

# Getting the Most from Your Automated Testing Tools

---

Laura Rose  
Quality Assurance Engineer  
Rational Software

TP325, 09/01

|  |          |
|--|----------|
| <b>Automated Testing: Filling the Quality Gap.....</b>   | <b>1</b> |
| <b>Test Early .....</b>  | <b>2</b> |
| <b>Understand Your Testing Environment.....</b>  | <b>2</b> |
| <b>Modularize and Reuse Test Cases.....</b>  | <b>3</b> |
| <b>Use a Configuration Management (CM) System.....</b>   | <b>3</b> |
| <b>Don't Reinvent the Wheel .....</b>  | <b>4</b> |
| <b>Share Unit Tests with Developers.....</b>   | <b>4</b> |
| <b>Create Playback Scripts and Leverage Comparator Features for<br/>Regression Testing .....</b> | <b>4</b> |
| <b>Combine Unit Test Scripts for Integration Testing.....</b>                                    | <b>5</b> |
| <b>Conduct Non-intrusive System Testing .....</b>  | <b>6</b> |
| <b>Stress Test for Multiple Users on Different Machines .....</b>                                | <b>6</b> |
| <b>Use Load Testing Tools for Performance Testing.....</b>                                       | <b>6</b> |
| <b>Let Developers Do the Sanity Testing .....</b>  | <b>6</b> |
| <b>Collect Field Diagnostics.....</b>  | <b>7</b> |
| <b>Use Test Scripts for Vendor Calibration and Performance Verification .....</b>                | <b>7</b> |
| <b>Use Test Scripts for Loading and Lab Setup.....</b>   | <b>7</b> |
| <b>Challenges and Limitations of Automated Testing .....</b>                                     | <b>7</b> |
| <b>The Redesign Challenge.....</b>   | <b>7</b> |
| <b>The Human Factor .....</b>  | <b>8</b> |
| <b>References.....</b>   | <b>9</b> |

## Introduction

In today's competitive software market, customers expect quality products, and they gauge quality according to the frequency and severity of post-implementation problems. What's the key to reducing these problems? The answer is deceptively simple: Test the software sufficiently and correct the sources of error before the product is shipped. As every tester and quality assurance person knows, however, the actual business of testing for, and then correcting, errors is not a simple matter. Here are just a few of the factors that make it complicated:

- Statistics show that even the most carefully constructed code averages one to three defects per hundred statements.<sup>1</sup>
- Studies indicate that inspections can uncover about 60 percent of total product defects; the remaining errors show up in application testing.<sup>2</sup>
- Application testing is a labor-intensive and expensive undertaking. Studies show that it consumes at least 50 percent of total product labor costs.<sup>3</sup>
- Manual test efforts tend to find the majority of defects at the end of the release effort or during beta testing, where the errors are more expensive to fix.
- =Few programmers like testing, and manual tests are often executed inconsistently.

## Automated Testing: Filling the Quality Gap

The past five years have seen a rapid proliferation of tools that address some of these issues by testing software automatically. Automatic testing tools can detect errors early in the development process. The earlier the detection, the easier and cheaper these errors are to fix, and the less impact the correction cycle will have on the customer. Figure 1 shows a benchmark comparison of manual vs. automated effort for various test steps.<sup>4</sup> The testing involved 1,750 test cases and 700 errors.

Hours For Manual VS. Automated Testing<sup>1</sup>

| Test Steps                         | Manual Testing | Automated Testing | Percent Improvement with Tools |
|------------------------------------|----------------|-------------------|--------------------------------|
| Test Plan Development              | 32             | 40                | -25%                           |
| Test Case Development              | 262            | 117               | 55%                            |
| Test Execution                     | 466            | 23                | 95%                            |
| Test Result Analyses               | 117            | 58                | 50%                            |
| Error Status/Correction Monitoring | 117            | 23                | 80%                            |
| Report Creation                    | 96             | 16                | 83%                            |
| Total Duration (Hours)             | 1090           | 277               | 75%                            |

<sup>1</sup>Taken directly from the November 1995 issues of QA Quest, The Newsletter of the Quality Assurance Institute. Note that percent of hours may be a more valuable benchmark for individual organizations than actual hours.

Figure 1: Hours for Manual vs. Automated Testing

As you can see, automation makes a significant difference in all areas of testing, especially executing tests and producing reports.

