Legacy Systems Management: Facts and Fallacies

[ By Young-Gul Kim, KAIST, Graduate School of Management, Seoul, Korea ]

Whether we want it or not, much of IS human resource is devoted to the maintenance of existing software. What makes software maintenance time-consuming and challenging is that many of the old software are hard to understand, thus hard to maintain. We call them legacy systems. In this article, we define the legacy system and look into IS people's dilemma in dealing with the system. We then classify the legacy systems and introduce techniques to improve them, suggesting a match between the legacy types and the improvement techniques. Finally, we address and clarify common fallacies people hold about legacy systems.

What is a Legacy System?

Despite its ubiquitous usage, the term, legacy system, suffers from ambiguity of its definition. Everyone seems to have his or her own perception and understanding of what a legacy system is. It is commonly understood as “a large and old piece of software that is hard to maintain”. Bennet [2] defined it as “large software systems that we
don’t know how to cope with but that are vital to our organization”. While the name “legacy” implies some age, we do not seem to know how old a system should be to qualify as a legacy. Neither is the size of the software a readily applicable criteria to categorize it into the legacy class. Then, perhaps a less controversial definition of a legacy system can be “a software system in operation that is hard to maintain”. Many are old; built in the 60s and 70s. Majority of them are large, batch processing systems, handling the organization’s high-volume business transactions on mainframe.

Typical problems of such a legacy system are: poor documentation, spaghetti code without good control structure, lack of knowledgeable personnel, and lack of performance and adaptability. Some of these problems are due to the lack of proper management actions while others stem from the technical constraints at the time of their birth. Poor documentation and lack of modularity makes the job of the maintenance programmer, who probably was not involved in the original development project, extremely difficult. On the technical front, since many of these systems were developed in a time when main memory optimization and processing speed were key design considerations, they lack the flexibility and user-friendliness and are unable to exploit the today’s vastly changed computing environment (cheap and huge main memory, optical storage, powerful graphics, parallel processing architecture, etc.).

**Legacy Dilemma**
Legacy systems represent the years, and sometimes decades, of organization's investment in hardware, software and human resource. This type of investment is non-trivial for most organizations. The real asset value of a legacy system, however, lies in the years of accumulated business rules, policies, expertise, and knowhow embedded in such a system. If corporate knowledge is defined as the company's organized experience and knowhow from its problem solving activities, the legacy
system represents the corporate knowledge repository. This is due to the information system's role of acting as an invisible but intelligent agent of the corporate problem solving process. On the other hand, because of the poor maintainability and technical outdatedness, legacy systems are incurring significant organizational burdens. Organizations are suffering from piling application backlogs as the request for new information systems increase while many of its IS personnel are fighting to understand and maintain the legacy systems.

So we have a dilemma. Legacy systems are handling critical corporate transactions in a reliable way. Giving it up or replacing it with a new system will bring not only huge financial burden but also potential managerial chaos arising from user resistance and massive conversion of data, forms, and procedures. At the same time, legacy systems became too expensive to maintain and the rest of the company is hurting from inflexible and unavailable IS services. Assuming that totally giving up on a legacy system is not feasible for most organizations in the immediate future, we expect the solution to be found in improving the legacy system so that it becomes more maintainable.

**Classification of Legacy Systems**

According to our definition of legacy system as "a software system in operation that is hard to maintain," there is a variety of different legacy systems. Some may have been in existence since 1960s while others were developed in 1980s and 1990s. Some are enterprise-wide batch transaction processing systems while others are used within an
end-user department for on-line analysis and reporting purposes. So, before we hurry to find “the” legacy solution, we need to classify legacy systems into several types so that we can apply different approaches and techniques for each type.

Let’s start with the first type, which we will call “first generation legacy systems (FGLS).” FGLSs are the systems developed between mid 1960s and late 1970s, some in machine language but mostly in assembly or early versions of the third generation programming languages such as COBOL or FORTRAN. Most of them are batch, transaction processing systems running on mainframes. They rarely have development or operational documentation and no one in the organization fully understands their innerworking mechanism or nature and purpose of changes made over the years. FGLSs tend to show a very low modularization and suffer from a serious scalability problem to accommodate the growing business demands.

Next, we find the “second generation legacy systems (SGLS)” which were developed from late 1970s and throughout the 1980s. SGLSs differ from the FGLSs in many respects. Unlike FGLSs, many SGLSs do possess some level of modularity and development documentation. Many are used for on-line transaction processing (OLTP) tasks and running not only on mainframes but also on a diverse set of mid range computers. While majority of them were still written in COBOL, some were written in fourth generation languages and are based on early Database Management Systems (DBMS). Despite these differences, they share with the FGLSs the common symptoms of a
legacy system: lack of documentation on changes, lack of knowledgeable maintenance personnel, and uncontrolled redundancy in data and functionality.

