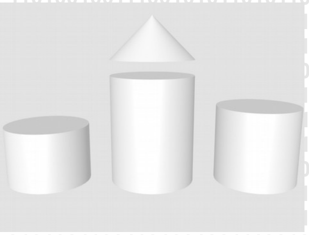


ODABA^{NG}

Test Framework

Summary

In order to manage different kind of tests, ODABA provides a command line test framework combined with a **TestBrowser**. **TestBrowser** is a GUI tool based on a data stored in the file system in order to run an independent external test frame work.

The interface to the test frame work is a work area directory containing test data and test procedures. Test data and test procedures are copies from test suites and test cases represented as directories in the file system. **TestBrowser** does not care about content (test case description, test data, procedures etc.) but provides fast access to test resources, prepares test runs and registers results for creating test logs and protocols.

For executing tests, any test framework with a command shell interface may be used. The **TestBrowser** delivery comes up with a simple command shell based test framework, that might be used for unit tests as well as for component or system tests.

run Software-Werkstatt GmbH
Weigandufer 45
12059 Berlin

Tel: +49 (30) 609 853 44
e-mail: run@run-software.com
web: www.run-software.com

Berlin, October 2015

-

Contents

1 Rules and principles for test frameworks

Although, testing is not only related to IT projects, this section mainly refers to software tests, especially considering an approach to automated tests.

The first topic explains basic concepts and common rules for test frameworks. In the second topic, an example based on command shells (cmd, sh, bash etc.) illustrates, how to provide a simple test framework by means of shell procedures. The last topic describes some use cases and how to modify the example in order to achieve more appropriate results.

The concepts and examples mainly are related to black box tests, which are typically used for component or system tests. The ODABA API test suite is also demonstrates the use of **TestBrowser** for running unit tests together with component and system tests.

-

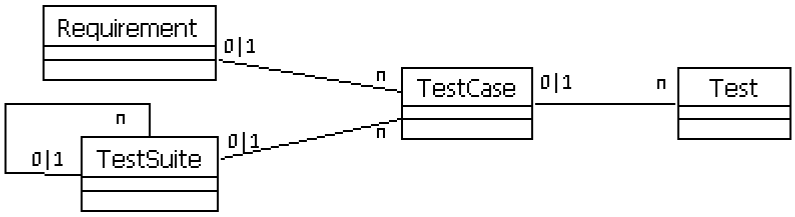
1.1 Principles for arranging tests in test frameworks

The **test framework** is an appropriate environment, which allows running requested test scenarios. The test framework controls test runs including any number of test suites and manages test resources for running tests. Ideally, the test framework also communicates with the test management in order to return results for executed tests or test runs.

A simple test framework is the command shell (bash, cmd or other). Depending on implementation language and type (GUI or command line components) one may also refer to more advanced test frameworks (as Abbot, OpenCTF, Fitnesse etc.). Common principles can be identified in different frameworks. Because of better transparency, command shell procedures (**bash**, **cmd**) are used for providing essential test framework functions.

The task of a test framework is to perform tests conceptually described in test cases. Usually, test cases for component tests describe the way to test the component against specifications in requirements. In case of requirement driven tests, for each requirement any number of test cases may be defined. A **test case** defines a specific test aspect (for a requirement), e.g. borderline checks for a value or parameter. In order to completely test a test case, several tests may be necessary. A **test** provides rules (actions) and test data for running the test and expected test results.

Similar test cases are collected in test suites. Tests suites may form hierarchies by defining upper test suites containing a number of similar subordinated test suites. In order to execute automated tests, test suites are provided in a test framework. A **test suite** is the environment that provides necessary resources for running one or more test cases.



Test suites may be arranged in hierarchies, where each test suite includes all subordinated test suites. **TestBrowser** as well as the test frame work expect one and only one topmost test suite, which is called **main suite**.

A **test run** is the execution of a number of tests defined in one or more test suites. Within a test run any number of tests from different test suites may be collected. Typically, a test run, contains all tests for a test suite and its subordinated test suites.

For supporting statistics on test runs and for comparing test runs, results of each test run need to be stored. In the example, test run results are stored in a directory structure *test run identifier/test identifier*, but any other systematic including test run and test identifiers might be used. .

In order to support automated tests, naming conventions will help a lot. Usually, it does not matter, which naming conventions are agreed upon, as long as names can be filtered by means of regular expressions (e.g. having a specific prefix).

1.1.1 Test suite and test case

A test suite covers different kind of resources to run tests and provide results in a proper way. Each test case belongs to a test suite that provides common resources for test cases described below the test suite. Usually, a test suite covers one or more test cases. Hence, test cases may extend or overwrite test suite resources by adding additional resources or replacing existing resources. Test suite and test case provide the following kinds of resources:

- Tests - Implementation of one or more tests to be executed within the test suite
- Test set - Input data required for running the test
- Actions - Actions provided for running the test
- Expected output - Expected output for the test

For each test run following resources are typically created:

- Test output - Output data created by the test run
- Reports - log and result files

Within a test suite, a number of test cases, which refer to the test set of the test suite, may be executed by calling appropriate actions. A test defines the sequence of action calls. Each test run creates test output. Test output will be compared with expected output and the result of comparison is written to a result file.

Ideally, a separate location (directory) is defined for each element type. Depending on kind of processing, other elements may be added to a test suite.

In order to mark a directory as suite, a file **suite** has to be created in the directory.

Test

A **test** is described as a sequence of actions to be executed. The execution of a test takes place within a test run for the test or a test suite. In the ODABA test framwork tests are defined as **run** actions for a test case or a test suite.

Each test execution within a test run returns either **true** or **false**. The values may be named differently (e.g. success or failed as in the example), but running a test always returns one of these result values. When a test has failed (**false**), it means that it did not match the expected test output. In case of testing an error situation, expected output may contain an error message or code and the test returns **true**, when the expected error has been produced. In order to provide more information for false tests, one may add additional information.

The result of each test is as to a result data for the test/test run. In addition, each test should provide a protocol (log file) when being executed (e.g. start and stop time for the test).

Test set

A **test set** provides the data for running all test cases defined in a test suite. Test data defined for the test suite must not be complete, i.e. it may be extended or replaced by each single test case. Test sets are mainly a mean for reducing the amount of test data, i.e. reducing costs for test data maintenance. Local test set for a test case or test suite is defined in the **data** directory.

The final test set for a test results is a combination of test sets of all test suites in the test suite hierarchy, while files in test sets on lower level in the hierarchy overwrite files in higher test suite levels.

Before running a test, test resources are copied to a test work area, where the test will be executed.

Actions within a test suite

In order to run tests within a test suite, several actions may be defined. It does not matter, how actions are implemented, but it should be as less and as simple as possible. There is a typical scenario, which is referred to in the example and which is quite sufficient for many tests. This scenario consists of following actions:

- `preprocessing` - Run special (sub)actions for preparing test case data.
- `run` - Run the required test functions.
- `postprocessing` - Run special actions after running the test. Typically, this action compares test results with expected test results.

Actions are implemented in order to be executed within a single test. Actions do not manage test runs for one or more test cases. For this purpose, the test frame work provides appropriate techniques (command line procedures). The example provides a number of procedures (bash or cmd files) for managing test suites under Linux and MS Windows.

Since actions defined within the test suite may be overloaded, those are called **dynamic action**s. Actions or procedures for performing test framework tasks are called **global action**s.

Expected output

Expected output describes the expected result of a test being executed. This may be the value returned from a program or one or more output files created by the test. Typically, expected data will be compared with output data produced by running test. When the output and expected data are the same, the test returns **true**, otherwise **false**. Locally defined expected output is defined in the **expected** directory.

Often, this is not as simple, since comparing data may include removing disturbing parts in the test output (e.g. creation time stamp for an output file). This does not change principles used for comparing test output, but evaluation technologies, only.

Similar to test sets (test data), expected data is inherited from the test suite hierarchy, i.e. common expected test data may be provided on higher level test suites.

Test output

The location for storing test output in a common sense depends on the component or unit to be tested. Test output may be a return code, but also a collection of files or any other kind of electronic readable output. As soon as test output is not readable (e.g. a test may produce a beep), it becomes more difficult running automatic tests.

When running tests in a work area, output will be created in the work area. In order to save more detailed information for a test run, relevant part of output may be copied to a location identified by test run and test or test suite identifier.

Results

Results are created by test runs and are stored on test run/test level. Usually, there is a result file containing the result (**true** or **false**) for each test executed. Moreover, a summary file may be provided for the test run containing e.g. start and stop time for each test, duration or other relevant information.

Result and log file content refers to common test run information not depending on component or component properties.

1.1.1.1 Hierarchical test suites

When considering several hundred or thousand test cases, maintaining corresponding resources may become a problem. Test case resources may be reduced by arranging test suites similar to requirements and test cases in hierarchies. This also allows introducing advanced testing features as test suite inheritance.

E.g. a test suite defined for value domain check that refers to 5 subordinates test cases (defining tests for lower than or equal to minimum value, between minimum and maximum and equal to or greater than maximum) may share its data with another value domain check by using the same parent test suite.

Finally, how test suites are structured, depends on practical requirements. In general, test suites are required when resources for tests differ, i.e. within a test suite, all tests refer to the same test set and same set of expected results, but may be called in different ways.

Hierarchical test suites provide the advantage, that resources may be inherited from test suite parents, which allows defining reusable test resources.

- main_suite - providing default resources
- test suite 1
  - test suite 11
  - ...
- ...
- test suite n

Since running a test suite includes running all test cases and related tests defined in subordinated test suites, too, ordering test suites in a hierarchy provides an additional way of filtering a set of test suites to be executed. Moreover, hierarchical test suites allow referring to advanced test run features as resource inheritance and action overloading.

Using resource inheritance and action overloading features, borderlines between test suites as well as between test suite and test case become blurred. Nodes in the hierarchy may be interpreted as test suite, but also as test case.

1.1.2 Running a test

In order to run a test in a test framework, several typical steps are required:

- Preparing test environment - This includes providing global settings for the test run and preparing the test work area by collecting necessary test resources.
- Running the test - This may be simply a function call but also a more complex process.
- Evaluating test output: This typically includes comparing test output with expected output and creating test results for the test.

In the example `testRun` performs the necessary steps for a single test by calling:

- `settings` - provide common environment variables for running tests (**settings.cmd**)
- `prepare` - prepare test work area for execution (**prepare.cmd**)
- `preprocessing` - execute test specific preprocessing actions (**preprocessing.cmd**)
- `run` - run the test (**run.cmd**)
- `postprocessing` - execute test specific post processing, e.g. comparing output (**postprocessing.cmd**)
- `report` - create test run report information **(report.cmd**)

This demonstrates, how one may manage automated tests with just a few lines in a command shell.

Since the work area will be removed after testing or replaced by data for the next test run, detailed information about last test run will get lost. In order to keep test run output, the work area data may be copied to a test run/test area (e.g. by zipping the complete work area directory). In any case, test output and results should be stored separately, i.e. in a separate test run directory or in appropriate test run directories below each test suite. It might also be sufficient to copying output data differing from expected results to a test run/test directory.

1.1.3 Managing tests within a test run

Managing tests within a test run includes the following basic features:

- Selecting tests to be executed
- Running selected tests
- Providing test run summary reports

For a given test suite hierarchy two simple ways of selecting tests can be used. By selecting a single test suite for test run, tests for the test suite and all subsequent test suites are executed. The other way is manually selecting or deselecting tests in the test suite hierarchy. In the example, tests are indicated by a file with the name **test**. Removing or renaming this file will exclude a test suite from being tested. Nevertheless, there are many other ways of filtering tests for run as filtering by name.

Beside managing test runs, the test framework has to manage test run resources. This includes providing test run locations for storing results from tests within the test run. Running tests within a test run should also provide a summary of test results (**true**/**false**). Formatting the output property (e.g. as csv-file), usually allows importing test results into a connected test management system (ODABA Teat Browser or Project Manager, Polarion or other).

While running tests, also test events (e.g. start/stop time) may be written to a test run protocol. As other test run resources, this should be stored in general in separate test run location for the test suite. In order to be able to store test run summaries, each test run should provide its own test run location (e.g. directory), which contains test results and test output for executed tests.

-

1.1.4 Advanced concepts

There are several ways for improving maintenance of test suites. Because of the large amount of test data required for testing a component, techniques are required, that allow reducing the amount of data, especially avoiding unnecessary duplicates.

Following advanced features are considered in the next topics:

- Test suite patterns
- Test suite templates
- Test suite inheritance
- Test suite versioning
- Interface to test management

Here, advanced concepts are not discussed in detail, but some useful solutions are shortly illustrated.

1.1.4.1 Test suite patterns

Test suite patterns provide typical test suite definitions. Test suite patterns are not executed directly, but copied to become a specific test suite after updating special parameters. The advantage for test suite pattern is, that those provide common rules for test suites in a specific environment.

A test suite pattern defines the typical resources for a test suite. Providing a number of test suite patterns supports standardizing test processes. An alternative to test suite patterns ate test suite templates, which support automatic test suite generation.

A test suite pattern may be copied and adapted to become a part of a test suite. In case of test suite hierarchies, one may also provide a complete pattern hierarchy in order to define kind of nested patterns.

Considering the value domain problem (including tests for lower than or equal to minimum, between minimum and maximum and equal to or greater than maximum), one may define a two level test suite:

- pattern_suite - providing common resources
- test suite 1 (value < minimum)
- test suite 2 (value = minimum)
- test suite 3 (minimum < value < maximum)
- test suite 4 (value = maximum)
- test suite 5 (value > maximum)

Such a pattern makes sense, when all 5 test cases require different input data. One might, however, also define a test suite with five tests below, when the test sets for the test do not differ (e.g. testing parameters passed to a program)

- pattern_suite - providing test suite resources
- test 1 (value < minimum)
- test 2 (value = minimum)
- test 3 (minimum < value < maximum)
- test 4 (value = maximum)
- test 5 (value > maximum)

1.1.4.2 Test suite templates

Test suite templates are another feature for providing generic test suites. A typical case for defining test suite templates are value domain tests. When testing value ranges for maximum and minimum value, five test cases result from the requirement (lower than and equal to minimum, between minimum and maximum, equal to and greater than maximum). Normally, one will not define all those test cases, but just say "test p1 to be equal to or greater than 0 and equal to or lower than 10". Passing p1, 0 and 20 to a value domain test suite template allows generating the five required tests or test suites (e.g. as subordinated test suites) and running those afterward.

One solution for this approach is illustrated within the example framework by providing a specific `preprocessing` action, which generates and fills the 5 required subordinated test suites, which will be called automatically after finishing generation.

It depends on specific test strategies, whether test templates are executed once or for each test run. When generating the test suite once, test run conditions will not change when rerunning the test. On the other hand, template improvements will apply on a test run, only, when generating test suites all the time a test run is executed.

1.1.4.3 Test suite inheritance

Arranging test suites in a hierarchical order allows inheriting test suite resources along the hierarchy. In this case, the work area will be prepared by copying resources from different level in a parent test suite hierarchy. During this process, lower level resources always overwrite higher level resources.

The advantage using test suite inheritance is, that common resources for several test suites may be defined once on a higher level in the test suite hierarchy. Hierarchical test suites not only allow replacing data, but also test expectations and actions. Thus, default actions defined for the main suite may be overloaded by actions defined in subordinated test suites. Actions provided on lower levels will always overwrite actions with the same name defined on higher levels in the test suite hierarchy.

Test suite inheritance is a feature, which allows reducing test resources extremely by sharing those between several test suites. Thus, the effort for test suite maintenance can be reduced. On the other hand, test suite inheritance includes additional risks, since it is not obvious, which resources are finally used within a test run. In order to be able to review test runs, one may store the test work area before (and/or after) running the test.

The example below shows the work area resulting from test suite hierarchy. In general, the main suite should contain as much resources as possible, but should not contain any resource, which may be omitted in any of the subsequent test runs. Nevertheless, in such cases the `preprocessing` action may solve the problem by removing unnecessary files.

1.1.4.4 Test suite version control

When running several versions of a product, test suites versions have to be controlled. Whatever framework is used for testing, it has to support version control in any way. This becomes simple when using a command shell framework, since one may check in the complete test root or main suite into any kind of version control system system (GIT, SVN, ...). Advanced test frameworks provide their own version control.

1.1.4.5 Interface to test management

While the test management manages test cases (and requirements) on a conceptual level, the test framework allows running required tests. Nevertheless, it is sometimes desirable to start test runs from test management and import test results and progress information.

The way of solving this problem depends on the test management system. The ODABA **TestBrowser** supports starting test runs by configuring the test frame work and generating framework specific information for the test suites to be executed. The connection between test cases and test suites may be provided by test suite identifiers referenced in the test case or by any other algorithm deriving a test suite identifier from the test case definition. Thus, test management and test execution is loosely coupled and allows in a simple way connecting **TestBrowser** to different kind of test frameworks.

1.2 Common example

The example provides a simple test suite as described below. After executing the test suite hierarchy, success and failure messages are written to console and to result file.

The following chapters describe the structure of the example, the principles for running a test including a simple test case and the features provided by the command shell test framework.

The test technology used for component and unit tests is based on a few simple procedures and third party tools (Linux tools for MS Windows: **find**, **diff**, **grep**, **sed**). In order to run tests the following steps are necessary:

1. Before running tests, data and expected results have to be provided in an appropriate structure below a test root directory (see "Preparing data" below). The location of the test root directory does not matter, since all path definitions are relative.
2. When default actions or files do not match the requirements for some tests, those have to be overloaded by appropriate **actions** and **data** sub directory in the test suite directory.
3. Test suites or tests have to be executed in test runs and results have to be checked by calling `RunTestSuite`. In case of test failures, single tests may be corrected and repeated by calling `RunTest`.

When a test fails, there are always two possibilities: Either the tested component is not correct or the expected result has been defined improperly. Both must taken into account.

In order to manage test suites and test cases more comfortable, a GUI application (**TestBrowser**) has been provided. **TestBrowser** created a database from the file system (directory structure). All relevant information is stored in the file system. When starting the **TestBrowser** the first time, the database will be created from the file system data. Running the **TestBrowser**, database and file system are automatically synchronized.

Example structure

Within the example, following directories are used:

- **data** - contains test set and results
- **expectations** - contains expected results
- **actions** - contains test run actions

In order to support inheritance features in a simple way, the example refers to a hierarchical test suite arrangement. The example provides a simple test suite hierarchy below the main suite:

-

main_suite	suite
main_suite/1	suite
main_suite/1/1	suite, run
main_suite/1/1/0	suite, run
main_suite/1/1/1	suite
main_suite/1/2	suite, run
main_suite/1/3	suite
main_suite/1/3/1	suite (disabled)
main_suite/1/3/2	suite (disabled)
main_suite/2	suite, run

Test suites marked as *run* are tests and will be executed in a test run. Test suites marked as *suite* allow providing resources for a test in the work area. In the example, each directory may define a test suite and a test at the same time. The example framework does not support more than one test for a test suite.

It is assumed, that test suites are provided according to a test case hierarchy below a common test root directory (`ROOT_DIR`). The test root contains common resources as test frame work procedures, tools and executables (**binaries**) and the work area (**work_area**) The top level for a test suite hierarchy for testing a component is a directory with the name **main_suite** below test root. This contains common resources for all the subsequent test cases. Sub directories are numbered (1, ... n).

In order to mark test suites as suite and/or test, two special files may be provided in each test suite directory:

- **suite** - Each test suite in the hierarchy contains a file named **suite**. The system is constructed in a way, that it will stop recursion when the first folder is met, which does not contain a **suite** file. The file may be empty or may contain a short description of the test suite concept (e.g. test intend).
- **run** - Test suites, which are also tests and have to be executed, contain a file named **run**. The **run** file may contain a description or comments for the test run to be executed.

The content of these files is of no relevance for the test framework.

Default resources may be defined in `main_suite`, which may be replaced or extended by resources defined further down in the test suite hierarchy. `main_suite` is usually marked as *suite* but not as *run*.

Besides some procedures managing complex test runs, the test root directory contains following directories:

- **main_suite** - top entry for test suite hierarchy (`ROOT_DIR`/**main_suite**)
- **test_runs** - contains a sub directory for each test run
- **binaries** - binaries or executables to be tested (`BIN_DIR`)
- **binaries/tools** - third party tolls used for running tests (`TOOLS_DIR`)
- **work_area** - work area (`WORK_AREA`)

Component to be tested

Ea component to be tested, **MyTest.cmd** is called from the binaries directory. the shell echo command had been selected. The procedure simply calls the `echo` command for creating an output file containing the test suite pass for the test suite currently tested.

```
test root/binaries/MyTest.cmd
  rem 1 - current test suite location in hierarchy
  echo %1>>myTestResult.txt
```

Notes: Running tests will not only test the component, but also the test data and expectations. Usually, it will take a while, until test data and expectations work properly.

