Reprinted from:

*Journal of Systems and Software*
Volume 28, Number 1
February, 1995
Pages 179–184

# Controversy Corner

It is the intention of the *Journal of Systems and Software* to publish, from time to time, articles cut from a different mold. This is one in that series.

The object of the CONTROVERSY CORNER articles is both to present information and to stimulate thought. Topics chosen for this coverage are not just traditional formal discussions of research work; they also contain ideas at the fringes of the field's "conventional wisdom."

This series will succeed only to the extent that it stimulates not just thought, but action. If you have a strong reaction to the article that follows, either positive or negative, write to Robert L. Glass, Editor, *Journal of Systems and Software*, Computing Trends, P.O. Box 213, State College, PA 16804. We will publish the best responses as CONTROVERSY REVISITED.

# The Importance of Ignorance in Requirements Engineering

Daniel M. Berry

*Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania*

This paper examines a number of successful requirements engineering efforts carried out by the author and determines that a critical element in the success of these efforts was the author's ignorance of the client's domain.

*Where ignorance is bliss*
*'tis folly to be wise*

— Thomas Gray, 1742

## 1. INTRODUCTION

The problem is that of a team of one or more requirements analysts helping a team of one or more client representatives arrive at complete, consistent, and unambiguous requirements for a system that the client wants built. This system may be anything from a simple closed, P type program (Lehman, 1980) to the next iteration of an evolving E-type system. In any case, the requirements analysts must take the fuzzy, often mutually inconsistent and collectively incomplete ideas of the client representatives and somehow arrive at a shared and unambiguous consensus that captures everything the system should be doing, no more and no less. This consensus must be reached despite any requirements analyst's ignorance of the application domain and any client representative's ignorance of computing.

This consensus is both a hindrance and an advantage. It is a hindrance because it requires work to derive it from the varied views, and it is an advantage because reaching it is a way to sort through inconsistencies among views and complete the incompleteness in individual views.

Arriving at the consensus requires hard thinking on the part of all involved. It is often useful to have techniques to help the thinking be more systematic and give something concrete to do when the thinking stalls.

This article reviews a paper that I and Orna Berry published in 1983 about a joint requirements engineering experience in 1979 (Berry and Berry, 1983). As described in detail in sections 2–5, that experience was successful beyond expectations and the paper examined the experience of attempting to identify the reasons for success and posit methodological, technical, and managerial components of a general approach to successful requirements engineering. These components and the reasons for and against their effectiveness are discussed in sections 6–8. Since 1983, both Orna and I have applied the approach separately to a variety of projects with success, albeit never as much as the first ex-

*Address correspondence to Daniel M. Berry, Computer Science Department, Technion, Haifa 32000, Israel. E-mail address: dberry@cs.technion.ac.il*

© Elsevier Science Inc.
655 Avenue of the Americas, New York, NY 10010 0164-1212/94/$7.00

perience. Was it beginner's luck? Suddenly, in 1993, I applied the approach with as much success as the first time! Sections 9–11 examine this experience and attempt to identify the properties in common with the first event (it cannot be beginner's luck!) that might account for the unusual success.

At the time we wrote the first paper, I thought that the problem was to overcome the ignorance I had about the problem domain or to operate effectively in spite of it. After the most recent experience, I have determined that ignorance was more than something to overcome on the way to writing the requirements. Instead, it was a necessary component in my ability to expose the tacit assumptions that prevent consensus. It seems that among experts, a common disease is the presence of unstated assumptions. Because they are unstated, no one seems to notice them. Worse than that, it seems that no two people have the same set of assumptions, often differing by subtle nuances that are even more tacit than the tacit assumptions. It is the slight differences in these assumptions that confound attempts to arrive at consensus, particularly because none of the players is even consciously aware of his or her own assumptions and certainly not of the differences between the players' assumptions about the same things.

## 2. BACKGROUND

The 1983 paper, titled "The Programmer-Client Interaction in Arriving at Program Specifications: Guidelines and Linguistic Requirements"[1] (Berry and Berry, 1983) dealt with lessons learned during a 1979 experience in which I played a requirements analyst and Orna Berry played a client during a joint development of a specification of the requirements for a statistical experiment simulation program that the client then implemented herself, as part of the research for her M.A. degree in Statistics at Tel Aviv University.