<sup>1</sup> Boris Beizer, Software Testing Techniques. Van Nostrand Reinhold, 1990.

<sup>2</sup> B.W. Boehm, Software Engineering Economics. Prentice-Hall, 1980.

<sup>3</sup> Boris Beizer, Op.Cit.

<sup>4</sup> From QA Quest, *The Newsletter of the Quality Assurance Institute*, November 1995. Note that percent of hours may be a more valuable benchmark for individual organizations than actual hours.

But how do you use automated testing to achieve a maximum positive impact on overall product quality? We'll discuss some approaches below.

## Test Early

In a good software development process, test planning and design parallels the steps needed to produce the working code. Product quality begins with documenting rules and restrictions that define what the software must do (functional requirements). Once you complete the first draft of your functional specification, you should evaluate it using four criteria:

- Completeness
- Consistency
- Feasibility
- Testability

Pay special attention to item #4. During the functional specification phase, you should determine your test automation strategies. If you keep test automation at the forefront of your thinking, then you can easily shape your product design and coding standards to provide the proper environment for getting the most out of your test tools.

**As a software development project evolves, it's important for testers to be aware of what is under development so that they can introduce automated tools at strategic junctures.** Early iterations and prototypes often undergo only ad hoc, superficial usability testing. But once a product design has been selected and the specification is in place, testing becomes more serious. The focus is on features. Ideally, each feature is tested with various inputs and constraints, and several iterations are required to fully cover the product. On most projects, however, it soon becomes obvious that the testing team cannot possibly examine all interactions between all components manually in a timely and cost effective way. This is where automation really shines. It is a viable solution to these complex testing challenges. In a mature testing environment, automation can help provide quality measurements, useful analyses, and optimization of staff testing time.

Too often, teams wait until the features are complete before hauling out their automated testing tools. At that point, they are racing delivery deadlines, so they restrict their activities to regression testing. This not only leads to serious post-implementation problems for customers; it also robs the test team of an opportunity to get their money's worth from those tools. The earlier you incorporate an automated test approach into your development cycle, the greater the return on your investment.

## Understand Your Testing Environment

Many factors affect testing during the software development lifecycle, but let's look at why it's important to have in-depth knowledge about — — and control over — four factors that can ultimately have a major influence on product quality.

**Know your staff.** Each staff member has different strengths, and it's worth taking some time up front to assess these before deciding how to assign various testing activities. In addition, because automation changes how work is done, you can expect complementary changes in people's jobs. Some may disappear; other, more technical, jobs may materialize. Still other jobs may change focus. Accept those changes and try to match up each person's skills with these changing roles. If necessary, provide new skills training for your staff to lay the groundwork for success.

**Know your customer.** If the goal of testing is to reduce the frequency with which your customer will encounter a defect, then your tests should ultimately reflect the customer's environment:

- Mirror the customer's configuration in the test lab.
- Match the test data to the customer's environment.
- Make sure the workload characteristics correspond to the customer's performance and load requirements.

**Know your product.** The more background you have on the product you are testing, its design emphasis, and its customer base, the closer your test cases will be to the *real world use*. In a good software development process, test planning and design should parallel the steps needed to produce the working code. The test plan should start during the generation of the requirements.

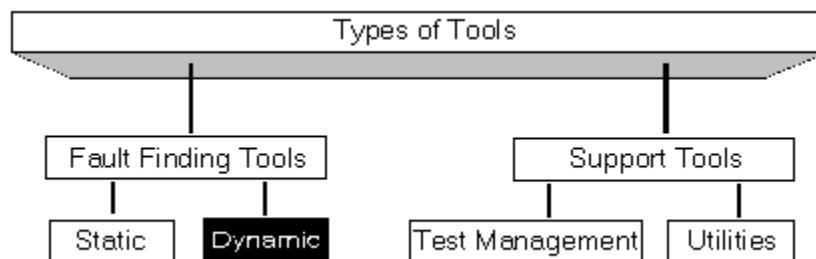
**Know your test tool.** Figure 2 illustrates the four basic categories of automated test tools:

**Static tools** analyze the application's source code.

**Dynamic tools** actually run the application to inspect real output: screens, reports, database records, and input/output (I/O) activity.

**Test management tools** assist in managing the test process.

**"Other utilities"** is a catchall category for software that does not perform or manage testing, but in some way aids the test process.