Data used

The simple component to be tested does not require input data, but will produce an output file, only. Hence, the **data** directories are empty.

Except test suite **1/2**, all the other test suited marked as run contain a file **myTestResult.txt** with the expected text data in sub directory **expectations**. The corresponding file in test suite 1/2 has different content and will, hence, fail when running tests.

After executing the test suite hierarchy, results.out and logfile.out contain more or less the subsequent data. In order to get a more detailled protocol, one may call TestOut.cmd.

```
test root/test_runs/run_X/results.out (after calling RunTestSuite)
  1\1\\; success
  1\1\0\; success
  1\2\\; failed
  2\\\; success

test root/test_runs/run_X/logfile.out
  1\1\\ started : 23.02.2014 18:45:53,83
  1\1\\ finished: 23.02.2014 18:45:53,83
  1\1\0\ started : 23.02.2014 18:45:54,06
  1\1\0\ finished: 23.02.2014 18:45:54,06
  1\2\\ started : 23.02.2014 18:45:54,42
  1\2\\ finished: 23.02.2014 18:45:54,42
  2\\\ started : 23.02.2014 18:45:54,87
  2\\\ finished: 23.02.2014 18:45:54,87
```

-

1.2.1 Running a test

In order to run a test for one or more test cases, a test run environment (run root) has to be provided, which contains test runs in the same hierarchy as in the **main_suite**. For running tests, only the test case directory structure is required (without data, actions and expected directory).

- run root (`RUN_PATH`)
- **main_suite**
  - actions (`run`, `preprocessing`, `postprocessing`, `compare`, ..)
  - **data**
- binaries (executable files)
  - *MyTool*
  - **tools** (`diff`, `find`, `grep`, `sed`)
- **work_area**
- data (in/out)
- actions
- expected

The **binaries** directory contains tools and programs needed for running the test. **work_area** provides all test specific resources, i.e. the work area may be used for one test at the time. Before running a test, the work area has to be prepared for the test. This is a task of the test framework.

For running a test, the **main_suite/actions** directory contains a `run` action. In case of requested additional actions within a test, the `run` action has to be updated and stored in the directory of the corresponding test case or test suite. Moreover, the `preprocessing` action has to be provided in the **main_suite/actions** directory and is overwritten in some test suites in order to perform actions in the preprocessing step.

After running the test, test results are evaluated by comparing data collected in the **expected** directory with the data created in **data** directory (`postprocesssing` and `compare` action). In case of differences, result files differing are copied from **data** to **failed**.

Since the work area will be recreated for each test, default procedures for actions are stored in **main_suite/actions**. At least three default actions `preprocessing`, run and `postprocessing` have to be provided. Since actions may be replaced by other actions with the same name (overloading), these actions are called **dynamic actions**.

Common settings for environment variables are defined in `settings` action, which is part of the test framework.

Pre-processing

The `preprocessing` action allows providing specific actions to be called before running the test. Typical actions are copying additional data or manipulating test data for meeting test case requirements. The default action does nothing, but may be overloaded in subordinated test cases (e.g. generating test suites from a test suite template).

The `preprocessing` action is called and defined differently for Windows (cmd) and Linux (bash).

Notes: Overloaded `preprocessing` action in order to set the class name for running unit tests.

Executing test

Running the test suite depends on the test to be performed. Typically, the `run` action is implemented on a higher level in the hierarchy (main suite). When, however, considering another hierarchy, where each call gets its own test suite below the main test suite, the `run` action could be implemented on the next lower level.

As first and only parameter the current test or test suite is passed (path relative to **main_suite**).

Summary information (start and stop time) is written to a log file (`LOGFILE_OUT`) in the test run directory.

`run` calls the table compiler and writes start and stop information to **logfile.out**. An example for overwriting the run action was the requirement calling the table compiler twice. An example is given in the ODABA test suite under **Utilities/CopyDB/actions.**

The `run` action is called and defined differently for Windows (cmd) and Linux (bash).

Notes: The `run` action includes additional actions for creating statistics before and after copying the database in order to compare copied database content.

Post-processing

The default `postprocessing` action compares all files defined in the **expected** directory with the files created in the **data** directory by calling `compare` after copying all error files (**.err**) to the test suite directory.

The `Compare` action compares the file passed with a file having the same file name in the data directory in the work area. When comparing returns differences, those are written to the `COMPARE_OUT` location, which has been defined in settings (**compare.out** in test run directory). In case of errors (e.g. missing file in data directory), error messages are written to the error location `ERRORS_OUT` (**errors.out** in test run directory).

When comparing a file fails, the file is copied to the **failed** directory in the test suite (`copyFailed`) and an error notice is written to **compare.out**.

More enhanced compare could be easily achieved by calling `grep` (remove lines not to be compared) and `sed` (change lines, i.g. for removing comments) before comparing files.

Usually, the `postprocessing` action has to be provided for each test system ones on top of the hierarchy. But it may also be overloaded for any test suite in the test suite hierarchy.

The `postprocessing` action is called and defined differently for Windows (cmd) and Linux (bash).

1.2.2 How to use the example frame work

The following chapters will explain, how to use the test framework. It looks a bit complicate (and it is, indeed) because of a rather complex directory structure, but tests need a lot of details and are complex by nature. The problem is to manage complexity in a proper way. The test framework is a good mean for running tests and managing test runs, but it becomes a nightmare when trying to prepare tests just by creating and editing files and directories in the file system. For managing test data, it is much more comfortable using tools like **TestBrowser**, which will be explained later. This chapter provides a look behind the scene and will improve understanding about what is going on when preparing and running tests with tools like **TestBrowser**. On the other hand, the test framework may be used as standalone tool, also.

When explaining how to prepare, execute and evaluate tests, the ODABA release test delivered with ODABA **TestBrowser** is referenced as example. It consists of a combination of unit and system tests to be performed when releasing a new ODABA version. The example demonstrates principles when using the test framework. The first test suite refers to unit tests for the ODABA **API** containing two test suites for testing the **Local** and **ClientServer** version. The second test suite **OSI** covers a series of tests for OShell commands (**OShell**) and special OSI operations (**Operation**). The last test suite contains test cases for ODABA utility programs (**Utilities**).

- **main_suite**
- **API**
    - **Local**
        - **Application**
        - **Binary**
        - **Database**
        - ...
    - **ClientServer**
    - **Application**
    - **Binary**
    - **Database**
    - ...
- **OSI**
    - **OShell**
    - **Operation**
    - **Extensions**
    - **Templates**
    - ...
- **Utilities**
- **BackupDB**
- **CheckDB**
- ...

Green names in the hierarchy denote test cases, which are associated with tests in one or more test runs. Blue names denote test suites. The **main_suite** is stored in the directory **ODABATest/TestRun**, which is the root directory for the test framework. This directory also contains the command line procedures of the test frame work.

-

1.2.2.1 Preparing tests

The testable unit for the test framework is a test case. Each test case is represented by a directory with an appropriate test case name. Test cases may be grouped in test suites by locating test case directories as sub directories below a test suite directory. Test suites may be grouped, again, in upper test suites etc.

Each test suite directory must contain a **suite** file, which contains a short description of the for the test suite. Test case directories need a **suite** file, which allows storing explanations for the test and a **run** file, which may also contain special hints for executing the test case. Content of **suite** and **run** files is needed for documentation purpose, only, i.e. files may also be empty.

In order to run a test case, one has to prepare at least the **run** file in the **actions** sub directory for calling actions to be executed. This and other actions that may be overloaded (e.g. **preprocessing**) are stored in the **actions** sub directory. When not defining **preprocessing.cmd** and **postprocessing**, those are taken from the next higher test suite directory that provided appropriate command files in its actions sub directory. The test framework always collections actions along the test suite hierarchy, where command line files provided on lower level will overwrite command line files with the same name provided on higher levels.

Test specific data has to be provided in the **data** sub directory for each test case. Similar to command line files in actions directories, data common for several test cases may be stored in the **data** sub directory of test suites on different hierarchy levels. When running test cases, test data will be collected along the test suite parent hierarchy.

Expected test results are provided in the **expected** sub directory. The **expected** sub directory must contain at least one file with expected data. Also, expected data may be provided in test suite **expected** sub directories.

1.2.2.2 Making use of data inheritance

Test data maintenance becomes a problem, when a new software version requires new test data, actions or expected data has to be changed. When each test case provides its own data, this may result in a lot of test data updates. Sometimes, this can be done automatically, but often this has to be done manually. In order to reduce the amount of test data (actions, data, expected), test data may be inherited from test suite. All test data provided in **actions**, **data** and **expected** directory of the upper test suite is available for all test cases belonging to the test suite. In case that files that are provided for test suite and for a test case, test data from the test case is used, i.e. test case test data overwrites test data provided in the test suite.

The same way as test cases may share test data provided in the upper test suite, test suites may also share test data stored in the next higher test suite. Thus, test data may be reduced by storing it in the right place. Typically, actions are shared between test suites and test cases. Sometimes, also data is shared. Sharing expected data is, however, a rare case.

Before executing a test, actions, data and expected files are collected for each test case along the parent test suite hierarchy are collected and copied to **actions**, **data** and **expected** sub directory of the **work_area**. The disadvantage is, that test data stored in the file system is not transparent anymore, since one has to check the complete hierarchy in order to find out, what kind of test data is really involved in test. A work around is provided with the action `SetupWorkArea`, which creates the work area for the required test case. **TestBrowser** a.so shows "merged collections", which contain the test data as being merged from test case and parent directories.

1.2.2.3 Executing tests

For executing test cases, required resources (**actions**, **data**, **expected**) are copied from test cases and test suite directories to a work area, when calling appropriate test run commands:

For running a single test case, `RunTest` has to be called:

`RunTest` "API/Local/Application" "%cd%/test_runs/*TestRun001*" "%cd %/main_suite" "%cd%" ""

Instead of *TestRun001* any other test run name may be used. The test run directory will be created when not yet existing.

In order to execute the complete hierarchy, `RunMany` has to be called:

`RunMany` "%cd%/test_runs/*TestRun001*" "%cd%/main_suite"

from the test root directory (ODABATest/TestRun). For executing a specific test suite (e.g. Utilities)

`RunMany` "%cd%/test_runs/*TestRun001*" "%cd%/main_suite/Utilities"

may be called, which will execute all test cases below **Utilities**. All the commands will finally call `testRun`, which executes following actions:

- `settings` - set environment variables
- `prepare` - copy test data (recursively) to **work_area** directory
- `preprocessing` - perform test specific test preparation (has to be provided in test case or test suite **actions**)
- `run` - execute test (has to be provided in test case or test suite **actions**)
- `postprocessing` - typically, the procedure removes variable data, e.g. timestamps, from test data before being compared (has to be provided in test case or test suite **actions**). Finally, the procedure has to create a **compare.out** file (e.g. by comparing expected and test result files).
- `report` - writes success or failed state in a test protocol file

-

1.2.2.4 Test evaluation

Evaluating test results is typically done by comparing expected data with data created by running a test. The test framework only expects that test results for a test are summarized in a file **compare.out** (`COMPARE_OUT` environment variable). When no differences have been found between test results and expected data, the file is empty (success). When one or more files differ between expected data and test results, the differences should be listed in the **compare.out** file (failed). When test had not been executed for any other reason, the **compare.out** file does not exist.

Creating the **compare.out** file, is task of the `postprocessing` action. This is not part of the test framework, but has to be provided for a specific test environment. The ODABATest environment provides the procedures listed below:

- `postprocessing` - Preparing files for compare (`GetErrorsFromLog` and `Compare`)
- `GetErrorsFromLog` - Remove timestamps and directory paths from test **error.lst** and **output.lst** files, which are created by all test test runs.
- `Compare` - Compares updated files with expected results. In case of differences, those are written to `COMPARE_OUT` (**compare.out**) file and the test result file differing is written to the **test/failed** directory (`CopyFailed` action)
- `CopyFailed` - Save test result file differing from expected data in **failed** directory.

From the compare.out file, the results.out file for the complete test run is appended with the test result for the test (success or failed). In addition, the run procedure for ODABATest environment creates a log file **logfile.out** containing start and stop time for each test.

```
# postprocessing (Linux):
# before comparing remove time stamps, file locations and other variable
data
# compare all expected files with data created by test

  ${TEST_ACT}/GetErrorsFromLog
  cp -f ${WORK_AREA}/data/*.err ${TEST_RUN} 2>/dev/null
  for x in ${WORK_AREA}/expected/*; do ${TEST_ACT}/Compare $x; done

# GetErrorsFromLog (Linux):
# Remove time stamps and file locations

  if [ -f ${WORK_AREA}/data/output.lst ] ; then
    cat ${WORK_AREA}/data/output.lst | ${ODABA_TOOLS}/ReplaceTextL $
{WORK_AREA} ... | ${ODABA_TOOLS}/ReplaceTextL ${ODABA_ROOT} ... >$
{WORK_AREA}/data/output.out ;
  fi
  if [ -f ${WORK_AREA}/data/error.lst ] ; then
    cat ${WORK_AREA}/data/error.lst | sed "s/[0-9][0-9][0-9][0-9]\/[0-9]
[0-9]\/[0-9][0-9] [0-9][0-9]:[0-9][0-9]:[0-9][0-9]/                   /"
>${WORK_AREA}/data/Utility.err
  fi

# Compare (Linux):
# Compare two files and save result in case of differences
# 1 - file name (complete path) to be compared

  echo "Compare $1"
  diff -b "${WORK_AREA}/data/$(basename $1)"  "$1" >>${COMPARE_OUT} 2> $
{ERRORS_OUT} || ${TEST_ACT}/CopyFailed $1

# CopyFailed (Linux):
# copy files causing problems
# 1 - file name (complete path) to be copied

  if [ ! -d ${TEST_RUN}/failed ] ; then
    mkdir ${TEST_RUN}/failed
  fi
  cp "${WORK_AREA}/data/$(basename $1)" "${TEST_RUN}/failed/$(basename
$1)"
  echo "File ${TEST_RUN}/data/$(basename $1) missing or differs from
expected result" >>${COMPARE_OUT}
```

2 Test framework actions (Linux)

The bash test framework consists of a number of procedures for managing tests under Linux.

In order to run a test, all required resources are copied to the work area (**work_area**) which is structured according to the requirements of the component to be tested. When running a series of tests in the test framework, the work area will be overwritten each time, a new test is started. In order to guarantee traceability, the work area has to be copied to the test suite running the test (e.g. as zip file). Here, only differing files are stored to the **failed** directory in the test suite.

The test framework provides a set of common actions for managing test preparation and evaluation. In addition, each specific test environment has to provide adopted actions named `preprocessing`, `run` and `postprocessing` (bash file names without extension), which are usually stored in **main_suite/actions** directory, and possibly overloaded in several test cases or test suites. Test framework actions (procedures) are stored in the test root directory.

Several actions that are called internally by the framework are defined as functions in **TestSuite.sh**. Framework actions that may be called from command line are **RunTest.sh**, **RunMany.sh**, **SetupWorkArea.sh** and **TestOut.sh**. In order to support parallel testing, the work area location may also be passed as parameter to `RunTest` or `RunMany` action.

Starting test run(s)

In order to execute a single or a number test cases, framework actions `RunTest` or `RunMany` may be called.

For executing a single test case, `RunTest` is called. In order to locate the proper test case, the relative path has to be defined like **API/Local/Property**.

In order to run the complete test suite hierarchy, `RunMany` may to be called. The procedure executes all test case directories containing a **run** file. In order to filter test cases for execution, one may rename the **run** file in test case directories to be excluded.

For each test directory, `runSingle` will be called. `runSingle` executes a single test suite after checking the presence of a **run** file in the directory. When the **run** file exists, `testRun` is called.

```
# RunTest.sh
#!/bin/bash
source ./TestSuite.sh
# execute single test
  # 1 - relative test suite path (e.g. 0/1/2)
  test_suite=$1 # CHAR
  # 2 - test run directory (${RUN_PATH%})
  run_root=$2 # CHAR
  # 3 - main suite directory (${ROOT_PATH})
  main_suite=$3 # CHAR
  # 4 - test root directory ($PWD)
  root_dir=$4 # CHAR
  # 5 - work area location (${WORK_DIR})
  WORK_dir=$4 # CHAR

  settings ${test_suite} ${run_root} ${main_suite} ${root_dir} $
{work_dir} ;
  pushd ${RUN_ROOT} > /dev/null
  runSingle ./${test_suite}
  popd > /dev/null

# RunMany.sh
#!/bin/bash
source ./TestSuite.sh
# execute single test
  # 1 - relative test suite path (e.g. 0/1/2)
  test_suite=$1 # CHAR
  # 2 - test run directory (${RUN_PATH%})
  run_root=$2 # CHAR
  # 3 - main suite directory (${ROOT_PATH})
  main_suite=$3 # CHAR
  # 4 - test root directory ($PWD)
  root_dir=$4 # CHAR
  # 5 - work area location (${WORK_DIR})
  work_dir=$4 # CHAR

  settings ${test_suite} ${run_root} ${main_suite} ${root_dir} $
{work_dir} ;
  pushd %RUN_ROOT%
  find ./ -type d -exec ${ROOT_DIR}\runSingle.sh {} $1 ;
  popd

# run single test in hierarchy (TestSuite.sh)
runSingle() {
# 1 - test directory below main suite .../main_suite
  test_suite=$1 # CHAR

  #CODE
  if [ -f ${MAIN_SUITE}/$1/run ]; then
    testRun ${test_suite};
  fi
}
```

-

Running a test within the frame work

In order to run a test, the `testRun` action is provided by the test framework (**TestSuite.sh**). The `testRun` action updates settings for environment variables, prepares the work area and calls test environment specific actions for the selected work suite. `settings` and `prepare` are framework actions, which cannot be overloaded, while `preprocessing`, `run` and `postprocessing` are test environment specific actions, which have to be defined for each specific test environment in top or lower test suite or test case **actions** directory.

After setting test environment variables (`settings` action), the `prepare` action copies test data (**actions**, **data**, **expected**) to corresponding directories in **work_area** from the test suite hierarchy. Test data from test cases and test suites is copied recursively (`copyRecursive`) from the test suite hierarchy by copying resources from the test case to be tested and all its parent test suites. A directory is considered as test suite, when it contains a file **suite**. Copying stops, when the first directory was found, which is not a test suite. Files existing on lower levels in the hierarchy, always will overwrite files with the same name provided on higher levels.

When `NO_RUN` has been set to **YES** (`SetupWorkArea`), the `testRun` action terminates without executing the test run. Otherwise actions `preprocessing`, `run` and `postprocessing` are called.

