STSC

Software Test Technologies Report

August 1994

Gregory T. Daich
Gordon Price
Bryce Ragland
Mark Dawood

Test and Reengineering Tool Evaluation Project

This technical report was prepared by the Software Technology Support Center.

Preface

The purpose of this report is to increase awareness and understanding of software testing technologies in particular, test preparation, test execution, test evaluation, and source code static analysis tools. Use of this report should be the first step in transferring effective software test processes, methods, and tools into practical use. This report was written for organizations responsible for the development and maintenance of computer software. This report explains how the features of current testing technologies can improve software development and maintenance. It includes information about specific products in the marketplace. The information is aimed at those who must make decisions about acquiring advanced technology and prepare their organizations to use it effectively.

This August 1994 Software Test Technologies Report combines the February 1993 Test Preparation, Execution, and Evaluation [TPEE93] Software Technologies Report and the March 1993 Source Code Static Analysis [SCSA93] Technologies Report (Volumes 1 and 2) and updates those versions by providing

Updated test tool taxonomy and test technology information.

Guidance for improving the testing process using the Capability Maturity Model.

Additional testing tools and updated tool information.

Improved Product Sheet format that condenses relevant information.

Removal of redundant information between the TPEE and SCSA reports.

Updated information about STSC products and services.

Updated list of conferences and seminars.

Note that the Product Critiques and tool evaluation characteristics (criteria) have been removed from this technology report. Current Product Critiques and evaluation characteristics can be requested from the STSC at any time. This will permit better STSC service and support in the proper use of the information.

(This page has been intentionally left blank.)

Table of Contents

(This page has been intentionally left blank.)

1 Introduction

This report was written to help software development and support activities (SDSA)[1] identify, evaluate, and adopt effective software testing practices and technologies. There are several types of software testing technologies that can potentially improve an organization's ability to test software to find defects and to determine if the software satisfies its requirements. Section 1 introduces the Software Technology Support Center (STSC) and its mission. Section 2 describes some software testing terms and a classification of tools (taxonomy). Section 3 provides additional information about static analysis technologies. Section 4 presents the STSC's method for evaluating and selecting software test tools. Section 5 discusses improving the testing process using the Software Engineering Institute's (SEI) Capability Maturity Model (CMM).

1.1 The Software Technology Support Center

The STSC's mission is to assist Air Force software organizations with identifying, evaluating, and adopting technologies that will improve

The quality of their software products.

Their efficiency in producing software.

Their ability to accurately predict the cost and schedule of software delivery.

A planned approach is necessary for successful transition. In general, transitioning effective practices, processes, and technologies consist of a series of activities or events that occur between the time a person encounters a new idea and the daily use of that idea. Conner and Patterson's Adoption Curve [CONN82], shown in Figure 1-1, illustrates these activities. After encountering a new process or technology, potential customers of that technology increase their awareness of its usage, maturity, and application. If the process or technology is promising, customers try to better understand its strengths, weaknesses, costs, and applications. These first activities in the Adoption Curve take a significant amount of time.

Promising processes and technologies are then evaluated and compared. To reduce risk, customers usually try new processes or technologies on a limited scale through beta tests, case studies, or pilot projects. A customer then adopts processes or technologies that prove effective. Finally, refined processes and technologies become essential parts of an organization's daily process (institutionalization).

[1] A Software Development and Support Activity is a DoD or military service organization responsible for the software development or support of a designated Mission-Critical Computer Resource (MCCR). Adaptation is based on [MCCR90].

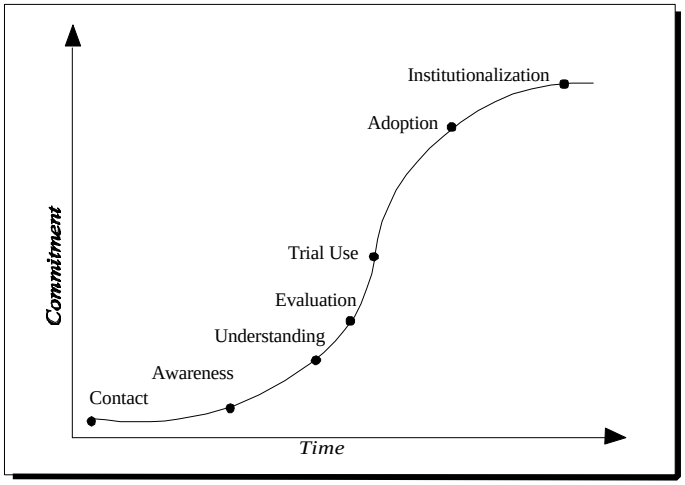

Figure 1-1. Adoption Curve.

Word processors are essential in most organization's daily operations. Yet, 30 years ago they did not exist. The institutionalization of word processors in many organizations followed a series of events similar to those identified in the Adoption Curve.

The STSC is researching and collecting information about technologies that will reduce the time and resources it takes to become aware, understand, evaluate, select, try, and adopt effective practices, processes, and technologies. The STSC has developed the following objectives to accomplish its mission:

Information Exchange

Facilitate the exchange of better software business practices, processes, and technologies within the DoD.

Technology Evaluation

Identify, classify, and evaluate effective processes and technologies.

Insertion Projects

Analyze and improve processes, support the adoption of new methods as needed, evaluate and recommend for selection effective tools, receive and provide appropriate levels of training, and support pilot projects to try out and confirm the technology insertion efforts.

Technology Champions

Develop technology champions who can infuse effective process and technology improvements through the use of available STSC, DoD, and industry products, services, and processes.

1.2 STSC Technology Transition Approach

This section describes the STSC's approach to meeting the objectives identified in the previous section.

1.2.1 Information Exchange

This objective involves exposing potential STSC customers to available technologies and, conversely, customer requirements to technology developers. Referring to the Adoption Curve, this objective focuses on contact, awareness, and understanding. STSC products and services that accomplish this objective include *CROSSTALK* (the DoD journal of defense software engineering), the annual Software Technology Conference, specific technology reports, electronic customer services, and introductory workshops.

1.2.1.1 *CROSSTALK*

Over 13,000 software professionals receive *CROSSTALK* monthly. This publication provides a forum for the exchange of ideas. Articles cover leading edge, state-of-the-art, and state-of-the-practice processes and technologies in software engineering.

1.2.1.2 Software Technology Conference

The annual Software Technology Conference is held each April in Salt Lake City, Utah. This conference brings together over 2,400 software professionals from government, industry, and academia to share technology solutions and exchange ideas and information.

1.2.1.3 Technology Reports

STSC technology reports provide awareness and understanding of each topic in preparation for evaluation and selection of corresponding technologies. Over 34,000 of these reports have been distributed. The current list of reports include the following:

Software Test Technologies Report (this report) .

Configuration Management [CM94].

Documentation [DOC94].

Project Management [PM93].

Reengineering [RE93].

Requirements Analysis and Design [RAD94].

Reuse [REUS94].

Software Engineering Environments [SEE94].

Software Estimation [EST93].

Software Management Guide [SMG93].

The STSC also provides a Metrics Starter Kit [MET94] that helps organizations adopt a measurement program that satisfies the Air Force Software Metrics Policy, 93M-017 [DRUY94].

1.2.1.4 Electronic Customer Services

Along with the services mentioned above, the STSC also provides customers with access to information via Electronic Customer Services (ECS). ECS includes a bulletin board system, which is available to obtain additional information, leave messages, add information, and confer electronically. ECS can be accessed via the INTERNET at address 137.241.33.1 or stscbbs.af.mil or by calling 801-774-6509 with modem at 2400 or 9600 baud, 8 bit word, 1 stop bit, and no parity.

1.2.1.5 STSC Introductory Workshops

Initial visits to potential STSC customers (see Section 1.2.3) can be arranged to introduce STSC products and services to Air Force and other government organizations. The STSC introductory workshops are customized to introduce concepts about specific technologies such as testing, metrics, and program management. An STSC introductory workshop provides valuable input to begin a technology insertion project.

1.2.2 Technology Evaluation

The objective of technology evaluation involves identifying and classifying processes, methods, and technologies that can potentially improve the quality or productivity of software development and maintenance. Many organizations are so focused on deadlines and customer needs that they lack the resources and time to thoroughly investigate options for improvement, leaving them vulnerable to the marketing hype of software vendors. The STSC has developed the infrastructure to provide information on several types of software development and maintenance technologies. Product critiques, essentially, brief tool evaluations from experienced technology users, are collected and distributed. Quantitative evaluations are detailed and objective evaluations performed as needed by experienced users and potential users. Quantitative evaluations are performed on the most promising tools, methods, or processes using the STSC's evaluation guidelines and practices.

1.2.3 Technology Insertion Projects

STSC technology insertion projects are customer-oriented projects that assist customers in evaluating, selecting, and piloting new processes, methods, and technologies. These projects include process definition, process improvement, methodology insertion, and tool insertion and are often initiated through introductory and customized workshops addressing specific technology areas.

Referring to the Adoption Curve (Figure 1-1), an insertion project helps cement understanding of a process or technology, tailors an evaluation of the process or technology for the customer, and pilots the use of that process or technology with appropriate levels of training. Customers move closer to adoption of the process or technology through hands-on experience. It is important to try out technology improvements in a pilot project to confirm that the technology is appropriate for the organization and that the organization is ready and able to adopt the new technology.

1.2.4 Technology Champions

Fowler and Przybylinski [FOWL88] propose that transitioning new technologies from a developer to a consumer requires an advocate to push the technology and a receptor to pull the technology into an organization. This concept is illustrated in Figure 1-2.

Effective change comes from within the organization. The objective of fostering technology champions is to develop technology receptors within individual Air Force SDSAs. These receptors, technology champions, are trained in the use of the STSC's information, products, and services to enhance their organization's ability to incorporate advanced practices, processes, and technologies.

Referring to the Adoption Curve (Figure 1-1), technology champions complete the trek to institutionalization. Champions that come from within the organization should be politically astute and aware of internal organizational requirements. They have the highest probability of influencing the adoption and daily use of effective business practices, processes, and technologies.

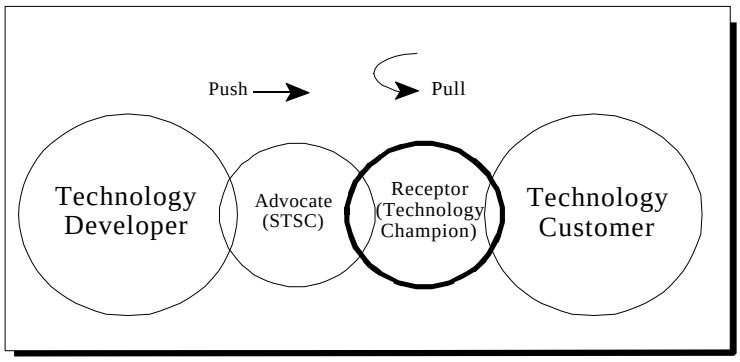

Figure 1-2. Transitioning Technology.

The STSC helps organizations manage their technological change processes by helping them define their strategic and tactical plans. Issues such as management and staff resistance during all improvement phases are addressed using resources from the Software Engineering Institute, etc. Planning for and confronting resistance in an atmosphere of understanding and support will help organizations institutionalize effective processes and tools.

2 Introduction to Software Testing Technologies

The STSC's Test Technologies Group has been researching software testing technologies including practices and tools. The tools include government-owned tools as well as commercial-off-the-shelf (COTS) tools. This report identifies a number of tools that support software testing in typical development and maintenance organizations of the government. Section 2.1 defines important testing terms. Section 2.2 characterizes current practices in software testing. Section 2.3 provides a test tool classification scheme.

2.1 Testing Terminology

Definitions of some commonly used testing terms will help introduce state-of-the-practice software testing technologies. To start with, Bill Hetzel's definition of testing is provided as follows:

> "Testing is any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results" [HETZ88].

This definition advances the concept of testing as a process of executing a program or system with the intent of finding errors [MYER79]. Hetzel suggests that there are many ways to evaluate (or test) a system without executing it. For example, you can test a requirements specification or design document by building test cases based on those specifications or documents. This activity involves testers early on the project and helps correct requirements and design problems before they are coded when they are more expensive to fix. Another point that Hetzel makes is that our intuitive understanding of testing is built on the notion of "measuring" or "evaluating," not trying to find errors. He says, for example, that we don't test students to find out what they don't know, but rather to allow them to demonstrate an acceptable understanding and grasp of the subject matter.

Both views of testing (any evaluation activity and executing code to find errors) are important. In this document, *testing* refers to the act of detecting the presence of faults in code or supporting documentation, or demonstrating their absence by confirming that requirements are met, and is distinguished from debugging where faults are isolated and corrected. This definition of testing, and the following five paragraphs, define several important testing terms and were adapted from a paper entitled, "An Examination of Selected Commercial Software Testing Tools" [IDA92].

An *error* is a mistake made by a software developer. Its manifestation may be a textual problem in the code or documentation called a *fault* or *defect*. A *failure* occurs when an encountered fault prevents software from performing a required function within specified limits.

Four test execution stages are commonly recognized: *unit testing*, *integration testing*, *system testing***,** and *acceptance testing*. In unit testing, each program module is tested in isolation, often by the developer. In integration testing, these modules are combined so that successively larger groups of integrated software and hardware modules can be tested. System testing examines an integrated hardware and software system to verify that the system meets its specified requirements. Acceptance testing is generally a select subset of system test cases that formally demonstrates key functionality for final approval and is usually performed after the system is installed at the user's site.

In *bottom-up* testing, the modules at the bottom of the invocation hierarchy are tested independently using *test drivers*, then modules at the next higher level that call these modules are integrated and tested, and so on. *Top-down* testing starts at the highest-level module, with *stubs* replacing the modules it invokes. These stubs are then replaced by the next lower-level modules, with new stubs being provided for the modules that these call, and so on.

Dynamic analysis approaches rely on executing a piece of software to determine if the software functions as expected. This can involve running the software in a special test environment with stubs, drivers, simulators, test data, and other special conditions or running the software in an actual operating environment with real data and real operating conditions. The effectiveness of any dynamic analysis technique is directly related to the test data used. Current tools attempt to detect faults rather than demonstrate the absence of faults. Additionally, most of these tools can only detect faults whose effects propagate to software outputs, unless the software has been specially instrumented to monitor internal data elements (*intrusive*) or special hardware monitors have been attached to the system (*nonintrusive*).

Static analysis refers to the evaluation of software without executing it using automated (tool assisted) and manual mechanisms such as desk-checking, inspections, reviews, and walkthroughs. Static analyzer tools can demonstrate the absence of certain types of defects such as variable typing errors. Static analysis alone cannot detect faults that depend on the underlying operating environment. Consequently, effective testing requires a combination of static and dynamic analysis approaches.

In support of dynamic analysis, different strategies or heuristics can be used to drive test data generation. Commercial automated support is currently available for both *functional* and *structural* strategies. Functional (*black box*) tests are derived from system-level, interface, and unit-level specifications. Structural (w*hite box*) tests require knowledge of the source code including program structure, variables, or both. With functional strategies, test data is derived

from the program's requirements with no regard to program structure. Functional approaches are language-independent. In structural strategies, test data is derived from the program's structure.

