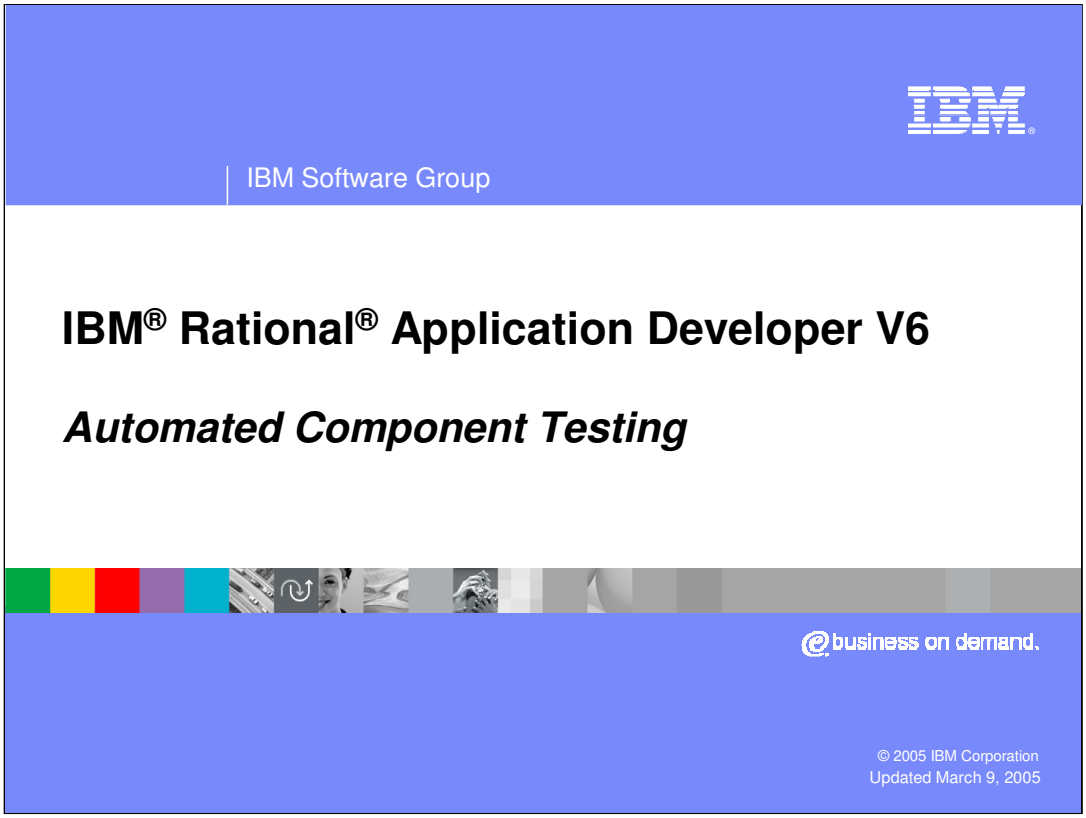
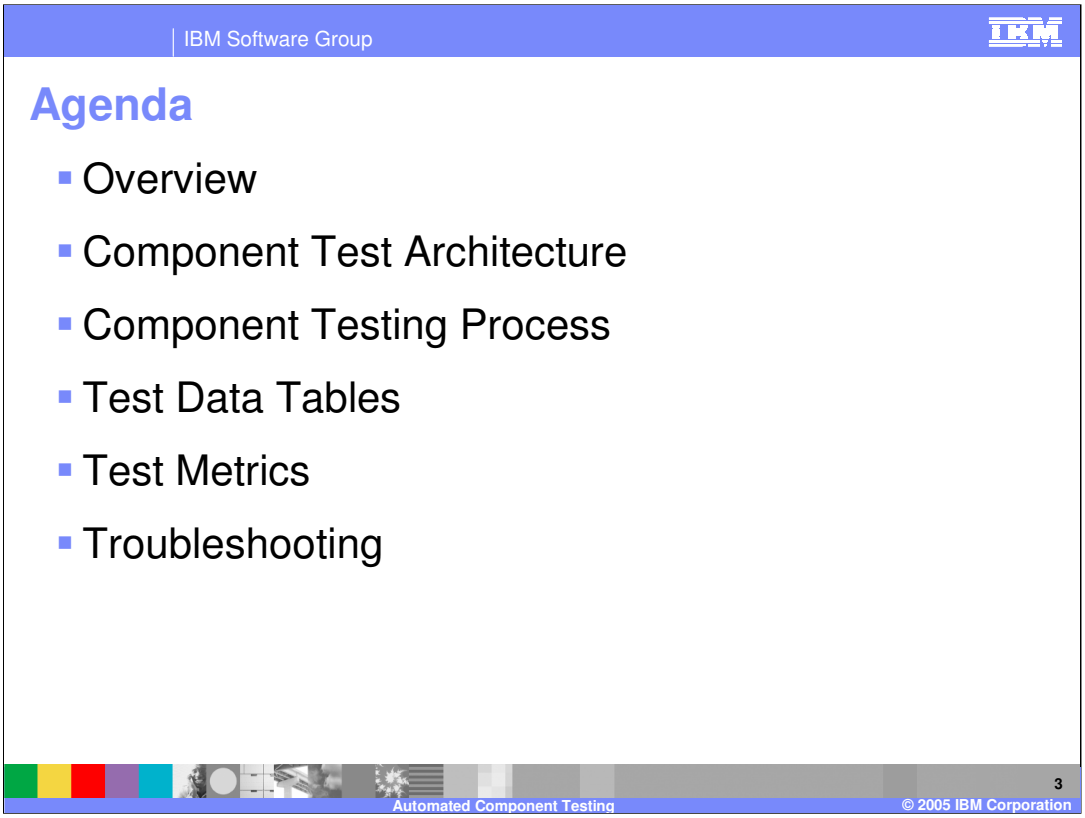


This presentation will focus on IBM Rational Application Developer V6 Automated Component Testing

Goals

- Understand Automated Component Test features
- Learn about the various Component Test artifacts
- Learn how to create and execute a component test

This presentation covers the Automated Component Testing function in the IBM Rational product suite. In this module, you will learn the features that are provided in Automated Component Testing. You will also learn about the various artifacts used in the testing process, and how to create and execute a component test, including defining test data and using test metrics to help guide you in building a comprehensive test plan.



This presentation will also provide an overview of Automated Component Testing, including the features of Component Testing and the typical process to use when creating and executing a test. It will also cover the usage of Test Data Tables and how to enter data for your tests. Test metrics and how they affect your test plans will also be discussed. Finally it will review some typical problems you might encounter.

There will also be two demonstrations, one showing you how to create and run a basic Java test, and another showing you how to create a test for a stub.

Overview

- Automated Component Testing is a feature in IBM Rational Application Developer and IBM Rational Software Architect
- Benefits include speed and quality
- Supports Java, EJB, Web Services
- Compliant with the OMG Testing profile and uses the JUnit testing framework and the Hyades open source project

Automated Component Testing is included in IBM Rational Application Developer and IBM Rational Software Architect. It is not included in the IBM Rational Functional Tester, IBM Rational Manual Tester, nor the IBM Rational Performance Tester.

Automated component Testing will speed the development of an Enterprise Application by enabling you to find and fix defects early in the development cycle, and by automating regression testing. It should also help you improve the overall quality of your application.

Developers can use this feature to create, execute and maintain unit tests for their Enterprise Application components, including

Java classes

EJBs (1.1, 2.0, and 2.1, both local and remote interfaces)

J2EE and .NET based Web Services

Automated Component Testing is compliant with the Object Management Group (OMG) UML testing profile, so many of the concepts and the terminology that you will see here are defined in the modeling specification for the UML testing profile which you can find at www.omg.org. The UML Testing Profile defines a language for designing and specifying the artifacts of test systems. OMG is an open membership, not-for-profit consortium that produces and maintains computer industry specifications for enterprise applications.

Automated Component Testing uses the JUnit testing framework which is a regression testing framework that can be found at www.junit.org. It is open source software that is used by the developer implementing unit tests in Java.

Automated Component Testing uses Hyades which is an open source integrated testing environment, based on Eclipse that provides standards and tools for the testing process. You can find out more at www.eclipse.org/hyades.

Features

- Testing Guidance
- Use of test patterns
- Data Driven Testing
- Stubbing
- Automated Test Deployment and Execution
- Import existing JUnit tests
- Integrated with Profiling
- Support for re-factoring
- Management of test assets

Automated Component Testing gives several views of the Enterprise Application components to optimize the unit testing phase.
Metrics are provided to help focus testing effort on the right components:
Structural metrics highlight dependencies between components.
Complexity metrics highlight intrinsic component complexity.
Coverage metrics assess the completeness of the tests.

Use test patterns to automatically generate unit tests.
Alleviates the need to write test scripts from scratch.
Helps you define more complex and complete unit tests.

Data Driven Testing
Data is externalized from the script into a Test Data Table (TDT).
Accelerates the creation and maintenance of tests, and allows you to re-use test scenarios with multiple sets of data.
Automates the regression testing process.

Stubbing
Stubbing allows you to replace one or more components with user defined stubs.
Enables you to test earlier in the development cycle, even if not all the components are ready.
Allows you to run tests without having to set up a database.