The research issue at hand was the validity of conclusions drawn from experiments involving a sequence of observations in which the data become unavailable from some point on (truncation) or in which some, but not all, data are missing (censored). The program was to permit numerous simulations of experiments with large observation vectors. An experiment was to generate a pair of vectors of observations, to both truncate one of the pair at some random point and censor one of the pair at some random points, and then to compare the conclusions drawn from the full vectors to those drawn from the truncated and censored vectors. Drawing conclusions and comparing them required calculating some well-known statistical measures.

The production of the requirements document took 4 weeks out of a total project time of 11 weeks. In the 7 weeks after acceptance of the requirements document, the client implemented the specified program, tested it, debugged it, did several production runs with a variety of input parameters, studied the output of these runs, did some additional production runs with additional input parameters, studied the output of these runs, wrote up the study and drew conclusions, defended the report as an M.A. thesis, modified the thesis but not the program and the requirements, and filed the thesis (Berry, 1979).

## 3. SUCCESSES

Despite the impediments listed in section 6 auguring against success, the requirements document and the subsequent program were wildly successful beyond all expectations (but not beyond all hopes!). That is:

- The eight-page requirements document did not need to be changed beyond correction of minor typographical errors (missing or misplaced punctuation, misspelled words, etc.) to keep it faithful to the program as implemented.
- The 60-page program worked after only a half dozen error-exposing test runs and very easy corrections.
- The program, through its generation of extra debugging output in anticipation of the testing phase, produced enough not originally required output that ended up satisfying all requests for additional useful computations that were expressed during the thesis defense!

In particular, the first component of success was a complete shock to me. It was the first time, in nearly 15 years of programming experience, that I had participated in a software development in which the requirements document did not need to be changed to keep it faithful to the software as finally implemented. On all previous occasions, either the requirements were changed or they were abandoned as irrelevant, unneeded, too costly to maintain, or impossible to reconstruct after the fact.

## 4. REQUIREMENTS

The form of the requirements document was that of an input-output specification:

1. The input specification consisted of a list of typed, constrained variable declarations for each variable whose value was to be read in.
2. The output specification consisted of a list of all

[1]For a reprint, please contact the author via e-mail at *dberry@cs.technion.ac.il.*

values and tables to be printed out, given input that met the types and constraints of the input specification.

The output specification in turn consisted of

1. an abstract data type *observation_vector* encapsulating all knowledge about the observation vectors and providing all operations for creating them, reading and printing all or part of them, and computing various statistical measures on them, and
2. a main section specifying the vectors to be created and the values and tables to be printed, all in terms of operations of the abstract data type taken as primitive.

No details were given as to how the operations of the type and the values and tables to be printed were to be computed.

## 5. IMPLEMENTATION

The implementation was carried out in FORTRAN by the client herself; recall that she was writing this software for her M.A. degree and had to write the program herself. The abstract type was implemented as a collection of subroutines and functions sharing a named common area that the main program did not see. Thus, encapsulation of the abstract data type was enforced by the program. The implementation details not spelled out in the requirements document were fleshed out in the program. The client learned various programming, testing, and debugging methods as she was writing the program.

## 6. IMPEDIMENTS TO SUCCESS

The impediments that were thought to make success highly unlikely were the following:

- The client, a statistician at the time, knew very little about programming and what was possible with a computer.
- I, the requirements analyst, a computer scientist, knew very little about probability and statistics owing to a phobia developed during a high school course on the subject, in which I received my lowest grade among all of my courses. My mind would freeze at the mere thought of a standard deviation.
- The native languages of the analyst and client are different. They are English and Hebrew, respectively.

