Costs Down 80%—Error Rates Down 75%
Applying the SEI Model to the SAS Programming Environment

by
Jeffrey Phillips and Robert Hall
Computer Sciences Corporation

Abstract

The Software Engineering Institute's Software Maturity Model has been measured to reduce costs by 80% and error rates by 75%. It provides a programming framework which offers significant time/cost savings to application development efforts. These savings are effected through the use of standardized procedures for all software activities. The paper describes the SEI model and offers some examples of the types of procedures that are traditionally developed. This paper then describes the applications of the SEI model to two CSC projects: a classical fixed-duration and fixed-cost application development effort and a broad-based data processing support effort. We will describe what procedures we implemented and how well they worked within each situation. The paper will describe where we achieved the most success in implementing the model and the impact it had on the software teams.

Introduction

The SAS System was among the first application development tools to focus on the information analysts' use of computer data. The original developers of SAS in the late 1960s and early 1970s sought to eliminate the barrier that existed between researchers, and their data, created by cumbersome computer languages such as FORTRAN and PL/1. In overcoming that barrier, SAS anticipated by almost a decade the revolution that occurred in computing when power formerly reserved for the mainframe computer programmers was placed on user desktops. SAS offered a combination of ease of use and data processing power that was and still is unique in the computer industry.

Unfortunately, SAS' success has had some unintended consequences. Because SAS is relatively accessible to non-programmers, it has been seen primarily as an end user analytical tool rather than a professional development platform, and SAS development is often done out of the production mainstream. Even when IS departments "do SAS", they often use it for ad hoc reporting or other non-production applications. Hence, SAS development never gets managed the way COBOL development does. On the other hand, because SAS is so powerful many SAS applications developed by analysts and other non-IS professionals have grown far beyond their original intent. There are many SAS applications that "sneaked into" mission-critical system roles in this way. Many such SAS applications were never properly designed or tested for their roles at the enterprise level, and their continuing enhancement and maintenance gives the departments who inherit them cost, schedule, and quality nightmares.

In short, SAS, along with other fourth and fifth generation tools, suffers from an extreme case of the condition
that afflicts the software industry in general. There is no problem with the tools themselves. Rather, the problem is with the process by which applications are being developed.

Software development is an extremely young industry, and its technology has far outpaced its methods and disciplines. Recently, however, there has been substantial movement toward formalizing and controlling the software development process, so that the term software engineering can be truly be used to describe it. One such effort is the Carnegie Mellon University Software Engineering Institute’s Capability Maturity Model (CMM), a program undertaken by the US government, the software industry, and academia to impose order on the software development process. This paper discusses the reasons for and origin of the CMM, and presents as a case study a Computer Sciences Corporation (CSC) partnership with one client to apply the CMM to the problems of software development in a mixed SAS/non-SAS environment.

Background

The Crisis in Software Development

In a world once ruled by mainframes and batch-processed COBOL, desktop computing, local area networks, and client/server technologies have suddenly emerged. Modern applications have gorgeous graphical user interfaces, can distribute their workload over multiple machines, and place tremendous control in the hands of the end user. Unfortunately, the breakneck pace of hardware and software tool evolution has far outstripped the development of the software engineering discipline. The results are all too plain: development costs and schedules out of control; applications that work poorly or never work at all; and users who feel constantly shortchanged by the promise vs. the reality of the “client/server revolution”. A recent article in Scientific American describes two well-known software development fiascoes: the Federal Aviation Administration’s Advanced Air Traffic Control System (AAS) and Denver Airports automated baggage control system. In the first case, the FAA largely abandoned its ambitious, state-of-the-art AAS because its costs had grown far out of proportion to original estimates, the schedule could not be met and there was growing evidence that the massive system might never be able to do what it was designed to do. At Denver, an entire new airport stood idle months after its scheduled opening date, in large part because of a much-heralded baggage routing system that has never worked in spite of the many attempts to correct its problems.

These examples are extreme illustrations of the problems most IS professionals and their customers and clients have experienced directly: software development is out of control. Developers often cannot predict how long a project will take, how much it will cost, or how good the result will be. Even if one project is a success, there is usually no guarantee that the next one will be, and customers are left with a less than trusting feeling about what an IS project manager tells them. Another aspect of the problem has recently been called
"client/server backlash:" IS and LAN administration staffs are struggling to maintain control over the thousands of applications developed both inside and outside their departments that adhere to few or no standards and are developed by people of varying backgrounds and talents.

