Software in the Large

Larry Bernstein

Large software projects—those that require more than 100 people to develop—are difficult to manage. They usually take more than one year to complete and only one in ten finish on time, within budget, and with the features users need. It is not the people, but how they are deployed, that is the critical issue in managing a large software project. One strategy is to partition the project into a collection of smaller ones, provide the technology and organizational structures to tie these parts together, employ common tools and processes, and schedule formal partial product delivery dates within the project. Management also must maintain a humanistic point of view to keep the project workers focused on their goals, as these workers typically are affiliated with different business units, work in different locations, and have different responsibilities.

Introduction

"The control of a large force is the same principle as the control of a small one. It is merely a question of dividing up their numbers and instituting signs and signals," wrote Sun Tzu in *The Art of War*. If the general was right, then the trick is to find a set of signs and signals that are needed to control a large software project.

For such projects, defined as those lasting a year or more and employing more than 100 people, maintaining the balance between too much control and not enough is one that requires constant monitoring and adjustment throughout the project's life.

This paper, reflecting my 30 years of experience as a software manager, discusses the role of the project manager, project development and control plans, project risks, and software design, manufacturing, and integration.

Why Is Software So Hard To Do?

Software technology has a weak theoretical foundation. Most of the theory that does exist focuses on the static behavior of the software—namely, analysis of the source code. There is little theory on its dynamic behavior—namely, how it performs under load. So we naturally try to avoid serious problems by over-engineering software systems, since we cannot realistically specify the expected load as there is no idea of the resources most applications will need until they are up and working.

Today's software theory cannot provide the design rules to assure scalability, robustness, and reliability. In their article on scaleable software libraries, Don Batory and his colleagues[1] argue that a large, feature-rich collection of software components is inherently unscalable. To go further, we cannot be certain that a small change in a software system will result in an equivalently small change in the system's performance. We either have to test and retest every time we make even the smallest change or we risk system crashes and the resultant cranky customers.

Indeed, experience has shown that small changes in the software or in the input data can lead to a predictable outcome—the system will be unstable and will experience

