

Building an Efficient Quality Testing Process

A Technical White Paper

By Aleksandr Kuzmin,
Sun Microsystems, Inc.

PRELIMINARY

Copyright © 2006 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A. All rights reserved. U.S. Government Rights - Commercial software. Government users are subject to the Sun Microsystems, Inc. standard license agreement and applicable provisions of the FAR and its supplements. Use is subject to license terms. Sun, Sun Microsystems, Java, and JavaTest are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd. Products covered by and information contained in this service manual are controlled by U.S. Export Control laws and may be subject to the export or import laws in other countries. Nuclear, missile, chemical biological weapons or nuclear maritime end uses or end users, whether direct or indirect, are strictly prohibited. Export or reexport to countries subject to U.S. embargo or to entities identified on U.S. export exclusion lists, including, but not limited to, the denied persons and specially designated nationals lists is strictly prohibited. DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Copyright © 2006 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, Etats-Unis. Tous droits réservés. L'utilisation est soumise aux termes de la Licence Sun, Sun Microsystems, Java, et JavaTest sont des marques de fabrique ou des marques déposées de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays. UNIX est une marque déposée aux Etats-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company, Ltd. see above

Les produits qui font l'objet de ce manuel d'entretien et les informations qu'il contient sont regis par la législation américaine en matière de contrôle des exportations et peuvent être soumis au droit d'autres pays dans le domaine des exportations et importations. Les utilisations finales, ou utilisateurs finaux, pour des armes nucléaires, des missiles, des armes biologiques et chimiques ou du nucléaire maritime, directement ou indirectement, sont strictement interdites. Les exportations ou reexportations vers des pays sous embargo des États-Unis, ou vers des entités figurant sur les listes d'exclusion d'exportation américaines, y compris, mais de manière non exclusive, la liste de personnes qui font objet d'un ordre de ne pas participer, d'une façon directe ou indirecte, aux exportations des produits ou des services qui sont régi par la législation américaine en matière de contrôle des exportations et la liste de ressortissants spécifiquement désignés, sont rigoureusement interdites. LA DOCUMENTATION EST FOURNIE "EN L'ETAT" ET TOUTES AUTRES CONDITIONS, DECLARATIONS ET GARANTIES EXPRESSES OU TACITES SONT FORMELLEMENT EXCLUES, DANS LA MESURE AUTORISEE PAR LA LOI APPLICABLE, Y COMPRIS NOTAMMENT TOUTE GARANTIE IMPLICITE RELATIVE A LA QUALITE MARCHANDE, A L'APTITUDE A UNE UTILISATION PARTICULIERE OU A L'ABSENCE DE CONTRE FAÇON.

Contents

Why Testing?	5
Testing in a Product's Life	6
Software Development Life Cycle	6
Capability Maturity Model	6
Importance of Different Types of Testing	7
Types of Tests	7
Black-box and White-box Test Development Techniques	12
Black-Box Tests	13
White-Box Tests	13
Test Process Overview	13
The Criteria of a Good Test	14
Test Harnesses	14
Test Harnesses for the Java Programming Language	15
CMM Maturity Levels	16
JavaTest Harness Compared With CMM Requirements	17
Initial (CMM Level 1)	17
Repeatable (CMM Level 2)	17
Defined (CMM Level 3)	18
Managed (CMM Level 4)	19

Optimized (CMM Level 5)	19
Examples	20
Conclusion	20
References	21
Links	21
Acronyms	21

Building an Efficient Quality Testing Process

This paper describes the steps for building an efficient test methodology that fulfils the requirements of most advanced software development methods and schemes such as Capability Maturity Model and eXtreme Programming. It reviews all types of tests applied during the testing of complex software products and the leading test harnesses available in the market, including JavaTest™ harness.

Why Testing?

According to various research works, almost half of software maintenance costs are attributable to fixing defects introduced in the initial software development. These defects should have been found by thorough and effective software testing. Thus, properly organized testing can significantly reduce software maintenance cost. It is important to properly fit the testing into product's evolution.