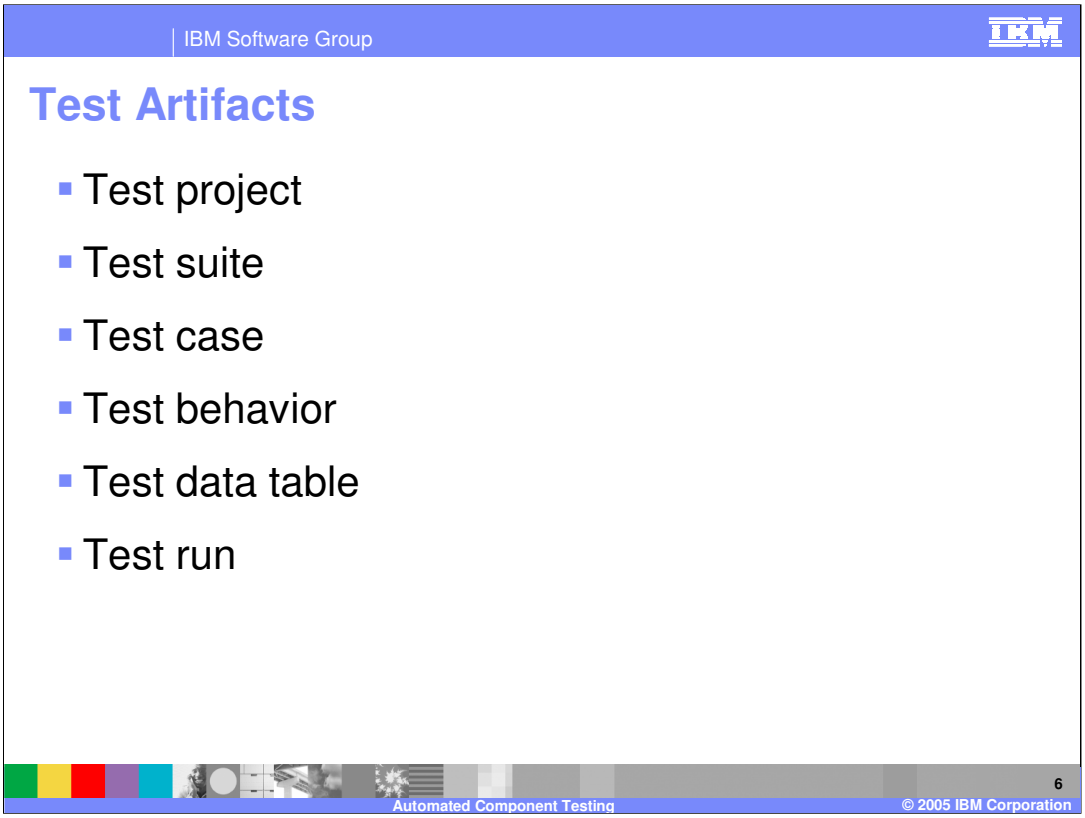
Automated Test Deployment and Execution
Using Automated Component Testing, you get complete automation of regression testing, eliminating the need to manually review the test results.
You also get automatic deployment of test components including EJBs and Web Services on the application server.

You can import existing JUnit tests, and automatically extracted data from the script for insertion into the Test Data Table.

Integration with Profiling allows you to engage any of the profiling features during test execution, such as code coverage measurement, or performance profiling.

Support for refactoring means you can update method signatures or method names and component tests will automatically be updated to reflect these changes.

Management of test assets provides
Integration with a repository of your choice, including ClearCase or CVS.
Integration with ClearQuest, enabling you to submit defects or feature requests directly from a test report. This requires installation of the ClearQuest plug-in.



The test project contains your component tests and test runs. You create this first to store the rest of the test artifacts.

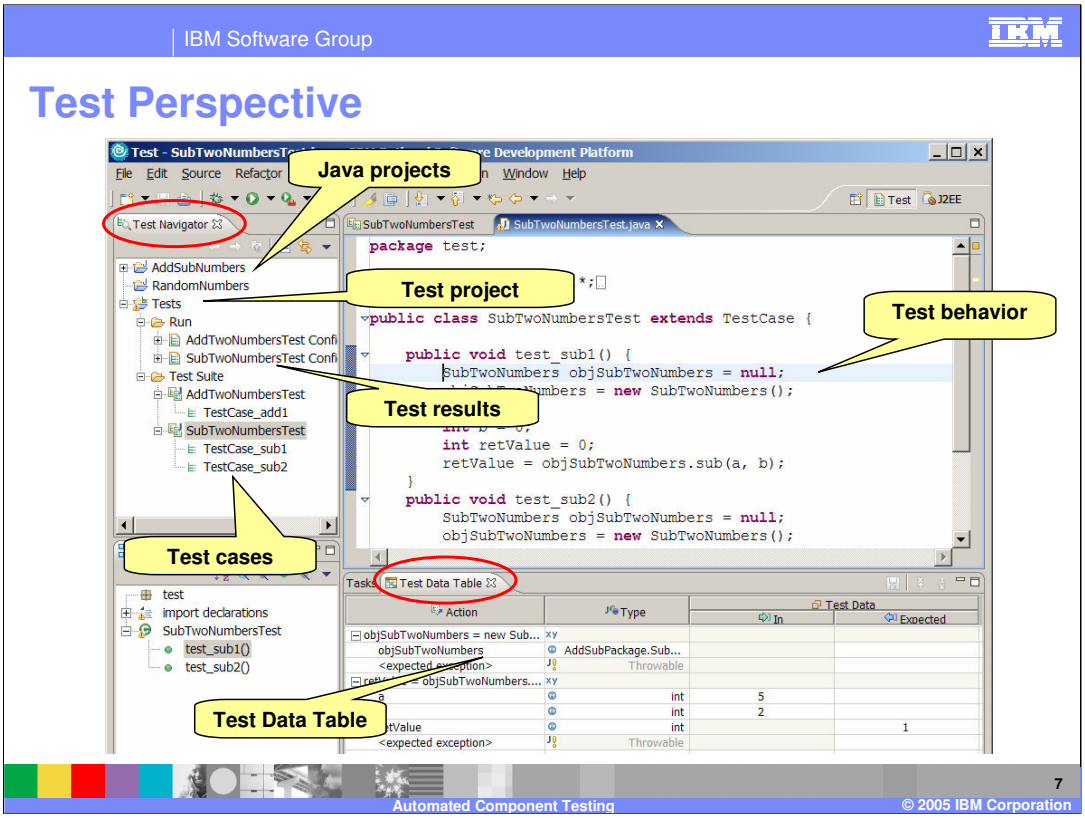
The test suite is a collection of test cases in a component test. Each time you run the Component Test wizard, it will add a test suite to the Test Suite folder in the test project.

A test case defines the application components being tested.

The test behavior is the Java code that is created automatically by the tool which defines your test cases. This may also be called the test script.

The test data table is where you place the input and output values for the test case.

A test run is the execution of the component test, and it contains the test results with verdicts.



Most testing is done from the Test Perspective.

The test navigator view shows you the list of test artifacts, including the Java projects and Test projects.

There are several data views including the test data table, stub data table and test data comparator. This example shows the Test Data table.

The component test editor is used to manage properties of a test suite.

The Java editor is used to edit your test behavior files.

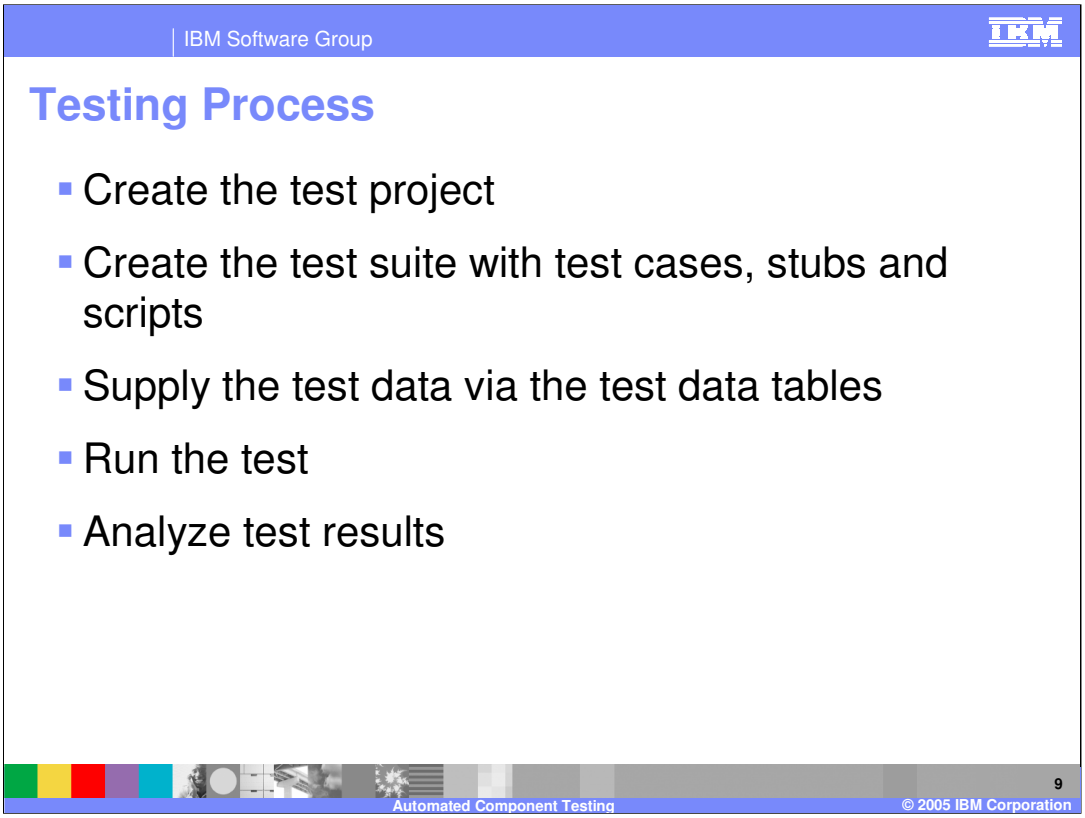
The test run editor is used to view the results of the test execution.