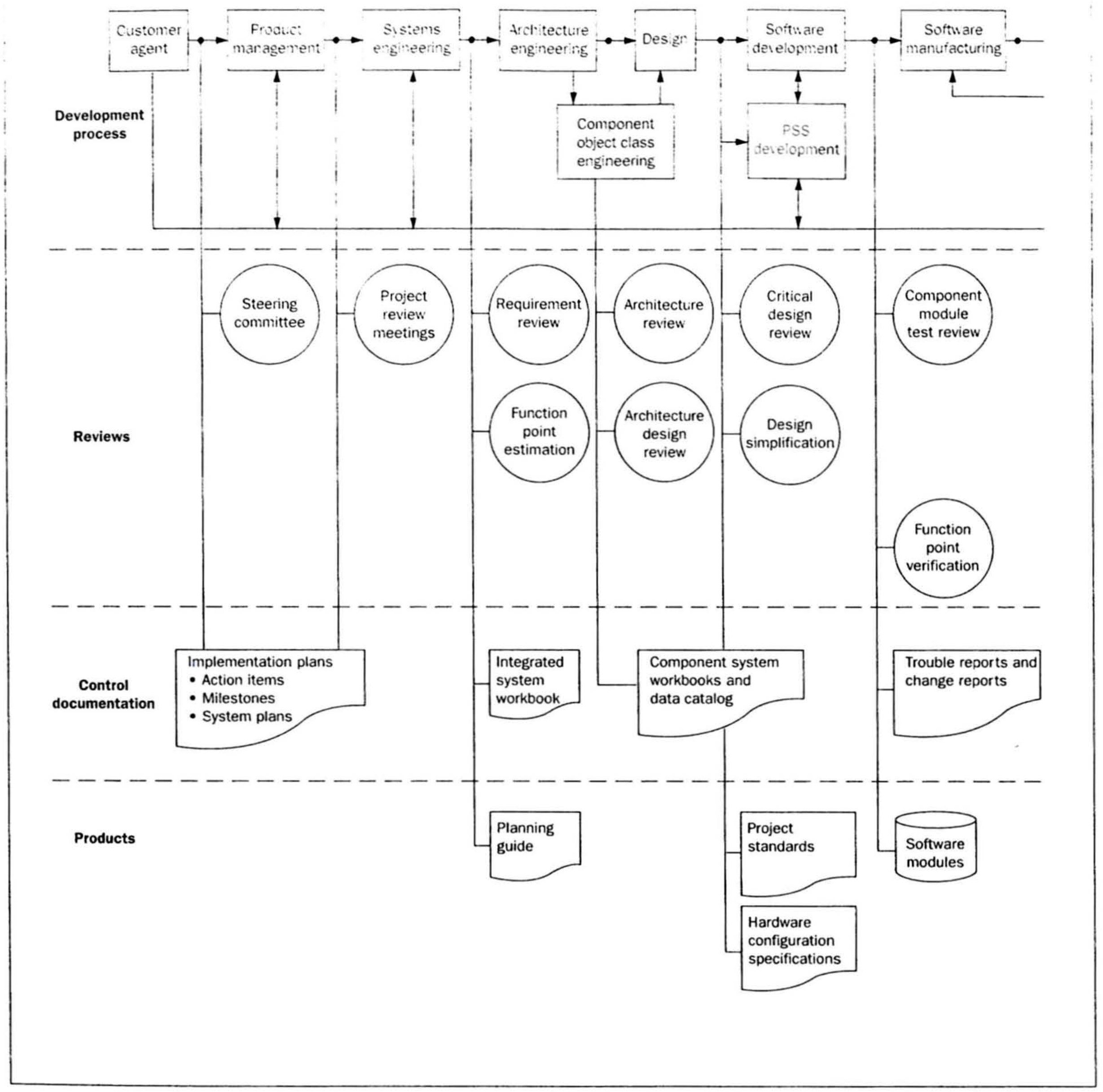
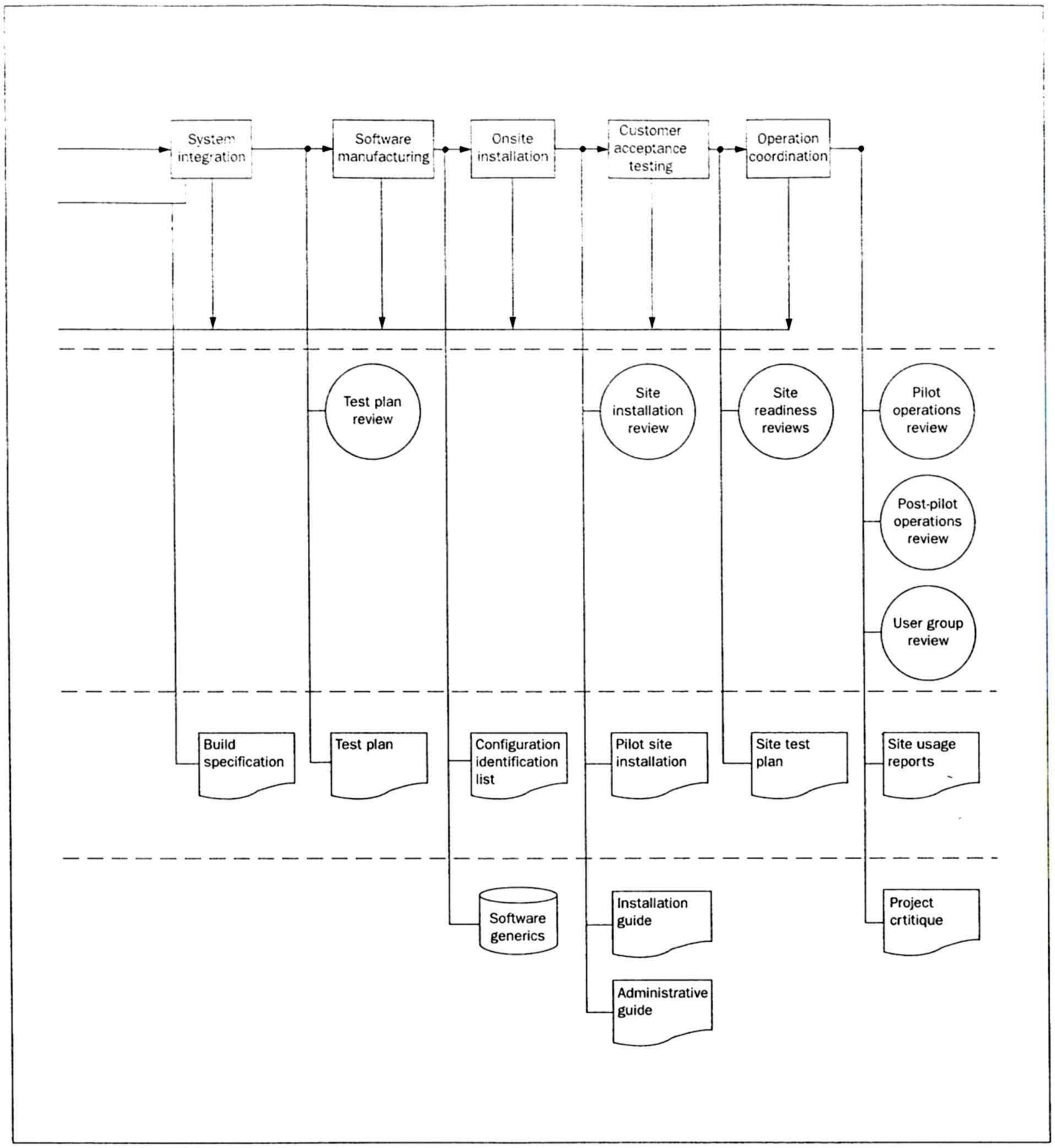


