# 16 CRITICAL SOFTWARE PRACTICES
# FOR PERFORMANCE-BASED MANAGEMENT

**Purpose**

This paper outlines the 16 Critical Software Practices that serve as the basis for implementing effective performance-based management of software-intensive projects. They are intended to be used by programs desiring to implement effective high-leverage practices to improve their bottom-line measures—time to fielding, quality, cost, predictability, and customer satisfaction—and are for CIOs, PMs, sponsoring agencies, software project managers, and others involved in software engineering.

The "16 Critical Software Practices for Performance-based Management" and Templates contain the 16 practices (9 best and 7 sustaining) that are the key to avoiding significant problems for software development projects. These practices have been gathered from the crucible of real-world, large-scale, software development and maintenance projects. Together they constitute a set of high-leverage disciplines that are focused on improving a project's bottom line. This document is intended to define the essential ingredients of each best and sustaining practice. These practices are the starting point for structuring and deploying an effective process for managing large-scale software development and maintenance. They may be tailored to the particular culture, environment, and program phases of a program. Of course, these practices cannot help "death march" programs that are expected to deliver under impossible schedule deadlines with inadequate funding and without the required staffing with essential skills.

## PROJECT INTEGRITY

### 1. Adopt Continuous Program Risk Management

**Practice Essentials**

1. Risk management is a continuous process beginning with the definition of the concept and ending with system retirement.
2. Risk management is a program responsibility impacting on and supported by all organizational elements.
3. All programs need to assign a risk officer as a focal point for risk management and maintain a reserve to enable and fund risk mitigation.
4. Risk need to be identified and managed across the life of the program.
5. All risks identified should be analyzed, prioritized—by impact and likelihood of occurrence—and tracked through an automated risk management tool.
6. High-priority risks need to be reported to management on a frequent and regular basis.

**Implementation Guidelines**

1. Risk management should commence prior to contract award and shall be a factor in the award process.
2. The DEVELOPER needs to establish and implement a project Risk Management Plan that, at a minimum, defines how points 3 through 8 will be implemented. The plan and infrastructure (tools, organizational assignments, and management procedures) will be agreed to by the ACQUIRER and the DEVELOPER and need to be placed under configuration management (CM).
3. DEVELOPER and ACQUIRER senior management should establish reporting mechanisms and employee incentives in which all members of the project staff are encouraged to identify risks and potential problems and are rewarded when risks and potential problems are identified early. The ACQUIRER needs to address risk management explicitly in its contract award fee plan, and the DEVELOPER needs to provide for the direct distribution to all employees in furtherance of establishing and maintaining a risk culture.
4. Risk identification should be accomplished in facilitated meetings attended by project personnel most familiar with the area for which risks are being identified. A person familiar with problems from similar projects in this area in the past should participate in these meetings when possible. Risk identification should include risks throughout the life cycle in at least the areas of cost, schedule, technical, staffing, external dependencies, supportability, and maintainability and should include organizational and programmatic political risks. Risk identification need to be updated at least monthly. Identified risks should be characterized in terms of their likelihood of occurrence and the impact of their occurrence. Risk mitigation activities need to be included in the project's task activity network.
5. Both the DEVELOPER and the ACQUIRER should designate and assign senior members of the technical staff as risk officers to report directly to their respective program managers and should charter this role with independent identification and management of risks across the program and grant the authority needed to carry out this responsibility.
6. Each medium-impact and high-impact risk should be described by a complete Risk Control Profile.
7. Periodically updated estimates of the cost and schedule at completion should include probable costs and schedule impact due to risk items that have not yet been resolved.
8. The DEVELOPER and ACQUIRER risk officers need to update the risk data and database on the schedule defined in the Risk Management Plan. All risks intended for mitigation and any others that are on the critical path and their status against the mitigation strategy should be summarized. Newly identified risks should go through the same processes as the originally identified risks.