Functional strategies can be applied at all testing levels. System tests can be defined at the requirements analysis phase to test overall software requirements. During the design phase, integration tests can be defined to test design requirements. During the coding phase, unit tests can be defined to test coding requirements.

Figure 2.1 illustrates the software development lifecycle with test development and execution activities. The left side of the figure identifies the specification, design, and coding activities for developing software. It also indicates when the test specification and test design activities can start. For example, the system/acceptance tests can be specified and designed as soon as software requirements are known. The integration tests can be specified and designed as soon as the software design structures are known. And the unit tests can be specified and designed as soon as the code units are prepared.

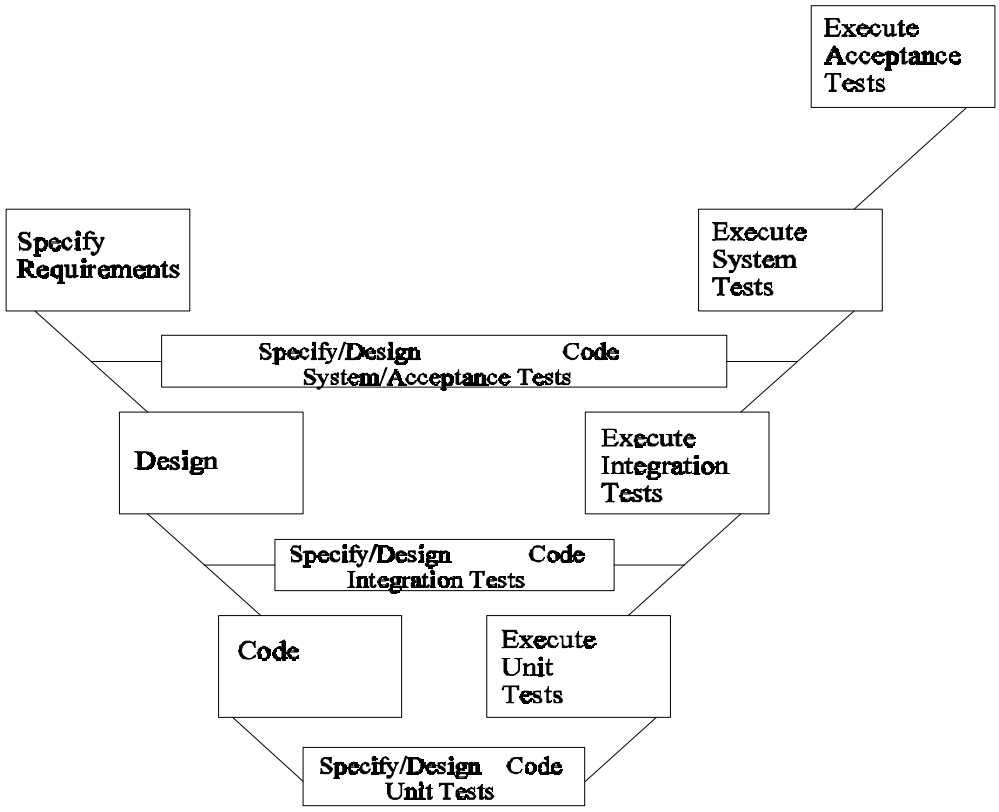

Figure 2-1. Software Development Lifecycle - Modified V Model.

Section 5.3.2.1 suggests that building tests may be the most objective type of testing for requirements and design specifications before code is available to execute. This may also be some of the least expensive testing that we can do. The right side of the figure identifies when the evaluation activities occur that are involved with executing and testing the code at its various stages of evolution.

Requirements-based test case generators create functional test cases by *random*, *algorithmic*, or *heuristic* means that can be applied at all functional test levels. Superior quality test case generators will use all three means. In random or statistical test case generation, the tool chooses input structures and values to form a statistically random distribution. In algorithmic test case generation, the tool follows a set of rules or procedures. Several popular algorithms or methods employed by test case generators include *equivalence class partitioning*, *boundary value analysis*, and *cause-effect graphing*. When generating test cases by *heuristic* or *failure-directed* means, the tool uses information from the tester. Failures that the tester discovered in the past are entered into the tool. The tool uses that history of failures to generate test cases [POST92].

Equivalence class partitioning involves identifying a finite set of representative input values that invoke as many different input conditions as possible [MYER79]. Test cases that exercise boundary conditions usually have a higher payoff than other types of test cases [MYER79]. Building test cases to exercise specific functions supports demonstration of requirements. Regarding cause-effect graphing, causes relate to distinct input conditions, and effects relate to the resulting output conditions or system transformations. Test data is derived from a combinational logic network that represents the logical relationships between causes and effects.

The structural strategies at the unit testing level include *statement coverage*, *branch coverage*, and *path coverage* testing. Statement coverage only detects which statements have been executed and is not considered adequate for structural testing. Branch coverage testing requires each conditional branch statement and the code segment whose execution is controlled by this conditional to be executed at least once. A variation of branch coverage involves exercising all feasible true and false outcomes of each logical component of compounded conditional statements. Path coverage testing requires execution of every path including loops. Path testing is the more stringent strategy but can incur unacceptable computational costs. There are variations of path coverage testing that require basis paths be executed (see [MCCA89] for a complete discussion of basis paths) or that require at least one loop iteration be executed for all loops in addition to all conditionals. A *structural coverage* strategy at the integration level requires that each pair of module invocations be executed at least once.

Prototyping is becoming more widely accepted and implemented as an iterative development activity on many projects. The use of this technique is being accelerated by the availability of more automated tools that enable quicker and easier prototyping of system components. Prototyping evaluates (tests) requirements specifications at the conceptualization phase or the requirements analysis phase and can save a considerable amount of development time when properly managed.

Verification is defined by the MIL-STD-2167A as "the process of evaluating the products of a given software development activity to determine correctness and consistency with respect to the products and standards provided as input to that activity" [216788]. Verification is a testing activity that occurs at all lifecycle phases using the inputs of a phase to evaluate the products of that phase. *Validation* is defined by the MIL-STD-2167A as: "the process of evaluating software to determine compliance with specified requirements" [216788]. Validation ensures that software meets the requirements as specified at all requirements definition phases, i.e., phases that define system requirements, software requirements, design requirements, and coding requirements.

2.2 Software Testing: Current Practices

Many organizations approach software testing with the same practices and tools they used 10 or more years ago. This was evident in a survey conducted by Software Quality Engineering, Inc. of leading software organizations [SOFT90]. While many recommended testing practices are more than 10 years old, there is more tool support available for testing software in today's market. Organizations should regularly review their testing practices and industry practices for potential opportunities for improvement.

Test development, execution, and analysis is still labor-intensive like most development activities. Testing can consume over 50 percent of software development costs (note that testing costs should not include debugging and rework costs). In one particular case, NASA's Apollo program, 80 percent of the total software development effort was incurred by testing [DUNN84]. In general, schedule pressure limits the amount of testing that can be performed. Furthermore, defects frequently lead to failure of operational software. Barry Boehm tells us that 3 to 10 failures per thousand lines of code (KLOC) are typical for commercial software, and 1 to 3 failures per KLOC are typical for industrial software [BOEH88]. With a rate of 0.01 failures per KLOC for its shuttle code, however, NASA has demonstrated that lower defect counts can be achieved [HEND94]. The cost of correcting defects increases as software development progresses for example, the cost of fixing a requirements fault during operation can be 60 to 100 times the cost of fixing that same fault during early development stages [PRES87]. Consequently, timely defect detection is important.

Improved practices and automated tools can reduce testing costs. In addition to eliminating some repetitive manual tasks, tools can promote effective dynamic analysis by guiding the selection of test data and monitoring test executions. Through capturing and reporting data gathered during the performance of testing activities, tools also support quantitative process measurement that is necessary for controlling the testing process. Benefits claimed by some of the tools discussed later include[2]

> A coverage analyzer tool that has saved a developer $15,000 or more per KLOC.
>
> A requirements-based test case generator that has given clients an 8:1 reduction in test development effort, and one client has achieved a reduction from 1.3 to 0.072 failures per KLOC.

Of course, testing tools are not the only mechanism for improving software quality, reliability, and productivity. Software inspections, for example, have been reported to find 60 to 90 percent of software defects, while reducing total development costs by as much as 25 percent [FAGA86].

2.3 Software Test Tool Classification

The STSC has been collecting information from several sources on classifying test tools. This information is being used to help identify, evaluate, and select software test tools. The following books or articles have contributed to STSC's Test Tool Classification Scheme:

> Evaluation and Validation (E&V) Reference Manual [CLAR91].
>
> A Complete Guide to Software Testing [HETZ88].
>
> Testing Tools Reference Guide [DURA93].
>
> CAST Report [GRAH93].
>
> A Complete Toolkit for the Software Tester [POST92].
>
> Software Testing and Evaluation [DEMI87].
>
> An Examination of Selected Commercial Software Testing Tools: 1992 [IDA92].

The STSC Test Tool Classification Scheme is primarily based on the *E&V Reference Manual*, which contains a wealth of information about development functions, maintenance functions, and quality attributes for evaluating Ada Programming Support Environments

[2]**Tool names are omitted since these claims have been neither validated nor invalidated.**

(APSEs). This information is relevant to many development environments besides Ada and identifies lifecycle functions required for development and maintenance using the MIL-STD-2167A.

Some popular terms for software development and maintenance tools such as Computer-Aided Software Engineering (CASE) and Computer-Aided Software Testing (CAST) have evolved over the last few years. Unfortunately, these terms have caused some confusion because they can be interpreted to mean that *any* software tool that supports a software engineering function, such as a compiler, is a CASE tool. Equally, *any* software tool that supports testing, such as a timing analyzer, can be considered a CAST tool. No attempt is made in this report to limit the scope of tools that these terms address. However, as the industry moves toward building Software Engineering Environments (SEEs) or APSEs, the CASE and CAST tools of particular interest and importance are those that can integrate development and testing lifecycle activities or perform multiple related functions within a lifecycle activity.

Some testing tools perform more than one testing function or support more than one lifecycle activity. Since vendors package various capabilities within their tool sets, the same tool can appear under several tool classifications. Multiple classifications are used in tool catalogs such as the *Testing Tools Reference Guide* [DURA93]. The important point here is that software engineers need to be able to find tools that perform the required functions.

The STSC tries to find tools that support all functions in all lifecycle phases of development or maintenance for most major platforms. However, some tools may provide a specific capability, but not be listed under that corresponding tool type. The tool should be recognized by the industry as that specific type of tool. Classification of tools can be very difficult given the variety of published test tool hierarchies, the number and "flavor" of tools in the market, and the terminology used to describe them. The STSC classification scheme is not intended to be complete, nor is it intended to conflict with other schemes. As new technologies emerge and are made commercially available, new types will be added. Lists of tool characteristics are then provided to help customers focus their testing requirements and help them find the right tools.

The industry has accepted the notion of testing at all lifecycle phases. The testing activities may be called by different names but for the purposes of grasping the extent of the role of testing throughout the software lifecycle, all tools that perform a test support or evaluation role are categorized as testing tools. For example, test planning is an essential element of the testing role although it doesn't directly produce test results in and of itself. Testing activities can be initially classified under three major headings:

Test resource management.

Testing at requirements analysis and design phases.

Testing at implementation and maintenance phases.

Figure 2-2 presents the STSC Test Tool Classification Scheme. This classification scheme is not intended to dictate how organizations should classify test tools but rather provides a starting point from which to discuss automated testing capabilities. Tools that manage test resources and several types of tools that support testing at the requirements analysis and design phases are discussed in other STSC technology reports (references are provided below).

This report focuses on software test tools that support the implementation and maintenance phases and, more particularly, those that support source code static analysis, test preparation, test execution, and test evaluation of the actual software system at the unit, integration, system, and acceptance testing levels. Note that requirements-based test case generators and test planners appear under two major headings. This was done to recognize the important role that these tools play during the requirements analysis and design phases and during the implementation and maintenance phases.

2.3.1 Test Resource Management Tools

Management of testing resources is required at all lifecycle phases. The following paragraphs describe several types of test resource managers:

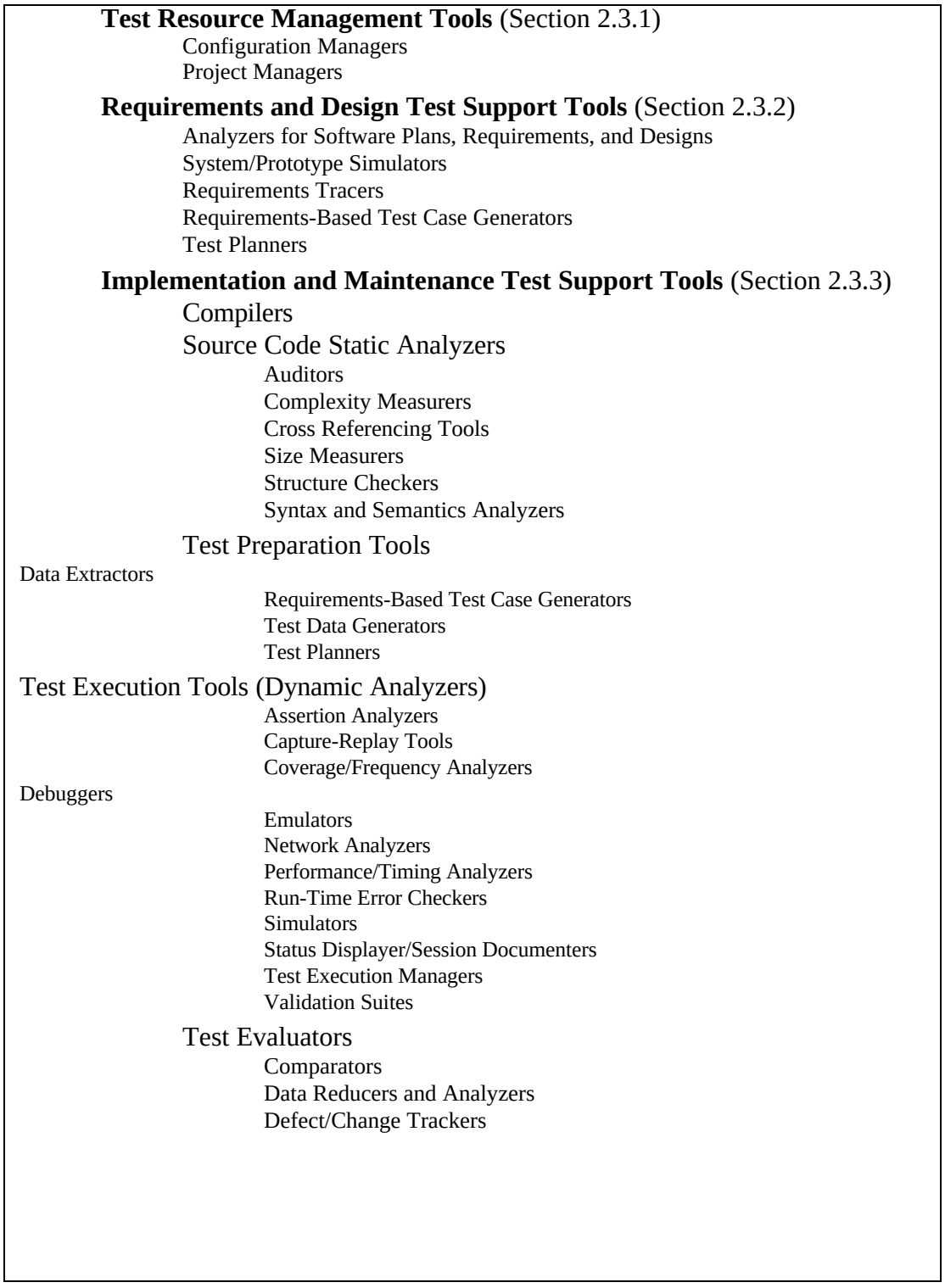

Figure 2-2. STSC Test Tool Classification Scheme.

a. *Configuration managers* monitor and control the effects of changes throughout development and maintenance and preserve the integrity of released and developed versions. Change control of software and test documentation (including test plans, test requirements, test procedures, and test cases) must be carefully managed. Configuration managers can be some of your most powerful testing tools. Code version retrieval, defect

tracking, change request management, and code change monitoring capabilities are typical features of configuration management systems. Configuration management technologies are analyzed and published in the *Configuration Management Technologies Report* [CM94].

b. *Project managers* help managers plan and track the development and maintenance of systems. These tools document the estimates, schedules, resource requirements, and progress of all project activities. Test planning activities are often neglected, delayed, or impacted by delays in early lifecycle phases. Project managers can elevate the role of testing in a project with management's documented commitment (see Section 5, Improving the Testing Process Using the Capability Maturity Model, for more information on this subject). Project management technologies are analyzed in the *Project Management Technologies Report* [PM93].

