When a data processing department inherits software

by JOAN R. ZAK
Henry S. Miller Companies
Dallas, Texas

ABSTRACT

This paper discusses some of the problems that occur in dealing with inherited software and some of the basic procedures necessary to manage successfully a data processing department or group that is converting and/or maintaining software that has been imposed on them and that they neither designed nor implemented. A significant constraint is the impossibility of contacting the author(s) for guidance and help.

The two situations described are very different, yet they share some problems. The first involves an engineering company with a relatively scientific, real-time application—the design of a security system for a nuclear power plant. The second is a real estate management company with fairly typical business applications.

This paper does not hope to provide all the answers, but instead to help raise some of the questions that arise when dealing with inherited software.
INTRODUCTION

Every maintenance programmer is faced with the problems of inherited software—trying to fix or improve a program that someone else wrote. However, a whole new level of complexity is introduced when the entire shop is working with software they neither wrote nor designed. In this case there is no one to go to who understands what the system does, no one who might have a clue about what the author was thinking when the program was written. The programmers can have difficulty in understanding some of the broader background issues involved at the time the system was written—issues which may or may not still be applicable. The process is painful, and, if not well managed, can cost the company a lot of time and money.

I have been in two situations where an entire group or data processing department has been working with inherited software; and I would like to share pitfalls, special problems, and my recommendations for working in this kind of environment.

SITUATION 1—THE DISASTER

Background Information

The first situation was an engineering company in Texas, whose sister company had won a bid to build a security system for a nuclear power plant. The sister company folded after working on the project for a year and collecting about 90% of the money. The programmers assured the people who were coming to take over the project that the system was almost complete. In fact, they grumbled that if they had just been given another month or two, the entire system would have been working.

The security system was supposed to run on a PDP-11/34 with 256K bytes of memory and 15 megabytes of disk space. The system was badly underconfigured, since over 2,000 detection and alarm devices were supposed to be connected to it. The relations with the customer were very negative. The customer had been burned once by the other company and was not open to suggestions about buying more hardware.

The engineering company that took over the project had a big problem with lack of expertise, especially in software. They hired, as programming manager, a woman who claimed she had 7 years’ experience on a similar operating system, but who turned out to have very little knowledge about this system. They hired a number of consultants to work on the project, all claiming expertise in one part of the system or another. By the end of the project two company employees, nine consultants full time, a representative from the customer’s company, and a Florida consulting firm were involved.

The Scope of the Project

Documentation consisted of the 300-page requirements document prepared for the customer by another consulting firm, a 300-page functional specification, and a system flow chart. The requirements document asked for things that were almost impossible to do, the functional specification was vague and contradictory, and the system flow chart was incomprehensible.

Of course, there were the programs themselves, consisting basically of undocumented FORTRAN and assembly language code.

Initial Attempts

The primary directive for the first 9 months of the project was to use as much of the existing software as possible. One of the employees started the job with about 9 months’ programming experience, then spent the first 6 months of her job reading the programs and trying to understand what they were trying to do. Another consultant spent 3 months trying to modify the existing alarms programs so they would work. He was unsuccessful. My solution was to look at the code for about 10 minutes, announce that it wasn’t worth saving, and begin to write the program from scratch. Unfortunately, I was one of the more productive members of the group.

In the long run just about every piece of software that the bestowing group had written was thrown out and completely rewritten. In some cases, although the program would have worked all right, it had to be discarded because it had been written in FORTRAN and was therefore too big to fit on the system.

The system was finally delivered—8 months late, and not meeting all the performance criteria.

Effects of the Decision to Use Existing Software

The most devastating effect of the decision to try to live with the existing software was that no meaningful attempt was made to do system design. All design decisions made by the first group were kept, which meant that no one had a real, thorough understanding of how the individual pieces of the system composed an integrated whole. They also did not have
the proper tools to make an adequate analysis of the system. In particular, the three basic maintenance productivity aids—tools, techniques and training—were almost totally ignored. For an asynchronous system, those involved were using synchronous tools, the system flow chart and the functional specification.