Testing in a Product's Life

It is useful to consider test methodology in the context of a software product's entire life cycle, as follows:

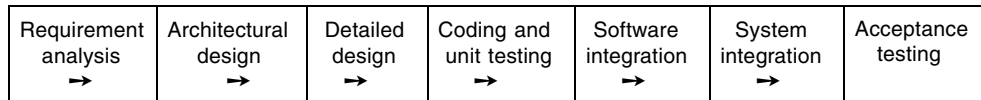


FIGURE 1 Software Product Life Cycle

Software Development Life Cycle

The typical product development cycle can be represented by the V-model or “waterfall” diagram, in which the important steps of product development cycle follow each other. It is important to develop such a test strategy and a variety of tests to gain the highest quality at the end of cycle.

Almost all modern test theories suggest that product testing should begin at the lowest level, and proceed along the product life cycle through integration testing, up to the final acceptance testing stage, where the product is nearly ready to be released to customers. Product development methods recommend this approach in cases where the end system needs to be reasonably robust. Reliability is achieved through the different granularity of testing, providing a greater degree of confidence in the quality of the software at every step forward.

Capability Maturity Model

The Capability Maturity Model (CMM) developed by Carnegie Mellon Software Engineering Institute is a very popular and efficient methodology used for developing and optimizing an organization's software development process.

CMM establishes a framework for continuous process improvement and is more explicit in terms of standards and definitions than the old basic PLC model in defining the means to be applied.

Importance of Different Types of Testing

Obviously, a system that is built from thoroughly tested components is more likely to be of higher quality than a system that is built from components that are not thoroughly tested. Tests may be automated or run manually, using the black box or white box methodologies.

Types of Tests

In general, the following types of tests are available:

- Compatibility
- Functional
- Regression
- Benchmarks
- Application-like
- Unit
- Stress
- Integration

Compatibility Tests

Compatibility tests are the most explicit way of verifying the correctness of public interfaces and API methods. It is customary to develop compatibility tests using the black-box technique. Compatibility tests must strictly follow the specification against which they are developed. They should be written in a way that excludes any doubt in test scenario reading. A test harness for compatibility tests must be able to manage a large amount of tests and it should support a test exclusion mechanism. This mechanism should be easy to configure for new platform environments. JavaTest harness is an ideal tool for compatibility testing because it is designed with these requirements in mind.

Functional Tests

Functional tests verify that a product conforms to its design specification document and correctly performs all its required functions. This includes a number of tests that perform a module-by-module verification of assertions, using a wide range of regular and erroneous input data. This can involve testing of the product's user interface, security, installation, networking, and so on according to the design

specification. While unit or compatibility tests verify the correctness of preselected methods and classes, functional tests exercise the entire system the way the user will run it, including rich scenarios and subcases.

The link in this footnote lists a number of well-known functional test tools.¹

Functional tests are no replacement for compatibility tests or pre-integration unit tests. They strive to test a piece of functionality as the user sees it. Thus, they may not methodologically include every aspect of the product. A functional test is more focused on a single “vertical” of functionality, typically linked to a particular use case.

The first phase of a functional test case involves setting up initial conditions. Most modern programming languages, such as the Java™ programming language and C++, allow the hiding of internal variables from external software modules. This is known as *encapsulation*, and is in direct conflict with the requirement to initialize that internal data for testing purposes. The concept of data hiding is generally regarded as a beneficial feature of a programming language, but is actually a liability when the needs of testing are considered. In such cases, a white-box testing approach might help.

Regression Tests

Similar to functional tests, regression tests allow consistent, repeatable validation of each new version of a product. Such testing ensures reported product defects are fixed for each new build or release and that no new regressions were introduced during defect fixing or the normal development cycle. In the context of automation, regression tests usually simply play back previously developed test scenarios to compare the latest build of the product to the golden test suite established at an earlier point in time. Regression testing is intended to reveal any differences that might have been introduced into the application since the last verification. Testers can then investigate these differences to determine whether or not they’re actual defects.

