REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
22-07-2019	Final	

4. TITLE AND SUBTITLE
Test Operations Procedure (TOP)
01-1-063 Enterprise Software Reliability

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHORS

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
U.S. Army Electronic Proving Ground
Enterprise Systems and Software Test Division
2000 Arizona Street
Fort Huachuca, AZ 85613-7063

8. PERFORMING ORGANIZATION REPORT NUMBER
TOP 01-1-063

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
T&E Policy and Standardization Division (CSTE-TM)
U.S. Army Test and Evaluation Command
6617 Aberdeen Boulevard
Aberdeen Proving Ground, MD 21005-5001

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
Same as item 8

12. DISTRIBUTION/AVAILABILITY STATEMENT

Distribution Statement A. Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
Defense Technical Information Center (DTIC), AD No.:

14. ABSTRACT
This document describes a systematic data collection approach for determining enterprise software reliability. The overall objective is to streamline and consolidate all the approaches to increase enterprise software reliability as well as to communicate best practices for performing reliability assessments of enterprise systems.

15. SUBJECT TERMS
software, reliability, system of systems, C4I, feature, load, regression, development

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**B. ABSTRACT**	**C. THIS PAGE**			**19b. TELEPHONE NUMBER** *(include area code)*
Unclassified	Unclassified	Unclassified	SAR	29	

(This page is intentionally blank.)

U.S. ARMY TEST AND EVALUATION COMMAND
TEST OPERATIONS PROCEDURE

*Test Operations Procedure 01-1-063 22 July 2019
DTIC AD No.

ENTERPRISE SOFTWARE RELIABILITY

1. SCOPE.

Software Reliability is the probability that software will not cause the failure of a system for a specified time under specified condition[1**]. This document describes the procedures necessary for increasing enterprise software reliability through testing. The topics in this document includes guidance for conducting various software tests and systematic data collection for increasing enterprise software reliability. These enterprise systems are currently under acquisition and therefore testing is focused on establishing reliability for the system during the full development lifecycle. The identified tests in the Test Operations Procedure (TOP) allows the testing organization to select appropriate events during the acquisition lifecycle to obtain over time increased reliability of the system by eliminating defects of the System Under Test.

1.1 Purpose.

a. The purpose of this TOP is to outline a process for determining Enterprise Software Reliability using government and industry best practices, as well as convey a data collection approach. To better understand the need for increasing software reliability, one can compare the unique characteristics of software reliability with that of hardware reliability.

b. Figure 1 shows the failure characteristics of hardware. Failure rate during burn-in (hardware development process) goes down as development of the hardware matures. The hardware is released to the customer when the failure rate reaches a certain tolerance point. During the useful-life of the hardware, there are no changes in the failure rate, which is due to being no changes to the nature of the hardware. However, at some point, all hardware becomes worn. This happens during the end-of-life phase. The failure rate goes up exponentially as the physical decay takes effect. This is commonly known as the “bathtub curve” for hardware reliability.

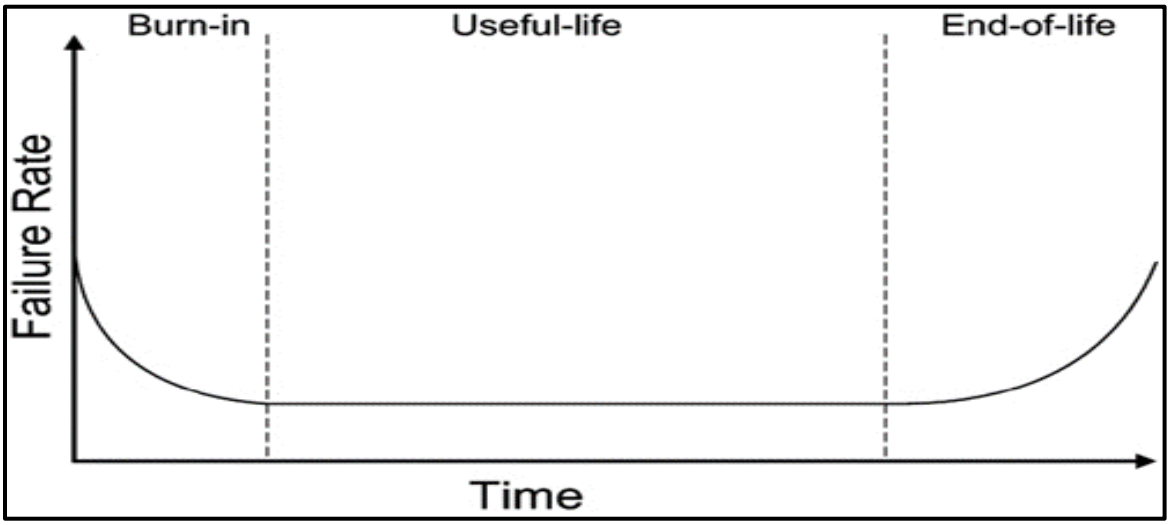

Figure 1. Failure rate of hardware vs. time[2].

* Superscript numbers correspond to Appendix B, References.

c. In comparison, the nature of software reliability is different as it relates to failure rate. Figure 2 shows the failure characteristics of software. During the Software Development Life-Cycle (SDLC) the failure rate goes down as the software matures, similar to hardware development. In the Obsolescence phase, software does not have the increasing failure rate as hardware does. Also during Obsolescence, there is no motivation for any upgrades or changes to the software, therefore, the failure rate does not change.

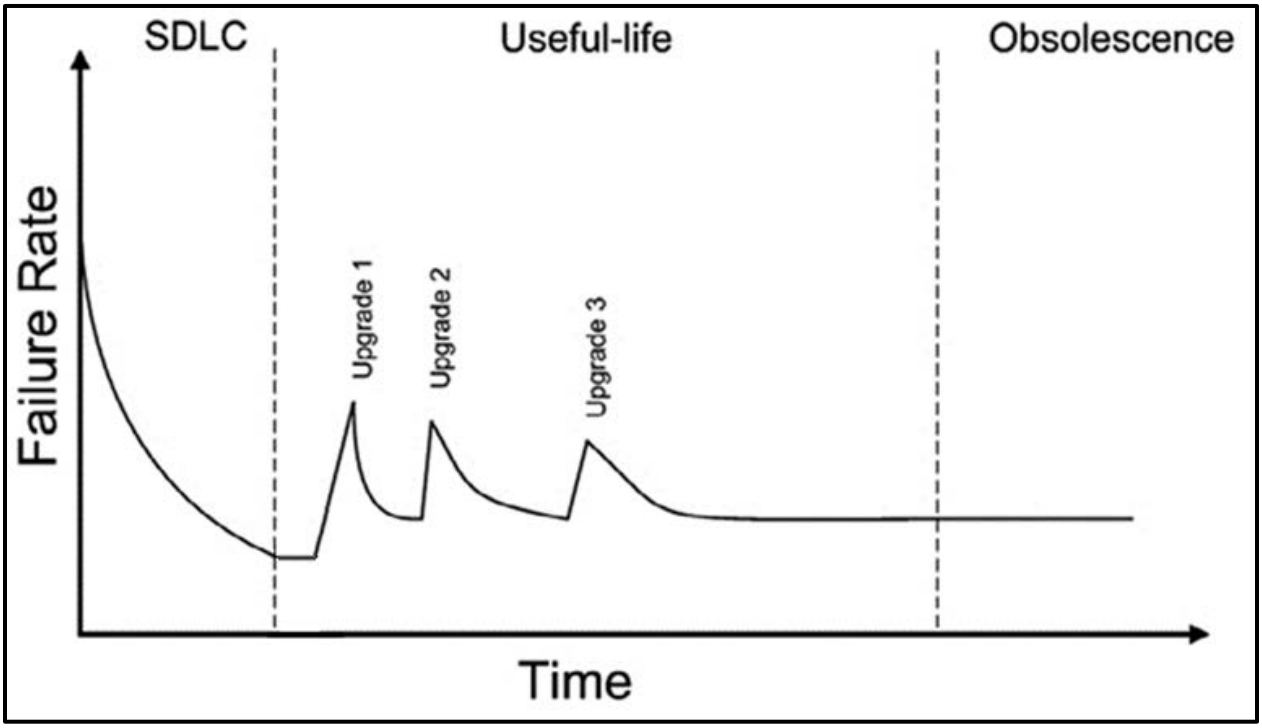

Figure 2. Failure rate of software vs. time[1].

d. Another difference to hardware failure rates, can be seen in the Useful-life phase. Software will experience a drastic increase in failure rate each time an upgrade is made of the software. The upgrades in Figure 2 imply capability enhancements and not reliability improvements. For capability upgrades, the complexity of the software is likely to be increased, since the functionality of the software is enhanced. The failure rate levels off gradually after each upgrade, because of the detection of defects that are then corrected. Finally, defect corrections may contribute to more software failures as they can introduce additional defects into the software.

