

JADE Test Suite

USER GUIDE

Last update: 12-January-2005 JADE3.4

Authors: Elisabetta Cortese (TILAB)

Giovanni Caire (TILAB)

Rosalba Bochicchio (TILAB)

JADE - Java Agent DEvelopment Framework is a framework to develop multi-agent systems in compliance with the FIPA specifications. JADE successfully passed the 1st FIPA interoperability test in Seoul (Jan. 99) and the 2nd FIPA interoperability test in London (Apr. 01).

Copyright (C) 2000 CSELT S.p.A. (C) 2001 TILab S.p.A. (C) 2002 TILab S.p.A. (C) 2003 TILab S.p.A. (C) 2004 TILAB

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, version 2.1 of the License.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

TABLE OF CONTENTS

1 INTRODUCTION3

1.1 Scope and target audience..... 3

1.2 System requirements 3

1.3 How to install..... 3

1.4 How to Compile and quick run..... 3

2 THE TESTSUITE FRAMEWORK.....3

2.1 Objective 3

2.2 The Test Suite Model..... 4

2.3 Main classes 4

2.4 TestGroup arguments 5

2.5 The TestSuiteAgent 5

2.6 XML Files 6

3 CREATING A TEST7

3.1 Test Suite directory/package structure..... 7

3.2 Developing a Functionality-Test 8

3.3 Adding an atomic-test for a specific aspect. 8

3.4 An example 9

4 RUNNING TESTS 10

4.1 Running Tests from the TestSuiteAgent GUI 10

4.1 Skipping tests..... 12

4.2 The Test Suite Daemon 12

1 INTRODUCTION

1.1 Scope and target audience

This document describes the JADE Test Suite that is distributed as a JADE add-on since March 2004. The JADE test suite is a framework designed to test JADE-based agent systems. We decided to develop a new testing framework from scratch and not to take advantage of commonly used ones such as JUnit for the following reason. The main difference between agents and normal objects is autonomy: objects expose methods that external entities may invoke when and how they want. Agents do not expose methods. External entities may interact with them by sending messages that agents will process when and how they (the agents) “want”. Existing testing frameworks such as JUnit are designed to test objects and stimulate them by calling their methods. Since agents do not expose methods we believe that existing testing frameworks are not suited to test agent systems.

The JADE Test Suite is also used by the JADE team to test JADE itself. Developers discovering bugs, problems or unexpected behaviors using JADE are encouraged to provide tests that can be executed by the JADE test suite highlighting the problem they found. This ensures that the problem can be immediately addressed (and hopefully fixed) by the JADE team or someone other in the community.

This document is intended for JADE developers and therefore we suppose the reader is already familiar with JADE.

1.2 System requirements

In order to use the JADE Test Suite you need JADE 3.1 or later, and Sun Java SDK 1.4 or later.

1.3 How to install

The distribution file of the TestSuite must be *unzipped* into the directory where jade is installed. It does not interfere either overwrite any existing file. It just adds this tutorial into the `jade/doc/tutorials` directory and all the sources of the testsuite into the `jade/add-ons/testSuite` directory. Throughout this tutorial we will refer to the `jade/add-ons/testSuite` directory as the “test suite home directory” and we will indicate it as `<test-suite-home>`.

1.4 How to Compile and quick run

To compile the TestSuite you need *ant 1.6.5* or later. Besides the `src` directory, containing all test suite sources, the test suite home directory also contains an ANT build.xml file. Moving to the test suite home directory and typing *ant* compiles the test suite and starts it. Of course be sure the JADE jar files are present in the `jade/lib` directory before launching the compilation and startup process.

The Test suite sources includes both the framework classes and some sample tests that can immediately be executed.

As a result of the compilation a jar file called `testSuite.jar` and located in the `<test-suite-home>/lib` directory is produced. This can be linked to JADE based projects to create automatic tests as will be described in the following sections.

2 THE TESTSUITE FRAMEWORK

2.1 Objective

The JADE Test Suite is a framework build on top of JADE that permits to create tests that can be executed in a **uniform** and **automatic** way that is:

- Uniform: all tests follow the same pattern of execution and produce results in a similar format.
- Automatic: all tests do not require user intervention neither during execution nor to detect whether the test passed or failed.

