WHY IS SOFTWARE ALWAYS LATE?

J. L. Lawrence

ABSTRACT

Despite all of the advances in software engineering practice, despite all the newly developed languages and software tools, and despite case study after case study, software is almost always late. It does not seem to matter what the product is or what the industry is. The cry of frustration is almost always the same: "Why is software always late?" In this article, the author discusses the software development cycle and the LOC/day productivity measure in an attempt to explore some of the reasons for failure and some of the factors contributing to success. The opinions expressed herein are solely those of the author and do not necessarily reflect views or procedures of the General Electric Company or its employees.

INTRODUCTION

An amazing amount of progress has been made in the computer field since the early pioneering days of the 1950s. Desktop computers, realistic graphics, high level languages, artificial intelligence, computer vision, and many other advancements have been made in the space of only about 30 years. In addition to technology advancements, new application areas are being discovered virtually every day. Some describe the still infant field as a science while others call it an art and still others view the whole thing as black magic. Almost everyone appreciates the power and flexibility that computers have added to our lives. However, despite all the wonderful progress and potential, managers are invariably frightened and frustrated by this new technology. Software is viewed as unpredictable in terms of development time and effort and is hence viewed as a management nightmare. The universal cry is "Why is software always late?"

There is no simple answer to the question raised. There are in fact multiple factors contributing to the problem and it is easier to describe the factors than to formulate the solutions. It is unfortunate but true that most of the problems related to late software are outside the domain of the software development team. But, software development is not always late and does not have to be a nightmarish managerial headache. As an example, the author's companion article, *The RC2000: A Software Success Story*, describes a software development effort that was consistently on time and under budget. In the article which follows, the software development cycle and current productivity measures will be discussed to point out some of the factors that lead to and guarantee disaster. At the end of the article, some suggestions are made for ways to improve the situation and plan for success.

A TYPICAL SCENARIO

Before considering the problem in any detail, consider how many projects come about. Marketing does an analysis of the existing product line, surveys the marketplace, and decides that a new *widget* needs to be built. A preliminary schedule is thrown together to sell the idea to upper management to justify the development effort. Once the idea has been accepted, engineering is informed that they are to build a new *widget* with a certain set of features. If engineering is fortunate, a product planning specification might be included with the request. At this point, engineering goes off to write a functional specification of the new *widget* with inputs from the engineering staff and product planning. Meanwhile, marketing is out hustling up orders or doing more forecasting to see what the marketplace wants. Typically, there will be some normal changing of direction as marketing, product

planning, and engineering struggle to determine what the new *widget* should be. Morale and expectations are high because of the excitement over a new product that will be an engineering marvel, that will make the company lots of money, and will provide the company with a leadership position in the marketplace.

When the functional specification writers are done, a response is sent back to product planning and marketing which defines the new *widget* with the question, "Is this what you meant by a *widget* ?" By this time, firm schedules have been generated and committed to and sales forecasts have been made with the schedules assumed. Marketing, after deliberating over the contents of the functional specification, agree in principle but insist that certain new features must be added and the schedules must be pulled in by at least a quarter to meet the marketing window. Engineering, to save time, begins the initial design of the system using the "almost correct" functional specification. While some of engineering is doing the design work, others are incorporating the newly requested features into the functional specification and still others are generating detailed schedules for the software team. The detailed schedules must allow for delivery of the entire feature set identified by marketing and must meet the delivery dates already committed to.

When the new functional specification is ready for release, the high level design is either complete or nearing completion. Marketing reviews the new feature set, makes changes, and hands it back to engineering. "And by the way, engineering, if you could just move the schedule in by 2 months, there is this really prime sales opportunity we can capture that requires the new features we requested. We have already begun discussions with the customer, confident that you can deliver on time. In fact, we have scheduled a demonstration of the basic features already, so could you skip over some of the design work and begin writing code?" Meanwhile, the software development team has begun detailed design of a high level design that is based upon a functional specification that is at least two iterations out of date and are struggling to meet schedules that were generated long before the product was defined.

