SMC-TR-99-27

AEROSPACE REPORT NO.
TR-2000(8550)-1

Software Acquisition and Software Engineering Best Practices

A White Paper Addressing the Concerns of
Senate Armed Services Committee Report 106-50
On Software Management Improvements

15 November 1999

Prepared by

S. ESLINGER
Software Engineering Subdivision
Computer Systems Division

Prepared for

SPACE AND MISSILE SYSTEMS CENTER
AIR FORCE MATERIEL COMMAND
2430 E. El Segundo Boulevard
Los Angeles Air Force Base, CA 90245

Engineering and Technology Group

THE AEROSPACE CORPORATION
l Segundo, California

This report was submitted by The Aerospace Corporation, El Segundo, CA 90245-4691, under Contract No. F04701-93-C-0094 with the Space and Missile Systems Center, 2430 E. El Segundo Blvd., Los Angeles Air Force Base, CA 90245. It was reviewed and approved for The Aerospace Corporation by R. Sudakow, Principal Director, Software Engineering Subdivision. Michael Zambrana was the project officer for the Mission-Oriented Investigation and Experimentation (MOIE) program. Col. Adrian Gomez was the Space-Based Infrared System (SBIRS) High Program Manager.

[illegible]

Michael Zambrana
SMC/AXE

[illegible]

Col. Adrian Gomez
SMC/MTS

Acknowledgements

The author wishes to thank the SBIRS High Program Office, especially James E. Slattery (SBIRS High Program Principal Director) and Milo E. Whitson (SBIRS High Ground Systems Director), for the opportunity to prepare this white paper. The preparation of this paper was funded by the SBIRS High Program Office and the Software Acquisition Mission-Oriented Investigative Experimentation (MOIE) task.

The author also wishes to thank the following members of the Computer Systems Division (CSD) and the Space-Based Surveillance Division (SBSD) for their contributions to this paper:

Richard J. Adams, Senior Engineering Specialist, CSD

Rita C. Creel, Section Manager, CSD

Roberta F. Gleiter, Project Engineer, SBSD

Sharon K. Hoting, Director, CSD

James E. Slattery, Principal Director, SBSD

Mary J. Trafton, Sr. Project Engineer, SBSD

Bonnie R. Troup, Engineering Specialist, CSD

Executive Summary

1. Introduction

1.1 Purpose and Scope

The purpose of this white paper is to address the issues raised in the recently published Senate Armed Services Committee Report 106-50[1] concerning Software Management Improvements for the Department of Defense (DoD). The text, titled "Software Management Improvements," extracted from Title VIII (Acquisition Policy, Acquisition Management, and Related Issues) of Senate Report 106-50, is given for reference in Table 1-1 of the body of this report.

This paper recommends a set of software acquisition and software engineering best practices that addresses the issues raised in the Senate Report. These recommendations are based upon the experience of The Aerospace Corporation in supporting the United States Air Force (USAF) and the National Reconnaissance Office (NRO) in the acquisition of DoD space systems. The domain of application of the recommended best practices, therefore, is the acquisition and development of large software-intensive, mission-critical systems, such as space systems, which are for the most part unprecedented.

"Best practices" are defined to be practices that have been identified through experience as significant contributors to the success of a software development effort. Thus, a best practice is not one that has been proven to be "best" in any analytical sense. Rather, it is a practice that has been used effectively on numerous successful software development projects and has been demonstrated to "help bring order, predictability, and higher levels of productivity and quality"[2] to the software project.

This paper addresses both software acquisition and software engineering best practices. Here, "software acquisition" is defined to be the set of processes (i.e., the methods, tools, techniques, procedures, etc.) used by the Government to acquire software. "Software engineering" is defined to be the set of processes used by the developers to build software.

Clearly, one of the principal components of a successful software development project is the software engineering processes used. This statement is based on the well-established fact that the quality of a software product is highly dependent upon the quality of the processes used to develop and maintain

[1] This report accompanies Senate Bill S. 1059 and is titled "NATIONAL DEFENSE AUTHORIZATION ACT FOR FISCAL YEAR 2000 REPORT [TO ACCOMPANY S. 1059] ON AUTHORIZING APPROPRIATIONS FOR FISCAL YEAR 2000 FOR MILITARY ACTIVITIES OF THE DEPARTMENT OF DEFENSE, FOR MILITARY CONSTRUCTION, AND FOR DEFENSE ACTIVITIES OF THE DEPARTMENT OF ENERGY, TO PRESCRIBE PERSONNEL STRENGTHS FOR SUCH FISCAL YEAR FOR THE ARMED FORCES, AND FOR OTHER PURPOSES TOGETHER WITH ADDITIONAL VIEWS".

[2] DoD Software Program Manager's Network, *The Program Manager's Guide to Software Acquisition Best Practices*, Version 2.2, June 1998, p. iv.

that product. However, software acquisition processes are also very influential in achieving a successful software development project. The software acquisition processes used can positively encourage, or adversely constrain, the developers in their application of high-quality software engineering processes to a software development effort.

The software acquisition and software engineering best practices recommended in this paper are limited to those that address the issues raised by the Senate Armed Services Committee in the above-cited report. The paper does not attempt to describe a complete set of recommended software acquisition and software engineering best practices. In addition, the recommended best practices are limited to those that directly address software. Systems engineering best practices, therefore, will not be addressed, although it is recognized that the quality of the developer's systems engineering processes affects the quality of the software products and the overall success of the software development effort.

1.2 Background and Definitions

Since one of the principal concerns of the Senate Report is the issue of "rework to correct product defects," this paper will emphasize software acquisition and software engineering best practices that, when applied effectively, have been demonstrated to reduce rework. "Rework" can be defined to be the amount of work expended to fix defects in any software product.[3] Here, software products include all products produced throughout the software development life cycle, including software development plans, requirements, architectures, designs, code, test plans and procedures, and user and maintainer documentation, not just the final operational software itself.

It is beyond the current state of the art to develop defect-free software products, especially in large, complex, software-intensive systems. Moreover, it is a well-documented fact that the later defects are identified, the more effort must be expended to correct the defect. Therefore, software acquisition and engineering processes must emphasize finding and fixing defects as early as possible in the software life cycle.

By its nature, engineering is an iterative activity, producing successive refinements in requirements, architectures, and designs until acceptable solutions are reached. This is true of software engineering as well as other engineering disciplines. The effort to produce the iterations and refinements that occur during the engineering process are not considered part of rework. There is also additional effort that is considered part of the normal engineering process and is, therefore, not considered rework. This includes the effort to understand requirements, technologies, Commercial Off-the-Shelf (COTS) hardware and software, and reuse software. It also includes the effort to bring prototype software or other reuse software up to the required quality to be incorporated into the operational system.

The question, therefore, arises as to the point at which modifications to software products cease being part of the normal engineering process and become rework to fix defects. The accepted boundary between normal engineering effort and rework is the quality gate for the completion of the product. A well-defined process for the development of any intermediate or final product has certain charac-

[3] Paraphrased from DoD's *Practical Software Measurement: A Foundation for Objective Project Management*, Version 3.1, April 1998.

teristics. These characteristics include entrance criteria, tasks to be performed to accomplish the development, quality checks (e.g., peer reviews, quality assurance reviews, independent product reviews, management reviews) to determine that the tasks have been properly accomplished, and exit criteria. The exit criteria for the completion of the software products include the completion of the required quality checks and the correction of any defects identified by those quality checks.

Rework is generally defined to begin after the product has passed its exit criteria quality gates. Thus, the effort to perform the quality checks that are part of the product's exit criteria is not part of rework. Also, the effort to correct any defects identified by those quality checks is considered part of the normal engineering process for developing that product and is not part of rework. However, all effort to correct any defects identified in the product after this point in time is considered rework. Rework, therefore, consists of the effort to fix defects in products found during a successor activity that uses the product after the product has passed its exit criteria quality gates.

Effort to modify software products due to changes in requirements imposed upon the software may or may not be considered rework, depending upon the source of the change. The effort to make changes due to new or modified requirements allocated to software imposed by the Government is not considered part of rework. These changes have their source external to the development effort and do not involve the correction of defects inserted into the products during their development. However, the effort to modify software products due to changes in the requirements allocated to software imposed by the systems engineering process is considered rework when those changes result from corrections to fix defects in the systems engineering products (e.g., system/segment/element requirements and requirements allocations to hardware and software items).

2. Recommended Best Practices

This section contains the recommended software engineering and software acquisition best practices that address the issues raised in the Senate Report. These recommendations are based upon the experience of The Aerospace Corporation in supporting the USAF and the NRO in the acquisition of DoD space systems. The best practices, the rationale for their inclusion, and essential elements for their effective application are described in Section 2 of the body of this report.

2.1 Software Engineering Best Practices

The set of recommended software engineering best practices, with a brief description of each, is shown in Table 1. These twenty best practices are listed in this table in alphabetical order so as not to imply an order of importance. It should be noted that the recommended best practices are not independent of each other; in fact, effective use of a particular best practice may require concurrent use of other best practices. The recommended software engineering best practices are described in more detail in Section 2.1 of the body of this report.

Table 1. Recommended Software Engineering Best Practices

Software Engineering Best Practice	Description
Binary quality gates	Using objective quality gates that must be passed by each software development task before the task can be considered complete
Configuration and change management	Rigorously enforcing configuration control and change management on all software products
Customer/user involvement	Maintaining close involvement of the customer and/or user throughout the software development life cycle
Defect root cause analysis	Analyzing data collected on software product defects to identify the root cause of the defect and to determine how to avoid similar defects from occurring
Formal inspections	Conducting formal inspections (a specific type of peer review) as mandatory quality gates for all software products
Iterative life cycle models	Developing the software iteratively in a series of builds, each fully implementing a subset of the functionality
Prototyping	Rapidly implementing specific subsets of software functionality to solve problems and reduce risks
Quantitative cost and schedule management	Using earned value management as part of managing the software development project's cost and schedule
Realistic cost and schedule estimation	Performing realistic estimation of software cost and schedule based on valid historical data and appropriate use of software cost models
Requirements traceability	Developing and maintaining accurate bi-directional traceability between levels of requirements, requirements and design, requirements and code, and requirements and test cases/procedures/results
Software architecture definition	Defining the software architecture early in the system life cycle
Software engineering process improvement	Analyzing current software engineering processes for deficiencies and implementing new/modified processes to correct those deficiencies
Software interface management	Rigorously defining and controlling all internal and external software interfaces
Software metrics	Using metrics as an integral part of the developer's software project management throughout the system life cycle
Software process standards	Using a robust software process standard for defining the software engineering processes
Software reliability engineering	Using a full life cycle approach to ensuring that the operational software meets its required reliability
Software requirements definition	Using a software requirements definition methodology to define and specify the software requirements for each software item
Software risk management	Using a continuous process of software risk identification, assessment, prioritization, mitigation, and control throughout the system life cycle
Software systems engineering	Including software as an integral part of the systems engineering processes
Test coverage exit criteria	Using objective and rigorous test coverage exit criteria for software development testing and software qualification testing

2.2 Software Acquisition Best Practices

The set of recommended software acquisition best practices, with a brief description of each, is shown in Table 2. These ten best practices are listed in this table in alphabetical order so as not to imply an order of importance. It should be noted that the recommended software acquisition best practices are not independent of each other; in fact, effective use of a particular software acquisition best practice may require concurrent use of other software acquisition best practices. In addition, the effective applications of the software acquisition best practices are related to the effective applications of the software engineering best practices. This is because the Government's software acquisition processes can positively encourage, or adversely constrain, the developer's use of software engineering best practices. The recommended software acquisition best practices are described in more detail in Section 2.2 of the body of this report.

