ooRexxUnit: A JUnit Compliant Testing Framework for ooRexx Programs

Rony G. Flatscher (Rony.Flatscher@wu-wien.ac.at), Wirtschaftsuniversität Wien

"The 2006 International Rexx Symposium", Austin, Texas, U.S.A.

April 9th – April 13th 2006.

Abstract: *`ooRexxUnit` is an Open Object Rexx implementation of the `JUnit` testing framework (cf. [W3JU3]) It allows for creating and running ooRexx test cases, which assert whether application specifications are met. One usually creates such test cases according to the specification parallel to the development of the application and employs them every time the application is developed further or changed because of maintenance purposes, e.g. because bugs need to get fixed.*

This article will introduce the `ooRexxUnit` framework, give examples of how to employ and implement systematically such unit tests. One aim will be to show and demonstrate, how easy it is with such a testing framework in place to test Rexx and ooRexx programs.

1 Introduction

In software engineering it has become a state-of-the-art standard to define test cases at the same time specifications for algorithms and applications as a whole are developed. This helps to ensure that the implementation of specifications is really carried out correctly. In addition, whenever existing code needs to be changed or enhanced such defined test cases can be used for regression testing which should assert that after alterations of code have been carried out, the specifications are still met.

The more algorithms and the more complex applications as a whole get developed, the more test cases need to be created and maintained. Having dozens – in some cases even hundreds or thousands – of test cases can pose a serious handling problem for application developers or testers. Therefore, the maintaining and running of such test cases should be automatable.

In the Java world there is one testing framework which has become a standard for defining and running such test cases and is called "`JUnit`" (cf. [W3JU3]). In this framework there are interfaces defined which then are used to invoke and control the execution by so called "test runner" programs. Each test case carries out assertions and a test case is said to have run successfully if all of its assertions hold. Test cases are usually organized into collections named "test suites". Support for defining and running test suites is defined as well.

This article introduces the "`ooRexxUnit`" framework which has been modelled closely to `JUnit` version 3.8 in order to allow application developers who are acquainted with JUnit tests to apply their working knowledge right away. On the other hand, people who have never worked with a JUnit-like testing frameworks would gain the ability to research and read the wealth of articles, tutorials, and discussions that have been conducted for many years (cf. [W3JU3Cook], [W3G06]).

The JUnit framework is defined with object-oriented concepts, i.e., using classes.

2 "ooRexxUnit" ("OOREXXUNIT.CLS")

This chapter first introduces the four core classes Assert, TestCase, TestSuite and TestResult, defining their abilities in form of the implemented methods adhering to the JUnit framework. Therefore additional information can be always gathered by researching the JUnit documentation, articles, and tutorials.

Firstly, the four public classes and the public routines stored in the file "OOREXXUNIT.CLS" are introduced, which can be accessed, once a Rexx program has called or required[1] it. At the end of this chapter a few examples, from simple to more comprehensive, demonstrate how to put the ooRexxUnit framework to work.

2.1 The ooRexxUnit Framework Classes

Figure 1 depicts the four classes Assert, TestCase, TestSuite and TestResult and the existing specialization relationships. It follows from that class overview that TestCase is a specialization of Assert, and that TestSuite specializes TestCase directly and Assert indirectly.

From the inheritance feature of object-oriented systems it follows that all methods

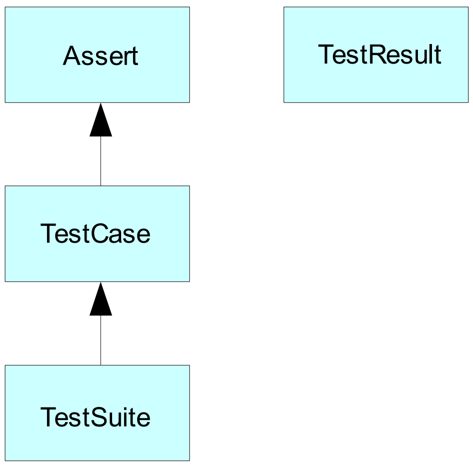

Figure 1: The ooRexxUnit Classes.

[1] The requires directive is carried out by the ooRexx interpreter before the program gets started by carrying out the very first statement (in the first line) of the program, by calling the required Rexx program. Whatever public classes and public routines are encountered will be available upon return in the program. This way the public classes of the "OOREXXUNIT.CLS" program can be made available for subclassing them in the program, in addition to gaining access to its public routines.

of the class Assert are available to instances of the class TestCase, and all methods of the TestCase class are also available to the TestSuite class. Therefore it is possible to invoke the Assert methods by merely sending the appropriate assert messages to TestCase and TestSuite objects.[2]

2.1.1 The “Assert” Class

The Assert class defines all the assertion methods that test cases may need to use for testing assertions.

Figure 2 documents all available methods. The argument failMsg in the assert methods is optional and used only if an assertion fails to record the author's message with the failure.

Method Signature	***Description***
assertCount	Returns the number of successful assertions so far (ooRexxUnit only).

assertEquals(val1, val2)
assertEquals(failMsg, val1, val2)
	Compares val1 and val2 for equality using the = operator semantics. If they are not equal the method fail will be invoked.[3]

assertFalse(val)
assertFalse(failMsg, val)
	Expects the Boolean value .false (the string “0”) as the result of some operation. If val does not have the .false value the method fail will be invoked.

assertNotEquals(val1, val2)
assertNotEquals(failMsg, val1, val2)
	Compares val1 and val2 for not being equal (using the = operator semantics). If they are equal the method fail will be invoked.[4]

assertNotNull(val)
assertNotNull(failMsg, val)
	Expects any value, but the .nil object. If val does refer to the .nil object the method fail will be invoked.

assertNotSame(val1, val2)
assertNotSame(failMsg, val1, val2)
	Compares val1 and val2 for not being identical (using the == operator and comparing its result with .false). If they are identical the method fail will be invoked.

assertNull(val)
assertNull(failMsg, val)
	Expects the .nil object as the result of some operation. If val does not refer to the .nil object the method fail will be invoked.

assertSame(val1, val2)
assertSame(failMsg, val1, val2)
	Compares val1 and val2 for identity using the == operator. If they are not identical the method fail will be invoked.

assertTrue(val)
assertTrue(failMsg, val)
	Expects the Boolean value .true (the string “1”) as the result of some operation. If val does not have the .true value the method fail will be invoked.

expectCondition(conditionName)	Expect a condition with the given conditionName. If the expected condition is not raised in the test case method, the method fail will be invoked.
expectSyntax(errorCode)	Expect a syntax condition with the given errorCode. If the expected syntax condition is not raised in the test case method, the method fail will be invoked.

[2] In ooRexx *only objects* can conceptually invoke methods. Programmers can invoke methods of an object only indirectly, by sending the object a message with the name of the method the object should invoke on behalf of the programmer.

[3] The compared values can be ordered or unordered collections as well. Ordered collections are regarded to be equal, if their MAKEARRAY method yields the same objects in the same order. Unordered collections are regarded to be equal, if their renderings to a relation object yields two collections that are subsets of each other.

