SPECIFYING AND PROTOTYPING: SOME THOUGHTS ON WHY THEY ARE SUCCESSFUL

{Daniel M. Berry[1] , Jeannette M. Wing[2]}

Computer Science Department
University of California
Los Angeles, CA 90024

Computer Science Department
University of Southern California
Los Angeles, CA 90089

U. S. A.

Abstract

Two methods that have been successful in producing good software are 1) specifying and then implementing and 2) prototyping and then implementing. This paper identifies what the two methods have in common, namely that the implementation is the second time through carefully thinking about the problem. It proposes that perhaps this common aspect is more important to the successes of the methods than other aspects of the methods.

1. Introduction

1.1. Our Background

We both work actively in the fields of specification and verification. In addition to doing research in these fields, we both consult for a company, SDC, which specializes in implementing secure systems by first formally specifying them, then verifying that the specifications meet some of the desired properties, and then finally implementing the systems. We both also have implemented software by first prototyping it and then implementing it a second time.

1.2. Purpose of this Paper

The purpose of this paper is to promote some needed discussion of the reasons why projects are successful *when* they are successful*. Since it is difficult to conduct controlled experiments on such projects, our conclusions are at best conjectures. We hope that we can provoke sufficient debate so that more accurate conclusions can be reached by a consensus of the actual workers in such projects. Regardless of what the ultimate conclusions, the thoughts presented in this paper impact the choice of methods for software development, project management, and tool development. In this paper, we are thinking in print with hopes that the community joins us.

1.3. Our Claim

We have observed a number of successful software development projects. Some were developed by a method that we call *specifying*, some others were developed by a method that we call *prototyping*, and still others were developed by other methods which are not discussed here. These terms are defined in more detail in the next section. For now, it suffices to say that in each method, the method is named by what is done in the first stage. In this first stage the product of the method, either a specification or a prototype, is thoroughly checked with the help

[1] This work was supported in part by the University of California MICRO Program, SDC, A Burroughs Company, and NCR Corporation.

[2] This work was supported in part by the National Science Foundation under Grant No. ECS-8403905.

* Note that we are not attempting to determine why projects fail. We hope that in determining why successful projects succeed, the ideas can be applied to increase the probability of success in all projects. Examination of the reasons for project failure is also necessary, but space simply does not permit it.

of automated tools and possibly even executed. The second stage in both is the implementation stage, in which what is learned in the first stage is applied to produce a suitable production version of the software. What is common to the two methods is what we call the ***second time phenomenon***. That is, the delivered, production quality software is a second pass through the problem which follows a formally stated and machine-checked first pass. In the specifying case, the specification is formal and it can be checked by verifying, with the aid of a verification program, that it meets desired properties and that it is consistent. In the prototyping case, the prototype, i.e., the program, is formal, and it can be checked by a compiler or interpreter, the run-time environment, and the users when running the program.

We wonder if the most important factor in the success of these projects is the fact that the delivered software is a second pass after a formal, machine-checked first pass. That is, we wonder if this second time phenomenon is more critical to the successes than any other factors arising from the particulars of specifying and prototyping *per se*.

1.4. Outline of Rest of Paper

In Section 2, we clarify what we mean by the terms "specifying" and "prototyping." In Section 3, we elaborate on our claim by focusing on the similarities between the two methods and enumerating some successful applications of both methods. In Section 4, we explore the implications of our claim as it impacts on software methods, languages, tools, and project management. In Section 5, we present arguments that favor one method over the other by focusing on some differences between the two. Finally, in Section 6, we briefly state our conclusions.

2. Clarification of Terms

The two methods, *specifying* and *prototyping*, both start with informal requirements and have two major development stages. They have the same second stage, which is the implementation stage, and differ only in their respective first stages. In this section, we define what we mean by the terms "specifying" and "prototyping" by describing what happens in their first stages. These definitions are important because if any ingredient is left out, then the implementation cannot rightfully be called a second formal and machine-checked pass through the problem.

By *specifying*, we mean the following process:

1. writing a formal specification of the proposed software using some precisely defined and machine-processable specification language such as Affirm [Aff81], Gypsy [Gyp78], Ina Jo® [Ina80], Larch [GH83], and SPECIAL [Hdm79], and
2. checking this specification with the aid of its language processor and other tools. This checking includes as much of the following as possible
   a. syntax checking,
   b. type checking,
   c. verifying that the specification is formally consistent,
   d. verifying that the specification meets stated correctness criteria such as invariants, and
   e. (possibly) exercising the specification on actual or symbolic data with the help of a symbolic evaluator, such as UNISEX for the Ina Jo language [KE83] and the symbolic evaluator of McMullin and Gannon [McG83].