2.3.2 Requirements and Design Test Support Tools

It is widely acknowledged that testing must be considered at both the requirements analysis and design phases. Software requirements and design information provide primary input to define test requirements and prepare the test plans. CASE tools that support the requirements analysis and design phases are often called Upper-CASE tools. The following paragraphs describe several requirements and design test support tools:

a. *Analyzers for software plans, requirements, and designs* evaluate the specifications for consistency, completeness, and conformance to established specification standards. These tools are reviewed in the *Requirements Analysis and Design Technologies Report* [RAD94].

b. *System/Prototype simulators* merge analysis and design activities with testing. Requirements are refined while obtaining rapid feedback of analysis and design decisions. Requirements can be initially validated, altered, or canceled by demonstrating critical system functions much quicker than is possible under normal full-scale development.

   Prototyping tools allow more efficient consideration of design alternatives, evaluation of user interfaces, and feasibility testing of complex algorithms. Prototyping must be carefully monitored and controlled to ensure that appropriate full-scale-development objectives are considered. Otherwise, prototyping can result in ad hoc development with no documentation. It is anticipated that prototype simulators will be reviewed, and a report will be written as STSC customer interest prescribes.

c. *Requirements tracers* can significantly reduce the work effort of tracing requirements to associated design information, source code, and test cases for large projects. These tools provide links between requirements and design, code, and test cases. What has normally been a lengthy, manual process can be significantly automated using these tools. Requirements tracers are reviewed in the *Requirements Analysis and Design Technologies Report* [RAD94].

d. *Requirements-based test case generators* can help developers evaluate requirements and design information during the requirements analysis, design, and code phases. Early consideration of the types of test cases that a tool builds encourages developers to improve requirements and designs to pass those kinds of tests. See Section 2.3.3.b for more information.

e. *Test planners* assist developers in planning and defining acceptance, system, integration, and unit-level tests. Test cases (including test inputs, expected results, and test procedures) should be defined and designed early in the development lifecycle to help prevent errors. See Section 2.3.3.b for more information.

2.3.3 Implementation and Maintenance Test Support Tools

Some CASE tool vendors advertise full lifecycle support because they provide consistency and completeness checking of the requirements and design specifications, some of which offer no testing support at the implementation and maintenance phases. This has increased development time because unit, integration, and system-level testing needs were not properly considered at the requirements analysis and design phases [SIEG91]. Note that some of these types of tools are built in-house and then discarded (or adapted for other projects) after use. The following paragraphs describe implementation and maintenance test support tools:

a. *Compilers* are not generally considered testing tools, even though testing source code is a major function of these tools. It is anticipated that compilers (particularly Ada compilers) will be reviewed, and a report will be written as STSC customer interest prescribes.

b. *Source code static analyzers* examine source code without executing it. Static analyzers extend the analysis performed by compilers. Various kinds of static analysis tools are available. See Section 3 for more information on static analysis technologies.

(1) *Auditors* analyze code to ensure conformance to established rules and standards. Typical rules and practices include adherence to structured design and coding constructs, use of portable language subsets, or use of a standard coding format [DEMI87].

(2) *Complexity measurers* compute metrics from the source code to determine various complexity attributes associated with the source code or designs written in a program design language (PDL). This is accomplished by evaluating program characteristics such as control flow, operands/operators, data, and system structure.

(3) *Cross referencing tools* provide referencing between various entities. Some of these tools provide a comprehensive on-line cross-referencing capability. Some of the types of data that cross referencing tools provide include cross indexes of statement label, data name, literal usage, and intersubroutine calls.

(4) *Size measurers* count source lines of code (SLOC). SLOC counting tools typically provide counts for comments, executable lines, semicolons, declarations, total

lines, etc. Some of these tools automatically collect code information and provide historical databases that track code growth, changes, and trends.

(5) *Structure checkers* identify some structure anomalies and portray the structure of the source code through graphics or text. Examples of typical charts that are produced are di-graphs (directed graphs), structure charts, data flow diagrams, flowcharts, and call trees. Note that the static data flow and static path flow analyzer tool types identified in [TPEE93] and [SCSA93] have been included with structure checkers to simplify the classification scheme in this report.

(6) *Syntax and semantics analyzers* have been traditionally called static analyzers. Compilers perform these functions but usually with limited scope. FORTRAN syntax and semantic analyzers are often needed to identify type conflicts in calling arguments of separately compiled subroutines. Ada compilers do not have this problem due to the characteristics of the Ada language. Other syntax and semantic analyzers (such as UNIX lint) identify unused variables.

Appendix A.3 contains lists of source code static analyzers.

c. *Test preparation tools* include tools that prepare test data or test case information that may require various levels of follow-on formatting. Also included with the test preparation tools are the test planners.

(1) *Data extractors* build test data from existing databases or test sets. Call the STSC for a current list of data extractors.

(2) *Requirements-based test case generators* help developers evaluate code requirements by building test cases from requirements written following the rules of the tool's formal specification language. (See Sections 2.1 and 2.3.2.d for more information.) Appendix A.3 contains a list of requirements-based test case generators.

(3) *Test data generators* build test inputs that are formatted (or can be readily formatted) in the required files. Some test data generators build statistically random distributed test data sets. (See Section 2.1 for more information.) Call the STSC for a current list of test data generators.

(4) *Test planners* assist developers in planning and defining tests. (See Section 2.3.2.e for more information.) Call the STSC for a current list of test planners.

d. *Test execution tools* dynamically analyze the software to be tested.

(1) *Assertion analyzers* instrument the code with logical expressions that specify conditions or relations among the program variables. Call the STSC for a current list of assertion analyzers.

(2) *Capture-replay tools* automatically record test inputs (capture scripts) and replay those test inputs (playback scripts) in subsequent tests after code changes. These

tools can dramatically improve tester productivity. Some of these tools can fully automate regression testing when combined with the capability to automatically compare previous results with current outputs. Many communications programs (e.g., PROCOMM PLUS) provide scripting capabilities that could support some testing activities such as capture of test sequences. (These tools were not included because testing was not an advertised function.) Appendix A.3 contains a list of capture-replay tools.

(3) *Coverage/frequency analyzers* assess the coverage of test cases with respect to executed statements, branches, paths or modules. These tools generally require instrumentation of the code to be able to monitor coverage; that is, special code is added to monitor the execution paths. (See Section 2.1 for more information.) Appendix A.3 contains a list of coverage/frequency analyzers.

(4) *Debuggers* often directly support the testing effort even though their prime intent is to locate errors resulting from testing. Some debuggers have coverage analysis capabilities that directly support testing. Debuggers can be used to perform various low-level testing functions and are the only test execution tools that some organizations have. Contact the STSC for a current list of debuggers.

(5) *Emulators* may be used in place of missing or unavailable system components. Emulators are generally hardware simulations of various system components that usually operate at the real-time speed of the components being emulated. Terminal emulation is a capability commonly found in communications tools, some of which may be used for testing software. Emulation is usually done for economic or safety reasons. Either the emulated components are not yet available or they are too expensive to waste by permitting some destructive functions to be tested. Call the STSC for a current list of emulators.

(6) *Network analyzers* are a special class of testing tools that draw upon the technologies of several other types of tools. These tools have the capability to analyze the traffic on the network to identify problem areas and conditions. These analyzers often allow you to simulate the activities of multiple terminals. Call the STSC for a current list of network analyzers.

(7) *Performance/timing analyzers* monitor timing characteristics of software components or entire systems. Appendix A.3 contains a list of performance/timing analyzers.

(8) *Run-time error checkers* monitor programs for memory referencing, memory leaking (using memory outside program space), or memory allocation errors. These tools may also automatically monitor the use of stacks and queues. Appendix A.3 contains a list of run-time error checkers.

(9) *Simulators* are used in place of missing or unavailable system components. Simulators are generally software implementations of hardware components in which only the necessary characteristics are simulated in software. Example types of simulators include environmental, functional, and instruction simulators. Like

emulation, simulation is also done for economic or safety reasons. Call the STSC for a current list of simulators.

(10) *Status displayers/session documenters* provide test status information and record selected information about a test run. Call the STSC for a current list of status displayers/session documenters.

(11) *Test execution managers* are a general classification of test tools that automate various functions of setting up test runs, performing a variety of tests, and cleaning up after a test to reset the system. Test execution managers perform many of the same functions as capture-replay tools by automatically executing test cases using scripts and sets of input files. Test execution managers additionally maintain a test results history. This category includes tools that have been termed *test drivers*, *test harnesses*, and *test executives*. Appendix A.3 contains a list of test execution managers.

(12) *Validation suites* validate software against a well-defined standard such as the Ada coding standard ANSI/MIL-STD-1815A, 12 Jan 83, and are often used with compilers and operating systems. Call the STSC for a current list of validation suites.

e. *Test evaluators* include a variety of off-the-shelf and system specific tools that perform time- consuming, error-prone, and boring functions.

(1) *Comparators* compare entities with each other after a software test and note the differences. Capture-replay tools often provide a dynamic comparison capability, which compares entities while the software is under test. Appendix A.3 contains a list of comparators.

(2) *Data reducers and analyzers* convert data to a form that can be more readily interpreted and can sometimes perform various statistical analyzes on the data. Key information can be extracted from execution logs to determine correctness or appropriateness of system behavior [CLAR91]. Call the STSC for a current list of data reducers and analyzers.

(3) *Defect/Change Trackers* keep track of error information and generate error reports. Accounting for requirements and design errors should be considered in these tools. defect/change trackers are often part of configuration management systems, (see Section 2.3.1.a). Also, they can be integrated in Software Engineering Environments to automatically keep track of error information, saving engineers and managers considerable error reporting time. Refer to the *Software Engineering Environments Report* [SEE94] for more information about integrating defect tracking into an SEE. Appendix A.3 contains a list of defect/change trackers.

3 Static Analysis Technologies

A group of testers from the Institute of Defense Analysis identified several static analysis techniques in their paper, *SDS Software Testing and Evaluation: A Review of the State of the Art in Software Testing and Evaluation with Recommended R&D Tasks* [YOUN89]. These techniques complement the static analysis functions listed in the *E&V Reference Manual*. Christine Youngblut categorized the static analysis techniques into four groups:

> "The first group consists of those techniques which produce general information about a program, for example, symbol cross-referencers, rather than search for actual faults. These are relatively common and often provided by a compiler. The second group, static error analysis ... techniques, are designed to detect specific classes of faults or anomalous constructs ... . While some types of static error analysis can be automated, others are restricted to manual application. In contrast, the third group, symbolic evaluation techniques, are entirely automated. The final group consists of manual review techniques, namely code inspections and structured walkthroughs" [YOUN89].

These four techniques can be generally grouped into manual analysis and automated (tool assisted) analysis techniques. Note that the automated tools can be vendor supplied (commercial-off-the-shelf) or can be developed for a specific project. This report reviews several types of source code static analysis technologies.

3.1 Manual Static Analysis

Manual static analysis consists of *inspections*, *reviews* (formal and informal), *walkthroughs*, and *desk checking*. Watts Humphrey, the originator of the Software Engineering Institute's Software Process Maturity Model, expresses the importance of inspections:

> "Software inspections provide a powerful way to improve the quality and productivity of the software process ... . Since we tend not to see evidence that conflicts with our strongly held beliefs, our ability to find errors in our own work is impaired. Because of this tendency, many programming organizations have established independent test groups that specialize in finding problems. Similar principles have led to software inspections. With large-scale complex programs, a brief inspection by competent co-workers invariably turns up mistakes the programmers could not have found by themselves" [HUMP90].

Tom Gilb points out that there are significant differences between inspections, walkthroughs, and reviews.

> "Reviews and walkthroughs are typically peer group discussion activities - without much focus on defect identification and correction ... . Walkthroughs are generally a training process, and focus on learning about a single document. Reviews focus more on consensus and buy-in to a particular document ... . Inspection is a quality

improvement process for written material. It consists of two dominant components; product (document itself) improvement and process improvement (of both document production and Inspection)" [GILB93].

Desk checking is basically an informal, self-checking review technique that is smart practice and should be employed after producing any kind of software or documentation. However, as quoted from [HUMP90] above, desk-checking has a limited potential for finding defects because of our inability to find all errors in our own work. Note that tools are being developed to specifically support software inspections. These types of tools are not yet reviewed in this report.

3.2 Automated Static Analysis

Source code static analysis tools are useful for browsing, measuring, displaying, decomposing, reengineering, and maintaining source code [ARNO90]. Some examples of types of tools that support these functions are listed beside Youngblut's static analysis techniques as follows:

General Information - cross referencing tools, size measurers, complexity measurers.

Static Error Analysis - syntax and semantic analyzers, structure checkers.

Symbolic Evaluation - symbolic evaluators, proofs of correctness.

The types of source code static analysis tools that this document reviews include those tools that support the General Information and Static Error Analysis techniques. Section 2.3.3.a introduced source code static analyzer tools. This section provides additional guidance and motivation for using source code static analyzers. Note that there are very few products that were designed to perform only one major static analysis application; most tools provide multiple capabilities.

3.2.1 Auditors

Source code quality is a primary concern within the software community. Due to this concern, many agencies are establishing Total Quality Management (TQM) policies. The idea of quality software (or quality systems) is not new, but with the complexity of today's systems, it is receiving added emphasis.

