Rational® Test RealTime

Online Tutorial

VERSION: 2003.06.00

WINDOWS AND UNIX

support@rational.com
http://www.rational.com

Legal Notices

©2001-2003, Rational Software Corporation. All rights reserved.

Any reproduction or distribution of this work is expressly prohibited without the prior written consent of Rational.

Version Number: 2003.06.00

Rational, Rational Software Corporation, the Rational logo, Rational Developer Network, AnalystStudio, , ClearCase, ClearCase Attache, ClearCase MultiSite, ClearDDTS, ClearGuide, ClearQuest, ClearTrack, Connexis, e-Development Accelerators, DDTS, Object Testing, Object-Oriented Recording, ObjecTime, ObjecTime Design Logo, Objectory, PerformanceStudio, PureCoverage, PureDDTS, PureLink, Purify, Quantify, Rational Apex, Rational CRC, Rational Process Workbench, Rational Rose, Rational Suite, Rational Suite ContentStudio, , Rational Summit, Rational Visual Test, Rational Unified Process, RUP, RequisitePro, ScriptAssure, SiteCheck,SiteLoad, SoDA, TestFactory, TestFoundation, TestStudio, TestMate, VADS, and XDE, among others, are trademarks or registered trademarks of Rational Software Corporation in the United States and/or in other countries. All other names are used for identification purposes only, and are trademarks or registered trademarks of their respective companies.

Portions covered by U.S. Patent Nos. 5,193,180 and 5,335,344 and 5,535,329 and 5,574,898 and 5,649,200 and 5,675,802 and 5,754,760 and 5,835,701 and 6,049,666 and 6,126,329 and 6,167,534 and 6,206,584. Additional U.S. Patents and International Patents pending.

U.S. GOVERNMENT RIGHTS. All Rational software products provided to the U.S. Government are provided and licensed as commercial software, subject to the applicable license agreement. All such products provided to the U.S. Government pursuant to solicitations issued prior to December 1, 1995 are provided with "Restricted Rights" as provided for in FAR, 48 CFR 52.227-14 (JUNE 1987) or DFARS, 48 CFR 252.227-7013 (OCT 1988), as applicable.

WARRANTY DISCLAIMER. This document and its associated software may be used as stated in the underlying license agreement. Except as explicitly stated otherwise in such license agreement, and except to the extent prohibited or limited by law from jurisdiction to jurisdiction, Rational Software Corporation expressly disclaims all other warranties, express or implied, with respect to the media and software product and its documentation, including without limitation, the warranties of merchantability, non-infringement, title or fitness for a particular purpose or arising from a course of dealing, usage or trade practice, and any warranty against interference with Licensee's quiet enjoyment of the product.

Third Party Notices, Code, Licenses, and Acknowledgements

Portions Copyright ©1992-1999, Summit Software Company. All rights reserved.

Microsoft, the Microsoft logo, Active Accessibility, Active Client, Active Desktop, Active Directory, ActiveMovie, Active Platform, ActiveStore, ActiveSync, ActiveX, Ask Maxwell, Authenticode, AutoSum, BackOffice, the BackOffice logo, bCentral, BizTalk, Bookshelf, ClearType, CodeView, DataTips, Developer Studio, Direct3D, DirectAnimation, DirectDraw, DirectInput, DirectX, DirectXJ, DoubleSpace, DriveSpace, FrontPage, Funstone, Genuine Microsoft Products logo, IntelliEye, the IntelliEye logo, IntelliMirror, IntelliSense, J/Direct, JScript, LineShare, Liquid Motion, Mapbase, MapManager, MapPoint, MapVision, Microsoft Agent logo, the Microsoft eMbedded Visual Tools logo, the Microsoft Internet Explorer logo, the Microsoft Office Compatible logo, Microsoft Press, the Microsoft Press logo, Microsoft QuickBasic, MS-DOS, MSDN, NetMeeting, NetShow, the Office logo, Outlook, PhotoDraw, PivotChart, PivotTable, PowerPoint, QuickAssembler, QuickShelf, RelayOne, Rushmore, SharePoint, SourceSafe, TipWizard, V-Chat, VideoFlash, Virtual Basic, the Virtual Basic logo, Visual C++, Visual C#, Visual FoxPro, Visual InterDev, Visual J++, Visual SourceSafe, Visual Studio, the Visual Studio logo, Vizact, WebBot, WebPIP, Win32, Win32s, Win64, Windows, the Windows CE logo, the Windows logo, Windows NT, the Windows Start logo, and XENIX, are either trademarks or registered trademarks of Microsoft Corporation in the United States and/or in other countries.

Sun, Sun Microsystems, the Sun Logo, Ultra, AnswerBook 2, medialib, OpenBoot, Solaris, Java, Java 3D, ShowMe TV, SunForum, SunVTS, SunFDDI, StarOffice, and SunPCi, among others, are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

Purify is licensed under Sun Microsystems, Inc., U.S. Patent No. 5,404,499.

Licensee shall not incorporate any GLOBEtrotter software (FLEXlm libraries and utilities) into any product or application the primary purpose of which is software license management.

BasicScript is a registered trademark of Summit Software, Inc.

Design Patterns: Elements of Reusable Object-Oriented Software, by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Copyright © 1995 by Addison-Wesley Publishing Company, Inc. All rights reserved.

Additional legal notices are described in the legal_information.html file that is included in your Rational software installation.

Tutorial Contents

Preface vii

Audience vii
Contacting Rational Technical Publications vii
Other Resources viii
Customer Support viii

Overview 1

Additional Information 2
C and C++ Track 2
About this Tutorial 2
Example File Locations 3
Mobile Phone Simulator 3
UMTS Base Station 3
Host-based Testing vs Target-based Testing 4
Goals of the Tutorial 5
Java Track 6
About this Tutorial 6
Example File Locations 7
Mobile Phone Simulator 7
JDK Installation 7
Host-based Testing vs Target-based Testing 9
Goals of the Tutorial 10

Runtime Analysis 13

C and C++ Track 13
Runtime Analysis for C, C++ and Ada 13
Runtime Analysis 13
Runtime Analysis Exercises 15
Conclusion 36
Java Track 37
Runtime Analysis for Java 37
Runtime Analysis 38
Runtime Analysis Exercises 40

Conclusion 62

Component Testing **64**

C, C++ and Ada Track 64

Automated Component Testing 64
Component Testing for C and Ada 64
Component Testing for C++ 79
System Testing for C 89

Java Track 103

Automated Component Testing 103
Component Testing for Java with Rational Test RealTime 103
Component Testing for Java Exercises 104
Conclusion 113

Conclusion **115**

Regression Testing 115
Proactive Debugging 115
Questions? 117

Preface

Welcome to Rational Test RealTime.

This tutorial is designed to introduce software developers to the power and simplicity of Rational Test RealTime. Of course, the first goal of these lessons is to teach you how to use these tools in your current development project. However, there is a second goal as well. Test RealTime is meant to enhance your development effort, not get in its way. To that end, you will be offered insights into process as well. Hopefully, you will come away from this tutorial with an understanding of how best to make Test RealTime a fully integrated member of your development desktop.

Test RealTime is a complete runtime analysis and testing solution for real-time and embedded systems. It addresses all runtime analysis needs and all test levels including component and system testing for the C, C++, Ada, and Java programming languages.

General information about Test RealTime can be found in the *Test RealTime User Guide*.

Advanced usage of the product is described in the *Test RealTime Reference Manual*.

Audience

This guide is intended for Rational software users who are using Test RealTime for the first time, such as application developers, quality assurance managers, and quality assurance testers.

You should be familiar with the selected Windows or Linux platform as well as your Ada, C, C++ or Java development environment.

Contacting Rational Technical Publications

To send feedback about documentation for Rational products, please send e-mail to our technical publications department at techpubs@rational.com.

Keep in mind that this e-mail address is only for documentation feedback. For technical questions, please contact Customer Support.

Other Resources

All manuals are available online, either in HTML or PDF format. The online manuals are on the CD and are installed with the product.

For the most recent updates to the product, including documentation, please visit the Product Support section of the Web site at:

http://www.rational.com/products/testrt/index.jsp

Documentation updates and printable PDF versions of Rational documentation can also be downloaded from:

http://www.rational.com/support/documentation/index.jsp

For more information about Rational Software technical publications, see:

http://www.rational.com/documentation.

For more information on training opportunities, see the Rational University Web site:

http://www.rational.com/university.

Customer Support

Before contacting Rational Customer Support, make sure you have a look at the tips, advice and answers to frequently asked questions in Rational's Solution database:

http://solutions.rational.com/solutions

Choose the product from the list and enter a keyword that most represents your problem. For example, to obtain all the documents that talk about stubs taking parameters of type “char”, enter "stub char". This database is updated with more than 20 documents each month.

When contacting Rational Customer Support, please be prepared to supply the following information:

- About you:
  Name, title, e-mail address, telephone number
- About your company:
  Company name and company address

- **About the product:**
  Product name and version number (from the **Help** menu, select **About**).
  What components of the product you are using
- **About your development environment:**
  Operating system and version number (for example, Linux RedHat 8.0), target compiler, operating system and microprocessor. If necessary, send the Target Deployment Port **.xdp** file
- **About your problem:**
  Your service request number (if you are calling about a previously reported problem)
  A summary description of the problem, related errors, and how it was made to occur
  Please state how critical your problem is
  Any files that can be helpful for the technical support to reproduce the problem (project, workspace, test scripts, source files). Formats accepted are **.zip** and compressed tar (**.tar.Z** or **.tar.gz**)

If your organization has a designated, on-site support person, please try to contact that person before contacting Rational Customer Support.

You can obtain technical assistance by sending e-mail to just one of the e-mail addresses cited below. E-mail is acknowledged immediately and is usually answered within one working day of its arrival at Rational. When sending an e-mail, place the product name in the subject line, and include a description of your problem in the body of your message.

> **Note** When sending e-mail concerning a previously-reported problem, please include in the subject field: "**[SR#***<number>***]**", where *<number>* is the service request number of the issue. For example:

```
Re:[SR#12176528] New data on Test RealTime install issue
```

Sometimes Rational technical support engineers will ask you to fax information to help them diagnose problems. You can also report a technical problem by fax if you prefer. Please mark faxes "**Attention: Customer Support**" and add your fax number to the information requested above.

Location	Contact
North America	Rational Software, 18880 Homestead Road, Cupertino, CA 95014 voice: (800) 433-5444 fax: (408) 863-4001 e-mail: support@rational.com

Europe, Middle East, and Africa	Rational Software, Beechavenue 30, 1119 PV Schiphol-Rijk, The Netherlands voice: +31 20 454 6200 fax: +31 20 454 6201 e-mail: support@europe.rational.com
Asia Pacific	Rational Software Corporation Pty Ltd, Level 13, Tower A, Zenith Centre, 821 Pacific Highway, Chatswood NSW 2067, Australia voice: +61 2-9419-0111 fax: +61 2-9419-0123 e-mail: support@apac.rational.com

Overview

This tutorial is comprised of three primary lessons, or browse sequences. Those interested in C, C++ and Ada are asked to follow the track labeled with those languages; those interested in Java are asked to follow the Java track. You are welcome to complete both tracks, of course, but each has been designed to be finished in its entirety - that is, perform the entire Java tutorial track before initiating the track for C, C++ and Ada, and vice versa. Keep in mind that though there are some feature differences between the support for C, C++, Ada and Java, the majority of product features are the same.

> **Note** The Evaluation Version of Test RealTime only supports the Windows platform and the Microsoft Visual C++ Target Deployment Port. If you are evaluating the product, please disregard the Java track of this tutorial.

> **Note** For those interested specifically in Ada - The C, C++ and Ada track uses a pure C/C++ example. Ada support consists of component testing and code coverage analysis; a discussion of C language support for these two features should be considered equivalent to a discussion of Ada support. In addition, some of the Example projects shipped specifically with Test RealTime contain Ada code, giving you the opportunity to hone your skills for component testing.

Follow the lessons in order; this may take you 4 to 5 hours, depending on your prior knowledge of the Test RealTime feature-set and on your comfort level with software development. The three primary lessons are:

- Preparation for the Tutorial
- Runtime Analysis with Rational Test RealTime
- Automated Component Testing with Rational Test RealTime

To maneuver through the browse sequences:

- **On Windows:** Click the browse sequence pages at the top of the tutorial window.
- Other platforms: Use the Next Page and Previous Page links on each page.

Additional Information

While it is the objective of this tutorial to prepare you for the use of Rational Test RealTime, occasions will arise when you have questions beyond its scope. Be sure to take advantage of the online Help, which is designed to address all issues associated with the testing and runtime analysis of embedded software using Rational Test RealTime.

To access the Help, click the **Help** menu, then select **User Guide**. In the **Help** viewer, use the **Contents**, **Index**, and **Search** tabs to navigate to the information you need.

For information related to command-line usage and test script programming, click the **Help** menu, then select **Reference Manual**.

For information related to the Target Deployment Technology and its associated TDP Editor (both of which are discussed later), run the TDP Editor and select the menu item **Help->Target Deployment Guide**. To access the TDP Editor - within Test RealTime , select **Tools->Target Deployment Port Editor->Start**.

C and C++ Track

This tutorial can be performed on all Test RealTime supported development platforms - Windows, Solaris, Linux, HP-UX and AIX.

> **Note** The Evaluation Version of Test RealTime only supports the Windows platform and the Microsoft Visual C++ Target Deployment Port. If you are evaluating the product, please disregard any references to other platforms or development environments.

Since efforts are always being made to update or improve this tutorial - as well as the products themselves - a customer-only webpage has been created. This page contains news, patches and documentation updates for current users. Feel free to check this page for updates before usage of this tutorial.

To access the Test RealTime Support Web site:

1. From the **Help** menu, select **Rational Test RealTime on the Web** and **Latest News and Updates for Users**.

About this Tutorial

This tutorial demonstrates how to make the most of Test RealTime through a sample UMTS mobile phone application, comprised of:

- A mobile phone simulator, running a basic embedded application
- A UMTS base station demonstrating the communication system

UMTS - Universal Mobile Telecommunications System - is a Third Generation (3G) mobile technology that will enable 2Mbit/s streaming not only of voice and data, but also of audio and visual content. A UMTS base station is a switching network device enabling the communication of multiple UMTS-enabled mobile phones.

Example File Locations

Source files for the base station (the mobile phone executable is provided) are located within the product installation folder, in the folder **\examples\BaseStation_C\src**.

If you do not have write permission to the installation location of Test RealTime, you must copy the **examples** folder and its contents to a new location. Otherwise, you will be unable to perform any part of the Tutorial that creates or modifies files.

Mobile Phone Simulator

The mobile phone simulator consists of both a Graphical User Interface (GUI) as well as of internal logic. The GUI is constructed from OS-independent graphical C++ classes; the logic within the simulator is constructed from OS-independent C and C++ code.

The mobile phone executable is located within the Test RealTime/PurifyPlus RealTime installation folder, in the folder **\examples\BaseStation_C\MobilePhone**. The name of the executable depends on your operating system:

- Windows: MobilePhone.exe
- Solaris: MobilePhone.SunOS
- Linux SuSE: MobilePhone.Linux
- Linux RedHat: MobilePhone.Linux_redhat
- HP-UX: MobilePhone.HP-UX
- AIX: MobilePhone.AIX

A launcher shell script - **MobilePhone.sh** - is provided for the non-Windows platforms as well.

UMTS Base Station

The UMTS base station is fully operational, constructed from OS-independent C++ code. You are provided with both the source code and an executable for the base station. The UMTS base station executable is located within the Test RealTime installation folder, in the folder **\examples\BaseStation_C**. The name of the executable depends on your operating system:

- Windows: BaseStation.exe

- Solaris: BaseStation.SunOS
- Linux SuSE: BaseStation.Linux
- Linux RedHat: BaseStation.Linux_redhat
- HP-UX: BaseStation.HP-UX
- AIX: BaseStation.AIX

A launcher shell script - **BaseStation.sh** - is provided for the non-Windows platforms as well.

Host-based Testing vs Target-based Testing

The testing and runtime analysis that you will perform for this tutorial take place entirely on your machine. However, one of the greatest capabilities of Rational Test RealTime is its support for testing and analyzing your software directly on an embedded target. Does this mean you will need to change how you interact with your application when switching from host-based to target-based testing? Will your tests have to be rewritten, for example?

Not at all.

Thanks to Rational's versatile, low-overhead **Target Deployment Technology**, all tests are fully target independent. Each cross-development environment - that is, every combination of compiler, linker, and debugger - has its own Target Deployment Port (TDP). In addition, any TDP can be modified via the Test RealTime user interface at a more granular level, letting you customize a particular test or runtime analysis interaction without affecting neighboring interactions. Such granular tailoring is supported by the concept of *Configurations*. Each Configuration can support one or more TDP and can apply separate customization settings to each interaction assigned to it.

Over thirty reference TDPs, supporting some of the most commonly used cross-development environments, are supplied out-of-the-box. After creation of a project (you will be doing this in a few moments), you can access a list of TDPs installed on the machine.

To view a list of currently installed TDPs:

1. From the **Project** menu, select **Configuration**.
2. Select **New...**
3. Use the dropdown list to scroll through the available TDPs

Target Deployment Port Web Site

As new reference TDPs become available, they are first posted on a customer-accessible Web site. Check this site periodically for news of the latest TDPs to be made available to the Rational Test RealTime and PurifyPlus RealTime community.

To access the Test RealTime Web site:

1. From the **Help** menu, select **Rational Test RealTime on the Web** and **Target Deployment Ports**

Creating and Editing Target Deployment Ports

Does your organization target an environment for which no TDP yet exists? Using the **Target Deployment Port Editor** you can create support, just as many of Rational's customers have done before you.

The reference TDPs supplied with Test RealTime can guide your TDP creation efforts; Rational also provides professional services should you choose to contract out their creation.

> **Note** The Target Deployment Port Editor is not included with the evaluation version of the product.

To access the Target Deployment Port Editor:

1. From the **Tools** menu, select **Target Deployment Port Editor** and **Start**.

For more information about the Target Deployment Port Editor, please refer to the **Rational Test RealTime Target Deployment Guide**.

Every Test RealTime feature is accessible regardless of the environment within which you will be executing your tests. Rest assured, your intended targets are supported.

Goals of the Tutorial

The UMTS base station has been pre-loaded with errors; your responsibility, during the tutorial, will be to uncover:

- a memory leak
- a performance bottleneck
- a logic error in C code
- a logic error in C++ code

In addition, test completeness will be achieved by:

- improving the code coverage of your tests
- improving your understanding of the code via runtime tracing

Finally, you will

- simulate virtual actors in order to validate base station network messaging

To accomplish the above, you will first manipulate the UMTS base station through manual interaction with a mobile phone simulator. Afterwards, automated hands-free interaction will be used.

Regardless of the programming language you intend to use on your development project, make sure to perform the runtime analysis tutorial.

For component testing and system testing, however, only certain sections of the Tutorial may apply:

- for C users - Component Testing for C and Ada, System Testing for C
- for Ada users - Component Testing for C and Ada
- for C++ users - Component Testing for C++

To continue this tutorial, follow the C, C++ and Ada track in the next lesson: Runtime Analysis with Test RealTime.

Java Track

This tutorial can be performed on all Test RealTime supported development platforms - Windows, Solaris, Linux, HP-UX and AIX.

> **Note** The Evaluation Version of Test RealTime only supports the Windows platform with a JDK 1.3.1 or 1.4. If you are evaluating the product, please disregard the Java track of this tutorial.

Since efforts are always being made to update or improve this tutorial - as well as the products themselves - a customer-only webpage has been created. This page contains news, patches and documentation updates for current users. Feel free to check this page for updates before usage of this tutorial.

To access the Test RealTime Support Web site:

1. From the **Help** menu, select **Rational Test RealTime on the Web** and **Latest News and Updates for Users**.

About this Tutorial

This tutorial demonstrates how to make the most of Test RealTime through a sample UMTS mobile phone application, comprised of:

- A mobile phone simulator, running a basic embedded application
- A UMTS base station demonstrating the communication system

UMTS - Universal Mobile Telecommunications System - is a Third Generation (3G) mobile technology that will enable 2Mbit/s streaming not only of voice and data, but also of audio and visual content. A UMTS base station is a switching network device enabling the communication of multiple UMTS-enabled mobile phones.

Example File Locations

Source files for the base station (the mobile phone executable is provided) are located within the product installation folder, in the folder **\examples\BaseStation_Java\src**.

If you do not have write permission to the installation location of Test RealTime, you must copy the examples folder and its contents to a new location. Otherwise, you will be unable to perform any part of the Tutorial that creates or modifies files.

Mobile Phone Simulator

The mobile phone simulator consists of both a Graphical User Interface (GUI) as well as of internal logic. The GUI is constructed from OS-independent graphical C++ classes; the logic within the simulator is constructed from OS-independent Java code.

> **Note** Test RealTime supports both J2ME and J2SE; however, only J2SE is covered in this Tutorial.

The mobile phone executable is located within the Test RealTime/PurifyPlus RealTime installation folder, in the folder **\examples\BaseStation_C\MobilePhone** - that is, the executable is not located in the BaseStation_Java folder. The name of the executable depends on your operating system:

- Windows: MobilePhone.exe
- Solaris: MobilePhone.SunOS
- Linux SuSE: MobilePhone.Linux
- Linux RedHat: MobilePhone.Linux_redhat
- HP-UX: MobilePhone.HP-UX
- AIX: MobilePhone.AIX

A launcher shell script - **MobilePhone.sh** - is provided for the non-Windows platforms as well.

JDK Installation

Performance of the tutorial assumes access to the J2SE 1.3.1 or 1.4.0 SDK.

> **Note** If you are using the evaluation version of Test RealTime, only the Windows option is available. You must have the full versions of Test RealTime in order to perform the tutorial on Solaris, Linux, HP-UX and AIX.

If neither J2SE distribution is currently installed on your machine, you can freely download them as described here. The following are the recommended J2SE distributions. Technically, any SDK that is 100% J2SE 1.3.1 or 1.4.0 compliant can be used with Test RealTime. However, only the following distributions have been verified as supported.

To install J2SE 1.3.1 on Windows:

1. Go to http://java.sun.com/j2se/1.3/download.html
2. Select the SDK download link for "Windows (all languages)"
3. Download and install the SDK onto your machine

To install J2SE 1.4.0 on Windows

1. Go to http://java.sun.com/j2se/1.4/download.html
2. Select the SDK download link for "Windows (all languages, including English)"
3. Download and install the SDK onto your machine

To installJ2SE 1.3.1 on Solaris:

1. Go to http://java.sun.com/j2se/1.3/download.html
2. Select either SDK download link for the Solaris SPARC
3. Download and install the SDK onto your machine

To install J2SE 1.4.0 on Solaris

1. Go to http://java.sun.com/j2se/1.4/download.html
2. Select the SDK download link for the 32-bit or 64-bit Solaris SPARC distribution
3. Download and install the SDK onto your machine

To install J2SE 1.3.1 on Linux (both RedHat and SuSE):

1. Go to http://java.sun.com/j2se/1.3/download.html
2. Select the SDK download link for "Linux GNUZIP Tar shell script"
3. Download and install the SDK onto your machine

To install J2SE 1.4.0 on Linux (both RedHat and SuSE)

1. Go to http://java.sun.com/j2se/1.4/download.html
2. Select the SDK download link for "Linux GNUZIP Tar shell script"
3. Download and install the SDK onto your machine.

To install the Java SDK HP-UX

1. Go to http://www.hp.com/products1/unix/java/java2/sdkrte1_3/index.html
2. Select the **Downloads** link
3. Select the link **SDK (includes RTE) for Java 2 version 1.3.1.05 (April, 2002)**
4. You will have to accept a license agreement and then register with HP; do so, and then download and install the SDK onto your machine.

To install the Java SDK on AIX

1. Go to http://www-106.ibm.com/developerworks/java/jdk/aix/index.html
2. Select the **Register and Download** link
3. Select the Java 1.3.1 download link appropriate for your version of AIX
4. You will have to register with IBM; do so, and then download and install the SDK onto your machine.

Host-based Testing vs Target-based Testing

The testing and runtime analysis that you will perform for this tutorial take place entirely on your machine. However, one of the greatest capabilities of Rational Test RealTime is its support for testing and analyzing your software directly on an embedded target. Does this mean you will need to change how you interact with your application when switching from host-based to target-based testing? Will your tests have to be rewritten, for example?

Not at all.

Thanks to Rational's versatile, low-overhead **Target Deployment Technology**, all tests are fully target independent. Each cross-development environment - that is, every combination of compiler, linker, and debugger - has its own Target Deployment Port (TDP). In addition, any TDP can be modified via the Test RealTime user interface at a more granular level, letting you customize a particular test or runtime analysis interaction without affecting neighboring interactions. Such granular tailoring is supported by the concept of *Configurations*. Each Configuration can support one or more TDP and can apply separate customization settings to each interaction assigned to it.