Generally speaking, several more iterations occur in which feature contents are changed that require changing designs which require changing working code. If the code is already in some form of testing, it may be decided that the system just does not "look" right and hence spur a lot of other changes to the user interface to make it more "user friendly". To save time, documentation and design reviews are abbreviated or dropped so as to increase the probability of meeting the delivery dates committed to.

As the delivery date nears, it becomes apparent that the software is going to be late. So, in conjunction with product planning and marketing, a reduced feature set is arrived at that is at least marginally acceptable for a first release. Some sales may be lost, but this is better than not having any product. In fact, an opportunity is realized in that the product can be cost reduced. This feat can be accomplished by merely taking out features that are not part of the first release and hence reduce memory costs. Thus, software engineering is told to restructure the product so as to make certain features easy to remove. In addition, more people are added to the project to insure its timely completion.

To cut the story short, the product is released late at higher cost than anticipated and with poorer performance. Many desirable features are missing and the competition has just come out with a new product that obsoletes or at least directly competes with many of the features in the new *widget* and to make matters worse, the competition's product is cheaper. At this point, management decides that development costs have to be reduced. This is accomplished by reducing staff or funding or both. Further, to appease customers, promises are made and schedules are set for making everything right with the customer. Thus, engineering simply can not fail in meeting the new feature set and the new schedules. To make sure that such a disaster never occurs again, a "witch hunt" is conducted to find the guilty parties and a reorganization is done to better align the organization to meet the new challenge. Morale is low as people are disillusioned, there are rumors of more layoffs, there are rumors that the project will be canned, and people begin to analyze the problem and

wonder what went wrong.

Meanwhile, product planning and marketing survey the market place and decide that a new *gadget* is needed.....

SOFTWARE DEVELOPMENT CYCLE

While the above scenario may be exaggerated a little and may not be completely fair in some respects, it probably hits closer to home than one is willing to admit. In fact, it may make one somewhat uncomfortable. One difficulty in solving the "software problem" is in admitting and realizing that the above scenario is all too descriptive of the way development work is done. The maintenance and feature enhancement cycle is not even mentioned, but suffice it to say that these work efforts only add to the problem. To analyze what went wrong and why software is late, one must begin with an understanding of the software development cycle.

Functional Specification Phase

The software development cycle is a progressive cycle of design, inspection, and rework. Rework must be scheduled for, especially in the face of the reality that the product definition will be changed several times during the development cycle. A rough rule of thumb is that at least 10% of the time should be scheduled for rework due to design flaws or misunderstandings. Extra time must be scheduled in case of Functional Specification or program direction changes. The amount of rework time required is difficult to predict beforehand.

The development cycle must begin with the Functional Specification Phase. Some prefer the term Requirements Analysis, but regardless of the term used, the first step is to clearly identify what the software is to accomplish. The Functional Specification should be viewed as engineering's response to the Product Planning Specification. More importantly, the Functional Specification is a "contract" between engineering and marketing which describes what is being built. As such, it must be a joint effort between product planning and engineering. All too often this step is omitted and design is begun based upon a product planning document. This is undesirable because a Product Planning Specification, if properly done, will not be technical enough in content to resolve the many important technical details that must be attended to. Frequently engineering is forced to develop a Functional Specification without a Product Planning Specification and little or no input from marketing and product planning.

The Functional Specification Phase is the proper time to define, in engineering terms, what the product baseline is to be and to precisely define all features that are to be part of the product. In conjunction with product planning, the Functional Specification should define what the product releases are to be and what features are to be product options. The product option packaging is important so that engineering can design features as options to be easily removed or added in. Furthermore, the choice of which features are options and how the product is to be packaged has a tremendous impact upon how the software is to be designed, how manufacturing is to produce the product, how field updates are to be accomplished, and how configuration management is to be done. An additional benefit is that an early option packaging provides marketing with inputs for product pricing.

