Legacy systems present a fundamental challenge to those who own and operate them: they have begun to age but continue to provide vital services. They were designed following requirements and an implementation approach that existed earlier in the organization’s life cycle. Then they were released into environments different from those planned. Now, years and sometimes decades later, they are still expected to operate efficiently, solve problems, and incorporate changes in technology and business practices for many years to come.

Because legacy software systems are so crucial to an organization’s survival, they are not retired or redesigned without compelling reasons. Major changes require a huge investment in new technology, with the significant risk that the new systems may fail to deliver the required services. Therefore, organizations maintain functionality, correct defects, and upgrade legacy systems to keep up with changing business conditions.
Keeping legacy systems up to date involves two levels of change. One is technological, such as moving a system from a mainframe to desktops on a local area network. The other entails modifying the application to increase its functionality or ease of data access. Indeed, it is sometimes difficult to draw the line between preserving a legacy system by making a few enhancements or completely redesigning it.

In any event, cost is the essential driver for redesigning a legacy system. Cost and schedule overruns cause many redesign projects to be abandoned. If the risk of redesigning a system appears too high, the legacy system is preserved. However, there are plenty of reasons not to preserve legacy systems. For all aging systems, stability decreases over time. Modifying the legacy system therefore becomes increasingly cumbersome because those maintaining it don’t always understand the impact of small changes on the overall architecture. Response time to changes and corrections increases due to the need to preserve existing functionality. New functionality is difficult to add and engineers are reluctant to do so because any change requires extensive regression tests.

We must therefore evaluate legacy systems based on whether they are

♦ flexible enough to handle changes in requirements and

♦ adaptable enough to handle new requirements.

Increasingly, organizations are considering alternatives to maintaining a legacy system until it is no longer supportable. Some have begun redeveloping or replacing their systems with commercial products (such as COTS) where available. This means, however, that organizations must face the prospect of maintaining commercial products and the required interfaces between them.

**How Long Should Software Live?**

What business reasons exist for keeping a legacy system? Why design for maintainability instead of redesigning systems wholesale? What factors influence whether to install new software or spend additional effort on the legacy software?

Our Point-Counterpoint section addresses these issues. Nicholas Zvegintzov argues that software should live longer—that organizations should put more effort into maintaining and updating them—while John Munson asserts that systems far outlive their usefulness. Obviously, the truth lies between these extremes and depends on many factors. The major question is, what is software’s optimal lifetime? A legacy system might be more stable compared to a new design, but over time each small change introduces new defects caused by ripple effects that are difficult to find and correct. The cost of maintaining a legacy system could eventually exceed the cost of installing a new system.

Munson likens legacy software systems to old houses. Obviously, nice century-old houses with a significant history exist in most cities. Their design did not include many of today’s amenities, but they survived with changes needed to keep them in good shape and thus allow their owners to enjoy living in them. However, other houses—some built much later—were demolished because they no longer served their owners’ needs. While a natural life span seems to exist for construction work as well as for software, initial quality will affect a system’s longevity, just as a well-built house will last longer.

**Emergence of Component-Based Systems**

Information technology applications are no longer monolithic blocks. Increasingly, software engineers compile components with heterogeneous origins. Organizations outsource many of these components, some of them COTS, while engineers design others in-house. Object-oriented languages such as Java, together with glue languages such as Visual Basic, facilitate building componentware. As systems grow, changes in the components may become necessary—but this could be difficult with an outside vendor’s components. The user may have to wait until the next vendor release to obtain the desired functionality.

Thus the future maintainability of COTS and components developed in-house is called into question. How do testers prepare for regression
testing of such components? Jeffrey Voas answers these questions and suggests ways to emphasize the maintainability of component-based systems during the design process.

Lessons from a Restoration

Spencer Rugaber and Jim White address the practical aspects of maintaining a legacy system. Using the interesting analogy of the restoration of the Sistine Chapel, they discuss the restoration of an automatic call distribution system. Although telecommunications switching systems have a long history of successful operation as legacy systems, we know little about why they succeed.