[4] This method is ooRexxUnit specific and is able to test collections as described in footnote 3 above.

Method Signature	***Description***

`fail()`
`fail(failMsg)`
	Invoking this method causes the test case to fail, i.e., no more statements from the test case method will be executed. If the optional error message `failMsg` is supplied this message will be recorded with this failure.

Figure 2: Methods of the `Assert` Class .[5]

2.1.2 The "TestCase" Class

Test cases are created by defining a class ("test class") that is a subclass of `TestCase`, having specific test methods[6] implemented that each represent an individual test case.

The class `TestCase` defines what a test case is able to do:

- optional methods to set up (`setUp`) and tear down (`tearDown`) a testing environment, e.g., creating the testfiles a test case may need to use and removing them upon termination of the test case,
- a method that starts the test case (`run`),
- methods to get (`getName`) and to set the name (`setName`) of the test case,
- a method (`createResult`) that creates a test result object (an instance of the class `TestResult`) which is used to protocol (store) the results of running the test case. `createResult` is used only, if the `run` method does not receive a test result object as an argument.

As the class `TestCase` is subclassing the class `Assertion`, all of the methods of the Assertion class are available due to inheritance.

Figure 3 lists all methods defined for the `TestCase` class with a brief description of their purpose.

Method Signature	***Description***
`testCaseInfo`	*Class* attribute: a directory object which allows storing information about all test cases (`ooRexxUnit` only).
`defaultTestResultClass`	*Class* attribute: allows storing the class object the instance method `createResult` uses (`ooRexxUnit` only) for creating a `TestResult` object. Defaults to the `TestResult` class object.
`testCaseInfo`	A convenience instance attribute (a directory object) which allows storing information about a particular test case (`ooRexxUnit` only).
`countTestCases`	Returns the number of test cases (used e.g. in the `TestSuite` subclass).
`createResult`	Returns a `TestResult` object (an instance of the class object stored in the class attribute `defaultTestResultClass`).
`countTestCases`	Returns the number of available test cases (could be more than 1 in the case of a test suite).

[5] The methods `expectCondition` and `expectSyntax` allow to test for expected ooRexx conditions to be raised while a test method runs and were created and supplied by Rick McGuire at the 2006 International Rexx Symposium.

[6] Each test method of the "test class" will test and assert a specific aspect of an application or routine.

Method Signature	***Description***
`getName`	Returns the name of the test case method that gets run.
`init(nameOfTestCaseMethod)`	`nameOfTestCaseMethod` determines the name of the test method that should run for this test case.

`run`
`run(aTestResult)`
	Runs the test case and returns the `TestResult` object that was used to log the results of running the test case. If no `TestResult` object is supplied as an argument a new one is created from the class stored in the class attribute `defaultTestResultClass` (defaults to the `TestResult` class).

`setName(nameOfTestCaseMethod)`	Allows to set the test case object to name a different test method to run.
`setUp`	NOP[7] method: gets invoked, but does nothing. If a test case needs to set up a specific environment before its tests are run, then the test class needs to override it by implementing itself a method named `setUp`.
`string`	Creates a human-readable representation of the test case object, indicating the name of the test case and the name of the test class (`ooRexxUnit` only).
`tearDown`	NOP method: gets invoked, but does nothing. If a test case needs to tear down a specifically set up environment or clean up after a test case ran, then the test class needs to override it by implementing itself a method named `tearDown`.

Figure 3: Methods of the `TestCase` Class.

2.1.3 The “TestSuite” Class

The `TestSuite` class allows to define a set, a “suite” of test cases all of which should be run. As this class specializes `TestCase` all of its methods are inherited. For that reason it is possible to override e.g. `startUp` and `tearDown` in the case that a suite of test cases need the same environment set up before and torn down after running the test cases.

Figure 4 lists all methods of the `TestSuite` class with a brief description of their purpose.

Method Signature	***Description***
`getTestMethods(testClass)`	Class method that returns a “stem array”[8] which contains all test methods of the `testClass` in ascending order, i.e., all methods which names start with the string “`TEST`”.
`addTest(aTestCase)`	Adds the given test case object to the test suite.
`countTestCases`	Returns the number of test cases in the test suite.
`init([aTestClass])`	If the optional argument `aTestClass` is given, then all methods of that test class that start with the string “`test`” are regarded to be test case methods. Hence, for each such method a test case instance gets created from the class and added to the test suite.

`run`
`run(aTestResult)`
	Runs all the test cases in the test suite and returns the `TestResult` object that was used to log the results of running the test cases. If no `TestResult` object is supplied a new one is created from the class stored in the class attribute `DefaultTestResultClass` (defaults to the TestResult class).

Figure 4: Methods of the `TestSuite` Class.

[7] “NOP” is the acronym for “Null operation”.

[8] A “stem array” is a stem that uses integer number as indices. The stem index “`0`” returns the number “`n`” of elements stored with the stem, starting with the index “`1`” through “`n`”.

2.1.4 The “TestResult” Class

The result of running test cases[9] are logged with an instance of the `TestResult` class which serves as a log container of the run test cases. A test case may run successfully (all the assertions hold), it may fail (an assertion does not hold) or there may be an unexpected execution error (any condition the ooRexx interpreter raises, e.g. if there is a syntax error, etc.).

A test result object is used in the `run` method of the `TestCase` and the `TestSuite` class for logging the assertions, failures and errors, as well as learning whether carrying out test cases should be stopped prematurely. A test runner application uses a test result object to control the execution of running test cases, which it can stop prematurely by sending the test result object the `stop` message. In addition a test runner application can use the logged results to analyze and report about the test run(s).

Figure 5 lists all methods of the `TestResult` class with a brief description of their purpose.

Method Signature	***Description***
`logQueue`	Returns a queue containing directory objects created by the `run` method. For each test case a new directory object is queued for running the methods `startTest` and `endTest`, as well as in the case of a failure and error, which will cause the running of its `addFailure` and `addErorr`. The directory object will store the date, the time, the name of the test case method, and the name of the file in which the test case got defined. The index name for retrieving this information encoded as a string from the directory object is “`OOREXXUNIT.CONDITION`“. Figure 6 defines the encoding of that string. In the case of a failure or error the directory object in addition contains all ooRexx information supplied with its condition object as received by using the ooRexx built-in function (BIF) `condition(“Object”)`.
`testCaseTable`	Returns a table, whose indices are the individual test case objects and whose associated item is a queue object. For each invocation of the methods `startTest`, `endTest`, `addFailure` and `addError` an encoded string is queued.
`addError(aTestCase, aDir)`	Queues the directory (condition) object to the `logQueue`, the `errors` queue and adds an appropriate encoded string to the `testCaseTable` queue.
`addFailure(aTestCase, aDir)`	Queues the directory (condition) object to the `logQueue`, the `failures` queue and adds an appropriate encoded string to the `testCaseTable` queue.
`assertCount`	Returns the number of successful assertions.
`endTest(aTestCase)`	Encodes the date and time into a string and queues it as part of a directory object to the `logQueue` and as a string to the appropriate `testCaseTable` queue.
`errorCount`	Returns the number of errors.
`errors`	Returns the queue that contains the error related directory objects.
`failures`	Returns the queue that contains the failure related directory objects.
`failureCount`	Returns the number of failures.
`run(aTestCase)`	Runs the given test case and returns the itself (a test result object).
`runCount`	Returns the number of run test cases.
`shouldStop`	Returns a Boolean value indicating whether running test cases should stop.

