ooRexx Documentation 5.0.0

Open Object Rexx

The ooTest Framework Reference

ooRexx Documentation 5.0.0 Open Object Rexx
The ooTest Framework Reference
Edition 2022.12.22

Author	W. David Ashley
Author	Rony G. Flatscher
Author	Mark Hessling
Author	Rick McGuire
Author	Lee Peedin
Author	Oliver Sims
Author	Jon Wolfers

Preface

ooTest in the broadest sense represents all the tools and components used to test ooRexx and all of the tests written to date. Minus the written tests, ooTest is a testing framework that facilitates the testing of the ooRexx interpreter. It sits on top of the ooRexxUnit framework which is a generic testing framework that anyone can use to test their Rexx applications.

The audience for this book is primarily those interested in testing the ooRexx interpreter. Either by executing the test suite or by writing test cases. In the process of explaining how to write test cases for the ooRexx interpreter, the book covers the ooRexxUnit and ooTest frameworks. For this reason portions of the book may be of interest to persons wishing to use ooRexxUnit to test their own Rexx applications.

This book does not provide information on how to program using ooRexx. The assumption is that the reader is already familar with the intepreter and, to some degree, the use of objects in ooRexx. The book provides a reference to **`testOORexx.rex`**, the ooRexxUnit framework, the ooTest framework, and information on how to write tests of the ooRexx interpreter package using the ooTest framework.

The book is roughly organized this way:

- An overview (*Chapter 2, Overview of ooTest*) of ooTest.
- A chapter documenting testOORexx.rexx (*Chapter 3, testOORexx.rex the Automated Test Driver*) which is currently the one and only test driver propgram.
- Chapter(s) documenting the ooRexxUnit (*Chapter 4, The ooRexxUnit Framework*) framework.
- Chapter(s) documenting the ooTest (*Chapter 5, The ooTest Framework*) framework.
- Chapter(s) giving some guidence (*Chapter 6, Writing Tests*) in writing test cases using the ooTest framework.

1. Document Conventions

This manual uses several conventions to highlight certain words and phrases and draw attention to specific pieces of information.

1.1. Typographic Conventions

Typographic conventions are used to call attention to specific words and phrases. These conventions, and the circumstances they apply to, are as follows.

`Mono-spaced Bold` is used to highlight literal strings, class names, or inline code examples. For example:

> The **`Class`** class comparison methods return **`.true`** or **`.false`**, the result of performing the comparison operation.
>
> This method is exactly equivalent to **`subWord(`*`n`*`, 1)`**.

`Mono-spaced Normal` denotes method names or source code in program listings set off as separate examples.

> This method has no effect on the action of any `hasEntry`, `hasIndex`, `items`, `remove`, or `supplier` message sent to the collection.

```
-- reverse an array
a = .Array~of("one", "two", "three", "four", "five")
```

```
-- five, four, three, two, one
aReverse = .CircularQueue~new(a~size)~appendAll(a)~makeArray("lifo")
```

Proportional Italic is used for method and function variables and arguments.

> A supplier loop specifies one or two control variables, *index*, and *item*, which receive a different value on each repetition of the loop.
>
> Returns a string of length *length* with *string* centered in it and with *pad* characters added as necessary to make up length.

1.2. Notes and Warnings

Finally, we use three visual styles to draw attention to information that might otherwise be overlooked.

Note

Notes are tips, shortcuts or alternative approaches to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.

Important

Important boxes detail things that are easily missed, like mandatory initialization. Ignoring a box labeled 'Important' will not cause data loss but may cause irritation and frustration.

Warning

Warnings should not be ignored. Ignoring warnings will most likely cause data loss.

2. How to Read the Syntax Diagrams

Throughout this book, syntax is described using the structure defined below.

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

  The **>>---** symbol indicates the beginning of a statement.

  The **--->** symbol indicates that the statement syntax is continued on the next line.

  The **>---** symbol indicates that a statement is continued from the previous line.

  The **---><** symbol indicates the end of a statement.

  Diagrams of syntactical units other than complete statements start with the **>---** symbol and end with the **--->** symbol.

- Required items appear on the horizontal line (the main path).

```
>>-STATEMENT--required_item------------------------------------><
```

- Optional items appear below the main path.

```
>>-STATEMENT--+---------------+--------------------------------><
              +-optional_item-+
```

- If you can choose from two or more items, they appear vertically, in a stack. If you must choose one of the items, one item of the stack appears on the main path.

```
>>-STATEMENT--+-required_choice1-+-----------------------------><
              +-required_choice2-+
```

- If choosing one of the items is optional, the entire stack appears below the main path.

```
>>-STATEMENT--+------------------+-----------------------------><
              +-optional_choice1-+
              +-optional_choice2-+
```

- If one of the items is the default, it appears above the main path and the remaining choices are shown below.

```
              +-default_choice--+
>>-STATEMENT--+-----------------+------------------------------><
              +-optional_choice-+
              +-optional_choice-+
```

- An arrow returning to the left above the main line indicates an item that can be repeated.

```
              +-----------------+
              V                 |
>>-STATEMENT----repeatable_item-+------------------------------><
```

  A repeat arrow above a stack indicates that you can repeat the items in the stack.

- A set of vertical bars around an item indicates that the item is a fragment, a part of the syntax diagram that appears in greater detail below the main diagram.

```
>>-STATEMENT--| fragment |-------------------------------------><
```

  fragment:

```
|--expansion_provides_greater_detail----------------------------|
```

- Keywords appear in uppercase (for example, **`PARM1`**). They must be spelled exactly as shown but you can type them in upper, lower, or mixed case. Variables appear in all lowercase letters (for example, **`parmx`**). They represent user-supplied names or values.

- If punctuation marks, parentheses, arithmetic operators, or such symbols are shown, you must enter them as part of the syntax.

The following example shows how the syntax is described:

```
          +-,------+
          V        |
>>-MAX(----number-+--)-----------------------------------------><
```

3. Getting Help and Submitting Feedback

The Open Object Rexx Project has a number of methods to obtain help and submit feedback for ooRexx and the extension packages that are part of ooRexx. These methods, in no particular order of preference, are listed below.

3.1. The Open Object Rexx SourceForge Site

The *Open Object Rexx Project*[1] utilizes *SourceForge*[2] to house the *ooRexx Project*[3] source repositories, mailing lists and other project features. Over time it has become apparent that the Developer and User mailing lists are better tools for carrying on discussions concerning ooRexx and that the Forums provided by SourceForge are cumbersome to use. The ooRexx user is most likely to get timely replies from one of the mailing lists.

Here is a list of some of the most useful facilities provided by SourceForge.

The Developer Mailing List

> You can subscribe to the oorexx-devel mailing list at *ooRexx Mailing List Subscriptions*[4] page. This list is for discussing ooRexx project development activities and future interpreter enhancements. It also supports a historical archive of past messages.

The Users Mailing List

> You can subscribe to the oorexx-users mailing list at *ooRexx Mailing List Subscriptions*[5] page. This list is for discussing using ooRexx. It also supports a historical archive of past messages.

The Announcements Mailing List

> You can subscribe to the oorexx-announce mailing list at *ooRexx Mailing List Subscriptions*[6] page. This list is only used to announce significant ooRexx project events.

The Bug Mailing List

> You can subscribe to the oorexx-bugs mailing list at *ooRexx Mailing List Subscriptions*[7] page. This list is only used for monitoring changes to the ooRexx bug tracking system.

Bug Reports

> You can create a bug report at *ooRexx Bug Report*[8] page. Please try to provide as much information in the bug report as possible so that the developers can determine the problem as quickly as possible. Sample programs that can reproduce your problem will make it easier to debug reported problems.

[1] http://www.oorexx.org/
[2] http://sourceforge.net/
[3] http://sourceforge.net/projects/oorexx
[4] http://sourceforge.net/mail/?group_id=119701
[5] http://sourceforge.net/mail/?group_id=119701
[6] http://sourceforge.net/mail/?group_id=119701
[7] http://sourceforge.net/mail/?group_id=119701
[8] http://sourceforge.net/tracker/?group_id=119701&atid=684730

Documentation Feedback

You can submit feedback for, or report errors in, the documentation at *ooRexx Documentation Report*[9] page. Please try to provide as much information in a documentation report as possible. In addition to listing the document and section the report concerns, direct quotes of the text will help the developers locate the text in the source code for the document. (Section numbers are generated when the document is produced and are not available in the source code itself.) Suggestions as to how to reword or fix the existing text should also be included.

Request For Enhancement

You can suggest ooRexx features at the *ooRexx Feature Requests*[10] page.

Patch Reports

If you create an enhancement patch for ooRexx please post the patch using the *ooRexx Patch Report*[11] page. Please provide as much information in the patch report as possible so that the developers can evaluate the enhancement as quickly as possible.

Please do not post bug fix patches here, instead you should open a bug report and attach the patch to it.

The ooRexx Forums

The ooRexx project maintains a set of forums that anyone may contribute to or monitor. They are located on the *ooRexx Forums*[12] page. There are currently three forums available: Help, Developers and Open Discussion. In addition, you can monitor the forums via email.

3.2. The Rexx Language Association Mailing List

The *Rexx Language Association*[13] maintains a mailing list for its members. This mailing list is only available to RexxLA members thus you will need to join RexxLA in order to get on the list. The dues for RexxLA membership are small and are charged on a yearly basis. For details on joining RexxLA please refer to the *RexxLA Home Page*[14] or the *RexxLA Membership Application*[15] page.

3.3. comp.lang.rexx Newsgroup

The *comp.lang.rexx*[16] newsgroup is a good place to obtain help from many individuals within the Rexx community. You can obtain help on Open Object Rexx or on any number of other Rexx interpreters and tools.