A test harness has the following specific requirements for regression testing:

- The ability to manage a large number of tests
- A mechanism to exclude certain tests from a test run
- The support for time stamps in test result files

Benchmark Tests

Benchmark tests are designed to measure and establish a baseline to track performance changes.

1. <http://www.aptest.com/resources.html#app-func>

Field-reported problems are often not system crashes or incorrect system responses, but system performance degradation or problems handling required system throughput. In such cases, the software system is often not properly tested to assess its expected performance.

Performance testing issues differ in a number of ways from functional testing issues.

To efficiently test software performance, it is necessary to set performance requirements defined by a concrete, quantifiable formula or specification. Test monitors that run benchmark tests must be able to record, archive, and timestamp the results for comparisons with other versions of the product.

The number and variety of benchmark tests should be enough to obtain a multidimensional and extensive picture of performance. A test harnesses should be able to manage a large number of tests.

On many occasions, the quality assurance team at Sun Microsystems has successfully applied the JavaTest harness to run benchmark tests.

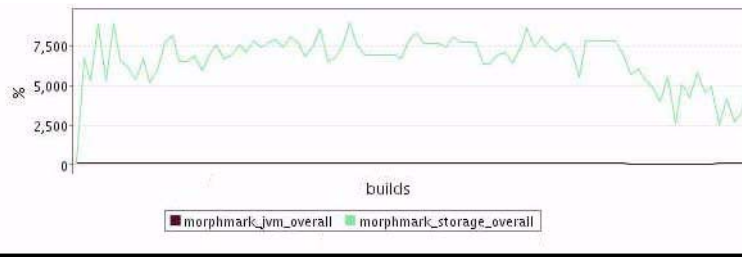


FIGURE 2 *Sample Java Wireless Client Data by Sun Microsystems, Inc.*

Unit Testing

The unit test is the lowest level of testing performed during software development, in which individual units, classes or files of a product are tested without connection to other parts. The cornerstone of the Extreme Programming (XP) methodology is the unit test. In this methodology it is implied that all units, classes or files in the system are unit tested. Developers must release unit test code to the code workspace with the code it tests. Unit level testing is an effective way to increase software reliability, and should always be performed in parallel with software design and implementation.

Unit testing verifies correct functionality of individual software units such as procedures or functions in a procedural language or classes in an object-oriented programming language. Any external functions called are replaced with stub (or wrapper) functions to allow full control of the test environment.

The best practice for unit tests is to use a separate source file called a test driver to set up initial conditions, to make calls to the software under test, and to verify the results. Three requirements must be met by the software under test to support the testing process:

Accessibility - An external test driver must be able to initialize the data.

Controllability - An external test driver must be able to direct the flow of control.

Visibility - An external test driver must be able to check the data.

In addition, other practices can help to ensure easy and successful unit testing:

- Requirements must be complete and documented.

Software behavior cannot be verified if correct behavior is not known. Unit level requirements (not the same as system requirements) are a valuable business asset.

- Test infrastructure must be built.

A complete test infrastructure includes test drivers, stub functions, libraries for checking data, code parsing, and coverage instrumentation tools. It does not make sense to regenerate this testing infrastructure for each project when a single automated testing environment can do this for all projects.

- Only appropriate tools must be used.

Sometimes developers attempt unit testing with simple debugging tools, exercising their code with various input conditions and with simple ad hoc testing. Instead, unit tests should be maintained as product code and run under a test harness.

Features to simplify testing (testability) must be embedded into the design of the product.

Many publications warn about this obstacle to unit testing, because the software often has to be rewritten to build a successful test. It takes time to redesign software, and the modifications to support testability are typically not incorporated into the official source code archive, which presents the risk that the software being tested is not the same software that will be delivered in the final product.

