Cash Cow in the Tar Pit: Reengineering a Legacy System

W. STEPHEN ADOLPH, British Columbia Institute of Technology

We all have them: those hand-me-down systems bequeathed to us by our forebears, those old machines that have generated the lion’s share of company revenues for the past decade — and that none of us want to touch for fear of being exiled to some program-maintenance hell.

Software engineering is now more than 30 years old and has generated billions of lines of code. Many of the systems designed in the ’60s and ’70s are still with us in some form. How many of them still have bizarre restrictions such as 64-Kbyte segments or 80-column records? Eventually, our users clamor for features that cannot be grafted onto the old system, constant patching makes the system’s reliability questionable, or the host-computer manufacturer discontinues that product line. Eventually, we must wade back into the tar pit and build a new system.

Many old systems are still with us and burden us with baggage. But replacing them, particularly when they are legacy systems, is not as straightforward as it seems. The author imparts lessons learned on a legacy-replacement project.
RISK AND LEGACY

The enemy of all projects is risk: the probability of project failure. Risk may manifest itself as exceeded budgetary constraints or a failure to satisfy functional requirements. Factors affecting the level of risk include development-staff competency, requirements volatility, tool capability, target-platform availability, and the reliability of third-party software. Successful projects reduce risks to manageable proportions.

Identifying and managing risks is part of the software project-planning process. However, when replacing a legacy system, software professionals often underestimate risk because they assume the legacy system is one of the organization’s core competencies and that it represents a set of stable requirements. This perception is true only if the replacement-project development team understands the legacy system and uses it to reduce the risk associated with uncertain requirements. Otherwise, schedules developed under this assumption of design reuse are erroneous. Even if the legacy system’s design is reused, it is unlikely that the original methodology and tools will also be reused.

OUR CHALLENGE

In 1989, when I worked at Alcatel-SEL, one of our largest clients came to us with a problem: They were planning a major extension of their transit railway, which was already taxing the limits of their legacy system’s host computer. Our initial estimates predicted that the cost of adapting the legacy system to satisfy the client’s new requirements was comparable to the cost of replacing the system. We were highly motivated to push for a replacement for other reasons as well: It was becoming very difficult to hire people who were interested in maintaining a 20-year-old assembly-based system, and we had a pent-up dislike of the tedious work the legacy system required.

In 1990 the decision was made to replace the legacy system. Our mandate was to:

- resolve the performance and memory limitations by moving the system to a 32-bit hardware platform,
- modernize the software by reorganizing the databases and removing the operating restrictions imposed by the old host’s memory constraints,
- improve development-staff productivity by using modern development tools and high-level languages, and
- use a development methodology that complied with relevant standards.

Our economic justification for this project — that the cost of replacing the legacy was comparable to the cost of adapting it — was the major constraint on the project.

Out with the old. Our client’s legacy system was an enhancement of one originally installed in 1971 to maximize the use of rail tunnels between Switzerland and its neighboring countries. The system is responsible for safely operating automatic trains on urban mass-transit railways. The system guides trains to a location on the guideway called a target point. As the target point is updated, the system leads the train to a new location, much like a carrot on a stick leads the donkey forward. This approach to railway signaling is known as the moving block.

There were several problems with the old legacy system.

- The host computer was a triplex of General Automation GA900s, which were becoming difficult to obtain and service. In addition, the system was being deployed to handle increasingly larger transit systems, and the number of trains the system could manage was limited by the host computer’s processing capacity. Thus, the capacity of billion-dollar transit systems was limited by a few tens of thousands of dollars worth of old computer hardware.
- Memory limitations imposed by the host computer limited the information that could be stored about a train and the guideway; previous developers were thus forced to make assumptions about the behavior of trains to minimize data requirements. As a result, implementing new features was often difficult, if not impossible.
- Maintaining code was difficult because there were few programmers who had ever heard of — let alone programmed in — GA900 assembly language. The host also had a primitive development environment: just a simple line editor, assembler, and linker.
- After years of evolutionary development and bug patches, much redundant code had accumulated in the system. For example, conditions were hard-coded rather than data-driven; if a change was made to a single condition to accommodate a new feature, an exhaustive search was required to find all conditions that made the same test.

There were also the inevitable smaller problems that afflict older, memory-limited real-time systems. For example, train names were numeric and based on the train’s database index key or the memory efficient — but cryptic — command language.