Bill Hetzel states, "The cost of quality falls into three broad categories: prevention, appraisal, and failure" [HETZ88]. Lack of attention in the prevention and appraisal categories

implies, there's never enough time to do it right, but there's always enough time to do it over. Tight schedules, little money, and the lack of the appropriate processes and effective tools encourage the overly optimistic view that there won't be defects and failures.

Some automated support is available for analyzing code to ensure that there have been no violations of coding standards such as variable naming conventions, structuring conventions, code nesting depth checks, etc. Auditors can play a significant role in reducing the code inspection effort. Inspection checklists don't need to include checks for characteristics that auditors evaluate. Source code auditors determine adherence to coding standards by automatically comparing the developed code to established coding standards.

3.2.2 Complexity Measurers

Years ago, the only way of measuring the size or complexity of code was to count the number of lines of code. Today, there are still many organizations that use this as their only method of measurement. There is still a lot of discussion about what constitutes a line of code; are comments to be counted? What about blank lines? Knowing the size of a program is still a vital piece of information, and it really doesn't matter what the project (or company) determines constitutes a line of code as long as they are consistent. But don't fall into the trap of trying to compare your productivity, i.e., lines of code/month, against someone else's published metrics; you will probably be comparing apples and oranges.

Counting the lines of code shows how big the module is, but gives little indication of its complexity. Consider the following two lines of code as an example:

```
c = 25;

if (((a <= 0) and ((b > c) or (b < d and c > b))) then
```

Both of the above examples are a single line of code, but the complexity of the second is obviously greater than the first. Complexity of the software can be considered how "busy" the code is. The "busier" the code, the more difficult it is to develop and ultimately to maintain. Some of the benefits of measuring the complexity of the software are

Determining how difficult the code will be to test and maintain. This information can help when allocating testing resources.

Identifying which sections of code should be rewritten.

Obtaining an estimate of the number of defects to expect. Research has shown a correlation between complex code and defects.

Complexity measurers can analyze code and identify potential problem areas that may be too difficult to effectively maintain. Complexity measurers can be used as early as the design phase by analyzing program design language (PDL) and during development to help minimize complexity problems as early as possible.

The industry is beginning to seriously use complexity metrics as additional input to determine the test schedules. The two most widely recognized complexity metrics are McCabe's cyclomatic complexity and Halstead's software science metrics. McCabe's complexity metric can be defined in several ways, the simplest of which is to count the number of decisions in a routine, and add one. Knowing the cyclomatic complexity of the code can help the developer determine the minimum number of independent unit tests that are required to execute every line of code, and every decision path, within a module. Halstead's complexity metrics consist of a series of metrics based on the total and unique numbers of operators and operands. The metrics include theoretical estimates of length, program volume, program level, effort, etc.

3.2.3 Cross Referencing Tools

Interfaces between software modules is another problem that programmers have always had to deal with. Cross referencing tools provide the information so that when a variable is modified in one section of code, the programmer knows all other places that the code will have to be modified. This is also extremely helpful for the maintainers of the code. Some of the capabilities provided by good cross referencing tools are

Allows the programmer to perform cross-referencing on the entire system.

Allows cross-referencing while still within an editing session.

Provides on-line, interactive cross-referencing, and viewing capability.

Note that most good compiler systems provide some cross-referencing capability.

3.2.4 Size Measurers

Because of the interest in tools that measure SLOC and function points as a result of the new Air Force Software Metrics Policy 93M-017, size measures was added to the static analyzer tool classification. The metrics policy requires that all Air Force organizations and contractors that do business with the Air Force identify and collect a minimal set of metrics that have the attributes of size, effort, schedule, quality, and rework for each of their projects that consist of 20,000 SLOC or more.

There are several types of size measurement tools including SLOC estimators, functions point estimators, and actual SLOC measurers. This report focuses on measurement of existing code. Actual SLOC measurers generally count comments, total lines, executable lines, and declarations. Some of these tools provide historical data bases to track code growth and changes. An attribute of some SLOC measurers is the capability to automatically collect SLOC counts at predetermined times. Most code complexity measurement tools provide some SLOC counts.

3.2.5 Structure Checkers

Understanding the structure of the software is another problem that maintenance programmers must deal with. As the code grows and becomes more complex during development, a programmer often has a difficult time remembering which modules are called by other modules. These tools are particularly useful when maintaining or reengineering code. Structure checkers provide increased visibility of systems under development and maintenance. Structure checkers report through graphic or textual means the module call structure. Structure checkers also analyze code structure for "dead code" (not capable of being executed), recursion (in languages like FORTRAN where it is undesirable), and nonstructured constructs, e.g., GO TOs. Note that some code may be logically "dead," and can only be identified by dynamic execution of the code.

3.2.6 Syntax and Semantics Analyzers

Syntax and semantics analyzers enhance the analysis capability of average compilers. Often, when someone thinks of source code static analysis tools, what comes to mind are the functions under this category of tools. *Syntax* is defined as the rules governing the structure of a language, and *semantics* is the relationship of characters and their meanings. Compilers check for syntax of the language to make sure that the rules of the language are being followed. Because of the nature of the language, Ada compilers perform more semantics checking than other language compilers. Syntax and semantic analyzers typically perform the following functions:

Check for uninitialized variables.

Check calling argument compatibility between the calling routine and the routine being called.

Check for certain semantic errors. For instance, a variable that is defined and not used.

4 Evaluation and Selection of Test Technologies

As can be seen from Section 2, there are many types of testing tools. There are also many test tool functions and categories not listed in this publication. This report provides evaluation and selection information about the major types of test tools. Some new tools and testing technologies are appearing in the marketplace and will be included in future releases of this report as information becomes available and as the technology matures or becomes more prevalent.

This section describes the appendices that catalog tool information, conferences and seminars, references, and a glossary of testing terms. This section also addresses the process for evaluating and selecting software testing technologies from a wide assortment of test tool types and their specific offerings. Many methods have been devised to assist with product evaluation and selection. Most of these methods are difficult to follow because some of the information that they recommend is difficult to obtain. Even the task of surveying tools can be difficult and time consuming when starting from scratch. This technology report provides a good place to start the evaluation and selection process by providing

Introductory information about testing technologies and practices.

Classification of test tools.

Guidelines for evaluating and selecting software test tools.

A catalog of candidate tools.

Guidance for improving the testing process using the Software Engineering Institute's Capability Maturity Model.

Test technology references, glossary, and conferences.

Preliminary or brief evaluations (product critiques) from experienced practitioners are also available upon request from the STSC. Finally, detailed evaluations (quantitative evaluations) are available upon request as are the characteristics for performing the detailed evaluations.

4.1 Test Information Catalog

Appendix A contains a test tool catalog (list). Appendix B contains information about STSC's product critique system. Appendices C through E contain additional information about references, testing conferences and seminars, and a glossary of testing terminology, respectively. The following sections further describe the contents of the appendices:

4.1.1 Test Tool Lists

Test tools can be identified by reading trade publications (including periodicals, books, and other tool catalogs), attending conferences, and networking with peers. Appendix A catalogs test tools by tool name, vendor name, and tool type. Note that some types of test tools are not listed in this report because of space limitations. A current list of those tools (or any type of tools on which the STSC has information) can be obtained upon request from the STSC.

Product Sheets by Tool Name. The product sheets are completed by the vendors and include the tool name, the version number, vendor name, phone, fax, E-mail address, vendor address, point of contact, tool types (classifications), languages supported, and platforms/operating systems supported. These sheets also contain product descriptions, pricing, site license availability, number sold, release date, product sheet dates, and an indication of training, user group, newsletter availability. See Appendix A.1. Abbreviated product sheets can also be provided when faxing to interested organizations to minimize the size of the tool information.

Test Tool Lists by Vendor Name. Test tools are reported for each vendor in Appendix A.2.

Test Tool Lists by Test Tool Types. Test tools are reported for each test tool type classification in Appendix A.3.

Some other tool catalogs that are helpful include

Software Quality Engineering, *Testing Tools Reference Guide*, 800-423-TEST [DURA93].

Software Maintenance News, *Software Maintenance Technology Reference Guide*, 415-969-5522 [ZVEG94].

CASE Consulting Group, *CASE Outlook*, 503-245-6880 [FORT91].

Grove Consultants, *CAST Report*, 44-625-616279 (United Kingdom) [GRAH93].

Sentry Publishing Company, *Applications Development Tools Product Guide*, 508-366-2031 [SENT94].

4.1.2 Product Critiques

Product critiques highlight unique or noteworthy tool capabilities and significant or annoying problems. The STSC is soliciting product critiques from experienced practitioners and providing this information upon request. Customers can read opinions from experienced colleagues in a similar manner to the way friends ask each other for their opinions when buying a new car or some other expensive item or service. Contact the STSC for product critiques of any

test tools that may be available. Included in Appendix B is a *CROSSTALK* article soliciting product critiques [PETE92]. The article explains STSC's product critique system and gives instructions for completing the form. We invite all experienced practitioners to participate in the product critique system by completing a product critique form (included in Appendix B) for each software development or maintenance tool used.

4.1.3 References

Appendix C contains cited references to publications that discuss test-related technologies.

4.1.4 Testing Conferences and Seminars

Appendix D contains a list of conferences and seminars that are of particular interest to software testers and quality assurance personnel.

4.1.5 Glossary

Appendix E contains a glossary of software testing technology terms.

4.2 Evaluating Test Technologies

The STSC has adopted a simplified approach to tool evaluation that does not require the use of detailed test evaluation procedures to verify functionality and determine quality characteristics about software products. STSC's recommended test tool evaluation guidelines start by obtaining this technology report, defining and funding the evaluation project, and proceeding as follows:

(1) Identify candidate technologies for information that is desired.

(2) Training is encouraged early for state-of-the-practice testing techniques to obtain more information about available testing technologies that are supported by automated tools. Attendance at testing conferences and seminars is also recommended.

(3) Perform needs analyses to document what test technologies are needed. A defined testing process is highly recommended; however, complete and accurate software requirements are essential in determining specific testing needs for a given system. Candidate tools need to be identified and cost estimates obtained. The basic costs must be estimated for tools, additional hardware and software, training, and integration for the top few candidate tools. Estimates on the impact to the current processes are recommended for those tools that require significant "ramp-up" before they can be used effectively, e.g., coverage analyzers may not require much time to learn and use properly whereas requirements-based test case generators will require training and an associated "learning curve."

(4) Review or request product critiques or any available tool evaluation information on hand at the STSC for an initial look at tool performance and market acceptance.

(5) The top few technology candidates need to be evaluated in more detail if the tools are a major investment in terms of tool costs, support, training, and adoption into the organization. [FIRT87] contains generic software tool questions important for many software development tools. [IEEE92] criteria lists also provide an excellent source of important evaluation characteristics. Scores can be computed in a similar manner to Mosley's Efficient Tool Assessment Methodology [MOSL92] (see Section 4.3.8) or you can use the weighted system provided in [IEEE92]. Evaluations are performed with the user's needs in mind to minimize the evaluation effort. This process ensures that efforts and expenditures are minimized for each evaluation while providing the most current available data.

(6) DoD organizations that require detailed (quantitative) evaluations should contact the STSC. Funding for quantitative evaluations generally would come from the DoD user (requesting) organization. Costs could be amortized depending on user interest in the evaluation information. Expert tool users would then be identified from the product critique authors and would be invited to perform quantitative evaluations based on known user needs. Availability of evaluators will need to be negotiated between the DoD user organization, the STSC, and the evaluating organization.

Evaluators will be provided lists of characteristics that specify the criteria for tool evaluation and scoring. More than one evaluation of a given tool will be sought, and discrepancies will be resolved. Lists of evaluation characteristics can be provided upon request, which cover issues such as ease of use, power, robustness, functionality of the various types of tools, ease of insertion, and the quality of vendor support.

(7) Depending on the amount of the total investment, a special test case may need to be developed. Each tool or technology would then be evaluated by potential users (technology champions) as to how well it supports their testing effort. Special emphasis should be placed on evaluating user interfaces, which may be the only significant difference between various types of testing tools. For example, candidate coverage analyzers may perform the same basic functions, but the number of required commands and the conciseness of the outputs may make one tool far superior to another.

4.3 Selecting Test Technologies

Before selecting and installing the tool, the organization needs to consider the impact that the tool will make on its process. Meaningful metrics need to be accumulated for measuring the performance and quality improvements. Even if a tool is not selected and implemented, organizations should begin collecting metrics that are meaningful for them, and organizational progress should be monitored as required by the new Air Force Software Metrics Policy [DRUY94]. The following issues need to be considered when selecting automated software test technologies:

Test processes

Test lifecycle

Budgets

Personnel

Schedules

Project issues

Training requirements associated with test tools

Tool scoring techniques

Manual testing is defined as the execution of the developed software program or system through the normal user interface and manually observing results. *Computer-assisted* or *automated testing* is defined as the use of software test tools to automatically perform one or more testing functions.

4.3.1 Test Processes

According to the Software Engineering Institute (SEI), all software development organizations should establish a Software Engineering Process Group (SEPG). This group may consist of one technology champion or several representatives of an organization's development team who would be charged with the responsibility of finding ways to improve software development processes. New technologies, including methodologies, techniques, and tools, should be researched and appropriately inserted in a planned, orderly fashion.

Obviously, if an organization's software development process has instituted the use of selected automated test tools, then those tools should be used. When any development phase is upgraded with new technologies, the testing process must be reconsidered since *all* phases

involve some form of testing. When new CASE technologies are inserted to support requirements analysis and design, testing strategies may need to be reworked throughout the development lifecycle.

An important (but easily overlooked) advantage to automated testing is the increase in technological sophistication within the organization that comes with the use of automated tools. The benefits of this increase in skills may be difficult to quantify, but technological sophistication builds on itself. Cultivating the required technical skills is a necessary element of process improvement. Fundamentally, a testing methodology needs to be recognized and uniformly adopted by the organization before supporting tools can be integrated into the testing process; otherwise, the tools may be viewed as an interference to the status quo.

An organization waiting for the perfect set of tools or for prices to come down may become less competitive in its ability to efficiently and effectively perform its testing roles. Many organizations will not use some of the new automated tools on the market today. Organizations that have acquired and used these tools have a major productivity advantage over those that do not.

4.3.2 Test Lifecycle

Since so many software test activities are on the critical path of the development schedule, test development should be scheduled and accomplished as early as possible to be prepared for unit testing, integration testing, system testing, and acceptance testing. This implies that the test effort and software development effort should be started concurrently, and that a software test development lifecycle should be identified and coordinated with the software development lifecycle (for example, see Figure 2-1). Static analysis activities apply to all phases of the software development lifecycle. One published estimate reports that 50 percent or more of the software errors are due to incorrect or modified requirements specifications. It is a well-known fact that software reviews can significantly reduce the number of errors in the later phases of development. Testers should be involved in early analysis and design reviews.