Overcoming the Crisis: Tools, Methods, and Procedures

Ironically, the application software industry has responded to the problem of managing software development primarily by throwing more software at it. Advertisements abound in any professional journal for better development tools, CASE tools, version control packages, application generators, and project management tools. In the SAS world, SAS/ASSIST, the Display Manager, the DATA step debugger, and other such enhancements to the basic package have been created to help with the application development process. Tools, however, do not solve the problem. At best they can make an already good developer more productive; at worst they isolate the developer from the code being created and generate applications that can be extra hard to maintain because of the template code they contain. CASE tools are worse than useless if developers do not have a firm grasp of their underlying methodology, and the most powerful CASE tools are also the most complex, adding yet another layer of software to learn. While an integrated suite of tools can add value to a development team that is well versed in their use, tools in and of themselves do not solve the underlying problems of consistency, predictability, and quality that beset the software industry.

Project management packages can be extremely helpful in controlling a software development effort, but are only useful if (1) the software requirements are managed well; (2) the planning is based on accurate data about how long it should take for x module to be done by y number of programmers; and (3) the effort is tracked and measured as it proceeds so that the project plan continues to have meaning. Too often, a project is laid out based on incomplete requirements, wishful thinking about productivity, and the desire to meet deadlines and budgets that come from corporate boardrooms, not from a solid basis of facts about how long things should take and how much they should cost. Later, no one on the project has time to update the plan based on actual progress, and the only time the plan sees the light of day is when the executives who set the deadlines in the first place ask why they have not been met.

Methodologies are another incomplete response to the development crisis. Beginning in the 1960s with structured programming, continuing into the 1970s when Yourdon, DeMarco and others began to discuss the system development life cycle, and moving on through the 80s and 90s as object oriented software engineering (OOSE) emerged, methodologies have been offered to provide a theoretical basis for software development. Advances in methods have been useful in advancing the industry; SAS developers have benefitted from each of the paradigms discussed above. However, nothing
inherent in a methodology forces developers to use it consistently or correctly. The current problem in software development is not that methods do not exist, but that they are inconsistently applied and that programmers are poorly trained in their use.

No tool or method, however sophisticated, can overcome a poorly managed software development process. It is the process that determines whether staff are given good tools, trained to use them, and forced to use them on every phase of every project. Only a disciplined process can dictate that project plans must be based on objective reality and updated and reviewed regularly -- and that corrective actions are taken as soon as progress deviates significantly from the plan. Paradoxically, in a period when the overwhelming trend is to decentralize application development and "free the user," the only real answer to solving the software development crisis may be to exert authority and control over the fundamental activities that go into producing applications. What is needed is not a better tool or a better methodology, but a better process.

**The Capability Maturity Model**

If process improvement is required, then the question becomes how to go about it. The first step might be to define a good process, and then work backward from there. If one considers the process of developing software as not too different from that of building a manufactured item, the elements of a good process become relatively easy to imagine. The ideal software development process would be:

- repeatable
- well defined
- managed
- optimizable

The ideal process would be *repeatable*, so that results could be predicted from project to project, regardless of normal personnel turnover. The process would be clearly *defined* throughout the organization so that all developers would know the process and be able to follow it as well as make suggestions for changing it when required. The process would be managed -- that is, measurements would be taken at every step to keep tabs on how well the process worked in terms of cost, schedule, and quality. Finally, the process would be *optimizable*, so that the measurements taken could be used to improve it continuously.

If this ideal process is compared to the typical one at most software organizations, the distance between "ought to be" and "is" becomes depressingly clear. Where the ideal process is repeatable, most software development organizations exist in a state where good things are done, but no one can really say why. The quality of the product is often directly dependent on the talent and productivity of individuals, and no manager can say with any certainty that if project "A" succeeded and was done on time that project "B" will also be successful. Where the ideal process is well-defined, most software shops have little or no written procedures that govern software development -- and many which do rarely follow them. Where the ideal process is
managed, most software managers in the real world are too busy responding to crises to plan well, much less to take systematic measures of the effectiveness of their software development efforts. Finally, where the ideal process is optimizable, most real software development processes cannot be optimized because they are neither repeatable, defined, nor managed. If the ideal software process is summed up by the word "control," the real world software process is, for most of us, summed up by the word "chaos".