Over thirty reference TDPs, supporting some of the most commonly used cross-development environments, are supplied out-of-the-box. After creation of a project (you will be doing this in a few moments), you can access a list of TDPs installed on the machine.

To view a list of currently installed TDPs:

1. From the **Project** menu, select **Configuration**.

2. Select **New...**

3. Use the dropdown list to scroll through the available TDPs

Target Deployment Port Web Site

As new reference TDPs become available, they are first posted on a customer-accessible Web site. Check this site periodically for news of the latest TDPs to be made available to the Rational Test RealTime and PurifyPlus RealTime community.

To access the Test RealTime Web site:

1. From the **Help** menu, select **Rational Test RealTime on the Web** and **Target Deployment Ports**

Creating and Editing Target Deployment Ports

Does your organization target an environment for which no TDP yet exists? Using the **Target Deployment Port Editor** you can create support, just as many of Rational's customers have done before you.

The reference TDPs supplied with Test RealTime can guide your TDP creation efforts; Rational also provides professional services should you choose to contract out their creation.

> **Note** The Target Deployment Port Editor is not included with the evaluation version of the product.

To access the Target Deployment Port Editor:

1. From the **Tools** menu, select **Target Deployment Port Editor** and **Start**.

For more information about the Target Deployment Port Editor, please refer to the **Rational Test RealTime Target Deployment Guide**.

Every Test RealTime feature is accessible regardless of the environment within which you will be executing your tests. Rest assured, your intended targets are supported.

Goals of the Tutorial

The UMTS base station has been pre-loaded with errors; your responsibility, during the tutorial, will be to uncover:

- poor memory management
- a performance bottleneck
- a logic error in Java code

In addition, test completeness will be achieved by:

- using code coverage to add new tests
- improving your understanding of the code via runtime tracing

To accomplish the above, you will first manipulate the UMTS base station through manual interaction with a mobile phone simulator. Afterwards, automated hands-free interaction will be used.

To continue this tutorial, follow the Java track in the next lesson: Runtime Analysis with Test RealTime and PurifyPlus RealTime.

Runtime Analysis

C and C++ Track

Runtime Analysis for C, C++ and Ada

You will start your tour with the runtime analysis features provided by Test RealTime. The automated component testing features provided by Test RealTime will be discussed in the chapter entitled **Component Testing with Rational Test RealTime**.

Runtime analysis refers to Test RealTime's ability to monitor an application as it executes. There are a variety of advantages to be gained from this monitoring:

- memory profiling
- performance profiling
- code coverage analysis
- runtime tracing

Runtime Analysis

Memory Profiling

Dynamically working with system memory can be quite a complicated affair. If you're not careful, your code might either:

- Fail to free memory - referred to as a memory leak
- Mistakenly reference non-allocated memory - referred to as an array bounds read or array bounds write

A memory leak detection utility monitors an application as it executes, keeping an eye on memory usage to ensure the above problems don't occur. If they do occur, the detection utility points out the sequence of events leading up to the poor usage of memory, helping you deduce the cause of the error and thereby repair your code.

This function is provided in Rational Test RealTime by the memory profiling feature for the C and C++ languages.

Performance Profiling

Optimal performance is, needless to say, crucial for real-time embedded systems. Measuring performance can be quite difficult, however, particularly when it comes to determining the specific functional bottlenecks in your system.

That's where performance profiling monitors come in. These tools watch your application as it executes, measuring statistics such as:

- How often a function is called
- How long it takes for that function to execute
- Which functions are the bottlenecks of your application

With this information you can optimize your code, ensuring all real-time constraints placed upon your system are accommodated.

This function is provided in Rational Test RealTime and Rational PurifyPlus RealTime by the performance profiling feature for the C and C++ languages.

Code Coverage Analysis

One of the greatest difficulties a developer experiences is a failure to determine the portions of code that have gone untested. For many embedded systems, failure is not an option, so every part of an application must be thoroughly tested to ensure there is no unhandled scenario or dead code.

In addition, project managers need a concrete measurement to determine where the team is in the development cycle - in particular, how much more testing needs to be done. A decreasing number of defects does not necessarily mean the product is ready; it might simply mean the portions of code that have been tested appear to be ready.

Code coverage measurement tools observe your running application, flagging every line of code as it executes. Advanced tools - such as Test RealTime and PurifyPlus RealTime - are also able to differentiate different types of execution, such as whether or not a **do-while** loop executed 0 times, 1 time, or 2 or more times. These advanced measurements are critical for software certification in industries such as avionics.

This function is provided in Rational Test RealTime and Rational PurifyPlus RealTime by the code coverage feature for the C, Ada and C++ languages.

Runtime Tracing

As all embedded developers quickly learn, intentions don't necessarily translate into reality. There can often be a vast difference between what you want to happen and what actually happens as your application executes.

This problem becomes more severe when the code is inherited. Yes, you could try to piece things together yourself, but system complexity might just undercut your efforts at understanding the code.

And what about multi-threaded applications? If you've ever encountered race conditions or deadlocks, you know how difficult it can be to uncover the source of the problem.

This is where runtime tracing monitors come in. These utilities graphically display the sequence of function or method calls in your running application - as well as the active threads - illustrating through pictures what is actually happening. With this information, unexpected exceptions can be easily traced back to their source, complex procedures can be distilled to their essence, threading conflicts can be resolved and inherited code can jump off the page and display its inherent logic.

This function, using the industry standard Unified Modeling Language for its graphical display, is provided in Rational Test RealTime by the runtime tracing feature for the C and C++ languages.

Runtime Analysis Exercises

The following exercises will walk you through a typical use case involving the four runtime analysis features of Test RealTime to which you have just been introduced. Pay close attention not only to the capabilities of these features but also to how they are used. The better you understand these features, the more quickly you will be able to adopt them within your own development process.

If you have never run this tutorial before, make sure your machine has a temporary folder in which you can store the test project you will be creating. For the tutorial, it is assumed that the test project will be stored in a folder called **tmp.**

If you have run this tutorial before, do not forget to undo the source file edits you made the last time you ran through it. The following files are modified during the tutorial:

- PhoneNumber.cpp
- UmtsCode.c
- UmtsServer.cpp

If you intend to use Microsoft Visual C/C++, but installed it after installing the product you will need to update the associated TDP. If the product was installed after Microsoft Visual C/C++ then no changes need to be made.
See this page if you need to update the TDP.

To run the tutorial without Microsoft Visual Studio:

1. For Windows: install a recommended GNU C and C++ compiler - click here for instructions. For Solaris, Linux, HP-UX and AIX: use the native C and C++ compiler already installed on your machine
2. During installation of the product:
    - **On Windows:** A local Microsoft Visual Studio compiler and JDK are located, based on registry settings. Only the compiler and JDK located during installation will be accessible within the product.
    - **On UNIX or Linux:** The user is confronted by two interactive dialogs. These dialogs serve to clarify the location of the local GNU compiler and (if present) local JDK. Only the GNU compiler and JDK specified within these dialogs will be accessible within the product.

To make a different compiler available for the product:

1. From the **Tools** menu, select **Target Deployment Port Editor** and **Start**
2. In the Target Deployment Port Editor, from the **File** menu, select **Open**
3. Open the **.xdp** file corresponding to the new compiler for which you would like to generate support
4. In the Target Deployment Port Editor, from the **File** menu, select **Save and Generate**
5. Close the Target Deployment Port Editor

Exercise One

In this exercise you will:

- create a new project in which the UMTS base station source code will be referenced

If you need a refresher about the application you will be using during this tutorial, look here; otherwise, please proceed.

Creating a Project

Typically, there is a one-to-one relationship between your current development project and a Test RealTime project. Although your development project may consist of more than one application, these applications often possess a common theme. Use the Test RealTime project to enforce that theme.

To create a project in Test RealTime:

1. Start Rational Test RealTime:
   Windows: use the **Start** menu
   UNIX: type **studio** on the command line
2. Select the **Get Started** link on the left-hand side of the Test RealTime Start Page. Two links appear on the top of the page: **New Project** and **Open Project**.
3. Select the **New Project** link.
   You should now see the **New Project Wizard**.
4. In the **Project Name** field, enter **BaseStation** (no spaces).
   In the **Location** field, select the ... button, browse to the folder in which you want the BaseStation project to be stored and then select it. For this Tutorial, let's assume that the project has been stored in the **C:\tmp** (Windows) or **\usr\tmp** (UNIX) folder.
   Click the **Next** button.
5. Select, from the list of Target Deployment Ports currently installed on your machine, the one you intend to use to compile, link, and deploy your source code and the test or runtime analysis harness. Since the UMTS base station consists of C++ code, you should choose either **C++ Visual 6.0** if you have Microsoft Visual C++ 6.0 installed, or, if you are using a GNU/native compiler, select the item appropriate for your operating system:
   - Windows - **C++ Gnu 2.95.3-5 (mingw)**
   - Solaris - **C++ Solaris - SC5.1**
   - Linux - **C++ Linux - Gnu 2.95.2**
   - HP-UX - **C++ HP-UX - aCC compiler**
   - **AIX - C++ AIX - IBM C++ Compiler**

   Do not be concerned if the version of the GNU compiler you have installed does not match the version mentioned for the TDP. The differences are not relevant for this tutorial and thus other versions are supported equally as well
6. Click the **Finish** button.

That's it. The project has been created - named **BaseStation** - and a project node by the same name appears on the Project Browser tab of the Project Explorer window on the right-hand side of the UI:

BaseStation

> **Note** A project created in Rational PurifyPlus RealTime could also be used in Rational Test RealTime; a project created in Test RealTime, opened in PurifyPlus RealTime, will be limited to runtime analysis - that is, no tests can be executed and no test reports can be viewed.

Starting a New Activity

Now that you have created a project, it is time to specify:

- your development project's source files
- the type of testing or runtime analysis activity you would like to perform first

To start a new activity:

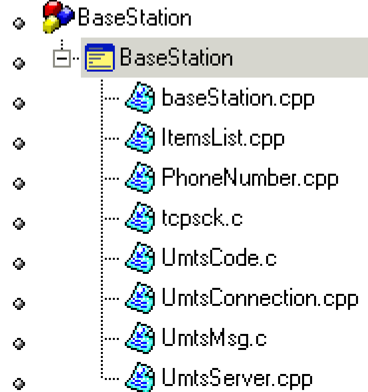
1. Once a project has been created, the user is automatically brought to the **Activities** page. In this tutorial you are starting with a focus on runtime analysis functionality, so select the **Runtime Analysis** link. This will bring up the **Runtime Analysis Wizard**.
2. In the window entitled **Application Files**, you must list all source files for your current development project. For this tutorial, you will directly select the source files. Click **Add** .
3. Browse to folder into which you have installed Test RealTime and then access the folder **\examples\BaseStation_C\src**
4. Make sure **All C++ and Header Files** in the **Files of Type** dropdown box is selected, then left-click-hold-and-drag over all of the C and C++ source files. Now click the **Open** button.
   You should see a set of **.c**, **.cpp** and **.h** files listed in the large listbox of the **Application Files** window.
   Click **Next**.
5. At this time, an analysis engine parses each source file - referred to as tagging. This process is used to extract the various functions, methods, procedures and classes located within each source file, simplifying code browsing within the UI.
6. In the window entitled **Selective Instrumentation** you have the ability to select those functions, procedures, methods or classes that should not be instrumented for runtime analysis. Such selective instrumentation ensures that the instrumentation overhead is kept to a minimum. For this Tutorial, you will be monitoring everything and thus all items should be checked. This should happen by default; if not, click **Select All** . Click **Next**.
7. You have now reached the window entitled **Application Node Name**. Enter the name of the application node that will be created at the conclusion of the Runtime Analysis Wizard; since you will be monitoring execution of the UMTS base station, type the word **BaseStation** within the text field labeled **Name**.
8. The **Application Node Name** window also gives you the opportunity to modify build settings associated with the TDP you selected when creating the Test RealTime project. Some changes may need to be made, depending on your operating system. (Note that these changes do not affect the actual TDP; you will be making changes to a Configuration. A Configuration lets you modify a

variety of settings on a node-by-node basis within a project. You can even reference multiple TDPs in the same Configuration.):

- For Windows:
    - Select the button on the bottom of the **Application Node Name** window entitled **Configuration Settings**.
    - In the window that has just appeared, named **Configuration Settings**, expand the **Build** node in the tree on the left-hand side and left-click the **Compiler** node.
    - In the **Compiler flags** edit box on the right-hand side of the window, add the flag **-MLd** to the end of the list, separated by a space from the flag **-GR**
    - In the **Preprocessor macro definitions** edit box, add the macro **_DEBUG** (make sure to include the preceding underscore, and use only capital letters).
    - Select the **OK** button on the bottom of the window.
- For Solaris only:
    - Select the button on the bottom of the **Application Node Name** window entitled **Configuration Settings**.
    - In the window that has just appeared, named **Configuration Settings**, expand the **Build** node in the tree on the left-hand side and left-click the **Linker** node.
    - Add the following two library flags to the **Additional objects or libraries** edit box on the right-hand side of the screen.
      `-lnsl -lsocket`
    - Select the **OK** button on the bottom of the window.

9. Click **Next**.
10. You are now confronted with the Summary window. Everything should be in order, so click the **Finish** button.

The **BaseStation** application node has now been created. The Project Browser tab of the Project Explorer window should appear as follows:



Additional Build Customization

In this example, the UMTS base station consists of a mix of C and C++ source files. Some C++ compilers can handle both the C and C++ languages; other compilers are not able to do this.

Recall that you selected the TDP for the C++ compiler on your machine. On Windows, the **Visual C++ 6.0** TDP can process both C and C++ files. For the GNU compiler on Windows, and for the native compilers on Solaris, Linux, HP-UX and AIX, you need to specify a C language TDP for the **.c** source files:

If you're using the GNU compiler on Windows, or the native compilers on Solaris, Linux, HP-UX and AIX:

To set a C language TDP for .c files:

1. In the Project Browser, right-click the **tcpsck.c** child node of the **BaseStation** application node and select **Settings**.
2. Position the **Configuration Settings** window that has opened so that you can easily see the Project Browser.
3. Expand the **General** node in the tree on the left-hand side of the window and left-click the **Host Configuration** child node.
4. Click the dropdown list for the **Target Deployment Port** setting. It's current value is the TDP selected when you created the project.
5. Expand the dropdown list - either by left-clicking the field one more time or by selecting the dropdown list arrow to the right - and select the corresponding C language TDP for your machine. Click **Apply** once the new TDP is selected.
6. Back in the Project Browser, select the node for the file **UmtsCode.c** and then follow steps 4 and 5 above.

7. Select the node for the file **UmtsMsg.c** in the Project Browser and then follow steps 4 and 5 above.

8. In the **Configuration Settings** window, click **OK**.

   Note Only the settings for these specific file nodes have been changed; all other file nodes continue to use the default TDP settings.

Conclusion of Exercise One

Have a look at the right side of your screen. This is the Project Explorer window, and within it two tabs are visible.

The first - the **Project Browser** tab - contains a reference to all group, application and test nodes created for the active project. The project node, named **BaseStation**, contains an application node named **BaseStation**; the application node contains a list of all of the source files required to build the UMTS base station application. (Though the project and application nodes have the same name, this is not a requirement.)

The second tab - the **Asset Browser** tab - lets you browse all of your source and test files. If the selected **Sort Method** is **By File**, you are presented with a file-by-file listing of test scripts, source code and source code dependents (such as header files). Note how each header file can be expanded to display every class, function, and method declaration, while each source file can be expanded to display every defined object and method or function. Double-clicking any test script/source file/header file node will open its contents within the Test RealTime editor; double-clicking any class declaration or method definition node will open the relevant source file/header file to the very line of code at which the definition/declaration occurs.

There are two other sort methods as well on the Asset Browser. The first, **By Object**, lets you filter down to classes and methods, independent of the source files. The second, **By Directory**, is primarily applicable to Java packages.

You may have noticed along one of the toolbars at the top of the UI that the TDP you selected in the New Project Wizard is listed in a dropdown box. In fact, this is not a reference to the TDP, it is a reference to the Configuration whose base TDP was the one you selected in the wizard - in the case of this tutorial, it is a TDP supporting C++. (Recall that the Configuration allowed you to select the TDP designed for use with C language files. Configurations are initially named after their base TDP, but this name can be changed.) Should you have multiple configurations for the same project, use this dropdown box to select the active Configuration for execution.

Finally, to the right of the Configuration dropdown list is the **Build** button. This button is used to build your application for application nodes and the test harness for test nodes. The test harness consists of:

- source files needed to build the application of interest
- stubs

- a test driver

The **Build Options** button lets the user decide from which point the build process should initiate and what runtime analysis features should be used. The runtime analysis features do not have to be used at the same time; this Build Options window provides a quick and simple method for deselecting undesired runtime analysis features immediately prior to execution of the build process.

Armed with this knowledge, proceed to Exercise Two.

Exercise Two

In this exercise you will:

- build and execute the UMTS base station application
- manually interact with the UMTS base station application
- view the runtime analysis reports derived from your interaction

Building and Executing the Application

When performing runtime analysis, your source code must be instrumented. Instrumentation, by default, is enabled for all four runtime analysis features - that is, for memory profiling, performance profiling, code coverage analysis and runtime tracing. All four features are turned on by default.

To build and execute the application:

1. In order to instrument, compile, link, and execute the UMTS base station application in preparation for runtime analysis, simply ensure the **BaseStation** application node is selected on the **Project Browser** tab of the Project Explorer window, and then click the **Build** button.
   Do so now.

   Note More information about the source code insertion technology can be found in the **User Guide**, in the chapter **Product Overview->Source Code Insertion**.

2. Notice that in the Output Window at the bottom of the screen, on the **Build** tab, you can watch the preprocessing, instrumentation, compilation, and link phases of the build process as they occur. A double-click on an error listed within any of the Output Window tabs opens the relevant source code file to the appropriate line in the Test RealTime Editor.

3. The build process has completed, and the UMTS base station is running, when the UML-based sequence diagram generated by the runtime tracing feature appears. (More about this feature in a moment.)

4. Close the Project Explorer window on the right-hand side of the UI by clicking the **Close Window** button.

Notice how the graphically displayed data in the **Runtime Trace** viewer dynamically grows - this is because the UMTS base station is being actively monitored. The UMTS base station endlessly searches for mobile phones requesting registration; the Runtime Trace viewer reflects this endless loop. If you wish, use the Pause button on the toolbar to stop the dynamic trace for a moment (the trace is still being recorded, just no longer displayed in real time). In addition, use the **Zoom In** and **Zoom Out** buttons on the toolbar to get a better view of the graphical display (or right-click-hold within the Runtime Trace viewer and select the **Zoom In** or **Zoom Out** options). Undo the Pause when you're ready to proceed.

You'll look at the Runtime Trace viewer in more detail later. Of primary importance right now is interaction with the UMTS base station. You'll do this by using the mobile phone simulator mentioned earlier in the Overview section of this tutorial. Through this manual interaction you will expose memory leaks, performance bottlenecks, incomplete code coverage, and dynamic runtime sequencing.

Interacting with the Application

To run the application:

1. Start the mobile phone by running the provided mobile phone executable built for your operating system. The mobile phone executable is located within the Test RealTime installation folder in the folder **\examples\BaseStation_C\MobilePhone**. The name of the executable depends on your operating system:

   - **Windows:** MobilePhone.exe
   - **Solaris:** MobilePhone.SunOS
   - **Linux:** MobilePhone.Linux
   - **HP-UX:** MobilePhone.HP-UX
   - **AIX:** MobilePhone.AIX

   (A launcher shell script - **MobilePhone.sh** - is provided for the non-Windows platforms as well.)

2. Click the mobile phone's On button ( ).

3. Wait for the mobile phone to connect to the UMTS base station (if you watched the Runtime Trace viewer closely, you would have noticed a display of all the internal method calls of the UMTS base station that occur when a phone attempts to register). The current system time should appear in the mobile phone window when connection has been established.

4. Once connected, dial the phone number **5550000**, then press the button to send this number to the UMTS base station (again, try to see the Runtime Trace viewer update).

5. Unfortunately, the party you are dialing is on the line so you'll find the phone is busy. Shut off the simulator by closing the mobile phone window via the button in its upper right corner.

   The UMTS base station is designed to shut off when a registered phone goes off line. Not a great idea for the real world, but it serves the Tutorial's purposes well. Alternatively, you could have just used the **Stop Build** button located next to the Build button on the toolbar.

6. The UMTS base station has stopped running when the green execution light next to the execution timer - located beneath the Project Explorer window on the lower right-hand side of the UI - stops flashing ( ). Wait for it to stop flashing.

   Everything that occurred at the code level in the UMTS base station was monitored by all four runtime analysis features. Once the UMTS base station stopped (i.e. once the instrumented application stopped), all runtime analysis information was written to user accessible reports that are directly linked to the UMTS base station source code. In order to look at these reports:

7. Reopen the Project Explorer window by selecting the menu item **View->Other Windows->Project Window**

8. In the Project Explorer window, on the Project Browser tab, double-click the **BaseStation** application node. All four runtime analysis reports will open. (Alternatively, right-click the **BaseStation** application node and select **View Report->All**.)

9. Close the Project Explorer window and the Output Window (at the bottom of the UI) to create room for the now-opened reports. You may also want to resize the left-hand window to gain additional room.

Understanding Runtime Tracing

To view the UML sequence diagram report:

1. Select the **Runtime Trace** tab.

2. As you recall, the Runtime Trace viewer displayed all objects and all method calls involved in the execution of the UMTS base station code. Using the toolbar **Zoom Out** button, zoom out from the tracing diagram until you can see at least five vertical bars.

3. Make sure you are looking at the top of the runtime tracing diagram using the slider bar on the right.

   What you are looking at is a sequence diagram of all events that occurred during the execution of your code. This sequence diagram uses a notation taken from the Unified Modeling Language, thus it can be correctly referred to as a UML-based sequence diagram.

   The vertical lines are referred to as lifelines. Each lifeline represents either a C source file or a C++ object instance. The very first lifeline, represented by a stick figure, is considered the "world" - that is, the operating system. In this UMTS base station tracing diagram, the next lifeline to the right represents an object instance named **Obj1**, derived from the **UmtsServer** class.

   Green lines are constructor calls, black lines are method calls, red lines are method returns, and blue lines are destructor calls. Hover the mouse over any method call to see the full text. Notice how every call and call return is time stamped.

   Everything in the Runtime Trace viewer is hyperlinked to the monitored source code. For example, if you click on the **Obj1::UmtsServer** lifeline, the header file in which the UmtsServer class declaration appears is opened for you, the relevant section highlighted. (Close the source file by right-clicking the tab of the Text Editor and selecting **Close**.) All function calls can be left-clicked as well in order to view the source code. Look at the very top of the **Obj1::UmtsServer** lifeline. It's "birth" appears to consist of a **List()** constructor first, then a **UmtsServer()** constructor. Why a call to the List() constructor if the object is an instance of the UmtsServer class? Click on the **UmtsServer()** lifeline again - see how the UmtsServer() constructor inherits from the List() class? This is why the List() constructor is called first. Click the two constructor calls if you wish to pursue this matter further.

   Notice how the window on the left-hand side of the user interface - called the **Report Window** - contains a reference to all classes and class instances. Double-clicking any object referenced in this window will jump you to its birth in the Runtime Trace viewer. This window can also be used to filter the runtime tracing diagram.

4. In the left-hand window, close the node labeled **NETWORKNODE.H** - notice how all objects derived from the NetworkNode class declared in this header file are reduced to a single lifeline.

5. Reopen the node labeled **NETWORKNODE.H**.

   You've probably noticed the vertical graph with the green bar to the left of the Runtime Trace viewer. This is the **Coverage Bar**. It highlights, in synchronization with the trace diagram, the percentage of total code coverage achieved during execution of the monitored application. The Coverage Bar's caption states the percentage of code coverage achieved by the particular

interaction presently displayed in the Runtime Trace viewer. Scroll down the trace diagram; note how code coverage gradually increases until a steady state is achieved. This steady state is achieved following the moment at which the mobile phone has connected to the UMTS base station. Dialing the phone number increases code coverage a bit; shutting off the phone creates a last burst of code coverage up until the moment the UMTS base station is shut off. Can you locate where, on the trace diagram, the mobile phone simulator first connected to the UMTS base station? (The Coverage Bar can be toggled on and off using the right-click-hold menu within the Runtime Trace viewer.)

Note If the C++ code in the UMTS base station spawned multiple threads, the Coverage Bar would be joined by the **Thread Bar**, a vertical graph highlighting the active thread at any given moment within the trace diagram. A double-click on this bar would open a threading window, detailing thread state changes throughout your application's execution. This thread monitoring feature is also available for the Java language.