Finally the test result is checked by comparing data and writing differences to `COMPARE_OUT` (**compare.out**) file in test run directory. When `COMPARE_OUT` is empty, the test status is **success**, and **failed** otherwise. The result is written to console and to **results.out** in the test run directory by the `report` action.

```
# selected actions called by testRun in TestSuite.sh
# execute singele test
testRun() {
# 1 - test directory below main suite .../main_suite
  test_suite=$1 # CHAR

  #CODE
  pushd . > /dev/null
  # prepare test work suite ... (global actions)
  settings ${test_suite} ${RUN_ROOT} ${MAIN_SUITE} ${ROOT_DIR}
  prepare
  source ${TEST_ACT}/preprocessing
  if [ ! "${NO_RUN}" == "YES" ]; then
    # run test ... (actions may be overloaded)
    preprocessing ${test_suite}
    ${TEST_ACT}/run ${test_suite}
    ${TEST_ACT}/postprocessing ${test_suite}
    # create test report
    report ${test_suite}
  fi
  popd > /dev/null
}
```

```
# Set global test frame work variables
settings() {
  # 1 - relative test suite path (e.g. 0/1/2)
  test_suite=$1 # CHAR
  # 2 - test run directory (${RUN_PATH%})
  run_root=$2 # CHAR
  # 3 - main suite directory (${ROOT_PATH})
  main_suite=$3 # CHAR
  # 4 - test root directory ($PWD)
  root_dir=$4 # CHAR
  # 5 - temporary work directory (${WORK_DIR})
  work_dir=$5 # CHAR

  #CODE
  if [ "${root_dir}" == "" ]; then
    export ROOT_DIR=$PWD;
  else
    export ROOT_DIR=${root_dir};
  fi

  if [ "${main_suite}" == "" ]; then
    export MAIN_SUITE=${ROOT_PATH};
  else
    export MAIN_SUITE=${main_suite};
  fi

  if [ "${run_root}" == "" ]; then
    export RUN_ROOT=${RUN_PATH};
  else
    export RUN_ROOT=${run_root};
  fi
  if [ ! -d ${RUN_ROOT} ]; then
    mkdir -p ${RUN_ROOT};
  fi

  if [ "${test_suite}" == "" ]; then
    export TEST_SUITE=${MAIN_SUITE};
  else
    export TEST_SUITE=${MAIN_SUITE}/${test_suite};
  fi

  if [ "${work_dir}" == "" ]; then
    if [ "${WORK_DIR}" == "" ]; then
      WORK_AREA=${ROOT_DIR}/work_area;
  else
      WORK_AREA=${WORK_DIR};
  fi
  else
    WORK_AREA=${work_dir};
  fi
  if [ ! -d ${WORK_AREA}/actions ]; then
    mkdir -p ${RUN_ROOT}/actions;
  fi
  if [ ! -d ${WORK_AREA}/data ]; then
    mkdir -p ${RUN_ROOT}/data;
  fi
  if [ ! -d ${WORK_AREA}/expected ]; then
    mkdir -p ${RUN_ROOT}/expected;
```

-

```
  fi

  export WORK_AREA
  export TEST_ACT=${WORK_AREA}/actions
  export BIN_DIR=${ROOT_DIR}/binaries
  export TOOLS_DIR=${BIN_DIR}/tools
  export TEST_RUN=${RUN_ROOT}/${test_suite}

  export LOGFILE_OUT=${RUN_ROOT}/logfile.out
  export RESULTS_OUT=${RUN_ROOT}/results.out
  export REPORT_OUT=${RUN_ROOT}/report.out
  export ERRORS_OUT=${TEST_RUN}/errors.out
  export COMPARE_OUT=${TEST_RUN}/compare.out

  export LD_LIBRARY_PATH=/usr/local/lib
  export ODABA_TOOLS=/usr/local/lib/odaba/tools

  export TEST_NAME=$(basename ${TEST_SUITE})
}

# Initialize test folder
prepare() {
  #CODE

  # delete last test suite results
  rm -f ${COMPARE_OUT}
  rm -f ${ERRORS_OUT}

  # delete work suite data
  rm -rf ${WORK_AREA}/data/*
  rm -rf ${WORK_AREA}/expected/*
  rm -rf ${WORK_AREA}/actions/*
  rm -rf ${TEST_RUN}/failed/*

  pushd ${TEST_SUITE} > /dev/null
  copyRecursive
  popd > /dev/null
}

# Hierarchical copies contents of data, expected and actions into the $
{work_area}
copyRecursive() {
  #CODE
  if [ -f suite ]; then
  pushd .. > /dev/null
    copyRecursive
    popd > /dev/null
    cp -r data/* ${WORK_AREA}/data 2>/dev/null
    cp -r expected/* ${WORK_AREA}/expected 2>/dev/null
    cp -r actions/* ${WORK_AREA}/actions 2>/dev/null
  fi
}

# evaluate termination code
report() {
# 1 - test directory below main suite .../main_suite
  test_suite=$1 # CHAR
  test_result=success
  #CODE
  if [ -f ${COMPARE_OUT} ]; then
    diff "${ROOT_DIR}/main_suite/compare.top" "${COMPARE_OUT}" >
/dev/null || test_result=failed
  else
    test_result=failed
  echo "... nothing to compare (expected empty)" >${COMPARE_OUT}
  fi
  echo "${test_suite}; ${test_result}" >>${RESULTS_OUT}
  echo "${test_suite} terminated: ${test_result}"
  if [ ! -f ${TEST_RUN}/run ]; then
    cp ${TEST_SUITE}/run ${TEST_RUN}/run
  fi
}
```

-

Output test protokol

Remarks about the test runs may have been stored in each test run directory in the run file. In order to print out a comprehensive test protocol, one may call `TestOut` framework action (**TestOut.sh**). The action calls `outRun` for each sub directory containing a **run** file.

More reporting features are provided with ODABA **Test Browser** application.

```
# TestOut.sh
#!/bin/bash
source ./TestSuite.sh
# execute single test
  # 1 - relative test suite path (e.g. 0/1/2)
  test_suite=$1 # CHAR
  # 2 - test run directory (${RUN_PATH%})
  run_root=$2 # CHAR
  # 3 - main suite directory (${ROOT_PATH})
  main_suite=$3 # CHAR
  # 4 - test root directory ($PWD)
  root_dir=$4 # CHAR

  settings ${test_suite} ${run_root} ${main_suite} ${root_dir} ;

  rm -f ${REPORT_OUT}

  pushd ${RUN_ROOT}
  find ./ -type d -exec ${ROOT_DIR}/outRun.sh {} ${test_suite} ;
  type ${RESULTS_OUT}
  popd

# create test run summary and print to REPORT_OUT (TestSuite.sh)
outRun() {
# 1 - test directory below main suite .../main_suite
  test_suite=$1 # CHAR

  #CODE
  settings %1 %ROOT_DIR%
  if [ -f ${TEST_SUITE}/run ]; then
    pushd ${TEST_RUN} > /dev/null
    echo Tested ${test_suite}>>${REPORT_OUT}
    if [ -f run ]; then
    type run >>${REPORT_OUT}
  fi
    echo .>>${REPORT_OUT}
    echo ----------------------------------------->>${REPORT_OUT}
    popd > /dev/null
  fi
}
```

3 Test framework actions (Windows)

The cmd test framework consists of a number of procedures managing tests under Windows.

In order to run a test, all required resources are copied to the work area (**work_area**) which is structured according to the requirements of the component to be tested. When running a series of tests in the test framework, the work area will be overwritten each time, a new test is started. In order to guarantee traceability, the work area has to be copied to the test suite running the test (e.g. as zip file). Here, only differing files are stored to the **failed** directory in the test suite.

The test framework provides a set of common actions (command line procedures) for managing test preparation and evaluation. In addition, each specific test environment has to provide adopted actions named `preprocessing`, `run` and `postprocessing` (command line procedures with extension **.cmd**), which are usually stored in **main_suite/actions** directory, and possibly overloaded in several test cases or test suites. Test framework actions (procedures) are stored in the test root directory.

Framework actions that may be called from command line are **RunTest.cmd**, **RunMany.cmd**, **SetupWorkArea.cmd** and **TestOut.cmd**. In order to support parallel testing, the work area location may also be passed as parameter to `RunTest` or `RunMany` action. Default is the **work_area** directory below the test root directory.

Starting test run(s)

In order to execute a single or a number test cases, framework actions `RunTest` or `RunMany` may be called.

For executing a single test case, `RunTest` is called. In order to locate the proper test case, the relative path has to be defined like **API/Local/Property**.

In order to run the complete test suite hierarchy, `RunMany` may to be called. The procedure executes all test case directories containing a **run** file. In order to filter test cases for execution, one may rename the **run** file in test case directories to be excluded.

-

For each test directory, `runSingle` will be called. `runSingle` executes a single test suite after checking the presence of a **run** file in the directory. When the **run** file exists, `testRun` is called. `testRun` is called via `callNormalized`, which normalizes the directory path by removing ./ at the beginning of the path name (this type of path names are not supported by older command line functions as **type** or **copy**).

```
// RunTest.cmd
  @echo off
  rem execute single test
  rem 1 - relative test suite path (e.g. 0/1/2)
  rem 2 - test run directory (%RUN_PATH%)
  rem 3 - main suite directory (%ROOT_PATH%)
  rem 4 - test root directory (%cd%)
  rem 5 - work directory (%WORK_DIR%)

  call %4\settings.cmd %1 %2 %3 %4 %5

  pushd %RUN_ROOT%
  call %ROOT_DIR%\runSingle.cmd ./%~1
  popd

// RunMany.cmd
  @echo off
  rem execute test run(s)
  rem 1 - test run directory (%RUN_PATH%)
  rem 2 - main suite directory (%ROOT_PATH%)
  rem 3 - test root directory (%cd%)

  if "%3" == "" (call %cd%\settings.cmd . %1 %2) else (call
%3\settings.cmd . %1 %2 %3)

  pushd %2
  %TOOLS_DIR%\find . -type d -exec %ROOT_DIR%\runSingle.cmd {} ;
  popd

// runSingle.cmd
  @echo off
  rem run single test in hierarchy
  rem 1 - test folder name below main suite ...\main_suite

  if exist %MAIN_SUITE%\%1\run %ROOT_DIR%\callNormalized.cmd %ROOT_DIR
%\testRun.cmd %1

// CallNormalized.cmd
@echo off
rem convert Linux path to MS Windows path (./a/b... --> a\b...)
rem 1 - procedure to be called
rem 2 - "find" name for directory or file
FOR /F "tokens=2,3,4,5 delims=/" %%G IN ('echo %2') DO call %1 %%G\%%H\%
%I\%%J %2
```

Running a test within the frame work

In order to run a test, the `testRun` action is provided by the test frame work (**testRun.cmd**). The `testRun` action updates settings for environment variables, prepares the work area and calls test environment specific actions for the selected work suite. `settings` and `prepare` are framework actions, which cannot be overloaded, while `preprocessing`, `run` and `postprocessing` are test environment specific actions, which have to be defined for each specific test environment in top or lower test suite or test case **actions** directory.

After setting test environment variables (`settings` action), the `prepare` action copies test data (**actions**, **data**, **expected**) to corresponding directories in **work_area** from the test suite hierarchy. Test data from test cases and test suites is copied recursively (`copyRecursive`) from the test suite hierarchy by copying resources from the test case to be tested and all its parent test suites. A directory is considered as test suite, when it contains a file **suite**. Copying stops, when the first directory was found, which is not a test suite. Files existing on lower levels in the hierarchy, always will overwrite files with the same name provided on higher levels.

When `NO_RUN` has been set to **YES** (`SetupWorkArea`), the `testRun` action terminates without executing the test run. Otherwise actions `preprocessing`, `run` and `postprocessing` are called.

Finally the test result is checked by comparing data and writing differences to `COMPARE_OUT` (**compare.out**) file in test run directory. When `COMPARE_OUT` is empty, the test status is **success**, and **failed** otherwise. The result is written to console and to **results.out** in the test run directory by the `report` action.

```
// testRun.cmd
  @echo off
  rem execute test suite
  rem 1 - work suite directory name below main_suite
  pushd .
  rem prepare test work suite ... (global actions)
  call %ROOT_DIR%\settings.cmd %1 %RUN_ROOT% %MAIN_SUITE% %ROOT_DIR%
  call %ROOT_DIR%\prepare.cmd
  if "%NO_RUN%" == "YES" goto end
  rem run test ... (actions may be overloaded)
  call %TEST_ACT%\preprocessing.cmd %1
  call %TEST_ACT%\run.cmd %1
  call %TEST_ACT%\postprocessing.cmd %1
  rem create test report
  call %ROOT_DIR%\report.cmd %1
  :end
  popd

// settings.cmd
  rem Set test environment variables
  rem 1 - relative test suite path (e.g. 0/1/2)
  rem 2 - test run directory (%RUN_PATH%)
  rem 3 - main suite directory (%ROOT_PATH%)
```

-

```
  rem 4 - test root directory (%cd%)
  rem 5 - work directory (%WORK_DIR%)

  rem global test environment
  if "%4" == "" (set ROOT_DIR=%cd%) else (set ROOT_DIR=%~4)
  if "%3" == "" (set MAIN_SUITE=%ROOT_PATH%) else (set MAIN_SUITE=%~3)
  if "%2" == "" (set RUN_ROOT=%RUN_PATH%) else (set RUN_ROOT=%~2)
  if "%1" == "" (set TEST_SUITE=%MAIN_SUITE%) else (set TEST_SUITE=
%MAIN_SUITE%\%~1)
  set TOOLS_DIR=%ROOT_DIR%\binaries\tools
  set BIN_DIR=%ROOT_DIR%\binaries

  rem test work area settings
  if "%~5" == "" (set WORK_AREA=%WORK_DIR%) else (set WORK_AREA=%~5)
  if "%WORK_AREA%" == "" set WORK_AREA=%ROOT_DIR%
  set WORK_AREA=%WORK_AREA%\work_area
  set TEST_ACT=%WORK_AREA%\actions

  rem prepare test run directory
  if not exist %RUN_ROOT% mkdir %RUN_ROOT%
  set TEST_RUN=%RUN_ROOT%\%~1
  set LOGFILE_OUT=%RUN_ROOT%\logfile.out
  set RESULTS_OUT=%RUN_ROOT%\results.out
  set REPORT_OUT=%RUN_ROOT%\report.out
  if not exist %TEST_RUN% mkdir %TEST_RUN%

  rem error protocols
  set ERRORS_OUT=%TEST_RUN%\errors.out
  set COMPARE_OUT=%TEST_RUN%\compare.out
  :end

// prepare.cmd
  rem Initialize test folder
  rem delete last test suite results
  cd /D %TEST_SUITE%
  if exist %COMPARE_OUT% del %COMPARE_OUT%
  if exist %ERRORS_OUT% del %ERRORS_OUT%

  rem delete work suite data
  del /Q /S %WORK_AREA%\data\*.* > nul
  del /Q /S %WORK_AREA%\expected\*.* > nul
  del /Q /S %WORK_AREA%\actions\*.* > nul
  del /Q /S %TEST_RUN%\failed\*.* > nul

  rem copy resources
  call %ROOT_DIR%\copyRecursive.cmd

// copyRecursive.cmd
  rem Copy test data recursively from test case hierarchy
  if exist suite (
      pushd ..\.
    call %ROOT_DIR%\copyRecursive.cmd
    popd
    if exist data xcopy /Y data\*.* %WORK_AREA%\data\*.* > nul
    if exist expected xcopy /Y expected\*.* %WORK_AREA%\expected\*.* >
nul
    if exist actions xcopy /Y actions\*.* %WORK_AREA%\actions\*.* > nul
  )
```

```
// report.cmd
  rem evaluate termination code
  rem 1 - work suite directory name below test root\0

  set RESULT=success
  if exist "%COMPARE_OUT%" (
    %TOOLS_DIR%\diff "%ROOT_DIR%\main_suite\compare.top" "%COMPARE_OUT%"
>> nul || set RESULT=failed
  ) else (
    set RESULT=failed
    echo ... nothing to compare (expected empty) >%COMPARE_OUT%
  )
  echo %1; %RESULT% >>%RESULTS_OUT%
  echo %1 terminated: %RESULT%
  if not exist %TEST_RUN%\run copy %TEST_SUITE%\run %TEST_RUN%\run >>nul
```

-

Output test protokol

Remarks about the test runs may have been stored in each test run directory in the run file. In order to print out a comprehensive test protocol, one may call `TestOut` framework action (**TestOut.cmd**). Via calling action callNormalized in order to change Linux paths to Windows path the `outRun` is called for each sub directory containing a **run** file.

More reporting features are provided with ODABA **Test Browser** application.

```
// TestOut.cmd
  @echo off
  rem list test results
  rem 1 - relative test suite path (e.g. 0/1/2)
  rem 2 - test run directory (%RUN_PATH%)
  rem 3 - main suite directory (%ROOT_PATH%)
  rem 4 - test root directory (%cd%)

  call settings.cmd %1 %2 %3 %4
  if exist %REPORT_OUT% del %REPORT_OUT%

  pushd %RUN_ROOT%
  %TOOLS_DIR%\find . -type d -exec %ROOT_DIR%\outSingle.cmd {} %1 ;
  type %RESULTS_OUT%
  popd

// outSingle.cmd
  @echo off
  rem run single test in hierarchy
  rem 1 - test folder name below top suite ...\main_suite
  call %ROOT_DIR%\callNormalized.cmd %ROOT_DIR%\outRun.cmd %1

// outRun.cmd
  @echo off
  rem 1 - test folder name below main suite ...\main_suite
  call %ROOT_DIR%\settings.cmd %1 %ROOT_DIR%
  if exist %TEST_SUITE%\run (
    pushd %TEST_RUN%
    echo Tested %1>>%REPORT_OUT%
    if exist run type run >>%REPORT_OUT%
    echo .>>%REPORT_OUT%
    echo ------------------------------------------>>%REPORT_OUT%
    popd
  )
```

4 TestBrowser

In order to manage file system data in a comfortable way, an ODABA GUI application **TestBrowser** has been provided, which allows creating, importing and removing test cases, updating test files and expected data, running tests and analyzing and displaying test results.

TestBrowser is an ODABA application based on file system data and the ODABA test framework (based on file system and command line procedures). The database content is completely created from file system information, which allows recreating the database completely from file system at any time. One may also provide a subset of a test environment and create a database for this subset. The database is created automatically, when no data base is available when starting **TestBrowser**.

In case of changes made in the (external) file system, the database is synchronized in many cases automatically or may be synchronized explicitly by user action. It is also possible to synchronize the complete database with the file system while running **TestBrowser.**

Changes made from within **TestBrowser** (e.g. deleting files or directories, changing file content) are automatically reflected in the file system.

-

The chapter "Using TestBrowser" describes the most important actions for creating and running tests by means of a simple example. More details about **TestBrowser** functionality is provided in chapter "Action Reference". The chapter "Database access" contains a detailed description of database model and implemented functions, which may be used in extensions or when customizing **TestBrowser**. Since the complete application is written with OSI script language, it becomes quite simple to make any kind of changes.

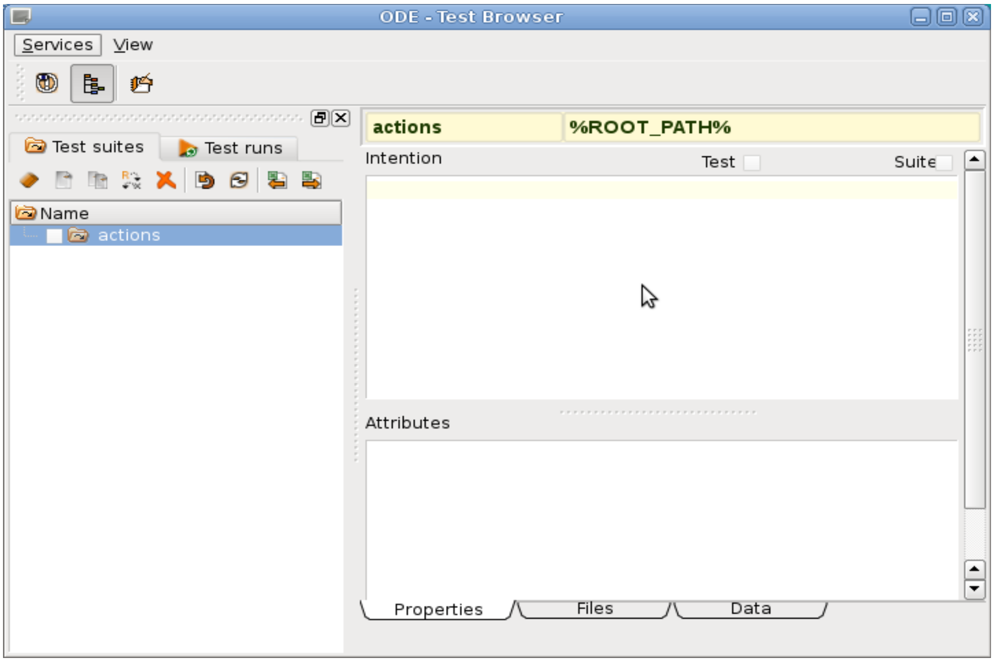
4.1 Using TestBrowser

In order to run **TestBrowser**, one may either call **TestBrowser** command from ODABATest test environment, which will start the ODABA release test environment. One may also create a new test environment by copying the ODABA test environment (**.../TestRoot/Linux** or **...\TestRoot\Windows**) template to a location, which becomes the test root.

4.1.1 Creating a new test environment

The chapter will demonstrate by means of a simple example, how to build a test environment with the **TestBrowser**. Everything done with **TestBrowser** is stored in the file system, i.e. **TestBrowser** is just a tool to simplify test data management. In order to demonstrate the test environment, a **CommandShell** test suite will be created for "testing" the **echo** and **copy** command (test cases).

In order to create a new test environment, the easiest way is to copy the test root template delivered with **TestBrowser** installation. After copying the test root template for Linux or Windows (**.../TestRoot/Linux** or **...\TestRoot\Windows**) to test root location (referenced in future as **test_root** directory), one may call the `TestBrowser` action in the **test_root** directory. Than, an empty **TestBrowser** application pops up showing a single actions directory:



-

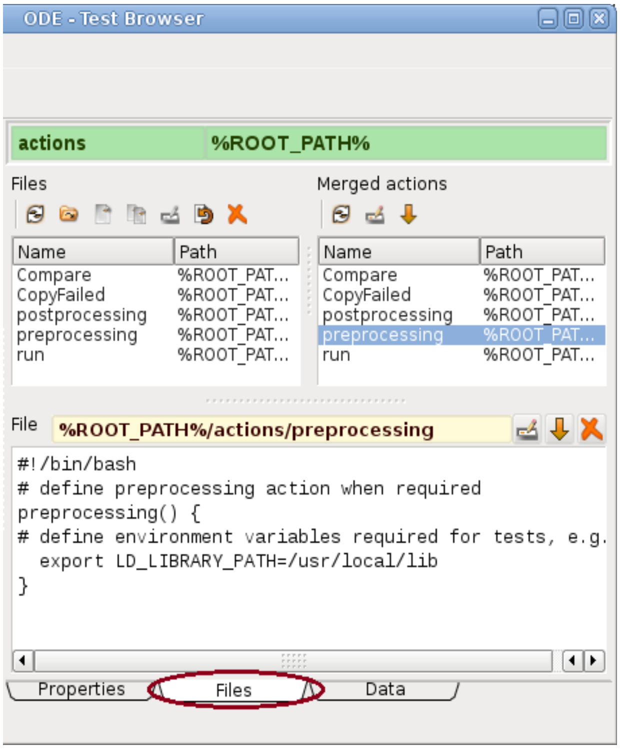
The new test environment does not contain any test suite or test case, but patterns for required test environment actions `preprocessing`, `run` and `postprocessing`.

