A Framework for Improving Software Maintenance Efficiency

Hossein Saiedian and James Henderson
Department of Computer Science, University of Nebraska, Omaha, U.S.A.

Software maintenance has emerged as a major effort within many software organizations. In this paper we review the facts on software maintenance, its unavoidability and associated cost. We analyze three primary types of maintenance, and search for a means of increasing software maintenance efficiency. We examine factors affecting a system's maintainability and see that significant improvements can be achieved by emphasizing preventative maintenance. A framework for implementing a preventative maintenance program based on the re-engineering of individual subroutines within a system is suggested. The suggested framework proposes criteria for selecting candidate subroutines whose reengineering will yield a high return on investment. The framework gives considerations for reengineering these candidates subroutine and discusses post-coding activities. Finally, we will briefly examine some potential areas for further study.

Keywords: Maintenance Techniques, Preventative Maintenance, Maintenance efficiency factors, Reengineering

1. Introduction

When the software development team completes the creation process and gives birth to a new program, the work is far from complete. In fact, the long, challenging, and expensive work of maintenance begins. This process may last only a few years and involve only the programmers who originally developed the system. Yet, the process frequently goes on for many years and can involve several generations of programmers. Due to their uniqueness or importance to the organization, we keep many systems in service for many years beyond their expected life spans. These systems require high levels of maintenance to meet changing circumstances that they were not originally designed to handle.

As an example of the magnitude of the problem, Weinman [WEI91] states that, "software inventory worldwide comprises more that 100 billion lines of working source code, with Cobol making up more than 80% of that. Conservative estimates of the cumulative cost of producing those systems is $2 trillion." Osborne and Chikofsky concisely explain why much of our software is difficult to maintain as follows:

Much of the software we depend on today is on average 10 to 15 years old. Even when these programs were created using the best design and coding techniques known at the time, they were written when program size and storage space were the principal concerns. They were then migrated to new platforms, adjusted for changes in machine and operating-system technology, and enhanced to meet new user needs — all without regard to overall architecture [OSB90].

As a starting point in attempting to control the software maintenance process, we will examine software maintenance, its unavoidability and associated cost. We will examine then three primary types of maintenance and some ways we can attempt to improve these areas. Following that, we will attempt to find a means of increasing overall efficiency through a framework for preventative maintenance. Finally, we will close with an examination of possible areas for future research.

2. Software Maintenance

What is software maintenance? It is the act of taking a software product that is being used by a customer, and keeping it working satisfactorily [GLA92, pp. 181]. With a system of significant size, used for a significant period of time, in a changing environment, we cannot avoid maintenance. We cannot expect to avoid maintenance by applying the lash to the developers and stating, "We will have error free software!" Even if they could produce error free software, a highly questionable prospect, that would not free us of maintenance. At some point, the users will want another option or report, or we will have to upgrade to a new computing environment and 'voila' — here again we have maintenance. Glass [GLA92, pp. 181-182] cites two reasons why we cannot avoid maintenance:

1. It is regrettably true that we simply don't know how to produce error free software.

2. Software change (not error correction) is the major component of software maintenance. That change often comes about because the customers of a software product have new vistas opened for them at product delivery, and come to understand new software capabilities they would like to have that they had not envisioned before. Therefore, the majority of software maintenance comes about not because someone did something wrong, but because someone did something right.

Many researchers have examined these issues and made significant contributions to this field. Lehman and Belady [LEH85] addressed this problem and came up with five laws relevant to software evolution. See Figure 1.

We can see several reasons why maintenance is essentially unavoidable in any significant system. Thus, this yields an important problem when we look at the cost and effort involved. Parikh [PAR87] had the following comments relevant to cost:

- Results of a survey of 149 MVS installations with programming staffs ranging from 25–800 programmers indicated that maintenance represented from 55 to 95% of their work load.

- Estimates that $30B is spent each year on maintenance ($10B in the US) with 50% of most companies' DP budgets going to maintenance and that 50–80% of the time of an estimated 1M programmers or programming managers is spent on maintenance.

- An MIT study which indicates that for every $1 allocated for a new development project, $9 will be spent on maintenance for the life cycle of the project.

Pressman further states, "If nothing is done to improve our maintenance approach, many companies will spend close to 80 percent of their software budget on maintenance by the mid–1990s" [PRE92, pp. 667]. In fact, as Glass [GLA92, pp. 183] points out, "In some organizations, there remains no new software development. All work is maintenance."

We have seen that maintenance is essentially unavoidable and is a significant organizational expense. Now we will examine three traditional types of maintenance and look for means of optimization within them.

1. The law of continuing change: A program that is used in a real-world environment necessarily must change or become progressively less useful in that environment.

2. The law of increasing complexity: As an evolving program changes, its structure tends to become more complex. Extra resources must be devoted to preserving and simplifying the structure.

3. The law of large program evolution: Program evolution is a self-regulating process. System attributes such as size, time between releases, and the number of reported errors are approximately invariant for each system release.

4. The law of organizational stability: Over a program's lifetime, its rate of development is approximately constant and independent of the resources devoted to system development.

5. The law of conservation of familiarity: Over the lifetime of a system, the incremental system change in each release is approximately constant.

