SOFTWARE PROJECT MANAGEMENT AND QUALITY ENGINEERING PRACTICES FOR COMPLEX, COUPLED MULTIPHYSICS, MASSIVELY PARALLEL COMPUTATIONAL SIMULATIONS: LESSONS LEARNED FROM ASCI

D. E. Post
R. P. Kendall

LOS ALAMOS NATIONAL LABORATORY,
LOS ALAMOS, NM, USA (POST@LANL.GOV)

Abstract

Many institutions are now developing large-scale, complex, coupled multiphysics computational simulations for massively parallel platforms for the simulation of the performance of nuclear weapons and certification of the stockpile, and for research in climate and weather prediction, magnetic and inertial fusion energy, environmental systems, astrophysics, aerodynamic design, combustion, biological and biochemical systems, and other areas. The successful development of these simulations is aided by attention to sound software project management and software engineering. We have developed "lessons learned" from a set of code projects that the Department of Energy National Nuclear Security Agency has sponsored to develop nuclear weapons simulations over the last 50 years. We find that some, but not all, of the software project management and development practices (rather than processes) commonly employed for non-technical software add value to the development of scientific software and we identify those that we judge add value. Another key finding, consistent with general software industry experience, is that the optimal project schedule and resource level are solely determined by the requirements once the requirements are fixed.

Key words: Software engineering, verification, validation, software project, management, computational science

Acknowledgments

The authors are grateful for discussions with and suggestions from Tom Adams, Marvin Alme, Bill Archer, Donald Burton, Gary Carlson, John Cerutti, William Chandler, Randy Christiansen, Linnea Cook, Larry Cox, Tom DeMarco, Paul Dubois, Michael Gittings, Tom Gorman, Dale Henderson, Joseph Kindel, Kenneth Koch, Robert Lucas, Tom McAbee, Douglas Miller, Pat Miller, David Nowak, James Rathkopf, Donald Remer, Richard Sharp, Anthony Scannapieco, Rob Thomsett, David Tubbs, Robert Weaver, Robert Webster, Daniel Weeks, Robert Weaver, Robert Webster, Dan Weeks, Don Willerton, Ed Yourdon, Michael Zika, and George Zimmerman.

The International Journal of High Performance Computing Applications,
Volume 18, No. 4, Winter 2004, pp. 399–416
DOI: 10.1177/1094342004048534

1 Introduction

In the middle of 1996, the Department of Energy (DOE) launched the Accelerated Strategic Computing Initiative (ASCI) to develop an enhanced simulation capability for the nuclear weapons in the US stockpile. The Los Alamos National Laboratory (LANL) and Lawrence Livermore National Laboratory (LLNL) were tasked with developing this capability for the physics performance, and the Sandia National Laboratory (SNL) for the engineering performance of weapons systems. The ASCI program is now almost eight years old and now has been renamed to Advanced Simulation and Computing (ASC). It is an appropriate time to assess the progress and to develop "lessons learned" to identify what worked and what did not. This paper presents the "lessons learned" for successful code development during the ASCI project so far. The major points are summarized in Table 1.

In the absence of testing, improved nuclear weapons simulation capability is needed to sustain the US defensive capability. Following the fall of the Soviet Union and the cessation of testing nuclear weapons by both Russia and the US in the early 1990s, the US inaugurated the "Stockpile Stewardship" program to maintain its nuclear stockpile. Even though the Russian Federation poses a much reduced threat to the US compared to the Soviet Union, history, particularly the history of the twentieth century, has amply demonstrated that any nation that does not possess a strong defense based on modern military technology can – and often will – fall victim to an aggressor. The US and Russia have been in the process of reducing their stockpiles from the level of tens of thousands of warheads needed to counter a "first strike" to the thousands of warheads needed for deterrence. The nuclear weapons mission is to sustain and maintain the US reduced stockpile for the foreseeable future. The existing stockpile consists of weapons systems highly optimized for specific missions and for the maximum yield to weight ratio. They were designed for a 15–30 year shelf life with little consideration given to possible longer-term aging issues. The weapons program now has the challenge of adapting the existing warheads for different missions, and extending their lifetimes to 40 to 60 years without the ability to test the nuclear performance. The strategy developed for "Stockpile Stewardship" has four major elements:

Table 1
Code development "lessons learned" from the ASCI program at LANL and LLNL.

- Build on the successful code development history and prototypes for your organization
- Good people in a good team are the most important item
- Software project management: run the code project like a project
- Risk identification, management, and mitigation are essential
- Schedule and resources are determined by the requirements (goals, quality, team survival and building, and added value)
- Strong customer focus is essential for success
- Better physics is much more important than better computer science
- Use modern but proven computer science techniques; do not be a computer science research project
- Train the teams in project management, code development techniques and the physics and numerical techniques used in the code
- Software quality engineering: use best practices to improve quality rather than processes
- Validation and verification are essential

1. active surveillance of the stockpile to identify problems and issues so that they can be fixed;
2. revival of the capability for manufacturing and refurbishing weapons;
3. development of enhanced fidelity computer simulations for nuclear weapons;
4. development of an active experimental program to validate the new simulations.

The nuclear weapons community has been developing and using complex codes since 1943. Indeed, weapons simulations were among the first applications for computers. A typical mature nuclear weapons simulation code has about 500,000 lines of Fortran or C code (Post and Cook, 2000). Its development and support usually involves an investment of about 200 to 400 man years spread out over 10–20 years. The code development and support group usually has 5–15 members who generally stay with the same group for 10 years or more. The code is used daily by a group of 10–50 weapons designers to analyze various weapons systems and experiments (Table 2).

The scale of the code development task is truly immense. The "legacy" weapons simulations codes were only able to model variations in one or two spatial dimensions, provided under-resolved solutions to severely simplified equations, and used physical data and materials models with a largely semi-empirical basis. They were successfully used to provide interpolated results between experimental results from underground nuclear tests. In a sense, they were highly sophisticated regression fitting algorithms. Without the ability to field test weapons to determine the impact of new conditions due to aging and modifications, the DOE turned to simulations. The outgrowth of this new emphasis on simulation was the ASCI program. Clearly new tools were required with a much more reliable prediction capability.

The new ASCI codes thus have two and three spatial dimensions, provide adequately resolved solutions of exact equations, and employ more accurate physical data and materials models. The increase in computing power from 1995 to 2004 required to achieve these advances is about 10^5. The ASCI nuclear weapons simulation development program is complemented by a strong effort in the development of solution algorithms, physical data, materials data, code development tools and massively parallel computer platforms and operating system software. Legacy codes generally took 10 to 20 years or more to mature and were often used for 30 to 40 years (Figure 1). They were developed by small teams – often just 3 to 5 professionals. The ASCI codes are needed in about 10 years or less, and, since they have as many or more components – each with more complexity – than the fully mature legacy codes, parallel component development is required. This results in larger code teams, up to 20 or 30 staff.

2 Build on the Successes of Your Institutional Software Development History

One of the most successful approaches is to look at your organization and similar organizations and see what has worked and what has not. This not only appears to be a successful approach at LANL and LLNL but is recommended by the authors of myriad software books and courses (e.g. McConnell, 1997; Remer, 2000). A quantitative database of successful software projects is required for good estimation for resources and schedules (Jones, 1998).

LLNL has been successfully developing the capability for simulating the performance of nuclear weapons since the late 1950s (Post and Cook, 2000). They have successfully developed state-of-the-art simulations for every major supercomputer platform since the 1950s (Figure 1). They have successfully coped with massive changes in computer languages, operating systems, platform architectures, and memory structures. It is not uncommon for a nuclear weapons code to have a useful life of 30–40

Table 2
Characterization of nuclear weapons simulations (Post and Cook, 2000).

Property	LANL/LLNL
Code complexity	20–50 independent packages to simulate different physics phenomena; massively parallel, iterative physics
Product	Working code delivered to local users, support for individual users common
User base	Small, homogeneous and collocated, tight coupling of users and developers
Code size	100,000 to 1,000,000 lines of executable code (loc); typically ~ 500,000 loc
Code updates releases	Major, 1–2; minor, 20–100 or more per year
Computer hardware risk	New, bleeding edge, new, "beyond the state-of-the-art" platforms
Technology risk	Algorithm R&D is necessary to develop new methods to solve physics problems
Funding	Level of effort, only loosely determined by scale of task, resource elements only roughly estimated by historical trends
Fault tolerance level	High; users can recognize and filter faults and defects and get rapid fixes from code groups; often users suggest fixes
Size of code groups	3–25 professionals (computational physicists, engineers, programmers, computer scientists, computational mathematicians, theorists, etc.)
Project lifetime	10–35 years, usually ongoing continuous development of codes
Responsiveness	Code groups must respond rapidly to user requests, hours to days for simpler fixes, months to years for new algorithms and physics
Requirements	Largely captured in prior codes and corporate experience, also users can set requirements
Multiple use codes	Codes are used for both research and production; design experiments, analyze problems with stockpile
Code evolution	Continual replacement of code modules as better techniques are developed
Module coupling	Modules are tightly coupled across disparate time and distance scales, usually operator splitting is adequate, but iterative solvers are beginning to be used for closely coupled physical phenomena
Verification	Comparison of calculated results with analytic test problems, comparison with other codes, checks on conserved quantities, regression test suites, infrequent convergence tests
Validation	Comparison with data from underground nuclear tests; comparison with past and current above ground experiments

years. The major elements of their success appear to be the following.