End-User Tests

End user testing (also known as application-like testing) involves creating rich, stress-like workloads simulating the behavior of real-life applications. In addition, at this stage of testing, it's important to focus on the user experience. Very often application-like tests are developed as interactive (manual) tests. Here we see a requirement to create a convenient environment for a test operator to understand the expected results of the test, to allow a time to conduct the test, and to report the

results of test execution. Application-like tests help to unveil unexpected defects that could not be found during static analytical testing or runtime testing. The JavaTest-based version of Java Device Test Suite has many end-user tests.

Stress Tests

Stress testing investigates the stability or response of the system under peak loads. It involves testing beyond normal operational capacity. Testing is conducted to evaluate a system or component at or beyond the limits of its specified requirements to determine the load under which it fails and how it fails. The desired result is a graceful degradation under load leading to non-catastrophic failure. Stress testing is performed using the same process as benchmark testing, but introduces a very high level of simulated load.

Integration Tests

Integration tests are designed to ensure that all elements of the product are tested with supplementary services and applications as part of an end-to-end solution. The focus is not necessarily on finding code bugs, but on validating the functioning of the entire platform. Integration tests usually test the entire product using a black-box technique. Beyond that, the tools used to write an integration test are the same as those to write stress tests or application-like tests. Only the approach to testing is different. Whether you need one, or the other, or both, depends on the level of rigor you need in your tests, and how your system is constructed. In general, though, true integration tests are rarer than functional tests and unit tests.

Large systems might require many integration phases, beginning with assembling modules into low-level subsystems, then assembling subsystems into larger subsystems, and finally assembling the highest level subsystems into the complete system. It is important to understand the relationship between functional module testing and integration testing. In one view, modules are rigorously tested in isolation using stubs and drivers before any integration is attempted. Then, integration testing concentrates entirely on module interactions, assuming that the details within each module are accurate. At the other extreme, module and integration testing can be combined, verifying the details of each module's implementation in an integration context.



The following figure is cited in *Applied Software Measurement*.¹ It compares the time needed to prepare tests, to execute tests, and to fix defects (normalized to one function point), showing that unit testing is about twice as cost effective as integration testing, and more than three times as cost effective as system testing.

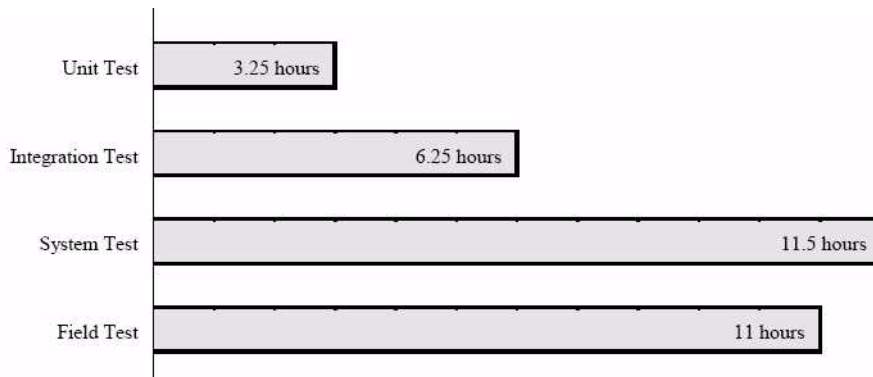


FIGURE 3 Cost of Fixing Defects at Various Stages

Black-box and White-box Test Development Techniques

Test development methodologies can be broadly divided into *white-box* and *black-box* testing.

1. Jones, C., *Applied Software Measurement, Assuring Productivity and Quality*, McGraw-Hill, New York, NY, (1991).

Black-Box Tests

The main idea of black-box testing is to test a class purely by its public interface, without assumption or knowledge of internal structure of the class implementation. The usual approach is to set up a series of test scenarios (or test cases) in each of which a call or series of calls are made typically to one or more public member functions. The checking of return values and modifiable parameters determine whether the object is doing what it is supposed to be doing. Examples of black box tests include compatibility tests, integration tests, and application-like tests.