All this is not to say that everyone produces bad software in a state of utter confusion all the time. In fact, a great deal of very good work goes on in our so-called "real world". The problem is that software managers have lacked a roadmap to help them identify and keep the good while weeding out the bad. What has been lacking is a plan to get from "chaotic" to "optimizable". In the early 1990s, the federal government, in partnership with Carnegie Mellon University, came up with just such a plan: the Capability Maturity Model (CMM).

The Purpose and Scope of the CMM

The CMM grew out of a research project on the software development process jointly sponsored by the Defense Advanced Research Projects Agency (DARPA) and Carnegie Mellon University. DARPA and Carnegie Mellon set up a research institution called the Software Engineering Institute (SEI) and invited academics and software development experts from government and industry to come, study, and share their results. The CMM, first published in 1991, was one of the results of their work.

The purpose of the CMM is to focus purely on the process of building software, without regard to the purpose of the software, the platforms, languages, or methods used. The CMM attempts to abstract the key elements of any software development effort and to present them in such a way that an organization can use the model to build a coherent process for itself and move down the path from "chaotic" to "optimizable" in a systematic fashion.

The Five Levels of Process Maturity

The CMM defines five levels of process maturity. Each level focuses on one aspect of the ideal software process, and is defined by a number of "key practice areas," or KPAs. Level 1, "Initial," corresponds to the situation discussed as "chaotic" above. Level 2, "Repeatable," focuses essentially on writing down the steps that an organization undertakes to develop software, whether they are effective or not, and then on turning those steps into a repeatable set of procedures. Level 3, "Defined," concentrates on publishing those procedures organization-wide, setting up a panel or group to review and publish new procedures, and in general on providing an organizational focus to process improvement. Level 4, "Managed," focuses on taking metrics at each step along the way of a software project to gather data for improvement. Level 5, "Optimized," concentrates on feeding back the metrics to improve the process. The steps are "foundational" -- each is a building block for the next, and
an organization cannot skip a level. According to the SEI, most software organizations in the United States are at the initial level; only a handful are at Level 5. An organization can be certified at Level 2 or higher only by undergoing a detailed software process audit, conducted by an independent SEI-certified auditor. The time it takes to move from one CMM level to another is measured in years, not weeks or months, since the institution of procedures involves the difficult task of changing the way a group of people do their jobs. The process of improvement is arduous, but the gains in productivity, lowered costs, and quality by moving even from Level 1 to Level 2 are enormous.

**Common Features of the CMM Levels**

There are several factors critical to an effort to change, and especially to standardize the way people work in a field:

- does the organization have the ability to undertake the program of change?
- does the organization have the commitment to the change?
- is the organization actually doing what it says it will do?
- is the change being institutionalized so that it will live beyond its original architects?

The CMM addresses these areas through its definition of common features in each key practice area. The common features are: commitment to perform, ability to perform, performance itself, measurement of performance, and verification of performance. Commitment to perform addresses management's willingness to get behind the process improvement effort -- if change is not desired from the top down, it will probably not be successful. Ability to perform addresses the organization's willingness to provide funding, training, and time for people to work on improving the software development process. Measurement speaks to the need to track and record the success and failure of the new procedures in each KPA. Verification comes full circle, and asks whether the organization's management is verifying that the software procedures and practices are indeed being followed. Each key practice area in each of the four levels 2-5 has the same common features, and the specific procedures, standards, and practices built for an organization at each CMM level are built around the common features. This standard framework for a procedure helps to insure that the organization can not only undertake but succeed at institutionalizing change.

**The Relationship of CMM to Other Efforts**

The CMM offers an organization guidance on what needs to be done to improve its software process -- it does not dictate how. Thus, CMM does not define the proper software or management tools and techniques, does not tell an organization what methodology is the best one to use, and does not set specific standards. Furthermore, CMM is not the same thing as quality assurance (QA). In fact, as discussed above, one of the key process areas for CMM Level 2 is QA; the CMM
documentation says little about what constitutes QA, only that it should be done.

