The Tree of Root Cause

Project Test Smells

a smell that is usually noticed by the project manager or the customer, who does not look at the test code or run the tests

Developers Not Writing Tests

we hear that our developers aren't writing tests

Not Enough Time
developers may have trouble writing tests in the time they are given to do the development

Root Cause
overly aggressive development schedule

Hard-to-Test Code
see **Code Test Smells**

Root Cause
supervisors/team leaders who instruct developers 'Don't waste time writing tests

Wrong Test Automation Strategy
test automation strategy that leads to Fragile Tests or Obscure Tests that take too long to write

Root Cause
ask the 5 why's

Production Bugs

we find too many bugs during formal test or in production

Infrequently Run Tests
we hear that our developers aren't running the test very often

Root Cause
slow tests that slow down the pre-integration regression testing

Root Cause
Unrepeatable Tests that force developers to restart their test environment

Untested Code
just know that some piece of code in the SUT is not being exercised by any tests

Root Cause
SUT includes code paths that react to particular ways that a depended-on component (DOC) behaves and we haven't found a way to exercise those paths

Untested Requirement
we just 'know that some piece of functionality is not being tested

Root Cause
SUT includes behavior that is not visible through its public interface

Buggy Tests

a project-level indication that all is not well with our automated tests

Fragile Test
see **Behavior Test Smells**

Hard-to-Test Code
see **Code Test Smells**

Obscure Test
see **Code Test Smells**

High Test Maintenance Cost

test code needs to be maintained along with the production code it verifies

Fragile Test
see **Behavior Test Smells**

Hard-to-Test Code
see **Code Test Smells**

Obscure Test
see **Code Test Smells**

Lost Test
The number of tests being executed in a test suite has declined (or has not increased as much as expected).

Root Cause
Test Method or Testcase Class that has been disabled or has never been added to the AllTests Suite

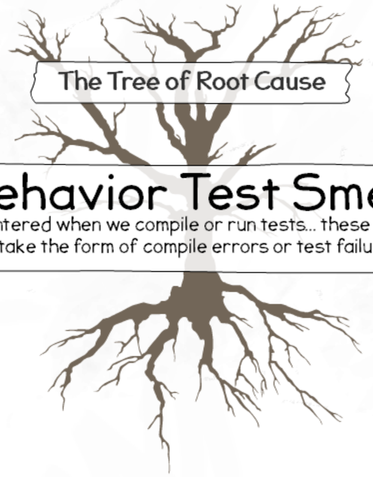
Missing Unit Test
all the unit tests pass but a customer test continues to fail

Root Cause
when a team focuses on writing the customer tests but fails to do test-driven development using unit tests

Neverfail Test
we just 'know' that some piece of functionality isw not working, even though the tests for that functionality pass

Root Cause
caused by improperly coded assertions such as assertTrue(aVariable, true) instead of assertEquals (aVariable, true)

Source:
Gerard Meszaros: xUnit Test Patterns: Refactoring Test Code
Addison-Wesley Professional. May 2007



The Tree of Root Cause

Behavior Test Smells

encountered when we compile or run tests... these smells will take the form of compile errors or test failures

Assertion Roullette

it is hard to tell which of several assertions within the same test method caused a test failure

Root Cause
Eager Test trying to minimize the number of unit tests

Missing Assertion Method

test fails...upon examining the output of the Test Runner, we cannot determine exactly which assertion failed

Root Cause
caused by the use of Assertion Method calls with identical or missing Assertion Messages

Eager Test

a single test verifies too much functionality

Root Cause
trying to minimize the number of unit tests (whether consciously or unconsiously) by verifying many test conditions in a single Test Method

Fragile Test

a test fails to compile or run when the SUT is changed in ways that do not affect the part the test is exercising

Behavior Sensitivity

when changes to the SUT cause other tests to fail

Root Cause
functionality the refression tests use to set up the pre-test state of the SUT has been modified

Root Cause
functionality the regression tests use to verify the post-test state of the SUT has been modified

Root Cause
code the regression tests use to tear down the fixture has been changed

Context Sensitivity

when a test fails becuase the state or behavior of the context in which the SUT executes has changed

Root Cause
the behavior of some other code or system(s) on which the SUT depends has changed

Root Cause
the functionality they were verifying depends in someway on the time or date

Data Sensitivity

a test fails because the data being used to test the SUT has been modified

Root Cause
SUT is being exercised with inputs that reference missing or modiified data

Interface Sensitivity

ask ourselves whether the tests are failing to compile; if so, Interface Sensitivity is to blame

Erratic Tests

one or more tests behave erratically; sometimes they pass and sometimes they fail

Interacting Tests

tests depend on other tests in some way

Root Cause
when several tests use a shared Fixture makes it hard to run tests individually

Resource Leakage

test or the SUT consume finite resources

Root Cause
tests or SUT consume finite resources by allocating those resources and failing to free them afterward

Lonely Test

test can be run as part of a suite but cannot be run by itself

Interacting Test Suites

a special case of Interacting Tests, the tests are in different test suites

Root Cause
when tests in separate test suites try to create the same resources

Slow Tests

the tests take long enough to run th at developers don't run them every time they make a change to the SUT

Slow Component Usage

a component of the SUT has high latency

Root Cause
interacting with a database in many of the tests

Root Cause
using slow components

Too Many Tests

there are so many tests that they are bound to take a long time to run regardless of how fast they execute

Root Cause
having so many tests

Root Cause
running too many of these tests too frequently

General Fixture

tests are consistently slow because each test builds the same over-engineered fixture

Root Cause
constructung a large General Fixture each time a Fresh Fixture is built

Test Run War

test failures occur at randome when several people are running tests simultaneously

Root Cause
globally Shared Fixture that various tests access and sometimes modify

