LA-UR-02-888

Approved for public release;
distribution is unlimited.

Title: LANL ASCI Software Engineering Requirements

Authors: LANL ASCI SQE Working Group

Los Alamos
NATIONAL LABORATORY

[This page intentionally left blank]

LANL ASCI SOFTWARE ENGINEERING REQUIREMENTS

Version 1.4

LA-UR-02-888

Revision Date
February 18, 2002

Major Contributors:
ASCI SQE Working Group

Lawrence J. Cox, X-5 (chair, editor)

Hilary M. Abhold, CCN-12
Billy J. Archer, CCN-12
David L. Aubrey, CCN-12
Gregg C. Giesler, CCN-12
Dale B. Henderson, X-1
Nelson M. Hoffman, X-1
Joseph M. Kindel, X-1
Kenneth R. Koch, X-DO
Stephen R. Lee, CCS-DO
Jonathon Parker, CCN-12
Maysa M. Peterson, CCN-12
Jungjo Pyun, X-4
Robert W. Robey, X-3
David L. Tubbs, X-2
Daniel D.Weeks, X-2

[This page intentionally left blank]

MANAGEMENT APPROVALS

Name	Date
Donald G. Shirk Acting Deputy Associate Laboratory Director for ASC	2/21/2002
John F. Morrison CCN Division Leader	2/22/2002
Andrew B. White, Jr. Acting CCS Division Leader	2-21-2002
Earle-Marie Hanson ESA Division Leader	2/25/02
Rodney B. Wood-Schultz X-Division Leader	2/22/02

[This page intentionally left blank]

Preface

The purpose of this document is to enhance the software engineering practices for ASCI application codes by building on the 1999 SQE Assessment while meeting the expectations of the Los Alamos ASC Program and NNSA. Certification relies on analyses of systems using simulation codes to predict complicated weapon physics phenomena. The integration of small-scale precision experiments with a validated simulation capability and scientific judgment form the basis of weapon system certification today. NNSA and DoD are requiring quantitative assessments of predictive capability confidence levels and associated uncertainties.

This report represents an opportunity for us to improve the methods and practices for code development at Los Alamos. It was prepared by a group of computer scientists, computational physicists, and software engineers who are intimately involved in the development of the ASCI codes at Los Alamos to define ways the can improve the way they develop applications codes at the Laboratory. The specific recommendations and the renewed commitment from the representatives of all the ASCI code teams can help us improve the quality of our codes and provide an objective record of how we would accomplish this goal.

Today, in this document, we have a framework for practices and standards that are advocated by the leadership of the ASCI project teams. The ASC program at Los Alamos and the Applied Physics (X) Division are strongly committed to improving our software development and software engineering methods to meet customer expectation standards and nuclear facilities safety standards. The authors of the report have surveyed the practices utilized by developers of many different types of software, and have identified the specific practices that are most useful to us. A self-analysis through the questionnaire in the final appendix is the next step.

These practices can be implemented in a graded approach, with three levels leading to a higher degree of quality and efficiency: the Basic SQE level, the Intermediate level, and the Advanced level. Starting each code with the Basic Level goals, we will assign some teams higher goals reflecting their current stronger status and/or programmatic positions as relevant. At the Basic SQE Level each team is expected to employ:

- *Life Cycle Definition,*
- *Software Requirements Management (documented and controlled),*
- *Software Project Planning and Tracking,*
- *Software Configuration Management,*
- *Regression Testing,*
- *Integrated System Testing,*
- *Documentation (including requirements, functional specifications, critical software practices, physics and algorithm description and a users manual),*
- *Risk Assessment and Mitigation Strategy,*
- *Preliminary Metric collections, and*
- *Mentoring and Training*

Management of requirements and functional specifications is essential for successful code development. It allows us to respond to changing requirements in a non-disruptive fashion and on a realistic time scale. It also allows us to set requirements and functional specifications for other groups that supply us with data and other support.

We need to conduct our software development efforts with goals, tasks, schedules, resource estimates, while monitoring and tracking the progress. We can identify the critical path items, reallocate resources, target additional contributions, and manage risks. This will also allow the ASCI code projects to credibly request assistance from other groups when needed.

The area of the strongest positive findings in 1999 was that of configuration management. We need to share the best practices ands methods of version control, and code release formalisms. The ability to trace a given floor release to its test and verification history is crucial; these data must be maintained by configuration management. Bug tracking is also an important adjunct to configuration management.

Testing is an essential element of the development process. Testing is needed for verifying the implementation of physics algorithms, the structure and logic of the code, and almost every aspect of code execution. Frequent testing with automated regression tests suites can alert the code team to problems and effects caused by changes in the code or operating system. A strong emphasis on systematic testing is needed for each code project.

Our codes are expected to have a life of at least 20 years and probably much longer. There will likely be several generations of physicists and computer scientists working on the code over its life. Complete documentation of the code (including the design of the code, the requirements, the physics and computational algorithms and the input and output) is essential for the next generation code developers and users. Almost all of the code projects have developed User's Manuals and documents that describe the physics and algorithms employed in the code. We need to compliment these with Programming Manuals and a living Software Design Document (SDD) for each code.

Risk management is important. Each team needs to develop a specific risk management strategy, identifying software components and the critical development path. The specific risk-management strategies must also be consistent with the overall ASCI risk-management protocol. The broadly developed software quality standards and practices recommended in this report are important because they result from an ASCI cross-team collaboration. Working together on the implementation of these goals will benefit us all. Confidence in the correctness and predictive capability level of the codes is essential to code developers, designers and analysts, and weapon certification.

We are indebted to the members of the LANL ASCI-SQE working group for developing this document. It was started over a year ago, and went through many revisions. We would also take this opportunity to thank Dr. Lawrence J. Cox who spearheaded this effort with enthusiasm, diplomacy, and perseverance.

Table of Contents

Tables

Figures

1 INTRODUCTION

1.1 Scope and Intent

One objective of this document is to identify a required set of software development practices for the Department of Energy (DoE) Advanced Simulation and Computing (ASCI) Program at Los Alamos National Laboratory (LANL). Another objective is to identify the implementation process for these required practices. The practices selected are those deemed to cost-effectively promote the reliability and maintainability of software which DoE Order 203.1 mandates. In addition to scientific simulation codes, this includes software developed under the Applications Development (APPS), Verification and Validation (V&V), Materials and Physics Models, Problem Solving Environment (PSE), Distributed Communication and Computing (DisCom^2), Path Forward, Visual Interactive Environment for Weapons Simulation (VIEWS), and other software funded by the ASCI Program. This document defines the process that will be used at Los Alamos to respond to federal law and to this DoE order. Other sources that generated suggestions and requirements for important software engineering activities are listed in references [1-4].

This document defines an approach to improving software quality in the ASCI Program based on the DoE graded approach to operating practices [3-6]. It includes a discussion of practices at various levels of sophistication with the required goal of continuously improving quality of software developed in support of stockpile stewardship. Although we are concerned with formal software engineering practices, to be consistent with preceding documentation we refer to these practices as Software Quality Engineering (SQE).

The goal is to provide a set of practices that can be applied in a stepwise manner to improve the overall quality of software developed on ASCI projects. These activities will help minimize the cost and schedule of producing reliable, maintainable software. Each project will develop their own time lines for implementing the SQE practices defined in this document. These time lines will vary from project to project. Additionally, all such time lines are subject to program and line management review and approval.

This document defines a continuum of software engineering practices characterized by three levels (SQE Levels)—Basic, Intermediate and Advanced—in which the listed practices are introduced to improve the efficiency of development, and quality of developed software. Application of these practices will increase the quality of projects' developed software over a period of time. These SQE levels should be implemented as appropriate to a given project as the project matures during its lifecycle.

Although this document does provide definitions of practices, it is not a general software management guidebook. Such a document would describe the entire software process, from requirements and functional specifications gathering to design, development, testing, implementation, and maintenance. It is beyond the current scope to pull these topics together into a single document for ASCI. This document does not provide a roadmap to begin a software project. It provides a basic set of activities to properly frame expectations for the improvement in the quality of software developed for ASCI.

1.2 Document Overview

This first section is the introduction. It is a discussion of the scope and intent of this document. Section 2 gives an overview of LANL's graded approach to best practices and relates them to SQE levels. Details of these best practices are given in Section 3. The process of evaluating and planning for the implementation of our software engineering evolution is described in Section 4. References follow Section 4. Measures and metrics are discussed in Appendix A. A table of audit items is given in Appendix B. Questionnaires to be used for self-evaluation and to guide internal and external assessments are given for each best practice in Appendix C. Italics are used when introducing a term whose definition is used frequently in the document.

2 GRADED APPROACH

The DoE has defined the term graded approach in reference to nuclear facilities and quality assurance to mean the tailoring of operating practices to a specific degree of formality comensurate with the risk and impact of failure [6]. In ACSI software development, we at LANL recognize the value of using a graded approach for evaluating the set of software and quality engineering practices required to be put in place by a software project based on the project's importance to the DoE mission and the goals of the ASCI Program.

It is important to understand that the software used to reach these goals is part of a broad set of systems used by a diverse workforce operating in different domains. Practices must be commensurate with the level of complexity of the software, the application domain, the associated risks of failure and impact to the DoE mission. The selection of specific software engineering practices, the rigor of their implementation, and the path forward for improving practices is based on these factors.

The list that follows describes how software projects will be categorized with respect to impact on laboratory missions. Three levels of impact are introduced: Critical, Moderate and Minimal. Software risk impact is usually rated by the consequences of failure in five domains: environmental, health & safety, programmatic, technical, and security. Software can also be categorized by how it is used, who uses it, and where it is applied. Depending on the intended end-use, additional considerations may be required.

Development and use of *ASCI software cannot and does not directly affect health & safety or the environment* since is it used or intended to be used for simulation purposes and the support thereof. Therefore these two domains are not directly considered here in the evaluation of the level of impact. End users of ASCI software will use the results of simulations to guide decision processes. In cases where software will be used to guide decisions that can have direct health & safety or environmental impact and where only one software simulation option is available (or in development), the software project shall be automatically considered to be of ***Critical Impact***.

Critical Impact: The software plays a vital role and is a critical element of the Laboratory's mission to ensure global security, threat reduction and stockpile stewardship.

Programmatic: Software project delay or failure can result in possible program cancellation due to inability to meet cost/schedule constraints. Such cancellation can lead to dire impact on the ability to fulfill the military and national security portions of the LANL mission

Technical: Faults in the software could lead to reliance on results that are invalid, but not known to be invalid, in the making of key decisions affecting portions of the LANL mission.

Security: Use of the software system may result in a classified material breach.

Uses: Software is used (or is intended to be used) by scientific end-users directly or indirectly for stockpile certification purposes or

other portions of the LANL mission having direct impact on the security of the USA.

Moderate Impact: The software plays an indirect but supportive role in meeting the Laboratory's mission.
Programmatic: Software project delay or failure can result in major program modification; cost/schedule impact is visible. Delay in software delivery can affect the ability to meet portions of the LANL mission.
Technical: Faults in the software could lead to use of results that are invalid for supportive, exploratory or preliminary calculations.
Security: Use of the software system may result in unintended release of unclassified, limited distribution information, cause export control violations, or reveal sensitive subject matter:
Uses: Software is used (or intended to be used) by scientific end-users for exploratory calculations. Software supports or is part of the infrastructure of certification software (or is intended to be such).

Minimal Impact: The software plays a remote/indirect role and has minimal affect on meeting programmatic goals.
Programmatic: Software project delay or failure can result in minor program modification, some impact on cost and/or schedule.
Technical: Faults in the software could impair performance or impact results but with limited or no impact on technical decisions.
Security: Use of software may result in unintended release of unclassified, unlimited distribution information.
Uses: Software is not used (or is not intended to be used) for stockpile certification but other programmatic uses exist; other scientific software with no existing user base.