## 2. Estimate Cost and Schedule Empirically

**Practice Essentials**

1. Initial software estimates and schedules should be looked on as high risk due to the lack of definitive information available at the time they are defined.
2. The estimates and schedules should be refined as more information becomes available.
3. At every major program review, costs-to-complete and rescheduling should be presented to identify deviations from the original cost and schedule baselines and to anticipate the likelihood of cost and schedule risks occurring.
4. All estimates should be validated using a cost model, a sanity check should be conducted comparing projected resource requirements, and schedule commitments should be made.
5. Every task within a work breakdown structure (WBS) level need to have an associated cost estimate and schedule.  These tasks should be tracked using earned value.
6. All costs estimates and schedules need to be approved prior to the start of any work.

**Implementation Guidelines**

1. Estimate the cost, effort, and schedule for a project for planning purposes and as a yardstick for measuring performance (tracking). Software size and cost need to be estimated prior to beginning work on any incremental release.
2. Software cost estimation should be a reconciliation between a top-down estimate (based on an empirical model; e.g., parametric, cost) and a bottom-up engineering estimate.
3. Software cost estimation should also be subjected to a "sanity check" by comparing it with industry norms and specifically with the DEVELOPER's past performance in areas such as productivity and percentage of total cost in various functions and project phases.
4. All of the software costs need to be identified with the appropriate lower-level software tasks in the project activity network.

---

## 3  Use Metrics to Manage

**Practice Essentials**

1. All programs should have in place a metrics program to monitor issues and determine the likelihood of risks occurring.
2. Metrics should be defined as part of definition of process, identification of risks or issues, or determination of project success factors.
3. All metrics definition need to include description, quantitative bounds, and expected areas of application.
4. All programs need to assign an organizational responsibility for identification, collection, analysis, and reporting of metrics throughout the program's life.
5. Metrics information should be used as one of the primary inputs for program decisions.
6. The metrics program needs to be continuous.

**Implementation Guidelines**

1. Every project should have a project plan with a detail activity network that defines the process the team will follow, organizes and coordinates the work, and estimates and allocates cost and schedule among tasks. The project plan needs to include adequate measurement in each of these five categories.
    a. early indications of problems,
    b. the quality of the products,
    c. the effectiveness of the processes,
    d. the conformance to the process, and

       e.   the provision of a basis for future estimation of cost, quality, and schedule.

2. Metrics should be sufficiently broad based. Data should be collected for each process/phase to provide insight into the above 5 categories.
3. To use these metrics effectively, thresholds need to be established for these metrics. These thresholds should be estimated initially using suggested industry norms for various project classes. Local thresholds will evolve over time, based upon experience (see 1.e above). Violation of a threshold value should trigger further analysis and decision making.
4. Examples of data, initial thresholds, and analysis of size, defect, schedule, and effort metrics can be found at http://www.qsm.com.
5. Continuous data on schedule, risks, libraries, effort expenditures, and other measures of progress should be available to all project personnel along with the latest revision of project plans.

---

## 4. Track Earned Value

**Practice Essentials**

1. Earned value project management requires a work breakdown structure, work packages, activity networks at every WBS level, accurate estimates, and implementation of a consistent and planned process.
2. Earned value requires each task to have both entry and exit criteria and a step to validate that these criteria have been met prior to the award of the credit.
3. Earned value credit is binary with zero percent being given before task completion and 100 percent when completion is validated.
4. Earned value metrics need to be collected on a frequent and regular basis consistent with the reporting cycle required with the WBS level. (At the lowest level of the work package, the earned value reporting should never be less frequent than 2 weeks).
5. Earned value, and the associated budgets schedules, and WBS elements need to be replanned whenever material changes to the program structure are required (e.g., requirements, growth, budget changes, schedule issues, organizational change).
6. Earned value is an essential indicator and should be used as an essential metric by the risk management process.

