Software Quality Journal **Vol. 12 No. 1, 2004**

Editorial: The Role of Prognostication in Software Design

Software design decisions are based in large part on expectations about how a system will evolve. Both the quality of an evolving system and the ease of adapting it depend on the early design. The only hope for making informed design decisions leading to systems that remain high-quality and adaptable is to improve the ability of designers to prognosticate. Rather than use a crystal ball, comprehensive studies of how existing systems have evolved in the past can provide solid evidence into the connection between early design decisions and the evolving adaptability and quality of software systems.

In a prior editorial published in the *Software Quality Journal*, I wrote the following:

> "I see two key reasons that often-used software is usually `clunky' --- hard to use, with many glitches. Early developers could not predict future user demands, and if they could they often did not have the resources to build the most flexible, scalable design." [1]

Adapting these systems required many inelegant design and coding tricks to meet the new demands of a larger and more diverse user community. These demands were not anticipated, and the early designs could only be modified inadequately with great effort.

In theory, a carefully done design with a well-thought out software architecture will be relatively easy to adapt. However, the choice of a software architecture and design is still based on expected future changes. And the adaptability of the system depends upon the accuracy of the predictions.

A particular software design may make it very simple to add one kind of feature, but make it very difficult to add other features. For example, the use of a Visitor design pattern [5] to implement a programming language compiler makes it easier to add new compilation phases, such as type checking or performance optimization. To add a new phase, a developer only needs to add one new concrete visitor class. However, using a Visitor pattern will make it much more difficult to add new constructs to the programming language itself. To add a new construct, a developer must add the construct class, and add a new method to every concrete visitor class for the new construct. Clearly, the value of a design structure depends on how the design will be used and adapted.

When a developer uses a design pattern, he or she is making an implicit prediction of future demands for changes. The quality of the design cannot be evaluated without knowledge of these future demands. So, it looks like we need a soothsayer to evaluate the adaptability of a software design.

Some advocates of *extreme programming* argue that we should not expend great effort on early designs, and employ what may be called "just in time design" --- refactor a design or program when it is difficult to make a change [4]. The design is modified enough to ease the next change. The danger is that a design evolves incrementally in response to early demands for changes. However, these early changes may not be similar to later required changes. Thus, a design may still evolve into one that is inflexible.

We can understand how systems will evolve in the future only by studying how they have evolved in the past. A key need is to understand the relationship between design

structures and their relative ease of adaptation. The software engineering community needs answers to the following questions:

1. How do systems evolve and what is the nature of software change?
2. What design structures and design strategies really lead to more adaptable systems?
3. Can we provide support to aid developers to design and/or implement in a manner that is consistent with these structures and strategies?

These questions can be answered by studying the evolving designs of real systems of substantial size.

I took part in a study that suggests that software designers often make design choices that conflict with future demands for changes. We examined multiple versions of five systems; in four of the five systems, classes that play roles in design patterns are among the most change prone classes in the systems [2, 3]. Yet, these pattern classes should be less, not more change prone. These results may be due to problems with the design patterns themselves, or to the information available to the developers when they made the decisions to employ particular patterns.

We can improve our ability to perform relevant prognostication only with a much deeper understanding of how systems have evolved. Or, to paraphrase the philosopher George Santayana, those who cannot remember how past systems have evolved into disordered jumbles "are doomed to repeat" the process.

James Bieman
Fort Collins, Colorado
U.S.A.

References

[1] J. Bieman. Why good software goes bad. *Software Quality Journal*, 10:3(201-203), September 2002.

[2] J. Bieman, D. Jain, and H. Yang. Design patterns, design structure, and program changes: an industrial case study. *Proc. Int. Conf. on Software Maintenance (ICSM 2001)*. 2001.

[3] J. Bieman, G. Straw, H.Wang, R. Alexander. Design Patterns and Change Proneness: An Examination of Five Evolving Systems. *Proc. Tenth Int. Software Metrics Symposium (Metrics 2003)*, pp. 40-49, 2003.

[4] M. Fowler. *Refactoring: Improving the Design of Existing Code* Addison Wesley, Reading MA, 1999.

[5] E. Gamma, R. Helm, J. R., and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading MA, 1995.