Java files and some test folders are not displayed in the Test Navigator so you might want to add the Package Explorer to the Test Perspective. You could also add the test views to the Java perspective.

Automated Component Testing: Demonstration

- Open the Test Perspective
- Create a Test project
- Create a Component Test
- Update the Test Data Table
- Run the test
- Check the verdict

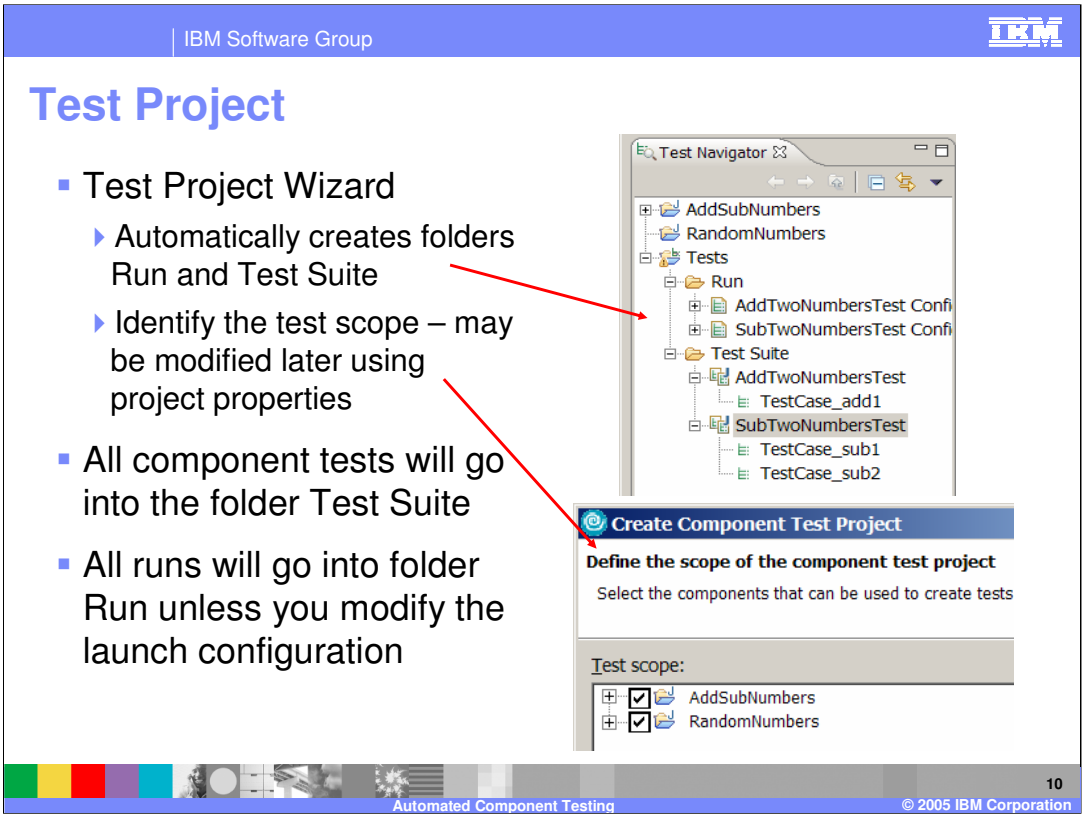
Click the Show Me icon for a demonstration that will show you how to perform Automated Component Testing. This demonstration will show opening the Test Perspective, creating a test project, creating a component test, adding data to the Test Data table, running the test, and checking the verdict.



This slide shows you the basic process of component testing, which also maps closely to what you saw in the demonstration.

First, you create the test project which will hold your test artifacts. Then you create the component tests or stubs for your test cases. Next, you add your input data and expected values to the test data tables.

After you run a test, you use the test data comparator to check for verdicts. You can also generate HTML reports of the test run results. To do this, use menu option File > Export > Component test HTML report.



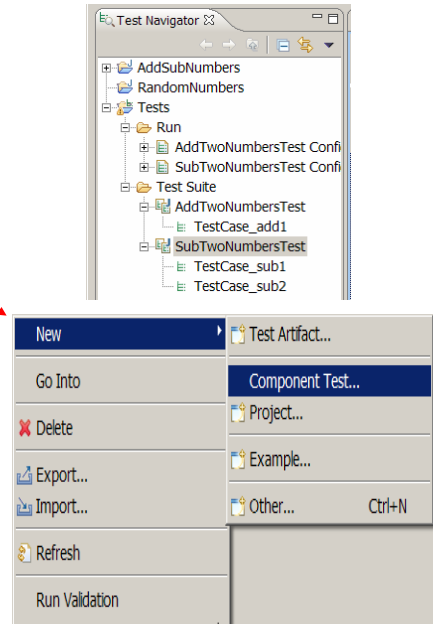
The Test Project will contain your test artifacts, including the component tests, execution results and test behavior scripts. Use the Project Wizard to create your test project. The wizard automatically creates the Run folder and Test Suite folder. All component tests will go into the Test Suite folder. All test executions will go into the Run folder, unless you modify the launch configuration and select a different folder.

The wizard will prompt you to define the scope of the project, so you can pick the Java projects that you want to have associated with this test project. You can change the scope later using the project properties dialog.

This example shows a test project called Tests. In the Run folder there are two test executions, and in the Test Suite folder there are two test suites created.

Test Suite

- The folder Test Suite contains your component test suites
- Use the Component Test wizard to create a new component test
- Do not use the Test Artifact menu option for creating artifacts for Automated Component Testing

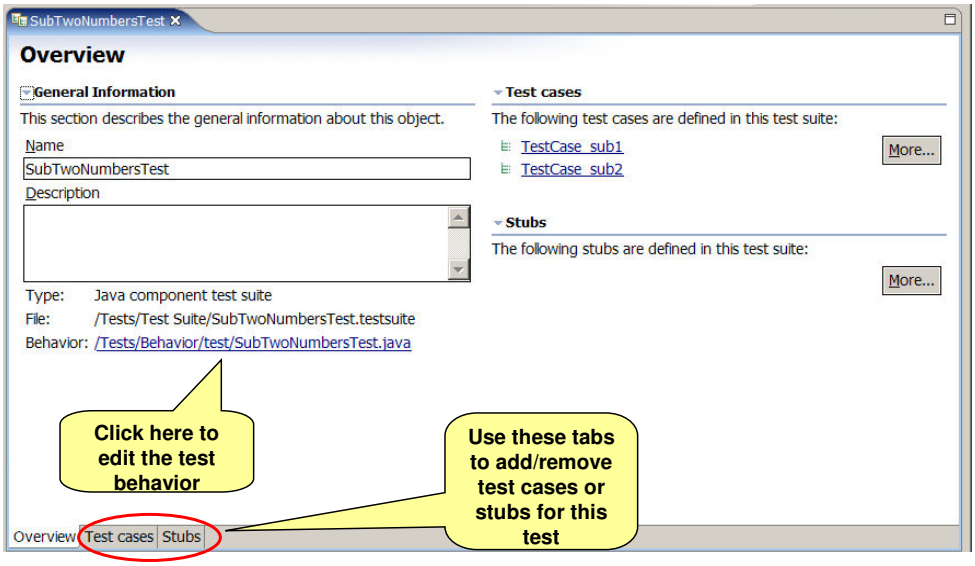


The Test Suite folder contains your component test suites, which you create using the Component Test wizard.