Unrepeatable Test

a test behaves differently the first time it is run compared with how it behaves on subsequent test runs

Root Cause
the use, either deliberate or accidental, of a Shared Fixture

Nondeterministic Tests

test failures occur at random, even when only a single Test Runner is running the tests

Root Cause
using different values each time a test is run

Resource Optimism

a test that depends on external resources has nondeterministic results depending on when or where it is run

Root Cause
resource that is available in one environment is not available in another environment

Asynchronous Test

a few tests take inordinately long to run; those tests contain explicit delays

Root Cause
...because of the variability in how long it takes for these threads or processes to be started, the test usually needs to include a long delat "just in case"

Frequent Debugging

manual debugging is required to determine the cause of most test failures

Root Cause
lack of Defect Localization in our suite of automated tests

Root Cause
missing the detailed unit test that would point out a logic errir inside an individual class

Manual Intervention

person running the test must do something manually either before the test run or partway through the test run

Manual Event Injection

person must intervene during test execution to perform some manual action before the test can proceed

Root Cause
many events in a SUT are hard to generate under program control

Manual Result Verification

we can run the tests but they almost always pass - even when we know that the SUT is not returning the correct results

Root Cause
if the tests we write are not Self-Checking Tests, we can be given a false sense of security because tests will fail only if an error/exception is thrown

Manual Fixture Setup

person has to set up the test environment manually before the automated tests can be run

Root Cause
lack of attention to automating the fixture setup phase of th e test

Source:
Gerard Meszaros: xUnit Test Patterns: Refactoring Test Code
Addison-Wesley Professional. May 2007

The Tree of Root Cause

Code Test Smells

the 'classic' bad smells that were first described by Martin Fowler in 'Refactoring'... these smells must be recognized by test automaters as they maintain test code

Obscure Test

it is difficult to understand the test at a glance.... as long as wee see a green bar, we think we are 'good to go', in reality, we may have created a test that never fails

General Fixture
the test builds or references a larger fixture than is needed to verify the functionality in question

Root Cause
test depends on mysterious external resources, making it difficult to understand the behavior that it is verifying

Mystery Guest
the test reader is not able to see the cause and effect between fixture and verification logic because part of it is done outside the Test Method

Root Cause
test depends on mysteriouis external resources, making it difficult to understand the behavior that it is verifying

Indirect Testing
the Test Method interacts with the SUT indirectly via another object, thereby making the interactions more complex

Root Cause
test depends on mysteriouis external resources, making it difficult to understand the behavior that it is verifying

Irrelevant Information
the test exposes a lot of irrelevant details about the fixture that distract the test reader from what really affects the behavior of the SUT

Root Cause
a test contains a lot of data

Root Cause
when we include all the code needed to verify the outcome rather than using a much more compact declarative style to specify the outcome

Eager Test
test verifiies too much functionality in a single Test Method

Root Cause
thinking in terms of manual testing (it makes sense to group logical assertions together if it's a live person doing the testing)

Hard-Coded Test Data
data values in the fixture, assertions, or arguments of the SUT are hard-coded in the Test Method, obscuring cause-effect relationships between inputs and expected outputs

Root Cause
test contains a lot of seemingly unrelated Literal Values

Root Cause
when we use 'cut and paste' reuse of test logic

Conditional Test Logic

a test contains code that may or may not be executed

Root Cause
use of if statements to steer execution to a fail statement or to avoid executing certain pieces of test code when the SUT fails to return valid data

Root Cause
use of if statements to avoid tearing down nonexistent fixture objects

Root Cause
use of loops to verify the contents of collections of objects

Root Cause
use of Conditional Test Logic to verify complex objects or polymorphic data structures

Root Cause
use of Conditional Test Logic to initialize the test fixture

Production Logic in Test
a form of Conditional Test Logic found in the result verification section of our tests

Root Cause
wanting to verify multiple test conditions in a single Test Method

Flexible Test
the test code verifies different functionality depending on when or where it is from

Root Cause
lack of control of the environment

Root Cause
test automater probably wasn't able to decouple the SUT from its dependencies

Conditional Verification Logic
when the tester tries to prevent the execution

Root Cause
when the tester tries to prevent the execution of assertions if the SUT fails to return the right objects

Root Cause
when the tester uses loops to verify the contents of collections returned by the SUT

Multiple Test Conditions
test tries to apply the same test logic to many sets of input values, each with its own corresponding expected result

Root Cause
trying to test many test conditions using the same test logic in a single Test Method

Complex Teardown
leaves the test environment corrupted if it does not clean up after itself correctly

Root Cause
wanting to verify multiple test conditions in a single Test Method

Test Code Duplication

the same test code is repeated many times

Reinventing the Wheel
accidentally write the same sequence of statements in different tests

Root Cause
lack of awareness of which Test Utility Methods are available

Root Cause
predisposition to write one's own code rather than reuse code written by others

Cut-and-Paste Code Reuse
'cut and paste'... results in many copies of the same code

Root Cause
lack of refactoring skills or refactoring experienced

Root Cause
time pressure

Hard To Test Code

code is difficult to test

Untestable Test Code
body of a Test Method is obscure enough or contains enough Conditional Test Logic that we wonder whether the test is correct

Root Cause
too much trouble to bother with in all but the most unusual circumstances

Highly Coupled Code
class cannot be tested without also testing several other classes

Root Cause
poor design

Root Cause
lack of object-oriented design experience

Root Cause
lack of reward structure that encourages decoupling

Asynchronous Code
class cannot be tested via direct method calls
start an executable... wait until its start-up has finished...

Root Cause
The code that implements the algorithm we with to test is highly coupled to the active object in which it normally executes

Source:
Gerard Meszaros: xUnit Test Patterns: Refactoring Test Code
Addison-Wesley Professional May 2007