An item that is frequently ignored or minimized during the Functional Specification Phase is the user interface. The user interface is a feature of the product as much as anything else and should be as completely specified as early as possible. To make matters worse, practically everyone has an opinion on how the user interface should operate. Leaving this vital decision until coding begins or until marketing sees the final product will create a disaster and needless friction. A section of the Functional Specification called the Operator Scenarios is an appropriate place to describe in fairly great detail how the

product will look to the customer. This section of the document can later form the basis for a user's manual. An important point to remember about the user interface, or any other decision deemed too important for software to make, is that if is important it must be clearly spelled out in the Functional Specification.

It is critical that software engineering be adequately represented during this phase of the development cycle. Too often the Functional Specification writers have little or no software background and frequently make poor decisions about how software will be done. As products become more and more software intensive, the tendency to ignore software concerns at this stage will more and more guarantee failure.

Another frequent mistake that occurs at about or just before this phase is the development and commitment to "cast in concrete" schedules. During the Functional Specification Phase, preliminary schedules only should be generated which give a rough indication of what features will be available at what time. In many cases, as in the scenario depicted above, schedule generation and commitment is done long before there is any understanding of what is being built.

The Functional Specification Document provides the engineering cornerstone for the entire development effort. It is the overall blueprint from which all other work is derived. Proceeding with development without such a blueprint is a guaranteed way to create unimaginable problems later on during the development effort. Engineering should refuse to proceed with the development effort until the Functional Specification is approved through a "sign-off" process. Still, in an attempt to speed up the process, proceeding without a clear functional specification is often mandated. When changes in the product feature set are made, the Functional Specification must be updated to reflect the changes and all activities which are affected by the change (design, code, test) must be updated also. Additionally, when feature sets change, schedules must be examined again to access the impact. Failure to do so guarantees problems later on. Dropping one feature to include another or adding additional resources does not necessarily mean that the original schedules can be maintained. Frequently such changes are made with an acknowledgement of the associated risk that is quickly forgotten later on.

High Level Design Phase

Once the Functional Specification Document is completed, software engineering's task really begins. While there is usually little software input into the Functional Specification process, there is frequently an abundance of "help" with the development of the high level design. Although other engineering (*not* marketing *nor* product planning) functions should be represented during this phase, this task is primarily a software engineering exercise.

During the High Level Design Phase, the software design is decomposed into a specification of major software subsystems, task definitions, and subsystem interfaces. It is extremely important that interfaces be completely specified (actual code is preferable) during this phase. Changing a subsystem interface amounts to a design change which will be disastrous if postponed until the Module Design or Module Coding Phases.

The High Level Design Phase should include the identification of an overall software design philosophy and methodology that will be followed throughout the software product development. This assures consistency during the development process.

In conjunction with the Functional Specification writers (or whoever has ultimate engineering responsibility), the High Level Design developers establish memory/performance/feature tradeoffs. With this clearly in mind, the software engineering group can define a software baseline which may or may not coincide with the feature set for a product baseline. Once this is complete, a set of detailed schedules can be generated and committed to. Changes to the product further upstream (Functional Specification Phase) will necessitate some amount of rework in this phase and potentially have a schedule impact.

Subsystem Design Phase and Module Design Phase

Once the High Level Design Document is complete, a set of software subsystems will have been identified. These subsystems are designed in the Subsystem Design Phase by decomposing them into Modules. Notice that the High Level Design Document acts as a requirements document for the Subsystem Design Phase. Based upon the Subsystem Design Documents, modules are defined which are in turn designed during the Module Design Phase. The Subsystem Design Document, in addition to defining what the modules are, should completely specify the intermodule interfaces.

The Module Design Document normally includes pseudo-code for the module. If properly done and the tools are in place, it is possible to generate much of the code directly from the pseudo-code.

Coding/Debugging Phase

When a Module Design Document is completed, actual coding can begin. Some amount of testing is done in this phase, but the real testing is described below. The Coding/Debugging Phase is no place to do design work. Problems may be encountered through the debugging process, but these should be resolved by going through the design process (typically just the Module Design Phase) again.