Table 2. Recommended Software Acquisition Best Practices

Software Acquisition Best Practices	Description
Contractor capability evaluation	Performing a formal evaluation of the contractor's software development capability as part of source selection
Contractual software process commitment	Obtaining a contractual commitment that the contractor will follow mature, well-disciplined software engineering processes
Independent technical reviews	Performing independent technical reviews of the contractor's software products and processes throughout the development life cycle
Realistic cost and schedule constraints	Imposing realistic software cost and schedule constraints based on valid historical data and appropriate use of software cost models
Software acquisition metrics	Using metrics as an integral part of the Government's software acquisition processes
Software acquisition process improvement	Analyzing current software acquisition processes for deficiencies and implementing new/modified processes to correct those deficiencies
Software acquisition risk management	Using a continuous process of software acquisition risk identification, assessment, prioritization, mitigation, and control throughout the system life cycle
Software quality incentives	Using award fees and other incentives to positively motivate the contractor to use software engineering best practices
Software system acquisition	Including software as an integral part of the systems acquisition processes
Software-inclusive performance requirements	Including software in the specification of system performance requirements

3. Managing the Risk in Using COTS and Reuse Software

The principal expected benefits of using COTS and reuse software are cost and schedule savings due to a reduction in the amount of software to be developed. In addition, higher reliability and maintainability benefits are also expected due to the use of software that, in some sense, has already been proven. The expected extent of these benefits, however, is rarely met due to the current state of the practice in COTS and reuse software.

The reality of using COTS and reuse software is frequently very different from the promised benefits of its use. There is generally less COTS and reuse software that can be used than expected. This is due to such factors as the requirements for the new system not being met, the algorithms not being appropriate for the new system, the design not being compatible with the new architecture, and the reuse software not being designed for reuse. This results in additional newly developed software being required. There is also frequently more new software to be developed than expected due to large amounts of "glue" code being needed in order to integrate the newly developed code, the reuse code, and multiple COTS software packages.

In addition, the COTS or reuse software is frequently unreliable due to latent defects in the code or due to unexpected side effects of unnecessary code that cannot easily be removed or disabled. Reuse software is often difficult to modify and maintain due to poor design and implementation, out-of-date or non-existent documentation, and the presence of obsolete technologies. When COTS software is incorporated into a software-intensive system, there are frequently impacts to the software development effort when new versions of (or patches to) COTS packages are released by the vendors. There are often networks of interdependencies among COTS packages, where one package cannot be updated until one or more other packages are updated. Often these updates are being performed by different vendors on schedules incompatible with project needs.

The bottom line is that the incorporation of COTS and reuse software into software-intensive systems will almost always require additional cost and schedule over that originally estimated to complete the development effort. This is exacerbated by the demands of the competitive procurement process that cause the amounts of COTS and reuse software bid by the developers to be significantly overestimated, and the amounts of integration effort and "glue" code to be significantly underestimated.

In spite of the problems involved, affordability constraints are causing increasing reliance on COTS and reuse software. Software-intensive systems are now too large, complex, and costly to be built completely from scratch. COTS and reuse software, however, must be considered as significant risk areas that must be mitigated and managed throughout the system life cycle. Furthermore, the software acquisition and software engineering best practices described in Section 2 need to be applied to the full software development effort, including any COTS and reuse software, as well as the newly developed software.

One of the significant problems in the use of COTS and reuse software is the lack of a thorough evaluation before the decision is made to incorporate the COTS and reuse software into the system under development. Such an evaluation should be the basis of any decision to use, or not use, particular COTS software packages or reuse software, and will assist in determining risk mitigation efforts that should be carried out. Essential criteria recommended by The Aerospace Corporation for evaluating COTS and rëuse software are shown in Table 3-1 of the body of this report.

4. Conclusion

This paper recommends a set of software acquisition and software engineering best practices that address the issues raised in the Senate Report. These recommendations are based upon the experience of The Aerospace Corporation in supporting the USAF and the NRO in the acquisition of DoD software-intensive space systems. These software acquisition and software engineering best practices are strongly focused on reducing rework to correct defects in software products since rework is one of the principal concerns of the Senate Report. The Senate Report also encourages the DoD to maximize the use of COTS and reuse software. To address this issue, the paper discusses the risks inherent in such use and presents criteria for the evaluation of COTS and reuse software to assist in identifying and mitigating these risks. Appendix A to this paper provides a cross-reference mapping between the recommendations presented in Sections 2 and 3 of this paper and the Senate Report excerpt to show the coverage of the Senate Report issues by this paper.

Unfortunately, the knowledge of software acquisition and software engineering best practices is ahead of their implementation, both by the Government and by its development contractors. The DoD's acquisition reform initiatives have focused upon acquisition streamlining to reduce the projected program cost and schedule. There has been very little emphasis on the resulting quality of the systems being acquired and on the cost of rework. This is especially the case in the area of software, even though improving software product quality and reducing rework will result in considerable cost and schedule savings over the life of the program. A "contracting for quality" initiative is needed as part of the DoD's acquisition reform efforts to provide a more balanced approach to acquisition reform. This initiative should define acquisition strategies, performance requirements, incentives, and contract provisions designed to ensure that the contractors rigorously apply software engineering best practices in the development of the DoD's software-intensive systems.

The DoD also needs to take the lead in defining and transitioning software acquisition best practices to its acquisition organizations. A "software acquisition improvement" initiative is needed to stimulate software acquisition process improvement in the DoD acquisition organizations. The Software Acquisition Capability Maturity Model® (SA-CMM®) is recommended for use to provide a formalized framework for software acquisition process improvement.[4]

[4] Capability Maturity Model and CMM are registered trademarks of the Software Engineering Institute.

Contents

Tables

1. Introduction

1.1 Purpose

The purpose of this white paper is to address the issues raised in the recently published Senate Armed Services Committee Report 106-50[5] concerning Software Management Improvements for the Department of Defense (DoD). The text, titled "Software Management Improvements," extracted from Title VIII (Acquisition Policy, Acquisition Management, and Related Issues) of Senate Report 106-50, is given for reference in Table 1-1.

1.2 Scope

This paper recommends a set of software acquisition and software engineering best practices that address the issues raised in the Senate Report. These recommendations are based upon the experience of The Aerospace Corporation in supporting the United States Air Force (USAF) and the National Reconnaissance Office (NRO) in the acquisition of DoD space systems. The domain of application of the recommended best practices, therefore, is the acquisition and development of large software-intensive, mission-critical systems, such as space systems, which are for the most part unprecedented. Other domains relevant to the DoD, such as the acquisition and development of management information systems, may have different or additional best practices and are not addressed.

"Best practices" are defined to be practices that have been identified through experience as significant contributors to the success of a software development effort. Thus, a best practice is not one that has been proven to be "best" in any analytical sense. Rather, it is a practice that has been used effectively on numerous successful software development projects and has been demonstrated to "help bring order, predictability, and higher levels of productivity and quality"[6] to the software project.

This paper addresses both software acquisition and software engineering best practices. Here, "software acquisition" is defined to be the set of processes (i.e., the methods, tools, techniques, procedures, etc.) used by the Government to acquire software. "Software engineering" is defined to be the set of processes used by the developers to build software.

[5] This report accompanies Senate Bill S. 1059 and is titled "NATIONAL DEFENSE AUTHORIZATION ACT FOR FISCAL YEAR 2000 REPORT [TO ACCOMPANY S. 1059] ON AUTHORIZING APPROPRIATIONS FOR FISCAL YEAR 2000 FOR MILITARY ACTIVITIES OF THE DEPARTMENT OF DEFENSE, FOR MILITARY CONSTRUCTION, AND FOR DEFENSE ACTIVITIES OF THE DEPARTMENT OF ENERGY, TO PRESCRIBE PERSONNEL STRENGTHS FOR SUCH FISCAL YEAR FOR THE ARMED FORCES, AND FOR OTHER PURPOSES TOGETHER WITH ADDITIONAL VIEWS".

[6] DoD Software Program Manager's Network, *The Program Manager's Guide to Software Acquisition Best Practices*, Version 2.2, June 1998, p. iv.

Table 1-1 Extract from Senate Armed Services Committee Report 106-50, Title VIII, Acquisition Policy, Acquisition Management, and Related Issues

"Software management improvements The Department of Defense has a history of costly and long-standing software development and acquisition problems. These problems are documented in many General Accounting Office (GAO), Inspector General, and Department studies. The committee is concerned that, although these problems have been well documented, not enough has been done to adopt management best practices to the acquisition, development, and maintenance of software defense-wide. Industry and academic studies show that 35 to 50 percent of the development and maintenance work on software is rework to correct product defects. As result, these studies identify rework as the single largest cost driver of the $11.0 billion the Department invests annually in information technology to support business operations and the tens of billions more spent annually on information technology that supports weapon systems. The committee requests the Department report to Congress by February 1, 2000 on its efforts to identify and adopt best practices in software development. Included in the report, the Department should address: (1) how risk management is used in a project or program's software development process; (2) the process used to control and manage requirements changes during the software development process; (3) metrics required to serve as an early warning of evolving problems, measure the quality of the software product, and measure the effectiveness of the software development or acquisition process; (4) measures used to determine successful fielding of a software product; (5) how the Department ensures that duplication of ongoing software development efforts are minimized, and commercial software and previously developed software solutions are used to the maximum extent practicable; and (6) the portion of defense software expenditures (including software developed for national security systems, as defined by section 5142 of the National Defense Authorization for Fiscal Year 1996) used for rework. The committee also directs the GAO review and comment on the Department's report on software best practices by April 1, 2000."

Clearly, one of the principal components of a successful software development project is the software engineering processes used. This statement is based on the well-established fact that the quality of a software product is highly dependent upon the quality of the processes used to develop and maintain that product. However, software acquisition processes are also very influential in achieving a successful software development project. The software acquisition processes used can positively encourage, or adversely constrain, the developers in their application of high-quality software engineering processes to a software development effort.

The software acquisition and software engineering best practices recommended in this paper are limited to those that address the issues raised by the Senate Armed Services Committee in the above-cited report. The paper does not attempt to describe a complete set of recommended software acquisition and software engineering best practices. In addition, the recommended best practices are limited to those that directly address software. Systems engineering best practices, therefore, will not be addressed, although it is recognized that the quality of the developer's systems engineering processes affects the quality of the software products and the overall success of the software development effort.

1.3 Background and Definitions

Since one of the principal concerns of the Senate Report is the issue of "rework to correct product defects," this paper will emphasize software acquisition and software engineering best practices that, when applied effectively, have been demonstrated to reduce rework. "Rework" can be defined to be the amount of work expended to fix defects in any software product.[7] Here, software products include all products produced throughout the software development life cycle, including software development plans, requirements, architectures, designs, code, test plans and procedures, and user and maintainer documentation, not just the final operational software itself.

It is beyond the current state of the art to develop defect-free software products, especially in large, complex, software-intensive systems. Moreover, it is a well-documented fact that the later defects are identified, the more effort must be expended to correct the defect. Research has repeatedly shown that the effort to correct a defect increases by a factor of 2 to 10 for each life cycle phase after the defect was inserted into the software. Therefore, software acquisition and engineering processes must emphasize finding and fixing defects as early as possible in the software life cycle.

By its nature, engineering is an iterative activity, producing successive refinements in requirements, architectures, and designs until acceptable solutions are reached. This is true of software engineering as well as other engineering disciplines. The effort to produce the iterations and refinements that occur during the engineering process is not considered part of rework. There is also additional effort that is considered part of the normal engineering process and is, therefore, not considered rework. This includes the effort to understand requirements, technologies, Commercial Off-the-Shelf (COTS) hardware and software, and reuse software. It also includes the effort to bring prototype software or other reuse software up to the required quality to be incorporated into the operational system.