Rugaber and White offer some insight as they describe both the upgrading of the entire system within the existing architecture and the many component improvements. Lessons learned include the need for tool support, knowledge dissemination, and project management.

Legacy System Stability

Many models provide techniques for evaluating the reliability of new software by measuring detection and correction of defects. In the last article in this Focus section, Norman Schneidewind presents a unified approach for assessing the stability of the maintenance process in terms of the reliability and risk of deploying software. He shows the results of evaluating the NASA Space Shuttle flight software, a highly reliable system that has grown in functionality over the last 15 years.

Seeking Middle Ground

There is no universal answer to the question of whether to preserve or redesign a legacy system, because the costs and benefits will differ in each case. The same applies to the related question of how long to maintain a legacy system before redesigning or replacing it. Most organizations do not rush to replace legacy systems because their very survival may depend on the system's continued operation.

If the decision were entirely in the users' hands, they would maintain systems longer than is presently the case. However, like automobile and consumer appliance manufacturers, information technology suppliers engage in a policy of planned obsolescence. While the new technology is frequently innovative and attractive to users, the new hardware, operating systems, and application programs are incompatible with their legacy systems. This forces users to eventually replace their systems to take advantage of the new technology.

However, it is not necessary to choose one extreme or the other. A third alternative maintains the existing system while developing a replacement system. This permits thorough inspection and testing before putting the new system into service. It may also be possible to install the replacement system in stages, thus minimizing disruption to the existing system. Of course, this alternative assumes there are resources available to maintain the existing system while developing its replacement. If this is the case, organizations can capitalize on new technology without incurring the risk of redesigning and replacing the existing system while it is operational.

Acknowledgment

The authors acknowledge the ideas contributed by Twyla B. Courtot of AT&T Solutions.
About the Authors

Norman F. Schneidewind is professor of Information Sciences and director of the Software Metrics Research Center at the Naval Postgraduate School. He developed the Schneidewind software reliability model used by NASA to assist in predicting the reliability of the Space Shuttle's software. Previously, Schneidewind held several technical management positions in the computer industry, where he directed IT projects in both the public and private sectors. Schneidewind received a BS in Electrical Engineering from the University of California, Berkeley, an MSEE and MSCS from San Jose State University, and an MS in Operations Research (MSOR) from the University of Southern California. He is a Life Fellow of IEEE.

Christof Ebert has been with Alcatel in Antwerp, Belgium, since 1994. He is currently a software engineering process group leader in Alcatel's Switching Systems Division, where he is responsible for the software metrics program. His research interests include software metrics, software process analysis and improvement, and requirements engineering. Ebert earned a PhD in software engineering from the University of Stuttgart. He is a member of IEEE, GI, VDI, and Alpha Lambda Delta honor society. He is also a member of the IEEE Software Editorial Board.

Address questions about this article to Schneidewind at Code IS/Ss, Naval Postgraduate School, Monterey, CA 93943; e-mail nschneid@nps.navy.mil.

September/October Issue

Autonomous Space Systems

To enable more frequent and more intensive space exploration missions at lower cost, NASA's new era of solar system exploration is being designed around the concept of sustained intelligent presence on the space platforms themselves. AI, spacecraft engineering, mission design, software engineering, and system engineering all have a role to play in these developments.

Guest edited by JPL’s Richard Doyle, this issue will feature articles on spacecraft planning, scheduling & resource management; spacecraft executives; onboard science data analysis and knowledge discovery; software architectures for spacecraft autonomy; and testing and validation of autonomy software—and more!

Also appearing in upcoming issues

- Configuration: From Mass Production to Mass Customization
- Self-Adaptive Software
- Knowledge Representation: Ontologies
- Intelligent Agents: The Crossroads between AI and Information Technology
- Intelligent Vehicles

IEEE Intelligent Systems (formerly IEEE Expert) covers the full range of intelligent system developments for the AI practitioner, researcher, educator, and user.