Software projects are graded based upon importance to the overall ASCI and Stockpile Stewardship goals using the above list. The highest overall level is used when determining at which level a particular software project falls. For example: a project might have minimal security risk but moderate programmatic risk, thus the overall ranking for the purposes of a graded approach for this project would be at the moderate level. In the next section, we discuss the concept of assessing the current state of SQE and what it takes to improve the situation.

2.1 Approach to Improvement

The intent of this section is to describe how we address the graded approach to improvement by recognizing that for a given graded approach risk ranking, a particular set of software engineering practices are required. We define three levels of software engineering practices or rigor. These levels roughly correlate to risk level. A Target SQE level will be established for each project by a process of self-evaluation in concert with program and/or management review and approval. Projects may begin their development improvement efforts with an already defined set of practices. They shall be required to improve their practices over time to their Target SQE level based on the graded approach described above. If their current practices already exceed the targeted level of rigor, they shall continue to operate at least at that level.

In the initial stages of consideration, determination of location in the levels may be simple, but determining destination may be difficult. Using the information from the graded approach, coupled with a general goal of improving a software system, one can determine the desired Target Level. Assuming that Basic SQE is adequate for Low Impact software, Intermediate SQE is adequate for Moderate Impact software, and Advanced SQE is ultimately required for Critical Impact software is the most obvious approach. However, convergence to the most adequate SQE level for a given project may initially be an admixture of the software engineering practices described for these three levels. Furthermore, such an evaluation cannot be made without some basic operating assumptions.

2.1.1 Assumptions

Prior to providing some guidelines for determining position and destination, there are some underlying assumptions of doing business. We list them as follows:

- Projects shall be given time and resources (including human resources) to phase in different practices.
- Projects shall keep track of their status with respect to implementing the practices required by this document.
- Projects shall identify and explain the omission of any of the required SQE elements.
- Improvement activities shall be prioritized by project according to risk reduction, resources and return on investment.

Each project shall establish a baseline evaluation of its current software development practices with respect to the practices defined in this document. This baseline will be used in planning for and measuring the effect of these practices on the quality of the software and the impact on the mission.

Given this baseline, the project shall identify its shortfalls (if any) with respect to the Basic SQE Level (defined below). With the same baseline, the project shall identify its shortfalls with respect to its Target SQE level. The project shall prioritize the practices that are missing or in need of improvement focusing first on missing or deficient elements of the Basic SQE level. Program and line management review and approval of this priority list is required.

2.1.2 Striving for Continuous Improvement

As resources (time, staff, training, etc.) are available, the identified high priority practices shall be put in place. Missing or deficient elements of the Basic SQE level should be considered first. However, elements of higher levels that are relatively easy to put in place can also be staged in early in the improvement process. Factors that should be considered in selecting a practice for introduction or improvement include the level of effort required and its potential for impact on quality.

As improvements are made, the project shall update its SQE baseline assessment and reprioritize any remaining shortfalls. Progress to the plan and the updated improvement plan shall be reported to program and line management at least quarterly.

When all the Basic SQE requirements are in place, work shall continue to advancing to the assigned Target Level.

Each new or modified critical activity should be accompanied by appropriate metrics that can be used to assess its impact on software quality. Metrics should be utilized to help project management minimize the effort and schedule required to produce reliable software and they provide the opportunity to calibrate practices across projects, thus leveraging collective experience. Therefore, some of the metrics of interest to SQE are simply the effort-and-schedule measures of our existing or anticipated software engineering practices. Not all metric data needs to be reported to all levels of upper management.

The reporting of measure and metric data to program and/or line management requires an environment of trust and consultation between the project and management. Many useful metrics are simple numbers that are used to summarize complex information. Such numbers can easily be misconstrued or misinterpreted in the absence of detailed explanation by the project team. Management must take care not to violate this implicit trust by making sure that the project team in consulted on the correct interpretation of the data. Otherwise there will be a reluctance to share and report the raw data. Management also has an obligation to help collect measure and metric data—such as staff-hours charged—to which project teams do not have ready access so that duplicate reporting systems are not created.

See Appendix A for definition of the measures and metrics that shall be collected and reported to program and line management.

2.1.3 Best Practices and SQE Levels

"Best practices" is a formal software engineering term that refers to a set of techniques or activities that, based on the experience of the software industry, are the best-known means to help minimize the cost and schedule of developing reliable and maintainable software. These practices, when applied to scientific software development projects such as those within ASCI, will help to improve the quality of developed software over time. In this document, we have divided the use of various practices into three different levels (Basic, Intermediate, and Advanced). Where appropriate, the term "process" is used to refer to methods or procedures that involve the use of our best practices.

Before embarking on a detailed discussion of best practices, we first introduce the notion of a software lifecycle and relevant terminology.

2.1.4 Lifecycle Management

A software lifecycle is a model that describes all the activities that go into creating and supporting a software product. It delineates the time-based *flow* of activities, and breaks the modeling and software development processes into steps or phases that need visibility, thus aiding in the delivery of quality software that conforms to program requirements. Every software project uses a lifecycle of one kind or another—explicitly or implicitly.

Consideration of our software systems in terms of a lifecycle model will streamline our projects and help ensure that each step moves the project closer to one or more of its goals.

The four basic elements in a software lifecycle are:

- Requirements and functional specifications;
- Design and coding;
- Implementation and Testing;
- Release & Support.

One illustration of the interplay between these elements for a project undergoing re-engineering is shown in Figure 1. Two important aspects of ASCI software projects are that they have been under development for some time and they have current users with immediate needs.

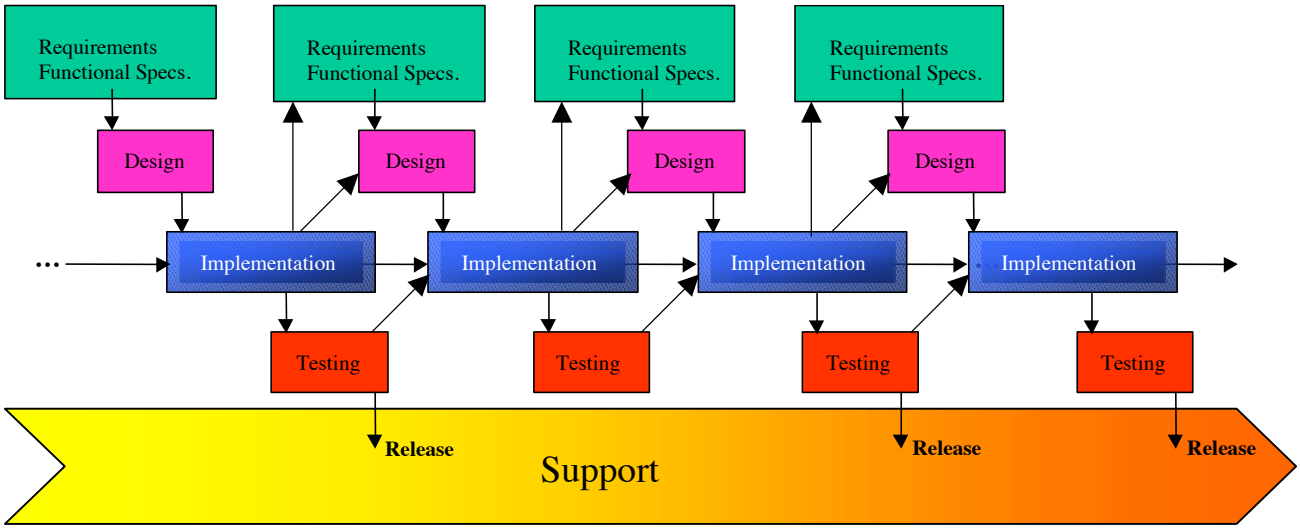


Figure 1. General Flow Through Lifecycle Activities

The continuous support band along the bottom reflects a typical LANL ASCI project cycle of maintenance of existing features and gradual introduction of new features. In this sense, features are described in the project requirements and functional specifications. As shown, this representation of a lifecycle is an abstraction of various incremental delivery lifecycle models.

The advantages of recognizing a software development lifecycle include increased knowledge of how much of the software is "complete" through identification of the completed "phases". Phases progress from left to right in the illustration. Another advantage is identification of work that needs to be done that might be overlooked if phases are not explicitly defined. Work in phases is made "*visible*"—through documentation, or review of work that was done by others on the team. Visible work products can be examined for correctness at pre-defined times, and improvements suggested.

With this cursory introduction of the notion of a software lifecycle, we now introduce our definitions of levels of SQE practices.

2.2 Levels and Definitions

This section defines three levels of software engineering practices (SQE Levels: Basic, Intermediate and Advanced). Application of the practices related to these three levels is anticipated to increase the quality of software teams' developed products over time. These practices should be *applied as appropriate* as a given software project matures during its lifecycle.

The first level—referred to as Basic SQE—is the set of practices that are considered to be a minimal set that shall be required of all ASCI software development projects at LANL.

The goal of the Basic Level is to ensure that all software development projects and project team members are aware of a minimal, established, and documented set of simple, yet important, SQE practices. By following Basic level practices, a project can ensure that the quality of a software product does not degrade as a function of time (and can survive the loss of staff or capability). Although not discussed in this document, Project Management should be in place at this level. The Basic Level also represents the starting point to advance to other levels, if needed.

The goal of the Intermediate Level is to introduce more formality into the software development process and to emphasize documentation. Metrics should be collected to assess the progress of the development of a software product as a function of time.

The goal of the Advanced Level is continual software quality improvement through the collection and analysis of metrics as formalized feedback used to directly improve the software. Risk management should be addressed.

Each SQE level is characterized by an increasingly formal set of practices for software management that should lead to improved software quality and improved documentation. The characterization of these levels is summarized in Table 1.

Table 1 **The Three SQE Levels**

SQE Level	Characterization
Basic	Minimal, established, and documented set of simple, yet important, SQE practices.
Intermediate	Introduce more formality into the software development process. Emphasize documentation. Use of as many of the SQE practices as deemed applicable to a given software system.
Advanced	Continual software quality improvement through the evaluation of metrics as formalized feedback used to directly improve the software lifecycle.

It is important to note that these three levels are not discrete. By defining these levels, a minimum expectation is established, but above that level, the practices put into use will exist in a continuum with specific activities varying between the different projects. These levels describe the minimal project standards for that level. Each level roughly corresponds to a risk ranking. A low risk project will only be required to implement the Basic Level of practices but may choose to implement others as well.

The specific practices associated with each of these SQE levels are outlined below. Section 3 of this document provides detailed descriptions of each of the specified practices and how they are related.

2.2.1 Basic SQE Level

The Basic SQE Level is the fundamental set of processes and practices that shall be put into place by all software projects in the ASCI program. This is the foundation level and represents the minimum that should be met. The Basic level has many elements, but many of them are either already in place on most ASCI projects or are easy to implement.

This level contains the following required elements:

- Software Requirements Management (documented and controlled)
- Software Configuration Management (SCM) consisting of:

    - Build System (automated)
    - Version control (configurable items including critical software components)
    - Release process (elementary)
    - Bug/Issues tracking
- Regression testing (manual, include Build/Environment testing)
- System testing (consider integrating new system tests into a regression test suite)
- Documentation for the following:
    - Requirements and Functional Specifications Document (preliminary)
    - Critical practices and basic processes
    - Build environment
    - Preliminary User Manual
- Risk assessment (identify critical software components)
- Informal reviews
- Metrics (preliminary)
- Mentoring and training

2.2.2 Intermediate SQE Level

The Intermediate SQE Level requires all of the aspects of the Basic SQE level and adds the following required elements and improvements:

- Software Requirements Management (documented, reviewed, tracked, and controlled)
- SCM supplemented with:
    - Build system (robust)
    - Version control (complete)
    - Release process (complete)
    - Bug/Issues tracking (generate reports, coordinated with change control)
    - Formal software change control (documented in SCM plan)
- System testing (process for adding to regression suite defined and documented)
- Regression testing (automated, unit coverage analysis)
- Documentation for the following:
    - Process documentation updated to include new practices and processes
    - SCM Plan (documenting all SCM processes and practices)
    - Preliminary Coding Practices
    - Software Test Plan (elementary)
    - Programmers Manuals (including physics documentation, if appropriate)
    - User Manual (complete and up to date)
- Risk Assessment (elementary)
- Regular metrics collection and analysis as feedback into Project Management
- Structured reviews
- Preliminary project-level Integrate Software Management
- Coordination with the ASCI Independent V&V effort

2.2.3 Advanced SQE Level

The Advanced level builds on the previous two levels and strives to drive the project toward a continuously improving software product. Developing a continuous and formal quality

improvement process for a software project requires all of the elements in all of the levels discussed to work in concert. At the project level, this is what is meant by Integrated Software Management. Each practice should supply information to at least one related practice.