1.2 Application.

This TOP is appropriate for all business and medical enterprise software systems either acquired or developed.

1.3 Limitations.

This TOP does not endorse any specific development methodology and is applicable to all enterprise medical and business software products.

2. FACILITIES AND INSTRUMENTATION.

2.1 Facilities.

a. Test facilities will vary depending on the development stage of the System Under Test (SUT); however, as a general rule, the support of a computerized environment that will host SUT and data collection instrumentation is required. Experience has shown the value in performing the data collection on site with subject matter experts (SMEs) and novice users present. This helps the evaluation team to identify and eliminate unintentional user inputs in collected data sets that can impact overall reliability results. Data collection for larger enterprise systems should be centralized to allow for efficient and timely harvesting.

b. All facilities listed in Table 1 are appropriate for collecting reliability data.

TABLE 1. TEST FACILITIES AND OBJECTIVES

FACILITY	OBJECTIVE
Material Developer's software Test and Evaluation (T&E) facility	To support the development, testing, and analysis of software code.
Material Developer's system laboratory facility	To support integration testing in an environment designed to replicate the true operational environment to the extent possible.
Operational facility	To conduct complete-system testing in a live environment.

2.2 Instrumentation.

Instrumentation can vary depending on the type of test being conducted, as well as when the test is executed in the Software Development Life-Cycle. The instrumentation required on an early Unit Test will differ from instrumentation required on a later System Acceptance Test, even though it may be testing the same software which may be part of the same SUT.

3. TYPES OF TESTING FOR RELIABILITY.

Software testing for determining reliability can be categorized into three types: (1) Feature Testing, (2) Load Testing, and (3) Regression Testing[3].

3.1 Feature Testing.

a. Feature testing checks the features provided by the software and is conducted in the following steps:

(1) Each operation in the software is executed at least once.

(2) Interaction between two operations is reduced.

(3) Each operation is checked for its proper execution.

b. A software feature can be defined as the change made in the system to add new functionality or modify existing functionality. A good feature is said to have characteristics that is designed to be useful, intuitive, and effective.

3.2 Load Testing.

a. Load testing is executed to check the performance of the software under maximum work load. Any software performs better up to some amount of workload, after which the response time of the software starts degrading. For example, a web site can be tested to see how many simultaneous users it can support without performance degradation. Load testing can also collect data on performance, which checks how the software performs under a specific workload.

b. Load testing is most relevant for multi-user systems, and is often performed on software that runs on a client/server system such as a web application[4]. However, other types of software systems can also be load tested. For example, a word processor or graphics editor can be forced to read an extremely large document, or a financial package can be forced to generate a report based of several years' worth of data. The most accurate load testing simulates actual use, as opposed to testing using theoretical or analytical modeling. This provides relevant information about the software prior to operational deployment.

c. Load testing is often performed with specialized software. The software would subject the SUT to different numbers of virtual and live users while monitoring performance measurements under these different loads. Load and performance testing is usually conducted in a test environment identical to the production environment before the software system is permitted to go live. This is done to protect the actual working system from any potential damage caused by the load test. However, we are seeing a trend in deploying software in a limited operational environment such as a Test Bed or smaller Operational Environment such as a Pilot Site.

3.3 Regression Testing.

a. As software is fixed or changed, emergence of new faults and re-emergence of old faults are common. Sometimes re-emergence occurs because a fix gets lost through poor revision control practices. Often a fix for a problem will only correct the problem in the narrow

case where it was first observed, but not in the more general cases which may arise over the lifetime of the software.

b. Regression testing is a type of software verification which confirms that software, which was previously developed and tested, still performs correctly after it was changed[5]. Changes may include software enhancements, patches or configuration changes. During regression testing, new software bugs may be uncovered.

c. The purpose of regression testing is to ensure that changes such as those mentioned previously have not introduced new faults. One of the main reasons for regression testing is to determine whether a change in one part of the software affects other parts in the program.

4. TEST PROCEDURES.

4.1 Feature Testing.

a. The software development life cycle has implications for Feature Testing in that it generally determines the level of testing[6]. The levels of testing echo the levels of abstraction found in the waterfall model of the software development life cycle, which is useful for identifying distinct levels of testing and for clarifying the objectives that pertain to each level. An example of a common variation of the waterfall model is shown in Figure 3.

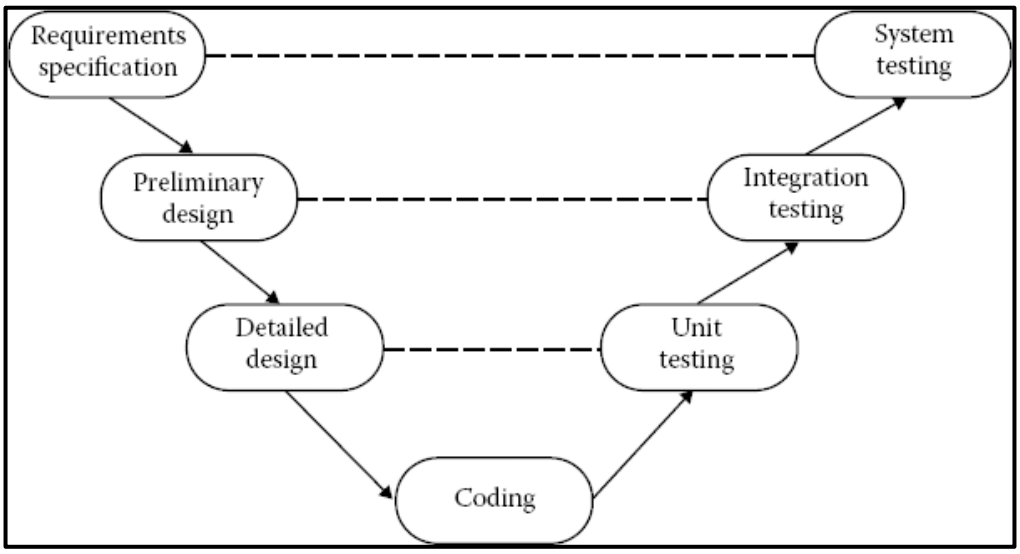

Figure 3. Software development lifecycle waterfall model[6].

b. The waterfall model for software development has implications on software testing, but the model has drawbacks. It is considered by many to be too rigid for the complexities associated in real-life software development. A widely accepted alternative with a less rigid structure is the Agile software development method. The Agile approach favors:

(1) Customer driven development.

(2) Bottom-up development.

(3) Flexibility with respect to changing requirements.

(4) Early delivery of fully functional components[6].

c. Most Agile development methods break product development work into small increments that minimize the amount of up-front planning and design. Iterations, or sprints, are short time frames that typically last from one to four weeks. Each iteration involves a cross-functional team working in all functions: planning, analysis, design, coding, unit testing, and system acceptance testing. At the end of the iteration a working product is demonstrated to stakeholders. This minimizes overall risk and allows the product to adapt to changes quickly. An iteration might not add enough functionality to warrant a release, but the goal is to have an available release at the end of each iteration. Multiple iterations might be required to release a product or new features[7]. Figure 4 shows a diagram with the processes associated with a typical agile project.