## 7. CONJECTURED REASONS FOR SUCCESS

When I analyzed for the writing of our paper why the requirements analysis effort had been so successful despite the impediments, I identified three techniques that had been applied, beyond good ol' fashioned common sense:

1. abstract data typing
2. strong typing
3. Jewish Motherhood

The first two are normally considered applicable only to design and programming languages, which are precisely specified artificial languages with restricted expressibility. However, there is a natural language analogue to these that proved to be applicable to the client's verbal descriptions of the problem and the working requirements document being written in mostly natural language

Specifically, an imperative sentence can be considered an application of its verb (procedure) to its object (arguments), in which the objects can be either direct or prepositions with indirect objects. When a particular verb is suddenly used with a different number of objects or with different kinds of objects, it is clear that something might be wrong. In the view of an imperative sentence as a procedure application, these different uses of a verb can be considered type mismatching errors. In this sense, there is a strong typing that can be understood in imperative sentences. Moreover, a collection of imperative sentences that share argument kinds can be thought of as an abstract data type for the shared object kinds. The sentences are the operations by which the objects are manipulated. Certainly the implementation of the abstract data type is hidden, simply because no implementations have been written yet!

An abstract data type is an implementation of a data type in which the data structures are hidden or encapsulated in a module that exposes only the procedures and functions for operating on the data (Liskov and Zilles, 1974; Parnas, 1972). In the experience, the encapsulation that is normally used to hide implementation details so that they can be changed easily was used to hide my ignorance of domain concepts so that I could work with them as built-in primitives. All I had to do was be certain that the encapsulated concepts were well enough understood that they could be implemented straightforwardly. Then I could fake my way through the problem so well that I could find inconsistencies in what the client was saying even though I did not understand the meaning of these concepts.

The ability to find these inconsistencies came from strong typing (van Wijngaarden et al., 1975; Schwartz and Berry, 1979). Strong typing is that property of a programming language that ensures that the types of all expressions and subexpressions can be computed at compile time. As mentioned, in terms of the natural language that was used as the medium of exchange between the client and the requirements analyst, a type inconsistency manifested itself by the use of a given verb with a different number or different kinds of objects. My ability to notice these differences in sentences uttered by the client allowed me to find all inconsistencies in what she was saying, even though I did not understand the substance of what she said in these sentences. Whenever I found inconsistent sentences, I explained the nature of the inconsistency and asked the client to resolve the problem. The resolution was either that one, the other, or both uses were wrong, the operation was overloaded, or the operation took a variable number of arguments, with the missing ones getting default values. It is no surprise that strong typing proved to be enough to find all inconsistencies. It is well known how effective strong typing is at identifying program errors at compile time and in preventing these errors from happening at runtime (Gannon, 1977; Eggert, 1981). It is the power of redundancy. It is for the advantages of this power that the designers of Ada (Department of Defense, 1983) put it into the language constructs for building strongly typed abstract data types.

Jewish Motherhood perhaps needs the most explanation. It is the ability to keep nudging the client to tell you all that needs to be told and to make the client feel guilty for leaving anything out. It is also the ability to detect from body language when the client has failed to tell you something important. Dan Greenburg in his famous book *How to be a Jewish Mother* (1993) explains how to nudge and inculcate guilt and makes the important point that one has to be neither Jewish nor a mother to be a Jewish Mother. Therefore, it is possible for, say, an American male programmer to be a Jewish Mother. In the experience, I, the requirements analyst, successfully nudged all the necessary information from the client. I was able to detect waffling in answers to my questions and follow up with questions to get answers given with conviction.

These concepts formed the basis for an adaptive requirements elicitation method that Orna and I have put into practice in our own work since the experience. A full narrative description of one application of the method is found in our original 1983 article.

## 8. COUNTERINDICATIONS

Although it is nice to believe that a method that worked in one situation will work in all, the reality is that each problem seems to beget its own method or variation of a method. Besides, there are a number of reasons why this particular experience might very well have been successful independently of the method applied:

1. There was a heavy dose of beginner's luck.
2. The program was small compared to industrial-strength E-type systems (Lehman, 1991).
3. There was only one client representative.
4. There was only one requirements analyst.
5. The client became the programmer.
6. The problem has a strong mathematical basis.

On the other hand, in the 14 years since the first experience, Orna and I have independently applied the method in about two dozen software development projects, both in university research efforts and industrial projects, to produce software for sale. The method has been generally successful and has never failed to help produce good, usable requirements. However, until very recently, no effort felt quite as successful as the first experience. So we felt that there *was* a strong beginner's luck component.