Fig. 1. Lehman and Belady's Five Law of Software Evolution

3. Traditional Approaches to Maintenance

There are three types of software maintenance commonly identified: *corrective, adaptive*, and *perfective*. Corrective maintenance is the correction of software errors. Adaptive maintenance is the changing of the software enabling it to handle new computing environments, which is often necessary since so many software systems are maintained for so long. Perfective maintenance is the changing of the system to provide the user with additional capabilities. Lientz and Swanson found that approximately 65% of maintenance was perfective, 18% adaptive, and 17% corrective [LIE80]. There are things that we can do to ease our maintenance in each of these areas.

To limit our corrective maintenance we need to implement a defined process for software testing and configuration management. A thorough testing procedure including unit, integration, and system testing can help to limit the number of errors introduced into the software. This is especially true if we use carefully considered and thorough test plans at each level and make maximum use of regression test sets containing standardized tests that should be run after any system modifications. Configuration management also can help to limit our corrective maintenance by acting as a central point for controlling software change. With configuration management, we ensure that only thoroughly tested modules are allowed into the system and that the correct versions of each module are used at each step in the procedure.

One of the best ways we can reduce our adaptive maintenance is by using standard language constructs as much as possible in our software so that the constructs we use will still be supported in a changing environment. When we cannot use standard language constructs, we should try to separate all non-standard constructs and version-dependent data into an interface module. This interface module should then be the target of most of our maintenance when changing environments.

Perfective maintenance is user-driven so we have less ability to limit our exposure to it, but there are things we can do to anticipate it and to make it as efficient as possible. When developing or when doing perfective maintenance, we need to try to fully understand the user's needs. We must elicit, as much as possible, all the capabilities the user needs and may need in the future. If possible, we can include then these capabilities or design such a system that adding those capabilities in the future will be relatively painless. Additionally, a careful analysis, representation, and validation of the users' requirements will ensure that we are using our time efficiently. We cannot afford to waste our time working on non-existent problems while ignoring the real problems because we failed to understand fully what the user needed.

As discussed, there are things that we can do to optimize our maintenance activities within each of three common types of maintenance. But, is there anything we can do to improve the efficiency of our overall maintenance ? We'll do some of those nice things in the future when we can, but we're already stuck with this system — what can we do to improve our maintenance now?

4. Preventative Maintenance

Perhaps the most significant thing we can do for our software now is to realize that we should be doing something in addition to the required corrective, adaptive, and perfective maintenance. This additional task is a *preventative* maintenance. Pressman defines preventative maintenance as the activity that occurs when software is changed to improve future maintainability or reliability, or to provide a better basis for future enhancements [PRE92, p 664]. Under this broad definition, we can include such things as: *restructuring, reverse engineering*, and *re-engineering.*

One feature of these efforts that makes them harder to justify is that our traditional users are not the direct beneficiaries of this effort. We directly benefit, as does our organization and those that come after us. We will rarely give any form of preventative maintenance a higher priority than the user-related maintenance, but we might form a working group dedicated to these efforts while other groups perform the user-related maintenance.

4.1. Restructuring

Restructuring is the transformation of a software system from one representation form to another, usually at the same relative abstraction level, while preserving the subject system's external behavior (functionality and semantics) [CHI90]. This technique is a realization that those who came before us probably didn't apply many of the structured programming concepts that we might today and that the system might be easier for us to maintain if it adhered more fully to those concepts.

With little understanding of a system or subroutine, we can still make significant alterations to its structure with some degree of certainty that we haven't changed its functionality. For example, we can convert loops created with GOTOs to modern REPEAT-UNTIL or DO-WHILE loops or change a long series of IFs to a CASE statement. With a deeper understanding of the meaning of the system or a subroutine, our changes can go deeper. We can change the type of data being handled or change the modularity of a system or subsystem to reflect better the system's overall functionality.

Some degree of restructuring can significantly reduce the time it takes for a new programmer to understand a given piece of software. This becomes particularly significant if we restructure the software at our leisure, but later programmers are trying to understand it to meet a critical deadline.

4.2. Reverse Engineering

Reverse engineering is the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form at a higher level of abstraction [CHI90]. This generally involves examining a working software system to capture its underlying design and requirements. However, it can be performed at any stage in the life-cycle to ensure that the abstraction we derive from a phase matches the output from an earlier phase (e.g., reverse engineering from our design to requirements should yield something resembling our original requirements). In a reverse engineering effort, we are not changing the system, we are simply trying to gain a better understanding of it. Chikofsky and Cross [CHI90] suggest that six key objectives of reverse engineering are to: cope with complexity, generate alternate views, recover lost information, detect side effects, synthesize higher abstractions, and facilitate reuse.

4.3. Re-engineering

Chikofsky and Cross [CHI90] describe re-engineering as follows: "Re-engineering, also known as both renovation and reclamation, is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form. Re-engineering generally includes some form of reverse engineering, to achieve a more abstract description, followed by some form of forward engineering or restructuring."

When re-engineering, we generally don't change what the system does — we simply change how it does it. To do this properly, we must truly understand what the software is doing, and why, before we begin, hence the typical reverse engineering or analysis step.

