HPWorld 2001 - Chicago - August 20-24

Building Better Software

User-centric Techniques That Endure

Bob Green

Robelle Solutions Tech Inc

Telephone: 1-604-582-1700

Fax: 1-604-582-1799

Email: bgreen@robelle.com

Paper: www.robelle.com/library/papers/better

As long as there has been software, there has been a software quality problem. Software continues to be less reliable than hardware, and may even be getting worse, at least for PC and Web software. Programmers seek solutions to making software more reliable and closer to what the users want. The "soft", flexible nature of software encourages infinite demands upon it - adding dozens of new features before the old ones are working. And each new advance in software engineering and hardware just increases the demands.

Software is badly made. More than that, it is often horribly made. It is developed with the sort of irresponsible abandon that would be unconscionable if it were applied to bridge-building, car-making and possibly even plumbing... And the internet has only made matters worse by encouraging dot-com companies to rush products out ever faster, despite the fact that software is now more complex than ever. Desperate to ride stock-market hysteria and the sea of investment dollars for dubious projects and websites, software companies cram their wares through on shorter and shorter timelines, with no latitude for serious planning, testing or concern for quality. "It's ship first and ask questions later," says one weary programmer, a survivor of a database company. Clive Thompson.

I write from a background in the IT industry, creating business applications first, then tools for IT professionals. What I look for are strategies and techniques for software development that provide a promise, not of perfection, but of improved quality.

Don't despair - there are solutions. Once you understand what creates quality, you can set out to achieve it.

Contents:

About the author

Bob Green

Software developer, award winning author and speaker, member HP Hall of Fame, Bob Green created Robelle to develop irresistible software.

www.robelle.com
With 22 years of success, 14,000 product installations world-wide and over 10% of the Fortune 500 as Robelle customers, they must be doing something right. Bob and his small staff also published dozens of technical papers - strangers imagined they must be a huge company.

Bob welcomes your comments at:
bob.green@robelle.com

Is Failure Inevitable?

Creating software is challenging and humans are fallible, so mistakes and typos are bound to happen. But is this a good enough excuse to give up on quality?

> *If you ask them, developers will claim that "perfect software" is impossible. There is much truth in this. Once a program goes over a few million lines of code, no one person can hold the structure neatly in their head.* Clive Thompson

Many programmers react to this difficulty by throwing in the towel and lowering their quality. It doesn't matter how many bugs there are - ship it. Windows 2000 with about 30 million lines of code is reported to have shipped with 63,000 defects, although only 28,000 are really bug-bugs and the high number may be due mostly to better measurement. In spite of this high count, reviewers found Windows 2000 more stable than Windows NT.

Deadlines are Tight, Pressure is Intense

Software engineering systems assume that you have some control over your schedule and resources. But what if you don't? What if the deadline is imposed by outside forces?

What about the need for speed? The PC revolution reduced project deadlines from years to 6 months and now the Internet revolution has reduced them again to weeks.

External forces demand short deadlines, while programmers who strive for quality often insist that long deadlines are needed. Which is correct and how do we achieve quality when time is tight?

Shaky Foundations?

Most software engineering and quality papers assume that you have some sort of reasonably good architecture in place. What happens if you don't?

What if you are at the point of making architecture decisions? Think of web-enabling a big, old, badly-designed minicomputer application written in COBOL, with a proprietary database and screen interface. Developers need strategies for dealing with the real world where they must build on top of questionable architectures. How do you make quality decisions about the architecture, when it seems that it will be months to years until you can actually demonstrate it?

I know a smart, young software developer named Tyler Close. Tyler started his own software firm, Waterken, using awards he won in Canada for engineering excellence. When asked about the lure of year-plus projects to "really get things right", his immediate response was "Never, ever, under any circumstances, go years without having something to demonstrate. It's depressing for the developers, worrisome for the managers and fatal for investors. My rule of thumb is to plan to have something to show every 4 months and not let it slip to any more than 8. (Perhaps a habit left over from my Waterloo University co-op days, when you only had four months to work with)."

One way to avoid year-long projects is to build on existing architectural elements with known behavior. In this way, aberrant results can be traced to a single new object. But what if the existing elements that you depend upon are badly designed, unreliable, and inconsisent?

Tyler Close suggested building a wall between you and the legacy system. Turn it into a black box, define your messages and responses, then test them and only use those messages: "When selecting what functionality from the old system to use, be a minimalist. Avoid including stuff that might be nice to have but isn't necessary for the project at hand. In software design, never build something until you have a pressing need for it, since if you don't really need it, you don't really know what the requirements are. If you don't know what the the requirements are, you won't build something useful."