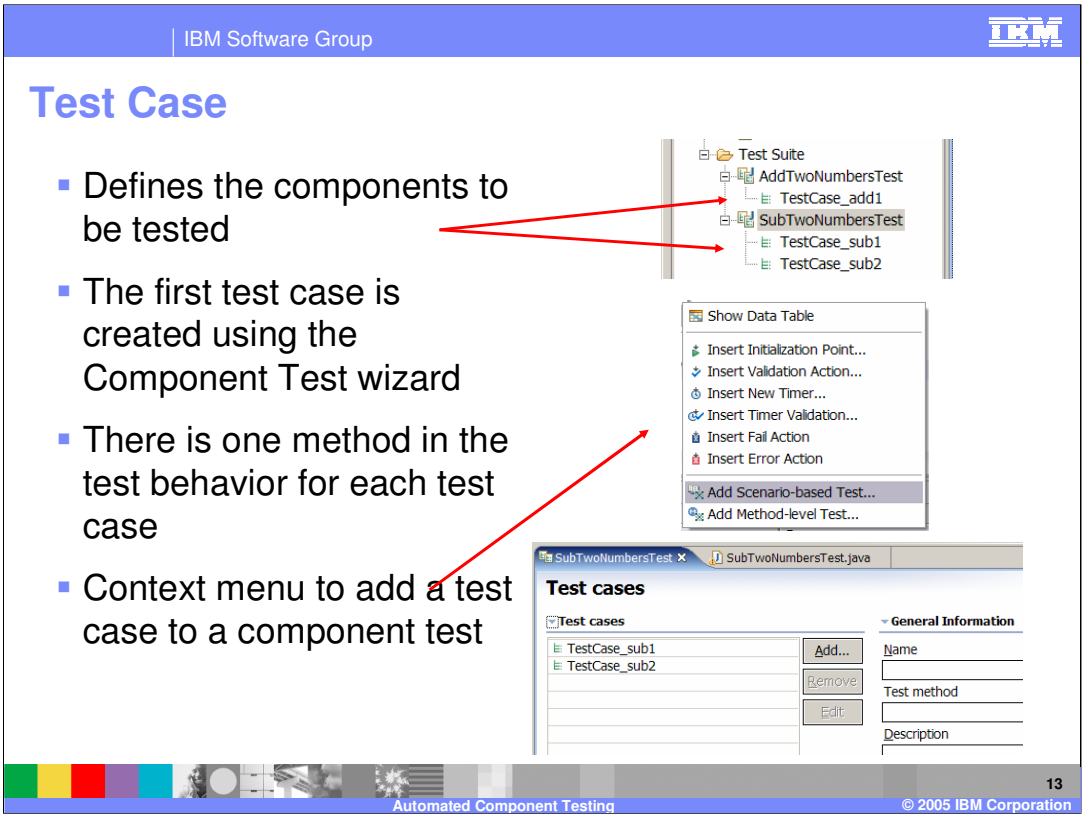
This example shows the Test Suite folder which contains two test suites.

To start the Component Test wizard, use the context menu in the Test Navigator. This slide shows the context menu. You should not select the 'Test Artifact…' menu item for creating your Automated Component tests. That option is primarily used for creating tests using the JUnit testing framework.

Test Suite Editor



Here is an example of the Test Suite Editor. It has a link to the behavior file, so you can click it to bring up the Java editor on the behavior file. You can also edit your test cases and stubs that are defined for the test. In this example you see that there are two test cases and no stubs defined. Use the tabs or the More buttons to edit them.



A test case defines the components to be tested. In this example you see one test case defined for one test suite and two test cases defined in another test suite. All of these are stored in the Test Suite folder in the test project.

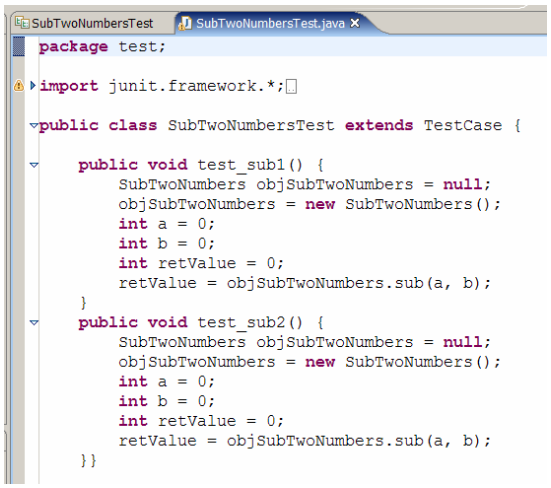
You create the first test case using the Component Test wizard. This wizard automatically creates a test script which will contain one method for each test case.

After you have created a test case, you might want to add another test case to a component test. To do this, bring up the test behavior script, and then use the context menu in the test behavior editor. Here is an example of the context menu, showing that you can then add a scenario- based test or a method-level test.

The third example shows the test suite editor which lists all the test cases in the test suite, along with their descriptions.

Test Behavior

- Java code defining the test
- Automatically created but you can modify it
- There is one behavior for each test suite
- One method for each test case



The Test Behavior artifact is the Java code that defines the test. The Component Test Wizard creates this for you automatically, but you can modify it.

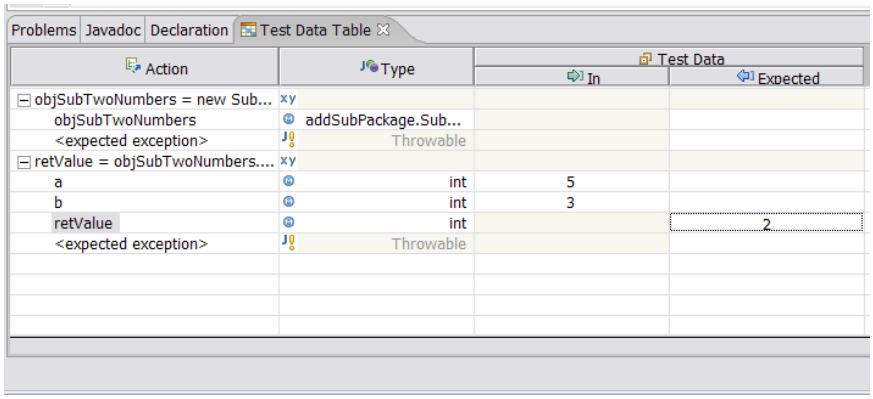
There is one behavior for each test suite, and one method in the behavior file for each test case.

This example shows you the test suite SubTwoNumbersTest which contains two methods for the two test cases in the suite. Notice that the behavior is implemented as a subclass of the JUnit TestCase class.

Also, the behavior file is stored in the Behavior folder in the test project, but this folder is not viewable using the Test Navigator. Add the Package Explorer to the Test Perspective is you wish to see them. However, even without the Package Explorer, you can access the behavior file using a link in the Test Suite editor.

Test Data Table

- Defines the inputs and outputs for the test
- There is one test data table for each test case

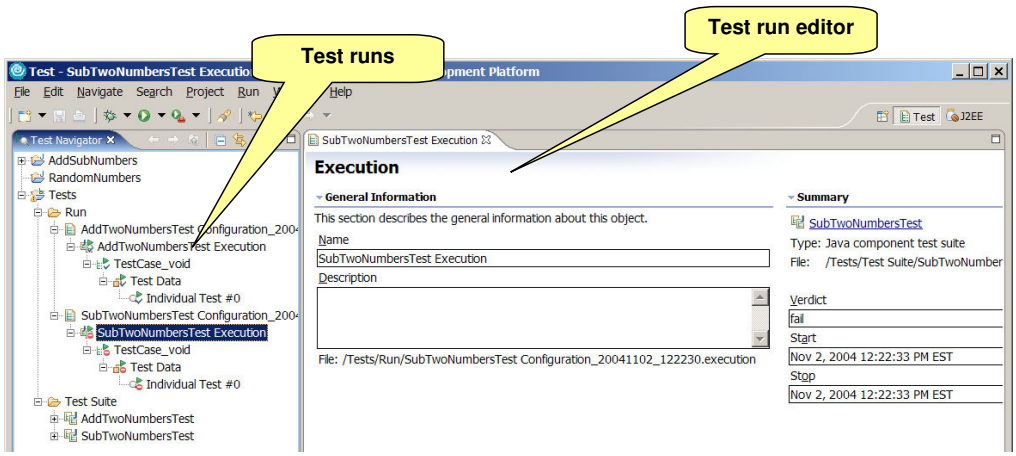


The Test Data Table defines the inputs and outputs for the test. You will edit this table and add the inputs and the expected outputs for the test. The outputs are really the expected values that will be used as verification points, which are used to provide a verdict. This example tests a method which subtracts two numbers and returns a result. So in the test data table, the input values are 5 and 3, and the expected result of the subtraction is 2.

Click inside a method in the test behavior to show the table corresponding to the test case represented by the clicked method. If the table is not visible, use the context menu in the behavior and select Component Test > Show Data Table

Test Run

- Contains the results of the execution of a test
- Default location is in the folder Run in the test project



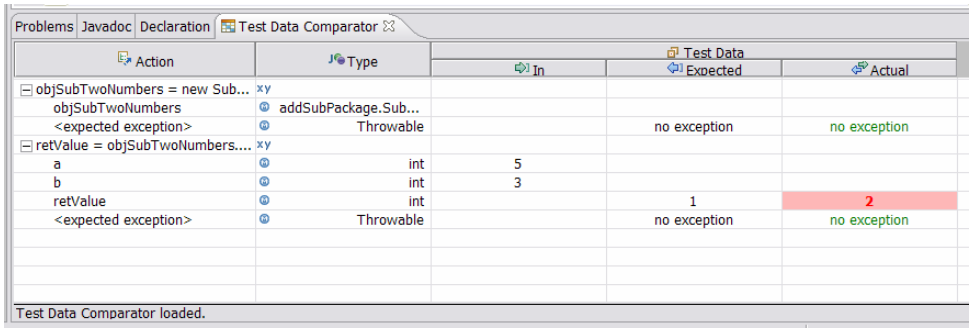
The test run contains the results of the execution of a test. The default location is in the Run folder in the test project.