2.2 The Test Suite Model

The TestSuite framework is based on a two-level model. At the first level we identify macro functionalities like the inter-platform communication functionality. In order to check whether a given functionality works properly a number of different cases must be tested. This leads us to the second level where we identify specific aspects of a functionality. For example, testing the sending and receiving of messages across different platform is a specific aspect of the inter-platform communication functionality. In the followings we will call “*functionality-test*” the set of tests related to all the aspects of a given functionality. Tests on specific aspects, on the other hand, will be referred to as “*atomic-test*”.

2.3 Main classes

In order to reflect the two-level model presented in 2.2, the Test Suite framework provides the classes

1. `test.common.TestGroup` representing the group of all the tests related to a given functionality (i.e. a functionality-test).
2. `test.common.Test` representing the test of a specific aspect of a given functionality (i.e. an atomic-test).

A TestGroup object is basically a collection of Test objects. The list of atomic-tests to be included in a TestGroup is described into an XML file according to the syntax presented in 2.6. The name of that file is passed to the TestGroup in its constructor.

The TestGroup class provides the `initialise()` and `shutdown()` methods that are called by the Test Suite framework just before and just after the execution of the group of tests. By re-defining these methods the user can perform initialisation/clean-up operations common to all the atomic-tests in the group.

The Test class has the following methods:

- 1) `load()`. This method is intended to perform test-specific initializations and returns a Behaviour that is responsible for actually executing the test.
- 2) `clean()`. This method is intended to perform test-specific clean-up operations.

In order to create a new test the user has to extend `test.common.Test` and redefine these two methods as follows:

```
public class MyTest extends Test {
    public Behaviour load(Agent a) throws TestException {
        // Perform test specific initialisation
        Behaviour b = // Create the Behaviour actually performing the test
        return b;
    }

    public void clean(Agent a) {
        // Perform test specific clean-up
    }
}
```

Where `a` is the agent that is going to execute the Behaviour returned by the `load()` method.

3) `passed()`. This method must be called (typically from within the Behaviour performing the test) to indicate that the test has been completed successfully.

4) `failed()`. This method must be called (typically from within the Behaviour performing the test) to indicate that the test has failed.

Calling `passed()` or `failed()` automatically makes the test terminate as soon as the `action()` method of the Behaviour performing the test returns (no matter of the value returned by the Behaviour `done()` method).

5) `log()`. This method can be used to log some information. The current implementation just call `system.out.println()`, but we expect to produce some more sophisticated report in future versions.

6) `setTimeout()`. This method can be used to specify a maximum amount of time for the test. If the test does not complete within the specified timeout it automatically terminates as if the `failed()` method was called.

In the Test Suite framework model a `TestGroup` object is executed by a “*tester agent*” i.e. an agent of a class that extends `test.common.TesterAgent`. Each tester agent has a behaviour (instance of the `jade.common.TestGroupExecutor` class) that

- Calls the `initialise()` method of the `TestGroup`
- For each test in the group {
 - Gets the next `Test` from the `TestGroup`
 - Calls the `load()` method of the `Test` and adds the returned `Behaviour` to the agent scheduler
 - Waits for that `Behaviour` to complete and logs its result
 - Calls the `clean()` method of the `Test`
- }
- Calls the `shutdown()` method of the `TestGroup`

A tester agent gets the `TestGroup` to execute by calling its `getTestGroup()` method. The user has to redefine this method so that it returns the right `TestGroup` object.

The Test Suite framework is completed by the `test.common.TestUtility` providing methods to easily create and kill agents and launch new JADE containers/platforms.

The classes described in this paragraph are summarized in the class diagram presented in Figure 1.