- Strong emphasis on building and supporting code development teams and expertise.
  - Long term (5–10 to 20 years or more) support of code teams with low turnover rates.
  - Code teams composed of a mixture of senior, experienced staff and younger staff, with the senior staff mentoring the less experienced staff.
  - Computer scientists as an integral part of code development teams, usually through a long-term matrix assignment or direct hire into the code development team.
- Three-phase life cycle (Figure 2).
  - Development of initial capability and initial use by customers: 5–10 years.
  - Enhancement of initial capability and support of heavy use by customers: 10–30 years.
  - Retirement and phase-out, support of declining user base: 5–10 years.
- Strong customer focus and continuous direct interactions with customers, either by embedding the code development teams in the user organization, or in organizations that are closely coupled to the user organization.
- Requirements come from customers (the users), requirements creep minimized.
- Prior projects serve as prototypes for new projects.
- Code development proceeds in steps: develop a core capability with a small team; let the users try it; if successful, add more capability (i.e. incremental delivery).
- Strong emphasis on the improvement of the physics capability.
- Conservatism with regard to computer science.

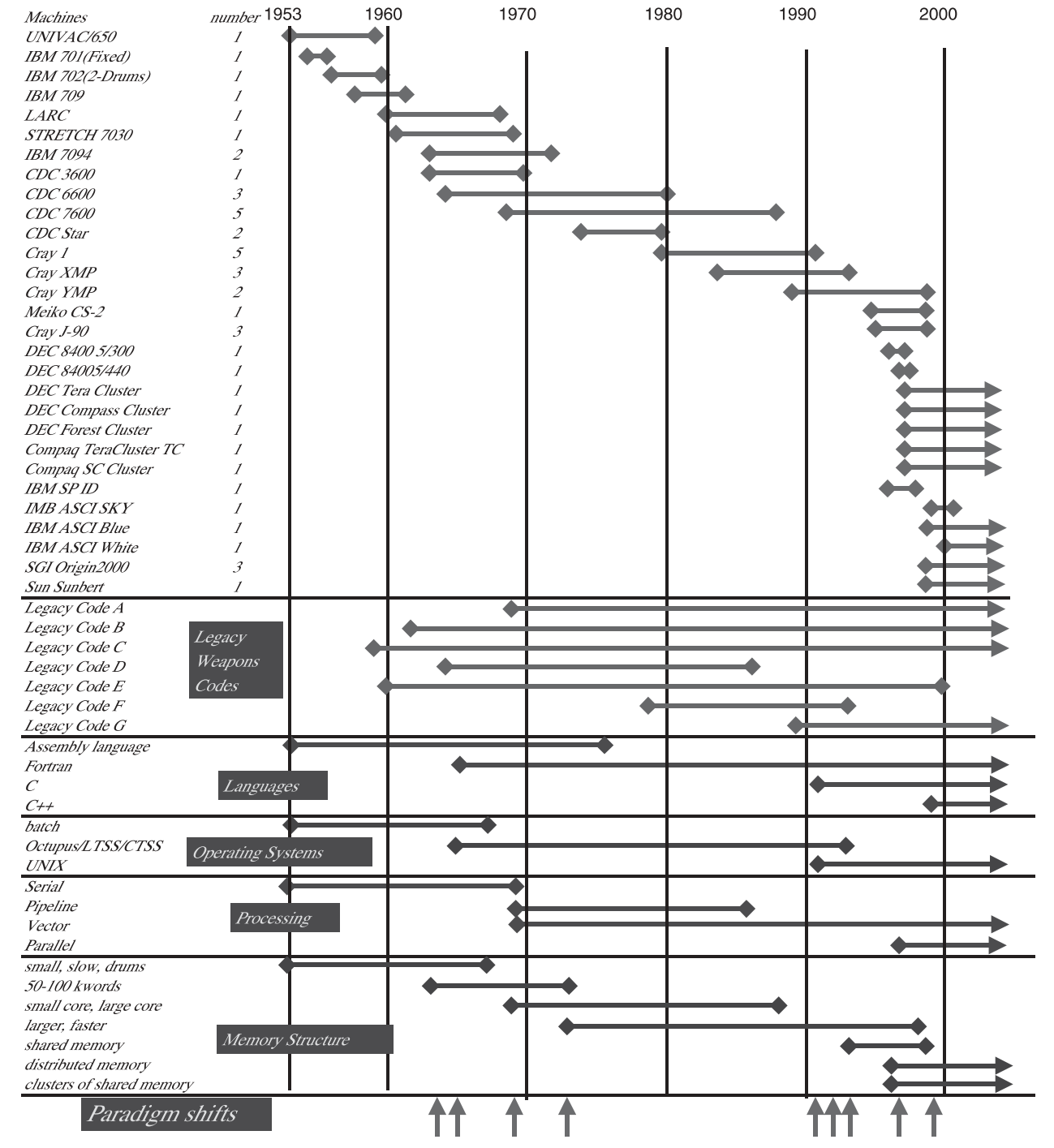

Fig. 1 LLNL computer history.

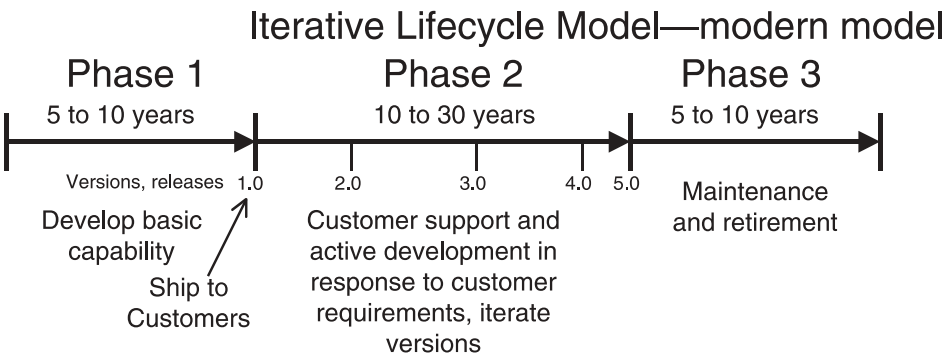

Fig. 2 Weapons code lifecycle.

A perhaps surprising element of the LLNL experience is the lack of formal requirements for new projects. The need for a formal set of requirements is mitigated by the close interactions between the code developer and users, the use of prior simulation codes as prototypes (each new code provides improved capability with relatively few completely new features), and the corporate knowledge of the code developers. As the code projects have become larger and the average experience level has dropped, the importance of more formal requirements has grown.

LANL has had similar experiences, although they have started fewer codes than LLNL. Their strategy has generally been to import codes with initial capability from other institutions, then modify and enhance the imported code to provide the capability needed by the users. LANL's efforts to develop new codes from scratch have been unsuccessful. The customer focus has generally been provided by embedding the code teams in the user groups and by collocation. LANL has attempted to have a stronger emphasis on innovative computer science (e.g. Oldham, 2002), but has not realized the expected benefit from that emphasis. In fact, the code projects with a heavy emphasis on advanced computer science have been among the least successful. LANL has experimented with different organizational structures, including separating the code developments from the user groups, but has been most successful with collocated code development teams and user groups. This has proven to preserve a strong customer focus.

3 Good People in Good Teams are the Basis of Successful Code Development Programs

DeMarco (1997) states that there are four essentials of good management:

- "get the right people";
- "match them to the right jobs";
- "keep them motivated";
- "help their teams to jell and stay jelled" ("all the rest is administrivia").

The experience at LANL and LLNL confirms this. DeMarco's statements are true for software projects in general, and are even more true for technical software projects such as large-scale, multiphysics complex computational simulations. Such simulations consist of different physics modules that interact strongly. There should be at least one team member who is an expert in each required physics and computer science capability, and preferably more than one to provide risk mitigation and a critical mass for discussion and resolution of difficult issues. All of the team members must have the full professional respect and trust of the other team members. The team members must be able to openly discuss issues and difficulties with each other and with management.