This example shows the test run editor which shows the test suite and the verdict. The verdict is also given in the Test Navigator at each level with icons so that a pass verdict is indicated with the green checkmark and the fail verdict is indicated with the red error icon.

Notice that under a run node, there are several levels of nodes. The first one is the test suite, then the test case, then the data set, and finally the individual tests. There may be more than one individual test if the test includes multiple data sets, or if it uses ranges or sets in the test data table.

Test Data Comparator

- Shows the actual results of each method invocation in the test case



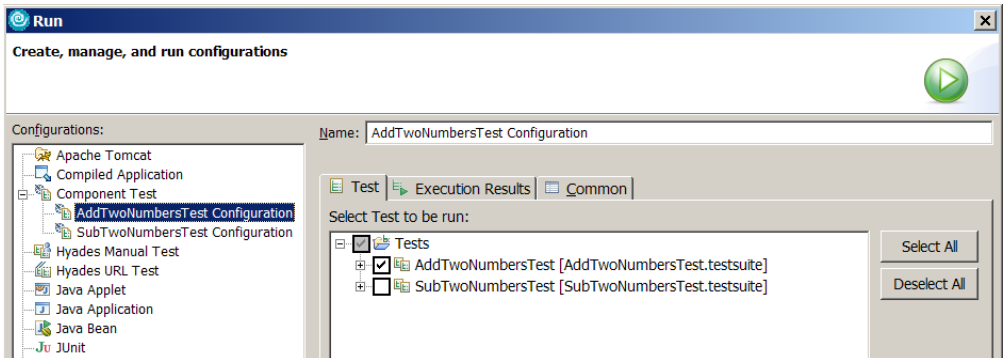
The Test Data Comparator shows the actual results of each method invocation in the test case. Passing results are listed in green, and failing verification points are highlighted in red. Exceptions are highlighted only if not expected.

In this example, the routine subtracts two numbers and returns the result. The test case supplied numbers 5 and 3 as inputs, and the expected output was entered as a 1. The method actually returned a 2, so this is listed as an exception. In fact, the method worked properly. It is the test data that is in error.

This is accessed by opening the test run folder, then drilling down to an individual test and then double clicking on the individual test.

Running A Test

- Run an individual test case or full test suite
- Use launch configurations to manage the run
  - Specify tests to run in the 'Test' tab
  - Specify Run folder in the 'Execution Results' tab



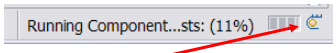
You can run an individual test case or you can run the entire test suite. Just click on the artifact of choice in the Test Navigator, the bring up the context menu and click the Run option.

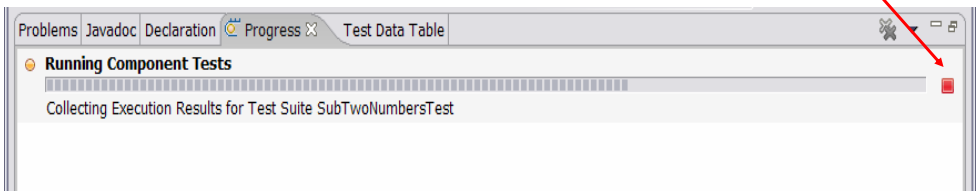
If you want to run the test immediately, the you can select Run > Component Test, and the launch configuration is created for you automatically.

If you want to create and edit the launch configuration then select 'Run…' in the context menu. Then you can pick the tests to run on the Test tab, and you can also pick the Run folder on the 'Execution Results' tab.

Progress Of A Test Run

- The test runs as a batch process
- Click here to open the Progress view
- Click the red 'Cancel Task' icon to stop the test





The component test runs as a batch process, so you can see the status in the lower right hand corner of the window. If you want to see the details of the batch run, then click on the progress icon and you will open up the Progress view where you can watch the test results. You can also cancel it if you desire.

Creating Tests For Java Components

- Java component test patterns
  - Method-level testing
  - Scenario-based testing
  - Tests for abstract classes, interfaces, superclasses

When creating tests for basic Java components, there are several testing patterns that you can choose from in the wizard.

Method-level testing

Use method level testing to test an individual method.

In this case, one test case is created for each method under test.

Scenario-based testing

Use scenario based testing to test a sequence of methods.

In this case, one test case is created for the sequence of methods under test.

Tests for abstract classes, interfaces, superclasses

Use this option to create an abstract test.

These tests cannot be run until they are made concrete with an implementing class.

When you create the abstract test, there will be one test behavior for the abstract test but no test suite. When implemented, there will be a test suite containing the test case and the behavior for each implementation of the abstract test.

You can make the abstract test concrete at the time you create the abstract test if there are implementing classes available. Or you can make them concrete at a later time. To do it at a later time, just start the Java Component Test wizard and pick the abstract test pattern.

Creating Tests For Java Component Stubs

- Used to guarantee behavior of a called component to ensure isolating the component under test
- Example: MyClass1 calls MyClass2
  - Create a stub for MyClass2
  - Create a test suite for MyClass1
  - Add the stub for MyClass2 to the test suite for MyClass1

A stub is defined as a class that provides a replacement implementation for the actual classes that the code you are testing interacts with.

You use a stub to guarantee the behavior of a called component to ensure component isolation under test.

For example, if MyClass1 calls MyClass2, you would create a stub for MyClass2, create a test suite for MyClass1, and add the stub to the test suite for MyClass1. Then, when you run the test for MyClass1, MyClass 2 is not called, but the stub is used instead to create return values.

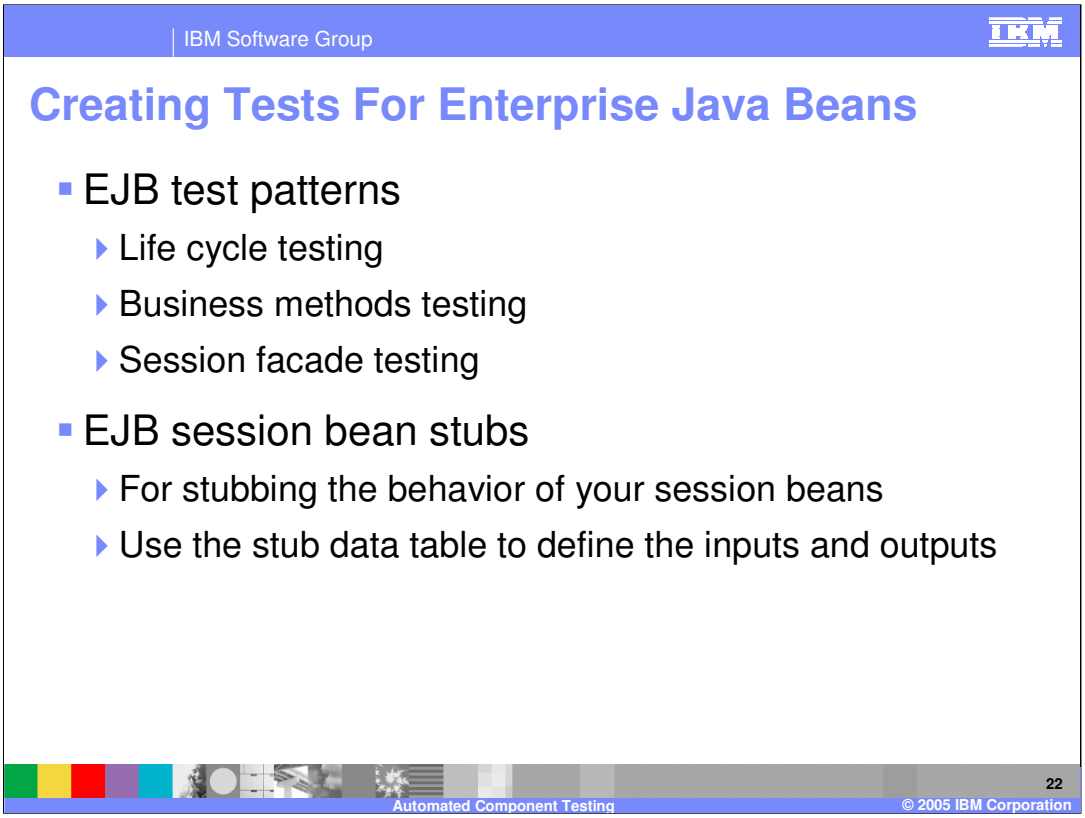
A stub is defined by behavior and data.

The behavior is stored in the stub folder of the test project, viewable in the Package Explorer but not the Test Navigator.

Use the stub data table to simulate the stubbed class by specifying the input and return values of the methods in the stubbed class.

After creating the stub, add it to the test suites that require it.

Test cases that call the stubbed method will automatically use the stubs defined in the suite.



You can also create component tests for Enterprise JavaBeans. You can create tests for session beans and entity beans, and you can test them using their local or remote interfaces. You can also create stubs for your session beans.

When you run the EJB test wizard, you can choose from several test patterns.

You can choose lifecycle testing for stateless session beans, stateful session beans and entity beans.

Test scenarios include creation, finding, setting state, checking state, and removal of EJBs.

You can also test the business methods for your EJBs. Just like Java component testing, you use the test data table to define your input data and expected results.