Because CMM does not fill in the details of what a software organization should do to improve itself, it does not interfere with other efforts such as total quality management (TQM) programs, methodology selection, tool selection, standard setting, or training. Rather, CMM provides a means of integrating all these efforts and of assessing their success against an objective scale of overall process maturity.

Case Study: Implementing CMM Level 2

This paper began by posing the problem of a “crisis” in software development, and made the claim that organizations involved in SAS software development have special problems controlling and improving their software processes because SAS has not usually been perceived as a professional programmer’s tool but rather as an end user analysis tool. Thus, SAS developers were not usually subject to the rigors, for example, of a production, COBOL-oriented, MIS shop. The following case study discusses CSC’s attempt to bring a combination SAS, PowerBuilder*, and Paradox* development environment from Level 1 to Level 2, beginning in 1994.

The Software Support Project

One of CSC’s longtime clients is a major insurance carrier with a major office in Washington, DC. The Information Systems department for the office support approximately 200 people with financial and decision support applications that run on both mainframe and the Novell LAN/Windows platform. There are approximately 10 Information Systems professional staff augmented by approximately a dozen CSC staff. In the past two years, the pace of application development has increased in response to the overall growth of the business, health care reform, and new technologies.

The arrival of powerful desktop tools such as SAS for Windows, and Borland’s Paradox was immediately followed by an IS decision to begin client/server development using a Microsoft SQL Server/PowerBuilder platform. The end result of this increased demand and rapid technological change has been a heightened awareness within the IS organization and among the CSC staff that our development process had several flaws: it was too dependent on the talents and skills of specific individuals; results were not always predictable; and costs could not be properly justified, especially if an estimate was overrun. Both CSC and the client realized that these problems would only get worse as client/server development came on-line.

There were specific financial considerations as well. On the company’s side, the IS budget was going up very rapidly, due in large part to increased demand, and IS customers in user departments were asking whether they were getting good value for their money. On the CSC side, we knew that we were about to undergo a contract recompetition at the end of 1994, and that our ability to provide solid
project estimates, good value, and a consistent software product would be a major factor in keeping our client's business.

**CSC's CMM Program**

CSC's project management team at the client site began to consider a formal TQM-type program in January, 1994. We were somewhat new to CSC at the time, having just been acquired as part of CSC's purchase of Atlantic Research Corporation's (ARC) Professional Services Group. Not being sure where to start, we contacted CSC Systems Engineering Division's (SED) Director of Quality for assistance. As it turned out, SED had begun a division-wide CMM-based process improvement program in 1991, and had recently completed a massive effort to move a major defense contract staff from Level 1 to Level 2. The defense contract, based in Illinois, was to become a model for the rest of the division's software process improvement efforts. A division-wide Software Process Improvement Plan was being drafted at the time we contacted SED for help, and SED had set itself the goal of reaching Level 2 for its software projects by March, 1995, and Level 3 by March, 1997. Given SED's tremendous size (approximately $350 million annually), this was and is an ambitious goal. However, it meant that most of the mechanisms our contract and client staff would need to make the jump to Level 2 were in place already.

We were left with two problems. The first was to convince our client's IS management that a CMM-based software process improvement program was the best way to go; the second was to arrange a satisfactory cost sharing arrangement. A three-hour seminar taught by SED's Quality Director quickly convinced our client that substantial benefits would result from an process improvement program. Our long relationship with the client made establishing a partnership in which both client and contractor participated an obvious idea. Costs were a more difficult matter: SED's estimate, based on CSC and SEI experience, was that somewhere between 5 and 10% of a total IS budget would be required to maintain a process improvement program. Given the size of the annual IS budget that meant either substantial costs to our client, substantial, unbillable labor for CSC, or both. The time commitment required by CMM, also between 5% and 10%, would also mean more pressure on deliverables. Both parties felt that the effort was worth the cost, however, and after some negotiation, CSC and the client agreed to share the expense of getting to Level 2, with each side picking up 50%.

**Planning**

Planning for the process improvement project began in March of 1994. CMM Level 2, Repeatable, focuses primarily on the management aspects of the software development process. The key practice areas for Level 2 are Requirements Management, Project Planning, Project Tracking and Oversight, and Configuration Management. At our Level 1 initial stage, these were terms we used only in the most informal sense, and we really had no more than a vague idea just how it was that we developed software. Our first step was to set a
goal: to reach Level 2 by the end of 1994, and to be ready for an independent assessment by the end of the first quarter of 1995. Most sites are cautioned that it will take at least a year to move from Level 1 to Level 2; we felt, however, that because of our small size and the fact that we had substantial SED experience to draw upon, that we could accelerate the timetable somewhat.