In order to create a new test suite, one may use the action buttons above the directory tree. After creating test suites and test cases, one may create a test run by selecting any number of test cases or suites from the directory tree (action buttons above the directory tree). In order to execute a test run, the **Test runs** tab above the directory tree has to be selected. The following sections will explain these steps more detailed.

Notes: The examples are written as Linux bash procedures, but work the same way under Windows, except the procedures are cmd procedures and not bash procedures.

4.1.1.1 Update environment specific actions

Environment specific actions are `preprocessing`, `run` and `postprocessing`. Actions delivered with the test environment template are considered as example have to be updated, usually. After selecting the **Files** tab below the edit window, provided actions are shown:



Name	Path
Compare	%ROOT_PAT...
CopyFailed	%ROOT_PAT...
postprocessing	%ROOT_PAT...
preprocessing	%ROOT_PAT...
run	%ROOT_PAT...

Name	Path
Compare	%ROOT_PAT...
CopyFailed	%ROOT_PAT...
postprocessing	%ROOT_PAT...
preprocessing	%ROOT_PAT...
run	%ROOT_PAT...

```
#!/bin/bash
# define preprocessing action when required
preprocessing() {
# define environment variables required for tests, e.g.
  export LD_LIBRARY_PATH=/usr/local/lib
}
```

At least the `run` action requires some modification. Actions can be updated in the lower edit box of the window. When leaving the box, changes are saved to the file automatically. Here, the `run` action will be overloaded later in test cases. The example below shows a pattern for a typical `run` action as delivered with the test environment template.

The `postprocessing` action delivered as pattern provides a compare mechanism for comparing all files in the **expected** directory with test result files having the same name, which will be sufficient for the example. Usually, evaluating test results becomes a bit more difficult, since variable information as timestamps or file locations have to be eliminated from test results and expected data.

For the simple demonstration, action patterns need not to be updated. The run action will be overloaded later in test cases.

-

Linux hints

In order to support settings for environment variables, `preprocessing` actions have to be defined as function. The **preprocessing** file is included in the framework action `testRun`. The action patterns provided with the test environment pattern are listed below:

```
// preprocessing
#!/bin/bash
# define preprocessing action when required
preprocessing() {
# define environment variables required for tests, e.g.
  export LD_LIBRARY_PATH=/usr/local/lib
}
// run
  #!/bin/bash
  source ${ROOT_DIR}/TestSuite.sh
  # run test
  # 1 - current test suite location in hierarchy
  test_suite=$1 # CHAR
  echo ${test_suite} started : $(date +'%Y/%m/%d %T') >>${LOGFILE_OUT}
  cd ${WORK_AREA}/data
  ${BIN_DIR}\test.exe parm1 parm2
  echo ${test_suite} finished: $(date +'%Y/%m/%d %T') >>${LOGFILE_OUT}

// postprocessing
  #!/bin/bash
  # source ${ROOT_DIR}/TestSuite.sh
  # compare results ... a typical implementation

  ${TEST_ACT}/GetErrorsFromLog
  cp -f ${WORK_AREA}/data/*.err ${TEST_RUN} 2>/dev/null
  for x in ${WORK_AREA}/expected/*; do ${TEST_ACT}/Compare $x; done

// Compare
  #!/bin/bash
  # source ${ROOT_DIR}/TestSuite.sh
  # 1 - file name (complete path) to be compared
  echo "Compare $1"
  diff -b "${WORK_AREA}/data/$(basename $1)"  "$1" >>${COMPARE_OUT} 2> $
{ERRORS_OUT} || ${TEST_ACT}/CopyFailed $1

// CopyFailed
  #!/bin/bash
  # copy files causing problems
  # 1 - file name (complete path) to be copied

  if [ ! -d ${TEST_RUN}/failed ] ; then
    mkdir ${TEST_RUN}/failed
  fi
  cp "${WORK_AREA}/data/$(basename $1)" "${TEST_RUN}/failed/$(basename
$1)"
  echo "File ${TEST_RUN}/data/$(basename $1) missing or differs from
expected result" >>${COMPARE_OUT}
```

Windows hints

The `preprocessing` action actually does nothing but has to be provided in the **actions** folder under **main_suite**. The **binary/tools** directory contains some Linux tools (**diff**, **find** etc.) for running the delivered `postprocessing` action. The action patterns provided with the test environment pattern are listed below:

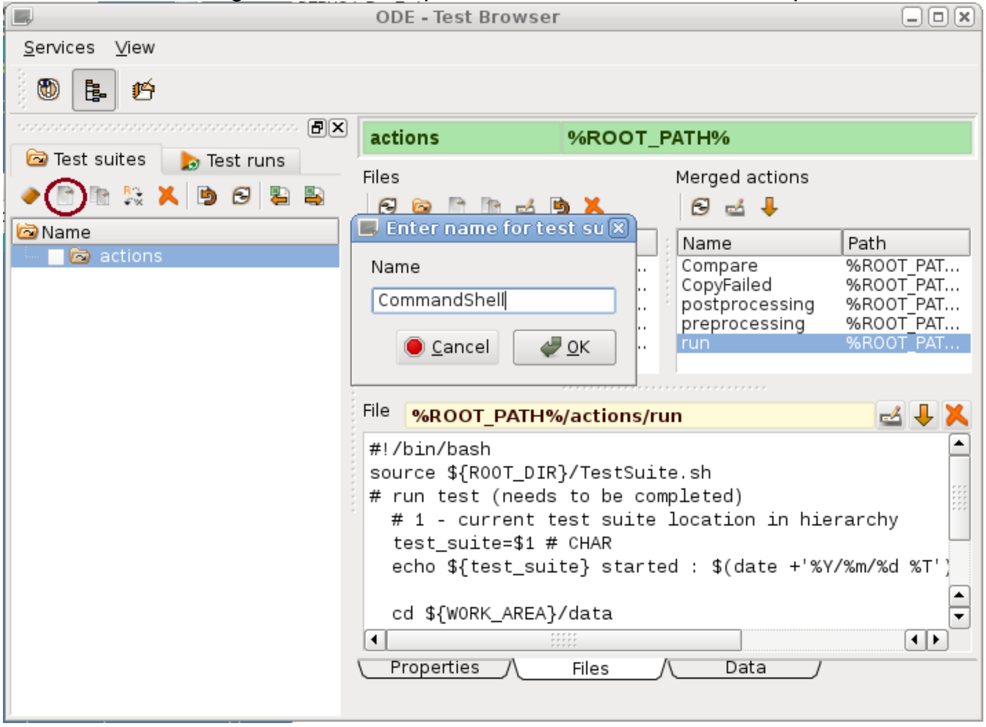
```
// preprocessing.cmd
rem run pre-processing actions
rem in order to do something, the action has to be overwritten below

// run.cmd
rem Compile table
rem 1 - current test suite location in hierarchy

echo %1 started : %Date% %Time% >>%LOGFILE_OUT%
cd /D %WORK_AREA%\data
rem include test function call like %BIN_DIR%\test.exe parm1 parm2
echo %1 finished: %Date% %Time% >>%LOGFILE_OUT%

// postprocessing.cmd
rem compare results ... a typical implementation
copy /Y %WORK_AREA%\data\*.err %TEST_RUN%\. >nul
for %%x in ( %WORK_AREA%\expected\*.* ) do call %TEST_ACT%\Compare.cmd %
%x

// Compare.cmd
rem 1 - file name (complete path) to be compared
%TOOLS_DIR%\diff "%WORK_AREA%\data\%~nx1" "%1" >>%COMPARE_OUT% 2>>
%ERRORS_OUT% || call %TEST_ACT%\CopyFailed %1

// CopyFailed.cmd
rem 1 - file name (complete path) to be compared
%TOOLS_DIR%\diff "%WORK_AREA%\data\%~nx1" "%1" >>%COMPARE_OUT% 2>>
%ERRORS_OUT% || call %TEST_ACT%\CopyFailed %1
```

-

4.1.1.2 Create new test suite

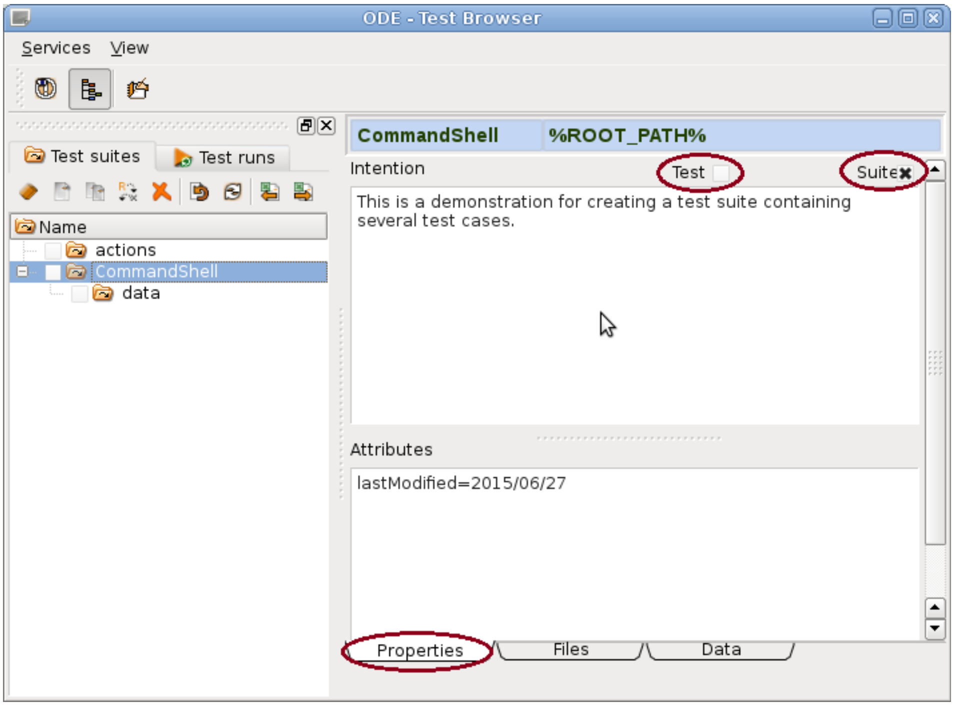
Creating a new test suite is done by clicking the action button above the test suite tree and entering the name for the new test suite. The new test suite (or test case) is inserted at the currently selected level in the tree.

After creating a new test suite, the **Properties** tab should be selected in the edit window for entering a short description for the test suite and required test suite



```
#!/bin/bash
source ${ROOT_DIR}/TestSuite.sh
# run test (needs to be completed)
  # 1 - current test suite location in hierarchy
  test_suite=$1 # CHAR
  echo ${test_suite} started : $(date +'%Y/%m/%d %T')

  cd ${WORK_AREA}/data
```

attributes:



This is a demonstration for creating a test suite containing several test cases.

lastModified=2015/06/27

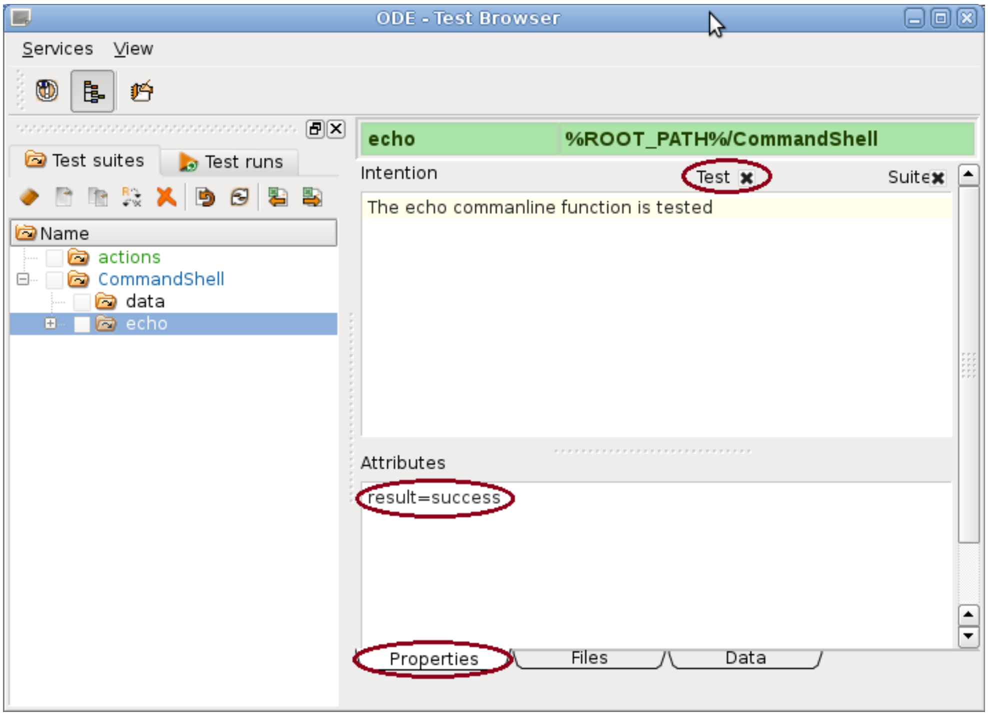
For test suites, the **Test** option must be off (is is on after creating the test suite). The **Suite** option must be on (is set after create). Activating the suite option creates a file **suite** in the test suite directory, which contains the description entered in the **Intention** field. The **Attributes** field may contain any number of attributes beginning with the name which is separated by = sign from the attribute value. Attributes are stored in a file attributes in the test suite directory. From within test browser OSI functions one may access attributes via functions **`DirEntry`**`::GetAttribute()` and **`DirEntry`**`::SetAttribute()`.

Below the test suite directory, a **data** directory has been created, which may contain common test data for the test suite. When the test suite provides common actions, one may also create an **actions** sub directory by selecting data in the tree and and using the "create a new test suite" action button above the test suite tree. After creating the **actions** directory, **Test** and **Suite** options have to be switched off.

-

4.1.1.3 Creating a new test case

In order to create a new test case for a test suite, any entry below the test suite (e.g. **data**) must be selected before clicking the action button above the test suite tree. After entering the name for the new test case the new test case is inserted:



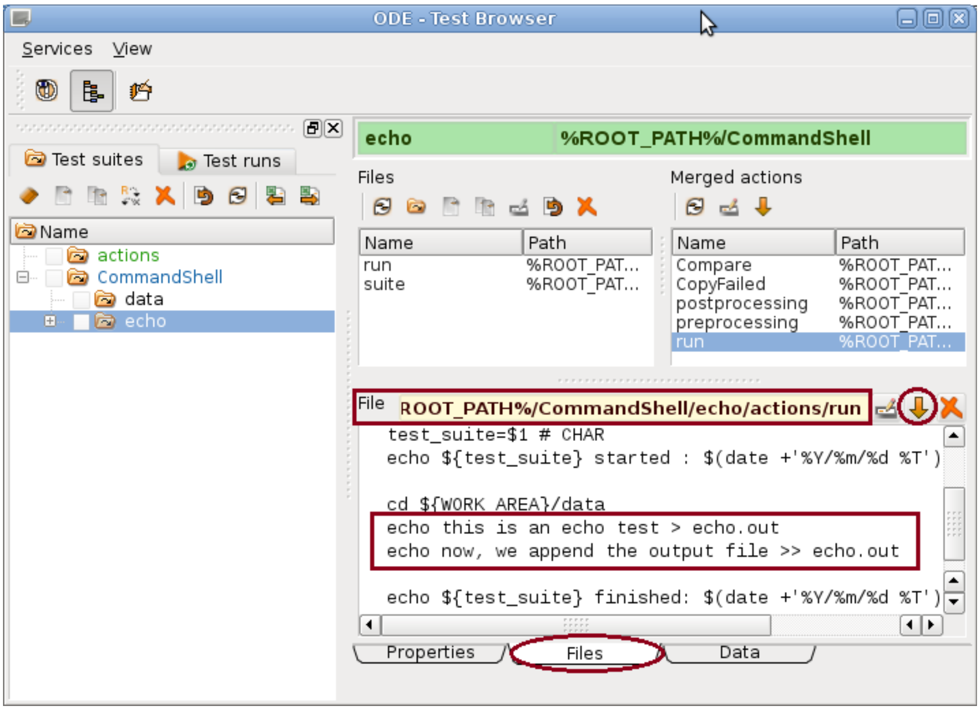
echo %ROOT_PATH%/CommandShell

The echo commanline function is tested

result=success

Test and Suite option must be switched on. After creating a new test suite, the **Properties** tab should be selected for entering a short description for the test case and required test suite attributes. The result attribute should be used to indicate, whether a successful test is expected or not.

Below the test case, a data directory has been created and a run file is stored to the test case directory in order to mark the directory as test case directory. In order to update the run action to be executes for the test case, the Files tab should be selected:



Name	Path
run	%ROOT_PAT...
suite	%ROOT_PAT...

Name	Path
Compare	%ROOT_PAT...
CopyFailed	%ROOT_PAT...
postprocessing	%ROOT_PAT...
preprocessing	%ROOT_PAT...
run	%ROOT_PAT...

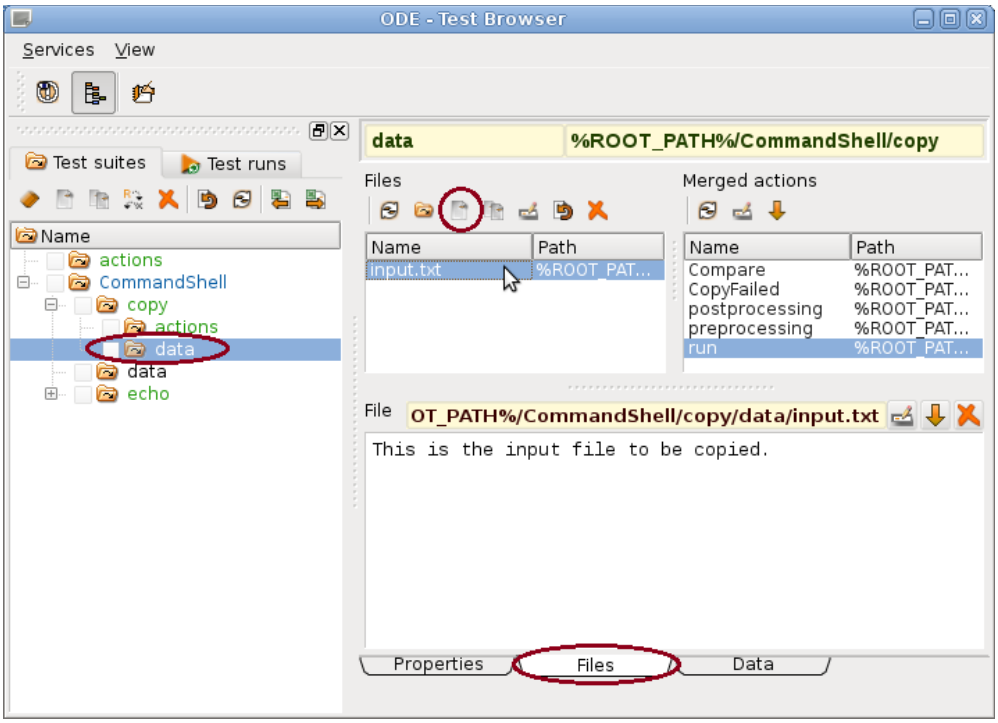
File ROOT_PATH%/CommandShell/echo/actions/run

```
test_suite=$1 # CHAR
echo ${test_suite} started : $(date +'%Y/%m/%d %T')

cd ${WORK_AREA}/data
echo this is an echo test > echo.out
echo now, we append the output file >> echo.out

echo ${test_suite} finished: $(date +'%Y/%m/%d %T')
```

The **Merged actions** list contains all actions inherited from upper test suites (in this case from the **main_suite**). When changing data as inserting the two echo lines in the example, the action will be localized automatically, i.e. the updated action will be stored in the **actions** directory for the selected test case. This is also the case for **Merged data** and **Merged expected** lists shown when selecting the **Data** tab.

The same way, the second test case for testing the copy function may be created. The difference is, that the copy test requires input data to be copied. Input data can be defined directly in the file system but also after selecting the data directory below the copy test case:

-



After updating (and localizing) the run action, the data directory below the copy test case directory has to be selected. The "Create new file" action button above the files list may be clicked. After entering the file name and confirming the action, the file appears in the list. In case of text files it may be edited directly in the edit filed below the file path. More complex structured test data may be edited using the edit button left of the file path. In order to copy test data from another location, one may also use the file dialog button above the **Files** list.