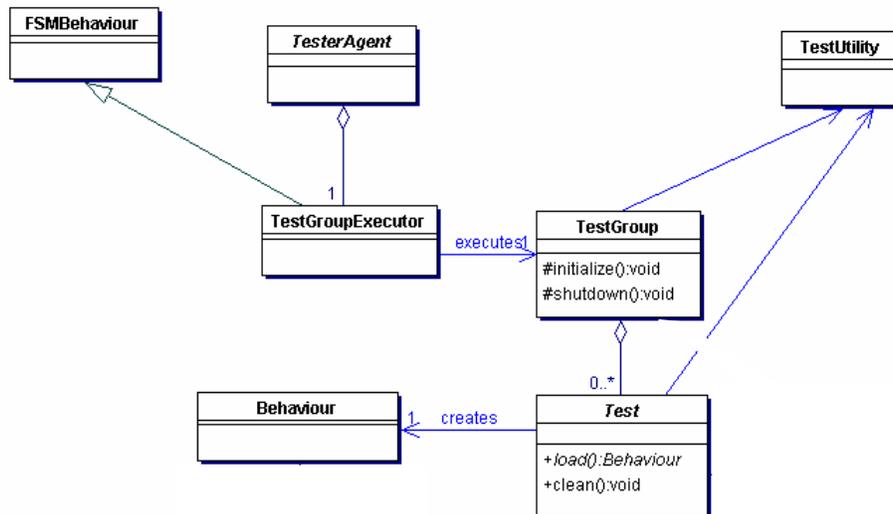


Figure 1. Test Suite framework main classes

2.4 TestGroup arguments

FIXME: To be done.

2.5 The TestSuiteAgent

Functionality-tests can be executed by simply launching the associated tester agent (e.g. through the JADE GUI).

A more convenient way to perform them however is by means of the `TestSuiteAgent`. This is an agent (`test.common.testSuite.TestSuiteAgent`) that provides a GUI by means of which it is possible

to

- View information related to functionality-tests and all atomic-tests they include.
- Select and execute functionality-tests.
- Execute all functionality-tests in sequence and print a final report indicating, for each functionality-test, the number of atomic-tests passed and failed.

As for the tests in a test group, the list of functionality-tests available for execution within the Test Suite framework is described into an XML file whose syntax is presented in 2.6.

2.6 XML Files

As already mentioned the list of functionalities that can be tested by the JADE Test Suite and the list of atomic-tests to be performed for each of them are described by means of XML files. More in details there is a single XML file (`test/testerList.xml`) that contains the list of functionalities under test and one XML file for each functionality that contains the list of atomic-tests to be executed.

The functionality list file structure is described through the following excerpt:

```
<?xml version='1.0' encoding='utf-8'?>
...
...
<TesterList>
  <Tester name="Content" skip="false">
    <ClassName>test.content.ContentTesterAgent</ClassName>
    <Description> Test the ACL content message </Description>
    <TestsListRif>test\content\contentTestsList.xml</TestsListRif>
  </Tester>
  ...
  ...
  ...
</TesterList>
```

Every `<Tester>` element describes a functionality-test and has two attributes: *name* and *skip*, the first one indicates the functionality name and the second one indicates (true or false) if this tester will be skipped during the “*RUNALL* execution” (see 4.1). Each `<Tester>` element on its turn includes the following elements

- `<ClassName>`: specifies the tester class name;
- `<Description>`: gives a brief description about the functionality to test;
- `<TestsListRif>`: specifies the path where to find the XML file including the list of atomic-tests for that particular functionality.

The tests list file structure is described through the following excerpt:

```
<?xml version="1.0" encoding="utf-8" ?>
...
...
<TestsList>
  <Test name="Integer" skip="false">
    <TestClassName>test.content.tests.TestInt</TestClassName>
    <WhatTest> Tests a ACLMessage content.</WhatTest>
    <HowWorkTest>
      It includes a concept with an attribute of type int.
    </HowWorkTest>
    <WhenTestPass> Test passed if no exception is fired.</WhenTestPass>
  </Test>
  <Test name="Boolean" skip="false">
    <TestClassName>test.content.tests.TestBoolean</TestClassName>
    <WhatTest> Tests a ACLMessage content.</WhatTest>
    <HowWorkTest>
      It includes a concept with an attribute of type boolean.
    </HowWorkTest>
    <WhenTestPass> Test passed if no exception is fired.</WhenTestPass>
  </Test>
```

```

...
...
...
</TestList>

```

Every <Test> element describes an atomic-test and has two attributes: *name* and *skip*, the first one indicates the test name and the second one indicates (*true* or *false*) if this test will be skipped during the execution (see 4.1). Each <Test> element on its turn includes the following elements

- <TestClassName>: specifies the test class name;
- <WhatTest>: gives a description of the specific aspect that is going to be tested;
- <HowWorkTest>: gives details about the test main steps;
- <WhenTestPass>: describes the criterion used by the test to detect success or failure.
- <Argument> (0 or more): defines an argument for the test that can be retrieved by means of the `getTestArgument()` method of the `Test` class. The <Argument> element has two attributes:
 - *key*: the key to be passed to the `getTestArgument()` method
 - *value*: the value that will be returned by the `getTestArgument()` method

3 CREATING A TEST

This section describes how to develop a new functionality-test and to add atomic-tests to it.