5. Maintainability Factors

Software maintainability is the ease with which a software system can be corrected when errors or deficiencies are discovered and can be expanded or contracted to satisfy new requirements [McC92]. We must examine the factors that impact the maintainability of a piece of software before deriving a framework for preventative maintenance. The presence of some factors will make a body of code easier to maintain, while other factors will make it more difficult to maintain. Then, our preventative maintenance framework will attempt to enhance the positive factors and eliminate or mitigate the negative factors.

A major portion of a systems maintainability is related to its understandability. We can define understandability as the ease with which one can decipher the function of a program and the way in which that function is performed by examining the code and its associated documentation. McClure [McC92, pp.20–21] found that software maintainers spend approximately 47%

of their time in trying to gain an understanding of the software. Obviously, if maintainers spend that much of their time attempting to understand, then any improvement in understandability will yield a significant savings in time.

A significant component of maintainability is the level of documentation. Documentation can be non-existent, sparse, adequate, or voluminous. Additionally, it can be correct, incorrect, or correct but out of date. Ofttimes incorrect documentation can be worse than no documentation at all. If a programmer takes incorrect documentation at its word, it could give a faulty understanding of what a program does or how it does it, leading to potentially dangerous coding errors.

As we would expect, module size and complexity are significant factors in program maintainability. Gremillion [GRE84], Lientz and Swanson [LIE80], and several others have found high correlations between module size and complexity and the number of errors present. The complexity of a module can be measured in various ways including simply counting lines of code, McCabe's Cyclomatic Complexity, and Halstead's Effort [NAV86]. Each metric measures different combinations of factors such as module size, number of control constructs, or number of variables. No metric has achieved universal acceptance, but any measure can give some insight if applied to a representative sample of modules. As Chapin [CHA83] states, "good ovidence exists that increases in complexity in application software go hand in hand with increases in the cost of maintenance and declines in the morale and productivity of programmers and analysts."

Language can be a factor in the maintainability of a module in two ways. First, the language that the software is written in may introduce some limitations or possibilities and will have a significant impact on the availability of skilled programmers. The second consideration in language use is to what degree programmers will be limited to standard language features. Some compiler-specific features may add flexibility or strength, but they can be a significant negative when trying to port the system to another environment that is not supported by the same compiler.

Consistency, or adherence to standards, can also be a factor in maintainability. Consistency of coding style implies that the program contains uniform notation, terminology, and symbology that comply with corporate naming conventions and standards [McC92]. Brooks [BRO82, pp. 42] refers to consistency as conceptual integrity and calls it "the most important consideration in system design." A consistent subroutine layout, for instance, will make it much easier to find relevant comments, find variable declarations, follow logic flow, etc.

Finally, the use of the simplest, most straightforward logic can affect maintainability. There are those among us who delight in obscure or arcane logic over any other. We should try to specify logic paths in the simplest possible terms even if this doesn't use the fewest possible lines of code.

We have looked at several factors that can affect the maintainability of our software, many of them related to its understandability. Now, keeping these factors in mind, we can attempt to express a framework for our preventative maintenance. Houtz and Miller [HOU83] cite the following goals for a program to improve software:

- Improve software maintenance and control
- Reduce delays in responding to users' needs.
- Improve software quality.
- Increase programmer productivity.
- Decrease software maintenance costs.
- Institutionalize processes.
- Change software from a reactive to proactive state.
- Extend the software's life.
- Put the organization in a position to take advantage of new and emerging technology.

The decision to undertake a program for preventative maintenance should not be taken lightly, but should also not be avoided. Britcher [BRI90] states that "without a well-defined rationale and plan, re-engineering could at best be too expensive, and at worst the wrong thing to do." Osborne and Chikofsky [OSB90] suggest that "we must have a continuing concern for future system maintainability, better support for the people who manage software systems, and a change-management discipline." Eliot and Weinman [ELW91] sum up this difficult decision this way, "making significant changes in

old systems is frightening proposition, but re-engineering can help avoid the even more daunting destruction wrought by applications that are nearing the end of their useful life."

6. Selecting A Framework for Preventative Maintenance

Having decided to undertake a preventative maintenance program, we still must decide upon our approach and the details thereof. We have a choice of three primary approaches: completely re-engineering the entire system as a unit, restructuring individual subroutines, or re-engineering individual subroutines on a priority basis. This is pictorially shown in Figure

The first approach is perhaps an optimal solution for many systems. If reverse engineering techniques are properly applied, we should derive first a complete set of system requirements and specifications and a thoroughly analyzed design approach. Then, we will forward engineer from these documents to a modern, structured system. Eliot [ELI91] states that to do this we would, "identify all of the changes that must take place, make the changes in as much an off-line mode as possible, and then make the entire shift all at once." The end result should be a system that is well documented, easy to maintain, and has a high degree of conceptual integrity. Yet, a large percentage of today's systems are of such a size that this effort would be truly monumental and thus expensive and difficult to get management approval for. Additionally, since we are making a large shift all at once, this approach generally entails a higher risk. Houtz and Miller [HOU83] maintain that, "because of the large amount of software that exists in most ADP organizations, most software can't be improved in one lump sum."