Unit Test Phase

Once a module has been coded, testing begins. The Unit Test is a test of the module in isolation from the system. Its purpose is to ensure that for a set of inputs, a valid set of outputs are generated. It will usually be necessary to write software stubs and drivers to be able to adequately generate test cases for the Unit Test. Once the Unit Test has been successfully completed, one has a high degree of confidence that the module itself internally meets the requirements.

Integration Test Phase

During the Integration Test Phase, modules which have successfully passed the Unit Test Phase are combined together into a "system". Stubs and drivers are replaced by actual code. The Integration Test Phase is primarily directed towards testing out interfaces since logic tests have pretty much been completed during the Unit Test Phase. The Integration Test is a "white box" testing approach in which tests are constructed by knowing what the interfaces are and what the results are supposed to be.

Functional/Acceptance Test Phase

The last phase of testing is a series of "black box" tests. These tests are a rigorous comparison of the product with the original Functional Specification and Product Planning Documents. To prevent bias, the Functional/Acceptance Test Phase should be conducted by individuals other than the product developers.

Development Cycle Summary

Whatever the actual terms used, software development projects go through the phases listed above. Maintenance and feature enhancements are not described because they can essentially be thought of as new "products" being developed on top of an existing baseline (the current product). Support activities such as computer facilities and configuration management are also not mentioned in the development cycle. These activities are important in the development cycle, but they are issues in their own right and are not as

pertinent to the immediate discussion.

Experience has shown that roughly 40% of the development effort is in the specification/design phase, 20% in the coding/debugging phase, and 40% in the testing phase. Thus, it is clear that the actual software writing process occurs for a relatively short period of time and that most of the work is in planning and testing. This is important to realize because when one talks about productivity improvements, the amount of time spent in each phase of the development cycle must be considered. Given the above remarks about the development cycle and where most of the time should be spent, it becomes clear why changes in feature content can have such a tremendous impact upon software productivity.

The development cycle described depends upon individuals with appropriate skills at each level. The Functional Specification Phase must be done primarily by a systems engineering component with heavy Marketing/Product Planning input and software input appropriate to the product. The High Level Design Phase is primarily a software engineering function and should be conducted by a chief architect (or small team of architects) with appropriate consultation with the Functional Specification writers. The Subsystem Design Phase is performed by a software engineer who consults with the chief architect as necessary. Module Design, Coding/Debugging, and Unit Test are performed by individual software engineers or programmers as appropriate for the complexity of the module under consideration. Integration Testing must be performed by the software team itself. Finally, Functional/Acceptance testing is done by an independent team. Functional/Acceptance testing is a good way to train field service personnel for the new product.

It is important to emphasize again the impact of changes at any point during the development cycle. A change at any level may affect each level beneath it. Changes at the Functional Specification level are among the most frequent and most undesirable because all levels are affected by it. It is also important that the objective of each level be kept in mind. Specifically, the testing phases are no place for design work to be done. An individual doing functional testing should certainly point out perceived weaknesses in the human interface, but before changes are made at the module design or coding level, the Functional Specification must be changed because a change to the human interface normally equates to a new feature.

CLASSIFICATION OF SOFTWARE PERSONNEL

Before diving into the question of software productivity, a few classifications are in order. While there is no general agreement on how to classify skill levels for software personnel, the following titles are submitted as a representation of the range of skill levels. A comparison (hopefully accurate) with hardware is also presented.

Software Engineer

A software engineer has the skill and experience to be able to design the software portion of the product. The most experienced software engineers would be chief architects. Less experienced software engineers would typically be responsible for subsystem design and relatively complex module designs. An analogy with hardware is to compare the software engineer with the hardware engineer who would design a computer down to the board level and would then turn the boards over to more junior engineers to design at the board level.

Programmers

Programmers take over where the software engineer leaves off. That is, a programmer would typically take a module and design it. The major differences between a programmer

and software engineer are in their experience level and the complexity of the design task they are given. An analogy with hardware would be to liken a programmer to a hardware engineer who takes a board description and does the circuit design for it, or who takes a complicated board and does a circuit design for one part of the overall board.