Resist getting sucked into the complexity of the "bad software". It helps if you construct two object models of the existing system: one for how the system currently works, and another for how you would like it to work if you had the time and resources to rewrite it. Having this better model to program to is an aid in keeping the ugliness of the bad system from seeping through your wall. Tyler suggests being humble enough to consider alternatives: "Avoid hubris. It is very hard to look at a single, poorly designed system and generalize from that to a flexible, good design. The same way your eyes need at least two vantage points to see in 3D, your design intuition needs at least two examples to see the general design patterns in a complex system."

Lots of Methods, But Disappointment

Over the years, I've seen many proposals for better ways to do programming. They all sounded great to me at first. Structured programming, abstract data types, waterfall life cycle, spiral model, rapid prototyping, formal inspections, test then code, 2167a, structured analysis & design, CASE tools, cleanroom, object oriented programming: when I read about each one, I was sold. At last, a method that would eliminate bugs! I'd read further, talk over the new method with my colleagues, and try to apply it to whatever the current project was. ... Every method had its disappointments. Tom Van Vleck.

Don't be afraid to try new techniques, but don't expect magic bullets either. It isn't that they don't help - they do help. But we still produce low quality software, sometimes even more than in the past.

Why?

Perhaps we haven't correctly identified what **quality** is all about.

Superior Value to the Customer

Defining Quality

The market value of a product is not an intrinsic value, not a "value in itself", hanging in a vacuum. A free market never loses sight of the question: of value to whom? Ayn Rand

A good way to start any inquiry is to define your terms. According to my Little Oxford Dictionary, *Quality* is a noun meaning "degree of excellence". *Excellence* is defined as "surpassing merit", *Merit* as "goodness", and *Goodness* as "virtue".

So what we have here is an ethical issue, not a technical issue at all: Quality is the relative virtue of a thing, compared to alternatives.

Quality Is Not The Same as Effort

What the software cost you to make does not matter. How long and hard you worked on it doesn't matter.

Your software has quality to the extent that it provides Value to some living, breathing people with choices and options. If another program solves a similar problem in a way that the person values more, it has higher quality.

Quality Demands Constant Attention

Quality improvement is a never-ending journey. There is no such thing as a top-quality product or service. All quality is relative. Each day each product or service is getting relatively better or relatively worse, but it never stands still. Tom Peters

Just as quality is not an intrinsic value, it is also not static. Quality is dynamic.

The passage of a few months of Internet time can convert your high quality program into a "has been" in the eyes of the customers.

Quality and Technology Life Cycle

Since value to the customer is the heart of "quality", your efforts must adapt as the customer's values adapt. This is especially important in the "life cycle" of a software product (or any technology product), where the values of the pioneering first customers are quite different from those of the final conservative customers before the software is dropped.

Over the life-cycle of a software program, it may be offering different values to different styles of users:

Innovators - Technology Enthusiasts.
Early Adopters. Visionaries.
Early Majority - Pragmatists.
Late Majority - Conservative.
Laggards - Skeptics.
Geoffrey A. Moore

For example, the PC as a technology product is just entering the Late Majority of its life cycle. My parents have a PC, but only use it for a few things. My grandparents are still skeptical. The PC is coming down in price (some as low as "free") to bring in these laggards, but the usability and bulletproofness still has to improve dramatically to be a value to them.

The **Innovators** value excitement, newness, Wow! features. They will forgive aborts, lack of documentation, incompatibilities between releases and much more. The **Early Adopters** are looking for practical results, but also the headstart value of being first in their area. And they value special enhancements to the software made just for them. The **Early Majority** don't care how gee-whiz your software is, only what problems it can solve for them, without introducing much risk into their organizations. They value a reputation of proven track record. The **Late Majority** are price-sensitive, unforgiving of bugs, and highly value commodity products (dependable, and well known, easy to use). The **Laggards** don't value your software at all!

Quality is Different From Correctness

Quality is not the same thing as "Correctness", which is producing a program that exactly implements the design specifications. What if the design does not specify what the customers need and want?

A creation can have engineering correctness and still fail. For example, I once programmed an accounts receivable system that matched the design specifications when done. The software was fast and reliable, and it produced detailed reports on who owed what. It was a *correct* system.

Unfortunately, the collections from customers went down instead of up after the software took over. We had overlooked one key element of the old manual system. Our clerks would pencil small notes on the account cards to explain difficulties in matching payments to invoices. When the cards were copied and mailed as monthly statements, the recipient clerks would read those notes to iron out difficulties such as tax overcharges. We failed to reproduce these informal notes in our formal system, so the errors were not getting resolved. Communication had stopped. And it was the largest customers who had the most errors, and they used any excuse not to pay their bills. Once we gave our clerks the ability to pen notes on any receivable item, the problem resolved itself. But it taught me a valuable lesson.