   Typically, the first two of these checks are done by the language's processor. The conjectures for the two verification checks are generated by this processor, and the conjectures are proved to be theorems with the aid of an associated, possibly interactive, theorem prover.

By *prototyping*, we mean the following process:

1. writing a first version of the software and bringing this version to a running state using an implemented pro-

®Ina Jo is a trademark of SDC, A Burroughs Company.

gramming language, which may be different from that used to write the production version,
2. checking this first version with the language's processors and tools, and
3. subjecting this first version to the end-users' acceptance tests.

By *first version*, we include also possibly incomplete versions written for exploration and experimentation by the programmers and clients [Flo84].

The checks done during the second and third steps of this process include syntax checking, type checking, and interface checking, and run-time checking. The first three checks are typically done by a compiler of the language and the fourth done with the code generated by this compiler perhaps in conjunction with a special debugging run-time system. Alternatively, these checks may be done by an interpreter of the language.

Observe that a programming language is a formal language; it has precise syntax and semantics just as any other language more traditionally considered to be a specification language. In the same light, it is clear that a program is just as much a formal statement of an algorithm as is a more traditional first-order predicate calculus specification of the algorithm.

3. Elaboration of Our Claim

3.1. Similarities Between Specifying and Prototyping

Close examination of the two methods shows that they have much in common and one wonders if what they have in common is the major reason for their success. In both cases, one must write a complete formal description of the system before beginning to code the system in its deliverable, production form. In the specifying case, the formal description is written typically in a first-order predicate calculus language, in a set theoretic language, or in an algebraic framework. In the prototyping case, the formal description is the first implementation, possibly in a language other than the production version language, i.e., in a so-called very high level language. Thus, in either case, one crucial result is a formal description of the software.

In both cases, the formal description is then subjected to a thorough battery of machine checks. These include syntax and type checking. These include interface consistency checks. In addition, in the specifying case, these may include the generation and subsequent verification of theorems that assert the consistency of the specification and that it meets stated requirements. These checks may also include execution with test data with the aid of a symbolic evaluator. In the prototyping case, the program is run with test data. Besides this testing against the data, given a suitable language implementation, the run-time system also performs a number of run-time semantic tests such as checking that variables are initialized before use, subranges and array bounds are observed, nil pointers are not dereferenced, etc. In addition, there may be symbolic evaluators, execution tracers, snapshot generators, etc. that allow the testers to observe the details of the program's execution. In either case, by the time the specification or prototype is accepted as done, the writers have had to eliminate many, many bugs and to iron out many, many wrinkles.

These machine-aided tests are crucial. They help to eliminate conceptual errors in the understanding of the problem that lead to serious design flaws. Anyone who has written a specification or prototype to completion knows how picky the machine tests are. Anyone who has written one of these without the benefit of machine processing knows how easy it is to handwave one's way into overlooking major design flaws and major processing errors. Machine-processing and checking help prevent cheating.

Furthermore, formality of the language used in specifying or prototyping is critical. If the language were not formal, then it could not be machine-processed and checked. The language's semantics would remain sufficiently fuzzy to permit human ambiguity. This ambiguity is useful for human-to-human contact, but is potentially disastrous to the completion of a software project.

Therefore, with either method, the writing of the production version of the software, i.e., what is done in the second stage of each method, constitutes a second pass through the problem, in which the first pass has had the purpose of finding many, if not all, of the tricky corners of the problem. We believe that *the fact that this is the* *second pass through the problem* is more important to the success of software projects than in what language the first pass was written and whether the machine processing involved proving theorems or executing the program on test data.

3.2. *In Support of Our Claim*

To substantiate our claim, we list in this section some successful examples of specifying and prototyping. In addition to these examples of successful projects, some documented "folklore" also lend support to our observation of the *second time phenomenon* and the commonality between specifying and prototyping. Many books on software engineering, e.g., [CL76, LHN79, KP74], admonish the programmer not to be afraid to throw programs out and start all over. Brooks [Bro75] even suggests *planning* to throw early versions away.

3.2.1. *Some Successful Applications of Specifying*

The successful projects that have used specifying as its method include the LSI Guard done at I.P. Sharp [Sta81], the COS/NFE project done at Compion [SW82], the SCOMP project done at Honeywell [Fra83], the SIFT project done at SRI [MS82], the Message Flow Modulator done at Austin [GSS82], the Secure Release Terminal done at SDC [HAK83], and the signalling system done at the General Electric Company [CCI81]. In each case the system was formally specified and the resulting specification was verified to meet its requirements with the help of an automatic or interactive theorem prover. The system was then implemented and is now running. Landwehr has a longer list of all such projects, successful and not so successful [Lan83].