**Figure 2: Taxonomy of Automated Test Tools**

It's important to understand which category your tools fit into because there are several different test phases in a product cycle, and some tools are useful only for specific phases. Dynamic tools, the most versatile, may be used in all major test phases, as we will discuss below.

**Know your data.** It is very important to control the application data you are testing. To get meaningful test results, you must ensure that you are inputting consistent data to the test cases. If developers and testers are allowed to independently add or remove information from a test database, chaos may result. You won't know which version of the data you are testing, and it will not be clear why you are seeing variations in the results.

## Modularize and Reuse Test Cases

A significant cost of test automation is keeping up with the inevitable changes associated with normal software development. One method for minimizing the effect of changes to the product and test lab environment is to modularize the test cases. To do this effectively, you must plan and design your test cases ahead of time. Be sure your feature specifications are complete enough to create the test plan and test cases in parallel with feature coding.

Choosing the proper automation strategy for each different test phase can also help you cut costs because you can leverage your automated tests for reuse. There is no one method that fits all test types or test objectives. Although dynamic tools can be used in all test phases, their implementation will not be the same for each phase. If you carefully tailor your strategy to each phase, then it may be relatively easy to adapt your tests for reuse during the same phase on your next project.

## Use a Configuration Management (CM) System

Tests, like development code, change throughout the development and maintenance cycles. Therefore, it is wise to use a Configuration Management (CM) system for test code.

- CM dictates the rules for storing and managing the software, hardware, and tools used in the development cycle. These rules give you traceable control of changes during the different phases of development.
- CM also helps you coordinate the efforts of the development and test project teams. It allows the teams to change code and test scripts without creating confusion or hampering each other's work. Additionally, it provides control over baseline and intermediate releases of the software, test scripts, documentation, and development tools.

- CM also allows everyone to share test libraries and test data. Developers and testers can execute the same test cases simultaneously, in different configuration views or working directories, without overwriting each other's test results. This reduces test time in two important ways:
  - The engineers can run the same tests cases prior to their code submission, allowing them to catch the errors before the final software build.
  - Testing on different hardware platforms can be done in parallel instead of serially.

## Don't Reinvent the Wheel

Automated test tools can provide tremendous productivity gains for a software development organization. Unfortunately, many software developers waste time reinventing the testing wheel. On most projects, test tools are the last requirements to be considered, and then they must mesh with already-specified development tools. Plus, the budget and schedule are often stretched, leaving little of either for bringing testing tools online. As Richard Morin explains:

Although some commercial packages for bug tracking and regression testing are available, most companies write their own. This might have to do with local conditions (assumed or real), deficiencies in the commercial offerings, or cost factors. Regardless, the result is that many programmers currently are busy writing and maintaining private versions of these software systems. What's worse, these systems really aren't solving the problems at hand.<sup>5</sup>

If you select the proper family of automated test tools, however, you can design your software product, from the beginning, to take advantage of the tool's entire feature set.

## Share Unit Tests with Developers

Historically, to produce high-quality software, you had to test each function, module, or class (in object-oriented programming). This practice, called **unit testing** (or module testing), while effective, is extremely time consuming and labor-intensive, and is usually performed by the software developer.

Automated test tools offer a way to make this process easier. Capture/Playback is an essential component of rapid script development, and most Capture/Playback tools can automatically collect all user interactions with the application into an easily edited test script that is later used for playback. So if you use a Capture/Playback tool, a test browser, and a CM system, you are already on your way to creating reusable tests.

In most organizations, development engineers both outnumber test engineers, and know more about problematic areas. Often, testers don't take full advantage of developers' testing knowledge. After all, these people have already performed much of the testing, even though their tests are typically designed to be temporary and disposable. Testers and developers can coordinate their efforts if developers design both program and unit tests for modularity, and then incorporate these tests into a regression suite or library. They can also write basic utility scripts and configure them into a library. Then, developers can browse through these components and use them to create unit tests. The code for these tests can also be configured into the same library. The unit tests can be combined or reworked into integration tests to be shared throughout the development cycle.

With some test automation tools, you can modify scripts to include external routines, and the software department's shared utilities (error routines, debug routines, message routing, etc.) can be configured directly into the test. These tools are extremely flexible, and the tests they create can often be used in more than one test phase.