Management needs to trust and support the team and team members (DeMarco and Lister, 1999; Cockburn and Highsmith, 2001; McBreen, 2001). The job of management is to facilitate, guide, and coach the team. Management needs to provide resources, solve problems that the team cannot solve, and provide a stable work environment. Thomsett (2002) puts it as, "People, not resources,

work on projects." The care and feeding of project teams is the main task of upper-level management for successful code development organizations (DeMarco and Lister, 1999). Good teams of competent people are vastly more important than good processes (Cockburn and Highsmith, 2001; McBreen, 2001). Good teams are characterized by the ability to work together informally and share ideas. Management has the duty to provide the type of environment that nurtures and supports good teams. Finally, the team provides the continuity of corporate knowledge that forms a basis for future code projects. Thomsett quotes Bill Gates as stating that "every code project at Microsoft has two deliverables: a working code and a solid code development team" (Thomsett, 2002). Without good teams, an institution cannot maintain and support its existing codes or build new ones. Many, if not most, of the present ASCI code project teams are led by "heroes". As Frederick Brooks points out in "The Mythical Man-Month", all good code projects need "conceptual integrity", a clear vision for the code and how to realize that vision. A coherent vision can be developed only by a few individuals, or just one individual: the standard role for "heroes". While "heroes" have been essential to successful code projects, particularly code projects with less than ten staff, larger-scale code projects need to evolve into more of a structured organization. As the ASCI projects grow, the myriad tasks become more than one person, no matter how talented and hard working, can accomplish. The hero needs to be able to bring in others to help realize the vision. At some scale, the hero becomes a single point of failure instead of a single point of success. The "heroes" need to evolve into senior mentors and code architects who communicate their vision and expertise to the other code team members, and mentor the more junior staff.

While the central importance of teams may seem obvious, a number of senior managers at LANL and LLNL have questioned the value of teams and have ended up destroying successful teams. They did not appreciate that it takes years to build up a good team, but only minutes to destroy one.

4 Sound Software Project Management is Essential for Success

Each code project should be managed as much like a standard software project as possible (DeMarco, 1997; Peters and Pedrycz, 2000; Remer, 2000; Highsmith and Cockburn, 2001; Humphrey, 2001; Pressman, 2001; Thomsett, 2002). However, due allowance has to be given to the differences between highly complex scientific software involving a healthy dose of research to develop improved solution algorithms and projects that do not require such research. Contingency allowances and planning and risk management and mitigation are essential for success. The project leader needs to have control of the budget, the staff, and other resources to continually adjust the task assignments so that the project goals can be met in a reasonable time. The project leader must be able to ensure that the code team members cooperate and work together constructively. The leader must have the support of the project sponsors and the stakeholders, and must have their strong backing to manage the project. Responsibility and authority go together. The project leader must be able to remove disruptive team members. If the project leader does not have this general level of authority, the project leader is not a project manager, but only a project "cheerleader" (Remer, 2000). The project should have clearly understood goals, a stable budget and resources, a realistic schedule, and support from the external stakeholders for elements of the project that are essential for success, but are to be provided by external stakeholders. The status of the project should be measurable, and each team member should have access to the status at all times. Training in software project management has proved to be very useful as well. All of this seems obvious, but it has been surprising how often one or more of the above tenets have been violated at LANL and LLNL.

5 Risk Identification, Risk Management and Risk Mitigation and Key Elements of Success

DeMarco lists five major risks for software projects (DeMarco and Lister, 2002):

1. uncertain or rapidly changing requirements, goals and deliverables;
2. inadequate resources or schedule to meet the requirements;
3. institutional turmoil, including lack of management support for code project team, rapid turnover, unstable computing environment, etc.;
4. inadequate reserve and allowance for requirements creep and scope changes;
5. poor team performance.

To these we add two more:

6. inadequate support by stakeholder groups that need to supply essential modules, etc.;
7. tackling a problem that cannot be solved with the available resources in the available time.

It is revealing that only item 5 is the responsibility of the team. It has been frequently assumed that most unsuccessful software development efforts fail because the team did not perform, but that is not the experience in the software industry or at LANL and LLNL. DeMarco and

Lister (2002) state that uncertain requirements and inadequate resources and schedule are the most common causes of project failure, and that is the experience at LANL and LLNL. This is where the generally informal requirements specification process at the labs is dangerous. It has worked as well as it has so far because the users have been able to adjust their expectations to what is actually delivered and have been strongly involved in the process at every level. The labs have over 50 years of experience in nuclear weapons modeling, and thus know in considerable detail what physics needs to be in the codes. The only real issue is the degree of improved capability. The degree of involvement lessens as the code project teams become larger, and a more formal requirements process would add value. Some flexibility in the requirements specification phase is essential because it is difficult to predict when (or if) a research program to invent or discover a new algorithm or model will be successful (e.g. successful development of a new materials model that provides better fidelity).

Inadequate schedule and resources have also been project killers at the labs. Often, resources are picked at a level that can be obtained or are in the externally determined budget and the schedule is picked to match the desires of the program for a capability. If these are picked without regard to what can actually be accomplished, failure usually occurs. In his book, "Death March", Yourdon (1997) documents that "overly ambitious schedules" have killed many code projects and destroyed many code teams, mostly unnecessarily. We discuss the importance of schedule and resource estimation in the next section. Remer (2000) states that annual code team turnover rates of 15% or more will doom a multiyear complex code project because the corporate memory will vanish and too many team members are taken away from the development work to train the new team members. Poor team performance is often blamed for project difficulties, but close examination almost always shows that, while it may be a factor, the other risk items usually dominate, and usually contribute to poor team performance. For example, several major security incidents at LANL led to much institutional turmoil that contributed to poor morale and poor performance for the code projects at LANL. However, this is really an institutional turmoil issue rather than a team performance issue. Also, poor team performance does not suddenly become evident overnight. Management has a responsibility to identify poor performers and disruptive team members and remove them from the team. Poor performers not only do not complete their work but also impede and discourage their team members. Disruptive team members will destroy the team. Also, if the code project is relying on another organization to provide an essential package or module, and the other organization does not deliver a workable package, the project will fail. Early involvement by upper-level management is usually crucial to avoid this problem. This cannot be fixed by the project manager or the project team. It can only be fixed by the project sponsor who is part of upper level management (Thomsett, 2002). Finally, it must be possible to solve the equations that represent the system being modeled with available techniques and computing power. Tackling a problem that cannot be solved will not lead to a successful code project.

The team needs to identify its risks and build contingency into the development plan to increase the chances of success in event of problems. An example is the need to pursue multiple approaches for the development of algorithms and modules on the critical path. If one approach turns out not to be feasible, there is a second approach already being developed. Similarly key staff members need backups.

6 Requirements, Schedule and Resources Must Be Consistent

Successful completion of a software project, indeed any project, requires that the project have a consistent and realistic set of requirements, schedule and resources (Post and Kendall, 2002). A cardinal rule (Verzuh, 1999) for successful technical projects is that one can specify at most two but not three of these. What is not appreciated as well is that software development projects are much more restrictive. For these projects, one can specify only the requirements (Jones, 1998; Highsmith, 2000). Specification of requirements determines the optimum schedule and resource level. One can do worse than the optimum, but not better. The general experience at LANL and LLNL and in the commercial software industry bears this out. In particular, we judge that this is one of the main reasons that many of the ASCI projects have often failed to complete their prescribed milestones on schedule. This lack of success in meeting their milestones has generally given the false impression that these projects were failures when in fact the prescribed schedule was unrealistically optimistic. We are beginning to find that, given adequate time, most of projects could meet their requirements. The ASCI milestones were set at the beginning of the ASCI program in 1996 with the implicit assumption that the project requirements, schedule and resources could be independently specified. An approach better suited to ultimate success would have been to specify the project requirements and then develop the project schedule, milestones and necessary resources as part of a planning process for each code project. The ASCI program is now in the process of reformulating the milestones so that they incorporate a realistic schedule and are better reflections of overall programmatic goals.

Accurate estimation of software project schedules and resource requirements requires quantitative data that char-

acterize code capability and performance, and the time and resources required to develop that capability. Without such data, accurate estimation is difficult for new projects and estimates must be derived during the initial stages of the project (McConnell, 1997). Data from similar projects can also be used. Unfortunately, such data exist at an only approximate level within the ASCI program. In lieu of detailed historical data, we have adapted empirical scalings from Jones (1998) calibrated using the LANL and LLNL weapons code history. Jones and others measure the capability of software in terms of "function points" (FPs), a weighted total of inputs, outputs, inquiries, logical files and interfaces (Symons, 1988; Jones, 1998). While FPs do not capture all of the complexity of scientific software, they are the best metric available in a simple form. Single lines of executable code can be converted to FPs (e.g. equation (1)). Jones (1998) lists the equivalent single lines of code (SLOC) per FP for the common computer languages, since computer languages have different information densities:

$$\mathrm{FP} = \left(\frac{\mathrm{C{+}{+}\ SLOC}}{53} + \frac{\mathrm{C\ \ SLOC}}{128} + \frac{\mathrm{F77\ \ SLOC}}{107} \right) \quad (1)$$

$$\text{schedule (months)} = \mathrm{FP}^{x};\ 0.4 < x < 0.5;\ \text{use } x = 0.47 \quad (2)$$

$$\text{real schedule} = \text{contingency} \times \text{function point schedule} + \text{delays} \quad (3)$$

$$\text{team size} = 3 + \frac{\mathrm{FP}}{150} * 0.6. \quad (4)$$

Using FPs, Jones presents semi-empirical scalings (equations (1)–(4)) for the time required to complete the project (schedule) and the recommended average team size that we modified.