The Advanced SQE level includes all of the previous practices, as well as the following *additional* elements and improvements:

- Software Requirements and Functional Specifications (tracked)
- SCM supplemented with:
    - Release process tied to bug/issues tracking and regression testing
- System testing (integrating new tests into regression suite formalized and required)
- Regression testing (system integration, optimal coverage, tested against requirements)
- Documentation (under change control) including the following:
    - All software lifecycle process and practices
    - Detailed Coding Practices
    - Software Test Plan (coordinated with Physics V&V)
    - Physics Verification and Validation (V&V) Plan (coordinated with the ASCI Independent V&V effort)
- Risk assessment, management, and documentation (complete)
- Regular analysis of metrics to improve project processes
- Provide metrics and analysis to integrated software management
- Formal reviews
- Project-level Integrated Software Management

3 BEST PRACTICE DETAILS

Each of the practices (and associated documentation) listed in §2.2 is discussed below. An effort is made to explain how a particular practice or associated process may be implemented at a particular SQE level.

3.1 Requirements Management and Functional Specifications

Software requirements for a project are meant to be a complete description of the problem that the software is to solve and how it interacts with the environment in which it operates. Knowledge of the problem that is being solved is essential to any software development. Requirements management consists of the activities of capturing, tracking, and controlling requirements, as well as any changes to them. This establishes and maintains a common understanding, between users and development teams, of the requirements to be fulfilled.

An agreement on requirements should be the basis for planning and managing a software project. Overall project requirements shall be negotiated in writing between code development teams and appropriate levels of management (program and/or line) prior to the establishment of new software development activities. It is understood, though, that requirements change. Provisions must be made for such change and it is important to inform all parties involved when such change occurs.

Functional Specifications are statements of services that the software should provide, data that it will generate, and how it should react to particular inputs/outputs in particular situations. Functional Specifications may also explicitly state what the software should not do. Changes to Functional Specifications lead to changes in the features of a software system. Thus, an agreement on Functional Specifications should also be part of the basis for planning and managing a software project.

There is a strong aspect of change control to requirements and functional specifications management. If requirements or functional specifications are allowed to change by the whim of management, hardware developers, or users/developers without due consideration of effects on cost and schedule and program requirements, a software project can quickly become poorly defined and out of control. Control of changes to requirements is essential. Sometimes the best solution to a now-different software problem is a complete change of approach.

Therefore, it is extremely important that software projects and their managers evaluate the effect of changing requirements or functional specifications on all parts of the system, as well as determine whether the change is really necessary. To minimize the risks of such change, each development team should document the changes acceptable to them and their users. Having a system for evaluating whether a change is necessary and the effect of the change on each piece of the system are the advantages of requirements management.

It is expected that software development teams will negotiate with their users over the preliminary requirements and specifications and come to agreement on them prior to any development activity. A simple, clear contract documenting this agreement shall be established. Additionally, at the Basic SQE Level, the project manager shall document and control the basic requirements and functional specifications for that project.

At the Intermediate SQE Level, these preliminary requirements and functional specifications shall be detailed and documented. The requirements may then be peer-reviewed. All changes to the requirements are also tracked as part of controlling the requirements. Progressing to the Advanced Level includes using these requirements in defining tests and reviewing the testing results to ensure product compliance.

3.2 Software Configuration Management

Software configuration management (SCM) is used to ensure that the integrity of the software is both known and preserved throughout its development, maintenance, and deployment. *Configurable items* can include (but are not limited to) source code, libraries (static and dynamic), compilers, test problems, data, test results, software tools (text/image processors, script interpreters, system utilities, etc), documentation, operating systems, and in special cases, hardware. Critical software components are those items fundamental to the operation of the overall software and applications and the quality of physics and numerical algorithms in the applications. Critical software components need to have an added degree of control to provide disaster recovery. The four main activities within SCM are:

- *Build System Definition and Management;*
- *Version (or Change) Control;*
- *Release Management;*
- *Issue Tracking*.

These activity areas are discussed in the following sections.

At the Intermediate SQE Level, these activities shall be documented in a Software Configuration Management Plan (SCMP). This document shall include the software change control process.

It is strongly recommended that projects avoid the use of specialized tools for these activities when commonly used tools can do the same job. The balance between cost and maintainability in terms of existing system utilities and existing expertise at LANL should be considered when selecting configuration management tools. Use of common tools will provide increased uniformity across the projects and allow for a higher level of organizational support.

3.2.1 Build System Definition and Management

A *build system* consists of a suite of tools and scripts used to construct the code from its basic components. A *robust* build system is one that is portable to all supported platforms and that applies rules to completely construct all applications within the software project with all required compiler and library options. If the construction of a version is accomplished with the execution of one script or one computer command, the build system is *automated*.

Any project-specific tools or scripts developed as part of the build system shall be considered as critical software components even at the Basic SQE Level and placed under configuration management.

An automated, robust build system enables software executables to be generated in a controlled, repeatable fashion. These types of build systems reduce the amount of manual work that developers must perform, are easily maintained and standardize builds from version to version.

At the Basic SQE level, the build system shall be included in regression testing process. At the Intermediate SQE level, the build system shall be extended towards including all builds needed for all employed compiler and system options. At this level (Intermediate), the build system should reach the robust stage.

3.2.2 Version Control

Version Control is the identification and control of the versions of all products, both by individual pieces (e.g., software module) and by appropriate groupings (e.g., set of software modules that constitute an executable program).

The advantages of version control are simple, but powerful. It allows you to reproduce any released version of the software at any time, to know what version that you have released, to know what version that bugs were in that have been fixed, and prevents one from losing old versions of software. It helps you control changes to software, and to identify when and by whom changes were made. It allows you to revert to a prior version whenever necessary or desirable.

A *repository* shall be established which controls accesses and changes to the source code and other configuration items. The repository and SCM activities should be able to perform the following functions:

- Provide control and archiving of identified software components from a defined baseline;
- Provide a journaling (history) capability to allow configuration management records to be kept (and be extracted at any time);
- Provide for the "stamping" of repository versions to preserve state (even after other modifications have been made);
- Allow multiple developers to simultaneously "check out" and "commit" snapshots of configurable items in a controlled manner;

In addition, the version control process shall be documented and clearly define:

- The process to check for code "collisions" when multiple changes are made to the same software component at the same time.
- The process(es) of placement of new or modified configurable items into the repository;
- The process for deciding when to "stamp" the repository to preserve state;
- The testing requirements for promoting versions. This may require testing of other codes that depend on this code;
- The process of ensuring that the entire software repository is backed up to tertiary storage on a regular basis.

The specific tool set used is unimportant, so long as the functionality is repeatable. Many other functions can be provided, but this is the minimal set provided as a guide for getting started.

At the Basic Level, only the critical software components need to be placed under the SCM. At higher levels, SCM control of all configurable items is required. As software processes evolve towards the Intermediate SQE level, the version control system should be complete.

3.2.3 Release Management

Release management is the control of product promotion, from development to production use. This includes internal and external releases of the executable and related documentation. Internal *alpha* and *beta* releases can include releases for review, testing, porting, or as new baselines for further development by the development team. External (or *floor*) versions can include releases for independent testing or customer use. In most cases user acceptance testing occurs to ensure that the users are satisfied with the product.

At the Basic SQE level, a fundamental characteristic of release management is the ability to recover the complete computer and hardware environment necessary for recovering a version of the software and its executable. This relies heavily on the cooperation and coordination of many support teams at LANL including the hardware support teams.

At the Intermediate SQE level, the process for releasing an application to a user community must be defined and documented. The version number should be tied to the repository database to allow for simple retrieval of a particular code version and thereby enhance the ability to provide relevant user support. A documented announcement process should also accompany release to the user community.

At the Advanced SQE level, the release process can be strongly tied to the issues tracking database and regression test process, and documentation and test results can be automatically generated.

The advantage of managing releases is clear: No Chaos. All Releases are controlled in some fashion. Each release may have a specific purpose, such as porting, testing, etc.

3.2.4 Bug/Issue Tracking

Bug/Issue tracking is the identification and tracking of problems, change requests, and issues with the configurable items and associated corrective actions. It allows you to keep track of what errors and issues users or developers reported before and after release, to monitor the status of fixes to those errors, and to maintain information about resolution of defects. It provides a system of control; all information is in one place, team members can look up this information, and provides a place for people to enter information so that they don't "forget" that their most important user had an issue. Bug Tracking systems can allow you to prioritize defects, and enter information on the level of impact of the defect.

This practice is introduced at the Basic level and certainly well-established at the Intermediate level. Issues can be code defects (bugs), feature requests, problems encountered during testing, etc.

At the Intermediate SQE level, issues related to all configurable items (not just critical software elements) should be added to the tracking system. This system should be readily accessible to all developers, and where appropriate, to users.

At the Advanced SQE level, the issue tracking system shall be closely coupled to the version control system. It should provide capabilities to generate development reports that include lists of changes to controlled items, extent of such changes, and level of effort required. Metrics, at this level, assist in tracking and project management.

3.3 Software Testing

Software testing is an activity that occurs at all levels of software development. For the purposes of describing the ASCI SQE activities, it is included here in order to present a guide for developing the early stages of testing code and configurable items. In this requirements document, software testing refers to the subset of verification testing that is described below. It is not intended to be comprehensive physics verification and validation (V&V) testing.

Comprehensive verification (*testing that the code solves the equations correctly*) and validation (*testing the level of agreement with physical reality of code's equations*) are activities performed not only within code-development projects but also by Independent Verification and Validation. As identified in §2.2, each code project will coordinate the development of a V&V Plan specific to its Target SQE Level with the ASCI Independent V&V effort. The Independent V&V effort shall provide documented guidance for the development of each project's V&V plan. Program and line management shall approve each project's V&V plan.

Given the complexity of testing, at the Intermediate SQE level, a preliminary test plan shall be written. At the Advanced SQE level, the test plan shall be detailed and coordinated with the project's V&V plan and the independent V&V effort.

3.3.1 Unit Testing

Unit testing refers to testing various components of a software system. At the highest Levels, unit testing is also the activity of testing code units against their requirements, specifications, and design to verify that components work as designed and conform to the system requirements and functional specifications applicable to the components tested. The advantages of unit testing are increased confidence in the component and the removal of defects that are not easily found by inspection or in a debugging process.

3.3.2 System Testing

System testing (also called *Integration* Testing) is used to verify that the system works as designed and conforms to the overall system requirements (i.e. they work together correctly). The advantages of system testing are increased confidence that the system works as designed and specified, and that components work together correctly.

At the Basic SQE level, new system tests should be considered for integration into the regression test suite. The process for doing this integration should be defined and documented at the Intermediate SQE level. The process of integrating new tests into regression testing should be formalized and required at the Advanced SQE level.

3.3.3 Regression Testing

Regression testing refers to the repetitive execution of a test or a suite of tests. It is the activity of regularly building the code and executing a series of tests designed to verify that the code works as expected for all computational platforms supported. This activity includes the development and maintenance of a regression test suite. This test suite should be designed to exercise as many of the code features as possible. (However, it is beyond the scope of this document to prescribe how to accomplish this.) The regression test suite should include system and unit tests, as appropriate. Regression testing provides confidence that, as changes are made to code and as development proceeds, changes are not incompatible with past versions and results.