White-Box Tests

White-box tests gain access to the private part of a class to increase efficiency. Typical advantages gained by applying this technique are the opportunity to set or check the value of private data items, and the ability to call private or package functions directly. Sometimes you need a test case that verifies the behavior of the object in a hard-to-reach state. One way to do this is to guess the sequence of public method calls which would put the object in the desired state. Alternatively, you can reach this state using the white-box approach. Unit tests and functional tests quite often use the white-box testing approach.

Test Process Overview

With either black-box or white-box methodology, the test process consists of the following steps:

1. The test methodology begins in the *requirements* phase of the software development life cycle. (Refer to FIGURE 1 on page 6.)
2. During the *design* phase, testers work with developers in to determine what aspects of a design are testable and under what parameters those tests work.
3. \During the test *planning* phase, testers develop test strategy and test plan documents.
4. The test *development* phase involves the creation of test procedures, test scenarios, test cases, and test scripts to use in testing software.
5. Testers *execute* the software based on the plans and tests and report any errors found to the development team.
6. Once testing is completed, testers generate metrics and make final *reports* on their test effort and whether or not the software tested is ready for release.

7. Testing is repeated to make sure the defects were eliminated by the development team.

The Criteria of a Good Test

According to the best practices established by CMM and XP, a formal written test case specifies a known input and an expected output before a test is executed. The known input must test a precondition and the expected output must test a postcondition.

Test Harnesses

This sections reviews how a test harness fits into the software development process. It details the general attributes that characterize an effective test harness.

In a properly managed project, the testing team is involved in all steps of product development. When considering all the steps from test development through the handling of test results, it's useful to elaborate the practical requirements for a test harness.

The proper organization of test sources and descriptions keeps the testing procedure on track and measures the quality of the product under development against its functional specification. An effective test harness will conveniently link test descriptions and requirements with a corresponding test execution log, ensuring traceability throughout the test execution process. Using a test monitor, a user should easily see what percentage of the test execution is done, how many of these tests have been completed, how many are still to be run, and how many have passed or failed because of a functional defect or a configuration error.

After the test design and development cycles have been completed, the testing team is ready to start running tests. To test the system as a whole, testers need to perform various types of testing – compatibility, functional, regression, benchmark, unit and integration – each with its own set of requirements, schedules and procedures. A test monitor should allow testers to configure the test environment to run unattended, to execute tests in parallel, and to handle situations where tests might hang or when the system might encounter resource limitations. A test monitor should have an adjustable time-out (the time that the test monitor pauses and waits for test completion). It should have protection against manual modification of test result data, and a mechanism for exclusion of problematic tests. A test monitor should be able to operate with large amounts of data. It should generate reports that are browsable over a network.

Test Harnesses for the Java Programming Language

The following tables summarize the most popular oriented test harnesses.¹ The first table lists that are oriented towards the Java programming language. To one degree or another, all of them demonstrate the general attributes that characterize an effective test harness.

TABLE 1 Test Harnesses for the Java Programming Language

Name	Description
GrandTestAuto	GrandTestAuto is a tool for unit testing applications written in the Java programming language
JavaTest	A test framework suitable for all programming languages and platforms.
JTestCase	A data-driven testing add-on for JUnit
JUnit	Popular Java programming language regression and unit testing framework.
TBrun	Automated Unit Test tool
TestMentor	A functional test and test modeling tool for the Java programming language
TestGen4J	TestGen4J is a collection of the tools that automatically generates unit test cases
VectorCAST	Automates the testing of individual software components prior to system test

1. <http://www.aptest.com/resources.html#app-jsource>

Test Harnesses for Other Popular Programming Languages

In addition to Java technology-based test harnesses, the next table summarizes the leading general-purpose test execution systems provided for other popular programming languages. Again, to some degree all of these have the general attributes that characterize an effective test harness. However, some do not provide support for the Java programming language.

