Software Testing Overview

Preprint · May 2022

1 author:

David Young
GDIT & Alabama Supercomputer Center
13 PUBLICATIONS **1,705** CITATIONS
SEE PROFILE

Software Testing Overview

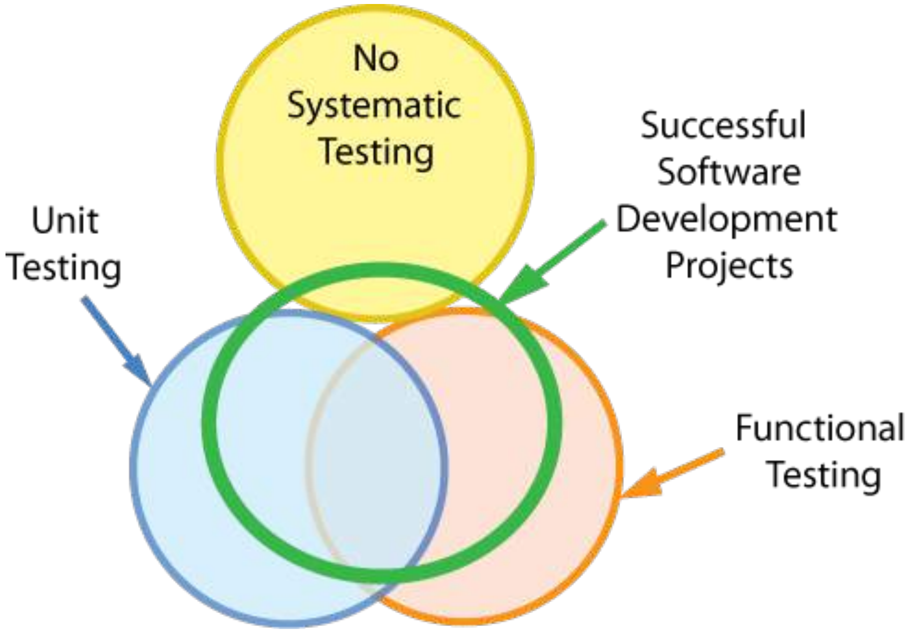
About this white paper: This paper was written by David C. Young, an employee of GDIT. It was written as supplemental documentation for use by the HPC account holders at the Alabama Supercomputer Center (ASC). This was written in 2015 and updated in 2022.

Why is software testing important?

Most people have had the experience of purchasing a product that didn't work, or didn't work correctly. In addition to refunds and complaints from a dissatisfied customer, the customer may avoid additional purchases from the manufacturer for years to come. This loss of current and future revenue can be devastating to any company. In software development, it can be particularly difficult to know what flaws are in the product, so that they can be addressed. Software testing encompasses a wide range of techniques to find those flaws.

To be successful, multiple types of software testing must be done. More importantly, they must be integrated into every stage of the software development process. Too many software development teams have waited until the week before product release to do significant testing, just to find that the software had fundamental flaws that would delay that product release by months.

Software developers are encouraged to use systematic testing processes. These include both unit tests of individual functions and functional tests of the entire program. For most software developers the success of the software in passing tests (either internally or customer usage) is a factor in getting jobs, getting promotions, and quite possibly losing jobs. Also, a software developer can be proud to list a robust piece of software on their resume. Consider the following Venn diagram software depicting the relationship between successful software development and software testing.



As indicated by the diagram above, there is a slim chance of successfully developing computer software without any systematic, formal testing strategy or plan. Those cases where *ad hoc* testing is sufficient tend to be projects that are relatively small and simple by software development standards. Even in such a case there are often a few remaining bugs that only manifest under less typical usage scenarios. That small problem can snowball into a much larger problem if more complex functionality is built on top of the original software.

The diagram also distinguishes between functional testing and unit testing. Functional testing consists of tests run on the completed program, or the full program at the current stage of development. Unit tests are done to test each program function, procedure, method, or object class outside of the full program. Functional testing is more easily done and identifies the most important bugs that will be readily visible to the end user of the software. However, functional testing does not give any indication as to where in the program the bug is located. Unit testing is much better for finding the specific function and ultimately line of code that needs to be corrected.

As software packages get larger and more complex, drilling down from functional tests to find the specific line of code to be corrected becomes more difficult and time consuming. Eventually a large project with only functional testing practically grinds to halt as it takes weeks to find just a few bugs. At that point either the project stops, or unit tests must be implemented. Implementing unit tests for a project after years of development is a tedious process that takes a long time to show significant results. The best practice is that any software development project expected to take more than thirty days should have both functional tests and unit tests built into the development process from day one. Opinions vary as to whether that threshold should be thirty days, three months, or up to a year. Usually, people pushing for a long threshold want to complete the project sooner or only do the more interesting work of writing the software. Proponents of doing unit testing on all projects are usually people who want top quality results or who have experienced the pain of a long-term project without unit tests.

Software testing (and the fixing of the identified problems) certainly improves the quality of the software being developed. However, developers should be aware that testing and fixing bugs is an ongoing process. If all the bugs are worked, afterwards there may be half as many bugs. This is because 20% won't be fixed on the first try, 20% of the fixes will have inadvertently created new bugs, and new bugs (about 10%) will be discovered. These percentages are from p. 108 of the Hass book listed in the bibliography. This gives rise to a diminishing returns situation in which more and more detailed testing results in finding bugs that are fewer in number and less likely to be noticeable in the typical usage of the software. There have even been hand-wavy “proofs” that it is impossible to have perfect, completely bug free software for a project of any significant complexity. While we disagree with the view that perfection is impossible, the reality is that time and labor cost nearly always put an end to testing before a perfect product is obtained. Most software development organizations start testing too late, and stop testing too early, before obtaining their desired level of software quality. In practicality, the goal of a good software testing plan and process is to get the best possible result for any given amount of time and resources devoted to software testing.

Software packages that are more complex and/or developed over a longer period of time tend to need more types of more thorough testing. Clearly more complex programs have more places in the code where a bug could occur. Software packages developed over a long period of time are more likely to have turn over in the development staff, and less likely to have everyone on the project remembering all of the previous testing and reasons for architectural decisions. In this case, sections of code and design choices are sometimes revisited by later developers, who may make changes that break something that was working previously. A good set of regression tests will ensure that the broken item is found soon rather than months later.

Types of software testing

If a novice software developer or test engineer is asked to test the software, what do they test? The simplest obvious answer is to run the software with some typical set of input options to see if it works. That is a type of test is called a smoke test. However, a smoke test only finds the biggest, most obvious problems. Many other software bugs may remain even if the smoke test passes.