Figure 1. This example of a generic project control plan itemizes the concrete *signs* that will indicate to everybody the health and condition of the project. Dealing with processes, it outlines the relationships that link the major project functions, the development process, project reviews, control documentation, and the end products from each functional area.


System integration
Software manufacturing
Onsite installation
Customer acceptance testing
Operation coordination
Test plan review
Site installation review
Site readiness reviews
Pilot operations review
Post-pilot operations review
User group review
Build specification
Test plan
Configuration identification list
Pilot site installation
Site test plan
Site usage reports
Software generics
Installation guide
Project crtitique
Administrative guide

degradation in system performance that is remarkably similar to the workings of chaos theory.[2]

Yet even when we get the system to finally work after these changes, can we be assured it will continue to work? As its load grows, will the system respond properly to out-of-range data? Will it handle data that arrives when it shouldn't? And when the load reaches system thresholds, will its performance degrade predictably? These are several of the 18 items Robyn Lutz[3] proposes in a safety checklist that all project developers should use to measure any changes to their designs. But for those still not convinced that the problem is serious, consider the article in *Forbes*[4] magazine fingering a three-line change to a 2-million line program that caused multiple catastrophic failures.

The Project Manager

The first challenge to the successful completion of a large software project is to find an experienced project manager. Since successful managers of large projects are rare, the alternative is to look for someone who has a proven track record on three or more medium-sized projects involving 20 to100 people and lasting 9 to 24 months.

As illogical as it may seem, one frequent failure at this initial stage is that management neglects to appoint a project manager, or appoints someone who has only been successful in leading small teams.[5] But small-team experience does not scale up to large projects. In addition to hands-on experience, the potential project manager should have been formally trained in project management, for there is much to be said for learning from the mistakes of others.

To ensure that the project manager has the administrative leverage to do the job effectively, he or she should have control of all project resources—including its funding, for the person in charge of funding has enormous influence on the behavior and direction of others.

Many think that the software manager's life is as orderly and as well structured as the software produced. Would it were so easy. A software project cannot be managed from afar with a prescriptive set of processes—it is a hands-on job that will have more than its share of ups and downs, even in the hands of the most skilled project manager.

Three keys to successful project management are visible leadership, defining the problem, and insisting on decisions based on facts. The leader sets the vision and goals for the project, which must remain crisp and understood throughout the project. The concept of a clear project vision should not be taken lightly, since it has been proven time and again that people are more motivated and perform better when they are making what they regard as a meaningful contribution, and not just improving the bottom line.

The goals, though less lofty in tone, are just as critical. When the development organization aligns its activities with the business goals of its customers, the developers can understand better how their particular effort contributes to the customer's expectations, and the organization's decisions can be more confidently and effectively delegated throughout the organization. Examples of such goals are "Eighty percent of all assignments untouched by human hands," "Terminal switching between hosts with no changes in the mainframe," "Four times the performance of the existing system with no loss of functionality," and "No traffic congestion on Mother's Day."