In session facades, a session bean wraps a subsystem of entity beans. When you use the EJB component test wizard, you will be prompted to select methods to test for the façade and also for the entity beans behind the façade.

Just like Java component test stubs, you can create stubs for your EJB session beans. You will create the stub behavior for the stubbed EJB, and also enter your data in the stub data table, and finally add the stub to the test suites that you want to use them.

Session bean stubs are supported only on WebSphere Application Server V5 servers in this release. Entity bean stubs are not yet supported but will be in a later release.

Automated Component Testing also supports Web services. You can test a Web service client or a Web service server.

To test a Web service client:

Create a stub for the service using the Web Service Component Stub Wizard.

Use the WSDL to automatically create the stub behavior in the Stub folder.

Set up responses for client requests in the Stub Data Table.

Create a test behavior and test data table for the client.

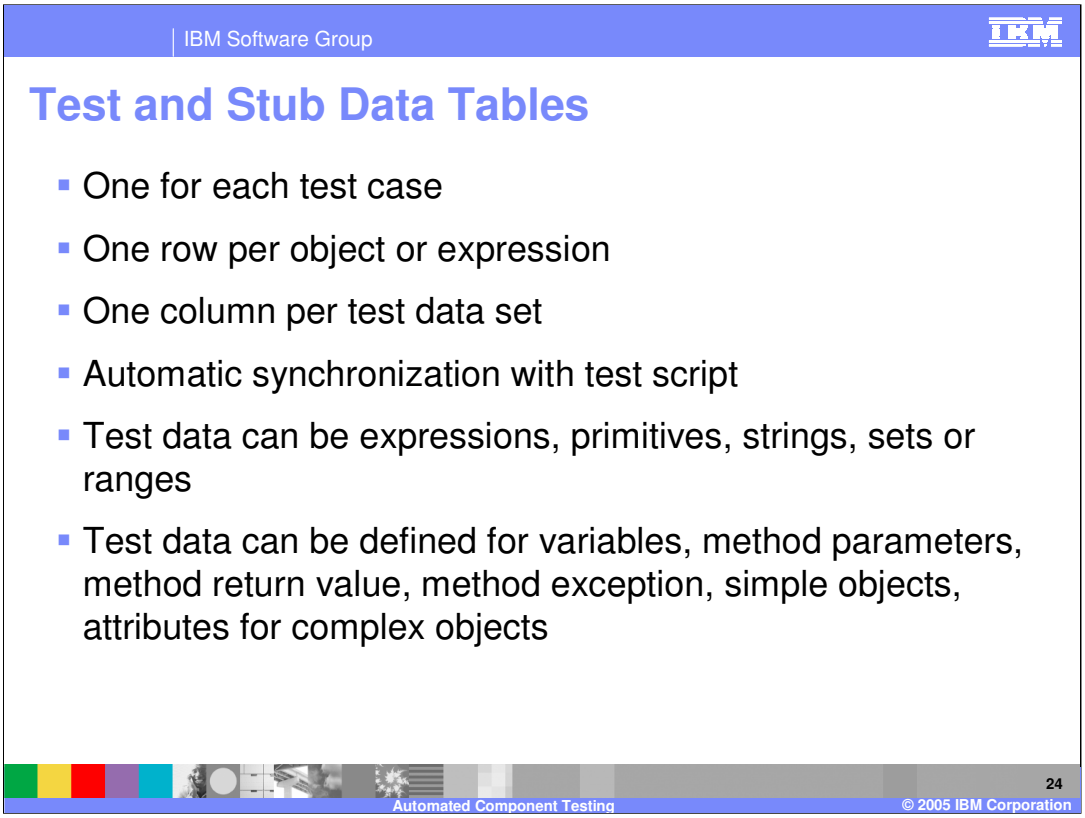
Replace the service URL in the client with the service stub URL.

To test a Web service server:

Create a test suite for the service using the Web Service Component Test wizard.

Use the WSDL to automatically create a test client in the JavaSource folder of the test project.

Set up input and expected return values in the Test Data Table.



Test Data Tables (TDT) and Stub Data Tables are used to provide inputs and expected outputs of the components under test.

There will be one TDT for each test case.

There will be one row in the TDT for each object or expression in the test script.

Each column in the TDT represents one test data set. For each data set, you will get a separate individual test result in the Run folder after execution of the test.

There is automatic synchronization with the test script, so the script is automatically modified as you make changes to the TDT.

Test data can be expressions, primitives, strings, sets or ranges. Test data can be defined for variables, method parameters, method return value, method exception, simple objects, attributes for complex objects.

When using a set or a range for input values, you create multiple tests from a single data set. Each test execution run represents one value from the range or set. Therefore you must be careful using this feature, because you could easily create numerous tests. This is especially true when using more than one set or range in a dataset, because the number of test executions is equal to the number of possible combinations of all sets and ranges in the test case.

Test Data Table Elements

- Expressions – 10, “myString”, myObj.method()
- Sets – {“item1”,”item2”,”item3”}
- Ranges – [100..1000]/50
- Validation actions
- Initialization points
- Timing constraints

This slide describes elements in a Test Data Table (TDT).

Any valid Java expression that can appear as the source of an assignment statement can be used. This could include numbers, strings in double quotes, variables, arrays, constructor calls and method calls. Other examples include true, null, and java.io.InvalidObjectException. For arrays, you can use one dimensional arrays or multidimensional arrays, and to add elements to the array, each row in the table will represent a unique element in the array.

A set can be used for defining multiple inputs or outputs in a test. The values can be any data type including objects.

A range can be used to supply multiple incremental numerical values. In the example, values are generated starting at 100, incrementing by 50, until a maximum of 1000 is reached.

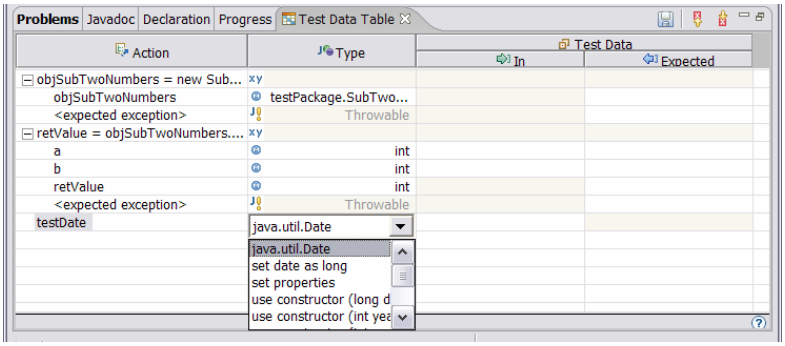
A validation action is used in a TDT to validate the value of a variable. When you create the validation action, you pick from a list of variables previously defined in the test script, and then you provide an expected output value in the TDT.

An initialization point is used to save the value of an object, for later reference in the TDT. When you create the initialization point, you define a new variable and the value to be assigned to it.

A timing constraint is used to measure the duration of a method call or a sequence of method calls. You will see two rows in the TDT for it, one to initialize it and the second one to measure the elapsed time. You can enter a value such as ‘>= 5 seconds’ to check the elapsed time. You can also use milliseconds, nanoseconds, minutes, hours or days.

Defining Attributes for Complex Objects

- In this example, several sets of defining attributes are provided for the object of class Date
  - The first set (set date as long) is from the specialized support for the class.
  - The second set (set properties) is from the properties of the JavaBean.
  - The remaining sets are from the available constructors.



For complex objects, you can pick from several different sets of attributes to provide the value of the object. When you click on the type column in the TDT for a complex object you will be presented with several options:

The available constructors for the object.

The properties for a JavaBean object.

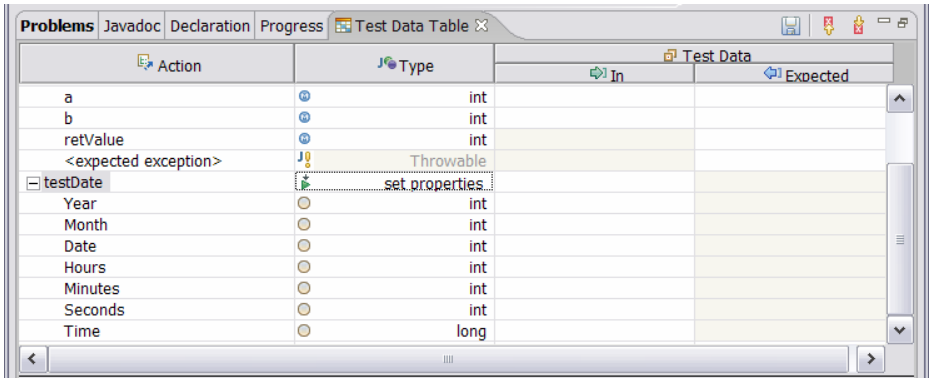
The specialized Component Testing support for the class.

In this example, you see the different options presented for the date object.

When complex objects are compared in a validation action, the way the object is initialized determines the way the comparison works. For objects initialized with constructors, the equals method is used. For objects initialized with properties, each property is compared. For objects initialized with specialized classes, the same specialized support is used for comparison.