Involve the Customer From the Start

After the state spent $20 million and nearly seven years trying to computerize its public-assistance program, the first caseworkers to use the system made their own discovery: They could figure out a customer's benefits faster by hand than with the computer. Seattle Times

The traditional method of developing an IT system includes endless user interviews, voluminous specifications, official approval by confused users, programming phase, integration phase, testing phase, user training, and endless bureaucracy. Notice that this method does not deliver any working programs to the customers until the very end.

Why Do Important Projects Often Go Wrong?

> *COSMOS was a gigantic government project to save money calculating eligibility for social assistance such as food stamps and welfare. "After spending over $20 million, the Washington State government decided to swallow its losses and terminate COSMOS. A consultant's report recommended scuttling COSMOS. The report cited poor management, an overly complex design, difficulty to use by caseworkers, and the use of untested software."* Seattle Times

This software disaster includes most of the things that can be done wrong. The state contracted with an outside firm to design and implement the system. What started as a Big Project, grew into a Giant one. State officials bragged that COSMOS would use artificial intelligence. It was seven years before the first pilot installation, when workers found it took up to twice as long to figure out a customer's eligibility with COSMOS as it did manually. Creating an IT system without delivering anything to the customers until everything is done is like constructing a complete office building on its side, then trying to lift it into position.

A Learning Process

> Moon's Maxim: *The process of developing a system uncovers information about the system that no one could have known at the offset.* Richard Moon

Why is it that systems designed with great care, using experienced analysts and the latest design techniques, can totally fail to solve the customer's problem?

Because of the difficulty of finding out what users really want, and need.

Users are not, and cannot be expected to be, systems analysts. And systems designers cannot think like users. The customer often cannot describe what he wants -- he does not realize how important exceptions are. Even when the analyst extracts all his wants from him and defines them in an enormous specification, he has no idea what is critical and what is frosting. In an attempt to wrench precise specifications from the customer, some shops spend so long on the design that by the time they are done, the customer's needs have changed.

The customers are often shut out after the general design phase. They are asked to approve the specifications so that the programmers can get to work. One thing you can be sure of: the customers may not be able to tell you what they want, but they can tell you what they don't like when you finally deliver the code.

Get the Program to the Customer

It was Michel Kohon who first pointed out to me the reason why it is difficult for the customer to visualize the result of a program, especially an interactive one:

> *A program is not static. The actions it performs vary dynamically,*

depending on the information that is entered. It is a moving body and is unlikely to be adequately described without using jargon. The same applies to mathematics or astronomy, or films. How can we visualize a film from a script? This is why the sooner you show the program to the user, the better it will be for his understanding. Michel Kohon.

When you get the program into the customer's hands, you find out what you don't know. Once you get a reaction from the customer, you can revise the program to closer meet his needs.

Iterative Development

Moon's Second Maxim:
Development methodologies that do not support iterative development are doomed to failure. Richard Moon

It is virtually impossible to get a software design right by just studying and interviewing. As soon as you start implementing, you will start learning new facts and want to revise the design. There are a number of iterative methods that aim to get user feedback and cycle it into the development process. - the one that we use is called the Step By Step Method

---

Start Small and Grow in Steps

The bigger the software project, the bigger the challenge, and the more risk of total failure. An amazing number of large projects never reach their objectives.

The Advantages of Small Projects and Pilots

I have observed that small teams seem to produce quality results more often than large teams. Others have observed the same:

At Pacific Bell, a system was required for automating a million transactions. Two estimates were received, one from a big, outside firm (three years, $10 million) and one from a major Pacific Bell unit (two years, $5 million). Meanwhile, three South California employees took a crack at the task--and did it in sixty days for $40,000. Tom Peters

Small projects have the advantage that they can be cut off or modified quickly. Big projects are hard to cancel, because of the political flak over all the money already spent, and are hard to modify, because of the complex planning that goes into them.

Small projects, especially pilot projects, are ideal for testing new ideas. Even large goals, such as a new aircraft design at Boeing, can be done as a series of small projects. Parts of new aircraft are tried out as redundant systems on current aircraft.

Limit the Time as Well as the Staff

Brooks Law: *Adding manpower to a late software project makes it later.* Frederick Brooks

Why is it that increasing the resources never seems to get the work done faster? One reason is economics. To produce programs, you will assign programmers, but there are never enough. Because the customer's demands will always increase to match your supply of programmers.

