Test Result Handling

Determining how a test case detects a product failure requires several test case design trade-offs. These trade-offs include the characteristics of test data used and when comparisons are done. This document addresses how result checking impacts test design. The following questions are grouped into the areas addressed by this document about where and when do you check results.

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 [MESZ] and anti-patterns to describe the details of why you choose a particular test result handling method to automate a test design. A **pattern** is a solution to a problem in a given context. Each pattern is a three-part rule, which expresses a relation between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain software configuration which allows these forces to resolve themselves. An **anti-pattern** is a pattern that tells how to go from a problem to a bad solution. A good AntiPattern also tells you why the bad solution looks attractive (e.g. it actually works in some narrow context), why it turns out to be bad, and what positive patterns are applicable in its stead. A well know Anti-Pattern is Big Bang Testing.

Context

You are designing tests and need to consider how to check the results of the test.

If you don't agree that conceptually all tests have the three parts below, then you won't find these patterns helpful.

- 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 or **post-condition** of a set of **effects** associated with the software being tested.

The above description and these patterns apply to the three most common testing interfaces: Graphical User Interfaces (GUI), Command Line Interface (CLI), and Application Programming Interface (API). Most of the examples show CLI and API tests because of their brevity. Most automated GUI tests are tightly related to a test tool, typically Capture/Replay.

You are choosing how to automate tests that have been designed and need to make the tests as understandable and maintainable as possible.

If your test oracle is composed of stored expected output and a test result oracle which compares expected and actual output, then the storage of the expected output is an issue you must deal with in creating an Automated Test Oracle.

This paper doesn't apply to ad hoc testing, exploratory testing or testing without an Oracle.

Test Data Location (data patterns):

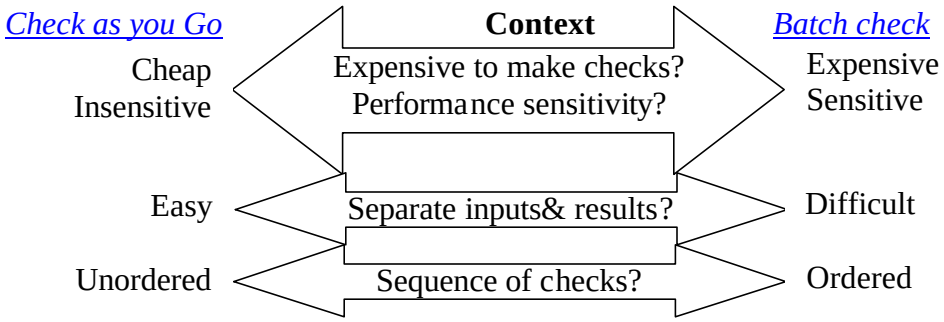
The input, expected output, and actual output may be coded directly into the test (program code, test script, etc.) or apart from it (input file, expected result database, etc.). The *Co-locate Data* pattern proposes that we make the data all internal or all external for increased readability, understandability, and maintenance, and compares contexts for further pattern choosing.

Co-location of the data results in either the *Self-contained data* pattern for all internal data or *Data-driven data* pattern for all external data. When the input and expected output are co-located, but not the actual output, then the helper patterns *Move actual to Internal* and *Move actual to External* facilitate 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 expected output.

Comparison Timing (check patterns)

You can *Check as you Go* the post-conditions or *Batch check* them depending on the context. While *Batch check* is frequently used, it has several potential problems that mean sometimes it is chosen incorrectly as the pattern to follow.



Forces on Repeatable Tests

- Repeatable tests require a clear set of inputs and expected outputs, and an understanding of how to generate actual outputs.
- Comparisons of actual and expected outputs must be efficient, preferably both in time and space. Efficient in space means using very little data to represent the expected output. Efficient in time means using the most efficient comparison algorithms and avoiding unnecessary comparisons.
- Clarity of which actual and expected outputs don't match is paramount.
- Avoiding false positives is more important than avoiding false negatives:
  People usually question and investigate a failure, so if it is false it will be discovered[1].
  People rarely question (or audit) the desired outcome (Pass) to see if it really was correct[2].

		Actual Output: **Correct**	Actual Output: **Incorrect**
Test Result	**Pass**	OK	False positive
	Fail	False negative	OK

- Hard-coding data in a test makes it difficult to update if the software under test changes.
- Input and Output are naturally separated streams and are usually not mixed.
  For example, the input is located in a file separate from the output in the Unix filter style of transformations.
- Placing data externally makes it easier to lose or to get out of synchronization with the test code. It also means reviewers have two things to look at and compare to verify correctness.
- Constantly interspersing checking code in testware can make the logic of a test case unclear. This is similar to the error checking code in software. One reason for adding exception handling to modern languages is to make the main logic clearer.
- Most software changes over time, producing new (different) actual output.
  Expected output must change to match changes in software output.