Continue to look around the trace diagram. Can you locate the repetitive loop in which the UMTS base station looks for attempted mobile phone registration (it always starts with a call to the C function **tcpsck_data_ready**)? You can filter out this loop using a couple of methods. One is to simply hover the mouse over a method or function call you wish to filter, right-click-hold and select **Filter Message**. An alternative method would be to build your own filter. You will do both.

6. Hover the mouse over any call of the **tcpsck_data_ready** function, right-click-hold and select **Filter Message** - the function call should disappear from the entire trace.
7. Select the menu item **Runtime Trace->Filters** (you'll see the filter you just performed listed here)
   Click the **Import** button, browse to the installation folder and then the folder **\examples\BaseStation_C**, and then **Open** the filter file **filters.tft**
8. Check that **BaseStation Phone Search Filter** is selected. Select it if necessary.
9. Click the **OK** button.

   The loop has been removed.

   Not only can the runtime tracing feature capture standard function/method calls, but it can also capture thrown exceptions.
10. View the very bottom of the runtime tracing diagram using the slider bar.

Do you see the icon for the catch statement - catch exception (you may have to drag the slider bar slightly upward; closing the **NETWORKNODE.H** node in the left-hand report window will also make things easier to see)? This **Catch Exception** statement is preceded by a diagonal **Throw Exception**. Why diagonal? Because when the

exception was thrown, prior to executing the Catch statement, the **LostConnection** constructor and **UmtsMsg** destructor were called. Click various elements to view the source code involved in the thrown exception and thus decipher the sequence of events.

This exception occurred by design, but it is clear how the runtime tracing feature, through the power of UML, would be extremely useful if you have:

- inherited old or foreign code
- unexpected exceptions
- questions about whether what you designed is occurring in practice

And you are guaranteed the identical functionality for application execution on an embedded target.

Understanding Memory Profiling

The Memory Profile viewer displays a record of improper memory usage within the application of interest.

To read the Memory Profiling report:

1. Select the **Memory Profile** tab.

   First, block and byte memory use is summarized for you in a bar chart, immediately followed by a textual description to the same information. What you have is a record of:

   - total number of blocks/bytes allocated for the entire run
   - total number of non-freed blocks/bytes allocated for the entire run
   - total number of blocks/bytes in use at any one time

   If any memory errors were detected, or if any warnings are warranted, those comments are listed next. The Report Window on the left hand side of the screen gives you a quick look at the contents of the report - your manual interaction with the UMTS base station via the simulated mobile phone has resulted in the creation of **Test #1**. If you click an item in the Report Window, the memory profiling report will scroll to the proper location.

2. On the Report Window, left-click the **ABWL** error.

Apparently, the memory profiling feature has detected a **Late Detect Array Bounds Write** (**ABWL**) - in other words, the UMTS base station code attempted to add data to an array element that does not exist. This error report is followed by the call stack, with the last function in the call stack listed first. Notice how each function is highlighted; clicking on the functions in the call stack will jump you to the relevant source code. Each source code file is highlighted at the line in which memory was

requested - in this particular case, some part of the UMTS base station code overwrote an array, thereby causing the ABWL error.

The **ABWL** is followed by one **File In Use** (**FIU**) and four **Memory Leak** (**MLK**) warnings. The **File In Use** warning references **<internal use>** - in other words, the file is being used by the memory profiling feature. As for the memory leaks - well it looks like you have some work to do here. Although it is conceivable the memory leak occurs by design (e.g. perhaps some clean-up code has not yet been written), assuredly the UMTS base station is not meant to have any.

Finally, the exit code is printed - look for the informational/warning note in the viewer starting with the words **Program exit code**. The memory profile report lists the exit code as a warning if it is of any value other than 0.

Notice how easily this information has been acquired; no work was required on your part. A real advantage is that memory leak detection can now be part of your regression test suite. Traditionally, if embedded developers looked for memory leaks at all, it was done while using a debugger - a process that does not lend itself to automation and thus repeatability. The memory profiling feature lets you automate memory leak detection.

And again, the identical functionality can be used on either your host platform or on your embedded target.

Understanding Performance Profiling

The Performance Profile viewer displays the execution time for all functions or methods executing within the application of interest, thereby allowing the user to uncover potential bottlenecks. First, the three functions or methods requiring the most amount of time are displayed graphically in a pie chart (up to six functions will be displayed if each is individually responsible for more than 5% of total execution time). This is then followed by a sortable list of every function or method, with timing measurements displayed.

To read the Performance Profiling report:

1. Select the **Performance Profile** tab.

   Notice how the function **tcpsck_data_ready** was responsible for around 45% to 50% of the time spent processing information in the UMTS base station. By looking at the table, where times are listed in microseconds, we can see that this function's average execution time was between 1 to 2 seconds (it will vary somewhat based on your machine) and that it has no descendents - i.e. it never calls and then awaits the return of other functions or methods (which explains why the **Function** time matches the **F+D** time). Is this to be expected? If you wished, you could click on the function name in the table to jump to that function to see if its execution time can be reduced.

   Each column can be used to sort the table - simply click on the column heading.

2. Click the column heading entitled **F+D Time**

   It is probably no surprise that the **main()** procedure - combined with its descendents - takes the longest time to execute overall. Notice, though, that the **main()** procedure itself only takes around 300μs (depending on the operating system) to execute - so there doesn't appear to be any bottleneck here. The **main()** procedure spends its life waiting for the UMTS base station to exit.

As with the memory profiling feature, notice how easy it was to gather this information. Performance profiling can now also be part of your regression test suite. And again, as with every other runtime analysis feature, performance profiling functionality is identical whether it is used on your host platform or on your embedded target.

Understanding Code Coverage

And finally, here you have the code coverage analysis report. The code coverage feature exposes the code coverage achieved either through manual interaction with the application of interest or via automated testing.

To view the Code Coverage report:

1. Select the **Code Coverage** tab.

   On the left hand side of the screen, in the Report Window, you see a reference to **Root** and then to all of the source and header files of the UMTS base station. **Root** is a global reference - that is, to overall coverage. For each individual source and header file, a small icon to the left indicates the level of coverage (green means covered, red means not covered).

   In the Code Coverage viewer, on the **Source** tab, a graphical summary of total coverage is presented in a bar chart - that is, information related to **Root**. Five levels of code coverage are accessible when the source code is C++, and those five levels are represented here. (Four more levels of coverage are accessible when working with the C language - up to and including Multiple Conditions/Modified Conditions. These levels are required by stringent certification standards such as aviation's DO-178B.) Notice how, on the toolbar, there is a reference to these five possible coverage levels (F E B I L).

2. Deselect **Loops Code Coverage** (L)

   Notice how the bar chart is updated.

3. Reselect **Loops Code Coverage** (L)

4. In the Report Window to the left, select the **PhoneNumber.cpp** node.

   The **Source** tab now displays the source code located in the file **PhoneNumber.cpp**. This code is colored to reflect the level of coverage

achieved. Green means the code was covered, red means the code was not covered.

5. In the Report Window, expand the **PhoneNumber.cpp** node and then select the **void PhoneNumber::clearNumber()** child node

   The **clearNumber()** function should now be visible on the **Source** tab. Notice how its **for** instruction is colored orange and sitting on a dotted underline. This is because the **for** statement was only partially covered.

6. Click on the orange **for** keyword in the **clearNumber()** function

   As you can see, the **for** loop was only executed multiple times, not once or zero times. Why should you care? Well some certification agencies require that all three cases be covered for a **for** statement to be considered covered. If you don't care about this level of coverage, just deselect **Loops Code Coverage**:

7. On the toolbar, deselect **Loops Code Coverage** (L).

   Now the **for** loop is green. If you would like to add a comment to your code indicating how this loop is not covered by typical use of the mobile phone simulator, have a look at the code by right-clicking the **for** statement and selecting **Edit Source**.

8. Select the **Rates** tab in the Code Coverage viewer

   The **Rates** tab is used to display the various coverage levels for

- the entire application
- each source file
- individual functions/methods

   Click various nodes in the Report Window in order to browse the Rates tab. Note how a selection of the Root node gives you a summary of the entire application.

9. From the **File** menu, select **Save Project**.

Conclusion of Exercise Two

With virtually minimal effort, you have successfully instrumented your source code for all four runtime analysis features. Manual interaction (in your case, via a mobile phone simulator) was monitored, and the subsequent runtime analysis results were displayed for you graphically. Source code is immediately accessible from these reports, so nothing prevents the developer from using the results to correct possible anomalies.

In addition, using the Test by Test option provided with each runtime analysis feature (introduced in the Further Work section for code coverage), you can easily discern the effectiveness of a test, ensuring maximal reuse without waste.

Your next step is to use the runtime analysis results to remove memory leaks, improve performance, and increase code coverage.

Exercise Three

In this exercise you will:

- Improve the UMTS base station code by eliminating memory leaks and by improving performance
- Increase code coverage
- Rerun the manual test to verify that the defects have been fixed

Using Memory Profiling to Remove Memory Leaks

By using the call stacks displayed in the Memory Profile viewer, you will deduce the corrections that need to be made to eliminate memory errors.

To locate and fix memory errors:

1. Select the **Memory Profile** tab.
2. Select the **ABWL** error node in the Report Window on the left hand side of the screen.

   Have a look at the call stack for the Late Detect Array Bounds Write error. Three C++ methods are listed.
3. Select the last function first, the one that occurs inside **main()**

   Within the main() procedure a **UmtsServer** object is instantiated. Nothing looks out of sorts here, so return to the call stack.
4. Close the source file for the main() procedure, and then click the second function from the bottom in the call stack referenced by the **ABWL** error - the **UmtsServer** constructor.

   The next function in the stack is the **UmtsServer** constructor. The line in the constructor that is flagged, the creation of a **NetworkNodes** object, is a call to the **List** constructor. Continue to follow the sequence of events.
5. Close the source file for the **UmtsServer** constructor, and then click the top function in the call stack referenced by the **ABWL** error - the **List** constructor.

   The highlighted line is a call to malloc. A quick look at this function shows that a return to the UmtsServer constructor is fairly quick, and nothing seems unusual. You should continue to track the string of events as they happened to see if the ABWL error shows itself. Return to the UmtsServer constructor.

6. Close the source file for the **List** constructor, and then click the second function from the bottom in the call stack referenced by the **ABWL** error - the **UmtsServer** constructor.

   What happens next? The **NetworkNodes** object was assigned 3 **List** objects in an array. Immediately following the call to the **List** constructor, 4 elements are assigned to this array. Not good. The **NetworkNodes** object should be an array of 4 **List** objects, not 3.

7. In the source code, change the line

```
networkNodes = new List(3);
```

   to

```
networkNodes = new List(4);
```

8. From the **File** menu, select **Save**. The revised file **UmtsServer.cpp** is saved.

This should fix the ABWL error. Before redoing you manual test to verify if the memory error was fixed, move on to the Performance Profile viewer and see if you can streamline the performance of the UMTS base station code.

As for the other memory warnings - that's for you to figure out!

Using Performance Profiling to Improve Performance

Now you will use information in the Performance Profile viewer to determine if you can improve performance in the UMTS base station code.

To locate and fix performance bottlenecks:

1. Select the **Performance Profile** tab.

2. Within the table, left-click the column title **Avg F Time** (Average Function Time) in order to sort the table by this column. (You may want to scroll down the report a bit to view more data elements in the table.)

   For this exercise you have sorted by the Average Function Time - that is, you're looking at functions that take, on average, the longest time to execute. This isn't the only potential type of bottleneck in an application - for example, perhaps it is the number of times one function calls its descendants that is the problem - but for this exercise, you will look here first.

   As the developer of this UMTS base station, you would know that the C function **tcpsck_data_ready()** does take a fair amount of time to execute - so you won't look here first (although feel free to have a look if you wish). Instead look at a different function in the table.

3. Select the link for the C function **checkUmtsNetworkConnection()**

   A quick look at the source code shows you that the developer treated this as a dummy function, inserting a "time-waster" to make it appear as if the function were executing. Simply comment out the line.

4. Change the code from
```
doSomeStuff(1);
```
to
```
// doSomeStuff(1);
```
5. From the **File** menu, select **Save**

This way, the **checkUmtsNetworkConnection()** method will do nothing at all. The next time you perform the manual test, this C++ method should have an execution time of 0.

There is another UmtsServer class method that also needs to be improved. Have a look, if you wish.

Using Code Coverage Analysis to Improve Code Coverage

You will now use the information gathered by the code coverage analysis feature to modify the manual test in such a way as to improve code coverage.

To improve coverage of your code:

1. Select the **Code Coverage** tab.
2. If necessary, select the **Source** tab of the Code Coverage viewer
3. In the Report Window on the left-hand side of the screen, open the **UmtsConnection.cpp** node and then select the **processMessages()** child node
4. Drag the slider bar down slightly until you see the line:
```
if (strcmp(msg->phoneNumber,"5550001")==0)
```

Notice how the **if** statement was never true - the **else** block is green, but the **if** block is red. In order to improve coverage of this **if** statement, you need to make the boolean expression evaluate to true.

According to this code, the **if** expression would evaluate to true if mobile phone sends the phone number **5550001**. You should do that.

You will now rerun the UMTS base station executable, restart the mobile phone simulator, and dial this new phone number. When you have finished, you will check the memory profiling, performance profiling, and code coverage analysis reports to see if you have improved matters.

Redoing the Manual Test

You have changed some source code, so some of the UMTS base station code will have to be rebuilt. The integrated build process of Test RealTime is aware of these changes, so you do not have to specify the particular files that have been modified.

To rebuild your application:

1. Select the menu item **View->Other Windows->Project Window**.
2. From the **Window** menu, select **Close All**.
3. Select the **Project Browser** tab in the Project Explorer window that has now appeared on the right-hand side of the UI.
4. Right-click the **BaseStation** application node and select **Rebuild**. When you select **Rebuild**, all files are rebuilt, whereas **Build** simply rebuilds those files that have been changed. If no files had been changed, you could have just selected **Execute BaseStation**.
5. Once the UMTS base station is running (indicated by the appearance of the Runtime Trace viewer), run the mobile phone simulator as before.
6. Click the mobile phone's **On** button ( ).
7. Wait for the mobile phone to connect to the UMTS base station (if you watch the dynamic trace closely, you'll notice a display of all the actions that occur when a phone registers with the server). The time should appear in the mobile phone window.
8. Once connected, dial the phone number **5550001**, then press the button again to send this number to the UMTS base station (again, watch the dynamic trace update).
9. Success! You have connected to the intended party. Stop right here to see the results of your work. Close the mobile phone window by clicking the button on the right side of its window caption. As you may recall, this action will shut down the UMTS base station as well.
10. The UMTS base station has stopped running when the green execution light next to the execution timer - located beneath the Project Explorer window on the lower right-hand side of the UI - stops flashing (). Wait for it to stop flashing.
11. In the Project Explorer window, on the **Project Browser** tab, double-click the **BaseStation** application node. All four runtime analysis reports will open with refreshed information. (Alternatively, right-click the **BaseStation** node and select **View Report->All**.)
12. Close the Project Explorer window to the right and the Output Window at the bottom.

So have you improved your code and increased code coverage?

Verifying Success

Was the memory leak eliminated?

To check that the memory leak was fixed:

1. Select the **Memory Profile** tab.
2. Maximize the window
3. In the Report Window on the left-hand side of the screen, look inside the node labeled **Test #2** - do you see the **ABWL** error anymore?

You successfully eliminated the **ABWL** error. Have you improved performance?

To check that performance was improved:

1. Select the **Performance Profile** tab.
2. Select the menu option **Performance Profile->Test by Test**
3. In the Report Window on the left-hand side of the screen, left-click the node labeled **Test #2**
4. Sort the table by **Avg F Time** - do you see the function **checkUmtsNetworkConnection()**?

You successfully improved performance. Was code coverage improved?

To check that code coverage was improved:

1. Select the **Code Coverage** tab.
2. In the Report Window on the left-hand side of the screen, open the node for **UmtsConnection.cpp** and then left-click the method **processMessages()**
3. Scroll down until you can see the **if** statement for which you have attempted to force an evaluation of true - did you? Has code coverage been improved?

   You successfully improved code coverage. Note, by the way, that you can discern what this second manual interaction has gained you in terms of code coverage.
4. Select the menu option **Code Coverage->Test by Test**
5. In the Report Window on the left-hand side of the screen, reselect the method **processMessages()**
6. With your mouse anywhere within the **Source** tab of the **Code Coverage** viewer, right-click and select **CrossRef**
7. Scroll the Code Coverage viewer to expose the line of code that has been newly covered and then left-click it:
   ```
   strcpy(response.command,cmd_accepted);
   ```

Notice that only **Test #2** is mentioned. However, what tests are listed for the **if** statement itself?

8. Left-click the line
```
if (strcmp(msg->phoneNumber,"5550001")==0)
```
   Both **Test #1** and **Test #2** are listed. As further proof, do the following.

9. With your mouse anywhere on the **Source** tab of the **Code Coverage** viewer, right-click and deselect **Cross Reference**

10. In the Report Window, on the left-hand side of the screen, open the **Tests** node and deselect the checkbox next to **Test #2**.

Since you have deselected **Test #2**, all you are left with is the code coverage that has resulted from running **Test #1**, and **Test #1** never forced the **if** statement to evaluate to true. Thus the newly covered code has become red again - in other words, unevaluated.

Conclusion of Exercise Three

After correcting the UMTS base station code directly in the Test RealTime Text Editor, you simply rebuilt your application and used the mobile phone simulator to initiate further interaction. A second look at the runtime analysis reports validated the accuracy of your changes. Consider the speed with which you could perform these monitoring activities once you are familiar with the user interface...

Conclusion

Conclusion - with a Word about Process

Automated memory profiling, performance profiling, runtime tracing, and code coverage analysis - no less important in the embedded world than elsewhere in software. So why is it done less often? Why is it so much harder to find solutions for the embedded market? It is because embedded software development involves special restrictions that make these functions more difficult to achieve, particularly when speaking of target-based execution:

- strong real-time timing constraints
- low memory footprints
- multiple RTOS/chip vendors
- limited host-target connectivity
- complicated test harness creation for target-hosted execution

- etc.

Rational Test RealTime and Rational PurifyPlus RealTime have been built expressly with the embedded developer in mind, so all of the above complications have been overcome. Nothing stands between you and the use of a full complement of runtime analysis features in both your native and target environment.

So use them! It should be automatic - part of all your regression testing efforts (discussed in greater detail in the Tutorial conclusion). As you have seen, these functions are only a mouse-click away so there is absolutely no drain on your time.

You may be concerned about the instrumentation - "But I don't want my final product to be an instrumented application. Doesn't it have to be if I'm testing instrumented code?" No, it does not have to be:

11. Using the code coverage feature, generate a series of tests that cover 100% of your code
12. Instrument that code for full runtime analysis
13. Uncover and address all reliability errors as you test (e.g. memory leaks, overly slow functions, improper function flow, untested code)
14. Now uninstrument your code - that is, simply shut off all runtime analysis features and rebuild your application
15. Run your regression suite of tests once more, this time looking only for functional errors
16. No errors? Time to ship

Make it part of your development process, just another step before you check in code for the night. Rational Test RealTime and Rational PurifyPlus RealTime simplify runtime analysis to such an extent that there is no longer a reason not to do it.

Test RealTime users may now proceed to the next lesson: Automated Component Testing with Test RealTime

Java Track

Runtime Analysis for Java

You will start your tour with the runtime analysis features shared by Test RealTime. The automated component testing features provided by Test RealTime will be discussed in the chapter entitled **Component Testing with Rational Test RealTime**.

Runtime analysis refers to Test RealTime's and PurifyPlus RealTime's ability to monitor an application as it executes. There are a variety of advantages to be gained from this monitoring:

- memory profiling
- performance profiling
- code coverage analysis
- runtime tracing

Runtime Analysis

Memory Profiling

One of the reasons for Java's success is its ability to perform memory management - that is, Java is designed to ensure memory is properly allocated and freed. Does this mean you, as a developer, no longer have any responsibility regarding your software's usage of memory?

No.

There are two primary reasons for a developer to remain vigilant:

- Java applications CAN leak memory. Not in the traditional way, where memory is no longer referenced by your application and yet not accessible by the system OS - such a problem can not occur. However, if you allocate memory, use it, then fail to free (i.e. dereference), then the Java garbage collector will never reclaim it. Do this enough and your system will still run out of memory.
- Excessive memory usage can result in application slowdown. Do you know how much memory your application is using at any given time? If you have access to limited memory, do you know how much your application has allocated? Are there places in your code that could be optimized to use less memory, thereby freeing systems resources for other activities?

A memory profiling utility indicates a running tally of allocated memory as well as those portions of your code that reference memory at a specified moment in time (such as when the program exits). Such information can be used to ensure all unnecessary memory has been dereferenced and that memory usage has been optimized.

This function is provided in Rational Test RealTime by the memory profiling feature for the Java language.

Performance Profiling

Optimal performance is, needless to say, crucial for real-time embedded systems. Measuring performance can be quite difficult, however, particularly when it comes to determining the specific functional bottlenecks in your system.

That's where performance profiling monitors come in. These tools watch your application as it executes, measuring statistics such as:

- How often a function is called
- How long it takes for that function to execute
- Which functions are the bottlenecks of your application

With this information you can optimize your code, ensuring all real-time constraints placed upon your system are accommodated.

This function is provided in Rational Test RealTime by the performance profiling feature for the Java language.

Code Coverage Analysis

One of the greatest difficulties a developer experiences is a failure to determine the portions of code that have gone untested. For many embedded systems, failure is not an option, so every part of an application must be thoroughly tested to ensure there is no unhandled scenario or dead code.

In addition, product managers need a concrete measurement to determine where the team is in the development cycle - in particular, how much more testing needs to be done. A decreasing number of defects does not necessarily mean the product is ready; it might simply mean the portions of code that have been tested appear to be ready.

Code coverage measurement tools observe your running application, flagging every line of code as it executes. Advanced tools - such as Test RealTime and PurifyPlus RealTime - are also able to differentiate different types of execution, such as whether or not a **do-while** loop executed 0 times, 1 time, or 2 or more times. These advanced measurements are critical for software certification in industries such as avionics.

This function is provided in Rational Test RealTime by the code coverage feature for the Java language.

Runtime Tracing

As all embedded developers quickly learn, intentions don't necessarily translate into reality. There can often be a vast difference between what you want to happen and what actually happens as your application executes.

This problem becomes more severe when the code is inherited. Yes, you could try to piece things together yourself, but system complexity might just undercut your efforts at understanding the code.

And what about multi-threaded applications? If you've ever encountered race conditions or deadlocks, you know how difficult it can be to uncover the source of the problem.

This is where runtime tracing monitors come in. These utilities graphically display the sequence of function or method calls in your running application - as well as the active threads - illustrating through pictures what is actually happening. With this information, unexpected exceptions can be easily traced back to their source, complex procedures can be distilled to their essence, threading conflicts can be resolved and inherited code can jump off the page and display its inherent logic.

This function, using the industry standard Unified Modeling Language for its graphical display, is provided in Rational Test RealTime by the runtime tracing feature for the Java language.

Runtime Analysis Exercises

The following exercises will walk you through a typical use case involving the four runtime analysis features of Test RealTime and PurifyPlus RealTime to which you have just been introduced. Pay close attention not only to the capabilities of these features but also to how they are used. The better you understand these features, the more quickly you will be able to adopt them within your own development process.

If you have never run this tutorial before, make sure your machine has a temporary folder in which you can store the test project you will be creating. For the tutorial, it is assumed that the test project will be stored in a folder called **tmp**

Do you have JDK 1.3.1 or 1.4.0 installed? This is necessary for performance of the tutorial.

During installation of Rational Test RealTime:

- on Windows - A local Microsoft Visual Studio compiler and JDK are located, based on registry settings. Only the compiler and JDK located during installation will be accessible within Test RealTime.
- on Unix/Linux - The user is confronted by two interactive dialogs. These dialogs serve to clarify the location of the local GNU compiler and (if present) local JDK. Only the GNU compiler and JDK specified within these dialogs will be accessible within Test RealTime.

If you have run this tutorial before, don't forget to undo the source file edits you made the last time you ran through it. The following files are modified during the tutorial:

- LogServer.java
- NetworkLoadMonitor.java
- PhoneNumber.java

To make a different JDK accessible in Test RealTime:

1. Run Test RealTime
2. From the **Tools** menu, select **Target Deployment Port Editor** and **Start**.
3. In the TDP Editor, from the **File** menu, select **Open**.
4. Open the **.xdp** file corresponding to the new JDK for which you would like to generate support
5. In the TDP Editor, from the **File** menu, select **Save**.
6. Close the TDP Editor

Exercise One

In this exercise you will:

- create a new project in which the UMTS base station source code will be referenced

If you need a refresher about the application you will be using during this tutorial, look here; otherwise, please proceed.

Creating a Project

Typically, there is a one-to-one relationship between your current development project and a Test RealTime project. Although your development project may consist of more than one application, these applications often possess a common theme. Use the Test RealTime project to enforce that theme.

