A methodology for minimizing maintenance costs

by LINDA BRICE and JOHN CONNELL
Los Alamos National Laboratory
Los Alamos, New Mexico

ABSTRACT

Research conducted in the case study of a large applications system shows that the two primary causes of high maintenance costs are

1. The frequency of user-requested changes to software
2. The psychological complexity of the software

A “tool kit” is suggested that, when applied to the design of new systems or rewrites, will

1. Produce systems that users are less likely to need changed
2. Contribute to the reduction of psychological complexity of code, making it easier to change when necessary

The tool kit is easy to use, can be applied to large or small systems in any language on any equipment, and requires no purchase of hardware or software.
INTRODUCTION

Maintenance costs escalate when software must be changed. Sometimes there are user-requested changes because the system does not meet the user's needs, and sometimes there are "bugs" because the systems and the individual program modules composing those systems are not well structured. All changes, whether necessary to fix bugs or desired to improve or add features, are difficult when program code is psychologically complex.

Quantifiable costs associated with software applications include the following: computer resources used by the application; programmer staff time plus computer-resource costs expended to maintain the application; and associated user time spent trying to learn and to use the end product. The focus of this paper is on programmer staff time expended to maintain the application. Maintenance will be defined as all changes required to keep a system running according to the user's needs, including:

- Corrections to programs necessitated by coding errors or misunderstanding of user requirements;
- Changes to programs required owing to changes in environment or legal/regulatory changes not under the control of the user;
- Enhancements or optimizations that alter the processing environment, often including minor new features.

Research performed in the case study of a large applications system has shown that the number of changes applied to a system and the psychological complexity (in particular, the misuse of branching instructions) of the code undergoing change both correlate positively with maintenance costs in terms of programmer effort. The term psychological complexity, as used here, refers to those elements of programming style that make the resulting software difficult to maintain.

This paper is written to suggest several aids for the reduction of software maintenance costs. The first suggestion is for the data processing professional to employ certain metrics to estimate the expense of maintaining software. Software shown to be expensive to maintain may then be subjected to a break-even/payoff analysis for economic justification of a rewrite. When rewrites appear to be economically feasible, care must be taken so that the new system is indeed easier to maintain than the old.

Many data processing shops continue to maintain production systems, despite high maintenance efforts, simply because they work. It would be helpful to have a method of deciding just when psychological complexity contributes enough to the maintenance costs to be economically unfeasible. There comes a time when, because of psychological complexity due to poor initial program design, or due to many "patches," rewriting the program (or set of programs) is more economically justifiable than continuing to maintain it.

In order to develop new systems and rewrites of existing ones that will have lower maintenance costs, a methodology is needed for designing with future maintenance in mind. Because psychological complexity is causally related to maintenance costs, the methodology should provide a means for minimizing such complexity. Since it has been demonstrated that requests from the user for changes correlate significantly with maintenance costs, the methodology should also aim at maximizing user satisfaction with new systems and rewrites in order to reduce future service requests.

WHEN TO REWRITE

One method for deciding when to redesign existing computer applications involves deriving an economic break-even/payoff analysis using a five-step process:

1. Track maintenance costs for a time period and then project future costs, using straight-line trend analysis.
2. Measure the complexity of the existing code using a demonstrated metric. This step does not contribute directly to the break-even/payoff analysis, but it does provide confidence that program complexity contributes to maintenance costs.
3. Estimate cost of rewrites.
4. Estimate costs for the maintenance of the new system after implementation.
5. Prepare a break-even/payoff analysis. In this projection (Figure 1), maintenance costs for the present system are shown as a straight line. Total cost for the proposed

![Figure 1 — Time in months (assuming rewrite takes two people six months)](From the collection of the Computer History Museum (www.computerhistory.org))
TOOL KIT FOR REWRITE

The life of software systems is traditionally viewed as a cycle or sequence of iterative events. Recently, the life-cycle concept has come under fire.6,7 This paper is not intended to pass judgment on the life-cycle concept—many versions exist, not all without merit. What is proposed here are a few techniques that we hope will reduce the costliness of maintenance. In order to describe the helpful tools, it is necessary to assume that the software to be maintained is written, not purchased, and that the development of that software proceeds in some order decreed by management. It is suggested that in order to minimize the number of post-implementation requests from users for changes, users be involved in setting objectives, and that production of output facsimiles and prototyping occur early in the development process.