Staffing. The success of the legacy system was a major problem for our new project. We were bidding on and winning contracts on the basis of the legacy system, and the delivery dates for these contracts were well ahead of that for the new system. The immedia-
Figure 1. The 12 development stages, divided according to function. The first four stages were performed sequentially.
ment team experienced in the application domain. To create a schedule that assumed the design team was familiar with the application was to ignore these cost drivers and therefore underestimate the schedule risk.

**CAPTURE REQUIREMENTS (READ THE FINE CODE)**

The most important output of the system analysis was a new data model that described the relationship between a train and the guideway elements. The original system’s train and guideway data structures were designed to accommodate the memory restrictions of the host computer and so were highly encoded and compact. This encoding made the addition of new features difficult at best — and in many cases impossible. Our new database unpacked the data and explicitly represented the relationships between data items rather than relying on the restrictive and implicit assumptions made by the code.

**Ripple effect.** Although these changes were necessary, their effect rippled throughout the entire system. The new data model made many of the legacy system’s modules obsolete. Thus, a designer had to distinguish between actual train-management code and the coding “tricks” used to access the train data. This task became the software equivalent of separating the wheat from the chaff and effectively eliminated the possibility of a direct translation from GA900 assembly to C. The focus of our design effort became the capture of the essential train-management algorithms. This led to our second major problem.

A significant number of design packages failed their design reviews because they had not properly captured all the functional requirements represented by the legacy code. Most designers were attempting to design their part of the system with only a partial set of requirements: the functional requirements for the new system and the design documentation for the legacy system. They would rarely look at the legacy system’s code because they found the task of reading old assembly tedious. Naturally the problem of human nature creeps in: If something is difficult, tedious, and slow, we’ll try to avoid doing it. Unfortunately, as with most legacy systems, there was a wide gulf between the documented description of the system’s function and its actual function.

**Harsh measures.** We chose a brutal solution: All staff were required to read and comprehend the legacy code. We told them that it was an important part of their job to understand the code they were translating and their understanding would be reflected in their job-performance review. We took three steps to make reading the original system’s code easier:

- The principal engineer mapped the existing train data structure in the legacy system to the new data model. This was one of the most useful design documents prepared during the project.
- We cobbled together a few Awk scripts to generate a cross-reference list of modules, function calls, and data structures. All macros were listed and commented. Although our CASE tool had a reverse-engineering component, it did not work for assembly code — and least of all GA900 assembly code.
- Principal engineers started giving lunchtime seminars on the theory of railway signaling and the history of the legacy system. We even purchased a visual aid: a small electric train set to help us step through and understand the legacy code.

**CONTROLLING CHANGE (JUST SAY NO)**

A significant problem that often afflicts projects is that designers go beyond the scope of their mandate. In our project, the redesign of the train database gave some designers the impression that they were free to redesign other parts of the system. However, although there were many features in the old system that were considered ugly, they were fundamental to the system’s operation.

For example, several weeks were spent exploring how to create a new coordinate system for trains. The existing system employed two different coordinate systems: one for regular train movements and one for close-up operations such as coupling and uncoupling. The use of two different coordinate systems was one of the tricks that had been used to squeeze the legacy system into the host computer’s limited memory. This trick really offended our computer-science sensibilities, and we proposed a new consistent coordinate system that would have been better from a purist’s point of view. However, this new coordinate system was a serious risk from a project point of view because we would be throwing away 20 years of operating experience to adopt something that we thought was conceptually better.

Uncontrolled change can increase project risk, especially in a legacy-replacement project. By changing exist-
ing system features, we were throwing away a working, accepted feature and replacing it with an untested feature that dramatically increased the risk to the project. Although new features are often much more elegant and powerful, they are also much more complex and require more development time. New, complex features also have new, complex failure modes. We knew the shortcomings of the existing system, but we were now proposing to create a new mechanism whose operational behavior was uncertain.

Fortunately, the project manager put a stop to this improvement effort and set up an overriding project directive: No features of the legacy system were to be changed unless changing the feature reduced the cost of implementing the new system.

This was probably the most important decision made to insure the project's success. As a side effect of this decision, we were able to reuse all of the legacy system's existing test plans. The only additional test plans we had to write were the test plans for the enhancements we added to the new system.

**CASE**

**A FOOL WITH A TOOL ...**

An often unaccounted-for side effect of a legacy-system replacement project is the replacement of the legacy system's development methodology. The risks accompanying this side effect are often grossly underestimated or even ignored. In my opinion, most of the serious problems we encountered resulted from underestimating this risk and introducing CASE tools.