To create a project:

1. To start Rational Test RealTime
    - Windows - use the **Start** menu
    - Solaris/Linux/HP-UX/AIX - type **studio** on the command line
2. Select the **Get Started** link on the left-hand side of the Test RealTime user interface (UI). Two links will appear on the right-hand side of the UI - one called **New Project** and one called **Open Project**. Select the **New Project** link. You should now see the **New Project Wizard**.

    In the **Project Name** field, enter **BaseStation_Java** (no spaces).

    In the **Location** field, select the ... button, browse to the folder in which you want the BaseStation project to be stored and then select it. This Tutorial will assume that the project has been stored in the **tmp** folder.
    Click the **Next** button.

3. Select, from the list of Target Deployment Ports currently installed on your machine, the one you intend to use to compile, link, and deploy your source code and the Test RealTime testing and/or runtime analysis harness. This is the same TDP you configured earlier in the tutorial. It is either:

    - Java JDK 1.3.1
    - Java JDK 1.4.0

    Click the **Finish** button.

That's it. The project has been created - named **BaseStation_Java** - and a project node by the same name appears on the Project Browser tab of the Project Explorer window on the right-hand side of the UI:

> **Note** A project created in PurifyPlus RealTime could also be used in Test RealTime; a project created in Test RealTime, opened in PurifyPlus RealTime, will be limited to runtime analysis - that is, no tests can be executed and no test reports can be viewed.

Starting a New Activity

Now that you have created a project, it is time to specify:

- your development project's source files
- the type of testing or runtime analysis activity you would like to perform first

To start a new activity:

1. Once a project has been created, the user is automatically brought to the **Activities** page. In this tutorial you are starting with a focus on the runtime analysis features, so select the **Runtime Analysis** link. This will bring up the **Runtime Analysis Wizard**.
2. In the window entitled **Application Files**, you must list all source files for your current development project. For this tutorial, you will directly select the source files. Select the **Add** button.
3. Browse to folder into which you have installed Test RealTime and then access the folder **\examples\BaseStation_Java\src\baseStation**
4. Make sure **All Java Files** in the **Files of Type** dropdown box is selected, then left-click-hold-and-drag over all of the eleven Java source files. Now click the **Open** button.
   You should see these eleven files listed in the large listbox of the **Application Files** window.
   Click **Next**.

5. At this time, an analysis engine parses each source file - referred to as tagging. This process is used to extract the various methods and classes located within each source file, simplifying code browsing within the UI.

6. In the window entitled **Selective Instrumentation** you have the ability to select those classes/methods that should not be instrumented for runtime analysis. Such selective instrumentation ensures that the instrumentation overhead is kept to a minimum. For this Tutorial, you will be monitoring everything, so simply click the **Next** button.

7. In the window entitled **Configuration Settings for Java**, you need to define your application's class path as well as the fully qualified name of the main class for your application.

   In the **Class path** text box, click the ... button, then the button, and then browse to and select the folder **\examples\BaseStation_Java\src** (located in the Test RealTime installation folder). The package used by the Java-based UMTS base station is named **baseStation**, and it's located in the **src** folder you just referenced.

   Note For Windows users, if a folder in the path has a name containing a space, change that name following the DOS 8.3 naming convention rules (such as replacing **C:\Program Files** with **C:\Progra~1**).

- In the **Java main class** text box, select the **BaseStation** class from the dropdown list. Your screen should look like this:

Now click the **Next** button.

8. You have now reached the window entitled **Application Node Name**. Enter the name of the application node that will be created at the conclusion of the Runtime Analysis Wizard; since you will be monitoring execution of the Java-based UMTS base station, type the word **BaseStation** within the text field labeled **Name**.

9. You also need to make some minor changes to the way you would like the TDP to be used. These modifications are specifically aimed at the memory profiling feature and are being used simply to illustrate additional concepts within the Tutorial.
   At the bottom of the Application Node Name window, click the **Configuration Settings** button.

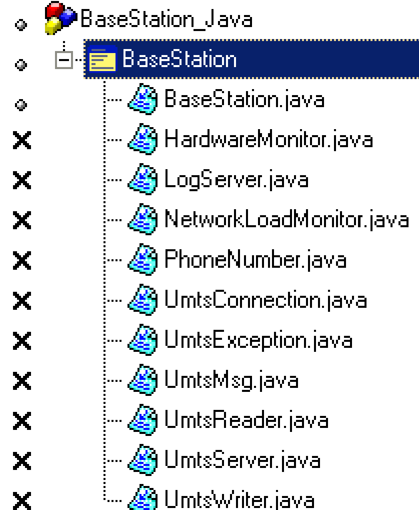
10. Expand the **Runtime Analysis** node on the left-hand side of the **Configuration Settings** window, expand the **Memory Profiling** child node, and then left-click the **JVMPI** child node.

11. Test RealTime uses the JVMPI interface of supported JVMs to acquire memory profiling information. The following custom changes should be made to the Configuration for the purposes of this tutorial:

- On the right-hand side of the window, set the Value of the **Take a Snapshot** setting to **After Each Garbage Collection**. Though it is possible to interactively take memory snapshots during execution, setting this option ensures you will have sufficient data to work with in this tutorial.
- Set the Value of **Display Only Listed Packages** to **baseStation** (the Value is case-sensitive, so enter it carefully). This setting ensures you filter out references to objects derived from classes not explicitly defined within the application-under-test.
- Set the Value of **Collect Referenced Objects** to **Yes**. By collecting referenced objects, the memory profiling diff functionality will provide greater visibility into whether or not the application-under-test is properly allocating/deallocating objects.

12. In the **Configuration Settings** window, click **Ok**.
13. In the **Application Node Name** window, click **Next**.
14. You are now confronted with the Summary window. Everything should be in order, so click **Finish**.

The **BaseStation** application node has now been created in the Project Explorer window, on the Project Browser tab, located on the right-hand side of the user interface. If you expand the **BaseStation** application node, you should see the following:



Why is the **exclude** ✖ status indicated for all but one **.java** file? This is because the build process need only reference the source file containing the main Java class when calling the Java compiler. This source file is **BaseStation.java**.

Conclusion of Exercise One

Have a look at the right side of your screen. This is the Project Explorer window, and within it two tabs are visible.

The first - the **Project Browser** tab - contains a reference to all group, application and test nodes created for the active project. The project node, named **BaseStation_Java**, contains an application node named **BaseStation**; the application node contains a list of all of the source files required to build the UMTS base station application.

The second tab - the **Asset Browser** tab - lets you browse all of your source and test files. If the selected **Sort Method** is **By File**, you are presented with a file-by-file listing of test scripts, source code and source code dependents (this last is applicable to C, C++ and Ada only). Note how each source file can be expanded to display every class declaration and method definition within them. Double-clicking any test script/source file node will open its contents within the Test RealTime editor; double-clicking any class declaration or method definition node will open the relevant source file/header file to the very line of code at which the definition/declaration occurs. (To close a Text Editor window, right-click its associated tab and select **Close**.)

There are two other sort methods as well on the Asset Browser. The first, **By Object**, lists classes and methods independent of their associated source files. The second, **By Directory**, sorts source files based on their associated Java packages.

You may have noticed along one of the toolbars at the top of the UI that the TDP you selected in the New Project Wizard is listed in a dropdown box. In fact, this is not a reference to the TDP, it is a reference to the Configuration whose base TDP was the one you selected in the wizard - in the case of this tutorial, it is a TDP supporting JDK 1.3.1 or 1.4.0. (Configurations are initially named after their base TDP, but this name can be changed.) Should you have multiple configurations for the same project, use this dropdown box to select the active Configuration for execution.

Finally, to the right of the Configuration dropdown list is the **Build** button. This button is used to build your application for application nodes and the test harness for test nodes. The test harness consists of:

- source files needed to build the application of interest
- stubs
- a test driver

The downward-facing arrow associated with the Build button lets the user decide from which point the build process should initiate and what runtime analysis

features should be used. The runtime analysis features do not have to be used at the same time; this Build options window provides a quick and simple method for deselecting undesired runtime analysis features immediately prior to execution of the build process.

Armed with this knowledge, proceed to Exercise Two.

Exercise Two

In this exercise you will:

- build and execute the UMTS base station application
- manually interact with the UMTS base station application
- view the runtime analysis reports derived from your interaction

Building and Executing the Application

When performing runtime analysis, your source code must be instrumented. Instrumentation, by default, is enabled for all four runtime analysis features - that is, for memory profiling, performance profiling, code coverage analysis and runtime tracing. All four features are turned on by default.

To build and execute the application:

1. In order to instrument, compile and execute the UMTS base station application in preparation for runtime analysis, simply ensure the **BaseStation** application node is selected on the **Project Browser** tab of the Project Explorer window, and then click the **Build** button.
   Do so now.

   Note More information about the source code insertion technology can be found in the **User Guide**, in the chapter **Product Overview->Source Code Insertion**.

2. Notice that in the Output Window at the bottom of the screen, on the **Build** tab, you can see the instrumentation and compilation phases of the build process as they occur. A double-click on an error listed within any of the Output Window tabs opens the relevant source code file to the appropriate line in the Test RealTime Text Editor.

3. The build process has completed, and the UMTS base station is running, when the UML-based sequence diagram generated by the runtime tracing feature appears. (More about this feature in a moment.)

4. Close the Project Explorer window on the right-hand side of the UI by clicking the **Close Window** button; do the same for the Output Window at the bottom of the UI.

Notice how the graphically displayed data in the **Runtime Trace** viewer dynamically grows - this is because the UMTS base station is being actively monitored. The UMTS base station endlessly searches for mobile phones requesting registration; the Runtime Trace viewer reflects this endless loop. If you wish, use the **Pause** toolbar button to stop the dynamic trace for a moment (the trace is still being recorded, just no longer displayed in real time). In addition, use the **Zoom In** and **Zoom Out** buttons on the toolbar to get a better view of the graphical display (or right-click-hold within the Runtime Trace viewer and select the **Zoom In** or **Zoom Out** options). Undo the Pause when you're ready to proceed.

You'll look at the Runtime Trace viewer in more detail later. Of primary importance right now is interaction with the UMTS base station. You'll do this by using the mobile phone simulator mentioned earlier in the Overview section of this tutorial. Through this manual interaction you will expose careless memory usage, performance bottlenecks, incomplete code coverage, and dynamic runtime sequencing.

Interacting with the Application

To run the application:

1. Start the mobile phone by running the provided mobile phone executable built for your operating system. The mobile phone executable is located within the Test RealTime installation folder in the folder **\examples\BaseStation_C\MobilePhone**. The name of the executable depends on your operating system:

    - Windows: MobilePhone.exe
    - Solaris: MobilePhone.SunOS
    - Linux Suse: MobilePhone.Linux
    - Linux Redhat: MobilePhone.Linux_Redhat
    - HP-UX: MobilePhone.HP-UX
    - AIX: MobilePhone.AIX

    A launcher shell script - **MobilePhone.sh** - is provided for the non-Windows platforms as well.

2. Click the mobile phone's On button ( ).

3. Wait for the mobile phone to connect to the UMTS base station (if you watched the Runtime Trace viewer closely, you would have noticed a display of all the internal method calls of the UMTS base station that occur when a phone attempts to register). The current system time should appear in the mobile phone window when connection has been established.

4. Once connected, dial the phone number **5550000**, then press the button to send this number to the UMTS base station (again, try to see the Runtime Trace viewer update).

5. Unfortunately, the party you are dialing is on the line so you'll find the phone is busy. Shut off the simulator by closing the mobile phone window via the button in its upper right corner.

   The UMTS base station is designed to shut off when a registered phone goes off line. Not a great idea for the real world, but it serves the Tutorial's purposes well. Alternatively, you could have just used the **Stop Build** button in the toolbar.

6. The UMTS base station has stopped running when the green execution light next to the execution timer - located beneath the Project Explorer window on the lower right-hand side of the UI - stops flashing ( ). Wait for it to stop flashing.

   Everything that occurred at the code level in the UMTS base station was monitored by all four runtime analysis features. Once the UMTS base station stopped (i.e. once the instrumented application stopped), all runtime analysis information was written to user accessible reports that are directly linked to the UMTS base station source code. In order to look at these reports:

7. Reopen the Project Explorer window by selecting the menu item **View->Other Windows->Project Window**

8. In the Project Explorer window, on the Project Browser tab, double-click the **BaseStation** application node. All four runtime analysis reports will open. (Alternatively, right-click the **BaseStation** application node and select **View Report->All**.)

9. Close the Project Explorer window to create room for the now-opened reports. You may also want to resize the left-hand window to gain additional room.

Understanding Runtime Tracing

The sequence diagram produced by the Runtime Tracing feature uses a notation taken from the Unified Modeling Language, thus it can be correctly referred to as a UML-based sequence diagram.

To view the UML sequence diagram report:

1. Select the **Runtime Trace** tab.

2. As you recall, the Runtime Trace viewer displayed all objects and all method calls involved in the execution of the UMTS base station code. Using the toolbar

buttons , zoom out from the tracing diagram until you can see at least four vertical bars.

3. Make sure you are looking at the top of the runtime tracing diagram using the slider bar on the right.
4. Right-click within the runtime tracing diagram and select **Hide Memory Usage Bar**. Repeat in order to select **Hide Coverage Bar** and **Hide Thread Bar**. You will return to these bars in a moment.

What you are looking at is a sequence diagram of all events that occurred during the execution of your code.

The vertical lines are referred to as lifelines. Each lifeline represents a Java object instance. The very first lifeline, represented by a stick figure, is considered the "world" - that is, the operating system. In this UMTS base station tracing diagram, the next lifeline to the right represents an object instance named **Obj0**, derived from the **UmtsServer** class.

Green lines are constructor calls, black lines are method calls, red lines are method returns, and blue lines are destructor calls. Hover the mouse over any method call to see the full text. Notice how every call and call return is time stamped.

Everything in the Runtime Trace viewer is hyperlinked to the monitored source code. For example, if you click on the **Obj0::UmtsServer** lifeline, the source file in which the UmtsServer class definition appears is opened for you, the relevant section highlighted. (Close the source file by right-clicking the tab of the Text Editor and selecting **Close**.) All function calls can be left-clicked as well in order to view the source code. Look at the very top of the **Obj0::UmtsServer** lifeline. It's "birth" consists of a **UmtsServer()** constructor. Left-click the constructor if you wish to view the steps that occur when an object of the UmtsServer class is instantiated.

Notice how the window on the left-hand side of the user interface - called the **Report Window** - contains a reference to all classes and class instances. Double-clicking any object referenced in this window will jump you to its birth in the Runtime Trace viewer. This window can also be used to filter the runtime tracing diagram; closing a node associated with a source file or class will collapse all of the associated lifelines into a single, consolidated lifeline.

Filters

Continue to look around the trace diagram. Can you locate the repetitive loop in which the UMTS base station looks for attempted mobile phone registration (it always starts with a call to the method **baseStation.LogServer.checkLog()**)? You can filter out this loop using a couple of methods. One is to simply hover the mouse over a method or function call you wish to filter, right-click-hold and select **Filter Message**. An alternative method would be to use a predefined filter. You will do both.

To use sequence diagram filters

1. Hover the mouse over any call of the **baseStation.LogServer.checkLog()** method, right-click-hold and select **Filter Message** - the function call should disappear from the entire trace.
2. Select the menu item **Runtime Trace->Filters** (you'll see the filter you just performed listed here)
   Click the **Import** button, browse to the installation folder and then the folder **\examples\BaseStation_Java**, and then **Open** the filter file **filters.tft**
3. Select **BaseStation Phone Search Filter** if necessary.
4. Click the **OK** button.

   The loop has been removed.
5. Using the Zoom Level dropdown list on the toolbar, select a level of 50%:

Memory Usage Bar

The **Memory Usage Bar** is a graphical representation of the amount of memory allocated by the monitored application at any moment represented within the runtime tracing diagram.

To use the Memory Usage bar:

1. Right-click-hold in the Runtime Trace viewer and select **Show Memory Usage Bar**.

   You can now see, along the left-hand side of the runtime tracing diagram, a red, vertical bar. The caption of the Memory Usage Bar indicates the maximum amount of allocated memory that occurred during execution, while the mouse tool tip can be used to discern the amount of allocated memory at any moment along the graph. (Depending on your JVM, you may also notice garbage collection, indicated by areas where there is a sudden drop in the number of allocated bytes.)

   This diagram can be used to expose memory intensive parts of your program that may in fact be needless churn that slows down overall execution time. You could trigger garbage collection immediately prior to suspect moments within your application, using the Runtime Trace viewer to help you decide where the garbage collection should occur, to study whether or not memory usage has become excessive. Note that this feature is specific to Java support.
2. Right-click-hold in the Runtime Trace viewer and select **Hide Memory Usage Bar**.

Coverage Bar

The **Coverage Bar** highlights, in synchronization with the runtime tracing diagram, the percentage of total code coverage achieved during execution of the monitored application. The Coverage Bar's caption states the overall percentage of code coverage achieved by the particular interaction presently displayed in the Runtime Trace viewer.

To use the Coverage bar:

1. Right-click-hold in the Runtime Trace viewer and select **Show Coverage Bar**.

   Scroll down the runtime tracing diagram; note how code coverage gradually increases until a steady state is achieved. This steady state is achieved following the moment at which the mobile phone has connected to the UMTS base station. Dialing the phone number increases code coverage a bit; shutting off the phone creates a last burst of code coverage up until the moment the UMTS base station is shut off. Can you locate where, on the runtime tracing diagram, the mobile phone simulator first connected to the UMTS base station? Note that the Coverage Bar is available for all supported languages.

2. Right-click-hold in the Runtime Trace viewer and select **Hide Coverage Bar.**

Thread Bar

The UMTS base station is actually a multi-threaded application; the Thread Bar graphically indicates the active thread at any given moment within the runtime tracing diagram.

To use the Thread bar:

1. Right-click-hold in the Runtime Trace viewer and select **Show Thread Bar**.

   Now you are looking at the **Thread Bar**. (Hovering your mouse over the Bar reveals the name of the active thread within a tool tip.) A left-click on the Thread Bar opens a threading window, detailing thread state changes throughout your application's execution. Pressing the Filter button in this detail window specifies the state of each thread within the region of the Thread Bar that was double-clicked. Note that this thread monitoring feature is also available for the C++ language.

2. Right-click-hold in the Runtime Trace viewer and select **Hide Thread Bar.**

   Not only can the runtime tracing feature capture standard function/method calls, but it can also capture thrown exceptions.

3. View the very bottom of the runtime tracing diagram using the slider bar.

Do you see the icons for the catch statement - catch exception ? The second **Catch Exception** statement is preceded by a diagonal **Throw Exception**. Why diagonal?

Because when the exception was thrown, prior to executing the Catch statement, the **UmtsException** constructor was called. Click various elements to view the source code involved in the thrown exception and thus decipher the sequence of events.

This exception occurred by design, but it is clear how the runtime tracing feature, through the power of UML, would be extremely useful if you have:

- inherited old or foreign code
- unexpected exceptions
- questions about whether what you designed is occurring in practice

And you are guaranteed the identical functionality for application execution on an embedded target.

Understanding Memory Profiling

The Memory Profile viewer displays a memory usage report for the application of interest.

To view a Memory Profile report:

1. Select the **Memory Profile** tab.
2. Select the menu item **Memory Profile->Hide/Show Data->Hide/Show Referenced Objects**.

The Report Window on the left-hand side of the UI displays a list containing each memory snapshot and the time at which they occurred; as you may recall, the TDP Configuration was updated so that a snapshot would occur immediately following each garbage collection. The Memory Profile tab contains a sortable table (i.e. sortable via a left-click on a column header) with the following information:

- **Method** - Each method that, when called, resulted in the instantiation of an object. A left-click on any method names brings you to the portion of source code in which this method has been defined.
- **Referenced Object Class** - If any method in the first column continues to reference an object at the time of the snapshot, the object is listed in this column. Of course, many objects are allocated and deallocated before a snapshot - in this case, the object allocation is recorded but the object reference is not.
- **Allocated Objects**- Total number of objects created by a method throughout execution of the monitored application.
- **Allocated Bytes** - Total number of bytes associated with the objects created by a method.

- **L + D Allocated Objects** - Total number of objects created by the "local" method and by any descendant methods - that is, by any method that was called as a result of the specified method.
- **L + D Allocated Bytes** - Total number of bytes associated with the objects created by the "local" method and by any descendant methods.

Note how this table is referred to as a "snapshot" at the very top. A user is able to predefine moments at which a memory snapshot should take place - this is done via Configuration Settings. At each snapshot, the JVMPI interface of the targeted JVM is queried and information about each individual method is acquired. For example, if you have designed a particular, cyclic portion of your code to deallocate all unnecessary memory prior to each iteration, set a snapshot to occur each time the cycle is entered. The Memory Profile report contains diff functionality - you will explore this capability later - that can tell you if additional memory remains allocated when the cycle is reentered.

Notice how easily this information has been acquired; no work was required on your part. A real advantage is that memory profiling can now be part of your regression test suite. Traditionally, if embedded developers looked for careless memory allocation/deallocation at all, it was done while using a debugger - a process that does not lend itself to automation and thus repeatability. The memory profiling feature lets you automate memory leak detection.

And again, the identical functionality can be used on either your host platform or on your embedded target.

Understanding Performance Profiling

The Performance Profile viewer displays the execution time for all methods executing within the application of interest, thereby allowing the user to uncover potential bottlenecks. First, the one or more methods requiring the most amount of time are displayed graphically in a pie chart. Up to six functions are displayed if each is individually responsible for more than 5% of total execution time. This is then followed by a sortable list of every method, with timing measurements displayed.

To view the Performance Profile report:

1. Select the **Performance Profile** tab.

   Notice how the function **checkLog()** was responsible for around 75% to 85% of the time spent processing information in the UMTS base station. By looking at the table, where times are listed in milliseconds, we can see that this function's average execution time was between 6 to 7 seconds (it will vary somewhat based on your machine) and that it has no descendents - i.e. it never calls and then awaits the return of other functions or methods, which explains why the **Function** time matches the **F+D** time. Is this to be expected? If you wished, you

could click on the function name in the table to jump to that function to see if its execution time can be reduced.

Each column can be used to sort the table - simply click on the column heading.

2. Click the column heading entitled **F+D Time**

   Interestingly, though **checkLog()** clearly uses the largest amount of execution time, it is not the "slowest" method when considering descendants. That distinction goes to **readMsg()**; though quick by itself, it's execution time when including descendants is the slowest of all. However, a quick investigation of the **readMsg()** function would reveal that this function calls - and that awaits the return of - **readString()**, which explains why the execution time of **readMsg()** takes longer than **readString()**.

Of course, since this is a multi-threaded application, it is possible for one function to reveal itself as the slowest performer while, overall, the monitored application is typically busy doing other things. This would explain why the runtime tracing diagram does not indicate monopolization of UMTS base station execution following a call to the **checkLog()** method (have a look; search for *checkLog* using the **Find** button from the toolbar), and thus why performance profiling is such a valuable supplement to code optimization.

As with the memory profiling feature, notice how easy it was to gather this information. Performance profiling can now also be part of your regression test suite.

Understanding Code Coverage

And finally, here you have the code coverage analysis report. The code coverage feature exposes the code coverage achieved either through manual interaction with the application of interest or via automated testing.

To view the code coverage report:

1. Select the **Code Coverage** tab.

   On the left hand side of the screen, in the Report Window, you see a reference to **Root** and then to all of the source files of the UMTS base station. **Root** is a global reference - that is, to overall coverage. For each individual source file, a small icon to the left indicates the level of coverage (green means covered, red means not covered).

   In the Code Coverage viewer, on the **Source** tab, a graphical summary of total coverage is presented in a bar chart - that is, information related to **Root**. Five levels of code coverage are accessible for Java, and those five levels are represented here. (Four more levels of coverage are accessible when working with the C language - up to and including Multiple Conditions/Modified Conditions.) Notice how, on the toolbar, there is a reference to these five possible coverage levels (**F**, **E**, **B**, **I** and **L** toolbar buttons).

2. Deselect the **L** toolbar button to disable Loops Code Coverage.

   Notice how the bar chart is updated.

3. Select Loops Code Coverage again by selecting the **L** button.

4. In the Report Window to the left, select the **HardwareMonitor.java** node.

   The **Source** tab now displays the source code located in the file **HardwareMonitor.java**. This code is colored to reflect the level of coverage achieved. Green means the code was covered, red means the code was not covered.

   Within the **run()** method you should see a **while** statement that is colored orange and sitting on a dotted underline. This is because the **while** statement was only partially covered.

5. Click on the orange **while** keyword in the **run()** method.

   As you can see, the **while** loop was only executed multiple times, not once or zero times. Why should you care? Well some certification agencies require that all three cases be covered for a **while** loop to be considered covered. If you don't care about this level of coverage, just deselect **Loops Code Coverage**:

6. Deselect the **L** toolbar button to disable Loops Code Coverage.

   Now the **while** loop is green. If you would like to add a comment to your code indicating how this loop is not covered by typical use of the mobile phone simulator, access the code by right-clicking the **while** statement and selecting **Edit Source**.

