, the objective (the house to be built) is represented in the referent (the conceptual model of the house including its construction constraints) by the concept of the house itself. A generative model can be thought of as a model *of* its referent intended *for* building its objective.

A generative model can be represented graphically as in Figure 2. As for any RPC-E model, the modeling relationship is shown as a projection from the referent onto the model, but in addition, a generative model *projects* part of its referent onto its objective.

This definition of a generative model allows formulating the design process in terms of modeling.

5. The Design Process As Modeling

The design process for targeted technology development begins with the recognition of a need in the real world. Under appropriate conditions, this leads to the formation of an initial conceptual model. This is a generative model whose referent is the identified target reality and whose objective is the eventual system to be developed. The conceptual model evolves throughout the design process. As it evolves, it must be continuously validated against the target reality. An intermediate objective of the conceptual model is a concrete specification for the

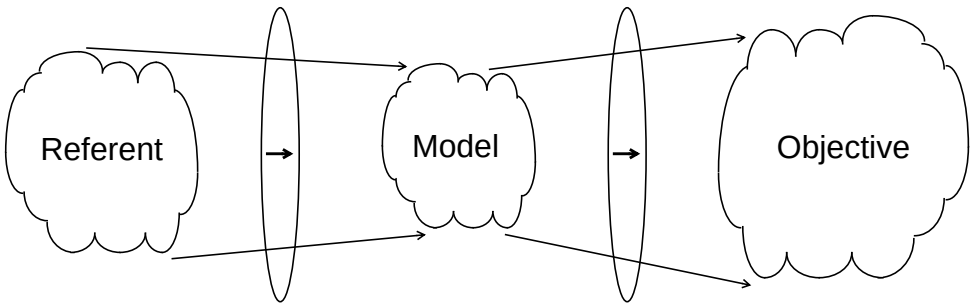

Figure 2: A generative model

eventual system; this evolves along with the conceptual model. This specification is itself a generative model whose immediate referent is the conceptual model (though its ultimate referent is the target reality, which it typically accesses via the conceptual model) and whose objective is again the eventual system. The design process consists of refining the conceptual model, using it to produce an evolving concrete specification, and implementing the eventual system according to this specification.

The conceptual model must be refined and validated with respect to the target reality, yet there are few ways of accomplishing this. The most obvious way is to build the eventual system and try it out, but this involves tremendous risk. The next best available technique is to use prototyping to "preview" selected aspects of the eventual system prior to building it.

A prototype is a partial, concrete realization of one aspect of a conceptual model; that is, it implements one aspect of the eventual system that is the conceptual model's objective. Though a prototype appears at first glance to serve as a model of the eventual system, it is more useful to think of it as a model of the conceptual model itself. In terms of the RPC-E interpretation, the purpose of a prototype (as a model) is to improve the understanding of the conceptual model and to validate and refine the conceptual model, in order to improve the eventual system. It is more cost-effective to use the prototype to validate and refine the conceptual model in this way than it would be to use the conceptual model directly, since the prototype can be "tried out" in ways that the conceptual model cannot. The prototype therefore satisfies the RPC-E interpretation of a model: it is used in place of the conceptual model because it is more cost-effective for its purpose, which is to understand, validate, and refine the conceptual model.

Evaluative prototyping can answer specific questions about the target reality, providing validation of certain aspects of the conceptual model. However, strawman prototyping has an even greater potential for refining and validating the conceptual model by means of direct end-user interaction. In both cases, the purpose of the prototype is to validate and refine the conceptual model.

A prototype can also be used as a partial specification for the eventual system. In this case, the prototype serves as a generative model and therefore has a different (or additional) purpose: its referent is still the conceptual model, but it now takes on the eventual system as its *objective*. This is analogous to the blueprint or "inventor's prototype" discussed above.

6. Conclusions

This discussion has attempted to analyze various aspects of software development in terms of modeling. In so doing, it has further developed and extended a view of modeling based on the criteria of Reference, Purpose, and Cost-effectiveness (here called the "RPC-E interpretation of modeling"). This interpretation suggests, among other things, that prototypes should be viewed as models of the conceptual model for an eventual system. This insight should help focus the prototyping enterprise and improve the understanding of the design process for computerized information systems.

Acknowledgments

The author wishes to thank Tora Bikson, Norm Shapiro, and Clairice Veit for their many invaluable insights on the subject of modeling.