**Tool selection.** We acquired Cadre's Teamwork CASE tools for structured analysis and structured design. We chose the Teamwork tools because they were the best available PC-based tools. We wanted to use PCs because most of our staff had PC experience while only a few had worked on Unix. Also, Unix machines were two to three times more expensive than their PC counterparts, and personal productivity software for Unix—such as word processors and project-management tools—often cost 5 to 10 times more. At the time, however, PC technology was not what it is today; in retrospect, we might have been better off going with Unix and avoiding time lost in developing tools and patching around problems in the PC-network.

Overall, the CASE tool was invaluable in coordinating the designs of 15 individual developers and led to a higher-quality system. However, we experienced far more startup problems than we had anticipated.

**The experience factor.** First, only two members of the development team had CASE experience and had used a design methodology such as Hatley/Pirbhai. We almost naively believed that the transition to CASE meant little more than purchasing a product and sending the development team away for two weeks of training. The result was that many designers made all the classic errors of novice users in their early models. For example, the level of formality and detail specified in the mini-specs varied widely. It was common to see a statement of the form:

```
if train is invalid then call fail_train.
```

With no specification of what an "invalid train" was in this context, how could this statement be reviewed or understood? Never did Frederick Brooks' words "be prepared to throw one away" ring so true. Numerous early designs had to be discarded and redone—something we didn't account for in the schedule.

**Checking consistency.** During our early system specification, we encountered the second problem with our use of CASE tools: semantic inconsistencies in the data dictionary. The CASE tool had a consistency checker that the development staff used to verify the data dictionary. What the staff failed to realize was that the tool could only verify the syntax and hierarchical derivation of data items. It could not verify if the contents of the data dictionary made semantic sense.

This problem was caught during an early review of the data dictionary by a developer who had CASE-tool experience. It took two developers more than a week to revise the data dictionary, and another week for all developers to revise their designs to use the new data dictionary. To prevent a recurrence of this problem, an appendix was added to the development methodology describing how entries were to be defined in the data dictionary, and a naming convention was created.

**Packaging problems.** The third problem we encountered involved the procedure for extracting designs from the CASE database into review packages. A review package consisted of the structure charts for all modules included in the package, the mini-spec pseudocode for each module, the relevant portions of the data dictionary, functional-requirements cross reference, and the package test plan. Preparation of the package was horrific and time-consuming. The interface between the CASE tool and the desktop publishing system required an average of four hours of manual effort to define the layout, export the data from the CASE system, then edit the package before it could be distributed. This problem became so bad that at one point four junior programmers were spending all their time preparing
packages for reviews.

We pointed out this problem to our CASE vendor, who said that they had a wonderful new interface for extracting documentation from the database — one that wouldn't be available for their PC-based system for at least six months (translation, one year). It was quite clear that the PC version of the CASE tool was the poor second cousin of its Unix counterpart.

We eventually fixed the problem ourselves by hiring a junior programmer to write some Postscript filters and some “glue” code for generating our review packages. The output of our home-brew system was less pretty, but we were able to reduce the effort required to produce a package by an order of magnitude.

Managing CASE. A final problem with the CASE tools was that no one person was assigned the responsibility for managing them. The result was that a senior designer with CASE experience took it upon himself to be responsible for the tools. The problem, of course, was that CASE-tool maintenance was not part of his job description and he began to fall behind in his scheduled design work. Although much pressure was placed on him to catch up on his design work, no relief was offered for the burden of maintaining the CASE tools. The problem was not resolved until the designer quit, and a decision was made to hire a toolsmith.

We should not have been surprised by these startup problems. Studies have shown that the benefits of CASE are not observed until the later stages of the project or even until the next project. CASE tool vendors make varying claims on the productivity improvements that result from using their tools. However, that higher productivity is only obtained after the designers become proficient with the tool and the methodology. In the first project with a CASE tool, our experience confirms the experiences of others: You should expect at least a 30-percent decline in designer productivity on your first project.

ANALYSIS

Our project was successful in that it delivered the system with a very high level of software quality, satisfied our customer, and was reliably cut into operation. The most significant factor leading to this success was the project manager's emphasis on reducing or eliminating the risk associated with the introduction of new features and his keeping the design teams focused on the project's mandate. However, the software was delivered late and missed its initial budgetary targets.