Notes: When switching off the suite option, the suite file will be deleted and the test intention will disappear.

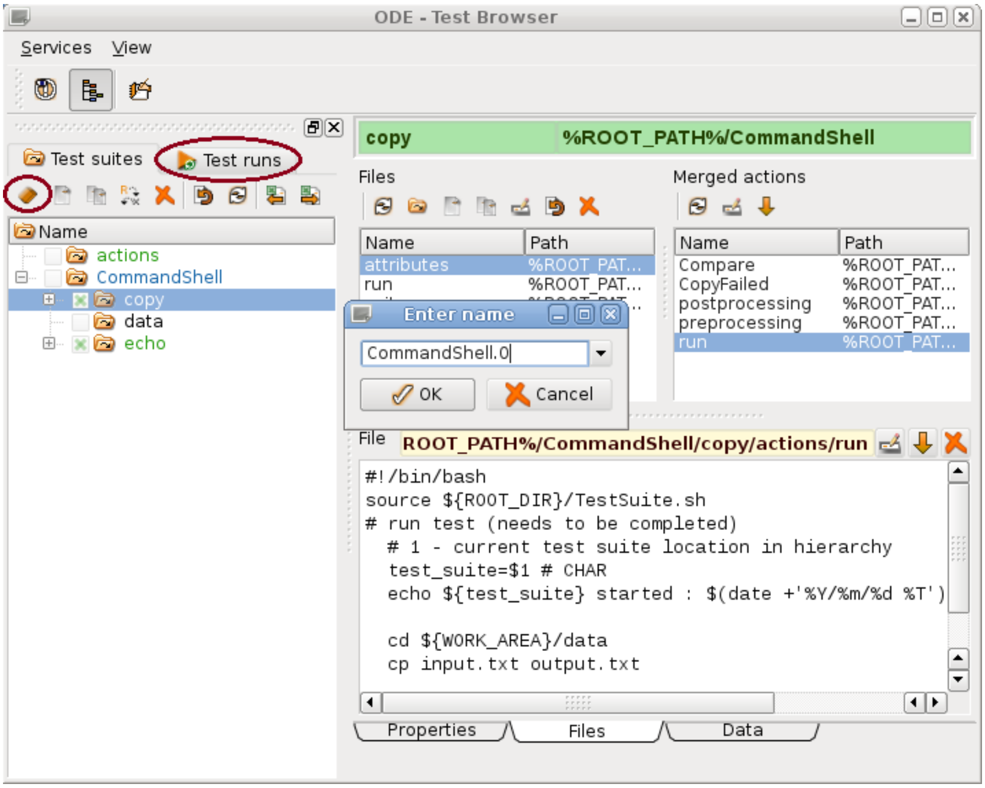
4.1.1.4 Create expected data

Expected data are one or more files, that contain the expected test results. All files in the expected directory (Merged expected list) will be compared after running the test with files having the same name in **work_area/data**. Theoretically, expected data should be created before running the test, but practically, this becomes nearly impossible in many cases and later we will see a more practical way for creating expected data. Nevertheless, there is a way to create expected data files within **TestBrowser** or within the file system. Expected data for a test case has to be added to the expected directory below the test case. Usually, expected data cannot be shared and will be provided for each test case separately. In general, however, expected data could also be provides for a number of test cases in a common parent test suite. When no expected directory has been created so far, the expected directory has to be created. Select an entry below the expanded test case (e.g. data) and click

4.1.1.5 Create a test run

After defining a number of test cases, one may create a test run. A test run refers to all test cases to be tested in the test run. A test run gets a name, which is requested when creating the test run. Before creating the test run, all test cases to be added to the test run have to be checked in the test suite tree. When checking a test suite, all test cases below the test suite will be added to the test run.

-



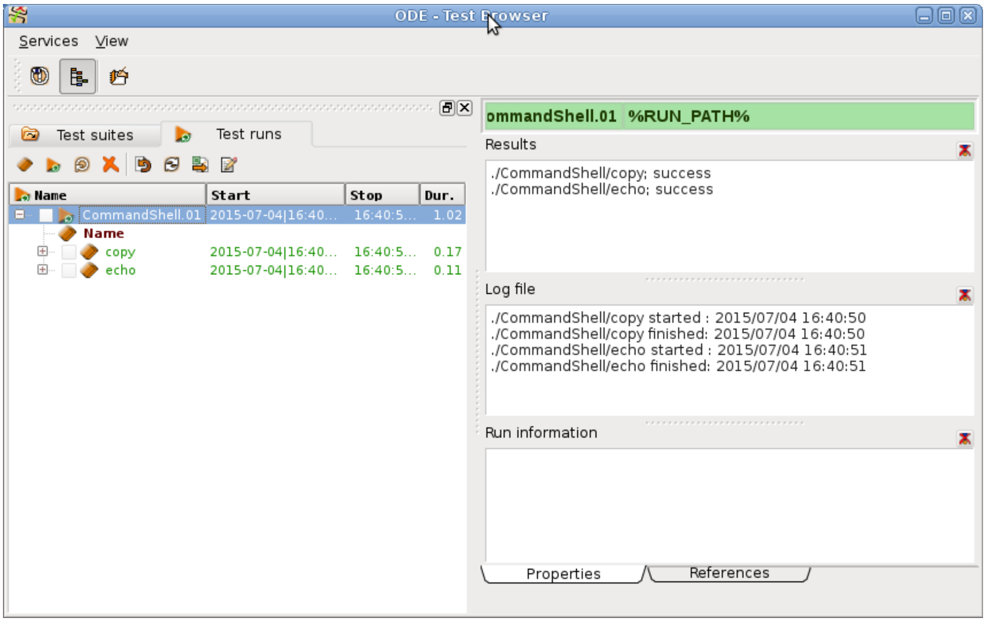
```
#!/bin/bash
source ${ROOT_DIR}/TestSuite.sh
# run test (needs to be completed)
  # 1 - current test suite location in hierarchy
  test_suite=$1 # CHAR
  echo ${test_suite} started : $(date +'%Y/%m/%d %T')

  cd ${WORK_AREA}/data
  cp input.txt output.txt
```

Since any number of test runs may be created for the same set of test cases, test runs should get a number or any other indicator after the name referring to the test run content. After entering and confirming the test run name, the "Create a new test run" action has to be clicked in the action button list above the test suites tree. Now, the test run may be executed by selecting the Test runs tab.

After a test run has been created, it may be expanded by selecting any number of test cases and/or test suites by activating corresponding check boxes in the tree. Clicking on the "Create a new test run" button above the tree and selecting an existing test run from the drop list in the dialog (instead of entering a new name) will expand the selected test run. When test suites have been selected, all test cases defined for the test suite and all subordinated test suites are added to the test run.

4.1.1.6 Execute test run

In order to run a single test run, i.e. all test cases for the test run, one has to select the test run in the tree and click the run button above the list.



```
Name              Start                Stop       Dur.
CommandShell.01   2015-07-04|16:40...  16:40:5... 1.02
  Name
  copy            2015-07-04|16:40...  16:40:5... 0.17
  echo            2015-07-04|16:40...  16:40:5... 0.11
```

ommandShell.01 %RUN_PATH%

Results

```
./CommandShell/copy; success
./CommandShell/echo; success
```

Log file

```
./CommandShell/copy started : 2015/07/04 16:40:50
./CommandShell/copy finished: 2015/07/04 16:40:50
./CommandShell/echo started : 2015/07/04 16:40:51
./CommandShell/echo finished: 2015/07/04 16:40:51
```

Run information

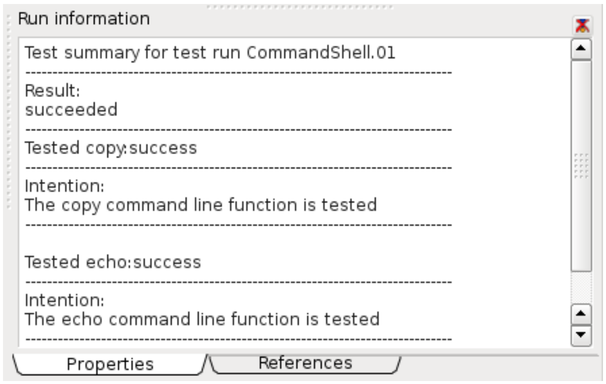
One may also select several test runs by activating check boxes besides the test nun name. In order to run a single test or several single tests, tests have to be selected by clicking the check boxes beside the tests (green entries).

After running the test, tests successfully executed are displayed with green color. Tests failed become red. On the right side test results are listed for all executed tests and start/stop statistics are shown in the log file box as well as in the tree. Log file statistics are created by the command shell test framework while tree statistics are created by **TestBrowser**.

Before rerunning a test one may clear log file and result data by clicking the "clear file content" button above the text field.

In order to get test summary information, one may use the "Create test summary ..." button above the list.

-


Run information

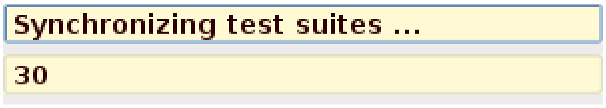
```
Test summary for test run CommandShell.01
--------------------------------------------------------------------------
Result:
succeeded
--------------------------------------------------------------------------
Tested copy:success
--------------------------------------------------------------------------
Intention:
The copy command line function is tested
--------------------------------------------------------------------------

Tested echo:success
--------------------------------------------------------------------------
Intention:
The echo command line function is tested
--------------------------------------------------------------------------
```

The layout for test run summaries may be overloaded by an external OSI function **`TestRun`**`::Protocol(TextFile &`*`file`*`)`. The file created is stored in the file **TestSummary.txt** in the test run directory below **TestRun/test_runs**.

4.1.2 Rebuild test database from directories

The test database is just a mean for managing test resources in a more comfortable way. The database can always be rebuild from information stored in directory and several files. Thus, one may also define test cases in the file system and starting **TestBrowser** later.

When no test database exists when starting **TestBrowser**, it will be created automatically by reading test suites, test cases and test runs from directory structure and connecting test suites with test runs. While rebuilding the database, a progress window shows the number of directories processed.


Synchronizing test suites ...

30

For large databases with several thousands tests this may take several minutes. Hence, it is better to run test databases by themes rather than having everything in one test database. This makes it also easier to manage test environments in a sub versioning system as SVN or GIT.

More details about **TestBrowser** functionality is provided in chapter "Action Reference".

4.2 Action reference

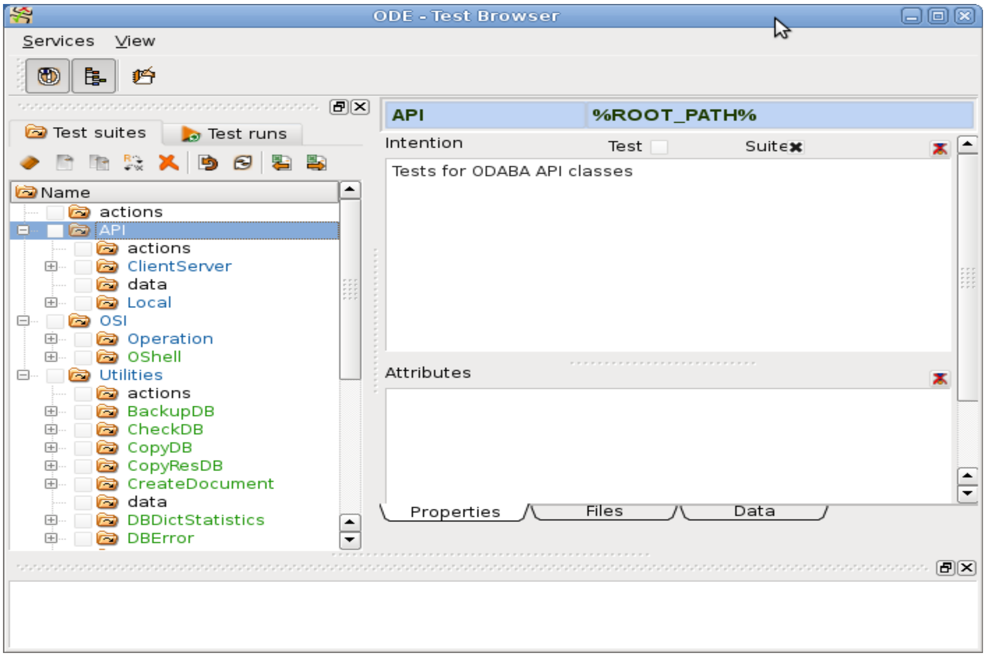
The action reference guide describes actions supported by **TestBrowser**. The chapter describes different actions supported by **TestBrowser** ordered by themes

As example, we use the ODABA release test database, which provides tests foe each new release.

4.2.1 Running ODABA tests

In order to view or run ODABA tests, which are delivered in file **ODABATest** (**.zip** or **.tar.bz2**), one may start **TestBrowser** from the installed test root directory (**TestBrowser.sh** under Linux and **TestBrowser.cmd** under Windows).

Usually, the database is available and the **TestBrowser** starts immediately. When no database has been provided (or when it has been deleted), a new database will



Tests for ODABA API classes

be created. While importing test environment data from test suites and test runs a progress window appears displaying the number of directories already processed (about 300 test suite and 500 test run directories).

The tree shows three test suites (**API**, **OSI** and **Utilities**). Test suites are displayed with blue letters. The action directory on top contains test environment specific actions. Action directories and other directories not marked as test suite or test case are displayed with black letters. After expanding the tree test suites, several subordinated test suites and test cases are displayed. Test cases are displayed with green letters (e.g. **OShell** or **CheckDB**).

4.2.2 Main menu and main toolbar

There are just a few actions provided in via main menu or main toolbar, which mainly refer to application layout.

4.2.2.1 Show/hide message area

The toggle button allows showing or hiding the message area. Usually, the message area is shown automatically, when a message is written to the area. This, however, depends on the applications and sometimes, the output area has to be activated explicitly.

4.2.2.2 Show/hide selection tree

The action allows hiding and showing the main tree. Usually, the main tree is visible when starting the application. When the application has been closed with hidden selection tree, the tree is also hidden, when restarting the application.

4.2.2.3 Edit common and user-defined settings

The action opens the option dialog, which allows updating or creating option values for the application. Options may be defined as common and as user-defined options. When starting the applications, COMMON options are loaded first and might be overwritten by user-defined (or default) **option** settings.

Options may be referred to in OSI actions (`Option` class) and in application designer. Since options are stored within the database, they will get lost, when recreating the database. In order to reuse options updated or created later on, those have to be exported to an ini- or configuration (xml) file when being changed.

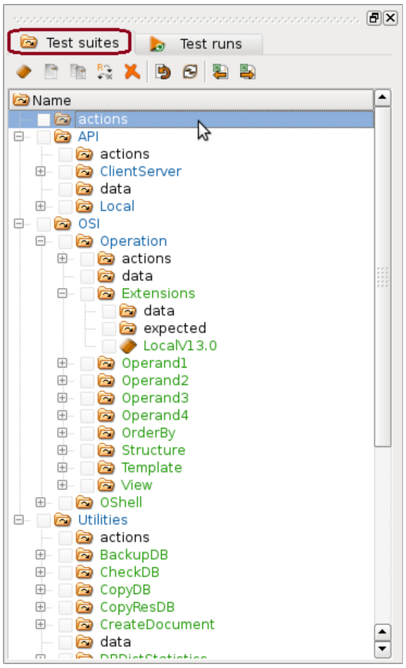
4.2.2.4 Exit application

The action terminates the application. Changes made in the application are stored automatically.

4.2.3 Test suite tree

The test suite tree is a directory tree that contains entries for (hierarchical) test suites and subordinated test cases. Besides test suites and test cases, other directories not identified as test suite or test case are displayed.

The tree shows test cases with green letters and test suites with blue letters. Other directories are displayed with black letters. When test cases are associated with est runs, test runs are listed at the end of the subordinated directory list using the test run icon instead of the directory icon.



Except test runs the tree exactly reflects the directory structure in the file system below the test root directory.

Usually, each test root directory contains an action directory with default actions for `preprocessing`, `run` and `postprocessing` actions.

Since test cases require a test suite parent, the top level in the tree must not contain test cases. The application allows test cases on top level, but the command line test framework may get problems.

The parent of a test case or test suite (except top test suites) should be a test suite. Inheriting data, expected and actions stops when a parent is not a test suite.

Available actions for the tree are shown in the toolbar above the test suite tree. Default actions are provided via context menu (right mouse click on the list. Actions are described in following topics. Actions are listed according to toolbar button sequence from left to right.

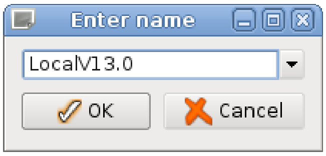
Usually, the test suite tree automatically updates as soon as something has changed (e.g. new directories (**expected**, **actions** created because of marking a test suite as test case). When tree has not been refreshed, it may be refreshed explicitly via context menu action `Refresh`.

-

4.2.3.1 Create or extend test run

The action creates or extends a test run with test cases selected from the test suite tree. Before starting the action, test cases for which run entries (tests) are to be created should be marked by activating the check box. Activating the check box for a test suite will include all subordinated test cases as well as all test cases of subordinated test suites.

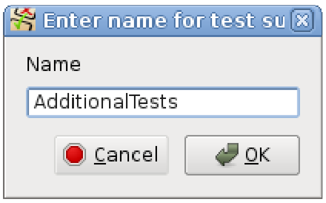
The action starts with a dialog requesting the name for the test run. One may enter a new test run name or select a test run from the drop list of the name field. When an existing test run has been selected, the selected test run will be extended. Selected test cases that have got already run entries in the selected test run are ignored.



4.2.3.2 Create test environment directory

The action creates a new test environment directory in the tree. In order to select the region in which the entry should be created, one existing entry in the region has to be selected. The type of the directory created (test suite, test case, other) also depends on the selected tree entry. Nevertheless, the tape may be changed later.

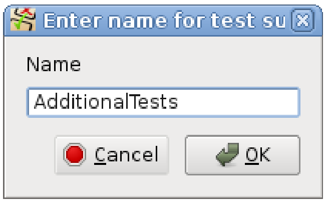
The action pops up with a name dialog for entering the directory name. After entering the name and confirming, the new directory is created in database and file system. In case of test suites, also a **data** sub directory is provided. For test cases, an **actions** and **expected** directory will be created in addition. All directories area created in the database as well as in the file system.



4.2.3.3 Copy test environment directory

The action copies the entry selected in the test case tree within the current region. Copying the entry includes copying all subordinated database entry. The action also copies the file system directory and all its subordinated files and directories.

The action pops up with a name dialog for entering the new directory name. After entering the name and confirming, the new directory is created. Names for subordinated entries and directories remain unchanged.



Name
AdditionalTests

In order copy or move a test directory to another region in the tree, one may use file system functions and reload the complete test suite tree or parts of it.

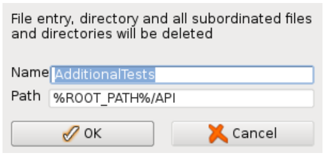
4.2.3.4 Rename test environment directory

The action allows changing the name for a test suite directory in the database and in the file system. Renaming default directories data, actions and expected will prevent renamed directories from hierarchical inheritance.

4.2.3.5 Delete test environment directory

The action deletes all selected test environment directories (check box activated in the test suite tree) and all its subordinated directories and files in the database as well as in the file system. When no check box is activated in the list, the currently selected entry is deleted, i.e. in order to delete a single test environment directory the corresponding line has to be selected, only, before clicking the button. By default, deletion happens immediately without warning.

In order to obtain a warning dialog before deleting a selected entry, the option variable ASK_BEFORE_DELETE has to be set to **true** (**TestBrowser** ini- or configuration file). Then, a deletion confirmation is required by popping up a deletion dialog.



File entry, directory and all subordinated files and directories will be deleted

Name AdditionalTests
Path %ROOT_PATH%/API

4.2.3.6 Update test environment directory status

The test environment directory status (test case, test suite) depends on existence of files **test** and **suite** in the test environment directory. In case of **test** file exists, the directory is supposed to define a test case. In case of **suite** file exists (but no **test** file), the directory is considered to be a test suite directory. When **test** and **suite** files do not exist, the directory is considered to be any other test environment directory.

The test environment directory status is reflected in database directory entries in attributes test and suite. When the files have been changed in the file system, the status in the database is automatically updated only for entries selected in the tree. In order to update directory status for all entries, this action may be called.

-

After updating directory status, `Refresh` action should be called via context menu for updating tree colors.

4.2.3.7 Reload test environment directory structure

The action may be called in order to synchronize file system and database directory structure. The file system has higher priority, i.e. all files and directories that do not exist anymore in the file system are also deleted in the database. Files and directories that exist in the file system but not in the database will be added to the database.

