

Keeping the lights on: Legacy systems and the maturing workforce

Level: Introductory

Ben Lieberman, Principal Architect, BioLogic Software Consulting

15 Sep 2005

from The Rational Edge: This article examines the critical role that older workers with extensive knowledge about legacy systems should play in maintaining, integrating, and replacing those systems. Based on his consulting experiences, the author describes effective techniques for discovering legacy system characteristics and leveraging the wisdom of experienced staff members.

Many enterprises still execute critical business operations -- billing, order management, customer relations, inventory, sales tracking, payroll, accounting, materials management, shipping, manufacturing, and communications -- via older software systems that run on large, mainframe computers rather than individual PCs. To meet changing business needs, these companies continually update, extend, and integrate their systems.

Paradoxically, many of these companies also have policies that threaten the single greatest source of knowledge about their older systems: their most senior personnel. Although the aging workforce represents a vast pool of talent and experience, these businesses neither actively recruit senior workers nor provide incentives to retain those on staff.¹ Instead, they mistakenly assume that they can hire younger, lower-paid people to perform the same tasks.

On the flip side, other businesses refuse to acknowledge that certain of their legacy systems have become obsolete. They spend a huge percentage of their budget on maintenance, and live in constant fear of losing the few workers who know the system inside out. Meanwhile, younger managers often dismiss these older workers as "grey-beards" or "curmudgeons," and neglect to train them on new technology or in new ways of doing business.

These attitudes and behaviors create business risk and waste. Senior workers represent a precious resource not only for understanding, maintaining, and integrating aging software systems, but also for replacing them with new technology that will enable the company to move forward without compromising current operations. By offering senior workers the advantages of training and development in automated tools and techniques, companies can get the best of both worlds: They can retain and document critical business knowledge and also update and increase the ability of existing staff to help in designing and developing better replacement systems.

Assessing legacy applications

Once a system leaves the *development* stage, it enters the *maintenance* stage, which often lasts many times longer, perhaps even decades. Maintaining systems is very different from creating them, because it involves repairing flaws (i.e., "bugs") resulting from the initial development effort, which often includes corner-cutting, design trade-offs, and other compromises. Precisely when the organization is trying to gain a return on investment, software operating costs may start to climb. Worse, at the company's urging, the senior personnel who fully understand the programming logic may have moved on to other projects, other organizations, or even early retirement. Of course, these problems are compounded if systems have undergone continuous updating and integration over a period of many years. At this point, support costs can start to consume a larger



and larger part of the IT budget, severely limiting new investments.

The first step toward controlling these spiraling costs is to understand the advantages and risks associated with each system in your organization, so that you can make intelligent decisions about whether to maintain, integrate, or replace them. This requires a *technology audit*. For a truly objective assessment, it is usually best to engage an external consultant who is not involved with system maintenance. However, senior organization members are an invaluable resource for these consultants. In fact, they are often the *only* source consultants can turn to for an understanding of why a system came into the organization and what business functions it has supported. In my assignments, establishing relationships with senior team members has helped me uncover system information that I could not have gained in any other way: what the original business drivers were, how exceptions are handled, and what functionality is used or not used.

The steps for assessing a legacy system are as follows:

- Interviews with senior business personnel
- Interviews with senior software developers
- Interviews with information services (IS) staff
- Modeling of business operations, based on interview results and document review
- Capture of software functional flows (e.g., UML Activity Diagrams)
- Determination of system integration points and dependencies

You can perform these steps to gain an understanding of the "as-built" system status. Beginning with the senior business and software personnel helps you save a great deal of time that might otherwise be wasted on tracking down system information. Moreover, since written documentation rarely (if ever) gets updated when systems change, the best/only source of system information are the people tasked with developing or maintaining those systems.

The audit should focus on specific system information (as outlined in Table 1). This information is key to making decisions regarding the support/replacement of any particular legacy system.