3.2.2. *Some Successful Applications of Prototyping*

The successful prototyping projects are too numerous to list completely and include the following well-known (at least to the authors) examples: the UNIX® operating system done at Bell Labs and at UC Berkeley [Unix], the Device Independent TROFF done at Bell Labs [Ker82], the Cedar system at done at Xerox Parc [Tei84], the REVE term rewriting system generator done at MIT and the University of Nancy [Les83, FG83], the Affirm specification and verification system done at USC's Information Sciences Institute [Aff81], the EMAS operating system done at the University of Edinburgh [SRSY77, SYRS80, RS82], and the S-port portable version of SIMULA done at the Norwegian Computing Center [NCC??]. In each case the current delivered version of the software is at least the second, or is built based on experiences with at least one other, earlier system.

Further discussions of prototyping may be found in the *Software Engineering Notes* issue containing the working papers submitted to the ACM SIGSOFT Rapid Prototyping Workshop [Pro82]. Of relevant interest is the healthy dosage of papers relating prototyping with specifying, including those on executable specifications which are therefore prototypes [Smo82, GM82, Dav82, Fea82, BGW82] and the process of prototyping specifications [Mac82, KK82, HH82]. Also, the recent proceedings, *Approaches to Prototyping* [BKMZ84], thoroughly explores many aspects of prototyping. Of relevant interest, the first article by Floyd [Flo84] attempts to reconcile the wide variety of views as to what is prototyping; in all of these views of prototyping, the prototype is a first version of at least a two-version progression.

3.3. *Qualifications to Our Claim*

We do not mean to imply that two times is necessary or even sufficient for success. There are and will be first-time projects done well and there are and will be second-time projects done poorly even if the first pass is done well. All we are doing is trying to identify the major reason for success in the two methods and the cited projects.

We acknowledge that the perception of success via either specifying or prototyping may be more psychological and attributable to learning than anything else. For instance, some successful applications of prototyping may not even have started off as attempts to prototype. They may have been projects in which the software was done twice because the first effort, though satisfactory, was not perfect and because other desired enhancements were

®UNIX is a trademark of AT&T Bell Laboratories.

discovered or requested. Similarly, in [GHW82], the authors state that the process of specifying, i.e., understanding and learning about the problem to be specified, is at least as or often more beneficial than having the resulting specification.

4. Implications of Our Claim

This section describes some implications of accepting the validity of our claim. These are offered to explain the methodological impact of the claim.

4.1. *What Should Not Work*

The reader should correctly infer that we do not believe that a first pass that is done without the benefit of machine-processing is likely to lead to as successful an end-product. Thus, for example, the use of non-processed specifications or specifications involving second-order logic (which is not processable) should be discouraged. Likewise, we discourage prototyping that is too rapid or haphazard. More specifically, the following two activities are less useful than the activities of specifying and prototyping as defined in Section 2.

1. Using non-machine checked formal specifications. A non-machine checked formal specification is as good as an informal specification. Although both may be useful pieces of documentation, to prevent the specifier from "cheating" and to ensure at least the consistency of the specification, it is more valuable to use and rely on machine-checked specifications. We maintain that the process of specifying is still valuable, whether or not the product is eventually checked. With the proper social processes [DLP79], i.e., many people carefully poring over such specifications, these too can lead to successful projects, e.g., the Ada® compiler done with the help of the Vienna Definition Method [CO84]. A specification, however, must be written at least with the intention that it be checked, and ideally with tools that help perform the checks.
2. Rapid prototyping when done too haphazardly or with the intention of throwing out versions. Probably no one would advocate non-systematic approaches to software development. A prototype should be written with as much attention paid to good planning and design as that paid in any implementation effort, even if it is not intended to be the final version. Making modifications made to a well-designed prototype should go faster than making them to a poorly-designed one. Also, systematic planning and recording of what modifications are made at each stage can speed up the entire process itself.

4.2. *What Should Work: Combining Specifying and Prototyping*

4.2.1. *Methods*

The traditional software life cycle method includes one or more specification phases in which specifications in varying degrees of formality are written. In practice, however, rapid prototyping is a method often used to get a working system up quickly. According to our claim these two methods are not incompatible since one can view writing a formal specification as writing a first prototype. Instead of choosing one method over another, it may prove beneficial either to follow both in parallel or to interleave the two activities and to compare intermediate results at their intersecting formal specification/first prototype step.