[1] However, a common problem is a known failure that continues to occur because its fix has been deferred. Some people stop running the test or discount its result. Now if a new type of failure is discovered, it will be ignored since the test is already known to find a failure. This can be avoided by distinguishing between unexpected (or new) failures and known (or old) failures.

[2] False positives are usually discovered only when a failure is noticed in the field and is traced back to a test reporting passed (when it should have reported failed). This is of course way too late – the test didn't do its job.

Test Maintenance cost is frequently ignored or deferred.
For repeatable tests, Test Maintenance cost should be a consideration.

- Some software behavior is timing dependent. Adding checks slows down tests which may affect their ability to trigger the timing dependencies of the software.

Pattern Summary

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***

The first pair of patterns for test data location conceptualize where the input and outputs are stored.

The second pair of patterns are complementary comparison timing patterns for solving the same problem of when to check. *Check as you Go* is the generally preferred method for ease of understanding. *Batch check* (or *Benchmark*) is particularly useful for changing the manual judging of result into a solved example [see Related Patterns in Appendix].

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

Pattern Usage Example

The table below contrasts bad and good practice. On the left is the frequently followed anti-pattern **Pass Each Post-Condition** solution and not following the *Co-Locate data* pattern. On the right is a solution following *Whole Function*, *Check as you Go* and *Self-contained data* patterns. An alternative solution using *Batch check* and *Data-driven data* patterns is also shown.

Problematic solution	**Pattern-following solution**

`BeginTest`
# Test operation: 	
`Insert database record`
`Verify` `successful return code`
# post-condition1: 	
`EndTest`
 	
`BeginTest`
`Retrieve same database record`
# post-condition2: 	
`Print actual retrieved record`
`Verify` `actual printout matches`
`  expected printout`
`EndTest`
	`BeginTest`
	`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`

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

Notice that there are two verify steps in the *Check as you Go* case, right column, above. Either verify step can cause the test to mark the feature as failed. The timing and number of verifies is not prescribed. It is not even required that 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 necessary comparisons are successful. 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.

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

You are creating reusable tests and must store the test data for usage from one test run to another.

Problem

Where do you store the test data? Inputs come from various sources including the test script, 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.
- It is easiest to keep data where it naturally occurs (in code or external to the code).
- Reviewers need to see input and output together to judge correctness.

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 (perhaps using *Move actual to Internal* to get *Self-contained data*) or the centralized external location (perhaps using *Move actual to External* to get *Data-Driven data*).

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.

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**.

Resulting Context/Consequences

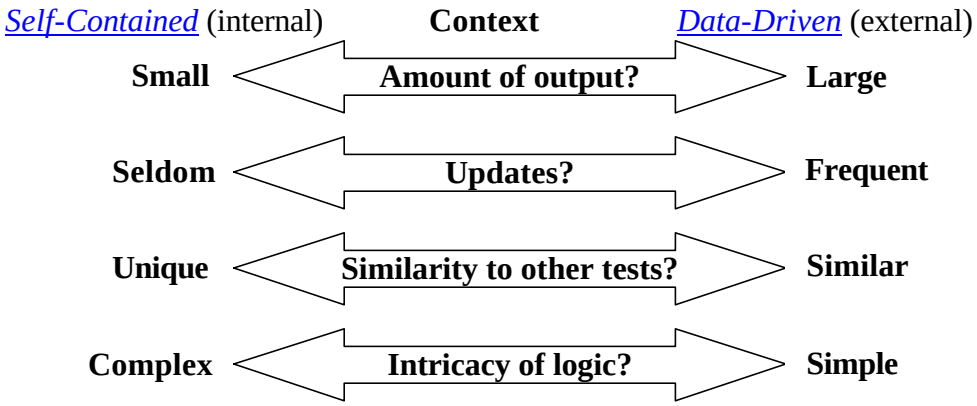
Verification of the correct expected output is easier since it is all in one place.
Filtering logic may be needed to separate input from output.