## Create Playback Scripts and Leverage Comparator Features for Regression Testing

As stated in the IEEE Standards,

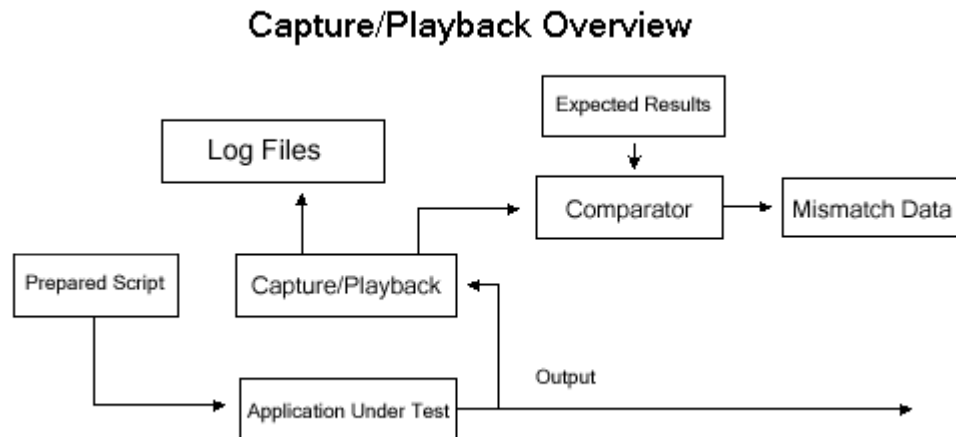
**Regression Testing** is the selective retesting [of software] to detect faults introduced during modifications of a system or system component, to verify that modifications have not caused unintended adverse effects, or to verify that a modified system or system component still meets specified requirements. (IEEE 1990).

---

<sup>5</sup> From Richard Morin, "Distributed Quality Assurance." *UNIX Review*, Sept. 1993 v11 n9 p170.

In other words, regression testing answers the question, "Does everything still work after my fix?" Tools can automate the regression testing phase of the development cycle and allow you to utilize your testing resources efficiently (and avoid the monotony and error-prone results that characterize manual quality testing). Tools enable you to deliver high quality software with tremendous time and cost savings.

Most automated test tools capture the right answer from a previous version of the software and store it. During regression testing, the test scripts are rerun, and their actual results are compared with the previous version's expected results (see Figure 3).



**Figure 3: Capture/Playback Overview**

To perform reliable regression testing, the environment of the system under test must remain the same between the time of the original recording and the playback. One way to achieve this is to create a script that records the state of the original test system. Then, prior to regression testing, play back the script to bring the system to the original state.

Since regression testing implies the use of a comparator (a program that compares the actual with the expected results), the more flexible the comparator, the less rework is required. Some features of screen comparators to look for include:

- Ignore a field or region.
- Include only a field or region.
- Ignore video attributes.
- Ignore color shading or change in color palette.

By programming to automatically ignore information that changes from one release to the next, you are reducing maintenance for the *expected results*. For example, the release or version number changes with each release, as do the date and time stamps on screens and files. If this information is captured in the expected results file and compared with the next release, then it will fail. If you take advantage of comparator features to mask those fields, however, then the test will not fail.

## Combine Unit Test Scripts for Integration Testing

Testing a specific feature together with other newly developed features is known as **integration testing**. Testing the interface of two components is a way to explore how components interact with each other. Integration testing inspects not only the variables passed between two components, but also the global variables. This test phase assumes that the components and the objects they manipulate have all passed their local unit tests.

Previously captured unit test scripts can be combined, with minimum effort, to create a variety of integration test cases. For instance, a unit test script that tested an ADD function can be scheduled with other unit tests for DELETE and COPY to create an integration test of the entire file maintenance system with little rework.

## Conduct Non-intrusive System Testing

**System testing** is designed to reveal bugs that cannot be attributed to either individual components or the interaction among components and other objects. System testing studies all implementation aspects of the design that are similar to those in the customer's environment. The issues and problem behaviors it targets can only be exposed by

testing either the entire integrated system or a major part of it. System testing includes testing for performance, stress, security, accountability, configuration sensitivity, usability, data integrity, start-up, and recovery.

Verifying these characteristics is especially important for products intended for users on diverse OS and hardware platforms. You can test a variety of platforms using the same test scripts and capture/playback record files, then compare the results. That way, you can readily associate the causes of variations with either a specific hardware platform or a generic problem.