The question, therefore, arises as to the point at which modifications to software products cease being part of the normal engineering process and become rework to fix defects. The accepted boundary between normal engineering effort and rework is the quality gate for the completion of the product. A well-defined process for the development of any intermediate or final product has certain characteristics. These characteristics include entrance criteria, tasks to be performed to accomplish the development, quality checks (e.g., peer reviews, quality assurance reviews, independent product reviews, management reviews) to determine that the tasks have been properly accomplished, and exit criteria. The exit criteria for the completion of the software products include the completion of the required quality checks and the correction of any defects identified by those quality checks.

Rework is generally defined to begin after the product has passed its exit criteria quality gates. Thus, the effort to perform the quality checks that are part of the product's exit criteria is not part of rework. Also, the effort to correct any defects identified by those quality checks is considered part of the normal engineering process for developing that product and is not part of rework. However, all effort to correct any

[7]Paraphrased from DoD's *Practical Software Measurement: A Foundation for Objective Project Management,* Version 3.1, April 1998.

defects identified in the product after this point in time is considered rework. Rework, therefore, consists of the effort to fix defects in products found during a successor activity that uses the product after the product has passed its exit criteria quality gates.

Effort to modify software products due to changes in requirements imposed upon the software may or may not be considered rework, depending upon the source of the changes. The effort to make changes due to new or modified requirements allocated to software imposed by the Government is not considered part of rework. These changes have their source external to the development effort and do not involve the correction of defects inserted into the products during their development. However, the effort to modify software products due to changes in the requirements allocated to software imposed by the systems engineering process is considered rework when those changes result from corrections to fix defects in the systems engineering products (e.g., system/segment/element requirements and requirements allocations to hardware and software items).

1.4 Contents of This Paper

Section 2 of this paper contains the software engineering and software acquisition best practices recommended by The Aerospace Corporation to address the issues in the Senate Report. The recommended software acquisition and software engineering best practices are strongly focused on reducing rework to correct defects in software products since rework is one of the principal concerns of the Senate Report. The Senate Report also encourages the DoD to maximize the use of COTS and reuse software. To address this issue, Section 3 of this paper discusses the risks inherent in such use and presents criteria for the evaluation of COTS and reuse software to assist in identifying and mitigating these risks. Section 4 presents the conclusion to the white paper. Appendix A provides a cross-reference mapping between the white paper recommendations given in Sections 2 and 3 and the Senate Report excerpt to show the coverage of the Senate Report issues by this paper.

2. Recommended Best Practices

This section contains the recommended software engineering and software acquisition best practices that address the issues raised in the Senate Report. These recommendations are based upon the experience of The Aerospace Corporation in supporting the USAF and NRO in the acquisition of DoD space systems. This paper is not intended to be a tutorial in the application of these best practices. Rather, it briefly describes the best practices, the rationale for their inclusion, and essential elements for their effective application. Software engineering best practices are presented first, followed by the software acquisition best practices. In general, the software acquisition best practices require a basic understanding of the software engineering best practices.

2.1 Software Engineering Best Practices

The set of recommended software engineering best practices is shown in Table 2-1. These twenty best practices are discussed below in alphabetical order so as not to imply an order of importance. It should be noted that the recommended best practices are not independent of each other; in fact, effective use of a particular best practice may require concurrent use of other best practices.

Table 2-1. Recommended Software Engineering Best Practices

- Binary quality gates
- Configuration and change management
- Customer/user involvement
- Defect root cause analysis
- Formal inspections
- Iterative life cycle models
- Prototyping
- Quantitative cost and schedule management
- Realistic cost and schedule estimation
- Requirements traceability
- Software architecture definition
- Software engineering process improvement
- Software interface management
- Software metrics
- Software process standards
- Software reliability engineering
- Software requirements definition
- Software risk management
- Software systems engineering
- Test coverage exit criteria

The terms "project" and "organization" are used with specific meanings in this paragraph and its subparagraphs. A project is responsible for a single software development effort (e.g., the software development for a particular program that is developing a software-intensive system). A project has one or more software development teams performing the software development effort. An organization, on the other hand, is responsible for multiple software development projects (e.g., for multiple programs).

2.1.1 Binary Quality Gates

The use of quality gates as exit criteria for software development activities and products was discussed in Section 1.3 above. The use of binary quality gates means that each lowest-level software development task has an objective quality gate that must be passed for that task to be considered complete. Passing of a quality gate includes fixing any defects identified by the quality checks involved. Products of a task may not be used in subsequent tasks until they have passed their quality gate.

This best practice has two important benefits. First, the use of a binary quality gate for completion of a software development task, when rigorously applied, provides a mechanism for the earliest identification and removal of product defects. Second, this practice, when rigorously applied and used with an earned value measurement system, enables one to obtain an exact status of the software development project cost and schedule at any point in time.

Effective use of this best practice requires rigorous enforcement of the quality gate, even under the severest schedule pressure. Spending time up front in finding and removing defects has been demonstrated to reduce cost and schedule in the long run since correcting defects late in the life cycle or after fielding is much more expensive and time consuming. Effective use of this best practice also requires the software development project to be planned to the level of "inch-stones." The lowest-level tasks should be no larger than that which one person can accomplish in one to two weeks.

2.1.2 Configuration and Change Management

The configuration and change management best practice involves the rigorous enforcement of configuration control on all software products. Products are placed under configuration control after passing their quality gates, and all changes after that point are controlled by a process that includes review by all product stakeholders. In general, there is a hierarchical structure to the configuration management of the software products. Some products are controlled by the development team, some by the Software Configuration Management group and the Software Configuration Control Board (SCCB), and some by the Program Configuration Management group and the Program Configuration Control Board (CCB). Software development products that must be placed under configuration control include, as a minimum, software development plans, both qualitative (e.g., processes) and quantitative (e.g., schedule, effort); software standards and procedures (e.g., work instructions); software requirements, architectures, and designs, including interface requirements and designs; code; unit, integration, and qualification test plans, procedures, data, tools/drivers, and results; and operator and maintainer documentation.

The essential benefit of this best practice is defect prevention. Tight control of product configuration ensures that all members of the development team are working from the same version of the software products and thus prevents defects caused by use of multiple conflicting versions. This is especially important for interface requirements and designs. In addition, review of product changes by all product stakeholders ensures that all viewpoints have been considered before changes are made. This protects against making changes that adversely affect some product stakeholders.

Effective use of this best practice requires rigorous enforcement of change control. It requires clear communication of and access to the latest approved versions of all products by development team members. It also requires maintenance of uniquely identified and consistent product sets (i.e., sets of products that are uniquely identifiable and consistent with each other and with respect to a particular requirements baseline). In addition, effective use of this best practice requires all levels of configuration management to employ an equally rigorous configuration and change management process. Software products should be controlled at a level appropriate to their impact. Thus, internal software design may be controlled by the development team, while software requirements should be controlled by the SCCB. Higher-level requirements, along with their allocations to software and hardware, should be controlled by the Program CCB. The use of this best practice is optimal when the same configuration management tools are used at all levels.

2.1.3 Customer/User Involvement

This best practice requires the close involvement of the customer and/or user (if different from the customer) in the software development project throughout the life cycle. This involvement includes the customer's/user's technical support contractors, such as Federally Funded Research and Development Center (FFRDC) and Systems Engineering and Technical Assistance (SETA) personnel. The customer/user should be involved, for instance, in the software development planning; the software requirements, architecture, and design (including the human-computer interface design, algorithm design, and critical interface design, as well as the software design); qualification testing of the software to verify requirements; software process definition, enforcement, and improvement; and quantitative software project management (e.g., software metrics interpretation and use, and software cost and schedule management).

Close involvement of the customer/user throughout the software development life cycle will reduce rework due to misinterpretations by the developers of customer/user requirements and needs. In addition, participation by qualified customer/user personnel and their technical support contractors in software product quality checks can assist the developers in identifying and correcting product defects early in the life cycle. Close involvement of the customer/user in software qualification testing can also assist the developers in defect identification, and can provide an early assessment of the suitability of the software products for meeting operational needs.

The effective application of this best practice requires consistent and continuous participation of qualified customer/user personnel and a good working relationship between the customer/user and the developers.

2.1.4 Defect Root Cause Analysis

Defect root cause analysis involves analyzing data collected on software product defects to identify the root cause for the occurrence of the defect. The root cause analysis includes identifying the activity during which the defect was inserted, how the defect came about, and why the defect was not detected during quality checks. The results of the analysis are then used to improve the processes so that future defects of a similar type will either not occur or will be detected by the product's quality gates. The data for defect root cause analysis must be collected for all software products after they have passed their initial quality gates, whether the products are controlled by the development team, the SCCB, or the Program CCB. The defect root cause data are collected as part of the project's closed-loop corrective action process.

Defect root cause analysis, with effective process improvement based on the results, clearly will result in the overall reduction of future similar defects. This technique has been demonstrated to be extremely effective in improving software quality and reducing rework.

Defect root cause analysis is usually applied by a Software Engineering Process Group (SEPG) at the organizational level. This results in the improvement of the organizational processes and is a necessary application of the technique. However, the most effective application of defect root cause analysis occurs when it is applied both at the individual project level and at the organizational level. Application within a project can result in highly effective, immediate improvements to the processes being used on the project and in reducing rework due to defects. Application at the organizational level results in the overall improvement of organizational processes, which is of principal benefit to future projects.

2.1.5 Formal Inspections

This best practice involves conducting formal inspections as one of the mandatory quality gates for all software products. Formal inspections are a particular type of peer review. A peer review is a technical review of a software product by co-workers without management presence. There are various types of peer reviews used in the software development industry, with wide variation in their formality. Research into peer reviews has shown that the effectiveness of peer reviews in identifying defects increases with the level of formality of the peer review, with the most effective form of peer review being the formal inspection. For this best practice, the software products that are to undergo formal inspections include software development plans; software standards and procedures; software requirements, architectures, and designs; code; unit, integration, and qualification test plans, procedures, and results; and operator and maintainer documentation.

Formal inspections have been demonstrated to be effective at identifying defects in the software products under review and, therefore, at reducing rework due to defects not identified until later in the life cycle. While the time and effort required to hold a peer review increases with the level of formality, the formal inspection has proven to be extremely cost effective. This is due to the larger number of defects found by this technique when compared to peer reviews of less formality.

In order to be effective, sufficient time must be allocated to all participants to prepare for the formal inspections. In addition, the quantity of material to be reviewed must be organized into meaningful (and sufficiently small) packages to enable review of the material to the required depth within a reasonable inspection meeting duration. Data are available that give guidance in these areas, based on experience with formal inspections on numerous past projects. In addition, for formal inspections to be effective, participation in formal inspections must be a recognized part of each developer's workload. This means that the developers' deadlines for accomplishing their own development work must take into account their participation in formal inspections for other developers' products. This participation includes both the time to prepare for the formal inspections and the time to participate in the inspection meetings.

2.1.6 Iterative Life Cycle Models

The use of iterative life cycle models for the development of large, complex software-intensive systems has been demonstrated to reduce software development risk. In an iterative life cycle model, the software is developed in a series of builds where each build implements a subset of the full software functionality. Each build adds capabilities to the previous builds until the software is completely developed. There are multiple types of iterative life cycle models with the most popular being the incremental, evolutionary, and spiral. In the incremental life cycle model, all software requirements are developed first, before beginning the builds that design, implement, and test subsets of the software. In the evolutionary life cycle model, each build includes defining the software requirements for that build as well as performing the design, implementation, and test of the software that implements those requirements. In the spiral model, each spiral (which may consist of one or more builds) is focused on specific risk reduction goals.