Finally, we encounter the “third generation legacy systems (TGLS)” which has a sharply different origin and characteristic from those of FGLSs or SGLSs. TGLSs are the systems developed since late 1980s which suffer or will suffer from even worse maintenace problems than its predecessors. Many of them were developed using the graphical user interface (GUI) development tools in the rapid application development(RAD) style, skipping most of the conceptual and logical level modeling of data and processes. This almost turns the clock of IS development practice back to the first generation. This time, only its maintenance nightmare seems bigger because, compared to the relatively stable COBOL era, the new GUI development tool environments are far more diverse and going through much faster changes in its functionality, structure, and in some cases, existence. Maintenance programmers of the TGLSs will soon assimilate themselves to an unfortunate enduser who is lost in the sea of documents written in different versions of the multiple word processing software without any indexing or document type information. This problem of TGLS is not limited to the systems developed in-house. Companies that adopt the popular enterprise resource planning(ERP) packages may find their state of the art integrated client-server solution quickly turning into a TGLS if they customize it too much without care. I recently observed a Korean firm customizing 40% of the SAP R3 package without the systematic documentation of those changes.
Techniques for Legacy System Improvement

Improving a legacy system for higher maintainability involves understanding of the following four software engineering techniques: redocumentation, restructuring, reverse engineering, and reengineering. According to Chikofsky and Cross’s taxanomy [4], these techniques differ from one another in terms of the tasks they perform and the level of abstractions they address. First, **redocumentation** is “the creation or revision of a semantically equivalent representation within the same relative abstraction level.” Adding program comment lines or reflecting changes of data definition in the original data model are examples of redocumentation efforts. The goal of redocumentation is to reinforce the originally deficient documentation or keep it up to date by tracking the changes after the initial development. Of the four legacy improvement techniques, it is the simplest and the oldest one with the least change impact. All three types of legacy systems can benefit from this technique.

Second, **restructuring** is “the transformation from one representation form to another at the same relative abstraction, while preserving the subject system’s external behavior (functionality and semantics).” Breaking a monolithic COBOL program with generic PERFORM statements into a set of more cohesive modules and CALL statements will be a common example of restructuring. But, restructuring is not limited to the programs. Normalization of relational database tables and the consequent redesign of COBOL’s data divisions will also
belong to the restructuring efforts. The goal of restructuring is to achieve the benefits of modern programming (understandability) and data design (integrity) with the minimal, syntactic level change effort. Among the three legacy types, FGLSs will benefit most from restructuring due to its unstructured denss in processing logic and data design while SGLSs may also benefit to some degree.

Third, reverse engineering is "the process of analyzing a subject system to identify the system's components and their relationships and create representations of the system in another form or at a higher level of abstraction." Creation of a data model from the program's data descriptions or structure chart from the program's procedure division will be an example of the frequently performed reverse engineering tasks. The purpose of reverse engineering is to ease the human user's understanding of the underlying semantics of the subject system by recovering it at a higher, therefore more user-friendly, level of abstraction. Both FGLSs and TGLSs, which are deficient of such high-level models, can be the ideal target systems of the reverse engineering activities. SGLSs may also benefit from reverse engineering but mostly in the form of complementing the original models.

Finally, reengineering is "the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form." Reengineering usually involves some form of reverse engineering, followed by either forward engineering or restructuring activities. So, among the four legacy improvement techniques, reengineering is considered to have the most comprehensive
scope and involves the most challenging change effort in terms of resource investment and project complexity. The main objective of reengineering is to lengthen the life of the target software system by modifying it to become more understandable to human users and more adaptable to changes in the business environment. While both FGLSs and SGLSs may benefit from reengineering, considering costs and benefits of such initiative, SGLSs will be the prime target of reengineering initiatives. Reengineering of FGLSs, despite the visible benefits, will require excessive resource investment for reverse and forward engineering activities due to the enormous program understanding requirement and the outdated development environment. Table 1 summarizes the match between the three types of legacy systems and the four legacy improvement techniques.

Table 1. Match between Legacy Types and Improvement Techniques

<table>
<thead>
<tr>
<th></th>
<th>Redocument’n</th>
<th>Restructuring</th>
<th>Reverse Eng.</th>
<th>Reengineering</th>
</tr>
</thead>
<tbody>
<tr>
<td>FGLSs</td>
<td>**</td>
<td>**</td>
<td>**</td>
<td>*</td>
</tr>
<tr>
<td>SGLSs</td>
<td>**</td>
<td>*</td>
<td>*</td>
<td>**</td>
</tr>
<tr>
<td>TGLSs</td>
<td>**</td>
<td>**</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

where ** - highly applicable,
    * - applicable
Legacy Fallacies

Despite its widespread use among the IS professionals, the term, “Legacy System”, is perceived and treated differently by people with different perspectives and experiences, creating misconceptions and somtimes leading to critical investment mistakes. Based on the previous discussions, we will address and clarify three common fallacies people hold about legacy systems.