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 the bad solution of having the input not co-located with the expected output.
The table below lists patterns in (blue for link) *italics* and anti-patterns using **bold blue** 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*

Context affecting choice of *Self-contained* versus *Data-driven data*:



Pattern Name: *Self-contained data*

Aliases: *internal data*

Context

You are creating a repeatable test where the test data must be preserved. The test design indicates amount of test data and other tests that might be similar. The test is relatively unique in it its parameters or sequences of actions (e.g. navigating in a GUI).
The *Move actual to Internal* pattern may already have been applied to transform actual output from external to internal to match the other data.

Problem

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

Forces

- Putting all pieces of test in one place makes it easier to find them.
- Reviewing is easier when cause and effect are clear.
- Data hard-coded within the test increases future maintenance effort.
- Data hard-coded within the test reduces future reusability of the test.
- External data can be hard to review and can get out of synchronization with the code.

Solution

Because of the relative uniqueness of the test data or its small size, we can emphasize putting all pieces in one place and ignore the reusability and future maintenance costs as they are unlikely to have a great effect in this context. Include the data in the test script or test code. You may still wish to avoid totally hard-coding the data by using constant names or literals within the code. Thus MaxInt is still better than 32,767 because it conveys more meaning.

Indications

The amount of input and output per result is relatively small, easy to understand and verifiable with the test.

Rationale

Test script logic becomes less complicated or more obvious when data 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. It is easier to run the test because if has fewer dependencies.
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

Frequently used with *Check as you Go*.
Contrast with *Data-driven data* which has same problem, but different context (as illustrated in *Co-locate data*).

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.

```
BeginTest
Insert database record                  # database record value in code
Verify successful return code           # successful value in code
Retrieve same database record
# expected output same as input database record from the code:
Verify actual retrieved record matches (expected output) input record.
EndTest
```

Pattern Name: *Data-driven data*[3]

Aliases: *external data*

Context

You are creating a repeatable test where the test data must be preserved. The test design indicates amount of test data and other tests that might be similar. You have many data combinations to be tested. The output may be voluminous. The output may be difficult to predict and can frequently change from release to release. The test data shouldn't intricately drive the test logic.
The *Move actual to External* pattern may already have been applied to transform actual output from internal to external to match the other data.

Problem

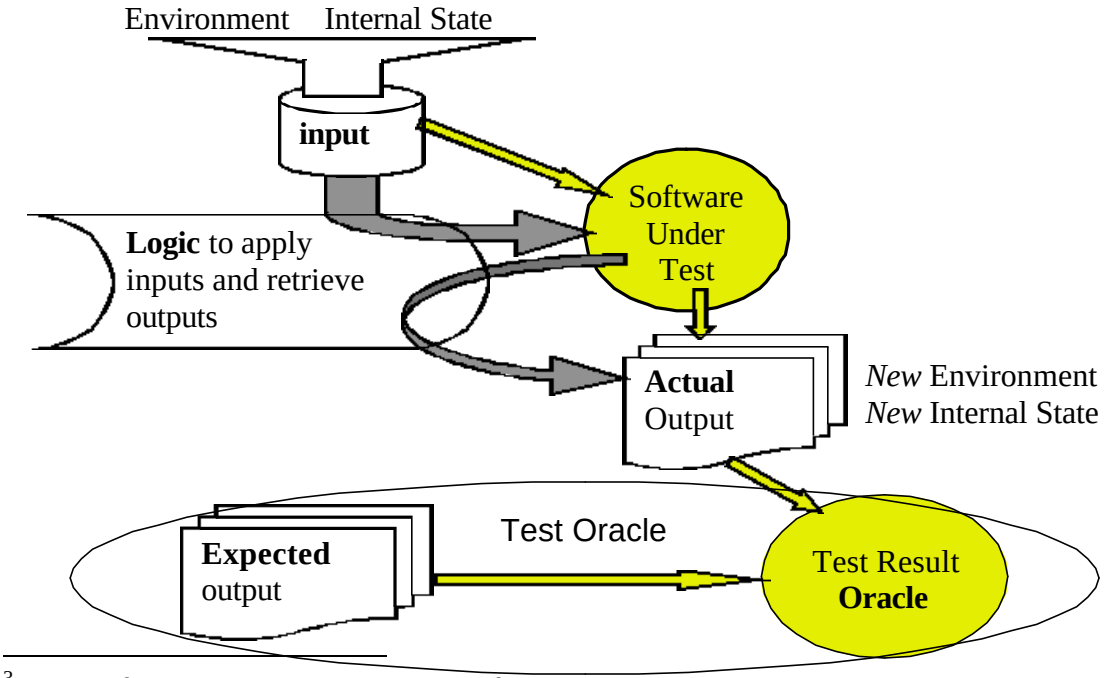
Where do you keep the test input data and test expected output data?

