SOFTWARE TEAM MANAGEMENT IN THE FACE OF DECLINING BUDGETS

Lynn D. Stuckey Jr. and David C. Gross
Boeing Defense & Space Group
Huntsville, Alabama

ABSTRACT

One of the results of the decreasing defense budget is that competition and value returned are not just hot topics, they are cold facts. Critics have long contended that defense programs were not managed efficiently because these programs were not driven by the laws of the market place. But even if this is only partially true, the perception of a diminished threat (and the resulting diminishing defense budget) is forcing a re-evaluation of the way we manage simulator software projects. The government wants more for less. In the software arena there are a number of advances in both technology and in philosophy that may allow us to do our jobs faster and cheaper. However, as Dr. Deming says, one obstacle to progress is the supposition that automation, gadgets, problem solving, and new machinery will transform industry.

Competition is forcing us to really address software project management. The paper presents an approach to solving the pertinent issues of software project management. How many people are really required to do the job? What kind of people should they be? Who should be the system architect and what kind of person should he be? What are the underlying productivity drivers? Which tools are cost effective for our team, and which are simply neat toys? In which direction should we drive team communication and structure? The issues involved sound so familiar as to be old-hat -- a complacency which blocks productivity enhancement. We can not afford this attitude because only the most productive organizations will survive.

ABOUT THE AUTHORS

Lynn D. Stuckey, Jr. is a software systems engineer with the Missiles & Space Division of the Boeing Defense & Space Group. He has been responsible for software design, code, test, and integration on several Boeing simulation projects. He is currently involved in research and development activities dealing with software development for weapon and threat simulators. Mr. Stuckey holds a Bachelor of Science degree in Electrical Engineering from the University of Alabama, in Huntsville and holds a Master of Systems Engineering from the University of Alabama, in Huntsville. His thesis presents a systems engineering approach to software development.

David C. Gross is a software systems engineer with the Missiles & Space Division of the Boeing Defense & Space Group. He has worked in all phases of simulation and training systems from requirements development through delivery. He is currently involved in basic research for software engineering and applied research for system simulation. Mr. Gross has a Bachelor of Science in Computer Science/Engineering from Auburn University and is working toward a Master of Operations Research at the University of Alabama at Huntsville. His thesis investigates the utility of high level languages such as C++ and Ada for simulation.

SOFTWARE TEAM MANAGEMENT IN THE FACE OF DECLINING BUDGETS

By Lynn D. Stuckey Jr. and David C. Gross
Boeing Defense & Space Group
Huntsville, Alabama

INTRODUCTION

Every engineering endeavor seeks to build a product. As these endeavors grow in size/complexity, technical management of them becomes more critical. There are significant management challenges to overcome in addition to the technical obstacles. The management challenges change as the technology on which the product depends matures (see Figure 1). We find it convenient to divide the process of technology maturation maturation into three stages. In the first stage, the technology is not widely-known or repeatable. Frequently, there is only a single method or technique which can succeed. The management task here is simply to identify the technical geniuses who bring the necessary knowledge with them. In the second stage, the technology is more widely known and there are a number of techniques that will yield some success. It is at this point that the marketplace for the product of the endeavor grows exponentially, because customers begin to identify the product as a potential solution to their problems. The management task in the second stage is simply to maximize production.

It is the third stage which presents a real challenge. At the same time that customers begin to gain a real sophistication in a product, they have slaked their immediate driving thirst for the product, which made it so desirable earlier. In the case of hardware manufacturers, this is sometimes called the problem of "installed base". Everyone *needs* my computer, but how many really *need* my upgrade? The design and delivery of simulator software has entered the third stage, through the gradual maturing of