In order to have a high degree of confidence that the software is working correctly, a variety of types of tests must be performed. Functional and unit tests described in the previous section are two broad categories of tests, but there are many specific types of things those tests can be designed to find. Some of these are applicable to any type of software. A few are only applicable to certain types of software. The following table lists various types of software tests. Nearly all software testing falls into one or more of these categories.

Test type	What it tests	When used
Valid outputs	Correct behavior on correct input	Frequently when writing the applicable routine, and periodically (at least nightly) for regression.
Invalid inputs	Robust handling of invalid input	During code development, and for regression (perhaps less often)
Boundary values	Valid and invalid inputs likely to cause problems	Development & regression
Trivial cases	Valid inputs likely to cause problems	Development & regression
Permutations of inputs	Thorough selection of inputs	Development & regression
Smoke test	Quick test of major functionality	Development & installation
Sanity check	Check that results are physically reasonable	Development
Data transformation	Data conversions	Development & regression
Interaction	How an object sends and receives data to interact with other objects.	Development & regression
Standards	Conformance to established standards	Development & regression
Simulation	Software for systems where the "real" system can't be used for testing.	For all tests when testing in the live environment is impractical
Performance	Speed, memory, I/O, etc.	When tuning code performance.
Load testing	Behavior under heavy load	Periodically during development.
Multithreaded code	Errors due to threading	Periodically during multithreaded code development.
Presentation layer	Graphical display	For regression.
Localization	Versions for multiple languages.	As part of graphic interface testing.
Mean time to failure	Software reliability	Periodically during development.
Precision analysis	Accuracy of floating point values	As necessary to verify the validity of crucial sections of code.
Requirements	Project progress	More often than it should be.
Code coverage	Thoroughness of testing	Typically, as the product approaches alpha testing
Mutation testing	Thoroughness of testing	As an alternative to code coverage analysis.
Usability	Program interface design.	During the design phase, then again when the interface nears completion.
Security testing	Potential security exploits	As each release nears completion.
Static testing	Problems that can be identified by examination of the source code	Occasionally during development
Code reviews	Coding conventions.	At initial stages of the project, then periodically thereafter.
Maintainability	Future readiness of code	Once per major version & initial design
Portability	OS & hardware dependence	Once per major version & initial design
Documentation testing	That the documentation is well written.	To evaluate the first draft of the documentation
White box & black box	Designations that indicate whether or not the tester has access to the source code	More white box testing is done early in the development cycle, and more black box later in development
User testing	The fully functioning program.	Just before the release of a stable version.

Valid Inputs

The most basic unit test is to call a function with valid input values, then verify that it returns the correct result. This is testing that the function works correctly when given correct data. Just as computer codes can become rather complex, so can tests for correct behavior. This section discusses the simplest form of a test for correct results. Some of the subsequent chapters discuss more complex situations. Here is a simple unit test. This example is written in a C++ like pseudocode, which will be the example code format for this document.

```
int liTemp
cout << "Testing absolute_value with valid inputs." << endl
liTemp = absolute_value( -3 )
if ( liTemp != 3 )
  {
    cout << "ERROR: absolute_value failed integer test" << endl
  }
```

Examples like the one above can be written in the same programming language as the software, or in any scripting language that can call functions written in that language. The code above works, but it's more convenient to use a unit test harness that has assert functions to make it more convenient to do this, usually with better reporting. Thus we can revise this code to look like this.

```
int liTemp
cout << "Testing absolute_value with valid inputs." << endl
liTemp = absolute_value( -3 )
assert_eq( liTemp, 3 )
```

If our absolute_value function were written incorrectly, the assert_eq function might output an error message like this.

```
ERROR: in assert_eq
       called from line 19 of script.
       3 != 4
```

Unit test harnesses usually have assert functions for a whole range of needs. For example, there may be an assert_eq function for comparing two double precision numbers. However, checking equality of floating precision values is dangerous, since floating point numbers are expressed to a limited precision in binary computers, and floating point results can be slightly different on different CPUs due to the difference in unit round errors. Consider the following

```
// The wrong way to compare floating point values
// unless it is exactly representable in binary
cout << "Incorrect test of floating point division" << endl
float lfTest
lfTest = 22 / 7
assert_eq( lfTest, 3.142857 )
```

The above test reports that lfTest has the wrong value, even when it is correct. lfTest might contain this value to one more or one less significant digit. The way to compare these floating point numbers without this problem is like this.

```
cout << "Correct test of floating point values" << endl
float lfTest
lfTest = 22 / 7
assert_almost( lfTest, 3.142857, 0.0001 )
```

The above code tests that the two values are the same out to the fourth decimal place. If they are different beyond that point, that difference will be ignored by assert_almost.

The choice of input values is important. There should be a test where typical input values are given. There should also be tests with certain input values that can give incorrect results, even when other input values work. For numeric arguments, try numeric values that are positive, negative, 0, 1, and -1. Also try floating point numbers that are greater than 1, between 1 and 0, between 0 and -1, and less than -1. For character or string arguments, try letters, numbers, space, tab, \, /, #, @, ^, and other characters other than the letters and numbers.

Boundary Value Tests

Some of the most important valid values to test are boundary values. These are the extremes of what the function can, or should be able to, accept as inputs. With any binary representation of a number, there is a minimum and maximum value that can be represented by those binary bits. Likewise, there can be a zero length string. There may also be a maximum string length to test. For any type of pointer argument, consider whether it should be valid to pass it a NULL. Some unit test harnesses have many built in constants that are common boundary values.

Trivial Case Tests

Another type of boundary value is what a mathematician would call a trivial case. For example, we have seen matrix algebra functions that failed when given a 1x1 matrix (a single number). Often 0 or 1 are trivial case tests for numeric functions.

Permutations of Inputs

Another type of valid input testing is reading any valid permutation of the input file data that the software can read or import. Build up a collection of valid input files with one data point, many data points, trivial data (i.e. a molecule with just one atom), very large inputs, etc.

Often testing all valid permutations of inputs just isn’t practical. A good solution is to do pair-wise testing in which enough permutations (or pairs) of inputs are tested to have some reasonable level of confidence in the code.

Smoke Test

A surprisingly useful test is a smoke test. This is a simplest test to quickly check that major things are working. A smoke test to verify correct installation of software may take only a few seconds to run the simplest non-trivial problem. A smoke test to help the programmers check that nothing major is broken might take fifteen minutes to run one of every type of function on a simple data set.