The use of iterative life cycle models can reduce risk and rework in software development since it provides for early integration of the software components. The use of iterative life cycle models can also reduce risk by enabling difficult or high-risk areas of the software to be prototyped in early builds/spirals. The evolutionary life cycle model is particularly effective when the software requirements are not clearly understood at the beginning of the development effort. Defining the software requirements within each build as the requirements are better understood can reduce software requirements rework due to incomplete understanding of the requirements at the beginning of the project. Furthermore, the use of iterative life cycle models facilitates obtaining early customer/user feedback on the software by enabling each build to be demonstrated as it is completed. More robust customer/user feedback can be obtained by providing the opportunity for hands-on execution of the builds by the customer/user. Such iterative customer/user feedback can reduce rework caused by misunderstanding of customer/user needs.

Effective application of iterative life cycle models requires that the complete software architecture be defined before the builds are initiated. Without a well-defined architecture for the full software product, use of iterative life cycle models can actually result in extensive rework when attempting to add the capabilities in later builds. In addition, effective application of iterative life cycle models requires the implementation of the software infrastructure (i.e., the architectural foundation) in the early builds. This ensures that the infrastructure is available for use by the application software being developed in those and subsequent builds.

2.1.7 Prototyping

Prototyping involves rapid implementation of specific subsets of software functionality in order to reduce risk in the later implementation of that functionality. Examples of areas where prototyping has been proven to be of benefit include algorithms; critical interfaces, especially for non-standard interfaces; the human-computer interface (e.g., user screens); and integration of COTS packages (e.g., to determine whether the COTS packages will satisfy the requirements and to determine the amount of "glue" code required to integrate the COTS with the developed software).

Prototyping reduces rework in the operational software by enabling technical problems, issues, and areas of high risk to be resolved with minimal effort early in the life cycle. The design of the operational software can then be based upon known solutions.

To be effective, prototyping must be performed using rapid development techniques that enable exploration of various solutions. Using such techniques, the resulting prototype software must not be expected to conform to the required quality and robustness of the operational "industrial strength" software. Effective use of prototyping requires that it be developed with the expectation that it will be discarded after resolution of the problems, issues, and risks for which it was built. Before attempting to reuse prototype software in the operational system, a comprehensive cost-benefit analysis must be performed. This analysis must evaluate whether or not it is cost effective to add any additional required functionality and to re-engineer the prototype software to conform to the required quality and architecture of the operational software. Frequently, it is more expensive to add functionality and re-engineer the prototype software than to develop new software to perform the operational functionality. Prototype software is almost never suitable for incorporation into the operational software without significant re-engineering.

2.1.8 Quantitative Cost and Schedule Management

This best practice involves the use of earned value management in order to manage the software development project's cost and schedule. To use earned value techniques, a detailed work breakdown structure for the software development project is developed, work packages are identified and divided into small tasks with objective entry and exit criteria, and activity networks for the tasks are created. An explicit verification process is used to determine whether a task has met its exit criteria and thereby "earned" its value. The best practice of earned value management uses only binary credit for earned value. This means that the earned value for a task is zero until its exit criteria have been verified as met, and at that point the task earns 100% of its value.

The most effective application of earned value management occurs when binary quality gates are used as part of the exit criteria for each software development task. The verification process used as part of the earned value technique, when rigorously enforced, then ensures that the required quality checks have been performed on the software products and the identified defects corrected before the value is earned. This will result in the reduction of defects found in subsequent tasks or after fielding. In addition, quantitatively managing the software cost and schedule by earned value enables an exact status of the project to be obtained at any point in time. Deviations from the plan can then be identified as they occur and corrected immediately.

Effective use of this best practice requires the software development project to be planned to the level of "inch-stones." The lowest level tasks should be no larger than that which one person can accomplish in one to two weeks. The definition of these tasks must match the detailed software engineering processes in use on the program. Top-level planning of the entire software development effort (including the work breakdown structure, work packages, and activity network at the work package level) is performed at the beginning of the project. However, most of the detailed task planning requires more information about the structure of the software than is known at the beginning of the project. Therefore, the detailed planning at the individual task level is performed for rolling increments of the schedule throughout the development life cycle.

2.1.9 Realistic Cost and Schedule Estimation

This best practice involves making realistic software cost and schedule estimates based upon valid historical data and appropriate use of software cost models. Initial software cost and schedule estimates are based upon experience with similar past programs, while updates to software cost and schedule estimates add more accurate data from the current project (e.g., actual effort data, more accurate software size estimates).

Realistic software cost and schedule estimation not only increases the predictability of the cost and schedule and, therefore, decreases the probability of overruns, it also contributes to improved quality of the software product. Sufficient time and effort to develop the software products, including performing the quality checks and correcting any identified defects, is essential to reducing rework. Experience has demonstrated that one of the largest contributors to poor software quality is schedule pressure. When large amounts of overtime (paid or unpaid) are required to meet the software cost and schedule constraints, software product quality suffers. In addition, research has repeatedly shown that a software schedule compression of more than 25% off nominal is impossible to meet. Thus, with more than 25% schedule compression, the software product will not meet the schedule, and in addition, the resulting schedule pressure will cause poor software product quality due to shortcuts being taken in the software development processes.

For realistic software cost and schedule estimation to be effective, accurate historical software cost and schedule data from current and past projects are necessary. Many development organizations do not account for unpaid overtime in their historical data. This results in the calculation of large productivity values that are erroneous. Estimates of future efforts using these productivity values are then based upon the assumption of large amounts of unpaid overtime by the software development personnel. This situation will result in poor software quality and increased rework. Realistic cost and schedule estimation also requires proper use of the software cost models. This includes appropriate decomposition of the software for costing, appropriate values for parameter settings, addition of effort and time for activities not covered by the software cost models, and validation of the software cost model results against historical data. Furthermore, realistic software size estimation is necessary for realistic software cost and schedule estimation. Software size is notoriously underestimated at the beginning of a project. This includes unrealistically optimistic estimates for amounts of COTS and reuse software that can be used and unrealistically small estimates for amounts of new code to be developed.

2.1.10 Requirements Traceability

The best practice of requirements traceability involves the development and maintenance of accurate bi-directional traceability between all levels of requirements, from system requirements through individual software and hardware item requirements; between software requirements and software architecture components; between software requirements and software detailed design units; between software requirements and code; and between software requirements and software test cases, test procedures, and test results.

Bi-directional requirements traceability is a tool for determining whether the system requirements allocated to software are fully implemented by the software requirements. It is also a tool for determining whether all software requirements are being met by the evolving software products (e.g., architecture, detailed design, and code) as development proceeds. Furthermore, it is a tool for determining whether all software requirements have been fully verified by the software qualification testing. By using the bi-directional requirements traceability as part of software product quality checks, defects due to improper requirements implementation can be avoided. Bi-directional traceability is also essential to the accurate analysis of impacts due to changes in higher-level requirements, software requirements, or other software products.

The most effective application of bi-directional traceability requires ensuring the consistency, completeness, and correctness of the traceability information. Development, maintenance, and quality checking of the traceability information is most efficiently and effectively performed if the requirements traceability is implemented in an automated tool that provides automated checking and flexible selection and viewing capabilities. To be effective, the traceability must also be readily available to the technical and management personnel that need the information.

2.1.11 Software Architecture Definition

This best practice involves the early life cycle definition of the architecture of the software portion of the software-intensive system. A complete description of the software architecture consists of multiple architectural views that describe various aspects of the software (e.g., the logical view, describing the functional behavior; the process view, describing the dynamic execution behavior; the physical view, describing the allocation of software to hardware; the development view, describing the static structure of the software; the user view, describing the operational behavior from the user's viewpoint). DoD mission-critical, software-intensive systems are generally operational for long periods of time, sometimes decades. During this time, changes will occur to system requirements and interfaces, to the available base of commercial hardware and software, and to various computer systems technologies. The software architectures for such systems need to be appropriately designed so that the systems can readily adapt to change and can evolve as the system environment and requirements evolve.

A complete definition of the software architecture early in the life cycle will help reduce rework later in the life cycle by providing the framework for the design and implementation of the operational software. This is especially true when using iterative life cycle models since the early definition of the software architecture provides the framework for all subsequent builds. Without a complete architectural definition, functionality being implemented in later builds can require major rework to the contents of previous builds. The essential benefit of designing software architectures so that they readily support change and evolution is the reduction in the cost of necessary environmental and requirements changes. This benefit applies both during development and after fielding. While the effort to accommodate such changes is not considered rework, this effort is a large component of the total system development and maintenance cost.

The most effective application of software architecture definition requires the use of a Computer-Aided Software Engineering (CASE) tool that supports the techniques being used for describing the architectural views. The techniques used for describing the architectural views must be compatible with the methodologies in use for software requirements definition and software design. In addition, there are a number of accepted principles for structuring software architectures that readily support change and evolution. These principles include the use of layered architectures and open systems standards (e.g., those mandated by the DoD's Joint Technical Architecture initiative). Software architectures are currently a topic of major interest in computer science research.

2.1.12 Software Engineering Process Improvement

Software engineering process improvement involves analyzing the current software processes for deficiencies and putting new and/or modified processes in place to correct those deficiencies. Process improvement can be done informally or with the use of a formal model for process improvement, such as the Software Engineering Institute's (SEI's) Capability Maturity Model® for Software (SW-CMM®).[8] The SW-CMM® has become widely accepted worldwide and is now the industry standard for software process improvement efforts.

Data collected by organizations that have instituted software process improvement efforts have shown process improvement to be extremely cost effective due to fewer product defects being identified downstream and due to greater predictability of software cost and schedule.

[8]Capability Maturity Model and CMM are registered trademarks of the Software Engineering Institute.

Software engineering process improvement is usually applied by an SEPG at the organizational level. This results in the improvement of the organizational processes, and is a necessary application of the technique. However, process improvement is most effective when it is applied both at the individual project level and at the organizational level. Application within a project can result in highly effective immediate improvements to the processes being used on the project and in reducing rework due to defects. Application at the organizational level results in the overall improvement of organizational processes, which is of principal benefit to future projects. In addition, software engineering process improvement will not be effective unless the improved processes are used and rigorously enforced on each project.

2.1.13 Software Interface Management

Software interface management requires the early definition and rigorous configuration control of all software interfaces, both external and internal. Software interface requirements and detailed designs must be documented and agreed to by all stakeholders, and conformance to these interface specifications (both requirements and detailed design) must be rigorously enforced. This best practice applies to all levels of software interfaces: interfaces with external systems, interfaces between segments, interfaces between software items within segments, and interfaces between software units within each software item.

One of the most prevalent problems found during software and system integration is incorrectly specified and/or mismatched software interfaces. The early definition and configuration control of software interfaces will help prevent this type of defect from occurring. Checks for interface specification conformance during software product quality checks (e.g., formal inspections) will aid in detecting this type of defect prior to software and system integration, where such defects are more expensive to correct.

The effective application of this best practice requires all software interfaces to be defined and documented early in the life cycle before implementation of the software on either side of the interface. It also requires rigorous enforcement of change control. Furthermore, it requires clear communication of and access to the latest approved versions of all interface specifications by development team members. Most software projects define, document, and control external software interfaces and interfaces between software items both within and between segments. However, for maximum defect prevention in this area, it is equally important to define, document, and control the internal software item interfaces between software units, especially where these interfaces must be shared among multiple software developers. Software interfaces should be controlled at a level appropriate to their impact. Thus, internal software item interfaces may be controlled by the development team, while interfaces between software items within and between segments should be controlled by the SCCB or Program CCB. External software interfaces should always be controlled by the Program CCB.