Updating the complete database may take a while. A progress window shows the number of directory entries already processed. In order to reload part of the file system below a selected directory, the `Reload` action from test suite tree context menu may be used.

4.2.3.8 Import test environment from CSV file

The action allows importing test suites and test cases from a CSV file. The action prompts for an output file name and imports test suites and test cases from the selected file, which is assumed to be a CSV (tab or semicolon separated) file.

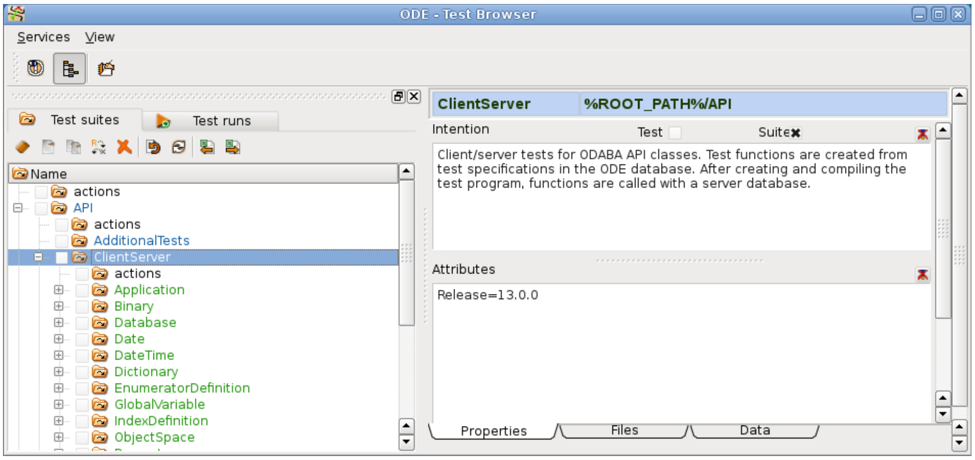
The import logic depends very much on specific requirements, i.e. it makes a big difference importing e.g. requirements from Doors in order to create test suites for a requirement driven test environment or importing test cases from another test environment. Hence, the import logic has to be implemented by customer in **`DirEntry`**`::ImportFromCSV` function (see programmer's reference).

4.2.3.9 Export test environment to CSV file

The action allows exporting test suites and test cases to a CSV (semicolon separated) text file. The action prompts for an output file name and exports test suites and test cases to the selected file.

The export logic depends very much on specific requirements. Hence, the export logic has to be implemented by customer in **`DirEntry`**`::ExportToCSV` function (see programmer's reference). Programming examples are provided.

4.2.4 Edit test suite, test case and other directories



Client/server tests for ODABA API classes. Test functions are created from test specifications in the ODE database. After creating and compiling the test program, functions are called with a server database.

Release=13.0.0

When selecting a test directory in the test suite tree, relevant data from the selected test environment directory are displayed in the edit window on right side:

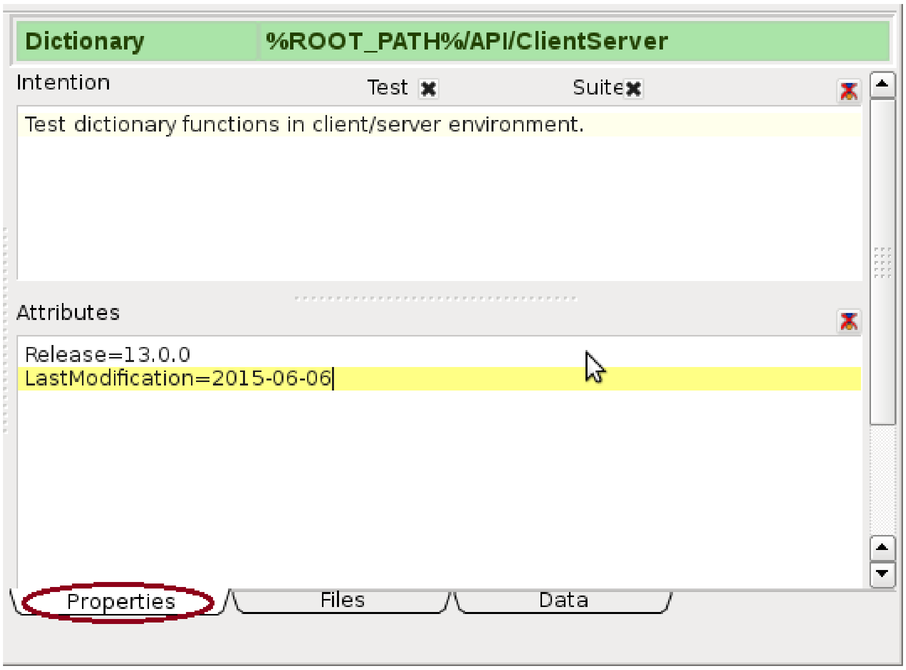
The headline shows the directory name and the path relative to the test root directory (ROOT_PATH environment or option variable). Depending on directory type the title background is blue (test suite), green (test case) or yellow (other directory).

Below the edit window are three tabs for selecting different views, which are explained in subsequent topics.

-

4.2.4.1 Edit test environment directory properties

The properties window displays the test environment directory properties.


Dictionary %ROOT_PATH%/API/ClientServer

Intention Test ✖ Suite ✖

Test dictionary functions in client/server environment.

Attributes

Release=13.0.0
LastModification=2015-06-06

For test suites, he test **Intention** describes the area or requirement covered by the suite. For test cases, he test intention describes test conditions and expectations. Since the data in this field is stored in the suite file for the directory, intention can be entered for directories marked as suite, only. When the directory is not yet marked as suite, it becomes a test suite directory automatically after entering an intention text.

The "Clear file content" button above the **Intention** field can be used for clearing the complete text field. It does not delete the file but stores an empty file.

The **Test** check box indicates, whether the directory describes a test case or not. Test case directories contain a **test** file, which is usually empty. When activating the check box, the file will be created. It will be deleted, when deactivating the field.

The Suite check box marks a test suite directory. Test suite directories contain a **suite** file that contains the description for the test intention. When activating the check box, the file will be created. It will be deleted, when deactivating the field. In order to store intention description for test cases, test case directories are marked often as test suite, too.

The **Attributes** field contains extension attributes for the directory, which are stored in the **attributes** file of the directory. The attributes file is created automatically, when entering data in the field. Attributes have to be defined in the form

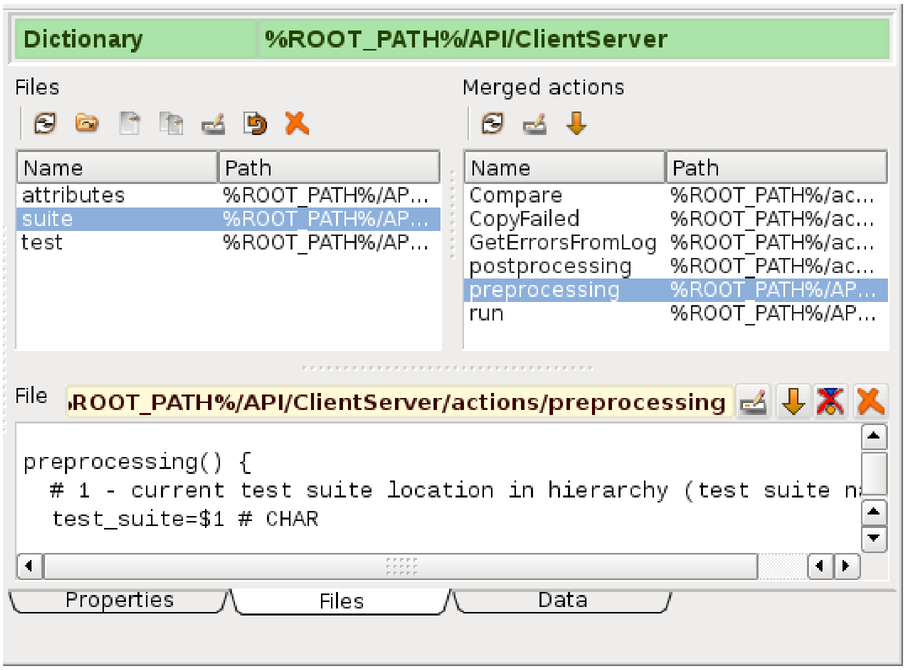
```
attribute_name=value
```

The value must not contain line break. Spaces before and after attribute_name are considered as part of the name. Attributes can be accessed from within OSI functions by calling **DirEntry**::GetAttribute() and **DirEntry**::SetAttribute().

The "Clear file content" button above the **Attributes** field can be used for removing all attributes from the **attributes** file without deleting the file.

4.2.4.2 Edit test environment actions and directory files

The files window displays directory files and test suite/case actions. In the upper part there are two file lists. The lower part provides the content for the last file selected in one of the lists (`preprocessing` action in the image below).


Dictionary
%ROOT_PATH%/API/ClientServer
Files
Merged actions
File
ROOT_PATH%/API/ClientServer/actions/preprocessing
Properties
Files
Data

Name	Path
attributes	%ROOT_PATH%/AP...
suite	%ROOT_PATH%/AP...
test	%ROOT_PATH%/AP...

Name	Path
Compare	%ROOT_PATH%/ac...
CopyFailed	%ROOT_PATH%/ac...
GetErrorsFromLog	%ROOT_PATH%/ac...
postprocessing	%ROOT_PATH%/ac...
preprocessing	%ROOT_PATH%/AP...
run	%ROOT_PATH%/AP...

```
preprocessing() {
  # 1 - current test suite location in hierarchy (test suite n
  test_suite=$1 # CHAR
```

The **Files** list displays the files stored in the directory. File lists support several actions via action buttons in the toolbar above the list, which are described in the following subtopic "File list actions".

The **Merged actions** list contains the actions that are called when running a test. All files contained in **actions** sub directory for the current and all parent directories are collected. Actions in lower **actions** directories get higher priority, i.e. those will

-

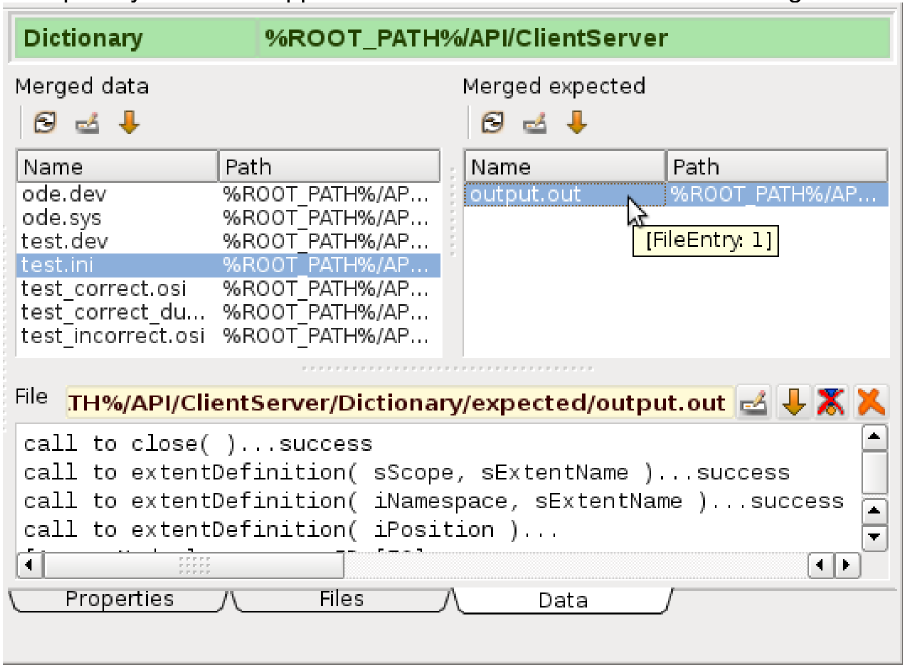
overwrite actions in higher **actions** directories. The current **actions** directory has got highest priority. Actions supported for the list are described in "Merged actions list actions".

The **File** text field displays the file content. The file content edit field is described in a separate chapter "Edit file content".

4.2.4.3 Edit test environment data

The files window displays merged test data and expected files. In the upper part there are two file lists. The lower part provides the content for the last file selected in one of the lists above (expected **output.out** in the image below).

The **Merged data** list displays the files that are used when running the test. All files contained in **data** sub directory for the current and all parent directories are collected. Data files in lower **data** directories get higher priority, i.e. those will overwrite data files in higher **data** directories. The current **data** directory has got highest priority. Actions supported for the list are described in "Merged data list


Dictionary
%ROOT_PATH%/API/ClientServer
Merged data
Merged expected
File
TH%/API/ClientServer/Dictionary/expected/output.out
[FileEntry: 1]
Properties
Files
Data

Name	Path
ode.dev	%ROOT_PATH%/AP...
ode.sys	%ROOT_PATH%/AP...
test.dev	%ROOT_PATH%/AP...
test.ini	%ROOT_PATH%/AP...
test_correct.osi	%ROOT_PATH%/AP...
test_correct_du...	%ROOT_PATH%/AP...
test_incorrect.osi	%ROOT_PATH%/AP...

Name	Path
output.out	%ROOT_PATH%/AP...

```
call to close( )...success
call to extentDefinition( sScope, sExtentName )...success
call to extentDefinition( iNamespace, sExtentName )...success
call to extentDefinition( iPosition )...
```

actions".

The **Merged expected** list contains the expected files that are used for comparing test results after running the test. All data files contained in **expected** sub directory for the current and all parent directories are collected. Data files in lower **expected** directories get higher priority, i.e. those will overwrite data files in higher **expected** directories. The current **expected** directory has got highest priority. Actions supported for the list are described in "Merged expected list actions".

The **File** text field displays the file content. The file content edit field is described in a separate chapter "Edit file content".

4.2.4.3.1 File list actions

File lists show files in a directory (**Files**, **Work area**) or merged file lists containing files from all sub directories ao certain kind (data, actions, expected) in the test suite parent hierarchy. Depending on different file lists, a set of actions is supported shown in the toolbar above the file list.

Refresh list content

Sometimes the displayed list is not up to date. In order to update data displayed in the list with current database content, the action may be clicked. For merged lists the collection is re-evaluated.

Insert file

The function allows inserting a file to the selected directory. A file dialog pops up for selecting a file in the file system. After confirming, the file is copied to the selected directory and added to the file list.

Create a new file

The action creates a new file in the selected directory. In order to enter the file name, a dialog pops up. After entering the name and confirming, en empty file is created in the selected directory.

Copy file

The action copies the selected file within the selected directory. In order to enter the file name for the copy, a dialog pops up. After entering the name and confirming, a copy of the file is created in the selected directory.

Edit file

External editors may be called directly via this action. In order to call file editors via file extension, a file association has to be defined in the system. When this is not the case, an ODABA file association may be defined in the ini- or configuration file in. Finally, one may define a test browser file association by adding a section with the extension name and define a variable CALL below, that provides the external editor function call (see `FileEntry::Edit()` in "TestBrowser Programmer's Guide").

Localize selected file

Localizing the file will create a copy of the file in the appropriate sub folder (**data**, **actions** or **expected**) of the current test suite or test case directory. When the file does already exist in this directory, nothing will happen.

-

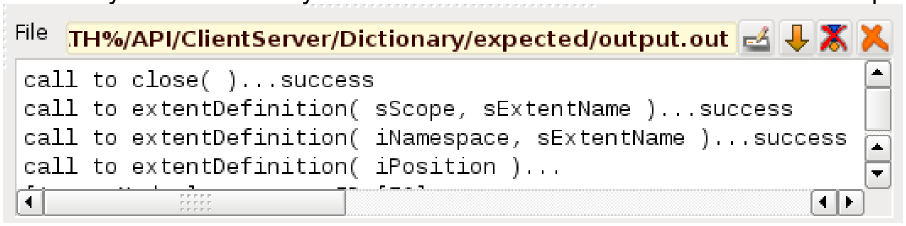
Copy file to expected directory

The action copies the selected file to the **expected** directory of the selected test suite or test case. Typically, this action is used for creating expected data files in the expected directory after running the test and expecting the result in the **work_area/data** directory (**Work area**).

4.2.5 Edit file content

File content may be displayed or updated with an internal or external file editor. The internal file editor may be used for editing text files up to 1 MB. External file editors are usually called for none text files or larger files.

For text files, the content may be edited directly in the text box below. When updating data that is not stored for the selected test suite or test case, the data will be localized automatically, i.e. the updated content is not stored in an upper test suite directory (**data**, **actions** or **extended** sub directory), but in the corresponding sub directory of the currently selected test suite or test case. In order to update



```
call to close( )...success
call to extentDefinition( sScope, sExtentName )...success
call to extentDefinition( iNamespace, sExtentName )...success
call to extentDefinition( iPosition )...
```

common data, one has to select the test suite owning the file.

The file editor supports the actions described below.

Use external editor

External editors may be called directly via this action. In order to call file editors via file extension, a file association has to be defined in the system. When this is not the case, an ODABA file association may be defined in the ini- or configuration file in. Finally, one may define a test browser file association by adding a section with the extension name and define a variable CALL below, that provides the external editor function call (see **`FileEntry`**`::Edit()` in "TestBrowser Programmer's Guide").

Localize selected file

Localizing the file will create a copy of the file in the appropriate sub folder (**data**, **actions** or **expected**) of the current test suite or test case directory. When the file does already exist in this directory, nothing will happen.

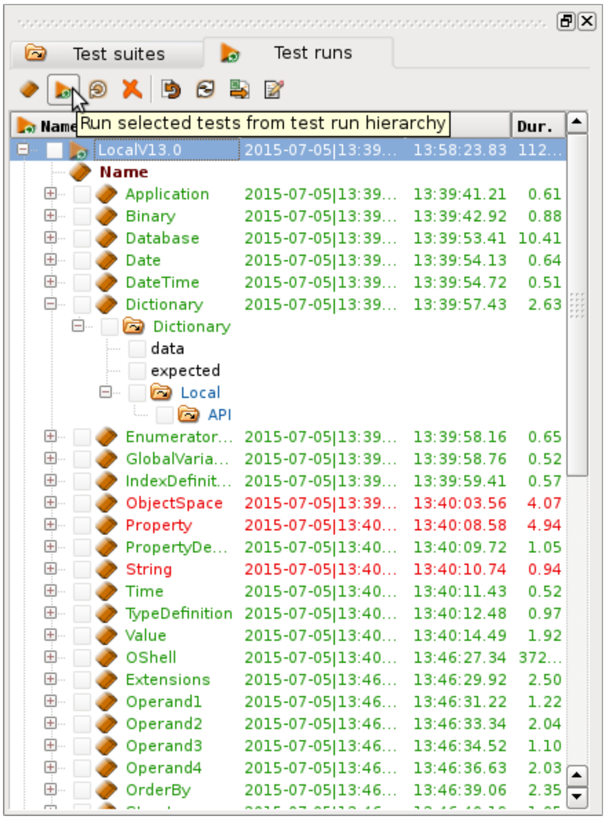
Clear file content

The action can be used for clearing the complete file content. It does not delete the file but stores an empty file. When the file is not owned by the selected test suite or test case, it will be localized before being updated.

Delete file

When the file is owned by the selected test suite or test case directory, it will be deleted. Otherwise, the action does nothing.

4.2.6 Test suite tree



Name			Dur.
LocalV13.0	2015-07-05|13:39...	13:58:23.83	112...
Name			
Application	2015-07-05|13:39...	13:39:41.21	0.61
Binary	2015-07-05|13:39...	13:39:42.92	0.88
Database	2015-07-05|13:39...	13:39:53.41	10.41
Date	2015-07-05|13:39...	13:39:54.13	0.64
DateTime	2015-07-05|13:39...	13:39:54.72	0.51
Dictionary	2015-07-05|13:39...	13:39:57.43	2.63
Dictionary			
data			
expected			
Local			
API			
Enumerator...	2015-07-05|13:39...	13:39:58.16	0.65
GlobalVaria...	2015-07-05|13:39...	13:39:58.76	0.52
IndexDefinit...	2015-07-05|13:39...	13:39:59.41	0.57
ObjectSpace	2015-07-05|13:39...	13:40:03.56	4.07
Property	2015-07-05|13:40...	13:40:08.58	4.94
PropertyDe...	2015-07-05|13:40...	13:40:09.72	1.05
String	2015-07-05|13:40...	13:40:10.74	0.94
Time	2015-07-05|13:40...	13:40:11.43	0.52
TypeDefinition	2015-07-05|13:40...	13:40:12.48	0.97
Value	2015-07-05|13:40...	13:40:14.49	1.92
OShell	2015-07-05|13:40...	13:46:27.34	372...
Extensions	2015-07-05|13:46...	13:46:29.92	2.50
Operand1	2015-07-05|13:46...	13:46:31.22	1.22
Operand2	2015-07-05|13:46...	13:46:33.34	2.04
Operand3	2015-07-05|13:46...	13:46:34.52	1.10
Operand4	2015-07-05|13:46...	13:46:36.63	2.03
OrderBy	2015-07-05|13:46...	13:46:39.06	2.35

The test run tree is a directory tree that lists test run directories on top (e.g. **LocalV13.0**) and test directories (e.g. **Dictionary**) on next lower level. Test directories (**`RunEntry`** instances) are arranged in a similar hierarchy as test suites and test cases, but below the **test_runs** directory. In the list, tests are displayed on same level in order to increase readability.