This is a common result in human interactions. When they opened a new freeway in the City of Vancouver, a highway expert said not to expect any lessening of traffic on other routes. The reason: by making it easier to travel downtown, the new freeway would entice more suburban motorists to take trips. The traffic expands to fill the roads available, just as software projects expand beyond the number of programmers available.

Many observers have suggested limiting the size of project teams as a way of fighting expansionism. A logical extension of that approach is to *explicitly limit the time allowed for a project*. Pick a small time frame like 2 weeks or a month and organize what can be done in that time with a limited staff.

This is what the Step by Step method proposes: divide larger projects into two-week chunks, then deliver each chunk to the customer for actual use.

*Let's imagine for a moment that we've said we have two weeks to program our system with the existing manpower. No more than two weeks. How can we best solve the problem in the amount of time given? The natural way will be to put on paper what the **musts** and the **wants** are. If both can be produced in two weeks, we will program both, but that is unlikely... The most important objective is to find the absolute **musts** which can be produced with the current staff in a limited period of two weeks....Never go back on the two weeks allowed. It **must** be done in two weeks. Try to imagine that in two weeks' time, it will be the End of of the World. Users will laugh, but they will, as well, appreciate your concern.* Michel Kohon

This has a number of useful results. It involves the customer directly and enthusiastically in the design of the system, it means you never have to write off more than two weeks' work if your design is wrong, it means you can make constant adjustments in your goals as you get realistic feedback, and it eliminates the difference between the development and maintenance programmer. Everyone becomes a maintenance programmer, charged with delivering increasing value to the customer in each step.

Continual Improvement

In my experience, most successful software has continual updates, fixes and enhancements after the initial release. If the developers assume they can get the design 100% correct on the first release, they are greatly increasing their risk of failure. Even if there software elicits a flurry of interest, it will often die out if customer's feedback is not quickly built into an update. When a large software supplier makes this mistake, they create openings for more nimble, smaller suppliers.

One form of continual improvement that has gotten some press recently is called "rapid prototyping":

> *But the real power of rapid prototyping comes less from the technical momentum it generates than from the human interactions it facilitates. It isn't "show and tell," it's "show and ask." It creates conversations between people that would not otherwise take place.* Michael Schrage

If you want to see the tremendous power of continuing, unrelenting, tiny improvements, you just have to look at the Japanese success in manufacturing. They treat every product as an ongoing experiment. When the experiment fails, fails, try to understand why it had gone wrong and then break the corrective process into small steps.

When I sent this paper for review to JP May who develops multi-media and web-based applications at Interesting Software Ltd., he pointed out a similar approach called *fast-tracking:* "I was quite influenced by the book Skyscraper: The Making of a Building, which has nothing to do with software. It is about building the first big skyscraper that was "fast-tracked". They just went ahead and dug the hole while they were still arguing about what scope of building to build; they went ahead and put the spine up while they were still trying to choose an architect, and so on."

> *Sure, you make mistakes, but so what? As Steve Jobs points out "creativity is inefficient". Ultimately, it is more efficient, because, I believe, it is evolutionary and organic. Particularly with web applications, it's just a useless pain to try to "perfectly spec the system" first. For instance, just now we're doing a "job tracking" intranet system for a large print shop. We spent very little time trying to conventionally "spec" it. We just jumped in, found the "one thing" they really need, and did that, then let people use it and work with it. This is much more productive. In large complex systems, clients have no idea what they want until they see something ... and you can build from there.* JP May

Revolutions Too!

This is evolution, not revolution. How to apply this same approach to a revolutionary technology? You must balance long-term needs against shorter-term views of quality. Reconciling these in terms of resource management is difficult.

However, even in the middle of an architecture revolution, you can apply the Step By Step method. It is just more difficult. Some of the steps will be pure architecture steps, without benefit of customer feedback. The longer you go without customer feedback, the more danger you are in.

One of the rules for implementation in e-business is to start small and move quickly refining the application as you go. Over 60% of mega projects fail according to industry stats, so the odds don't favor that approach. Also, most user's don't know what they want because they have never seen it. Giving them something to try gets them thinking about what they really need. Constant updating and refinement is the key to success these days. Arthur S. Mackin

Consider Cisco, whose phenomenal growth was partly made possible by a revolutionary new self-service support system that used the Internet. Did they design and deploy their entire on-line customer service system at once? No! They rolled out self-service "agents" one at a time, starting with the one that customers wanted the most.