At the Basic SQE level, the initial regression testing process defined here includes performing well-defined tests of critical software components on a regular basis. The tests do not have to be numerous or automated at this fundamental level, but shall form the foundation for expanded testing at the higher levels. The regression test set should be designed to cover as many of the overall project requirements as practical. It is a good idea to begin the design and construction of a testing process and framework that it is extensible. At this level, build and environment tests (defined in §3.3.4) should be included.

At the higher SQE levels a complete set of regression tests should be developed to cover much of the software and many of the configurable items. These tests should include all elements of the software environment that are required to execute the software products under various circumstances. Furthermore, these tests should be rolled into an automated system which will execute the test, detect differences within some tolerance, and report results to the project leader and all software developers that "committed" changes to the software repository since the last regression test cycle. The process should be documented, and the entire system, including the tests, scripts, documentation, and anything else required shall be placed under configuration management. Analysis of the code-coverage of the regression tests is encouraged. At the Advanced SQE level, a continuous process should be defined in an effort to plug holes in the testing. Furthermore, a formal process for designing tests should be created so that no new software elements are added to the software without a designed regression test to go with them

3.3.4 Smoke Testing/Build and Environment Testing

This type of testing is the process of building and testing the complete code on a periodic basis. On large code projects it should be done on a regular, deliberate schedule; on smaller projects, it should be done whenever changes are made to the code or the runtime environment. In an environment where hardware, compilers and libraries are continually evolving, smoke tests may be required even if the code itself does not change. The purpose of smoke testing is to assure that the code still works as expected, and that no side effects have been introduced by code or environment changes. Typically, the smoke tests are a subset of the full regression test suite that can be automatically run on a regular schedule deemed appropriate for the project [13].

3.4 Critical Practices and Processes Documentation

Critical processes must be documented at the Basic SQE level. A list of required documentation is given in §2.2 and in Table 10 (Appendix B). The documentation should be updated as needed. If the software system is sufficiently complex, some consideration should also be given to the creation of tutorials or other similar "mentoring" programs to bring new people on the project team up to speed quickly on the process requirements.

When new developers come on board, the "way to do things" can be confusing. Documentation of processes such as building and releasing code helps these developers come up to speed, allowing them to become productive sooner and reduce the learning curve. The risk of experienced developers leaving your team and taking knowledge with them is also mitigated if the process is documented. Documentation of a process discourages creeping changes of the process that may be incorrect. From a program-management point of view, documentation allows program managers to be assured that you know what you are doing. However, keeping documentation up-to-date is not trivial, and frequently changing processes lead to out-of-date documentation.

As a project progresses towards the Intermediate SQE level, new processes and practices will be defined and introduced. These shall be documented and updated as needed.

At the Advanced SQE level, a complete set of documentation for the entire SQE practices and processes should exist and be under change control.

3.5 Risk Assessment and Management

Risk Assessment and Management is the set of activities related to identifying, analyzing, tracking, controlling, and mitigating sources of risk before they become threats to successful completion of a project. One breakdown of these activities is summarized in Table 2. The advantage of risk assessment and management is the early identification and prioritization of things that might affect the success or failure of the project.

At the Basic SQE level, an elementary risk assessment and analysis shall be done as specified in §2.1.1. Elementary risk assessment means that the software project leader, along with the code team and end users, identifies *critical software components* and the risks affecting them, analyze those risks, and prepare a plan to manage or mitigate them. These components may include basic physics packages or numerical algorithms, or important frameworks or back planes. These components shall be placed under configuration management. (See §3.2)

Informal discussions should also occur to ascertain other elements of risk to the software project. These risks can include reliance on fragile technology, single individuals, specialized build system tools, or other issues.

The risk-management process started at the Basic SQE level shall be further developed in the Intermediate SQE level to provide a more formal risk-mitigation plan and documentation of that process which also allows for tracking of risk. This process can be formulated using what was learned during the examination of critical software components, but should be general enough to apply to all components. The process should address new components as development begins as well as assessments and mitigation planning at regular intervals during the lifecycle of the software.

At the Advanced SQE level, a completion of the risk management process started in earlier levels shall be completed. At this level risks should be tracked and controlled. Communication of risks should be available at all levels with progressively increasing formality.

Table 2 The Six Aspects of Risk Management

Identify	Search for and locate risks before they become problems adversely affecting the project
Analyze	Process risk data into decision-making information
Plan	Translate risk information into decisions and actions (both present and future) and implement those actions
Track	Monitor the risk indicators and actions taken against risks
Control	Correct for deviations from planned risk actions
Communicate	Provide visibility and feedback data internal and external to your program on current and emerging risk activities

3.6 Software Reviews

Sub-teams within the project or the project lead should periodically review software work products: code, libraries, scripts, documentation as well as critical software components. Critical software components include those elements fundamental to the operation of the overall software and applications and the quality of physics and numerical algorithms in the applications. Reviews are an additional way to find defects in work products. Testing is one way of finding defects in code; reviews and assessments can be used to find defects in code that are obvious to the naked eye but hard to find with testing. Reviews also can be used to find defects in processes and work products that have not yet been coded. For example, reviews can detect designs that have errors, designs that do not meet all requirements, etc.

At the Basic SQE level, the reviews can be informal; no formal minutes are taken and no follow-up procedure is defined. Informal reviews can be walk-throughs by the project lead or project sub-team that meets to review developed software and comment on it, peer-reviews using people from outside the project, or even pair-programming (from Extreme Programming).

At the Intermediate SQE level, reviews are more structured. A structured work product review allows one to evaluate a process or product to find defects, omissions, and contradictions. Defect resolution is mandatory and rework is formally verified. At this level, a project shall define and document the structure of its review process. Defect data found during reviews shall be systematically collected and stored.

At the Advanced SQE level, the review process is completely formalized and includes formal documentation of reviews that are stored as part of the historical record of the development of the software. A template should be developed for these reviews that can be filled out by a recorder during the review. Results of the review should feed into other project management processes as appropriate. Metrics on the conformance to the project's defined processes and practices should be collected, analyzed and compared to ideal and historical measures. The metric data collected should also be added to the historical record.

3.7 Metrics

Each critical activity should be accompanied by appropriate metrics that can be used to assess its impact on burden and software quality and project management. Metrics are useful because they help software engineers and project leaders minimize the effort and schedule required to produce reliable software and they provide the opportunity to calibrate our practices across projects, thus leveraging collective experience.

At the Basic level the preliminary metrics of Appendix A shall be collected and provided to management, both at the project and program level. Explanations of the reported metric data should be added to the reports to management to aid in understanding the meaning and to reduce the potential for misunderstanding.

At the Intermediate level additional project specific metrics should be defined. All metrics shall be tracked over time. Project teams shall analyze the metrics to monitor the project status.

At the Advanced level the analyzed metrics are used as part of Integrated Software Management (§3.11) to improve the processes of the project, and as input into the program-level process development.

3.8 Standardized Coding Practices

At the Basic SQE level, there should be informal agreement between the team members on coding practices.

At the Intermediate SQE level, a coding practices document or a "style guide" for the project software shall be established. An example of a basic coding style guide is given in reference [5].

Each project shall develop and document their own coding practices at a detail level commensurate with the overall design and philosophy of the project source code. At a minimum, guidance on what code language(s) to use, which code language(s) to avoid, file and variable naming conventions, and comment-style guidelines should be provided. At the Advanced SQE level, the coding practices shall be documented in detail and include coding constructs and conventions that will and will not be used [7].

This document shall be used to formalize the project's software quality expectations. Therefore, it shall define the methods that will be used to determine and enforce compliance. For example, at the Basic SQE Level, an honor system may be sufficient. Alternatively, a visual inspection by another project member may be required to determine if work is within tolerances. At the Intermediate SQE Level, compliance to code and style standards shall be part of the formal code review/inspection process. At the Advanced SQE Level, automated scripts should be developed to perform most of the coding standards and style checking and correction.

Coding practices and style guidelines should not be so strict as to stifle individual style preferences. They should be flexible and avoid unnecessary, rigid constraints.

3.9 Training and Mentoring

Training and mentoring implies the collection of documentation into presentation form for training as well as the identification and training of mentors on project team in key areas to sit with and train staff. This prevents new people from wasting both their own time and the time of experienced team developers. It also allows them to receive accurate information up front. Training should not just be documentation. It can include training sessions that provide both information to new people, and allow current developers to gain valuable information about how hard the system is to use, to learn, etc.

Development of such a training program shall be started at the Basic SQE level and include training materials on all defined processes, practices and standards. Part of the process documentation effort for existing activities should be the creation of training materials. Training materials associated with new practices and processes should be developed as they are introduced. Training activities shall be refined and continuously improved for the Advanced SQE level.

3.10 General Software Documentation

At the Basic SQE Level, software documentation shall include a full Users Manual and Physics Manual that describe how to use the application software. This includes issues such as problem setup, mesh generation (if applicable), job submission, execution, results, output files, and so on. To the extent needed, the manuals should also be tailored to the site-specific runtime environments. This can be especially important on the ASCI computational platforms due to their high degree of complexity.

Full documentation of the software infrastructure should be generated to allow new project team members to become quickly productive in the development environment. This software development documentation should address all technical information required to enable anyone to develop a new software component for the project. At the Intermediate SQE Level, a Programmers manual shall be provided. Inherent in these documents will be a description of physics and numerical models (and their limitations) for most of the software development projects, and for others it may be hardware, operating systems, data manipulation tools, etc. Anyone outside the project should be able to pick this document up and understand what the software does, how it does it, and all of the technical details required to develop components for the software. To the extent needed, this document should also be tailored to the site-specific development environments.

3.11 Integrated Software Management

Integrated Software Management (ISM) is the process to improve the processes. ISM should exist at both the project-level and the program-level. At the project-level ISM can range from informal to a documented, systematic effort for process improvement. The specifics for each SQE level are discussed below.

Project SQE should not take place within a vacuum. The multiple projects within the ASCI program should not have to reinvent processes or process improvements that other projects have already developed. The only way to assure this sharing occurs is to have central coordination at the program-level ISM. As the projects start to implement the requirements of this document the program should start to collect, review, and share metrics, best practices, processes, and lessons-learned. Eventually, program-wide standard software development processes and program specific models for interpretation of metric data should emerge.

Project–level Integrated Software Management (ISM) is started in the Intermediate SQE level and continually improves through the Advanced SQE level. Program-level ISM can and should start immediately.

At the Basic SQE level, teams start collecting project management metrics such as those listed in Table 8 (Appendix A), distribute them internally and to higher levels of management. This collection of program-wide metric data is the first stage of program-level ISM.

At the Intermediate level, the project shall track the metric data over time and perform simple analyses of the metric data to monitor the project status. This is the first stage of project-level ISM.

At the Advanced SQE level there shall be a systematic effort to improve the processes of the project and to provide feedback on program-wide processes. Metric data should be analyzed to determine deviations from estimates. Further investigation should be undertaken to discover root causes of the deviations. Solutions to the discovered problems should then be incorporated into the project-level processes. Additionally, a project-level formal lessons-learned and best-practices improvement program shall be put in place for the collection, review, dissemination, and implementation of process improvements. Improvements to processes should be provided to the program-level ISM.

4 IMPLEMENTATION PLAN

4.1 Establishment of a Baseline for Planning Purposes

DoE Order 203.1 requires that current SQE processes and practices be documented for each project to provide a baseline for evaluation and planning. To accomplish this task, each code project will be asked to update its SQE practices questionnaire with respect to the practices described in this document. This will establish an ASCI wide "development practices" baseline as well as project specific "development practices" baselines. Note that each project could develop this information into a project-specific "development practices" baseline document, if desired. A revised set of questionnaires is provided for this purpose in Appendix C.