One form of a smoke test is a use case test. This simply consists of performing a real-world task as the end user would typically use the software. In itself, this is a minimal smoke test. However, some organizations err on the side of only doing use case testing, which results in brittle, poorly tested software. Brittle code is software that works correctly as long as you follow a very scripted set of steps on a prescribed set of input data, but fails if features are called in a different order or if using other inputs that the end user would generally consider valid input. The end users of the software will try using it in many different ways, so it should be designed and tested to handle any set of inputs in any order.

Sanity check

Testing can become abstract. After a while it appears to be a string of numbers going in and numbers going out. Occasionally it is beneficial to step and back and ask if those numbers look reasonable. For example, you may run a series of tests to figure out how fast a car will travel with various engines and wind resistance. A sanity check would be to verify that all of the speed values are between 0 and 300 miles per hour since cars don't travel at speeds outside that range in the real world. One advantage of this is that it can quickly check for major problems, without the drudgery of hand checking what the correct results should be for a bunch of examples.

Exceptions & strategies

The discussion in the above paragraphs assumes that there is a single correct output result that should be generated for a set of correct input values. The exception to this is testing random number generators, which require completely different tests. There is a comprehensive set of tests for random number generators published in "The Art of Computer Programming, Volume 2: Seminumerical Algorithms (3rd Edition)" by Donald Knuth, Addison Wesley, 1998.

There are many programs and functions that take a number of input values, that can take on a number of different valid values. In this case, it may be impractical if not impossible to test all possible combinations of all possible input values. The solution to this is to do a combinatorial selection of the possible inputs. This selection should make sure that each input is tested with a number of values, such as typical values, boundary values, etc. It should also give a sampling of tests with various combinations of input values, an expansion on the idea of pair wise testing. This is necessarily a non-comprehensive testing scheme. Combinatorial tests can be augmented with a good mean time to failure test, which feeds randomly selected values into the function and thus may catch problem combinations that weren't included in the valid input tests.

Invalid inputs

In defining the coding practices for a software package, or even an entire software company, it is necessary to define an error handling mechanism. When a function encounters an error, does it print an error message, or perhaps print an error then halt execution? For this application, is there a distinction between errors and warnings? The function could return an error value, thus putting the onus on the code calling the function to catch the error. Or errors could be completely ignored, thus leaving the code to continue executing with garbage data in the variables, or continually throwing uncaught exceptions. Tests with invalid inputs are intended to test that the intended error handling mechanism works correctly.

As a general rule of thumb, it is not good to ignore errors. If the code is allowed to continue execution with bad data, it may not be identified until hundreds more functions have executed. At that point, the programmer may have a daunting task of trying to work backwards through the code to identify the place that the error was first generated.

Not putting in error traps at all can lead to the worst possible situation. This tends to give a very unforgiving code, where inputting one item incorrectly causes the program to crash. Worse yet, finding where and why it crashed can be incredibly difficult as the code becomes more complex.

Regardless of what the error reporting mechanism is, unit tests can be written to verify that the errors are indeed generated as they should be. Admittedly, it may be easier to write tests to verify text messages sent to a log file, than to verify graphical dialog boxes. This document shows examples of checking text output. For examples of verifying dialog box wording, see the discussion of presentation layer testing.

Various types of numeric values can be invalid as arguments to functions. Depending upon the function, positive values, negative values, or zero may be nonsensical. If the function increments or decrements a value, the largest or smallest values representable in the specified precision may be invalid. An invalid numeric input that is often missed is a combination of inputs that causes the internal equations to have a divide by zero. Numeric tests should always include the special numeric values such as, NaN, -0, inf, and MAX_INT. Any value that is the absolute maximum or minimum representable by a given variable type could cause program failures with certain algorithms, and thus should be tested. Each of these maps to a particular binary value that a numeric variable can hold. Thus each is a potential input that may be passed to the function being tested. Some other invalid inputs to test are the minimum valid value minus one and the maximum valid value plus one.

Functions that take pointers as an argument may fail on a NULL pointer. The function may also assume that the pointer being passed to it is the first or last item in a linked list, the root node of a binary tree, or some other specific item.

Functions that process strings may fail on a zero length string, or strings with symbols, or characters that are not part of the printable ASCII character set. There can also be problems associated with ASCII verses Unicode string data.

There is a caveat to building automated tests that the code is generating the error messages that it should. The test program should not report that the test failed when it gets these error messages. The test program should report a problem only when the error message received does not match the one that it should be getting. Here is an example of a bourne shell script to verify that results match a saved set of previously verified results.

```
// prior to this, a test calculation created file test04.out
num_errors=`diff test04.out test04.out.correct | wc -l`
if [ "$num_errors" -gt 0 ] then
  # print diff out to the console
  diff test04.out test04.out.correct
  echo $num_errors" errors in test04"
fi
```

As mentioned above, there are some special values that should be tested. Unit test harnesses make these values available often to a scripting layer. The following are some examples of pseudocode for calling functions with invalid inputs.

```
// set ldInput to NaN
ldInput = nan
cout << "using NaN input ldInput = " << ldInput << endl
ldOutput = absolute_value( ldInput )
cout << "absolute_value = " << ldOutput << endl << endl

// set ldInput to inf, infinity
ldInput = inf
cout << "using inf input ldInput = " << ldInput << endl
ldOutput = absolute_value( ldInput )
cout << "absolute_value = " << ldOutput << endl << endl

// another NaN test - badly written square root
ldOutput = square_root( -1.2 )
cout << "square_root( -1.2 ) = " << ldOutput << endl << endl

// another NaN test - square root with input checking
ldOutput = square_root2( -1.2 )
cout << "square_root2( -1.2 ) = " << ldOutput << endl << endl
```

The output from running this section of test script may look like this.

```
using NaN input ldInput = nan
absolute_value = nan

using inf input ldInput = inf
absolute_value = inf

square_root( -1.2 ) = nan

ERROR: square_root2 called with invalid negative argument.
       square_root2( -1.2 )
```

In this example, absolute_value and square_root do not in themselves give any indication that they were called with invalid arguments. The function square_root2 has been written more robustly to fail with an error message upon receiving an invalid value. Some code development projects will choose to wrap built in functions with error traps, as has been done in square_root2.

Another variation on incorrect input testing is to have examples of input files with as wide a selection of incorrect variations on the input possible. For example, have both Unix and Windows text file formatting, which differ in the characters used to signal a new line. Also try files with missing lines, missing fields, non alpha-numeric characters in fields, unmatched brackets, etc.

Data transformation tests

Data transformations are events where the format of the data changes, but the information it carries doesn't. This sometimes means that the data can be restored to the original form. An example would be compressing data, in which case it can be uncompressed to get back the original data. Other examples are encryption, conversion between ASCII and binary, or writing data to a file.