Our next step was to lay out the project. CSC appointed one of our task leaders as QA Manager, and sent him to special training at our Falls Church, Virginia headquarters to learn more about the CMM, how to conduct a project, and to learn special skills such as group facilitation that would be needed during the course of the project. From that time on, the QA Manager reported to the Director of Quality rather than solely to the on-site project manager so that CSC would have a direct view into how well the SPI project was going. The client manager of software development took on the equivalent role for the client. These two individuals and the CSC project manager began to lay out the schedule of activities that would, we hoped, get us to Level 2 by the end of the year. The schedule outline was as follows:

- form a group to review the progress of procedure development, known as a Software Engineering Process Group (SEPG)
- gain staff-wide familiarity with the CMM and especially with the Level 2 key practice areas
- develop a model, consisting of a flow diagram of how software is developed at our site
- form teams to write procedures describing how we will perform each of the four KPAs
- write and review the procedures
- Implement the procedures

With that outline in hand, and a great deal of previous work by CSC to guide us, we officially began the SPI project in April, 1994.

Pitfalls and Successes

Our first step was to form our Software Process Engineering Group (SEPG). Though this is technically a Level 3 activity, we felt that we needed a forum that included more than just managers to help determine the course of the project. Because of our small size, we simply declared that every professional staff member, from data technician on up, would be a member. The SEPG would meet as a whole only to review completed procedures or to decide on major courses of action; smaller teams would do the actual writing and research.

Our next step, developing a model of how we "do software," was easier said than done. One of the basic tenets of the CMM approach is that everyone has a set of procedures and steps for developing software, even if (1) they are not written down and (2) are not always followed perfectly. In fact, we were told we would be surprised at how much of a procedure we really did have. We found that to be true, as Figure 1 shows. As with many activities that are done intuitively, however, the many informal discussions, meetings, and decisions that
we used to develop an application were very hard to write down, purely because of the many exceptions to the rule. It took us a couple of months to write down and agree upon the model itself.

Our next job was to begin to write procedures to describe how we would do the things in our software development model corresponding to the Level 2 KPAs. The Level 2 KPAs are actually the long horizontal boxes at the top and bottom of the model. The main set of software development activities such as code and test, in the center of the diagram, is dealt with in subsequent levels of the CMM, once the management aspects are under control. At this point we hit our next "snag" -- a very human one. Do to our small staff and relatively informal atmosphere between IS department and the contract staff, we had a rather "anti-bureaucratic" attitude. The guidance we received from CSC was based on a very large (500 person military project, and is such as absolutely full of acronyms, boards, authorities, reviews, and other mechanisms and bodies. Both client and CSC staff reacted extremely negatively to this, and the whole idea of the SPI was thrown into jeopardy as staff began to associate it with excess bureaucracy instead of improved procedures. It took several meetings and much discussion to move on. The effort was further hampered because the first procedure we needed to write was called SP-100: a procedure for writing procedures. Convincing staff of the need to do this was very hard; finally, our QA Manager drafted SP-100 by himself and asked the SEPG only for review and comment.

It was not until fall of 1994 that we began to prepare to write procedures: the requirements of day-to-day deliverables and other pressures kept us from moving faster. Among the major problems was project participation. CSC staff, used to accounting for billable time hour-by-hour, had difficulty with work that was for the general good as opposed to work requested by a specific client. On the client side, the IS staff began to feel not only that SPI was low-priority in a busy schedule, but also that the SPI effort was primarily a CSC activity, and some withdrew from active participation altogether. Funding also played a part: though both CSC and client upper management had agreed to the SPI project, they were never comfortable with the costs as they showed up from month to month. The implicit pressure on staff, then, was to keep SPI activity to a minimum in favor of "real work". This pressure affected another of our project steps, and that was training the staff on CMM and Level 2 terminology. Only our QA Manager received specific training; others were required to read the SPI manuals and help files provided on whatever time they could squeeze in between other work. The time and budget pressure discussed above, plus the admittedly dry nature of the material, kept staff from being over-diligent in their reading, and this in turn made procedure writing and review more difficult.