As long as test runs are not executed, tests are displayed with blue characters and run time statistics (start, stop, duration) are empty. After executing a test run or a single test the line becomes green, when test succeeded and red otherwise. Time statistics are filled. For better traceability, below each test the corresponding test case (here **Dictionary**, too) is displayed. This allows viewing test case properties (description, attributes etc.)

Below the test case, the parent test suite (e.g. **Local**) is displayed, that provides the requirement tested with the test case or in general, the test intend. Below the parent test suite, the parent's parent test suite (e.g. **API**) is shown etc., i.e. the tree shows the inverse test suite hierarchy. This directory hierarchy contains all information that may influence the test, i.e. data, actions and expected results

Selecting one of the tree entries will show entry type specific properties on the right side property window. This also displays the path to the selected directory within the file system.

Tests that cannot be expanded cannot be executed, since test data and actions defined in associated test case and parent test suites is not available. Usually, when removing a test case, all associated tests are removed as well. In case that unlinked tests appear in the list, those should be removed.

Available actions for the tree are shown in the toolbar above the test run tree. Default actions are provided via context menu (right mouse click on the list. Actions are described in following topics. Actions are listed according to toolbar button sequence from left to right.

Usually, the test suite tree automatically updates as soon as something has changed (e.g. new tests created via test suite tree). When tree has not been refreshed, it may be refreshed explicitly via context menu action `Refresh`.

4.2.6.1 Create a test run

The action creates a test run without tests. The action starts with a dialog requesting the name for the test run. After entering a new test run name and confirming, the test run will be created.

4.2.6.2 Run selected tests from test run tree

The action executes all selected tests (check box activated) in the test run tree. When test runs are checked, all tests defined in the test run are executed. When no check box is activated in the list, the currently selected entry is executed, i.e. in order to run a single test or execute a single test run, the corresponding line has to be selected, only, before starting the action.

4.2.6.3 Provide test data in work area

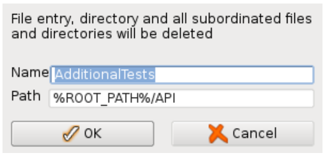
The action restores test data in the work area. When test execution failed, it might be necessary repeating the test by running a debugger. In this case, test data must be provided as at the beginning of test execution (since work area data may change during test).

Restoring test data is a feature of the command line test framework (global action `SetupWorkArea`), which is called when executing this action.

4.2.6.4 Delete selected tests from tree hierarchy

The action deletes all selected tests (check box activated) in the test run tree. When test runs are checked, all tests defined in the test run are deleted. When no check box is activated in the list, the currently selected entry is deleted, i.e. in order to delete a single test or a single test run, the corresponding line has to be selected, only, before clicking the button.

In order to obtain a warning dialog before deleting each selected tree entry, the option variable ASK_BEFORE_DELETE has to be set to **true** (**TestBrowser** ini- or configuration file). Than, a deletion confirmation is required by popping up a deletion dialog.


File entry, directory and all subordinated files and directories will be deleted

Name AdditionalTests

Path %ROOT_PATH%/API

4.2.6.5 Update test and test run status

The test run and tests status depends on existence of files **compare.out** and **errors.out** in the test directory. Since the content of this files may have changed (e.g. because of external test execution), the test and test run state may be not up to date. The consequence is a possible incorrect color when displaying the tests and test runs in the tree. This action may be called in order to update the test and test run status.

4.2.6.6 Reload test runs and tests

The action may be called in order to synchronize file system and database directory structure. The file system has higher priority, i.e. all files and directories that do not exist anymore in the file system are also deleted in the database. Files and directories, that exist in the file system but not in the database will be added to the database. Updating tests also includes providing the link to the associated test case.

Updating the complete database may take a while. A progress window shows the number of directory entries already processed. In order to reload part of the file system below a selected directory, the `Reload` action from test run tree context menu may be used.

4.2.6.7 Export test environment to CSV file

The action allows exporting test runs and tests a csv (semicolon separated) text file. The action prompts for an output file name and exports test runs and tests to the selected file.

The export logic depends very much on specific requirements. Hence, the export logic has to be implemented by customer in **`TestRun`**`::ExportToCSV` function (see programmer's reference). Programming examples are provided.

-

4.2.6.8 Create test summary for selected test run

The action allows creating a test summary text file for the selected test run. The test summary is written to the test run directory (**.../test_runs/selected_test_run/TestSummary.txt**). The content of the generated file is displayed in the edit window in the **Test summary** field.

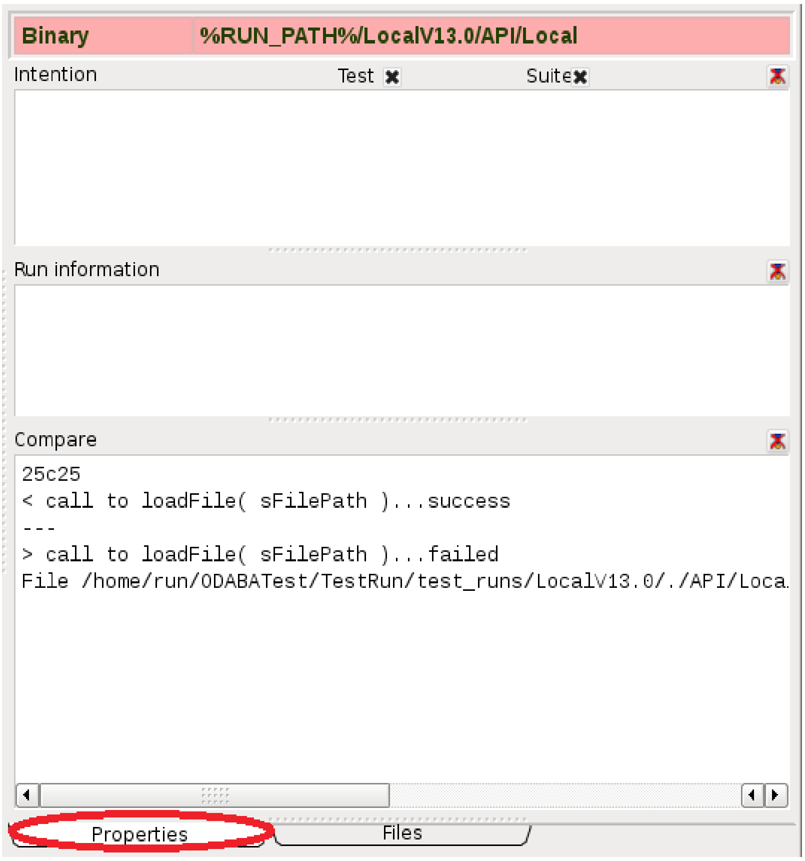
The layout and content of the test summary depends very much on specific requirements. Hence, the test summary logic has to be implemented by customer in `TestRun::Protocol` function (see programmer's reference). Programming examples are provided.

4.2.7 Run entry

For each test case, several run entries may be executed (e.g. for each new release). Results for each executed test are stored for the run entry. The run entry refers to a number of text files stored in the run entry directory. Data for a run entry may be displayed in the **Properties** of **Files** view for the run entry.

4.2.7.1 Run entry Properties view

Run entry properties show the results for a single test. One may also provide test run specific information for the test.



```
25c25
< call to loadFile( sFilePath )...success
---
> call to loadFile( sFilePath )...failed
File /home/run/ODABATest/TestRun/test_runs/LocalV13.0/./API/Loca
```

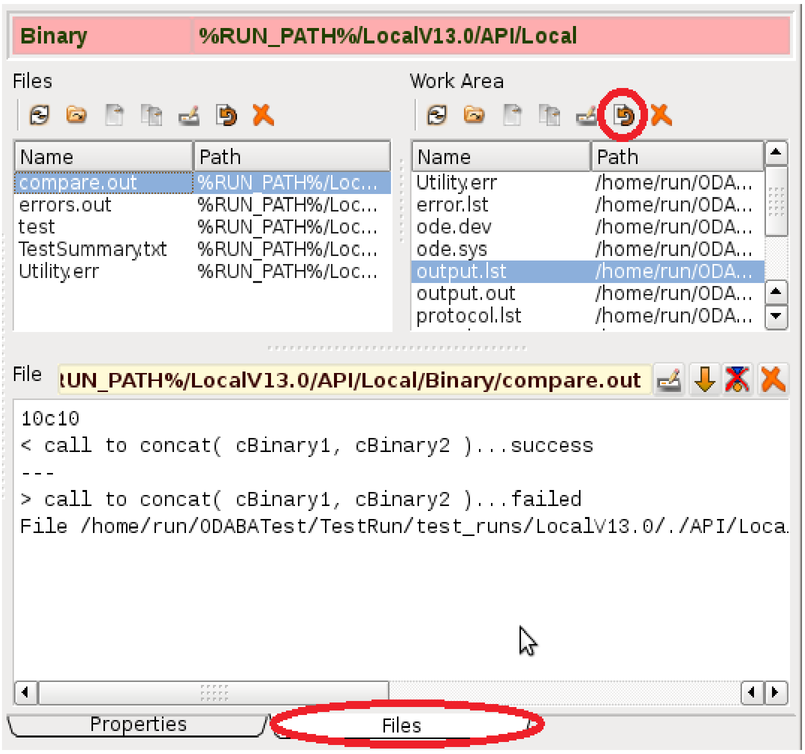
Data shown in the fields is the content of several files stored for the test run entry. **Intention**: The test intend is taken from the test case and should not be changed here (test). In **Run information** (**TestSummary.txt**) one may add special events happened during test. The test result is obtained by comparing expected data and data created by test. Differences detected by calling `diff` are shown in the **Compare** area (**compare.out**).

Data in all fields may be edited. In order to clear one of the fields, the clear button right above the field may be used. Clearing the content of the field also means clearing the content of the file displayed in the field.

-

4.2.7.2 Run entry Files view

The **Files** view for a run entry shows the files stored in the run entry directory.



Name	Path
compare.out	%RUN_PATH%/Loc...
errors.out	%RUN_PATH%/Loc...
test	%RUN_PATH%/Loc...
TestSummary.txt	%RUN_PATH%/Loc...
Utility.err	%RUN_PATH%/Loc...

Name	Path
Utility.err	/home/run/ODA...
error.lst	/home/run/ODA...
ode.dev	/home/run/ODA...
ode.sys	/home/run/ODA...
output.lst	/home/run/ODA...
output.out	/home/run/ODA...
protocol.lst	/home/run/ODA...

File ŁUN_PATH%/LocalV13.0/API/Local/Binary/compare.out

```
10c10
< call to concat( cBinary1, cBinary2 )...success
---
> call to concat( cBinary1, cBinary2 )...failed
File /home/run/ODABATest/TestRun/test_runs/LocalV13.0/./API/Loca
```

The **Files** view shows two file lists. **Files** shows the files stored in the run entry directory. The work area file list shows the files from last test (input and output files). The list will be refreshed when running the next test. When selecting a file entry in one of the lists, the file content is displayed in the file edit area below the lists.

When a test failed to run (red title), the reason is either a program error or invalid test data. In case of invalid test data, the result in the work area has to be checked. When the result corresponds to actual expectations, the new result file may be copied to the test suite expected directory by clicking the "copy to expected" button above the work area file list.

4.2.7.2.1 File list actions

File lists show files in a directory (**Files**, **Work area**) or merged file lists containing files from all sub directories ao certain kind (data, actions, expected) in the test suite parent hierarchy. Depending on different file lists, a set of actions is supported shown in the toolbar above the file list.

Refresh list content

Sometimes the displayed list is not up to date. In order to update data displayed in the list with current database content, the action may be clicked. For merged lists the collection is re-evaluated.

Insert file

The function allows inserting a file to the selected directory. A file dialog pops up for selecting a file in the file system. After confirming, the file is copied to the selected directory and added to the file list.

Create a new file

The action creates a new file in the selected directory. In order to enter the file name, a dialog pops up. After entering the name and confirming, en empty file is created in the selected directory.

Copy file

The action copies the selected file within the selected directory. In order to enter the file name for the copy, a dialog pops up. After entering the name and confirming, a copy of the file is created in the selected directory.

Edit file

External editors may be called directly via this action. In order to call file editors via file extension, a file association has to be defined in the system. When this is not the case, an ODABA file association may be defined in the ini- or configuration file in. Finally, one may define a test browser file association by adding a section with the extension name and define a variable CALL below, that provides the external editor function call (see `FileEntry::Edit()` in "TestBrowser Programmer's Guide").

Localize selected file

Localizing the file will create a copy of the file in the appropriate sub folder (**data**, **actions** or **expected**) of the current test suite or test case directory. When the file does already exist in this directory, nothing will happen.

Copy file to expected directory

The action copies the selected file to the **expected** directory of the selected test suite or test case. Typically, this action is used for creating expected data files in

-

the expected directory after running the test and expecting the result in the **work_area/data** directory (**Work area**).

Delete file

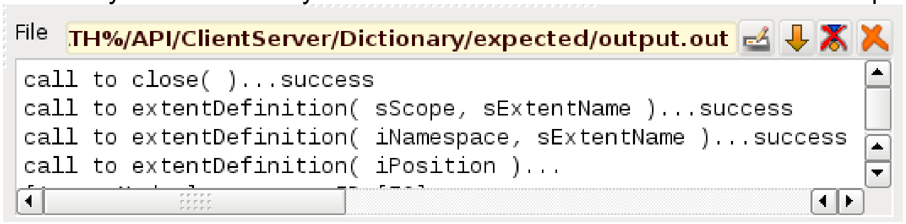
When the file is owned by the selected test suite or test case directory (merged list) or when the file is a file of the directory, it will be deleted. Otherwise, the action does nothing.

When deleting a file displayed via a merged list, the localized version is deleted and a corresponding file owned by the next available file from higher parent test suite will be displayed.

4.2.7.2.2 Edit file content

File content may be displayed or updated with an internal or external file editor. The internal file editor may be used for editing text files up to 1 MB. External file editors are usually called for none text files or larger files.

For text files, the content may be edited directly in the text box below. When updating data that is not stored for the selected test suite or test case, the data will be localized automatically, i.e. the updated content is not stored in an upper test suite directory (**data**, **actions** or **extended** sub directory), but in the corresponding sub directory of the currently selected test suite or test case. In order to update



TH%/API/ClientServer/Dictionary/expected/output.out

```
call to close( )...success
call to extentDefinition( sScope, sExtentName )...success
call to extentDefinition( iNamespace, sExtentName )...success
call to extentDefinition( iPosition )...
```

common data, one has to select the test suite owning the file.

The file editor supports the actions described below.

Use external editor

External editors may be called directly via this action. In order to call file editors via file extension, a file association has to be defined in the system. When this is not the case, an ODABA file association may be defined in the ini- or configuration file in. Finally, one may define a test browser file association by adding a section with the extension name and define a variable CALL below, that provides the external editor function call (see **`FileEntry`**`::Edit()` in "TestBrowser Programmer's Guide").

Localize selected file

Localizing the file will create a copy of the file in the appropriate sub folder (**data**, **actions** or **expected**) of the current test suite or test case directory. When the file does already exist in this directory, nothing will happen.

Clear file content

The action can be used for clearing the complete file content. It does not delete the file but stores an empty file. When the file is not owned by the selected test suite or test case, it will be localized before being updated.

Delete file

When the file is owned by the selected test suite or test case directory, it will be deleted. Otherwise, the action does nothing.

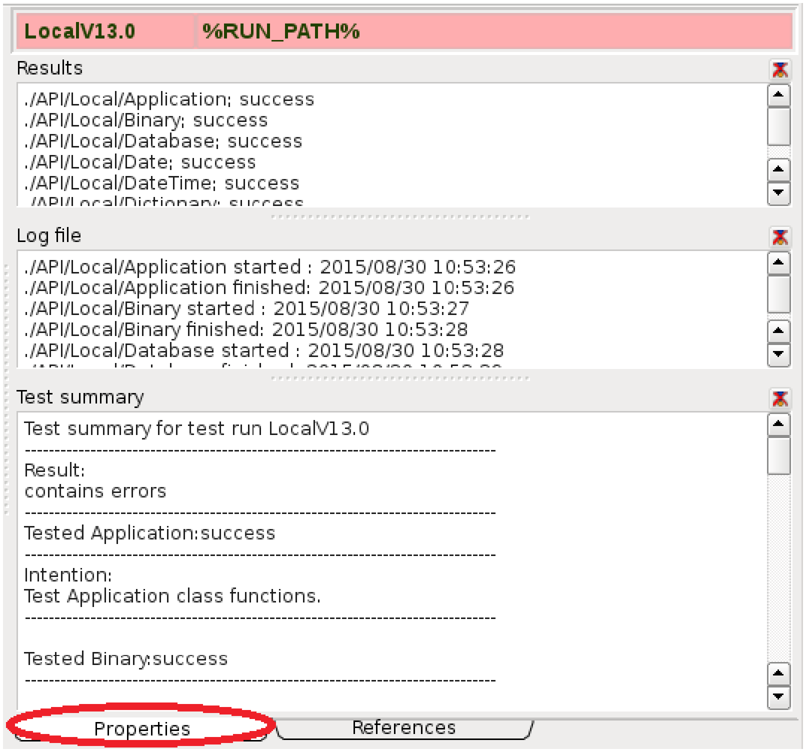
4.2.8 Test run

Test runs represent a list of tests from one or more test suites. Test runs are created for running release tests but also for collecting tests for a certain area (e.g. local and client/server tests). Results for each executed test run stored in the test run directory containing a number of text files. Data for a test run may be displayed in the **Properties** of **References** view for the test run.

-

4.2.8.1 Test run Properties view

Test run properties show the results for tests in the test run. As long as data in protocol fields is not cleared, test information for each test is appended at the end



LocalV13.0 %RUN_PATH%

Results

```
./API/Local/Application; success
./API/Local/Binary; success
./API/Local/Database; success
./API/Local/Date; success
./API/Local/DateTime; success
```

Log file

```
./API/Local/Application started : 2015/08/30 10:53:26
./API/Local/Application finished: 2015/08/30 10:53:26
./API/Local/Binary started : 2015/08/30 10:53:27
./API/Local/Binary finished: 2015/08/30 10:53:28
./API/Local/Database started : 2015/08/30 10:53:28
```

Test summary

```
Test summary for test run LocalV13.0
--------------------------------------------------------------------------------
Result:
contains errors
--------------------------------------------------------------------------------
Tested Application:success
--------------------------------------------------------------------------------
Intention:
Test Application class functions.
--------------------------------------------------------------------------------

Tested Binary:success
--------------------------------------------------------------------------------
```

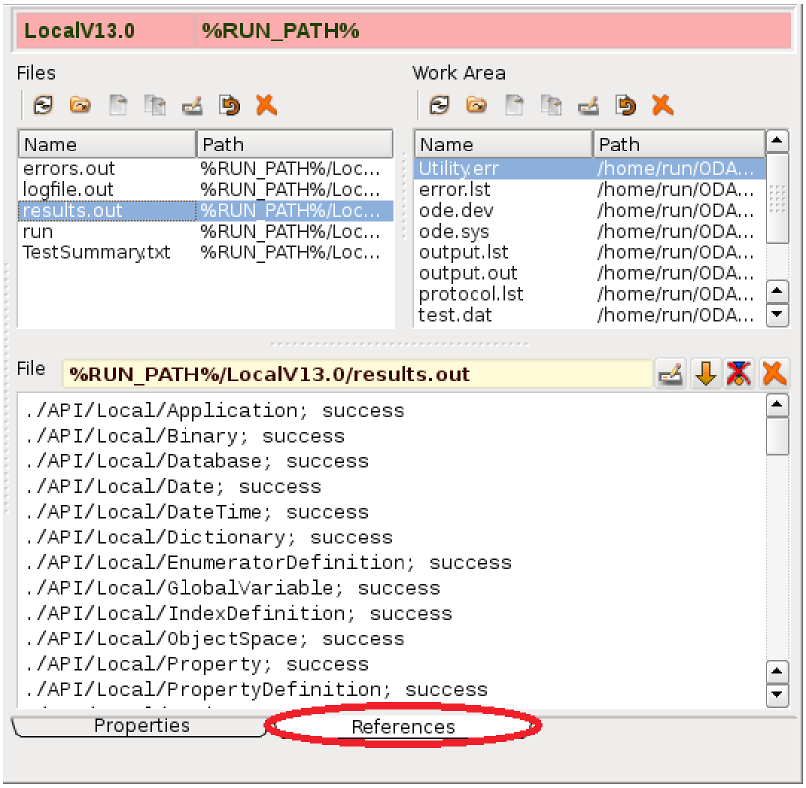
of each protocol list.