Table 1: Areas and techniques for an organizational technology audit

Audit area	Technique
Business processes and software support	Capture detailed business flows in diagrams, indicating areas where software systems are used.
Inter-system dependency	Note where software systems depend on information or functionality in another system. In particular, note where two or more systems have circular dependencies.
Monitors	Describe system performance monitors.
Security	Note roles and responsibilities for all system users; detail system privileges available to each role.
Operational criticality	Rate the importance of each system to overall business operation (e.g., essential, important, supporting, marginal).
Concurrency	Determine the average and peak user loads for each system.
Transaction complexity	Rank the complexity of the data operations (e.g., complex, detailed, simple). For example, if a business operation involves hundreds of data elements, then it is complex.
Rate of content change	Note the frequency with which users present content changes (e.g., hourly, daily, etc.). This relates primarily to informational applications (e.g., Web sites).
Reporting	Define business reporting needs.
Performance (e.g., "real-time")	Define requirements for system performance/uptime. These may be affected by concurrency and transactional complexity.
Data integrity	Note methods (or the lack thereof) for ensuring data integrity, including backups and recovery.

System configuration	Describe the system configuration's complexity (e.g., complex, detailed, simple). A complex system may have hundreds or thousands of associated configuration values.
----------------------	---

In general, the audit report should help you identify which systems you can still maintain cost effectively, and those you should replace. These decisions can be tricky; since one person may find a particular system inadequate, while another may have learned "tricks" to perform difficult tasks. I prefer a two-pronged approach: an objective ranking based on maintenance cost plus a more subjective ranking from system users. I combine these two values to see which systems are most in need of replacement or enhancement. For example, if a system is critical to business operations and has moderate maintenance costs compared to its return on investment for the company, then the best approach is probably to maintain it rather than replace it. A system that is in general but not critical use and is also inadequate for the task may be a good candidate for replacement.

Maintaining legacy systems

If your technology audit indicates that a particular system is critical but too expensive to replace, then you will want to maintain that system. Your objectives will be to keep costs as low as possible but still provide enough system flexibility and enhancements to meet new business needs. Based on both my personal experience with large legacy systems and published information,² I can recommend the following:

- **Establish special teams for maintaining legacy systems.** Clearly, senior personnel with the greatest experience and system knowledge should form the core of these teams. However, a maintenance team should never be entirely dependent upon one person (senior or not) for complete (or near-complete) system knowledge. That leads to my next piece of advice.
- **Make sure that system information resides in several heads.** If only one employee understands a critical system, that person might try to hold the company hostage and demand ridiculous perks to remain on staff. This can easily happen if the original system team was very small, and only one or two members stuck around to see the final product. In one consulting situation I encountered, the single knowledge holder simply didn't trust anyone else to handle critical sections of the system. Left unattended, situations like this create dangerous dependencies; the organization might be left "high-and-dry" if the individual moves to another position, retires, or even dies.
- **Leverage the experience of senior personnel.** This is especially important when these people have specialized knowledge that would take other team members a great deal of time to acquire. Older team members can offer invaluable insight into the original intent behind the system, hidden exception handling, and the overall life-history of repairs, extensions, and integrations. It is a failing of our culture that older workers are not given the respect they are due,³ although many organizations have recently made encouraging cultural changes, instituting ways to reward and retain older employees.
- **Leverage the language expertise of senior personnel.** The languages that old systems use have fallen out of general use, and younger employees see little value in learning them. Consequently, senior personnel who know these languages are increasingly scarce and increasingly valuable. Do you recall what happened when we had to upgrade legacy system dates from two to four digits before Y2K? Companies scrambling for developers with COBOL expertise had to coax people out of retirement and compensate them handsomely. The lesson: If you have people with COBOL, FORTRAN, or BASIC expertise within your organization (and systems written with these languages), treat them well!
- **Let senior personnel mentor newer staff members and learn to use new tools and technologies.** On the best maintenance teams I have worked with, more experienced team members initially took the hardest requests but over time began moving these responsibilities to younger team members. After several months, all team members were able to handle any request, and the senior personnel could take on other, high-profile projects without placing the original system at risk. This collaboration had another benefit: Younger team members gained respect for the older ones and also discovered their willingness to learn how to use the latest tools and technologies. This paved the way for pairing younger and older staff members on other projects that reversed the mentoring role -- so that senior folks could learn from the junior people. It also enabled senior staff to document their system

knowledge via models and other artifacts rather than simply passing it on verbally.