Project management demands attention to detail, a sense of humor when things go awry, flexibility when the customers find out they do not know what they really need, and most of all a spirit of optimism when things seem most bleak. People working on the project need to believe that their leader has a very clear idea of where they are headed, appreciates the problems they face, and has a plan—solidly supported by data—for any contingency.

Since problems inevitably will occur,[6] the mantra of "Fix the problem, then fix the process, but don't fix the blame," establishes an environment of trust among project members, and yet indicates a willingness to take risks. The ability to see a way around seemingly intractable problems is the hallmark of a great project manager. When this is not possible, however, the ethical manager will stop the project.

And this leads to one caveat which must be borne in mind by the project manager's own upper management, particularly when, in their frustration, they want to blame software managers when projects are late, over cost, or worse yet, canceled. In the best of hands, large-scale software development is very tricky, and thus very risky, and executives must bear this business risk in mind when such problems occur.

Developing and maintaining the confidence of the customer also is the project manager's responsibility. The project manager needs to make on-site customer visits to see, first hand, how the product will be used and to ensure that the customer's concerns are being accurately interpreted through the project's goals.

Thus, we can regard the project manager as a corporate fire fighter, harvesting the proper resources for the project, backfilling when there is a problem, making or changing decisions based on carefully gathered information, and representing the project to the customer and its own corporate hierarchy.

Knowing that software development is an intensely human effort, software people are skeptical that manufacturing techniques proven on the hardware side of the house will lead to best-in-class development when applied to software.[7] The answer is that we need formal approaches that allow for dynamic changes in the product. The best approach is to:

- Make the software work,
- Make it work right, then
- Make it work better.

The spiral[8] development process provides a formal scheme that encourages incremental design and risk management, while keeping the entire project under control. The spiral process provides for several iterations of prototyping and an assessment of project risk before a commitment is made to a formal development plan. This approach must be coupled with formal baselined specifications, and contingency plans must be thought through and nimbly acted upon when the unanticipated happens.

Another mistake often made at the beginning of a project is bringing on too many people before they are needed. In this premature "staffing up" scenario, as many people as might eventually be needed at the project's peak activity are hired at the very beginning.

This is not the way to go. During the definition phase, long before the project functions and requirements are clear, and before General Tzu's "signs" and "signals" are needed to control a large staff, only a small team is needed. This team should have a free hand to define the project development plan, architecture, tools, and technology that will be used in building the system. At this point, loose controls, such as status reports and project meetings, are more appropriate. While this definition phase is under way, the rest of the staff can be recruited and trained as needed for the task ahead.

To summarize, project managers set the goals for their organizations and decide what must be done to achieve them, structure the work into manageable activities, foster teamwork, communicate constantly, help people develop themselves, and reach out frequently to customers. The project manager needs to encourage the team and seek opportunities to cite individual and group accomplishments, for while software people typically find their jobs meaningful, they sometimes are unsure of the importance of their results.

Project Development and Control

A project development plan is the *signal* system that keeps the project focused and headed in the right direction. It is a living document, summarizing the proposed course of action as well as containing the detailed supporting plans for all the components. The development plan specifies at the outset what work will be accomplished, though particular features and activities may still be modified throughout the project's life. The plan states who will do the work, how the staff will be organized, and what the interfaces will be.

The development plan is very important, because the architecture of a system often follows the architecture of the organization building it. The plan includes lists of the resources that will be required, and estimates of how long the project will take in increments, in phases, and by milestones. Finally it sets out the strategies, technologies, and methodologies that will be used, with special emphasis on quality and robustness of design.

Figure 1 shows a generic project control plan. It itemizes the concrete *signs* that will indicate to everybody the health and condition of the project. It outlines the relationships that link the major project functions, the development process, project reviews, control documentation, and the end products from each functional area. It deals with processes.

Project direction is managed through periodic project meetings and customer steering boards. Quality is maintained by architecture reviews that must be held on the work before it passes through predefined quality gates. These reviews ensure that the project is feasible and that the approach is sound.