**Implementation Guidelines**

1. Progress towards producing the products should be measured within the designated cost and schedule allocations.
2. THE DEVELOPER should develop and maintain a hierarchical task activity network based on allocated requirements that includes the tasks for all effort that will be charged to the program. All level of effort (LOE) tasks need to have measurable milestones. All tasks that are not LOE should explicitly identify the products produced by the task and have explicit and measurable exit criteria based on these products.
3. No task should have a budget or planned calendar time duration that is greater than the cost and schedule uncertainty that is acceptable for the program. The goal for task duration is no longer than two calendar weeks of effort.
4. Each task that consumes resources needs to have a cost budget allocated to it and the corresponding staff and other resources that will consume this budget. Staff resources should be defined by person hours or days for each labor category working on the task.
5. For each identified significant risk item, a specific risk mitigation/resolution task should be defined and inserted into the activity network.
6. The cost reporting system for the total project needs to segregate the software effort into software tasks so that the software effort can be tracked separately from the non-software tasks.
7. Milestones for all external dependencies should be included in the activity network.
8. Earned value metrics need to be collected for each schedule level and be made available to all members of the DEVELOPER and government project teams monthly. These metrics are: a comparison of Budgeted Cost of Work Scheduled (BCWS), Budgeted Cost of Work Performed (BCWP), and Actual Cost of Work Performed (ACWP). A comparison of BCWP and ACWP, a Cost Performance Index, a Schedule Performance Index, and a To-Complete Cost Performance Index.
9. The lowest-level schedules should be statused weekly.
10. The high-level schedules should be statused at least monthly.
11. Earned value reports should be based on data that is no more than two weeks old.

## 5. Track Defects against Quality Targets

**Practice Essentials**

1. All programs need to have pre-negotiated quality targets, which is an absolute requirement to be met prior to acceptance by the customer.

2. Programs should implement practices to find defects early in the process and as close in time to creation of the defect as possible and should manage this defect rate against the quality target.
3. Metrics need to be collected as a result of the practices used to monitor defects, which will indicate the number of defects, defect leakage, and defect removal efficiency.
4. Quality targets need to be redefined and renegotiated as essential program conditions change or customer requirements are modified.
5. Compliance with quality targets should be reported to customers on a frequent and regular basis, along with an identification of the risk associated with meeting these targets at delivery.
6. Meeting quality targets should be a subject at every major program review.

**Implementation Guidelines**

1. The ACQUIRER and the DEVELOPER need to establish quality targets for subsystem software depending on its requirements for high integrity. A mission-critical/safety-critical system may have different quality targets for each subsystem component. System Quality Assurance needs to monitor quality targets and report defects as per the Quality Plan.
2. Quality targets can be under change control and established at the design, coding, integration, test, and operational levels.
3. Quality targets should address the number of defects by priority and by their fix rate.
4. Actual quality or defects detected and removed should be tracked against the quality targets.
5. Periodic estimates of the cost and schedule at completion should be based on the actual versus targeted quality.

---

## 6. Treat People-as the Most Important Resource

**Practice Essentials**

1. A primary program focus should be staffing positions with qualified personnel and retaining this staff through the life of the project.
2. The program should not implement practices (e.g., excessive unpaid overtime) that will force voluntary staff turnover.
3. The staff should be rewarded for performance against expectations and program requirements.
4. Professional growth opportunities such as training should be made available to the staff.
5. All staff members need to be provided facilities, tools, and work areas adequate to allow efficient and productive performance of their responsibilities.
6. The effectiveness and morale of the staff should be a factor in rewarding management.

**Implementation Guidelines**

1. DEVELOPER senior management needs to work to ensure that all projects maintain a high degree of personnel satisfaction and team cohesion and should identify and implement practices designed to achieve high levels of staff retention as measured by industry standards.  The DEVELOPER should employ focus groups and surveys to assess employee perceptions and suggestions for change.
2. DEVELOPER senior management should provide the project with adequate staff, supported by facilities and tools to develop the software system efficiently.  Employee focus groups and surveys should be used to assess this adequacy.
3. The training of DEVELOPER and ACQUIRER personnel should include training according to a project training plan in all the processes, development and management tools, and methods specified in the software development plan.
4. The DEVELOPER and the ACQUIRER should determine the existing skills of all systems, software, and management personnel and provide training, according to the needs of each role, in the processes, development and management tools, and methods specified in the Software Development Plan (SDP)
.