2.1.14 Software Metrics

This best practice involves the use of metrics as an integral part of software project management to assist in managing the software products, processes, and resources. The software/system development project attributes recommended for measurement by The Aerospace Corporation for software-intensive systems are shown in Table 2-2. For a software metrics program to be effective, the work performed on the lowest-level tasks must be made measurable, and the metrics must be consistent with and integrated into the project's software engineering processes. This best practice requires project management to establish an environment that fosters the use of metrics and assures project personnel that the collected data will never be used against individuals. It also requires project management to ensure that metrics are used as a

Table 2-2. Recommended Software/System Attributes to Measure

PRODUCT-PROCESS

- Quality and Performance Attributes
  - Volatility
  - Traceability
  - Problem Reports/Action Items/Issues
  - Defect Density/Inspection Effectiveness
  - Fault Density/Test Effectiveness
  - Size
  - Structure (Complexity, Coupling, Cohesion, etc.)
  - Target Resource Utilization
- Progress Attributes
  - Completeness
  - Integrated Progress

PROJECT RESOURCES

- Capability and Capacity Attributes
  - Staffing Levels/Skills
  - Turnover Rates
  - Engineering Environment Resource Availability/Utilization
  - Test Environment Resource Availability/Utilization

management tool, not an end in themselves, and that adherence to well-disciplined software engineering processes is not subverted in order to make the metrics appear "good."

An effective software measurement program provides high-level visibility into the health and status of the evolving software product and lower-level data for timely problem detection, isolation, and impact assessment. High-quality, accurate metrics reduce information overload; enable software process evaluation and improvement; provide assessment of product quality and project progress; and facilitate early detection, understanding, and resolution of software problems. When accurate, well-defined data are collected from multiple software projects by an organization, this historical data can be analyzed to determine expected ranges of values for the various metrics. These expected ranges enable a project to easily identify areas where corrective action is needed.

An effective software metrics program requires comprehensive metrics planning. This includes selecting an appropriate set of metrics to be collected throughout the life cycle. In addition, to ensure a clear understanding of the metrics and metrics data, the following detailed information must be defined for each selected metric: benefits/costs (e.g., effort, resources); general definition; raw data items to be collected; computations to be performed; data collection methods, frequency, and responsibility; data reporting frequency, responsibility, and users; analysis, interpretation, and feedback guidelines; and relationship to other metrics. Techniques are available to assist in selecting an appropriate set of metrics for an individual program, based upon program goals and risks. Care must be taken to ensure that the selected metrics cover the entire software and system life cycle and all software development processes, products, and resources. Once selected, the metrics must be defined to be consistent across the entire program (i.e., the same definition for all development teams, for any type of teaming arrangements); consistent across all levels of the program (i.e., the same definition for all levels of the specification tree); and integrated with other disciplines (e.g., reliability; cost and schedule management). The most efficient application of a software metrics program utilizes automated collection, computation, and reporting of the metrics data to the maximum extent possible.

2.1.15 Software Process Standards

Using a robust software process standard as the basis for defining software engineering processes provides an ensured level of completeness, stability, and quality for project and organizational processes. The current best software process standard for software-intensive systems is IEEE/EIA J-STD-016-1995, which is the commercial version of MIL-STD-498.[9]

A robust software process standard, such as IEEE/EIA J-STD-016-1995, defines the software development activities that must be performed and the tasks that must be accomplished for each of those activities. A robust software process standard also provides detailed instructions for the content of the various software development products. Rigorous adherence to such a standard will reduce defects since it will guarantee that all necessary software engineering tasks will be accomplished and all necessary engineering work will be performed for each software product.

Effective application of a robust software standard requires careful translation of the requirements of the standard into the project and organizational software engineering processes and rigorous enforcement of the resulting processes. Extreme caution must be used in tailoring out (i.e., deleting) features of the standard since this usually reduces the quality of the software engineering processes and their resulting products. This is especially true with respect to a standard's required contents of software products. The document contents specified in IEEE/EIA J-STD-016-1995 provide checklists for the engineering work that needs to be performed, independent of the form that the results of the engineering work will take. Thus, tailoring out document contents can result in the definition and use of processes that do not require the engineering work necessary to produce quality software products.

2.1.16 Software Reliability Engineering

Software reliability engineering is a full life cycle program whose goal is to ensure that the operational software will meet its required reliability (generally measured in terms of Mean Time Between Failures). Actual system failures experienced during operations will be caused by both hardware failures and software faults. Thus, the operationally experienced system reliability will always be less than a system reliability estimation based upon hardware alone. Operational system reliability that includes both hardware and software is an excellent measure of the successful fielding of the software product.

The first step in software reliability engineering is the allocation of system reliability requirements to both hardware and software components based upon the system design. Data on software failures experienced during integration and qualification testing are collected, and a software reliability model is used to estimate the software reliability. Testing can then be continued until the software meets its reliability requirement. Effective use of a software reliability program will, therefore, result in fewer defects in the software when transitioned to operations.

Effective use of software reliability engineering requires the implementation of a full life cycle set of software engineering processes geared toward reducing software product defects. Examples of such processes are those that implement the software engineering best practices described in this section. In

[9] ISO/IEC 12207, the international standard for software life cycle processes, is a high level standard that covers acquisition, supply (the prime contractor role), development, operations, and maintenance. The U.S. implementation of ISO/IEC 12207 (IEEE/EIA 12207.0-1996, 12207.1-1997 and 12207.2-1997) is not recommended for use in a stand-alone mode since it is not sufficiently robust in the areas of software development and maintenance processes. However, since IEEE/EIA J-STD-016-1995 is compliant with ISO/IEC 12207 for software development and maintenance, using IEEE/EIA J-STD-016-1995 in conjunction with the U.S. implementation of ISO/IEC 12207 is recommended.

addition, a reliability demonstration of the software should be performed using a realistic operational profile of software features under realistic operational workload conditions.

2.1.17 Software Requirements Definition

This best practice involves using a software requirements methodology for the definition and specification of the software requirements for each software item in the system. Software requirements methodologies provide an organized, disciplined technique for understanding the system requirements allocated to software and for defining the detailed requirements that the software must satisfy. The most popular software requirements methodologies currently in use are the structured and object-oriented methods. Both of these methodologies result in a model of the software requirements that is used for understanding and specifying the individual requirements.

One of the largest sources of defects in software products is the software requirements. Improper understanding, incompleteness, incorrectness, inconsistency, and improper level of detail (either too high to understand what the software must do or so low that the design is over-constrained) are common problems in software requirements in large, complex, software-intensive systems. Use of a software requirements methodology helps to avoid these problems and thus reduces the incidence of defects due to software requirements errors. Ensuring the completeness of the software requirements is especially important in order to eliminate problems later in the life cycle.

The effective application of software requirements methodologies requires rigorous adherence to the rules and standards of the methodology. These methodologies are most efficiently applied by using a CASE tool that supports the methodology. In addition, effective use of a software requirements methodology requires at least some of the software requirements definition personnel to be experienced in using that methodology. All software requirements personnel need to be trained both in the methodology itself and in the particular CASE tool being used.

2.1.18 Software Risk Management

Software risk management is an effective mechanism for reducing the impact of potential problems on the software project. Software risk management involves a continuous process of risk identification, assessment, prioritization, mitigation, and control throughout the life cycle of the project.

The use of software risk management enables a software project to understand its risks (i.e., future problems that might occur) and, where possible, to mitigate the risks that are assessed to have the largest potential impact on the software development effort. All software development projects have risks, and the cost to the project of resolving a problem (i.e., an actualized software risk) later in the life cycle is often significantly larger than mitigating that same risk early in the life cycle. Thus, effective software risk management can result in reducing the cost and schedule impact of problems by having alternative technical solutions or workarounds identified before problems occur.

To be effective, the software risk management process must be practiced by all development personnel and at all levels of the software development project. An environment that rewards the identification of risks must be fostered by the developer's program management so that technical personnel are encouraged to identify risks and devise appropriate mitigation strategies. Large programs frequently have a risk management process at the program level with the top program risks being closely monitored by developer and Government upper management. However, on large, complex DoD programs with major hardware and software development, individual software risks are seldom considered to have a large enough potential cost impact to reach the program risk list until those software risks have become significant program problems. Frequently, the only risk management process practiced on such a program is the program level process. An effective software risk management process must be practiced at every level of the software development project, including the lowest-level software development teams. Each software development team should use the software risk management process to identify its risks, mitigate and control those risks that are within its scope of control, and elevate those risks with sufficiently high impact to higher-level development teams.

2.1.19 Software Systems Engineering

This best practice involves the inclusion of software as an integral part of the systems engineering processes in the development of software-intensive systems. Software systems engineers (i.e., personnel with computer systems knowledge and experience, both hardware and software) must be members of the teams responsible for the definition of the system and other higher-level requirements for all levels of the specification tree. Software systems engineers must also be members of the system architecture and design team(s) and must be key participants in the allocation of higher-level requirements to computer subsystems and to individual software and hardware items. Similarly, software systems engineers must also be members of the system integration and verification teams. These teams are responsible for the integration of all system components, both hardware and software, and the verification of all higher-level requirements for all levels of the specification tree above the individual software and hardware items. Furthermore, software specialty engineers must be key participants in the system specialty engineering disciplines that involve both hardware and software elements, including reliability/maintainability/availability (RMA), supportability (including testability and integrated system diagnostics), safety, security, and human factors.

The inclusion of software as an integral part of the system requirements, architecture, and design definition helps to reduce rework to correct defects caused by inappropriate or infeasible higher-level requirements allocated to computer hardware and software. It also helps to reduce rework to correct the inappropriate or incorrect inclusion of computer hardware and software in the system architecture and design. Inclusion of software as an integral part of system integration and verification helps to ensure the thorough verification of higher-level requirements implemented in software. This will result in reducing the number of latent software defects remaining after all system requirements verification is completed. In addition, without consideration of software in system specialty engineering, rework can be caused by the final system (composed of both hardware and software components) not meeting its specialty engineering requirements. Finally, the inclusion of software along with hardware in the integrated system diagnostics will improve software maintenance by enhancing the capability to detect and isolate software faults.

The effective incorporation of software as an integral part of the systems engineering processes requires the developer's program management to establish an environment where software is a highly respected part of the program, equivalent in importance to hardware. In addition, the program must be structured to provide effective lines of communication, responsibility, and authority among the software development teams and the other program teams involved in the systems engineering processes. Finally, the software engineering processes and systems engineering processes must be defined so that they are consistent and integrated with each other.

2.1.20 Test Coverage Exit Criteria

This best practice involves the use of objective and robust test coverage exit criteria for all levels of software development testing (including software unit testing, unit integration testing, and hardware/software integration testing) and for software qualification testing. Robust test coverage exit criteria for software unit testing should include unit test cases covering, as a minimum, the correct execution of all statements and branches; all error and exception handling; all unit interfaces, including limits and boundary conditions; start-up, termination, and restart (when applicable); and all algorithms. Robust test coverage exit criteria for unit integration testing should include integration test cases covering, as a minimum, the correct execution of all interfaces between units, including limit and boundary conditions; integrated error and exception handling across the units under test; all end-to-end functional capabilities through the units under test; all software requirements allocated to the units under test; performance testing, including operational input and output data rates and timing and accuracy requirements; stress testing, including worst-case scenario(s); start-up, termination, and restart (when applicable); fault detection, isolation, and recovery handling (e.g., fault tolerance, failover, data capture and reporting); and resource utilization measurement (e.g., CPU, memory, storage, bandwidth). Robust test coverage exit criteria for hardware/software integration testing are similar to the criteria for unit integration testing. Robust test coverage exit criteria for software qualification testing should include, as a minimum, verification of all software requirements under conditions that are as close as possible to those that the software will encounter in the operational environment (e.g., operational databases, operational input and output data rates, target hardware configurations); verification of all software interface requirements, using the actual interfaces wherever possible or high-fidelity simulation of the interfaces where not possible; verification of all software specialty engineering requirements (i.e., reliability/maintainability/availability, supportability, testability, safety, security, human factors, as applicable), including in particular verification of software reliability requirements and fault detection, isolation, and recovery requirements; stress testing, including worst-case scenario(s); and resource utilization measurement (e.g., CPU, memory, storage, bandwidth). Unit integration, hardware/software integration, and software qualification testing should always be performed on target hardware configured to be as close as possible to the operational hardware configuration.