Coders

Coders take well defined, designed modules and turn them into actual code. The information given to a coder might be in the form of a flow chart or some other detailed specification. The coder is like the hardware technician who takes a schematic and soldering iron to build a board.

Classification Summary

The classifications above are not exact and often overlap. Just as with hardware engineers, there is often a gray line between skills. Experience and training are typically the deciding factors in placing an individual in the categories above. The important thing to note is the job responsibility for each classification and the training/experience level required. Software engineers require quite a bit of training and experience to be able to make intelligent decisions about design alternatives for large software systems. It is the author's view that software design is usually more complex than hardware design because of the large number of interfaces. (This view is certainly debatable.) As a result, training and experience level is of paramount importance. Few colleges and universities prepare "computer science" students to be software engineers upon graduation. Most graduating students are really in the category of programmers. That is, they are able to take small, yet significant, portions of a software project and do the design work. Finally, coders require the least training and experience of all. Most people can make good coders and many colleges offer two year programs for just such jobs. Again, the categories presented are not clear cut but give a general idea of the range of skills present in a software project.

To add to the confusion, job duties often overlap during a project. For example, the author's team does not include any coders because all of the software engineers and programmers do their own coding tasks. The problem occurs when the qualifications do not match the task. Many times software engineers or programmers are hired by a company and given only coding tasks to do. This leads to job dissatisfaction and a high turnover rate. Likewise, companies often give individuals with minimal coding skills the software engineering tasks. Of course, this is not done intentionally but happens nonetheless. The difficulty occurs because of the mistaken idea that just because an individual has several years experience with a product, they are qualified to do the software design. This leads to frustration, overly complex software designs, and poor performance. Likewise, there is a tendency to allow a software engineer with no product experience to define a product. Both extremes will guarantee problems and must be avoided.

To rephrase the problem as a question that is frequently asked, "Why does the software take so long when my 15 year old nephew can do it on his home computer?" The obvious hardware analogy is "Why does hardware take so long when my 15 year old nephew designs and builds crystal radio sets at home?" There is a world of difference in writing a program and producing a product. The electronic products produced by professional engineers are far more sophisticated and complex than a simple crystal radio set. The same is true for software products. Another factor to keep in mind is that a simple program being done at home for entertainment purposes is easier because there is little or no need to worry about quality, long term maintenance, or manufacturing concerns. Furthermore, the feature content of a home program in general does not begin to approach that of a real product. There are certainly some good software products coming out of home "garage shops", but most such developers really are in the category of coders.

SOFTWARE PRODUCTIVITY

The above discussion described the software development cycle and the types of individuals that are required for software development. Several pitfalls were pointed out that can be avoided by proper discipline, planning, and management support. It is now time to consider the central issue of software productivity.

Measuring Productivity

The most commonly used measure of software productivity is lines of code per day (LOC/Day). This measure is fairly easy to arrive at once the code is written and is used by several planning models to predict software development costs. However, it is the author's opinion and experience that LOC/Day is a poor measure for several reasons. Before elaborating upon this, consider how the LOC/Day measure is arrived at. The LOC/Day figure is calculated by counting up all of the lines of source code that are in the product delivered to a customer and dividing that by the number of man-days required to produce the product. Only *delivered* source code counts. Thus, software written as stubs, drivers, or as tools does not count in the LOC/Day number even though resources were required to do the work.

There is hardly universal agreement on how to measure several of the factors that are part of the LOC/Day measure. What counts as a line of code? Do comments for documentation purposes count or only executable statements? How should deleted lines of code or modified lines of code be counted? How does reusable code count? How does one account for assembly code versus high level language code? When does the counting start? Does it begin during the requirements phase, design phase, or coding phase? Who gets included in the man-day count? Do managers, clerical support, and non-software engineers get counted or only those who actually produce code? These questions can be irrelevant for a local group that is trying to measure gains in productivity. Unfortunately, that is not typically how such LOC/Day measures are used. They are frequently used to measure productivity against other industries, other projects, or the Japanese. Thus, how the LOC/Day measure is arrived at becomes a critical issue because by judicious choices, productivity can be made to look extremely good or extremely poor.