4.3.3 Budgets

Short-term and long-term interests will most likely clash with regard to budgets. In many organizations, management requires quantitative estimates of the return-on-investment (ROI) for new software tool purchases. These are often difficult to obtain without knowing the costs of software to start with. Sometimes subjective justifications are all we have to help persuade management to purchase a new tool. Note that it may be easier to justify large initial costs if the tools and training can be used on other projects. This requires long-term strategic planning that spans many projects and organizations.

4.3.4 Personnel

An often overlooked point is that the test development effort is just that: a development effort, requiring personnel with development skills, tools, and positive attitudes. Test plans, test cases, test software, and test documentation are developed. Software testers have to be very creative in devising ways to increase test effectiveness while reducing effort. Boris Beizer has a good list of the essential characteristics of a good tester in his book, *Software Testing Techniques* [BEIZ90]. A few of these characteristics include not easily intimidated, has integrity, has tenacity, is detail conscious and goal oriented, and must have a certain amount of "cop" in them. Testers, like cops, are rarely rewarded openly and directly for a good job. If they do a good job, it can easily mean added work or embarrassment for someone else.

There is no question that the more sophisticated tools will require an attendant increase in skills. Application domain knowledge and mature development experience provide significant advantages for testers in building effective test cases. Developers and testers then have better communications and confidence in each other. However, the tendency is often to throw together an available group of varied skills and personalities and call it the test group. This lack of personnel consideration will likely result in a project with many difficult problems.

4.3.5 Schedules

Technology acquisition schedules and project planning schedules need to be considered when selecting new test technologies.

4.3.5.1 Technology Acquisition

If schedules do not allow enough time to select the proper new tools and obtain the required training, current techniques must be used. Ad hoc development and manual testing may be effective on very small projects with the right personnel. But if investments in effective technologies are not made on large projects, the organization will probably pay for it during maintenance.

4.3.5.2 Project Planning

Whether automated test tools or manual testing techniques are used, test planning should begin as early in the development lifecycle as possible, i.e., during requirements definition. Testing must be treated and scheduled as an integral part of the development effort. Strong leadership commitment is required to elevate the importance of test preparation if an organization is still not committing test resources until after code is produced.

4.3.6 Project Issues

In large or complex projects, computer-aided testing may be mandatory to automatically generate test cases, to keep track of configurations, to automatically retest changed components, etc. Requirements tracing can easily get out of hand without modern tools. Manual checking of design and code structure can become burdensome. How do you stress test a network that has the capacity to support several hundred nodes? How do you test this network continuously over several days? If the inputs and responses happen too fast for a human to keep up, then computer-assisted testing also is mandatory.

Employee turnover can cripple a project if a mechanism is not in place to retain as much project knowledge as possible. It is very likely that key personnel will leave during a long-term project and take their skills with them. Some automated testing tools can be thought of as skills repositories to preserve critical capabilities during the life of the product. People, for a variety of reasons, cannot be relied upon to completely remember or document all development or test details. Capture-replay tools can record all test inputs (document automatically or permit editing of test scripts). Tests can be run automatically and the results can be compared automatically to the expected outputs with no human intervention.

It may be difficult to justify the cost in time and money of automated test tools and training if the product is a one-of-a-kind or one-time-only product, unless the project's complexity, size, and precision requirements or the future plans for tools dictate automated testing.

4.3.7 Training Requirements Associated with Test Tools

Along with the added sophistication of new technology, come the added costs in time and money for the training necessary to use this technology. There is no easy way around this. The learning curve may require that more immediate projects not be included in the new development strategy. The cost of this technology can be amortized over a large number of projects. There is a good chance that test automation, incorporated with a proper level of software inspections and reviews, will reduce test and rework efforts and budgets over time while providing higher quality systems.

A hasty, ill-considered tool selection will be a costly burden to bear for a long time. This is not a new problem. A word processor or database management system that is not carefully selected can be a long-time thorn in productivity's side. The game is the same, but the stakes are getting higher in the software development world. The costs of a bad tool selection, in retraining time alone, may doom an entire product, organization, or company. Sophisticated technology is a double-edged sword that can help you or hurt you.

Development support activity training (test in particular) is usually low on most priority lists. Training, or more aptly retraining, can be the winning or losing edge in the sophisticated software world of which we are a part. Training cannot be treated as an afterthought. There is too much at stake.

4.3.7.1 Formal Training

Software testing is not a subject that is commonly taught at the college level as a separate subject. If taught at all, it is usually included as part of the general software engineering courses. There are a few specific software testing classes to be found, however, and probably more to come as quality becomes a hotter topic.

4.3.7.2 Vendor Training

Vendors will usually supply some kind of training. The main drawback here is the costs for time, classrooms, and manuals (printed or electronic). Customer-site training is often available and depending on the number of people involved can be cost-effective. Otherwise, individuals can go to the vendor site; however, this requires significant resources. Vendor manuals are the most common form of vendor training. The learning process is slower, but you may learn just what you need to get the job done. Some vendors provide on-line tutorials or computer-based instruction to help reduce the costs of training while broadening the base of potential trainees.

4.3.7.3 Seminars, Conferences, Meetings, and Workshops

Seminars, conferences, meetings, and workshops are taught at more of a generic level, unless the tool in question is very common and possibly dominates its tool domain. These courses can sometimes be more informative than those supplied by the vendor. Costs usually reflect this reality. Some seminars are built around a particular method or tool that is currently getting generous press coverage. Seminars are good for learning about new technologies and are an effective way of obtaining contacts within the industry. Government and industry groups offer a wide variety of testing conferences that deal with all aspects of development and testing.

Appendix D and the STSC's periodical publication, *CROSSTALK*, contain lists of upcoming conferences and seminars that may be of interest to readers.

4.3.7.4 Individual Research

By far, the most common forms of education in the computer world, and testing in particular, are independent reading and consulting with peers. Industry trade journals abound and cover nearly every subject imaginable. Many publications are free if the reader is employed in that subject area, but others can be costly and should be bought by an organization's library for

everyone's benefit. Relations with peers are valuable and should be nurtured. Often, there are personnel in the immediate organization that have first-hand experience who can and should be tapped for information.

4.3.8 Tool Scoring Techniques

All of the tool selection sections under Section 4.3 impact a tool selection decision to a greater or lesser degree. After consideration of these issues, it makes sense to compare similar tools to identify strengths and weaknesses based on the detailed evaluation scores obtained by following the guidelines in Section 4.2. The STSC has defined spreadsheets that automatically sum up weighted evaluation scores. A sample scoring spreadsheet or matrix (with fictitious data) is presented in Figure 4-1. Features of the Tool Scoring Matrix include

All of the first-level descendent characteristics are listed prior to listing lower-level descendants. This makes it easy to view the important issues rather than having to potentially scan several pages to find scores for those first-level descendants. Also, briefing charts can be easily prepared to report the most significant information.

Only ones and zeros were used at the lowest-level descendent characteristics to indicate whether a tool has a capability or not. (The score for a characteristic was left blank if it was not evaluated.) Any scoring range could have been used (depends on user desires) but you may want to keep the scoring range simple.

Weights can be defined according to user desires; however, assigning weights that are not equal at low-level groups is usually subjective and may be difficult to obtain consensus. The sum of the individual characteristic weights appears on the top line of the group under the weight column.

Group scores consist of the summation of the weighted, normalized scores (between 0 and 1) of the descendent characteristics.

Call the STSC for information about computing scores and selecting tools based on evaluation information.

A Comparative Evaluation of Tool_A and Tool_B 31 May 94

ID	CHARACTERISTICS	TOOL_A	TOOL_B	WEIGHTS
	Complexity Measurer	**0.80**	**0.90**	**1**
1	Ease of Use			0
2	Power			0
3	Robustness			0
4	Functionality	0.80	0.90	
5	Ease of Insertion			0
6	Quality of Vendor Support			0
7	Other			0
...				
4	**Functionality**	**0.80**	**0.90**	**2**
4.1	Methodology Support			
4.2	Correctness			
4.3	Types of Metrics	0.60	0.80	1
4.4	Types of Reports			
4.5	Database Support	1	1	1
...				
4.3	**Types of Complexity Metrics**	**0.60**	**0.80**	**5**
4.3.1	McCabe Metrics	1	1	1
4.3.2	Halstead's Metrics	0	1	1
4.3.3	Nesting Depth	1	1	1
4.3.4	Variable Span of Reference	0	1	1
4.3.5	Design Structure	1	0	1
...				

Figure 4-1. Sample Tool Scoring Matrix.

(This page has been intentionally left blank.)

5 Improving the Test Process Using the CMM

Last year's test technology report, [TPEE93], mentioned in its Section 5.2 that the next updated test technology report would provide additional testing guidance at the Software Engineering Institute (SEI) Capability Maturity Model (CMM) Level 2. This section contains guidance for improving test management practices at Level 2 and some test engineering practices at Level 3 through 5. This section identifies testing (or test supporting) activities that exist in the CMM and recommends some supplemental testing activities based on current effective industry practices.

5.1 CMM Background

The CMM provides a formidable tool that software development[3] organizations can use to improve their software processes. It is a software process maturity framework "that describes key elements of an effective software process" [PAUL93]. The CMM defines five maturity levels:

(1) Initial - Ad hoc, occasionally chaotic process.

(2) Repeatable - Disciplined process.

(3) Defined - Standard consistent process.

(4) Managed - Predictable process.

(5) Optimizing - Continuously improving process.

Each maturity level is composed of several Key Process Areas (KPAs), which are related activities "that, when performed collectively, achieve a set of goals at that maturity level" [PAUL93]. Also, the maturity levels suggest an orderly progression to develop effective management practices at Level 2, then improve technical and organizational practices at Level 3. With the foundation of the previous levels, Level 4 initiates statistical processes through establishing improved measurements and process improvement goals and finally, Level 5 focuses on continuous process improvement through defect prevention and improved practices and technologies.

5.1.1 CMM Testing Issues

Software test management practices need to be reviewed early in a process improvement initiative. The CMM tells us that until sound management practices are in place, such as those identified at the Repeatable Maturity Level, "the benefits of good software engineering practices are undermined by ineffective planning and reaction-driven commitment systems" [PAUL93].

[3] Software development also refers to software maintenance in this section.

However, there are some software testing issues that need additional attention in the CMM, particularly in the area of test management. Modern testing principles recognize that software test development parallels software development. Software test development has a lifecycle that starts with planning, proceeds through analysis of test objectives (test requirements) and design of test cases, then follows with implementation (building test scripts, test data, etc.). Evaluation of the test work products (testware: test plans, test procedures, test cases, etc.) occurs at each test lifecycle phase. Some test management and planning issues are either not addressed in the CMM or are embedded in the CMM in several general development guidelines that do not specifically address testing. This can make them difficult to apply to improve software testing.

5.1.2 Example Concerns

The Repeatable Maturity Level does not require the process to have defined test objectives nor a specific test plan to address those objectives. Proper management of test development requires defined test objectives. Software requirements cannot be exhaustively tested under all data input conditions and process states; therefore, test objectives are required to define how conformance to requirements will be evaluated. Test objectives include functions and constraints, software states, input and output data conditions, and usage scenarios to test for at the various testing levels used on the project, e.g., unit, integration, system, acceptance [GELP94]. As an industry, we have learned the importance of defining and managing requirements. We don't want to be making fundamental requirements and design decisions while we're coding the system; similarly, we don't want to be making decisions about test objectives and test designs while we're building (coding) the test procedures and test data.

Obtaining early agreement from the customer organization on the test objectives can avert many problems, such as unprepared and uninformed testers with no testing goals, that can otherwise occur near the end of a project. Test objectives defined early can affect and actually guide requirements definition and analysis, design, and coding. Test cases, including success criteria, are designed from the test objectives. Test objectives are also required to effectively track test status and repeat progress on future projects.

Determining test objectives is a technical activity, but managing those objectives is a test management activity. Unfortunately, a common perception is that most test activities are engineering activities to be addressed at Level 3, and they do not need to be considered in the early stages of process improvement. A CMM-Based Appraisal for Internal Process Improvement (CBA IPI), formerly called Software Process Assessments (SPAs), may ignore test management issues in the resulting process improvement Action Plans. Some software management principles in the Requirements Management KPA need to be applied specifically to

test development. Section 5.3.1.2 lists some recommended test management practices for managing test objectives.

5.2 CMM Benefits

More organizations are recognizing the benefits of using the CMM to improve their software development practices. This section discusses several important benefits of using the CMM to improve software processes, specifically testing processes. Organizations should focus their improvement efforts on "the few issues most critical to software quality and process improvement" [PAUL93]. Improved testing practices can provide significant software quality and process improvements. The following sections discuss several important benefits of using the CMM to improve software testing processes.

5.2.1 Advocate

The CMM is an advocate for software developers and testers who have managers with unreasonable expectations. The CMM is also an advocate for managers who have staff members who practice ad hoc methods with no accountability. Top-level DoD management is committed to achieving higher software process maturity levels [MOSE93]. Developers, testers, and managers will have a greater respect and appreciation for planning and monitoring progress with the adoption of CMM practices.

The CMM advocates senior management sponsorship, written organizational policies, proper levels of training for each activity, and resource availability to do the job. If any of these controls are missing, the project is at risk and will be difficult to complete on time, within budget, and with acceptable quality. As an example, Raymond Dion of Raytheon expressed the importance of training in his organization, "Project managers who once insisted that all training be funded by overhead are now accepting bids that include training costs, because we can demonstrate a direct benefit to the project" [DION93].

5.2.2 Structure

The CMM has an interesting horizontal and vertical structure that encourages solving related problems at each level in priority order as well as evolving software development practices from level to level. While the CMM says that "key process areas have been defined to reside at a single maturity level," many software practices evolve as the organization matures [PAUL93]. For example, defect handling practices mature from policies for dealing with problems at Level 2, to documented practices for tracking and analyzing defects at Level 3, to quantitatively predicting defects at Level 4, to defect prevention at Level 5.

We're encouraged to adopt practices that improve our maturity to the next level since they provide the foundation for further improvements. However, some high-level practices, like setting up a software engineering process group (SEPG) which is a Level 3 activity, exert a maturity-pull effect for Level 1 organizations. "They can facilitate the introduction and acceptance of other practices and so should be introduced early" [CORD93]. Cord's theme is that defect-causal analysis (a Level 3-5 activity) also exerts a maturity-pull effect for Level 1-2 organizations. Being careful not to take on more than we can handle, adopting improved practices (acting mature) is the way we start to improve our processes.

Note that we have to be careful to adopt enough of a recommended practice in order not to develop bad habits. For example, if you don't consider your organization's optimum inspection[4] rate when first introducing peer reviews or inspections, you will have limited success. For peer reviews to be effective, you need to monitor your review rates to determine and abide by your optimum rates [GILB93]. Measuring optimum inspection rates is basically a Level 4 activity, but it is important when first implementing peer reviews.

The internal structure of each KPA helps you consider important issues such as your organization's ability to do the job and relevant configuration management, quality assurance, evaluation, and progress measurement activities to ensure a good job is done. Each KPA contains an opening discussion that describes its purpose. Then, specific goals of the KPA are listed. Finally, a set of Common Features are provided that address the following:

Commitment to perform with required actions to ensure the process is established.