The second approach, restructuring, is perhaps at the opposite end of the spectrum from the first in terms of risk and size of undertaking. In restructuring, we can make significant alterations in a subroutine's structure with some degree of certainty that we haven't changed its functionality even if we don't fully understand it. The risk involved is thus quite low and the project can be performed piecemeal, giving some immediate results. The final product won't necessarily include updated system requirements, specifications, and design information, or significantly change the modularity of the system. However, the result should be a group of subroutines that are at least marginally easier to understand and maintain.

The third approach, re-engineering individual subroutines on a priority basis, is in many ways a hybrid of the previous two. It is "less risky [ELI91]" than re-engineering the system as a whole, yet will yield some of the same information, though at a lower level. It is quite similar to restructuring, but taken a step farther. We think that this approach holds the highest cost-benefit ratio of the three for most systems. This is based on the famous 80/20 rule, which in our case holds that 20% of the subroutines should contain 80% of the problems. Therefore, significantly improving a relatively small percentage of the subroutines will yield considerable improvements in overall productivity. We propose a framework for preventative maintenance based on this approach. Many of the considerations in this frame work would be equally applicable to a restructuring approach.

If your system is small or your organization is such that a significant effort can be devoted to re-engineering, then perhaps re-engineering the system as a whole would be the best approach. This might especially be true if your organization has previous experience in re-engineering since you will be aware of potential problems and understand the returns.

If your organization would like to realize some improvement in maintainability, but can't afford the higher effort involved in re-engineering, then perhaps restructuring would be the preferred approach. This might also be true if your organization is hesitant to accept even a fairly low level of risk or if a large percentage of your personnel have low experience levels.

7. A Proposed Framework for Preventative Maintenance

In this approach, we will select individual subroutines that we believe are likely candidates to benefit from re-engineering based on a set of criteria. We use reverse engineering techniques on the subroutine level to create functional descriptions for these individual subroutines. We