3.1 Test Suite directory/package structure

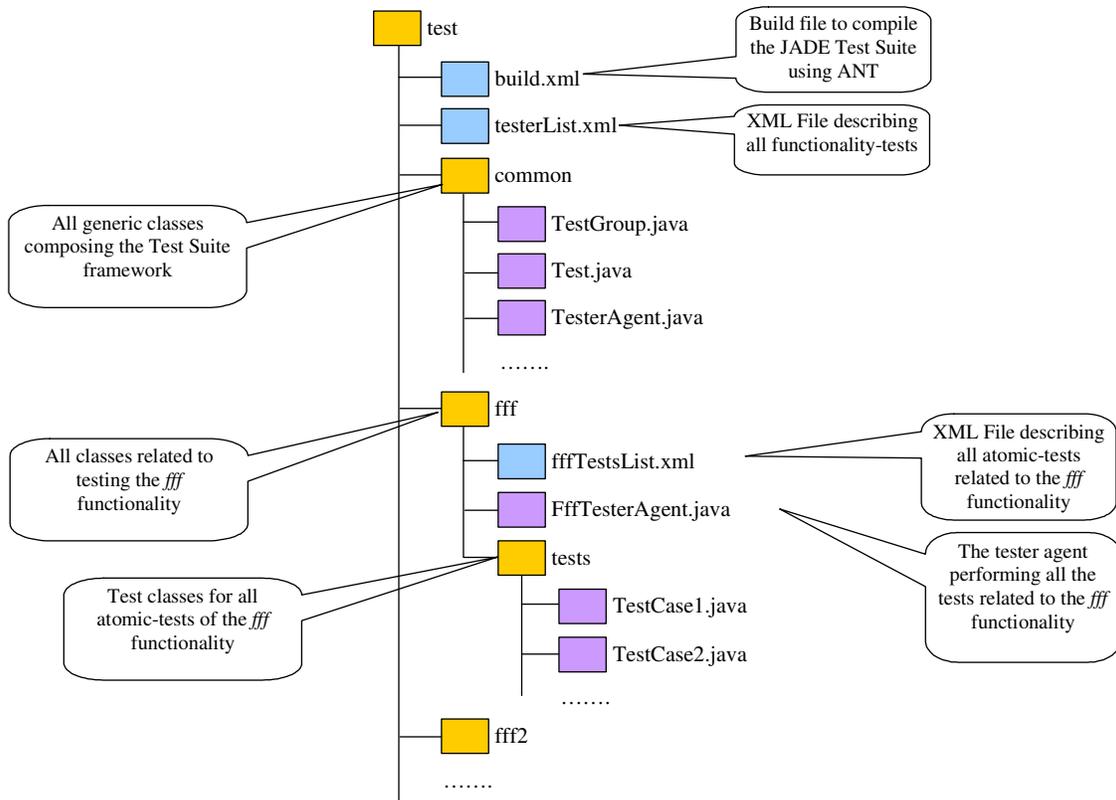


Figure 2 Directory Structure

Figure 2 depicts the directory/package structure we recommend to set up to organize your tests. For each

functionality `fff` under `test` add the package `test/fff` including the `TesterAgent` for the `fff` functionality (`FffTesterAgent.java`) and the xml file describing the atomic tests for the `fff` functionality (`fffTestsList.xml`). Moreover add the package `test/fff/tests` including all tests related to the `fff` functionality.

This package/directory structure perfectly reflects the model presented in 2.2. Moreover adopting it ensures that all xml files are properly handled by the `build.xml` file required to compile the test suite using ANT. Both the `build.xml` file and the `testerList.xml` file describing the functionalities under `test` are located in the `test` directory.

The files composing the Test Suite framework are all included into the `test/common` package.