4.2.2. *Languages*

In order to support specifying and prototyping as compatible activities, the problem of which languages to use arises. There are three languages for which choices must be made: the specification language, the prototyping language, and the (eventual) implementation language. One must choose which particular language to use for each and whether any should be the same. These decisions can greatly influence the speed and cost of software development.

® Ada is a trademark of the U. S. Department of Defense (AJPO).

Traditionally, specification and implementation languages were different, but currently the distinction between the two is blurring. Executable specification languages, such as OBJ [GT79] and GIST [BGW82], can be used as high-level programming languages. Very high level programming languages, such as SETL [KS84], and applicative programming languages, such as Prolog [CM81] and FP [Bac78], can be used as specification languages. More consistent with our claim, however, is to use an executable specification language as a prototyping language, though not necessarily as the eventual implementation language. Thus, specifications must be executable, a viewpoint which many designers of specification languages currently advocate [GM82, Zav84, Orl84], or at least be subject to machine-aided semantic checks [GHB3]. Furthermore, if the specification and prototyping languages are chosen to be the same, but different from the implementation language, the transition from the specification/prototyping language to the implementation language must still be made.

4.2.3. Project Organization

Given that methods and languages overlap, one needs to rethink how to organize a project to obtain reliable and correct software as efficiently and economically as possible. Most software projects are organized along a traditional life cycle approach. If prototyping is to be accepted as a viable parallel activity or otherwise somehow integrated into the life cycle approach, then guidelines for managing, controlling, and budgeting for prototyping need to be made with the same concern as for the other activities in the life cycle. In any project in which reliability, security, safety, correctness, or user-friendliness is important, i.e., all but the most trivial or private programs, the project must be organized and budgeted to allow for the two times through the problem.

5. Differences Between Specifying and Prototyping

So far, we have focused on the commonality between the two activities of specifying and prototyping as ways of systematically developing software. Now, in order discharge our duty to attempt to find an *absurdum* of an unverifiable hypothesis, we will discuss some of their differences. We have made the distinctions between specifying (the activity) and a specification (the product of specifying), and between prototyping and a prototype. The important differences between specifying and prototyping are not differences between performing the two different activities, but are differences between the products of having performed them, i.e., specifications and prototypes, and their intended uses.

As stated in Section 1.2, we wish to promote some needed discussion on why specifying and prototyping are both successful methods. To provide grounds for further discussion, in this section we examine arguments first in favor of specifying and then in favor of prototyping. The arguments for specifying are the traditional ones propounded by the specification community; those for prototyping are taken from conclusions from documented experiments, which were conducted to compare specifying and prototyping. For each of the arguments, we propose counterarguments to show how the argument either does not hold in practice or presumes definitions of specifying or prototyping different from those we gave in Section 2.

5.1. In Favor of Specifying

5.1.1. Main Argument

5.1.1.1. Specifications are Independent of Implementations

A specification is written independently of any of its implementations. Consequently, specifications serve more easily than prototypes do for the following two uses. First, a specification is a contract between a user and an implementor. On one side of the contract, a user need be concerned with only the specification and not with any of its possible implementations. In principle, to understand the behavior of an implementation, the user need look only at its specification and neither look at the implementation nor execute it. On the other side of the contract, the implementor need be concerned with only satisfying a specification without any knowledge about any of the users of the implementation. This argument holds for whether a user is a person or another piece of software.

The second use is that a specification is a common reference point among the several implementors on the same project. This use is especially important for large software projects where implementation work is divided among a team of programmers and a specification is composed of specifications of pieces of the software. Each member of the team is thus concerned only with implementing and maintaining a piece of the software and making sure that it satisfies a piece of the entire specification.

5.1.1.2. A Counterargument

On the other hand, the above advantages can be ascribed to a well-written program that is taken to be a specification. The point is that the above main argument presumes the following three beliefs:

1. It is better to understand the behavior of a system by reading a specification of its behavior than by running the system.
2. It is easier to understand the behavior of a system by reading a specification than by reading an implementation, i.e., the program text.
3. Specifications are written in a more abstract manner than are implementations.