The Project Meeting

The project meeting is a key function throughout the life of the project. It should start out being held weekly and, once the definition stage is complete, it can be held biweekly. Less frequent meetings tend to allow the project to get out of control; more frequent meetings waste time.

The project meeting sets the culture for the project. It is a regularly scheduled ritual held on the same day, in the same place, with the same fixed agenda, and for the same duration. Every module manger is expected to attend, along with the architect, integrator, and support staff. The project manager chairs the meeting, but it is best to have an assistant prepare the material for the meeting and quickly write the meeting minutes for prompt distribution throughout the project.

Think of the farmers market as a metaphor for the project meeting. Here people come at fixed times not only to exchange goods but also to get the news and to gossip. The market creates a culture of free information exchange and a democratic spirit of information sharing. People can rely on others they need to talk to being there, and they avoid the awful problems of telephone tag and cascading ad hoc meetings to solve specific problems.

The Developers' Culture

A procedure manual is needed to define the relations between the various groups. This process works best when it becomes ingrained in the minds of the people and acculturated within the organization. Once after an outside auditor visited a project of mine, he commented that it was strange that everybody followed the same procedures, although there seemed to be no formal procedures to follow.

He asked, "How did everybody know what to do?" What wasn't apparent to him was that the procedure manual was written and widely distributed three years before his visit and the procedure flow was contained in a flow chart, supported by associated control documents and regularly scheduled meetings. These disciplines were followed routinely and passed on from person to person and group to group because that was just the way things were done. The manual itself had long since been forgotten; even the managers had forgotten it existed.

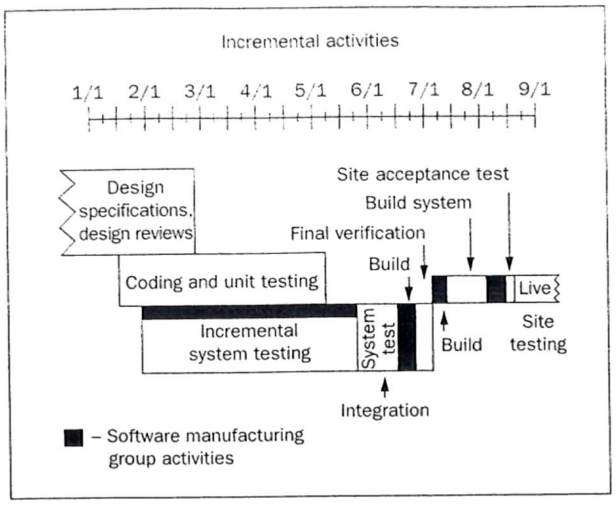


Figure 2. The key for a procedure manual to be a successful guide is not to have it too detailed, but it should clearly define handoffs from group to group at key points in the process. This figure shows where these handoffs occur while incremental development is carried out.

The development procedures had became a way of life.

The key for such a manual to be a successful guide is not to have it too detailed but to have it clearly define handoffs from group to group at key points in the process. Figure 2 shows how these handoffs typically occur while incremental development is carried out.

These handoff points are determined and measured by the project manager, while each person or group on the chain defines a set of acceptance criteria for a handoff so that if the delivery organization does not live up to the quality standards, the receiving organization has the *right of refusal*. This is especially important for the test teams, who often are erroneously expected to test in the quality, which is three times more costly to do than designing quality in. Of course, if problems are found during testing that have to be fixed in the field, the cost is 30 times more than ensuring the quality in design.

Software Design

Customers focus on price, features, and schedule before they get the product. Once they have it up and running, their focus turns to availability, throughput, and response time. The successful project manager must balance all six factors during the design phase, insisting on a formal design simplification effort. The goal is to reduce the design complexity by 40 percent during this design phase, which starts once the requirements are baselined and ends with the product implementation.

Using the spiral method in this phase is very useful and fully 30 percent of the project schedule should be allotted to design. With good design, the total development schedule can be reduced by 40 percent as redundant features are eliminated, opportunities for software reuse are found, and the algorithms implemented are simplified and made directly applicable to the problem. Gold-plating also is eliminated.

The interfaces between modules are critical to get right and the design phase gives the developers the opportunity to try different approaches without committing to a formal implementation. If the approach proves too expensive in execution time, then it can be changed without jeopardizing the project schedule. This is the spirit of "failing small and succeeding big."