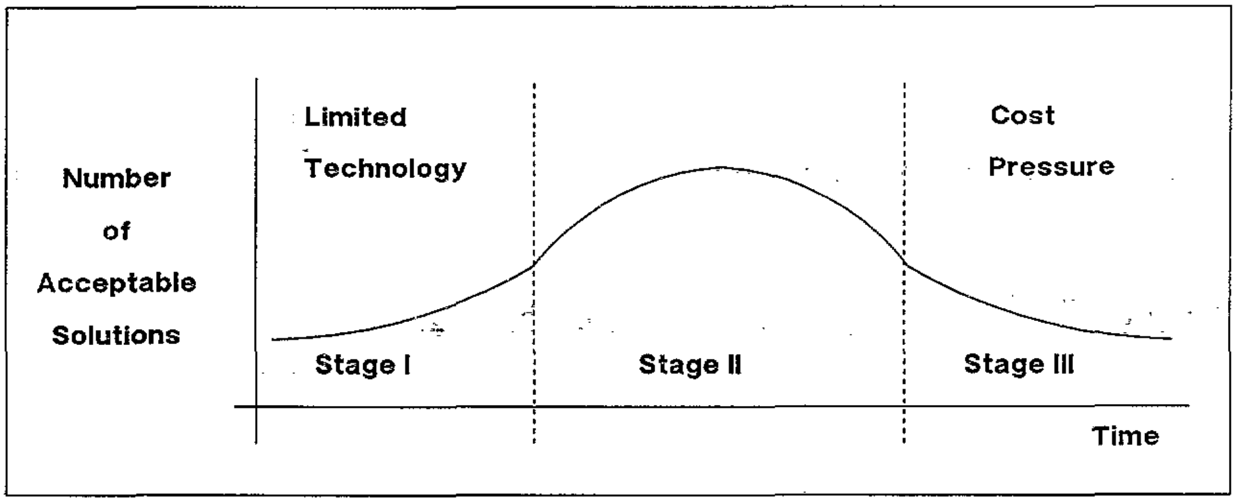

Figure 1

The Stages of Technology Management

software technologies and the pressure of declining budgets.

The third stage is characterized by the elimination of less productive processes. The customer's new position in stage three gives him a commanding posture in negotiations. His advantages mean that he is no longer required to deal with organizations that do not produce quality work -- either consuming too much or failing to satisfy. The result is that organizations employing inferior processes simply go out of business. The counterpart is also true, namely, productive organizations prosper. Productivity is classically defined as the amount (or rate) of resources consumed to produce a unit. Productivity is a measure of the *efficiency* of a process. However, productivity alone does not sufficiently reflect the commanding position that the customer holds in stage three technologies. Productivity does not measure the *effectiveness* of a method in meeting the customer's desires.

We define *quality* as the joint measure of a method's efficiency and effectiveness. High quality software will be made with a minimum of resources (manpower, computer-power, schedule, ...) and fulfill the customer's expectations. We care about quality because the only way we can stay in business is to make high quality software. But what exactly is quality software, and how do we get it?

The foundational work on improving the quality of goods arises from the work of W. Edwards Deming. Dr. Deming's central assertion is that while common wisdom suggests the workers cause poor quality, in actual fact, quality is a management problem. He calls current management practices *retroactive* management, excessively focused on the end product rather than the production process.[2] Notice that such management methods can be successful in stage two technological processes but have obvious failings in stage three. He suggests that the correct management approach is team-oriented, in which everyone shares responsibility and authority for quality.

Team-oriented strategies have important potential for improving software quality. We have seen the success of such methods on software-intensive programs such as the Ada Simulator Validation Program, the Modular Simulator Systems, and so forth. However, creating a team by declaration is not sufficient (and perhaps not necessary). The purpose of this paper is to layout how to organize and use software teams.

TEAM IMPACT ON THE PROCESS

As we stated earlier, quality is based on the process not the product. In order to define the process it must be measured. One way to objectively measure the software development process of an organization is the Software Engineering Institute's (SEI) process maturity model.[3] The central notion of the process maturity model is that all software organizations are *not* created equal -- some get better results than others! SEI contends that there are discernible characteristics that can serve as "leading indicators" of end product quality. The SEI model divides the universe of software processes into five levels as illustrated in Figure 2. According to SEI's definition of the levels, and the audits of organizations they have performed, most organizations are at level I.