Fallacy #1: Legacy systems are symbols of the developer’s past sins!

Many people believe that legacy systems are the result of IS developer’s lack of interest in documentation, structured programming, data modeling, 4GLSs, etc. Rather, the more fundamental problems of the legacy systems result from the management’s inaction in setting realistic project deadlines, providing necessary tools and education, and establishing explicit and clear guidelines and procedures on handling changes of program codes and data definitions. Persevering throughout the years of undercare and countless business requirement changes and still handling bulk of the company’s business transactions, legacy systems ought to be seen as symbols of the success and survival.

Fallacy #2: Legacy systems should be replaced ASAP.

With the advent of new GUI development tools and movement toward distributed computing platforms, young IS blood argue for the total and immediate replacement of their bread and butter legacy systems
either by building a new software or by purchasing an integrated, enterprise-wide IS packages from the vendors. Proceeding without careful cost/benefit analysis of both replacement and improvement alternatives, however, will prove to be a critical and very expensive mistake. Moving from the central host-based legacy systems to the distributed computing environment involves not only the rewriting of software in a new language but also the non-trivial tasks of installing and managing networks, distributed databases, multiple servers, and hundreds sometimes thousands of personal computers. I recently observed a major Japanese advertizing agency, after investing five years and over $100M on replacing its mainframe legacy system with a client-server system of 40 servers and 900 clients, suffering from the unacceptably slow performance and high system management cost. In contrast, US Department of Defense’s reverse engineering of data requirements [1] and the Boeing’s payroll system reengineering project [5] demonstrate the successful improvement of the very large-scale legacy systems.

Fallacy #3: Legacy improvement can be automated with tools.

Emergence of several CASE tools claiming their support for reverse engineering or reengineering of legacy systems misleads IS managers into believing that “running the codes through the tools will automatically recover higher level designs and generate the new system.” This is not true, yet. While many of the tools were able to support the redocumentation and restructuring tasks to some extent, their support of the reverse engineering and reengineering tasks remains at the
basic understanding of the syntactic constructs and far from automatic understanding of the program and data semantics as required in such techniques. According to Biggerstaff [3], design recovery as a core subset of reverse engineering “must reproduce all of the information required for a person to fully understand what a program does, how it does it, why it does it, and so forth. Thus it deals with a far wider range of information than found in conventional software engineering representations or code.”

**Legacy System: Is there a future?**

General McArthur, the Korean War hero, said, “Old soldiers never die, they just fade away.” Same can be said of the millions of the legacy systems around the world. Some of them will disappear soon. But, many of them, when properly improved through the techniques described in this paper, will survive the end of this millennium and continue to be the workhorse of their organizations well into the 21st century. After all, who said COBOL would be dead by the end of 1980s?
References


Author Biography

Young-Gul Kim is an assistant professor of MIS at the Graduate School of Management, Korea Advanced Institute of Science and Technology (KAIST), Seoul, Korea. He finished his Ph.D. in MIS at the Carlson School of Management, University of Minnesota and taught as an assistant professor at the Katz Graduate School of Business, University of Pittsburgh from 1990 to 1993.
Legacy Systems Management: Facts and Fallacies

Young-Gul Kim
Graduate School of Management
Korea Advanced Institute of Science and Technology
[KAIST]
P.O. Box 201, Cheong-Ryang
Seoul, 130-650, Korea
Tel: 82-2-958-3614, Fax: 82-2-958-3604
E-mail: ygkim@msd.kaist.ac.kr
Ms. Vivian Rothschild
Editor, Information Systems Management
Auerbach Publications
Warren Gorham Lamont
One Penn Plaza
New York, NY 10119, USA
Tel: 212-971-5000

April 22, 1996

Dear Vivian:

How have you been? I am very pleased to be able to finish the
“Legacy Systems” paper as scheduled. Length of the paper is about 2600
words including references and title. As you specified, I enclosed a
diskette along with the two hard copies of the paper. Inside the diskette,
you will find three files: legacy.txt, legacy.wp5, legacy.doc. Legacy.txt
is a text file as a back-up. Legacy.wp5 is the wordperfect 5.* file for
windows. Legacy.doc is the Microsoft Word 6.0 file just in case the
above two files do not work on your system.

I certainly enjoyed writing this paper for your journal and hope to
work with you and your journal again in the future. If you have any
questions on the paper, please send me an e-mail at
ykgkim@msd.kaist.ac.kr or fax me at 82-2-958-3604. Thanks.

Sincerely,

Young-Gul Kim
Assistant Professor
Graduate School of Management
KAIST, Seoul, Korea