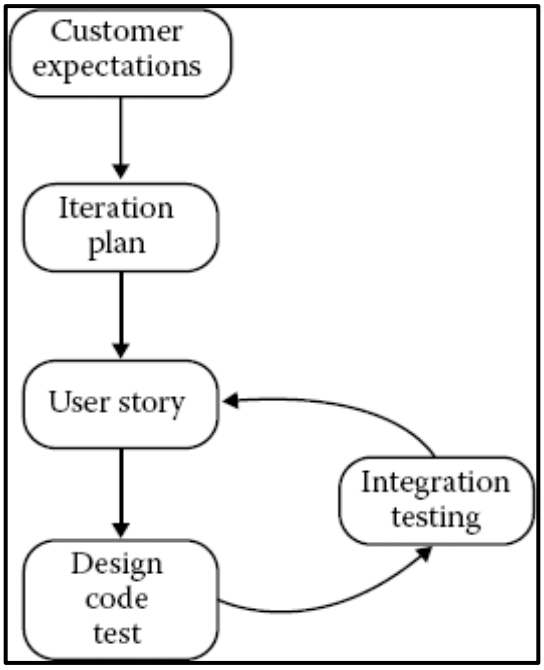

Figure 4. Generic Agile lifecycle[6].

d. Due to the Defense Acquisition Regulations System (DARS), a Department of Defense (DOD) sponsored software project will still most likely incorporate the rigid structure outlined in the traditional waterfall method. Some smaller development teams within a large acquisition project may incorporate Agile principles to ease complexity and/or resource

constraints. Agile testing focuses on repairing faults immediately, rather than waiting for the end of the project. When testing occurs at the tail end of a project as with most traditional methods, it can sometimes be sacrificed in terms of duration and quality to meet critical schedules and budget restrictions. Costs are expected to go down as the time between development and testing feedback decreases. With shorter feedback loops, bugs fixes and reworks require less time as developers spend much less time reengaging the code's context as they move on to new problems and projects[8]. Agile testing teams often rely on software testing tools to solve challenges that can ultimately speed-up the release. Most teams look for collaboration features, automated or customized reporting and finding ways to avoid repeated efforts. Choosing the right tool will depend on the requirements of each team. As software development and testing evolves in the DOD, look for more Agile approaches mixed in with traditional software development and testing methods.

4.1.1 Unit Testing.

Unit Testing typically happens during and after coding and is usually the start of any formal software testing. It is categorized as a Developmental Test because it happens before a final deliverable is released. Most unit testing is performed by the vendor due to unit testing tools being integrated into development tools.

a. The term “unit” needs explanation because there are several interpretations about exactly what constitutes a unit in software testing. In a procedural programming language, a unit can be:

(1) A single procedure.

(2) A function.

(3) A body of code that implements a single function.

(4) Source code that fits on one page.

(5) A body of code that represents work done in a short time frame.

b. In an object oriented programming language, there is general agreement that a class is a unit, however, methods of a class may be limited by any of the definitions of a unit for procedural code. The bottom line is that “unit” is probably best defined by the organization implementing the code.

c. Unit testing is commonly automated, but may be still performed manually[9]. The choice of either an automated or manual unit test will differ depending on project specifics. The objective in unit testing is to isolate a unit and validate its correctness. A manual approach to unit testing may employ a step-by-step instructional document. However, automation is efficient for achieving this, and enables many benefits. By using an automation framework, the developer codes criteria that is known to be good into the test to verify the unit’s correctness. During test case execution, the framework logs tests that fall within a certain criteria. Many automation

frameworks will also automatically flag these failed test cases and report them in a summary, and depending upon the severity of a failure, the framework may halt subsequent testing. Figure 5 shows a screenshot of a unit test tool with test results on the left hand side.

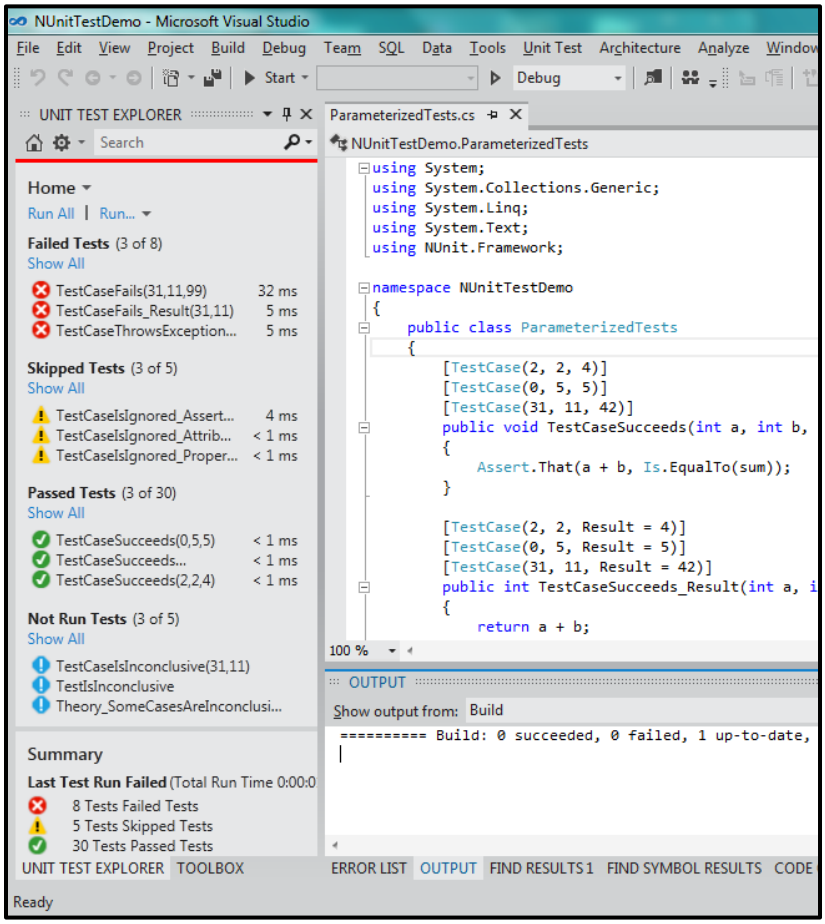

Figure 5. Unit testing tool.

4.1.2 Integration Testing.

For a large DOD Program, the post-tested Units are integrated into a larger Unit or System. Integration Testing can either be a standalone test, or if the test occurs later in the lifecycle, part of a larger System Test.

a. Integration testing is the phase in which individual software modules are combined and tested as a group[10]. This occurs after unit testing and before system testing. Integration testing takes as its input modules that have been tested, groups them in larger aggregates, applies tests defined in an integration test plan to those aggregates, and delivers as its output the integrated output, ready for system testing. Some different types of integration testing are big-bang, bottom-up, top-down, and mixed (hybrid).

(1) In the big-bang integration testing approach, most of the developed modules are coupled together to form a complete software system or major part of the system and then used for integration testing. This method is very effective for saving time in the integration testing process. However, if the test cases and their results are not recorded properly, the entire integration process will be more complicated and may prevent the testing team from achieving the goal of integration testing.

(2) Bottom-up integration testing is an approach to integrated testing where the lowest level components are tested first, then used to facilitate the testing of higher level components. The process is repeated until the component at the top of the hierarchy is tested. All the bottom or low-level modules, procedures or functions are integrated and then tested. After the integration testing of lower level integrated modules, the next level of modules are integrated and subsequently tested. This approach is helpful only when all or most of the modules of the same development level are ready. This method also helps to categorize the levels of software developed and makes it easier to report testing progress in the form of a percentage.