There may be an argument over the details of SEI's model, however, in its broad strokes few would disagree with it. Most organizations agree generally that it is the ladder of quality improvement that must be climbed. While discussions about how to get from level IV to level V may be interesting, they are of little utility to most of us. The question for most organizations is how to bootstrap from level I to level II. And this is where team management can play an important role. One characteristic of level I organizations is their tendency to look for quick fixes and easy solutions. However, the primary problem in level I processes is *cultural,* not *technical!* Level I organizations are very individual oriented whereas level II organizations have succeeded in forming a foundation cooperation and group interest between the software engineers.

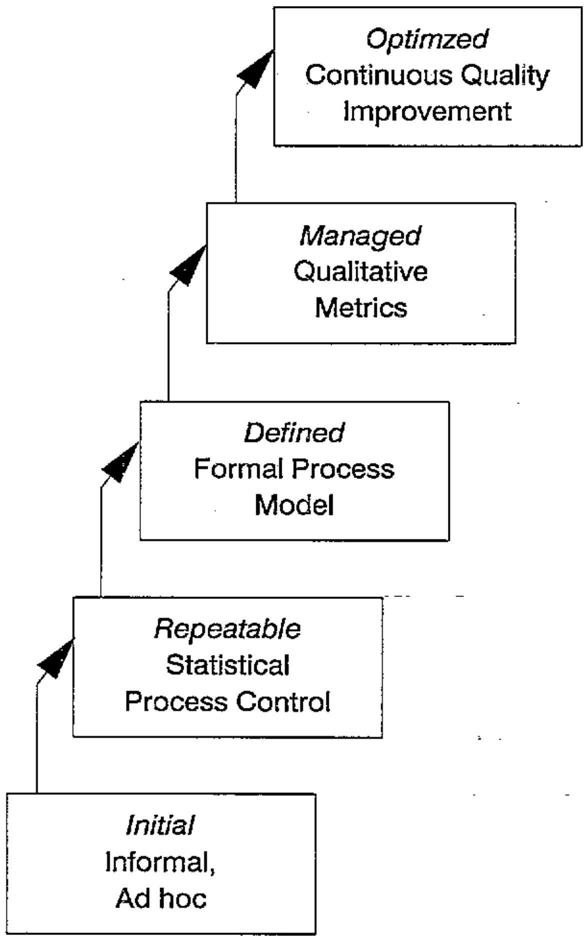

Figure 2
SEI Process Maturity Levels

Good software team management can empower individuals and organizations. But it is absolutely necessary to understand that this will require a fundamental change in the *perspective* of most software engineers! This is a most difficult challenge, after all, managing software people has been compared to herding cats. We have seen how difficult it is to introduce technical change into the software development process. One example is the resistance to the Ada programming language. However, changing an individual's perspective is a much more difficult task than simply changing a technology. It is therefore easy to predict that software organizations will drag their feet over process measurement and improvement. That is why process improvement must begin with team management. Good software team management will create a cultural that enables continuous quality improvement of the software development process. Poor team management will continue to increase the variance in software quality between individuals.

TEAM FOCUS

Obviously, simply calling a group of people a team, does not make them one! What are the critical differences that forge an organization into a team? We believe that there are three important focuses which provide the themes and motives for real teams.

The first focus is that every team member must have ***systems mindset***. At first glance, this would seem to suggest that every software engineer must in fact be a systems engineer. While this is not the case, every software designer must have knowledge and respect for the project concerns outside of his role on the software team. Quality software for large projects cannot be developed by software trolls that want to be left alone to work their magic. Training can assist in developing this focus. Courses in inter-personal communications have proved to have some effectiveness. Introductory courses in systems engineering may succeed in shaking the parochial attitude of some engineers.