## CONSTRUCTION INTEGRITY

### 7. Adopt Life Cycle Configuration Management

**Practice Essentials**

1. All programs, irrespective of size, need to manage information through a preplanned configuration management (CM) process.
2. CM has two aspects: formal CM, which manages customer-approved baseline information, and development CM, which manages shared information not yet approved by the customer.
3. Both formal and development CM should uniquely identify managed information, control changes to this information through a structure of boards, provide status of all information either under control or released from CM, and conduct ongoing reviews and audits to ensure that the information under control is the same as that submitted.
4. The approval for a change to controlled information must be made by the highest-level organization which last approved the information prior to placing it under CM.
5. CM should be implemented in a centralized library supported by an automated tool.
6. CM needs to be a continuous process implemented at the beginning of a program and continuing until product retirement.

**Implementation Guidelines**

1. CM plans need to be developed by the ACQUIRER and the DEVELOPER to facilitate management control of information they own.  The CM procedures of the ACQUIRER serve as the requirements for the CM plan that describes and documents how the DEVELOPER will implement a single CM process.  This plan should control formal baselines and will include engineering information, reports, analysis information, test information, user information, and any other information approved for use or shared within the program.  The CM process should include DEVELOPER-controlled and -developed baselines as well as ACQUIRER-controlled baselines.  It should also include release procedures for all classes of products under control, means for identification, change control procedures, status of products, and reviews and audits of information under CM control.  The CM plan needs to be consistent with other plans and procedures used by the project.
2. The two types of baselines managed by CM are developmental and formal.  Developmental baselines include all software, artifacts, documentation, tools, and other products not yet approved for delivery to the ACQUIRER but essential for successful production.  Formal baselines are information/products (software, artifacts, or documentation) delivered and accepted by the ACQUIRER.  Developmental baselines are owned by the DEVELOPER while formal baselines are owned by the ACQUIRER.
3. All information placed under CM as a result of meeting task exit criteria need to be uniquely identified by CM and placed under CM control.  This includes software, artifacts, documents, commercial off-the-shelf (COTS), government off-the-shelf (GOTS), operating systems, middleware, database management systems, database information, and any other information necessary to build, release, verify, and/or validate the product.
4. The CM process should be organizationally centered in a project library.  This library will be the repository (current and historical) of all controlled products.  The ACQUIRER and the DEVELOPER will implement an organizationally specific library.  The library(s) will be partitioned according to the level of control of the information.
5. All information managed by CM is subject to change control.  Change control consists of:
   a. Identification
   b. Reporting
   c. Analysis
   d. Implementation
6. The change control process needs to be implemented through an appropriate change mechanism tied to who owns the information:
   a. Change control boards, which manage formal baseline products.
   b. Interface boards, which manage jointly owned information
   c. Engineering review boards, which manage DEVELOPER-controlled information.

7.  Any information released from the CM library should be described by a Version Description Document (Software Version Description under 498).  The version description should consist of any inventory of all components by version identifier, an identification of open problems, closed problems, differences between versions, notes and assumptions, and build instructions.  Additionally, each library partition should be described by a current version description that contains the same information.

## 8.  Manage and Trace Requirements

**Practice Essentials**

1.  Before any design is initiated, requirements for that segment of the software need to be agreed to.
2.  Requirements tracing should be a continuous process providing the means to trace from the user requirement to the lowest level software component.
3.  Tracing shall exist not only to user requirements but also between products and the test cases used to verify their successful implementation.
4.  All products that are used as part of the trace need to be under configuration control.
5.  Requirements tracing should use a tool and be kept current as products are approved and placed under CM.
6.  Requirements tracing should address system, hardware, and software and the process should be defined in the system engineering management plan and the software development plan.
.
**Implementation Guidelines**

1.  The program needs to define and implement a requirements management plan that addresses system, hardware, and software requirements.  This plan should be linked to the SDP.
2.  All requirements need to be documented, reviewed, and entered into a requirements management tool and put under CM.  This requirements information should be kept current.
3.  The CM plan should describe the process for keeping requirements data internally consistent and consistent with other project data.
4.  Requirements traceability needs to be maintained through specification, design, code, and testing.
5.  Requirements should be visible to all project participants.