[9] `TestResult` objects are used to log the running of a test case in the `run` method of the T`estCase` and the `TestSuite` class.

Method Signature	Description
	.true should stop any outstanding test cases from running.
stop	Sets shouldStop to return .true, i.e., stop running any outstanding test cases.
wasSuccessful	Returns .true, if no failures or errors were encountered while running all of the test cases, .false otherwise.

Figure 5: Methods of the TestResult *Class.*

Figure 6 depicts the string encodings the ooRexxUnit methods use to log the sorted date, the long time, the test case (method) name, the class (object) in which the test case got implemented, the failure and/or the error message.

Method	Encoding with Sample Data
startTest	[20151105 17:51:52.150000]:˽[startTest]˽testCase:˽[testMethod1]˽(a RgfTest@A440F21F)
addError	[20151105 17:51:52.161000]:˽[error]˽testCase:˽[testMethod1]˽(a RgfTest@A440F21F)˽--->˽errMsg Where "errMsg" is one of the following strings: condition [X] raised unexpectedly.Y condition [SYNTAX a.b] raised unexpectedly.Z Where "X" is replaced by the Rexx symbol(s) denoting the condition's name[10], "Y" represents the TAB ("09"x) character. The second form is used for syntax conditions where the string "a.b" is replaced by the actual syntax error number and "Z" represents the TAB ("09"x) character immediately followed by the full error message.
addFailure	[20151105 17:51:52.162000]:˽[failure]˽testCase:˽[testMethod1]˽(a RgfTest@A440F21F)˽--->˽failMsg Where "failMsg" is one of the following strings: @assertFailure assertEquals: expected=[E], hashValue="he"x], actual=[[A], hashValue="ha"x].Z @assertFailure assertFalse: expected=[0], actual=[A].Z @assertFailure assertNotEquals: expected=[\= [E], hashValue="he"x], actual=[[A], hashValue="ha"x].Z @assertFailure assertNotNull: expected=[\= [.nil]], actual=[.nil].Z @assertFailure assertNotSame: expected=[\== [E], hashValue="he"x], actual=[[A], hashValue="ha"x].Z @assertFailure assertNull: expected=[.nil], actual=[A].Z @assertFailure assertSame: expected=[[E], hashValue="he"x], actual=[[A], hashValue="ha"x].Z @assertFailure assertTrue: expected=[1], actual=[A].Z @assertFailure check4ConditionFailure: expected condition [E] was not raised.Z Where "E" is replaced by the expected value, "he" is the hash value of the expected value, encoded as a hexadecimal string. "A" is replaced by the actual value, "ha" is the hash value of the actual value, encoded as a hexadecimal string. "Z" represents the TAB ("09"x) character, which can be immediately followed by a failure message, which the programmer supplied as the first argument to the appropriate assert method.
endTest	[20151105 17:51:52.170000]:˽[endTest]˽testCase:˽[testMethod1]˽(a RgfTest@A440F21F)

Figure 6: The String Encoding of Logged Test Case Run Information.

[10] In the case of a syntax condition the string "SYNTAX" will indicate the condition, immediatley followed by a blank, immediately followed by the the syntax error number, e.g. "SYNTAX 42.3" (the syntax error message in this example would be: "Arithmetic overflow; division must not be zero.").

2.1.5 Overview of the Classes and their Methods

Figure 7 depicts graphically the structure and methods of the ooRexxUnit classes.

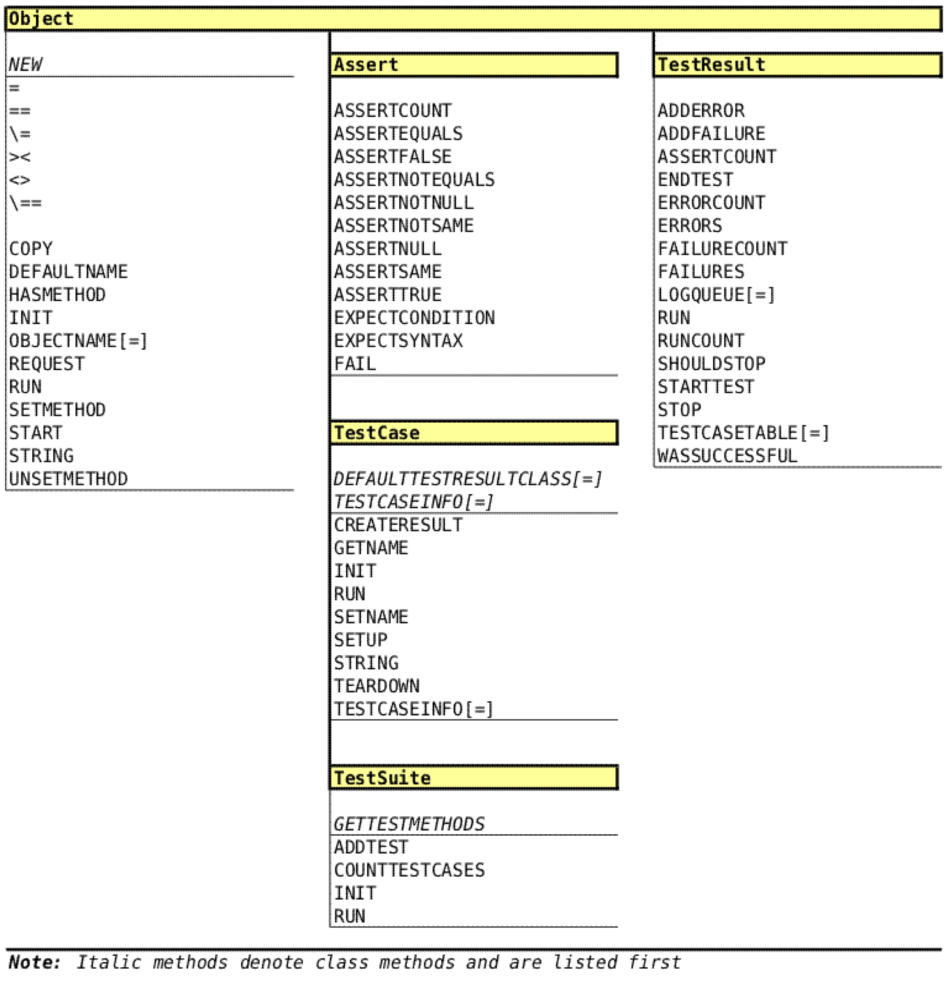

Figure 7: Overview of the ooRexxUnit Classes and their Methods.

2.2 The ooRexxUnit Framework Routines

Figure 8 depicts the available public routines, provided for the convenience of the framework users, the most important being `simpleDumpTestResults(…)` which dumps statistical information from the supplied test result object.