The first belief is always true under circumstances in which running the system is either dangerous or prohibitively expensive. Examples of such systems are real-time systems with potential consequences of loss or destruction of life. It is also true under circumstances where the user of a system is concerned with the properties a system guarantees and not with the operational details of the system. Examples of such properties are security [HAK83], reliability [MS82], and performance [Zav82]. This first belief, however, is not necessarily true in other circumstances. There may be problem domains in which it is better to run the system than to read its specification because it would be easier for the user to understand the system by observing its behavior than by reading about it. For example, users may rather run a series of acceptance tests over systems that are heavily dependent on human interaction, e.g., interfaces to text editors, database query systems, or CAD/CAM systems, instead of (or in addition to) reading their specifications.

The second belief is not always true. Many specifications are hard to read because of the language they are written in, their size, or lack of machine support to aid in understanding them. Conversely, many implementations are easy to read because of the language they are written in, their design (their modular decomposition and proper choice of data types), and richness of machine support (structured editors, libraries).

Similarly, the third belief is not always true. An unskilled specifier may expose implementation details or make premature design decisions in his or her specification. Conversely, a highly skilled programmer who makes use of the abstraction power of the implementation language may hide implementation details in order to make the resulting software more easily modified [Par72, Mye79].

5.1.2. Three Other Arguments

One: One can deduce properties of a system from a specification without running the system. As argued previously, there are some circumstances in which one cannot or should not run a system. Also, one may want to check for desired (or undesired) properties of a system even before implementation has begun in order to see how design decisions interact. In both cases, a specification can be used to derive properties that may or may not have been originally desired by the user. Such feedback can be used to accept or reject the system, or it may induce a change to the system or to the specification itself [Win84]. In practice, however, only a few experiments have been performed on non-trivial examples where non-trivial properties were derived [Hen79, MS79, GH80]. In some cases, it is even difficult to know what properties one might want to deduce [Win80]. Finally, this difference between specifications and prototypes can be counted as an advantage only if the specification is small enough to do proofs by hand or if there is sufficient software support to do proofs by machine.

Two: A specification is intended to be a consistent and complete description of a system whereas a prototype is intended to be an approximation of the eventual system. This difference points out that a prototype is typically written with intentional incompletenesses in mind and even with intentional inconsistencies. A prototype may leave the implementation of certain features of the desired system for future versions; it may even differ in contradictory ways from the final system. This difference between a specification and a prototype counts as an advantage for specifications if one views a specification as system documentation. A prototype that is incomplete or inconsistent cannot be used as a reliable piece of system documentation. However, at least for the reason of consistency, we exclude this sort of prototype as a basis for the method we described. Furthermore, a specification can easily be written, perhaps intentionally, with the same incomplete coverage as the above described prototype. Thus again, the argument boils down to how the specification or prototype is actually written.

Three: Specifications can be written in a language that is more abstract than any programming language and that is possibly non-executable. Not bound to traditional programming languages, a specifier is free to write specifications with assertions that take full advantage of the power of set operations or that quantify over infinite sets. The properties of expressibility and understandability should guide the choice of a specification language. The various branches of mathematics provide their own languages for expressing properties of their systems. Hence, mathematics provide a rich set of languages from which to base specification languages. On the other hand, mathematical languages are not easily understood, especially by those not trained in mathematics. Educating non-mathematicians* in specification languages remains an important practical problem. In addition, a skillful prototyper can modularize an implementation using the same high level abstractions with well-named functions so that its structure is identical to that of a highly abstract mathematical specification.

5.2. *In Favor of Prototyping*

In order to argue in favor of prototyping and to present counterarguments, we begin by summarizing the main results of two experiments documented in the literature.

Experiment 1: Gray, Boehm, and Seewaldt [BGS84] describe an experiment comparing prototyping to the more traditional life cycle approach, which includes specifying the software; this specifying, however, may be only informal. The purpose of their experiment is to determine which method produces the best software. They had several student groups apply the two methods to the same software product development. They found that prototyping, as opposed to specifying,

1. tends to produce a smaller product with roughly equivalent performance with less effort,
2. tends to produce higher equivalent user satisfaction per person-hour but lower delivered source instructions per person-hour,
3. tends to produce better human-machine interface, continual availability of a running version, and reduced deadline effects,
4. tends to produce software that is perceived to be easier to maintain, but
5. tends to lead to less planning and more testing and fixing, and to more difficult integration.

Experiment 2: Alavi [Ala84] describes the results of interviewing groups of users and systems analysts to determine which method was more satisfying. She finds that prototyping, as opposed to the more traditional life cycle methods, is perceived to facilitate better communication between users and designers during the design and implementation of the system and thus to facilitate better utilization of the system by the users. The users appeared more satisfied with the accuracy and the helpfulness of the output of a prototyped system than of one developed by life cycle methods. She concludes, however, that in designing innovative systems with fuzzy, not well-understood requirements, prototyping should be done by skilled people. She also concludes that in some cases prototyping may not be useful.