This lack of coordination was felt in other areas. Although at one point there were 15 programmers working on the system (most of whom had not known each other, much less worked together before this project), there were no coding standards, and no teams were formed. This is somewhat surprising, considering that Lientz has found that almost 80% of the shops he surveyed used at least chief programmer teams.

All programming was done on an individual basis, with no walkthroughs or group checking. One attempt by the programmers to hold weekly status meetings met with management disapproval; and although the meetings were extremely useful when they were held, they finally disintegrated. Thus the system was coded in 15 different styles, which is going to be a real maintenance headache.

**What Should Have Been Done**

Now that you have a picture of what the project development process was like, let me explore with you what should have been done. I left the project before it was finished, so I don’t know if a post-mortem evaluation was done, but I doubt it.

**Systems analysis**

First, the company should have evaluated the system requirements by using an analysis tool that could give them an overview of the entire system, such as Data Flow diagrams or Warnier-Orr diagrams. This would have helped the managers and the participants to see what the system was doing. As it was, only one or two people (if any) had a reasonable idea of how the system fit together, which meant that a great deal of redundant and counterproductive code was generated. In a way this recommendation is rather silly, because none of the people on the project knew that these kinds of tools existed—or, if they did, wouldn’t admit it.

**Evaluate system requirements**

The second step that should have been taken is to evaluate the limits of the system’s requirements. The project started in October, and the following September I finally did a memory map showing which pieces of code needed to be in memory at the same time and what their size requirements would be. Not surprisingly, the amount of memory available was far less than the amount needed. Had an evaluation of the limits been made at the beginning, the company would have realized that FORTRAN was going to be impractical for most of the programs in the system.

**Documentation**

The third step should have been to document the existing system. Using data flow diagrams, structure charts or data structure charts, and Chapin charts, the company could quickly have documented what had been done and could immediately have seen which parts of the system were usable and which needed to be thrown away. This would also have fostered a team approach to maintenance and development rather than the severely individual, egocentric programming that occurred.

**Determine staffing needs**

The fourth step should have been to consider the expertise the company had and what they needed, as well as to determine how many people would be necessary for the project. They tried to do this on a gut-feeling basis and started with five people, ending with 17. They did a fairly good job of matching expertise with needs when they added people, although they never had anyone who excelled in systems design, maintenance, or documentation. Since they had so many consultants, who were not going to be responsible for the maintenance function, there was no attempt at standardization, and no consideration was given to maintainability.

**Set coding and design standards**

The fifth step would be to set up coding and, as necessary, design conventions.

**Set up an implementation plan**

The final step would be to set up an implementation plan with review procedures. Had structured methodology tools been used, this step would be fairly obvious. A Gantt chart was finally developed in July, but until then no one knew what was expected. Since there were no regular status meetings and little communication, the chart was meaningless when it was put up, especially since there was no faith in the reasonableness of the estimates. The result was that several people sat around for many months not accomplishing much. One programmer sat from July to December before anyone wanted to see the results of his code. Predictably, when he ran his programs, it was discovered that they did not perform as required. This caused bad feelings and a significant loss of time.

The plan should have included an evaluation period for each section of software to determine the extent of modifications necessary. By the end of that period a decision could be made about whether that section should be modified or rewritten. As it was, some people spent far too much time trying to work with existing software, whereas others looked at it for about 15 minutes before deciding (and always deciding) to throw it away.

The project had too many strikes against it from the beginning to ever be really successful; but had it been handled correctly, it could have been termed a valiant and effective salvation effort.

From the collection of the Computer History Museum (www.computerhistory.org)
SITUATION II—AN ACCEPTABLE SITUATION

Background Information

The second situation makes a far happier story. This one takes place at my current place of employment, Henry S. Miller Co., in Dallas, Texas. Henry S. Miller is a real estate and property management company. The company’s data processing systems were developed on a leased IBM System/3 Model 10 during the period 1972 to 1976. In 1974 the process was contracted to a service bureau, which also assumed the hardware. In 1978 another company acquired the service bureau and assumed responsibility for providing processing and programming services to Henry S. Miller. The overall level of support and service received was not always satisfactory, so the company hired a local management consulting firm to review the support services. Over time the service requirements of users intensified to the point where an in-house computer was considered feasible. The consulting firm then helped Henry S. Miller write a request for proposal and select a hardware vendor. 12

Despite the long association with IBM, a Data General computer was chosen. The choice was based on performance tests conducted, and it considered primarily current needs. The machine was delivered in late December 1981 and was operational in early January. (The staff, which consisted of a programmer/analyst and a nonprogramming data processing manager, had begun to use a computer at the Data General office starting in October.)