The use of these scalings requires mapping from the commercial software environment to the environments at LANL and LLNL. We therefore added the additional time it takes to recruit, hire, train and process security clearances for code development staff as compared to conditions in the commercial software industry. We estimate this additional time to be at least one year and probably more like two to three years. It is less for staff recruited from other parts of the laboratories, and at least two years or more for new hires. A reasonable average is about 1.5 years. Commercial software companies typically have much shorter times of three to six months or less.

A contingency factor is also necessary. We used a standard contingency model developed by the LLNL engineering directorate and described in the Course on Software Project Management by Remer (2000). The contingency accounts for the additional "viscosity", risks and uncertainties involved in developing codes for the complex and challenging LANL and LLNL computing environments compared to the commercial software development environment, typically on mature single processor Windows or Unix boxes with stable compilers and operating systems, less stringent security requirements, and relatively straightforward algorithms. Examples of items that add viscosity for LANL and LLNL weapons code projects include:

- computing on two disjoint and unconnected computing systems (classified and unclassified);
- delays and low worker efficiency due to the instability and immaturity of the new ASCI platforms and the software for code development (compilers, operating systems, debuggers, etc.);
- the extra work required because the platforms, operating systems and code development environments change every two to three years;
- the paradigm shift from single processor systems to massively parallel platforms;
- the general complexity of the multiphysics models in the codes;
- the transition from two-dimensional to three-dimensional models;
- the need for extensive algorithm research and development.

The LLNL contingency model with the assumptions made for each of these factors yields a contingency factor of 1.6 for new projects. A contingency of 1.6 is somewhat high for standard engineering projects, but is not atypical of engineering projects that have significant technical risk and require a substantial amount of technology R&D (Remer, 2000).

To develop semi-empirical scalings to estimate schedule and team size on LANL/LLNL simulation code projects, we modified the form of the Jones scalings to account for the LANL/LLNL environment and then incorporated the LLNL contingency factor. Then we calibrated these scalings using the experience of seven weapons code projects from LLNL and LANL, including six ASCI code projects and one legacy code (Table 3, Figure 3). We also modified the scaling for team size to provide a better fit to the data (equation (3)). The minimum team size of three is a reflection of the complexity of multiphysics projects, and the factor of 0.6 reflects the challenges of integrating complex multiphysics codes that limit the size of the code team and the high degree of specialized training for the code team (cf. Brooks, 1987). The team size is the peak staffing level for the code project (Figure 3).

We analyzed seven code projects, three at LLNL and four at LANL (Table 3). We have identified the LLNL

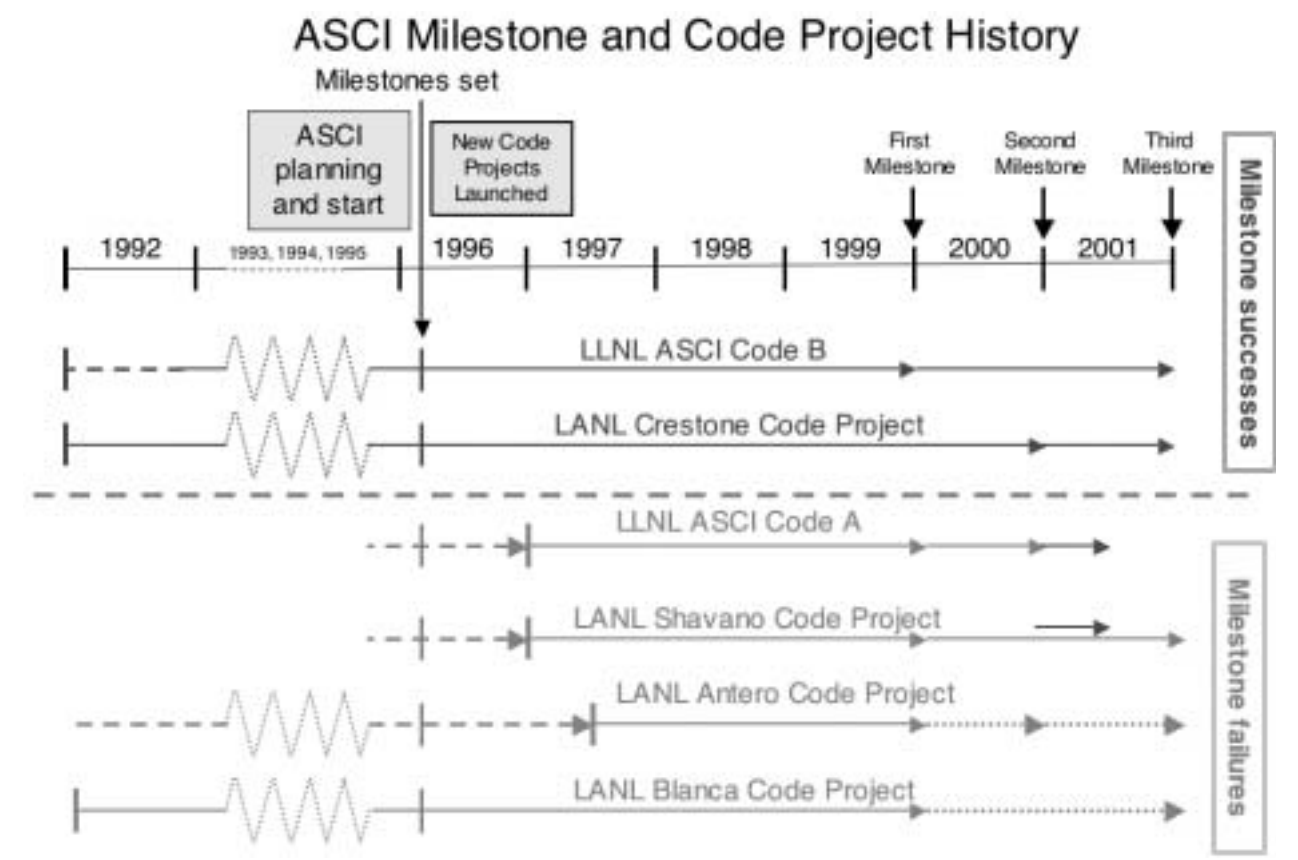

Fig. 3 ASCI planning, milestone and code project schedules.

codes with the letters A and B to avoid classification issues. Table 3 lists the size of the code in FPs, the time estimated by equation (4) to develop the initial capability of the code project, the actual age of the code at the point it was expected to accomplish its first milestone, whether or not the project succeeded, the optimal code team size estimated from equation (3), the actual size of the team, and the estimated man years and actual man years for the project. The ASCI planning and milestone schedule are summarized in Figure 3. The scalings indicate that codes of the ASCI class should take at least seven to nine years to develop, consistent with the ages of the projects that either succeeded in meeting their prescribed milestones or did not. The scalings for the time required for a code project to succeed (i.e. meet its ASCI milestones) are consistent with the observed historical data if a contingency factor of 1.6 is used. The codes that were successful were all older than the age that was estimated to be necessary by equation (2) (generally about eight years). Those that were unsuccessful in meeting their milestones on schedule are all younger than the estimated required time. The average team size is between 15 and 25, close to the observed team sizes.

Four of the ASCI codes were started before ASCI began in 1996 (ASCI B, Legacy A for LLNL, and the Crestone and Blanca code projects for LANL). ASCI B was started in 1992 and had a working prototype in 1994. The Crestone code project was started before 1992. ASCI A and the Shavano code project were started in late 1996 and early 1997. The Antero and Blanca code projects had roots that extended back before 1992. The core of the Blanca code project was a working parallel Fortran code imported from another institution. The Blanca team decided to "modernize" it by completely rewriting it in C++ with

Table 3
Software resource estimates for the LLNL and LANL code projects. Items in bold denote computed numbers (equations (1)–(5)) and items not in bold are historical data.