- **Rotate team members so no one takes on the full burden of maintenance requests.** Maintenance has a reputation for being dull and uninteresting. In my experience, maintenance personnel would prefer to be working on the newest technology rather than making sure monthly budget reports are run on time. However, maintaining a system is a great way to learn how to integrate it with others. What better way to understand system idiosyncrasies than to spend time fixing issues? Again, having senior personnel mentor younger, less experienced staff members is the most efficient way to transfer system knowledge. And that brings up the next bit of advice.
- **Create a comprehensive unit test suite.** If the system does not already have a comprehensive unit test suite, creating one should be a primary goal after the maintenance organization receives the system. As outlined by M.C. Feathers in his recent book, *Working Effectively with Legacy Code*,⁴ an automated test system helps the team gain a better understanding of the system, guarantees that repair and enhancements will not break existing functionality, and permits system refactoring to reduce design degradation over time. A number of test harnesses are available for systems built using Java or .NET (i.e. JUnit and NUnit), but for older systems you may need to create one from scratch. The tests should isolate functional areas so that you don't need network resources (e.g., a database) to run most of them. They should also be as low-level as is reasonable. That is, you should test interfaces with data that generates a success as well as exceptions, including hard-to-generate exceptions. A solid test framework can help you increase the speed and accuracy of your maintenance development and make the system more stable over time. Again, senior team members are invaluable resources. They can guide the development of a unit test suite by identifying high-risk areas in the application -- those that will benefit most from a comprehensive test suite.
- **Perform modeling, baselining, and testing.** To preserve and maintain company systems -- and particularly to modernize them -- an organization must perform three ongoing tasks. First, create current system models to guide development efforts. Second, create system baselines, so that you can back out of detrimental changes to a stable configuration. Third, develop automated testing that will reduce overhead and decrease the likelihood of breakage with the addition of new functionality.

In the context of all these critical maintenance functions, providing training and development on automated tools and new technologies for mature staff will benefit the company three times over: once by giving senior workers an incentive to remain with the company and support the existing infrastructure; once by enabling these workers to produce system models and testing artifacts that everyone in the organization can access; and once by modernizing a capable workforce to prepare for eventual system replacement.

Two cases in point

To understand the real value of resident experts in maintaining legacy systems, let me tell you about two cases I worked on. The first was an all-around success. The other taught me valuable lessons about the dangers of over-reliance on outdated systems and older workers.

Senior staffer's assessment information guides legacy upgrade. On one assignment, I was asked to research and document the business processes of a moderate-sized financial management group that had become dependent on an internally developed, twenty-year-old, PICK Universe-based financial tracking system. The system had a client interface that was essential to business operations but needed to be replaced with a Web application that would permit clients better access to research and track their holdings. This involved creating a data access layer to retrieve the client data (held in the proprietary PICK format), and then developing a telnet-based screen scrape interface⁵ to use the underlying system algorithms.

To ensure that we created a correct replacement for the client tool, I had to interview many of the organization's personnel. Remarkably, one of the original software system developers was still with the organization. This person was a gift; he offered invaluable information on key system algorithms and structures that had taken many years to develop and could not easily be replaced. I relied heavily on his expertise to understand how each part of the business operated via the legacy system. Then, we used this

information to construct a series of business use cases that drove development of the replacement client system. The result was a successful launch that both satisfied the company's current clients and served as an excellent marketing tool for potential clients. Had I not been able to tap into the senior system expert's knowledge, the risks associated with this project would have been far greater.