As was already stated, it is the author's opinion that LOC/Day numbers are poor productivity measures. The major reason for this is that there is a great tendency to misinterpret the meaning of such numbers. For example, if it is stated that the productivity of a particular group is 6 LOC/Day, management response is typically "Why is it so low? Get rid of those lousy programmers and hire some good ones." Taken out of context, LOC/Day numbers are as easy to misunderstand and misrepresent as statistics. If properly used to measure progress in productivity gains, much of this argument can be discounted. Furthermore, it is important to realize that the LOC/Day measure defined is a project productivity measure, *not* just a software productivity measure.

A second reason for disagreeing with the LOC/Day measure is that by its very nature, there is a built in but unmeasured penalty for Functional Specification or program direction changes. Thus, there is a tendency to again misinterpret the results and see the whole problem as a software issue when in fact it may be a procedural and discipline problem. Because only delivered lines of code count, much necessary work is often a penalizing factor. It is often the case that more code must be written to test a module than actually comprises the module itself! Yet, the test code takes resources and is hence a negative productivity factor! Examples of this situation abound in practically every development effort. In one specific instance, this author developed and debugged 40K of code in one day (much was reusable code), but since the code was not part of the product, that day was "wasted" by LOC/Day measures. In another case, a major software module was rewritten at least three times due to fundamental changes in the Functional Specification Document resulting in a low productivity number for that module.

Perhaps the most dangerous aspect of the LOC/Day measure is that it attempts to describe 100% of the development cycle by measuring only 20% of the effort. Recall from the discussion earlier that actual coding is only about 20% of the total effort. Hence, documentation effort, design effort, tool development, configuration management, and testing are not adequately measured.

Because of the LOC/Day measure, code size estimates are often used to try and determine software effort. This seems perfectly reasonable but suffers from many of the same problems as the LOC/Day measure. The fundamental flaw is the derivation of the unknown (software development effort) from a series of unknowns. Code estimates determined by inexperienced software people estimating code that has never been written before for a product definition that is constantly changing are bound to be in error! To make matters worse, such estimates are derived at or about the same time as the schedules – before the Functional Specification is even written! While it is perfectly reasonable to want to estimate effort and measure progress based upon LOC/Day, in practice it rarely works well because of misinterpretations, unmeasured factors, and events outside the control of the development effort. If the code size estimates turn out to be accurate and schedules are met, the claim is made that the schedules were too easy and not challenging enough. If the code size estimates turn out to be wrong and schedules are not met, a "witch hunt" is sure to follow.

A related and important difficulty with the LOC/Day measure is that measurements can be taken only during or after the Coding Phase. This means that 40–60% of the development cycle is over before measurements are taken and if the project is in trouble, the proof comes too late. Used in this manner, LOC/Day becomes an autopsy report rather than a health checkup.

Given the comments made concerning LOC/Day and code size estimates, should such measures be abandoned? The answer is not a clear cut yes or no. It depends upon how the information is to be used. If the data is used as a planning guideline and measurement tool because no other measurements are available, LOC/Day can be very valuable. This pretty much means that the LOC/Day and code size measures should be used only for internal monitoring. If the data is used as a test to determine "who's the witch", inestimable damage has been done by the data collection. In reality, such measures are most likely presented to upper level management with the accompanying misinterpretation and more harm than good is done.

It has been the author's experience that LOC/Day measures and code size estimates can be valuable aids as long as schedules are met and immediate managers understand the validity of such measures. Still, LOC/Day is not the measurement of choice. What is preferable is to use measurements and estimation techniques that account for each phase of the development cycle. With this scheme, to take the Coding Phase as an example, code size estimates would be made during the design phase to predict Coding Phase effort. An LOC/Day measure at the end of the Coding Phase would measure the productivity and success of the effort.