"The first agent we put on the Net was the status agent. It allowed you to go on the Internet, enter a P.O. number to find out what the status of your order was, and if it was shipped, hotlink directly to FedEx or UPS and get the exact location. ... Because [this] was successful, we started introducing new agents every three months, approximately. The pricing agent, the lead time agent, the configuration agent, the order-placement agent, the invoice agent, the service order agent, and the service contract agent." Pete Solvik, quoted in Cyber Rules

---

Solve the Actual Problem First

After you have done your analysis and design, you know as much about the customers as you are going to know without deploying the new software system. Your design and implementation plan include a number of steps to reach your objective. How do you order the implementation of those steps? One way is to first implement the steps that seem logically required by the later steps, but this is not the Step By Step way.

Let the Value to the Customer Order the Steps

If we are going to involve the customer intimately in the design process, start small, and improve the software in continual steps, we need a way to set priorities during implementation.

Programmers have a tendency to want to work on the technical challenges first, since that is what they know best. But a beautiful screen doesn't help the customer unless it has data on it that are important to him.

Step by Step aims to discover the customer's actual requirements and *eventually* program them all. One way to prioritize the steps is to order the objectives from the maximum customer payoff to the minimum. Suppose a customer is having cash flow problems? He asks you to provide an order processing system, expecting that the more efficient invoicing will bring in cash more quickly.

The typical response is to give him an order processing system. If you could provide a complete working order processing system in two weeks, including invoicing, you would indeed solve his cash flow problem. But you can't, so you conduct a long study and install order entry as phase one. This is more work for him and does not solve his most pressing problem.

Step by Step challenges you to deliver something in the first step that will make a big contribution toward solving the customer's most pressing problem. This is not easy to do -- it takes creative thought. You might automate just the invoices with the largest dollar amount. Or just the simplest ones, leaving the staff free to deal with the ugly invoices manually. Think of solving the 20% of the cases that generate 80% of the benefit.

Until you deliver a program to the customer, you have not accomplished anything, and you haven't started receiving the objective feedback that will ensure a quality system. The advantage of going after the immediate problem first is two-fold: it gets the customer on your side, and it uncovers facts about the problem that may make the rest of the project simpler or unneccesary.

This is the most demanding part of Step by Step: you must analyze the customer's problems sufficiently to identify the most critical problems. For complex customers, this could be a major study. Remember, your goal is to solve your customer's problems, not just create beautiful technology.

Seeking a Perfect Architecture

Try not to get sidetracked creating the ultimate infrastructure, if it isn't essential to your objectives. You can use object-oriented techniques to isolate the infrastructure from the application logic, so you can go back and replace the infrastructure.

In discussing this issue with Tyler Close of Waterken, I hinted that had he spent a lot of time building underlying infrastructure instead of delivering the capabilities-based exchange system that was his final aim.

Tyler pointed out that there is always a continuum of strategies in regard to infrastructure, each choice involving tradeoffs. If you succeed in building a more supportive infrastructure, you can reap many benefits from it; however, it might take a lot longer.

Tyler first built the Lock™ object database in Java. Writing a new object database required a greater development investment than using an existing RDBMS, but Tyler decided that the investment would pay off in simpler application code. Tyler then wrote the Droplets™ Java servlet. This time, Tyler decided that it would be easier to make his applications fit within the WWW's architecture, than it would be to develop, and especially to deploy, custom network software.

This decision meant accepting several restrictions to his final capability environment, including: the limited request-response networking that HTTP is based on, the centralized approach of having users interact through a web server, the slow response of DNS to address changes, the expense of purchasing an SSL certificate for each host computer in the application and the graphical limitations of using HTML for the GUI. With the Droplets™ product completed, Tyler was finally able to code his Ferex exchange system.

Mark Miller of ERights.org has made different design decisions in his development of a capabilities environment. Mark decided that he could provide a much simpler and more robust environment for writing capability-based applications by creating a new programming language. Mark therefore made a large development investment in the E language. Mark also decided that he wanted his capability environment to support easy peer to peer networking. This meant another large investment in custom network software. The E environment includes its own DNS like service, VLS (Vat Location Service), as well as its own SSL like transport protocol, Pluribus. As of March, 2000 Mark estimates that another 1 or 2 years of development are needed before an exchange system could be deployed.

Both E and Droplets™ are reliant on their host OS for perimeter security. All current operating systems fail at this task. There was a security bug in SSH reported as recently as December, 1999. Worse, each new application added to the host computer brings with it a new set of security vulnerabilities. Mark and Tyler reason that if they can limit the software running on a computer they can limit that computer's security vulnerabilities. They have also designed their systems so that a breach of one machine does not compromise the entire platform.

Researcher Jonathan Shapiro sees the above approach as unacceptable. While it may be possible to limit the software running on a server, it is not reasonable to limit the software running on a user's computer. Jonathan has therefore invested effort in creating a new OS, EROS, The Extremely Reliable Operating System. EROS implements its capability environment at the OS level. This means that it will be possible to add new applications to a computer without introducing new security vulnerabilities. There is currently no timeline for the completion of EROS.