(3) Top-down integration testing is an approach to integrated testing where the top integrated modules are tested and the lower level modules are simulated. Stubs act as temporary replacements to the lower modules, but give the same output as the actual product.

(4) Mixed or hybrid integration testing is an approach that combines top-down testing with bottom-up testing. The process for this varies greatly depending on the SUT.

b. Integration testing may be performed manually or take advantage of automated tools, depending on the goals of the project and the software to be tested. For example, a networked enterprise application may have all the software components integrated in some type of networked test bed. An example of an automated tool may be a packet inspector to find out how well each integrated component communicates with each other on the network. Figure 6 shows an example of a common tool used for inspecting packets over a network.

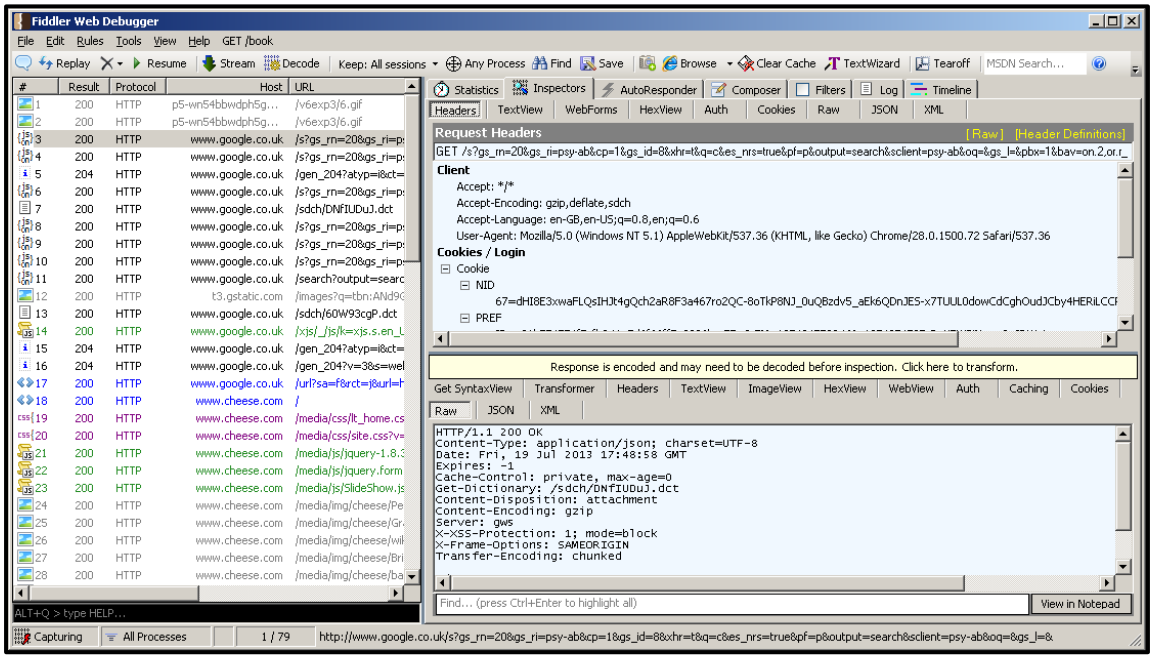

Figure 6. Tool used for inspecting network packets.

4.1.3 System Testing.

System Testing typically happens at the end of the Lifecycle and involves the entire System instead of just Components or Units. It is typically performed by government and military personnel who would be the actual users of the Software or System. This makes each scenario more true to life to mimic the actual usage of the real life customer. Vendor support is kept to a minimum to help keep the test scenarios as true to life as possible.

a. Of the three levels of testing, the system level is the closest to everyday experience[6]. The goal is to not only find faults, but also to demonstrate correct software behavior. Because of this, system testing is commonly approached from a specification-based standpoint instead of a development based one. System testing is closely coupled with requirements specified in previous iterations of the SDLC. A typical System Test will verify each of the requirements outlined in the Requirements Specification phase (Figure 3). Typically a system test is performed in the context of a Functional Requirement Specification (FRS) and/or System Requirement Specification (SRS). It tests not on the design, but the behavior and even the believed expectations of the customer. It also tests up to and, in many cases, beyond the boundaries defined in the software requirements specifications.

b. System Testing falls within the scope of black-box testing in which the tester requires no knowledge of the inner workings of the software code or logic to successfully perform the test[11]. System testing is performed on the entire system and can be traced back to the requirements defined in the first iteration of the SDLC (Section 4.1). Each software project has different requirements for System Testing which vary greatly depending the SUT. Because of

this, the TOP will describe some frequently executed types of software System Tests, which may or may not be required, depending on the Program:

(1) Requirements Testing: Perhaps the most common type of System Test, Requirements Testing simply executes each set of defined requirements at least once. When actual results differ from expected requirements, the flaws are identified as defects. Finding these defects and implementing the resulting repairs contributes to improved reliability. Many tools can help with keeping track of requirements and results such as Matrices, Configuration Management Tools, and Test Tracking Tools.

(2) Graphical User Interface (GUI) Testing: Test cases to exercise the GUI itself. A GUI may have additional operations that needs to be tested as opposed to testing a regular Command Line Interface (CLI). GUI Testing exposes defects related to user's graphics-based controls. A relatively small GUI may have many more operations than a comparable CLI. The number of operations can easily be an order of magnitude larger. In some cases, an automatic screen capture tool may help the tester with the amount of data which needs to be addressed during the test scenario. Figure 7 displays a screen capture tool capable of capturing real time video of steps executed.

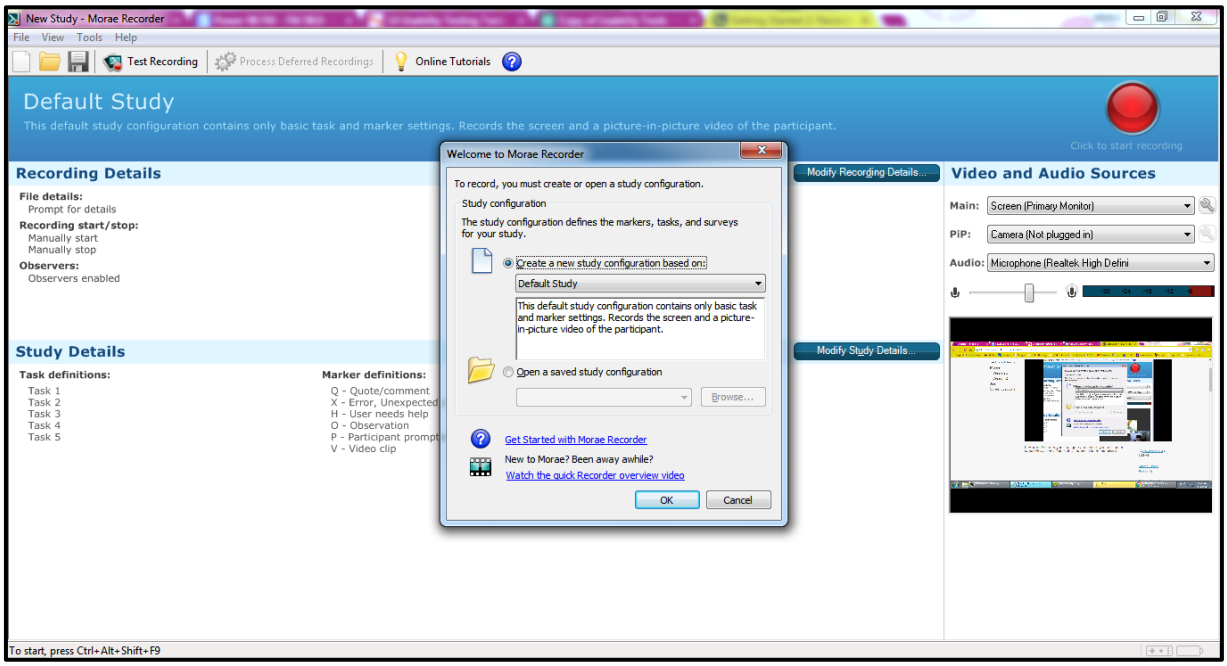

Figure 7. Screen capture tool.