4. Related Information

See also: *Open Object Rexx: Reference*

[9] http://sourceforge.net/tracker/?group_id=119701&atid=1001880
[10] http://sourceforge.net/tracker/?group_id=119701&atid=684733
[11] http://sourceforge.net/tracker/?group_id=119701&atid=684732
[12] http://sourceforge.net/forum/?group_id=119701
[13] http://www.rexxla.org/
[14] http://rexxla.org/
[15] http://www.rexxla.org/rexxla/join.html
[16] http://groups.google.com/group/comp.lang.rexx/topics?hl=en

Quick Guide to Getting Started

1.1. First Steps

We'll start with a quick explanation of how to get started and a little bit of an overview.

1.1.1. ooRexxUnit Snapshots

Download one of the ooRexxUnit snapshots from SourceForge. Through out this discussion I use the term ooRexxUnit and ooTest interchangeably. ooRexxUnit is a generic testing framework and the term first used when talking about testing ooRexx. Some things retain that name. What I am really concerned with is ooTest which is a framework whose sole purpose is to test the ooRexx interpreter.

As the ooRexx interpreter evolves, so does the ooTest framework. For that reason you need to grab a snapshot that matches the version of ooRexx that you will be writing and executing your tests with.

It can not be stressed enough how valuable a contribution to the ooRexx project test cases are. You do not need to write and execute test cases using the most current non-released version of ooRexx. Most people should write and execute test cases using the most recent released version of ooRexx. This provides a solid base of good test cases with known results. These test cases then make it far easier for the developers to improve and enhance ooRexx. After any code changes the suite of known working test cases can be executed to ensure the new code did not break anything.

An ooRexxUnit X.X.X Snapshot is a package released by the ooRexx team that contains both the testing framework and the set of tests used in testing the ooRexx interpreter at that X.X.X level. As the version name implies, it is a snapshot of ooRexxUnit at that point in time. The package contains all the currently available test groups written to test the X.X.X interpreter. These test groups are an excellent source of techniques for using ooRexxUnit.

The primary purpose of releasing snapshot versions of ooRexxUnit is to encourage individuals and organizations to contribute test cases for the interpreter.

Writing test groups to test the interpreter requires nothing other than an installed X.X.X interpreter and a framework package. If an individual has limited knowledge of ooRexx programming, then writing test cases will be a excellent way to learn the language.

Individuals wishing to contribute to ooRexx through writing test case can get advice and help by joining the oorexx-devel list. The list is free and open to anyone. Go to: *Subscribe oorexx-devel list* [1] to join the list.

In addition the RexxLA mailing list is also a good place to seek advice or help in writing test units. If not already a member of RexxLA, more information on the group can be found at: *RexxLA Home Page*[2]

1.1.2. Download

Go to the file download section of the ooRexx project on SourceForge: *SourceForge.net Files* [3] and download the snapshot for your version of ooRexx.

[1] https://sourceforge.net/mail/?group_id=119701
[2] http://rexxla.org/
[3] https://sourceforge.net/project/showfiles.php?group_id=119701

Most of the time the snapshots are packaged in both zip and tar format. However, the ooTest framework and files are completely platform independent. Either package will work on any system that ooRexx is working on. Pick the packaging type that is most convenient for your needs.

1.1.3. Extract the Files

Open a console window and unzip or untar the snapshot in a convenient spot. After unpackaging you will end up with a directory structure similar to this:

```
ooRexxUnit.X.X.X
   |
   *------*framework
   |          |
   |          *-----<subdirectories>
   |
   *------*misc
   |         |
   |         *-----<subdirectories>
   |
   *------*ooRexx
             |
             *-----<subdirectories>
```

The framework directory and subdirectories contain additional documentation and examples.

The ooRexx directory and subdirectories contain all the tests implemented using the ooTest framework. These sub-directories contain the tests used in testing the ooRexx interpreter.

The misc directory has, well miscellaneous stuff. The most important of which for this discussion is a template file that can be used to start your test group files and several examples of test group files.

1.1.4. Test Your Install

After unpackaging the snapshot, cd into the top level directory and read the ReadMe.first file. The top level directory will look similar to below:

```
E:\work.ooRexx\ooRexxUnit\3.2.0>dir
 Volume in drive E is Blackfoot
 Volume Serial Number is 9C3D-6D2A

 Directory of E:\work.ooRexx\ooRexxUnit\3.2.0

07/03/2008  08:00 PM    <DIR>          .
07/03/2008  08:00 PM    <DIR>          ..
07/03/2008  08:00 PM    <DIR>          framework
07/03/2008  08:06 PM    <DIR>          misc
07/03/2008  08:06 PM    <DIR>          ooRexx
11/29/2007  05:47 PM            11,839 CPLv1.0.txt
12/19/2007  09:15 PM             2,822 directory.structure.tests
06/30/2008  05:47 PM             1,737 Expected.Results
01/18/2008  10:39 AM            69,323 ooTest.frm
06/30/2008  05:47 PM             5,918 ReadMe.first
01/02/2008  08:03 PM             9,139 runTestGroups.rex
12/04/2007  08:07 PM             2,437 setTestEnv.bat
12/04/2007  08:06 PM             3,062 setTestEnv.sh
01/13/2008  01:15 PM             4,441 testOORexx.rex
01/12/2008  08:44 PM            18,190 worker.rex
              10 File(s)        128,908 bytes
               6 Dir(s)  14,359,519,232 bytes free
```

Note

For the sake of this document I am going to show examples on Windows. But, the same general thing applies to Linux. Just translate the slashes. In addition the examples are from the 3.2.0 version of ooTest. The same general principles apply to whatever snapshot you have.

The program file **`testOORexx.rex`** is what drives the automated execution of the test cases. Provided that you have a standard ooRexx install, you can execute the entire test suite as follows. This command will execute the entire test suite. Depending on your system itwill take several minutes to finish.

```
E:\work.ooRexx\ooRexxUnit\3.2.0>rexx testOORexx.rex -V 5
Searching for test containers.......................................
Executing automated test suite...........................................
...........................................................................
..........

ooTest Framework - Automated Test of the ooRexx Interpreter


Interpreter: REXX-ooRexx_3.2.0(MT) 6.02 30 Oct 2007
ooRexxUnit:  2.0.0_3.2.0        ooTest: 1.0.0_3.2.0

Tests ran:           16600
Assertions:          537835
Failures:            2
  (Known failures:)  42
Errors:              2
Exceptions:          0
Skipped files:       0
Messages:            0

[failure] [20080819 17:37:57.809000]
  Test:   TEST_MULTIPLE_INHERITANCE_WITH_MULTIPLE_METACLASSES
  Class:  Class.testGroup
  File:   E:\work.ooRexx\ooRexxUnit\3.2.0\ooRexx\base\class\Class.testGroup
  Line:   576
  Failed: assertEquals
    Expected: [['123.'], identityHash="495954478"]
    Actual:   [['231.'], identityHash="421451928"]

[failure] [20080819 17:37:57.825000]
  Test:   TEST_SUBCLASSES
  Class:  Class.testGroup
  File:   E:\work.ooRexx\ooRexxUnit\3.2.0\ooRexx\base\class\Class.testGroup
  Line:   684
  Failed: assertTrue
    Expected: [1]
    Actual:   [[0], identityHash="535806184"]

[error] [20080819 17:37:57.809000]
  Test:  TEST_MIXINCLASS_01
  Class: Class.testGroup
  File:  E:\work.ooRexx\ooRexxUnit\3.2.0\ooRexx\base\class\Class.testGroup
  Event: [SYNTAX 97.1] raised unexpectedly.
    Object "NOT_AN_EXISTING_CLASS" does not understand message "NEW"
    Line:    509
   509 *-* cl=.object~mixinclass("subTest_01", not_an_existing_class)
```

```
[error] [20080819 17:37:57.825000]
  Test:  TEST_SUBCLASS_01
  Class: Class.testGroup
  File:  E:\work.ooRexx\ooRexxUnit\3.2.0\ooRexx\base\class\Class.testGroup
  Event: [SYNTAX 97.1] raised unexpectedly.
    Object "NOT_AN_EXISTING_CLASS" does not understand message "NEW"
    Line:    616
   616 *-* cl=.object~subclass("subtest_01", not_an_existing_class)


Interpreter: REXX-ooRexx_3.2.0(MT) 6.02 30 Oct 2007
ooRexxUnit:  2.0.0_3.2.0        ooTest: 1.0.0_3.2.0

Tests ran:           16600
Assertions:          537835
Failures:            2
  (Known failures:)  42
Errors:              2
Exceptions:          0
Skipped files:       0
Messages:            0

File search:        00:00:51.456000
Suite construction: 00:00:02.000000
Test execution:     00:02:17.632000
Total time:         00:03:11.620000


E:\work.ooRexx\ooRexxUnit\3.2.0>
```

You should get results similar to the above. Included with the snapshot is a file called **`Expected.results`**. That file should show you results similar to what you actually get.

1.1.5. Understanding What you See

We looked at the entire output from running the automated test suite. Let's examine some of the sections in more detail. The below stats show:

```
Interpreter: REXX-ooRexx_3.2.0(MT) 6.02 30 Oct 2007
ooRexxUnit:  2.0.0_3.2.0        ooTest: 1.0.0_3.2.0

Tests ran:           16600
Assertions:          537835
Failures:            2
  (Known failures:)  42
Errors:              2
Exceptions:          0
Skipped files:       0
Messages:            0

File search:        00:00:51.456000
Suite construction: 00:00:02.000000
Test execution:     00:02:17.632000
Total time:         00:03:11.620000
```

that 16,600 tests ran, comprising 537,835 assertions, taking about 3 minutes total to finish. There were 2 failures, 42 known failures, and 2 errors.