"With Ferex, and other applications I am planning, I am creating secure collaborative group applications. This means that the centralized architecture of the WWW is not really a hindrance, since my applications need to be centralized anyways. Writing smart contracts in Java is more difficult than doing so in E, but I am not expecting my users to write smart contracts at all, just to use smart contracts provided on the host computer. Finally, since all of the software is server side, I can limit what software gets run on the host computer and in this way, hopefully, limit my perimeter security risks.

I have a less robust platform to work with than E or EROS eventually will, but I am ready to deploy now even after starting my work years later than E or EROS. So I guess the moral is, decide what your application domain is and stick to it. Don't build infrastructure that is outside the scope of your domain. Mark and Jonathan are going after a larger domain than I am." Tyler Close

Know The Customer

Since you're not the customer you have no way of knowing what's important and what's not important about the product. Anonymous Hewlett-Packard customer, from Corporate Quality

From the Customer's Point of View

A program is inseparable from the manual, sales brochure, packaging, delivery, training, support, and installation that come with it. In the same way, an application system is inseparable from the operating system and tools that it depends upon. All of these elements go into creating the customer's experience. It takes a total team effort to ensure high quality.

For example, one brand of disposable contact lenses comes in a plastic package that keeps them sterile and moist until use. However, peeling off the foil seal leaves a sharp edge that can cut your hand. The contact lens may give 20-20 vision, the marketing may be superb, the sales team helpful, and the product distribution speedy, but if the customer cuts his hand opening the package, that undercuts the quality of the entire product.

Did anyone on the contact lens team ever watch new customers trying to use the product? Obviously not!

It is very hard to know what the user of a software program or web site really "wants" or how easily they find it with your creation.

As one tester for an email-software firm told me, "A lot of the time, customers don't know what they want. So you have to give them everything, so that everyone will have one little thing they need. And that feature creep is what knocks you out." Clive Thompson

Who is the Customer?

When I wanted a screensaver for my Anguilla News web site, I bought Slide Show Toolbox to convert JPG files into a Windows screensaver. I was the customer of that tool, but the actual target users were the visitors to my web site.

Therefore, the screensaver needed to be as small to download as possible, since the web was the delivery method, not a CD. And installation had to be as simple and foolproof as possible, since many of my visitors were computer novices. My "design" called for an executable download that could install itself and become the active screensaver.

I used the tool to build an auto-install screensaver and I tested it on every computer I could find. The screensever installed and ran fine on Win95, on

WinNT 4.0, on an older Win95, and on a new Win98. But on a Win95 PC upgraded with Active Desktop (halfway to Win98), it aborted with a disturbing message and did not install the screen saver.

Since web users get more conservative ever month due to the vast numbers of new users with no previous computer experience, I decided against using the auto-install. Even though only a few users might run into the problem, my judgement was that they would be very upset. Instead, I wrote detailed installation instructions on how to do a manual installation. Several months later the tool vendor found a way around the problem in Windows, but it came too late for me.

Break Loose: Let Programmers Answer the Phone

If knowing the customer is so important, it is deadly that many organizations deliberately cut their IT staff off from the customers. To shake up such an organization, you need to break down barriers between customers and programmers.

My first suggestion is to let programmers answer technical support calls on their products (and sales calls later). There is nothing like hearing directly from an irate user of a piece of software that you wrote to motivate you to improve it.

Let Programmers Visit Customer Sites

My second suggestion is to send programmers out of the office to see their software in an actual customer environment.

When you are on site, people mention problems that irritate them but which they won't call about. You see them use your product in unexpected ways and use ingenious workarounds for unsuspected design flaws. Users group meetings are another good place to meet customers.

Implement Bug-Tracking Software

There is nothing worse than ignoring a cry for help. Once you have a reputation as a "black hole" (bug reports are never heard of again), customers stop looking to you for solutions, or products.

My third suggestion is to implement bug-tracking software. Not only does this create the opportunity for real-world measurements of your efforts (i.e., outstanding defects, defects uncovered and corrected per week, etc.), but it creates opportunities for improved communication as well.

At Robelle, we use electronic mail and a keyworded database to collect trouble calls and route them to interested parties, including programmers. Anyone can append comments to the original call report and the comments are distributed via E-Mail also. This "conferencing" makes a lab programmer feel almost as if he is on the technical support line.

In selecting a trouble-call system, the key attributes are the ability to build a knowledge base history, communication links to all interested parties, and, nowadays, web-enabled self-service so that customers can easily follow up on their calls.

