Automating Test Oracles and Decomposability

Conceptually all tests have three parts:

- the input (including the environment and internal state) is the stimulus provided by a test designer
- the expected output (including environment and internal state) is what the test requires to consider the software correct, it is generated via some Test Oracle
- the actual output (including environment and internal state) is the result of executing the software with the given input.

The output is a result of a set of post-conditions associated with the software being tested.

Where and how do you check results? What information do you record about results?
How does result checking impact test design?

Test Data Location questions: Where do the input, expected output, and actual output reside? In the test case code? In a separate set of files or a database?

Comparison Timing questions: How close in time after the execution of a test is the check of the result made? As each step of the test is made? After the entire test? After a set of tests?

If there were only one answer to these questions, they would have been solved long ago. Instead, a series of trade-offs (forces), helps determine when each method is most appropriate. This paper uses the concept of patterns[1] to describe the details of why you choose a particular test result handling method to automate a test design.

Context

You are designing tests and need to consider how much to reasonably include in a given test.
You are choosing how to automate tests that have been designed and need to make the tests as understandable and maintainable as possible.
This doesn't apply to ad hoc testing, exploratory testing or testing without an Oracle.

Test Data Location (data patterns):

The input, expected output, and actual output may be coded directly into the test (program code, test script, etc.) or apart from it (input file, expected result database, etc.). The *Co-locate Data* pattern tells us that we should make them all internal or all external resulting in the *Self-contained data* or *Data-driven data* patterns respectively. The helper patterns *Move actual to Internal* and *Move actual to External* allow transformations into the all internal or all external patterns. The anti-patterns, **Cause without Effect** and **Effect without Cause**, show what results from separating the input from the output.
The table below lists patterns in italics and anti-patterns using outlined text.

Input - Cause	Output – Effect		***Pattern* / Anti-Pattern**
	Expected	Actual	
Internal	Internal	Internal	***Self-contained data***
Internal	Internal	**External**	*Move actual to Internal*
Internal	**External**	Internal	**Cause without Effect**
Internal	**External**	External	**Cause without Effect**
External	**Internal**	Internal	**Effect without Cause**
External	**Internal**	External	**Effect without Cause**
External	External	**Internal**	*Move actual to External*
External	External	External	***Data-driven data***

[1]If necessary see "*A Pattern Language for Pattern Writing*" by Gerard Meszaros and Jim Doble at http://www.hillside.net/patterns/Writing/patterns.html for the more details about the meaning of the headings in the patterns used in this paper.

Comparison Timing (check patterns)

Many of the features of software that are tested result in multiple post-conditions. Each post-condition must be checked, but is each post-condition its own test? See **Pass Each Post-Condition** anti-pattern for why using only a single post-condition as a test is misleading. The *Whole Function* pattern considers a single test to include the **evaluation** of all of the relevant post-conditions.

Thus the following is considered incorrect:

```
BeginTest
Insert database record         # Test operation
Verify successful return code # post-condition1
EndTest
BeginTest
Retrieve same database record
# post-condition2
Verify actual retrieved record matches (expected) input record.
EndTest
```

Instead the following should occur using either the *Check as you Go* pattern or the *Batch check* pattern described later:

```
BeginTest    [Check as you Go using Self-contained data]
Insert database record              # Test operation
Verify successful return code       # post-condition1
Retrieve same database record
# post-condition2
Verify actual retrieved record matches (expected output) input
record.
EndTest
```

or

```
BeginTest    [Batch check using Data-driven data]
Read input for database record
Insert database record         # Test operation
Print return code              # post-condition1
Retrieve same database record
Print actual retrieved record # post-condition2
Verify actual printout matches expected printout
EndTest
```

Notice there are two verify steps in the *Check as you Go* case above. Either verify step can cause the test to mark the feature as failed.

However the timing and number of verifies is not prescribed. It is not even required that a verify be a direct part of the test code. Sometimes several tests will output their results before any of the comparison (verification) is done. However, each test is not considered complete *until* the results of all successful comparisons are done. It is acceptable to not complete the comparisons if a discrepancy has already been shown. The primary consideration for continuing comparisons after a discrepancy is whether it would provide additional useful information for diagnosing the failure. It does not impact the outcome of the test.