7. Select the **Rates** tab in the Code Coverage viewer

   The **Rates** tab is used to display the various coverage levels for

- the entire application
- each source file
- individual methods

   Click various nodes in the Report Window in order to browse the Rates tab. Note how a selection of the Root node gives you a summary of the entire application.

8. Select the menu item **File->Save Project**

Conclusion of Exercise Two

With virtually minimal effort, you have successfully instrumented your source code for all four runtime analysis features. Manual interaction (in your case, via a mobile phone simulator) was monitored, and the subsequent runtime analysis results were displayed for you graphically. Source code is immediately accessible from these

reports, so nothing prevents the developer from using the results to correct possible anomalies.

In addition, using the Test by Test option provided with each runtime analysis feature (introduced in the Further Work section for code coverage), you can easily discern the effectiveness of a test, ensuring maximal reuse without waste.

Your next step is to use the runtime analysis results to remove memory leaks, improve performance, and increase code coverage.

Exercise Three

In this exercise you will:

- Improve the UMTS base station code by correcting memory usage errors and by improving performance
- Increase code coverage
- Rerun the manual test to verify that the defects have been fixed

Using Memory Profiling to Remove Memory Leaks

Although, from one perspective, memory leaks are not possible with Java, failure to dereference objects will still, in the end, monopolize memory and potentially cause problems with your software. By using the diff functionality of the memory profiling feature, you will uncover poor object allocation/deallocation practice within the code.

To locate and detect memory problems:

1. Select the **Memory Profile** tab.
2. If you performed the Further Work section for memory profiling, skip this step; otherwise, select the menu item **Memory Profile->Show/Hide Data->Diff with Previous Snapshot**.

   Two new columns have appeared - **Referenced Objects Diff AUTO** and **Referenced Bytes Diff AUTO**. These columns contain a diff between each snapshot and the previous snapshot for every listed method; the word "referenced" refers to those objects for which a reference exists following a snapshot. It is also possible for the user to diff any two selected snapshots; this custom diff would be labeled USER to differentiate it from the AUTO diff you will be studying. (Note that a blank cell in any diff column means the object did not exist in the previous snapshot.)

   Recall that the snapshots for this Tutorial occurred immediately after each garbage collection. This means that any object references uncovered by a diff are suspicious; referenced objects can not be deallocated by the garbage collector.

3. Sort by the column **Referenced Objects Diff AUTO** by clicking on the column header.

4. Search the various snapshots for a method that recurrently is responsible for continuously referenced objects.

   Have you noticed that the **GetChannels()** method reappears throughout? Perhaps you should look at the code to understand why this method is so often associated with continuously referenced objects.

5. Left-click any reference to the **GetChannels()** method in the first column of the table.

6. Scroll the Text Editor until you can view the **GetChannels()** method.

   Inspection of the **GetChannels()** function reveals that it creates ten new channels each time it is called - which means ten channels should be removed (i.e. dereferenced) elsewhere in the code. This dereferencing is the responsibility of the **ReleaseChannels()** method, located right below the definition of **GetChannels()**, and the **for** statement of this method has been improperly written. Currently, the **ReleaseChannels()** method only dereferences nine objects. You need to fix the code.

7. Modify the **for** statement of the **ReleaseChannels()** method as follows (you are adding an =):

   Change the code from

```
for (i=0;i<10;i++)
```

   to

```
for (i=0;i<=10;i++)
```

8. Select the menu item **File->Save**.

9. Right-click the tab for the source file you have just modified and select **Close**.

This should fix the problem. Before redoing you manual test to verify if the memory error was fixed, move on to the Performance Profile viewer and see if you can streamline the performance of the UMTS base station code.

As for the other methods that appear to continuously reference objects following garbage collection - are they also leaking? That's for you to figure out!

Using Performance Profiling to Improve Performance

Now you will use information in the Performance Profile viewer to determine if you can improve performance in the UMTS base station code.

To locate and improve performance issues:

1. Select the **Performance Profile** tab.

2. Within the table, left-click the column title **Function Time** in order to sort the table by this column.

   For this exercise you have sorted by the Function Time - that is, you're looking at functions that take the longest time, overall, to execute. This is isn't the only potential type of bottleneck in an application - for example, perhaps it is the number of times one function calls its descendants that is the problem - but for this exercise, you will look here.

   As the developer of this UMTS base station, you would know that the method **read_string()** takes a fair amount of time to execute - so you won't look here first (although feel free to have a look if you wish). Instead look at the second function in the table.

3. Select the link for the method **checkLog()**.

   A quick look at the source code shows you that the developer has added an inexplicable loop - perhaps a dummy function to act as a "time-waster". Simply comment out the line.

4. Change the code from
   ```
   for (x=1,y=100000;x<=100000;x++) y=y/x;
   ```
   to
   ```
   // for (x=1,y=100000;x<=100000;x++) y=y/x;
   ```

5. Select the menu item **File->Save**.

6. Right-click the tab for the source file you have just modified and select **Close**.

You have now eliminated a loop that was adding significant execution time to the **checkLog()** method.

Using Code Coverage Analysis to Improve Code Coverage

You will now use the information gathered by the code coverage analysis feature to modify the manual test in such a way as to improve code coverage.

To improve code coverage:

1. Select the **Code Coverage** tab.

2. If necessary, select the **Source** tab of the Code Coverage viewer.

3. In the Report Window on the left-hand side of the screen, open the **UmtsConnection.java** node, then open the **baseStation.UmtsConnection** child node, and then select the **run()** child node.

4. Drag the slider bar down slightly until you see the line:
   ```
   case_connected:
   ```

Notice how the **if** statement was never true - only the **else** block is green, but the **if** block is red. In order to improve coverage of this **if** statement, you need to make the boolean expression evaluate to true.

According to this code, the **if** expression would evaluate to true if mobile phone sends the phone number **5550001**. You should do that.

You will now rerun the UMTS base station executable, restart the mobile phone simulator, and dial this new phone number. When you have finished, you will check the memory profiling, performance profiling, and code coverage analysis reports to see if you have improved matters.

Redoing the Manual Test

You have changed some source code, so some of the UMTS base station code will have to be rebuilt. The integrated build process of Test RealTime is aware of these changes, so you do not have to specify the particular files that have been modified.

To rebuild the application:

1. Select the menu item **View->Other Windows->Project Window**.
2. From the **Window** menu, select **Close All**.
3. Select the **Project Browser** tab in the Project Explorer window that has now appeared on the right-hand side of the UI.
4. Right-click the **BaseStation** application node and select **Build**
   (If you select **Rebuild**, all files will be rebuilt. **Build** simply rebuilds those files that have been changed. If no files had been changed, you could have just selected **Execute BaseStation**.)
5. Once the UMTS base station is running (indicated by the appearance of the Runtime Trace viewer), run the mobile phone simulator as before. (Note how the runtime trace appears to stop - this is because the filter is still applied and thus the recurrent loop is not visible.)
6. Click the mobile phone's On button ().
7. Wait for the mobile phone to connect to the UMTS base station (if you watch the dynamic trace closely, you'll notice a display of all the actions that occur when a phone registers with the server). The time should appear in the mobile phone window.
8. Once connected, dial the phone number **5550001**, then press the button again to send this number to the UMTS base station (again, watch the dynamic trace update).

9. Success! You have connected to the intended party. Stop right here to see the results of your work. Close the mobile phone window by clicking the X button on the right side of its window caption. As you may recall, this action will shut down the UMTS base station as well.

10. The UMTS base station has stopped running when the green execution light next to the execution timer - located beneath the Project Explorer window on the lower right-hand side of the UI - stops flashing (00:00:19). Wait for it to stop flashing.

11. In the Project Explorer window, on the **Project Browser** tab, double-click the **BaseStation** application node. All four runtime analysis reports will open with refreshed information. (Alternatively, right-click the **BaseStation** node and select **View Report->All**.)

12. Close the Project Explorer window to the right and the Output Window at the bottom.

So have you improved your code and increased code coverage?

Verifying Success

Was the memory leak eliminated?

To check that the memory leak was fixed:

1. Select the **Memory Profile** tab.

2. In the Report Window on the left-hand side of the UI, left-click the first snapshot for **Test #2**.

3. Select the column header for **Reference Bytes Diff AUTO**, then select the column header for **Reference Objects Diff AUTO**.

4. Scroll down and study each of the snapshots for Test #2 - is the **GetChannels()** method still responsible for referenced objects?

You successfully eliminated the memory leak. Have you improved performance?

To check that performance was improved:

1. Select the **Performance Profile** tab.

2. Select the menu option **Performance Profile->Test by Test**

3. In the Report Window on the left-hand side of the screen, left-click the node labeled **Test #1** in order to deselect it.

4. Sort the table by **Function Time** if it is not sorted by this value already.

5. Do you see the function **checkLog()**?

You successfully improved performance. Was code coverage improved?

To check that code coverage was improved:

1. Select the **Code Coverage** tab.
2. In the Report Window on the left-hand side of the screen, open the node for **UmtsConnection.java**, open the **baseStation.UmtsConnection** child node, then left-click the **run()** node.
3. Select the menu option **Code Coverage->Test by Test.**
4. Scroll down until you can see the **if** statement for which you have attempted to force an evaluation of true - did you? Has code coverage been improved?

   You successfully improved code coverage. Note, by the way, that you can discern what this second manual interaction has gained you in terms of code coverage.
5. With your mouse anywhere within the **Source** tab of the **Code Coverage** viewer, right-click and select **CrossRef**
6. Scroll the Code Coverage viewer to expose the line of code that has been newly covered and then left-click it:
   ```
   message.setCommand(UmtsMsg.ACCEPTED);
   ```
   Notice that only **Test #2** is mentioned. However, what tests are listed for the **if** statement itself?
7. Left-click the line
   ```
   if (message.getPhoneNumber().equals("5550001"))
   ```
   Both **Test #1** and **Test #2** are listed. As further proof, do the following.
8. With your mouse anywhere on the **Source** tab of the **Code Coverage** viewer, right-click and deselect **Cross Reference**
9. In the Report Window, on the left-hand side of the screen, open the **Tests** node and deselect the checkbox next to **Test #2**.

Since you have deselected **Test #2**, all you are left with is the code coverage that has resulted from running **Test #1**, and **Test #1** never forced the **if** statement to evaluate to true. Thus the newly covered code has become red again - in other words, unevaluated.

Conclusion of Exercise Three

After correcting the UMTS base station code directly in the Test RealTime Text Editor, you simply rebuilt your application and used the mobile phone simulator to initiate further interaction. A second look at the runtime analysis reports validated

the accuracy of your changes. Consider the speed with which you could perform these monitoring activities once you are familiar with the user interface...

Conclusion

Conclusion - with a Word about Process

Automated memory profiling, performance profiling, runtime tracing, and code coverage analysis - no less important in the embedded world than elsewhere in software. So why is it done less often? Why is it so much harder to find solutions for the embedded market? It is because embedded software development involves special restrictions that make these functions more difficult to achieve, particularly when speaking of target-based execution:

- strong real-time timing constraints
- low memory footprints
- multiple RTOS/chip vendors
- limited host-target connectivity
- complicated test harness creation for target-hosted execution
- etc.

Rational Test RealTime and Rational PurifyPlus RealTime have been built expressly with the embedded developer in mind, so all of the above complications have been overcome. Nothing stands between you and the use of a full complement of runtime analysis features in both your native and target environment.

So use them! It should be automatic - part of all your regression testing efforts (discussed in greater detail in the Tutorial conclusion). As you have seen, these functions are only a mouse-click away so there is absolutely no drain on your time.

You may be concerned about the instrumentation - "But I don't want my final product to be an instrumented application. Doesn't it have to be if I'm testing instrumented code?" No, it does not have to be:

- Using the code coverage feature, generate a series of tests that cover 100% of your code
- Instrument that code for full runtime analysis
- Uncover and address all reliability errors as you test (e.g. poor memory management, overly slow functions, improper function flow, untested code)
- Now uninstrument your code - that is, simply shut off all runtime analysis features and rebuild your application

- Run your regression suite of tests once more, this time looking only for functional errors
- No errors? Time to ship

Make it part of your development process, just another step before you check in code for the night. Rational Test RealTime and Rational PurifyPlus RealTime simplify runtime analysis to such an extent that there is no longer a reason not to do it.

Test RealTime users may now proceed to the next lesson: Automated Component Testing with Test RealTime

Component Testing

C, C++ and Ada Track

Automated Component Testing

You have just completed a variety of what are, in essence, reliability tests on the UMTS base station. In other words, you have verified the absence of memory leaks, the optimization of performance, the sensibility of process flow, and the completeness of your testing.

But does the base station code do what it is designed to do? And wouldn't it be useful to create automated tests rather than rely solely on manual interaction?

Runtime analysis completes the picture, but functional testing of your code gets to the heart of the matter - that is, will your application generate the results it was designed to achieve. Rational Test RealTime provides you with three automated testing features to address your testing needs.

- **Component Testing for C:** For use with C functions and Ada functions and procedures.
- Component Testing for C++: For use with C++ classes.
- **System Testing for C:** For use with C threads, tasks, processes, and nodes.

You'll start with a look at the component testing feature for C and Ada.

Component Testing for C and Ada

Component Testing for C and Ada

Component Testing for C and Ada

When speaking of C and Ada programs, the term "component testing" - also sometimes referred to as "unit testing" - applies to the testing of C functions and/or Ada functions and procedures. A function/procedure is passed a possible set of inputs, and the output for each set is validated to ensure accuracy. This can be done with either a single function/procedure, a group of unrelated functions/procedures,

or with a sequential group of functions - i.e. one function calling another, verifying the overall or integrated, result.

Sounds simple but, unfortunately, in the embedded world its practice can be quite difficult. Why?

- What if the function you wish to test relies on the existence of other functions that have not yet been coded?
- How will you call the function-under-test in the first place?
- How will you create and maintain a variety of potential inputs and associated outputs - that is, how will you make data-driven testing manageable?
- What kind of effort and knowledge is required to run the test on your target architecture - that is, in the intended, native environment?

The component testing feature of Rational Test RealTime for the C and Ada languages provides a means for automating and verifying the above concerns. Through source code analysis:

- Yet-to-be coded functions and procedures are "stubbed"; in other words, these functions are created for you
- A test driver is generated to facilitate communication between your running code and the test
- A test harness, independent of your test, is constructed to ensure adoption of your target architecture and thus enabling in-situ test execution

Plus, thanks to a powerful test script API:

- Define stub responses to varied input generated by the function(s) under test
- Enable highly detailed data definitions for data-driven testing

With the assistance of the Target Deployment technology, the end result is an extensible, flexible, automated testing tool for component and integration testing.

Testing Exercises

Exercise One

In this exercise you will:

- uncover a part of the UMTS base station C code that requires further testing
- create a new activity in which you build a unit test

If you need a refresher about the application you will be using during this tutorial, look here; otherwise, please proceed.

Using Code Coverage to Find Untested Code

During the code coverage review, you surely noticed a fair amount of untested code. For this tutorial, you will focus on one particular section.

To select a particular section of code:

1. First, select the menu item **File->Save Project**
2. If necessary, select the **Code Coverage** tab.
3. In the Report Window on the left-hand side of the screen, open the **UmtsCode.c** node and then left-click the **code_int()** function

This function contains two partially covered **while** statements - focus on the second **while** statement (you may need to scroll down a bit):

```
while (x!=0)
```

A left-click on the **while** statement shows you that of the three possible types of coverage, only one type was achieved - 2 or more loops. You should really create one or more tests to appropriately cover this **while** statement - but first, perhaps you should spend a little more time understanding what the **code_int** function does.

Code Review

It doesn't make much sense to test a function without understanding it first.

To locate the source code:

1. Right-click-hold the mouse over the **while** statement you have just inspected and select **Edit Source**

The objective of the **code_int** function is to place a given integer at the end of a buffer with the following format:

```
I[length of number][lowest order digit]....[highest order digit]
```

Thus the number **1234** would be stored at the end of the buffer as **I44321**.

That's about it. Have a look at the code you're about to test if you wish. Once ready, proceed to the next step in which you will build your test.

Adding a New Configuration to Your Test RealTime Project

Since you will be testing a C function, you should use a Target Deployment Port for the C language. Rather than modifying the existing Configuration, you will now create a new one whose base TDP is a TDP for the C language.

To add a new configuration to a project:

1. Select the menu item **Project->Configuration**

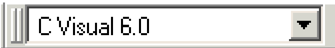
Note, in the **Configurations** window that has just appeared, the existence of the Configuration you have been working with up to now.

2. Click the **New...** button
3. In the dropdown list of the **New Configuration** window, choose either **C Visual 6.0** if you have Microsoft Visual C++ 6.0 installed, or, if you are using GNU/native compilers, select the item appropriate for your operating system:

- Windows - C Gnu 2.95.3-5 (mingw)
- Solaris - C Solaris - SC5.1
- Linux - C Linux - Gnu 2.95.2
- HP-UX - C HP-UX compiler
- AIX - C AIX - IBM C Compilers

Do not be concerned if the version of the GNU compiler you have installed does not match the version mentioned for the TDP. The differences are not relevant for this tutorial and thus other versions are supported equally as well.

4. Click the **OK** button to close the **New Configurations** window.
5. Click the **Close** button to close the **Configurations** window.
6. In the toolbar dropdown list that mentions the current Configuration - named after the C++ TDP you selected at the beginning of the tutorial - select the new Configuration, based on the C TDP you just added to the project. The following is what the box should look like if you're using the Microsoft Visual C/C++ TDPs:



Now the C language TDP will be used by any new node generated via the Activity Wizard.

Creating a C and Ada Component Test

Using the Component Testing Wizard, you will now create a test for all functions in the file **UmtsCode.c** - including the **code_int** function that contains the **while** statement for which you wish to improve coverage.

To create a component test:

1. If the Project Explorer window is not visible, from the **View** menu, select **Other Windows** and **Project Window**.
2. From the **Window** menu, select **Close All**.
3. Click the toolbar **Start** button to relaunch the Start Page.

4. Select the **Activities** link on the left-hand side of the Start Page.
5. Select the **Component Testing** link that has now appeared.
6. In the **Application Files** window, notice how all the C source files of your development project are already visible.

   Select the **Compute static metrics** option. This allows the measurement of code complexity from which you can prioritize your test campaign.

   Click the **Next** button.
7. In the **Components Under Test** window, you are asked to specify which functions you would like to test. There are a variety of ways for making this decision. One method is to use the static metrics that have just been automatically calculated. Certain measurements of code complexity are listed for you:

- V(g) - Also called the Cyclomatic Number, it is a measure of the complexity of a function that is correlated with difficulty in testing. The standard value is between 1 and 10. A value of 1 means the code has no branching. A function's cyclomatic complexity should not exceed 10
- Statements - Total number of statements in a function.
- Nested Level - Statement nesting level.

   Sorting by any of these metrics columns - by left-clicking a column header - lets you prioritize your test selection, choosing the more complicated functions first.

   Additional metric information can be viewed by selecting the **Metrics Diagram** button on the lower right-hand side of the screen. Selection of this button opens a graph enabling visualization of two, selected static metrics graphed against one another. Select a data point in this graph to indicate your desire to test the associated functions.

   For this Tutorial, your test selection is based on the desire to increase code coverage, so the static metrics do not affect your decision. You need to test the **code_int** function. However, to help you get a better understanding of how the component testing feature of Test RealTime works, you should select all functions in the file UmtsCode.c.

8. Left-click the box to the left of every function in the source file UmtsCode.c (there are five functions in total).
9. Click the **Next** button.

   In the **Test Script Generation Settings** window, you are asked to make two decisions

- If you've selected more than one function to test, do you want all functions to be part of the same test script (Single Mode) or do you want each function to be assigned to its own test script (Multiple Mode). A single test script would be easier to manage, but multiple test scripts let you provide custom Configuration settings to each test.
- Do you want Test RealTime to make some basic assumptions about test harness and test stub generation? If so, use Typical Mode; if not, use Expert Mode.

10. Type **UmtsCode** in the **Test Name** field - that is, name the test node after the source file whose functions you will be testing. Leave the default selections. You will be creating a single test script that automatically stubs all referenced but undefined functions. Click the **Next** button.
11. You should now be viewing the **Summary** window. Click the **Next** button.

    The Component Testing Wizard now analyzes the source code in **UmtsCode.c** and creates a test for every function within it.
12. When test script generation has completed, click the **Finish** button.

In the Project Browser tab of the Project Explorer window on the right-hand side of the screen, you should now see a component test node named **UmtsCode**.

Conclusion of Exercise One

The advantages of automated testing is that it enables regression testing - that is, it ensures nothing regresses. Just because code appeared to be functional in Build X, doesn't mean that code will continue to be functional in Build X+1.

Few would dispute the usefulness of component testing, but many would claim there is not enough time to do it. Every effort has been made to simplify this process in Rational Test RealTime so that you can simply focus on making good tests, getting readable results, and making quality code.

Exercise Two

In this exercise you will:

- review the autogenerated component test
- improve the autogenerated component test
- execute the component test

The Autogenerated Component Test for C and Ada

The Component Testing Wizard analyzed the file **UmtsCode.c** and produced a test script called **UmtsCode.ptu**. What does this test do?

To edit the generated .ptu script:

1. In the **Project Browser** tab on the right-hand side of the screen, open the file **UmtsCode.ptu** by double-clicking it.
2. Maximize the test script window that has just opened, closing the lower Output Window to free up some additional space.
3. Click the **Asset Browser** tab on the right-hand side of the screen and select the **By File** sort method.

   On the **Asset Browser** tab you now see each of the five UmtsCode.c functions listed as a child of the test script **UmtsCode.ptu**. Each requires its own test; all test scripts are stored in the .ptu file. Back on the Project Browser tab, you'll notice that the .ptu file is associated with the source file upon which it was based. The idea is that when you build the UmtsCode component testing node, you are actually building a test harness comprised of the .ptu file, the original source file, the referenced header file and any stubs required for the simulation of as yet undeveloped code. The build process and test execution, as you recall, is managed by the information stored in a Configuration which, in turn, is based on the information stored in a Target Deployment Port.

   Component testing scripts for C and Ada are written with a compiler-independent test script API. For detailed information about the script layout, take advantage of the **Test RealTime Reference Guide** accessible via the **Help** menu. For the tutorial, only critical script elements will be pointed out.
4. In the **Asset Brower** tab, double-click the node **code_int** (child node of UmtsCode.ptu).

   Of particular note are the **Service** blocks in a test script. Each function in the file under test is assigned its own **Service** block. Each **Service** block can consist of one or more **Test** blocks. Each **Test** block consists of data-driven calls to the function under test.

   Here you see the **Service** block for the **UmtsCode.c** function **code_int**. This is then followed by native C language calls (indicated by the # symbol) used to declare the variables **x** and **buffer[200]** that are passed to the function **code_int**, the function containing the **while** statement for which we intend to improve code coverage. As a reminder, here is the declaration for **code_int**:

   ```
   void code_int(int x,char *buffer)
   ```

   The variable declarations are followed by an **Environment** block. The **Environment** block is used to define input (called **init** - i.e. initial) and output (called **ev** - i.e. expected value) values for the variables passed to the function under test. In the **Environment** block for the **code_int Service** block, **x** is initialized to 0 and has an expected value of **init** - that is, a value of 0, the initial value. **buffer** is initialized to nothing - which means each of its 200 array elements are set to 0 - and it has an expected value of **init** as well.

The **Test** block for **code_int** consists of a call to this function. Have you noticed that there is no mention of a return value? Since **code_int** returns **void**, nothing is returned - there is no return value to check.

5. In the **Asset Browser** tab on the right-hand side of the screen, open the **decode_int** node and then double-click on the icon for test **1**.

Look at the **Test** block for the function **decode_int** - in this case, a return value is expected - referred to as **#ret_decode_int**. Notice how the **Environment** block for the **decode_int** function includes an expected value for **#ret_decode_int**.

You now understand the essence of Rational Test RealTime component testing test script for C and Ada.

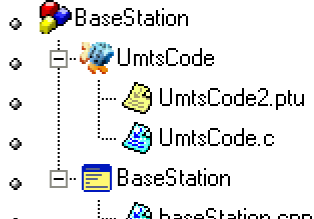
For the purposes of performing useful work, the test script needs to be more detailed than it is immediately following generation. You need to create good tests that supply relevant input values and then verify appropriate output values. Rather than writing it yourself, a revised test has been created for you.

A Customized Component Test

A customized component test script has been created for you. This test will be used to test the functions within **UmtsCode.c** - in particular, the function **code_int**, which contains the **while** statement of interest.