For a little terminology. When you write a test case there are 2 expected outcomes. It is expected that the test case either passes or fails. So in ooTest, a failure is a test case that did not pass. An error is an **unexpected** event. We are going to ignore the errors for now. In general they indicate something is wrong, maybe with the framework, maybe with the interpreter.

I'm going to just explain one of the failures. You have this output:

```
[failure] [20080819 17:37:57.809000]
  Test:   TEST_MULTIPLE_INHERITANCE_WITH_MULTIPLE_METACLASSES
  Class:  Class.testGroup
  File:   E:\work.ooRexx\ooRexxUnit\3.2.0\ooRexx\base\class\Class.testGroup
  Line:   576
  Failed: assertEquals
    Expected: [['123.'], identityHash="495954478"]
    Actual:   [['231.'], identityHash="421451928"]
```

The above shows that the name of the failed test was **`TEST_MULTIPLE_INHERITANCE_WITH_MULTIPLE_METACLASSES`** and it can be found on line 576 in the file **`Class.testGroup`**.

What failed was an assertion that two things were equal. The thing was expected to be equal to '123' but it actually was '231'

Unfortunately these failing two test are a little complex, maybe not the best for an intro, but to continue with what we have. If we look at the failing test file on line 576 we see:

```
  self~assertEquals("'123.'", .class123~info)
```

Okay, it has a method, but just bear with me for a second. This is the heart of writing a test case.

```
  assertEquals("123", .class123~info)
```

Think of the method just as a routine for now with some prefix on it that you do not need to worry about. The routine name is assertEquals. The routine has 2 args and what we are doing is asserting that the 2 args are equal.

If they are not equal, the test case fails and the ooTest framework takes care of all the details of reporting the failure. That is the print out you see above. For now, don't worry about how the framework does this, just accept that it does it.

Next, let's look at the lines around 567, and you see this:

```
::method "test_MULTIPLE_INHERITANCE_WITH_MULTIPLE_METACLASSES"

     self~assertEquals("'123.'", .class123~info)
```

Think of the method as a routine, for now, that is named: **`test_MULTIPLE_INHERITANCE_WITH_MULTIPLE_METACLASSES`**. And that is the test name, the name reported in the print out. Next is the assertion, **`assertEquals()`**.The test writer is asserting that **`"123"`** is equal to **`.class123~info`**. The assertion failed, the test failed.

You and I don't need to figure out what **`.class123~info`** is right now.

Enough for an introduction.

1.2. Starting to Write a Test Case

In the last section we looked at this, which I said was the heart of a test case:

```
  ::method "test_MULTIPLE_INHERITANCE_WITH_MULTIPLE_METACLASSES"
```

```
        self~assertEquals("'123.'", .class123~info)
```

In this section we will start to write our own test case.

1.2.1. A First Test Case

Say we want to test that the interpreter is adding 2 + 3 correctly. We expect our test case either to show that the interpreter is adding 2 + 3 correctly or that it is not.

To write a test case, it has to be in a method, which always starts as follows:

```
    ::method
```

That is boilerplate, just type it. Then the next key thing is that the framework executes each method whose name begins with 'test', case not significant, as a test case. We need to name the test case method, so we do this:

```
    ::method test_simpleAddition
```

That's also boilerplate, make up a name, start it with test. I usually add the underscore, but it is not needed.

What's the test? We know that 2 + 3 is 5, so if the interpreter adds 2 + 3 we would expect the result to be 5. We write some code that adds 2 + 3 and then assert that the result is 5:

```
    ::method test_simpleAdditon
        val = 2 + 3
        self~assertEquals(5, val)
```

That is all there is to it. The above 3 lines of code will test that the interpreter adds 2 plus 3 correctly. The framework takes care of all the other details.

Well there is a little more to it, we need to add the test case methods to a file, I'll show that later.

To summarize, to get started: Think of **`::method`** as a the beginning of a function definition, where the function name comes right after the **`::method`**. Think of **`assertEquals()`** as a function call. And just note that the function call has to start with a prefix of **`'self~'`**. You do not need to really understand the **`'self~'`** part at first. Just know that you have to add it. We'll worry about learning about classes some other time. This is enough to write test cases.

1.2.2. Contributing to the ooRexx Project

At this point, I'm going to interject a little info on contributing to the ooRexx project / contributing to the ooRexx test suite.

The test suite is enormously helpful to the project. Everyone and any one can contribute to this effort. Even though we have a large number of tests, we still have a lot of holes in the coverage.

There are 4 main ways someone can contribute:

- You could write a whole new **`*.testGroup`** file.
- You can add tests to an existing **`*.testGroup`** file.

- You can examine existing test case code for correctness and correct or improve wrong or weak test cases.
- You can help organize or promote the contribution of test cases into the project.

It is my strong opinion that all the **`.testGroup`** files in the test suite are works in progress. In addition, we could really use more pairs of eye examining the existing .testGroup files with a critical eye.

Any one and every one is encouraged to either add to an existing file or to say hey this test is wrong, it should be written this way. Or to say, hey the **`Lines.testGroup`** does not have a test for this option, here's a method that does test it.

The other thing that is really important, is that just because something is working now, that doesn't mean there shouldn't be a test for it. There is no guarantee that anything that is working today will be working after I make my next commit. If I break something with my next commit, and we have a test case for it, it will show up right away and get fixed.

For example, you may think that there is no way the say instruction could be broken. But what about if you have this in a Rexx file:

```
/* Simple.rex */
say 'Hello World'
```

and then run it like this:

```
E:\>rexx Simple.rex > myTest.file
```

You would expect **`myTest.File`** to contain 1 line consisting of:

```
 Hello World<endOfLineCharacter>
```

Well, a real life bug we just had was that instead of the normal Windows end of line which is 0x0d0a we were actually getting 0x0d0d0a.

Now, running the test suite did catch this, but not because we had a test for it. The test suite caught it because some other tests were failing with what appeared to be no reason. It took quite awhile to locate what was really wrong.

We did not, and still don't, have a simple test case that asserts that a say statement redirected to a file produces what it should. The simple test should assert that the bytes in the file are exactly 13 and then assert that byte 1 is 0x48, byte 2 is 0x45, ... byte 12 is 0x0d, byte 13 is 0x0a.

Next section, create a testGroup file from the template and start a test group.

1.3. Starting a Test Group from Scratch

In the ooTest framework, tests are orgainized as follows. An assertation is a single test. A single method represents a test case. Each test case would contain at least one assertion, but often a test case will contain several assertation. Then a number of test cases for a similar area are gathered together in a file, which we call a test group.

1.3.1. Starting a Test Group

I'm going to assume that the reader has been following along in the thread and try not to repeat myself. The idea behind this thread is to show how to start writing test cases even if you do not understand classes by using boilerplate code.

If you downloaded a snapshot and unzipped it, it will have created a directory tree. We start in the root of the tree which will be named ooRexxUnit.X.X.X.

There is the misc/ subdirectory. It has a template file. Pick a name for the test group and figure out roughly where it should go. Copy the template to the subdirectory renaming it in the copy. I am going to work with the stream BIF, since we do not even have a test group started for that important BIF. This example is on Linux

```
Raven:/ooRexxUnit.3.2.0 # cp misc/template.testGroup ooRexx/base/bif/STREAM.testGroup
```

You don't even have to put it in the proper subdirectory. To get started you could just put it anywhere under the ooRexx subdirectory.

```
Raven:/ooRexxUnit.3.2.0 # cp misc/template.testGroup ooRexx/STREAM.testGroup
```

1.3.2. Editing STREAM.testGroup

Open the file in an editor. The first thing I do is a search and replace of **`template.testGroup`** with **`STREAM.testGroup`**. I'm not going to give editor lessons, but if you do not understand classes, be sure you don't skip this first step.

Then starting at the top of the editor we have the first 41 lines that you just ignore and leave alone. The first few are:

```
    #!/usr/bin/env rexx
    /*
     SVN Revision: $Rev: 2267 $
     Change Date:  $Date: 2008-01-18 09:41:04 -0800 (Fri, 18 Jan 2008) $
    */
```

Which are just book keeping. The **`#!/usr/bin/env rexx`** causes the file to execute as a script on Linux, etc. The next is just svn book keeping. Then there is the license text.

Starting on line 42 through 51 we have what I think of as the entry point to an ooRexx program that has directives in it. You do not need to understand this to start off with, it is the code that lets the framework automate the execution of tests. You just need to be sure you changed **`template.testGroup`** to **`STREAM.testGroup`**. The lines look like:

```
  parse source . . s

  group = .TestGroup~new(s)
  group~add(.STREAM.testGroup)

  if group~isAutomatedTest then return group

  testResult = group~suite~execute~~print

 return testResult
 -- End of entry point.
```

Briefly what this does is:

- Create a new TestGroup object using the full path name of the file.
- Add the **`STREAM.testGroup`** class to the test group
- Magically test if the invocation of the program is part of an automated test. If so the group object is returned and the code execution is done.

- If it is not an automated test, it is a stand alone test. The test group executes all the tests it contains and prints out the results, then returns the test result object. This allows you to execute the tests by just invoking the file as an ooRexx program. I am not going to go into details about that now.