Software development testing and software qualification testing are the last opportunities for identifying and correcting software defects before the system is integration tested, qualification tested, and fielded. The rigorous enforcement of objective and robust test coverage exit criteria for all levels of software testing will ensure a thorough test program that maximizes defect identification.

Effective application of this best practice requires the software to be tested under conditions as similar as possible to those that will be encountered during operations. This requires early planning of the entire software test effort so that sufficient time and funding are allocated for the development of high-fidelity simulators and other necessary test tools and for the procurement and/or development of high-fidelity ground testbeds. Effective application of this best practice also requires the inclusion of COTS and reuse software in the robust exit criteria for all levels of testing. Reuse software, as a minimum, should be subjected to the same exit criteria as newly developed software for unit testing, unit integration testing, and hardware/software integration testing for all modified units, for all units where the track record indicates potential problems (even if the units have not been modified), and for all critical units (even if the units have not been modified). Unmodified reuse software and all COTS software, as a minimum, should be subjected to the same exit criteria as newly developed software for unit integration testing and hardware/software integration testing. Finally, all software requirements must be verified during software qualification testing, whether they are satisfied by COTS, reuse (modified or unmodified), or newly developed software.

2.2 Software Acquisition Best Practices

The set of recommended software acquisition best practices is shown in Table 2-3. These ten best practices are discussed below in alphabetical order so as not to imply an order of importance. It should be noted that the recommended software acquisition best practices are not independent of each other; in fact, effective use of a particular software acquisition best practice may require concurrent use of other software acquisition best practices. In addition, the effective applications of the software acquisition best practices are related to the effective applications of the software engineering best practices. This is because the Government's software acquisition processes can positively encourage, or adversely constrain, the developer's use of software engineering best practices.

The phrases "software acquisition project" and "acquisition organization" are used with specific meanings in this paragraph and its subparagraphs. A software acquisition project is responsible for the acquisition management of a single software development effort (e.g., the software development for a particular program that is developing a software-intensive system). A software acquisition project has one or more software acquisition teams performing the software acquisition management effort. An acquisition organization, on the other hand, is responsible for the acquisition management of multiple software development efforts (e.g., for multiple programs).

Table 2-3. Recommended Software Acquisition Best Practices

- Contractor capability evaluation
- Contractual software process commitment
- Independent technical reviews
- Realistic cost and schedule constraints
- Software acquisition metrics
- Software acquisition process improvement
- Software acquisition risk management
- Software quality incentives
- Software system acquisition
- Software-inclusive performance requirements

2.2.1 Contractor Capability Evaluation

This best practice consists of performing a formal evaluation of the contractor's software development capability as part of the source-selection process for software-intensive systems. There are currently two principal methods in use for performing such a formal evaluation: the SEI's Software Capability Evaluation (SCE^{SM}) and the USAF's Software Development Capability Evaluation (SDCE).[10] Both methods are effective tools for obtaining insight into the offerors' software development processes, and both methods provide strengths, weaknesses, and risks for use in the source-selection evaluation.

The primary purpose for performing a contractor capability evaluation is to increase the likelihood of selecting a contractor capable of developing the required software within the program constraints. It is well established that risk in software development is reduced by selecting a contractor with mature software engineering processes since the quality of a software system is largely governed by the quality of the processes used to develop and maintain it. A secondary purpose for performing a contractor capability

[10] SCE is a registered service mark of the Software Engineering Institute.

evaluation is to identify risks associated with the selected contractor to facilitate managing these risks beginning at contract award. Another secondary purpose is to obtain a contractual commitment from the selected contractor to adopt processes that instill and support effective software engineering discipline.

For a contractor capability evaluation to be effective and to achieve its goals, the contractor capability evaluation must occupy a position of sufficient importance in the source-selection evaluation criteria to affect the results of the source selection. It is strongly recommended that the contractor capability evaluation be an entire subfactor of the Mission Capability factor.

2.2.2 Contractual Software Process Commitment

This best practice involves obtaining a contractual commitment that the contractor will follow specific, well-disciplined software engineering processes. This commitment may be made by contractually requiring adherence to a robust commercial standard for software development, such as IEEE/EIA J-STD-016-1995. Alternatively, the commitment may be made by contractually requiring adherence to the developer's Software Development Plan (SDP). In some acquisition environments, this contractual requirement may be specified by the Government in the contract (e.g., by making IEEE/EIA J-STD-016-1995 or the contractor's SDP be compliance documents). In other acquisition environments, the contractor may be required to specify their own compliance documents as part of their contractually compliant Integrated Master Plan.

Since the quality of the software products is highly dependent upon the quality of the processes used to develop them, adherence to mature, well-disciplined software engineering processes is essential to delivery of a high-quality software-intensive system. Contractual commitment helps to ensure that the developer will adhere to the required software engineering processes.

The most effective application of this best practice occurs when the developer's adherence to the contractually compliant software development processes is evaluated as part of the award and/or incentive fees. The use of award and/or incentive fees will provide significant motivation to the developer for process compliance. Care must be taken to ensure that the contractually compliant software processes are of sufficiently high quality to support the software development necessary for the software-intensive system under contract. If IEEE/EIA J-STD-016-1995 is used, the Government should have approval over any tailoring used in order to ensure that essential aspects of the standard's required software engineering processes have not been deleted. If the developer's SDP is used, the Government should have approval of the initial version of the SDP and any modifications to it before any version becomes contractually compliant. This will help ensure that the SDP defines sufficiently high-quality software engineering processes to support development of the required software.

2.2.3 Independent Technical Reviews

This best practice consists of independent technical reviews of the developer's software products and processes throughout the development life cycle. These independent technical reviews are performed by the Government software acquisition team, which includes Government software acquisition personnel and their software technical support contractors (such as FFRDCs, SETA contractors, and/or Independent Verification and Validation contractors).

The objective of independent technical reviews of the developer's software products is to assist the developer in identifying defects in the software products. Independent technical reviews of software products by the Government software acquisition team generally identify defects not found by the developer's quality checks (e.g., defects due to misinterpretations of requirements, defects due to incorrect use of new technologies). This is due to the difference in perspective of the independent reviewers compared to the development personnel. The objective of independent technical reviews of the developer's software processes is to assist the developer in improving their software development processes and their process compliance. These independent process reviews focus on determining whether the developer's documented software development processes are effective and whether the developers are, in fact, following their documented processes. Independent process reviews by the Government software acquisition team may be informal or formal. One formal method for software process review is the SEI's SCE^{SM}, which can be applied for contract monitoring as well as source selection. Other types of independent technical reviews, called independent assessments, focus on identifying problems and risks in the software development project, assessing actual software status and performance, and developing solutions to critical problems.

A good working relationship and good communication between the Government software acquisition team and the developer's software development team will enhance the effectiveness of the independent technical reviews. The most effective application of independent technical reviews occurs when the results of the Government software acquisition team's reviews are used by the development contractor to correct and improve their products and processes. Highly effective application of independent technical reviews also occurs when the results of the Government team's independent technical reviews are used as input to the award and/or incentive fee determination process.

2.2.4 Realistic Cost and Schedule Constraints

This best practice involves the Government making realistic software cost and schedule estimates based upon valid historical data and the appropriate use of software cost models. The contractual cost and schedule constraints imposed upon the developer must then be based upon these realistic software cost and schedule estimates. Early life cycle software cost and schedule estimates should be based upon historical data collected by the acquisition organization on similar past programs. Later in the life cycle, updates to the Government's software cost and schedule estimates should include more accurate data collected on the current software acquisition project.

Realistic software cost and schedule estimation not only increases the predictability of the cost and schedule, and, therefore, decreases the probability of overruns, it also contributes to improved quality of the software product. Sufficient time and effort to develop the software products, including performing the quality checks and correcting any identified defects, is essential to reducing rework. Experience has demonstrated that one of the largest contributors to poor software quality is schedule pressure. Research has repeatedly shown that a software schedule compression of more than 25% off nominal is impossible to meet. Thus, the Government should never impose contractual schedule requirements that would constrain the software schedule to be greater than, or even to approach, 25%. When such extreme schedule constraints are imposed upon the developer, the schedule will not be met. In addition, the resulting schedule pressure will result in poor software product quality since the contractual schedule constraints will force the developer to take shortcuts in the software development processes.

For this best practice to be effective, the acquisition organization must develop and maintain an accurate historical database of software size, cost, and schedule data from past and current software development projects. In order to use this data for statistical analysis, the data must be uniformly defined and consistently collected across projects. Another important part of this best practice is that the Government's software cost and schedule estimates should be independent; that is, the Government should perform its software cost and schedule estimates should be independent; that is, the Government should perform its own software cost and schedule estimation and not rely solely upon the developer's estimates. The data collected from the developer should provide important input into the Government's estimation process. However, the independence of the Government's estimates is necessary to eliminate the biases present in the developer's estimates (e.g., underestimation of amount of new code to be developed, overestimation of amount of COTS and reuse code, overly optimistic parameters used in the software cost models, overly optimistic productivity data).

2.2.5 Software Acquisition Metrics

This best practice involves the use of metrics as an integral part of the Government's software acquisition processes. A software acquisition metrics program includes measurement of both the development and the acquisition processes. This means that the Government software acquisition team needs metrics data from the developer to assess the software development products, processes, and progress. Table 2-2 shows the software/system attributes recommended for measurement by the development project. In addition, measurement of similar attributes of the work performed by the Government software acquisition team itself are needed for effective management of its own work. This best practice also requires the acquisition organization to collect, maintain, and analyze historical software engineering and software acquisition metrics data across multiple programs to enhance understanding and predictability of future efforts.

An effective software acquisition measurement program provides high-level visibility into the health and status of the software engineering and software acquisition products and lower-level data for timely problem detection, isolation, and impact assessment. High-quality, accurate metrics reduce information overload; enable software engineering and acquisition process evaluation and improvement; provide assessment of product quality and project progress; and facilitate early detection, understanding, and resolution of problems. When accurate, well-defined software acquisition and software development data are collected from multiple projects by an acquisition organization, this historical data can be analyzed to determine expected ranges of values for the various metrics. These expected ranges enable a software acquisition project to easily identify areas where corrective action is needed.

An effective software acquisition metrics program requires comprehensive metrics planning, to include selecting an appropriate set of software engineering and software acquisition metrics to be collected throughout the life cycle and defining each selected metric in detail. The selected metrics should be based upon an analysis of the goals of the acquisition organization and of the questions that the metrics are intended to address. For example, to answer the Senate Report's question about the amount of expenditures spent on software development rework, the proper data must have been collected on all defense software development projects. In order for the historical software acquisition data to have maximum utilization across an acquisition organization, it is important that the data be defined and collected in the same way across all software development projects managed by that acquisition organization. This is necessary in order to perform any kind of valid statistical analysis on the collected data. As an example, consider the question of determining an expected value of software productivity (in source lines of code per person-month) for newly developed code. In order to compute this value correctly from historical data, each software development project would need to count lines of code in exactly the same way. To accomplish this type of data collection, the acquisition organization needs to place a uniform set of contractual requirements for data collection and reporting on each of its development contractors. Similarly, the acquisition organization would need to define a uniform set of requirements for acquisition data collecting and reporting by each of its acquisition projects.