Table 3 summarized the responsibilities defined in this chapter at each organizational level for each phase of the path forward. The number codes used to designate the organizational levels in Table 3 are defined in Table 4. The action codes used in Table 4 are defined in Table 5.

4.2 Grading of ASCI Code Projects with respect to Goal for Level of SQE Practices

ASCI projects at LANL will, using the grading criteria, be assigned a grade of Basic, Intermediate, or Advanced. The Target Level is to be proposed by the project itself and approved (after review) by program and line management. See Table 3 for the actions required by the different management levels.

This level will be the Target Level of SQE Practices that each project should plan to strive to attain.

4.3 Achieve the Basic SQE Level

After the establishment of a project baseline of software development practices and processes, this information shall be used to evaluate what processes and practices specified as Basic are not being done by the project. If not already accomplished, the first priority at this point is to develop plans to achieve the Basic Level.

Each project shall examine the list of elements of the Basic Level, and first and most importantly, come to an understanding of what advantages each element would offer to their project. Each item on the Basic SQE level list was placed there because it offers clear advantages to software projects in general. If the project cannot discern the advantages or benefits of a required element, they shall consult with other teams and/or management to resolve whether there is benefit or if the element should be dropped from the list of required elements.

Next, each project should identify the change in current processes necessary to implement each item, and the resources necessary. Note here that each process can be done in many different ways, using tools or manual processes, in "light" fashion or heavily controlled fashion. The project needs to identify the most appropriate process for their project. Tasks required to implement each process should be identified, and time and people resources estimated. The project may identify that their team does not have the necessary individuals to accomplish the implementation of these processes.

Each item outstanding on the Basic List should then be prioritized based on perceived advantages, necessary resources, project risk, and time necessary.

The purpose of these plans is not only to allow the project to plan for upgraded development practices, but also to allow the ASCI management to monitor program wide the status of their implementations, as well as to provide appropriate additional resources if necessary.

4.4 Implement the Targeted Level of SQE Practices

A project's plan to achieve the "Targeted" Level of SQE should follow the same process described in the previous section: understand the advantages offered by each process, identify the change in current processes and the resources necessary to implement those processes, prioritize the order of implementation.

4.5 Reevaluate Target Level Plan

Progress towards achieving the target level should be monitored and frequently compared to the plan. There shall be an annual review of progress on the plan with a reevaluation of both the plan and assigned target level.

4.6 Processes Improvement

Continuous evaluation of process improvement is needed to help leadership at all levels to continue to direct its software engineering efforts. Each project shall formally evaluate the process improvement activities at least yearly. The evaluation must be measurement based to verify if the process improvement activities are effective or not. If an activity is proven to be ineffective, management must re-scope and direct their effort in another direction or manner. The results of the process improvement evaluation shall be reported to higher levels of management to allow for program wide aggregation and evaluation.

Table 3 Individual, Team and Organization responsibilities by phase of the path forward.

Phase	Individual/Team/Organization Responsible				
	1	2	3	4	5
Establish Baseline	I	R / A	R	R	E
Assign Target Level	I	A	R	C / A	E
Achieve Basic Level					
Plan	I	R	R	C / A	E
Allocate resources	I	R	E / R	E	E
Carry out plan	I	I	I	R	E
Monitor progress	I	I	E	E / R	E
Achieve Target Level					
Plan	I	R	R	C / A	E
Allocate resources	I	R	E / A	E	E
Carry out plan	I	I	E / R	E / R	E
Monitor progress	I	I	E	E / R	E
Reevaluate Target-Level Plan	I	A	E / R	C / A	E
Process Improvement	I	E / R	E / R	E / A	E

Table 4 Definitions of Individuals, Teams and Organizations used in Table 3.

Number	Individual/Team/Organization
1	Associate Laboratory Director for Weapons Physics
2	Deputy Associate Laboratory Director for ASCI
3	Division Level (DL, DDL or authorized delegate)
4	Group Level (GL, DGL or authorized delagate)
5	Project Level (PL and project team)

Table 5 Definitions of Action Codes used in Table 3.

Action Code	Description of Action
A = Approve	Approve the activity and deliverable. This action requires a formal signature for sign-off from the individual, team or organization.
E = Execute	Responsible for the work being performed to complete the deliverable. This action represents the individual, team or organization that will perform the work.
R = Review	Review the deliverable and provide feedback.
C = Consult	Individual, team or organization that is consulted for information pertaining to the work or deliverable. This individual, team or organization may not be responsible for providing this information but should be consulted before any action is taken.
I = Inform	Individual, team or organization is notified of an event for their information.

REFERENCES AND SUGGESTED READING

This section contains the numbered references that appear throughout this document.

1. Patty Trellue, et al., **Tri-Lab Survey: Software Engineering Practices - Software Process Assessment Laboratory Assessment Report (U)**, unpublished (May 26, 1999) Official Use Only [LANL ASCI Code Projects].
   Dwayne Knirk, et al., **Tri-Lab Survey ASCI Software Engineering Practices Final Report**, unpublished (August 9, 1999) [Overall summary report to DoE]

2. Ken Koch, **LANL V&V ASCI & Legacy Code Software Engineering Practices Questionnaire and Responses**, unpublished (2000).

3. Ann Hodges, et al., **ASCI Software Quality Engineering: Goals, Principles and Guidelines**, U. S. Department of Energy Document DoE/DP/ASC-SQE-2000-FDRFT-VERS2 (February 2001).

4. **U. S. Department of Energy Notice 203.1 Software Quality Assurance**, (October 2, 2000).

5. [LJCox-1] Cox, L., **Standards for Writing and Documenting Fortran 90 Code**, X–5–RN(U) –00–28, 2000

6. LIR 230-01-02.0 and LIR 300-00-01.2 and LIR 230-04-01.1

7. Les Hatton, **Safer C**, McGraw-Hill, London, 1994

8. Software Quality Assurance Subcommittee of the Nuclear Weapons Complex Quality Managers, **Guidelines for Software Measurement**, United States Department of Energy, Albuquerque Operations Office, Quality Report SQAS97-001, April 1997.

9. Anita D. Carleton, Robert E. Park, Wolfhart B. Goethert, William A. Florac, Elizabeth K. Bailey, Shari Lawrence Pfleeger, **Software Measurement for DoD Systems: Recommendations for Initial Core Measures**, Technical Report, CMU/SEI-92-TR-19, ESC-TR-92-019, Carnegie Mellon University, September 1992.

10. Goethert et al., **Software Effort & Schedule Measurement: A Framework for Counting Staff hours and Reporting Schedule Information**, Technical Report CMU/SEI-92-TR-21, Software Engineering Institute, Pittsburgh, PA, July 1992. See http://www.sei.cmu.edu/pub/documents/92.reports/pdf/tr21.92.pdf.

11. William A Florac, **Software Quality Measurement: A Framework for Counting Problems and Defects**, Technical Report CMU/SEI-92-TR-22, Software Engineering Institute, Pittsburgh, PA, September 1992. See http://www.sei.cmu.edu/pub/documents/92.reports/pdf/tr22.92.pdf.

12. Software Engineering Laboratory Series, **Manager's Handbook for Software Development, Revision 1**, SEL-84-101, NASA, Goddard Space Flight Center, Greenbelt, MD, November 1990. See http://sel.gsfc.nasa.gov/website/documents/docs/84-101.pdf.

13. Steve McConnell, **Software Project Survival Guide** , Microsoft Press, Redmond, WA, 1998.

14. Dr. Victor R. Basili, **Software Measurement Implementation and Practice**, LANL Training Class, August 2000.

15. **IEEE Standard for Software Reviews,** IEEE Std 1028-1997. 9 December 1997.

Appendix A Program Metrics

1 Introduction

The purpose of this appendix is to lay out a small set of standard measures and derived metrics that all codes, regardless of level, can easily collect and calculate. A common set of metrics allows ASCI program management to have a consistent view of all the projects.

Items that can directly counted or collected are measures. Examples include hours of effort, the number lines of code, etc. Measures are facts and usually cannot directly be used to pass judgment on a process.

Metrics are usually derived quantities that show how measures change over time. Examples include source code growth rate, defect density, etc. Interpreting metrics requires a model to compare them to. For example, is a particular source code growth rate too fast or too slow? The answer depends on the model. The definition of standard metrics will allow the collection of enough data to develop LANL ASCI models.

This appendix defines common measures and metrics. It does not define models because to be accurate they should be built from historical data and documented experience of a particular organization.

2 Measures

The DoE Software Quality Assurance Subcommittee of the Nuclear Weapons Complex Quality Managers [8] adopted the measures in Table 6.

Table 6 Measures adopted by the DoE-SQAS

Type of Measure	Examples of Measure Units	Characteristic Addressed
Size	Counts of physical source lines of code (SLOC)[1]	Size, progress, reuse
Effort	Counts of staff hours expended	Effort, cost, resource utilization, rework
Progress to Schedule	Calendar dates (events/milestones)	Schedule, progress
Defects	Counts of software problems & defects	Quality, acceptability for delivery, improvement trends, user satisfaction

These measures are identical to those that the Software Engineering Institute (SEI) recommended for the Department of Defense (DoD) [9]. Each measure is discussed in more detail below.

[1] The alternative measure of function points has been omitted because they count input/output operations and have little proven relevance to numerically intensive software.

2.1. Size

The DoE-SQAS recommendation is to:

> Adopt physical source lines of code (non-comment, non-blank source statements) as the measure of software size.

This does not preclude projects from using more complicated measures, however source lines of code (SLOC) shall always be collected so there is a consistent measure across projects. Notice that the definition of SLOC is non-comment, non-blank source statements.

The reasons given by the DoE-SQAS for using SLOC are given below:

1. SLOC is easy to measure; measurements are made by counting end-of-statement markers, filtering out the blank and comment lines.
2. Counting methods strongly depend on the programming language used. One need only to specify how to recognize statement types not counted (e.g., comments, blank lines). Automated counters for physical SLOC measures are available.
3. Most historical data for constructing the cost models used for project estimating are based on measures of source code size.
4. Empirical evidence to date suggests that counting physical SLOC is generally as effective as any other existing size measure.

2.2. Effort

The DoE-SQAS recommendation is to:

> Adopt staff hours as the principal measure of effort.

The reasons to use staff hours are:

1. No standard exists for the number of hours in a labor month. Practices vary widely across projects
2. Staff months often do not provide the granularity needed for measuring and tracking individual activities and processes, particularly when the focus is on process improvement.
3. Measuring effort by staff weeks or staff days presents many of the same problems as staff months, as well as additional ones. E.g.: the length of a workday, overtime and weekend work, the number of days in a work week, holidays and vacation.
4. Staff month, staff week, and staff day measures can be calculated from staff hours should these measures be needed for presentations or other summaries.

The measurement of effort can be done using various granularities. Most common is to measure total effort, which is used for project management purposes. A more granular approach is to measure effort for specific tasks such as requirements development, defect correction, etc. No matter what granularity is used, the unit of measure shall be staff hours.

2.3. Progress to Schedule

The DoE-SQAS recommendation is to report:

> Dates (both planned and actual) associated with project milestones, reviews, audits, and deliverables; and exit or completion criteria associated with each date.

Schedules are one measure that can also serve as a metric; this will be revisited in the metrics section. Although not required herein, the DoE-SQAS recommends that projects adopt a structured method such as the one recommended by SEI [10] to define the schedules.

2.4. Defects

The DoE-SQAS recommendation is that:

> Counts of software problems and defects and rework time should be used to help plan and track development and support of software systems.

Unfortunately the DoE-SQAS does not define "defect". The SEI [11] defines a software defect very broadly as:

Any flaw or imperfection in a software work product or software process.