When working on just your own test group file, it is best to just run the single test group by itself through the frame work. Right now you have a complete test group that will execute. You can run it using this command:

```
Raven:/ooRexxUnit.3.2.0 # ./testOORexx.rex -R ooRexx -f STREAM
```

(The above should work, but I can not try it right now, so I am going to show it on Windows.)

```
E:\ooRexxUnit.3.2.0>testOORexx.rex -R ooRexx\ -f STREAM
Searching for test containers..
Executing automated test suite..

ooTest Framework - Automated Test of the ooRexx Interpreter


Interpreter: REXX-ooRexx_3.2.0(MT) 6.02 30 Oct 2007
ooRexxUnit:  2.0.0_3.2.0        ooTest: 1.0.0_3.2.0

Tests ran:           4
Assertions:          2
Failures:            0
Errors:              0
Skipped files:       0

File search:        00:00:00.047000
Suite construction: 00:00:00.000000
Test execution:     00:00:00.000000
Total time:         00:00:01.000000

E:\ooRexxUnit.3.2.0>
```

Okay, that's it. You created and executed your first group of tests. 4 tests ran using 2 assertions with no failures or errors. On my system the total execution time was 1 second.

That takes care of all the initial steps. You have a working test group.

The next installment will show adding some test cases to the new test group.

1.3.3. STREAM.testGroup Continued

Picking up with writing the new test group **`STREAM.testGroup`**.

The last section showed the top boilerplate from the the template file. This picks up with the actual test case class. I'm not to going to explain too much about the class part of this, I'll save that for later. This is about just editing the boilerplate to quickly get started.

This is the rest of the file, leaving out some comment lines. Format your code however you like.

```
-- End of entry point.

::requires 'ooTest.frm'

::class "STREAM.testGroup" public subclass ooTestCase

::method test_YYY
    self~assertTrue(.true)
```

```
  ::method  test_XXX
      self~assertSame('dog', 'dog')
```

The **`::requires`** line pulls in the ooTest framework. That is what provides the backing code for our test cases.

Then we have our **`STREAM.testGroup`** class which is a subclass of ooTestCase. Once you do the replace of **`template.testGroup`** with **`STREAM.testGroup`** you are done with those lines. You can just leave the boilerplate alone and figure out what this means later.

Then, to restate what I've said earlier. Each individual test is a method of the test case class where the method names starts with 'test' Here our test case class is: **`STREAM.testGroup`** and we see that we have 2 methods: **`test_YYY`** and **`test_XXX`**.

That is why executing the **`STREAM.testGroup`** actually runs some tests, even though we didn't add any tests to the file yet.

If you don't understand classes yet, just think of the two methods as two routines, or functions, or procedures. Whichever terminology you are comfortable with. I'm going to call them methods, because that's what they are.

The 2 methods are in the template to jog your memory and get you started. I start out by editing the first method and coding an actual test. We are doing a group of tests for the stream BIF, so an easy way to get started is to look at the documentation for the stream BIF.

If you are following along and don't understand stream at all, the advice is that the principles here apply to writing any test cases, pick some area of Rexx you do understand. Plus, I have to add, if you don't understand stream, then digging in to it enough to write some test cases is a great way to learn about stream.

Looking at the doc, we see that it says stream returns a string, which string is dependent on the args to stream, and that the first arg names the stream to be worked with. The second arg can be one of: State, Command, or Description.

For the Description arg it says that it returns the same string as State, but with a colon at the end, maybe followed by some text describing an error or not ready condition. That suggests a test. If we use the Description arg, the returned string must have a colon in it.

1.3.4. Finally, the Interesting Part

This is the interesting part. If you like to program, then it is usually fun to think of a way to code a test for this. Here is one way.

The mechanics of this are to change the name of the first method. The restriction is that every method in the class has to have an unique name. If there is a failure, the name of the method gets printed out. If the name of the method reflects the test, it is a little easier to discern what the test is about. But, you could just name every method test_001, test_002, test_003 and so on.

Here goes, rename the method:

```

  ::method test_description_arg
```

One of the most common types of streams is file input or output. The name of the stream is the file name. One approach is to use a file name for the stream name, use the Description arg, and validate the return:

```
::method test_description_arg
    streamName = ???
    retString = stream(streamName, "Description")
    <check retString>
```

Remember this is going to be an automated test that should run on any system. We don't want to use a file name that is on your system, but is not on Rick's system. What file name are we absolutely sure exists?

The **`STREAM.testGroup`** file, for sure, because that is the file we are using. We can get that name from parse source. If you don't know parse source, look it up in the doc.

```
::method test_description_arg
    parse source junk notUsed streamName
    retString = stream(streamName, "Description")
    <check retString>
```

Now we just need to code the check of retString.

1.3.5. The Core, Validating Results

Here we are at the core of the ooTest framework. We validate expected results by using one of the assertXXX() methods. I will list the different assertXXX a little later.

In the template code we already had:

```
    self~assertTrue(.true)
```

The meaning of that should be easy enough to discern. We are asserting that .true is true.

Okay, our test is about the stated fact that the Description arg must return a string with a colon in it. We need to write some code that we can get true out of. There are a number of ways to do this. I'll use a BIF since this is directed toward people who may not know classes too well.

We know that if the returned string has a colon in it, then the pos bif will return a non-zero position for the colon. Here is the complete test case:

```
::method test_description_arg
    parse source junk notUsed streamName
    retString = stream(streamName, "Description")
    p = pos(retSting, ":")
    self~assertTrue(p > 0)
```

1.3.6. The Finale, Executing our Test

And here is the output from executing this test group (stripped of a little bit):

```
C:\ooRexxUnit.3.2.0>rexx testOORexx.rex -R ooRexx -f stream
...

Tests ran:           4
Assertions:          1
Failures:            1
Errors:              0
Skipped files:       0

[failure] [20080701 07:48:09.882000]
 Test:   TEST_DESCRIPTION_ARG
```

```
    Class:  STREAM.testGroup
    File:   C:\work.ooRexx\ooRexxUnit\3.2.0\ooRexx\STREAM.testGroup
    Line:   68
    Failed: assertTrue
      Expected: [1]
      Actual:   [[0], identityHash="535806184"]
```

This is perfect! Okay, the output shows that the test named **TEST_DESCRIPTION_ARG** failed. Whoa, that's the one I just wrote. It shows that what failed was assertTrue and that it was on line 68.

What was expected was 1 (true) but the actual was 0 (false) Let's look at the test case again:

```
p = pos(retSting, ":")
self~assertTrue(p > 0)
```

I always forget the args to pos. They are needle, haystack. I coded haystack, needle. Well the astute observer may have noticed that I also coded retSting rather than retString. The fixed test case:

```
 ::method test_description_arg
     parse source junk notUsed streamName
     retString = stream(streamName, "Description")
     p = pos(":", retString)
     self~assertTrue(p > 0)
```

The output:

```
 Interpreter: REXX-ooRexx_3.2.0(MT) 6.02 30 Oct 2007
 ooRexxUnit:  2.0.0_3.2.0        ooTest: 1.0.0_3.2.0

 Tests ran:           4
 Assertions:          2
 Failures:            0
 Errors:              0
 Skipped files:       0

 File search:        00:00:00.160000
 Suite construction: 00:00:00.000000
 Test execution:     00:00:00.000000
 Total time:         00:00:00.160000
```

Alright. Now, if someone rewrote the stream libraries, a lot of work, and had some trivial error that forgot to tack on the ":" for the Description arg, we have a test case that would catch it.

1.3.7. One More Quick Test

Let's do one more quickly.

The doc also says that the Description arg returns the same string as the State arg, with a colon and some other possible text added after the colon. That suggests a test.

Here is the code. Notice the use of **`assertSame()`** which really fits in with the semantics of the test. We are testing that the Description and State args return the same string:

```
 ::method  test_description_state_same
     parse source junk notUsed streamName

     retDiscrpt = stream(streamName, "Description")
     retState   = stream(streamName, "State")
```

```
      retDiscrpt = left(retDiscrpt, pos(":", retDiscrpt) - 1)

      self~assertSame(retDiscrpt, retState)
```

Here is the output:

```
C:\ooRexxUnit.3.2.0>rexx testOORexx.rex -R ooRexx -f stream
...
Interpreter: REXX-ooRexx_3.2.0(MT) 6.02 30 Oct 2007
ooRexxUnit:  2.0.0_3.2.0        ooTest: 1.0.0_3.2.0

Tests ran:           4
Assertions:          2
Failures:            0
Errors:              0
Skipped files:       0

File search:        00:00:00.160000
Suite construction: 00:00:00.000000
Test execution:     00:00:00.000000
Total time:         00:00:00.160000
```

To code the 2 tests, we really just used classic Rexx. The 'object' stuff is confined to some boilerplate code that you should be able to use without really understanding it. The process of just writing this, will allow the 'object' stuff to start seeping in.

Next installment: I am always threatening to write the documentation for the ooTest framework, but it is an *empty* threat. Or - what *are* the assertXXX methods?

1.4. Some Reference Documentation

This section will provide some basic documentation for the ooTest framework. In the spirit of the rest of this document, it is minimalistic. Just enough to write simple tests.

The best way to get more information if you are stuck is to join the ooRexx-Devel list and ask questions. Asking questions on that list will help in several ways. You'll get your question answered. Other people that have the same question will benefit from the answer. The question and answer become part of the permanent archive. And, hopefully, that information will migrate from the list into the ooTest documentation in a similar manner to the way the information in this document was migrated.

1.4.1. The assertXXX() Methods

The main thing that is left is to list the different assertXXX() methods. They are:

- **`assertEquals(expected, actual,[msg])`**
- **`assertNotEquals(expected, actual,[msg])`**
- **`assertNull(actual,[msg])`**
- **`assertNotNull(actual, [msg])`**
- **`assertSame(expected, actual,[msg])`**
- **`assertNotSame(expected, actual,[msg])`**
- **`assertTrue(actual,[msg])`**
- **`assertFalse(actual,[msg])`**

Some key points.