More work is definitely needed in the areas of productivity measurement and software planning tools. There are other ways to measure productivity that have been suggested in the literature. It might be possible to measure and predict the Functional Specification Phase on the basis of a technique called function point analysis. Metrics such as the McCabe metric might be used to measure design complexity and hence indirectly indicate productivity and quality. Finally, during the testing phases, the number of errors detected are probably a good starting point for measuring quality and productivity.

Factors Affecting Productivity

Regardless of how one measures software productivity, it is apparent that software productivity must be increased. Fortunately, it can! Several factors affect software

productivity. The amount of code to write, the programming language chosen, the complexity of the software, and the use of modern programming practices all have a great deal of effect upon the success of a software project. These factors have been explored extensively in the literature and are almost to the point of being "motherhood". Factors which have the most impact on productivity are often overlooked because they lie outside the domain and authority of the development team. Some of these negative factors will be considered now.

(1) Without a doubt, the single largest negative factor affecting software development is in attempting to build and design against a moving target. A frequently changing or out of date Functional Specification Document is a guaranteed formula for failure. It need not be that way.

(2) Schedules that are generated and mandated are often unrealistic and incomplete. Schedules defined and committed to without engineering input are not likely to be accurate. All too often such schedules are made up based strictly upon marketplaces requirements without much consideration for what is doable. Incomplete schedules lack sufficient time for proper documentation or rework and generally are not revised even when the feature set changes.

(3) The "Make it Work" syndrome puts on pressure to limit or even omit "nonproductive" development phases. The pressure to "quit designing and start coding" guarantees that poor designs will be implemented and set the stage for ultimate failure. It is true that one can not design forever or until the optimum solution is found, but the balance seems to have swung too far in the opposite direction.

(4) The "Anyone Can Write Software" syndrome causes a skills mismatch that is frequently not realized until it is too late. Coders performing software engineering tasks assure missed schedules, poor quality, and missed performance budgets. Assigning coding tasks to software engineers guarantees low morale, high turnover rates, and disillusionment.

(5) The "It's Only Software" syndrome results in the legislation of how software is to be done. Rules governing software development are frequently made up by people who have never done software development before or have limited exposure to modern programming practices. A common belief is that since software is so easy to write and change, anyone can do it and hence not only are all software developers interchangeable, procedures and rules developed for one type of software industry (such as data processing) are applicable to all types of software industry (such as real time software).

(6) Most development efforts suffer from the lack of adequate tools. Software engineering may be in the "Bronze Age" but many are using only "Stone Age" tools or "Bare Hands". Unfortunately, few managers and software developers are aware of what tools are available or are trained in their use.

Improving Software Productivity

There are a lot of reasons why software productivity is low, why software is "always" late, and why software projects often fail. Eliminating or at least improving the negative factors described above may well be enough to make the overwhelming majority of the software projects successes. There are specific changes which are immediate hits.

(1) "Chill" the Functional Specification. It is generally impossible to completely "freeze" a Functional Specification Document and guarantee that no further changes will be made. But, the amount of changing can be minimized by doing the upfront job correctly and requiring a realistic schedule impact assessment as features are changed.

(2) To facilitate understanding by marketing and product planning, develop rapid prototypes during the Functional Specification Phase. Software prototypes have several advantages. First of all, they provide a concrete vehicle for communication between engineering and product planning rather than some abstract document.

Secondly, prototypes allow all concerned to see and evaluate alternatives at an early stage when radical changes in direction are the least expensive to recover from. Thirdly, prototypes give engineering valuable insight into the problem at hand and thus give potential direction for proceeding with design. Finally, prototypes give everyone an early indication that the product is real and what it is going to look like. It is perhaps heresy, but there is no less a need for prototypes in software than in any other discipline. Yet, while other disciplines develop throw away prototypes, software is expected to deliver correctly on the first attempt. This can lead to disasters.

(3) Improve up front planning by utilizing existing tools (COCOMO model or alternatives) to at least have a best guess estimate of the effort involved. Estimates should be made and reviewed by qualified software engineers. Management and marketing must understand the questionable validity of such estimates.