Ability to perform with preconditions that must exist.

Activities performed with roles and procedures to implement the KPA.

Measurement and analysis with associated example measurements.

Verifying implementation with steps to ensure compliance.

[4] Further references to peer reviews include inspections.

5.2.3 Other Test Support Activities

The CMM has recognized the importance of requirements, which may be reflected in the placement of the Requirements Management KPA as the first KPA in the Repeatable Maturity Level.

The CMM includes peer reviews in addition to internal reviews with management and formal technical reviews with the customer. Peer reviews have been accepted as an important industry practice that cost-effectively identifies many defects in the phase they were created and effectively prevents them from being passed on to subsequent phases. If any software or testware is worth building, it is worth planning, analyzing, designing, implementing, and evaluating similar to production software. Note that managers are not included in peer reviews, which helps improve the effectiveness of finding defects in an ego-less, team environment during a specific development phase.

The CMM's key practices were written "in terms of what is expected to be the normal practices of organizations that work on large, government contracts" [PAUL93]. Teamwork is emphasized along with practices that improve the planning, tracking, and oversight management capabilities of the organization.

The CMM has a future. The Software Engineering Institute has a CMM Configuration Control Board that entertains suggestions for improvements. Future editions may begin to address technical and human resource maturity issues. The next release of the CMM may appear in 1996.

5.3 Concerns

An organization improving its software development practices will want to consider the testing process in light of all of its problems in a priority order. Unfortunately, the CMM does not focus much on software test management and planning, especially at the Repeatable Maturity Level. Many fundamental test planning activities are not addressed until Level 3. The following sections address software test management and test engineering concerns. Each section is further divided into subject areas that identify, discuss, and recommend improvements for specific concerns about CMM test practices or concepts that need additional consideration.

5.3.1 Software Test Management Concerns

Improving test planning, estimating, and tracking practices will give management increased visibility and control over our test development, execution, and analysis activities [ROYE93]. As mentioned earlier, the CMM does address some test management activities. However, it is difficult to get a clear picture of the CMM's recommendations because test

management guidelines are embedded within several general software development practices. Many of those practices don't specifically address testing nor do they show enough software test related examples. Beizer states,

> "Although programmers, testers, and programming managers know that code must be designed and tested, many appear to be unaware that tests themselves must be designed and tested - designed by a process no less rigorous and no less controlled than used for code" [BEIZ90].

With this in mind, a Software Test Management KPA is proposed to supplement the CMM at the Repeatable Maturity Level. Subject areas as important as software configuration management and software quality assurance warrant particular attention in their KPA. Since software testing generally involves a separate organization (except for unit-level testing) and because it has separate goals and concerns from software building activities, a separate KPA might better address the needs of managing test development efforts. Also, because separate plans are often required for the various testing levels, managing the development of those plans needs to be clearly outlined and a Software Test Management KPA could help provide the needed attention to this important activity.

The business of writing an additional KPA may seem unnecessary or even contradictory with the CMM; however, note for example, that there is no Database Management KPA because it is not a universal concern to the industry[5]. If you need a Database Management KPA, the SEI recommends that you develop one to support your process.

The following subject areas identify major CMM software test management concerns. The recommendations in these subject areas form a framework for a Software Test Management KPA that could be used by all organizations to better support their testing practices. This framework of recommended practices parallels several CMM Level 2 practices but specifically addresses the test group, the test activities, and the test work products.

5.3.1.1 Risks

Testing is not associated with risk assessment in the Repeatable Maturity Level. Risks and available resources dictate the scope of testing. Identifying risks is the heart of testing and determines test objectives. If you believe something is a risk, you test for it. If you want a group of people to agree about when to stop testing, then consensus on risk assumptions must be obtained early in the development effort. This is fundamental to the management of the project.

[5]**The Software Subcontract Management KPA also does not universally apply but was included because of its importance. Note that there are several practices expected of subcontractors that are not expected of developers.**

"This makes the testers job doable" and increases test effectiveness [GELP94]. Also, the risks and test strategy will mold the test objectives and is fundamental to planning and managing the testing effort. The Software Test Management KPA should require that test objectives and test plans be based on assessed risks.

5.3.1.2 Test Objectives

As mentioned in Section 5.1.2, Example Concerns, test objectives are not adequately addressed at the Repeatable Maturity Level. Some example recommended Software Test Management key practices to manage test objectives (derived from the Requirements Management KPA) include

Goal 1	Test objectives are controlled to establish a baseline for testing software.
Commitment 1	The project follows a written organizational policy for managing test objectives.
Ability 1	For each project, responsibility is established for analyzing software requirements and risks to determine test objectives.
Ability 2	The test objectives are documented.
Ability 3	Adequate resources and funding are provided for managing test objectives.
Activity 1	The software test group and the software engineering group review the test objectives before test cases are designed to meet those objectives.
Activity 2	The software test group uses the software requirements and the test objectives as a basis for test plans, test work products, and testing activities.
Measurement 1	Measurements are made and used to determine the status of the activities for managing test objectives.
Verification 2	The activities for managing the test objectives are reviewed with the test manager and project manager on both a periodic and event-driven basis.

5.3.1.3 Test Planning/Tracking

The Software Project Planning and Software Project Tracking and Oversight KPAs in the Repeatable Maturity Level do not address software test planning, tracking, and oversight specifically enough because the test objectives, test responsibilities, and test work products are not discussed. Examples of software work products at the Repeatable Maturity Level do not include test work products. The only test plan mentioned at Level 2 is the acceptance test plan required by subcontractors.

The CMM basically treats test planning as a Level 3 engineering activity. Effective management at the Repeatable Maturity Level requires fundamental test planning, tracking, and oversight practices that address test objectives, test work products, and test activities [GELP94]. We need more visibility into the test process at the Repeatable Maturity Level. For test development, we need that same process visibility required at the conclusion of each major software development phase. You can't manage what you don't see. Note that one of the major objectives of software quality assurance is to make sure that you are performing as planned. When testing is not planned adequately, the controls are not in place for SQA and management to evaluate conformance and progress.

The CMM and the DOD-STD-2167A prescribe early software project planning. However, many organizations do not take full advantage of early test planning and test development practices [PAUL93] [216788]. Testing is still often regarded as a necessary evil, and detailed planning efforts are delayed until code is available for testing. Brooks told us long ago that "testing is the most mis-scheduled part of development" [BROO75]. As mentioned earlier, test development has a lifecycle. Recognition of that lifecycle will go a long way to improving our capabilities for estimating and scheduling test activities. Brooks also said, "Developers don't remember that they don't understand - they ... happily invent their way through the gaps and obscurities" [BROO75]. This certainly applies to test development. The goal should be to remove as many test development activities from the test execution window as possible. These are test management issues that need to be addressed early in a process improvement program by Level 1 organizations.

Some example recommended Software Test Management key practices for test planning (derived from the Software Project Planning KPA) include

Commitment 1	A software test manager is designated to be responsible for negotiating testing commitments and developing the software test plan.
Ability 2	Responsibilities for developing the software test plan are assigned.
Ability 3	Adequate resources and funding are provided for planning the test activities.
Activity 1	The software test group participates on the project proposal team.
Activity 5	A test development lifecycle is defined and coordinated with the software lifecycle.
Activity 6	The project's test plan is developed according to a documented procedure.

Activity 7 The test plan is documented.

Activity 12 The testing schedule is derived according to a documented procedure.

Activity 13 Software testing focuses on software risks associated with cost, schedule, and technical aspects of the project.

Measurement 1 Measurements are made and used to determine the status of the test planning effort.

Some example recommended Software Test Management key practices for tracking and oversight of testing (derived from the Software Project Tracking and Oversight KPA) include

Commitment 2 The project follows a written organizational policy for managing the test effort.

Ability 2 The test manager explicitly assigns responsibility for test work products and activities.

Activity 8 The project's test schedule is tracked, and corrective actions are taken as necessary.

Activity 9 Test engineering technical activities are tracked, and corrective actions are taken as necessary.

Activity 12 The test group conducts periodic internal reviews to track technical progress, plans, performance, and issues against the test plan.

5.3.1.4 Regression Testing

The requirement for regression testing in the Software Configuration Management KPA is optional. This KPA states that "reviews and/or regression testing are performed to ensure that changes have not caused unintended effects on the baseline" [PAUL93].

Though the intent of the Repeatable Maturity Level deals with being able to repeat the same basic process for other projects with the same likelihood of success, the very concept of repeatability is in question with respect to the project. The importance of regression testing is not adequately addressed at Level 2. The Software Configuration Management guidelines for regression testing and configuration control of test work products may need to be strengthened after changes are made. This will improve our capabilities to repeat our process on current and subsequent projects and will better support reuse of appropriate test work products.

5.3.1.5 Timing of Test Process Improvements

Because testing is treated as a technical activity, important test management issues may not surface as high-priority problem areas when CBA IPIs (or SPAs) are conducted using the CMM as a guide. Phil Koltun, of Harris Corporation, wrote, "The SEI's CMM postpones treatment of software testing until Level 3. That's unfortunate because it discourages less mature organizations from devoting process improvement attention to this vital activity" [KOLT93].

Unless we consider software test management practices early in our process improvement efforts, we may be headed for the same kinds of problems that we experienced in the past as we involved unprepared testers late in our development projects. Effective testing practices improve overall development effectiveness. A Software Test Management KPA will give added emphasis to testing to improve software development capabilities.

5.3.2 Test Engineering Concerns

The role of software testing has evolved in the industry over the last several decades from demonstration, to detection of errors, to evaluation, and finally to defect prevention [GELP94]. Unfortunately, testing's image has remained poor in many organizations because of excessive costs and ineffective practices. An example of this poor image is reflected in the CMM's statement, "During a crisis, projects typically abandon planned procedures and revert to coding and testing" [PAUL93]. Part of this poor image has resulted from testing being burdened with having to include debugging and rework costs. Test costs should not include debugging and rework. Perhaps the CMM should say, "... coding, testing, and rework."

The application of effective test development practices (a CMM Level 3 issue) and appropriate levels of automation of test practices (management and engineering) will reduce costs and improve the quality of our software. The following subject areas describe some CMM test engineering concerns and recommendations for enhancing the CMM to better address effective test engineering practices.

5.3.2.1 Test as a Process Improvement Mechanism

Testing was not specifically mentioned as a process improvement mechanism in the CMM at any CMM level. The example software-related defect prevention groups at CMM Level 5 did not include the software test group.

Many organizations believe that you cannot test quality into software. While this is true after the software is built, it is not true during development. Test development activities can play a significant role in preventing defects during early development phases [GELP94]. In fact, building test cases from requirements may be the most objective and effective mechanism to

evaluate requirements (and determine testability - a Level 2 practice) during requirements analysis and design. Lt Col Mark Kindl considers testing as "the most important method for producing error data necessary to guide process improvement" [KIND92]. Software testers can make significant contributions to prevent defects and should be consulted with respect to process improvement. Note that test process improvement must always include an increased capability to measure development progress.

5.3.2.2 Independence Versus Interdependence

The discussion about the *independence* of the test group needs to be enhanced in the Defined Maturity Level to address an *interdependence* of the test group and the software development group. Independence discourages the partnership that should exist between developers and testers in producing quality software. It also discourages early involvement of testers on the project. Admittedly, evaluation activities conducted by testers must be planned and conducted separately from development activities to be effective. However, there are significant benefits to having an interdependence between the developers and testers to provide a common understanding of requirements and tests [GELP94]. The concern about the developers knowing what the testers will be testing for is unfounded. We want systems that have passed a "comprehensive" set of tests, where comprehensiveness is based on an adequate consideration and coverage of known concerns.

5.3.2.3 The Role of Automation

Because the CMM generally focuses on management concerns at Level 2, some people say that adopting CASE tools is a Level 3 activity. The CMM wisely councils that "the benefits of better methods and tools cannot be realized in the maelstrom of an undisciplined, chaotic process" [PAUL93]. While it is important not to automate chaos, some elements of the software development process, when understood well enough, can be considered for automation early in an improvement effort. The CMM also states that, "Software engineering technical activities are tracked and corrective actions are taken as necessary" [PAUL93]. Correcting involves improvements to activities that use improved practices and tools. Effective testing demands some amount of automation for systems of any appreciable size. Michael Deutsch says "the evaluation of the output data, if performed manually, is likely to be a tedious, time-consuming, and error-prone process for all but the most elementary of tests" [DEUT82].

An important question about software testing isn't whether or not an organization should automate but rather, "Are there technologies and tools available that will enable organizations to effectively test their software and are the tools economical in terms of (1) speeding up the development processes and (2) maximizing the quality of all product releases to minimize future error correction efforts?" These questions need to be asked for each project. Our investigations

have found that there are a number of software test tools that can significantly help testers on a number of platforms and software engineering environments. Many of these tools can be used by organizations with a Level 1 rating. Note that several Level 2 Ability Common Features discuss providing adequate resources and funding, which includes the availability of appropriate tool support.

Tool adoption requires careful planning, evaluation, and trial use. However, not all CASE tools[6] are implemented alike nor do they require the same level of maturity to adopt. Tools that don't cause developers and testers to have to rethink their problems but rather help them better view their problems are good candidate tools in the initial stages of process improvement. Several types of testing tools can be considered early in a process improvement effort and can reduce resistance to organizational change. Jerry Durant believes that, "Test tools should provide immediate benefits" [DURA93]. Some of those benefits are intangible but still very important, such as an increased level of confidence in testing accuracy and comprehensiveness. Some example types of test tools that encourage process improvement are discussed in the next several paragraphs.

a. Defect tracking tools can help manage the test and rework effort at all maturity levels.

b. Static analyzers can automate many code review tasks (and can sometimes give you more diagnostics and code measurements than you might initially care for). Many static analyzers are considered "no brainers" to use (as simple as compilers) and can be adopted at all maturity levels. Note that sophisticated code measurement systems may require that mature practices be in place before adopting them.

c. While many coverage analyzers have found their way to a dusty shelf, coverage analysis is fundamental to the test process and is considered "criminal" not to know how much code was exercised during development [BEIZ90]. Maybe coverage analyzers should be considered at Level 2. However, adopting coverage analyzers will require some organizational changes. Note that coverage analysis was promoted to a CMM Level 3 test engineering activity from a Level 4 measurement activity in the old SEI software process maturity model published in 1987 [HUMP87].

d. Capture/replay and test management tools automate much of the test execution process and can vastly improve regression testing capabilities. But a significant investment may be required to buy and learn the tools before benefits can be derived. Changes may require a major rework of the test scripts, especially in graphical user environments. However, those changes may also require considerable rethinking in a manual or ad hoc testing environment if effective testing is a goal. Note that there are no savings with capture/replay tools during the first time a test is run [GRAH93].

---

[6] CASE tools include tools that support requirements analysis, design, code, test, documentation, etc. [IEEE92].

e. Requirements-based test case generators provide an interesting capability that automates some test development activities at the software requirements analysis and design phases. Perhaps the greatest benefit that these kinds of tools offer is the promotion of gathering and improving requirements information. Test case generators build several classes of test cases (function, logic, boundary value, equivalence partitions, etc.) from available requirements information. These tools may also require a significant investment to adopt in the organization, but the benefits can be very impressive [PATR92].