Forces

- Putting all pieces of test in one place makes it easier to find them.
- Reviewing is easier when cause and effect are clear.
- Data hard-coded within the test increases future maintenance effort.
- Data hard-coded within the test reduces future reusability of the test.
- External data can be hard to review and can get out of synchronization with the code.

Solution

Because of the amount of change or reuse anticipated, it is best to avoid hard-coding the data. Separate test data from scripts. This makes it easier to create multiple related tests. The greater difficulty in finding the pieces is more than made up by lower maintenance and creation costs.



[3] Much of this material is a derivation from *Data-driven testing* pattern by Bret Pettichord and Paul Szymkowiak presented at PoST 1, Jan. 2001

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. Sometimes, the test script logic may become more complicated or less obvious when data is separate. Other times there is cleaner indication of test data versus the test logic.

Related Patterns

Frequently used with *Batch Check*. Contrast with *Self-contained data*.

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) as protocols evolve or the implementation improves, e.g. adding caching may cause the stubs to change from version to version.

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*.

```
BeginTest
Read input for database record # External input
Insert database record
Print return code              # internal input converted to external
Retrieve same database record
Print actual retrieved record  # External output created
Verify actual printout matches expected printout
EndTest
```

Anti-Pattern Name: Effect without Cause

Aliases: ***output only***

Context

Expected output is recorded without knowing the input. Output is in separate stream from input. Output changes more frequently than the input (e.g. each release of a product might have different output for the same input).

Problem

Outputs can match, but for the wrong reasons.

Forces

- Avoiding false positives is more important than avoiding false negatives.
- Store output separately from input to ease maintenance of changing output.

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 as external files, this input is separate from the expected output file. Correct way is shown below::

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

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

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.
Data from the environment is dynamically needed to evaluate correctness.[4]
Either the checks are very 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.
Correctness requires specific relationships to occur, for example complex data structures like trees or receiving an asynchronous event within a specific time period.

Problem

Do you compare actual results to 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.
- Special code may need to be run to gather diagnostic info depending on the failure.
- Output data may have specifics (e.g. node name or time) the test doesn't care about.
- Constantly interspersing checking code in testware can make the logic of a test case unclear.

Solution

Check each result or post-condition in-line as finely as possible immediately after its inputs are submitted. You may need to incorporate current specific data to have the results compare.
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. This concept applies both to testing post conditions within an individual test case and execution of test cases within an automated test suite. A major feature of *Check as you Go* is to only log the software under test as passing after all relevant post-conditions have been checked. Typical GUI Testing using *Check as you Go* might look like:

```
Produce Screen1
        Verify Screen 1
Produce Screen2
        Verify Screen 2
  . . .
```

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.

[4] 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.

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 context.

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.

Java/C++

```
result = squareRoot(1);
if (result != 1) {
     LogFail( "squareRoot(1) resulted in "+result
            +" where 1 was expected" )
     }
     else LogPass( "squareRoot(1)" );
result = squareRoot(4);
if (result != 2) {
     LogFail( "squareRoot(4) resulted in "+result
            +" where 2 was expected" )
     }
     else LogPass( "squareRoot(4)" );
```

Shell

```
result=`squareRoot 1`
if [ "$result" != "1" ] ; then
     logFail "squareRoot 1 resulted in $result, where 1 was expected."
else
     logPass "squareRoot 1"
fi
result=`squareRoot 4`
if [ "$result" != "2" ] ; then
     echo "squareRoot 4 resulted in $result, where 2 was expected."
else
     logPass "squareRoot 4"
fi
```

Pattern Name: *Batch check*

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

Context

You have pre-specification of the test results or you can manually judge the output for correctness when exact expected results are not necessarily known. The post-conditions are independent, that is don't use when you need to check one post-condition before choosing to check another one. 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. Don't need to check result of each step to determine if future steps of the test are valid.

Problem

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

Forces

- Future results may be invalid if early results don't pass.
- Special code may need to be run to gather diagnostic info depending on the failure.
- Output data may have specifics (e.g. node name or time) the test doesn't care about.
- Constantly interspersing checking code in testware can make the logic of a test case unclear.
- Avoiding false positives is more important than avoiding false negatives

Solution

Provide a benchmark file of expected results. Collect actual results as the test executes. At the end, compare the expected and actual results. Use filtering programs to ignore specifics the test doesn't care about.

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, then reused again and again. 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 [BACH] creating a false positive! A way to ameliorate this is to occasionally have an independent person spot check the results.

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 tests are ignoring.