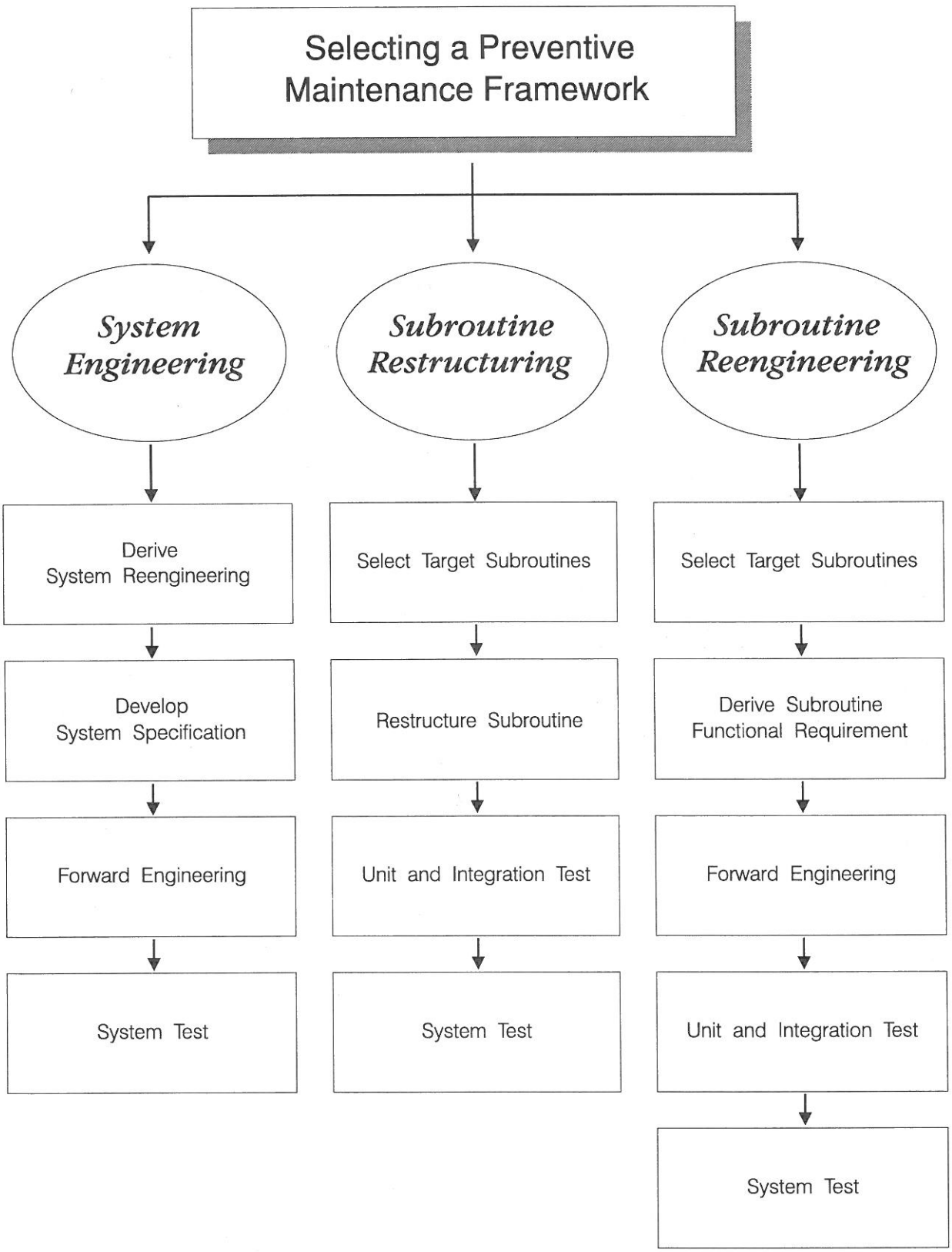

Fig. 2. Selecting a Preventive Maintenance Framework

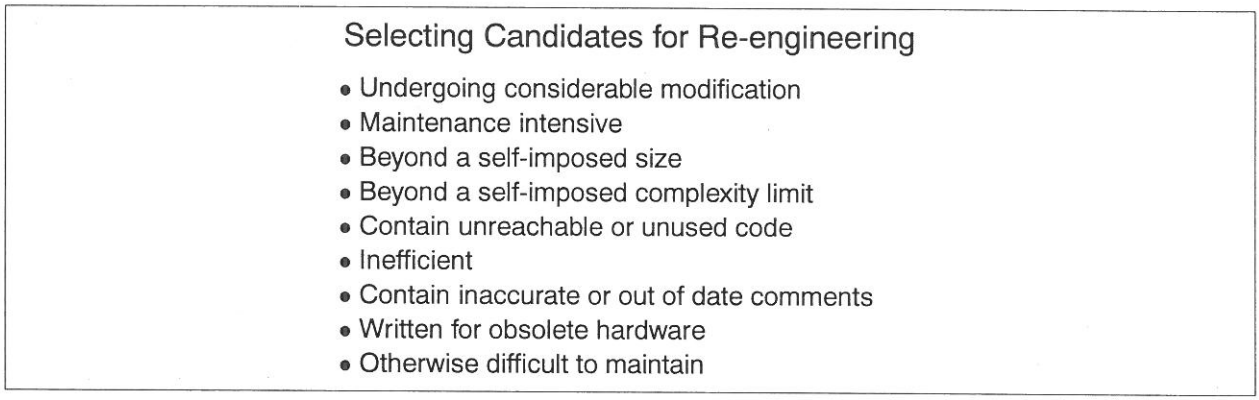

Fig. 3. Re-engineering Factors

then attempt design recovery for each subroutine which "must reproduce all of the information required for a person to understand fully what a program does, how it does it, why it does it, and so forth [BIG89, pp. 36]." Further, we examine the logical decomposition of these subroutines, their parents, children, and siblings. We will have the understanding to make real changes in modularity, if necessary, and greater changes in the underlying code structure. When we have completed the re-engineering of this group of subroutines, we will then carefully validate them and select a new set of candidate routines.

This approach won't yield soon the seamlessly integrated and consistent package that an all-out approach might, but we will be able to show progress in a much shorter period of time. It also has the benefit of being easier for reluctant organizations to accept because the minimum required effort is much smaller. We can do as little as one subroutine at a time or as many as one, or more, subroutines per programmer. Ultimately this approach can mate in with a system approach by providing functional descriptions and design information which will be very valuable in constructing overall system requirements and design information. Houtz and Miller [HOU83] state that this type of piecemeal approach has four basic advantages in that it:

- minimizes uncertainty and risk by maximizing the utilization of testable components.
- preserves the value of past software investment as much as possible.
- enables the project to be broken into small, manageable pieces with an operational system at each phase; and
- is iterative in nature, which allows for the tasks performed to be repeated in an orderly, incremental fashion with constant achievement, growth, and feedback, until the overall objectives of the project are met.

7.1. Selecting Candidates for Re-engineering

As depicted in Figure 3, there are many factors to be considered when selecting candidate subroutines for re-engineering.

Perhaps a good place to start is by considering those subroutines that are already being significantly modified for user-related maintenance. McClure [McC92, pp. 29] states that those components that require a major enhancement should be among the top candidates for re-engineering. One main advantage of selecting such routines is that we are going to be forced to investigate them thoroughly anyway in order to complete the work request. Additionally, since they are already being significantly changed, and thus will have to be strongly tested, our re-engineering changes don't necessarily add a significant additional load of testing.

One frequently cited criteria [McC92, pp. 29], [SNE91, pp. 170–171] is the selection of those subroutines that are maintenance-intensive, being changed an inordinate number of times within a set period of time. These subroutines tend to exemplify the 80/20 rule. Changes made in these subroutines are likely to give us the

greatest return since we will either be reducing the problems in these subroutines or simplifying them for the next time we do have to deal with them.

An additional method is to select those subroutines that exceed a set limit for lines of code. This limit will be dependent on your programming language, type of application, and local standards. No specific number has been found to be particularly meaningful as a limit, but you should set some limit reasonable for your software. Then, any subroutines that exceed this limit should be investigated because they might not be reasonably modularized. Gremillion [GRE84], Lientz and Swanson [LIE80], among others, have noted a strong correlation between program size and complexity and number of errors present. Thus, if we work on those that should have the highest concentration of errors, we will likely enjoy the swiftest reduction of errors.

Similarly to size, we can examine any subroutine that exceeds a set limit for any selected measure of code complexity. Again, no measure of complexity is universally accepted, but if you select some measure that seems representative for your software and apply it across a wide range of your subroutines — the results should be somewhat telling. Any subroutine exceeding this limit should be examined.