Data transformation tests are often done by converting the data, then making checks on the converted data. Sometimes the test will convert the data back again to verify that it comes back identical to how it started out. Here is a simplest case piece of code for file comparison in pseudocode.

```
file_toupper( "test31.dat1", "test31.dat2" )
file_match( "test31.dat2", "test31.dat3" )
```

In this example, the function file_toupper takes the file test31.dat1 and creates a file test31.dat2 in which all characters have been converted to upper case. Then the file_match function tests that the new file is identical to a stored, correct result.

A big problem is that the stored correct results must indeed be correct. If an incorrect output accidentally gets copied to the correct result file, then the test will stop working and there will be no outward indication that the test is broken. Several mechanisms are used to avoid this problem. Some organizations simply educate developers about this potential problem and warn them to be very cautious about updating correct results files. Some keep the correct results files with read only permissions, so that more conscious effort is required to change them. Some organizations require two different people to independently verify that the correct results, and any changes to them are indeed correct.

Interaction tests

The pieces of a software suite may communicate with one another in multiple ways. In the simplest case, there are function arguments and return values. Some programming languages have the ability to throw and catch exceptions. Some software may pass data via stdin and stdout. Even more complexity arises if the software accesses a network with a given data transfer protocol. Failures in these interactions are one source of bugs, and sometimes security flaws.

Testing interactions in software components tends to be specific to the language and type of interaction. Function arguments and return values are readily tested with unit tests. Software written in programming languages that throw exceptions often include a catch all function to catch any exceptions that are otherwise uncaught. There are testing tools designed for testing network communication.

Standards testing

Some programs are designed to generate output that conforms to a published standard. Consider the example of a program that generates files to be viewed with a web browser. It is necessary to test that the information will be displayed correctly when imported into a web browser. In order to make such programs useful to the widest possible customer base, many organizations specify that it should be possible to display the output on certain versions of from three to six of the most popular web browsers. In spite of the fact that the web was designed to make it possible to distribute data to heterogeneous systems, this can be surprisingly difficult to accomplish. This is a worse case situation for standards testing because not all browsers conform to all of the published standards. Some may be a bit behind the current standard, while others may have proprietary extensions beyond the standard. Unfortunately, this is not an uncommon situation to encounter when generating data to be imported into programs from multiple vendors.

It is wise to do some sort of standards testing any time that a program will import data generated by another software package, or export data to be read by another software package, particularly ones from other vendors. This even includes the case where the other software packages are written by the same company.

The primary goal of standards testing is to ensure that the two programs work correctly together, because they correctly implement a standard for communication. However, there is a secondary goal of ensuring that reasonable error messages or behavior is seen when the data is corrupted, or unrecognized. Even though two programs may work perfectly together, at some point in the future the user may well upgrade one of them and not the other thus giving a situation where the two are no longer compatible. Do not assume that standards are implemented correctly just because you use a third-party library for the functions that handle that standard…. Test it anyway.

Standards testing is sometimes performed as a functional test of the entire software package, and sometimes as a unit test of the individual routines that import and export data. Ideally, the import and export calls should be confined enough to individual methods that unit tests can be done, then both unit tests and functional tests are run. The unit tests verify that the methods are writing out the data in the correct format, while the functional tests verify that the correct data is being sent to those methods.

There are also quality standards for how the code is written. A programming project in which the software must meet some quality or robustness requirement may mandate that the code is written to comply with one of these standards. Some of the existing standards are ISO 9126, McCall & Matsumoto, IEEE 830, ESA PSS-05, and the BCS working group report. Testing for these types of quality standards is generally done through a code review process.

Simulation tests

It is extremely important to test computer software, but there are some things that just can't be tested. Consider the task of writing control software for a nuclear reactor. The software should be tested to make sure that an accidental input error from one of the operators will not cause it to accidentally melt down the reactor core. Obviously, this test should not be run on a live nuclear reactor. The solution to this problem is to write a second program that is a nuclear reactor simulator. The control software can be tested by having it talk to the simulator so that no harm is done if a problem in the unfinished code would have caused a catastrophic failure. This software substitute for the real thing may be called a "simulator" or a "mock object". A mock object typically has some type of programmed behavior. This distinguishes it from a “stub” which is typically an empty function that always returns the same value for testing purposes.

Another example where a simulator would be useful is testing job queuing software. It could take years to thoroughly test queue software in a real cluster computing environment where the jobs may take weeks to complete. A simulator can be written so that various scheduling algorithms can be tested at the rate of thousands of jobs per second. Writing a simulator would also be much less expensive than purchasing a collection of different supercomputers of different architectures on which to test the scheduling algorithm. Simulators can also be used in place of a live database system in testing front-end software for a database.

Sometimes simulation testing is done as part of the unit tests. Sometimes it is done as part of the software package functional testing. Sometimes both are done. Which is appropriate is dependent upon the design of the software package.

One of the major keys to success in simulation testing is to first make sure that the simulator itself is thoroughly tested and functioning correctly. Obviously, the simulation tests will be useless if the simulator doesn't correctly reflect the behavior of the real system.

Performance testing

Performance testing and optimization is a field of expertise, some would say an art form, all of its own. A performance test is one that measures how long it takes for a program to execute, how much memory it uses, scratch file utilization, communication latency, bandwidth, etc.

Usually, functional tests of the entire program are run first. Then profiling tools are used to identify which methods are responsible for the less than desirable performance. Finally, unit tests may be used to aid with the optimization of the trouble spots.

There are a number of reasons that this is more difficult to accomplish than it may appear. The following is a discussion of why these difficulties arise, and how to get around them to get the job done.

The first major issue is that the results won't be the same every run, even if identical tests are run. Results can be affected by other software running on the system, current network traffic, operating system housekeeping processes, etc. Results will also change as the test is run on different hardware. This makes it necessary to write tests that pass when results are slightly different from previous results, but fail when results change more than a specified percentage.

Another aggravating part of performance testing is that the tools for measuring system performance are inaccurate. These tools use algorithms that were designed knowing that there would be a statistical error in results, but designed to keep that error small. The problem is that increasing CPU clock speeds from one generation of hardware to the next changes those statistical errors, thus making many performance monitoring tools become less accurate as the hardware improves. For a detailed technical discussion of this issue, see the section below, titled "The Evils of Jiffy Counters".

There are several ways to compensate for inaccuracies in performance measuring tools. One option is to only use selected tools that better compensate for those problems, at the expense of giving less information. Another option is to use performance tools with the understanding that the results are subject to systematic errors. For example, the function that a profiler shows as using the most CPU time probably does, even though the actual value of the amount of CPU time used may be significantly in error. Some testers take the low-level option of manually working out correction factors to compensate for the inaccuracies of a given performance monitoring tool for a given application on a specific hardware platform.