The assumption will be made that software writers' management and the end users' management agree on the events that must take place to get the system up and running. Those events should be scheduled in a visual form (Gantt charts, Figure 2). The events will vary from project to project, but will necessarily include consultation with the user to describe system functions and software development. DP and user managements should meet before each major step to review the schedule.

The major goal is inexpensive maintenance. The tools are recommended (a) to force users' participation in the design, which will cause them to request fewer changes later, and (b) to produce lucid code that requires less effort per change. They are:

1. **System requirements definition (SRD).**  
   Tool: Scheduling guideline

2. **System Design.**  
   Results in system design document (SDD).  
   Tools: Output facsimiles or prototypes  
   Data flow diagrams (DFDs)  
   Policy statements  
   Data dictionaries

3. **Internal Design.**  
   Results in Requirements Specification Package (RSP).  
   Tools: DFD's of proposed system (from SDD)  
   Approved output formats (from SDD)  
   Policy statements (from SDD)  
   Data dictionaries (completed from SDD)  
   Logic-flow charts  
   Program abstracts  
   Program design walkthroughs

---

**Figure 2—Gantt chart**

---

From the collection of the Computer History Museum (www.computerhistory.org)
The methods and tools mentioned do not depend on team makeup or on computer-based tools. None of the tools are original with this paper. What is proposed here is the integrated use of the tools to meet the stated goals.

**Systems Requirements Definition**

The systems requirement definition (SRD) will not be covered in depth in this paper because, except for the schedule, there are no specific tools recommended. The purpose of the SRD is to identify proposed objectives, define the project scope, define the organizational units involved, identify the end users, identify production or purchase approaches, and construct a rough schedule and cost/benefit for each alternative.

Cost/benefit analysis has already taken place when the system is a rewrite. It is inherent in the break-even/payoff analysis mentioned under “WHEN TO REWRITE.” If the system is an entirely new development, the assumption is that a cost/benefit study would be necessary for a go/no-go decision by management at this point, prior to any actual development effort.

The schedule is not intended to be rigid as to dates. It is intended to identify the tasks to be performed, the parties involved, and the order in which the tasks will be performed. When reviews are held prior to the end of each phase (task), the remainder of the schedule can be reviewed and adjusted for reasonableness.

Gladden warns that “system objectives are more important than system requirements . . . concentrating on objectives can go a long way to prevent a system from ‘evolving’ into one that the user does not want or need.” The life-cycle wheel model of system development, which concentrates on viewpoints, stresses that “... requirements analysis is viewed as a design activity from a user viewpoint. This design is synthesized from various (incomplete, inconsistent) user scenarios and other expressions of needs. The emphasis is on what functions the system is to perform, and how the system interacts with the users.”

The SRD is, then, the project’s starting point and the place where objectives are defined.

**System Design**

The ultimate degree of user satisfaction with a new system or rewrite is often determined in the early stages of analysis and design. It is recommended that intensive interviews be conducted with the user during this phase. Such interviews should concentrate primarily on net outputs—the part of the system that will be visible to the user after implementation.

Users may have little interest in how data will be massaged to produce these outputs.

**System design: document components**

Careful users will want to know how the accuracy of the information contained in the net outputs can be guaranteed.

A system design document (SDD) should, therefore, contain the following elements:

1. A brief description of the framework within which the proposed system will operate, including
   a. constraints imposed by the operating environment
   b. required hardware/software configuration
   c. allowances for future contingencies.
2. Samples of proposed net outputs, such as report layouts and screens.
3. Proposed formats for net inputs, showing how data will be captured at original collection points.
4. Visual diagrams of data flows for the present system (either manual or automated) and the proposed system.
5. Policy statements giving a decision method for each procedure shown in the above diagram(s).
6. Rigorous definitions of all data elements shown in the above diagram(s).

**System design: tasks**

Development of the SDD components need not be undertaken in the order given above. The SDD’s development guidelines may specify tasks to be performed in preparing such a document, and the order in which they should be performed. The following is a brief description of each of these tasks.

