

Chapter 39

Water Test: Building Self-Testing Software

IN THIS CHAPTER

- ◆ Creating test cases
- ◆ Making a test suite
- ◆ Storing tests and associating a test with an object
- ◆ Testing the execution time
- ◆ Creating assertions

WATER TEST DEFINES HOW TO CREATE and run tests within the Water language. Everyone knows that good testing improves software quality. In most languages, though, creating tests is difficult and testing is not formally supported in the language. Water Test includes a number of powerful tools for easily integrating testing into the development cycle.

Creating Test Cases

A test case can be created with a single click. If you want to create a test case for the following Water expression, select the expression by double-clicking at the start of the path; then right-click and choose "<test .../>".

```
10.<times 10/>
```

The preceding code is replaced by the following test case:

```
<test 10.<times 10/>
  100
/>
```

The IDE executed the code, wrapped a `test` call around the original code, and added a second argument to `test` that is the returned result. When you select the new expression and press Execute, it returns `true`.

By convention, the first argument is the call to test, and the second argument is the expected return value. If both arguments return values that are `equal`, the test returns `true`; otherwise, it returns `false`.



DEFINITION *Test* follows the following process: Execute the first argument and store in `var1`. Execute the second argument and store in `var2`. Return `var1.<equal var2/>`. If there is only one argument, execute it—if it returns `true` the test passed, otherwise the test failed.

The `test` call performs a regression test because the `test` call has the expected baseline result as the second argument of the call.



DEFINITION A *regression test* compares the results of execution with some expected baseline of a result.



XREF A call to `test` does not evaluate its arguments in the call. This is possible because of execution kinds. Execution kinds are described in Chapter 30 on Water Execute.

Testing with a custom comparison

The following example is another way to perform the same test as the preceding example, but using a single argument. The first argument has the call to test, the expected result, and the `equal` comparison. This is useful when you don't want to use the default `equal` comparator between the two arguments.

```
<test 10.<times 10/>.
  <equal 100/>
/>
```

The `equal` method takes arguments `whitespace_sensitive` and `case_sensitive`, which are particularly useful for testing the result of formatting methods.

```
<test
<thing x=10/>.<to_xml/>.
  <equal
```

```
"<thing><attributes>x=10</attributes></thing>"
whitespace_sensitive=false case_sensitive=false
/>
/>
```

Testing with setup code

If the test requires some setup code, put the setup code within a `do` call that ends with the call to `test`. If any local variables are used, they need to be defined within the first argument.

```
<test
  <do>
    <set x=10/>
    x.<times 100/>
  </do>
  1000
/>
```

Testing for an error

It is also possible to use the `try` clause from Water Flow to create a test that verifies an error occurs when executing a Water expression.

```
<test
  <try
    thing.no_field_by_this_name
    false
  >
  true
</try>
/>
```

Associating a test with an object

Tests can be associated with any object. The following example shows how to create a new test on the `integer` class.

```
integer.<test 1.<plus 1/> 2/>
```

Tests can be run on a particular object by calling `run_tests`.

```
integer.<run_tests/>
```

Other Testing

Other testing methods are described in the following sections.

Test suite

Test runs are accumulated into `test_suite`. The test results may be printed by calling `test_suite.<report/>`. To clear the test results, call `test_suite.<clear_report/>`. Each failure shows the code for each argument and the returned value for each.

```
test_suite.<clear_report/>
<test 1.<plus 1/> 2/>
<test 2.<plus 2/> 3/>
test_suite.<report/>
```

Output as text:

```
Test Suite Report:
Tests Run: 2
Tests Failed: 1
test failed with:
  2.<plus 2/> returns 4
  3 returns 3
```

Storing tests and running multiple tests

By convention, tests are usually placed right after the object they are testing, or in a dedicated test file. If a file is named `my_program.h2o`, the convention is to place the tests for `my_program` in a metafile named `my_program_f_tests.h2o`. A metafile is a file associated with another file. The convention of using `_f_` unambiguously identifies an associated file and it follows the same convention as Water Metafield.



Chapter 26 describes Water Metafield in detail.