Memory usage is analyzed both to determine how much memory is required to run the software, and to identify memory leaks. Analyzing memory usage is particularly difficult when using a language with automated garbage collection, such as Java. A garbage collection system does not ask users when they are done with memory, but analyzes it dynamically and occasionally frees the memory. One problem this creates is that the memory isn't freed immediately, so it can be difficult to determine which function was using that memory. Another problem is that garbage collection systems often expand to use different amounts of memory depending upon how much is installed and available on the hardware platform. Sometimes an additional function call can be added into the software to trigger a garbage collection cycle.

It’s necessary to investigate the nature of the performance testing tools in use. Some may give more accurate results at the expense of giving only a few pieces of information. Others may give a very rich set of information, but leave handling the expected variability of results to the developer. For this reason, multiple performance testing tools may be used in the project.

The Evils of Jiffy Counters

The following discussion is a more detailed look at why many performance monitoring tools are inaccurate. This gets deeper into some technical issues than other sections of this document. The reader can safely skip this section if they are willing to accept our statement that many performance monitoring tools are inaccurate. For those that want to know why these tools give inaccurate results, continue reading this section of this paper.

Most computer operating systems in use today are multitasking. The computer appears to be doing multiple things at once, but the CPU is really running one process at once (one per CPU core), and switching between tasks after a number of milliseconds. These multitasking time slices are called "jiffys" and have been set to 100 milliseconds on most operating systems, all the way back to the first Unix versions created in the 1970s. The 100 millisecond time slice is based on the speed of human reflexes. It is the longest time slice that will allow a human to perceive the two processes as running simultaneously. However, since the 1970s CPUs have increased in clock speed by orders of magnitude, thus increasing the number of CPU clock ticks in a jiffy from around 1000 to around 1,000,000.

When utilities such as the Unix "time" or "sa" commands report how much CPU time a process used, they are polling jiffy counters in the operating system kernel. Jiffy counters are integer values the kernel uses to count how many jiffys have executed for the process. The problem is that the program doesn't necessarily execute for the entire duration of the jiffy. The kernel switches which process is being executed at the end of each jiffy (called preemptive multitasking) but it also switches which processes is being executed in the middle of the jiffy if the current process hits an I/O wait state while it waits for disk I/O or even a page fault from main memory into cache (called non-preemptive multitasking). In the 1970s when there were far fewer CPU clock ticks in a jiffy, this mid-jiffy swap happened far less often statistically, and thus was responsible for negligible errors in the performance monitoring data. With present day CPUs being 1000 times faster, the probability of a mid-jiffy swap happening is 1000 fold higher, and thus has increased the error in the performance monitoring by a factor of a thousand.

On a recent supercomputer, we compared the sum total of the performance data collected by the jiffy counter based linux "sa" command to the total CPU load reported by the "sar" command. The sar command is time averaged and thus not subject to jiffy counting errors. Current systems typically see a 50% error between jiffy counter based accounting and time averaged accounting. A number of Linux builds in the 2003-2010 year range had an error in the kernel code that gave a 10x error in jiffy counter based results thus under representing the CPU time utilization.

A crude way around this problem is to only use wall clock times, which only take into account when a program started and when it ended. A somewhat better way around this error is to use an averaged system load, as is done by the linux "sar" or "top" commands. This allows the time that a program executed to be measured more accurately. However, load averaged accounting can't get down to the lower level of profiling how many milliseconds each function in the program spent running the computation.

Fortunately, there is a better way to do low level performance monitoring. It is a system called microstate accounting. In microstate accounting the real clock time is logged every time a process is swapped in or out of the CPU, even in the middle of a jiffy. This gives much more accurate performance monitoring, at the expense of adding some overhead to the operating system. This overhead can make the computer appear to run slow, even if you aren't doing any program profiling at that moment. At the time this document was written, the only operating system in existence with microstate accounting was Solaris variant of Unix from Sun. There are rumors of microstate accounting being put into the Linux kernel, but no distributions yet available.

Load testing

Load tests (also called stress tests) are constructed similar to performance tests, but load tests are run for a different reason, and thus made somewhat differently. The purpose of a load test is to find out at what point the program fails, or shows significant performance degradation due to handling excessively large amounts of data, or running a particularly large calculation of some sort.

Load tests are generally setup like performance tests, in that they measure the performance of the code with various input values. Load tests might have a whole series of inputs of various sizes. These are run successively to find out at what point the program has problems. Like performance tests, load tests give considerably different results on different hardware platforms.

In designing load tests, the failure criteria must be defined. For tests of increasing memory utilization, the failure point could be the point where the computer starts using virtual memory, or the point where the program halts execution with a memory error. For software that is used interactively, such as a web-based application, a pause in the software of more than a few seconds may be unacceptable.

A test for performance degradation can be constructed by monitoring CPU utilization. If the program typically displays 100% CPU utilization, there can be a larger input for which CPU utilization goes down to around 10%, perhaps due to virtual memory swapping or being limited by disk I/O.

One trick to running load tests is to run them on an older computer with a slower CPU and less memory. Problems will often show up more quickly on outdated hardware than on the most recent systems, where hardware performance can compensate for software issues. Often some load testing is done on systems one generation older than the software manufacturer expects to recommend as a minimal system configuration.

Load tests can be unit tests if a specific subroutine is primarily responsible for the majority of the system utilization. More often, load tests are functional tests of the whole program. Load test might be run daily if the nature of the software is such that load handling is a primary design criterion. More often, load tests are only run occasionally during the software development process.

Testing multithreaded code

Programs can be written to use multiple threads (ideal to run on multiple core processors) in a number of ways. These include pthreads, OpenMP, Java Threads, MPI and a number of other parallelization mechanisms. Using multiple threads allows the program to get better performance or more graphic interface functionality than is possible with single threaded programs. However, this also results in a new set of things that must be done in order to write the code, debug the code, and test the code. There are even some types of errors that are unique to multithreaded codes.

Multithreaded programs can give a significant decrease in performance if they run into a starvation condition. For example, if one master thread is serving data out sequentially a host of slave threads, each slave thread might spend more time waiting for the master thread to reply for its request for more data than the amount of time it spends actually doing the computing. This can be particularly prone to happening when the program has been given synchronization code to ensure that certain tasks are done in a certain order, thus forcing other processes to wait. Sometimes a careful code redesign can give better performance, and sometimes there just isn't a better option. In either case it is valuable to know whether a starvation condition is occurring.