Question	Answer	***Pattern*** / **Anti-Pattern**
Where to store test results?	All internal	***Self-Contained Data***
	All external	***Data-Driven Data***
When to check?	At each step	***Check as you Go***
	At the end	***Batch Check***
Depend on previous test?	No	***Separable Tests***
	Yes	***Smart Dependency Checking***
Pass a test for each post-condition?	No	***Whole Function***
	Yes	**Pass Each Post-Condition**

The following patterns are a linkage between the Test "Oracle Micro-Patterns" for "Pre-Specification Oracles" (S*olved Example, Simulation, Approximation, and Parametric*) and "Test Automation Design Patterns" approaches of *Built-in Test* and *Test Cases* as described in *Testing Object-Oriented Systems: Models, Patterns, and Tools* (*http://www.rbsc.com/TOOSMPT.htm*). [See appendix for summaries of these patterns.]

The second pair of patterns are complementary patterns for solving the same problem. *Check as you Go* is the generally preferred method for ease of understanding. *Batch check* (or *Benchmark*) is particularly useful for changing the Oracle *Judging* pattern into *Solved Example*.

The third pair of patterns are supplemental Test Automation Designs. *Separable Tests* (or *Atomic Tests)* deals with selection and independence of tests. *Smart Dependency Checking* is used when decomposability to satisfy the *Separable Tests* pattern precludes the second test from being independent.

Whole Function (or *All behavior)* deals with post-conditions of an Item Under Test (IUT).

Note that patterns can be combined as the situation demands. You might use *Check as you Go* for some of the post-conditions which increases the ability to create *Separable tests* and yet use *Batch check* for voluminous post-conditions which may change frequently.

The rest of this paper presents the patterns followed by an appendix with pointers to other patterns and references.

Pattern Name: *Co-Locate Data*

Context

Reusable tests are being created and the data for the tests must be stored.

Problem

Inputs come from various sources including the test script, the environment, or internal state. Similarly output consists of various sources including the environment, internal state, test script, or output files (including standard out, standard error, log files, and database files).

Forces

Input and Output are naturally separated streams and are usually not mixed.

Solution

Put the input, expected output, and actual output either within the test code or put them all centralized external to the test code. It should be possible to see the input and expected output together (either in the code, in a file, or in an extract from a database). Transform input or output from other sources either into the code or the centralized external location.

Indications

Input data and expected output data are separated.

Rationale

Having the input and expected output in the same place (either internal or external) increases readability and understandability (including for maintenance). If you separate them, then you end up with the anti-patterns: **Cause without Effect** and **Effect without Cause**.

Generally, tests are designed to transform the cases where the expected and actual output reside in different locations, into the cases where they are all the same.

Resulting Context/Consequences

Verification of the correct expected output is easier since it is all in one place.

Related Patterns

See ***Self-contained data*** for putting the data inside the test script or program.
See ***Data-driven data*** for keeping the data outside the test script or program.
See *Move actual to Internal* or *Move actual to External* to make the actual data co-located with the input and expected output.
The anti-patterns **Cause without Effect** and **Effect without Cause** describe problems with bad solution of having the input not co-located with the expected output.

Pattern Name: *Self-contained data*

Aliases: internal data

Context

You are creating a reusable test.

Problem

Where do you keep the test input data and test expected output data?

Forces

- Want to read and understand a test with as few external references as possible.
- Tests are each relatively unique in their parameters (or sequences of actions).

Solution

Include the data in the test script or test code.

Indications

The amount of input and output per result is relatively small and easy to understand.

Rationale

Test script logic becomes less complicated or more obvious when it is included directly with the logic.

Resulting Context/Consequences

Makes it impossible to lose part of the test, if it is only in one file.
Sometimes reduces maintainability if bulk updates of expected results are needed.
Reduces code reusability if inputs and results are hard-coded or expressed as symbolic constants in the code.

Related Patterns

Contrast with *Data-driven data*.
See also *Move actual to Internal*.

Examples/Known Uses

Typically used in API tests, for example Posix Verification Suite.

Code Samples

The *Check as you Go* pattern Code Sample demonstrates simple *Self-contained data*.
The *Batch check* pattern Code Sample demonstrates input and output contained within the test script.