1.) All of these are methods, so to use them in your tests you do, for example:

```
self~assertTrue(.true)
```

If you are hazy on objects, just think of that as a routine with a mandatory prefix of 'self~'

2.) Where you have expected and actual args, expected always comes first.

```
val = 2 + 3
self~assertSame(5, val)
```

5 is expected, val is the actual.

3.) Where you just have the actual arg, it is the actual thing.

```

  val = .true
  self~assertTrue(val)
```

or

```

  val = 2 + 3
  self~assertTrue(val == 5)
```

4.) As the above shows, there are usually several ways to write the same assertion. Pick what you are comfortable with.

5.) All the assertions have an optional 'message' argument that is the last arg. The message is printed out if the assertion fails.

Here is an example:

```

  ::method test_addition
      val = 2 + 3
      self~assertSame(6, val, 'Special test, 2 plus 3 must equal 6')
```

and the output:

```

  [failure] [20080701 09:19:45.455000]
   Test:   TEST_ADDITION
   Class:  STREAM.testGroup
   File:   C:\work.ooRexx\ooRexxUnit\3.2.0\ooRexx\STREAM.testGroup
   Line:   82
   Failed: assertSame
     Expected: [[6], identityHash="535894080"]
     Actual:   [[5], identityHash="535906054"]
     Message:  Special test 2 plus 3 must equal 6
```

The intention of the message is to give some idea about what the test is testing. Here we see that the test writer has some special purpose in mind for this test.

Without the message and a non-descriptive test name we would see:

```
[failure] [20080701 09:22:45.367000]
 Test:   TEST_001
 Class:  STREAM.testGroup
 File:   C:\work.ooRexx\ooRexxUnit\3.2.0\ooRexx\STREAM.testGroup
 Line:   82
 Failed: assertSame
   Expected: [[6], identityHash="535894080"]
   Actual:   [[5], identityHash="535906054"]
```

which, if you were not the one that wrote the test, might leave you clueless. Then if you look at the test you would see:

```
::method test_001
    val = 2 + 3
    self~assertSame(6, val)
```

And you might really wonder what the test writer had in mind for this test in the STREAM test group. (Of course even with the message, we still don't know why 2 + 3 should equal 6.)

In general you don't need a message. But, it can help clarify what the test is about by using a message that states what is expected.

6.) The assertXXX() methods are pretty self explanatory, with maybe the exception of **`assertSame()`** and **`assertEquals()`**.

`assertSame()` (and **`assertNotSame()`**) assert that 2 things are stictly equal.

```
5 == 5
```

`assertEquals()` (and **`assertNotEquals()`**) assert that 2 things are loosely equal.

```
'dog ' = 'dog'
```

Equals is also used to test if 2 collections are loosely equal.

```
a1 = .array~of(1, 2, 3)
a2 = .array~of(1, 2, 3)
self~assertEquals(a1, a2)
```

`assertNull()` and **`assertNotNull()`** might also need some explaining. They test if the actual is **`.nil`** (or not **`.nil`**).

7.) The assertXXX() methods are located in the OOREXXUNIT.CLS file. Since this is open source you can always browse the file to see what assertXXX() methods are available to you. Which is what I do since I can never remember them and no one has produced easy to find documentation for the framework.

OOREXXUNIT.CLS is located in the framework subdirectory:

```
C:\ooRexxUnit.3.2.0>dir framework\OOREXXUNIT.CLS
 Volume in drive C has no label.
 Volume Serial Number is B4C0-DCBA

 Directory of C:\ooRexxUnit.3.2.0\framework
```

```
05/12/2008  01:15 PM            65,321 OOREXXUNIT.CLS

---------
```

Speaking of doc etc., in the misc subdirectory there are some sample test group files. Looking at those should be helpful. I believe I tried to comment them well.

```
C:\ooRexxUnit.3.2.0>dir misc
 Volume in drive C has no label.
 Volume Serial Number is B4C0-DCBA

 Directory of C:\ooRexxUnit.3.2.0\misc

01/18/2008  10:41 AM             5,755 SampleOLEObject.testGroup
01/16/2008  01:40 PM             5,638 Simplest.testGroup
01/16/2008  04:17 PM             5,775 SimpleWithOneTimeSetup.testGroup
01/16/2008  04:28 PM             5,609 SimpleWithSomeSetup.testGroup
```

There is also doc and more examples in the framework directory. However, those docs and examples are from the original ooRexxUnit framework.

The framework we are using here is the ooTest framework, which sits on top of ooRexxUnit. The ooRexxUnit doc does not necessarily apply to ooTest. And, I have made changes to ooRexxUnit, but have not been rigorous about maintaining the ooRexxUnit doc and examples.

1.5. How to Become a Committer

Armed with this quick start, I think that anyone who can program in Rexx and is using ooRexx to some degree can begin writing test cases for the ooRexx interpreter.

Ask questions when you get stuck. Contribute test cases to the project. Earn yourself the status of a committer.

About that last sentence. Want to have the exalted rank of a committer on an Open Source project? But you don't know C or C++. Contribute test cases to the ooRexx project and I guarantee I will push to make you a committer. And, Rick will vote in favor of making you a committer.

1.6. Negative Tests

In my original series of e-mail posts, Rick pointed out that I forgot to include how to do negative tests. This is really an important part of testing, and since I spent a lot of time in test, I'm kicking myself for overlooking it. **`<grin>`**

1.6.1. Example One

We'll continue with the **`STREAM.testGroup`** and look at what the doc says about the second arg. Which is exactly: "The second argument can be one of the following strings ..." and it the lists exactly 3 strings: State, Description, and Command.

So a negative test is what happens when you do something that is incorrect. In this case what happens if you don't follow a requirement. (Well there are other types of negative tests, but let's not get too complicated here.)

The requirement is that the second arg be one of only 3 different things. What we want to test, is what happens if we don't use of the 3 specified args. Say if we use FILESTATE ?

Staying with the same **`STREAM.testGroup`**. How do we test this? Maybe something along those lines?

```
::method test_wrong_2nd_arg
    parse source junk notUsed streamName

    retDiscrpt = stream(streamName, "FILESTATE")
```

Output:

```
[error] [20080701 12:06:53.206000]
 Test:  TEST_WRONG_2ND_ARG
 Class: STREAM.testGroup
 File:  C:\work.ooRexx\ooRexxUnit\3.2.0\ooRexx\STREAM.testGroup
 Event: [SYNTAX 40.904] raised unexpectedly.
   STREAM argument 2 must be one of SDC; found "FILESTATE"
   Line:    83
   83 *-* retDiscrpt = stream(streamName, "FILESTATE")
```

Now if you look closely at that, it was not a failure, it was an error.

If you remember back to what I said about terminology, I said with a test case you expect one of 2 outcomes. You either expect the test to pass, or you expect the test to fail.

Errors are something totally unexpected. We can see from the output that the interpreter is handling this the way it should. But, what we want is a test case for this that *passes*.

In this case we **expect** the error condition. So our test case will pass when we get an error. The framework provides for these types of test with the **`expectSyntax()`** method.

We code our test like this:

```
::method test_wrong_2nd_arg
    parse source junk notUsed streamName

    self~expectSyntax(40.904)
    retDiscrpt = stream(streamName, "FILESTATE")
```

And the output we get is:

```
C:\ooRexxUnit.3.2.0>rexx testOORexx.rex -R ooRexx -f stream
..
Interpreter: REXX-ooRexx_3.2.0(MT) 6.02 30 Oct 2007
ooRexxUnit:  2.0.0_3.2.0        ooTest: 1.0.0_3.2.0

Tests ran:           5
Assertions:          3
Failures:            0
Errors:              0
Skipped files:       0

File search:        00:00:00.160000
Suite construction: 00:00:00.000000
Test execution:     00:00:00.000000
Total time:         00:00:00.160000
```

Great, we now have 3 tests coded and they all pass.

1.6.2. Example Two

What else can we test in this way? We see from the doc that the first arg to stream is required. What happens if we leave it out? We should get some type of error. And the interpreter should not blow up.

Now how to code it? Similar to what we just did:

```
::method test_no_args

    --self~expectSyntax(40.904)
    retDiscrpt = stream()
```

But, what syntax error do we expect? The best thing to do, is to look up the error numbers and descriptions and decide for yourself what you think it *should* be.

The error message list is in the back of the doc, in an appendix. I don't want to drag this out, so I'm going to say, since the first error was in the 40. range, that is a good place to start. Looking there we see:

```
001
External routine "routine" failed
```

Okay, that seems reasonable. Though, there is also:

```
005
Missing argument in invocation of routine; argument argument_number is required
```

Aha, that is it. Argument 1 is required, says the doc. So the message: "Missing argument in invocation of STREAM; argument 1 is required" seems perfect.

Our test looks like:

```
::method test_no_args

    self~expectSyntax(40.005)
    retDiscrpt = stream()
```

and our output looks like this:

```
[error] [20080701 12:27:15.323000]
 Test:  TEST_NO_ARGS
 Class: STREAM.testGroup
 File:  C:\work.ooRexx\ooRexxUnit\3.2.0\ooRexx\STREAM.testGroup
 Event: [SYNTAX 40.3] raised unexpectedly.
   Not enough arguments in invocation of STREAM; minimum expected is 1
   Line:    90
   90 *-* retDiscrpt = stream()
```

Eureka! We found a bug! This is what it is all about, finding bugs. **`<grin>`** But, let's investigate a little. What is 40.3?

```
003
```

```
    Not enough arguments in invocation of routine; minimum expected is number
```

Making the message: Not enough arguments in STREAM; minimum expected is 1 Well, okay, that message also seems to fit. What to do? If you find something like this, that you think is a bug, then the thing to do is bring it up on the developer's list and lay out your case for why you think this is a bug.