## 9. MOST RECENT EFFORT

I carried out the most recent effort when I was called in as a consultant to help a start-up company write the requirements for a new multiport Ethernet switching hub.

I was called in even though I had protested that I knew nothing about networking and Ethernet beyond nearly daily use of networking applications such as telnet, ftp, and netfind. To give the reader some idea of how poor my understanding of Ethernet was, let it be said that in the early days of Ethernet, once when I had seen a cable lying on the floor, I worried out loud that the ether might spill out and evaporate.

The engineers in the company were almost exclusively hardware engineers. They had been struggling for 4 months to come up with a software requirements document and were getting nowhere fast. They knew about the technology but not how to structure its software well. They had not stated the requirements in full and had been cycling between requirements gathering and software design. They had a stronger understanding of how to specify the hardware, and consequently the hardware part of the project was on schedule whereas the software part was behind schedule.

When they hired me, I asked each person to supply me with a list of the pieces of the system and the features (operations) of each. I read these and began to build my abstractions with their operations. I noticed a number of inconsistencies in both the abstractions and the operations. I came in one morning and had a 2-hour meeting with all of the engineers. During the meeting, I nudged the engineers for resolutions to all the inconsistencies I had noted and any others that came up during the meeting. After the meeting, I worked for 4 more hours writing a new requirements document in the style of the first experience, that is, with a collection of abstract data types and processes that used these. At the end of these 4 hours I had a good first draft specification that seemed to have electrified the engineers. In only 6 hours, I had put down in words, types, objects, processes, and diagrams thereof what they had been trying to say over the last 4 months.

After that meeting, I worked with them over 2 months to help produce two documents, a functional specification and an architectural specification, which were carefully maintained to be consistent with each other.

## 10. IGNORANCE IS THE KEY

Even before this most recent experience, when I had described the first experience to a software engineering class at Carnegie Mellon University, a student, Jim Alstad, had wondered if the very fact that I knew so little about the client's application area had been a significant factor in the success of that first experience. By being ignorant of the application, I was able to avoid falling into the tacit assumption tar pit.

The latest experience seems to confirm the importance of the ignorance that the method is so good at hiding. It was clear that the main problem preventing the engineers at the company from coming together to write a requirements document was that they were all using the same vocabulary in slightly different ways, and none was conscious of any other's tacit assumptions. Each was wallowing deep in his or her own pit. My own ignorance forced me to ferret out these tacit assumptions, and my lack of assumptions forced me to regard the differences in the way each was talking about the same thing as signs of inconsistencies.

It is interesting that the two most successful applications of the method were in areas in which one of the requirements analysts was totally ignorant. In all other, less successful, applications of the method by either me or Orna, it was for a problem in which I or Orna was an expert. I was eliciting requirements for word processing software as part of my research in electronic publishing, and Orna was eliciting requirements for networking software as part of her job as chief scientist for a networking company.

It must be emphasized that I am not claiming that domain expertise is not needed. It is clear that at least one member of the requirements engineering team must be a domain expert. The requirements cannot be invented from total ignorance the way it appears that a magician pulls a rabbit out of a hat; in fact, there must be a rabbit already in the hat. That rabbit is provided by the domain experts.

Robert Glass (1993) reports about a software production company that has English majors helping to write maintenance documentation. He reports amazement that nonsoftware-knowledgeable English majors were consistently able to write high-quality descriptions of the grubby details about programs that are needed to allow them to be maintained by people other than their original developers. Glass and the professional programmers who worked with the English majors were surprised that these English majors, "by virtue of having poked and probed at a variety of the enterprise's software assets, probably knew as much about what software was out there as anyone else in the company." English majors are generally skilled in organizing ideas and expressing them in writing. Perhaps an additional phenomenon at play is ignorance being parlayed into an ability to recoil when the unspoken underlying assumptions inhibit understanding and to hunt them down among the more knowledgeable programmer members of the team.