(3) Scalability Testing: Testing of a software application to measure its capability to scale up or scale out in terms of its non-functional capability. The defects found during Scalability Testing may hinder future growth of the system to include additional users, capabilities etc. Depending on the application, different parameters can be tested. For example,

if a web page is being tested, a good scenario may be to determine the user limit for the web application and ensure that end user experience under a high load is not compromised. Also, other test scenarios may determine central processing unit (CPU) and network usage to find out what needs to be done to scale up the application. The test may determine if replacing a processor with a faster one is adequate for scaling up. For scaling out, a test may determine if setting up another server to run in parallel would be adequate to address new load.

(4) Compatibility Testing: Testing used to ensure compatibility of the software application with other objects such as other web browsers, hardware platforms, specialized users (i.e., foreign language users), other operating systems, etc. This type of testing helps find out how well an application performs in a particular environment that includes theses other objects. Defects found during this type of test may hinder deployment because the software may not interoperate with a previous software version.

(5) Security Testing: Intended to reveal flaws in the security mechanisms of the software. It is important to note that due to the limitations of testing, passing a security test is not an indication that no flaws exist or that the software adequately satisfies security requirements. Typical security requirements may include specific elements of confidentiality, integrity, authentication, availability, authorization and non-repudiation. Security testing has a number of different meanings and can be completed in a number of different ways. Some typical types of security tests for software include vulnerability assessments, penetration tests and discovery scans. The defects found during Security Testing may allow a hacker to take advantage of an existing vulnerability.

(6) Software Performance Testing: The testing performed to determine responsiveness and stability under a particular workload. It can also serve to investigate, measure, validate quality attributes of the system or software. In many cases, performance testing requires gathering certain metrics to measure up against baseline or expected figures. Performance Testing can also discover defects which may not be notice during other types of testing due to the increased stress put on the SUT. Figure 8 displays a performance measuring tool which is part of a popular web browser. The tool can be used to measure performance on web based software.

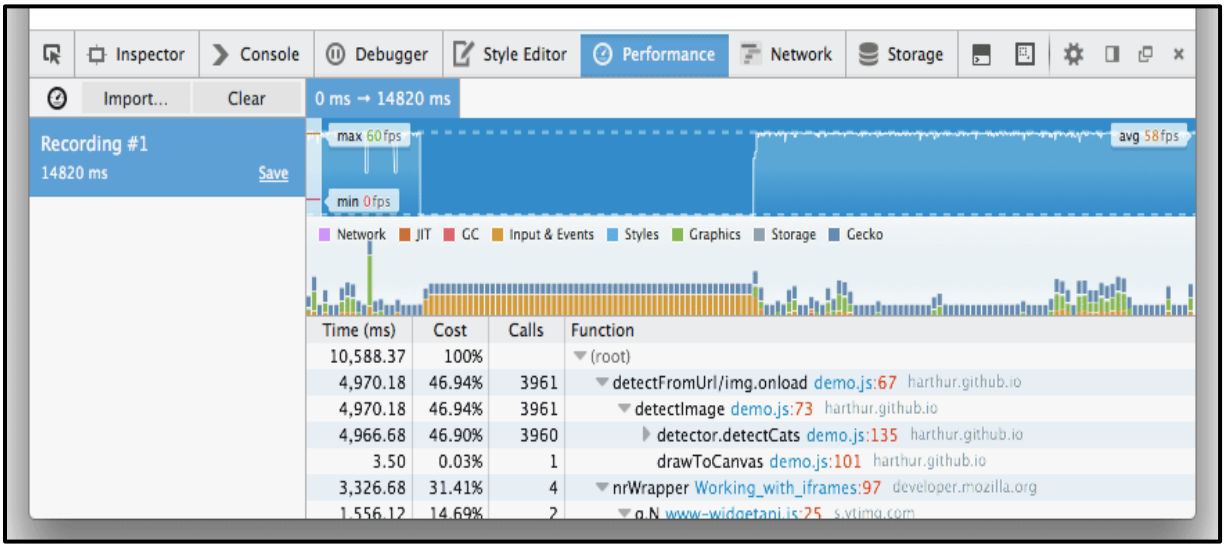

Figure 8. Performance data gathering tool.

(7) Smoke Testing: A preliminary test to reveal simple failures severe enough to reject a prospective software release. Smoke tests are a subset of test cases that cover the most important functionality of a software component. It is used to aid assessment of the main functions of the software. Smoke tests are commonly run on each new build to verify that the build is testable before the build is released into the hands of the test team. Smoke tests are also performed by testers before accepting a build for further testing. It can identify integration defect issues. Consequently, smoke testing is an important indicator if further in-depth programming is required before additional testing. Smoke testing can be performed manually or by using automated tools.

(8) Usability Testing: Testing focused on user-centered interaction to evaluate software. It measures the extent to which the software product can be used by specified users to achieve goals with effectiveness, efficiency, and satisfaction in a specified context[12]. Parts of usability testing may be performed manually using surveys to capture user experience and manually record metrics such as steps-per-task and errors-per-task. In some cases, an automated tool may be helpful for usability testing. Figure 9 displays a tool that captures the screen under test in video form while recording mouse movement and the image of the user to gauge facial reactions. These video captures are used in measuring system reliability when they capture defects.

Figure 9. An automated usability testing tool.

(9) Maintenance Testing: Performed on the already deployed software when it needs to be enhanced, changed, or migrated to other hardware. Software that undergoes maintenance testing usually needs to run at high availability, so maintenance testing is key for making sure there is little or no downtime. Examples of when maintenance testing is performed are when users need some more new features in the existing software, or when the end user might want to migrate the software to the latest hardware platform or change the environment such as the operating system or database version. As new capabilities are added to the software, new defects will often present themselves when executing maintenance testing events.

(10) Exploratory Testing: Exploratory Testing helps you expose defects that is often missed in either automated or manual testing since you are generally exploring new areas of the product. An approach that can be described as simultaneous learning, test design, and test execution by the personnel performing the test. It is a style of software testing that puts more emphasis on freedom and less on following a formal script. While the software is being tested, the tester learns things that may generate a new test to be run; perhaps during a more formal test.

It seeks to find out how the software actually works and to ask questions about how it would handle difficult and easy test cases. The expectations are more open in exploratory testing, which may lead to finding something which may be unexpected or missed during a more scripted formal test. In reality, many software system tests being performed has some type of exploratory aspect.

4.2 Load Testing.

a. Load testing are important events in identifying defects in an application related to buffer overflow, memory leaks, and mismanagement of memory. Within the DOD, Load Testing can be part of developmental testing (DT) or operational testing (OT) depending on the Program and SUT. Load Testing is performed to determine the software's behavior under both normal and anticipated peak load conditions. A very well-known and very common type of load testing involves applying ordinary stress to a software application to see if it can perform as intended. Load testing can deliberately induce failures so that the tester can analyze the risk involved at the breaking points to perhaps recommend changes. Load testing tools play a large part in being able to perform these functions. The tools allow the tester to generate loads, and in many cases, allow the tester to analyze the performance under different load types. Figure 10 displays a load testing tool which is able to create a custom load as well as display data analyzing the performance from the different loads.