An unhealthy dependence on senior staffers. This is a cautionary tale. On another assignment, I spent the better part of a year working with a large travel itinerary processing company whose core business depended on an extremely large, older legacy system. I saw directly how difficult it is to maintain such a system. For example, it took thirteen months to introduce a single bit change to a core data element; two weeks to perform the change, and then weeks more to regression test the application. This was because the organization had developed a sophisticated deployment process (including unit tests) to avoid introducing system errors. Error avoidance was crucial, because system performance standards were extremely high: greater than 99.999% availability in real-time. To maintain the system, the IT organization relied on older personnel familiar with the system's assembly language and data structures. In fact, these professionals tended to shuttle back and forth between this company and its largest competitor, which had a similar system.

This is where I learned that it is *possible* to ensure a huge, old, mission critical system against failure, but the cost is extremely high. I saw how vital it was for companies with these systems to hold onto their senior personnel for the short term, because attempting to bring younger personnel up to speed only increases maintenance costs, and willing trainees are hard to find. I urged my clients to prepare for the day when they would be forced to replace their mammoth systems, suggesting techniques I will discuss later in this article. But first, let's look at another aspect of maintaining legacy systems: integration.

Integrating legacy systems

Sooner or later, every system that is critical to business operations will need to interact with other business systems. Perhaps a billing engine will need to communicate with the systems for a new service offering. Or an inventory system and order management system will need to communicate about available merchandise. If these systems were built at different times, use different languages, and run on different platforms, the system integrator will face significant challenges, as the respective system designers may never have anticipated the need for such interactions.

Again, senior personnel with in-depth expertise on one or two systems are a great help in such situations. They can either create a new interface for the integration effort or provide information on existing interfaces that teams can leverage to reduce the development effort. And they let you reduce or eliminate the risk of problems caused by simple ignorance about the system. Just as for system maintenance, an integration effort will be more successful with a mixed team. Senior team members can feed the integrators "inside" information that helps them maximize their efficiency. They can also gain enough experience with the new integration software to support it after deployment.

Approaches to legacy system integration

The specific role that senior employees can play in legacy integration depends in part on the nature of the integration.

Gaining access to an application's underlying business data. This is one common motivation for integration. For example, you may want to create a sales support system that queries an existing customer database to verify customer identity. The simplest approach is to query the database directly, rather than going through the legacy application. However, this may be impractical if the database is not accessible via a standard mechanism (e.g. SQL), if the underlying data structures are embedded within the legacy application itself, or if the legacy system is performing complex data integrity algorithms. In such cases, the only way to access the data may be via the legacy application's user interface. This technique, known as "screen-scraping," is a common way to access data stored in mainframe ("green screen") applications. To set up a screen-scraping interface, you must determine the exact command flow necessary to generate the required screen and account

for potential error conditions.

Senior team members who have used the mainframe application will be an invaluable resource in this instance. They will know what system commands to use and be able to detail possible responses to a request.⁶ In addition, by training the most knowledgeable individuals to use automation modeling tools, the organization can create a win-win situation: Workers will gain valuable, highly marketable skills, and the company will reduce its dependence on individuals by documenting existing systems.

Another way to access legacy application data is through specified API calls, such as a defined byte stream. In this case, the system data is available in a well-defined format: Each byte -- and in many cases each bit position -- is defined. To read and write to these kinds of interfaces, you need to write a bit reader and a bit writer that will permit parsing of the byte stream according to the data specification. In many cases, this byte stream is not well documented or may have changed since the last revision; senior team members should be of great help in determining the exact data structures.