Routine Signature	***Description***
`iif(test, valTrue, valFalse)`	Returns the supplied `valTrue`, if `test` has the value `.true`, the supplied `valFalse` else.
`makeDirTestInfo(aTestCaseClass, arrLines)`	Processes an array of text in the form of "`keyword: value`", stores keyword in the class attribute `testCaseInfo` directory and creates a queue for it which receives the following text(s) until a new `keyword` is encountered.

Routine Signature	***Description***
`makeTestSuiteFromFileList(fileList [,aTestSuite])`	Returns a test suite object built from the files in the string `fileList`. If the second argument is supplied this test suite object gets used to add the test cases from the files and will be returned as the result of the routine.[11]
`pp(val)`	Returns the string value of `val` enclosed in square brackets.
`ppp(val)`	Returns the string value of `val` enclosed in square brackets. Non-printable characters are escaped as a hexadecimal Rexx literal.
`simpleDumpTestResults(aTestResult [,title])`	Analyzes the supplied test result object `aTestResult` and outputs brief statistics to standard output using the optional `title` string as the header ("title"). In the case of an error or failure all the given error or failure messages are output as well.

Figure 8: The Public Routines of the `ooRexxUnit` Framework.

2.3 Putting the ooRexxUnit Framework to Work

This section introduces sample programs that take advantage of the ooRexxUnit framework, starting out with very simple usages, concluding with a fairly comprehensive example.

The results of the test runs are always displayed with the public routine `simpleDumpTestResults(…)` which gives a brief overview of the test result object's information. It is possible that one writes an own "test runner" application which gives much more thorough information about the run test cases, those that ran successfully and those that did not.[12]

2.3.1 An Elementary Test Case "`sample01.rex`"

Figure "Code 1" below depicts a minimal program "`sample01.rex`" that puts the ooRexxUnit framework to work. It defines "`MyTestClass`" (a specialization of the ooRexxUnit framework's class `TestCase`) with the method named "`testMethod`", in which the result of the arithmetic addition "`1+2`" is asserted[13] to be equal to "`3`". Only if the result is not equal to "`3`" would the test fail.

The program starts out by creating an instance (object) of `MyTestClass` and supplying the name of its test method "`testMethod`" as an argument determining that that method's code should be run as the test case. Sending the object the message `run`

[11] If there are mandatory tests defined in the test unit files, then test cases are only created for these. Cf. section entitled "Structure of '`.testUnit`' Programs" below.

[12] As a matter of fact, such a test runner application, if it is able to store the test results in a standardized way could even compare the results of different test runs, indicating improvements and deterioration in the quality of the code that gets tested.

[13] The class `TestClass` specializes the ooRexxUnit framework's class `TestCase`, which itself specializes the framework's class `Assert`. Therefore all methods of the superclasses of `TestClass` are available. Sending the message `assertEquals` to `self`, causes the inheritance tree to be looked up, eventually getting to the `Assert` class where a method by the same name is found.

will cause the method's code to run. The TestCase's run method will first create an instance of the class TestResult for logging the results and then runs the test case method “testMethod”, returning the test result object upon completion.

Finally, using the ooRexxUnit framework's routine simpleDumpTestResults(…) and supplying the returned test result object will create the output that is shown in figure “Output 1”.

```
o=.MyTestClass~new("testMethod") /* create an instance and use the code of
                                    the method "testMethod" as the test case */
aTestResult=o~run /* runs the test case and returns a test result object     */

call simpleDumpTestResults aTestResult /* dump brief results                 */

::requires ooRexxUnit.cls  /* get access to the ooRexxUnit framework         */

::class MyTestClass subclass TestCase    /* a specialization of TestCase     */
::method testMethod           /* this method's code is the test case         */
  a= 1+2
  self~assertEquals(3, a)    /* using the Assert class via inheritance       */
  self~assertEquals("test: 3=(1+2)", 3, 1+2 ) /* supply a failure message    */
```

Code 1: The Rexx Program “sample01.rex”.

```
nr of test runs:             1
nr of successful assertions: 2
nr of failures:              0
nr of errors:                0
```

Output 1: Output of Running “rexx sample01.rex”.

2.3.2 Another Elementary Test Case “sample02.rex”

Usually, there will be a need to define more than one test case, most likely at least one for each feature that needs testing. In such a case multiple methods will be defined, each one representing a specific test case. For each such test method one will need to instantiate the test class and supply as the sole argument the respective test method's name, finally executing that test method's code by sending the returned instance the run message. In order to gather the results from all these test cases one should use the same TestResult object as the argument for all the run messages.

Figure “Code 2” depicts a program that contains two test methods, one named “anton”, and one named “berta”, both belonging to the test class named “MyTestClass”. Figure “Output 2” shows the statistics from the logged content stored in the TestResult object.

```
aTR=.TestResult~new        -- this TestResult object will be used to log all runs
o=.MyTestClass~new("anton")-- create an instance, denote test method to run
aTR=o~run(aTR)             -- run the test code, supply a TestResult object

   -- create an instance (denoting the name of the test method to run) and run it,
   -- supplying the same TestResult object as previously
.MyTestClass~new("berta")~run(aTR)

call simpleDumpTestResults aTR      -- show brief statistics

::requires ooRexxUnit.cls  --  get access to the ooRexxUnit framework

::class MyTestClass subclass TestCase -- a specialization of TestCase

::method anton                      -- this method's code is the test case
  self~assertFalse(.false)          -- assert that ".false" is false
  a=.false                          -- set variable to .false
  self~assertFalse(a)               -- assert that "0" is false
  self~assertFalse(0)               -- assert that "0" is false
  self~assertFalse(2)    -- ASSERTION WILL FAIL!   -- assert that "2" is false
  self~assertFalse('"0" is false in Rexx!', 0)     -- supply optional failure text
  self~assertFalse('is "2" false in Rexx?', 2)     -- assert that "2" is false

::method berta                      -- this method's code is the test case
  self~assertTrue(.true)            -- assert that ".true" is true
  a=.true                           -- set variable to .true
  self~assertTrue(a)                -- assert that ".true" is true
  self~assertTrue(1)                -- assert that "1" is true
  self~assertTrue('"1" is true in Rexx!', 1)    -- supply optional failure text
  self~assertTrue('is "2" true in Rexx?', 2)    -- WILL FAIL! -- assert that "2" is true
```

Code 2: The Rexx Program "`sample02.rex`".

```
nr of test runs:             2
nr of successful assertions: 7
nr of failures:              2
   [20150530 18:17:12.958000]: [failure] testCase: [anton] (a TESTCLASS@3CC5F21F) --->
@assertFailure assertFalse: expected=[0], actual=[2].○
   [20150530 18:17:12.958000]: [failure] testCase: [berta] (a TESTCLASS@6AD2F21F) --->
@assertFailure assertTrue: expected=[1], actual=[2].○is "2" true in Rexx?
nr of errors:                0
```

Output 2: Output of Running "`rexx sample02.rex`".[14]

A few remarks:

- The test method "`anton`" carries out six assertions, the test method "`berta`" five, giving a total of eleven assertions.
- There were two test cases run in which a total of seven assertions did hold, but two did fail (and two were not tested in the method "`anton`").
- Once an assertion fails, the test case ends prematurely, such that the remaining assertions are not tested. Therefore the last two assertions in the test method "`anton`" were not tested in the above run and hence are not reflected in the above brief statistics.
- If an assertion fails and no failure text was supplied, then the date and time, the name of the test method and the ooRexx default name for the test case object is given. If a failure text is given as the first argument to any of the assertion messages, it will be shown in the case of a failure as a trailing text with the failure information. In figure "Output 2" the failure message stored with the failure information is depicted in italics.