## 11. RECOMMENDATIONS

On the basis of these experiences, it seems clear that it is essential for each requirements engineering team, consisting of analysts and client representatives, to have at least one domain expert and at least one smart ignoramus! Generally, there is no difficulty finding the domain expert among the client's people. The ignoramus should be smart about computing, information hiding, and strong typing, but totally naive and ignorant in the client's domain. It is absolutely essential for the ignoramus to be attuned to spotting inconsistencies in what people are saying; this means that the ignoramus should have a good memory for what people say and a good sense of language to be able to detect when a verb is being used with a different set or number of objects. These language skills are required of lawyers, psychologists, psychiatrists, negotiation facilitators, etc.

In a team consisting of at least one ignoramus and at

least one expert, each kind of person will find different things about the requirements. The domain expert finds the basic information needed, but he or she falls too easily for tacit assumptions. The ignoramus has no assumptions and asks questions whenever he or she catches a sign of something left unsaid.

We have the interesting situation that ignorance is both necessary and something to overcome. It seems that if there were no ignorance to overcome, the consensus process would be stunted.

It should be stressed that although ignorance is necessary, stupidity is neither desired nor helpful; the ignoramus must be smart. Moreover, total ignorance is not helpful either. There must be some expertise on the teams to drive the quest for facts and provide facts when demanded by others on the teams.

When Linda McCalla read an earlier draft of this article, she observed that it is possible to develop a skill of faking ignorance. I personally have difficulties not using everything I know and thus being lulled into accepting the hidden assumptions. She suggested the technique of building a glossary of all terms before beginning the other steps of requirements elicitation. The list can be brainstormed and differences in meanings can fuel a resolution that exposes the hidden assumptions.

If the ideas presented here are accepted, perhaps we shall see a day in the future in which software engineers will proudly list on their resumes areas in which they are ignorant! In any case, this list is the one section of a resume that gets smaller and smaller as the software engineer gets more and more experienced.


## ACKNOWLEDGMENT

I thank Gregory Abowd, Orna Berry, Bruce Blum, Jorge Diaz-Herrera, Robert Firth, Robert Glass, Shmuel Katz, Linda McCalla, Bill Pollak, Dave Wood, and an anonymous referee for their useful comments. The description of the requirements process as arriving at a consensus is Bruce Blum's.

This work was sponsored by the U.S. Department of Defense.


## REFERENCES


Department of Defense, Ada Language Reference Manual, MIL-STD-1815A, U.S. Department of Defense, Washington, DC, 1983.

Berry, D. M. and Berry, O., The programmer-client interaction in arriving at program specifications: guidelines and linguistic requirements, in *Proceedings of IFIP TC2 Working Conference on System Description Methodologies*, (E. Knuth, ed.), 1983.

Berry, O., Comparison between Two Life Span Distributions Based on Small Samples with Censored Data, M.A. Thesis, Tel Aviv University, Tel Aviv, Israel, 1979.

Eggert, P. R., Detecting Software Errors before Execution, Ph.D. Thesis, Computer Science Department, UCLA, Los Angeles, California, 1981.

Gannon, J. D., An Experimental Evaluation of Data Type Conventions, *Commun. ACM* 20, 584–595 (1977).

Glass, R. L., Can English Majors Write Maintenance Documentation?, *J. Syst. Software* 21, 1–2 (1993).

Greenburg, D., *How to be a Jewish Mother*, Price/Stern/Sloan, Los Angeles, California, 1993.

Lehman, M. M., Programs, Life Cycles, and Laws of Software Evolution, *Proc. IEEE* 68, 1060–1076 (1980).

Lehman, M. M., Software Engineering, the Software Process and Their Support, *IEE Software Eng. J.* 6 (1991).

Liskov, B. H. and Zilles, S. N., Programming with Abstract Data Types, *SIGPLAN Not.* 9, 50–60 (1974).

Parnas, D. L., On the Criteria to be Used in Decomposing Systems into Modules, *Commun. ACM* 15, 1053–1058 (1972).

Schwartz, R. L. and Berry, D. M., A Semantic View of Algol 68, *J. Comp. Lang.* 4, 1–15 (1979).

van Wijngaarden, A., Mailloux, B. J., Peck, J. E. L., Koster, C. H. A, Sintzoff, M., Lindsey, C. H., Meertens, L. G. L. T., and Fisker, R. G., Revised Report on the Algorithmic Language Algol 68, *Acta Informat.* 5, 1–236 (1975).