Who is to Blame for the Y2K and Similar Bugs?

Alka Jarvis
Cisco Systems
Dr. Vern J. Crandall
Vern J. Crandall & Associates Inc.
Cindy Snow
Intel Corp.

In the beginning, programming and software architectural design were considered an art. Programmers used their creative skills to build software. Constraints were viewed in terms of "compilers" and "operating systems," not in terms of fundamental design criteria. Quality assurance personnel and testing organizations still are considered to come at the end of the "food chain" and their job is to find any errors that might exist in the software. These people are told, "You must accept the software product the way it has been designed and programmed, and make it work." The implication is that you are at the mercy of the software developers. The real problem is not inadequate testing. It is the lack of quality controls during the crucial design and coding stages. Insufficient controls at this stage interpolate to insufficient design and architecture. The year 2000 bug is a prime example of the lack of such quality controls. It could have been avoided. Someone is to blame.

Introduction

The United State's quality guru, W. Edwards Deming, won the respect of Japanese management after World War II by preaching a philosophy of commitment to quality and continuous improvement of company-wide processes. His recognition that quality cannot be inspected in, but must be designed in, raised an awareness in American businesses. The companies that are industry leaders and retain their customer base continuously evaluate their processes, services, and delivery mechanisms and improve them [1, 2].

We think Deming was right. We also think that the pioneers in the software field were on the right track. Their publications indicated both the right software development problems to be solved and the right way to solve them. These publications include:

- D.L. Parnas [3] with his definition and justification of modularity via information hiding.
- Wayne Stevens, Larry Constantine, and Glenford Myers [4] with their definition and justification of minimizing dependencies via composite design.
- Edsgar Djikstra [5] with his definition and justification of structured programming.

This paper focuses on modular and composite design, and structured programming — and the need for quality controls to insure such. Adherence to these three principles at the design and coding phases insures a higher quality product — even before it goes to test. These software development guidelines were known in the 1960's, early enough to have avoided the year 2000 bug (Y2K).

We choose to focus on the "quality" principles that, had they been followed, would have avoided the Y2K problem. These include both architectural and coding quality principles, defined below:

Architectural Quality

Architectural quality insures that software is designed to be modular, to minimize dependencies and states, and to maximize reusability. Architectural quality reduces the number of modules created by recognizing functional similarities and designing one generic module in place of several "one-offs."

A component that is reused is designed to be state independent. An anti-virus software product, for example, gives the user the ability to configure an automatic action if a virus is found. The user will want to choose the automatic action for real-time virus scanning, for screen-saver scanning, for scheduled scans — and for servers and clients. The configure automatic actions component should be designed, coded, tested, and translated once, but reused in all of the scenarios presented above. Driven by the need for reusability, the component will be designed to work the same regardless of which scan type is being defined. It will be state independent; independent of the context from which it was triggered. A good architecture also separates data from functionality. An example is a data-driven diagnostic tree where each node contains a trigger, an expected response, and further directions based on the actual response. A second example is a data-driven installation component. We have seen a successful setup architecture that compartmentalizes the functionality (add/delete/start/stop services; copy/delete files; etc.) then lists the actual files to be managed as data. Different applications can be installed, upgraded, and uninstalled with no code changes to the setup component, requiring only changes to the data.

How to Design-In Architectural Quality

We propose developing a Graphical User Interface (GUI) prototype based on marketing requirements. It is our experience that such a prototype hastens identification of reusable objects, is a good tool for communicating the design, and can be mapped to marketing requirements (one should be able to identify in the prototype how each function in the marketing document can be triggered). The approved prototype is examined for

objects that appear multiple times, such as the configure automatic actions component. Common functions are combined into one component/module. The goal is to minimize components and maximize reuse. Components are represented in the architecture. Architectural constructs include:

1. Executive modules: Modules in which high-level logic is packaged, making it possible for direct traceability back to the specification.
2. Fan-in modules: Normally, the term describes a situation where multiple modules call a single module. This concept is used for module reuse and for building libraries of reusable functions. We use the term fan-in to describe a method of removing dependencies. When dependencies have been identified across various pieces of code, the code is extracted from the modules in which it appears and fan-in is used to package that code, and dependency, in one place.
3. Massage modules: These modules are used when a fan-in module can almost be used. That is, a condition is slightly different than that expected by the fan-in module. The massage module then modifies the format so that a nonconforming condition is made to conform, and the fan-in module can be used.
4. Informational strength modules: If the dependencies within a set of modules are so different that they cannot be related by fan-in or massage modules, they can be packaged in informational strength modules. These modules have multiple interfaces and multiple entry points allowing each dependency to be considered as a separate program.
5. Transaction (event) processors: Transaction processors use the menu to identify the desired state and transfer the system to that state, defining it as an independent subsystem. Each transaction processor system has its own set of independent states.