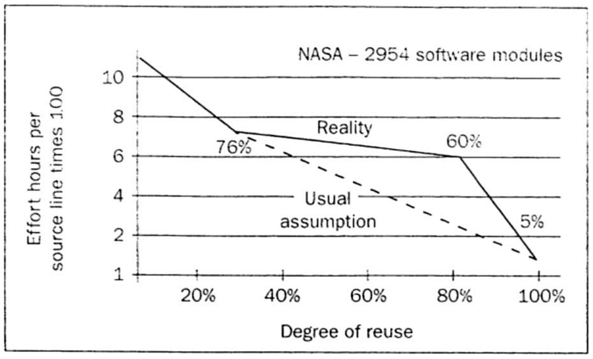


Figure 3. Data on the reuse of 2,954 modules of NASA programs[7] clearly show that to reap the benefits of the initial extra effort to make a module reusable, it must be reused unchanged. A slight change can increase the module cost by 60 percent, making it almost as expensive as a major revision of 70 percent of the software.

Managing the Risks

Since people are the critical resource in developing a large software project, gathering the large numbers required with the right skills and ability and organizing them into functioning units is a major risk.[9] People on large projects often feel lost and that their contributions are unimportant and unappreciated. Thus it is very important to communicate the status of successes and failures to all members of the project team.

This communication is best done by the project manager on a regular basis. One effective technique is writing a weekly status report to one's boss and distributing it to all the managers on the team, coupled with making the minutes of the project meetings promptly available to all.

When problems begin to develop, and schedule slippage becomes a growing possibility, there are signs that one should look for. For example, when the problems being worked are six to nine months ahead, the project is under control. When, however, the team is anticipating problems just one to three months into the future, it has things under control but it still can easily fall into crises. When problems are not openly identified and discussed because of the fear of being blamed, the manager can be certain the project is in deep trouble. And when the project is reacting to problems that occurred in the past and it is constantly trying to dig out, the project is in crisis. Thus, the concern is not the *number* of problems that the project is addressing, it is their *nature* that determines the project's state of health.[6]

A neat approach to dealing with problems is to use the concepts of *current estimates* and *negative slack*. The project manager owns the project schedule and all commitments, but the development managers are encouraged to provide an honest current estimate against their piece of the project. They express their difficulty in meeting the dates in terms of negative slack and not in terms of new dates. This isn't a semantic trick, it is a "don't-fix-the-blame" approach that recognizes unanticipated problems will develop and will require a reassessment of resources. This approach naturally leads to questions of what is on the critical path and what barriers are getting in the way of meeting the plan. The project manager can then choose to add resources, rearrange the sequence of activities, or redefine detailed goals to close the gaps—without the word spreading that there is a schedule slip.

When the project seems to be losing ground it is essential to define the problems carefully and then to get the facts to fill the gaps. Management by fact is easy to suggest but hard to accomplish as people clamor for a decision, any decision, just to move on. Knowing when to stop to get the facts is a skill managers of large projects develop. Stopping too often can lead to "paralysis by analysis," but not stopping enough often leads to huge investments in solving the wrong problem.

The experienced project manager knows to ask "What's changed?" when something that worked before no longer works.

Software Reuse

An important part of the design phase is to find opportunities to reuse software, although reuse becomes profitable only after its third use. Data shown in Figure 3 on the reuse of 2,954 modules of NASA programs[10] clearly leads to the shocking conclusion that to reap the benefits of the initial extra effort to make a module reusable, it must be reused unchanged. It had been assumed that incremental software changes result in incremental cost increases, as shown by the dashed line. Thus it wasn't surprising to learn that using but not changing the software increases the cost by only 5 percent. NASA's experience showed, however, that just a slight revision in the software drove the cost up by 60 percent, making it almost as expensive as a major revision that changes about 70 percent of the software.

Since software reuse is profitable only after its third use, the issues of who pays the extra cost of initially designing software for eventual reuse and who pays for ongoing support remain serious barriers to reuse. Reuse across multiple organizations can pay off, but it is a tricky effort to negotiate.