In this instance, you would not prevail with your case. The reason being that 40.3 is applicable and it is what has been used in the past. Therefore, changing it because you think 40.5 is better has the potential of breaking existing code that is checking for this specific error code.

So, the test looks like this:

```
    ::method test_no_args

        self~expectSyntax(40.003)
        retDiscrpt = stream()
```

output:

```
    [error] [20080701 12:39:59.063000]
     Test:  TEST_NO_ARGS
     Class: STREAM.testGroup
     File:  C:\work.ooRexx\ooRexxUnit\3.2.0\ooRexx\STREAM.testGroup
     Event: [SYNTAX 40.3] raised unexpectedly.
       Not enough arguments in invocation of STREAM; minimum expected is 1
       Line:    90
       90 *-* retDiscrpt = stream()
```

Great!! Now we REALLY found a bug! It is supposed to be error 40.003. Well, again, do a little research. What the doc says is:

```
    Some errors have associated subcodes. A subcode is a one- to
    three-digit decimal extension to the error number, for example, 115 in
    40.115
```

A subcode is a *one* to three digit number.

Okay, no bug. Our test looks like this:

```
    ::method test_no_args

        self~expectSyntax(40.3)
        retDiscrpt = stream()
```

output:

```
    Interpreter: REXX-ooRexx_3.2.0(MT) 6.02 30 Oct 2007
    ooRexxUnit:  2.0.0_3.2.0        ooTest: 1.0.0_3.2.0

    Tests ran:           6
    Assertions:          4
    Failures:            0
    Errors:              0
    Skipped files:       0
```

```
     File search:         00:00:00.160000
     Suite construction: 00:00:00.000000
     Test execution:     00:00:00.000000
     Total time:         00:00:00.160000
```

Now we have 4 tests of an area that hadn't been touched. The 4 additional tests add 0 seconds to the overall test execution time. 2 of the tests are positive tests, 2 are negative.

Overview of ooTest

The primary purpose of ooTest is to throughly test the ooRexx interpreter and its distribution pacakages. In general, to throughly test software involves automating as much as possible of the testing. The ooTest framework provides the foundation for automatically executing tests. It currently contains little or no components that would aid in automating the writing of tests.

Note

This document is written for the 4.0.0 and later versions of ooTest. It makes no effort to cover the 3.2.0 verison of ooTest, or explain any differences between the 3.2.0 and 4.0.0 versions. There are differences.

ooTest also contains all the tests written to date to that the ooRexx team uses to test ooRexx. This makes ooTest itself more than just a framework. It is the sum of the tests themselves, the framework to write the tests, and any other components used in the over all process of testing the ooRexx package.

This document serves as a reference to the different components of ooTest. It explains how to execute the tests, intepret the results, and gives some guidence on writing new tests. There is very little involved in obtaining and settin up ooTest. Indeed there is very little set up involved in ooTest. It is merely a matter of placing the ooTest files on the system used to test.

2.1. Install ooTest

Getting and installing ooTest is extremely easy. There is no set up involved, you merely place the ooTest directory structure on somewhere convenient on your system. The best method of obtaining the ooTest directory structure is to use a Subversion client to check out the ooTest repository. Subversion is widely avaliable, easy to install, and easy to use. Directions for obtaining Subversion have been posted numerous times to the lists related to ooRexx. There is a wealth of information on this subject on the Internet, simply use Google.

Once you have a Subversion client, the following command will check out ooTest and place its directory structure in the subdirectory named **`ooTest.4.0.0`**. The subdirectory can be named anything you wish.

```
C:\>svn co https://oorexx.svn.sourceforge.net/svnroot/oorexx/test/trunk ooTest.4.0.0
```

The command is exactly the same on any operating system.

The other way to obtain ooTest is to download one of the ooTest snapshots from the ooRexx project on SourceForge. The drawback to this method is that the latest snapshot may not be current and you can not use the Subversion tools to keep your copy of ooTest up-to-date. To download a snapshot go to this link on SourceForge: *SourceForge.net: Open Object Rexx Files*[1] and download the appropriate file. Then either untar or unzip the file whereever is convenient. For example:

```
C:\>unzip ooTest-4.0.0-snapshot04_windows.i386.zip
```

[1] https://sourceforge.net/project/showfiles.php?group_id=119701&package_id=251951

Once the directory structure is placed on your system, you are ready to go. Use the driver program, testOORexx.rexx (*Chapter 3, testOORexx.rex the Automated Test Driver*) to execute some tests. The tests should run and print out some statistics. This will ensure ooTest is working. The rest of this book goes into detail on all aspects of ooTest.

2.2. History, Foundation

ooTest started out as ooRexxUnit. ooRexxUnit is a *generic* testing framework designed to serve as a unit testing framework for anyone to use to test their Rexx programs. It is patterned after jUnit and originally strove to emulate jUnit as closely as possible. As a publicly released piece of software it was constrained by typical needs, such as providing backward compatibility, limiting changes that require the user to make changes to their exisiting testing efforts, etc..

Over time it became apparent that the constraints of ooRexxUnit were a hinderance to the ooRexx testing effort. Therefore, the ooTest framework was put in place. ooTest sits on top of ooRexxUnit, but it is not designed to be a generic testing framework, does not guarentee backward compatibility, and does not strive to closely emulate jUnit. The idea behind this was to leave ooRexxUnit as a generic testing framework for any one to use, and focus the ooTest framework on specificaly testing ooRexx.

However, because of its history, ooTest is conceptualy similar to jUnit. If you understand jUnit concepts, you will have little trouble understanding ooTest. The main difference is that jUnit is primarily a unit testing framework, while ooTest is primarily an automation framework, that heavily supports unit testing.

From the perspective of writing test cases, ooTest is very jUnit-like. The terminology of ooTest is very similar to that of jUnit. ooTest uses test cases, test case classes, test suites, test results, test runners, and asserts, just as jUnit does. ooTest uses the same design patterns that jUnit uses. Hardly surprising as ooRexxUnit was modeled after jUnit. jUnit has generated many tutorial and primer types of documentation on the Internet. Google 'jUnit tutorial' or 'jUnit primer' and browse some of the simpler ones. This will quickly give you a grasp of ooTest test cases.

2.3. Testing Concepts Used in ooTest

ooTest is a way of writing defined repeatable tests, collecting and storing the tests, and executing all or some of those tests, now and in the future. With ooTest, all tests are to test the ooRexx interpreter and its distribution package. The purpose of a test is to demonstrate that some facet of the interpreter works, both now and at the next point in time the test is executed. To do this ooTest uses objects and the following loosely discusses these objects and how they fit together.

Test Case

The test case is the basic concept of ooTest. It is what the collection of defined repeatable tests is built upon.

Assertion

x.

testOORexx.rex the Automated Test Driver

The execution of all the tests in the test suite is currently driven by one Rexx program, **`testOORexx.rex`**. This program has an extensive set of options that allow the execution of all the tests, a single test, or subsets of the tests. As is all of ooTest, the program is platform independent and runs on all platforms where ooRexx is available.

When run, testOORexx sets up the environment, gathers up the tests to be executed, executes them, and prints out the statistics for the test run.

There is nothing preventing the use of other driver programs with the tests for the interpreter. It is merely that no one has written one. In particular, a GUI test driver would be nice and relatively easy to produce. One approach would be for the GUI driver to present the interface and then call testOORexx under the covers to do the actual work.

3.1. Using testOORexx.rex

testOORexx.rex is a command line program. It is designed to, and should be, run from a console window. Starting testOORexx.rex is the same as starting any Rexx program. On Windows, with a default install of ooRexx where **`.rex`** has a file association with the interpreter, you can simply type testOORexx. For example:

```
C:\ooTest>testOORexx
```

On an unix-like system, testOORexx would typically be started like this:

```
[miesfeld@falcon]# ./testOORexx.rex
```

Of course testOORexx can always be started this way:

```
C:\ooTest>rexx testOORexx.rex

[miesfeld@falcon]# rexx testOORexx.rex
```

Usually you would start testOORexx in the root directory of ooTest. I.e., in the directory that contains **`testOORexx.rex`**. However, testOORexx.rex is designed to run from any location on the local machine's file system. The main purpose of this is to allow a developer to execute the test suite from within their build directory. But, as the following example illustrates, testOORexx does not need to be run from the ooTest directory or a build directory, it can be run from anywhere on the system:

```

D:\TravelDrive>c:\ooTest\testOORexx.rex -F SimpleTests Searching for test containers..
Executing automated test suite..

ooTest Framework - Automated Test of the ooRexx Interpreter


Interpreter:     REXX-ooRexx_4.0.0(MT) 6.03 21 Jun 2009
Addressing Mode: 64
ooRexxUnit:      2.0.0_3.2.0    ooTest: 1.0.0_4.0.0

Tests ran:           9
```

```
Assertions:          9
Failures:            0
Errors:              0
Skipped files:       0

File search:        00:00:00.000000
Suite construction: 00:00:00.000000
Test execution:     00:00:00.000000
Total time:         00:00:01.000000


D:\TravelDrive>
```