	LLNL			LANL			
	ASCI A	ASCI B	Legacy A	Antero Code Project	Shavano Code Project	Blanca Code Project	Crestone Code Project
Single lines of code	184,000	640,000	410,550	300,000	500,000	200,000	314,000
FPs (equation (1))	4800	6000	5400	2900	4800	3800	2900
Estimated schedule (equation (4))	8.7	9	6.9	6.6	8.1	7.4	6.7
Project age (at initial milestone date)	3	9	N/A	4	3.5	8	8
Successful in achieving initial ASCI milestone	No	Yes	N/A	No	No	No	Yes
Estimated staff requirements (equation (3))	22	27	24	14	22	18	14
Real team size	20	22	8	17	8	35	12

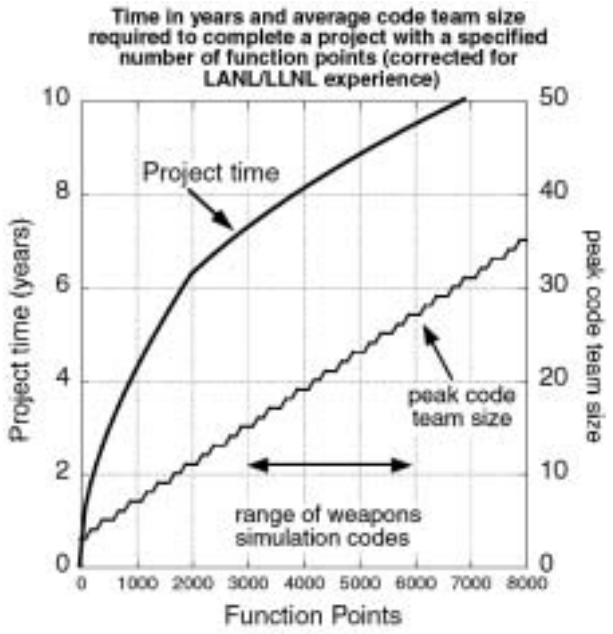

Fig. 4 Time required to complete a project and average code team size as a function of code capability measured in FPs (Symons, 1988; Jones, 1998).

admixtures of "advanced computer science" techniques that were then being developed. LANL retained the original Fortran imported code with minimal support (generally 0.5 to 1 staff). It outperformed the codes from the Blanca code project during the entire life of the Blanca code project. The Antero code project team sought to join two existing separate mature code packages that had been developed in the late 1980s and early 1990s. Neither the Blanca nor the Antero code project was successful in meeting their milestones because they violated many of the "lessons learned" in Table 1. These lessons were expensive. The LANL ASCI program spent about $50M on the Antero code project and close to $100M on the Blanca code project. Since we are able to match the history of weapons codes with scalings derived from the experience of the commercial software industry, we can also conclude that the constraints, computer science practices and management issues that generally apply to the commercial environment apply to the development of weapons codes (i.e. there is no "silver bullet" that can radically reduce the development time; Brooks, 1987).

The historical evidence and the estimation procedures indicate that it takes generally a minimum of eight years to develop an initial capability for a weapons code. The requirements for a weapons code are fixed by the physics necessary to simulate a nuclear weapon. LANL and LLNL have over 50 years of experience in this area, and know these requirements in detail. The requirements are thus not very flexible. In terms of FPs, weapons codes require at least 3000 FPs and some require up to 6000. For many reasons, it is difficult to field code teams of more than about 20 staff, but this is adequate according to the estimation procedures and historical evidence. These points are reflected in Figure 5.

Experience with the ASCI code projects is consistent with the general experience of the software industry embodied in standard software estimation methods when contingencies are included. ASCI B at LLNL and the Crestone code project at LANL have been very successful in meeting their mileposts. In each case, the age of the

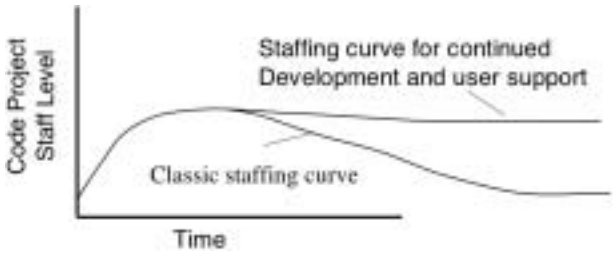

Fig. 5 Optimal staffing schedule (Remer, 2000; DeMarco and Lister, 2002).

successful project at the time they met their mileposts exceeded eight years. ASCI A at LLNL and the Antero and Shavano code projects were not successful at meeting their initial milestones. In those cases, the age at the time the milepost was due was several years less than the required time estimated from equation (4). In fact, at the time they failed to meet the relevant mileposts, their age was generally half of the age of the successful projects and of the required time estimated from equation (4).

The various projects had different computer languages, framework structures, project organizations, degrees of software quality processes, platform vendors, laboratory management structures, staff maturity levels, and many other factors. These factors undoubtedly played some role, but success seems to correlate consistently with age of the project. Adequate time was a necessary, but not sufficient, condition. The Blanca code project failed even though it had adequate time.

The software project management literature and software development experience (DeMarco, 1997; McConnell, 1997; Remer, 2000) stress the need to start a project with a small number of staff who can develop the concept and plan. Once the concepts and plan are developed, the team can then be staffed up to the level needed to accomplish the project. Too large a staff at the beginning leads to a confused plan (e.g. too many cooks spoil the soup) and locks the project into directions that often do not contribute to project success (Figure 5).

To determine the sensitivity of the estimates to the assumptions for the estimation formulae, we varied the exponents in equation (2) and the contingency (Figure 6). The estimates changed by 10–20%, but the conclusions we draw from Table 3 remain valid. Indeed, the initial ASCI milestones were shorter than even the estimates with no contingency.

A number of senior managers at LLNL and LANL have suggested that had the projects been more "aggressively" managed, they would have been successful at meeting

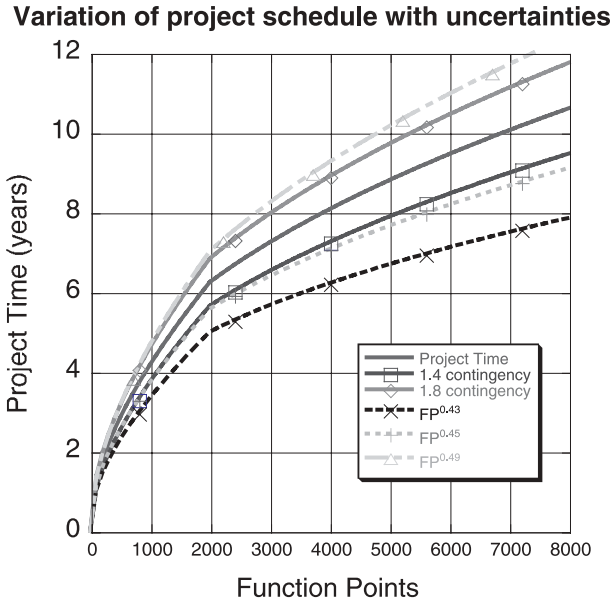

Fig. 6 Variation of estimated project times for different assumptions for the contingency and for the exponent in equation (2).

their milestones on schedule (cf. DeMarco, 1997). In fact senior management subjected much pressure on all of the code projects that did not meet their milestones. Extensive overtime was authorized. During the period just before the milestone, senior management monitored the progress on a weekly and sometimes daily basis. The code teams worked very hard, but were not able to achieve a sustained increase in their rate of progress. This is consistent with the experience in commercial software development (DeMarco, 1997; Yourdon, 1997). DeMarco (1997) notes in his books that "aggressive" management and management pressure are never successful in accelerating the code development schedules for more than a very short time interval. A basic reason is that the limiting factor in software development is the rate at which software developers think – the rate at which they can solve problems. "Management pressure and aggressive managers do not make people think faster" (DeMarco, 1997). In fact, excessive pressure slows projects down and retards delivery of the project, if it does not kill the project (Figure 7; Esque, 1999; DeMarco and Lister, 2002). Paraphrasing Yourdon (1997), "overly ambitious software development schedules are the leading cause of software project failure."

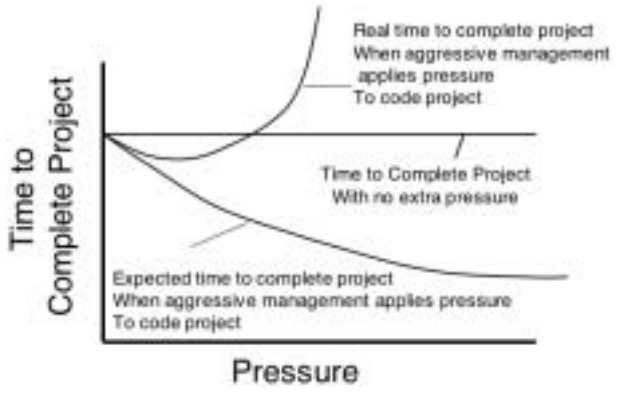

Fig. 7 Schematic illustration of the effect of management pressure on the schedule to complete a software project (DeMarco and Lister, 2002).

Another way to state the lesson to be drawn is that software development time has been very inelastic for the ASCI code projects, as it has been for the software industry in general. When resources have been added to a project that is late, it has not helped very much and has sometimes hurt. For one thing, the new personnel need to be trained and integrated into the team, and this often slows things down. In addition, the complexity of the project may not lend itself well to a large staff. The empirical scaling quoted by Remer (2000) is that the resource level, R, required to speed up a project behaves as $R \sim t^{-4}$ where t is the time for development. Decreasing the schedule by 20% requires a 150% increase in resources. This assumes that the resources can be utilized efficiently, something that is very challenging for complex, technical software projects.