The second focus for teams is on the actual ***products*** of the software design ***process*** rather than the ***deliverables*** required by the contract. Given our emphasis on meeting the customer's requirements, this seems contradictory. However, we must bear in mind that the customer really wants quality software, not useless paper. Some deliverables relate to the software design process, while some deliverables are incidental. If we treat the products of the software development process as feedback on the quality of the process, then we can use the product to improve the process. It is here we must concentrate our investment for quality. While some may argue that it is hard to distinguish between meaningful and incidental products, this is not the case. Simply decide what you would build and deliver if the contract specified no specifics!

The third focus flows directly out of the second. A fundamentally flawed approach to quality is the attempt to *inspect* it into the product, rather than *building* quality. How many organizations believe that they are building quality simply by sending their documents through a central drawing quality organization. The only thing drawing quality can give us is information -- they can not produce quality. It is possible to use such information to improve quality but this requires proactive management. Management must constantly seek the source of discovered flaws and change the production *process* to correct them.

TEAM ORGANIZATION

Simply put, the key to software quality improvement in the face of declining budgets is the software team. To be successful, management must properly organize the team according to size, membership and purpose. All the silver bullets that are introduced in the way of tools and enhancements will not change this basic fact.

Team Size

Surprisingly, to do a better job in software development, we need to reduce the size of our software development team. Software quality is *not* directly enhanced by adding engineers. Recall that quality is a function of the productivity of the team and the ability of the product to satisfy the customer. Adding people does not help because there is not a purely linear relationship between the number of people and the time to complete. For example, we cannot assume that a software project that requires one person nine months can equally well done by nine people in one month. This phenomenon is occasionally called the *mythical man-month.* Therefore, size is a critical decision in software team management.

An all too frequent approach for determining software team size goes something like this: we form multiple teams to address each aspect of a software project (usually without proper direction or clear requirements). When design and implementation problems arise, we find that software productivity is low. Since folklore holds that more people can solve the problem sooner, we bury the problem with as many software designers/programmers as we can afford. This mentality results in a loosely coupled large group of people constantly fighting "fires" in the design by starting new fires. In the end, cost and schedule overruns create a management attitude of "Just get something done. I don't care how." Many have come to the simplistic conclusion that software is always late and we cannot change it. This situation cannot continue.

On a number recent Boeing projects, the use of small teams was required due to either security requirements or limited personnel resources. When problems arose, the smaller team was forced to work out a solution amenable to all members of the team. The team communicated more frequently and did a better job of coordinating efforts. As a result, software quality rose dramatically. On one particular program an average team size of six engineers developed and fielded a flight simulator of 180,000 lines of code in just 19 months. Each member of the small software team had to function in multiple capacities, giving the team a firm systems understanding. The traditional barriers between large staff groups were eliminated. The team concentrated on improving the system as whole, with resulting improvements in the software development process. They required less effort for fire fighting and territorial skirmishes. The small team eliminated the common problems of poor communication and lack of direction. Figure 3 illustrates the communication difference on a small team versus a large team. Since each member's role was obviously crucial to success, the team (not management or corporate policy) had little tolerance for, or ignorance of, poor performance. A small team environment allows software engineers to greatly outperform a large team in software development.

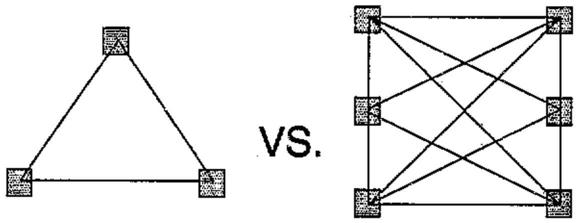

Figure 3
Inter-personnel Communication

Team Members

The use of a small team adds the responsibility of proper team membership. If the team is to be successful, the right skills and attitudes must be present. There are two important observations about team members. First are the qualifications of the general team member and second is the special role of the software systems engineer.