Although running testOORexx will correctly set up the PATH to execute the tests, some people may prefer to change the path in the environment. This will allow running testOORexx from anywhere without having to type in the complete path name of testOORexx, such as **`C:\ooTest\testOORexx`**. In addition, on Windows testOORexx is able to completely set up the environment for executing all the tests, but on unix-like systems the library path can not be set. On the unix-like systems, the library path **must be** set prior to executing any test requiring one of the external binaries (*Section 3.9, "External Binaries"*). Two small utility scripts, **`setTestEnv.bat`** and **`setTestEnv.sh`** are provided to handle these chores. On Windows:

```
C:\ooTest>setTestEnv
```

On a unix-like system be sure to source the utility script:

```
[miesfeld@falcon]# . ./setTestEnv.sh
Setting env for Linux
[miesfeld@falcon]#
```

Another reason to use the utility scripts is that this allows a TODO test group ADD LINK to be run as a stand alone Rexx program from its current directory. This is sometimes useful when first developing a set of test cases in a test group. For example:

```
C:\ooTest\ooRexx\base\class>rexx Array.testGroup

Interpreter:     REXX-ooRexx_4.0.0(MT) 6.03 21 Jun 2009
Addressing Mode: 64
ooRexxUnit:      2.0.0_3.2.0    ooTest: 1.0.0_4.0.0

Tests ran:           101
Assertions:          8007
Failures:            0
Errors:              0
Skipped files:       0

Test execution:     00:00:00.469000


C:\ooTest\ooRexx\base\class>
```

Executing a test group as a stand alone program is not possible unless the ooTest root directory is in the path because the interpreter will not be able to find the **`ooTest.frm`** file:

```
C:\ooTest\ooRexx\base\class>rexx Array.testGroup
    54 *-* ::requires 'ooTest.frm' -- load the ooRexxUnit classes
Error 43 running C:\ooTest\ooRexx\base\class\Array.testGroup line 54:  Routine not found
Error 43.901:  Could not find routine "ooTest.frm" for ::REQUIRES
```

```
C:\ooTest\ooRexx\base\class>
```

Note

As a reminder, on unix-like systems, sourcing **`setTestEnv.sh`** is a requirement when tests using the TODO external binaries ADD LINK will be executed. Unless of course the library path is manually set by some other means.

3.2. Modes of Operation

To Be Written

3.3. Understanding the Output

To Be Written

3.4. Static Options

The term *static* is used here for lack of a better one. These are the *known* or *implemented* options as opposed to the dynamic (*Section 3.5, “Dynamic Options”*) options formed by using the **`-D`** option.

Each of the known options has a short form and a long name form. The long name for the option is used when specifying the option in the options (*Section 3.6, “The Options File”*). The long name can also be used on the command line in combination with the -D (*Section 3.4.5, “The -D "define" option”*) option if it is easier to remember the long name.

The following sections discuss each of the known options individually.

3.4.1. The -a "allTestTypes" option

This option is a boolean option, specified by true or false. The default is false.

3.4.2. The -b "buildFirst" option

To Be Written

3.4.3. The -B "forceBuild" option

To Be Written

3.4.4. The -d "defaultTestTypes" option

To Be Written

3.4.5. The -D "define" option

To Be Written

3.4.6. The -e "testContainerExt" option

To Be Written

3.4.7. The -f "singleFile" option

To Be Written

3.4.8. The -F "fileList" option

To Be Written

3.4.9. The -I "testTypeIncludes" option

To Be Written

3.5. Dynamic Options

Options need not be known to the **`testOORexx`** program. Options can be specified dynamically by defining the name of the option and its value on the command line, or in the options file (*Section 3.6, "The Options File"*). These options are handled generically by testOORexx.

3.6. The Options File

To Be Written

3.7. The .testOpts Directory

To Be Written

3.8. Some Debug Help

To Be Written

3.9. External Binaries

To Be Written

The ooRexxUnit Framework

x

4.1. TestCase Class

A *TestCase* class

The following table summarizes the important information for a TestCase class, including all methods and attributes relevant to writing test cases. Most of these methods are inherited from *TestCase*:

Table 4.1. ooTestCase Summary

Item	...Description
Something	some link

4.1.1. ooTestType (Class)

```
>>--ooTestType--------------------------------------------><

>>--ooTestType=-------------------------------------------><
```

To Be Written

4.1.2. new (Class)

```
>>--new---------------------------------------------------><
```

To Be Written

4.2. Assert Class

An *Assert* class

The following table summarizes the important information for an Assert class, including all methods and attributes relevant to writing test cases. Most of these methods are inherited from *TestCase*:

Table 4.2. ooTestCase Summary

Item	...Description
Something	some link

The ooTest Framework

The ooTest framework requires the ooRexxUnit framework (*Chapter 4, The ooRexxUnit Framework*), that is it uses a ::requires directive for **`OOREXXUNIT.CLS`**. It extends the functionality of ooRexxUnit in ways that are not neccessarily intended to be generic, but rather are only intended to serve some specific need in testing ooRexx. Most of the main classes in ooTest are direct subclasses of a comparable class in ooRexxUnit, adding only minor functionality to the ooRexxUnit class.

This section of the book serves as a reference to the public classes, routines, and features of the ooTest framework. It starts with a general discussion of where and how the framework differs from a generic unit testing framework and the objects in the framework that implement these differences. The rest of the material is the reference documentation for the classes and routines that the framework provides.

The reference material is intended to be most useful to people writing test cases for ooRexx. Please **note** how this is structured. Many of the classes and methods of the framework are used to relieve the test case writer of much, or most, of the burden of implementation. The test case writer, for the most part, only needs the documentation for a few classes and usually only for a few methods of those classes. The rest of the classes and methods need only be understood by the maintainers of the overall ooRexx testing project.

However, it makes little sense to document part of a class here, the part used by test case writers, and the rest of the class in some other place, say a *maintainers* manual. So this strategy is used: the primary classes a test case writer would use are documented first. Within a class documentation, the methods and attributes most useful in writing test cases are also documented first. The rest of the classes and methods will be marked with the keyword: *internal*. This doesn't mean these classes or methods are only for internal use. It is merely meant to indicate to the reader, who is mostly interested in writing test cases, that she can probably ignore the classes and methods.

5.1. General Concepts

Description

5.1.1. Test Types

Test types are implemented through the ooTestTypes (*Section 5.3, “ooTestTypes Class”*) and ooTestCase (*Section 5.2, “ooTestCase Class”*) classes.

Unlike jUnit, which only does unit tests, ooTest is designed to automate all the testing for the ooRexx distribution package. There is of course more to software testing than unit tests. There are stress tests, performance tests, acceptance tests (does the distribution package install correctly, do all the samples at least run,) GUI tests, etc..

The value of an automated unit testing framework comes from developers running the unit tests often. It has been suggested that preferrably the unit tests should be run after every compile. But, if ooTest is going to be the framework used to run stress tests, acceptance tests, etc., the time to execute the test suite starts growing exponentially. In addition stress or performance tests typically use large amounts of system resources. If the developer's machine doesn't have the required horse power for a stress test, what then? If an ooRexx user wants to help by writing and executing test cases, but doesn't have access to a smtp server needed to run the socket test cases, how does he proceed?

Clearly, if ooTest is to serve as an all-purpose test framework, there needs to be some way of easily running subsets of the tests that are applicable at the time. The developer needs to be able to easily run only the unit tests. The user trying to contribute to the project, but is lacking a smtp server, needs some way to run the test suite, but exclude the socket tests.

ooTest handles this by using the concept of *test types*. In jUnit there is only one type of test, the unit test. In ooTest, each group of test cases in an ooTestCase (*Section 5.2, “ooTestCase Class”*) subclass belongs to one of any number of defined test types. The ooTestCase class has a class attribute that defines the test type the class contains. By default the test cases in an ooTestCase subclass are the unit test type.

When writing a ooTestCase subclass, the test type for the subclass can be over-ridden. This is done for example in the *Native_api* tests. The class init() method is over-ridden as in this example:

```
::class "CONVERSION.testGroup" subclass ooTestCase public

::method init class
  forward class (super) continue

  -- Over-ride the default test type
  self~ooTestType = .ooTestTypes~NATIVE_API_TEST
```

5.2. ooTestCase Class

An *ooTestCase* class is a test case class where methods of the class define individual test cases.

In order to make it easy to construct automated tests with large numbers of test cases, a convention is followed: Each method of an ooTestCase class that starts with *test* is considered an individual test case.

Each ooTestCase has a class attribute defining the test type (*Section 5.1.1, “Test Types”*) of the individual test cases the class contains. All test cases in a specific ooTestCase class are the same test type. The class *new()* method is intended to be over-ridden by test case writers to assign an appropriate test type when needed.

The following table summarizes the important information for an ooTestCase class, including all methods and attributes relevant to writing test cases. Most of these methods are inherited from *TestCase*:

Table 5.1. ooTestCase Summary

Item	...Description
Subclasses	TestCase (*Section 4.1, “TestCase Class”*)
Inherits	ooTestTypes (*Section 5.3, “ooTestTypes Class”*)
Inherits (indirectly through TestCase)	Assert (*Section 4.2, “Assert Class”*)
Class Attributes	
ooTestType (Class Attribute)	ooTestType (*Section 5.2.1, “ooTestType (Class)”*)
Class Methods	
new (Class method)	new (*Section 5.2.2, “new (Class)”*)

5.2.1. ooTestType (Class)

```
>>--ooTestType---------------------------------------------><
```

```
>>--ooTestType=--------------------------------------------><
```

To Be Written

5.2.2. new (Class)

```
>>--new----------------------------------------------------><
```

To Be Written

5.3. ooTestTypes Class

An *ooTestTypes* class

The following table summarizes the important information for an ooTestCase class, including all methods and attributes relevant to writing test cases. Most of these methods are inherited from *TestCase*:

Table 5.2. ooTestCase Summary

Item	...Description
Something	some link

5.3.1. ooTestType (Class)

```
>>--ooTestType---------------------------------------------><

>>--ooTestType=--------------------------------------------><
```

To Be Written

5.3.2. new (Class)

```
>>--new----------------------------------------------------><
```

To Be Written

Writing Tests

No content yet.