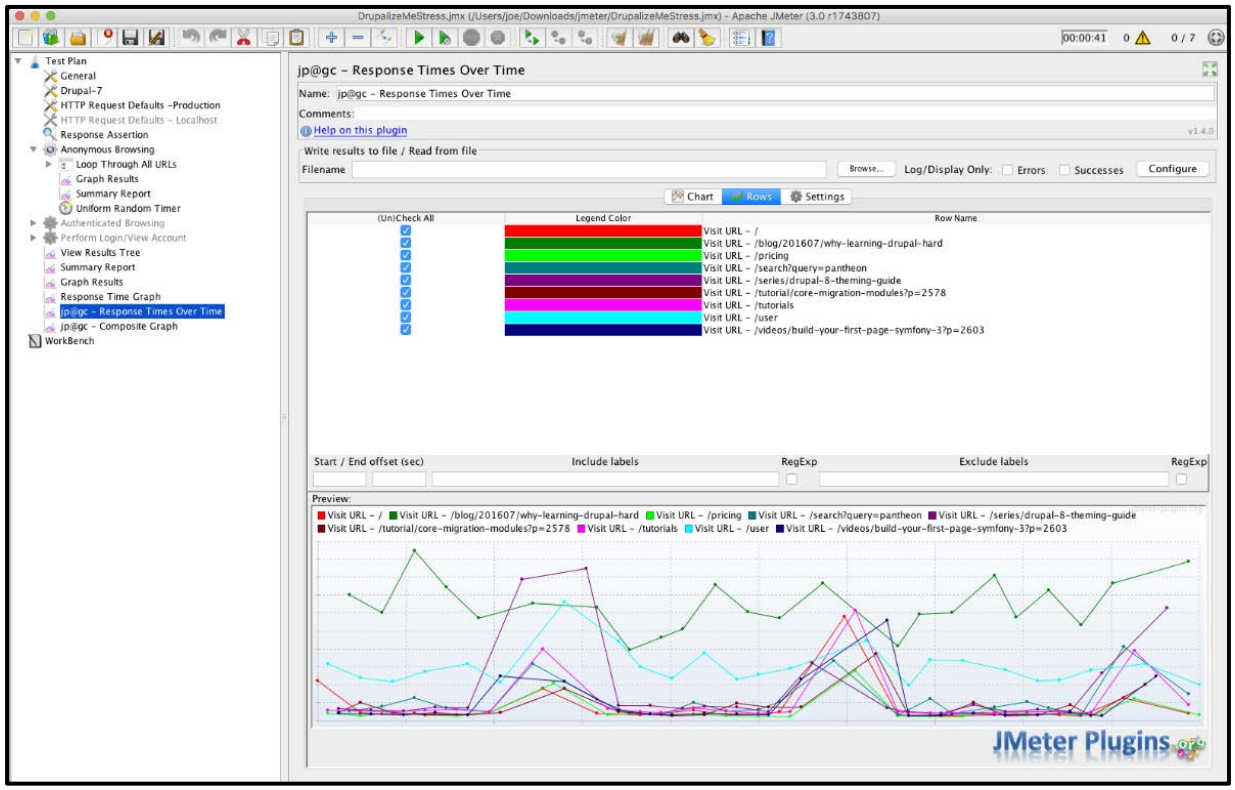

Figure 10. Load testing tool.

b. Because load testing is so closely related to the nature of the system being tested, load testing techniques are also application dependant[6]. Even though techniques may vary, there are a few common approaches to load testing.

4.2.1 Compression.

Compression is a strategy that takes an extreme load, but due to limitations of resources on a test, compresses the load test to a smaller and more manageable size. For example, an enterprise web application needs to support 100,000 users on Continental United States (CONUS) spread over four (4) application servers. Initial developmental testing is to take place in a special test facility. The Program Manager in charge of the test facility was only able to procure one (1) server. In this case, the load test will only need to simulate 25,000 users due to having only one server.

4.2.2 Replication.

Some load test requirements may be unusually difficult to actually perform. In some cases, the actual performance may destroy the system, or may not be economically feasible to perform the test. For example, imagery software needs to process up to 15 hours of imagery from an unmanned aerial vehicle (UAV) during a typical flight pattern. The load test will need to find out how well the software performs with all the imagery video data. Flying an actual UAV is not feasible due to airspace restrictions on scheduling a flight and the cost of manpower. Research has concluded that the UAV imagery load is similar to the load from video taken with a certain type of web camera. The web camera imagery was able to replicate the UAV imagery for a feasible load test.

4.3 Regression Testing.

Regression testing ensures that previously developed and tested software still performs the same way after it is changed or interfaced with other software. It finds defects which may be introduced from fixing other defects. Thus, regression testing is useful and needed, whenever a new version of a program is obtained by modifying an existing version[13]. Regression testing can be applied in each phase of software development. Within the DOD, Regression Testing can be part of DT or OT depending on the Program and SUT. There are several regression testing techniques.

4.3.1 Retest All.

a. This technique checks all test cases on the current software to test full integrity. It is the most resource intensive since it re-runs all test cases. While the retest-all strategy might be the least risky, it does suffer from the fact that it may be too expensive to execute. In many cases, automated tools are used to aid in making sure all test cases are executed. Figure 11 shows an automated tool used for making sure all test cases are executed for Unit Testing.

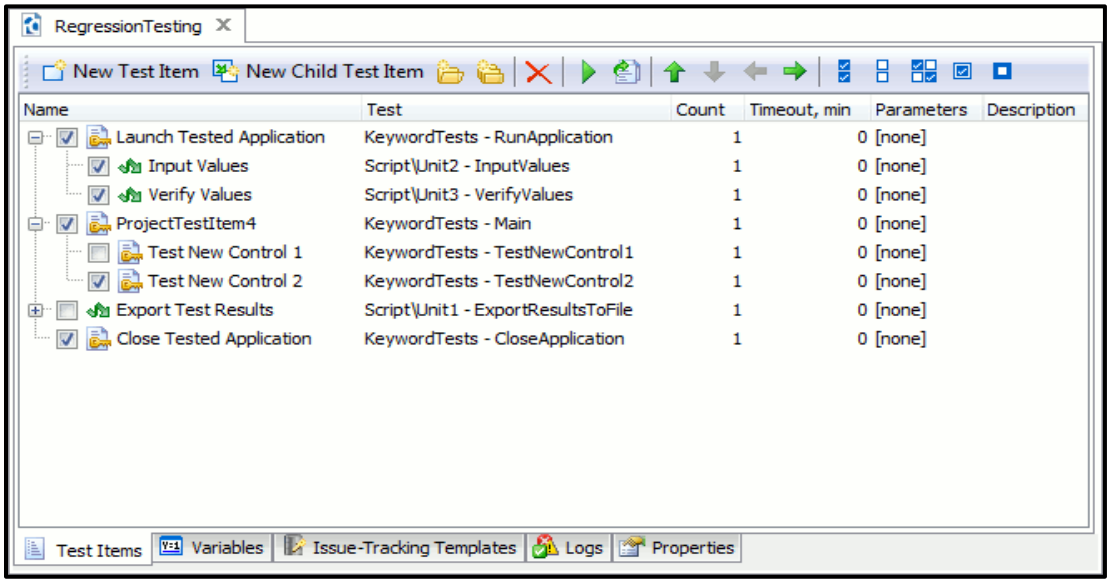

Figure 11. Automated regression test tool.

b. Retesting all unit tests with an automated tool is widely used since many unit testing, and even software development tools, come with automated regression testing capability. Retesting-all gets more resource intensive when the application of the strategy moves closer to a full system test as opposed to a unit test.

4.3.2 Random Selection.

Random selection of tests for regression testing is one possible method to reduce the number of tests. In this approach tests are selected randomly from the whole test suite. The tester can decide how many tests to select depending on the level of confidence required and the available time for regression testing. Under the assumption that all tests are equally good in their fault detection ability, the confidence in the correctness of the unchanged units increases with an increase in the number of test sampled and executed successfully. However, in most practical applications such an assumption is unlikely to hold because some of the sampled tests might now bear any relationship to the modified units, while others might. This is the prime weakness of the random selection approach to regression testing. Nevertheless, random selection is better than no regression testing at all.