Pattern Name: *Move actual to Internal*

Context

The result of running a program is some external post-condition, for example writing a message in a separate log-file, but the input and other post-conditions (expected output) are internal.

Problem

The check of post-conditions is distributed between external script code and internal script code.

Forces

- ❑ Post-conditions occur in different environments.

Solution

The external actual output is read into or checked by the test program to transform it into the internal actual output case.

Indications: Input and most output are internal.

Rationale: It is most convenient to compare between expected output and actual output in the same location. Since the input and exptect output are already internal, they are the majority, and the actual output must be made to match them.

Resulting Context/Consequences: You have ***Self-contained data*** after moving actual to internal.

Related Patterns: See ***Self-contained data***.

Code Samples

A call to lock a record in a database that is out of locks might have two post-conditions:

1. an exception is returned to the caller indicating no locks (internal), and
2. a message is written to the operations log indicating the database has run out of locks (external).

The first example shows an external post-condition (DBlog output) causing comparison being done external to the test program in addition to the internal post-condition (exception raised) because of the internal input.

Run DB lock test (internal)

```
for I=1,numlocks {getDBlock();}
try { getDBlock(); logFail(); }
  catch (TooManyLocksException e){/*Got expected exception*/}
  catch (Exception e) { logFail(); }

diff DBlog expectedDBlog (external)
```

Applying this pattern results in all post-conditions being checked for internally in the code

Run DB lock test (internal)

```
for I=1,numlocks {getDBlock();}
try { getDBlock(); logFail(); }
  catch (TooManyLocksException e){/*Got expected exception*/}
  catch (Exception e) { logFail(); };
System.exec("diff DBlog expectedDBlog");
```

The above shows all post-conditions being checked from within the test code even though external actual output is involved.

Pattern Name: *Data-driven data*[2]

Aliases: external data

Context

You are creating a reusable test and you have many data combinations to be tested.

Problem

Where do you keep the test input data and test expected output data?
Hard-coding data in test scripts makes it laborious to create lots of related tests.

Forces

- Output is voluminous.
- Output is difficult to predict and can frequently change from release to release.
- Additional, similar tests may need to be added.

Solution

Separate test data from scripts. This makes it easier to create multiple related tests.

Indications

Test data is to be provided externally, for example from domain experts.
You have existing legacy data.

Rationale

Separating data from procedure is a classic computer science technique for structuring code.

Resulting Context/Consequences

Mentally tracing through the test requires an extra level of indirection to substitute the data-driven values in the specific situation. Test script logic becomes more complicated or less obvious when data is separate.

Related Patterns

Frequently used with *Batch Check*. Contrast with *Self-contained data*.
See also *Move actual to Internal*.

Examples/Known Uses

Compiler tests.
SQL optimizer tests showing optimizer strategy (which may change release to release).
Stub generation for distributed methods (for example CORBA IDL, or Java RMI).

Code Samples

If the *Batch check* pattern Code Sample had an external existing `actualOutput` file, instead of creating it on the fly, it would demonstrate *Data-driven data*.
The *Move actual to External* code sample also shows *Data-driven data*.
The **Cause without Effect** code sample also shows *Data-driven data*.

[2] Much of this material is a derivation from *Data-driven testing* pattern presented at PoST 1, Jan. 2001

Pattern Name: *Move actual to External*

Context

The result of running a program is some internal post-condition, for example an exception being raised, but the input and other post-conditions (expected output) are external.

Problem

The check of post-conditions is distributed between external script code and internal script code.

Forces

- Post-conditions occur in different environments,

Solution

The internal post-condition effect is outputted to transform it into the actual external output case.

Indications

Input and most output are external.

Rationale

It is most convenient to compare between expected output and actual output in the same location. Since the input and expected output are already external, they are the majority and the actual output must be made to match them.

Resulting Context/Consequences

You have *Data-driven data* after moving actual to external.

Related Patterns

See *Data-driven data*.

Code Samples

For bad syntax a compiler is supposed to have two post-conditions:

1. display an error (external output), and
2. exit with a non-zero status code (internal to script check).