Because data processing was being done by the service bureau, no data processing staff was in place. A data processing liaison, hired in mid-1979, learned all about the systems Henry S. Miller was using. Then they hired a programmer/analyst and became the data processing manager. The problem they faced was converting systems neither of them had written from IBM to Data General.

In December another programmer/analyst was hired, and a consultant who had Data General experience was engaged. The conversion schedule was tight. Everything was supposed to be converted by June 1982. Most of the systems were converted fairly close to their scheduled date, but there are still a few problems. The fact that they came close is amazing, considering that it has been said that “conversion estimates should not be attempted by persons who do not have access to a sizable data base of information and who do not do this on a regular basis.”13

The biggest problem is documentation:3 The systems are not well documented. There are a few system flow charts, and theoretically there are source listings for each program. Not only are the programs not printed one module per page; a program does not even necessarily start at the beginning of the page (see Appendix 1). Program names are not descriptive; therefore, which program goes with a given system has to be determined from the JCL, of which there are copious quantities (see Appendix 2). Unfortunately, this occurs in more cases than should be expected.14

Deloitte, Haskins and Sells reviewed the accounting systems and recommended that they be completely rewritten. This firm felt that there was little point in trying to modify undocumented RPG programs. Because of time and staff limits, however, this was not possible.

I started with the company in May 1982 and immediately began lobbying for documentation to be done. The major drawback to this is user expectations.1 Like many companies, we are operating in the crisis mode most of the time. Fast on the heels of the IBM-to-Data-General conversion has come a conversion of the manual and automated accounting systems to accrual from modified cash, which meant further changes to newly converted programs.

Understanding the Systems

To help myself understand the systems I was working on, I made a few data flow diagrams (DFDs). I found that in a relatively short period of time I became conversant with those systems. I have not made DFDs for most of the systems because I have not had time; consequently, their operation remains far fuzzier to me than the systems for which I have data flow diagram documentation.

Special Conversion Problems

A troublesome problem in the conversion was the conversion of files. IBM sign conventions for packed fields are different from Data General conventions,15 and the records were not always created with a consistent sign character. The problem was not always immediately noticeable, and it caused problems that were hard to track. The conversion involved a painstaking process of looking at the data to see what was happening, and it consumed a lot of time.

Another problem was that the programs seemed to be written by adherents of the Magic School of Programming.1,15 That is, we were frequently heard to exclaim, “How could that ever have worked?” I’m sure if we spent long enough thinking about it, we could have figured it out; but frequently we just rewrote that piece. This problem seems to occur any time that someone else has written a program and has not followed strict coding standards.16

Scope of the Project

The primary directive, again, was to get the systems up and running as quickly as possible—in any way possible. The data processing manager gave the programmers a great deal of leeway in deciding whether to use an existing program; therefore, on most of the systems, a decision was made fairly quickly about whether to convert or rewrite a program. Thus, not a lot of time was spent trying to use unworkable programs, but some time was wasted in rewriting programs that probably could have been converted. This has not been a problem for us so far, but it needs to be watched in the future as rewrite decisions become more expensive.10,16

Handling User Requests

One recurring problem has been users’ wanting “just a little change—and how soon can we get it?”13 This had come about

From the collection of the Computer History Museum (www.computerhistory.org)
because special user requests were taken into consideration as
the systems were being converted. Users would come to each
member of the data processing team with requests. As a re­s­
result, some priority work was not getting done and user re­
quests were being forgotten. The solution was to institute a
special request form (shown in Appendix 3) and to channel
requests through the data processing manager. This has been
a tremendous help to us and is extremely important in dealing
with inherited software, since what on the surface looks like
because the way the program is coded. We are given the
request and asked to estimate the amount of time it will take
make the change. On the basis of our estimates the requests
can then be prioritized. As a result the users feel that they are
getting better response to their requests.