References

1. Balzer, R., T. E. Cheatham, and C. Green, "Software Technology in the 1990's: Using a New Paradigm", *IEEE Computer*, November 1983, pp. 39-45.
2. Bigelow, J., "Hypertext and CASE", *IEEE Software*, March 1988, pp. 23-27.
3. Bikson, T. K., C. Stasz, and D. A. Mankin, *Computer-Mediated Work*, The RAND Corporation, R-3308-OTA, November 1985.
4. Bikson, T. K., B. A. Gutek, and D. A. Mankin, *Implementing Computerized Procedures in Office Settings*, The RAND Corporation, R-3077-NSF/IRIS, October 1987.
5. Boehm, B. W., "A Spiral Model of Software Development and Enhancement", *Computer*, Vol. 21, No. 5, May 1988, pp. 61-72.
6. Boehm, B. W., "Verifying and Validating Software Requirements and Design Specifications", *IEEE Software*, January 1984, pp. 75-88
7. Boehm, B. W., T. E. Gray, and T. Seewaldt, "Prototyping Versus Specifying: A Multiproject Experiment", *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 3, May 1984, pp. 290-302.
8. Brooks, F. P., *The Mythical Man-Month*, Addison-Wesley, December 1979.
9. Brooks, F. P., "No Silver Bullet: Essence and Accidents of Software Engineering", *Computer*, Vol. 20, No. 4, April 1987, pp. 10-19.
10. Davis, M., S. Rosenschein, and N. Shapiro, *Prospects and Problems for a General Modeling Methodology*, The RAND Corporation, N-1801-RC, June 1982.
11. DeMarco, T., *Structured Analysis and System Specification*, Prentice-Hall, 1979.
12. Fetzer, J. H., "Program Verification: The Very Idea", *CACM*, Vol. 31, No. 9, September 1988, pp. 1048-63.
13. Gass, S. I. and R. L. Sisson, *A Guide to Models in Governmental Planning and Operations*, U.S. Environmental Protection Agency, 1974.
14. Greenberger, M., M. A. Crenson, and B. L. Crissey, *Models in the Policy Process*, Russell Sage Foundation, NY, 1976.
15. House, P. W. and J. McLeod, *Large-Scale Models for Policy Evaluation*, Wiley, 1977.
16. McCracken, D. D. and M. A. Jackson, "Life cycle concept considered harmful", *ACM SIGSOFT Software Engineering Notes*, pp. 29-32, April 1982.
17. Meta Systems, Inc., *Systems Analysis in Water Resource Planning*, Water Information Center, Inc., 1971.
18. Meyer, B., *Object-oriented Software Construction*, Prentice Hall, 1988.
19. Page-Jones, M., *The Practical Guide to Structured Systems Design*, Yourdon Press, 1980.
20. Quade, E. S., "Modeling Techniques" in *Handbook of Systems Analysis*, H. J. Miser and E. S. Quade (eds.), North-Holland, 1985.

21. Rothenberg, J., “Object-oriented Simulation: Where Do We Go From Here?”, *Proceedings of the 1986 Winter Simulation Conference*, (Washington, D.C., Dec. 8-10), J. Wilson, J. Henriksen, and S. Roberts (eds.), The Society for Computer Simulation, San Diego, CA, 1986, pp. 464-469.
22. Rothenberg, J., “The Nature of Modeling”, in *Artificial Intelligence, Simulation, and Modeling*, L. Widman, K. Loparo, and N. Nielsen (eds.), John Wiley and Sons, Inc., August 1989, pp. 75-92.
23. Royce, W. W., “Managing the Development of Large Software Systems: Concepts and Techniques”, *Proceedings of the Western Electronic Show & Convention*, August 1970, Session A/1, pp. 1-9.
24. Specht, R. D., “The nature of models” in *Systems Analysis and Policy Planning: Applications in Defense*, E. S. Quade and W. I. Boucher (eds.), Elsevier, 1968.
25. Walker, W. E., J. M. Chaiken, and E. J. Ignall (eds.), *Fire Department Deployment Analysis*, North-Holland, 1979.

About the author

The author is a Senior Computer Scientist in the Information Sciences Department of The RAND Corporation, where he pursues research in knowledge-based simulation and modeling.