Below is a mixture of checking a post-condition in the script (internal) and externally.

```
Compile <testInput >actualOutput
if [ $? != 0 ] ; then logFail(); # Internal
diff actualOutput expectedOutput # External
```

After applying the pattern you get:

```
Compile <testInput >actualOutput
echo "ResultCode $?" >>actualOutput
diff actualOutput expectedOutput  # External
```

The above shows all post-conditions being checked externally to the test code even though internal actual output is involved.

Anti-Pattern Name: Effect without Cause

Aliases: output only

Context

Expected output is recorded without knowing the input.

Problem

Outputs can match, but for the wrong reasons.

Forces

Solution

Record input with the outputs. It is also possible to derive the input as an extraction from the expected output.

Rationale

It is easier to review for correctness when the inputs and outputs are together.
It is easily verified if each effect occurs due to its cause.

Resulting Context/Consequences

Test may have to echo or otherwise copy the input into the output stream.

Code Sample

Input file:

```
      set onn   # should fail because it should be "set on"
      set Off   # should succeed because Off should be case insensitive
```

Actual & expected outputs:

```
      Illegal Set argument
```

Test marks product as passed because it got expected output. Although this has the input, expected output, and actual output all external files, this input is external to the expected output file.
Correct way:

Expected output:

```
      set onn   # should fail because it should be "set on"
      Illegal Set argument
      set Off   # should succeed because Off should be case insensitive
```

Actual output:

```
      set onn   # should fail because it should be "set on"
      set Off   # should succeed because Off should be case insensitive
      Illegal Set argument
```

Test marks product as failed.
The product only looks at first 2 letters ("on") and is not case insensitive.

Notice how the expected output is easy to understand since both the cause and effect show up in the file.

Anti-Pattern Name: Cause without Effect

Aliases: input only

Context

Input is recorded externally to the expected output.

Problem

Matching the expected output with the input is error-prone during maintenance.

Forces

Solution

Record expected output with the input.

Rationale

It is easier to review for correctness when the inputs and outputs are together.
It is easily verified if each effect occurs due to its cause.
Additional inputs can be easily added since their expected output is recorded with them.

Code Samples

Input file: 1 4 9 –1 0

Test code:

```
For I=1 to 3; do get num;
      if (square(squareRoot(num)) != num) print “fail $num”;
      done
For I= 1 to 2; do get num;
      if (squareRoot(num) != “illegal”) print “fail $num”;
      done
```

Note that the test code (internal effects) is tightly tied to the input (external cause) and changing either creates test (not product) failures. This is a very brittle coding style.

Better, using *Data-driven data* pattern is:
Input file:

```
        1  1
        4  2
        9  3
       -1  illegal
        0  illegal
```

Test code:

```
While read in_num, out_result; do
      if (out_result = “illegal”)
      then if (squareRoot(in_num) != “illegal”) print “fail $in_num”;
      else if (squareRoot(in_num) != $out_result) print “fail $in_num”;
      done
```

This prevents the brittle code and is easily expandable. You can add additional test cases by changing the input file without changing the test code. This is an example of Data-Driven Data.

Pattern Name: *Check as you Go*

Aliases: In-Line check

Context

You have pre-specification of the test results.

Problem

Do you compare actual results to known expected results at each step as you go, or all at the end?

Forces

- Future results may be invalid if early results don't pass (see also *Separable Tests*).
- Data from the environment is dynamically needed to evaluate correctness.[3]
- Correctness requires specific relationships to occur, for example complex data structures like trees.
- Either the checks are ve ry cheap to make or the test is not highly performance sensitive, that is, you can afford to spend time to do the checks during the test.

Solution

Check each result in-line as finely as possible immediately after its inputs are submitted. If a validation fails then log the failure and optionally don't proceed forward with the rest of the test. For example, when bounding the time of the result, if a connection isn't made within a timeout period, abort the test rather than waiting to get more output.

Indications

- The desired results of a test input are precisely known ahead of time.
- The test is programmable, that is, the result of each set of inputs is easily checked.
- The test is long running, and could be meaningless if there is an early discrepancy for one of the post-conditions.

Rationale

It generally aids comprehensibility of the tests if the expected results appear in the same file and as close to the inputs as possible.