Software Engineers. The emphasis of team members now has to be on quality, and the members ability to handle multiple aspects of the simulation process. This is simply an extension of the present situation. Simulation engineers are already made up of many of the engineering disciplines. Electrical, mechanical, and industrial engineers, as well as physicists, mathematicians, and computer scientists are used to form the present software teams. Why? Simulation requires a broader background than just computer programming.

In addition to specific specialities, there are three *common* abilities each team member must possess. We could look at this as the individual focuses which empower the team focus discussed previously. First, the engineers need to have some systems engineering background. As software development moves from a mystic art to engineered process, systems engineering will be a major part of the transition. Secondly, the engineers need to focus on continuous quality improvement. They have to realize that the way the product will improve and be cheaper to produce is through the improvement of our software development processes. There are no "silver bullets" that will cause dramatic leaps forward without time, effort, or money. Thirdly, the engineers must be trained. On the job training simply does not necessarily result in a quality product at a quality price. Both the management and the work force will have to be educated in the new methods including team work. Simply put, the software team members on future simulations cannot afford to depend on a "one-area specialist". On a small team, members must be able to pull their load in a variety of situations.

Software Systems Engineer. A software systems engineer is a systems engineer who is proactively involved in the design and production of the software. In the beginning of a simulation program, there are a group of systems engineers that are concerned with the system as a whole. The hardware, software, logistics, etc. are all part of the initial requirements analysis. The project needs to transition the software systems engineers from this stage of the program into the active participation in the software development.

Systems engineering has long been accepted as applicable to hardware systems; but in the case of software development, it has been either ignored or taught as inappropriate. Others have stated that systems engineering only applies to the ends of a software activity; e.g., the front end to provide requirements and in the back end to integrate the system and test compliance with requirements. However, a systems engineer is really responsible for the entire picture. In software this includes all phases of software development, production, and operation. Beyond providing a person with oversight responsibility for the system, systems engineering can provide us with a methodology to guide us through all phases of the software lifecycle, and in particular through the development phase.

Because designers imagine and implement complex systems in parallel, projects can suffer from the fracturing effect of multiple individual visions of the final product. The systems engineer is the only one who has a total system viewpoint. The problem in the past has been that the systems engineering group is separate and unrelated to the development groups responsible for design, code, test, and integration. This separation has lead to poor requirements adherence and less than optimal software solutions. This is the reason that in the software world, systems engineers need to transition through a project's phases. This means that the systems engineer needs to be the requirements engineer, the designer, the tester, and the integrator. Of course, not all the systems engineers can be a part of this transition, but a portion of the systems engineers should become software systems engineers, intimately involved in each of the software development phases. This allows a *real* implementation of the systems engineering process.

THE TEAM IN ACTION

Software Engineering Environment (SEE)

Most software development managers would agree that software engineering environments impact quality. They would probably not agree about the amount and kind of impact. The marvelous products (and their even more marvelous sales pitches) seem to promise all measure of increased productivity. However, the prudent manager will recall that productivity is not sufficient for success in the software marketplace of today. By itself, productivity does not meet the needs of the customer.

What we need is software development processes which produce quality software. At issue here is the debate between *tools for quality* versus *toys for the kiddies.* A classic example of this is the personal computer (PC) invasion. The decade of the eighties saw the introduction of PCs into virtually every office, yet the productivity of the average office worker is essentially unchanged. Why? The problem is not that PCs are no good. The problem is that they were not generally employed in a disciplined way to enhance quality.

But what tools improve quality? There is a tension between the *surface* quality of a product and its underlying *technical* quality (see Figure 4). The surface quality is the degree to which the process is efficient and effective, in the eyes of the customer at product delivery. The technical quality is the same measure, in the eyes of the technologist. The customer does not really care about technical quality at the time of delivery. Neither does a designer who lacks a system mindset. What they forget is that the vast bulk of any system's cost is incurred after product delivery, during maintenance. The level of technical quality will drive the maintenance cost long after anyone can do anything about it.