A far worse condition is a deadlock. A deadlock is a condition that can't be met. For example, thread A might be waiting for thread B to send it information at the same time that thread B is waiting for thread A to send it information. Thus, both are waiting and neither can do anything. The end result is that whole program locks up and does nothing.

Another error, called a race condition, can cause the program to sometimes work correctly and sometimes not work correctly. A race condition can occur if thread A and thread B must both report their results to the master thread. If thread A reports its results first the program may work correctly, if thread B reports its results first the program may work incorrectly. Which finishes first might be affected by what type of computer they are running on, what else is running on that computer, the data assigned to each thread, the current network traffic, etc. Race conditions are amongst the most difficult of the multithreaded program errors to reproduce, find, and analyze.

Race conditions and deadlocks can be eliminated by putting additional synchronization into the code. The synchronization forces things to happen in a certain order, even if some threads must wait for those conditions to be satisfied before continuing its execution. When the synchronization frequently puts threads in a wait state, it results in a starvation condition. The way to avoid, or at least minimize, this conundrum is for the software architect to be aware of these issues, and carefully design the way that threads perform their tasks and interact with one another to avoid such problems.

Errors specific to multithreaded coded are the result of interactions between different portions of the program. As such, these errors are tested as functional tests of the entire program the vast majority of the time. There has been a bit of work done on designing unit tests for threading errors. This has been done by having a multithreaded unit test program, which can initiate one function, insert a time delay, then initiate a second function, then test for correct behavior. It is typically necessary to initiate functions in different orders with different time delays in order to identify problems.

There are some techniques for finding threading errors that the reader should be aware of. Static analysis programs (i.e. lint and similar packages) examine the source code and suggest places problems might occur. The programs do identify many potential problems, including threading errors. The disadvantage of these programs is that they can generate a massive output that lists many potential problems that couldn't really happen because the programmer had put the necessary error traps further up in the code and static analysis program can't analyze all of the potential execution flows well enough to see that a given situation could never occur.

Dynamic analysis is the analysis of the software while it runs. This requires some sort of instrumentation of the code. Some simple testing can be put in using optional flags to some compilers. A common dynamic analysis task is checking for memory leaks and pointer issues. There are dynamic analysis programs specifically for identifying parallelization issues.

Another technique to be aware of is that many threading errors will first manifest themselves when the code is running under a very light load, a very heavy load, on very fast hardware, or very slow hardware. Thus, simplest case tests and load tests can be run on both very new and very old hardware as one of the checks for load and threading problems simultaneously. Testers may have other programs that they sometimes run at the same time that tests are running in order to find out what happens if another program is bogging down the CPU, using increasing amounts of memory, etc.

Presentation layer testing

Another type of testing with its own inherent difficulties is presentation layer testing. Most of the testing described thus far in this manual has been based around generating text outputs that report errors, or can be compared to the expected output. In the design of software packages with a graphic interface, there is a need to test that the graphic interface is being displayed correctly. In this case, there may be no error messages or text outputs to indicate a problem, only a display that does not look as it should.

In addition to testing that the graphic interface looks correct, it is also necessary to test that the graphic interface works correctly. That means making sure that the correct behavior is seen when the mouse is clicked or dragged on a particular area of the display. There are several ways to test the presentation layer, including manual testing, unit testing, and functional testing. These generally require additional components designed specifically for presentation layer testing.

The first step in developing presentation layer tests is usually writing up a procedure for manual testing. This is usually a document that tells the person doing the testing (often an intern or entry level position) how to click on each control of the graphic interface, then compare the resulting display to an image in the document. This is a very labor-intensive way to do software testing, but there are two reasons that manual tests are developed for graphic interfaces. First, the graphic interface may be changing daily at certain stages of the software development, thus making the continual update of automated interface tests too labor intensive. Edits to the manual-testing document are generally far less labor intensive than changes to an automated testing procedure. Second, the manual-testing document provides a set of steps to be implemented in the automated tests.

Automated presentation layer tests all depend upon having a way to compare the display to the expected display. For this reason, some programs will have a function for taking a snapshot of the open window embedded in the code, even if it is not accessible to the users. If there is not an internal mechanism for taking a window snapshot, an external utility must be used. The image of the present display must be compared to the image of the expected display using an image processing utility. At the crudest level, the Linux "diff" command can be used to tell whether two binary files are exactly identical (it can't tell anything else about how they differ).

There are tools for reporting a percentage difference between two images (i.e. screen shots) or even to movies (i.e. from an automated sequence of mouse clicks). These tools can be useful if some change in display is expected, such as a web page that displays a daily news headline. In this case, there may be a system to track how much variation is typical from day to day, and an error raised if it is more than that. If well implemented this can ignore changes in daily data, but will flag format changes. Another option is to compare how two programs display the same data and print an error if they differ significantly.

The majority of presentation layer testing are implemented as functional tests of the whole program, rather than unit tests. Some graphical programs will have internal scripting languages that can be used to write tests that drive the graphic interface, performing exactly the same actions that would be done with the mouse. If not, there are utilities designed to create an automated process for issuing mouse clicks to the screen. Utilities that control the mouse cursor can be used to test the presentation layer even if no internal functionality for this purpose exists.

Some portions of the graphic interface, such as dialog boxes, may be generated by a single subroutine. In this case, it is sometimes possible to test the dialog box and its embedded action listeners separate from the rest of the program. This is the less common way to test graphic user interfaces.

Localization testing

Large software packages are sometimes designed to be used in many countries. This is done with a compile time flag or configuration option to display all menus and dialogs in a different language. It may alter other aspects of the program behavior such as date and time display formats or monetary symbols. The means for testing this are often the same as for testing the presentation layer.

Estimating mean time to failure

Mean time to failure (MTTF) is a difficult thing to test or measure. However, it is an important metric because the user’s perception of the software quality is based on their personal observations of program failures. Unfortunately, mean time to failure is often one of the last tests implemented, if ever. Mean time to failure is sometimes called mean time between failures (MTBF). Both terms will be used interchangeably for the purposes of this discussion.

There are many types of failures that can be exhibited by a code after it has been released to the customers, even if it passed all of the tests that were devised during development. These failures could take on many forms, such as having the program crash, deadlocks, failing to complete a data analysis sequence, etc. Failures seen after passing all validation tests are most often due to a particular sequence of input values that the development and testing department personnel didn't think of when they were devising the code tests. They can also be due to timing issues when the code is run on faster or slower computers or networks. Thus, the goal of mean time to failure testing is to test the things that you forgot to test. That seems like a contradiction, but it can be done. The solution is in short... random number generators.

Some testing tools provide a rich set of options for generating random numbers or random characters. These are useful for testing code with values that weren't anticipated in the design of other tests.