2.2.6 Software Acquisition Process Improvement

Software acquisition process improvement involves analyzing the current software acquisition processes for deficiencies and putting new and/or modified processes in place to correct those deficiencies. Process improvement can be done informally or with the use of a formal model. The SEI has recently developed a formal model for software acquisition process improvement, the Software Acquisition Capability Maturity Model® (SA-CMM®).

Data collected in other domains (e.g., software engineering, manufacturing) have shown process improvement efforts to be very cost effective. While data on software acquisition process improvement do not yet exist, similar benefits are expected to apply to this domain. For example, improvements may result in fewer overruns due to greater predictability in software cost and schedule and reduced life cycle costs due to higher-quality software products being delivered. Furthermore, high-quality software acquisition processes have a beneficial effect on the developer's adherence to well-disciplined software engineering processes.

Software acquisition process improvement should be applied by an acquisition organization. This results in the improvement of the acquisition-level organizational processes and is a necessary application of the technique. However, acquisition process improvement is most effective when it is applied both at the individual acquisition project level and at the acquisition organizational level. Application within an acquisition project can result in highly effective, immediate improvements to the processes being used on the project. Application at the acquisition organizational level results in the overall improvement of organizational acquisition processes, which is of principal benefit to future acquisition projects. In addition, acquisition process improvement will not be effective unless the improved acquisition processes are rigorously adhered to by the acquisition projects.

2.2.7 Software Acquisition Risk Management

The best practice of risk management applies to software acquisition as well as to software engineering. Software acquisition risk management is an effective mechanism for reducing the impact of potential problems on the acquisition of a software-intensive system. Software acquisition risk management involves a continuous process of risk identification, assessment, prioritization, mitigation, and control throughout the life cycle of the software acquisition project, from mission needs identification through retirement.

The use of software acquisition risk management enables a software acquisition project to understand its risks (i.e., future problems that might occur) and, where possible, to mitigate the risks that are assessed to have the largest potential impact on the software acquisition effort. Effective software acquisition risk management can result in reducing the cost and schedule impact of problems by having alternative solutions or workarounds identified before problems occur. Software acquisition risk management by the Government's software acquisition team is necessary in addition to software engineering risk management by the developer's software engineering team. The developer's software engineering risks are almost always software acquisition risks. However, the program acquiring the software-intensive system generally has additional risks as well (e.g., risks related to reduction in staffing of the Government program office and their technical support contractors, risks related to program milestones). Furthermore, the Government's software acquisition team frequently will identify additional software development risks and will assess the importance of the developer-identified software development risks differently than the developer. Independent risk assessment of the software development risks by the Government's software acquisition team is an effective technique for identifying and assessing risks in the software development project.

To be effective, the software acquisition risk management process must be practiced by all software acquisition personnel and at all levels of the software acquisition project. Large programs frequently have a Government risk management process at the program level with the top program risks being closely monitored by Government program management. Just as frequently, however, the only risk management process practiced on large, complex software-intensive programs is the program level process. An effective software acquisition risk management process must be practiced at every level of the software acquisition project, including the lowest-level software acquisition team. Each software acquisition team should use the software acquisition risk management process to identify its risks, mitigate and control those risks that are within its scope of control, and elevate those risks with sufficiently high impact to higher-level Government program management.

2.2.8 Software Quality Incentives

The use of award fees and other incentives can positively motivate the development contractor to use software engineering best practices. Award fee is generally given at predetermined time intervals throughout the contract duration. To apply this best practice, the award fee criteria contained in the award fee plan need to include the quality of the software products produced during each award fee period. The award fee criteria also need to include an evaluation of the degree of compliance with documented software engineering processes during the award fee period and of the effectiveness of those processes. Incentive fees based on performance of the delivered software product in its operational environment can also be used to positively motivate the use of software engineering best practices during development. Incentive fees post-delivery can be based upon system performance measures such as system reliability and availability that include both hardware and software. They can also be based upon defect removal effectiveness measures such as the ratio of defects found during development to the defects found during operations.

Frequently, programs incentivize meeting target cost and target schedule without also incentivizing quality. The more constrained the contractual target cost and schedule, the more likely there are to be large incentives dependent upon meeting those constraints. Incentivizing target cost and schedule without also incentivizing product quality sends the message to the developer that product quality is not important to the Government. The developer will then be motivated to take shortcuts in their software engineering processes to earn the associated incentives, such as by eliminating the quality checks and reducing the robustness of the test program. The Government should be especially concerned about cost and schedule incentives that result in infeasible cost and schedule constraints for the software development effort.

Effective application of this best practice requires commitment from the Government in their execution of the software quality incentives. The Government must be willing to allocate sufficient amounts of funds to the software quality portion of the award fees and incentives so that the contractors are significantly rewarded for use of software engineering best practices. In addition, they should also ensure that the contractors are significantly penalized for process non-compliance and poor quality software products. Independent technical reviews of software products and processes by the Government software acquisition team are effective mechanisms for providing input on software product and process quality to the award fee process.

2.2.9 Software Systems Acquisition

For software-intensive systems, this best practice involves the inclusion of software acquisition as an integral part of the systems acquisition processes. Software acquisition personnel must be knowledgeable and must participate in the systems acquisition processes throughout the entire life cycle, from mission needs identification through retirement. In addition, software acquisition processes must be consistent and integrated with the systems acquisition processes.

It is very important for software acquisition to be an integral part of the system acquisition's pre-contract award activities, especially in defining the system acquisition and support strategies and in preparing the system performance requirements and request for proposal. Without effective participation of software acquisition knowledgeable personnel in the pre-contract award activities, the selected contractor may not be capable of performing the software development effort, the system performance requirements may not include necessary software-related requirements, and the contract resulting from the procurement may not be structured to encourage the developer to follow well-disciplined software engineering processes and produce high-quality software products. Post-contract award, software acquisition must be an integral part of the system acquisition contract management activities, especially for award and incentive fee determination, to encourage the best software development performance from the developer.

The effective incorporation of software acquisition as an integral part of the systems acquisition processes requires positive action by the Government program management. Government program management must establish an environment where the software acquisition is a highly respected part of the program, equivalent in importance to the hardware acquisition. In addition, the Government program must be structured to provide effective lines of communication, responsibility, and authority among the software acquisition teams and the other program teams involved in the systems acquisition processes.

2.2.10 Software-Inclusive Performance Requirements

This best practice involves the inclusion of software in the specification of the system performance requirements for the software-intensive system to be developed. Examples of performance requirements that should have both hardware and software components specified are reliability, maintainability, and availability; supportability, including testability and integrated system diagnostics; safety; security; mission performance timelines; computer resource reserves; and interoperability, including open systems interface requirements.

Since the system performance requirements become the contractual requirements for the system under development, it is very important that they have a complete system perspective, including both hardware and software. The system performance requirements should not be specified so as to only reflect the hardware contribution. For example, including software in the system RMA requirements will help ensure that the RMA actually experienced during operations will meet the specified requirements. When RMA requirements include only hardware, the operationally experienced RMA will be significantly less than specified in the requirements due to failures caused by software faults. This can result in the software-intensive system not being suitable for operations.

The most effective application of this best practice requires the participation of software acquisition personnel knowledgeable in specifying software-inclusive system performance requirements in the system performance requirements definition process.

3. Managing the Risk in Using COTS and Reuse Software

For the purposes of this paper, the term "COTS software" refers specifically to a software package offered for sale, lease, or license to the general public by a vendor. The term "reuse software," on the other hand, refers to any previously developed (non-COTS) software being incorporated into the software under development. Reuse software includes software previously developed by the development contractor, software furnished by the Government (e.g., from another Government program), and software in reuse libraries. Reuse software may be used with or without modification. It should be emphasized that software that is still under development should not be considered reusable until its development is complete.

The principal expected benefits of using COTS and reuse software are cost and schedule savings due to a reduction in the amount of software to be developed. In addition, higher reliability and maintainability benefits are also expected due to the use of software that, in some sense, has already been proven. The expected extent of these benefits, however, is rarely met due to the current state of the practice in COTS and reuse software.

The reality of using COTS and reuse software is frequently very different from the promised benefits of its use. There is generally less COTS and reuse software that can be used than expected. This is due to such factors as the requirements for the new system not being met, the algorithms not being appropriate for the new system, the design not being compatible with the new architecture, and the reuse software not being designed for reuse. This results in additional newly developed software being required. There is also frequently more new software to be developed than expected due to large amounts of "glue" code being needed in order to integrate the newly developed code, the reuse code, and multiple COTS software packages.

In addition, the COTS or reuse software is frequently unreliable due to latent defects in the code or due to unexpected side effects of unnecessary code that cannot easily be removed or disabled. Reuse software is often difficult to modify and maintain due to poor design and implementation, out-of-date or non-existent documentation, and the presence of obsolete technologies. When COTS software is incorporated into a software-intensive system, there are frequently impacts to the software development effort when new versions of (or patches to) COTS packages are released by the vendors. There are often networks of interdependencies among COTS packages, where one package cannot be updated until one or more other packages are updated. Often these updates are being performed by different vendors on schedules incompatible with project needs.

The bottom line is that the incorporation of COTS and reuse software into software-intensive systems will almost always require more cost and schedule than originally estimated to complete the development effort. This is exacerbated by the demands of the competitive procurement process that cause the amounts of COTS and reuse software bid by the developers to be significantly overestimated and the amounts of integration effort and "glue" code to be significantly underestimated.

In spite of the problems involved, affordability constraints are causing increasing reliance on COTS and reuse software. Software-intensive systems are now too large, complex, and costly to be built completely from scratch. COTS and reuse software, however, must be considered as significant risk areas that must be mitigated and managed throughout the system life cycle. Furthermore, the software acquisition and software engineering best practices described in Section 2 of this paper need to be applied to the full software development effort, including any COTS and reuse software as well as the newly developed software.

One of the significant problems in the use of COTS and reuse software is the lack of a thorough evaluation before the decision is made to incorporate the COTS and reuse software into the system under development. Such an evaluation should be the basis of any decision to use, or not use, particular COTS software packages or reuse software. If, after the evaluation, the decision is made to use the COTS or reuse software, the results of the evaluation will identify specific risks in incorporating that COTS or reuse software into the deliverable software product. Thus, the evaluation will assist in determining risk mitigation efforts that should be carried out. In addition, after the decision is made to use the COTS or reuse software, that software must be the subject of continuous risk management throughout the development life cycle. This includes frequent re-evaluation of the COTS and reuse software as the system and software development progresses. Essential criteria recommended by The Aerospace Corporation for evaluating COTS and reuse software are shown in Table 3-1.