Where the Project Stands Today

As of the time of this writing, we are just beginning to implement the new procedures on a few test projects. Our schedule, which called for us to be ready for at least a self-assessment of our Level
2 readiness by March 1995, has been pushed back by our various pitfalls and distractions, to June. In spite of the problems we encountered, however, by February 1995 we had written drafts of procedures for all the Level 2 KPAs, plus a full suite of management procedures for writing (and changing or waiving) procedures. We also developed a set of LAN-based application and GUI design standards that emerged as a side project from the SPI work, and we began a serious dialog with the end user departments that focused on who should be allowed to develop "applications" and under what circumstances -- a topic especially important to those end users who were SAS or Paradox programmers and who felt they needed free access to those tools in their jobs. Our greatest SPI successes were in the Tracking and Oversight and Project Planning KPAs, partly because of the specific personnel we had on those teams and partly because those were areas in which we were relatively strong to begin with. The Configuration Management procedure (Appendix 1) received a strong boost when it became tied to the so-called "LAN Application Standards," and by February was in excellent shape, leaving only Requirements Management, our most difficult area, needing continued work. Both CSC and client management are satisfied with our progress to date, and we are funded (under the same "50/50" arrangement as in 1994) to continue our work.

Quantitative benefits to our project have focused on the Level 2 problems of process management that all sites face. In comparison to a number of other CSC and client IS organizations, we appear to be running a more controlled software development process, which has great advantages as we move further into client/server and distributed development.

Conclusion

There is no question that a crisis exists in software development, one that is shared by all levels of development, from operating systems to decision support systems. That crisis is verified by article after article in both the professional and popular press. The crisis is not one of poor tools or methodologies, however. Better tools, such as SAS, can both help and hurt quality and productivity depending on how they are used. Methodologies are only helpful when they are consistently and intelligently applied. The real heart of the crisis lies in the fact that "software engineering" is too young a field to have acquired much of the discipline that exists for other types of engineering and manufacturing enterprises. The fact that computer hardware continually outpaces software in terms of processing ability, cost, and reliability is ample enough evidence of this, even if the Denver Airport and FAA projects did not stand as vivid examples. One model of introducing engineering discipline to software development is provided by the Carnegie Mellon Software Engineering Institute's Capability Maturity Model, which focuses on the process of software development rather than the specifics. CMM, which arose out of
years of research by industry, government, and academic participants, defines 5 levels of software process capability: initial, repeatable, defined, managed, and optimized. By far the majority of us are at Level 1; only a very few sites have reached Level 5.

Embarking on a systematic program of software process improvement based on the CMM can pay large benefits to an organization in terms of software quality, lowered costs, reduced turnaround time, and more satisfied clients and customers. However, these gains are not achieved with ease. Moving even from Level 1 to Level 2 requires great effort and expense of time, money, patience, and persistence. As a case study for SAS developers, this paper presented the example of a joint Computer Sciences Corporation/client effort to move a relatively small development organization to Level 2. The process and many of the problems encountered on the project were discussed with an eye toward giving others a sense of what the process improvement road ahead could be like. Although too early to be called an unqualified success, the project has returned many qualitative benefits already, and promises to yield the quantitative ones in the future. As the SAS System and SAS developers continue to take an increasingly important place in enterprise system development, we would do well to improve our process as much as we are improving our tools.

References


Robert Hall
Jeffrey Phillips
Computer Sciences Corporation
1301 Piccard Drive
Rockville, Maryland 20850
(301) 258-5300
(301) 258-6878 (FAX)

SAS® is a registered trademark of the SAS Institute, Cary, NC.

PowerBuilder® is a registered trademark of Powersoft, Inc.

Paradox® is a registered trademark of Borland Corporation
Figure 1: The Software Development Model

1. Project Management & Tracking
2. Software Quality Assurance
3. Configuration Management

**Requirements Repository**

**Baseline Requirement (BOP)**

**Analysis/Design**

**Development**

**Integration**

**QA Facilitation (All Reviews)**

- Internal Reviews
- Minutes, Actions

- Documentation
- User Testing
- Exception Report
- User Testing
- Audit