History – both in the ASCI program and in the software industry in general – shows that software project requirements, schedule and resources are not independent. The schedule and resources are determined by the project requirements. This important fact must be recognized if future ASCI and other large, complex multiphysics code development projects are to be successful and meet their milestones. Milestones must be determined as part of the detailed project plans for each project, not set independently. If projects have different maturity levels and similar requirements, they will succeed at different times, not at the same time. As noted before, there is also strong evidence that "aggressive" management retards success, rather than hastening it (DeMarco, 1997; Yourdon, 1997; Esque, 1999). The DOE ASCI program has, over the last year, worked with LLNL and LANL to modify the original milestone plan (Figure 3) to make the plan more consistent with the requirements of the weapons program and to ensure that the milestone schedule is realistic. This is a key step forward since it is essential that milestone schedules be consistent with sound software project planning and what the code development projects can actually accomplish, rather than what management thinks they ought to be able to accomplish. Otherwise, the ASCI code projects that do not have the time or resources to meet the milestones on schedule will not be able to do so. These projects will then unjustly and inaccurately be judged failures and the ultimate success of those projects will unnecessarily be jeopardized.

The need for consistent requirements, schedules and resources is actually more demanding for software projects than for normal construction projects or than stated above. Thomsett (2002) points out that there are in reality seven elements for a project, not three. A successful project should:

- meet the project's objectives/requirements/goals/deliverables;
- meet an agreed budget – resources, capital, equipment;
- schedule – deliver the product on time;
- satisfy the stakeholders;
- add value for the organization;
- meet quality requirements;
- provide a sense of professional satisfaction for the team.

Meeting requirements, budget and schedule are customary project criteria. Satisfied stakeholders (those outside the project team who sponsor or support the project or are the customers) are important for acceptance of the product. If the customers are not satisfied with the final product, the project will not be a success even if the requirements were met. If the expectations of the stakeholder support groups or the sponsor are met, then the project really has not succeeded in the eyes of the institution. Managing and meeting the expectations of the stakeholders is essential. Management and the project team both own this issue. Adding value to the institution is also an important criterion. For nuclear weapons simulations, this means that the code is a substantial improvement over prior codes, and engenders greater confidence in the customers, e.g. users, lab management, DOE, Department of Defense (DoD), etc., that they can make more accurate predictions. The code must also meet quality requirements. Not only should the code be relatively free of bugs and errors, but must be easy to use, and be robust and reliable. The DoD and DOE are beginning to impose software quality standards on codes developed with their funds. Improving software quality is expensive, and meeting objective standards for software quality is even more expensive. Finally, an essential outcome is a satisfied code project team. Otherwise, the institution will be unable to maintain the code and will not have good teams to develop the next code. Resource and schedule estimates need to include these additional issues.

7 Code Project Team must have a Customer Focus

One of the most important criteria for success, mentioned frequently above, is the customer focus of the code project. The most successful code projects have been collocated with the user groups and have interacted with the users on an almost daily basis. This helps to keep the code teams responsive and motivated, and helps to develop trust among the code development team, the users and management. The continuous interest by the users in the progress of the codes helps sustain the enthusiasm and dedication of the code teams. It is easy to lose motivation on a project that lasts years and has few incremental deliverables.

There are many examples of code projects that have failed because they lacked customer focus. The code developers (and possibly upper-level management) felt that they knew what the users needed better than the users. Almost always they were wrong and the new code was never used. In most cases it failed to deliver what the users actually needed. In the few cases where the code did have value, there was no user buy-in because of the adversarial relationship that had been established. Valuing and supporting the customers has been an essential part of every successful code project at LANL and LLNL.

Another reason for customer involvement is that the customers do much of the initial testing and almost all of the validation. Without their involvement, the code team will need to do all of the testing and validation. The code developers will not have the same level of credibility that the users do. They also will not be as familiar with how to carry out validation, will not have the same level of familiarity with the data, etc.

We observed that good teams, supported and nurtured by management, are normally much more focused on customers than teams that lack management support and nurturing – points made in Cockburn and Highsmith (2001). In particular, heavy-handed top-down management seems to discourage customer focus because the team then sees upper-level management as the customer rather than the real customer.

The development of critical modules for multiphysics code projects is often the responsibility of stakeholder groups not part of the project team. It is vital that the stakeholder groups understand that their customers are the code project teams. It is absolutely essential that the support groups view the success of their module within the integrated code project as more important as the success of the independent module on its own. The integrated code project will fail if critical modules do not function properly in the integrated code project. Achieving this has been especially challenging for "discipline" or "functional" organizations such as LANL. Several key LANL code projects have been seriously delayed or threatened with failure due to problems with delivery of modules from stakeholders outside the core project team or in other organizations. LLNL is organized more along project and matrix lines than LANL, but even LLNL occasionally has problems with support groups providing critical components.

8 Better Physics is the Key to Successful Prediction

The predictive value of the weapons simulations (or any physics simulation) depends on the quality of the physics in the codes (the right equations, good quality physical data and materials models) and accurate solution algorithms for the equations and adequate spatial and temporal resolution. To achieve the desired improvement in predictive ability, the ASCI program has supplemented the effort to develop better weapons simulation codes with programs to develop improved linear and non-linear solvers, better materials models, better physical data for equations of state, opacities and neutral particle cross-sections, better models for turbulent mix, and more accurate transport algorithms. This effort to develop better data and modules for the simulation codes has also been supplemented with a strong experimental program to provide experimental data to help improve the physics basis of the models and to validate both the detailed models and the integrated multiphysics calculations. When the weapons program was conducting underground nuclear tests, the weapons design codes could be benchmarked and calibrated with the test data from an actual weapons test. The test data usually provided information in integral form, i.e. the total performance of the device. Partially because of the hostile physical environment in the vicinity of a nuclear explosion, data on detailed effects were difficult to obtain. Since many of the materials models and other models had a semi-empirical basis and the physical data had uncertainties, it was generally possible to adjust the data and models within the range of uncertainties to fit the integral data. This was generally adequate for interpolation within the general problem domain bounded by nuclear test data, but was not adequate for extrapolation and prediction outside of that domain.

The paradigm shift from interpolation to prediction has been fundamental (Laughlin, 2002). Simulation errors are potentially limited when the domain is bounded and spanned by experimental data, so interpolation has some validity. Parameter adjustments can be made in order to maximize the cancellation of errors due to inadequate treatments of different effects. However, when predictions are made outside the domain of the test data, the errors are the sum of the errors due to each element of the calculation. It is not possible to rely upon compensating errors.

The only way to improve the predictive capability is to improve the quality of every part of the calculation. The same considerations generally apply to scientific software.

This paradigm shift has involved a major culture change. Managing the culture change in an evolutionary fashion has also proved to be a challenge. The code development programs have had to continue to support the old tools while developing the new tools. The funding agencies and review boards have had difficulty understanding the importance of maintaining the old tools that the users need until the new tools are fully functional. It is first necessary for the new codes to achieve the capability of the old tools before moving forward to better capability. Otherwise the connection to the validation database and the user community is not maintained. This issue needs to be explicitly featured in any strategy for massively upgrading the quality of a computational science effort. Otherwise, the code developers are caught in a "catch 22" situation where they spend much time in limbo trying to keep the old codes alive and healthy while trying to develop the capability to replace them with better ones and being unable to get sympathy for this from their sponsors.

9 Use Modern, but Proven Computer Science

An important element of success is minimization of risks. The general experience in the labs is that the major payoff comes from better physics. Developing a better treatment of the physics usually is sufficiently challenging that conservatism for the other aspects of the code development effort is essential. This particularly applies to the role of computer science in an application code project. Good, sound computer science is essential for a successful project, particularly those in the ASCI program. The codes have to run on the very latest massively parallel platforms with thousands of processors. The platforms and platform architecture change every two or three years. Sound configuration management is essential for a complex multiphysics code involving as many as 20, 30 or more separate, large-scale modules being developed and integrated by a team of 10–30 members of staff. Obtaining reasonably efficient performance on the complex platforms is a demanding computer science task. Visualization of large data sets (up to terabytes) is essential for debugging, problem generation, and analysis of the results. The computer scientists need to be fully integrated into the team and treated as professionals with equal professional status to the physicists. Use Fortran 90, not Fortran 2000 which has not been standardized. If you use C++, do not use templates and inheritance classes in all their glory. Do not use MPI II until it has been shaken out and so on. Do not participate in the latest computer science fad. Let the new ideas mature, and let someone else get all the glory and the pain of using the latest and greatest computer science. You will have enough challenges.

Given the challenges associated with successfully developing and integrating a complex, multiphysics code for multiple massively parallel new and unstable platforms, getting the physics right, and porting it to a new platform every two years, our record indicates that mixing in a strong component of new and unproven computer science usually makes the total task much more difficult and is often fatal. For instance, almost all of the successful ASCI projects have used Fortran or C as the basic language. Those using C++ and other advanced languages (if they succeed at all) have been slower to develop capability and have exhibited lower computer performance efficiencies than the C and Fortran codes. One factor is that C++ and many other advanced languages have a steep learning curve, and are more complex. Indirect addressing often diminishes performance to unacceptable levels. The successful code projects have emphasized modularity, transparency, simplicity, and portability rather than performance optimization.