3.2 Developing a Functionality-Test

As mentioned in 2.3, developing a functionality-test for a given functionality means developing a new tester agent executing a `TestGroup` including tests addressing specific aspects of that functionality. The steps necessary to develop a tester agent for a generic `fff` functionality are listed below.

- 1) Create a new package `test.fff`.
- 2) Create the file `test/fff/fffTestsList.xml` describing the list of atomic-tests to perform according to the syntax presented in 2.6.
- 3) Develop the class `test.fff.FffTesterAgent` extending `test.common.TesterAgent` and re-defining the `getTestGroup()` method as follows:

```
public TestGroup getTestGroup() {
    TestGroup tg = new TestGroup("test//fff//fffTestsList.xml") {

        // Re-define the initialize() method to perform initializations common to all
        // tests in the group
        public void initialize(Agent a) throws TestException {
            // Perform initializations common to all tests in the group
        }

        // Re-define the shutdown() method to perform clean-up operations common to all
        // tests in the group
        public void shutdown(Agent a) {
            // Perform clean-up operations common to all tests in the group
        }
    }

    // Specify group arguments (if any) to be inserted by the user

    return tg;
}
```

- 4) Add a `<Tester>` element to the `test/testerList.xml` file to make the new functionality-test executable from the `TestSuiteAgent` GUI. The `<ClassName>` element must be `test.fff.FffTesterAgent`.

3.3 Adding an atomic-test for a specific aspect.

The steps necessary to develop an atomic-test for a specific `xxx` aspect of a generic `fff` functionality are listed below.

- 1) Create the package `test.fff.tests` (unless already present).
- 2) Develop the class `test.fff.tests.TestXxx` extending `test.common.Test` and re-defining the `load()` and `clean()` methods as described in 2.3.
- 3) Add a `<Test>` element to the `test/fff/fffTestsList.xml` file. The `<TestClassName>` element must be `test.fff.tests.TestXxx`.

3.4 An example

Suppose you want to test the functionality of sending and receiving messages across different platforms. The steps are:

1. Develop a new tester agent:
 - a. Create the `test.interPlatform` package
 - b. Create the file `interPlatformTestsList.xml` and put it into the `test/interPlatform` directory
 - c. Extends `TesterAgent` to create the `test.interPlatform.InterPlatformCommunicationTesterAgent` class
 - d. Override the `getTestGroup()` method to return a `TestGroup` object that gets the tests list from the file `test/interPlatform/InterPlatformTestsList.xml`
 - e. Override the `TestGroup` methods that you need (`initialize(), ...`);
 - f. Update the `testerList.xml` file with a new `<Tester>` element to make the newly developed `InterPlatformCommunicationTesterAgent` executable from the `TestSuiteAgent` GUI.
2. Develop an atomic test for the functionality of sending and receiving messages across different platforms:
 - a. Create the `test.interPlatform.tests` package
 - b. Extends `Test` to create the `test.interPlatform.tests.TestRemotePing` class
 - c. Override the `load()` method to return a `Behaviour` which actually executes the test;
 - d. Override the `clean()` method if you need to clean some structure.

The `test/interPlatform/interPlatformTestsList.xml` file will look like the following:

```
<?xml version="1.0" encoding="utf-8" ?>
<TestsList>
  <Test name="Basic inter platform communication" skip="false">
    <TestClassName>test.interPlatform.tests.TestRemotePing</TestClassName>
    <WhatTest> Tests sending and receiving a message to/from an agent on a remote
platform.</WhatTest>
    <HowWorkTest>An agent is created on a remote platform and then a ping message is sent
to it</HowWorkTest>
    <WhenTestPass>
      The test passes if the agent on the remote platform receives the ping message and
replies correctly and no exception is fired.
    </WhenTestPass>
  </Test>
</TestsList>
```

The modified part of the `test/testerList.xml` will look like the following:

```
<Tester name="Inter Platform Communication" skip="false">
  <ClassName>test.interPlatform.InterPlatformCommunicationTesterAgent</ClassName>
  <Description> This item includes tests related to agent communication across different
platforms. </Description>
  <TestsListRif>test/interPlatform/interPlatformTestsList.xml</TestsListRif>
</Tester>
```

4 RUNNING TESTS

4.1 Running Tests from the TestSuiteAgent GUI

Having compiled the TestSuite source code as described in 1.3, the following command can be used to launch the TestSuiteAgent:

```
java -cp <classpath> test.common.testSuite.TestSuiteAgent
```

A quicker way to execute the same command is “ant run” (to be executed in the jade/src/test working directory).