To customize the test:

1. Select the menu item **Window->Close All**
2. Select the **Project Browser** tab on the right-hand side of the screen, select the **UmtsCode.ptu** node (child of the **UmtsCode** component testing node), and then select the menu item **Edit->Delete**.
3. Right-click the **UmtsCode** component testing node and select **Add Child->Files...**
4. In the **Files of Type** dropdown box, select the **C and Ada Test Scripts** option, then browse to the Test RealTime installation folder and **Open** the file **\examples\BaseStation_C\tests\UmtsCode2.ptu**
5. After this new test script is analyzed by Test RealTime, your screen should appear as follows:



6. Double-click the node **UmtsCode2.ptu**
7. Maximize the test script window.
8. Bring the **code_int** test blocks for UmtsCode2.ptu into view using the **Asset Browser** tab. (The **Asset Browser** tab continues to reference the original test script - **UmtsCode.ptu** - because it still exists on your machine - it is simply no longer referenced by any tests.)
9. As you can see, two **Test** blocks are now part of the **code_int Service** block. In the first **Test** block the initial value of **x** has been set to **3** and the expected value for **buffer** has been set to **I13**. In the second **Test** block, the initial value of **x** has been set to **34** and the expected value for **buffer** has been set to **I243**. These expected values should make sense based on the function review you performed back in Exercise One.

Running a Component Test for C and Ada

Running a component test is as simple as it was to build and execute the UMTS base station used in the runtime analysis exercises.

To execute the test:

1. From the **File** menu, select **Save Project**.
2. From the **Window** menu, select **Close All**
3. On the **Project Browser** tab, select the **UmtsCode** component testing node (the parent node of the **UmtsCode2.ptu** and **UmtsCode.c** nodes) and then press the **Build** toolbar button.
4. The test is executed as part of the build process - you will know the test has finished executing when the green execution light on the lower-right of the UI stops flashing.

   You may have forgotten that the runtime analysis tools are still selected in the Build options; the file under test - **UmtsCode.c** - was instrumented for the memory profiling, performance profiling, code coverage analysis and runtime tracing features of Test RealTime, which explains why the Runtime Trace viewer appears during the run. Notice how this feature tracked all of the calls made to functions in UmtsCode.c. Each call is a test in the component testing test script that just executed.
5. In the **Project Browser** tab on the right-hand side of the screen, double-click the **UmtsCode** component testing node in order to open the test report and all of the runtime analysis reports.

What is the result of your tests? Did you improve coverage on the **while** statement? That is the subject of the next exercise.

Conclusion of Exercise Two

The component testing test scripting language for C and Ada gives you enormous data-driven testing power with minimal effort. This compiler-independent language lets you build tests that can be used with any embedded target, so you'll never have to change your tests when the architecture you're writing for changes.

As for test script execution, this is accomplished through the Test RealTime interface regardless of the target. The Target Deployment Port takes care of everything; there is no distraction from the task at hand - making quality tests and then fixing problems as they are exposed.

Exercise Three

In this exercise you will:

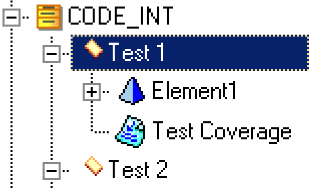
- analyze the results of the improved component test
- continue to increase code coverage
- repair the uncovered defect
- rerun your test to verify that the defect has been fixed

The C and Ada Component Test Report

The component testing report summarizes all of the test results. It is hyperlinked to the test script (the **.ptu** file) and can be browsed using the **Report Browser** on the left-hand side of the screen..

6. Close the **Project Explorer** window on the right-hand side of the screen as well as the **Output Window** at the bottom of the screen to free up space.
7. Select the **Test Report** tab to ensure the component testing report is active, and then maximize this window

   At the top of the report is an overall summary of test execution. Notice the **Passed** and **Failed** items - all eight tests in **UmtsCode2.ptu** passed. Good news.
8. In the Report Window on the left-hand side of the screen, double-click the node **Test 1** (a child node of the node **CODE_INT**:



Looking at the component testing report, you can see:

- General test information
- Initial, expected, and obtained values for all variables involved in a test
- Code coverage information

Have a look around if you wish. Your next concern should be whether code coverage on the **while** statement in the **code_int** function has been improved.

Checking the Code Coverage Report

Has code coverage been improved by running the unit test?

To analyze code coverage:

1. Select the **Code Coverage** tab.
2. In the Report Window on the left-hand side of the screen, open the **UmtsCode.c** node and then select the **code_int** node.
3. If necessary, scroll through the Code Coverage viewer **Source** window until the second **while** statement is visible.
4. Left-click this second **while** statement. You should see:

```
while
{
 *ptr
 x=x
 ptr+
 len++;
```

```
loop branches :
 0 loop
 1 loop
 2 loops or more
```

Code coverage has been improved somewhat, but the **while** statement has yet to be executed 0 times. To do this, you will have to create a new test. It would be preferable to do as little work as possible to create this new test. What other tests have forced the **while** statement to execute?

5. Select the menu item **Coverage->Test by Test**
6. With the mouse anywhere within the **Source** window of the **Code Coverage** viewer, right-click-hold and select **CrossRef**
7. Click any part of the line

```
while (x!=0)
```

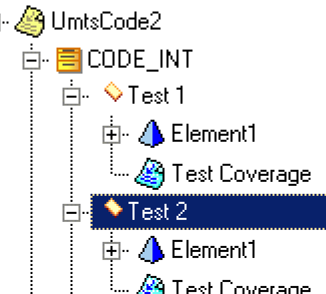
Perhaps not surprisingly, the two **code_int** tests covered this **while** statement. All you need to do is copy one of the two tests but make sure **x** equals 0 (i.e. when x is equal to 0, you will be achieving the highest level of coverage on this **while** statement).

Updating and Running the Component Test

Through reuse of existing test assets, your testing effort can be significantly reduced.

To reuse test elements:

1. Select the **Report Viewer** tab.
2. In the Report Window on the left-hand side of the screen, double-click the node **Test 2**, which is a child node of the node **CODE_INT**:



3. On the Test Report tab, left-click the green section header *1.2.3 ·Test 2*, located at the top of the screen.

   You are now looking at the code for the second of the two **code_int** tests. Since the objective is to execute the **while** statement where **x** has a value of **0**, reuse this second test block but assign **x** an initial value of **0** and buffer an expected value of - what? A value of I10.

4. In the Text Editor, copy all of the lines between **Test 2** and **End Test -- Test 2**, including these two lines:

```
TEST 2
FAMILY nominal
  ELEMENT
    VAR x,         init = 34      ev = init
    VAR buffer,    init = "",     ev = "I243"
    #code_int(x,buffer);
  END ELEMENT
END TEST -- TEST 2
```

5. Paste these lines immediately below the last line copied, and then rename the Test block to Test 3. It should look like the following:

```
END TEST -- TEST 2

TEST 3
FAMILY nominal
  ELEMENT
    VAR x,         init = 34      ev = init
    VAR buffer,    init = "",     ev = "I243"
    #code_int(x,buffer);
  END ELEMENT
END TEST -- TEST 3
```

6. Change the initial value of **x** to 0 and change the expected value of **buffer** to **I10**.

```
TEST 3
FAMILY nominal
  ELEMENT
    VAR x,          init = 0       ev = init
    VAR buffer,     init = "",      ev = "I10"
    #code_int(x,buffer);
  END ELEMENT
END TEST -- TEST 3
```

7. From the **File** menu, select **Save** to save your changes to the Unit Testing test script.
8. From the **View** window, select **Other Windows** and **Project Window**.
9. From the **Window** menu, select **Close All**.
10. In the **Project Browser** tab on the right-hand side of the screen, left-click the **UmtsCode** component testing node and then click the **Build** toolbar button.
11. The test has finished executing when the green execution light on the lower right of the UI stops flashing.

You should have now achieved proper code coverage. But were you looking at the Output Window? Why was there a warning?

Repairing a Defect

Unfortunately (or fortunately, depending upon how you look at it), increased code coverage can expose defects.

To fix a defect:

1. In the **Project Browser** tab on the right-hand side of the screen, right-click the **UmtsCode** component testing node and then select **View Report->Code Coverage**.
2. Maximize the **Code Coverage** report. Close the **Output Window** on the bottom of the UI
3. View the **code_int** code in the **Source** window of the **Code Coverage** viewer (using the **Report** tab, and then look at the **while** statement for which you have been trying to improve code coverage.

   The **while** statement is green, which means it is now fully covered (left-click it if you wish to be sure). Should you check the component testing report? Is it safe to assume that no defect has been uncovered by your effort to increase coverage?
4. In the **Project Browser** tab on the right-hand side of the screen, right-click the **UmtsCode** component testing node and then select **View Report->Test**

A failure is reported in the component testing report, so the effort to improve coverage has resulted in the discovery of a new defect. The **Report Window** on the left-hand side of the screen flags this error nicely.

5. In the **Report Window**, select the **Element1** node that has a **Failed** symbol to its left.

   Given **x** equal to **0**, the **code_int** function is supposed to assign **buffer** a value of **I10**. However, this did not happen. **buffer** has a value of **I0** - a defect.

6. Select the **Code Coverage** tab.

7. Hover the mouse over the **while** statement for which you have been trying to improve coverage, then right-click and select **Edit Source**.

   If you continue to read this section, you will be told what is wrong and how to fix the problem. Feeling adventurous? Don't read on and see if you can solve the problem yourself.

   When you're ready, just go to the next section in Exercise Three entitled Verifying the Success of Your Repairs.

   The problem is that because the **while** statement never handles **x** with a value of **0**:

   - the len variable - which contains the length of the number represented by x - is never increased beyond its initial value of 0

   - the value of x is never written to the buffer

   So a buffer value of **I0** reflects a length of **0** and an absence of the value of **x**. You need to take care of the special case in which **x** equals **0**.

8. Immediately before the **while** statement for which you have been attempting to improve coverage, add the following line:
   ```
   if (x==0) {*ptr='0';len++;ptr++;}
   ```

9. From the **File** menu, select **Save**.

This should fix the problem.

Verifying the Success of Your Repairs

As you have now learned, tests always need to be rerun and reports should always be checked.

To validate the repair:

1. From the **Window** menu, select **Close All**.

2. In the **Project Browser** tab on the right-hand side of the screen, left-click the **UmtsCode** component testing node and then click the **Build** toolbar button.

3. The test has finished executing when the green execution light on the lower-right of the UI stops flashing.
4. Double-click the **UmtsCode** component testing node to view all of the reports.
5. Select the **Report Viewer** tab.

   When looking at the **Report Window** to the left, you will find that the defect has been repaired. It's a good thing you tested all three possible coverage levels for the **while** loop!
6. Select the menu item **File->Save Project**.

Conclusion of Exercise Three

One can never be too vigilant in the embedded industry. Quality just isn't an option, so every care must be taken to ensure defects don't slip through the cracks. The last thing your team needs are frantic, last-minute code bashing sessions or - even worse - shipping what you know to be defective code. And of course, that's not even possible in industries with stringent certification standards.

You need to check everything. But how is this possible when shipping dates don't slip and you're under enormous pressure to produce? Rational Test RealTime is the answer. All the tedious tasks are automated, and great care has been taken to ensure you get your job done without losing precious development time.

Is it possible to develop a defect-free product? It's certainly not possible if you don't test. But if you do test, and test well, who knows...

Conclusion

Component testing is probably the type of testing that comes to one's mind when considering the minimal amount of effort one must make to ensure a defect-free product. As these exercises have shown, component testing is a non-trivial activity.

Imagine a world in which no tool exists that can automate stub, driver, and harness creation, in which no tool can automate data-driven tests. No wonder that testing is typically viewed negatively by developers. Again, it's not that anyone feels testing is unimportant. But how repetitive and work-intensive!

To make matters worse, without code coverage the best tests in the world are run in a vacuum. How do you know when you are finished? How do you know what test cases have been overlooked?

Use Rational Test RealTime to simplify your component testing of C functions and Ada functions and procedures. All the tedious tasks are automated so you can focus on good tests. Test boundary conditions. Try inputs that would "never" happen. And let the test scripting API generate an overabundance of inputs; why not, considering no additional effort is required on your part.

Perhaps now you can see how Rational Test RealTime, combined with the runtime analysis tools reviewed in the last group of exercises, provides you with full regression testing capabilities without having to sacrifice time better spent creating quality code.

Component Testing for C++

When speaking of C++ applications, the term "component testing" applies to the testing of C++ classes. As when working with C functions and Ada functions and procedures, embedded object testing requires the construction of a test harness (consisting of stubs and test drivers), the generation of suitable input data, and the subsequent passing of that data into the methods under test in order to verify the accuracy of the output data.

In addition to the overhead effort that was automated by the component testing feature for C and Ada - such as stub, test driver, and data generation - the component testing feature for C++ adds additional capabilities:

- Support for complex data types
- Support for private and protected class methods and variables
- Support for contract assertion checking - that is, the ability to verify properly obeyed preconditions, postconditions, and data invariants.

Such features are crucial for efficient, proactive debugging. Without them, you wouldn't have enough power at your disposal to catch all the defects in your C++ code.

The following exercise will highlight these additional capabilities.

Component Testing for C++ Exercises

Exercise One

In this exercise you will:

- uncover a part of the UMTS base station C++ code that requires further testing
- create a new activity in which you build a component test

If you need a refresher about the application you will be using during this tutorial, look here; otherwise, please proceed.

Using Code Coverage to Find Untested Code

Creating and executing a component test for C++ in Rational Test RealTime is much like the process for C and Ada component testing. Most steps are shared in common - the main difference is the content of the C++ component tests themselves, which you will see later.

As with the C and Ada testing exercises, the first step in these exercises is to use the code coverage feature of Test RealTime to determine which parts of your code require greater coverage.

To locate uncovered code:

1. From the **Window** menu, select **Close All**.
2. In the **Project Browser** tab on the right-hand side of the screen, right-click the **BaseStation** application node and select **View Report->Code Coverage** - that is, open the coverage information pertaining to your manual interaction with the UMTS base station.
3. Maximize the Code Coverage viewer
4. In the Report Window on the left-hand side of the screen, open the **PhoneNumber.cpp** node and then select the node for **PhoneNumber(unsigned int)**

Looking at the coverage for this constructor of the **PhoneNumber** class, you can see that the **for** loop has only been covered in one of three possible ways - you still need to cover 0 loop and 1 loop through the **for** statement.

To achieve this coverage, you would be wise to create an automated component test. Of particular interest would be to see what happens when a phone number of zero length is sent to the UMTS base station. The objective of this tutorial is to increase code coverage by ensuring this **PhoneNumber** constructor is called with a value of **0**.

Creating an C++ Component Test

Since you will be testing C++ code, the first order of business is to reselect the C++ TDP- based Configuration. Once done, you will follow virtually the same steps as you took for creation of a component testing test script for C and Ada. The difference? Accommodating the Test RealTime ability to implement assertion tests.

To create a C++ Component Testing test node

1. Select the menu item **Window->Close All** (and close the Output Window at the bottom of the UI if you wish to free up additional space).
2. In the toolbar dropdown list for Configurations, select the C++ TDP configuration you used in the Runtime Analysis exercises, thereby replacing the currently selected C TDP-based Configuration.
3. On the toolbar, click the **Start Page** button.
4. Select the **Activities** link on the left side of the Start Page.
5. Select the **Component Testing** link in the center of the Start Page.

6. In the window **Application Files -** Notice how all source and header files of your project are already visible (by selecting the C++ TDP-based Configuration, both C and C++ source files are accessible). No changes need to be made, so simply click the **Next** button.

7. In the window **Components Under Test**, select the checkbox next to the reference to the **PhoneNumber** class. (Since a single C++ class can be defined in multiple files, classes are listed by the Wizard rather than any implementation reference. This also explains why the file in which a class is declared is listed in the File Name column - there is only one declaration, while definitions can occur across multiple files.) Click the **Next** button.

8. In the **Test Name** field, enter the name **PhoneNumber**. Leave the default values and click the **Next** button.

9. You should now be viewing the Summary window. Click the **Next** button.

   The component testing wizard now analyzes the source code in **PhoneNumber.cpp** and **PhoneNumber.h** and creates a test for every class defined in the **.cpp** file.

10. Click the **Finish** button.

11. Select the menu item **File->Save Project**.

Notice now, in the **Project Browser** tab on the right-hand side of the screen, a C++ component testing node named **PhoneNumber** has been added to your project.

Conclusion of Exercise One

Use of the C++ component testing feature should have been a lot easier for you, considering your experience with test creation for C and Ada. Not much is different - and that's by design. All you need to do now is specify the exact test and assertion checks you would like to perform, and then execute the test. You will do that next.

Exercise Two

In this exercise you will:

- review the autogenerated C++ component testing script
- modify the test script in order to improve code coverage of the for loop
- execute the test

The Autogenerated C++ Component Test

Once you become familiar with the layout of the autogenerated test and contract check, the modifications you need to make to increase code coverage will become obvious.

To complete the test script:

1. In the **Project Browser** tab on the right-hand side of the screen, double-click the node **PhoneNumber.otd**.
2. Maximize the test script editor.

   This is the test driver script. In it you will perform those steps necessary to drive and test classes in the file under test.

   Along with the **.otc** contract-checking test - discussed in the next section - full C++ class testing is possible. The idea is that the files **PhoneNumber.cpp**, **PhoneNumber.otd**, and **PhoneNumber.otc** will be compiled and executed together (with execution taking place on the target specified by the currently selected Target Deployment Port Configuration).

   C++ component testing test scripts are written with a compiler-independent test script API. For detailed information about the script layout, take advantage of the **Test RealTime Reference Guide**. For the Tutorial, only critical script elements will be discussed.

   Each class used in the file under test is assigned its own **TEST CLASS** block - **PhoneNumber.cpp** only handles the **PhoneNumber** class, so there is only one **Test Class** block. Each **TEST CLASS** block is divided into a single **PROLOGUE** , one or more **TEST CASE** blocks, and then a single **EPILOGUE**.

   The **PROLOGUE** statement defines native code that is to be executed whenever the surrounding **TEST CLASS** execution begins. You typically use the **PROLOGUE** statement to declare and sometimes initialize the object instances of a class under test. In this exercise, the generated **PROLOGUE** creates an instance of the class **PhoneNumber**. The **EPILOGUE** structure defines native code that is to be executed whenever the execution of the surrounding **TEST CLASS** ends.

   The **TEST CASE** block generates a public method test of the class under test. The test case name is made up of the identifier of the method under test with the prefix **test**. This ensures correct overload handling.

   A typical test starts with the display of a trace (with the **PRINT** statement) and continues with the C++ native code that calls the method under test. This call is performed on the instance declared in the **PROLOGUE** block. Any parameter values are null. If the method under test returns a value, the test case continues with a CHECK statement. The test case ends with another trace display.

   For this tutorial, we would like to call the **PhoneNumber** constructor with an integer value of **0**. Since your goal is to simply increase code coverage, don't bother testing anything - just call the **PhoneNumber** constructor with a value of **0**.

3. In the **PROLOGUE** block, add **(0)** after the **obj0** identifier, so that it appears as follows:

```
PROLOGUE
{
  // Declarations of variables needed by this test class.
  // Actions to be performed before executing this test
  // class.
  #PhoneNumber obj0 (0) ;
}
```

   The # symbol indicates that the line contains native C++ code.

4. From the **File** menu, select **Save**.

Technically, you are finished. When this test script is executed, the **PhoneNumber** constructor will be called with an integer value of **0**. However, to give you some idea of how an assertion test would be useful, the next topic will take a look at the contract checking script.

The Autogenerated Contract Check

Use the contract checking test to ensure assertions are not violated. Assertions are parameter limits or restrictions that should be obeyed, but which are very often not explicitly enforced by the code. For example, it surely makes sense that a phone number never has zero digits. If that is the case then calling the PhoneNumber constructor with a value of 0 should violate this assertion. You will create this assertion.

To complete the test script:

1. In the **Project Browser** tab, double-click the node **PhoneNumber.otc**
2. Maximize the test script editor

   This is the C++ component testing contract checking script. In it you will perform those steps necessary to verify that assertions are not violated.

   Contract checking scripts are written with a compiler-independent test script API. For detailed information about the script layout, take advantage of the **Test RealTime Reference Guide**. For this Tutorial, only critical script elements will be discussed.

   For each class a **CLASS** block is created and this **CLASS** block can test for violations of:

   - invariants
   - pre-conditions/post-conditions
   - states
   - transitions

Since you wish to verify that the length of the phone number always exceeds **0**, then one possible contract check would be to ensure the **stringLength** variable of the **PhoneNumber** constructor is always greater than 0 (have a look at the source code if you wish to verify this approach yourself).

3. Scroll down the contract checking test script until you see the line:
```
WRAP PhoneNumber(unsigned int length)
REQUIRE ("Require PhoneNumber")
ENSURE ("Ensure PhoneNumber")
```
4. Modify the code as follows:
```
WRAP PhoneNumber(unsigned int length)
//REQUIRE ("Require PhoneNumber")
ENSURE (stringLength > 0)
```
5. From the **File** menu, select **Save.**

   The **WRAP** keyword lets you check for pre- and post-conditions of a class method. The **REQUIRE** keyword checks pre-conditions; the **ENSURE** keyword checks post-conditions.

   Another example of a contract check would be to verify that class invariants are never violated. For example, it certainly makes sense that the phone number can never be full and empty at the same time. This can never be, it is an invariant. The PhoneNumber class actually has these methods - isFull() and isEmpty() - so use them to verify this assertion.

6. Scroll up the contract checking test script until you see the line
```
// INVARIANT (/* expression */);
```
7. Modify this line as follows:
```
INVARIANT (!(isFull() && isEmpty()));
```
8. Select the menu item **File->Save**

Done. You are ready to compile and run the test and contract check.

Running a C++ Component Test

You have set up your tests to increase coverage of the **for** loop in one of the **PhoneNumber** constructors by calling it with a value of **0**. You have also set up two contract checks - one verifies that the phone number object is never full and empty at the same time, the other verifies that the phone number length is never set to 0. Time to run the test.

To execute the test node:

1. Select the menu item **Window->Close All**
2. Left-click the **PhoneNumber** test node (the parent node of the **PhoneNumber.otd, PhoneNumber.otc** and **PhoneNumber.cpp** nodes) and then press the **Build** toolbar button ( )

3. The test is executed as part of the build process - you will know the test has finished executing when the green execution light on the lower-right of the UI stops flashing.

4. Select the menu item **File->Save Project**.

As with your C and Ada component test, the runtime analysis features are still selected in the Build options; the file under test - PhoneNumber.cpp - was instrumented for the memory profiling, performance profiling, code coverage and runtime analysis features, which explains why the Runtime Trace viewer appears during the run. Notice how the runtime tracing feature tracked all of the method calls made throughout the execution of the test.

So have you improved code coverage? Were any of your assertions violated? That is the subject of the next exercise.

Conclusion of Exercise Two

The C++ component testing test scripting language gives you enormous object-oriented testing power with minimal effort. This compiler-independent language lets you build tests that can be used with any embedded target, so you'll never have to change your tests when the architecture you're supporting changes.

As for test script execution, this is accomplished through the Test RealTime interface regardless of the target. The Target Deployment Port takes care of everything; there is no distraction from the task at hand, which is to make quality tests and then fix problems as they are uncovered.

Assertion checking - often overlooked as too time consuming to pursue - is now easily achieved via the contract checking script. This ability gives you even greater confidence in the stability of your code.

Exercise Three

In this exercise you will:

- analyze the test results
- repair the defect discovered by the C++ component test
- rerun your test to verify that the defect has been fixed

The C++ Component Test Report

The C++ component test report summarizes all of the test results. It is hyperlinked to the test script (the **.otd** and **.otc** file) and can be browsed using the **Report Window**.

To analyze the test report:

1. In the **Project Browser** tab on the right-hand side of the screen, right-click the **PhoneNumber** component testing node and select **View Report->Code Coverage**.
2. Maximize the **Code Coverage** viewer
3. Using the **Report** tab on the left hand side of the screen, view the source code for the **PhoneNumber** constructor you called with a value of **0** in your test script.

   Have you covered the **0 loop** case of the **for** loop? Yes, indeed. (Notice the absence of coverage for **2 loops or more** - remember, in your component test, only the 0 case was tested. Your manual interaction with the UMTS base station via the mobile phone simulator was responsible for the **2 loops or more** coverage - and that coverage won't be listed here.)

   How about your contract checks?
4. In the **Project Browser** tab on the right-hand side of the screen, right-click the **PhoneNumber** test node and select **View Report->Test.**
5. Close the Project Explorer window to the right, and the Output Window at the bottom of the UI to give you more room to explore the report.

   Look at the **Report Window** on the lower-left side of the UI. Your method contract check failed - that is, the **stringLength** variable was not greater than **0**. It should come as not surprise that this assertion failed since you went out of your way to supply a length of 0. Sensibly, you should continue to test this assertion in all your regression testing of the UMTS server to ensure that "normal" phone number inputs never have a length of 0.

   Does anything else need to be done? Is everything else working properly?