In theory, a mean time to failure test is rather simple. It runs a large number of tests with randomly chosen input values, until something breaks. In practice, it is more difficult than it sounds to write a good mean time to failure test. However, a good mean time to failure test can be an incredibly powerful tool for finding errors that would otherwise be missed by even the most diligent software developers and testers.

Mean time to failure tests can be either unit tests of each specific function, or functional test of the entire code. Both types of tests have merits. Unit tests are more likely to give good code coverage. Theoretically mean time to failure unit tests can give complete code coverage, but they don’t test interaction between functions. Functional tests for mean time to failure have the advantage of being more likely to find problems that could be encountered by a user of the program.

Within both unit tests and functional tests for mean time to failure there are two variations; tests designed to test valid inputs and tests designed to test invalid inputs. Tests that allow invalid inputs are expected to generate error messages and halt execution. In fact, the purpose for invalid input tests is to verify that the code generates useful error messages and executes gracefully, not via a segmentation fault. This is usually a test for obviously invalid outputs, such as error messages or nonsensical numeric values (i.e. negative outputs from a function that should only give positive values, nan, inf, etc.). Tests with valid input values are used to find out if there are valid inputs for which the code fails.

There may be other practical constraints besides valid or invalid input values. For example, numeric algorithms may require input values to be limited so that they don't inadvertently hit a set of inputs that causes the calculation to run for months, or use all of the memory. There can be very complex dependencies such as requiring that argument 2 is greater than twice argument 1, but less than the cube of argument 4. Such complex dependencies are often uncovered during testing for mean time to failure.

Getting the program to fail is useless unless the cause of the failure can be determined. More specifically, there must be a way to reliably reproduce the failure. As such, mean time to failure tests must generate a log of what commands are being issued. Thus, the general sequence of events is to select values or menu picks at random, then write the command about to be issued to a log file, then issue the command. If the software package has a built-in macro language, the log file might be a macro to replay the exact sequence of inputs that caused the failure.

Mean time to failure tests should include some type of metric to give a measurement of whether the software is getting more or less robust. One way to do this is to run multiple tests, then compute an average number of iterations between failures. The value most often generated by a mean time to failure test may be the number of failures per test hour, or per month, or per thousand transactions. There are also more complex robustness metrics that take into account the severity of the failure as well as the frequency of failures. One type of robustness metric is a recoverability metric which takes into account the time to restore after a crash, the time to regenerate any lost data, and may include metrics for the backup redundancy and failover capability.

Mean time to failure tests are sometimes called maturity tests. An alternative way of testing maturity is by generating metrics on the programming process. There are a number of established maturity metrics, such as the software maturity matrix (SwMM), capability maturity model (CMM), or software maturity index (SMI). These maturity metrics typically reflect aspects of the software development process, such as whether code coverage testing is done, or if there is an established process for making improvements to the software development process. Although generating maturity metrics doesn't improve the software product, it does serve to suggest ways that the software testing and development process can be improved.

A related test is fault tolerance. This is testing how the software responds to failures. If the software communicates over a network, the tester may unplug a network cable while it is running to see how the software responds. This tests the failure mode of the software. Does it crash, freeze, or indicate that it is waiting for the network to be restored? Sometimes fault tolerance and recoverability can be combined into the one fail and recover test.

Precision analysis

There are two types of floating point precision errors that can be very difficult to analyze, find, and fix. These errors occur due to the binary representation of floating point numbers. Thus, the code can be written with every line doing exactly what it should, and execute without error, but give mathematical results that are inaccurate. This inaccuracy is particularly difficult to find and analyze because the output values may have the first few digits correct, followed by numbers that are the wrong values.

One of the easiest ways to get this type of loss of precision is with certain mathematical operations. For example, look at this subtraction

$$1.23459875 - 1.23456789 = 0.00003086$$

We started out with two numbers having 9 significant digits and got an answer with 4 significant digits. Thus, all subsequent calculations are accurate to only four significant digits. However, the result may be displayed in the output like this 3.08622314e-5. It appears as though we have 9 digits of accuracy, even though we really only have four digits of accuracy, and another 5 digits of whatever garbage was left in memory, perhaps left over from when the employee was browsing bass fishing web sites on their lunch hour.

An error that is slightly less painful, although possibly more difficult to fix, is a unit round error. Nearly all programming languages use a fixed number of bits to hold floating point numbers (usually 32 bits for type float, and 64 bits for type double). Because of this, there are only a certain number of digits that can be represented for any number. However, there are some numbers that would require an infinite number of bits to represent, such as Pi or 1/3. Thus, the last digit in memory will have some round off error as extra digits are truncated or rounded up. This is called a unit round error. If mathematical calculations are done on a collection of numbers with round-off errors in the last digit, the calculation can at some point give a result that has an error in the second to the last digit. Likewise, mathematical algorithms that do billions of iterations might have the last four or even eight digits in error due to accumulation of unit round errors.

Many types of mathematical software packages, such as quantum chemistry software, do all of the calculations in double precision, just to get an answer accurate to single precision. Half of the accuracy may be lost to these numeric errors.

From a testing standpoint, the problem is finding a way to test how much of the answer can be trusted and how much has been lost to precision errors. Traditionally, this has been done by comparing results to hand checked sample calculations. Hand worked calculations are an important verification of numerical codes. However, the high labor cost of doing hand checked examples is responsible for them being utilized very rarely. There isn't an ideal way to analyze for loss of precision errors. The following paragraphs discuss the pros and cons of a number of options.

Techniques exist for doing a hand analysis of the loss of precision errors inherent in the calculation, as opposed to one specific result. These techniques are well documented in most textbooks on numerical analysis. An error analysis should be done, but it isn't necessarily any less work than doing hand-checked examples. Again, the labor involved makes this an under-utilized technique.

Symbolic manipulation software packages, such as Mathematica and Maple, can do calculations with a very high precision. This comes at the price of running extremely slowly and using more memory than when the same equations are put into conventional languages. These shortcomings often make symbolic manipulation languages inappropriate for writing the end application, but they can be used for prototyping or testing mathematical algorithms. Rewriting the algorithm isn't a trivial amount of work, but once it is done many different trial inputs can be run.

If the algorithm is written to use single precision floating point variables, it is sometimes possible to use a compiler flag to automatically convert it to double precision. This allows making an executable to test with higher precision results. This is the easiest way to check for precision problems, but it doesn't do any good if the code is already written in double precision.

Static analysis programs such as some versions of lint can identify lines of code that are potential risks for loss of precision errors. The problem is that these programs can give a very large number of false warnings for every real problem uncovered. Checking all of these issues by hand can be nearly as much work as checking the whole code by hand.