(4) Define a product and software baseline before design begins. By doing this, a staged implementation strategy can be used for software development. Designing all features before any implementation begins is asking for trouble because time and patience will likely run out before anything workable is produced. A staged implementation strategy allows a base system to be built early and features added in stages in a "layered" or "peeled onion" manner.

(5) Increase the software input into all planning phases from management to marketing to product planning to systems engineering. This is required as products are more and more software oriented.

(6) Give software developers terminals to take home. The more access to a computer, the more work that will be accomplished. It is often the case that a developer will realize the solution to a problem after he has left work and if computer access is immediately available, he is likely to solve the problem then and there. Additionally, terminals at home allow engineers to submit batch jobs at the end of the day and monitor the progress from home.

(7) Let software people do their job. A software engineer is just as much an engineer as someone from another discipline and should be treated as such.

Most of the above "hits" can be accomplished reasonably quickly to improve productivity and the chances of success immediately. They require management support, discipline, and initiative. There are also longer term investments that will pay off in 1–2 years.

(1) Invest in tools. A programmer's tool kit should be developed and made available. Many tools are already available for systems with UNIX® and other operating systems while other tools may simply have to be developed in house. Great care must be taken not to view tools as a panacea or cure all. Most tool kits are oriented towards the Coding Phase and as such only address 20–30% of the development cycle.

(2) Automate testing. Test coverage tools can be generated fairly easily to assist in the testing phase. Such tools help determine if a module, subsystem, or system has been tested adequately. Tools can be developed or already exist to measure design complexity using some criteria such as the McCabe metric.

(3) Standardize development methodologies and procedures. "In order to form a more perfect union", this should be a "government by the people and for the people." In other words, methodologies and procedures should be developed by the people who are going to use them and who are most qualified to establish them. Standardized methodologies increase uniformity and make quality control problems easier. If developed by software developers in conjunction with a QA organization, they are more likely to be followed and succeed.

®UNIX is a trademark of AT&T Bell Laboratories.

(4) Automate configuration management. Manually controlling what software is required for what feature set and for what release can quickly become and unwiedly task. It can easily decrease productivity and slow releases.

(5) Reuse code wherever possible. This statement is also very close to "motherhood". Almost everyone agrees with the concept, but little is done to design for reusable code and little is done to make developers aware of what reusable modules might already exist.

(6) Appoint a chief architect with the authority and responsibility to make decisions. The chief architect concept is nothing new but it seems to be rarely used. A chief architect (or small team) should understand the entire scope of the project at hand. He should interface to systems engineering, product planning, and the developers. His task is to provide a single point of contact for developers who have issues that need to be resolved. These issues might be technical in nature, or may be interpretations of the Functional Specification Document.

The ideas suggested above can all improve productivity fairly quickly. However, productivity improvement is an ongoing process. Software developers, and their managers, must keep current with new advances that are being made. Other productivity enhancement areas should actively be investigated. These include productivity measurement alternatives, workstations versus mainframes, software estimation/planning tools, rapid prototyping tools, and machine independent compilers. The key is to establish an on going goal of improving productivity and a serious program for accomplishing the goal.

CONCLUSION

This article has discussed the central issue of late and unpredictable software by examining the software development cycle, measuring productivity, and factors contributing to low software productivity. It has been pointed out that many factors contributing to software disasters lie outside the domain of the software development team. Several suggestions, both inside and outside the software development team, have been given for improving the chances of software success.

ACKNOWLEDGEMENTS

The author is grateful to the past and present members of the Proffitt Road Gang for their dedication and inspiration. Their success has proven many of the concepts described above and validated many of the author's opinions. This development team is quite possibly the most productive, talented, dedicated, and dependable in existence.

ABOUT THE AUTHOR

The author received his undergraduate degree in mathematics from Western Kentucky University in 1976. His masters and doctorate degrees are both in computer science from the University of Missouri-Rolla and were granted in 1978 and 1980 respectively. The author was a member of the Mathematics and Computer Science Department of Western Kentucky University until 1981 at which time he joined the General Electric Company. He is presently the manager for Robotics Software and Software Engineering at General Electric's Charlottesville, Virginia facility.