Prospects for Design

Because the major effort has been the conversion of the
systems, little design work has been done. Some of the sys­
tems run inefficiently, and others are organized in such a way
that modifications are extremely difficult to make. There are
also redundant programs and files; and because our storage
space needs are very cyclic, we almost run out of space at
some times and use only about half our disk space at others.

A thorough system design needs to be made. We need to
get together with the users and find out everything that they
would like to include in their systems. The best way for us to
do this is to make a set of data flow diagrams of the existing
systems and then mini-models of the requested changes. From
this we can tell which changes can be made in the
existing systems and which will mean total redesign and
rewrite.

Comparison of the Two Situations

To compare the Henry S. Miller situation with that of the
engineering company shows decidedly that Henry S. Miller
was in a much better situation to begin with than was the
engineering company. The expertise of the two managers was
roughly equal, but at Henry S. Miller the manager is aware of
her lack of programming knowledge and has found reliable
people to depend on for advice. She has been open to sug­
gestions about documentation and respects programmer deci­sions involving rewrites. This has added tremendously to de­
partment productivity.

In both cases a lack of documentation has hurt the imple­
mentation effort. In Henry S. Miller’s case, however, at least
the programs were demonstrably executing before the con­
version effort began. In addition, at Henry S. Miller there is
still the possibility of doing good documentation.

Both companies made use of consultants, and both to some
extent gave up some control to the consultants. In both cases
the consultants were conscientious; however, the Henry S.
Miller consultant has made some effort to write programs that
are readable and maintainable.

In general, Henry S. Miller started with a better situation,
and through reasonable management it has kept the con­
version and maintenance effort under control.

SUMMARY

To summarize, there are several considerations involved when
a data processing department is dealing with inherited soft­
ware. First, a relatively experienced staff is required. The
learning curve is high and fast, and the programmer doesn’t
have time to deal with language concepts. Second, the staff
needs to be virtually overwilling to do documentation, since
most of the work involved in the conversion is documenta­tion.
Third, just as in new development, much front-end planning
needs to be done, including implementation planning that specifies a time limit for evaluating existing software.
This means that management has to accept the existing soft­
ware as a sunk cost—if it works, great, but if not, rewrites are
acceptable.

ACKNOWLEDGMENTS

I would like to thank Patricia Cathey, my data processing
manager, for supplying the background information and
keeping me honest in my description of the Henry S. Miller
situation.

I would like to thank Ned Chapin of InfoSci for his guidance
and encouragement, without which this paper would never
have been written.

I would like to thank LWFW, Inc., Group for the use of
their word processor.

Finally, I would like to thank my husband Bruce for doing
online literature searches on this topic and for his encour­
agement and suggestions.

REFERENCES

1. Liu, Chester C. “A Look at Software Maintenance.” Datamation, 22
3. Lientz, Bennet P., and F. Burton Swanson. Software Maintenance
6. Gane, Chris, and Trish Sarson. Structured Systems Analysis: Tools and
7. Orr, Ken. Structured Requirements Definition. Topeka, Kans.: Ken Orr and
   York: Yourdon, 1980.
   People, 27 (1978), 17–27.
10. Lyons, Michael J. “Salvaging Your Software Asset (Tools Based Main­
    tenance)”. AFIPS Proceedings of the National Computer Conference (Vol. 50),
    27–30.
12. LWFW, Inc., Group. Henry S. Miller Companies, Data Processing Request