## 9.  Use System-Based Software Design

**Practice Essentials**

1.  All methods used to define system architecture and software design should be documented in the system engineering management plan and software development plan and be frequently and regularly evaluated through audits conducted by an independent program organization.
2.  Software engineering needs to participate in the definition of system architectures and should provide an acceptance gate before software requirements are defined.
3.  The allocation of system architecture to hardware, software, or operational procedures needs to be the result of a predefined engineering process and be tracked through traceability and frequent quality evaluations.
4.  All agreed to system architectures, software requirements, and software design decisions should be placed under CM control when they are approved for program implementation.
5.  All architecture and design components need to be approved through an inspection prior to release to CM.  This inspection should evaluate the process used to develop the product, the form and structure of the product, the technical integrity, and the adequacy to support future applications of the product to program needs.
6.  All system architecture decisions should be based on a predefined engineering process and trade studies conducted to evaluate alternatives.

**Implementation Guidelines**

1. The DEVELOPER should ensure that the system and software architectures are developed and maintained consistent with standards, methodologies, and external interfaces specified in the system and software development plans.
2. Software engineers need to be an integral part of the team performing systems engineering tasks that influence software.
3. Systems engineering requirements trade studies should include efforts to mitigate software risks.
4. System architecture specifications need to be maintained under CM.
5. The system and software architecture and architecture methods need to be consistent with each other.
6. System requirements, including derived requirements, need to be documented and allocated to hardware components and software components.
7. The requirements for each software component in the system architecture and derived requirements need to be allocated among all components and interfaces of the software component in the system architecture.

**10. Ensure Data and Database Interoperability**

**Practice Essentials**

1. All data and database implementation decisions should consider interoperability issues and, as interoperability factors change, these decisions should be revisited.
2. Program standards should exist for database implementation and for the data elements that are included.  These standards should include process standards for defining the database and entering information into it and product standards that define the structure, elements, and other essential database factors.
3. All data and databases should be structured in accordance with program requirements, such as the DII COE, to provide interoperability with other systems.
4. All databases shared with the program need to be under CM control and managed through the program change process.
5. Databases and data should be integrated across the program with data redundancy kept to a minimum.
6. When using multiple COTS packages, compatibility of the data/referential integrity mechanisms need to be considered to ensure consistency between databases.

**Implementation Guidelines**

1. The DEVELOPER needs to ensure that data files and databases are developed with standards and methodologies.
2. The DEVELOPER needs to ensure that data entities and data elements are consistent with the DoD data model.
3. All data and databases should be structured in compliance with DII COE to provide interoperability with other systems.
4. Data integrity and referential integrity should be maintained automatically by COTS DBMSs or other COTS software packages.   The DEVELOPER should avoid developing its package, if at all possible. Before selecting multiple COTS software packages, the DEVELOPER should study the compatibility of the data/referential integrity mechanisms of these COTS packages and obtain assurance from the COTS vendors first.
5. Unnecessary data redundancy should be reduced to minimum.
6. Data and databases should be integrated as much as possible.  Except data for temporary use or for analysis/report purposes, each data item should be updated only once, and the changes should propagate automatically everywhere.

**11. Define and Control Interfaces**

**Practice Essentials**

1. Before completion of system-level requirements, a complete inventory of all external interfaces needs to be completed.
2. All external interfaces need to be described as to source, format, structure, content, and method of support and this definition, or interface profile, needs to be placed under CM control.
3. Any changes to this interface profile should require concurrence by the interface owners prior to being made.
4. Internal software interfaces should be defined as part of the design process and managed through CM.
5. Interfaces should be inspected as part of the software inspection process.
6. Each software or system interface needs to be tested individually and a test of interface support should be conducted in a stressed and anomalous test environment.

**Implementation Guidelines**

1. All internal and external interfaces need to be documented and maintained under CM control.
2. Changes to interfaces  require concurrence by the interface owners prior to being made.
3. Milestones related to external interfaces should be tracked in the project activity network.
4. Subsystem interfaces should be controlled at the program level.