Gaining access to an application's underlying business logic to perform complex operations. This is a second major motive for a systems integration project. The operations you want to perform may involve extremely complex algorithms that would be expensive and difficult to replicate (such as a fare calculation algorithm for a travel system). Or the operations may be beyond the scope of the current application (such as billing periods or tax calculations for a provisioning system). In such instances, you may require a more sophisticated mechanism for interaction between the two systems. For example, it may be necessary to implement a CORBA (Common Object Request Broker Architecture) interface to permit interoperability. Alternatively, it may be possible to create a direct connection using a native language call, such as with Java or .NET. Also, certain new technologies allow for simplified XML/SOAP-based method calls -- if the maintenance team can modify the legacy system to support such operations. Again, senior personnel are the people most likely to make these modifications successfully.

Case in point: Resident experts guide a legacy interface integration

Several years ago, I was involved in a complex integration effort to support a travel Web site deployment. The Web application had to interact with a fare quote and travel booking system that had been in production since the early seventies. The only available interface relied on defined byte stream data and offered little support for complex data operations. To perform the integration, we had to understand how each data element interacted on the host system to form a viable travel data construct. To determine this information, I turned to the organization's system engineers, most of whom had used the system for twenty to thirty years. These people were able to guide my team through the interface not only at the defined message level, but also in connecting messages together to form a meaningful transaction. Without their assistance, we would have been hard pressed to produce the necessary integration subsystems in a correct and timely manner. *With their assistance*, we were able to meet our schedule and enhance a system that continues to process tens of thousands of transactions each year.

Replacing a legacy system

In addition to the successful legacy system maintenance projects that I've described in this article, I have also worked on many legacy replacement projects. Eventually *all* software systems need to be replaced, whether the driving forces are rising support costs, limited functionality, competitive pressures, or loss of key support personnel. In many organizations, there is great resistance to replacing systems that have outlived their usefulness. In fact, the longer a system is in use, the less likely a company is to replace it. There are three main reasons for this:

- The system has demonstrated its value.

- Company personnel are familiar with the system's capabilities and limitations; they have devised workarounds for incomplete/flawed operations.
- Critical business processes depend upon it.

The first reason is self-explanatory. The second reason relates to a strong human tendency. Change is never easy, and we often choose the "devil we know" over trying something new, even though it might be better. Knowledgeable senior staffers who have an in-depth understanding of the old system may also have a strong desire to maintain the status quo. They may be worried about losing their positions or authority when the new system is introduced.

The third reason is complex, because companies tend to integrate multiple business systems over time, such as flow-through provisioning for a telephony network and its associated billing systems. This produces a more efficient, less error-prone process than a manual operation, but it also creates greater dependency upon the software systems involved.

To overcome individual resistance, you need to reassure the affected personnel that they will be equally important to the new system and involve them in every aspect of its creation. Even if some people are unwilling or unable to "migrate" to the new system, you can assure them that their skills will be valuable to the company for some time. System replacements typically take years, so they will have an opportunity to find other uses for their abilities.

When an organization does finally decide to replace an older system, it must build a new system that is at least as capable -- if not far more so. Unfortunately, by the time companies get to this point, typically most of the organizational knowledge about the system, especially system requirements, has long since vanished. So the first step in creating a reasonable replacement system is to discover what you already have.

Discovering current functionality

As opposed to the overview you can obtain with the system audit we described earlier, this step yields an in-depth discovery of *all* the system functionality, including algorithms, data structures, error handling, and external dependencies on other systems. Discovering the current functionality in a legacy system is a far from trivial task. Any system that has been around for awhile will have accumulated many "specialized" functions -- such as one-off user shortcuts in the interface, or a complex algorithm for calculating tax withholdings -- which have never been documented.⁷ However, again, senior development team members will be able to tell you just about everything you will need to know, especially about undocumented, "hidden" functions. By making allies of these individuals, you can greatly reduce your project risks.