TABLE 2 Other Leading Test Harnesses

Name	Description
ApTest	Powerful, configurable, easy to us. Good for interactive tests.
Gauntlet	Borland's all-purpose test automation system
SilkCentralTestManager	Segue's (recently Borland) automated software testing management
CTB	Test harness generator, unit/integration testing environment
GRS	Parasoft's all-languages test result control system
RationalTest	RealTime Unit Testing performs black-box/functional testing for C, C++ and ADA
TestDirector	Mercury's system of running tests and analyzing test results

In a most comprehensive way, the JavaTest harness has the attributes that characterize an effective test harness. It provides an intuitive user interface, the convenient presentation of test descriptions, and a traceable test execution log. The JavaTest harness monitor can operate with a large number of tests, running them simultaneously even on platforms with limited resources. The features which differentiate the JavaTest harness from other test harnesses are its flexibility, the support of tests written in any language, and the convenient format of test results that can be post-processed by external tools. In addition, it is optimized for testing conformance with Java technology.

CMM Maturity Levels

CMM details five maturity levels in software processes.

In the beginning, at the *initial* level, processes are disorganized, even chaotic. Success is likely to depend on individual efforts, and is not generally repeatable, because processes would not be sufficiently defined and documented to allow them to be replicated.

At the *repeatable* level, basic project management techniques are established, and successes could be repeated, because the requisite processes would have been made established, defined, and documented.

At the *defined* level, an organization has developed its own standard software process through greater attention to documentation, standardization, and integration.

At the *managed* level, an organization monitors and controls its own processes through data collection and analysis.

At the *optimizing* level, processes are constantly being improved through monitoring feedback from current processes and introducing innovative processes to better serve the organization's particular needs.

JavaTest Harness Compared With CMM Requirements

Here is an item-by-item comparison of CMM requirements and the capabilities of the JavaTest harness.

Initial (CMM Level 1)

Due to its flexibility, the JavaTest harness is a great aid in building a prototype version of any test suite. The pluggable test runners of the JavaTest harness allow rapid expansion of test framework functionality. These features are useful at the *initial* level of CMM.

Repeatable (CMM Level 2)

CMM states that:

If low-level testing is to be carried out, plans for the provision of tools will be required.

The incorporation of fully-tested components into the finished product is one of the best ways to promote that product's chances of being reliable. Sometimes the variety of tests, the number of tests in the test suite, and the dependency to system environment might push QA teams to sacrifice the integrity or automation of testing claiming that "it's too complex" or "too expensive."

Although test adoption in difficult cases can be implemented using wrappers, the JavaTest harness is perfectly suitable for all kind of tests and due to its compact distribution, can be provided with test suites or target products.

The purpose of testing is to find bugs. Once found, they must be tracked and fixed. Sometimes attempts to fix bugs fail, so tests must be rerun to check whether or not they are successful. It may also be the case that your fix does indeed solve the initial problem, but introduces a new problem. This is a form of regression. For both of these reasons it is advisable to make sure that your tests are easily repeatable and, if possible, automated. A CMM requirement:

If the software development plan specifies that low-level testing will be carried out, then standards and procedures for these steps must be defined.

The JavaTest harness provides a convenient mechanism for specifying low-level testing standards for both test quality and test quantity. To be *managed* according to CMM L4 requirements, test developers must produce reviewable test sources and input parameters and provide for the generation of convenient post-processable output. (This is exemplified by JavaTest *result* files) The reviewable pass or fail trigger mechanism of test results, in conjunction with the controlling checksums, removes doubt about whether or not a given unit has passed a test. In addition to this, informative and detailed output helps in debugging and reproducing test failures.

Defined (CMM Level 3)

CMM suggests the following:

1. The organization's standard process should be reviewed periodically and improved in the light of findings.
2. Successful methods, procedures and tools should be actively transferred though an organization.

The use of the JavaTest harness can help strengthen the development process. The use of successful tools usually spreads from project to project.

CMM suggests that:

The organization's standard process should be documented and that standards for its use should be created.

Here, software testing is specifically mentioned. In the next paragraph, CMM refers to well-known software life-cycle models. Use of the JavaTest harness creates efficient environment for successful procedures and standards in the area of QA testing.