The finalized architecture points out all modules that need coding. The GUI prototype will demonstrate how the code should look and feel. The combination allows for novel task assignment. Senior engineers should be working on proof-of-concept and new technology research. Junior engineers should be assigned to the basic coding tasks, and should represent the critical path. That way, if they get behind on the schedule, senior engineers can be pulled back to help catch up. If proof-of-concept, the most difficult to predict schedule for, takes longer than expected (within bounds) the critical path time is not affected.

We recognize that composite design stressing independence usually involves tradeoffs with performance and efficiency.

Code Quality

Code is produced for the modules identified in the architecture. Quality code reduces dependencies, is correct, is logical, and is easy to certify. Code quality is controlled by constraint to three canonical forms (sequence, iteration, and selection) defined later. Code quality is certified via manual pattern matching of each piece of code. We know of one company where two different reviewers look at code. If it does not follow the proposed constructs, or if it is not easily understood, it is looked at by a third reviewer who usually sends it back for redesign. Code quality should be verified before a module is sent to the test group.

Further, coded algorithms should be reviewed and certified for correctness. The quality controls proposed by this paper guarantee architectural quality and code quality as defined above.

Legacy Software Wisdom

As mentioned in the introduction, we consider three discoveries in software development over the last 35 years to be most significant. Each of these discoveries has been based on the work of others, and each has influenced various methodologies.

Modularity via Information Hiding

The major concept Parnas introduced was "information hiding" as the basis of modularity. This has been interpreted as the hiding of a function, leading to reusability. Reusability is maximized when dependencies are minimized. This leads to our approach.

Minimizing Dependencies via Composite Design

Composite design, based on the 1974 paper and presented by Myers [6, 7, 8], addresses the development of architectures focused on minimizing dependencies. While module strength, module coupling, and other design issues have been enumerated by those describing composite design or structured analysis and design or the "Yourdon Methodology," no one after Myers attempted to explain the reasons for the various module strengths and couplings in terms of dependencies — except, perhaps, Lawrence Peters [9] and Vern Crandall [2]. The elimination of dependencies, initially considered an integral part of composite design, is a guiding factor to insure architectural quality. We think it has a far more critical impact than might be expected.

Structured Programming

Another concept we will discuss regards the work of Djikstra. While he deserves the credit for publicizing structured programming, others have had an influence. The original paper leading to this approach to programming comes from professors C. Boehm and G. Jacopini [10] from Italy. (Actually, the concepts originally appeared in a 1938 textbook on linear algebra, but we cannot find the reference.)

The late Harlan Mills of IBM probably had as much to do with popularizing structured programming as anyone. He simplified the proofs of Djikstra, who proved the second half of the structure theorem, and publicized the approach throughout IBM — and the English-speaking world. Structured programming is based on the definition of a proper program in conjunction with the structure theorem, and the correctness theorem. These theorems and definition are paraphrased below:

Definition of a Proper Program

- It contains a single entry and a single exit for the entire structure and for any structures inside.
- It contains no "dead code," i.e. it contains no code or logic structures

that cannot be reached or have no effect on the results.
- It contains no "eternal loops," i.e. all loops are finite and cannot continue to run forever.

Some interpret a proper program "single entry/single exit" to mean "no GO TO's." GO TO's, however, may be necessary to implement structured programming forms in some languages such as assembly language and Basic.

Structure Theorem as Understood by R.C. Linger, H.D. Mills, and B.I. Witt [11]

- It can be shown that any proper program is the equivalent of one composed of sequence, iteration, and selection, plus some logical functions. In other words, any proper program can be written using only these three canonical forms.