Table 3-1. Recommended Criteria for Evaluating COTS and Reuse Software

Ability to provide required capability and meet required constraints – Ability to satisfy requirements – Ability to achieve necessary performance, especially with realistic operational workloads – Appropriateness of algorithms in the COTS/reuse software for use in the new system – Need for and ability to perform characterization/stress testing to determine actual capabilities and performance
Ability to provide required protection – Safety, security, and privacy
Reliability/maturity – As evidenced by an established track record
Testability – As evidenced by the ability to identify and isolate faults
Interoperability with other system and system-external elements – Compatibility with system interfaces – Adherence to standards (e.g., open systems interface standards)
Suitability for incorporation into the new system architecture – Compatible software architecture and design features – Absence of obsolete technologies – Need for re-engineering and/or additional code development (e.g., wraps, "glue" code) – Compatibility among the set of COTS software packages – Need for prototyping   – To determine compatibility, wraps, "glue" code
Ability to remove or disable features/capabilities not required in the new system – Impact if those features cannot be removed/disabled or are not removed/disabled
Availability of personnel knowledgeable about the COTS/reuse product – Training required – Hiring required
Availability and quality of documentation and source files – Completeness – Accuracy

Acceptability of software product licensing and data rights

- Restrictions on copying/distributing the software or documentation
- License or other fees applicable to each copy
- Acquirer's usage and ownership rights, especially to the source code
  - Ability to place source code in escrow against the possibility of the vendor/developer going out of business
- Warranties available

Maintainability, including:

- Likelihood the software product will need to be changed
- Feasibility/difficulty of accomplishing that change if changes are to be made by the program reusing the software product
  - Quality of design and code
  - Need for re-engineering and/or restructuring
- Feasibility/difficulty of accomplishing that change, if changes are to be made by the vendor or product developer (e.g., for COTS or proprietary software)
  - Priority of changes required by this program versus other changes being made
  - Likelihood that the current version will continue to be maintained by the vendor/developer
  - Impact on the system if the current version is not maintained by the vendor/developer

---

Potential for multiple baselines of the COTS or reuse software product

- Likelihood of modifications being made by the vendor/developer (e.g., a new version being released) after a particular version has been incorporated into the new system
- Feasibility/difficulty of incorporating the new version of the COTS/reuse product into the new system
- Impact if the new version is not incorporated
- Ability of the new architecture to support the evolution of COTS/reuse software products

---

Compatibility of planned upgrades of COTS or reuse software with software development plans and schedules

- Compatibility of planned upgrades with build content and schedules
- Impact on development cost and schedule to incorporate upgrades
- Dependencies among COTS software packages
  - Potential for an incompatible set of COTS packages
  - Potential for schedule delays until all dependent COTS products are upgraded

---

Criticality of the functionality provided by the COTS or reuse software

---

Short- and long-term cost impacts of using the COTS/reuse software

- Amount of management reserve needed in case less COTS/reuse software is usable and more newly developed software is required

---

Technical, cost, and schedule risks and tradeoffs in using the COTS/reuse software

4. Conclusion

This paper recommends a set of software acquisition and software engineering best practices that address the issues raised in the Senate Report. These recommendations are based upon the experience of The Aerospace Corporation in supporting the USAF and the NRO in the acquisition of DoD software-intensive space systems. These software acquisition and software engineering best practices are strongly focused on reducing rework to correct defects in software products since rework is one of the principal concerns of the Senate Report. The Senate Report also encourages the DoD to maximize the use of COTS and reuse software. To address this issue, the paper discusses the risks inherent in such use and presents criteria for the evaluation of COTS and reuse software to assist in identifying and mitigating these risks.

The Department of Defense is the world leader in sponsoring efforts to define and transition software engineering best practices to the DoD developers. Examples of DoD-sponsored organizations are the SEI, an FFRDC whose focus is to improve the state of software practice throughout the defense community; the DoD Software Program Manager's Network, whose focus is to bring about improvements in productivity, quality, timeliness, and user satisfaction by implementing best practices as a foundation for DoD software management; and the DoD Practical Software Measurement organization, whose focus is to provide program managers with the objective information needed to successfully meet cost, schedule, and technical objectives on software intensive programs. Individual services also have similar efforts, for example, the Air Force's Software Technology Support Center, which provides services and support to organizations responsible for software development and/or maintenance, including Government acquisition organizations, Government development/maintenance organizations, and contractors. In addition, the DoD has been the principal force behind the development of the existing commercial software product standards (e.g., open systems interface standards) and process standards (e.g., IEEE/EIA J-STD-016-1995).

Unfortunately, the knowledge of software acquisition and software engineering best practices is ahead of their implementation, both by the Government and by its development contractors. The DoD's acquisition reform initiatives have focused upon acquisition streamlining to reduce the projected program cost and schedule. There has been very little emphasis on the resulting quality of the systems being acquired and on the cost of rework. This is especially the case in the area of software, even though improving software product quality and reducing rework will result in considerable cost and schedule savings over the life of the program. A "contracting for quality" initiative is needed as part of the DoD's acquisition reform efforts to provide a more balanced approach to acquisition reform. This initiative should define acquisition strategies, performance requirements, incentives, and contract provisions designed to ensure that the contractors rigorously apply software engineering best practices in the development of the DoD's software-intensive systems.

The DoD also needs to take the lead in defining and transitioning software acquisition best practices to its acquisition organizations. A "software acquisition improvement" initiative is needed to stimulate software acquisition process improvement in the DoD acquisition organizations. The SA-CMM® is recommended for use to provide a formalized framework for software acquisition process improvement.

References

Air Force Materiel Command, Software Development Capability Evaluation, Volumes 1 and 2, AFMCP 63-103, 15 June 1994.

Air Force Software Technology Support Center, Guidelines for Successful Acquisition and Management of Software Intensive Systems, Version 2.0, June 1996.

Paul Byrnes and Mike Phillips, Software Capability Evaluation (SCE^{SM}) Version 3.0 Method Description, Software Engineering Institute, Carnegie-Mellon University, No. CMU/SEI-96-TR-2, April 1996.

DoD Software Program Manager's Network, The Program Manager's Guide to Software Acquisition Best Practices, Version 2.2, June 1998.

DoD Software Program Manager's Network, The Guidebook of Software Acquisition Questions, Version 1.0, July 1999.

Dorofee, A.; Walker, J.; Alberts, C.; Higuera, R.; Murphy, R.; & Williams, R. Continuous Risk Management Guidebook, Software Engineering Institute, Carnegie Mellon University, 1996.

Jack Ferguson, Jack Cooper, Michael Falat, Mathew Fisher, Anthony Guido, John Marciniak, Jordan Matejceck, and Robert Webster, Software Acquisition Capability Maturity Model® (SA-CMM®), Version 1.01, Software Engineering Institute, Carnegie-Mellon University, No. CMU/SEI-96-TR-020, December 1996.

Brian P. Gallagher, Christopher J. Alberts, and Richard E. Barbour, Software Acquisition Risk Management Key Process Area (KPA) - A Guidebook, Version 1.0, Software Engineering Institute, Carnegie Mellon University, CMU/SEI-97-HB-002, August 1997.

S. K. Hoting and R. J. Costello, Computer Systems Division Software System Metrics Approach, Revision 1, Aerospace Report No. TR-96(8617)-1, September 1996.

IEEE/EIA 12207.0 - 1996, Information Technology - Software Life Cycle Processes.

IEEE/EIA 12207.1 - 1997, Guide for Information Technology - Software Life Cycle Processes - Life Cycle Data.

IEEE/EIA 12207.2 - 1997, Guide for Information Technology - Software Life Cycle Processes - Implementation Consideration.

IEEE/EIA Interim Standard J-STD-016-1995, Standard for Information Technology, Software Life Cycle Processes, Software Development Acquirer-Supplier Agreement, 30 September 1995.

ISO/IEC 12207, Information Technology -- Software Life Cycle Processes, 1 August 1995.

Joint Logistics Commanders Joint Group on Systems Engineering, Practical Software Measurement: A guide to objective program insight, Version 3.1a, April 17, 1998.

Mark C. Paulk, Bill Curtis, Marybeth Chrissis, and Charles V. Weber, Capability Maturity Model® for Software, Version 1.1, Software Engineering Institute, Carnegie-Mellon University, No. CMU/SEI-93-TR-24, February 1993.

The Aerospace Institute, Introduction to Software Acquisition, Course Materials, January 1999.

Appendix A—Mapping Between This Paper and the Senate Report Excerpt

Table A-1 provides a mapping between the recommended software acquisition and software engineering best practices discussed in Section 2 of this paper and the topics addressed in the Senate Report excerpt on "Software Management Improvements" (see Table 1-1 of the body of this report). Also included in Table A-1 is a mapping between the COTS and reuse software information discussed in Section 3 of this paper and the topics addressed in the Senate Report excerpt.

The numbers in the column headings of Table A-1 refer to the item numbers in the Senate Report excerpt (see Table 1-1 in the body of this report). These items are the topics to be discussed by the DoD in the report required by the Senate. A summary of these topics is as follows:

(1) Software risk management

(2) Requirements control and change management

(3) Metrics for product quality, process effectiveness, and problem identification

(4) Metrics for successful fielding

(5) Maximizing use of COTS and reuse software

(6) Portion of expenditures spent on rework

Table A-1. Mapping Between This Paper and the Senate Report Excerpt

White Paper Paragraph	Senate Report Topic						
	Reducing Rework	(1) SW Risk Management	(2) Reqs. Control and Management	(3) Metrics for Development	(4) Metrics for Fielding	(5) Maximizing COTS and Reuse	(6) Rework Expenditures
2.1.1 Binary Quality Gates	X						
2.1.2 Configuration & Change Management	X		X				
2.1.3 Customer/User Involvement	X				X		
2.1.4 Defect Root Cause Analysis	X			X			
2.1.5 Formal Inspections	X						
2.1.6 Iterative Life Cycle Models	X						
2.1.7 Prototyping	X						
2.1.8 Quantitative Cost & Schedule Mgmt.	X			X			X
2.1.9 Realistic Cost & Schedule Estimation	X			X		X	
2.1.10 Requirements Traceability	X		X				
2.1.11 Software Architecture Definition	X					X	
2.1.12 Software Eng. Process Improvement	X						
2.1.13 Software Interface Management	X		X				
2.1.14 Software Metrics	X			X	X		X
2.1.15 Software Process Standards	X						
2.1.16 Software Reliability Engineering	X			X	X		
2.1.17 Software Requirements Definition	X		X				
2.1.18 Software Risk Management	X	X					
2.1.19 Software Systems Engineering	X		X				
2.1.20 Test Coverage Exit Criteria	X					X	
2.2.1 Contractor Capability Evaluation	X	X					
2.2.2 Contractual SW Process Commitment	X	X					
2.2.3 Independent Technical Reviews	X	X					
2.2.4 Realistic Cost & Schedule Constraints	X	X		X		X	
2.2.5 Software Acquisition Metrics	X	X		X	X		X
2.2.6 Software Acq. Process Improvement	X	X					
2.2.7 Software Acq. Risk Management	X	X					
2.2.8 Software Quality Incentives	X	X			X		
2.2.9 Software Systems Acquisition	X	X	X				
2.2.10 Software-Inclusive Performance Reqs.	X	X	X				
3. Managing Risk in Using COTS and Reuse Software	X	X				X	

Appendix B—Acronym and Abbreviation List

®	Registered Trademark
Acq.	Acquisition
AFMCP	Air Force Materiel Command Pamphlet
CASE	Computer-Aided Software Engineering
CCB	Configuration Control Board
CMM®	Capability Maturity Model®
CMU	Carnegie-Mellon University
COTS	Commercial Off-the-Shelf
CSD	Computer Systems Division
DoD	Department of Defense
EIA	Electronics Industries Association
Eng.	Engineering
FFRDC	Federally Funded Research and Development Center
GAO	Government Accounting Office
HB	Handbook
IEC	International Electrotechnical Commission
IEEE	Institute for Electrical and Electronics Engineers, Inc.
ISO	International Organization for Standardization
J	Joint
KPA	Key Process Area
Mgmt.	Management
MIL	Military
MOIE	Mission-Oriented Investigative Experimentation
NRO	National Reconnaissance Office
Reqs.	Requirements
SA-CMM®	Software Acquisition Capability Maturity Model®
SBIRS	Space-Based Infrared System
SBSD	Space-Based Surveillance Division
SCCB	Software Configuration Control Board
SCE^{SM}	Software Capability Evaluation
SDCE	Software Development Capability Evaluation
SDP	Software Development Plan
SEI	Software Engineering Institute
SEPG	Software Engineering Process Group
SETA	Systems Engineering and Technical Assistance
SM	Service Mark
STD	Standard
SW	Software
SW-CMM®	Capability Maturity Model® for Software
TR	Technical Report
U.S.	United States
USAF	United States Air Force