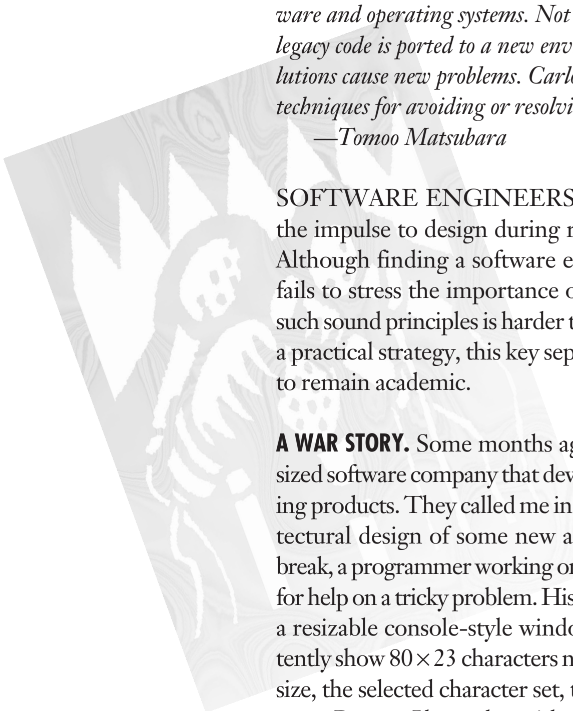
carlo pescio
independent consultant

# When Past Solutions Cause Future Problems

A forum for exchanging ideas, philosophy, and experience.

*Currently, a worldwide effort is underway to uncover the calendar-related programs embedded in our legacy software. When these applications were first coded, had programmers rigorously and repeatedly questioned how their code handled date validity, the Year 2000 problem—the largest crisis the information technology industry has yet faced—could have been greatly mitigated. Most code lasts far longer than its original programmers anticipated, and sometimes even outlives successive generations of hardware and operating systems. Not surprisingly, when this legacy code is ported to a new environment, inherited solutions cause new problems. Carlo Pescio suggests several techniques for avoiding or resolving these issues.*

*—Tomoo Matsubara*



SOFTWARE ENGINEERS SHOULD AVOID the impulse to design during requirements analysis. Although finding a software engineering book that fails to stress the importance of this is rare, *applying* such sound principles is harder than it seems. Without a practical strategy, this key separation is condemned to remain academic.

**A WAR STORY.** Some months ago, I visited a middle-sized software company that develops a family of banking products. They called me in to help with the architectural design of some new applications. During a break, a programmer working on another project asked for help on a tricky problem. His task was to implement a resizable console-style window that would consistently show 80 × 23 characters no matter the window's size, the selected character set, the display device, and so on. Because I knew the quirks in Microsoft Windows that we must work around to make such a window function, we solved the problem easily.

Late that afternoon, as activity on the new project wound down, I asked the programmer's project manager *why* they needed a console-style window. "That's obvious. We receive a number of 80 × 23 pages from a mainframe, and we have to show them in sequence. I want a Windows look and feel, so the user should be able to resize the window, change the font, and so on. That's even in our requirements document...." I didn't mind appearing childlike, so I again asked him *why* he had to show the pages sequentially in an 80 × 23 format. "Oh well, we do that across our entire product family. We started more than 20 years ago with dumb terminals, then we moved to text-based PCs, and now we have this Windows stuff...."

**The mainframe solution, adopted 20 years before, had gradually become an entrenched client-side problem.**

I wasn't convinced yet, so I tried his patience by asking to see a few examples of the mainframe pages. For each session, they all had a common header to identify the document, a portion of text, and a sequential footer—basically a page number. Later in the conversation, I discovered that 20 years ago they devised the paging scheme so that they could show "long" documents on dumb terminals. As the client technology evolved to text-based PCs and then GUIs, the mainframe side remained mostly untouched. The paging scheme, which was adopted as a *solution* 20 years before, had gradually become an entrenched *problem*, to the point where it assumed a life of its own.

Naturally, given the client technology's progress they had a much better option available now: transmit the entire document from the mainframe server and leave the formatting to the client. In the Windows environment, you move through a long document by scrolling, not by paging, which is not only more user friendly but also easier to implement in that environment. Although it took a little archeology to uncover the original problem, the project manager quickly agreed that the scrolling-based solution was better. We rapidly devised a method to keep the mainframe side untouched while we rebuilt the entire document on the client side by extracting and collating pages. The customers loved the new product because they weren't forced to move to the next page just to see the next line. Only the programmer who wasted time working on the 80 × 23 window was upset. A large investment of emotional energy always accompanies de-