You can run tests in multiple files by using the following pattern:

```
<test_suite.init_report/>
<file "C:/my_program_f_tests.h2o"/>.<execute/>
<file "C:/foo_f_tests.h2o"/>.<execute/>
test_suite.<report/>
```

Output as text:

```
Test Suite Report:
Tests Run:         350
Tests Failed:      0
```

The first line clears the report, the next two lines execute the tests within those files, and the last line returns a report that shows how many tests were run and the details of any tests that failed.

Timing Tests

Water provides a standard method named `time_it` for timing the execution of code. `time_it` is part of the `test_suite` package. The content of `time_it` can be any Water code. It returns the number of milliseconds it takes to execute the code.

```
test_suite.
<time_it>
  "some code to time goes here"
</time_it>
```

If the execution time is less than 20 milliseconds, you might want to repeat the execution multiple times. Longer tests usually give more consistent results. `time_it` takes the number of times to repeat the execution as its first argument.

```
test_suite.
<time_it 1000>
  "some code to time goes here"
</time_it>
```

A test call could be combined with `time_it` to verify that running some code always completes within a specific number of milliseconds. The following example verifies that returning the Water language home page takes less than five seconds.

```
<test
  test_suite.
  <time_it>
    www.<web "http://www.waterlang.org/">.content
  </time_it>.
  <less 5000/>
/>
```

Assertions

An assertion is a test that happens during normal program execution. Water Test has an `assert` method that verifies that the subject of the call to `assert` is of a given type. The following example shows a Water path without calling `assert`.

```
6.<plus 10/>
→ 16
```

The following example shows how to insert a call to `assert` within a path. An `assert` call can be placed in any path without affecting the result.

```
6.<assert number/>.<plus 10/>
→ 16
```

The first argument in a call to `assert` is a Water Type. The subject of the call is tested against the type by calling the `is_type_for` method. The following example shows a basic implementation of `assert`.

```
<defmethod assert a_type>
  <if> a_type.<is_type_for _subject/>
    _subject
  else
    <error "Assertion failure"/>
  </if>
</defmethod>
```



Chapter 4 describes Water Type in detail.

`assert` takes an optional second argument named `doc` that can contain any object, but it is typically a human-readable string that describes what was expected. The string is also used when reporting an assertion failure.

```
3.<assert doc="small number"/>.<plus 10/>
→ 13
```

If the assertion fails, several outcomes are possible. It could return an error immediately, a failure message could be written to the console, or the failure could be ignored. The assertion failure options are changed through the IDE. The following example shows an assertion failure that returns an error.

```
"6".<assert number/>.<plus 10/>
→
<error "Assertion failure: expected a number
      and got a string of '6'"
/>
```

Replacing local variables with asserts

In many languages, a local variable is used to document an intermediate value and declare its type. I have found that assertions are more effective for this purpose. The following example shows the use of a local variable.

```
<set x=3/>
<set small_value_of_x=x=type.<range_of max=6/> />
small_value_of_x.<plus 10/>
→ 13
```

The following example shows the same effect can be accomplished with more precision by using `assert`.

```
<set x=3/>
x.<assert doc="small number"> it.<less 6/> </assert>.<plus 10/>
→ 13
```

`assert` takes an optional second argument named `doc` that can contain any object, but it is typically a human-readable string that describes what was expected. The string is also used when reporting an assertion failure.

```
3.<assert doc="small number"/>.<plus 10/>
→ 13
```

`assert` may also have an expression in the content argument. The expression must return `true` for the assertion to pass. The local variable named `it` is bound to the subject of the call to `assert`. In the following example, `it` has the value of 3.

```
3.<assert> it.<less 6/> </>.<plus 10/>
→ 13
```

The `assert` and `test` methods are very useful because they can help document the expected behavior of a system. When something goes wrong, it becomes much easier to determine the cause of the failure.

Summary

This chapter described how to create many types of test cases. The tests can be stored in a file, added to a test suite, and run as a group. Tests and assertions make great documentation because they describe the expected behavior of a system. Everyone knows that testing is very important for improving software quality, but writing tests typically involve a lot of effort. The goal of Water Test is to make it so easy to create and maintain tests that more and better tests are written.