We may also wish to examine those subroutines that contain unreachable or unused code, perhaps retained from now unnecessary options. This is significant because your 'unreachable code' may not indeed be unreachable, which could lead to severe problems. Also, there is a significant correlation between size and number of errors [GRE84, LIE80], so any additional code can hide additional bugs.

One criteria that borders on perfective maintenance is to select subroutines that perform time-intensive operations using inefficient methods. Re-engineering these subroutines will allow us to carefully examine our methodologies and modifying these subroutines will produce benefits for both our customers and ourselves.

Clearly, the level and quality of documentation have an impact on the ease of maintenance. Therefore we may wish to examine those subroutines that have inaccurate or out-of-date documentation. As stated previously, inaccurate documentation can be worse than none at all since it can deceive programmers as to what is really going on. Out-of-date documentation might, for example, reference obsolete table sizes which could deceive programmers as to the maximum subscript values. Problems in documentation may not require or justify a full-scale re-engineering of a subroutine, but it is worth examining these subroutines since the documentation may be inaccurate because the subroutine is difficult to comprehend.

Sneed [SNE91, pp. 170–171] and McClure [McC92, pp. 29] both cite subroutine being written for a previous generation of hardware as a possible criteria for re-engineering a subroutine. These subroutines may use methodologies that simply didn't port well, may fail to take advantage of significant system capabilities, and may have been written before some useful capabilities existed. Instead, they might use rather obscure methods for achieving the

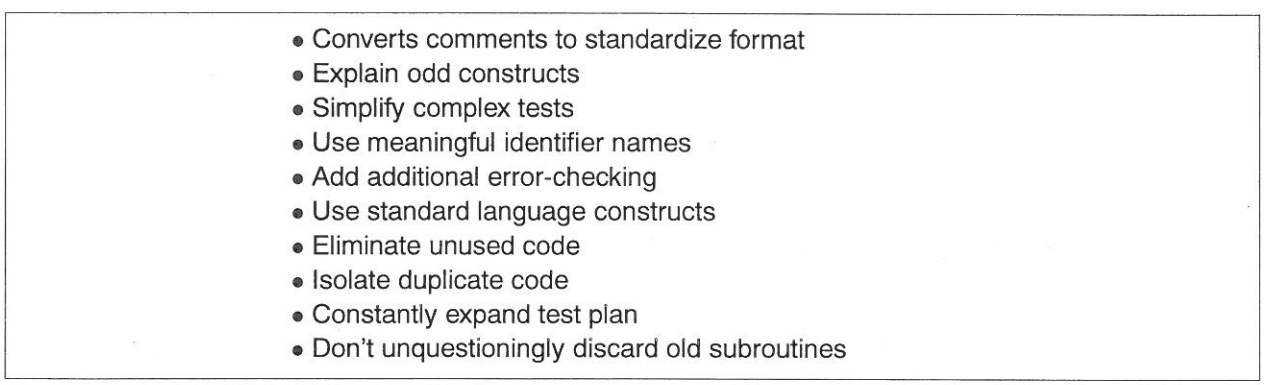

Fig. 4. Subroutine Re-engineering Considerations

same end that will in turn be difficult to maintain. One example of this might be old Fortran subroutines with character data stored in real or integer variables and thus manipulated.

Finally, we may wish to examine those subroutines that, for any combination of reasons, are exceedingly difficult to accurately maintain. There will be some subroutines which may not exhibit any of the previous criteria specifically, but which are, nevertheless, difficult to maintain. This could be for any combination of factors and will depend on a programmer's judgment to identify.

Once we select subroutines based on these criteria, there are certain things we should do before we actually start modifying the code. We should first reverse engineer the subroutine so that we can ensure that we fully understand its functional requirements. This will then help us to prepare a careful test plan to determine if our modified subroutine still performs its intended function(s), and no other. Finally, we need to generate a new subroutine design and ensure that it will lead to a subroutine that properly performs the functional requirements.

7.2. Considerations While Re-engineering

When we are actually ready to modify the code, there are many things we should consider as shown in Figure 4.

Among them are several things relative to comments. First, convert comments to a standardized format. This standardization can give us that conceptual integrity [BRO82, pp. 42] and allow for easier comprehension and thus easier maintenance. Next, validate existing comments for accuracy and completeness. Recalling the large percentage of programmer's time spent on understanding the software (close to 50%) [McC92, pp. 20–21] — the importance of this is easy to visualize.

In particular, add additional explanation of any odd constructs or methods including an explanation of why things weren't done in a more straightforward way. If things are done, for some good reason, in an odd way, you should pass this reasoning on to the next programmer or they may simply change it to the more straightforward method. If you did indeed have good reasons for not doing it this way in the first place, this could have serious consequences. An example of this could be using a slower sort because you need a certain type of stability more than you need optimization.

We can increase system clarity if we simplify complex conditional tests and use parentheses to clarify order of operations. Many languages have slightly different orders of precedence for logical and mathematical operations. Thus, it can be easy for another programmer to misunderstand what a complex statement is really doing, leading to a complete misunderstanding of a major construct.