Related Patterns

See *Check as you Go* with an alternate solution due to different context.

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.

An example of expensive checks goes like this:
The author was testing a new transactional capability of a database. The transactions allowed the commit or abort of a set of operations (insert, update, delete). The original test was written to update the database and keep a separate file of the expected results. The extra logging of expected results allowed enough time that many race conditions in the transaction system were missed when several transactions were supposed to be occurring in parallel. The *Batch check* solution was to rely on the concept of database integrity via a constraint. A particular column had the constraint that it was always divisible by 10,000. Transactions could do a series of operations such that the constraint would continue to hold if all or none of the operations occurred. For example, insert record A with column value of 5,000 and update record B's column value by adding 5,000. This allowed large numbers of concurrent transactions to be run at once. The batch check was to verify the integrity of the column constraint at the end of a set of transactions.

Code Samples

The abbreviated example below shows the expected output stored in a separate file followed by a batch comparison. Pattern Name: *Data-driven data* has an illustration typical of Batch check where the test executes first gathering output and then results are compared.

Shell:`cat <<EOINPUT >|expectedOutputinput output1 14 2EOINPUT# Set up for read from fd=4 with above data`

```
exec 4<expectedOutput

# Read (heading) line from expectedOutput file into input_value & output value
read -u4 input_value?"headings " output_value
# Put heading line in actualOutput file
echo $input_value $output_value >| actualOutput
# While lines to read, put value into input_value & output_value
while read -u4 input_value?"input and output" output_value; do
    # Echo input to actualOutput (without a linefeed/newline)
    echo "$input_value \c" >> actualOutput
    # Put actual result on end of previous line
    squareRoot $input_value >> actualOutput
done

diff expectedOutput actualOutput
```

Appendix

Inconsquential Pass (unwritten) describes how input data and test actions must be set up to distinguish if test actions really had any effect, and not whether the result is coincidentally correct.

Related Patterns
The patterns derived from *Co-Locate Data* 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* [BIND]

Batch check (or *Benchmark*) is particularly useful for changing the Oracle *Judging* pattern into *Solved Example.*

From *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.

Acknowledgements:

Luke Hohman inspired me to write the first draft in my Design Patterns course. A very early draft of this was workshopped at [PoST1] and a revised draft was workshopped at [PoST2] by Paul Szymkowiak, Melissa Mutkoski, Jennifer Smith-Brock, Florence Mottay, Nadim H. Rabbani, Amit Singh, Scott Chase, Cem Kaner, and Bret Pettichord. I also thank Brian Marick and the other PoST sponsors for setting up the PoST forum.

References

[BACH] "*Test Automation Snake Oil*" by James Bach
http://www.satisfice.com/articles/test_automation_snake_oil.pdf

[BIND] *Testing Object-Oriented Systems: Models, Patterns, and Tools* by Robert Binder. (*http://www.rbsc.com/TOOSMPT.htm*)

[HOFF] *Oracle Strategies for Automated Testing* by Douglas Hoffman,
Quality Week (tutorials) 2000
1
[MARI] *Craft of Software Testing* by Brian Marick

[MESZ] "*A Pattern Language for Pattern Writing*" by Gerard Meszaros and Jim Doble at
http://www.hillside.net/patterns/Writing/patterns.html

[PoST1] Patterns of Software Testing workshop1, Jan. 2001, Lexington, MA

[PoST2] Patterns of Software Testing workshop2, April. 2001, Melbourne, FL

[TET] Test Execution Toolkit http://tetworks.opengroup.org/
Documentation: http://tetworks.opengroup.org/download.html

Keith Stobie

Keith plans, designs, and reviews software architecture and tests for Microsoft. Keith directed and instructed in QA and Test process and strategy at BEA Systems. His most recent project was BEA WebLogic Collaborate, and previously WebLogic Enterprise. Keith was Test Architect at Informix, designing tests for the Extended Parallel Server product, and Manager of Quality and Process Improvement. With over 20 years in the field, Keith is a leader in testing methodology, tools technology, and quality process. He is a qualified instructor for Systematic Software Testing and software inspections. Keith is active in the software task group of ASQ, participant in IEEE 2003 and 2003.2 standards on test methods, published several articles, and presented at many quality and testing conferences. Keith has a BS in computer science from Cornell University.

ASQ certified Software Quality Engineer,
Member: ACM, IEEE, ASQ

Keith Stobie
Test Architect

16541 Redmond Way PMB 321-C U6
Redmond, WA 98052-4482
e-mail: sqe@stobie.org