Continuing, Intelligent Communication

My fourth suggestion is a newsletter. A dependable, concise technical newsletter builds customer loyalty. Even better nowadays is a well-edited web site and is updated every day with news of interest to your customers, including technical updates on your software.

The latest form of web newsletter is the *weblog*, an automated rolling set of news stories and links editied by a web journalist.

The key to a web site or newsletter is **selectivity**. Customers are busy and don't have time to read everything they receive every week. It is no use to your customer to publish a weekly report of all the known defects in the software, if you don't highlight the disastrous ones.

If the information flow is honest, timely, and intelligent, say via a web site with a constant flow of well-indexed news items, the customer will find it irresistable. And since the customers decides how often and how deeply to visit, they find it much less intrusive than piles of printed technical material piling up unread, week and week.

Difficult, But Not Impossible

Developing software reminds me of trying to clean up the basement in the dark. We crash around, running into each other and into unidentified obstacles, grunting and swearing. I encounter something and decide to move it; as soon as I set it down someone else finds it and moves it elsewhere. Adding more people to a process like this probably won't help. Tom Van Vleck

The evidence is clear: programming is hard and mistakes are common. In spite of that, many programmer's have written virtually defect-free software, even with just a pencil and paper: it can be done.

Developers can learn both from those who succeed and those who fail. Consider the experience below from another field, aerospace.

Projects Must Be Grounded in Reality

For a successful technology, reality must take precedence over public relations, for Nature cannot be fooled. Richard Feynman

Physicist Richard Feynman was a member of the Presidential Commission that investigated the crash of the Challenger shuttle. He concluded that NASA management exaggerated the reliability of the shuttle to the point of fantasy, then regularly and subtly reduced safety criteria to maintain the published launch schedule.

Fantasy by top management has a devastating effect on employees. If your boss commits you to producing a new accounting system in six months that will actually take at least two years, there is no honest way to do your job. Such projects usually appear to be on schedule until the last second, then are delayed, and delayed again. Management's concern often switches from the project itself to covering up the bad publicity about the delays.

> *Information from the bottom which is disagreeable is suppressed by big cheeses and middle managers ... Maybe they don't say explicitly, "Don't tell me," but they discourage communication ... it's a question of whether, when you do tell somebody about some problem, they're delighted to hear about it. If you try once or twice to communicate and get pushed back, pretty soon you decide, "To hell with it."* Richard Feynman

An objective project goal unleashes people's minds to discover solutions and attain the goal. An irrational goal just short-circuits the best within them.

In software projects, the *reality* that cannot be ignored is that customers seldom know exactly what will help them and that programmers are fallible. According to research by Watts Humphrey at the Software Engineering Institute, a typical programmer makes one mistake for every 10 lines of code.

Regression Testing

If you even touch the code, you may delete a line by mistake. Thus the advisability of automatic regression testing of each new version. At Robelle, we often run the test suite every night to check that day's changes. Ideally, each bug uncovered is verified with a test script that reproduces it. Without these tests, we have had old "fixed" bugs creep back into the software.

However, never think that testing is enough. It isn't because we can never test every situation.

> The impossibility of complete testing:
>
> We can’t test all the inputs to the program.
> We can’t test all the combinations of inputs to the program.
> We can’t test all the paths through the program.
> We can’t test for all of the other potential failures, such as those caused by user interface design errors or incomplete requirements analyses.
> Cem Kaner.

Testing is essential, but not sufficient to ensure quality. We also need pre-emptive strikes against the source of errors, which may vary from programmer to programer.

I now suggest that we confine ourselves to the design and implementation of intellectually manageable programs. E. Dijkstra

Testing is no excuse for ignoring the benefits of object-oriented programming, code walkthroughs, patterns of programming, and other programming techniques that encourages reliabilty.

Usability

To create things that users understand, we in turn have to understand the users and how they think. Good "usability" is about designing things so that they make sense to the people who use them. The interaction that users have with an object or system should meet their needs and wants as elegantly as possible -- whether it's a web site or a left-handed potato peeler. If it doesn't, they won't want to use it.

Engagement is about "fitness for purpose," both of the medium and the interaction. Fitness for purpose is about choosing the right technologies to use in designing and developing your site, and structuring your site so that users interact with it in a way that is natural to them, using their language, not yours. Claire Rowland

If the customer is key, then usability is essential to software quality. How do you find out if your software or web site is usable?

Jakob Nielsen is a usability specialist. His Useit.com web site has numerous articles on web usability, many of which can be applied to software in general. For example, in "Voodoo Usability", he teaches how **not** to measure usability.

Traditional market research methods don't work for the Web. The basic problem is that one cannot ask users what they want and expect the answer to have any relation to their actual behavior when they go online. Focus groups can often be directly misleading. Jakob Nielsen

In another article, Nielsen shows that the cost of valid user testing need not be excessive. In fact, you only need to test with five users.