Resulting Context/Consequences

Complete list of post-conditions being checked may be spread throughout the test code.

Related Patterns

See *Batch check* for the same problem, but different forces.

Examples/Known Uses

Frequently used for API/Class Drivers approach.

Junit – See http://www.junit.org/
Expect tool – See http://expect.nist.gov/
POSIX Verification Test Suite – See http://www.opengroup.org/testing/downloads/vsx-pcts-faq.html

Code Samples

Note below that the result is checked as you go in the code and not by some external entity.

[3] For example, you want to verify the timestamp on a log record. You can print the time before and after you expect the log record, but now your batch comparison requires relative checks (less than and greater than) instead of just equals. This is usually a significantly more difficult comparison algorithm.

Java/C++

```
	result = squareRoot(1);
	if (result != 1) {
		LogError( “squareRoot(1) resulted in “+result
			+” where 1 was expected” )
		}
	result = squareRoot(4);
	if (result != 2) {
		LogError( “squareRoot(4) resulted in “+result
			+” where 2 was expected” )
		}
```

Shell

```
	result=`squareRoot 1`
	if [ “$result” != “1” ] ; then
		echo “squareRoot 1 resulted in $result, where 1 was expected.”
	fi
	result=`squareRoot 4`
	if [ “$result” != “2” ] ; then
		echo “squareRoot 4 resulted in $result, where 2 was expected.”
	fi
```

Pattern Name: *Batch check*

Aliases: *Benchmark, baseline, golden results, canonical results, gold master*
- where a benchmark file containing expected results is used.

Context

You have pre-specification of the test results or the specification may be via the pattern *Judging* actual results when expected results are not necessarily known.

Problem

Do you compare results at each step as you go or all at the end?

Forces

- The set of inputs is not easily separable (for example compiler input file).
- The output can be compared easily with minimal filtering.
- The checks are expensive to make or the test is highly performance sensitive and it is relatively cheap to just record the results.

Solution

Provide a benchmark file of expected results. Collect actual results as the test executes. At the end compare the expected and actual results.

Indications

- A failure near the start of the test doesn't invalidate the results that follow.

Rationale

Tests are very easy to develop. Expected results can be generated by the program once, hand-checked for accuracy once, and then reused again and again. This changes the *Judging* pattern into *Solved Example*. Expected results can be updated without affecting any code (since they are in a separate file). Batch processing may be the nature of Item Under Test (IUT).

Resulting Context/Consequences

One dangerous Consequence frequently seen is testers get lazy when the expected output has to change and don't scrutinize the initial results carefully enough for correctness. In this case, the incorrect actual output gets canonized as the expected output. See Test Automation Snake Oil at http://www.satisfice.com/articles/test_automation_snake_oil.pdf

Batch check can make maintenance more difficult *if* the relationship between inputs and outputs is not very clear.

Frequently special filtering patterns (regular expressions) are needed to ignore uncontrollable extraneous differences, for example machine names, time stamps, etc. This filtering has a small risk of missing incidental problems, such as the time being reported incorrectly. Generally you rely on other tests to specifically verify what most of these types of tests ignore.

Related Patterns

See *Check as you Go* as an alternate method.

Examples/Known Uses

Compiler testing or any transformation type program. It is generally too expensive to test each feature completely individually, and a great deal of common setup exists to test any one feature.

Code Samples

The abbreviated example below shows the expected output stored in a separate file and then a batch comparison done.

Shell:

```
cat <<EOINPUT >|expectedOutput
input output
1 1
4 2
EOINPUT

# Set up for read from fd=4 with above data
exec 4<expectedOutput

read -u4 input_value?"headings " output_value
echo $input_value $output_value >| actualOutput
while read -u4 input_value?"input and output" output_value; do
   echo "$input_value \c" >> actualOutput
   squareRoot $input_value >> actualOutput
done

diff expectedOutput actualOutput
```

Pattern Name: *Separable Tests*

Aliases: *Atomic Tests, Independent Tests*

Context

You know the nature of the tests you want to run and easily identify exactly which existing tests or subset of tests you need.

Problem

How do you run as few tests as possible and as small a set of tests as possible if you know what you want to test?
How can you run tests in any order if they each require a different setup?