Boehm reaches this last conclusion on economic grounds [Boe81]. He argues that for well-understood problems, it is wasteful to prototype.

* Even mathematicians trained in one branch, e.g, numerical analysis, require additional training to understand languages used in other branches, e.g., number theory.

While the above described experiments clearly show the benefits of prototyping§ the results are strictly speaking not applicable to our definitions of the terms. First, the specifying or traditional life cycle method tested does not necessarily include machine-checkable formal specifications. Second, neither experiment is carried out through the second stage of the prototyping, i.e., when a production version is produced. It would be interesting to follow up these experiments up with this consideration.

Finally, Brooks points out one advantage of prototyping that may not be obtainable if specifying is done: when something gets running, the morale of the workers goes sky high. We have observed, however, the same jump in morale when an arduous effort in specifying finally results in a verified and consistent specification.

6. Conclusions

To do a good job on any large complicated project, one must understand the problem – a lack of understanding can lead to catastrophic failures. One is more likely to obtain this requisite understanding when building a "complete" model of the intended system, e.g., a formal specification or a prototype, than when simply handwaving through an incomplete design. The importance of machine-checking in this process is that it helps to ensure that one's model is complete and to prevent overlooking or ignoring tricky details. Thus, when going through a problem the second time, one can take advantage of the knowledge gained from having already gone through a formal understanding of the problem once.

We have attempted to expose the issues while suggesting our own favored conclusions. Of course, until some controlled experimentation can be done to test our hypotheses, the conclusions can only be conjectures. We hope that the conjectures are convincing and that this discussion has promoted useful debate.

Acknowledgements

The authors have benefited from discussions with Orna Berry, Myron Hecht, Tom Hinke, Nancy Leveson, and Ben Livson.

Bibliography

[Ala84] Alavi, M., "An Assessment of the Prototyping Approach to Information Systems Development," *CACM* 27:6, June, 1984.

[Bac78] Backus, J., "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," *CACM,* 21:8, Aug., 1978.

[BGW82] Balzer, R.M., Goldman, N.M., and Wile, D.S., "Operational Specification as the Basis for Rapid Prototyping," "Special Issue on Rapid Prototyping, Working Papers from the ACM SIGSOFT Rapid Prototyping Workshop," *SOFTWARE ENGINEERING NOTES* 7:5, Dec., 1982.

[BGS84] Boehm, B.W., Gray, T.E., and Seewaldt, T., "Prototyping vs. Specifying: A Multi-Project Experiment," *Proceedings of the Seventh International Conference on Software Engineering,* Orlando, FL, May, 1984.

[Boe81] Boehm, B.W., *Software Engineering Economics,* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[Bro75] Brooks, F.P., Jr., *The Mythical Man Month, Essays on Software Engineering,* Addison-Wesley, Reading, MA, 1975.

[BKMZ84] Budde, R., Kuhlenkamp, K., Mathiassen, L., and Züllighoven, H. (Eds.), *Approaches to Prototyping,* Springer-Verlag, Berlin, 1984.

[CCI81] CCITT, "Specifications of Signalling System No. 7," *Yellow Book, VI,* Fascicle VI-6, Recommendations Q.701-Q.741, 1981. Referenced in [Orl84] by J. Woodcock.

§ In some cases the benefits are only perceived, but the perception is the benefit.

[CL76] Chmura, L.J. and Ledgard, H.F., *COBOL with Style, Programming Proverbs*, Heyden, Rochelle Pk., NJ, 1976.

[CO84] Clemmensen, G.B. and Oest, O.N. "Formal Specification and Development of an Ada Compiler — A VDM Case Study," *Proceedings of the Seventh International Conference on Software Engineering*, Orlando, FL, May, 1984.

[CM81] Clocksin, W.F., and Mellish, C.S., *Programming in Prolog*, Springer-Verlag, Berlin, 1981.

[Dav82] Davis, A.M. "Rapid Prototyping Using Executable Requirements Specification," "Special Issue on Rapid Prototyping, Working Papers from the ACM SIGSOFT Rapid Prototyping Workshop," *SOFTWARE ENGINEERING NOTES* 7:5, Dec., 1982.

[DLP79] De Millo, R.A., Lipton, R.J., and Perlis, A., "Social Processes and Proofs of Theorems and Programs," *CACM*, 22:5, pp. 271-280, 1979.