A snapshot of the TestSuiteAgent GUI is shown in Figure 3

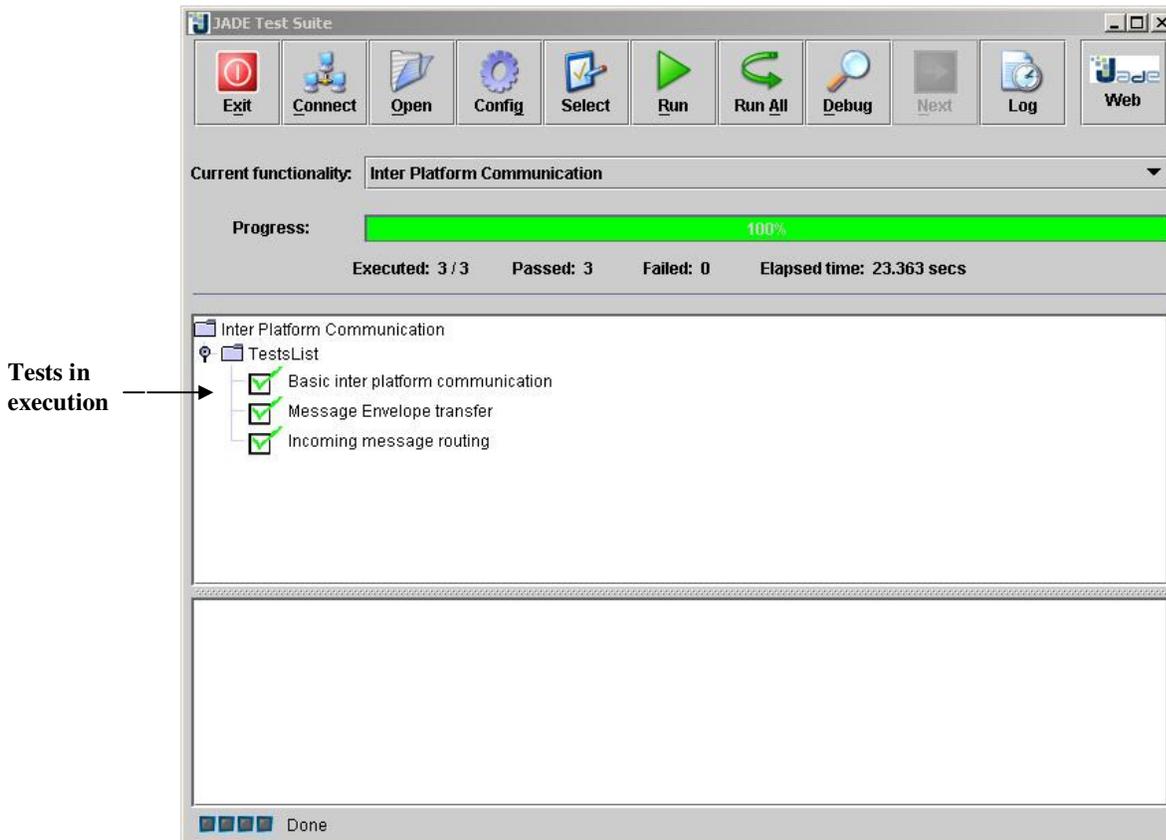


Figure 3 Test Suite command bar

where:

- Button “Exit”. exits the TestSuiteAgent.
- Button “Connect”. The TestSuiteAgent connects to a remote Test Suite Daemon (see 4.2) to launch other JADE instances remotely. This is particularly useful when communication involving different hosts must be tested.
- Button “Open”. The TestSuiteAgent displays the test selection panel where it is possible to view a description of all functionality-tests and atomic-tests available and select a tester agent to load. A snapshot of the test selection panel is shown in Figure 4.