Additionally, we should use meaningful variable names [LEW91, pp. 342]. This is helpful in that it saves another programmer the effort of referring to a data dictionary (assuming such exists) to understand the meaning of a variable. It is also much easier to catch a typo if it is in a variable name that is actually a word rather than in some meaningless combination of letters.

For the sake of robustness, we should add additional error checking to further verify user input [PRE92, pp. 680]. It is not at all uncommon, and generally not wise, to assume that the users of a system will only enter valid and reasonable values. If the previous programmers made this assumption, add additional error checking to make your system more robust and reliable.

Replace non-standard constructs with standard constructs where possible. This goes back to the fact that standard constructs are much more likely to port easily when changing environments. Likewise, standard constructs are more likely to be familiar to new programmers.

One consideration often overlooked is the reuse. Consider whether the subroutine is generic and reliable enough to be placed in a reuse library. As Lewis states [LEW91, pp. 348], "the best unit, procedure, function, or whole program is one that you do not need to write." This could save someone else from re-inventing the same wheel. If the subroutine as a whole is not usable in this fashion, portions of it or some of its algorithmic design might be.

Due to the correlation between size and number of errors [GRE84,LIE80], we can reduce the number of bug hiding places by carefully eliminating code that is no longer used or that is unreachable. Of course, we need to be quite sure that it is indeed no longer used (and is unlikely

to be used in the future) or unreachable (and shouldn't be reachable). Rather than actually deleting the code, we may want to investigate its value for reuse.

Similarly, look for sections of code that are duplicated elsewhere. If the same, or very similar, code is used in several places, that heightens the chances that one of those places might not be updated when all should be and introduces additional places for bugs to hide. Code of this sort might be worth separating into a more generic subroutine to add to your reuse library.

In the interests of thorough testing, we should add elements to our test plan as we think of them to ensure as thorough a test as possible. Any subroutine that has been re-engineered should receive thorough testing before being put into the production environment. We don't want to forget about an important test by delaying in writing it down.

Finally, don't throw away unquestioningly the old subroutine and rewrite it from scratch [PRE92, pp. 680]. You probably have a large amount of time and money invested in your current software so try to reuse concepts, designs, and code when possible. However, starting from scratch may be the best approach at times.

As shown in Figure 5, our work is far from complete when we finish the re-engineering of a subroutine. We must ensure that all related documentation (e.g., user's manual, maintenance manual, etc.) is up-to-date and reflects any changes made to the system.

We should update any design representations to reflect the new subroutine design. Perhaps most importantly, the programmer must conduct extensive unit-level testing according to his test plan and note and correct any errors found. This must then be followed by a careful integration and system-level testing that would preferably be conducted by an independent group of testers or configuration managers.

This proposal is just one method for conducting preventative maintenance, various other approaches could be used successfully. This approach should yield significant improvements in maintainability while guarding against the introduction of errors. Any other approach should cautiously balance benefits against risks and ensure that proper controls are in place.

7.3. The Promise of Preventative Maintenance

These preventative maintenance techniques start with the assumption that our existing software is indeed valuable but that, over time, our system hasn't been maintained up to today's standards. These techniques are varying approaches to the problem of improving the maintainability of our system. If used wisely, they should enable us to maintain our system in a way that we are more responsive to the customer and that the system is more reliable. They are a rejection, to some degree, of the old adage, "if it ain't broke — don't fix it." Software doesn't degrade over time as hardware does, but it can get farther out of line with our maintenance standards and thus degrade our ability to maintain it.

8. Conclusions and Further Research

We have seen that to retain its usefulness, software must change, and does. We have also seen that much of the software that we maintain today was written a significant number of years ago and that much of it didn't follow the software engineering standards of today. Therefore, our software may be quite difficult to maintain and programmers may require a sizable amount of time to simply decipher its purpose and logic before even beginning to maintain it. We have looked at some ways to optimize three standard

1. Update all related system documentations
2. Update all related design information
3. Conduct thorough unit, integration and system tests
4. Examine and record lesson learned

Fig. 5. Post Subroutine Re-engineering Tasks

types of maintenance to ease future maintenance and ensure greater reliability.

Finally, we suggested the need for some combination of preventative maintenance methods to improve the long term maintainability of our system and examined a framework for piecemeal re-engineering. We are often caught in a vicious cycle of scrambling to get a basic comprehension of a piece of software so that we may make the minimum changes necessary to meet the priority needs of the user.

To break this cycle, we must undertake preventative maintenance. By using preventative maintenance techniques, we can put our software in a more comprehensible form. A more comprehensible form should enable our programmers to understand it more rapidly (and fully), thus allowing them to maintain it more quickly and more reliably. Grady [GRA87] sums it up well, "because we cannot economically replace all our old software, we must find better ways to manage needed changes. Until we do, software maintenance will continue to represent a large investment — and software quality will not improve."

8.1. Areas for Future Research

Significant research has been done, and continues to be done, in software maintenance. One important area that will continue to be studied is the improvement of automated tools for software maintenance and analysis. Better automated tools may be able to generate much of the system documentation from the source code or restructure the source code to a programmer-defined standard.