The types of defects SEI recommends to count are:

- Requirements defect
- Design defect
- Code defect
- Operational document defect

The types of defect SEI recommends to not count are:

- Test case defects
- Other work product defects
- New requirements
- Hardware defects
- Operating system problems
- User mistakes
- Operations mistakes
- Not repeatable / cause unknown

Another question about defects is when do they exist? The short answer is that any problem that is found in a work product that has been turned over for testing is a defect. The activities that SEI believes can result in defect reports are:

- Product synthesis
- Inspections
- Formal reviews
- Testing
- Customer service

Product synthesis is the defined as the activities of planning, creating, and documenting the requirements, design, code, and documentation that constitutes a software product. An example is finding a requirement error when working on the design of a module.

To make the above list more concrete, the following are examples of activities that result in defects when problems are found:

- A requirements document is reviewed;
- A module is in detailed design and a requirement problem is found;
- A piece of code is inspected;
- A piece of code is released for testing;
- A code is publicly released.

Defects can be partitioned, thus counted, in many ways. Examples are origin (requirements, design, etc.), type (logical omission, control structure, routine interfaces, etc.), when it was found (review, testing, users, etc.), and so on. How to partition the defects depends on the metrics of interest. Recommendations will be given in the metrics section.

3 Metrics

The goal of project management metrics is to provide a clear view of the status of projects. This section will define basic metrics. Projects are encouraged to use more detailed metrics internally if they are helpful to the project.

3.1. Metrics Defined

Table 7 lists the project management metrics that are used by the NASA Software Engineering Laboratory [12]. An exception is "schedule" which was added to the list. Also shown are some of the questions that the metrics can help answer.

Table 7 Metrics and questions they answer.

Metric	*Questions*
System size estimate	How large is the system? Is the system size estimate reliable?
Source code growth rate	Is the actual size converging to the estimated size? Does the growth rate reflect the project's lifecycle?
Effort expended	Does the actual effort expended approximate the budgeted effort?
Schedule	Are the milestones, etc. stable? Are the milestones, etc. being meet?
Defect density	Is the defect density rising or falling? Does the defect density reflect the lifecycle of the project?
Defect status	Are defects being repaired faster than they are being found? Does the defect status reflect the lifecycle of the project?

These questions are looking for abnormalities that flag a project in trouble. For example, suppose a project's integration testing has reached its scheduled completion time, but both the defect density and the defect status are growing. In this case more integration testing is clearly called for.

Definitions of the metrics and how to display them are shown in Table 8.

System size estimates will vary with each project. The point is to provide a common metric that all projects can provide to the program office. Defect density and defect status are metrics that should continue to be collected on codes that have been released to the users. The after release results can give a good look at the actual quality of the released code.

Table 8 Project management metrics defined.

Metric	Collect	Display As
System size estimates	Estimated number of lines of code in the completed system.	Estimated number of thousand lines of code versus percentage project time.
Source code growth rate	Number lines code under configuration control.	Number of thousand lines of code versus percentage project time. Display on same plot as the system size estimate.
Effort expended.	Total effort expended to date in staff hours.	Percent of total effort expended per budgeted effort versus percentage of project time.
Schedule	Planned and actual completion dates of program milestones, project milestones, reviews, etc.	Table with planned and actual dates, and exit criteria for program milestones, project milestones, reviews, audits, and deliverables.
Defect density	Number of defects found in configuration-controlled code.	Total number of defects per thousand lines of actual code versus percentage project time.
Defect status	Number defects closed each week.	On a single plot that has percentage project time as x-axis, show total number of defects, total number of closed defects, and total number of open defects.

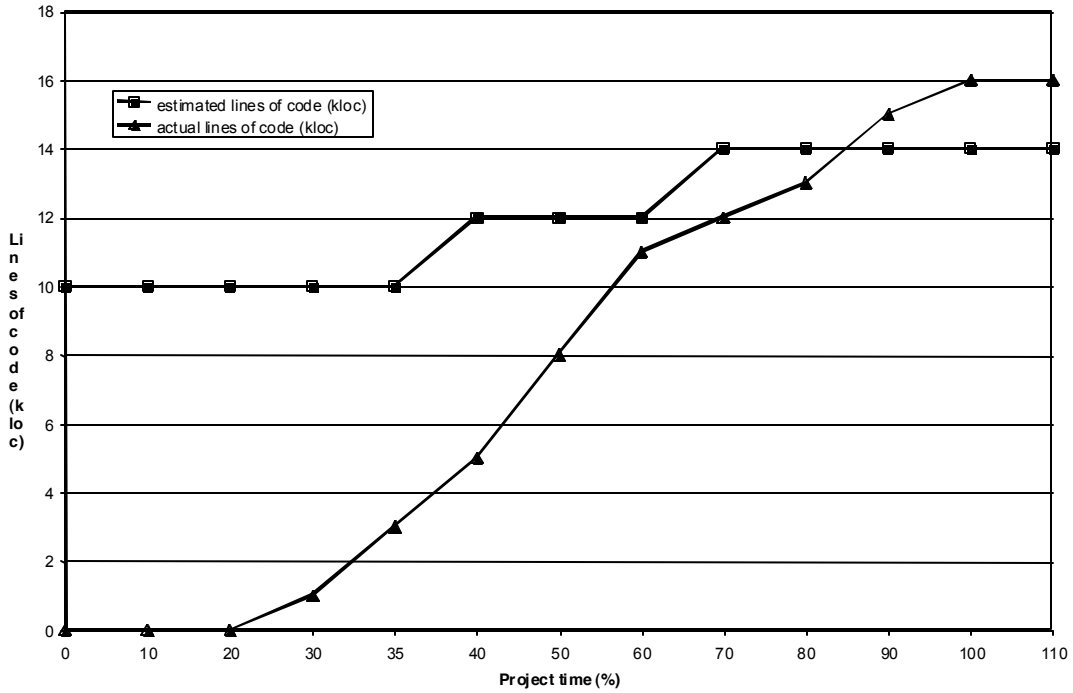
3.2. Examples

The examples show how the metrics are implemented and provides some guidance on interpreting the metrics.

Examples of the *source code growth rate* and *system size estimates* are shown in Figure 2. The estimated lines of code changed twice, at the delivery of the stage one and stage two releases. Coding did not start until about 20% into the project, prior to that were planning and

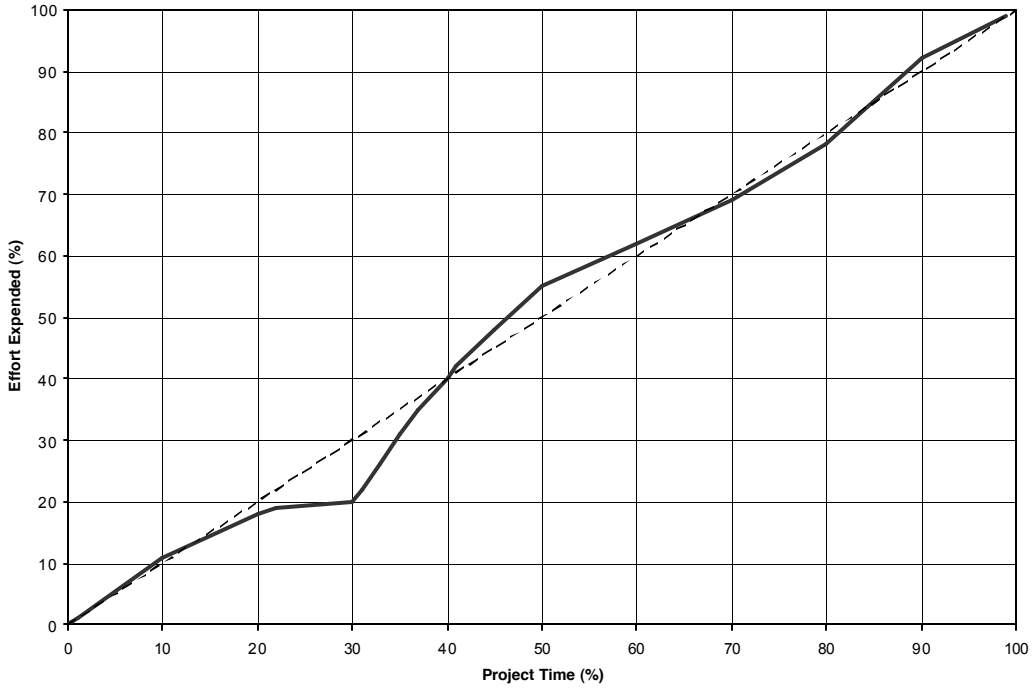
requirements gathering. The code grew steadily until the delivery of the stage-two release. By that point defects had become so prolific that testing took much of the effort and code growth slowed. This was followed by a burst of code growth, and then stability was achieved as final testing was conducted. Note that the project was delivered late at 110% of project time.

Figure 2. Example of source code growth rate and system size estimate.



An example of the *effort-expended* metric is shown in Figure 3. Initial planning actually took a little more than the projected effort (the dashed line). Requirements gathering—meeting with customers—was a slow process so the effort expended essentially ceased, bringing the effort expended below the projected effort. Requirements gathering completed at about 30% of the project time and code construction started. Construction quickly drove the expended effort back above the average level and then started tapering during integration and integration testing. The last surge of effort was around 85% of project time as final release preparations occurred.

Figure 3. Example of Effort Expended metric.



An example of the beginning portion of a *schedule* metric is shown in Table 9. The metric is the planned versus actual dates for starting and finishing the project milestones, reviews, audits, and deliverables. The criterion that determines when an event is completed is also shown.

Table 9 Example of Schedule metric.

Task Name	Planned Start Date	Actual Start Date	Planned Finish Date	Actual Finish Date	Exit Criteria
Develop functional requirements list	5/1/01	5/1/01	5/25/01	5/24/01	Preliminary capabilities document completed.
Develop User Manual	5/29/01	5/29/01	7/2/01	6/29/01	Preliminary User's Manual completed.
Internal requirements review	7/3/01	7/2/01	7/5/01	7/3/01	Review passed.
Revise requirements	7/6/01	7/4/01	7/11/01	7/5/01	Review recommendations implemented
Develop project plan	5/14/01	5/14/01	6/11/01	6/12/01	Preliminary Project plan completed.
Checkpoint review	7/12/01	7/10/01	7/12/01	7/10/01	Review gives "go ahead" on project.

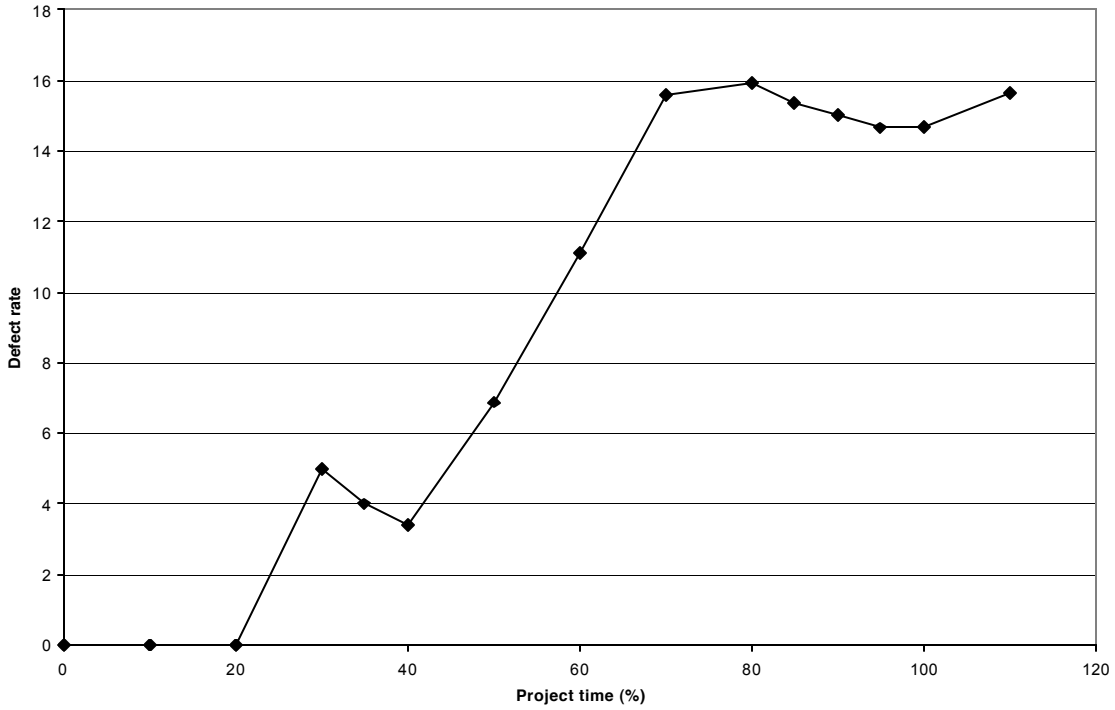


Figure 4. Example of defect density.