**System design: tasks—describe system environment.** At this stage, the system designer can recognize when the new system will exist in a physical environment that may impose constraints on the design. The task during this phase should involve documenting the nature of that environment and identifying areas that might impose design constraints.

**System design: tasks—describe net outputs.** Examples of proposed outputs can be produced rapidly without using actual applications software. The editor on any system can be used to produce a text file that, when copied to the line printer, will produce a facsimile report or screen layout. The main advantage of this approach is that content and format can be changed easily without modifying software. In addition, the facsimile reports provide an immediate focal point for user interviews. Users who tend to be vague about system requirements can often be coaxed into being more specific by discussing information contained in the new outputs. If the output formats are approved by the user before the system design begins, the result should be fewer design changes and service requests after implementation.

If an installation has available the necessary tools (i.e., flexible database systems), it is strongly recommended that a prototype system be brought up at this early stage. “It is now recognized . . . that although the customer may state his requirements very firmly at the beginning, his perception of the problem begins to change as he begins to consider how the solution development . . . is proceeding.” Peters, Gladden, and McCracken and Jackson all recommend rapid proto-
typing to combat wholesale requirements changes.8,7,10 The remainder of the SDD is charged with demonstrating that the approved sample net outputs can be produced accurately.

System design: tasks—describe net inputs. If the user is familiar with existing inputs, it is probably not necessary to produce samples. There may be, however, implications in the above components of the design for new methods of data capture or even entirely new data elements to be captured. In this case, it is important to solicit user approval of new input formats such as data entry screens. The method for providing examples of proposed input formats can be the same as that for output formats—sample forms produced with a text editor form an obvious, simple manner.

System design: tasks—produce data flow diagrams. For a visual representation of the flow of data between functions performed by a system, the use of the data flow diagram (DFD) is highly recommended. DFDs have been explained in Yourdon's structured analysis and design technique, and are described by Dr. Marco.11 Basically, these diagrams consist of bubbles, arrows, and parallel lines. The bubbles represent a procedure, the parallel lines represent a data store, and the arrows represent the flow of data between the procedures and data stores. The diagrams are ordered by degree of detail—the highest level (Level 0) contains only one bubble labeled with the system name and shows only net inputs to and outputs from the system (Figure 3). The lowest-level diagrams show elementary procedures and data elements (Figure 4). One suggestion for the number of descriptive levels is seven, plus or minus two. The rule also applies to the number of bubbles or procedures per level. The diagrams should remain visually digestible, since they are the tool for user interviews in this phase.

DFD demonstrate for the user how net inputs will be transformed into net outputs; therefore they serve as a primary check on the accuracy and completeness of the outputs. This technique tends to minimize unnecessary or over-complex procedures and maximize user satisfaction.

Each of the bubbles or procedures shown in the lowest-level DFD should have an associated policy statement that describes the decision method proposed to perform the procedure. These policy statements should be expressed in structured English or pseudo-code so that they are unambiguous yet still intelligible to the user (Figure 5). They should be developed in the interviews with the user so that they are, in fact, the user's policies. Each policy statement should correspond to a bubble on a low-level DFD.

These statements of user policy should eventually become online documentation for production source-code in the form of preludes (abstracts) for procedure modules. Initially, they serve as a guide to system design; later, they can serve as a maintenance aid.

System design: data dictionary

Each of the arrows in all of the levels of the DFDs will have a label. The SDD should include a "dictionary" defining each of these labels. The definition of a data label on a high-level diagram should be in terms of the labels on the diagram of the next lower level. At the lowest level, each label should also be defined as to how, when, and where that element will be captured.

If the dictionary is complete and rigorous, it serves as a proof that the user's requirements, as expressed in the policy statements, can be satisfied using the data defined therein. Each definition should correspond to the level of the DFD on which it can be found as the label of a data flow. This also answers the designer's question, "what data do I need, and where can I find it?"

Internal Design: Requirements Specification Package

Once the SDD has been approved by the user, "internal" design can begin. Here, internal design will only address those elements necessary to develop low-maintenance software. The requirements specification package (RSP) components will include

- Copies of the DFDs that identify program modules
- Approved output (reports)
- Data dictionary from the SDD
- Policy statements from the SDD
- Logic flow diagrams for each module (Chapin charts)
- Program abstracts.

