Software Quality Journal **Vol. 10 No. 1, 2002**

Editorial: Risks to Software Quality

As a software engineering researcher and teacher, I frequently hear comments about how awful our software is. Such comments come from both computer science and engineering academics, and practitioners. We all hear complaints about failures of commonly used applications such as browsers, word processors, presentation software, etc.

From my experiences collaborating with industry, I know that many, if not most, software systems are developed in a very ad hoc manner. We professors know many students who are convinced that our lectures on software development techniques are irrelevant. Thus, I have ample reason to believe that most software is likely to be of low quality.

Yet, I am continually impressed by how well our software actually works. My bank statements are correct, the phone system rarely crashes (such crashes make the news), fly-by-wire airplanes do fly, point-of-sale terminal systems work, I can usually make purchases using the Internet. I can go on and on. Our software usually works well, in spite of our expectations. Of course, the software that is of the highest quality is the software that must be of high quality.

Although software does generally work well, in spite of how it is developed, there are serious threats to future quality. These threats come from at least three interconnected sources:

1. The problems that we use software to solve have become much more complex. Years ago, we wrote software to automate a companies payroll. Today, we write software to automate an entire business. The domain knowledge and complexity of the relationships in the problem space of enterprise software ensures that our software will be incorrect, at least in the details. The trend towards increasingly complex problems that must be solved via software seems to be accelerating, at least at a pace that matches improvements in hardware.
2. The fast pace of development in our highly competitive business environment puts enormous pressure on procedures to assure quality. Warren Harrison described this problem in a recent editorial[1].
3. The methods that we use to solve software problems are evolving at a rate that is almost as fast as the increasing complexity of the problem space (source 1, above). New methods are adopted on the basis of the enthusiasm of the proponents of the methods, rather than on careful studies of effectiveness. These new development methods and tools may solve particular development problems, but they can have a negative impact on long-term quality.

[1] See Harrison, W. 2001. Editorial: software quality and internet time, *Software Quality Journal* 9(2):77-78, June 2001.

We have little control over the first two sources; our employers and clients bring the problems to us, and "internet time" is a result of the competitive business environment. However, we do have some control over what development methods we adopt.

We can trace the history of software development, in particular the techniques adopted in industry. Structured programming replaced unstructured programming in the 1970's; structured design replaced structured programming in the 1980's, to be replaced by object-oriented methods. During the last decade, developers started to use object-oriented design patterns, and now Extreme Programming methods.

Each of these developments does solve problems with the prior methods, but they can have a negative impact on long-term quality. Object-oriented software has added complexity over procedural code --- maintainers must understand complex relationships between objects with many identities unknown at compile time, since they involve dynamic binding. Object-oriented software systems tend to have many small classes with short methods. Control flow complexity and intra-module complexity is reduced, while the connections between modules is increased. Design patterns add indirection with additional abstractions, additional classes, and associations between objects. We still do not know whether object-oriented systems will be easier or harder to maintain and adapt than "old fashioned" procedural programs. The claimed benefits of Extreme Programming have not been convincingly demonstrated.

Another example is the emerging field of *aspect-oriented programming*[2]. When we develop software using object-oriented techniques, we develop an overall static structure, or *primary decomposition*, of a program, which is often represented with a class diagram. This primary decomposition is often inadequate for expressing crosscutting concerns such as security issues. Using an aspect oriented language, a developer writes an aspect --- the functionality for the crosscutting concern---and the language processor *weaves* the aspect code into the original program.

The aspect-oriented approach does make the aspect a named functional entity. Thus, there is a specific component addressing the aspect. However, the process of aspect development and its affect on the overall design integrity is unknown. One concern is that the weaving process can introduce content coupling, where a module references the internal elements of another module. Content coupling is generally considered to be the worst kind of coupling. The interactions between the code in an original function or method and the aspect that was woven in may be difficult to track.

I am not saying that we should not use object-oriented methods, design patterns, extreme programming, aspect-oriented programming, etc. Rather, we should use these methods carefully, and be skeptical. I encourage objective, scientific studies of new methods. Then developers can make informed, rather than risky decisions.

Please share your thoughts on these and other software quality issues. Send your comments to me at bieman@cs.colostate.edu.

James Bieman
Fort Collins, Colorado
U.S.A

[2] For a good introduction to aspect-oriented programming see the October 2001 issue of Communications of the ACM