---

**12. Design Twice, Code Once**

**Practice Essentials**

1. All design processes should follow methods documented in the software development plan.
2. All designs need to be subject to verification of characteristics, which are included as part of the design standards for the product produced.
3. All designs should be evaluated through a structured inspection prior to release to CM. This inspection should consider reuse, performance, interoperability, security, safety, reliability, and limitations.
4. Traceability needs to be maintained through the design and verified as part of the inspection process.
5. Critical components should be evaluated through a specific white-box test level step.
6. Design can be incrementally specified when an incremental release or evolution life cycle model is used provided the CM process is adequate to support control of incremental designs and the inspection process is adapted to this requirement.

**Implementation Guidelines**

1. When reuse of existing software is planned, the system and software architectures should be designed to facilitate this reuse.
2. When an incremental release life cycle model is planned, the system and software architectures need to be completed in the first release or, at most, extended in releases after the first without changes to the architecture of previous releases.
3. The system and software architectures will be verified using methods specified in the SDP. This verification will be conducted during a structured inspection of the software architecture and will include corroboration that the architecture will support all reuse, performance, interoperability, security, safety, and reliability requirements. The architecture will be under CM.

---

**13. Assess Reuse Risks and Costs**

**Practice Essentials**

1. The use of reuse components, COTS, GOTS, or any other non-developmental items (NDI) should be treated as a risk and managed through risk management.
2. Application of reuse components, COTS, GOTS, or any other NDI will be made only after successful completion of a NDI acceptance inspection. This inspection needs to consider the process used to develop it, how it was document, number of users, user experience, and compliance with essential program considerations such as safety or security.
3. Before a decision is made to reuse a product or to acquire COTS, GOTS, or NDI, a complete cost trade-off should be made considering the full life cycle costs, update requirements, maintenance costs, warranty and licensing costs, and any other considerations which impact use of the product throughout its life cycle.
4. All reuse products, COTS, GOTS, or NDI decisions should be based on architectural and design definitions and be traceable back to an approved user requirement.
5. All reuse components, COTS, and COTS need to be tested individually first against program requirements and in an integrated software and system configuration prior to release for testing according to the program test plan.
6. Reuse, COTS, GOTS, and NDI decisions will be continuously revisited as program conditions change.

**Implementation Guidelines**

1. The DEVELOPER will establish a reuse plan for the integration of COTS, GOTS, and in-house software. This plan needs to include discussion and allocation of whom and by what process reused software code is tested, verified, modified, and maintained.

2. The reuse plan should be in the SDP and document an approach for evaluating and enforcing reused functionality against system requirements.
3. The reuse plan should suggest a system engineering process that identifies software requirements by taking existing, reusable software components into account.
4. The test plan should identify the testing of the integrated reused code.
5. When integrating COTS, GOTS, and in-house software, ensure accurate cost estimation of integrating the reused code into the system.  The cost of integrating unmodified reused code is approximately one-third the cost of developing code without reuse.
6. The DEVELOPER and the ACQUIRER need to be able to plan for the estimated costs of obtaining the necessary development and run-time licenses over the system's life cycle and the maintenance/support critical to the product, including source code availability.

---

## PRODUCT STABILITY AND INTEGRITY

### 14.  Inspect Requirements and Design

**Practice Essentials**

1. All products that are placed under CM and are used as a basis for subsequent development need to be subjected to successful completion of a formal inspection prior to its release to CM.
2. The inspection needs to follow a rigorous process defined in the software development plan and should be based on agreed-to entry and exit criteria for that specific product.
3. At the inspection, specific metrics should be collected and tracked which will describe defects, defect removal efficiency, and efficiency of the inspection process.
4. All products to be placed under CM should be inspected as close to their production as feasible.
5. Inspections should be conducted beginning with concept definition and ending with completion of the engineering process.
6. The program needs to fund inspections and track rework savings.
.
**Implementation Guidelines**

1. The DEVELOPER will implement a formal, structured inspection/peer review process that  begins with the first system requirements products and continue through architecture, design, code, integration, testing, and documentation products and plans.  The plan needs to be documented and controlled as per the SDP.
2. The project should set a goal of finding at least 80 percent of the defects in every product undergoing a structured peer review or other formal inspection.
3. Products should not be accepted into a CM baseline until they have satisfactorily completed a structured peer review.
4. The DEVELOPER needs to collect and report metrics concerning the number of defects found in each structured peer review, the time between creating and finding each defect, where and when the defect was identified, and the efficiency of defect removal.
5. Successful completion of inspections should act as the task exit criteria for non-Level-of-Effort earned value metrics (and other metrics used to capture effectiveness of the formal inspection process) and as gates to place items under increasing levels of CM control.
6. The DEVELOPER should use a structured architecture inspection technique to verify correctness and related system performance characteristics.