4.3.3 Test Selection.

Another technique to reduce the number of tests is to select only tests that guarantee the execution of the modified units and the units that might be impacted by other modified units. This technique uses methods to determine the desired test subset and aim at obtaining a minimum regression test suite. Tests that traverse the units that had been modified are good candidates to select for regression testing. The sophistication of techniques to select modification traversing tests requires automation. It is impractical to apply these techniques to

large software systems unless a tool is available that incorporates at least one safe test selection technique. Further, while test selection appears attractive from the test effort point-of-view, it might not be a practical technique when tests are dependent on each other in complex ways and that this dependency cannot be incorporated in the test selection tool. Also, further minimization can be achieved by getting rid of redundant tests. A test is considered redundant if another test achieves the same objective. The objective in this context may be in terms of control flow, data flow, or requirements. This type of minimization may not be safe in the sense that tests not selected may actually be useful in revealing errors despite the fact they are considered redundant, according to some criterion.

4.3.4 Test Prioritization.

One approach to regression testing is through prioritization. In this approach, a suitable metric is used to rank all tests. A test with the highest rank has the highest priority, with the rank also determining the priority for the remaining tests. Prioritization does not eliminate any test from the test suite. Instead it allows the tester to decide which tests to select based on their relative priority and individual relationships based on sequencing and other requirements. A ranked list of tests can be used to decide when to stop testing in case not enough resources are available to run all the tests.

4.3.5 Hybrid Approach.

A hybrid technique of Test Selection and Test Prioritization can be a very effective regression test strategy. Even Random Selection can be incorporated into a hybrid strategy. If a handful of tests have the same prioritization, randomly selecting from that subset may be the most useful approach.

5. SOFTWARE RELIABILITY GROWTH.

As mentioned in Section 1.1, software systems are not subject to wear-out, fatigue, or other forms of degradation. It can, however, fail due to faulty functionality, timing, sequencing, data, and exception handling. Software reliability problems are deterministic in the sense that when each time a specific set of inputs is applied to the software system, the result will very well likely be the same. This is clearly different from hardware systems, for which the precise moment of failure, and the precise cause of failure, can differ from replication to replication. In some situations, reliability errors are attributed to a full system and no distinction is made between subsystems or components. This attribution is appropriate in many applications. However, as is the case for any failure mode, there are times when it is appropriate to use separate metrics and separate assessments of subsystem or component reliabilities, which can then be aggregated for a full-system assessment. This separate treatment is particularly relevant to software failures given the different nature of software and hardware reliability. Industry preferred reliability growth solutions are identified below:

5.1 Models.

a. Software reliability growth models use failure data from testing to forecast the failure rate. The models depend on the assumptions about the fault rate during testing, which can either be increasing, peaking, decreasing, or some combination of decreasing and increasing. Some models assume that there is a finite and fixed number of inherent defects while others assume that the value is infinite. Some models require effort for parameter estimation while others have only a few parameters to estimate. Some models require the exact time in between each failure found in testing, while others only need to have the number of failures found during any given time interval such as a day[14]. A list of software reliability growth models is provided in Table 2.

TABLE 2. LIST OF SOFTWARE RELIABILITY GROWTH MODELS[13]

MODEL NAME	INHERENT DEFECT COUNT (assumption)	EFFORT REQUIRED (parameter estimation)	REQUIRES EXACT TIME BETWEEN FAILURES
Increasing Fault Rate			
Weibull	Finite/not fixed	High	Yes
Peak			
Shooman Constant Defect Removal Rate Model	Finite/fixed	Low	Yes
Linearly Decreasing			
General exponential models including: · Goel-Okumoto (exponential) · Musa Basic Model · Jelinski-Moranda	Finite/fixed	Medium	Yes
Shooman Linearly Decreasing Model	Finite/Fixed	Low	Yes
Duane	Infinite	Medium	No
Non-Linearly Decreasing			
Musa-Okumoto (logarithmic)	Infinite	Low	Yes
Shooman Exponentially Decreasing Model	Finite/fixed	High	Yes
Log-Logistic	Finite/fixed	High	Yes
Geometric	Infinite	High	No
Increasing and then decreasing			
· Yamada (Delayed) · S-shaped	Infinite	High	Yes
Weibull	Finite/not fixed	High	N/A

b. Software reliability growth models have, at best, limited use for making predictions as to the future reliability of a software system in development for several reasons. Software reliability growth models have limited use for software that is complex. For most Army enterprise systems, existing models cannot be properly applied. Most important, the pattern of reliability growth evident during the development of software systems is often not monotonic because corrections to address defects will at times introduce additional defects. Other deficiencies in such models relevant to software are the substantial dependence on time as a

modeling factor, the dynamic behavior of software systems, the failure to take into consideration various environmental factors that affect software reliability when fielded, and hardware interactions. The dynamic behavior of software systems as a function of the environment of use, the missions employed, and the interactions with hardware components, all complicate modeling software reliability. Given the previously stated challenges, Reliability Growth Models are helpful early in system development, when they can help determine the scope, size and design of the testing programs[15].

5.2 Trend Charting.

a. While models help estimate reliability growth, reporting reliability growth trends is also vital and a better choice for later in the lifecycle. One of the most important principles found in commercial best practices are the benefits of displaying collected data in terms of trend charts for tracking progress. This can be best illustrated with the use of analytics dashboards in large-scale software systems. Analytics dashboards provide easily interpretable information that can help many users; including developers, managers, and testers. The dashboard shown in Figure 12 can cater to a variety of requirements.

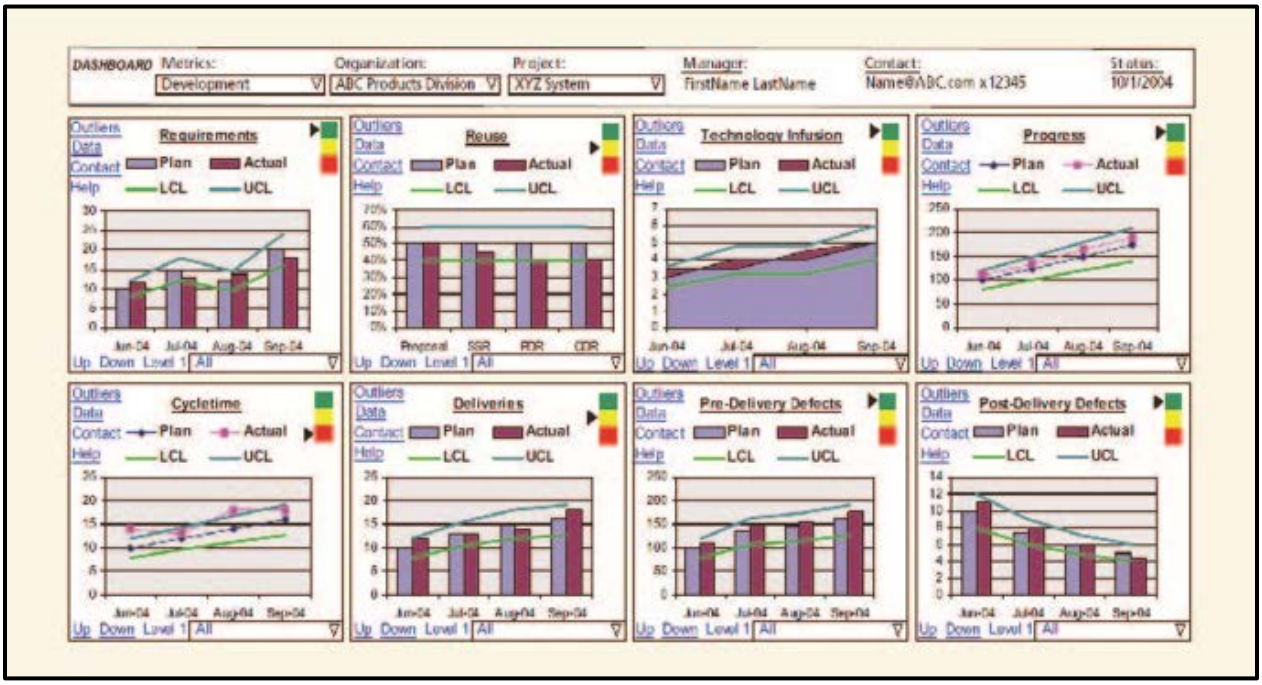

Figure 12. Software analytics dashboard[15].

b. Several of the metrics shown in Figure 12 (for example, the trend of post-delivery defects), can help assess the overall stability of the system. Organizations should define data trends that are reflective of success in meeting software requirements so that, over time, one could develop statistical tests that could effectively discriminate between successful and unsuccessful development programs. Analytics dashboards can also give context-specific help, and the ability to drill down to provide additional useful details[15]. While Reliability Growth

Modeling may be helpful in the early stages of system development, Trend Charting is just as important during the later stages of development and throughout all of software testing.