[Fea82] Feather, M. "Mappings for Rapid Prototyping," "Special Issue on Rapid Prototyping, Working Papers from the ACM SIGSOFT Rapid Prototyping Workshop," *SOFTWARE ENGINEERING NOTES* 7:5, Dec., 1982.

[Flo84] Floyd, C., "A Systematic Look at Prototyping," in [BKMZ84], pp. 1-18, 1984.

[Fra83] Fraim, L.J. "SCOMP: A Solution to the MLS Problem," *Computer* 16:7, July, 1983.

[FG83] Forgaard, R., and Guttag, J.V., "REVE: A Term Rewriting System Generator with Failure-Resistant Knuth-Bendix," in *Procceedings of an NSF Workshop on the Rewrite Rule Laboratory*, September, 1983, J.V. Guttag, D. Kapur, and D.R. Musser, (editors), available as a General Electric Technical Report No. 84GEN008, April, 1984.

[GM82] Goguen, J.A. and Meseguer, J., "Rapid Prototyping in the OBJ Executable Specification Language," "Special Issue on Rapid Prototyping, Working Papers from the ACM SIGSOFT Rapid Prototyping Workshop," *SOFTWARE ENGINEERING NOTES* 7:5, Dec., 1982.

[GT79] Goguen, J.A., and Tardo, J., "An Introduction to OBJ: A Language for Writing and Testing Formal Algebraic Program Specifications," *Proceedings Conference on Specifications of Reliable Software*, Boston, 1979.

[Gyp78] Good, D.I., Cohen, R.M., Hoch, C.G., Hunter, L.W., and Hare, D.F., "Report on the Language Gypsy, Version 2.0," Tech. Report ICSCA-CMP-10, University of Texas, Austin, Sept., 1978.

[GSS82] Good, D.I., Siebert, A.E., and Smith, L.M., "Message Flow Modulator," Final Report, Institute for Computing Science TR-34, University of Texas, Austin, Dec., 1982.

[GHW82] Guttag, J.V., Horning, J.J., and Wing, J.M., "Some Notes on Putting Formal Specifications to Productive Use," *Science of Computer Programming*, 2:1, Oct., 1982.

[GH83] Guttag, J.V., and Horning, J.J., "An Introduction to the Larch Shared Language," *Proceedings IFIP Congress 1983*, Paris, 1983.

[GH80] Guttag, J.V., and Horning, J.J., "Formal Specification as a Design Tool," *Proceedings Principles of Programming Languages Conference*, Las Vegas, 1980.

[Hen79] Heninger, K.L. "Specifying Software Requirements for Complex Systems: New Techniques and Their Application," *Proceedings Conference on Specifications of Reliable Software*, Boston, 1979.

[HAK83] Hinke, T., Althouse, J., and Kemmerer, R.A., "SDC Secure Release Terminal Project," *Proceedings of the 1983 Symposium on Security and Privacy*, Oakland, CA, April, 1983.

[HH82] Hooper, J.W. and Hsia, P., "Scenario-Based Prototyping for Requirements Identification," "Special Issue on Rapid Prototyping, Working Papers from the ACM SIGSOFT Rapid Prototyping Workshop," *SOFTWARE ENGINEERING NOTES* 7:5, Dec., 1982.

[Orl84] *International Workshop on Models and Languages for Software Specification and Design*, Orlando, Florida, Workshop Notes, R.G. Babb and A. Mili, editors, Département d'Informatique, Université Laval, Québec, DIUL-RR-8408, March, 1984.

[KE83] Kemmerer, R.A. and Eckmann, S.T., "A User's Manual for the UNISEX System," Department of Computer Science, UCSB, Santa Barbara, CA, Dec., 1983.

[Ker82] Kernighan, B.W., "A Typesetter-independent TROFF," Computing Science Technical Report No. 97, Bell Laboratories, Murray Hill, NJ, 1982.

[KP74] Kernighan, B.W. and Plauger, P.J., *The Elements of Programming Style*, McGraw-Hill, New York, 1974.

[KK82] Klausner, A. and Konchan, T.E., "Rapid Prototyping and Requirements Specification Using PDS," "Special Issue on Rapid Prototyping, Working Papers from the ACM SIGSOFT Rapid Prototyping Workshop," *SOFTWARE ENGINEERING NOTES* 7:5, Dec., 1982.

[KS84] Kruchten, P., and Schonberg, E., "The Ada/Ed System: A Large-Scale Experiment in Software Prototyping Using SETL," in [BKMZ84], pp. 398-415, 1984.

[Lan83] Landwehr, C.E., "The Best Available Technologies for Computer Security," *Computer* 16:7, July, 1983.