For every practice, the CMM advocates that adequate resources and funding be provided, and appropriate tools be made available to accomplish the work. SEPG-type organizations are reminded not to forget that test process improvement includes an appropriate level of technology and tool automation improvements. Another maturity-pull effect can be felt with the adoption of appropriate Level 5 Technology Change Management KPA practices when adopting tools.

5.4 Recommendations

A greater appreciation for the CMM will come from studying it and applying its principles. Perhaps the best way to understand something is to try to use it *and* improve it. Every organization should carefully consider the practices advocated by the CMM and adopt those that make sense for them in a planned, orderly progression.

Many of the same management practices used to manage software development can be effectively applied toward the management of test development. Obtaining management commitment and support as the CMM advises is fundamental to a software process improvement effort. The CMM, supplemented with the proposed Software Test Management Key Process Area, may enhance your organization's understanding of their testing roles and may advocate to your management and your technical staff that improved testing practices should be considered early in a software process improvement effort.

Effective software test engineering practices plays a vital role in improving an organization's overall software development capabilities. Early involvement of testers with developers helps build quality systems. Finally, you don't have to be a Level 3 organization before considering appropriate automation of well-known testing activities.

This technology report provides a useful forum for exchanging ideas between DoD, SEI, and industry. Because of the attention focused on the CMM, many organizations are looking to it to guide their process improvement efforts. Contact the STSC for more information about applying the principles of effective software test engineering and about improving the material in these reports.

Appendix A: Test Tool Lists

Note: The test tool lists are outdated, therefore are no longer listed in this report. Please contact Karen Rasmussen (rasmussk@software.hill.af.mil) at the STSC for a customized tool list. **Please include tool type(s), platform(s)/operating system(s) and language(s) needed.

Appendix B: STSC Product Critique System

In the December 1992 issue of *CROSSTALK*, the STSC published "Product Critiques Revisited," which discusses the STSC's Product Critique System. Sections B.1 through B.3 are a reprint of that article. Section B.3 contains a blank product critique. Completed product critiques for a number of software tools can be obtained by contacting the STSC.

B.1 Product Critique Concepts

Selecting the right software products to improve your software development process can incorporate hundreds of criteria. We use the term *technology* in the broad sense to include processes, methods, techniques, tools, tool sets, and environments. It is easy for engineers to get wrapped up in detailed evaluations of software products by listing criteria, setting up an evaluation matrix, developing a scoring scheme, determining weighting factors, and so forth. This is not wrong; detailed evaluations are essential once your options have been narrowed down to a manageable size. Yet, we often overlook the value of a less quantitative approach of collecting testimonials or critiques from experienced users to narrow those options.

When you buy a car, it is useful to talk with friends, relatives, and neighbors who own the same models you are considering. You want to leverage off both their good and bad experiences to narrow your choices and reaffirm your selection. Once your choices have been reduced, you can use evaluations, specifications, additional options, and test drives to select a car to buy.

Selecting a software technology is no different, except that most friends, relatives, and neighbors lack the experience you need. The STSC is facilitating the exchange of tool experiences with the product critique form, which you will find in Section B.3. If you are an experienced user of a particular software product, please take a few minutes to share your experience by filling in this form and mailing or faxing it to the STSC. Make copies of the blank form as needed for each product to be critiqued.

The goals of the STSC Product Critique System are

To convey actual experiences of product users to potential product buyers.

To help consumers of software technology reduce their candidate lists effectively and at low cost.

To provide software product developers with constructive criticism from users about the strengths and weaknesses of their products, particularly requirements analysis and design, testing, reengineering, documentation, reuse, project management, project estimation, and software engineering environment products.

To complement or summarize quantitative evaluations for use in the final selection of software technologies.

Note that the STSC emphasizes the importance of selecting a sound software development or maintenance methodology before selecting a software product to automate that methodology. Some products have a built-in methodology that will require some user training before the tool can be used effectively.

B.2 Product Critique Instructions

Every entry of the product critique should be brief and basically cover the issues of ease of use, power, robustness, functionality, ease of insertion, and quality of vendor support [FIRT87]. The intent is to capture your impressions of the product rather than have you list the tool's features. Most of the blocks and fields are self-explanatory, but some additional explanations on the following fields may be helpful:

a. **Operating Environment.** Describe the main components (hardware and software) that make up the system, particularly those parts of which the product is dependent for proper utilization. Include memory and disk requirements.

b. **Project Description.** Describe unclassified details about the project that would help product critique readers to better understand the technology's application in your environment.

c. **Keep name and company confidential.** Circle "Y" or "N" to indicate whether or not you would be willing to have other potential tool buyers or vendors call you for further information.

d. **Will you use this product on your next project? Do you prefer another product?** These are essentially bottom-line questions about your opinion of the technology. Enter supporting information in the blocks for strengths and weaknesses. Enter the name of the preferred product in the advice block.

e. **Notable Strengths of this Product.** Describe notable features and performance characteristics that make the technology indispensable or better than other technologies with similar features. Cite specific examples that helped you form your impressions.

f. **Notable Weaknesses of this Product.** Describe annoying problems or deficiencies that reduce the technology's effectiveness.

g. **Advice for Potential Users or Buyers of this Product.** This block provides information (not necessarily good or bad) that is important for a potential buyer; for example, you could write, "It is essential to use an accelerator card and a 20-inch color monitor."

h. **Vendor Comments.** The product reviewer does *not* complete this block. The vendor will be given the opportunity to respond to the strengths and weaknesses identified by the reviewer.

(This page has been intentionally left blank.)

Appendix C: References

[216788] Department of Defense, "Military Standard Defense System Software Development," February 1988.

[ADRI82] Adrion, W.R., et al., "Validation, Verification, and Testing of Computer Software," June 1982.

[ARNO90] Arnold, Robert S., "Tools for Static Analysis of Ada Source Code," ADA_STATIC_TOOLS_SURVEY-90015-n, Version 01.00.01, June 1990.

[BASI91] Basili, Victor R., et al., "The Future Engineering of Software: A Management Perspective," *IEEE Computer*, September 1991, p. 95.

[BEIZ90] Beizer, Boris, *Software Testing Techniques*, 2nd edition, Van Nostrand Reinhold, 1990.

[BOEH88] Boehm, B.W. and P.N. Papaccio, "Understanding and Controlling Software Costs," *IEEE Transactions on Software Engineering*, October 1988, Vol. 12, No. 9, pp. 929-940.

[BREW91] Brewin, Bob, quoting Paul A. Strassmann, "Corporate Information Management White Paper," *Federal Computer Week*, September 1991, p. 10.

[BROO75] Brooks, Fred P., Jr., *The Mythical Man-Month*, Addison-Wesley, Inc., 1975.

[CECO90] "Applying T to AFATDS A2A3 Documents," U.S. Army, CECOM, Technical Report 4, DAAB07-87-C-B025, 1990.

[CLAR91] Clark, Peter, and Bard S. Crawford, *Evaluation and Validation (E&V) Reference Manual*, TASC No. TR-5234-3, Version 3.0, Feb.14, 1991.

[CM94] Staib, Vicki, et al., *Configuration Management Technologies Report*, Software Technology Support Center, September 1994.

[CONN82] Conner, Daryl R., et al., "Building Commitment to Organizational Change," *Training and Development Journal*, Vol. 36, No. 4, April 1982, pp. 18-30.

[CORD93] Cord, David, "Defect-Causal Analysis Drives Down Error Rates," *IEEE Software*, July 1993.

[DACS79] Data and Analysis Center for Software (DACS), "The DACS Glossary: A Bibliography of Software Engineering Terms," October 1979.

[DAIC92] Daich, Gregory T., et al., "Testing Support at the SEI CMM Level 1," *CROSSTALK*, Software Technology Support Center, December 1992.

[DEMI87] DeMillo, Richard A., *Software Testing and Evaluation*, The Benjamin/Cummings Publishing Company, Inc., 1987.

[DEUT82] Deutsch, Michael S., *Software Verification and Validation*, Prentice Hall, 1982.

[DION93] Dion, Raymond, "Process Improvement and the Corporate Balance Sheet," *IEEE Software*, July 1993.

[DOC94] Sorenson, Reed, *Documentation Tools Report*, March 1994, Software Technology Support Center.

[DRUY94] Druyun, Darleen A., "New Air Force Software Metrics Policy," *CROSSTALK*, Software Technology Support Center, April 1994.

[DUNN84] Dunn, R.H., *Software Defect Removal*, McGraw-Hill.

[DURA93] Durant, Jerry, *Testing Tools Reference Guide*, Software Quality Engineering, 1993.

[EST93] Barrow, Dean, *Software Cost Estimation Technologies Report*, January 1993, Software Technology Support Center.

[FAGA86] Fagan, M.E., "Advances in Software Inspections," *IEEE Transactions on Software Engineering*, July 1986, Vol. SE-12, No. 7, 744-751.

[FIRT87] Firth, R., "A Guide to the Classification and Assessment of Software Engineering Tools," Technical Report CMU/SEI-87-TR-10, 1987.

[FORT91] Forte, Gene, *CASE Outlook*, CASE Consulting Group, Inc., 1991.

[FOWL88] Fowler, Pricilla, and Stan Przybylinski, *Transferring Software Engineering Tool Technology*, IEEE Computer Society Press, Washington, D.C., 1988.

[FREE90] Freedman, Daniel P., and Gerald M. Weinberg, *Handbook of Walkthroughs, Inspections, and Technical Reviews: Evaluating Programs, Projects, and Products*, Third Edition, Dorset House Publishing Co., 1990.

[GELP94] Gelperin, David, Course Notes from "Systematic Software Testing," Software Quality Engineering, 1994.

[GILB93] Gilb, Tom, and Dorothy Graham, *Software Inspection: An Effective Method for Software Project Management*, Addison-Wesley Inc., 1993.

[GRAH93] Graham, Dorothy R., *CAST Report*, Cambridge Market Intelligence, 1993.

[HAMM91] Hammer, Michael, "The Last World," *Computerworld Premier 100*, September 1991, p. 80.

[HEND94] Henderson, Johnnie A., "Software Quality Assurance on Onboard Shuttle Software," Proceedings of the Software Technology Conference (STC-94), April 1994.

[HETZ88] Hetzel, Bill, *A Complete Guide to Software Testing*, QED Information Sciences, Inc., 1988.

[HUMP87] Humphrey, W. S., and W. L. Sweet, *A Method For Assessing the Software Engineering Capability of Contractors*, CMU/SEI-87-TR-23, September 1987.

[HUMP90] Humphrey, W. S., *Managing the Software Process*, Addison-Wesley, 1987.

[IDA92] Youngblut, Christine and Bill Brykczynski, *An Examination of Selected Software Testing Tools: 1992*, Institute for Defense Analyses, IDA Paper P-2769, December 1992.

[IEEE83] IEEE Standard 729, "IEEE Standard Glossary of Software Engineering Terminology," February 1983.

[IEEE92] IEEE/ANSI 1209, "Recommended Practice for CASE Tool Evaluation and Selection", 1992.

[KIND92] Kindl, Mark R., Lt. Col., "Software Quality and Testing: What DoD Can Learn from Commercial Practices", *CROSSTALK*, Software Technology Support Center, December 1993.

[KOLT93] Koltun, Phil, "Testing ... Testing ... ," ShowCASE, May 1993.

[LAWL89] Lawlis, Patricia K., "A Supporting Framework for Software Evaluation and Selection," based on her Ph.D. dissertation, "Supporting Selection Decisions Based on the Technical Evaluation of Ada Environments and Their Components," December 1989.

[MCCA89] McCabe, Thomas J., and Charles W. Butler, "Design Complexity Measurement and Testing," *Communications of the ACM*, December 1989.

[MCCR90] "Mission-Critical Computer Resources Software Support," *Military Handbook 347*, Department of Defense, p. 10, May 1990.

[MET94] Houser, Judd, et al., *Metrics Starter Kit*, Software Technology Support Center, June 1994.

[MOSE92] Mosemann, Lloyd K., II, "Ada: Vital to the Industrial Base," Address at the *Ada's Success in MIS: A Formula for Progress Symposium*, George Mason University, Fairfax, VA, Jan., 14, 1992. (Complete text available in *CROSSTALK*, February 1992.)

[MOSE93] Mosemann, Lloyd K., II, "Software Technology Conference Closing Address," *CROSSTALK*, Special Edition 1993, Software Technology Support Center.

[MOSL92] Mosley, Vicky, "How to Assess Tools Efficiently and Quantitatively," IEEE Software, May 1992.

[MYER79] Myers, Glen, *The Art of Software Testing*, New York: John Wiley, 1979.

[NORM92] Norman, Dr. Stephen, *Dynamic Testing Tools: A Detailed Product Evaluation*, Ovum Ltd., 1992.

[PATR92] Patrick, Daryl, "Solving the Testing Puzzle," Proceedings of the 1992 Software Technology Conference, Software Technology Support Center, April 1992.

[PAUL93] Paulk, Mark C., Charles V. Weber, Suzanne M. Garcia, Mary Beth Chrissis, Marilyn Bush, "Key Practices of the Capability Maturity Model, Version 1.1," Software Engineering Institute, CMU/SEI-93-TR-25, February 1993.

[PETE92] Petersen, Gary, "Product Critiques Revisited," *CROSSTALK*, Software Technology Support Center, December 1992.

[PM93] Steadman, Todd, et al., *Project Management Tools Report*, Software Technology Support Center, December 1993.

[POST87] Poston, Robert, "Counting Down to Zero Software Failures," IEEE Software, Vol. 4, No. 5, 1987, pp. 54-61.

[POST90] Poston, Robert M., *Software Testing Policy*, Programming Environments, Inc., 1990.

[POST92] Poston, Robert M., et al., "Evaluating and Selecting Testing Tools," IEEE Software, May 1992.

[POST92] Poston, Robert M., "A Complete Toolkit for the Software Tester," *CROSSTALK*, October 1992.

[PRES87] Pressman, R.S., *Software Engineering, A Practitioner's Approach*, McGraw-Hill, 1987.

[PRIC94] Price, Gordon, "Software Testing Terminology," *CROSSTALK*, Software Technology Support Center, July 1994.

[RAD94] Grotzky, John, et al., *Requirements Analysis and Design Tools Report*, Software Technology Support Center, March 1994.

[RADC83] Rome Air Development Center, "Software Interoperability and Reusability," Vols. I and II, Technical Report RADC-TR-83-174, July 1983.

[RE93]Olsem, Mike, et al., *Reengineering Tools Report*, Software Technology Support Center, August 1993.

[REUS94] Smith, Larry, *Reuse Technologies Report*, Software Technology Support Center, 1994.

[ROYE93] Royer, Thomas C., *Software Testing Management: Life on the Critical Path*, Prentice-Hall, Inc., 1993.

[SCSA93] Ragland, Bryce, et al., S*ource Code Static Analysis Tools Report*, Software Technology Support Center, 1993.