   Notice how the test cases corresponding to the methods appear to have failed as well. Why should this be? As you recall, no test was actually performed in the **Test Case** block - you simply called the PhoneNumber() constructor. In fact, this failure implies the test was not able to finish properly. You should take a closer look at the runtime trace to ensure nothing unusual happened.
6. Select the **Runtime Trace** tab.

   Look closely. There are lifelines for:

   - the operating system
   - the test class block
   - the test case that calls the **PhoneNumber** constructor
   - a **PhoneNumber** object

Your assertion checks are flagged by notes - a green note means the assertion has been observed, a red note means the assertion has been violated. (Thus the note for the **stringLength** test is red.)

What about the unexpected exception? That can't be good. In fact, close inspection of the PhoneNumber lifeline shows that the destructor method was never called. Intuition probably tells you that this unhandled exception is directly related to your input of a phone number of **0** length.

The code needs to be fixed.

Repairing a Defect

The runtime tracing feature has uncovered what looks to be an unhandled case - that is, handling a phone number of **0** length. The code must be fixed.

To fix the defect:

1. In the Runtime Trace viewer, left-click the green **PhoneNumber** constructor call made by the **Test Case**

   Take a look at this **PhoneNumber** constructor (you may need to scroll down a bit in order to fully expose the function). In essence, a **numberString** object is being prepared to hold the phone number. What happens if the length of the phone number - the input to this constructor - is **0**? The **numberString** object is never created.

   The problem is the last line of this constructor. The **numberString** object is assigned a final value. How can this be if the **numberString** object is never created when the length of the phone number is 0? You need to add an extra line of code to ensure that the last line of the constructor is only executed if the length of the phone number is greater than 0.

2. Modify the source code of this **PhoneNumber** constructor as follows:

```
if(length > 0)
    numberString[length] = '\0';
```

   In other words, add the **if** statement.

3. Select the menu item **File->Save**

This should fix the problem. In the next topic, you will rerun your test to make sure the unexpected exception goes away.

Verifying the Success of Your Repairs

As you have now learned, tests always need to be rerun and reports should always be rechecked.

To check that defects have been resolved:

1. From the **View** menu, select **Other Windows** and **Project Window**.
2. From the **Window** menu, select **Close All**.
3. In the **Project Browser** tab on the right-hand side of the screen, left-click the **PhoneNumber** test node and then select the **Build** toolbar button.
4. The test has finished executing when the green execution light on the lower-right of the UI stops flashing.
5. From the **File** menu, select **Save Project**.
6. Expand the **Runtime Trace** viewer that appeared during the test run.

By looking at the Runtime Trace Viewer, you will find that the unexpected exception has disappeared and is now replaced with a call to the **PhoneNumber** destructor. One more defect has been eliminated. That was one defect you would not have caught without the assistance of the runtime tracing feature of Test RealTime.

Conclusion of Exercise Three

Rational Test RealTime, more than anything else, exposes two vital issues:

- True, error free code is guaranteed only through extremely vigilant testing and runtime analysis. Skip any part and defects might fall through - defects you either repair now, when you have time, or later, when code freeze looms and your reputation is on the line.
- With Test RealTime, this vigilance is easily accomplished. You achieve full testing and runtime analysis with minimal distraction and minimal focus on tedious, time-consuming tasks.

Again - is it possible to develop a defect-free product? It's certainly not possible if you don't test. But if you do test, and test well, who knows...

Conclusion

Now you have seen how to perform host- and target-based unit testing for C, C++, and Ada.

For all of these languages, notice how Test RealTime has allowed you to focus solely on your code. Notice how easily it has been to expose untested code and to generate new tests that not only test that code, but test it well. The time you spend testing can now be devoted to good tests - which increases the usefulness of your attention to testing in the first place.

Contract checking adds an extra layer of protection, so give some thought to using it when testing your C++ code. It's optional - you don't have to add assertion checking to your regression suite. Nevertheless, particularly if your code is called by someone

else's code, assertion checking is a simple and clean method for verifying that your code is properly used.

So are you finished? You've seen how to detect and repair:

- memory leaks
- performance bottlenecks
- functional defects

You've learned how to clarify:

- your code's call sequences
- the completeness of your testing

What's left?

System-level testing - the integration testing of distributed components. Up to now you have tested and monitored the code. Next you must see how to test the interaction of various threads, tasks, processes, and subsystems.

System Testing for C

What does embedded software testing at the system level focus on? It focuses on the interaction between two or more threads, tasks, processes, and subsystems. In this case, the communication mechanism is provided by a C language messaging API, and the system-under-test is stimulated by stubbed virtual actors.

As a tester, you have three primary interests at this point:

- does the system-under-test respond to the input signal as designed
- does the system-under-test respond to the input signal quickly enough
- can the system-under-test handle various loads that accurately reflect a working environment

The system testing feature for C-based messaging APIs enables system level testing. This is achievable because of Test RealTime's ability to define virtual actor behavior - or, using Test RealTime terminology - virtual tester (VT) behavior. There are two ways to define VT behavior:

- use the system testing test script API to define virtual tester actions
- use a probing technology to monitor system execution, recording the actions of system actors so that they can be played back one or more times simultaneously

The output virtual tester scripts not only define the message content sent to the system-under-test, but also define tests for the messages subsequently received - tests in which success or failure is based on message content, time-of-response, or both.

Once the virtual tester scripts are created, the system test deployment scheduler is used to configure the launch of one or multiple VT instances, including the machines upon which the deployment should occur (virtual testers can be executed on multiple machines, remotely, during a single test run). The resulting report consolidates all interactions, highlighting errors, while a runtime tracing diagram graphically displays system interactions. (Note how the ability to launch multiple, concurrent virtual testers lets you generate a load on the system under test, thereby enabling load and stress testing of the target system.)

This tutorial will focus on the first method suggested for generating a system test - that is, through the use of the System Testing test script API. For information about the probing technology, refer to the **Rational Test RealTime User Guide**.

System Testing for C Exercises

Exercise One

The goal of this system testing exercise is to ensure the signals sent from the UMTS base station to a registered mobile phone are accurate and timely. The signals consist of TCP/IP-based messages. (Note, however, that any protocol can be simulated by Test RealTime as long as a C-based messaging API is used.)

To accomplish this, rather than manually interacting with a mobile phone simulator, you are going to create a virtual phone that dials a number all by itself.

In this exercise you will:

- review what you have done so far, and discuss how one could adapt previous efforts to system testing

If you need a refresher about the application you will be using during this tutorial, look here; otherwise, please proceed.

Requirements for C-based System Testing with Test RealTime

Having performed the runtime analysis exercises, you have seen how a mobile phone simulator can be used to interact with the UMTS base station. The implication then is that signals were being traded between the two.

If this is the case - if, in fact, signals are passed between the mobile phone simulator and the UMTS base station - would it not be useful to "fake" the simulator with a test that can send signals to the base station and then analyze the content and timing of the signals that are returned?

You will be doing just that. You will be simulating the simulator, creating a test that can interact with the base station in a well-defined way and then test the returned signals. Put another way, you will be automating the manual interaction you performed in the Runtime Analysis portion of this tutorial.

This test is coded with a system testing test script API.

In order to build this system testing test script - that is, in order to create virtual testers - the test script code must have access to the C language messaging API used by the system under test. Without a messaging API, it would not be possible to define the signals sent from the virtual tester to the system under test, nor would it be possible to analyze the returned signals. The messaging API might be accessible in a preexisting library, accessible in source code used to build the system under test, or inaccessible (thereby necessitating manual creation of a referenceable messaging API file). In this tutorial, you will be reusing some of the UMTS base station source files; these files define the messaging API used to communicate with mobile phones.

In addition to having access to the messaging API, you must also define an adaptation layer. The adaptation layer describes how the API is to be used; in other words, how are messages sent and received.

Finally, your test script will need to describe the action of a virtual tester - indicated in a system testing test script with the reserved keyword INSTANCE. This is the part of the test script that specifies what signals are sent to the target, what signals are expected, and any timing requirements.

To summarize: When building a System Testing test, you are responsible for:

- creating or providing access to the C language messaging API
- coding the adaptation layer
- coding the INSTANCE blocks describing the simulated behavior and tests

You will not be responsible for creating any of these above items in the Tutorial - the files are provided for you - but their content will be reviewed.

For Solaris, Linux, HP-UX and AIX Users

You need to install the System Testing agent software, a daemon that must be running on the host to act as an interface between virtual testers and the machine running Test RealTime. The instructions for the installation process are located in the **Rational Test User Guide**, in the **System Testing Overview** chapter.

For Windows users, this daemon has already been installed.

Creating a System Test

As with the C, C++ and Ada component testing features of Rational Test RealTime, your first responsibility is to create a node in your project for the system test.

To create a System Testing node:

1. From the **View** menu, select **Other Windows** and **Project Window**, if necessary.
2. From the **Window** menu select **Close All**, if necessary.

3. Using the TDP Configuration selector on the toolbar, ensure the C TDP-based Configuration is selected. This is necessary to support the C language messaging API.

4. Activate the **Start Page** by selecting the toolbar button.

5. Select the **Activities** link on the left-hand side of the Start Page.

6. Select the **System Testing for C** link on the Start Page.

7. In the window **Create System Testing Node**, enter the name **MobilePhoneVT** and then click the **OK** button.

8. In the **Test Script Selection** window, do not opt to create a new test script. Rather, select the browse '**...**' button and select the file named **MobilePhoneVT.pts**, located within the Rational Test RealTime installation folder, in the folder **\examples\BaseStation_C\tests**.

9. On the same window, in the **Interface Files List** listbox, click the **Add** button and then browse to the UMTS base station source files located within the Rational Test RealTime installation folder, in the folder **\examples\BaseStation_C\src**.
   Open two of the C language header files:

   - tcpsck.h
   - UmtsMsg.h

   These two files define the messaging API used by the UMTS base station to communicate with mobile phones. They will be reused in order to define the messaging API employed by the virtual testers.

10. Now click the **Next** button on the **Test Script Selection** window.

11. The **Include Directories List** window contains a listing of directories that contain files referenced by the test script or the source files used for the test. This window currently references the proper directory, so simply click the **Next** button.

12. If you had decided to create a new test script, the System Testing Wizard would be finished at this point; it would now be time to code the adaptation layer and INSTANCE actions. However, since you are using a preexisting test script (**MobilePhone.pts**), the Wizard continues to the next automated phase, which is creating and configuring the virtual tester drivers.

The system testing node has already been created (you can see it on the Project Browser). The next step is to configure the test script that will reference the messaging API, define the adaptation layer, and describe virtual tester actions.

Configuring Virtual Tester Drivers

The System Testing Wizard has analyzed the preexisting test script - **MobilePhoneVT.pts** - noting the INSTANCE blocks defined within. Recall that the INSTANCE blocks describe the exact actions a virtual tester should take, including:

- what signals to send
- what signals are expected in response
- what tests should be performed

A test script can contain more than one INSTANCE definition. (The System Testing test script will be reviewed in Exercise Two.)

Your next responsibility is to create virtual tester drivers. A virtual tester driver is used to create one or more virtual testers - or, more specifically, one or more virtual testers for one or more of the INSTANCE blocks defined in the test script. A virtual tester driver can be configured to support only one INSTANCE block, or it can be configured to support multiple. The advantage of only supporting only one type of INSTANCE block is that the driver size is minimized.

To set up a Virtual Tester:

1. In the **Virtual Test Driver Creation**, next to the **Virtual Tester Driver List**, click **New**.
2. In the **Create Virtual Tester Driver** window, name this new binary **Driver1** and click **OK**.

   Now you must specify which INSTANCE blocks this driver supports and, if applicable, which SCENARIO and FAMILY blocks within the INSTANCE blocks are supported. (Again, the system testing test script language is discussed in Exercise Two.)

   Notice how the **General** tab on the **Virtual Tester Driver Creation** window lets you select which INSTANCE block is supported by the selected VT driver. In addition, the TDP configuration for this binary can be changed and modified as well. The **Scenario** and **Family** tabs let you deselect SCENARIO and FAMILY blocks you don't want the selected driver to support.
3. For this tutorial, you will only be using the **Driver1** driver, and you want this driver to support all INSTANCE blocks, so the **Implemented INSTANCE** dropdown list on the **General** tab should remain the same - that is, all INSTANCE blocks will be supported by this one driver.
4. In the **Target** dropdown list on the **General** tab, select the C-language TDP for your machine. (Since multiple drivers could be distributed across multiple execution environments, it is conceivable that each test driver would be assigned its own TDP.)
5. Click the **Next** button.

One step to go. You must now describe the deployment configuration - that is, you must create individual virtual testers, the VT driver from which each will be generated, and - if applicable - the INSTANCE block that will be executed. This window can also be used to create multiple, concurrent VTs of the same type.

Configuring the Deployment Configuration

Each virtual tester driver can be used to create one or more virtual testers. In addition, if the driver supports more than one **INSTANCE** block, then each specific INSTANCE block needs to be assigned a virtual tester. For this tutorial, you will just be running a test that consists of a single virtual tester.

To set up the Deployment Configuration:

1. In the **Deployment Configuration** window, click the **Add** button to create a virtual tester.
2. Select **phone1** in the **Instance** column

   The **Virtual Tester Driver** column is used to select the driver, the **INSTANCE** column is used to select the INSTANCE, and the **Network Node** column is used to specify the machine upon which the virtual tester(s) will be deployed. Since only one virtual tester is required for the tutorial, the column **Number of Occurrences** can remain equal to **1**.
3. Click the **Next** button.
4. Review the settings on the **Test Generation Summary** window if you wish, then click the **Finish** button.

   You're screen should appear as follows:

Note that if, at any point, you feel the need to modify the deployment configuration, you can right-click the test script node (in this tutorial that would be the **MobilePhoneVT.pts** node) and select the **Virtual Tester Driver Configuration** option.

One step remains. Recall that you will be using UMTS base station files to implement the messaging API. During the System Testing Wizard you selected the two header files that contain the API specification. What you must do now is reference the source files that implement the messaging API. This could not be done in the wizard because there was no messaging-API library to import. The source files for the messaging API need to be compiled along with the test script and thus must be added directly.

5. Right-click the virtual tester driver node **driver1** on the **Tests** tab and select **Add Child->Source Files**
6. Browse to the UMTS base station source files located within the Rational Test RealTime installation folder, in the folder **\examples\BaseStation_C\src**, and open all of the C language files:
    - tcpsck.c
    - UmtsCode.c
    - UmtsMsg.c
7. From the **File** menu, select **Save Project.**

There is no need to instrument the three C language files used to implement the messaging API, but rather than altering the entire TDP configuration using the **Build** dropdown menu, you are simply ensuring these three particular files won't be instrumented.

You are now ready to simulate the mobile phone and thus drive the UMTS base station, ensuring the base station responds to signals in a proper and timely fashion.

Conclusion of Exercise One

Distributed embedded environments can be highly variable. As you have seen, the System Testing Wizard and test script API accommodates this variability, enabling a highly adaptable test environment for your networked components.

The next section focuses on the test script itself, and then its execution.

Exercise Two

In this exercise, you will:

- review the system testing test script
- execute the test

The System Testing Test Script

A brief tour of the C-based system testing test script should clear up any further mystery about how the virtual testers are implemented.

To modify the test script:

1. Double click the **MobilePhoneVT.pts** node on the **Project Browser** tab.
2. Maximize the test script

Highlights, from top to bottom (use the **Rational Test RealTime Reference Guide** for detailed information regarding the system testing test script API):

- DECLARE_INSTANCE - Note how only one INSTANCE block exists in this test script.
- MESSAGE - These variables will contain the message sent from the UMTS base station to the mobile phone.
- PROC ... END PROC - Used to define a function that will be called multiple times.
- PROCSEND ... END PROCSEND - Part of the adaptation layer; describes the steps necessary for a virtual tester to send a message.
- CALLBACK ... END CALLBACK - Part of the adaptation layer; describes the steps necessary for a virtual tester to receive a message.
- INITIALIZATION ... END INITIALIZATION - Indicates those steps that must occur before any SCENARIO block executes. Only applies to those SCENARIO blocks at the same level as the INITIALIZATION block. In this case, the virtual tester opens a TCP/IP socket to the base station and then connects to it. (Note that the phone has not yet been registered to the base station; the INITIALIZATION block only opens a connection to the phone; with this connection, the mobile phone can then try to register.)
- TERMINATION ... END TERMINATION - Indicates those steps that must occur after every SCENARIO block finishes executing. Only applies to those SCENARIO blocks at the same level as the INITIALIZATION block.
- SCENARIO ... END SCENARIO - Highest level blocking of specific virtual tester actions. A SCENARIO block can consist of more than one child SCENARIO block. The INSTANCE blocks are typically defined in SCENARIO blocks.
- INSTANCE ... END INSTANCE - Contains code specific for a virtual tester instance.
- SEND - Sends a message.
- WAITTIL - Waits for a message, and tests the message for both content and promptness. Reports a failure if the received message does not match expected, was never received, or was received late.

Take a look around. Notice how the **call_busy** scenario uses the phone number **5550000**, and how the **call_success** scenario uses the phone number **5550001**. As you may recall, these were the phone numbers used in the runtime analysis portion of this tutorial.

Once you are comfortable with the test script, you can proceed to execute the test.

Running the Base Station in the Background

The objective of your system test is to test the UMTS base station. However, how will you run the base station application at the same time as the test? Normally, the tested

thread, task, process, or subsystem will be run somewhere on your network, but for the purposes of the Tutorial, you will have to manually run it yourself.

To execute the system under test:

1. From the command line (or via Windows Explorer on Windows) browse to the UMTS base station executable provided with the Tutorial. This file is located within the Rational Test RealTime installation folder, in the folder **\examples\BaseStation_C**, and is called:
   - on Windows - BaseStation.exe
   - on Solaris, Linux, HP-UX and AIX - **BaseStation.sh**

   (For Solaris, Linux, HP-UX and AIX you also have the option of selecting the base station executable itself, located in the same directory. The shell script referenced above simplifies matters.)
2. Run the base station executable. Windows users should minimize the command window that appears.

Executing the System Test

It might seem like a lot of work to get to this point, but consider what you have accomplished and what can be accomplished. You have:

- Modeled dynamic, distributed component interaction
- Created virtual testers that could, simply by specifying various IP addresses, execute on multiple machines
- Enabled load testing
- Provided a means for implementing scenario-based testing

Each step you performed, in reality, has hidden an enormous amount of complexity.

In this section, you will run the test.

To execute the test:

1. Run the System Testing agent software - that is, run the software that supports virtual tester execution. The agent executable is called **atsagtd** and it can be executed in one of two ways:
   - On Windows - In the **Start** menu, select **Programs > Rational Software > Rational Test RealTime > Tools->Rational Test RealTime System Testing Agent** (which is simply a link to the file **atsagtd.exe**, executable from the command line with a single argument - the port number to be used, 10000 in this case). Minimize the command window that appears.

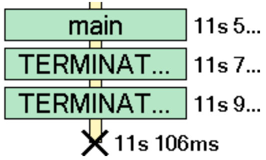
- On Solaris, Linux, HP-UX and AIX - This agent is already launched if you have followed the System Testing Agent installation instructions in the **Rational Test RealTime User Guide**, in the **System Testing Overview** chapter.

When test execution has completed, a post-execution trace of events will be created; this trace is used later in the tutorial. However, if you wish to monitor execution via an on-the-fly trace as well, follow the next five steps. Otherwise, skip to Step 7.

2. Right-click the **MobilePhoneVT** system testing node on the **Project Browser** tab and select **Settings**
3. Expand the **System Testing** node on the left-hand side of the **Configuration Settings** window, select the **Report Generator** node and then select **Yes** in the dropdown list associated with the property **Display using on-the-fly mode**.
4. Now select the **Target Deployment Port for System Testing** node (child of the **System Testing** node) on the left of the **Configuration Settings** window and then select **Yes** in the dropdown list associated with the property **Enable On-the-fly Runtime Tracing.**
5. Click **OK.**
6. From the **Window** menu, select **Close All**.

   Now run the test.

7. Left-click the **MobilePhoneVT** system testing node and press the **Build** button. (If you are asked to rebuild the nodes, click the **Yes** button.) The test harness is now built, deployed, and executed.
8. If you opted to create an on-the-fly trace: The Runtime Trace viewer will appear. The test has finished executing when the right-hand **phone1_0** lifeline in the viewer is stamped at its base by a black **X**:



   If you opted to not create an on-the-fly trace: Execution has completed when the green execution light in the lower-right of the Test RealTime GUI stops flashing (00:00:19)

9. From the **File** menu, select **Save Project**.
10. On Windows only - close the System Testing Agent.

The on-the-fly runtime tracing diagram shows interactions, as they happened, between the software-under-test (SUT) - that is, the UMTS base station - and the single virtual tester you had created for the system test. This virtual tester is named **phone1_0**. Such an on-the-fly diagram is useful for monitoring test execution; however, this diagram is not crucial to the extent that the information within it has also been captured for post-execution analysis in a separate runtime tracing diagram.

In the next exercise, you will look at this runtime tracing diagram and then study the system test report.

Conclusion of Exercise Two

The system test scripting language has been designed to accommodate the intricacies of distributed, scenario-based testing. In this type of testing, virtual actors send and receive signals derived from a C language messaging API. Supported communication links include:

- TCP/IP
- CAN Bus
- RS232/422
- ARINC Bus
- etc.

Execution can be distributed over multiple virtual testers performing multiple actions on multiple clients. The test can be monitored by the runtime tracing feature of Test RealTime to expose the test actions as they occur, letting you validate the accuracy of your test and ensuring all is operating on-the-wire as designed. Timing of each signal is also performed, enabling load testing of the component under test.

Exercise Three

In this exercise, you will:

- analyze the runtime trace viewer generated during test execution
- analyze the system test report

Analyzing the Execution-based Runtime Trace Viewer

A complete runtime tracing diagram of test execution was created at the conclusion of the test run.

To open the UML sequence diagram:

1. To gain additional space, close the Output Window at the bottom of the UI.

2. On the **Project Browser** tab, right-click the **MobilePhoneVT** System Testing node and select **Runtime Trace**.

3. Right-click-hold within the **Runtime Trace** viewer and select the option **Hide Coverage Bar**.

4. Make sure you are viewing the top of the runtime tracing diagram, using the right hand slider bar if necessary.

   The UMTS base station is represented by the lifeline labeled **SUT BaseStation**; the virtual tester lifeline is labeled **VT phone1_0** (that is, virtual tester 0 for the phone1 **INSTANCE** block you chose in the **Deployment Configuration** window - see the topic **Configuring the Deployment Algorithm** in the previous exercise to refresh your memory).

   The virtual tester first performs its initialization functions - represented by the **INITIALIZATION** note. Then it performs each of the three **SCENARIO** blocks located in the test script - named **connect**, **call_busy**, and **call_success**. Each is visually traced, consecutively, as they occur.

   The **main** block consists of the three **SCENARIO** blocks, performed one at a time. Each scenario consists of a single test - a **WAITTIL**. Recall that a **WAITTIL** command both checks the content of a received message as well as ensures the message is received within a specified amount of time.

5. Click on the **INITIALIZATION** node at the top of the runtime tracing diagram.

The system test report is opened. You will look at that report next.

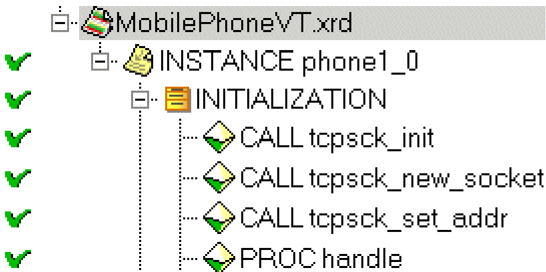
Analyzing the System Test Report

The Runtime Trace viewer shows you what happened, but it doesn't make any reference to the success or failure of each **WAITTIL**. All success or failure values for any system test are recorded in the system test report.

To open the test report:

1. Close the Project Explorer Window on the right-hand side of the screen to gain additional room for the runtime tracing diagram.

2. In the Report Window on the left-hand side of the UI, close the **INITIALIZATION**, **SCENARIO main**, and **TERMINATION** nodes. The window should appear as follows:



Look at the Report Window; notice the existence of a node named **INSTANCE phone1_0** - this is a reference to Virtual Tester 0 for the phone1 **INSTANCE** block. For every virtual tester executing the phone1 INSTANCE block, a separate node would exist in this browse tree. Since your test consisted of only one virtual tester, only one node exists in the tree.

By clicking the **INITIALIZATION** note in the Runtime Trace viewer, you were jumped to the INITIALIZATION section of the system test report. This section of the report could also be accessed by double-clicking the **INITIALIZATION** node in the Report Window.

3. Expand the **INITIALIZATION** node in the Report Window.

   Here, in the report, you see all of the **CALL**s made in the **INITIALIZATION** block of your system test. If any of these calls failed, that information would be found here.