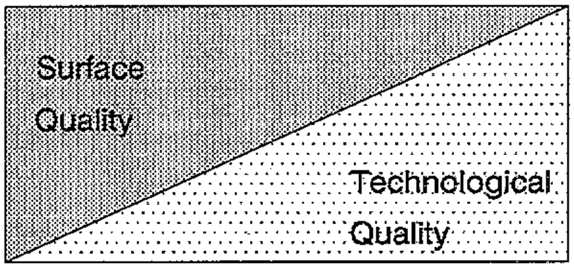

Figure 4
The Tension Between Qualities

This tension between kinds of product quality can only be resolved through teamwork. The tools which will contribute to improved quality are those which assist in building teams. One example is language selection. Some languages are well-suited for development in teams and some are not. While the latter may be well-suited to small projects, their impact on large projects, which require programming teams, is obviously negative. In Japan, Ada is the most popular general application language (not just for military projects).[1] This is because Ada well supports a team approach to software development. The impact of the Japanese emphasis on teamwork to improve processes has had an obvious impact on a variety of American industries. The software industry will be next if we do not learn how to use teams. This argument can be extended to the selection of compilers, development machines, target machines, and CASE tools.

A classic example of the impact of tools on teamwork is configuration management. A traditional attitude toward configuration management is that it exists to protect management from careless programmers. Therefore, the configuration management software is built to prohibit ready visibility or changes. However, programmers require visibility to the software and the ability to make changes if they are to correct errors and produce a product! In one project of our experience base, the configuration management software was so obtuse and slow, that the software engineers spent large amounts of project time just figuring out ways to get around it! A regular result was the loss of control of the software baseline. An obvious quality improvement would be to improve the configuration management process, simultaneously taking into account the needs of the software engineers (to make necessary changes) and management (to maintain a definable baseline).

Peer Reviews

One of the most important steps in the systems engineering process is the practice of design reviews. It is through these reviews that members of a team can affect the system as a whole and improve the software development process. *The problem is that these reviews have* become so blase´ that they have lost their purpose. The very name *design review* is indicative of the problem. Reviews are not only to check design correctness. They are just as important as avenues for sharing techniques, discussing interfaces, and checking compliance to development standards.

Peer reviews can give the software team the opportunity to fundamentally improve the software development process. The old purpose of reviews was the attempt to inspect out bad software. The new purpose is to build in quality from the beginning. If we treat the review as an effort to continuously improve the process, rather than anchoring on the errant details of a particular piece of code, each review can benefit all of the software. A secondary purpose of a review is to promote communication. Software engineers (like other people) need to feel that they can ask questions. The review should help drive out the fear that many engineers have about asking questions. Many employees are afraid to ask questions or to take a position, even when they do not understand what the job is or what is right or wrong. People will continue to do things the wrong way, or to not do them at all. The economic loss from fear is appalling. It is necessary for better quality and productivity that people feel secure.[2]

Once the need and purpose of a peer review is established there are two other questions that need to be addressed: (1) When do you have reviews? and, (2) Who's in charge of the review? In the past reviews have been forced to fit someone's schedule or timeline. Reviews should not be based on the calendar. They should be based on the development lifecycle. A software system has a lifecycle just as any other system. Reviews should be scheduled for each software object at the end of each phase. One very important point must be made about any development lifecycle, namely there are no crisp edges between phases in a lifecycle. The actual lifecycle is not linear and there is continual feedback. This means that some software objects may be in *review for design while another is being reviewed* at integration. Reviews need to be repeated if iteration is required in upper levels of the software hierarchy. The idea behind scheduling reviews based of lifecycle phasing is that it provides meaningful steps and information to be reviewed without constantly interfering with the engineer's progress.

The second question of who is in charge of a review may seem simple, but it is not. What has been conspicuously absent from this discussion on reviews is the role of management. Reviews are not primarily focused for managers. This means that management may be in a review but really does not need to preside over it. A lead *software engineer or perhaps the software systems* engineer needs to run the review. The decision is based on who has the authority to request changes in the software after the review. But, whoever is in charge must lead in the review process. The job of the leader is not to tell people what to do or punish them but to lead.