5.3 Selection of Reliability Methodology.

Models and Trend Charting are indicated as the best means of determining reliability growth for a software system. In the medical and business enterprise systems, the complexity of the environment would favor Trend Charting, which allows the program office to track the maturity of their software product over time. In addition, these programs have extended development cycles, often measured in years, and therefore Trend Charting allows for a consistent way for senior management in tracking progress in meeting milestones. The end goal is to deploy a reliable software product that the user community can use to execute their mission to support the warfighter.

5.4 Software Reliability Metrics.

Software reliability metrics can provide insight into the current reliability of the software as well as support trend analysis. Listed are the three main categories of software reliability metrics, as well as examples of the metric being described.

a. Frequency of failure metrics. These measures can be thought of with respect to time.

(1) Mean Time Between Software Failure (MTBSF). A measure of the usage time divided by the number of software failures.

(2) Failure on Demand (FoD). The rate of failure for a given type of software transaction. In distinction from MTBSF which is time based, the FoD metric focuses on the failure rate of a given task.

b. Defect status metrics. These measures could include open defects per priority or open defects or number of problems per size (e.g., lines of code) of the software.

(1) Defect count by priority. This metric should be presented at defined intervals to support trend analysis.

(2) Defect aging. This gives the number of unresolved defects by priority.

(3) Defect density. The number of known defects per size code (e.g., defects per 1,000 lines of source code).

c. Defect management metrics. These measures could include the efficiency and effectiveness of an organization removing defects from the software (such as defect removal efficiency, which is the percentage of defects removed from software compared to total number of known defects).

APPENDIX A. ABBREVIATIONS.

AD No.	Accession Number
ATEC	Army Test and Evaluation Command
C4I	command, control, communications, computers, and intelligence
CLI	command line interface
CONUS	Continental United States
CPU	central processing unit
DARS	Defense Acquisition Regulating System
DOD	Department of Defense
DT	developmental test
DTIC	Defense Technical Information Center
FoD	Failure On Demand
FRS	functional requirement specification
GUI	graphical user interface
IEEE	Institute of Electrical and Electronics Engineers
MTBSF	Mean Time Between Software Failure
OT	operational test
SDLC	software development life cycle
SME	subject matter expert
SRS	system requirement specification
SUT	system under test
T&E	Test and Evaluation
TOP	Test Operations Procedure
UAV	unmanned aerial vehicle

(This page is intentionally blank.)

APPENDIX B. REFERENCES.

1. IEEE 1633 Recommended Practice On Software Reliability, Standards Committee IEEE Reliability Society, The Institute of Electrical and Electronics Engineers Inc., (New York, NY) 22 September 2016.

2. Software Reliability, 18-849b Dependable Embedded Systems, Carnegie Mellon University, Jiantao Pan, Spring 1999.

3. "Software Reliability Testing", https://en.wikipedia.org/wiki/Software_reliability_testing, retrieved 27 March 2017.

4. "Load Testing", https://en.wikipedia.org/wiki/Load_testing, retrieved 29 March 2017.

5. "Regression Testing", https://en.wikipedia.org/wiki/Regression_testing, retrieved 29 March 2017.

6. Software Testing: A Craftsman's Approach, Paul C. Jorgensen, CRC Press (Boca Raton, FL) 2015.

7. "Agile Software Development", https://en.wikipedia.org/wiki/Agile_software_development, retrieved 20 April 2018.

8. "Agile Testing", https://en.wikipedia.org/wiki/Agile_testing, retrieved 20 April 2018.

9. "Unit Testing", https://en.wikipedia.org/wiki/Unit_testing, retrieved 27 March 2018.

10. "Integration Testing", https://en.wikipedia.org/wiki/Integration_testing, retrieved 27 March 2018.

11. "System Testing", https://en.wikipedia.org/wiki/System_testing, retrieved 27 March 2018.

12. TOP 01-2-700, Software Usability Testing, 08 November 2017.

13. Foundations of Software Testing 2nd edition, Aditya P. Mathur, Dorling Kindersley (India) 2013.

14. "List of Software Reliability Models", https://en.wikipedia.org/wiki/List_of_software_reliability_models, retrieved 29 March 2017.

15. Reliability Growth: Enhancing Defense System Reliability, Panel on Reliability Growth Methods for Defense Systems, National Academies Press (Washington, DC) 2015.

(This page is intentionally blank.)

APPENDIX C. APPROVAL AUTHORITY.

CSTE-TM

MEMORANDUM FOR

Commanders, All Test Centers
Technical Directors, All Test Centers
Directors, U.S. Army Evaluation Center
Commander, U.S. Army Operational Test Command

SUBJECT: Test Operations Procedure 01-1-063 Enterprise Software Reliability, Approved for Publication

1. Test Operations Procedure (TOP) 01-1-063 Enterprise Software Reliability, has been reviewed by the U.S. Army Test and Evaluation Command (ATEC) Test Centers, the U.S. Army Operational Test Command, and the U.S. Army Evaluation Center. All comments received during the formal coordination period have been adjudicated by the preparing agency. The scope of the document is as follows:

The topics in this TOP include guidance for conducting various software tests and systematic data collection for increasing enterprise software reliability. These enterprise systems are currently under acquisition and therefore testing is focused on establishing reliability for the system during the full development lifecycle. The identified tests in this TOP allow the testing organization to select appropriate events during the acquisition lifecycle to obtain over time increased reliability of the system by eliminating defects of the System Under Test.

2. This document is approved for publication and will be posted to the Reference Library of the ATEC Vision Digital Library System (VDLS). The VDLS website can be accessed at https://vdls.atc.army.mil/.

3. Comments, suggestions, or questions on this document should be addressed to U.S. Army Test and Evaluation Command (CSTE-TM), 6617 Aberdeen Boulevard-Third Floor, Aberdeen Proving Ground, MD 21005-5001; or e-mailed to usarmy.apg.atec.mbx.atec-standards@mail.mil.

FONTAINE.RAYMOND.G.1228612770 Digitally signed by FONTAINE.RAYMOND.G.1228612770 Date: 2019.07.24 08:43:06 -04'00'

RAYMOND G. FONTAINE
Associate Director
Test and Evaluation Management Directorate

FOR

MICHAEL J. ZWIEBEL
Director
Test and Evaluation Management Directorate

(This page is intentionally blank.)

Forward comments, recommended changes, or any pertinent data which may be of use in improving this publication to the following address: T&E Policy and Standardization Division (CSTE-TM), U.S. Army Test and Evaluation Command, 6617 Aberdeen Boulevard, Aberdeen Proving Ground, Maryland 21005-5001. Technical information may be obtained from the preparing activity: U.S. Army Electronic Proving Ground, 2000 Arizona Street, Fort Huachuca, AZ 85613-7063. Additional copies can be requested through the following website: https://www.atec.army.mil/publications/documents.html, or through the Defense Technical Information Center, 8725 John J. Kingman Rd., STE 0944, Fort Belvoir, VA 22060-6218. This document is identified by the accession number (AD No.) printed on the first page.