The major value of the Structure Theorem is in its proof of the reduction of the number of patterns necessary to produce any process. It also certifies that any "spaghetti program" can be "reverse engineered" to the three canonical forms, which helps greatly when searching out Y2K-type problems.

We further propose implementing the three canonical forms as suggested below:

Sequence: Sequence should flow forward. Do not use GO TO to jump back to code with a lower sequence number. Do not use C's ++ notation because it is error and typo prone. 'X++' can easily, more dependably, and more understandably be coded as 'x = x + 1.'

Iteration: Use DO WHILE loops (pre-test loops) exclusively. Do not use REPEAT UNTIL or FOR loops.

Selection: Use case statements instead of nested IF THEN ELSE structures, except in the case of success-oriented nesting. Success-oriented nesting requires a nested IF THEN ELSE structure where the innermost structure is the success situation. Test for the first or highest percentage error first in the nested set.

Limiting the coding structures to three canonical forms, and further specifying the already validated constructs used to implement the three canonical forms, provides an outline or pattern that can be quickly and easily pattern matched to prove correctness. One can easily outline the flow of any module to verify that it represents a proper program. If the solution has been correctly defined, then pattern matching can quickly validate the correctness of the code. This pattern-matching process is manual, but studies have confirmed that a manual code walk-through is as effective in locating code defects as any other method.

Correctness Theorem, Linger, Mills, and Witt [11]

It can be shown that if the formula of a program contains at most the three canonical forms (sequence, iteration, and selection), it can be proved correct by a tour of the program tree. In other words, if all steps of the program are correct in its decomposition, then the program will be correct.

If P is a proper program, then it can be equivalently written using only three canonical forms. This means that no matter how bad the "spaghetti" that appears in a program, or any process in the world — from a cooking recipe, to directions to reach a destination, to instructions for building computer programs — the logic or logic structure can be replaced with logic or logic structures involving simple sequences of instructions, iterations (loops), and selections (branches). This is a very powerful concept.

Architectural and Code Quality Minimize Inefficient Testing

Some prescribe running software systems for a year or more to "certify" the adequate correction of the Y2K problem. The software system is tested by repeatedly executing production runs, with the hope that rarely executed code or rare conditions will be triggered and point out remaining Y2K defects. We propose that the real test issue is not one of covering the code, it is one of covering the states the code generates. Most everyone who has attempted to install and execute complex software knows that the number of states the software can take on approaches infinity. As the number of states approaches infinity, traditional testing efficiency approaches zero, defined as the percentage of the software states actually tested.

Assuring that the dependencies have been eliminated or "certified" as being transparent to the millennium bug is a better approach. Events that are independent of other events cannot cause the other events to fail.

The Year 2000 Bug was Preventable

The cost of fixing the Y2K bug already is in the millions. The societal and business costs are unknown, and even the most conservative projections seem unbelievable. Surely "Divine Providence" will not let such a profound catastrophe affect mankind, especially now that the Cold War is over.

The entire issue can be wrapped up in terms of information hiding. The millennium bug is not that an algorithm was coded over a period of more than 30 years, which would become defective around the year 2000; the bug was that the knowledge of the problem was distributed through millions of lines of code. Had quality-oriented, well-known "information hiding" strategies been employed at the time they were known, the year 2000 bug would have been a 10- to 45-minute fix in even the largest programs, because all impacted code would have been located at one point, and the impact would have been restricted to only the "code actually necessary to make use of the need to restrict the year to two digits to save 'needed memory or storage space.'"

Industry leaders call attention to the lack of programmers who know the early programming environment, reasoning that such knowledge is a prerequisite to searching for Y2K defects. This should not be an issue. For years, forward-looking teachers have insisted on their students programming in "pseudo-code," the eternal principle of "design before implementation." That means that the entire software industry should contain programmers who program in English and code in any of thousands of implementation languages. The Structure Theorem of Structured Programming guarantees that any program, however large, which is a "proper program," can be "equivalently

written" using only the canonical forms. What this means is that any program — be it COBOL or Basic, Assembly Language or whatever — can be "reverse engineered" into a structured equivalent, which can easily be expressed in English. This has been taught to secretaries and clerks by one of the authors for more than 20 years in companies where the available personnel is limited. You do not need to be a professional programmer to "reverse engineer code." But once these people have performed their task, professional programmers can quickly decipher the code structure and determine how it functions. There are efficient, time-proven ways to determine the logic structure around a potential millennium bug and to create a solution.