Within an organization, however, success is possible. Before the BaseWorX™ platform was available from the Software Technology Center, most products consisted of no more than 10 percent of reused software and none contained 100 percent. After the platform became available, products often were made up with half their software coming from reused modules and one was made entirely of reused modules. The BaseWorX platform is not a universal solution to the intractable problems of reuse, but it has demonstrated that such platforms make reuse more likely.

In the category of currently intractable problems, it has been impossible so far to systematically reuse software across application domains. There is ongoing work in modeling application domains to determine the relationship between requirements and object types so that software architectures can be reused by selecting these features.[11]

Additionally, reuse even in the same application domain is successful only when throughput and response time are not overriding concerns, since systems with large amounts of reused software suffer performance penalties by as much as a factor of two when compared with custom-designed systems. Finally, it is not yet possible to maintain an asset base of software modules except when they are in packaged libraries and when they are utility functions. Where reuse is successful, there has been a high level of management attention to making it work and a willingness to invest in the upfront design for reusability.

Software configuration management assumes that there is an existing base of software components from which the components of a specific system are chosen, assembled, tested, and distributed to a user. Even then, exhaustive re-testing is still required to root out undesired interactions between modules.

The message on reuse is simple and to the point: *When you reuse software, don't change it.*

Software Manufacturing

The software manufacturing process begins when the integration teams need careful configuration management to control system changes and to schedule fixes. Module developers deliver fully functional and tested modules, test scripts, `makefiles`, source code, and documentation. The manufacturing group runs the change process, maintains and updates the project libraries, and operates the development environment.

The process of building software and controlling change demands special know-how. Skills are needed in control languages, databases, configuration tools, and machine scheduling. Volume production and quality control demand an assembly-line, product-oriented frame of reference, with an emphasis on adhering to procedures and inventory control. Once release engineers establish the build and control methods, a production staff can handle the detail to perform them.

Developers do not fully realize the burden that a software manufacturing group can relieve them of. The manufacturing group can build systems, ship to sites, control changes, and port the system to new environments.

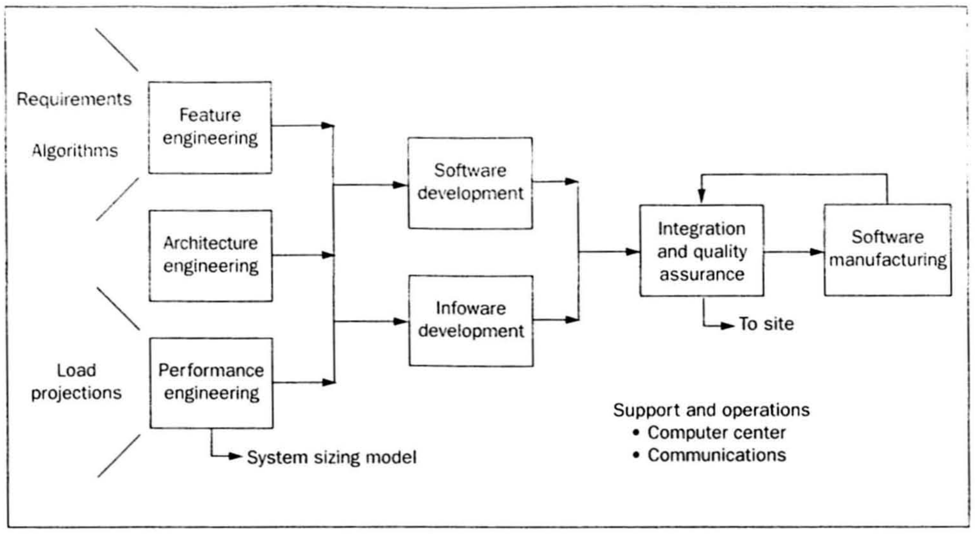


Figure 4. This organizational chart shows the relationship of the various development organizations as the product moves from initial design to software manufacturing and on to field use.

No longer do developers and their managers need to deal with generating code for a test or target machine, installing code, or even initializing a system.

Figure 4 shows where the software manufacturing group fits into the project organization. The manufacturing staff, with a controlled base, can cross-compile and objectively compare the results with the original. The software manufacturing group possesses an orderly collection of all viable software, something no one developer or development group can claim. Using this data, it can track builds for several machines simultaneously and, with initialization files clearly separated, configuring the application for a new machine becomes a much simpler job.