[SEE94] Hanrahan, Robert P., et al., *Software Engineering Environments Report*, Software Technology Support Center, 1994.

[SENT94] Sentry Publishing Co., "*Application Development Tools Product Guide*," 1994.

[SIEG91] Siegel, Shel, "*The CASE for CAST: Computer-Aided Software Testing*," 1991.

[SMG93] Rubey, Ray, *Software Management Guide*, Software Technology Support Center, October 1993.

[SOFT90] Software Quality Engineering, "*Software Measures and Practices Benchmark Study*," 1990.

[TPEE93] Daich, Gregory T., et al., *Test Preparation, Execution, and Evaluation Software Technologies Report*, Software Technology Support Center, February 1993.

[YOUN89] Youngblut, Christine, *SDS Software Testing and Evaluation: A Review of the State of the Art in Software Testing and Evaluation with Recommended R&D Tasks*, Institute for Defense Analysis, IDA Paper P-2132, February 1989.

[ZVEG94] Zvegintzov, Nicholas, *Software Maintenance Technology Reference Guide*, Software Maintenance News, Inc., 1994.

Appendix D: Testing Conferences and Seminars

Note: Appendix is no longer available.

Appendix E: Glossary

Acceptance Testing Formal testing conducted to determine whether or not a system satisfies its acceptance criteria - enables a customer to determine whether or not to accept the system [YOUN89].

Algorithmic Test Case Generation A computational method for identifying test cases from data, logical relationships, or other software requirements information.

Alpha Testing Testing of a software product or system conducted at the developer's site by the customer [YOUN89].

APSE Ada Programming Support Environment.

Automated Testing That part of software testing that is assisted with software tool(s) that does not require operator input, analysis, or evaluation.

Beta Testing Testing conducted at one or more customer sites by the end-user of a delivered software product or system [YOUN89].

Black-Box Testing Functional testing based on requirements with no knowledge of the internal program structure or data. Also known as closed-box testing.

Bottom-Up Testing An integration testing technique that tests the low-level components first using test drivers for those components that have not yet been developed to call the low-level components for test.

Boundary Value Analysis A test data selection technique in which values are chosen to lie along data extremes. Boundary values include maximum, minimum, just inside/outside boundaries, typical values, and error values [MYER79].

Branch Coverage Testing A test method satisfying coverage criteria that requires each decision point at each possible branch to be executed at least once [IEEE83].

CASE Computer-Aided (or Computer-Assisted) Software Engineering consists of the automated capability to implement the discipline of software engineering within a lifecycle phase or across multiple lifecycle phases.

CAST Computer-Aided Software Testing consists of the automated capability to implement software testing functions within a lifecycle phase or across multiple lifecycle phases.

Cause-Effect Graphing A testing technique that aids in selecting, in a systematic way, a high-yield set of test cases that logically relates causes to effects to produce test cases. It has a beneficial side effect in pointing out incompleteness and ambiguities in specifications [MYER79].

Clear-Box Testing Another term for white-box testing. Structural testing is sometimes referred to as clear-box testing since "white-boxes" are considered opaque and do not really permit visibility into the code. Also known as glass-box or open-box testing.

Cyclomatic Complexity A measure of the number of linearly independent paths through a program module [MCCA89].

Data Flow Analysis Consists of the graphical analysis of collections of (sequential) data definitions and reference patterns to determine constraints that can be placed on data values at various points of executing the source program [YOUN89].

Debugging The act of attempting to determine the cause of the symptoms of malfunctions detected by testing or by frenzied user complaints [BEIZ90].

Defect (1) A manifestation of an error made by a software developer [IDA92]. (2) An error made by a software developer that can appear in code or documentation.

Defect Analysis Using defects as data for continuous quality improvement. Defect analysis generally seeks to classify defects into categories and identify possible causes in order to direct process improvement efforts.

Defect Density Ratio of the number of defects to program length (a relative number).

Desk Checking A form of manual static analysis usually performed by the originator. Source code documentation, etc., is visually checked against requirements and standards [PRIC94].

DT&E Developmental Test and Evaluation focuses on the technological and engineering aspects of the system or equipment items [DACS79].

Dynamic Analysis The process of evaluating a program based on execution of that program [IEEE83]. Dynamic analysis approaches rely on executing a piece of software with selected test data.

Equivalency Class Partitioning A testing technique that involves identifying a finite set of representative input values that invoke as many different input conditions as possible [MYER79].

Error	(1) A discrepancy between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition [IEEE83], and (2) A mental mistake made by a programmer that may result in a program fault [YOUN89].
Error-Based Testing	Testing where information about programming style, error-prone language constructs, and other programming knowledge is applied to select test data capable of detecting faults, either a specified class of faults or all possible faults [YOUN89].
Essential Complexity	A measure of the level of "structuredness" of a program [MCCA89].
Evaluation	The process of examining a system or system component to determine the extent to which specified properties are present [YOUN89].
Execution	The process of carrying out an instruction or the instructions of a computer program by a computer [IEEE83].
Exhaustive Testing	Executing the program with all possible combinations of values for program variables [ADRI82].
Failure	The inability of a system or system component to perform a required function within specified limits. A failure may be produced when a fault is encountered [IEEE83].
Failure-Directed Testing	Testing based on the knowledge of the types of errors made in the past that are likely for the system under test.
Fault	A manifestation of an error in software. A fault, if encountered, may cause a failure [IEEE83].
Fault-Based Testing	Testing that employs a test data selection strategy designed to generate test data capable of demonstrating the absence of a set of pre-specified faults; typically, frequently occurring faults [YOUN89].
Fault Tree Analysis	A form of safety analysis that assesses hardware safety to provide failure statistics and sensitivity analyses that indicate the possible effect of critical failures [YOUN89].
Formal Review	Formal reviews are technical reviews conducted with the customer including the types of reviews called for in DOD-STD-2167A (Preliminary Design Review, Critical Design Review, etc.)

Functional Testing Application of test data derived from the specified functional requirements without regard to the final program structure [ADRI82]. Also known as black-box testing.

Function Points A consistent measure of software size based on user requirements. Data components include: inputs, outputs, etc. Environment characteristics include: data communications, performance, reusability, operational ease, etc. Weight scale: 0 - not present, 1 - minor influence, 5 - strong influence.

Heuristics Testing Another term for failure-directed testing.

Hybrid Testing A combination of top-down testing combined with bottom-up testing of prioritized or available components.

Incremental Analysis Incremental analysis occurs when (partial) analysis may be performed on an incomplete product to allow early feedback on the development of that product [YOUN89].

Infeasible Path Program statements sequence that can never be executed [ADRI82].

Inspection A formal evaluation technique in which software requirements, design, or code are examined in detail by a person or group other than the author to detect faults, violations of development standards, and other problems [IEEE83]. A quality improvement process for written material that consists of two dominant components: product (document) improvement and process improvement (document production and inspection) [GILB93].

Instrument To install or insert devices or instructions into hardware or software to monitor the operation of a system or component [PRIC94].

Integration The process of combining software components or hardware components or both into an overall system.

Integration Testing An orderly progression of testing in which software components or hardware components or both are combined and tested until the entire system has been integrated.

Interface A shared boundary. An interface might be a hardware component to link two devices, or it might be a portion of storage or registers accessed by two or more computer programs [IEEE83].

Interface Analysis Checks the interfaces between program elements for consistency and adherence to predefined rules or axioms [YOUN89].

Intrusive Testing Testing that collects timing and processing information during program execution that may change the behavior of the software from its behavior in a real environment. Usually involves additional code embedded in the software being tested or additional processes running concurrently with software being tested on the same platform.

IOT&E Initial Operational Test and Evaluation is the first phase of operational test and evaluation conducted on pre-protectional items, prototypes, or pilot production items and normally completed prior to the first major production decision. Conducted to provide a valid estimate of expected system operational effectiveness and suitability [YOUN89].

IV&V Independent Verification and Validation is the verification and validation of a software product by an organization that is both technically and managerially separate from the organization responsible for developing the product [IEEE83].

Lifecycle The period that starts when a software product is conceived and ends when the product is no longer available for use. The software lifecycle typically includes a requirements phase, design phase, implementation (code) phase, test phase, installation and checkout phase, operation and maintenance phase, and a retirement phase [IEEE83].

Maintenance The modification of a software product, after delivery, to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment.

Manual Testing That part of software testing that requires operator input, analysis, or evaluation.

Measure To ascertain or appraise by comparing to a standard; to apply a metric [IEEE83].

Measurement (1) The act or process of measuring. (2) A figure, extent, or amount obtained by measuring [IEEE83].

Metric A measure of the extent or degree to which a product possesses and exhibits a certain quality, property, or attribute [IEEE83].

Mutation Testing A method to determine test set thoroughness by measuring the extent to which a test set can discriminate the program from slight variants of the program [ADRI82].

Nonintrusive Testing Testing that is transparent to the software under test, i.e., testing that does not change the timing or processing characteristics of the

software under test from its behavior in a real environment. Usually involves additional hardware that collects timing or processing information and processes that information on another platform.

Operational Requirements Operational requirements are qualitative and quantitative parameters that specify the desired operational capabilities of a system and serve as a basis for determining the operational effectiveness and suitability of a system prior to deployment [YOUN89].

Operational Testing Testing performed by the end-user on software in its normal operating environment (DoD usage) [IEEE83].

OT&E Operational Test and Evaluation is formal testing conducted prior to deployment to evaluate the operational effectiveness and suitability of the system with respect to its mission [YOUN89].

Outside-In Testing A strategy for integration testing where units handling program inputs and outputs are tested first, and units that process the inputs to produce output being incrementally included as the system is integrated [YOUN89]. A form of hybrid testing.

Path Analysis Program analysis performed to identify all possible paths through a program, to detect incomplete paths, or to discover portions of the program that are not on any path [IEEE83].

Path Coverage Testing A test method satisfying coverage criteria that each logical path through the program be tested. Paths through the program often are grouped into a finite set of classes; one path from each class is tested [ADRI82].

Peer Reviews Peer reviews involve a methodical examination of software work products by the producer's peers to identify defects and areas where changes are needed [PAUL93].

Proof Checker A program that checks formal proofs of program properties for logical correctness [YOUN89].

Prototyping Prototyping evaluates requirements or designs at the conceptualization phase, the requirements analysis phase, or the design phase by quickly building scaled-down components of the intended system to obtain rapid feedback of analysis and design decisions.

Qualification Testing Formal testing, usually conducted by the developer for the customer, to demonstrate that the software meets its specified requirements [YOUN89].

Quality	The degree to which a program possesses a desired combination of attributes that enable it to perform its specified end use [YOUN89].
Random Testing	An essentially black-box testing approach in which a program is tested by randomly choosing a subset of all possible input values. The distribution may be arbitrary or may attempt to accurately reflect the distribution of inputs in the application environment [YOUN89].
Regression Testing	Selective retesting to detect faults introduced during modification of a system or system component, to verify that modifications have not caused unintended adverse effects, or to verify that a modified system or system component still meets its specified requirements [IEEE83].
Reliability	The probability of failure-free operation for a specified period.
Review	A review is a way to use the diversity and power of a group of people to point out needed improvements in a product or confirm those parts of a product in which improvement is either not desired or not needed [FREE90]. A review is a general work product evaluation technique that includes desk checking, walkthroughs, technical reviews, peer reviews, formal reviews, and inspections.
Semantics	(1) The relationship of characters or group of characters to their meanings, independent of the manner of their interpretation and use. (2) The relationships between symbols and their meanings [IEEE83].
Software Engineering Environment	A Software Engineering Environment (SEE) is a harmonious collection of tools that provides automated support for a software engineering system that includes all of the processes, methods, tools, information, and people an organization uses to do software engineering [SEE94].
Software Tool	A computer program used to help develop, test, analyze, or maintain another computer program or its documentation; for example, automated design tools, compilers, test tools, and maintenance tools [IEEE83].
Statement Coverage Testing	A test method satisfying coverage criteria that requires each statement be executed at least once.
Static Analysis	The process of evaluating a program without executing the program [IEEE83].

Structural Coverage Structural coverage requires that each pair of module invocations be executed at least once.

Structural Testing A testing method where the test data are derived solely from the program structure [ADRI82].

Stub A software component that usually minimally simulates the actions of called components that have not yet been integrated during top-down testing.

Syntax Syntax is (1) the relationship among characters or groups of characters independent of their meanings or the manner of their interpretation and use, (2) the structure of expressions in a language, and (3) the rules governing the structure of the language [IEEE83].

System A collection of people, machines, and methods organized to accomplish a set of specified functions [IEEE83].

System Simulation Another term for prototyping.

System Testing The process of testing an integrated hardware and software system to verify that the system meets its specified requirements [IEEE83].

Technical Review A technical review is a review that refers to content of the technical material being reviewed [FREE90].

Test Bed (1) An environment that contains the integral hardware, instrumentation, simulators, software tools, and other support elements needed to conduct a test of a logically or physically separate component. (2) A suite of test programs used in conducting the test of a component or system [PRIC94].

Test Case The definition of "test case" differs from company to company, engineer to engineer, and even project to project. A test case usually includes an identified set of information about observable states, conditions, events, and data including inputs and expected outputs.

Test Development The development of anything required to conduct testing. This may include test requirements (objectives), strategies, processes, plans, software, procedures, cases, documentation, etc. [PRIC94].

Test Driver (1) A software component that as a rule minimally simulates the actions of high-level components that have not yet been integrated during bottom-up testing. (2) Another term for test harness.

Test Executive Another term for test harness.

Test Harness A software tool that enables the testing of software components that links test capabilities to perform specific tests, accept program inputs, simulate missing components, compare actual outputs with expected outputs to determine correctness, and report discrepancies [CLAR91].

Test Plan A formal or informal plan to be followed to assure the controlled testing of the product under test [PRIC94].

Test Procedure The formal or informal procedure that will be followed to execute a test. This is usually a written document that allows others to execute the test with a minimum of training [PRIC94].

Testing Any activity aimed at evaluating an attribute or capability of a program or system to determine that it meets its required results [HETZ88]. The process of exercising or evaluating a system or system component by manual or automated means to verify that it satisfies specified requirements or to identify differences between expected and actual results [IEEE83].

Top-Down Testing An integration testing technique that tests the high-level components first using stubs for lower-level called components that have not yet been integrated and that simulate the required actions of those components.

Unit Testing Unit testing is the testing that we do to show that a unit (the smallest piece of software that can be independently compiled or assembled, loaded, and tested) does not satisfy its functional specification or its implemented structure does not match the intended design structure [BEIZ90].

Validation The process of evaluating software to determine compliance with specified requirements [216788].

Verification The process of evaluating the products of a given software development activity to determine correctness and consistency with respect to the products and standards provided as input to that activity [216788].

Walkthrough In the most usual form of the term, a walkthrough is a step-by-step simulation of the execution of a procedure, as when walking through code, line by line, with an imagined set of inputs. The term has been extended to the review of material that is not procedural, such as data descriptions, reference manuals, specifications, etc. [FREE90].

White-Box Testing Testing approaches that examine the program structure and derive test data from the program logic [YOUN89].