**Editor: Tomoo Matsubara**
Matsubara Consulting
1-9-6 Fujimigaoka,
Ninomiya Naka-gun,
Kanagawa 259-01
Japan
matsu@sran125.sra.co.jp

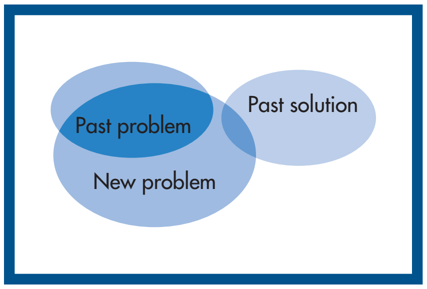


***Figure 1.*** *Too often we fail to see a new problem clearly because we color our perception of it with recollections of similar problems we've solved in the past. We also inject fragments of those past solutions into our thinking, further obscuring the current problem.*

velopment; throwing away a piece of code because the requirements were "wrong" is a painful episode in any programmer's life.

**PROBLEMS AND SOLUTIONS.** I'm a firm believer in software process improvement. I take every chance to reflect on the source of each problem and think about a prevention strategy. That's the only way I know of improving a process: by looking at *real* problems and trying to fix them, not by speculating on the desirable properties of a process in the safety of my study. So, I asked myself, how did a "wrong" requirement snake in?

Did the company suffer from a lack of process? Quite the opposite: They were ISO 9000-certified and allocated substantial resources to analysis and design. But it takes more than a documented process to make a good analysis. There are two frequently neglected issues that strongly influence analysis quality.

♦ As technical people, we thrive on finding solutions. We cannot look at a problem without feeling the wheels of thought spin. We automatically begin to generate solutions, generalize the problem, then seek even better solutions. If we don't have a problem at hand to solve, we look for one—that's the emotional root of research and where our attention first focuses. We need a deliberate act of will to *exclude* solutions from our thinking at the analysis stage so that we can concentrate on fully describing the problem.

♦ More often than we'd like, we inject past solutions into the problem domain during the next development round. More precisely, as Figure 1 shows, we inject part of the solution domain into the problem domain whenever the same or a similar project is revisited. When your system is maintained, ported, and reengineered over several years, it becomes harder and harder to properly recognize fragments of solutions that were injected earlier. They become accepted as part of the problem domain.

**SYMPTOMS AND SOLUTIONS.** Is it possible to clear these blind spots? Here are a couple of the symptoms I've identified that should trigger your internal alarm, and a few proactive steps you can take early on to avoid development pitfalls.

**Symptom 1.** If you are stretching and tweaking the technology, maybe your system really is on the cutting edge. Or maybe there's old technology in your requirements. Remember that making an 80 × 23 paging viewer in Windows was much harder than implementing a more user-friendly, scrolling viewer. Consider traditional query systems as another example. Most are conversational: users log in, use the system for a while, then log out. If you try to port the same architecture to the Web, you may encounter an impedance mismatch with the transactional nature of HTTP, which has no concept of "session." Better, then, to design a new system from the real user requirements than to force your old solution into a new shape.

**Symptom 2.** If your application does not exploit any of the advanced GUI, database, multimedia, and interoperability support provided by modern systems, perhaps your philosophy is *less is more*. Or maybe you simply ported your old solution to the new environment, unthinkingly restricting yourself to the old technology's limits. Consider, for example, data sonification in biomedical products. Although many applications, such as gastroenterology, could incorporate the technology to some advantage, it is mostly restricted to cardiology, where sounds have a long tradition of usefulness. In a similar sense, many so-called object-oriented programs are actually structured programs in disguise. Objects are used only as passive data structures, often with a centralized controller, recycling old solutions into the new paradigm.

**Step 1.** Document the whys. I see a lot of redundant documentation produced during development, mostly to satisfy paper-hungry managers and quality assurance personnel. I often see, for example, a class diagram *and* a detailed plain-text explanation of it. The latter is usually redundant, as it does not add significant information to the diagram. If it does, you should ask yourself why that information wasn't captured in the diagram itself, perhaps by adding a few callout text boxes. People are expected to read the diagram and understand the notation; if management insists on having wordy explanations, consider buying a CASE tool that converts diagrams into detailed reports.

On the other hand, you should invest more time in making a thorough diagram and then documenting *why* you did things a certain way, which external forces shaped the project, and so on. You should do that at each level, from analysis to coding. Expect some resistance. After all, explaining what's already in the diagram is much easier and less threatening than explaining why you chose a specific technique or method. Years ago, a "macho programmer" reacted to this proposal by saying that "good software does not need much explanation." He left the project shortly thereafter, and his legacy code, with its sparse accompanying explanations, remained a maintenance nightmare until it was finally replaced.