Managed (CMM Level 4)

CMM states that

A project's process and plan must be derived from an organization's standards through a set of de-cemented procedures.

If a project needs to run a rich variety and a large number of tests, managers need to look at tools support. If project uses the Java Platform, Micro Edition (Java ME), then the JavaTest harness is the ideal tool.

The CMM publication "Software Product Engineering" states:

A thorough software engineering process must exist and include testable requirements and verifiable design.

CMM is also explicit on the need for unit testing of code and details

... various approaches to different levels of software testing, including the unit and integration levels.

White-box and black-box testing approaches can be used as testing standards. In CMM, the need for tools is mentioned and the need for training in test planning and testing technique is detailed. The JavaTest harness (with various test execution environments) is designed specifically for multiple types of testing and permits various approaches to testing including black-box and white-box.

The next item in CMM states:

Integration testing should not begin until all relevant code has passed unit tests.

This is a requirement where the JavaTest harness can help significantly with its clear definition of passed-failed status.

Optimized (CMM Level 5)

To fulfill the aims of the *optimized* level, tools must be in place to produce test sets that are flexible and easily tailored. Such tools make functional, regression, compatibility and unit tests easy to configure and run. The exclude lists of the JavaTest harness manages test execution and test results very effectively.

Examples

In this figure, regression tests are running in a JavaTest harness window.

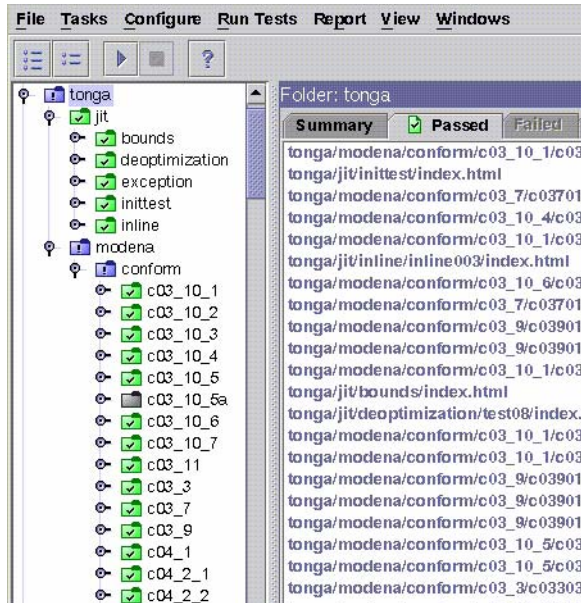


FIGURE 4 Regression Test Running in JavaTest Harness

Conclusion

The effectiveness of testing effort can be maximized by selection of a testing strategy that includes various types of tests, good management of the testing process, and the appropriate use of a harness such as the JavaTest harness to support the testing process. The results will be more reliable software at a lower development cost. Further benefits of simplified maintenance and reduced life cycle costs will also be realized. Effective testing is all part of developing an overall quality culture, which can only be beneficial to a software development business.

References

Capability Maturity Model, Carnegie Mellon Software Engineering Institute

R. Binder, "The FREE Approach to Testing Object-Oriented Software,"

<http://www.rbsc.com/pages/FREE.html>

"Performance Testing of Software Systems," AT&T Labs - Applied Tech.

Links

<http://www.aptest.com/resources.html#app-jsource>

<http://www.aptest.com/resources.html#app-func>

http://en.wikipedia.org/wiki/Software_testing

<http://www.extremeprogramming.org/what.html>

Acronyms

The following table defines acronyms commonly used in test methodology and JavaTest literature.

TABLE 3 Acronyms

Acronym	Definition
GUI	Graphical User Interface
Java ME	Java Platform, Micro Edition
Java SE	Java Platform, Standard Edition
Java EE	Java Platform, Enterprise Edition
MIDP	Mobile Information Device Profile
API	Application Programming Interface

TABLE 3 Acronyms

Acronym	Definition
PLC	Product Life Cycle
XP	Extreme Programming
CMM	Capability Maturity Model