The data dictionary may be revised during this phase of the project, and policy statements should contribute to the functions listed in the program abstract.

Internal design: Chapin charts

It is suggested that Chapin charts,12 Nassi-Shneiderman Structured Flowcharts,13 or the structured programming design method (SPDM)14 be used to describe logic flow for each program module. The three are similar in philosophy, and any one can be used to bridge the gap between module need (basic requirements) identification and executable code. The document will be referred to here as a Chapin chart.
The lowest-level DFDs in the proposed system section of the SDD represent processes in bubble format. Usually, each of these processes identifies a program module, as well as the inputs and outputs. Policy statements in pseudo-code or in structured English accompany the DFDs. The combination of inputs, outputs, and policy statements form the skeleton of a Chapin chart. If a database-management system is used, it will also have been defined in the SDD as "required software configuration" under the operating environment. If not, files or specific formats for data-transfer mechanisms must be specified prior to the construction of Chapin charts.

The Chapin chart is based on this cumulative knowledge, sometimes with the addition of special processing algorithms. The reader is referred to the references for in-depth explanations of this logic-flow chart. The method, in essence, consists of visually representing a set of program building blocks that allow single entry/exit and strictly limit branching, a practice known to increase psychological intelligibility. The set of program structures includes SEQUENCE, IFTHENELSE, DOWHILE, DOUNTIL, and CASE. When used properly, the set of combined structures lends itself to a well-structured program guide where arbitrary transfers of control are impossible. Figure 6 is an example of a Chapin chart.

The benefits of the Chapin charts are

- Provision of a "GOTO-less" map to be translated directly into a programming language.
Provision of a document that graphically depicts logic for the purpose of review (peer review, team walkthrough)

Provision of a test-bed guide. It has been noted that Chapin charts are not devices that provide functional hierarchy, interfaces, or data flow. The contention here is that there is no necessity that Chapin charts respond to those needs, since they are met by the DFD. What Chapin charts do well is control flow of executable code within a higher-level functional design. This toolkit provides the functional design via DFDs.

Internal design: walkthroughs

Approved DFDs showing processes (program modules), inputs, outputs, policy statements, functional hierarchies, interfaces, and data flows are available from the SDD phase; program-module logic design is graphically represented by the Chapin charts. Because of the importance of structuring program code for understandability and readability in the maintenance phase ("good" structure equals psychologically clear code and minimum branching), the Chapin charts should be subjected to a peer review before the coding phase. The review should not only insure the structure of the individual modules, but should double-check to see that elements are defined in the data dictionary, that the process will accurately perform what was intended in the higher-level diagrams, that the outputs conform to the early prototype specifications, and that a program abstract is present. The abstract would minimally consist of

- Purpose
- Input (arguments/files/other)
- Output (arguments/files/other)
- Functions (10 or less)
- Local variables
- Subprograms called
- Errors (fatal/non-fatal)
- Standards violations

An example of a program abstract is in Figure 7. The purpose of walkthroughs is improved (low-maintenance) quality of the product. The value of walkthroughs shows up ultimately in the maintenance phase. "The inspection process shifts the discovery and correction of errors and defects from software's operational period to the early design stages. Since the cost for software corrections during operations is many times the cost incurred in detecting problems during design, inspections provide an unusual leveraging of cost/benefit over the entire life cycle of the software." Although a heavy commitment is necessary for the time of team members and moderator participation, other benefits beyond low-maintenance code are

![Figure 6—Chapin chart](From the collection of the Computer History Museum (www.computerhistory.org))

![Figure 7—Program abstract](From the collection of the Computer History Museum (www.computerhistory.org))
accrued, such as "training and exchange of technical information among the programmers and analysts who participate in the walkthrough." \(^{18}\)

CONCLUSION

Use of this tool kit will not guarantee that the resulting system contains minimal psychological complexity and maximized user satisfaction. It is possible to misuse the tools. The intention of this paper was to explain some of the factors that cause software to be expensive to maintain, and to provide aids that may be useful in designing low-maintenance systems.

REFERENCES