**Step 2.** Maintain a childlike "why" attitude about the problem domain, even if you consider yourself an expert—*especially* if you consider yourself an expert. A few more "why" questions may be all you need to avoid design pitfalls. Again, be prepared for some resistance. Years ago, I assisted a biomedical software producer in the development of their next-generation product family. I wasn't very fond of their approach to database design, which started almost at the physical level. I warned them against the risks of neglecting higher-level steps. "We don't need no stinking analysis," they answered, "because we know our domain very well."

So well, they neglected to add, that they already had a solution in mind, based on previous experience. They simply wanted it implemented in the new environment. Months later, it turned out that several classes, including central ones like that for the patient, were too limited to support the new product's richer functionalities. Upgrading the classes required considerable reworking and some features were still sacrificed because the reworking cost necessary to implement them was unacceptably high.

**Step 3.** Remember that, like you, users are human. User involvement is an important ingredient in a successful project. However, you shouldn't believe that user involvement will prevent requirements pollution. Most often, users have a clear idea of their "preferred" solution in mind, which you cannot expect them to filter out for you. The user's solution will probably involve some technology familiar to them, be that paper forms, dumb terminals, personal computers, or graphics workstations. Sometimes, it's fine to build a system according to the user-specified *solution*, without investigating the *problem* domain further. In other cases, however—even though the users' suggestions and opinions about the problem domain are invaluable—you must be wary of incorporating their personal solution intact and unfiltered into the requirements. ◆

*Carlo Pescio has a doctoral degree in computer science and is a consultant and mentor for various European companies and corporations, including the Directorate of the European Commission. He specializes in object-oriented technologies and is a member of the IEEE Computer Society, the ACM, and the New York Academy of Sciences. He lives in Savona, Italy, and can be contacted at pescio@acm.org.*

**Wynn Systems –**

A leading technical job database. Wynnsystems has clients on its Internet site that are aggressively hiring engineers and programmers in the following areas:

| Hardware | Software |
|---|---|
| ASIC | Client Server/Oracle, Sybase |
| Circuit Designer | Windows 95/NT |
| Applications | Multimedia, Win, MAC |
| Engineering Chip-Board | C++, OLE, MFC |
| Systems Engineers | Graphic 2-D, 3-D |
| DVD/MPEG-2 | Audio Drivers |
| Analog IC, Digital IC, Layout IC | UNIX Applications |
| Verification | SNMP |
| Place and Route | Layout IC Internet JAVA, cookie |
| Network Communications IC | Quality Assurance |
| SCSI/Synthesizing | Technical Support |
| Sales Marketing | |
| Technical Support | |
| Info Systems | |
| Technical Writing | |

***A GREAT CAREER IS JUST A CLICK AWAY!***

Contact us at:

**wynnsystems.com**

(800) 995-4447

e-mail to office@wynnsystems.com.

**The MIT Press**

**THE DEFINITION OF STANDARD ML**

**Revised**

*Robin Milner, Mads Tofte, Robert Harper, and David MacQueen*

Written by the originators and designers of the language, this is the essential point of reference for Standard ML implementors and users.

128 pp. $25 paper

**NUMERICA**

**A Modeling Language for Global Optimization**

*Pascal Van Hentenryck, Laurent Michel, and Yves Deville*

"This is a well written presentation of a significant new modeling language that provides an exciting new tool for solving an important class of problems. The authors are careful to motivate their work and facilitate understanding through both abstraction and example. It serves a clear purpose with demonstrated success."
— Eugene C. Freuder, University of New Hampshire

232 pp. $25 paper

**TECHNOLOGY AND PRIVACY**

**The New Landscape**

*edited by Philip E. Agre and Marc Rotenberg*

"Commerce, bandwidth, equity of access, copyright—these are some of the battles that are taking place in the Internet environment, but the one that will affect all of us is the battle over privacy. *Technology and Privacy* is an excellent survey of the different issues on the different fronts. This is important reading for journalists, legislators, activists, and citizens."
— Steve Cisler, Advanced Technology Group, Apple Computer

"[These essays are] brilliant and clearly written. They offer the most innovative thinking about this issue in a decade and a half. Agre truly moves the privacy debate ahead...a significant new contribution to the privacy debate, even among professionals who deal with these issues regularly."
— Robert Ellis Smith, Publisher, *Privacy Journal*

"This collection represents work of many of the most interesting voices in the area of privacy and information technology, skillfully woven together. For anyone interested privacy, this is essential reading."
— Helen Nissenbaum, Princeton University

280 pp., 13 illus. $25

**http://mitpress.mit.edu**

To order call **800-356-0343** (US & Canada) or (617) 625-8569.
Prices higher outside the U.S. and subject to change without notice.