Example of the *defect density* is shown in Figure 4. At early times no code has been written but defects have been found in the requirements and design. Technically this means the defect density is infinite, however it has been truncated to zero because that portion of the curve is not interesting.

After code starts being written the defect density quickly jumps to about 5 per 1000 lines of code. The defect density then begins to drop, as the code is rapidly developed and testing falls behind. Around 40% of project time the testing program starts to steadily find defects and the defect density increases to about 16 per 1000 lines of code. Another burst of rapid code development results in the defect density going down as 100% of project time approaches. However, drawing on the earlier experience management delays release for another round of testing. The testing drives the defect density back to about 15 per 1000 lines of code.

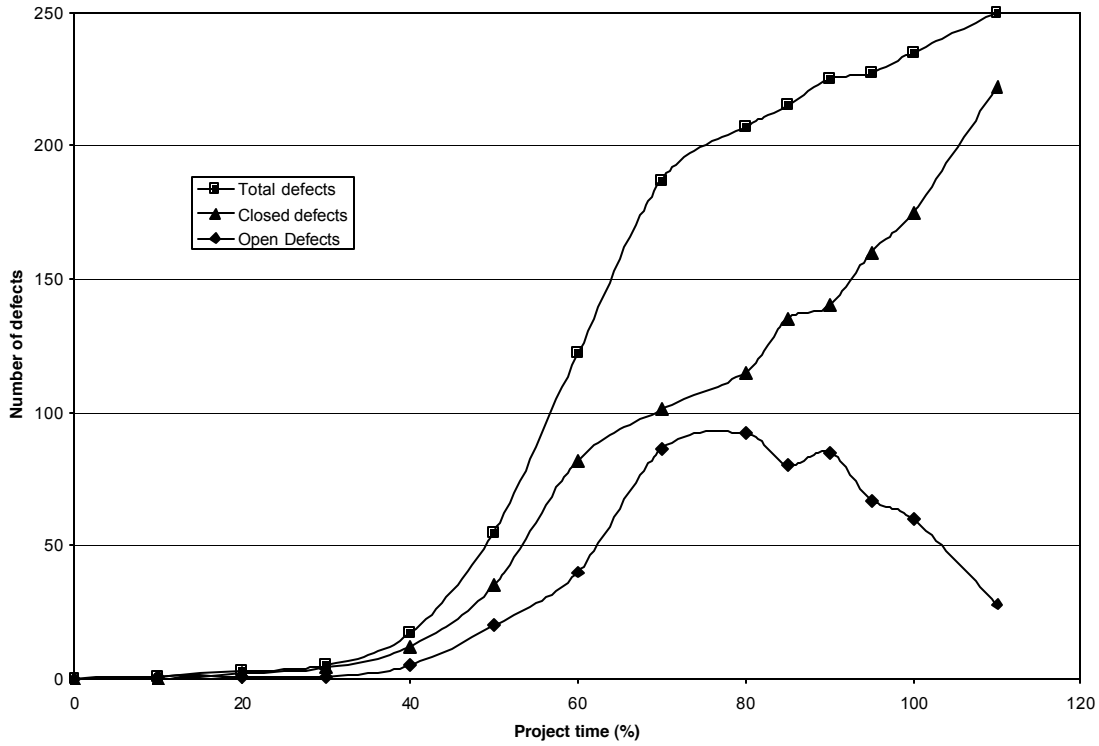


Figure 5. Example of defect status.

The *defect status* metric is closely tied to the *defect density* metric. An example of the *defect status* is shown in Figure 5. As expected the total number of defects steadily grows during the project. It grows fastest during the portion of the project with the fastest growth, and then slows as the project concentrates on fixing defects. By about 70% of project time the project is in crisis with a large number of open defects. The rest of the project is essentially spent in reducing the number of open defects to an acceptable level.

4 Further Reading

The required metrics in this section are very general and are intended for program management. However, some projects desire more detailed metrics that can help with project management, process improvement, or quality assurance. The following items are recommended for further reading.

The NASA Software Engineering Laboratory (SEL) Manager's Handbook [12] provides a good introduction to metrics and gives a list of possible metrics.

Time accounting can be looked at in many ways. McConnell suggests a comprehensive list of time-accounting categories. From these categories metrics can be developed such as effort expended per category. For a more detailed look at time-accounting metrics see the SEI report [10].

Defects can also be looked at in many ways. McConnell [13] suggests the information that should be on a defect report. With this information it is possible to design metrics that determine:

- When defects were created: requirements, architecture, design, construction, etc.;
- Effort required debugging and correcting defects;
- Which work products or routines are generating defects.

Victor Basili [14], a measurement expert, suggests also collecting the type of defect:

- Logical omission – a logical case is omitted (e.g. left out an error condition)
- Logical commission – a logical error was committed (e.g. input routine expects input records in the wrong order)
- Interface – routine interface errors (e.g. passed wrong type of variable)
- Initialization – routine initialization incorrect (e.g. counter variable not initialized to zero)
- Data – static data incorrect (e.g. value of pi wrong)
- Control structure – control structures used incorrectly (e.g. one off error in loop)

For more detail on defects and related metrics see the SEI report [11].

Appendix B Audit Items

The IEEE Standard for Software Reviews [15] gives the following introduction to audits.

> The purpose of a software audit is to provide an independent evaluation of conformance of software products and processes to applicable regulations, standards, guidelines, plans, and procedures.

The IEEE document lists 32 items that are subject to audit. Only a subset of the IEEE items is subject to audit by the ASCI program.

The basic philosophy is that audits can only check conformance to processes that have been documented. A mapping of the items subject to audit at each SQE Level is shown in Table 10. Also shown are the section numbers that first call out the use of the item at a particular SQE level and the description of the item.

Table 10 Items subject to audit at each SQE Level.

Sections	Item	Basic SQE	Intermediate SQE	Advanced SQE
2.2.1, 3.1	Requirements and Functional Specifications Document	✓	✓	✓
2.2.1, 3.2	Source Code	✓	✓	✓
2.2.2, 3.2	Software Configuration Management Plan	--	✓	✓
2.2.1, 3.2.1	Build Environment tools and plan	✓	✓	✓
2.2.1, 3.2.2	Version Control repository and tools	✓	✓	✓
2.2.2, 3.2.2	Version Control: plan in SCMP	--	✓	✓
2.2.1, 3.2.3	Release Process	✓	✓	✓
2.2.2, 3.2	Software Change Control Process	--	✓	✓
2.2.3, 3.2.4	Issue Tracking: plan in SCMP	--	--	✓
2.2.1, 3.2.4	Issue Tracking: tools / database	✓	✓	✓
2.2.1, 3.3	Test suites / tools	✓	✓	✓
2.2.2, 3.3	Software Test Plan	--	✓	✓
2.2.3, 3.3	Physics Verification and Validation Plan	--	--	✓
2.2.1, 3.4	Critical Processes Documented	✓	✓	✓

Sections	Item	Basic SQE	Intermediate SQE	Advanced SQE
2.2.1, 3.5	Risk Management Plan	--	✓	✓
2.2.3, 3.6	Software Reviews Documents	--	--	✓
3.2.1, 3.7	Metrics	✓	✓	✓
3.2.1, 4.9	Coding Practices Document	--	✓	✓
2.2.1, 3.9	Training Materials and Records	✓	✓	✓
2.2.3, 3.11	Integrated Software Management Plan	--	--	✓
2.2.1, 3.10	User's Manual	✓	✓	✓
2.2.2, 3.10	Programmers Manual	--	✓	✓

Appendix C Self-assessment Questionnaire for Best Practices

This appendix contains a set of questionnaires that are intended to guide a project in self-assessment of its status with respect to the requirements defined in this document. Most questions have a letter following them: B, I, A. These letters stand for Basic, Intermediate or Advanced respectively and reflect the SQE Level at which the topic should be addressed.

1 Requirements Management Questionnaire

Note: in this questionnaire the term "requirements" included both requirements and functional specifications as defined in the main document.

Capturing Requirements

1. Are your customers and users identified? (B)
2. Have overall project requirements been negotiated with Division/Program management? (B)
    a. Do developers and users agree to *detailed* requirements before development begins? (A)
3. Do users have input into the requirements? (B)
4. Do developers and users agree to basic requirements before development begins? (B)
5. Are requirements documented? (B)
6. Are requirements documented before development begins? (B)
7. Are requirements used as the basis for planning and managing the project? (B)

Controlling Requirements

1. Are requirements under configuration management? (A)
2. Are changes to requirements agreed to by developers and users? (I)
    a. Are changes to requirements documented? (I)
    b. Are changes to requirements placed under configuration management? (A)
    c. As requirements change, are adjustments made to software plans, work products, and activities? (B)
    d. Is there a process in place to evaluate the affect of requested changes on the project? (I)
3. Are requirements peer reviewed? (I)
4. Is there a training program in place for personnel involved with managing requirements? (I)

Tracking Requirements

1. Are software requirements tracked to be sure all are satisfied? (A)
2. Is each software component traceable to a requirement? (A)
3. Are requirements used to define tests? (A)
4. Are test results reviewed to confirm the product compiles with the requirements? (A)

2 Software Configuration Management Questionnaire

1. Are Configuration Items identified?
    a. Critical Source Code (B)
    b. All Source Code (I)
    c. Test problems (I)
    d. Documentation (I)
    e. Project-specific build scripts (B)
    f. Data (I)
    g. Software tools (test/image processors, scripts, system utilities) (I)
    h. Test results (I)
2. Does a release identify what versions of the following are used? (A)
    a. Operating Systems
    b. Libraries
    c. Compilers
    d. Hardware
3. Does the project's configuration management system provide for disaster recovery? (A)
    a. Of critical software components? (I)
4. Is the foundation of the project's build system based on common tools (i.e., GMAKE), or is it based on locally developed or maintained tools (i.e., MCNP's PRPR)? (B)
    a. If locally developed and maintained, do the benefits outweigh the costs? (B)
    b. Does the foundation of the build system allow for organizational support? (B)
5. Is the build system and code base portable to all supported platforms? (I)
    a. Does the build system construct all applications with all required compiler and library options? (I)
    b. Is the build system incorporated into the regression testing process? (B)
    c. Does the build system make use of:
        i. self-constructing builds? (A)
        ii. automatic metric collection? (A)
6. Is the project's process for version control documented and clearly defined? (B)
    a. Is all the documentation gathered together into an SCMP? (I)
    b. Does the SCMP describe the change control process? (A)
7. Does the project's release system allow for the reinstallation of past releases? (B)
    a. Is the release process defined and documented? (I)
    b. Is the numbering of configurable items ties to your repository database? (I)
8. Is there a documented release announcement process? (I)
    a. Is the release process strongly tied to issue tracking? (A)
    b. Is the release process strongly tied to the regression test processes? (A)
    c. Are documentation and test results automatically generated? (A)
9. Are bugs/issues identified, documented and tracked for critical software components? (B)
    a. For all software? (I)
    b. Is your issue tracking system easily accessible for developers and users? (I)
    c. Is your issue tracking system closely coupled to the version control system? (A)
    d. Are reports easily generated and include changes, extent of changes and effort required? (A)