Documentation

Documentation must be a part of any software team management approach because it presently requires a large level of wasted effort. Software documentation is one of the most expensive and least useful parts of a simulation. Let's face it, *software documentation was classically* required as a huggable surrogate for the etheral software that really was just bits on a computer. The customer no longer has the luxury of requesting the volumes of useless paper. We need to review the true need for documentation and only require/provide what enhances the simulation. We need to scope software documentation to those items that are required for the software engineer to develop and maintain the software. In essence, we need to develop documentation that we would write and use even if it were not required as a deliverable. This is the type of documentation that is a useful tool to the software team and not a hindrance.

The true purposes of documentation are for an aid to the transition of requirements into code and for software maintenance. In the days of assembly language, a person needed something in English to help him describe the design and layout of a software system. Today, we have software languages that are capable of being written in an understandable manner so that the explanation is not really required. Today's requirement for documentation must be that only the paper required should be considered. The government made a first step toward this when they implemented MIL-STD-2167A. This standard allowed the developer to customize his documentation, when the customer agreed.

The main tool in making self-documenting code is *to make the code understandable. Understandability* as a software goal is one of the easiest to implement, but is most often ignored. Understandability is the map of the transformation from problem to solution. It is also the measurement of the software's map to the real-world. This map refers to the ability of the software to accurately represent a real-world counterpart. For example, if the software is modeling a car, one would expect to find components and interfaces that are common for a car explicitly called out in the model. As the complexity of software systems increases so does the need for truly understandable software. The need for understandable software has been ignored because software has been written from the view point of the designer and not the reader. However, software is written once and read a thousand times. The reader's interpretation of the software is a good measure of understandability. The software ought to serve as a diagram of the requirements, design and implementation. An electrical engineer understands a system by looking at the circuit layout; the same should be true with the software engineer and his software. Understandability can also be a basis for the earlier goal of maintainability. Software that is understandable is much easier to maintain.

Understandable software is the basis for credibility. Credibility was the purpose of all the mounds of documentation in the past. We can gain more credibility, with users and maintainers, if the software documentation derives from the code itself. There are two approaches for this: (1) self-documenting code (see above) and (2) automatic documentation generators. Automatic documentation generators *give us the ability* to update the documents by feeding the current code to them. In addition, we can gain a synergy between projects, because the documentation begins to have a uniformity that cannot be produced by humans.

Another aspect of self-documenting code is that realization that a software methodology has to take this requirement into account. A prime example is *interface scheme*. A methodology that espouses a common data area or an extreme message passing scheme will result in code that does not provide interface definition to the reader. A methodology that requires a parameter list type of interface has the interface definition laid out for the user. An Interface Definition Document can be constructed automatically from the code with just the parameter lists and a corresponding types package. This is an example of planned self-documenting code.

CONCLUSION

Given the emphasis on software quality in order to survive the current marketplace, management must develop softwares teams which have the appropriate focus, organization, and activities. The crux of our position is this: teams are the *primary management tool for improving software* quality. A real software is a small cohesive group made up of multi-talented engineers with a systems mindset. Their focus must transcend the tools that the engineers use in software development. The peer reviews, and documentation must be primarily slanted toward helping the team do a better job. In this world of ever evolving technology, it is not the computers, languages, or methods that will make a simulation effort a success. It is the people on the software team and their focus that will be the key.

REFERENCES

[1] Richard Riehle, "Ada in Japan", *Embedded Systems Programming*, August 1991, pp. 28-33.

[2] Mary Walton, *The Deming Management Method*, Perigee Books, Putnam Publishing, New York, 1986.

[3] Edward Yourdon, *Decline & Fall of the American Programmer*, Yourdon Press, Englewood Cliffs, New Jersey, 1992.