[14] Lines depicted in italics were broken up by the word processor due to their length. They should go with the previous line. The character "○" represents the TAB ("09"x) character.

2.3.3 TestCases for Conditions

In this section two nutshell examples are introduced that demonstrate how to define test cases that assert that conditions arise while executing the code.

There are two groups of conditions, the syntax conditions defining an error number that identifies the syntax error, and all other conditions expressed by a Rexx symbol (in the case of user defined exceptions, two symbols are used, the first being the Rexx symbol `USER`).

If an expected condition has not been raised during the execution of the test case, then a failure is raised to document this fact.

2.3.3.1 Testing for Syntax Error Condition "`sample03.rex`"

An uncaught syntax error will always stop the execution of a test case. Therefore the code to lead to such an expected syntax error condition should be either the only code in the test case method or given at the end of it.

The `run` method will then check whether that particular syntax error was expected. If it was expected, then the number of successful assertions is increased by one, otherwise a failure is logged with the test result object.

Figure "Code 3" depicts a program that contains a test class named "`MyTestClass`" consisting of a test method named "`alpha`" with an expected syntax error, a test method "`bravo`" that does not raise an expected syntax error and a method "`caesar`" that just causes a non expected syntax error to be raised. Figure "Output 3" shows the statistics from the logged content stored in the `TestResult` object.

```
/* the following TestResult object will be used to log all test case runs */
aTR=.TestResult~new

   /* create three test case objects from the same test class but using
      different methods as test cases */
do methName over .list~of("Alpha", "BravO", "caesar")
   .MyTestClass~new(methName)~run(aTR) /* create and run the test cases    */
end

call simpleDumpTestResults aTR   /* dump brief results                  */

::requires ooRexxUnit.cls        /* get access to ooRexxUnit            */

::class  MyTestClass subclass TestCase
::method alpha                   /* the expected syntax error gets raised  */
  errorCode=42.3  -- syntax error "Arithmetic overflow; divisor must not be zero"
  self~expectSyntax(errorCode)    -- expect syntax error # 42.3
  a=1/0                           -- create divide by 0 syntax error

::method bravo                   /* an expected syntax error is not raised */
  errorCode=42.3  -- syntax error "Arithmetic overflow; divisor must not be zero"
  self~expectSyntax(errorCode)    -- expect syntax error # 42.3
```

```
::method caesar                  /* an unexpected syntax error gets raised */
  a=1/0                          -- create divide by 0 syntax error
```

Code 3: The Rexx Program "`sample03.rex`"

```
nr of test runs:              3
nr of successful assertions: 1
nr of failures:               1
   [20150530 18:25:50.292000]: [failure] testCase: [BravO] (a MYTESTCLASS@30CDF21F) --->
@assertFailure check4ConditionFailure: expected condition [SYNTAX 42.3] was not raised.○
nr of errors:                 1
   [20150530 18:25:50.292000]: [error] testCase: [caesar] (a MYTESTCLASS@A6D9F21F) --->
condition [SYNTAX 42.3] raised unexpectedly.○Arithmetic overflow; divisor must not be zero
```

Output 3: Output of Running "`rexx sample03.rex`".[15]

2.3.3.2 Testing for Other ooRexx Conditions "`sample04.rex`"

Testing for ooRexx conditions should be carried out condition by condition, separately coded in its own test case method, or at the very end of it.

The test case's `run` method will then check whether a particular condition was to be expected. If that particular expected condition was raised, then the number of successful assertions is increased by one, otherwise a failure is indicated that gets logged with the test result object.

Figure "Code 4" depicts a program that contains a test class named "`MyTestClass`" consisting of a test method "`testCase01`" with an expected user defined condition, a test method "`testCase02`" that does not raise an expected novalue condition and a method "`testCase03`" that just causes a non expected lostdigits condition to be raised. Figure "Output 4" shows the statistics from the logged content stored in the `TestResult` object.

```
/* the following TestResult object will be used to log all test case runs */
aTR=.TestResult~new

   /* create three test case objects from the same test class but using
      different methods as test cases */
do methName over .list~of("TESTCASE01", "testCase02", "testcase03")
   .MyTestClass~new(methName)~run(aTR) /* create and run the test cases    */
end

call simpleDumpTestResults aTR         /* dump brief results               */

::requires ooRexxUnit.cls              /* get access to ooRexxUnit         */

::class "MyTestClass" subclass TestCase
::method testCase01                    -- raise the expected condition
  self~expectCondition("USER RGF")     -- expect a USER condition
  RAISE USER RGF                       -- raise user defined condition

::method testCase02                    -- do not raise the expected condition
  self~expectCondition("NOVALUE")      -- expect a NOVALUE condition
```

[15] Lines depicted in italics were broken up by the word processor due to their length. They should go with the previous line. The character "○" represents the TAB ("09"x) character.

```
::method testCase03                    -- raise an unexpected condition
  signal on novalue
  a=b+1
  return

novalue:
  raise propagate
```

Code 4: The Rexx Program “`sample04.rex`”.

```
nr of test runs:             3
nr of successful assertions: 1
nr of failures:              1
   [20061230 18:33:21.451000]: [failure] testCase: [testCase02] (a MyTestClass@B2CCF21F) --->
@assertFailure check4ConditionFailure: expected condition [NOVALUE] was not raised.○
nr of errors:                1
   [20061230 18:33:21.451000]: [error] testCase: [testcase03] (a MyTestClass@24D9F21F) --->
condition [NOVALUE] raised unexpectedly.○
```

Output 4: Output of Running “`rexx sample04.rex`”.[16]

2.3.4 Additional ooRexxUnit Framework Conventions

It is conceivable that over time test classes gain a lot of test methods, such that it becomes desirable to automate the creation and running of test cases. In JUnit 3.8 [W3JU3] there is a convention by which all methods of a test class serve as test cases, whose names start with the string “test”. This is achieved by using Java's reflection mechanism which allows to determine at runtime all the methods that a class defines.

ooRexx, being an interpreted object-oriented language, possesses all the necessary means to use reflection at runtime to determine the methods that are defined for a class. This is realized by sending the class object the message “`METHODS`” which returns a supplier object that one can use to iterate over the defined (instance) methods. The `TestSuite`'s class method `getTestMethods` returns an ascendingly sorted array object containing the names of all test methods, i.e., methods whose names start with the string “`test`”. If one creates an instance of the `TestSuite` class by supplying the class object of a test class, then the `TestSuite`'s constructor will use the aforementioned class method `getTestMethods` to create test case objects for each test method and will add them to the test suite.