3 Software Testing Questionnaire

1. Are testing procedures and testing activities planned and documented? (I)
    a. How is the documentation distributed? (I)
    b. Provide a reference, hard copy or web address. (I)
    c. Is the documentation available to other teams and/or higher levels of management? (I)
    d. Is the documentation made available to the users of the project's codes?
2. Does the project team conduct regular meeting to identify areas for improving testing? (B)
3. Once a testing deficiency is identified, are these issues documented, addressed and eliminated? (A)
4. Does the project follow a written organizational policy for testing? (A)
5. Are the team members responsible for testing trained in this area? (I)
    a. When and how is new staff trained in this area? (I)
    b. Are team members provided with refresher training when the testing system changes? (A)
6. Are measurements used to determine the status of testing activities? (A)
7. Are the activities and work products subject to review and assessment? (A)
8. Are the code-input problems standardized? (A)
    a. Are they adapted to execute on multiple platforms? (A)
9. Are test results produced in a common format for comparison with other codes? (I)

Unit Testing

1. If your project has a software test plan, does it include unit testing? (I)
2. Does each functional specification that applies to a software component have its own test case? (A)
3. Are all defined/used data flow paths tested with at least one test case? (A)
4. Has the code been checked for data flow patterns that are unlikely to be correct? (I)
5. What about execution from restart dumps? (I)
6. Is a list of common errors been used to develop test cases for detecting errors that have occurred frequently in the past? (I)
7. Are all boundary conditions been tested? (max, min, off by one,) (A)
8. Do test cases check for incorrect data format? (B)
9. Is compatibility with old data tested? (B)
10. Are old versions of operating systems, hardware, software, libraries, and interfaces tested? (I)
11. Are defects documented and tracked? (See also Issue Tracking) (B)
12. Do you use any tools for testing, test coverage, code complexity analysis, or debugging? And are the issues and information generated from these tools documented. (I) If so, please list the tools uses and identify commercial versus custom tools.

13. What tests are performed in comparison to analytic or exact solutions? (I)
    a. Do the results converge? (I)

System Testing

1. If your project has a software test plan, does it include system/integration testing? (A)
2. Are the interfaces between components been specified well and in advance? (I)
3. Is there at least one test case for each interface? (A)
4. Do you incorporate your integration test cases with your regression test suite? (I)
5. What tests are performed in order to compare with experimental data (A)

Regression Testing

1. If you project has a software test plan, does it include regression testing? (I)
2. Is the percentage of the code covered by the tests calculated or measured? (I)
    a. If so, what is the coverage percentage?
3. Are your regression tests automated? (I)
    a. Is documentation of the results automatically generated? (A)
4. Are problems with analytic or exact solutions included in your regression suite? (I)
5. Are problems with results that can be compared with experimental data included in your regression suite? (A)

Smoke Testing/Build and Environment Testing

1. If you project has a software test plan, does it include smoke testing? (I)
    a. Is there a defined schedule for the smoke tests? (I)
    b. Is the smoke testing automated? (I)
    c. Does it include Environment testing? (I)
2. How is a compiling and execution environment standard enforced?
    a. Honor system (self-policing); (B)
    b. Visual inspection for compliance by another team member; (B)
    c. Detailed, formal inspection for compliance during formal code reviews; (I)
    d. Script-based compliance. (A)
3. What kind of static analysis is performed on your software? (I)
4. What kind of performance analysis is conducted? (A)

Verification and Validation

1. Are your software testing activities coordinated with the Independent V&V effort? (I)

4 Critical Practices and Processes Documentation Questionnaire

1. See Table 10 in Appendix B for a list of documentation that may be subject to audit at the different SQE levels. For existing documents, provide a reference, hard copy or web address.
2. Are critical processes and practices documented so that the loss of senior staff would not result in irreparable harm to the project? (B)
    a. Are practices and processes tailored to suit senior staff inflexibility?
3. Have critical practices and processes been identified? (B)
    a. List the identified critical processes and practices. (B)
        i. Requirements standards; (B)
        ii. Build system and environment; (B)
        iii. Version control processes and practices; (B)
        iv. Release process; (B)
        v. Issue Tracking process/tools; (B)
        vi. Test processes and practices; (B)
        vii. Others (list).
    b. Is each practice and process so identified been documented? (B)
    c. Is the documentation kept up-to-date as practices or processes evolve? (B)
    d. Is the documentation detailed enough to be used by a new staff member without additional guidance? (I)
    e. Is the documentation level appropriate for the maturity of the described practice or process (not *too* detailed)? (B)
    f. Does the project have tutorial materials for some or all of these practices and processes? (I)
4. As new practices and processes are introduced, are they documented and reviewed by the project team? (B)
5. Is the documentation of practices and processes current, complete and under configuration management? (A)
6. Are the effects of change in processes and practices tracked, analyzed and used to improve the practices and processes? (A)

5 Risk Assessment and Management Questionnaire

1. Does the project have a risk assessment? (B)
    a. Are the team members and end users involved in developing it? (B)
    b. Are the team members preparing the assessment and mitigation trained in this area? (B)
2. Is the risk assessment documented and readily available to team member and end-users? (B)
    a. How is the documentation distributed? Provide a reference, hard copy, or web address. (B)
    b. Is it available to other teams and/or higher levels of management? (B)
3. Does the risk assessment identify critical software components? (B)
    a. Does it identify the risks affecting them? (B)
    b. Does it analyze the possible effects of these risks?(B)
    c. Does it document methods to manage and/or mitigate these risks? (I)
    d. Does it identify other software components? (I)
    e. Are new components included as their development begins? (I)
4. Is this risk assessment prepared in a formal plan or formally addressed as a part of another plan? (A)
    a. Is this documented and readily available to team member and end users? (A)
    b. How is the documentation distributed? Provide a reference, hard copy, or web address. (A)
    c. Is it available to other teams and/or higher levels of management? (A)
5. Is the risk assessment updated periodically? (I)
    a. Are the team members and end users involved in the update? (I)
    b. Is the updated assessment documented and readily available to team member and end users? (I)
    c. How is it distributed? Provide a reference, hard copy, or web address. (I)
    d. Is it available to other teams and/or higher levels of management? (I)
6. Are the risk indicators regularly tracked and actions taken against risks? (I)
    a. Is this documented and readily available to team member and end users? (I)
    b. How is the documentation distributed? Provide a reference, hard copy, or web address. (I)
    c. Is it available to other teams and/or higher levels of management? (I)
7. Are the risks and planned actions under configuration control? (A)
    a. Is this documented and readily available to team member and end users? (A)
    b. How is the documentation distributed? Provide a reference, hard copy, or web address. (A)
    c. Is it available to other teams and/or higher levels of management? (A)

6 Software Reviews Questionnaire

1. Is the project/organization's software process assessed periodically? (I)
2. Are your software reviews conducted by peers or by independents? (I)
3. Is the software review process documented? (A)
    a. How is the documentation distributed? (A)
    b. Provide a reference, hard copy or web address. (A)
4. Are team members provided with feedback on their work with respect to a software review? (A)

7 Metrics Questionnaire

1. Are the basic metrics defined in Appendix A regularly collected? (B)
2. Are project specific metrics defined and regularly collected for critical processes? (I)
3. Are metrics regularly analyzed and feedback into the project management? (I)
4. Are metrics analyzed and used as feedback to improve the project processes? (A)
5. Are metrics and analysis provided as feedback into Integrated Software Management? (A)

8 Standardized Coding Practices Questionnaire

1. Does the project have a set of standard coding practices and/or a style guide? (B)
2. Are the coding practices and/or style guidelines documented and readily available to the members of the software project team? (I)
    a. How is the documentation distributed? (I)
    b. Provide a reference, hard copy or web address. (I)
    c. Is the documentation available to other teams and/or higher levels of management? (I)
    d. Is the documentation made available to the users of the project's codes? (I)
    e. Are the coding practices detailed including language constraints? (A)
3. Do the project's standards/styles seem overly rigid or inflexible to the project leader, the team members or any others consulted? (B)
    a. Are any of the standards/styles tailored to suit senior staff inflexibility?
4. Is the level of detail commensurate with the overall design and philosophy of the software project? (B)
5. Are the code languages that are to be used specified? (B)
    a. Are any code languages explicitly forbidden or discouraged? (B)
    b. Are specific features or constructs of the languages promoted/discouraged? (I)
6. Does the documentation define conventions for naming files and variables? (B)
7. Are guidelines provided for code commentary with respect to style and verbosity? (B)
8. How are these coding practices and style guidelines enforced?
    a. Honor system (self-policing); (B)
    b. Visual inspection for compliance by another team member; (B)
    c. Detailed, formal inspection for compliance during formal code reviews; (I)
    d. Script-based compliance checking and correction in addition to (c); (A)
    e. Other. (Describe)
9. Has the project team been trained in the specifics of the coding standards and/or style guide? (B)
    a. When and how is new staff trained in this area? (I)
    b. Are team member provided with refresher training when the coding standards or style guidelines change? (I)
10. Are team members provided with feedback on their work with respect to compliance with the project's code standards and style guidelines? (A)

9 Training and Mentoring Questionnaire

1. Do project members receive the training/mentoring necessary to perform their duties? (B)
    a. Is staff trained on all defined processes, practices and standards? (B)
    b. Is there a training/mentoring program for new team members and members who are changing roles? (B)
    c. Are adequate resources provided to implement the project's training program? (B)
2. Are adequate training materials available for all aspects of the project? (B)
    a. Are these materials kept up to date? (B)
3. Does your project have a written training policy? (I)
    a. Are metrics used to evaluate and improve the training program? (A)
4. Are training activities continuously refined and improved? (A)
    a. Are metrics used to evaluate and improve the training program? (A)

10 General Software Documentation Questionnaire

1. Does your project have documentation for developers describing critical coding processes, practices and standards? (B)
2. Does your project have developer documentation that describes the development environment? (B)
    a. Is your documentation clear enough for new team members to become productive quickly? (I)
3. Are the runtime environments documented for each supported platform? (B)
4. Is the underlying structure of the software documented? (I)
5. Are the major software components and related interfaces clearly documented? (B)
6. Does each routine have associated documentation describing its purpose? (I)
7. Are all software data structures documented? (I)
8. Do you have a data dictionary? (A)
9. Are the formats of input and output files thoroughly documented? (I)
10. Is there a Software Users Manual? (B)
    a. Is the Users Manual updated in a timely fashion? (B)
    b. Does the Users Manual contain examples of code usage? (I)
    c. Does the Users Manual contain all technical information necessary to fully exercise the software on all supported platforms? (A)
    d. Are the formats of input and output files thoroughly documented? (B)
11. Is there a Software Users Tutorial? (A)
12. Is there a Programmers Manual? (I)
13. (If applicable) Do all physics routines have documentation describing the underlying physics models and numerical algorithms? (I)
    a. Is there documentation describing the limitations of all physics models and related numerical algorithms. (A)
    b. Is there documentation describing appropriate problem application domains? (A)

11 Integrated Software Management Questionnaire

Metrics & Measures

1. Are project management measures and metrics defined? (B)
    d. Is the data collected? (B)
    e. Are these measures and metrics tracked over time? (I)
    f. Is the data analyzed? (I)
    g. Are the results of the analyses reported to Division/Program management? (A)
2. Are the trends in measure and metric data analyzed? (A)
    h. Are the trends reported to Division/Program management? (A)
    i. Are the trends used in making project estimates? (A)
    j. Are the trends used in revising project estimates? (A)
3. Are deviations from early project estimates analyzed? (A)
    k. Are deviations reported to Division/Program management? (A)
4. Are the measures, metrics and trends used in project performance evaluations? (A)

Lessons Learned

1. Are examples of best practices and lessons-learned collected? (I)
    a. Are these best practices and lessons-learned reviewed and analyzed? (I)
    b. Are these analyzed best practices and lessons learned shared? (A)
2. Is there a formal program for collecting, reviewing, analyzing, and distributing them? (A)
    a. Is this program documented? (A)