What are Software Testing Practice

The IDEAS Scientific Software Productivity Project

ideas-productivity.org/resources/howtos/

Motivation: Software requires regular extensive testing for several reasons:

- to maintain portability to a wide variety of (ever changing) systems and compilers;
- to allow *refactoring* or the addition of new features into library code that does not unknowingly introduce new errors, or reintroduce old errors; and
- to produce correct results for users.

In this document, we introduce some terminology of testing and discuss the benefits of testing and general approaches to testing.

Types and granularities of testing: Software engineering recognizes two main types of testing (see Definition and Categorization of Tests for CSE Software for full details)

- **Verification testing**: Tests that verify that the code is doing what it is intended to do.
- **No-change (often, perhaps mistakenly, called regression[1]) testing:** Checks that the code produces the same results (to an appropriate approximation) as it previously did. Having comprehensive no-change unit tests enable one to change the internals of the code (refactoring) but quickly verify that the results remain the same.

In addition, three granularities of testing are recognized:

- **Unit tests:** Focus on testing individual software units such as individual functions or individual classes.
- **Integration tests:** Focus on testing the interaction of larger pieces of software but not at the full system level.
- **System-level tests:** Focus on testing the full software system at the user interaction level. For example, a system-level test of a CFD code would involve providing complete input files, running the full simulation, and then checking the output and final solutions.

Managing and reporting on testing: The simplest test system compiles the code and runs one or more executables, saving the output into a text file for the developer or user to examine. Once a package becomes too complex, this approach to testing is no longer satisfactory. **Automatic testing systems** (sometimes called **test harnesses**) lower the (developer) burden of running the tests and adding new tests. For example, filters can be automatically applied to output only the text that indicates problems (e.g., *here* and *here*), and to display on a dashboard (using, e.g., a specific color) which build instantiations generated errors (e.g., *here* and *here*). This approach still requires developers to check a website on a regular basis and bombards them with all errors in the tests, not just those they caused. Arguably, some testing systems parse compiler/linker error output and use revision control system "blame" mechanisms to associate particular errors with particular developers and generate unique email messages to each developer, listing only those errors that they may have caused. If a developer did not cause any errors, then he or she would receive no messages. This approach minimizes the effort required to properly utilize the testing infrastructure.

Most packages use a combination of homegrown scripts and standard tools for running tests and reporting on the test results. Even the standard tools require customization to satisfy the unique

[1] Regression -- a return to a former or less developed state. Thus, *regression testing* is testing to prevent the need to return to a less developed state.

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.

requirements of each package. These custom scripts (e.g., in bash or python) limit reuse between different packages.

- CDash is an open source, web-based software testing server. It sends out emails for all configure, build, and test failures and provides flexible views and various types of queries to filter different builds and tests.
- Jenkins is an application that monitors executions of repeated jobs, such as building a software project, or jobs run by cron. Jenkins focuses on building/testing software projects continuously and monitoring executions of externally run jobs, even those that are run on a remote machine.

In addition to the high-level test drivers described above, there are finer-grained *unit testing frameworks* that enable quick writing and running of many smaller tests. Examples of such frameworks include *Google Test (gtest)* for C++, *Check* for C, and *pfUnit* for Fortran. "Pure" unit tests never touch the file system, do not launch additional processes, and therefore can execute hundreds (if not thousands) of such unit tests in under a second. But unit test frameworks are also useful for running tests that are not "pure" (e.g., reading and writing files or launching other processes).

Testing analysis tools either analyze the source code or run the executables in a special mode to help detect potential problems. For example, **code line coverage** tools determine what portion of possible (code) paths is actually tested. Achieving high code coverage for libraries is often difficult because of the sheer number of possible options the code may support. The Unix tool *gcov* can be used to obtain lines of code that have been covered in a test; but with libraries that may have hundreds of different regression tests, one needs a tool to combine the information obtained from those hundreds of runs with *gcov*. **Valgrind** monitors a running program to detect many errors, including incorrect memory access, memory overwrites, and memory that has not been freed.

Code reviews, although not explicitly a testing process, are closely related and complement testing as a way to find defects. Code reviews are procedures whereby other developers (who did not write a portion of code) examine that code for bugs or style issues. These reviews are sometimes held in a group meeting where the code is displayed with a projector and discussed verbally or done with online tools that allow examining and commenting on code (e.g., with pull requests on bitbucket or github). Some code reviews are a combination of both approaches.

This document was prepared by Ulrike Yang, Roscoe Bartlett, Glenn Hammond, Xiaoye Li, Barry Smith, and Jim Willenbring.

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.