Forces

- Tests may have setup dependencies.
- Tests should be as small and specific as possible.
- Tests need to be efficient and not repeat operations excessively.

Solution

Provide as fine grain a selection mechanism as possible. For API tests this means being able to select tests within a program via GUI, command line, or environmental options. Provide many ways to categorize tests (see *Test Keywords Management* in the Appendix).

Indications

- Developers or Managers request: can we run a test that just does X?
- Tests require different resources (for example network connections or database access)

Rationale

Quicker reruns of tests are usually possible if only a single small test must be run.
When development is doing defect reproduction or review, it is easier to understand small tests rather than large, multi-condition tests.

Resulting Context/Consequences

- ❑ Tests can be independently run based on each test's unique characteristics.
- ❑ Separating into very point-specific tests reduces the chances of serendipitous findings and may also avoid testing of any interactions – both of which can reduce bug-finding abilities of tests.[4]
- ❑ Running several small tests may take more time than running one large test. At a minimum there may be extra data recording time (start, stop, success or failure) for each small test.
- ❑ If a large test is broken into several smaller tests:
  - ❑ it is easier to run them in different orders, which can increase the chance of finding defects.
  - ❑ it is possible to run tests which might have been blocked by an early failure in the big test
  - ❑ there is a cleaner isolation of pass/fail for each feature

Related Patterns

Smart Dependency Checking describes methods of stating and satisfying test dependencies.

Test Keywords Management (in the Appendix) describes methods of selecting separable tests.

Examples/Known Uses

TET (http://tetworks.opengroup.org/) provides `tet_testlist` to allow "invocable components" within a program to be separately run.
Rational Test Manager – See http://www.rational.com/products/testmanager/index.jsp

[4] *Craft of Software Testing* by Brian Marick

Pattern Name: *Smart Dependency Checking*

Context

A test requires same environment setup as provided by another test. Conversely, a large test can be broken up into parts such that the early parts could be run without running the later parts and still be useful tests.

Problem

How do we allow running of any arbitrary test if the test requires other tests to run before it?

Forces

- Tests need to be efficient and not repeat operations excessively.

Solution

Provide methods of stating and satisfying test dependencies. Tests must indicate any dependencies required, and the execution harness must order tests to meet the dependencies.

Indications: Tests share a lot of common code that each must execute.

Rationale

By having each test state its dependencies, an intelligent execution harness can order the tests to satisfy the dependencies and run as few tests as possible.

Resulting Context/Consequences

Dependencies between tests are automatically accommodated.

Examples/Known Uses

TestFrameWork provides the `requires()`[5] method to automatically build the tree of required tests. For example a test of some data storage mechanism might have three test methods, `testBind()`, `testLookup()` and `testUnbind()`. Obviously you want to run the bind test first; the lookup test second; and the unbind test last and only if the bind test passed. To do this you'd write the methods this way. If you only require a test method to have had a chance to run, but not necessarily to have passed, you can prefix the test name with a '%'.

```
  public boolean testBind() { ... }
  public boolean testLookup() { require("Bind"); ... }
  public boolean testUnbind() { require("Bind,%Lookup"); ... }
```

Code Samples

A test for exceeding the database locks could be as shown in *Move actual to Internal*.
However, the database test would typically be decomposed into two tests: the first test to verify the expected maximum number of locks can be reached and a second test to verify the error condition. This would result in the overflow test being dependent upon maximum number of locks. But, the MaxLocks test can be run independently of the ExceedLocks test if that is all that is needed for testing.

```
BeginTest MaxLocks
for I=1 to numlocks {getDBlock();}
EndTest
BeginTest ExceedLocks requires(MaxLocks)
try { getDBlock(); logFail(); }
  catch (TooManyLocksException e){/*Got expected exception*/}
  catch (Exception e) { logFail(); };
System.exec("diff DBlog expectedDBlog");
EndTest
```

[5] "*Creating a Testing Culture*" by Keith Stobie, Quality Week 1999

Pattern Name: *Whole function*

Aliases: ***All behavior***

Context

An Item Under Test (IUT) has multiple post-conditions associated with it. Is each post-condition a separate test?

