Definition and Categorization of Tests for CSE Software

The IDEAS Scientific Software Productivity Project
ideas-productivity.org/resources/howtos/

Table of contents:

Purpose of this Document

This document provides common classification and definitions for tests for CSE software. These definitions and classifications are largely consistent with accepted definitions in the broader software engineering community (for example, as defined on *Wikipedia* and other online sources). The goal is to define a minimal set of definitions and classifications to cover the types of testing performed in many CSE projects. The goal is not to create an exhaustive list of all the types of testing that have ever been defined (such as on the *Wikipedia Software Testing page*).

In addition to defining these categories and tests, some of the consensus views of the broader (agile) software engineering community are injected. These views help to motivate and contrast the different types of tests and help to guide how they can be applied in an effective software development process.

Definitions and Categories of Tests

Tests can be categorized by the *granularity of the test* and the *type of test*. In addition, different types of *analysis tools/tests* can be run using an existing test suite.

Granularity of Tests

Tests can be defined at different levels of granularity. The levels of granularity vary from the smallest *units of the software* to the *entire software system*.

Unit tests are focused on testing individual software units such as individual functions or individual classes. By definition, unit tests must build fast, run fast, and localize errors. Unit tests are considered a foundation for modern agile software development methods (e.g. *test-driven development*) and also provide a foundation for fast efficient development and *refactoring efforts*. In order to make unit testing cost effective, it is important to use a well-designed and easy to use *unit test harness* (e.g. in the style of *xUnit*) tailored to the programming language and particular software being tested.
Integration tests are focused on testing the interaction of larger pieces of software but, not at the *full system level*. Integration tests typically test several different objects from several different types of

This material is based upon work supported by the U.S. Department of Energy Office of Science, Advanced Scientific Computing Research and Biological and Environmental Research programs.

classes together. Integration tests are contrasted from *unit tests* in that they typically don't build as fast, or run as fast or localize errors as well as unit tests. However, these types of more coarse-grained tests may still build and run fast enough to drive effective development and refactoring efforts in many cases (but not localize errors as well and therefore require more debugging effort when they fail).

System-level tests are focused on testing the full software system at the user interaction level. For example, a system-level test of a CFD code would involve passing in complete input files and running the full simulation code, and then checking the output and final solutions (by some criteria). System-level tests on their own are typically not considered a sufficient foundation to effectively and efficiently drive code development and code refactoring efforts.

Types of Tests

There are different types of tests that focus on different aspects of the software.

Verification tests are inwardly focused tests that verify that the code implements the intended algorithms correctly. These tests check for specific mathematical properties or other clear specifications. For example, a verification test for a linear conjugate gradient solver might check that a system with N unique eigenvalues fully solves the system in exactly N iterations (no more and no less). Other numerical algorithms have other special properties that can be tested for as well. By just looking at the test specification and the pass/fail criteria, it is clear that the code is meeting a specific requirement or behavior (as contrasted with *no-change tests*). Verification tests can be written at the *unit*, *integration*, or *system* level.

Acceptance Tests are outwardly focused tests that assert acceptable functioning for a specific customer or set of customers. For example, an acceptance test for a linear solver might be checking the linear solver convergence rates for a particular customer's linear systems. In the CSE domain, *Validation Tests* are a special class of acceptance tests where formal UQ-based methods are applied to validate a code against a specific set of problems for a specific range of conditions (typically using data from experiments). Acceptance/validation tests are usually applied at the *system level*. Acceptance tests are contrasted from the other types of tests defined here in that their focus is on user-level requirements for specific customers of the software. All of the other types of tests are owned by the software package itself.