Software Integration

Today, testing is an art, whether it is meticulous debugging or broader-brush scenario testing. Using only the system requirements and user documentation will not expose all the problems that turn up in the field.

A method for large projects that has proven effective is *cooperative testing.* This is done among groups outside the integration group by any designer who must touch another's module. The software manufacturing group helps with the builds, but this is a low-profile, friendly effort at intermodule testing without configuration control. It fosters cooperation among module developers and eliminates problems before they see the harsh glare of the integration team's searchlight.

Debugging is not the correct focus of integration activity. No matter how many errors are removed, no one knows how many remain—nor does the customer care. The critical measure of quality is how long the system will run before it fails and what the operational impact of that failure will be.

Testing emphasis has moved from randomly and inefficiently "looking for bugs" to truly ensuring that a system performs as expected. Scenarios that model the way the system is to be used are the most effective test cases. They model the life cycle of the system, including error paths and recovery paths. Few functional problems will be found in integration and these can be fixed quickly by the module developers. When there are too many functional problems, however, it is time to stop testing and declare a *quality hold* and return the modules to their respective developers to be fixed.

Summary

Managing large software projects requires special skills and techniques. Starting small, having great customer relations, and establishing a "can-do" culture with a heavy investment in modern tools, yield the best chance for success.

A ten-point model for building a large system is:

1. Start small,
2. Establish good customer relations,
3. Build prototypes,
4. Baseline the requirements,
5. Build project plans,
6. Enlarge the staff as necessary by building small teams,[5]
7. Invest in processes to encourage teamwork,
8. Hold periodic project meetings,
9. Manufacture the software for releases, and
10. Incrementally develop the features.

(Manuscript approved December 1995)

References

1. D. Batory, V. Singhal, M. Sirkin, and J. Thomas, "Scalable Software Libraries," SIGSOFT '93, December 1993.
2. James Gleick, *Chaos: Making a New Science*, R. Donnelly and Son, Richmond, Virginia, 1987 (See especially page 71).
3. Robyn R. Lutz, "Targeting Safety-Related Errors During Software Requirements Analysis," SIGSOFT '93, December 1993.
4. Philip E. Ross, "The Day the Software Crashed," *Forbes*, April 25, 1994, p. 150.
5. J. P. Moreland, E. Sieli, M. Socratous, and S. Udovic, "Small-Team Development in a Competitive Environment," *AT&T Technical Journal*, January/February, Vol. 75, No. 1, 1996, pp. 15-23.
6. R. N. Sulgrove, "Scoping Software Projects," *AT&T Technical Journal*, January/February, Vol. 75, No. 1, 1996, pp. 35-45.
7. N. R. Deutschen, E. J. Bowers, and G. W. Hankford, "ASCC: The Impact of a Silver Bullet," *AT&T Technical Journal*, January/February, Vol. 75, No. 1, 1996, pp. 24-34.
8. B. Boehm, "A Spiral Model of Software Development and Enhancement," *ACM SIGSOFT Software Engineering Notes*, Vol. II. No. 4, August 1986, pp. 14-26.
9. Capers Jones, *Assessment and Control of Software Risks*, Prentice Hall, Englewood Cliffs, N. J., 1994.
10. R. W. Selby, "Empirically Analyzing Software Reuse in a Production Environment," *Software Reuse: Emerging Technology*, edited by W. Tracz, IEEE Computer Society Press, Parsippany, N.J., 1988, pp. 176-189.
11. D. G. Belanger, Y.-F. Chen, N. R. Fildes, B. Krishnamurthy, P. H. Rank Jr., K.-P. Vo, T. E. Walker, "Architecture Styles and Services: An Experiment Involving the Signal Operations Platforms-Provisioning Operations System," *AT&T Technical Journal*, January/February, Vol. 75, No. 1, 1996, pp. 54-63.

Larry Bernstein *is Network Operations Vice President of AT&T Network Systems at Warren, New Jersey. He is responsible for software technology and the automated systems used for managing telecommunications networks. Mr. Bernstein joined the company in 1961. He has a B.S.E.E. degree from Rensselaer Polytechnic Institute in Troy, New York, and an M.S.E.E. degree from New York University in New York City.*