In order to help automate the creation and running of test case objects in a systematical way one can lay out conventions. This section will introduce such conventions and a short (“test runner”) program which takes advantage of the conventions to become able to run all adhering test programs.

2.3.4.1 Structure of “*`.testUnit`*” Programs

A “test unit” program contains one or more test classes, each of which may have

[16] Lines depicted in italics were broken up by the word processor due to their length. They should go with the previous line. The character “○” represents the TAB (`"09"x`) character.

test methods defined that represent individual test cases. Figure “Code 5” depicts the overall structure of such a test unit program, that gets stored with a file extension of “`.testUnit`” and contains annotations at the right margin (led in by “`/* ANN`”).

```
/* list of array objects (pairs of test class object and its test methods
*/
mandatoryTestMethods=.list~new   /* list of mandatory tests                  */      /* ANN 1 */
testUnitList=.list~of( .array~of(.testClass_01,  mandatoryTestMethods) )             /* ANN 2 */

-------------------------------------------------------------------------------
arrLines=.array~new              /* read top comment of this file            */      /* ANN 3 */
do i=1 to 150 until arrLines[i]="*/"
   arrLines[i]=sourceline(i)
end

aTestUnitClass=testUnitList~at(testUnitList~first)[1] /* get first testClass  */     /* ANN 4 */

   /* will parse the array lines and store result in class object's directory */    /* ANN 5 */
call makeDirTestInfo aTestUnitClass, arrLines
tmpDir=aTestUnitClass~TestCaseInfo  /* get directory containing parsed infos  */
parse source s       /*  op_sys invocationType fullPathToThisFile             */
tmpDir~setentry("test_Case-source", s)

do arr over testUnitList   /* add this directory to the other testCase classes*/     /* ANN 6 */
   if arr[1]=aTestUnitClass then iterate  /* already handled                  */
   arr[1]~TestCaseInfo=tmpDir             /* save info with class object      */
end

if .local~hasentry("bRunTestsLocally")=.false then /* entry not in .local ?   */     /* ANN 7 */
   .local~bRunTestsLocally=.true /* define entry, let all tests run           */

if .bRunTestsLocally=.true then  /* run ALL tests in this test unit          */      /* ANN 8 */
do
   ts=.testSuite~new             /* create a testSuite                       */
   do arr over testUnitList      /* iterate over testUnits                   */
      ts~addTest( .testSuite~new(arr[1])) /* create testSuite                 */
   end
   testResult=ts~run             /* now run all the tests                    */

   call simpleDumpTestResults testResult  /* show brief results              */
end

return testUnitList  /* return list of array objects containing the testUnits */     /* ANN 9 */


::requires ooRexxUnit.cls        /* load the ooRexxUnit classes              */     /* ANN 10 */

::class "testClass_01"  subclass TestCase public   /* test class             */     /* ANN 11 */

::method "test_01"               /* first test method                        */     /* ANN 12 */
   self~assertEquals("subTest_01", "1", 0+2-1)

::method "hi"                    /* some other test method                   */     /* ANN 13 */
   self~assertEquals("subTest_01", "1", 0+1)
```

Code 5: The Rexx Program “`generic.testUnit`”.

The following explanations refer to the annotations embedded at the right margin in “Code 5” above:

- “`/* ANN 1 */`”: This list may contain the names of those test methods which are regarded to be mandatory. If the list is left empty, all test methods (those that start with the string “`test`”) of a test class are regarded to be mandatory.[17]

- “`/* ANN 2 */`”: This list contains array objects, where each array stores the test class object at index 1, and its list of mandatory test methods at index 2.
- “`/* ANN 3 */`”: This code block reads the top comment from the testUnit file and stores each line in an array.[18]
- “`/* ANN 4 */`”: The very first test class object is retrieved from the list.
- “`/* ANN 5 */`”: Using the public routine `makeDirTestInfo` the supplied test class object will get a directory object stored with it, that gets created from the supplied array containing the top comment of the testUnit file. That directory object is then retrieved and the information about the name of the testUnit file is stored in that directory object as well.
- “`/* ANN 6 */`”: All remaining test class objects, if any at all, will get that directory object stored with them as well. This way it is possible to retrieve the test unit comments from all of the class objects that are defined in that test unit program.
- “`/* ANN 7 */`”: The environment `.local` is searched for an entry named `bRunTestsLocally`. If this entry is not available, then one is created with a value of `.true` and as a result of this can then be retrieved via its environment symbol “`.bRunTestsLocally`”![19] Usually this entry is set by test runner programs, and if set to `.false`, then the next block would not be executed.
- “`/* ANN 8 */`”: This block will be executed only, if `.bRunTestsLocally` is set to `.true`. It will create a test suite to contain one test suite per test class, which itself will contain one test case per test method.[20] The constructor of the `testSuite` class will by default pick only those test methods from the supplied test class object whose names start with the string “`test`” to create test cases from them.
- “`/* ANN 9 */`”: A test suite program will always return a list of array objects, where each array object will store the test case class at index 1 and the list of

[17] If the list is not empty, one could specify any method, i.e., the method's name does *not* need to start with the string “`test`”.

[18] Please note, that the end of the comment is assumed if the block comment end character sequence (`*/`) is found alone on a line.

[19] In ooRexx entries in environment directory objects like `.local` or `.environment` can be retrieved by turning the index name into an “environment symbol”, which is a symbol that starts with a dot, immediately followed by the index name.

[20] The code in block “`/* ANN 8 */`” creates a total of three test cases (two test suites, and one test case for a test method). Running the first (“main”) test suite will therefore cause all of its contained test suites to be run (one per test class), which in turn will run all the test cases that were created from test methods.

mandatory test methods at index 2. This way test runner programs are able to retrieve all the test cases and their test methods by merely calling test unit programs.[21]

- "`/* ANN 10 */`": This directive causes the interpreter to call the ooRexxUnit framework program, which will make all its public classes and routines available to the test unit program.
- "`/* ANN 11 */`": This defines a test class. By convention there is one test class per test unit program, but if desired, one can define as many as one sees fit.
- "`/* ANN 12 */`": This is a test method which qualifies as a test method, that the ooRexxUnit framework would pick via reflection, as it starts with the string "`test`".
- "`/* ANN 13 */`": This is a test method which does *not* qualify as a test method, as it does not start with the string "`test`". Such a test method could still be used as a test case by including its name in the list of mandatory test methods.

Figure Output 5 depicts the output of running the test unit "`generic.testUnit`" as listed in figure Code 5 above.

```
nr of test runs:             3
nr of successful assertions: 1
nr of failures:              0
nr of errors:                0
```

Output 5: Output of Running "`rexx generic.testUnit`".[20]

ooRexx programmers who wish to adhere to the conventions layed out in this section, need to edit the code sections in figure Code 5 annotated with "`/* ANN 1 */`", "`/* ANN 2 */`" and "`/* ANN 11 */`" following. Everything else should be left as is. The next section introduces a simple test runner program which takes advantage of these conventions.