---

### 15.  Manage Testing as a Continuous Process

**Practice Essentials**

1. All testing should follow a preplanned process, which is agreed to and funded.
2. Every product that is placed under CM should be tested by a corresponding testing activity.

3. All tests should consider not only a nominal system condition but also address anomalous and recovery aspects of the system.
4. Prior to delivery, the system needs to be tested in a stressed environment, nominally in excess of 150 percent of its rated capacities.
5. All test products (test cases, data, tools, configuration, and criteria) should be released through CM and be documented in a software version description document.
6. Every test should be described in traceable procedures and have pass-fail criteria included.

**Implementation Guidelines**

1. The testing process must be consistent with the RFP and the contract. The award fee should incentivize implementation of the testing practices described below.
2. The ACQUIRER and DEVELOPER need to plan their portion of the test process and document this plan with test cases and detailed test descriptions. These test cases should use cases based on projected operational mission scenarios.
3. The testing process should also include stress/load testing for stability purpose (i.e., at 95% CPU use, system stability is still guaranteed….)
4. The test plan should include a "justifiable testing stoppage criteria." This gives testers a goal. If your testing satisfies these criteria, then the product is ready for release.
5. The test process should thoroughly test the interfaces between any in-house and COTS functionality. These tests should include timing between COTS functionality and the bespoken functionality. The test plans need to pay serious attention to how to demonstrate that, if the COTS software fails, how to test that the rest of the software can recover adequately. This involves some very serious stress testing using fault injection testing.
6. Software testing should include a traceable white-box and other test process verifying implemented software against CM-controlled design documentation and the requirements traceability matrix.
7. A level of the white-box test coverage should be specified that is appropriate for the software being tested.
8. The white-box and other testing should use automated tools to instrument the software to measure test coverage.
9. All builds for white-box testing need to be done with source code obtained from the CM library.
10. Frequent builds require test automation, since more frequent compiles will force quick turnaround on all tests, especially during regression testing. However, this requires a high degree of test automation.
11. A black-box test of integration builds needs to include functional, interface, error recovery, stress, and out-of-bounds input testing.
12. Reused components and objects require high-level testing consistent with the operational/target environment.
13. Software testing includes a separate black-box test level to validate implemented software. All black-box software tests should trace to controlled requirements and be executed using software built from controlled CM libraries.
14. In addition to static requirements, a black-box test of the fully integrated system will be against scenarios—sequences of events designed to model field operation.
15. Performance testing for systems (e.g., performing 10,000 tests/second still yields response times under 2 seconds) should be tested as an integral part of the black-box test process.
16. An independent QA team should periodically audit selected test cases, test traceability, test execution, and test reports providing the results of this audit to the ACQUIRER. (The results of this or similar audits may be used as a factor in the calculation of Award Fee.)
17. Each test developed needs to include pass/fail criteria.

**16. Compile and Smoke Test Frequently**

**Practice Essentials**

1. All tests should use systems that are built on a frequent and regular basis (nominally no less than twice a week).
2. All new releases should be regression tested by CM prior to release to the test organization.
3. Smoke testing should qualify new capability or components only after successful regression test completion.
4. All smoke tests should be based on a preapproved and traceable procedure and run by an independent organization (not the engineers who produced it).
5. All defects identified should be documented and be subject to the program change control process.
6. Smoke test results should be visible and provided to all project personnel.

**Implementation Guidelines**

1. From the earliest opportunity to assess the progress of developed code, the DEVELOPER needs to use a process of frequent (one- to two-week intervals) software compile-builds as a means for finding software integration problems early.
2. It is required that a regression facility that incorporates a full functional test suite be applied with the build strategy.
3. Results of testing of each software build should be made available to all project personnel