Since **system testing** is aimed at testing the product as the customer will receive it, the test tool you use at this stage of the product development cycle should be non-intrusive. If the test tool requires the software application under test to be built or linked with specific runtime libraries or include files, then you are not testing the end product. A true non-intrusive approach requires no modifications to the test environment or application, and no special hooks into your application or libraries.

In today's marketplace, you can choose between intrusive and non-intrusive technologies for virtually every phase of testing. Rational recommends non-intrusive technologies for all test phases in the development cycle; they help you guarantee that the application you test in the lab is the same application you will deliver to your customer.

## Stress Test for Multiple Users on Different Machines

**Stress testing** determines whether a program can fulfill its defined requirements and work as it should under extreme conditions. Automated test tools can easily simulate multiple users and emulate several machines and applications. This is critical for both Client/Server and UNIX applications; the application itself might hold up fine with a 500-user load, but some hardware platforms might have lower load limits.

Avoid hard coding the number of emulated users within the script; instead, pass the size through environment variables, parameters, or test drivers. This allows the same set of test scripts to be used on other machines with different load limits.

## Use Load Testing Tools for Performance Testing

A software performance problem can cost millions of dollars in downtime, so you need **performance testing** to verify that an application meets specific performance efficiency objectives. Automated load testing tools can help you evaluate performance and response times by delivering a *workload of user activity* to your application. They then measure the application quality and response time that will be achieved in the actual environment. These automated test tools emulate *real* scenarios in order to truly test system performance and software functionality.

It's best to use a flexible test case scheduling mechanism so that you can easily specify and modify the order of the tasks, job mix, and rate at which jobs are submitted. Without this, it is difficult to reproduce the real workload, since the rate of test case submission is a critical component of the workload. This scheduling mechanism should be independent of the test cases, so that the workload dynamics can be modified without changing the actual test cases.

To avoid load dependent results, use the tool's text matching capabilities instead of matching on timing characteristics. For instance, match on the text response "DONE" instead of a response that returns in two seconds. Matching on timing characteristics means that the benchmark execution will be load dependent, and therefore not repeatable (since the load changes as you increase the number of users in a network environment or vary the configuration parameters). Investigate the use of synchronization mechanisms such as "wait on events." These mechanisms can be used to provide mutually exclusive access to critical data regions, as well as user cooperation on concurrent tasks.

## Let Developers Do the Sanity Testing

**Sanity testing** is used to verify that the software build is ready for System Test (or more extensive testing). The tests are very general, targeting the most frequently used functional areas. If an application does not pass the Sanity Test suite, then further testing activities are suspended.

If testers automate and share these tests with the development staff, then development engineers can quickly execute the tests prior to a software build to ensure that they are not handing off an unsatisfactory release. This eliminates several time-consuming cycles, including:

- Building the product.
- Handing the product to QA.
- Doing formal QA testing to discover and isolate the major error.
- Sending the product back from QA to engineering.

## Collect Field Diagnostics

Capture/Playback techniques are invaluable to customer support and field technicians. This technology collects all user interactions with the application (cursor movements, button activity, keystrokes, think time delays, and so on) into an editable test script that can be transmitted back to the development organization. Support staff can play back the scripts (with minimal modifications) in their test lab, emulating the customer's exact steps. This eliminates speculation about what actually transpired. The same customer scripts can be incorporated into the automated regression test suites, making yesterday's problems today's verification tools.

## Use Test Scripts for Vendor Calibration and Performance Verification

Another service automated test scripts can provide is vendor calibration and performance verification. When vendors receive software or hardware, they can run the same *performance* test scripts. These reports can be used as proof that the vendor-supplied items are working at agreed-upon levels.

You can use the same scripts to offer yearly performance maintenance runs on your product. Supply these reports to the customer as proof that the software is working at the same level it was when they purchased it.

## Use Test Scripts for Loading and Lab Setup

Automation can assist in several manufacturing procedures, including installation, initialization, and verification. Different scripts can be called to load environments, based upon customer configurations or software versions. These scripts can then be used to perform the final check before the equipment and software are shipped to the customer.

## Challenges and Limitations of Automated Testing

Test automation cannot bring order to chaos. Even the most advanced tools will work only as well as your process. If your organization has no active quality policy, no configuration management, no documented requirements, then even the best tools will disappoint. On the other hand, if your process is organized and continually working to improve quality, then automated tools can be extremely successful in reducing test cycle times while increasing test coverage.