Good test tasks can be written in one or two hours, recruiting can be outsourced to a focus group company (at a cost of less than $1,000 for five users), the actual test can be done in a day, and the results can be analyzed in a few hours. If you are a member of the design team, then there is no reason to write an extensive report which nobody will read, so reporting can be done in a one-hour meeting supplemented by a summary that takes 2-3 hours to write. In total, a discount usability study takes only two work days once you know what you are doing. Jakob Nielsen

As with regression testing, usability testing is essential, but not sufficient to ensure quality. A program or web site can be usable, but still fail in other parts of the system because of bad execution or a mismatch with user expectations.

Programming on Internet Time

Just as the web is dramatically remaking the bookseller and travel industries, so it is remaking the software industry.

The web is a software creation platform with a number of interesting attributes that distinguish it from your typical IT application or even PC program: *the customer can jump away* and select an alternative at any moment. Your IT customers don't have this choice - they are usually locked into long-term relationships that are difficult to change. Even PC users usually have to give their money first before they get to use the software.

On the web, the feedback is instantaneous and brutally honest. If you don't get everything right, the customer is gone!

Users don't care how clever the scripting and programming behind the site is, but they do care if the site keeps crashing. Their focus is on the experience of using the site, not on the means by which that experience is delivered.

The trouble with conducting business on the Web is that users don't have a lot of patience. They don't have to because it's really easy to hop from one site to another. Claire Rowland

With the speed an pressure-cooker atmosphere of the web, it is even more important that we programmers keep the essentials of quality programming in mind.

We shall do a much better programming job, provided that we approach the task with a full appreciation of its tremendous difficulty, provided that we stick to modest and elegant programming languages, provided that we respect the intrinsic limitations of the human mind and approach the task as Very Humble Programmers. E. Dijsktra

References

Brooks, Frederick P., **The Mythical Man-Month**, Reading: Addison-Wesley, 1975. Amazon link.

Close, Tyler. Conversations and emails with the author, March 2000: Anguilla. Waterken Inc.

Dijkstra, E., "The Humble Programmer", 1972 Turing Award Lecture at the ACM Annual Conference, Boston, on August 14, 1972.

Feynman, Richard P., "Personal Observations on the Reliability of the Shuttle,"

What Do You Care What Other People Think?, New York: W. W. Norton, 1988. Amazon link.

Green, Bob. This paper is based on an earlier work, Improving Software Quality. In writing "Building Better Software" I reviewed my earlier positions to see if they still apply in the Internet world and I updated all the examples and many of the references.

Hewlett-Packard Company. "Corporate Quality-Customer Visit Program", summary report, August 1989, reproduced in "Steps to Software Quality", Robelle: July 1990.

Humphrey, Watts S. Bugs or Defects? from Watts New column at the Software Engineering Institute: March 1999.

Kaner, Cem. Author of many articles on testing, including "The Impossibility of Complete Testing" from his web site.

Kohon, Michel, "Introduction to Step by Step", SMUG II Proceedings, Langley: Robelle, 1982.

Mackin, Arthur S. Email exchange with the author, March 2000: amackin@cisco.com

May, JP. Email exchange with the author, March 2000: Interesting Software Ltd..

Moon, Richard, "Managing 4GL System Development in the 1990's", **Conference Proceedings of the HP Computer Users Association**, Brighton, England, July 1989.

Moore, Geoffrey A. **Inside the Tornado**, New York: Harper Collins, 1999. Amazon link.

Nielsen, Jakob. His web site Useit.com is about web usability. Alertbox newsletter.

Peters, Tom, **Thriving on Chaos**, New York: Harper and Row, 1987. Amazon link.

Rand, Ayn. **Capitalism: The Unknown Ideal**, New York: New American Library, 1966. Amazon link.

Rowland, Claire. Usability Matters, Webreview.com: March 10, 2000.

Seatle Times. "State Bytes off more than it can chew - DSHS to scrap computer system," May 19, 1989.

Siebel, Thomas M. and House, Pat. **Cyber Rules: Strategies for Excelling at E-Business**, New York: Doubleday, 1999. Amazon link.

Shrage, Michael. "Faster Innovation? Try Rapid Prototyping", Harvard

Management Update, December 1999, Vol. 4, #12.

Thompson, Clive. "Bombsquad", *Shift* on-line magazine: July 1999.

Tom Van Vleck, author of three enchanting essays: "The Evolution of My Thinking", "Cleaning Up the Basement In the Dark", and "It Can Be Done", from Software Engineering Stories