Many aspects of a programmer's skill and training warrant additional attention. A greater understanding of how software maintainers do their work and what motivates them may enable managers to get a closer match between prospective employees and open positions. Butler and Corbi [BUT89] reference the fact that while software engineering is taught in a vast majority of Computer Science departments, almost none offer courses in things like "software renewal," "program comprehension," or "enhancement programming." Research into the necessity of specific training in such areas, and the best way to enhance skills in these areas, could lead to a generation of programmers much more adapt at software maintenance activities.

Finally, a more thorough analysis of large number of existing software systems will enable a better understanding of the challenges faced in software maintenance. This could provide valuable information such as the average system age, size, complexity, and frequency of change.

References

[BIG89] BIGGERSTAFF, T.J., Design Recovery for Maintenance and Reuse, *IEEE Computer*, July 1989, pp. 36–49.

[BRI90] BRITCHER, R.N., Re-engineering Software: A Case Study, *IBM Systems Journal*, vol. 29. no. 4, 1990, pp. 551–567.

[BRO82] BROOKS, F., Aristocracy, Democracy, and System Design, *The Mythical Man-Month*, Addison-Wesley, 1982.

[BUT89] BUTLER, R.B. and CORBI, T.A., Program Understanding: Challenge for the 1990's, *Scaling Up: A Research Agenda for Software Engineering, National Academy Press*, 1989, pp. 38–45.

[CHA83] CHAPIN, N., Attacking Why Maintenance is Costly — A Software Engineering Insight, *Proceedings of the Software Maintenance Workshop*, Monterey, CA, 6–8 December 1983, pp. 251–252.

[CHI90] CHIKOFSKY, E.J. and CROSS, J.H., Reverse Engineering and Design Recovery: A Taxonomy, *IEEE Software*, January 1990, pp. 13–17.

[ELI91] ELIOT, L.B., Overcoming Re-engineering Rigamarole, *CASE Trends*, Summer 1991, pp. 36–37.

[ELW91] ELIOT, L.B. and WEINMAN, E.D., Cleaning Up Old Software: Two Picture Perfect Case Studies in Re-engineering, *Information Week*, 22 April 1991, pp. 44–45.

[GLA92] GLASS, R.L., *Building Quality Software*, Prentice Hall, 1992.

[GRA87] GRADY, R.B., Measuring and Managing Software Maintenance, *IEEE Software*, September 1987, pp. 35–45.

[GRE84] GREMILLION, L.L., Determinants of Program Repair Maintenance Requirements, *Communications of the ACM*, vol. 27 no. 8, August 1984, pp. 166–172.

[HOU83] HOUTZ, C.A. and MILLER, K.A., Software Improvement Program — A Solution For Software Problems, *Proceedings of the Software Maintenance Workshop*, Monterey, CA, 6–8 December 1983, pp. 120–124.

[LEH85] LEHMAN, M.M. and BELADY, L.A., *Program Evolution: Processes of Software Change*, Academic Press, 1985.

[LEW91] LEWIS, T.G., *CASE: Computer-Aided Software Engineering*, Van Nostrand Reinhold, 1991.

[LIE80] LIENTZ, B.P. and SWANSON, E.B., *Software Maintenance Management*, Addison–Wesley, 1980.

[McC92] McCLURE, C., *The Three Rs of Software Automation: Re-engineering, repository, reusability*, Prentice Hall, 1992.

[NAV86] NAVLAKHA, J., Software Productivity Metrics: Some Candidates and Their Evaluation, *Proceedings of the 1986 National Computer Conference*, vol. 55, 1986, pp. 69–76.

[OSB90] OSBORNE, W.M. and CHIKOFSKY, E.J., Fitting Pieces to the Maintenance Puzzle, *IEEE Software*, January 1990, pp. 11–12.

[PAR87] PARIKH, G., Making the Immortal Language Work, *International Computer Programs Business Software Review*, no. 33, April 1987.

[PRE92] PRESSMAN, R.S., *Software Engineering: A Practitioner's Approach*, McGraw–Hill, 1992.

[SNE91] SNEED, H.M., Economics of Software Re-engineering, *Journal of Software Maintenance*, September 1991, pp. 163–182.

[WEI91] WEINMAN, E.D., The Promise of Software Re-engineering, *Information Week*, 22 April 1991, pp. 32–40.

Received: July, 1993
Accepted: March, 1994

Contact address:
Hossein Saiedian and James Henderson
Department of Computer Science
University of Nebraska, Omaha, NE 68182
Telephone (402) 554–2849
Fax (402) 554-2975
E-mail hossein@unocss.unomaha.edu

HOSSEIN SAIEDIAN is an assistant professor in the Department of Computer Science at the University of Nebraska at Omaha. Saiedian received his PhD degree from Kansas State University in 1989. He is member of the IEEE Computer Society, Sigma Xi, the ACM, and currently serves as the Chair of the ACM SIGICE (Special Interest Group in Individual Computing Environments). Dr. Saiedian's research articles have been accepted for publication in IEEE Computer, Computer Networks and ISDN Systems, Journal of Systems and Software, Journal of Information and Software Technology, Office Systems Research Journal, Computer & Security, Journal of Microcomputer Applications, and many others. His research interests include software engineering models, formal methods, and object-oriented computing.

JAMES HENDERSON received his M.S. degree in computer science from the University of Nebraska at Omaha in 1994. He has been directly involved in development of computer systems for the last 10 years. Mr. Henderson's research area includes software maintenance.