Most applications include areas that are especially difficult to test: interrupt handlers, exception handlers, critical sections relating to multitasking and relatively rare conditions (such as February 29th). To thoroughly test these chunks of code, it may be necessary to set up pre-test conditions or use debug techniques to set values, and this can be challenging.

**Naive user** and **usability** tests are also hard to automate. Automated tests run the same way each time (or travel the same paths), but different customers can use the product differently. It is impossible to cover all combinations. Normally, automated test scripts follow known paths. Once a new user discovers an unexpected path, however, you can update the automated tests to include it, thereby improving your regression suite coverage.

## The Redesign Challenge

Another limitation of automated testing is that when a new version of the software is released, the test scripts no longer work without redesign. Because test redesign was never built into the release schedule, often entire automation systems (representing thousands of hours of work) collapse, and the projects are completed with inadequate testing. A similar problem can occur with new platforms or operating systems.

When this happens, you may need to re-evaluate several processes:

- **Test Planning and Test Design.** At the start of a project, all potential platforms and features should be incorporated into the test design. Also acknowledge that software features and platforms will change and require test script modifications, and incorporate test maintenance into the development schedule. The task of designing and coding reusable automated tests for the full product cycle is complex, and must go beyond a capture/playback mentality.
- Also, remember that automated test tools are not the entire solution. Tests can be executed by an unattended machine, but planning and development must be done by a qualified engineer. Good software test development is sophisticated programming and requires testers who are comfortable in this environment.
- **Scheduling.** The use of automated test scripts requires that even small changes to the software product, anywhere in the development and maintenance phases, be carefully considered. Even the smallest change can impact test scripts or require additional test design and implementation to cover the new features or enhancements. This test case maintenance is an ongoing effort.
- **Test Tool Selection.** Frequently, development teams select test tools either after the product design phase or without even considering the product that is being designed. Those tools might not be right for your product.

## **The Human Factor**

Knowing where each member of your team stands with respect to test automation knowledge and experience is vital to the success of a new implementation. You can easily destroy an employee's morale if you assign him automated testing tasks that don't match his knowledge and abilities.

Also be aware that automation changes an organization. Interactions between testers and the rest of the software organization change. With automation, testers can do more for developers, release engineers, and customer service representatives. Sophisticated testing changes "educated guessing" into engineering. More people can run more tests earlier in the software development cycle. Testers can concentrate on more complex testing and analysis. But be prepared: Expectations change, too. Managers will expect higher quality code and faster deliveries. If you select the proper test tool at the start and implement it use early in the development cycle, however, you can meet those expectations. You will see tremendous productivity gains and greatly increase the return on your investment.

## References

Michael A. Foody, "When Is Software Ready for Release?" *UNIX Review*, March 1995.

IEEE Standards Board, "IEEE Standard for Software Verification and Validation Plans," 1982.

Nancy Jenkins and David M Chadwick, "Workload Engineering: Workload Models for Performance Measurement." Performance Awareness Corporation, 1990.

Jay Johnson, Rod Skoglund, and Joe Wisniewski, *Program Smarter, Not Harder: Get Mission-Critical Projects Right the First Time*. McGraw Hill, Inc., 1995.

Kathleen Lew, "Installing Quality into Software Development.", 1994

Richard Morin, "Distributed Quality Assurance." *UNIX Review*, Sept. 1993, v11 n9, p.170.

Dr. Stephen Norman, *Software Testing Tools*. Ovum Ltd., 1993.

William E. Perry, *A Standard for Testing Application Software*. Auerbach Publishers, 1992.

Gerald M. Weinberg, *Quality Software Management, Volume I: Systems Thinking*. Dorset House, 1992.

**Rational Software**  
Dual Headquarters:  
Rational Software  
18880 Homestead Road  
Cupertino, CA 95014  
Tel: (408) 863-9900

Rational Software  
20 Maguire Road  
Lexington, MA 02421  
Tel: (781) 676-2400

Toll-free: (800) 728-1212  
E-mail: [info@rational.com](mailto:info@rational.com)  
Web: [www.rational.com](http://www.rational.com)  
International Locations: [www.rational.com/worldwide](http://www.rational.com/worldwide)

Rational and the Rational logo, among others, are trademarks or registered trademarks of Rational Software Corporation in the United States and/or other countries. References to other companies and their products use trademarks owned by the respective companies and are for reference purposes only.

© Copyright 2001 by Rational Software Corporation.  
Subject to change without notice. All rights reserved