There have been several efforts to develop advanced backplanes and code development environments, e.g. POOMA (Oldham, 2002) and Sierra (Stewart and Edwards, 2001). The promise of such tools is great, but none has been an unqualified success so far. LANL unsuccessfully attempted to use the POOMA framework as the basis of the Blanca code project. As noted before, LANL spent over 50% of its code development resources (approximately $100M) on the Blanca code project without success. Only when the computer science research elements were replaced by an emphasis on physics and customer focus was some success realized. The generalized framework had poor performance and was not sufficiently flexible to accommodate all of the different types of modules and physics algorithms and mesh types that were needed for the code to be a success. The Sierra framework has been more successful for engineering applications, but has taken a long time to mature. The prudent approach is to support computer science research, but wait for the research to mature and produce a reliable product.

10 Develop the Team

As noted above, software is developed by teams, not systems, processes or organizations. The most effective training is mentorship of the newer team members by the more experienced team members. In addition, formal training has proven very useful to help the teams jell and to improve their skills. DeMarco and Boehm (2002) stress that "part of our 20-year-long obsession with process is that we have tried to invest at the organizational level instead of the individual level… (*we should be*)

investing heavily in individual skill-building rather than organizational rule sets." LLNL and LANL have brought in highly experienced software development leaders such as Tom DeMarco, Rob Thomsett, Ed Yourdon and Don Remer to give courses in risk management, software project management, estimation techniques, and software quality issues. This introduces a sense of best industry practices into the work. Not only does this train the team in the use of tools and methods, but also it helps to give the team members a sense of perspective on how things are done at other institutions. The experience of participating in a course with other members from the same team helps bond the team. Participating in a course with members of other teams helps develop bridges to the other teams, and enhances the sharing of experiences.

Team members should be encouraged to view code development as a professional activity. They should be encouraged to join and attend the conferences of the IEEE Computer Society and other computer-related organizations, and to subscribe to the relevant journals. We have found that forming an in-house library of software development books and journals for the staff also helps. Training in the tools, languages, and methods for computing is also essential. Even if the team members do not use advanced languages, every computer literate code developer should be trained in C, C++, PERL, Python, Unix and LINUX, etc. Similarly, training in the relevant physics issues and numerical techniques is important. Seminars and colloquia with internal and external speakers are also important. Inspiring speakers such as Fred Brooks, Tom DeMarco, and others make a positive difference. In addition to informal contacts, it has proven useful to have the customers formally address the team and describe how the simulation tools will be used and what the issues are.

The code development teams are the major asset of a code development organization and need to be nurtured and encouraged to grow and develop (Cockburn and Highsmith, 2001). Anything that can be done to improve the skills of the team members and management is worthwhile. Team building retreats and planning retreats have also proven useful.

11 Software Quality Engineering is Important: Practices and Processes

Software quality engineering (SQE) and software quality assurance (SQA) are major issues for the commercial software industry, especially industries developing software for the DoD. In the mid-1980s the Air Force and other parts of the DoD were experiencing major overruns and failures due to problems with delivery of software from contractors. Planes were crashing, rockets were not working, and satellites were failing due to software problems. The Air Force set up the Software Engineering Institute (SEI) at the Carnegie Mellon Institute of Technology to develop a set of standards and methods for prospective DoD software vendors (Paulk, 1994). The SEI developed the Capability Maturity Model (CMM) for vendors to adopt. The SEI surveys show that the CMM does improve repeatability and software delivery on schedule (Herbsleb et al., 1997). However, there are costs associated with implementing the CMM to improve the development process. In general, it takes of the order of two years to implement the first level of improvement, and much effort (Herbsleb et al., 1997; Remer, 2000). The CMM is often not recommended for ongoing projects (DeMarco, 1997). The strong emphasis on reliability and repeatability comes at a sacrifice of flexibility and innovation (Rifkin, 2002). It is tough to explain to others why everyone should not adopt the CMM or similar standards (e.g. ISO 9000). Who can be against quality? The problem is that one method does not fit all situations. Adopting SQA requires resources and time in addition to reducing flexibility and innovation, so a cost/benefit trade-off is essential.

We have found two ways of examining these issues that are helpful for developing a balanced perspective. The first is based on the taxonomy of Rifkin (2002) for technology development. Rifkin places software project values and goals into the three categories or attributes defined by Treacy and Wiersema (1995): operational excellence, product innovative, and customer intimate. In order to succeed, every project must have all three attributes, but must concentrate and excel in only one. This attribute must be the element of the product that matches the desires and needs of the customers. An example of an operational excellent product is a simple but robust computer program that must have no bugs because it must be 100% reliable, e.g. the software that controls a jet fighter. Innovative software has new capability that is essential, but necessarily entails some research and development, e.g. improved nuclear weapons simulation codes. Customer intimate products are focused on the detailed needs of the individual customer and must be flexible to respond to these varied needs of their customers. They typically feature large menus and many options. An example might include the latest version of MS Word, with so many features that a person can go crazy trying to turn most of them off. Rifkin sees process driven systems as being good at training an organization to produce operationally excellent software, but poor at producing the other types of software. Our needs are more in the product innovative category, and we need more flexibility than is achievable under highly process driven systems.

It is essential to use an organized approach to code development. It is necessary to appreciate that innovative products are developed by creative, highly trained scientists who are quite independent minded, highly skeptical and not very willing to accept things on authority. What

Table 4
Some essential elements for software project management

Software requirements management (documented and controlled)
Careful software design
Software project planning and tracking
Work breakdown structure for the project – lists of tasks to be accomplished
Estimation of the resource and schedule requirements for the project
Development of a plan for accomplishing the tasks
Development of a schedule for accomplishing the tasks
Identification of the resources needed to accomplish the tasks
Monitoring and tracking the plan
Risk assessment and management and contingency planning
Definition and control of the interfaces between project elements
Configuration management
Extensive regression testing
Integrated system testing and verification tests
Documentation (including requirements, functional specifications, critical software practices, physics and algorithm description, and a user manual)
Incremental delivery of capability
Continual interaction with customers
Construction of prototypes

the code projects are trying to accomplish is very ambitious: the schedules are wildly optimistic, the code teams are too small, and the users want the new code yesterday. We would never get them to be enthusiastic about laying everything aside for months or years to "improve their processes". Scientists learn early in their careers to identify and use things that add value to the way they carry out their work, and to avoid doing things that do not add value. Otherwise they would not have survived a PhD program in a scientific field. Scientists are trained to question authority and believe only what they can verify themselves. This is precisely what we value in them. We cannot expect them to adopt a different point of view when it comes to how they do their work. As Roache (1998) states, creative software developers and scientists and orderly process people are at the opposite ends of the Myers–Briggs personality scale and are naturally antagonistic. We have found that it is much more successful to advocate and employ "best practices" instead of "best processes" (Phillips, 1997). The LANL software requirements document (Cox et al., 2002), drawn up by the code project teams themselves, lists the set of best practices that the teams have judged useful for improving the way they do business (Table 4; Cox et al., 2002). Getting the

Table 5
Best practices: people issues

Users – understand them
Buy-in and ownership by everyone
Technical performance related to value for the business
Executive sponsor and support by stakeholders
Fewer, better people (management and technical)
Use of specialists
Clear management accountability

Table 6
Best practices: contained in the CMM level 2

Document everything
User manuals (as system specifications)
Documented requirements
Fight featuritis and creeping requirements
Cost estimation (using tools, realistic versus optimistic)
Planning and use of planning tools
Quality gates (binary decision gates)
Milestones (requirements, specifications, design, code, tests, manuals)
Visibility of plans and progress
Project tracking
Design before implementing
Risk management
Quality control
Change management

teams to develop their own list of the practices is a key element in getting them to adopt the practices. It is also a good way to encourage the code teams to share experiences and expertise among different teams. Teams usually cooperate and share practices more readily than algorithms and other "proprietary" packages where there is sometimes more competition.

Some of the distinctions above are a matter of perception. In fact, if we look closely, there is much in common among the CMM processes and best practices listed in Table 4. Table 5 lists the intersection of what we think is important for the code teams and the best people practices specified by the SEI. Table 6 lists the overlap of our best practices and the CMM level 2 processes/practices. Table 7 lists the overlap of our best practices and higher-level CMM processes/practices. The key difference is that the code teams can decide what adds value and what does not.

It is interesting that the SEI has begun to emphasize sound software project management as an important element of successful projects (Humphrey, 2000; 2001). Indeed, sound management practices appear to be as successful as institutional process improvement for reducing the defect rate (Humphrey, 2001). This matches our experience. There is no substitute for organizing the team and the work, monitoring the work as it progresses and rearranging the tasks as the project evolves.

Table 7
Best practices: contained in the CMM higher levels

Reviews, inspections, and walkthroughs
Reusable items
Testing early and often
Defect tracking
Metrics (measurement data)

Attention to software quality is important for other reasons too. The sponsor is accountable for the success of the project. If he does not think the team is giving adequate attention to quality, he may impose processes that are probably less well suited to the team than those they identify themselves.