[LHN79] Ledgard, H.F., Hueras, J.F., and Nagin, P.A., *Pascal with Style, Programming Proverbs*, Heyden, Rochelle Pk., NJ, 1979.

[Les83] Lescanne, P., "Computer Experiments with the REVE Term Rewriting system Generator," *Proceedings of Tenth Symposium on Principles of Programming Languages*, Austin, TX, Jan., 1983.

[Hdm79] Levitt, K.N., Robinson, L., and Silverberg, B.A., "The HDM Handbook," Vols. 1-3, SRI International, Menlo Pk., CA, 1979.

[Ina80] Locasso, R., Scheid, J., Schorre, D.V., and Eggert, P.R., "The Ina Jo Reference Manual," TM-(L)-6021/001/000, System Development Corporation, 1980.

[Mac82] MacEwan, G.H., "Specification Prototyping," "Special Issue on Rapid Prototyping, Working Papers from the ACM SIGSOFT Rapid Prototyping Workshop," *SOFTWARE ENGINEERING NOTES* 7:5, Dec., 1982.

[MG83] McMullin, P.R., and Gannon, J.D., "Combining Testing with Formal Specifications: A Case Study," *IEEE-TSE*, SE-9:3, May, 1983.

[MS82] Melliar-Smith, P.M., and Schwartz, R.L., "Formal Specification and Mechanical Verification of SIFT: A Fault-Tolerant Flight Control System," *IEEE Transactions on Computers*, C-31:7, July, 1982.

[Mye78] Myers, G.J., *Composite/Structured Design*, Van Nostrand Reinhold, New York, 1979.

[NCC??] "Programmer's Reference Manual for S-PORT SIMULA 67 System, Norwegian Computing Center, (date cannot be determined from document).

[Par72] Parnas, D.L., "On the Criteria to be Used in Decomposing Systems into Modules," *CACM*, 15:2, Dec., 1972.

[RS82] Rees, D.J., "The Kernel of the EMAS 2900 Operating System," *Software—Practice and Experience* 12, 655-667, 1982.

[SRSY77] Shelness, N.H., Rees, D.J., Stephens, P.D., and Yarwood, J.K., "An Experiment in Doing it Again, *But Very Well This Time*," CSR-18-77 Department of Computer Science, University of Edinburgh, December, 1977.

[SYRS80] Stephens, P.D., Yarwood, J.K., Rees, D.J., and Shelness, N.H., "The Evolution f the Operating System EMAS 2900", *Software—Practice and Experience* 10, 993-1008, 1980.

[Smo82] Smoliar, S.W., "Approaches to Executable Specifications," "Special Issue on Rapid Prototyping, Working Papers from the ACM SIGSOFT Rapid Prototyping Workshop," *SOFTWARE ENGINEERING NOTES* 7:5, Dec., 1982.

[Pro82] "Special Issue on Rapid Prototyping, Working Papers from the ACM SIGSOFT Rapid Prototyping Workshop," *SOFTWARE ENGINEERING NOTES* 7:5, Dec., 1982.

[Sta81] Stahl, S., "LSI GUARD System Specification (Type A)," MTR-8452, MITRE Corp., Bedford, MA, Oct., 1981.

[SW82] Sutton, S.A. and Wilut, C.K., "COS/NFE Functional Description," DTI Document 389, Compion Corp., Champaign, IL, Nov., 1982.

[Tei84] Teitelman, W., "A Tour Through Cedar," *Proceedings of the Seventh International Conference on Software Engineering*, Orlando, FL, May, 1984.

[Uni81] "The UNIX Programmer's Manual," Bell Telephone Laboratories, Murray Hill, NJ, June, 1981.

[Aff81] Thompson, D.H. and Erickson, R.W. (Eds.), "AFFIRM Reference Manual," USC Information Sciences Institute, Marina Del Rey, CA, Feb., 1981.

[Win80] Wing, J.M., "Experience with Two Examples: A Household Budget and Graphs," USC/ISI Affirm Memo-30-JMW, Aug., 1980.

[Win84] Wing, J.M., "Helping Specifiers Evaluate Their Specifications," *Proceedings Second International Conference on Software Engineering*, AFCET, Nice, France, June, 1984.

[Zav82] Zave, P., "An Operational Approach to Requirements Specification for Embedded Systems," *IEEE-TSE*, SE-8:3, May, 1982.

[Zav84] Zave, P., "The Operational Versus the Conventional Approach to Software Development," *CACM*, 27:2, Feb., 1984.