Defining Attributes: Example for 'Set Properties'

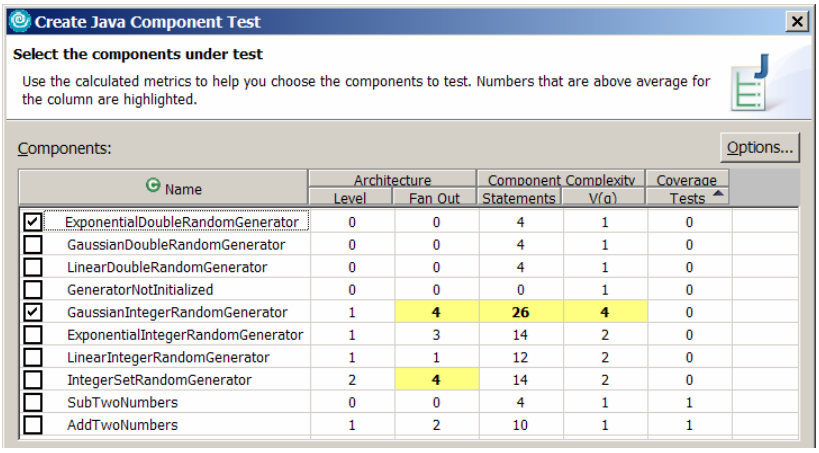
- After selecting 'Set properties' for the Date variable, the Test Data Table is populated with the properties for the bean.



Here is the result of selecting 'Set properties' for a date object. Notice that each property is listed on a separate row in the TDT. This makes it easy to set the value of each of the properties in the TDT.

Test Metrics

- Displayed in the wizard for creating component tests
- An aid to assist you in selecting components to test

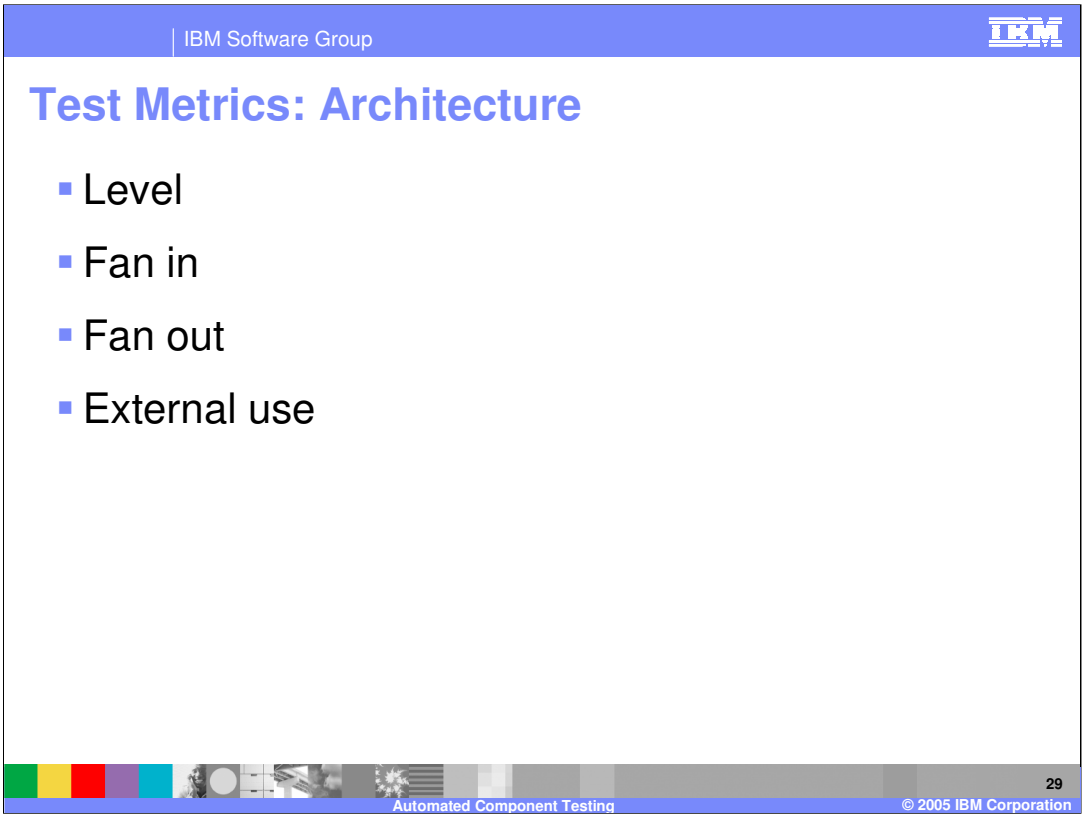


Name	Architecture		Component Complexity		Coverage
	Level	Fan Out	Statements	V(g)	Tests
ExponentialDoubleRandomGenerator	0	0	4	1	0
GaussianDoubleRandomGenerator	0	0	4	1	0
LinearDoubleRandomGenerator	0	0	4	1	0
GeneratorNotInitialized	0	0	0	1	0
GaussianIntegerRandomGenerator	1	**4**	**26**	**4**	0
ExponentialIntegerRandomGenerator	1	3	14	2	0
LinearIntegerRandomGenerator	1	1	12	2	0
IntegerSetRandomGenerator	2	**4**	14	2	0
SubTwoNumbers	0	0	4	1	1
AddTwoNumbers	1	2	10	1	1

Test metrics are displayed in the wizard when creating a new component test. These are used as an aid to measure the impact of the test and to help you define a test strategy. There are three categories of metrics which will be described on the next slides.

In this example, you see the default metrics. To add more, click on the Options button.

The yellow color coding is used to indicated the cells that are above average for their respective columns.



Level - indicates the level of class dependency, for example,

0 – references no other classes in the test scope

1 – references one or more level 0 classes

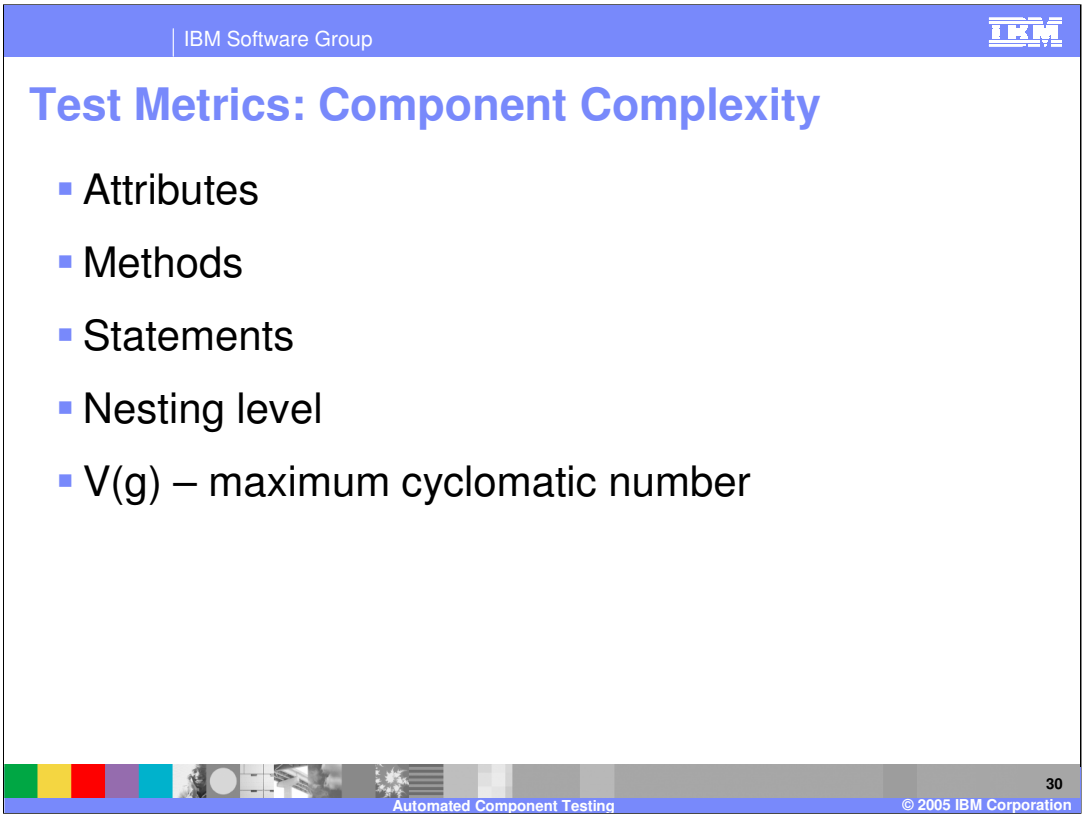
2 – references one or more level 1 classes

…

Fan in - the number of public methods and attributes of a class.

Fan out - the number of outside references to methods and attributes in a particular class.

External use - the number of classes with references to methods and attributes of the measured class.



Attributes - the count of attributes in a class.

Methods - the count of methods in a class.

Statements - the count of statements in a class excluding comments.

Nesting level – the maximum nesting of structures in a class, for example,

1 – no nesting

2 – if (…) { if (…) {} }

…

V(g) – the maximum cyclomatic complexity of any method in the class. For a method, count one for each decision point (if, for, while, case statement), and add one (for the entry point of the method).