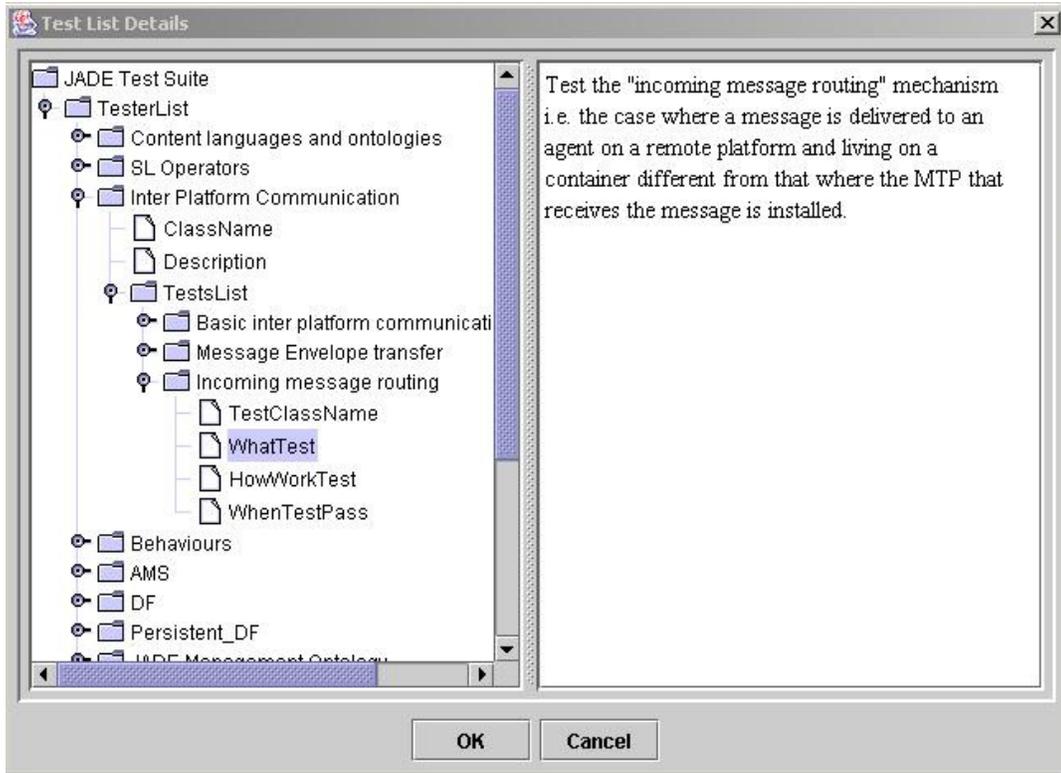


Figure 4: Test List Details

- Button “Config”. Allows to set the arguments, if any, required by the currently loaded tester agent.
- Button “Run”. Execute the currently loaded tester agent.
- Button “Run all”. Execute all functionality-tests that are listed in *testerList.xml*.
- Button “Select”. Select which tests to execute and which ones to skip.
- Button “Debug”. To debug the currently loaded tester agent, i.e. to execute it test-by-test.
- Button “Next”: Execute the next test when you have chosen the debug execution mode.
- Button “Log”: Select where to log tests result. Allowed loggers are console, text file and HTML file.
- Button “Web”: Allows user to browse the Jade Web Site.
- Label “Current Functionality”: Displays the currently loaded tester agent.
- Label “Progress”: Displays the progress status in the execution of the current test.
- Label “Executed”: Displays how many tests among the ones selected has been already run.
- Label “Passed”: Displays how many tests among the ones selected have been passed.
- Label “Failed”: Displays how many tests among the ones selected have failed.
- Label “Elapsed Time”: Displays the time elapsed from the beginning of tests execution.
- Tests in execution: the list of tests in execution is displayed on the graphical interface and a green

check is used to sign them when passed. Red checks are used to sign tests not passed.

4.1 Skipping tests

At run-time you can decide to modify the testers or the tests list to execute; for example sometime you may prefer to run only a subset of all the testers. In order to do that you have two options:

- in the file *testersList.xml* you can set to “*true*” the value of the *skip* attribute for the testers that you don’t want to execute.
- in the file *xxxTestList.xml* you can set to “*true*” the value of the *skip* attribute if you don’t want to run all the tests of a tester.

Yellow checks on the graphical interface are used to sign skipped tests.

4.2 The Test Suite Daemon

FIXME: to be done