No-change tests or ***Characterization tests*** simply compare current observable output from the code to some "gold standard" output, typically produced by a previous version of the software. For CSE numerical codes, examples include comparing the number of iterations and the final solution (e.g. on a mesh) to previous outputs for a set of generic test problems. The key difference between a non-change test and a *verification test* is that it is not clear if the code is "correct" by just looking at the definition of the no-change test. For a no-change test, one has to independently "verify" that the previous "gold standard" output from the code is "correct" (in some sense). The key difference between a no-change test and an *acceptance test* is that an acceptance test is targeted for a specific customer and not a generic customer. An extreme form of no-change tests in CSE codes is the binary compatibility test of floating point codes. This form of no-change test can severely hinder code refactoring efforts since any change in the order of floating point operations can change the binary output of floating point calculations. The primary problem with most no-change tests is that when behavior does change for the better (for any definition of "better"), one will often see that these tests have to be "rebaselined" in order to pass. This "rebaselining" is often done without what one would consider sufficient verification of the new updated "gold standard" outputs. No-change tests can be written at the *unit*, *integration*, or *system* level. Reasonably defined no-change tests (i.e. characterization tests) at the unit level are

This material is based upon work supported by the U.S. Department of Energy Office of Science, Advanced Scientific Computing Research and Biological and Environmental Research programs.

considered sufficient to drive software refactoring efforts. It is the higher levels of tests (i.e. integration and system-level) where no-change tests are considered most problematic.

Performance tests focus on the runtime and resource utilization of the software in question. Examples of performance tests include CPU time, CPU cycles, scalability to larger problem sizes, or more MPI processors, etc. This category of test is largely orthogonal from the previously discussed types. That is, all of the verification, validation, and no-change tests can pass but the code can run 10 times slower. As contrasted with simply putting a timer around an existing generic test, the performance tests described here are specifically designed to measure the performance of a particular piece of code or subsystem. Therefore, one has to specifically design these tests as opposed to just running some analysis tool for *memory usage error detection* or *code coverage*. Performance tests can be written at the *unit*, *integration*, or *system* level.

Test Analysis Tools

In addition to specific types of tests that are created, different types of analysis can be performed on a given set of existing executable tests. Some examples of this are *memory usage error detection* and *code coverage*.

Memory usage error detection is run on software written in unsafe languages like C, C++, and Fortran that checks for uninitialized variables, array out of bounds, memory leaks, and other memory usage errors using tools like valgrind, purify, etc. These tools run on any given existing test suite for the software of interest and report any issues found.

Code coverage investigates which lines of code are executed, what logical branches are run, etc. A coverage test tool is run on a given test suite for the software of interest and then the results are displayed for analysis.

Discussion

The *granularity of a test* and the *type (or focus) of a test* are typically independent of each other. For example, a verification test can be applied at the unit level or the system level. Also, while acceptance and validation tests are typically applied at the system level, they can also be applied at lower levels (e.g. unit or integration level) depending on the nature of experimental data, for instance.

A ***regression test suite*** is a set of tests which helps to check that a code is not losing capabilities and behaviors that it had in previous versions of the code (i.e. the code is not "regressing"). Any of the above types of tests (i.e. verification, acceptance, no-change, performance) and granularity of tests (i.e. unit, integration, system-level) as well as different types of test analysis/tools can be included in a regression test suite. In addition, a regression test suite can be defined in incremental pieces for pre-push tests, post-push tests, nightly tests, and weekly tests (e.g. see *nested layers of testing*) A common problem in CSE codes is that almost all of the tests in the regression test suite are non-change system-level tests. Such test suites are not considered by many to provide a sufficient foundation to efficiently and safely drive future development and refactoring efforts in many CSE codes.

Technically speaking, a ***non-regression test suite*** is a set of new tests that are developed to test new functionality. Such tests would include verification tests and acceptance tests (such as with *test-driven development* and *acceptance-test driven development*). If these tests are well defined and

This material is based upon work supported by the U.S. Department of Energy Office of Science, Advanced Scientific Computing Research and Biological and Environmental Research programs.

well automated, then they are good candidates to be added to the regression test suite to protect future development of the software.

Note that *memory usage error detection* and *code coverage* do not define new categories of tests in that one does not write specific memory usage error tests or code coverage tests. Instead, they are important diagnostic tools (especially memory usage error detection) that are run on a code using an already defined test suite.

This document was prepared by Roscoe A. Bartlett and Barry Smith with key contributions from James M. Willenbring, Michael A. Heroux and Ulrike Yang.

This material is based upon work supported by the U.S. Department of Energy Office of Science, Advanced Scientific Computing Research and Biological and Environmental Research programs.