Test Metrics: Coverage

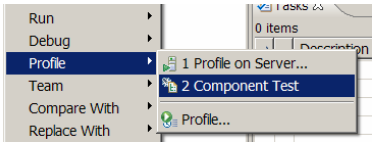
- Line(%)
- Tests

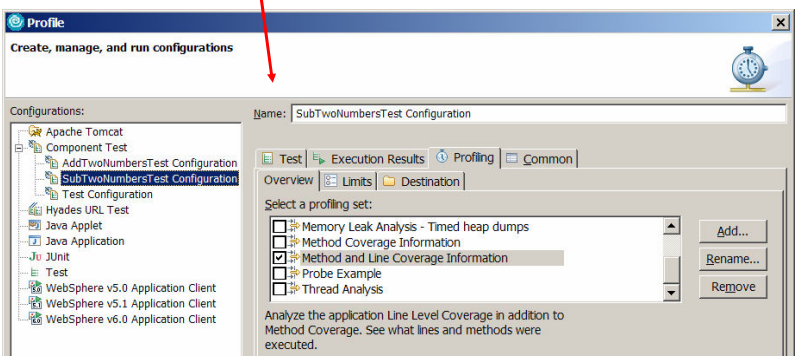
Line(%) – the percentage of the number of lines in the class that are covered by test runs in the workspace. To enable this you must run your tests with profiling using the profiling set for method and line coverage.

Tests – the number of times the class is referenced in test cases in the workspace.

Profiling a Component Test

- Use Profile menu
- Select your profiling set
- Review results



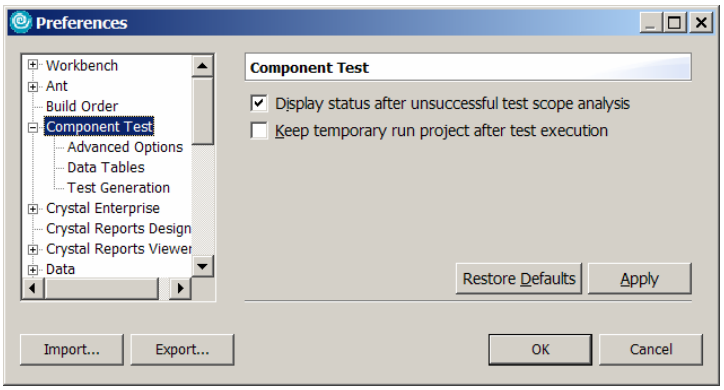


It is easy to use profiling while you run your component tests, so you can have all the usual profiling features at your disposal. It is not required that you use profiling, but you should if you plan on using the line percentage test metric. If you do not use profiling, this test metric will be zero.

To use profiling, select a test suite in the Test Navigator, then bring up in the context menu. The example above shows the profile menu. If you have already set up your profiling sets in a configuration, you can select Profile > Component Test. If not, select Profile > Profile…, and you will see the Profile dialog as above. This looks like the normal Run configuration dialog that you have seen for Component Testing, except with the addition of the Profiling tab, where you can select the profiling sets. This example shows the selection of Method and Line Coverage Information, which will ensure that the line percentage test metric is calculated. When you run this profile, the perspective is switched to the Profiling and Logging Perspective where you can see the results of the profiling sets that you have selected.

Preferences

- Advanced Options
- Data Tables
- Test Generation



Use the Advanced Options page to enable or disable statements in data tables. If you enable statements such as if, for and while, then the variables used in those statements will be represented by rows in the data tables and you will be able to set input and output values for them. If you do not enable them, you will not be able to set values for variables used in the statements.

Use the Data Tables page to set the styles of cells in your data tables, including font and color.

Use the Test Generation page to set the default values for your generated tests, including the names of your tests and packages. For Web Services you can also specify the runtime as IBM WebSphere or Apache Axis.

Supported Application Servers

Application Server	WTE	WAS	WTE	WAS	WAS/WTE	Apache Tomcat		.Net
Version	5.02 (?)	5.02 (?)	5.1	5.1	6.0	4.1	5.0	2003
Test Web Services	✓	✓	✓	✓	✓	✓	✓	✓
Stub Web Services	✓	✓	✓	✓	✓	✓	✓	N/A
Test EJBs Remote Interfaces	✖	✓	✖	✓	✓	N/A	N/A	N/A
Test EJBs Local Interfaces	✖	✓	✖	✓	✓	N/A	N/A	N/A
Stub EJBs Session	✖	✓	✖	✓	✖	N/A	N/A	N/A
Stub EJBs Entity BMP	✖	✖	✖	✖	✖	N/A	N/A	N/A
Stub EJBs Entity CMP 1.1	✖	✖	✖	✖	✖	N/A	N/A	N/A
Stub EJBs Entity CMP 2.0, 2.1	✖	✖	✖	✖	✖	N/A	N/A	N/A

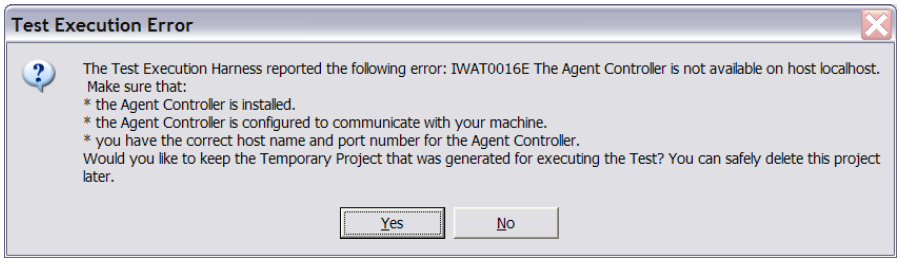
This charts shows which application servers are supported in this release for each major function in Automated Component testing.

The check marks indicate support in the current release.

The x's indicate not currently supported.

Troubleshooting

- Check if the Agent Controller (RAC) has been properly setup and is running



- Check the version of the application server on which the tests/stubs are deployed.

If you are having trouble, ensure that you have installed the Rational Agent Controller. See the appendix of this presentation for more information.

Also check to see that your server is supported.

Use the preference “Keep temporary run project after test execution”, and check that the project compiles successfully.

The folder names for these projects are similar to .cta_exec_200408031149199290 where the timestamp matches the time on the run results in the Test Navigator. Check for an error indicator on the folder in the Profiling or Resource Perspective.

Reference

- Help > Help Contents > Testing applications > Testing Java and enterprise application components

Check the Help Contents in the tool to find more detailed information about Automated Component Testing.

Installing Rational Agent Controller

- Run launchpad.exe in the folder <install files directory>\disk1
- Select 'Install Agent Controller' on the main menu
- When prompted for the Java runtime, specify <Install directory>\eclipse\jre\bin\java.exe
  - e.g., C:\IBM\RADBeta6.0\eclipse\jre\bin\java.exe
- When prompted for the location of the WebSphere Application Servers V5.1 and V5.0, leave both blank if using the integrated V6 server
- After installation, a service is created called 'Hyades Data Collection Engine', and it is automatically started
- To remove RAC, use Add/Remove Programs and select 'IBM Rational Agent Controller'

This slide provides details on installing Rational Agent Controller.

This concludes this presentation covering Automated Component Testing.

Trademarks, Copyrights, and Disclaimers

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both:

IBM	CICS	IMS	MQSeries	Tivoli
IBM(logo)	Cloudscape	Informix	OS/390	WebSphere
e(logo)business	DB2	iSeries	OS/400	xSeries
AIX	DB2 Universal Database	Lotus	pSeries	zSeries

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are registered trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel, ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks of Intel Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a registered trademark of Linus Torvalds.

Other company, product and service names may be trademarks or service marks of others.

Product data has been reviewed for accuracy as of the date of initial publication. Product data is subject to change without notice. This document could include technical inaccuracies or typographical errors. IBM may make improvements and/or changes in the product(s) and/or program(s) described herein at any time without notice. Any statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only. References in this document to IBM products, programs, or services does not imply that IBM intends to make such products, programs or services available in all countries in which IBM operates or does business. Any reference to an IBM Program Product in this document is not intended to state or imply that only that program product may be used. Any functionally equivalent program, that does not infringe IBM's intellectual property rights, may be used instead.

Information is provided "AS IS" without warranty of any kind. THE INFORMATION PROVIDED IN THIS DOCUMENT IS DISTRIBUTED "AS IS" WITHOUT ANY WARRANTY, EITHER EXPRESS OR IMPLIED. IBM EXPRESSLY DISCLAIMS ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NONINFRINGEMENT. IBM shall have no responsibility to update this information. IBM products are warranted, if at all, according to the terms and conditions of the agreements (e.g., IBM Customer Agreement, Statement of Limited Warranty, International Program License Agreement, etc.) under which they are provided. Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products in connection with this publication and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. IBM makes no representations or warranties, express or implied, regarding non-IBM products and services.

The provision of the information contained herein is not intended to, and does not, grant any right or license under any IBM patents or copyrights. Inquiries regarding patent or copyright licenses should be made, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

Performance is based on measurements and projections using standard IBM benchmarks in a controlled environment. All customer examples described are presented as illustrations of how those customers have used IBM products and the results they may have achieved. The actual throughput or performance that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput or performance improvements equivalent to the ratios stated here.

Note to U.S. Government Users - Documentation related to restricted rights-Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract and IBM Corp.