Appendix A. Notices

A.1. Trademarks

A.2. Source Code For This Document

The source code for this document is available under the terms of the Common Public License v1.0 which accompanies this distribution and is available in the appendix *Appendix B, Common Public License Version 1.0*. The source code is available at *https://sourceforge.net/p/oorexx/code-0/HEAD/tree/docs/*.

The source code for this document is maintained in DocBook SGML/XML format.

The railroad diagrams were generated with the help of "Railroad Diagram Generator" located at *http://bottlecaps.de/rr/ui*. Special thanks to Gunther Rademacher for creating and maintaining this tool.

Appendix B. Common Public License Version 1.0

THE ACCOMPANYING PROGRAM IS PROVIDED UNDER THE TERMS OF THIS COMMON PUBLIC LICENSE ("AGREEMENT"). ANY USE, REPRODUCTION OR DISTRIBUTION OF THE PROGRAM CONSTITUTES RECIPIENT'S ACCEPTANCE OF THIS AGREEMENT.

B.1. Definitions

"Contribution" means:

1. in the case of the initial Contributor, the initial code and documentation distributed under this Agreement, and
2. in the case of each subsequent Contributor:
   a. changes to the Program, and
   b. additions to the Program;

where such changes and/or additions to the Program originate from and are distributed by that particular Contributor. A Contribution 'originates' from a Contributor if it was added to the Program by such Contributor itself or anyone acting on such Contributor's behalf. Contributions do not include additions to the Program which: (i) are separate modules of software distributed in conjunction with the Program under their own license agreement, and (ii) are not derivative works of the Program.

"Contributor" means any person or entity that distributes the Program.

"Licensed Patents " mean patent claims licensable by a Contributor which are necessarily infringed by the use or sale of its Contribution alone or when combined with the Program.

"Program" means the Contributions distributed in accordance with this Agreement.

"Recipient" means anyone who receives the Program under this Agreement, including all Contributors.

B.2. Grant of Rights

1. Subject to the terms of this Agreement, each Contributor hereby grants Recipient a non-exclusive, worldwide, royalty-free copyright license to reproduce, prepare derivative works of, publicly display, publicly perform, distribute and sublicense the Contribution of such Contributor, if any, and such derivative works, in source code and object code form.
2. Subject to the terms of this Agreement, each Contributor hereby grants Recipient a non-exclusive, worldwide, royalty-free patent license under Licensed Patents to make, use, sell, offer to sell, import and otherwise transfer the Contribution of such Contributor, if any, in source code and object code form. This patent license shall apply to the combination of the Contribution and the Program if, at the time the Contribution is added by the Contributor, such addition of the Contribution causes such combination to be covered by the Licensed Patents. The patent license shall not apply to any other combinations which include the Contribution. No hardware per se is licensed hereunder.
3. Recipient understands that although each Contributor grants the licenses to its Contributions set forth herein, no assurances are provided by any Contributor that the Program does not infringe the patent or other intellectual property rights of any other entity. Each Contributor disclaims any liability to Recipient for claims brought by any other entity based on infringement

of intellectual property rights or otherwise. As a condition to exercising the rights and licenses granted hereunder, each Recipient hereby assumes sole responsibility to secure any other intellectual property rights needed, if any. For example, if a third party patent license is required to allow Recipient to distribute the Program, it is Recipient's responsibility to acquire that license before distributing the Program.

4. Each Contributor represents that to its knowledge it has sufficient copyright rights in its Contribution, if any, to grant the copyright license set forth in this Agreement.

B.3. Requirements

A Contributor may choose to distribute the Program in object code form under its own license agreement, provided that:

1. it complies with the terms and conditions of this Agreement; and
2. its license agreement:
    a. effectively disclaims on behalf of all Contributors all warranties and conditions, express and implied, including warranties or conditions of title and non-infringement, and implied warranties or conditions of merchantability and fitness for a particular purpose;
    b. effectively excludes on behalf of all Contributors all liability for damages, including direct, indirect, special, incidental and consequential damages, such as lost profits;
    c. states that any provisions which differ from this Agreement are offered by that Contributor alone and not by any other party; and
    d. states that source code for the Program is available from such Contributor, and informs licensees how to obtain it in a reasonable manner on or through a medium customarily used for software exchange.

When the Program is made available in source code form:

1. it must be made available under this Agreement; and
2. a copy of this Agreement must be included with each copy of the Program.

Contributors may not remove or alter any copyright notices contained within the Program.

Each Contributor must identify itself as the originator of its Contribution, if any, in a manner that reasonably allows subsequent Recipients to identify the originator of the Contribution.

B.4. Commercial Distribution

Commercial distributors of software may accept certain responsibilities with respect to end users, business partners and the like. While this license is intended to facilitate the commercial use of the Program, the Contributor who includes the Program in a commercial product offering should do so in a manner which does not create potential liability for other Contributors. Therefore, if a Contributor includes the Program in a commercial product offering, such Contributor ("Commercial Contributor") hereby agrees to defend and indemnify every other Contributor ("Indemnified Contributor") against any losses, damages and costs (collectively "Losses") arising from claims, lawsuits and other legal actions brought by a third party against the Indemnified Contributor to the extent caused by the acts or omissions of such Commercial Contributor in connection with its distribution of the Program in a commercial product offering. The obligations in this section do not apply to any claims or Losses relating to any actual or alleged intellectual property infringement. In order to qualify, an Indemnified

Contributor must: a) promptly notify the Commercial Contributor in writing of such claim, and b) allow the Commercial Contributor to control, and cooperate with the Commercial Contributor in, the defense and any related settlement negotiations. The Indemnified Contributor may participate in any such claim at its own expense.

For example, a Contributor might include the Program in a commercial product offering, Product X. That Contributor is then a Commercial Contributor. If that Commercial Contributor then makes performance claims, or offers warranties related to Product X, those performance claims and warranties are such Commercial Contributor's responsibility alone. Under this section, the Commercial Contributor would have to defend claims against the other Contributors related to those performance claims and warranties, and if a court requires any other Contributor to pay any damages as a result, the Commercial Contributor must pay those damages.

B.5. No Warranty

EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, THE PROGRAM IS PROVIDED ON AN "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, EITHER EXPRESS OR IMPLIED INCLUDING, WITHOUT LIMITATION, ANY WARRANTIES OR CONDITIONS OF TITLE, NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Each Recipient is solely responsible for determining the appropriateness of using and distributing the Program and assumes all risks associated with its exercise of rights under this Agreement, including but not limited to the risks and costs of program errors, compliance with applicable laws, damage to or loss of data, programs or equipment, and unavailability or interruption of operations.

B.6. Disclaimer of Liability

EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, NEITHER RECIPIENT NOR ANY CONTRIBUTORS SHALL HAVE ANY LIABILITY FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING WITHOUT LIMITATION LOST PROFITS), HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OR DISTRIBUTION OF THE PROGRAM OR THE EXERCISE OF ANY RIGHTS GRANTED HEREUNDER, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

B.7. General

If any provision of this Agreement is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this Agreement, and without further action by the parties hereto, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

If Recipient institutes patent litigation against a Contributor with respect to a patent applicable to software (including a cross-claim or counterclaim in a lawsuit), then any patent licenses granted by that Contributor to such Recipient under this Agreement shall terminate as of the date such litigation is filed. In addition, if Recipient institutes patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Program itself (excluding combinations of the Program with other software or hardware) infringes such Recipient's patent(s), then such Recipient's rights granted under Section 2(b) shall terminate as of the date such litigation is filed.

All Recipient's rights under this Agreement shall terminate if it fails to comply with any of the material terms or conditions of this Agreement and does not cure such failure in a reasonable period of time after becoming aware of such noncompliance. If all Recipient's rights under this Agreement terminate, Recipient agrees to cease use and distribution of the Program as soon as reasonably practicable.

However, Recipient's obligations under this Agreement and any licenses granted by Recipient relating to the Program shall continue and survive.

Everyone is permitted to copy and distribute copies of this Agreement, but in order to avoid inconsistency the Agreement is copyrighted and may only be modified in the following manner. The Agreement Steward reserves the right to publish new versions (including revisions) of this Agreement from time to time. No one other than the Agreement Steward has the right to modify this Agreement. IBM is the initial Agreement Steward. IBM may assign the responsibility to serve as the Agreement Steward to a suitable separate entity. Each new version of the Agreement will be given a distinguishing version number. The Program (including Contributions) may always be distributed subject to the version of the Agreement under which it was received. In addition, after a new version of the Agreement is published, Contributor may elect to distribute the Program (including its Contributions) under the new version. Except as expressly stated in Sections 2(a) and 2(b) above, Recipient receives no rights or licenses to the intellectual property of any Contributor under this Agreement, whether expressly, by implication, estoppel or otherwise. All rights in the Program not expressly granted under this Agreement are reserved.

This Agreement is governed by the laws of the State of New York and the intellectual property laws of the United States of America. No party to this Agreement will bring a legal action under this Agreement more than one year after the cause of action arose. Each party waives its rights to a jury trial in any resulting litigation.

Appendix C. Revision History

Revision 0-0	**Tue Aug 7 2012**	**David Ashley**

Initial creation of book by publican

Index