Other Preventable Bugs

The "Christmas bug" occurred when a new employee of a major corporation sent an e-mail Christmas card to several of his new colleagues. He included their aliases in the address. As the card was received by each friend, it was immediately forwarded to all on the friend's alias list, and their aliases. This proliferation recursed until the e-mail storm brought the entire mail network to a screeching halt. Either a failure mode was not comprehended in the original design, or recursion was improperly implemented.

Mistakes can happen to trained users under natural operating conditions. Have you ever hit the wrong key on the keyboard, or clicked the mouse at the wrong point as it swooped across the screen? Mistakes with huge consequences are sometimes explained away as viruses. Such accidents more appropriately represent failure modes that were not comprehended or appropriately planned for. Recursion problems are common with loops which process differently for each iteration, and with calling sequences which repeat themselves with different results for each sequence.

Another victim tells of an $8,000 check that did not get deposited to the right account. This caused his check to bounce. A new check to cover the bounced check was cashed twice, causing a $16,000 overdraft. Due to constraints on the banking software, all subsequent bounced checks had to be processed by hand. The knowledge of the overdraft was automatically available to Visa and MasterCard. They promptly cancelled. Credit reporting agencies then produced a damaging credit rating. The bank error rippled through the entire banking and credit rating system, and cost the bank $50,000 to correct. Knowledge of the first bounced check should have been contained in one spot, and then processing inactivated until an alert could be addressed. Instead, the knowledge was duplicated, without verification of the problem, to an undetermined number of places. Errors of this type appear in systems with extremely large numbers of states. Testing is not comprehensive even if it covers all code; it must cover all states the code generates. As the number of states increase, sometimes towards infinity, the software's complexity increases to the point where comprehensive testing is impossible to define or execute. State proliferation occurs when programmers do not pay attention to dependencies among input variables and functions.

New Bug Watch: The Year 2038 Bug

American National Standards Institute (ANSI) provides a standard for date/time representation called time_t. This time standard has received wide acceptance, most notably in the Unix world. The time value is represented by a 32-bit signed integer that denotes the number of elapsed seconds from Jan. 1, 1970. The maximum time-period (or epoch as it is called in date/time lingo) will rollover at 20:14:07 Jan. 18, 2038. Depending on implementations of this ANSI standard, once again computers will be faced with a date representation problem and may not be able to distinguish between 2038 and 1970. The problem is compounded by conversion functions and interpretations of the standards. Some known variations will actually run out as soon as 2036.

We propose that software development managers and engineers discuss the alternatives to time_t use (for both user interface [UI] and non-UI implementation) now. The alternatives are easily comprehended and easily implemented.

Summary

We claim that the year 2000 and similar bugs could have been avoided by adherence to the architectural and coding quality standards insured by modularity, structured programming, and composite design. At every point where these bugs occur, they were "designed in." We propose that such defects be "designed out." We think such defects can be eliminated, and at design time. If a bug must be allowed to exist, its impact — and the knowledge of its existence — should be contained in only one module.

The cost of ignoring architectural and code quality is now obvious. We encourage those responsible for solving the Y2K problem to concentrate on the real problem. We encourage them to add and execute quality controls at the design and coding stages. We hope that executives and software engineers have learned the awful cost of ignoring these simple quality principles.

About the Authors

Alka Jarvis, a senior quality consultant at Cisco Systems, in San Jose, Calif., has an international reputation in the area of software quality and last year was named Outstanding Woman Professional for Silicon Valley. She has written three books, *ISO 9000-3*, published in 1995 by Spriner-Verlag, *Inroads to Software Quality*, published in 1997 by Prentice Hall, and *Dare To Be Excellent*, published by Prentice Hall in December 1998. Jarvis is a Registration Accreditation Board-certified quality systems lead auditor (ISO 9000) and a certified quality analyst. She has 19 years of experience in software development, eight years of which have been in total quality management. Her background consists of management of large systems — development processes, quality control, specification reviews, management of the testing process for large companies, and teaching. She has been a frequent speaker on quality-assurance issues at international and domestic events and has worked in the quality management field in a variety of capacities at Cisco Systems,

Bank of America, Pacific Gas & Electric, Pacific Bell, Charles Schwab, and Apple Computers Inc. She serves as an instructor for University of California at Berkeley, Extension and University of California at Santa Cruz, Extension. She also is an adjunct professor at Santa Clara University in the field of quality and software engineering.