A particular area for discovery is the validation of data elements. Data validation is an often overlooked aspect of older systems. As it is almost always driven by business data relationships, it is best detailed in a business domain model. Figure 1, based on one of my projects for the aforementioned travel reservation company, depicts a limitation imposed by the legacy reservation system: Only seven passengers may be associated with a given itinerary. This limitation was not obvious from the structure of the data elements; I discovered it only with the assistance of an expert who had worked with the system for more than twenty-five years.

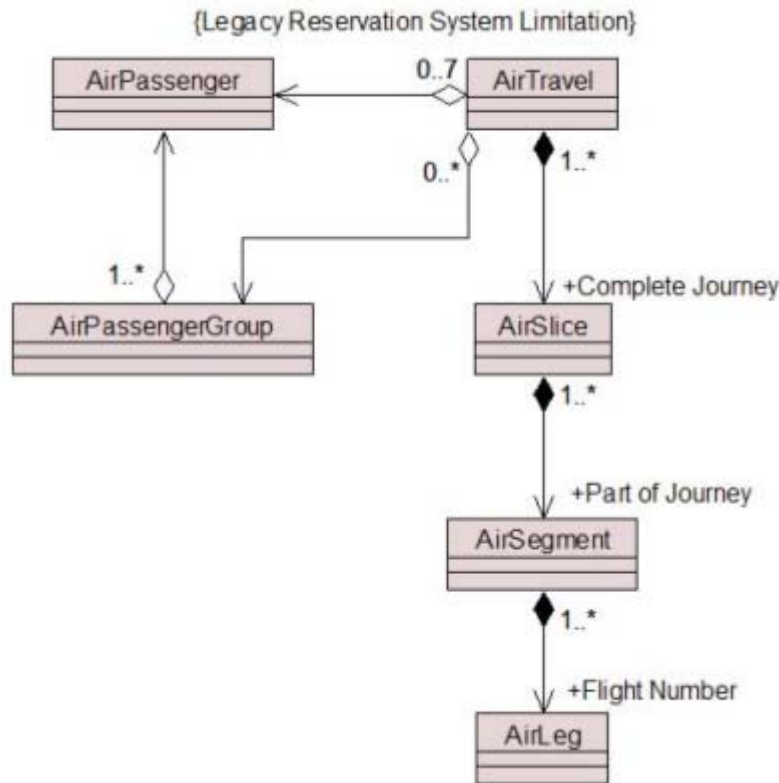


Figure 1: Model of a "hidden" limitation within a legacy system

I have found these models to be invaluable for capturing complex data dependencies. They also provide a visual mechanism to discuss business entity relationships with business personnel who are knowledgeable about the business domain but not necessarily technical.⁸

Discovering exception handling and workarounds

Information about exception handling is also critical to replacing an existing legacy system. Over time, users will turn up rare but important failure conditions that must be corrected. Typically, teams perform these corrections quickly, with little thought to changing the corresponding documentation, so few people know the exact purpose of a section of error handling code. However, there is risk involved in replacing a system without at least identifying the conditions the exception code was meant to handle. In fact, this is a major reason why organizations hang on to older systems longer than they should. Senior managers worry that not everything will be successfully transferred to the new system and failures will be inevitable.

Adding to this difficulty is the way in which each system handles errors. Some languages (such as Java, C#, and C++) have built-in, low-level error handling but few if any have structures for business-logic-level error handling. To handle cases in which one business transaction must be completed successfully for another to be valid, developers must create unique code. The code for handling such conditions is rarely built in a uniform way; so again, the analyst must rely on those people most familiar with the system to interpret the code.

Error handling also includes situations in which a system should have been able to recover from a problem condition, but actually required manual intervention. IBM is currently helping clients to construct autonomic computing systems that are more self-correcting and able to recover from exceptional situations. For example, if an external interface component is unavailable, the system should be capable of automatic failover to a back-up system, notification of the error condition, and correction of the condition when the primary interface is again made available.⁹

In addition to error handling, many senior workers have found ways to perform tasks with a system that has functional flaws or omissions. Such techniques are often called *workarounds*, because users must step outside

the boundaries of intended use to perform a task. These techniques should also be captured, since they are critical system features that were either never developed or developed incorrectly/incompletely. Equipping senior workers with an automated requirement tool will enable them to document these techniques in a form that can easily be translated into a model for the new system's design.