Many of our problems during system design were due to the development staff's lack of familiarity with the application domain and provided proper training. In new-systems development, this problem is not as severe because the research required to learn the application domain is part of the requirements-analysis process. In a legacy-replacement project, the requirements-analysis cycle is often compressed based on the assumption that the design information represented by the legacy system can be quickly captured. Unfortunately, this is only true if a significant number of project analysts are familiar with the application domain and the legacy system itself.

Clearly, we failed to adequately assess the risk of the new methodology and tools. There were no schedule adjustments made for the risk of the new tools; in fact, the schedule was compressed because it was assumed the development team would achieve productivity gains. Our experience with the development tools lends weight to the old saying, “Everything looks good until you know something about it.”

LESSONS LEARNED

What lessons did we learn?

Lesson 1: The legacy system is useless for reducing risk if the design staff does not understand it. We should have recognized the new staff's unfamiliarity with the application domain and provided proper training. In new-systems development, this problem is not as severe because the research required to learn the application domain is part of the requirements-analysis process. In a legacy-replacement project, the requirements-analysis cycle is often compressed based on the assumption that the design information represented by the legacy system can be quickly captured. Unfortunately, this is only true if a significant number of project analysts are familiar with the application domain and the legacy system itself.

Lesson 2: A legacy system cannot reduce the risk of methodology replacement. Be prepared to train your project team and progressively ramp up your design effort. The staff should have had much more training in the development tools and methodology. We should have used the CASE tools to capture the existing system's design before we started to use them to design the new system. This would have helped the new designers learn the existing system, given the designers' experience with the CASE tools, and, as a bonus, created up-to-date design documentation for the legacy system.
Lesson 3: Do not assume the CASE tools will look after themselves. For a project of any significant size, there should be a dedicated toolsmith. Having a designer manage the tools is a waste of time, especially if that designer has significant problem-domain experience.

Lesson 4: No matter how competent the design team is, there is a lot of history hidden away in the legacy. Make sure the design team has access to the experience they require to learn the legacy and do their jobs. The success of the legacy system and the immediacy of our contracts made it too risky to transfer experienced staff from the legacy system to the replacement project. While this decision reduced the legacy system’s risk, it dramatically increased the risk on the replacement project. A compromise solution would have been to temporarily relieve the engineering manager of his administrative responsibilities and make him project architect so he could mentor new staff. Like many engineering managers who are promoted from within, he had the most experience and knowledge regarding the legacy and the problem domain. Unfortunately, 75 percent of his time was taken up with administrative tasks.

In general, software-development managers assume that reengineering a legacy system is easier than building a new system from scratch. We tend to forget that in the case of a legacy replacement project, the only given is a set of formal requirements in the form of the existing system’s code. The software still must be analyzed and understood by the development staff, and a new design must be created, implemented, and tested. If the design methodology is changed, or if major requirements are added, the replacement project may take considerably longer.

If part of your justification for replacing a legacy system is that the legacy system will reduce risk (by providing stable requirements, for example), make sure that you schedule an analysis of the legacy system and that you actually use the legacy system’s design. New designers are not instantly going to understand the system just because you have the legacy code and documentation. Also, new designers may not examine “that old piece of crap” if not prodded.

In addition, clarity of purpose must be maintained throughout the project. Everyone on the project must know why the decision to replace the legacy was made and what the project objectives are. Is this simply an effort to port the system over to a new platform, or to do a complete redesign? This is good advice for any project, but for a legacy system it is even more important to ensure this because programmers’ dislike of the existing system and a desire to “strut their stuff” can increase their impulse to go wild.

If part of the replacement process includes replacing development methodology, you should ensure that the project schedule accommodates the learning curve. Do not assume that the addition of the new tools will reduce your development time; in fact, assume that it will take longer.

Whatever the reason, more often than not, the new replacement team collides with the egos of the original developers (“We can show those old guys”) But replacement developers should always remember: The system built by those old guys works and has been bringing in revenue for years. You are proposing to throw that cash cow into the tar pit of new systems development.

Acknowledgments

I thank my colleagues Arthur Bailey and Russ Trent for the time they took to review this article and for their recommendations.

References


W. Stephen Adolph is an instructor at the British Columbia Institute of Technology. His research interests are in object technology, software engineering, and distributed systems. He previously worked developing call-processing and railway-signaling software for MPR-Teletech and Alcatel-SEL, where he was design team leader on the legacy replacement project. Adolph received an MSc in computing science from Simon Fraser University.

Address questions about this article to Adolph at BCIT, Computer Systems Technology, 1770 Willingdon Ave., Burnaby, BC, Canada, V5G 3H1; FAOLSS5@BCIT.BC.CA.