12 Verification and Validation are Essential for an Accurate Simulation Capability

Verification and validation are essential elements for scientific codes (Lewis, 1992; Roache, 1998). All of us have refereed computational physics papers that we could not prove were right. At best, the results were plausible. In fact, if a code is not validated and verified, the users have no reason to believe that the results have any connection with reality (Laughlin, 2002). We define verification as ensuring that the code solves the equations in the code correctly, i.e. that there are no coding errors or mistakes in the code. Validation is defined as ensuring that the code results are a faithful reproduction of the natural world, i.e. that the models expressed in the code are correct. Verification is a mathematical exercise. Validation consists of comparing the code results with experimental data. Benchmarking is comparing the code results with the results of other codes. Benchmarking is useful during the code development process but does not give the same level of assurance of accuracy as verification and validation.

Both verification and validation are essential if the code results are to have credibility with the customers. For the nuclear weapons codes, validation is largely carried out by the users who compare the code results with the results from underground tests for full systems and above ground tests for specific physics features. Verification is carried out by comparing the code results for model problems that have analytic answers or by convergence rate tests. A third verification technique is "manufactured solutions". This involves the creation of hypothetical analytic solution that is turned into a real solution by adding a source term to the original equation that makes the manufactured solution the exact solution of the modified equation (Salari and Knupp, 2000; Roache, 2002). Validation proceeds at different levels. Each module must be validated by single purpose experiments, and then the integrated code must be validated. Integral experiments (ones that, for example, produce a single value to represent a complex system) are not sufficient. They do not test for compensating errors and effects. The nuclear weapons program is carrying out both small-scale experiments for validating single physics effect modules and large-scale experiments such as the National Ignition Facility at LLNL (Lindl, 1998) to provide validation data for multiphysics calculations.

13 Summary

After eight years of the ASCI program, a number of key conclusions and lessons we have learned are evident, as follows.

- Good teams of highly competent staff are essential. Everything else is second order.
- Schedules and resource levels are determined by the requirements. Setting them independently will wreck the code projects.
- Base the schedule and resource estimates on your institution's code development experience and history.
- Run the code project as an organized project.
- Identify the risks and provide mitigation. In particular, set clear requirements, insist on management and stakeholder support, and obtain adequate schedule and resources with contingency.
- Maintenance of customer focus is essential for success.
- Better physics is the most important product.
- Minimize risks, use modern but proven computer science; computer science research is not the goal.
- Invest in your people with training and support.
- Emphasize "best practices" instead of "processes".
- Develop and execute a verification and validation program.

AUTHOR BIOGRAPHIES

Douglass Post is a physicist in the Physics Division at the LANL. He was the Deputy Division Leader for Simulation in the Applied Physics Division at LANL from 2001 and 2002. From 1998 to 2001, he was the Associate Division Leader for Computational Physics for "A" and "X" Divisions at the LLNL. He graduated from Stanford University with a PhD in physics in 1975. He has 30 years of experience with the development of technical software and computational physics and project management in magnetic fusion, atomic and molecular physics, transport phenomena and nuclear weapons at LANL, LLNL, and the Princeton University Plasma Physics Laboratory. Doug was leader of the tokamak modeling group at the Plasma Physics Laboratory from 1975 to 1993. He was head of the Physics Project Unit for the International Thermonuclear Experimental Reactor Conceptual Design Phase (1988–

1990) and head of the In-Vessel Physics Group during the Engineering Design Phase (1993–1998). He is the Associate Editor-in-Chief of the IEEE/AIP publication "Computing in Science and Engineering", and a fellow of the American Physical Society and of the American Nuclear Society. His current professional interests include the development of software engineering methodologies for scientific computing as Team Leader for Analysis of Existing Codes for the Defense Advanced Research Projects Agency (DARPA) High Productivity Computing Systems Project.

Richard Kendall recently retired as the Chief Information Officer at LANL. He graduated from Rice University with a PhD in mathematics in 1973. He has over 30 years of experience in the development of technical software in the upstream oil and gas industry and in computer information and computer security technology. He started his professional career as Senior Research Mathematician at Exxon Production Research Company in 1972. In 1982 he left to join a start-up petro-technical software venture, J.S. Nolen & Assoc., as Vice-President. This company specialized in reservoir simulation codes for the then emerging vector supercomputer market. This company was acquired by Western Geophysical where Kendall became Chief Operating Officer of the Western Atlas Software division. He joined LANL in 1995. He is a contributing member of the Society of Petroleum Engineers (SPE) and the Society for Applied Mathematics (SIAM).

References

Brooks, F. P. 1987. No silver bullet: essence and accidents of software engineering. *Computer* 20(4):10–19.

Cockburn, A. and Highsmith, J. 2001. Agile software development, the people factor. *Computer* 34(11):131–133.

Cox, L. et al. 2002. LANL ASCI Software Engineering Requirements. LA-UR-02-888. Los Alamos National Laboratory, Los Alamos.

DeMarco, T. 1997. *The Deadline*. Dorset House, New York.

DeMarco, T. and Boehm, B. 2002. The agile methods Fray. *Computer* 35(6):90–92.

DeMarco, T. and Lister, T. 1999. *Peopleware, Productive Projects and Teams*. Dorset House, New York.

DeMarco, T. and Lister, T. 2002. *Risk Management for Software*. The Cutter Consortium, Arlington, MA.

Esque, T. J. 1999. *No Surprises Project Management.* ACT Publishing, Mill Valley, CA.

Herbsleb, J. et al. 1997. Software quality and the capability maturity model. *Communications of the ACM* 40:30–40.

Highsmith, J. A. 2000. *Adaptive Software Development*. Dorset House, New York.

Highsmith, J. and Cockburn, A. 2001. Agile software development: the business of innovation. *Computer* 34(9):120–127.

Humphrey, W. S. 2000. *Introduction to the Team Software Process*. Addision-Wesley, Reading, MA.

Humphrey, W. S. 2001. *Winning with Software: An Executive Strategy*. Software Engineering Institute, Pittsburg, PA.

Jones, T. C. 1998. *Estimating Software Costs*. McGraw-Hill, New York.

Laughlin, R. 2002. The physical basis of computability. *Computing in Science and Engineering* 4(3):27–30.

Lewis, R. O. 1992. *Independent Verification and Validation: A Life Cycle Engineering Process for Quality Software*. Wiley, New York.

Lindl, J. 1998. *Inertial Confinement Fusion*. AIP Press, New York.

McBreen, P. 2001. *Software Craftmanship*. Addison-Wesley, Reading, MA.

McConnell, S. C. 1997. *Software Project Survival*. Microsoft Press.

Oldham, J. D. 2002. Scientific computing using POOMA. *C/C++ Users Journal* 20:6–22.

Paulk, M. 1994. *The Capability Maturity Model*. Addison-Wesley, Reading, MA.

Peters, J. and Pedrycz, W. 2000. *Software Engineering: An Engineering Approach*. Wiley, New York.

Phillips, D. 1997. *The Software Project Manager's Handbook*. IEEE Computer Society, Los Alamitos.

Post, D. and Cook, L. 2000. A Comparison of Software Engineering Practices used by the LLNL Nuclear Applications Codes and by the Software Industry. UCRL-MI-141464. Lawrence Livermore National Laboratory, Oakland, CA.

Post, D. and Kendall, R. 2002. Estimation of Software Project Schedules for Multi-Physics Simulations. LA-UR-02-7159. Los Alamos National Laboratory, Los Alamos.

Pressman, R. S. 2001. *Software Engineering: A Practioner's Approach.* McGraw-Hill, Boston.

Remer, D. 2000. Managing Software Projects. UCLA Technical Management Institute, UCLA Extension Courses, Los Angeles, CA.

Rifkin, S. 2002. Is process improvement irrelevant to produce new era software? Software Quality – ECSQ 2002 7th European Conference, Helsinki, Finland. Springer-Verlag, Berlin.

Roache, P. J. 1998. *Verification and Validation in Computational Science and Engineering*. Hermosa, Albuquerque, NM.

Roache, P. J. 2002. Code verification by the method of manufactured solutions. *Transactions of the ASME* 124:4–10.

Salari, K. and Knupp, P. 2000. Code Verification by the Method of Manufactured Solutions. SAND2000-1444. Sandia National Laboratories, Albuquerque, NM.

Stewart, J. R. and Edwards, H. C. 2001. Parallel Adaptive Application Development using the SIERRA Framework. First MIT Conference in Computational Fluid and Solid Mechanics. Elsevier, Boston, MA.

Symons, C. R. 1988. Function point analysis: difficulties and improvements. *IEEE Transactions on Software Engineering* 14(1):2–11.

Thomsett, R. 2002. *Radical Project Management*. Prentice-Hall, Englewood Cliffs, NJ.

Treacy, M. and Wiersema, F. 1995. *The Discipline of Market Leaders: Choose Your Customers, Narrow Your Focus, Dominate Your Market*. Perseus Books, Reading, MA.

Verzuh, E. 1999. *The Fastforward MBA in Project Management*. Wiley, New York.

Yourdon, E. 1997. *Death March*. Prentice-Hall, Upper Saddle River, NJ.