CISCO Systems
360 Everett Ave., Suite 1A
Palo Alto, Calif. 94391
Voice: 408-325-2758
Fax: 408-527-7995
E-mail: ajarvis@cisco.com

Dr. Vern J. Crandall, president of Vern J. Crandall & Associates Inc., just completed serving as vice president of Digital Technology International of Springville, Utah. He received his doctorate degree at the University of Washington. For 25 years, he was professor of computer science at Brigham Young University, Provo, Utah. During this time, he wrote the first software engineering curriculum for BYU and also for IBM technical education. He has consulted with Hewlett-Packard, IBM, Microsoft, Corel WordPerfect, Novell, Pacific Bell, UNISYS, and approximately 45 other companies in the computer industry. He pioneered the first software testing course ever taught in a university and has given keynote addresses at major conferences in the areas of software engineering, comparative methodologies, enterprise-wide information management, getting software products to market, software testing, and software quality assurance. He also served as vice president of software development at Novell, and on the senior staff in the software engineering group at SunSoft, a division of Sun Microsystems.

Vern J. Crandall & Associates Inc.
3549 N. University Ave., Suite 200
Provo, Utah 84604-4417
Voice: 801-375-1415
Fax: 801-375-2295
E-mail: v_crandall@sprynet.com

Cindy Snow, an engineering manager at Intel Corp., in American Fork, Utah, has a background in aerospace and academia. She worked on Naval submarines and Marine radar systems at Hughes Aircraft in Fullerton. She taught in the mathematics department at Boise State University, Idaho and in both the mathematics and computer science departments at Brigham Young University for more than 10 years. Her teaching emphasis has been in the area of operating systems. Drawing on development experience with a variety of methodologies from Waterfall to Rapid Application Development (RAD), she also taught software life cycle classes at the university level. She has designed and managed production of multiple expert systems, including a medical diagnosis expert system, and a comprehensive on-line class scheduler for university students. Recently, Snow managed software product development at a consulting company using a 4GL while helping to design and implement an applicable RAD life cycle. She continues to study in the field of software product delivery to enhance the quality of, and reduce development time for, software products.

INTEL Corp.
734 E. Utah Valley Dr., Suite 300
American Fork, Utah 84003
Voice: 801-763-2274
Fax: 801-763-2895
E-mail: cindy.l.snow@intel.com

References

1. Jarvis, Alka, and Vern Crandall, *Inroads to Software Quality: A "How To" Guide with Toolkit*, Prentice-Hall, New York, 1997.
2. Crandall, Vern J, *Computer Science 427: Software Design and Implementation, Lecture Notes*, Alexander's Print Shop, Provo, Utah. 1984.
3. Parnas, D L, "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM*, December 1972, pp. 1053-1058.
4. Stevens, Wayne, Larry Constantine, and Glenford Myers. "Structured Design," *IBM Systems Journal*, Vol. 13, No. 2, pp. 115-139.
5. Dijkstra, Edsgar, "Go To Statement Considered Harmful," *Communications of the ACM*, Vol. 11, No. 3, pp. 147-178.
6. Myers Glenford, *Software Reliability Principles and Practices*, John Wiley & Sons, New York. 1976.
7. Myers, Glenford, *Composite/Structured Design*, Van Nostrand-Reinhold, New York. 1978.
8. Myers, Glenford, *Reliable Software Through Composite Design*, Van Nostrand-Reinhold, New York, 1979.
9. Peters, Lawrence, *Advanced Structured Analysis and Design*, Prentice-Hall, Englewood Cliffs, N.J. 1987.
10. Böhm, C., and G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules." *Communications of the ACM*, Vol. 9, No. 5, pp. 366-371.
11. Linger, R. C., H. D. Mills, and B. I. Witt, *Structured Programming: Theory and Practice*, Addison-Wesley, Reading, Mass. 1979.