The **Results** area writes a line for each test with success (or failed) information. The **Log file** contains information about start and stop time for each test. The **Test summary** area is filled when clicking the test summary button above the test summery list.

All fields display file content of files stored in the test run directory. One may change the field content, which automatically will change the file content. In order to clear result list, log file or test summary, the clear button right above the field may be clicked.

4.2.8.2 Test run References view

The **References** view for a test run shows the files stored in the test run directory.



Name	Path
errors.out	%RUN_PATH%/Loc...
logfile.out	%RUN_PATH%/Loc...
results.out	%RUN_PATH%/Loc...
run	%RUN_PATH%/Loc...
TestSummary.txt	%RUN_PATH%/Loc...

Name	Path
Utility.err	/home/run/ODA...
error.lst	/home/run/ODA...
ode.dev	/home/run/ODA...
ode.sys	/home/run/ODA...
output.lst	/home/run/ODA...
output.out	/home/run/ODA...
protocol.lst	/home/run/ODA...
test.dat	/home/run/ODA...

%RUN_PATH%/LocalV13.0/results.out

```
./API/Local/Application; success
./API/Local/Binary; success
./API/Local/Database; success
./API/Local/Date; success
./API/Local/DateTime; success
./API/Local/Dictionary; success
./API/Local/EnumeratorDefinition; success
./API/Local/GlobalVariable; success
./API/Local/IndexDefinition; success
./API/Local/ObjectSpace; success
./API/Local/Property; success
./API/Local/PropertyDefinition; success
```

The **References** view shows two file lists. **Files** shows the files stored in the test run directory. The work area file list shows the files from last test (input and output files). The work area list will be refreshed when running the next test. When selecting a file entry in one of the lists, the file content is displayed in the file edit area below the lists.

4.3 Database access

Database access provides extended features for evaluating or manipulating test data and results. Database entries are, in case of test suite hierarchy similar structured as directories in the file system below the **main_suite** directory. Test runs are collecting different tests to be executes in a list of subordinated run entries. In this case, the list of run entries, which is flat, does not reflect the directory structure for tests in the **test_runs** directory.

The database is an ODABA database, i.e. it provides data by means of en object-oriented data model. Thus, the database may be considered as kind of external memory. The OSI script interface provides enhanced query features, which look similar to Java or C# program code.

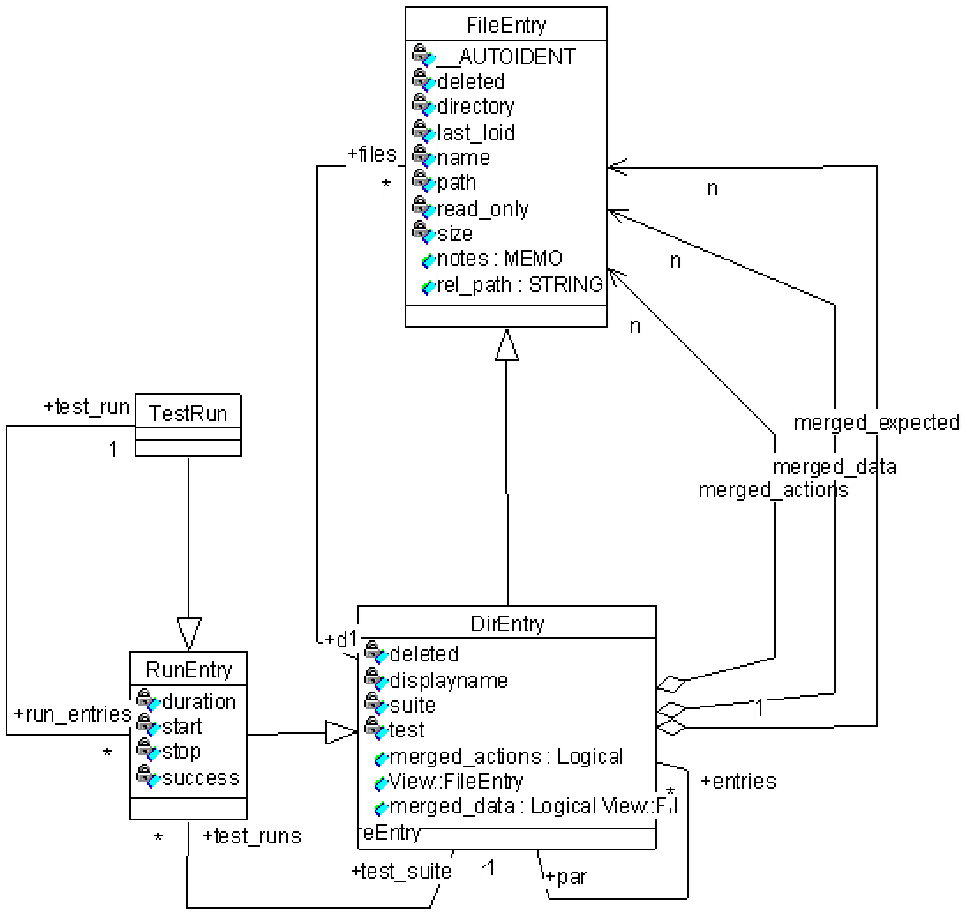
Since the **TestBrowser** application is completely written in OSI, one may also customize implemented function by providing those in an OSI overload directory (see examples at the end).

Besides database access functions, OSI provides a **`File`** class for accessing files and directories in the file system. It also supports data exchange between CSV and XML files for importing or exporting test requirements or results.

Also supported by **TestBrowser** is access to extension attributes stored in the **attributes** file for any directory.

4.3.1 Database model

The database model is quite simple as shown in the picture below:



The database provides several entries (extends) for accessing the database. Detailed description for the data types is provided in chapter "Complex data types". Extents are global database variables, the provide immediately access to data:

- `TopEntries` - Provides all main suites defined in the test system
- `DirEntry` - Provides all directory entries for test suites and test cases
- `TestRun` - Provides all test runs
- `RunEntry` - Provides run entries for all test runs

-

In order to access database instances, one starts with one of these entries following the traversal paths defined by the object model.

4.3.2 Accssing data in TestBrowser database

In order to access **TestBrowser** data in the **TestBrowser** database, one may run OShell, which provides a tool similar to command shell. Another way is using OSI scripts or combining OSI with OShell.

The example below shows how to create a final test protocol and a summary listing all successfully executed test intends. Since test intends are often not defined for single test cases, but for the test suite containing the test cases, the example tries to read intend from test case and if not existing from parent test suite.

```
// combined access
// enable OSI debugger
//set OSI_DEBUG=YES
set DSC_Language=English

// change database to data source TBDat (OShell.ini)
  cd TBDat

// activate user defined osi functions from OSILibrary path (OShell.ini)
  osi do
  dictionary.loadOSILibraries;
  end

// change collection of test runs
  cc TestRun
// change access key and locate instance
  co sk_Name
  loc "LocalV13.0"
// change to collection of test runs (RunEntry) for this TestRun
  cc run_entries
// run embedded osi function for listing successful executes test runs
osi begin
VARIABLES           // required for variable definitions, only, may be
omitted
  string     line;
  string     intend;
  int        total = 0;
  int        successful = 0;
  int        err = 0;
PROCESS             // required only in connection with VARIABLES section
  while ( next() )
    switch ( success ) { // 1: success;  0: error;  -1: not executed
    case 1 : line = displayname + '\t';
             if ( test_suite.tryGet(0) ) {
               intend = test_suite.ReadData("suite"); // file suite
contains intension description for testcase
         if ( intend == "" ) // inhrits intend from parent test suite
           if ( test_suite.par.tryGet(0) )
             intend = test_suite.par.ReadData("suite");
```

```
          line += intend;
        }
            Message(line);      // use File::writeLine for writing data
to file
        ++successful;
        break;
    case 0 : ++err;
            break;
  }
  Message("Testrun contains " + (string)count() + " tests.");
  Message("Successful: " + (string)successful);
  Message("Failed    : " + (string)err);
end
```

4.3.2.1 Database access via OShell

OShell acts similar to a common shell of the operating system. The difference is, that drives correspond to databases defined in data sources and directories in the file system correspond global and local collections, i.e. extents and local collections in object instances. Details for OShell commands are provided in ODABA Utilities/OShell.

In order to run an OShell script, an ini-file (**OShell.ini**) is required. The **TestBrowser** delivery provides both, an **OShell.ini** file and an OShell.cmd file for calling the OShell. When changing the environment after installing a **TestBrowser** system provided with default configuration, one has to adopt, probably, the common file as well as the ini-file.

When calling **OShell.ini** with ini-file parameter, only, a command prompt appears requesting further input. One may, however, also prepare OShell scripts in advance (like command procedures) and passing those as additional (second) parameter to the program call.

The Example below shows some features when running an OShell script simply by calling OShell.cmd.

```
Running L:\odet\OShell.exe with:
  ini-file: OShell.ini
  script file:
ODABA>cd TBDat
TBDat>cc TopEntries
TBDat/TopEntries>li
  1
  150
  200
  278
TBDat/TopEntries>loc 0
TBDat/TopEntries>p
  __AUTOIDENT=1
  name=API
  path=%ROOT_PATH%
  size=0
  directory=Y
  deleted=N
```

-

```
  read_only=N
  last_loid=0
  rel_path=
  notes=
  suite=Y
  test=N
  displayname=API
TBDat/TopEntries>co sk_Name
TBDat/TopEntries>li
  actions
  API
  OSI
  Utilities
TBDat/TopEntries>loc API
TBDat/TopEntries>p
  __AUTOIDENT=1
  name=API
  path=%ROOT_PATH%
  size=0
  directory=Y
  deleted=N
  read_only=N
  last_loid=0
  rel_path=
  notes=
  suite=Y
  test=N
  displayname=API
TBDat/TopEntries>cc entries
TBDat/./entries>p count
  count=6
TBDat/./entries>li
  actions
  ClientServer
  data
  expected
  Local
  odaba
TBDat/./entries>loc Local
TBDat/./entries>cc entries
TBDat/../entries>p count
  count=19
TBDat/../entries>li
  actions
  Application
  Binary
  data
  Database
  Date
  DateTime
  Dictionary
  EnumeratorDefinition
  expected
  GlobalVariable
  IndexDefinition
  ObjectSpace
  Property
  PropertyDefinition
  String
  Time
  TypeDefinition
  Value
TBDat/../entries>q
```

4.3.2.2 OSI script for data exchange

There are many ways for defining data exchanges. Here, some examples are provided in order to illustrate several possibilities. In order to provide OSI functions overloading default implementations in **TestBrowser**, those have to be provided in one or more files which are stored in the same directory.

When running **TestBrowser**, those function are loaded automatically, when the directory path has been provided in ODABA option or environment variable OSI_Library. In order to provide user defined OSI functions in OShell, one has to load those functions in addition as shown below.

In order to refer to external data sources, a file description for import/export may be defined. This setting is optional, since **TestBrowser** does provide dummy functions for Import/export, only, which have to be overloaded in any case. Those import/export functions define the way to refer to external files. The setting for IMPEXP_DEF is required for the example, only.

OSI script language provides many other features, which cannot be illustrated here. For more information see ODABA Script Interface.

```
// set OSI library path
set OSI_Library=%cd%/TestBrowser/osi/*.osi
// set external file description
set IMPEXP_DEF=%cd%/TestBrowser/osi/Imports.fsc

// OShell laoding user-defined OSI functions
// data source has to be opened first
  cd TBDat
// activate user defined osi functions from OSILibrary path (OShell.ini)
  osi do
    dictionary.loadOSILibraries;
  end
```

Example for simple export function

The export example creates a simple CSV output file filled with data from different levels in the test case hierarchy.

-

```
// the function overwrites default implementation in resource database
// and is called when pressing the export button in the Test Browser
application
// (above test suite tree).
collection bool DirEntry::ExportToCSV (string sPath )
{
VARIABLES
  SET < VOID >       exp;
PROCESS
exp.openExtern(objectSpace,sPath,Option("IMPEXP_DEF").toString,"",odaba::
AccessModes::Write,false);
  ExportToCSV_intern(exp);
  exp.closeAll;
FINAL
  return(true);
};

collection bool DirEntry::ExportToCSV_intern (set< VOID > &exp )
{
  top;
  while ( next ) {
    if ( test )
      ExportEntry(exp);
    entries.ExportToCSV_intern(exp);
  }
  return(true);
};

collection bool DirEntry::ExportEntry (set< VOID > &exp)
{
  exp.initializeInstance;
  if ( !selected ) {
    exp.test = "test";
    exp.isTest= "isTest";
    exp.dataRequirements= "dataRequirements";
    exp.result= "result";
    exp.testDescription= "testDescription";
    exp.testCase = "testCase";
    exp.caseDescription= "caseDescription";
    exp.suiteDescription = "suiteDescription";
    exp.suiteDescription = "suiteDescription";
    exp.comment = "comment";
    exp.save;
  } else {
    exp.test = displayname;
    exp.isTest= "true";
    exp.dataRequirements=(GetAttribute("Data"));
    exp.result=GetAttribute("Result");
    exp.testDescription=ReadData("suite");
    exp.testCase = par(0).displayname;
    exp.caseDescription = par(0).ReadData("suite");
    exp.testSuite = par(0).par(0).displayname;
    exp.suiteDescription = par(0).par(0).ReadData("suite");
    exp.comment = GetAttribute("Comment");
    exp.save;
  }
```

Import written in OSI code

The import example loads data from an CSV file, which defines a hierarchical structure for grouping test cases in test suites. Tests cases are provided based on software requirements, e.g. this is an example for requirement driven test. It is a bit more complex than the export example, but it demonstrates several features provided by OSI.

```
// The function overwrites the default implementation in TestBrowser
resource database
// and is called when pressing the import button in the Test Browser
application
// (above test suite tree).
//
// The function imports test cases from a requirements csv file with
following columns:
//  id           - requirement identifier
//  description  - requirement description
//  testCatecory - coded test category (see description below)
//  reviewState  - Requirement review state
//  lastModified  - last modification of requirement
// This names ate taken from CSV file head line or from external file
description file.fsc
// The test suite hietarchy is setup according to testCategory value,
which is defined
// for each line in the import file:
//  Hn_title - Heading level (n) with optional title (title)
//  Tn       - Test case with test priority (n)
//  TH       - table heading column (number of table columns results from
number of following
//             TH records. Each line in a table defines a test case
//  TCn      - Table cell data, where the first column in a table line
defines the test case
//             priority (n)
// Table lines result in one test case, but information is stored for all
cells in the test
// intend, which is filled with requirement description(s).
collection bool DirEntry::ImportFromCSV (string sPath )
{
VARIABLES
  extern ImportProgress   importProgress;
  SET < VOID >       imp;
  SET < DirEntry >  &curSuite = self;
  String             sCategory();
  string             description;
  string             title;
  string             category;
  string             id;
  string             str;
  string             type;
  string             sType;
  string             sTotal;
  int32              current;
  int32              priority = 0;
  int32              level = 0;
  int32              curLevel = 0;
  int32              colCount;
```

-

```
  int32             curColumn;
  string(100)       colTitle[20];
  bool              bTableHeader = false;
  bool              bTable = false;
  bool              bHeader = false;
  bool              bTest = false;
PROCESS
  imp.openExtern(objectSpace,sPath,Option("IMPEXP_DEF").toString,"",
               odaba::AccessModes::Read,true);
  sTotal = '/' + (string)imp.count;

  while ( imp.next ) {
    ++current;
    if ( current%10 == 0 )
      importProgress.Progress((string)current + sTotal);
    sCategory.assign(imp.testCategory);
    category = sCategory.splitTop('_');
    title = sCategory;
    type = category.left(1);
    sType = category.mid(1,1);

    switch ( type ) {
      case 'H'  : level = category.mid(1).toInteger;;
                  while ( level > curLevel + 1 ) {
                    curSuite &= curSuite.entries;
                    ++curLevel;
          str = "Missing headline level " + (string)curLevel;
                    curSuite.CreateSuite("",0,curLevel,str,"",imp);
                    Message(str);
                  }
                  if ( level == curLevel + 1 ) {
                    curSuite &= curSuite.entries;
                    curLevel = level;
                  } else while ( level < curLevel ) {
                    curSuite &= curSuite.parent;
                    --curLevel;
                  }
                  id = imp.id;
                  description = imp.description;
                  priority = 0;
                  bTable = false;
                  bTableHeader = false;
                  bTest = false;
                  bHeader = true;
                  break;
      case 'T'  : if ( !bTest ) {
                    curSuite &= curSuite.entries;
                    ++curLevel;
                    bTest = true;
                    if ( !curSuite.positioned ) curSuite.tryGet(0);
                  }
                  bHeader = false;
                  sType = category.mid(1,1);
                  switch ( sType ) {
                    case 'H' : if ( !bTableHeader ) {
                                 bTableHeader = true;
                                 colCount = 0;
                                 bTable = false;
                                 if ( curSuite.positioned ) {
                                   if ( curSuite.entries.tryGet("0") )
                                     curSuite.entries.Delete;
                                   if ( curSuite.entries.tryGet("1") )
                                     curSuite.entries.Delete;
                                 }
                               }
                               colTitle[colCount] = imp.description;
                               ++colCount;
                               break;
                    case 'C' : if ( bTableHeader || bTable ) {
                                 bTableHeader = false;
                                 bTable = true;
                                 if ( curColumn == 0 ) {
                                   id = imp.id;
                                   priority = category.mid(2).toInteger;
                                 }
                                 description += colTitle[curColumn] +
                                           ': ' + imp.description + '\n';
                                 ++curColumn;
                               } else
                                 Message("Missing table header before
                                          column with ID: " + imp.id);
                               break;
                    default  : priority = category.mid(1).toInteger;
                               id = imp.id;
                               description = imp.description;
                               bTable = false;
                               bTableHeader = false;
                  }
                  break;
    }
    if ( type == 'H' || type == 'T' ) {
     if ( bHeader ||
           (bTest && !bTable && !bTableHeader) ||
           (bTable && curColumn == colCount)     )
      {
        if ( bTable )
          curSuite &= curSuite.entries;
        curSuite.CreateSuite(id,priority,curLevel,title,description,
                             bHeader,imp);
        if ( bTable )
          curSuite &= curSuite.parent;
        description = "";
        title = "";
        id = "";
        curColumn = 0;
      }
    }
  }
  imp.closeAll;
FINAL
  return(true);
};
```

-

```
collection bool DirEntry::CreateSuite (string id, int32 priority, int32
level, string title, string description, bool bHeader, set< VOID > &imp )
{
    top;
    while ( next )
      if ( GetAttribute("ID") == id )
        break;
    if ( !positioned ) {
      Create(parent.FullPath,(string)NextNumber(count-2),true,false);
      if ( !bHeader ) {
        entries.Create(FullPath,"0",true,true);
        entries.SetAttribute("Result","success");
        entries.Create(FullPath,"1",true,true);
        entries.SetAttribute("Result","error");
      }
    }
    WriteData("suite",description);
    if ( id != "" ) {
      SetAttribute("ID",id);
    if ( bHeader )
        SetAttribute("Header",(string)level);
      else
        SetAttribute("Priority",(string)priority);
      if ( title != "" )
        SetAttribute("Title",title);
      SetAttribute("ReviewState",imp.reviewState);
      SetAttribute("LastModified",imp.lastModified);
  }
  displayname = DisplayName;
    return(true);
};
```

4.3.2.3 Common features

Besides access functions provided by OSI, **TestBrowser** provides extended features. OSI functions implemented for database structures are described in function reference and may be called from within any user defined OSI function. Functions are provided on different interface levels:

- ODABA database access API
- ODABA utility functions (services)
- **TestBrowser** object class functions

4.3.2.3.1 ODABA Access functions

ODABA provides a comprehensive API providing support for database access handles on different levels as well as for several helper classes. Nearly all classes and functions are accessible from within C++, OSI and .NET languages.

Details for the ODABA API are described in:

ODABA Online Documentation/**Reference documentation/ODABA Application Interface**, which is also provided as local HTML documentation when downloading ODABA.

4.3.2.3.2 ODABA service classes

In order to process files, XML structures, sending or receiving emails etc., ODABA provides several service functions (since version 13). Nearly all classes and functions are accessible from within C++, OSI and .NET languages.

Details for the ODABA service API are described in:

ODABA Online Documentation/**Reference documentation/ODABA Application Interface/Service Classes**, which is also provided as local HTML documentation when downloading ODABA.

4.3.2.3.3 TestBrowser functions

TestBrowser functions are provided for database object types (persistent data types). A description of supported functions is available in "**TestBrowser** Programmer's Guide".