Problem

What does it mean for a test to pass, or how fine-grained should comparisons be?

Forces

- Comparisons must be fine-grained.
- Tests must not report false positives (a successful execution when the product doesn't actually work).

Solution

Verify all of the post-conditions associated with a function being tested before considering a test as having shown the function passing.

Indications

Function has multiple post-conditions.

Rationale

If only one post-condition is checked, then a contradiction between post-conditions may not be detected, thus missing a defect!

Resulting Context/Consequences

A test can show a failure for multiple reasons (at least one per post-condition).

Related Patterns

Typically used in conjunction with *Separable Tests*.
See the anti-pattern **Pass Each Post-Condition**.

Examples/Known Uses

TET distinguishes between a "Test Purpose" and an "Invocable Component" which may have several test purposes, but can't be run separately.

Anti-Pattern Name: Pass Each Post-Condition

Context

An Item Under Test (IUT) has multiple post-conditions associated with it. Each post-condition is considered a separate test.

Problem

How to indicate that each post-condition has been checked?

Forces

- Management likes to see lots of tests.
- A post-condition can be thought of as a test.

Solution

Print pass or fail after each post-condition check or after each comparison.

Indications

Function has multiple post-conditions.

Rationale

The reasoning for not considering each post-condition as a test is as follows:
If the second test shows the second post-condition as failing, then it might be incorrect to say that the first post-condition in the first test succeeded. Yet, as written, the tests will indicate you should get a successful first post-condition even if the second post-condition fails.

For example:

```
BeginTest
Insert database record        # Test operation
Verify successful return code # post-condition1
EndTest
BeginTest
Retrieve same database record
# post-condition2 or test of retrieve?
Verify actual retrieved record matches (expected) input record.
EndTest
```

The reasoning for not considering these as two tests is as follows:
If the second test shows the database record is missing, then it might be incorrect to say that the insert in the first test succeeded and that the return code should have been a successful one. Yet, as written, the tests will indicate you should get a successful return code from a failing insert and say the Retrieve (Find) function has failed!

Resulting Context/Consequences

Functions are considered partially passing when they present inconsistent post-conditions, that is, there exists a “test” of the function that succeeds.

Related Patterns

See *Whole function* for correct usage pattern.

Code Samples

A typical example looks like:

```
insert (expectedRecord)
if insert doesn’t fail, then print PASS
else print FAIL
```

The other post-condition, that the insert had the desired effect is left to another test!
For example:

```
read (actualRecord)
if (expectedRecord != actualRecord)  the print PASS
else print FAIL
```

and the failure might be ascribed to the `read` instead of the `insert`. In fact, without additional tests, it is impossible to distinguish whether it is the `read` or the `insert` that failed. A good failure message in this case would be something like,

> “Read of actual record:<actual record> didn’t match expected record: <expected record> that was inserted.”

Appendix

To be written:
Test Keywords Management describes methods of selecting separable tests.

Logging strategies. – only logging on failure. Include and identify expected and actual results.
Known failure – providing a three way result Pass, Fail-known, Fail-unknown instead of typical Pass/Fail

From *Testing Object-Oriented Systems: Models, Patterns, and Tools* (*http://www.rbsc.com/TOOSMPT.htm*):

Oracle Patterns (micro-pattern schema)

Approach	Pattern Name	Intent
Judging	Judging	The tester evaluates pass/no-pass by looking at the output on a screen or at a listing, or by using a debugger or another suitable human interface.
Pre-Specification	Solved Example	Develop expected results by hand or obtain from a reference work.
	Simulation	Generate exact expected results with a simpler implementation of the IUT (e.g., a spreadsheet.)
	Approximation	Develop approximate expected results by hand or with a simpler implementation of the IUT.
	Parametric	Characterize expected results for a large number of items by parameters.

Test Automation Design Patterns

Capability	Pattern Name	Intent
Built-in Test	Percolation	Perform automatic verification of super/subclass contracts.
Test Cases	Test Case/ TestSuite Method	Implement a test case or a test suite as a method.
	Catch All Exceptions	Test driver generates and catches IUT's exceptions.
	Test Case / Test Suite Class	Implement test case or test suite as an object of class TestCase.