### APPENDIX 1

<table>
<thead>
<tr>
<th>O - W - L</th>
<th>SOURCE PROGRAM LIBRARY SYSTEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE: 03/13/81</td>
<td>TIME: 17:12:51</td>
</tr>
<tr>
<td>O</td>
<td>USE 29</td>
</tr>
<tr>
<td>O</td>
<td>SOFM 40</td>
</tr>
<tr>
<td>O</td>
<td>SALPM 69</td>
</tr>
<tr>
<td>O</td>
<td>STUMM 78</td>
</tr>
<tr>
<td>O</td>
<td>STNAME 84</td>
</tr>
<tr>
<td>O</td>
<td>CITY 110</td>
</tr>
<tr>
<td>O</td>
<td>OWNER 132</td>
</tr>
</tbody>
</table>

```plaintext
**[1111] END HMCRL15 PRODрев001.BLKS=001,ACTSP=0004,DELS=00002**
```

```plaintext
**[1111] END HMCRL15 PRODрев001.BLKS=001,ACTSP=0004,DELS=00002**
```

From the collection of the Computer History Museum (www.computerhistory.org)
APPENDIX 2

P.HGLAD0S V.M.O.O 17 BLOCKS PRIVATE SOURCE STATEMENT LIBRARY

**END P.HGLAD0S**

```plaintext
* **/ CYCLE HGLAD0S ADDS GENERAL LEDGER TRANSACTIONS 08-28-81 0000
// PAUSE MOUNT ** W S M S T ** ON ANY MCG 1 0001
// EXEC IPRESTRT 0002
* **/ DATA RESTART 0003
* **/ RESTART 0004
* **/ JOB HMLKEY1 TRANSACTION KEYTAPE RESTART= YES 0005
* **/ XLST CLASS+U.DIS+0,CPY+01,CMPACT+IP00,FCB+88F00C A 0006
// PAUSE MOUNT KEYTAPE 'HMLGL02' ON X:387' 0007
// ASSIGN SYSLST.IGN 0008
// ASSIGN SYS007.N:387' INPUT HMLGL02 0009
// TLBL EPRESTRT,'&0WORK1.W' OUTPUT 0010
// EXEC LSKEYTP 0011
* **/ DATA HMLGL02 0012
* **/ RESTART 0013
* **/ RESTART 0014
* **/ RESTART 0015
* **/ RESTART 0016
* **/ RESTART 0017
* **/ RESTART 0018
* **/ RESTART 0019
* **/ RESTART 0020
* **/ RESTART 0021
* **/ RESTART 0022
* **/ RESTART 0023
* **/ RESTART 0024
* **/ RESTART 0025
* **/ RESTART 0026
* **/ RESTART 0027
* **/ RESTART 0028
* **/ RESTART 0029
* **/ RESTART 0030
* **/ RESTART 0031
* **/ RESTART 0032
* **/ RESTART 0033
* **/ RESTART 0034
* **/ RESTART 0035
* **/ RESTART 0036
* **/ RESTART 0037
* **/ RESTART 0038
* **/ RESTART 0039
* **/ RESTART 0040
* **/ RESTART 0041
* **/ RESTART 0042
* **/ RESTART 0043
* **/ RESTART 0044
* **/ RESTART 0045
* **/ RESTART 0046
* **/ RESTART 0047
* **/ RESTART 0048
* **/ RESTART 0049
* **/ RESTART 0050
* **/ RESTART 0051
* **/ RESTART 0052
* **/ RESTART 0053
* **/ RESTART 0054
* **/ RESTART 0055
* **/ RESTART 0056
* **/ RESTART 0057
* **/ RESTART 0058
* **/ RESTART 0059
* **/ RESTART 0060
* **/ RESTART 0061
* **/ RESTART 0062
* **/ RESTART 0063
* **/ RESTART 0064
* **/ RESTART 0065
* **/ RESTART 0066
* **/ RESTART 0067
* **/ RESTART 0068
* **/ RESTART 0069
* **/ RESTART 0070
* **/ RESTART 0071
* **/ RESTART 0072
* **/ RESTART 0073
```

From the collection of the Computer History Museum (www.computerhistory.org)
APPENDIX 3

DATA PROCESSING USER REQUEST

System _____________________________ Date ______________________

Reported By _________________________ Charge Code __________________

REQUEST DESCRIPTION (Please include report samples, program names, etc.)

______________________________

Requested Completion Date __________

PROPOSED SOLUTION ____________________

Estimated Completion Date __________

User Advised of Proposed Solution: Date __________ By __________

Completion Date: ________________