Will Software Ever Work?

Henry Lieberman, MIT Media Lab, and

Christopher Fry, Bowstreet, Inc.

Introduction

Will software ever work? No, not if it's "business as usual" in the software industry. But we *could* make it work.

In the rest of this issue, you'll hear some amazing predictions for the future -- instant, universal communication, pervasive computing, new medical applications, etc. There's only one problem. The software for all these things might not work.

Certainly, if today's software is any indication, it won't. Today's software is appallingly full of bugs. A large, complex product like Microsoft Word is routinely released even when the vendor knows that thousands of bugs exist in it. A misplaced comma in a program caused a NASA space mission to fail. Computers crash or freeze, applications lose data or files, seemingly for no reason. Cryptic error message confuse users.

We could go on and on complaining about it, but, unfortunately, we don't need to. Every reader of this magazine, and every computer user has plenty of their own stories of the unreliability of modern software. Many of these problems are simply minor, time-wasting annoyances. But as computer applications enter more and more of our lives, as this issue is promising you, it becomes more and more important that the software really works.

But we're not writing this article to sound notes of doom and gloom. Cassandras such as the "Inside Risks" column in the back of this magazine can provide you with a copious collection of horror stories.

The problem is not, as many people assume, that system designers and programmers make mistakes. That, we can't avoid. To err is human. Of course, we know of many good software practices that can and should reduce error -- systematic design practices, good programming style, safer programming languages, better testing before release. But we can hardly hope to completely eliminate bugs before software is released. The problem is really that when errors do occur, we currently don't have any really good ways of discovering what went wrong and how to fix it. That's what we've got to change.

People make plenty of mistakes in social, economic and informational exchanges, but an important difference between people and machines is that when mistakes occur in human society, we have good ways of finding out what they are and of fixing them. If you think somebody is telling you something wrong, you can interact with them about it to find out what is wrong, and (assuming goodwill), correct it. You can ask them why they did what they did. You can verify what they are telling you with others. You can ask them what they can do to correct a mistake.

When something goes wrong with a computer, typically, you are stuck. You can't ask the computer what it was doing, why it did it, or what it might be able to do about it. You can report the problem to a programmer, but typically, that person doesn't have very good ways of finding out what happened, either. So bugs don't get fixed. It's that helplessness in the face of problems that causes interaction with computers to feel so frustrating.

Happily, we believe that this can be fixed. But not if the software industry goes on competing only on the ever-increasing accumulation of features. Instead, software development of the future will increasingly be oriented towards making software more self-aware, transparent, and adaptive. Software will still contain some bugs (perhaps fewer), but people will be able to fix bugs themselves by interacting with the software. Software developers will have better tools for systematically finding out where bugs are, and the software will give them help in correcting the bugs. Interacting with software will be a cooperative problem solving activity between the user and the system.

No, not if our economy can help it

Nevertheless, there are some strong forces working against software ever really working. The first is economic.

Given the competative marketplace, developers are often pressured to come up with innovations. Products that feature reliability get edged-out in the marketplace by products that offer more features. Another problem is the endless treadmill of software releases, where "version skew" occurs as Product A depends on Version 1 of Product B, but Version 2 of Product B breaks A. Asking the user to manually track and manage these relationships is disastrous. These social conditions practically guarantee that today's software will be unreliable.

There will have to be a "consumer revolt" against widespread unreliability, and willingness to reward reliability and improvability in products. Historically, such a revolt might be comparable to the American or French Revolutions in impact. A small, but encouraging sign is the recent commercial acceptance of the Palm Pilot, which proposed a simple, reliable, functional interface, winning over more "capable" but complicated and unreliable competitors.

No, not if today's programming culture can help it

Another obstacle is the "macho" culture of programming. "Real programmers" don't need debugging tools. People are psychologically reluctant to admit the prevalence of bugs in their programs, and that makes them unwilling to devote time and money to improving the process of dealing with them.

Barbara Liskov said, "Having a bug in your program is like having a cockroach in your kitchen", a distasteful metaphor suggesting that the presumed cause of a bug is negligence on the programmer's part. But we think it is this denial of the normalcy of bugs and debugging that have directly lead to the unreliability of software.

Yes, only if we provide the tools to fix it

It may sound silly to say it, but software will only ever work if we provide the tools to fix it when it goes wrong. Right now, we don't.

We see an important new direction in providing *end-user debugging tools*. These will be tools that users can use themselves to fix or improve their software. It's crazy that we can't ask a computer at any moment, "What are you doing?" or "Why did you do that?". If we can't get that kind of basic information, we won't be able to tell the programmers what's wrong.

An exciting new technology for giving end users the kind of procedural control that only programmers had is Programming by Example [2]. When the user wants to teach the computer how to do something new, or something different, an example is demonstrated step-by-step in the user interface and the computer records and generalizes a program.

Just because most users are end-users rather than programmers doesn't mean that the public shouldn't be concerned with tools for developers. The quality of debugging tools for developers has a direct impact on the quality of the resulting software, because if developers can't find and fix bugs the software will not improve quickly enough.

Boeing spent more than $50 million developing the interface to the cockpit of the Boeing 777, even though the "user community" is only a few hundred pilots. This expense is justified because those hundreds of pilots ferry around millions of passengers who pay the consequences of errors. Hundreds of programmers write programs for millions of people, yet no efforts of comparable scale have been mounted to improve the tools for debugging.

Moore's Law states that computers double in performance once every 18 months. Fry's Law says that programming environments double in performance once every 18 years. If that. We're not talking about simply the speed of running an application, but more importantly, the speed to develop reliable software functionality, regardless of how fast it runs.

One thing that can help is better programming languages. Languages that software is built on need not just to be reliable in themselves, they also need to be easy to learn, powerful, and extensible. One characteristic of language design not generally appreciated is the design for debuggability. Another is the design for ease of advanced tool creation. Its a lot easier to add good tools to a good langauge than the other way around.

It's not our place here to detail the many ways in which debugging tools can be improved. See [1] and [3] for a myriad of exciting new developments and directions. An important component of debugging tools is *software visualization*, using the considerable graphic capabilities of modern computers and our prodigious power of perception to quickly perceive spatially and dynamically what is going on in software. Other kinds of tools support the detective work of *localizing* bugs, *diagnosing* and analyzing problems, and *instrumenting* pieces of the software environment to monitor their behavior.

Ultimately, software will be something that we can use, not just for doing tasks, but for figuring out what it is that we really want. After all, almost any improvement to a piece of software could be viewed as "debugging it". So the process of debugging is really a process of improvement, and software is really a medium for "debugging" ourselves. Therein lies hope.

References

1. Henry Lieberman, ed. Special Section on the Debugging Scandal, Communications of the ACM, March 1997.

2. Henry Lieberman, ed. Your Wish is My Command, Morgan Kaufmann, 2001.

3. John Stasko, John Domingue, Marc Brown, and Blain Price, eds. Software Visualization, MIT Press, Cambridge, MA, 1998.

HENRY LIEBERMAN is a Research Scientist in the Software Agents group at the Massachusetts Institute of Technology Media Lab in Cambridge, Mass. *lieber@media.mit.edu*

CHRISTOPHER FRY currently works at Bowstreet, Inc. in Lynnfield, MA on tools for developing complex web sites. *cfry@bowstreet.com*