4. Expand the **SCENARIO main** node in the Report Window.

   Now you're looking at all of the functions that occur within each SCENARIO block. (Expanding the **SCENARIO main** block in the Report Window will let you maneuver through the three **SCENARIO**s.) Again, every action is listed. Successes are color-code pink.

5. In the Report Window, expand the **SCENARIO main** node if you haven't done so already, and then double-click the **WAITTIL** node located within the **SCENARIO connect** node:

   Look at the report. Notice how the **WAITTIL** section is broken down into a **WAITED EVENTS/RECEIVED EVENTS** section - specifically, into the expected message (called **MATCHING**) and the obtained message (called **mResponse**). The expected message defines what must be in the obtained message; in this case, the obtained message must contain a field named **command** with a string value of **CNX OK**. As you can see, the obtained message can contain more data

than was tested for; for example, the obtained message contains the additional fields **phoneNumber**, **simCardId** and **baseStationId**.

(The **WAITTIL** contains the clause **WTIME>1000**. This means that if it takes more than 10 seconds for the awaited message to arrive, a timeout would occur and the timeout error would be reported. The unit of measurement for this parameter can be modified via a TDP setting.)

6. To view the test summary, scroll to the top of the report in the Report Browser window.

Notice that 4 tests passed and 0 tests failed. This is a reference to the four **SCENARIO** blocks - the parent **SCENARIO** block named **main** and the three child **SCENARIO** blocks named **connect**, **call_busy**, and **call_success**.

Familiarize yourself with this report, noting that you can left-click all green-colored script functions performed by the virtual tester to view the test script itself.

Conclusion of Exercise Three

With the assistance of both the on-the-fly runtime tracing diagram as well as the post-execution runtime tracing diagram, test activity can be monitored, messaging sequences can be understood, and scenario-based system testing use cases can be visualized.

Once the test has been performed, the system test report succinctly summarizes the results, letting you focus directly on uncovered problems without the distraction of what might have been a large amount of collected data.

Conclusion

C and Ada component testing exposed problems at the function level in the UMTS base station C code. C++ component testing exposed problems at the class level in the UMTS base station C++ code. Finally, with system testing, problems that might exist at the signal passing level were exposed. The base station has been tested at all levels of complexity.

Message-passing defects can be very difficult to catch. Ideally, to uncover problems in this area:

- system actors should be simulated to ensure well-defined scenario use cases
- these system actors should be distributed to closely mirror the true target environment
- test data should be summarized and stored in a single, exportable file

The system testing feature of Test RealTime does all of these, with the additional benefits of:

- interactive source-code editing

- runtime observation capabilities
- target independence

The key to successful system testing is an understanding of realistic scenario use cases. You need to ask yourself what is really going to happen in your system, in what order it will happen, and what environmental constraints will exist at that time. Once determined, you should next consider the likelihood of environmental stress factors that could cause system degradation. If so, then load and stress testing should become a part of your testing regimen.

Assuming true component architectures have been used in your system, if defects are found at the system level - either improper or missing signals or signal delays - then the Test RealTime runtime analysis features should be used in conjunction with the testing features to narrow your focus and thereby find the root cause.

All of these tests should become part of a regression testing suite. This is the topic of the Tutorial Conclusion - combining all tests into a single regression testing suite.

Further System Testing Exercises

As the MobilePhoneVT.pts file is currently constructed, there are no failures. Can you make changes to the test script that will guarantee the UMTS base station fails to act appropriately?

Java Track

Automated Component Testing

You have just completed a variety of what are, in essence, reliability tests on the UMTS base station. In other words, you have verified the absence of memory leaks, the optimization of performance, the sensibility of process flow, and the completeness of your testing.

But does the base station code do what it is designed to do? And wouldn't it be useful to create automated tests rather than rely solely on manual interaction?

Runtime analysis completes the picture, but functional testing of your code gets to the heart of the matter - that is, will your application generate the results it was designed to achieve. Rational Test RealTime provides you with a component testing feature designed explicitly for the Java language.

Component Testing for Java with Rational Test RealTime

When speaking of Java applications, the term "component testing" - also sometimes referred to as "unit testing" - applies to the testing of Java classes. A method is passed

a possible set of inputs, and the output for each set is validated to ensure accuracy. This can be done with either a single method, a group of unrelated methods, or with a sequential group of methods - i.e. one method calling another, verifying the overall or integrated, result.

Sounds simple but, unfortunately, in the embedded world its practice can be quite difficult. Why?

- What if the function you wish to test relies on the existence of other functions that have not yet been coded?
- How will you call the function-under-test in the first place?
- How will you create and maintain a variety of potential inputs and associated outputs - that is, how will you make data-driven testing manageable?
- What kind of effort and knowledge is required to run the test on your target architecture - that is, in the intended, native environment?

The component testing feature of Rational Test RealTime for the Java language provides a means for automating and verifying the above concerns. Through source code analysis:

- Yet-to-be coded functions and procedures are "stubbed"; in other words, these functions are created for you
- A test driver is generated to facilitate communication between your running code and the test
- A test harness, independent of your test, is constructed to ensure adoption of your target architecture and thus enabling in-situ test execution

Plus, to ensure that software developers are not constrained by Test RealTime in their ability to create robust, highly flexible, data-driven tests:

- All tests are derived from the JUnit framework (www.junit.org), thereby maintaining the ease of use and simplicity of the industry's most popular test harness for the Java language. In fact, preexisting JUnit tests can be imported for use within Test RealTime - absolutely no modification is necessary.
- All tests use the Java scripting language - that is, write your tests using the same language leveraged to create the source code being tested.

With the assistance of the Target Deployment technology, the end result is an extensible, flexible, automated testing tool for component and integration testing.

Component Testing for Java Exercises

Exercise One

In this exercise you will:

- uncover a part of the UMTS base station code that requires further testing
- create a new activity in which you build a component test

If you need a refresher about the application you will be using during this tutorial, look here; otherwise, please proceed.

Using Code Coverage to Find Untested Code

During the code coverage review, you surely noticed a fair amount of untested code. For this tutorial, you will focus on one particular section.

To locate uncovered code:

1. First, select the menu item **File->Save Project**
2. If necessary, select the **Code Coverage** tab.
3. In the Report Window on the left-hand side of the screen, open the **PhoneNumber.java** node, then open the **baseStation.PhoneNumber** node, and then left-click the **remove_digit()** method.

This function has not been covered at all. You will achieve this coverage via component testing.

Creating a Java Component Test

Using the Component Testing Wizard, you will now create a test for all functions in the file **PhoneNumber.java** - including the **removeDigit()** method that contains has not yet been covered.

To create a Java test node:

1. From the **Window** menu, select **Close All**(and close the **Output Window** at the bottom of the UI if you wish to free up additional space.
2. On the toolbar, click the **Start Page** button.
3. Select the **Activities** link on the left side of the Start Page.
4. Select the **Component Testing** link in the center of the Start Page.
5. In the window **Application Files -** Notice how all source files of your project are already visible. No changes need to be made, so simply click the **Next** button. Static metrics are recalculated.

   In the **Components Under Test** window that has now appeared, you are asked to specify which classes you would like to test. There are a variety of ways for making this decision. One method is to use the static metrics that have been automatically calculated. Certain measurements of code complexity are listed for you:

- V(g) - Also called the Cyclomatic Number, it is a measure of the complexity of a function that is correlated with difficulty in testing. The standard value is between 1 and 10. A value of 1 means the code has no branching. A function's cyclomatic complexity should not exceed 10
- Statements - Total number of statements in a function.
- Nested Level - Statement nesting level.
- Call to Components - Number of calls to methods defined outside the class.
- Utilization of Variables - Number of uses of attributes defined outside the class.

Sorting by any of these metrics columns - by left-clicking a column header - lets you prioritize your test selection, choosing the more complicated functions first.

Additional metric information can be viewed by selecting the **Metrics Diagram** button on the lower right-hand side of the screen. Selection of this button opens a graph enabling visualization of two, selected static metrics graphed against one another. Select a data point in this graph to indicate your desire to test the associated functions.

For this Tutorial, your test selection is based on the desire to increase code coverage, so the static metrics do not affect your decision. You need to test the **removeDigit()** method, which is a part of the **PhoneNumber** class.

6. In the window **Components Under Test**, select the checkbox next to the reference to the **PhoneNumber** class. Click the **Next** button.

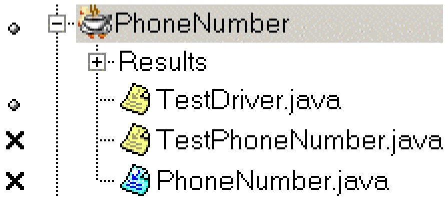
   In the **Test Mode** window that has now appeared you are asked to make two decisions:

   - If you've selected more than one class to test, do you want all classes to be part of the same test script (Single Mode) or do you want each class to be assigned to its own test script (Multiple Mode). A single test script would be easier to manage, but multiple test scripts let you provide custom Configuration settings to each test.
   - Do you want Test RealTime to make some basic assumptions about test harness and test stub generation? If so, use Typical Mode; if not, use Expert Mode.

7. In the **Test Script Generation Settings** window, enter the name **PhoneNumber** into the **Test Name** edit box then click the **Next** button.
8. You should now be viewing the **Summary** window. Click **Next**.
9. Click **Finish**.

10. From the **File** menu, select **Save Project**.

Notice how, in the **Project Browser** tab on the right-hand side of the screen, a Java component testing node named **PhoneNumber** has been added to your project.



Conclusion of Exercise One

The advantages of automated testing is that it enables regression testing - that is, it ensures nothing regresses. Just because code appeared to be functional in Build X, doesn't mean that code will continue to be functional in Build X+1.

Few would dispute the usefulness of component testing, but many would claim there is not enough time to do it. Every effort has been made to simplify this process in Rational Test RealTime so that you can simply focus on making good tests, getting readable results, and making quality code.

Exercise Two

In this exercise you will:

- review the autogenerated component test
- improve the autogenerated component test to increase code coverage as well as to verify proper functionality
- execute the component test

The Autogenerated Java Component Test

Once you become familiar with the layout of the autogenerated test and test driver, the modifications you need to make to increase code coverage will become obvious.

To modify the test script:

1. In the **Project Browser** tab on the right-hand side of the screen, double-click the node **TestPhoneNumber.java**.
2. Maximize the test script editor.

This is the test driver script. In it you will perform those steps necessary to drive and test methods in the class under test.

Combined with the test driver script - discussed in the next section - full Java class testing is possible. The idea is that the files **TestDriver.java** (the test harness), **TestPhoneNumber.java** (the test), and **PhoneNumber.java** (the relevant source file) will be compiled and executed together (with execution taking place on the target specified by the currently selected Target Deployment Port Configuration).

Java component testing test scripts are written using the Java language. For detailed information about the script layout, take advantage of the **Reference Manual**. For the Tutorial, only critical script elements will be discussed.

Each Java class under test is assigned a test class whose name, by default, is the name of the class under test preceded by the word **Test** - thus the test class for the **PhoneNumber** class is named **TestPhoneNumber**. Each test class inherits from the **TestCase** class, which is a part of the JUnit framework. Certain duties, such as constructors and set-up/tear-down functions - as defined within JUnit - are automatically generated for you by the Component Testing Wizard.

Your responsibility is to simply define the actual tests - just as you would do with JUnit alone. To ensure that Test RealTime is able to check the success and failure of your tests, each test should be a method of the test class assigned a name beginning with the word **test**. For example, to test the **removeDigit()** method, the TestPhoneNumber class should be supplied with a method named **testRemoveDigit()** (or testFoo, or testBar - anything that begins with **test**).

You now understand the essence of Rational Test RealTime component testing test script for Java. Rather than having you define tests yourself, a test script has been configured for you.

A Customized Java Component Test

A customized component test script has been created for you. This test will be used to test the methods of the **PhoneNumber** class - in particular, the method **removeDigit()**.

To customize the test script:

1. From the **Window** menu, select **Close All**.
2. Select the **Project Browser** tab on the right-hand side of the screen, select the **TestPhoneNumber.java** node (child of the **PhoneNumber** Java component testing node), and then select the menu item **Edit->Delete**.
3. Right-click the **PhoneNumber** Java component testing node and select **Add Child->Files...**
4. Browse to the Test RealTime installation folder and **Open** the file **\examples\BaseStation_Java\test\TestPhoneNumber.java.**
5. Select the **PhoneNumber** test node and click the **Settings** button.

6. In the Configuration Settings dialog box, select **Build** and **Compiler**.
7. Select the **Class Path** setting, and click the "**...**" button.
8. Click the **Add Directories** button and add both **\examples\BaseStation_Java\test** and **\examples\BaseStation_Java\src** to the Class Path.
9. Now, select the **\examples\BaseStation_Java\test\PhoneNumber** Class Path entry, and click the **Delete** button to remove it from the Class Path.
10. Click **OK** and close the Configuration Settings box.
11. Double-click the node **TestPhoneNumber.java**.
12. Maximize the test script window.
13. Scroll down the text editor until you can see the test methods for **addDigit()**, **removeDigit()** and **cleanNumber()**.

The tests should be very straightforward. The **isEmpty()** method is frequently used as an assertion check while the **verifyEquals()** method - inherited from JUnit - is used to ensure values are properly stored. Inherently, by calling the **removeDigit()** method via the **testRemoveDigit()** test method, code coverage on this method will be increased.

You are now ready to test - but first, take a quick look at the test driver.

The Autogenerated Java Test Driver

As defined by JUnit, each test class must be called by a test driver class - and that class is represented here, in the TestDriver.java file.

To view the Test Driver:

1. In the Project Window on the right-hand side of the screen, on the Project Browser tab, double-click the **TestDriver.java** node (child of the **PhoneNumber** Java component testing node).

Note how the **TestDriver** class contains **main()**, and note how the **TestDriver()** constructor contains a call to the test class **TestPhoneNumber**. If other test classes were generated by the Component Testing Wizard - that is, if you had selected multiple classes for testing and opted to have all classes tested as part of the same test driver - you would see them listed here.

Running a Java Component Test

Running a component test is as simple as it was to build and execute the UMTS base station used in the runtime analysis exercises.

To execute the test:

1. Select the menu item **File->Save Project**.
2. Select the menu item **Window->Close All**
3. On the **Project Browser** tab, right-click the **PhoneNumber** Java component testing node and then select the **Build** option OR left-click the **PhoneNumber** Java component testing node and then select the **Build** toolbar button ( ).
4. The test is executed as part of the build process - you will know the test has finished executing when the green execution light on the lower-right of the UI stops flashing. Wait for this to happen to ensure all test reports are created.

   You may have forgotten that the runtime analysis tools are still selected in the Build options; the class under test - PhoneNumber - was instrumented (within the source file PhoneNumber.java) for the memory profiling, performance profiling, code coverage analysis and runtime tracing features of Test RealTime, which explains why the Runtime Trace viewer appears during the run. You'll look at the runtime tracing diagram in a moment - but just to be sure, check the Code Coverage viewer to verify that you have achieved greater coverage.

5. In the Project Window, right-click the **PhoneNumber** Java component testing node and select **View Report->Code Coverage**.
6. In the Report Window on the left-hand side of the UI, expand the **PhoneNumber.java** node, then expand the **baseStation.PhoneNumber** node, then left-click the **removeDigit()** method node.

   As you can see, the majority of the **removeDigit()** method has been covered. Now to the runtime tracing diagram.

7. Select the **Runtime Trace** tab.
8. Close the Project Window to the right and the Output Window at the bottom of the UI, hide the **Coverage Bar**, **Thread Bar** and **Memory Usage Bar** in the runtime tracing diagram, and set a zoom level of around 75% for the Runtime Trace viewer.

   Notice how runtime tracing tracked all of the calls made by each test method. Of particular note is the Test Case lifeline, the fourth lifeline of the runtime tracing diagram. This lifeline represents each test class - that is, you should see a reference to **testAddDigit**, **testRemoveDigit** and **testClearNumber** as you move down the diagram.

   Along this test class lifeline you should notice three things:

   - at the very beginning, a note containing the name of the test class
   - method calls directed at an instance of the tested class - in this case, **obj0/obj1/obj2** of the **PhoneNumber** class

- tiny glyphs representing the actual tests performed in each test class - a pass is represented by while a failure is represented by . The mouse tool tip, when the mouse is hovered over each glyph, indicates the corresponding test.

A lot of information is contained in this runtime tracing diagram, which is why an additional test report is generated to simplify analysis of your test results.

9. Select the menu item **View->Other Windows->Project Window**.
10. On the **Project Browser** tab in Project Window, right-click the **PhoneNumber** Java component testing node and select **View Report->Test**.
11. Close the Project Window on the right-hand side of the UI.

What is the result of your tests? Did you improve coverage on the **while** statement? That is the subject of the next exercise.

Conclusion of Exercise Two

The ability to use Java in your component tests gives you enormous object-oriented testing power with minimal effort. And despite using the JUnit framework, your tests exist independent of any particular embedded target, so you'll never have to change your tests when the architecture you're supporting changes.

Every effort has been made to ensure there is no distraction from the task at hand, which is to make quality tests and then fix problems as they are uncovered.

Exercise Three

In this exercise you will:

- analyze the test results
- repair the defect discovered by the Java component test
- rerun your test to verify that the defect has been fixed

The Java Component Test Report

A component test report summarizes all test results. It is hyperlinked to the test script (in this case, the file **TestPhoneNumber.java**) and can be browsed using the **Report Browser** on the left-hand side of the screen..

At the top of the report is an overall summary of test execution. Notice the **Passed** and **Failed** items - of twelve tests, only ten passed. Not good news. If this was a long test report, you could use double-clicks on failed nodes in the Report Window to the left, or use the , or buttons on the toolbar to navigate through the report and better understand where the failures occurred. However, this is a short report, so you can easily see the following:

2.1.2 -Test Case TestPhoneNumber.testRemoveDigit

Expression	Status
verifyEquals(obj.isEmpty(),false)	Passed
verifyEquals(obj.toString(),"123")	Passed
verifyEquals(obj.toString(),"12")	Failed
verifyEquals(obj.toString(),"1")	Failed
verifyEquals(obj.isEmpty(),true)	Passed

Apparently, the **RemoveDigit()** method is not working properly when the phone number is reduced from 3 to 2 digits and from 2 to 1 digit - it's a good thing you didn't simply rely on simulation usage to increase code coverage! You need to check the code and verify if a repair is required.

Repairing a Defect

To fix a defect:

1. From the **View** menu, select **Other Windows** and **Project Window**.
2. Select the **Asset Browser** in the Project Window.
3. Select the **Sort Method** named **by File**, expand the **PhoneNumber.java** node, the **baseStation** child package and the **PhoneNumber** child node. Now, double-click the **removeDigit()** method node.

   Take a look at the **removeDigit()** method. What should happen is that whenever the **removeDigit()** method is called, the last digit of the phone number - actually stored as a string - should be removed. However, look at the line that actually removes the digit:

   ```
   _numbers.removeElementAt(0);
   ```

   In fact, the first digit of the phone number is being removed - this is a defect. The value passed to the **removeElementAt()** method should be the location of the last element in the string containing the phone number. You have to modify the code.

4. Change the following code from:

   ```
   _numbers.removeElementAt(0);
   ```

   to

   ```
   _numbers.removeElementAt(_numbers.size()-1);
   ```

5. Select the menu item **File->Save**

This should fix the problem. In the next topic, you will rerun your test to make sure the unexpected exception goes away.

Verifying the Success of Your Repairs

As you have now learned, tests always need to be rerun and reports should always be rechecked.

To validate the repair:

6. From the **Window** menu, select **Close All**
7. In the **Project Browser** tab on the right-hand side of the screen, left-click the **PhoneNumber** test node and then select the **Build** button.
8. The test has finished executing when the green execution light on the lower-right of the UI stops flashing. Make sure to wait until it has stopped flashing to ensure all test reports are updated.
9. From the **File** menu, select **Save Project**.
10. In the Project Browser to the right, right-click the **PhoneNumber** Java component testing node and select **View Report->Test**.

One more defect has been eliminated.

Conclusion of Exercise Three

Rational Test RealTime, more than anything else, exposes two vital issues:

- True, error free code is guaranteed only through extremely vigilant testing and runtime analysis. Skip any part and defects might fall through - defects you either repair now, when you have time, or later, when code freeze looms and your reputation is on the line.
- With Test RealTime, this vigilance is easily accomplished. You achieve full testing and runtime analysis with minimal distraction and minimal focus on tedious, time-consuming tasks.

Is it possible to develop a defect-free product? It's certainly not possible if you don't test. But if you do test, and test well, who knows... A defect is only a defect if you didn't know it was there when you have checked in your code.

Conclusion

Java Component Testing Conclusion - with a Word about Process

Component testing is probably the type of testing that comes to one's mind when considering the minimal amount of effort one must make to ensure a defect-free product. As these exercises have shown, component testing is a non-trivial activity.

Imagine a world in which no tool exists that can automate stub, driver, and harness creation, in which no tool can automate data-driven tests. No wonder that testing is typically viewed negatively by developers. Again, it's not that anyone feels testing is unimportant. But how repetitive and work-intensive!

To make matters worse, without code coverage the best tests in the world are run in a vacuum. How do you know when you are finished? How do you know what test cases have been overlooked?

Use Rational Test RealTime to simplify your component testing of Java classes. All the tedious tasks are automated so you can focus on good tests. Test boundary conditions. Try inputs that would "never" happen. And let the test scripting API generate an overabundance of inputs; why not, considering no additional effort is required on your part.

Perhaps now you can see how Rational Test RealTime, combined with the runtime analysis tools reviewed in the last group of exercises, provides you with full regression testing capabilities without having to sacrifice time better spent creating quality code.

Conclusion

Regression Testing

Regression testing involves the reuse of all tests to ensure your software experiences no regression - in other words, to ensure that the repair of one defect doesn't break some other feature that worked in the past. Frankly, software testing would be much simpler if nothing ever broke once it worked properly. Even manual testing efforts would be acceptable for some since the effort would only be focused on "new" code - a lot of testing at the beginning, but decreased testing as the development cycle matures and no new features are added into the project.

But things do break and manual testing is far from an achievable goal. Software is just too complicated and too interdependent to succeed without automated assistance.

With Rational Test RealTime you can create full regression tests that are comprised of all the testing and runtime analysis nodes created throughout your testing effort. It's as simple as creating a Group node and then copying and pasting your test and analysis nodes within it. Run the Group node as you would any other; every test and analysis node would (optionally) build and execute. When the Group execution has finished, a double-click on the Group node opens consolidated reports that let you easily determine where errors have been detected.

With regression testing you close the loop. Code might break, but it will never find its way into the finished product.

Proactive Debugging

As software complexity increases, developers must become more responsible for their contribution to the overall development project. It is becoming harder and harder for the developer to consider robust, end-to-end testing of their code an unachievable luxury.

In fact, developers need to proactively debug - that is, treat testing as an integral part of the development process, rather than waiting for defects to force their hand.

And why should this not be achievable? The advantage of proactive debugging is that it is manageable - testing is only performed on the code known intimately well by the developer (barring the case of inherited code, where the runtime tracing feature plays such a crucial role). There is little chance for confusion, so the time spent developing and deploying tests are optimized. Defects are eliminated early, ensuring that any system level defects that have slipped through the nets won't find their origin deep in the code. And test independence - due to the Target Deployment Port technology - ensures test reuse despite changes in target architecture.

Matters improve further when one considers the built-in integration that Test RealTime possess with other products in Rational's software development arsenal. Test RealTime is integrated with:

- Rose RealTime - Access all runtime analysis functionality from within Rose RealTime, the embedded industry's most robust UML-based code generation tool for the embedded space. Whether using RQA-RT to test your model or whether you simply wish to execute Rose RealTime generated code, get runtime analysis data traceable to the implemented use case. You can even visualize model coverage via color-coded state machines. Click here for access to the Rational Rose RealTime website.
- ClearCase - Out-of-the-box integration with ClearCase, the industry's clear market leader for version control software. Click here for access to the Rational ClearCase website.
- ClearQuest - Out-of-the-box integration to ClearQuest, the premier change management utility for diversified software teams. Submit context-sensitive defect reports directly from the Test RealTime interface. Click here for access to the Rational ClearQuest website.
- TestManager - Establish full traceability between a product requirement (stored in tools such as Rational RequisitePro), the test case for that requirement and the Test RealTime test implementing the test case. Ensure that when a test fails, you know which product feature has yet to be properly implemented; know which tests must be updated when features inevitably change.
- Rational Unified Process - Tool mentors help you implement various features of Test RealTime, conceived in the RUP framework - a mature, field-tested guide to the software development process. Click here for access to the Rational Unified Process website.

Questions?

Questions or comments? Want to share tips? Feel free to send us an e-mail at testrt-info@rational.com. Useful information will be shared on the Latest News and Updates page, accessible to Test RealTime customers from the Help menu in Test RealTime.