2.3.4.2 A Simple Test Runner Program

Figure Code 6 depicts a simple test runner program. It accepts two optional arguments, the switch "-r" for indicating recursive operation, and the name of a directory on which to operate. It will use the ooRexx `SysFileTree()` RexxUtil function to find all files matching the search pattern "`*.testUnit`". All found test unit files are then used to create a test suite which contains one test suite per testUnit file, which in turn contains all the repsective (mandatory) test cases. After running all test

[21] Such test runner applications will probably set the entry `bRunTestsLocally` in `.local` to `.false` which will inhibit the test unit programs to run their test cases locally.

cases, the brief results that got recorded in the test result obejct are shown with a heading.

```
/* usage: runTestUnits [-r[ecursive]] [directory]                      */
parse arg switch +2                 /* parse optional switch              */

hint="Processed"                    /* used in title for brief results below  */
switches="FO"                       /* SysFileTree()-switches             */
if switch~translate="-R" then       /* the "-R"[ecursive] switch in hand?     */
do
   switches=switches || "S"         /* add "S" switch for SysFileTree()       */
   parse arg . directory            /* parse optional directory           */
   hint=hint pp(switch)             /* used in title for brief results below  */
end
else
   parse arg directory              /* parse optional directory           */

   /* if running on Windows or OS2, then "\", "/" else   */
dirSlash=iif( pos(left(sysVersion(),1)~translate, "WO")>0, "\", "/")

if directory="" then directory="."        /* default to current directory     */
else if right(directory,1)=dirSlash then  /* remove possible trailing slash   */
   directory=directory~left(length(directory)-1)

searchFile=directory || dirSlash || "*.testUnit"   /* find all testUnit files */
call sysFileTree searchFile, "tests.", switches    /* search files            */
hint2="found" pp(tests.0) "file(s)" /* hint at the number of found files      */

list=.list~new                      /* create list of filenames           */
do i=1 to tests.0                   /* loop over found files              */
   list~insert(tests.i)
end

ts=makeTestSuiteFromFileList(list)  /* create testSuite object from file list */
testResult=ts~run                   /* run all the tests in the testSuite     */

   /* show test results as brief results  */
call simpleDumpTestResults testResult, hint pp(searchFile)"," hint2

::requires ooRexxUnit.cls
```

Code 6: The Test Runner Rexx Program "`runTestUnits.rex`".

The program in figure Code 6 is relatively short, because it takes advantage of the public routines `makeTestSuiteFromFileList()` and `simpleDumpTestResults()` that are defined in the ooRexxUnit framework.

Figure Output 6 displays the results of invoking the ooRexx program in figure Code 6 in a command line window containing the file "`generic.testUnit`" (figure Code 5 above) only, with the following command: "`rexx runTestUnits.rex`".

```
Processed [.\*.testUnit], found [1] file(s)

nr of test runs:             3
nr of successful assertions: 1
nr of failures:              0
nr of errors:                0
```

Output 6: Output of Running "`rexx runTestUnits.rex .`".

More advanced test runner applications could analyze and format all information about running the test cases as logged in the test result object.

2.3.4.3 A Concluding ".testUnit"-Program

Figure Code 7 depicts the concluding example ("finalExample.testUnit") of a test unit program, with some annotation markers on the right hand side margin. It defines the two test classes, "testClass_01" (cf. "/* ANN 7 */") and "testClass_02" (cf. "/* ANN 12 */"). For the first test class all methods starting with the string "test" should be used for creating test cases, hence the list for mandatory methods (cf. "/* ANN 1 */") is left empty. For the second test class only the method named "hi" (cf. "/* ANN 15 */") is regarded to be a mandatory test method, even though its name does not start with the string "test". Therefore the list of mandatory test methods is used to specify the name of this mandatory method (cf. "/* ANN 4 */").

```
/* list of array objects (pairs of testUnit class object and its test methods
*/
mandatoryTestMethods=.list~new   /* list of mandatory tests                   */      /* ANN 1 */
testUnitList=.list~of( .array~of(.testClass_01,  mandatoryTestMethods) )              /* ANN 2 */

      /* now supply the second test class that got defined in this file       */
mandatoryTestMethods=.list~of("hi")    /* define sole mandatory test            */      /* ANN 3 */
testUnitList~insert( .array~of(.testClass_02, mandatoryTestMethods) )                 /* ANN 4 */

--------------------------------------------------------------------------------
arrLines=.array~new              /* read top comment of this file            */
do i=1 to 150 until arrLines[i]="*/"
   arrLines[i]=sourceline(i)
end

aTestUnitClass=testUnitList~at(testUnitList~first)[1] /* get first testClass  */

   /* will parse the array lines and store result in class object's directory */
call makeDirTestInfo aTestUnitClass, arrLines
tmpDir=aTestUnitClass~TestCaseInfo  /* get directory containing parsed infos  */
parse source s       /*  op_sys invocationType fullPathToThisFile            */
tmpDir~setentry("test_Case-source", s)

do arr over testUnitList   /* add this directory to the other testCase classes*/
   if arr[1]=aTestUnitClass then iterate  /* already handled                  */
   arr[1]~TestCaseInfo=tmpDir             /* save info with class object      */
end

if .local~hasentry("bRunTestsLocally")=.false then /* entry not in .local ?   */
   .local~bRunTestsLocally=.true /* define entry, let all tests run           */

if .bRunTestsLocally=.true then  /* run ALL tests in this test unit           */
do
   ts=.testSuite~new             /* create a testSuite                        */      /* ANN 5 */
   do arr over testUnitList      /* iterate over testUnits                    */
      ts~addTest( .testSuite~new(arr[1])) /* create testSuite                 */      /* ANN 6 */
   end
   testResult=ts~run             /* now run all the tests                     */

   call simpleDumpTestResults testResult  /* show brief results               */
end

return testUnitList  /* return list of array objects containing the testUnits */


::requires ooRexxUnit.cls        /* load the ooRexxUnit classes               */

::class "testClass_01"  subclass TestCase public   /* test class              */      /* ANN 7 */
```

```
::method "test_01"                /* first test method                          */      /* ANN 8 */
   say "-->" pp(self~class~string)", method:" pp("test_01")                              /* ANN 9 */
   self~assertEquals("subTest_01", "1", 0+2-1)

::method "hi"                     /* some other test method                     */      /* ANN 10 */
   say "-->" pp(self~class~string)", method:" pp("hi")                                   /* ANN 11 */
   self~assertEquals("subTest_01", "1", 0+1)

::class "testClass_02"  subclass TestCase public   /* test class                 */      /* ANN 12 */
::method "test_01"                /* first test method                          */      /* ANN 13 */
   say "-->" pp(self~class~string)", method:" pp("test_01")                              /* ANN 14 */
   self~assertEquals("subTest_01", "ABC", "A"||"B"||"C")

::method "hi"                     /* some other test method                     */      /* ANN 15 */
   say "-->" pp(self~class~string)", method:" pp("hi")                                   /* ANN 16 */
   str=a||b||c
   self~assertEquals("subTest_01", "ABC", str)  /* comparing with "="          */
   self~assertSame("subTest_02", "ABC", str)    /* comparing with "=="         */
```

Code 7: The Rexx Program "`finalExample.testUnit`".