Take an incremental approach

After discovery of the detailed system requirements, the next step is to replace the legacy system with a new (hopefully better) one. Obviously, the more complex your legacy system is, the higher the risk of replacing it and the longer it will take. Many legacy systems have developed a large number of possibly unrelated functional areas. Replacing these systems is much like trying to defeat a many-headed monster: It is difficult to avoid breaking something while you are replacing something else. The best approach to this problem is divide-and-conquer, replacing smaller subsystems one by one rather than attacking the entire legacy application in one go. What at first seems an insurmountable task that requires untold commitment and energy will, over time, become a manageable project.

In addition, this approach will pay great dividends. Smaller deliveries allow for faster course correction, illustrate progress, and establish a pace that you can maintain for a long period of time. Just as a distance runner sets and holds a pace for the entire race, so should a development team plan for incremental discovery, building, and integration of the system replacement. By demonstrating consistent progress, you can overcome resistance, lower technical risk, and maintain team enthusiasm far more effectively than you can with "big-bang" development (i.e., attempting to replace the entire system in one piece).

Clearly, senior personnel with intimate knowledge of the systems slated for replacement should play a pivotal role in the project. An organization that does not recognize the unique value of these individuals for new system development as well as legacy system maintenance may find that it has wasted a few million dollars on a new system that is not as effective as the old one. By actively working to retain these senior team members, organizations can dramatically reduce the costs associated with legacy system maintenance as well as the risks associated with system integration and replacement. Providing incentives in the form of personal growth strategies, such as training in the use of automated tools, involvement with new company projects, and other continuous learning opportunities, will help entice older personnel to remain within the company, share critical system information, and make invaluable contributions to system modernization projects.

Notes

¹ K.Dychtwald, T. Erickson, and B. Morison, "It's Time to Retire Retirement." *Harvard Business Review*, March 2004. p. 17-25.

² For example, M. E. Bays, *Software Release Methodology*. Prentice Hall, 1999.

³ Dychtwald et al., *Op. Cit.*

⁴ Published by Prentice-Hall in 2005.

⁵ This is a technique that permits access to system functionality when there is no other programmatically accessible interface. I will describe it later in the article, when discussing integration to legacy systems.

⁶ When you use screen scraping, you may also need to filter out screen formatting codes that are sent along with the data stream during the parsing operation.

⁷ For techniques on recovering system requirements, see my *Rational Edge* article on "[Requirements Archaeology](#)."

⁸ For more information, see Christophe Tournier's *Rational Edge* article on "[Modeling Guidelines for Legacy Applications](#)."

⁹ I also built a system that was able to perform these exact tasks. When an interface was detected as unavailable the system went to "failover" mode and issued an alarm (email). When the condition resolved, the system automatically reconnected to the primary interface without any intervention.

About the author

As the principal architect for software architecture consulting firm BioLogic Software Consulting, Ben Lieberman provides clients with training, mentoring, and practical advice on architectural issues for software systems. He became what he calls a "consulting architect" more than five years ago, having arrived at this role via a circuitous route. Prior to that, he had spent nearly ten years as a research scientist involved in numerous software projects relating to biological laboratory research data collection and analysis (e.g., bioinformatics). He holds a B.S. in molecular biology from the University of California at Los Angeles, and a Ph.D. in biophysics and genetics, from the University of Colorado.

Rate this content

Please take a moment to complete this form to help us better serve you.

Did the information help you to achieve your goal? Yes No Don't know

Please provide us with comments to help improve this page:

How useful is the information? 1 2 3 4 5
(1 = Not at all, 5 = Extremely useful)

 **Submit**

[↑ Back to top](#)