All the test methods possess a statement that outputs the information about the test case class they got defined in as well as their names (cf. "`/* ANN 9, 11, 14, 16 */`"). This way it becomes possible to study the effects of defining mandatory methods.

Running this test unit program locally, i.e., invoking it directly with the Rexx interpreter (issuing the command "`rexx finalExample.testUnit`" from a command line window) will yield the output shown in figure Output 7a below. As can be seen, the information about mandatory test methods gets ignored, rather the default behavior takes place which honors only test methods that start with the string "`test`".[22] Therefore test cases were created and run for the test methods defined at the lines annotated with "`/* ANN 8 */`" and "`/* ANN 13 */`", the other test methods (cf. "`/* ANN 10, 15 */`") are ignored. Add to this the test suite in "`/* ANN 5 */`" and the two test suites that get created in "`/* ANN 6 */`" (one for each test class) which all run, adding up to a total of five (test) runs.

```
--> [The testClass_01 class], method: [test_01]
--> [The testClass_02 class], method: [test_01]
nr of test runs:             5
nr of successful assertions: 2
nr of failures:              0
nr of errors:                0
```

Output 7a: Output of Running "`rexx finalExample.testUnit`".

Using the Rexx program "`runTestUnits.rex`" from figure Code 7 (issuing the command "`rexx runTestUnits.rex .`" from a command line window) will produce the output as shown in figure Output 7b below. As can be seen from there, for the first test class ("`testClass_01`") all standard test methods are used to create the test cases (i.e., "`/* ANN 8 */`"), whereas for the second test class ("`testClass_02`") the

[22] This behavior follows from the `TestSuite` constructor, in which the test methods which start with the string "`test`" are chosen to be used as test cases. As in this example each method carries out one assertion, there is a total of two successful assertions.

mandatory test method "hi" (cf. "/* ANN 15 */") is used for creating a test case, all other test methods of that test class are ignored![23] The public routine "makeTestSuiteFromFileList(…)" will create a test suite object to which test suites created for each test class will be added. Altogether there are three test suites and two test cases that run, adding up to five (test) runs.

```
--> [The testClass_01 class], method: [test_01]
--> [The testClass_02 class], method: [hi]
Processed [.\*.testUnit], found [1] file(s)

nr of test runs:              5
nr of successful assertions:  3
nr of failures:               0
nr of errors:                 0
```

Output 7b: Output of Running "rexx runTestUnits.rex .".

The differences in the output between Output 7a and Output 7b above are marked with a bold font and highlighted with a bright yellow colour.

3 Summary and Outlook

This paper introduced and briefly described the new ooRexx based framework "ooRexxUnit" which allows for creating ooRexx test cases with those concepts that are described for the Java test unit framework "JUnit" [W3JU3]. The ooRexx classes Assert, TestCase, TestSuite and TestResult have been introduced, explained and demonstrated with short, "nutshell"-like ooRexx programs.

In addition there were ooRexxUnit conventions introduced that should enable the creation of more comprehensive test runner applications, which would help automate running the test units. Such test runners could possess a graphical user interface (GUI), or could store the results of running test units and allow to compare them, and the like.

There are a few test units that have been created for testing the ooRexx interpreter and can be located at [W3OU3]. It is hoped that the Rexx community will help out creating test units that cover all aspects of ooRexx in the years to come. To this end, this article is hoped to enable Rexx and ooRexx programmers to understand the ooRexxUnit framework in detail. The acquired knowledge could also be applied for creating test units for ooRexx applications, be they opensource, free or commercial.

A last word ad the "JUnit" framework. With the introduction of Java version 5 the programming language Java gained the ability to annotate Java types (classes, interfaces, enumerations), fields, methods, parameters, constructors, local variables and catch

[23] This behavior follows from "runTestUnits.rex" employing the ooRexxUnit framework's public routine named "makeTestSuiteFromFileList(…)", which honors the list of mandatory test methods.

clauses, annotations and package definitions. The JUnit version 4 framework (cf. [W3JU4], [W3JU4Doc], [W3G06]) takes advantage of Java annotations and, among other things, forgoes the need to force test method names to start with the string "test" in order to pick those methods that should serve as test cases via the Java reflection mechanism. Rather, Java annotations are used instead to inform the JUnit framework about what roles what methods play, e.g. the `@Test` annotation precedes those methods, that should be used as test cases.[24] As ooRexx does not allow for annotations, the ooRexxUnit framework cannot exploit them. It can be expected though, that once ooRexx gains annotations, the ooRexxUnit framework gets adapted accordingly.

Acknowledgements

The author wishes to thank Walter Pachl for his valuable comments, suggestions and for proof reading the article.

4 References

[Cow90] Cowlishaw, M.F.: "The REXX Language", Prentice-Hall (Second edition), 1990.

[Fos05] Fosdick H.: "Rexx Programmer's Reference", John Wiley & Sons, ISBN: 0-7645-7996-7, URL (as of 2006-04-01): `http://www.wrox.com/WileyCDA/WroxTitle/productCd-0764579967.html`

[cetRexx] URL (as of 2006-04-01): `http://www.cetus-links.org/oo_rexx.html`

[ooRexx] URL (as of 2006-04-01): `http://www.ooRexx.org`

[Rexx] URL (as of 2006-04-01): `http://www.Rexx.org`

[RexxInfo] URL (as of 2006-04-01): `http://www.RexxInfo.org/`

[RexxLA] URL (as of 2006-04-01): `http://www.RexxLA.org`

[VeTrUr] Veneskey G.L., Trosky W., Urbaniak J.J.: "Object Rexx by Example", Aviar. URL (as of 2006-04-01): `http://www.oops-web.com/orxbyex/`

[W3G06] Goncalves A.: "Get Acquainted with the New Advanced Features of JUnit 4", devx.com, July 24th 2006, URL (as of 2006-04-01): `http://www.devx.com/Java/Article/31983/1954?pf=true`

[W3JU3] Homepage of JUnit 3. URL (as of 2006-04-01): `http://junit.sourceforge.net/junit3.8.1/`

[24] The `@Test` annotation allows for two arguments, one that indicates whether an exception is expected, and one which allows to determine how much time a test case has available to complete. There are also annotations for indicating which methods serve the role of "startUp" and "tearDown", which ones should be explitily ignored for the time being, and the like.

[W3JU3Cook] N.N.: “JUnit A Cook's Tour”. URL (as of 2006-04-01): `http://junit.sourceforge.net/doc/cookstour/cookstour.htm`

[W3JU3Doc] Javadocs for JUnit 3.x. URL (as of 2006-04-01): `http://www.junit.org/junit/javadoc/`

[W3JU4] Homepage of JUnit 4. URL (as of 2006-04-01): `http://junit.sourceforge.net/`

[W3JU4Doc] Homepage of JUnit 4. URL (as of 2006-04-01): `http://junit.sourceforge.net/javadoc_40/`

[W3OU3] Code repositories for ooRexx and ooRexxUnit (as of 2006-04-01): `http://oorexx.cvs.sourceforge.net/oorexx/` or `http://oorexx.svn.sourceforge.net/viewvc/oorexx/`