There is a small movement to shift to the usage of interval arithmetic techniques. These are algorithms that keep track of the minimum and maximum possible results for every arithmetic operation. This range reflects the worst-case results due to loss of precision errors or unit round errors. At some time in the future, this may become a mainstream analysis technique, but at present it is used only in selected development projects where there is a driving need for this type of analysis. Because these techniques are not built into compilers, they must be integrated in by doing all arithmetic calculations through library calls, which slows down the code execution considerably.

Precision errors can create problems for other types of tests because each model of processor chip may have a different unit round algorithm. As such, two different computers may give different results in the last few digits. Both of them are correct if both are within the unit round error bounds. This is why floating point value results should be used for testing only if they are rounded to a number of digits that should remove unit round error.

Loss of precision errors can be analyzed either at the individual function level, or through functional test of the entire program. Both give valuable information. It is easier to analyze individual functions one at a time. However, functional tests of the entire code can identify errors due to calling functions with the arguments reversed.

Software requirements testing

One type of testing for proper behavior is requirements testing. Verifying that the software conforms to written requirements gives a quantifiable measurement of development progress, and in some cases shows that the developer's contractual obligations have been met.

Requirements testing can also be abused. Some individuals and organizations will try to hide behind the written requirements, using them as an excuse for writing brittle code. Individuals engaged in writing requirements, testing and coding should keep in mind that there is always an implied requirement that the software should work correctly with any input that would generally be considered valid by the users. There is a second implied requirement that invalid inputs should be prevented if possible (i.e. graying out menu options not valid at the present time). If invalid inputs cannot be prevented, the software should have a robust behavior (program doesn't crash if it can reasonably continue to function after encountering the error) and provide the user with an error message that tells them very clearly what aspect of the input was incorrect. The testing conventions described in this document are best practices that have evolved to ensure that codes will meet requirements, behave correctly with all manner of correct, or incorrect inputs, and generally be a well written, robust programs.

The legal staff may well disagree with the discussion of implied requirements in the previous paragraph. Indeed, most contracts are written to specify that there are no other implied warranties beyond those expressly stated in the contract. However, in many states there are laws specifying implied warranties of fitness for the intended use, which cannot be waived by a license agreement. Common sense still reigns. A company that gets a reputation for doing shoddy work will soon have no work at all. An experienced software developer will have his resume critiqued based on the quality of the software that he or she worked on in the past. Thus, you should always assume that the people that use the software you ship out the door will find every hidden problem in the code, and will be the ones that control your continued employment at this company and every company you interview with in the future.

Code coverage

Code coverage is a criterion for the design of a thorough test suite. The idea behind code coverage is that there should be at least one test that causes every section of code to be executed. This is defined more precisely by saying that there should be a test in place to follow each path at the branch points in the code (called branch testing), each result of a conditional statement (condition testing), each path through the program (path testing), each function entry and exit point (procedure call testing), or each line of code (statement testing). Good code coverage tends to be easier to accomplish through unit testing than through functional testing.

In practice the amount of labor required sometimes makes 100% code coverage impractical, thus resulting in using 100% coverage criteria only for critical sections of code. Projects using fast paced agile development methodologies often test with 20% to 70% code coverage. Even 100% code coverage doesn't guarantee bug free code, as some errors may occur only when certain input values are passed. However, more code coverage generally means better code, so do the best that business constraints allow.

Mutation Testing

Mutation testing fills a role similar to code coverage in that it is a test of the thoroughness of a testing suite. In mutation testing, random bugs are intentionally inserted into the software source code. This is typically done by altering code logic so that it will still compile, but function incorrectly. If the test suite finds the mutated sections of code, that provides some validation that the test suite is working well. Mutation testing has its limits since it only checks for whether the test suite can catch certain types of bugs. Some mutation testing tools are called fault seeding, or fault injection.

Testing software usability

Usability testing is a check on whether the program design, particularly a graphic interface, is intuitive and comfortable to use for a representative end user who has never seen this program before. Usability testing isn't a test for incorrect behavior as are the other tests discussed in this book. In fact, it is really a validation of the program design rather than a test. However, the most bug free software is still doomed to market failure if its design is so cumbersome to use that customers would prefer to use a competing product. As such, usability is no less important than the other criteria discussed in this paper.

Usability testing is intended to look for multiple aspects of the program design. Understandability is how well the user can interpret the information being presented by the program. Learnability is how easy it is for a first-time user to begin using the software. Operability is how convenient it is to use the software. Attractiveness incorporates the look, layout, and documentation format. Accessibility is the presence of features to allow the software to be used by many potential customers, such as those who may need larger fonts due to poor eyesight.

There are several ways to conduct usability tests. These tests may be informal or much more controlled, formal tests. Tests of software that is used via a graphic interface are necessarily different from tests of non-graphical package, such as those that are command line driven. There are also usability tests done before the software is even written, called paper prototype tests. The following paragraphs give examples of a number of these testing scenarios.

A product development team has been formed to write a new software product. After the version one functionality has been approved, the graphic interface design begins. The major elements of the graphic interface design are the layout of the main screen, and the arrangement of features into pull down menus. There are only a limited number of ways that the information can be formatted onto a screen size window. The entire team looks over hand drawn examples of each, and (almost) unanimously chooses one where the workflow will involve working first with the information on the left side of the screen, then moving on towards the controls on the right side of the screen. The arrangement of options on pull down menus is more problematic. The design team includes three experts on the software subject area. Each of these experts has independently put together a listing of which items should appear on each pull down menu, and there are some significant differences between them. The product manager selects one of these options that feels best to her and decides that a formal paper prototype test should be done on that design.

The formal paper prototype tests are done a few days later. Three subject area experts, who were not involved in the project design have been recruited from another department to be the test subjects. The team members conducting the test have made slips of paper showing each pull down menu and dialog box. A sheet of paper showing the main screen design is placed on the table and a video camera is focused on the tabletop. The test subject is given a general task to read in a certain file and do a given analysis on the data. The test subject verbally tells what they are doing and uses a pencil as a mouse pointer. When the test subject says they are clicking on a given item, the test conductor places the slip of paper with that pull down menu options in front of them. When the test subject has to try several different menus to find the correct option, the test conductor asks them to explain why they tried the other menus first. After going through this process with several test subjects, the team finds that a couple menu options did not have intuitive names, and weren't on the menu where the subject expected to find them. They also found that the main screen was confusing because it didn't look like the screen used by competing products. After careful consideration, the design team decides that the unusual screen layout will be a significant improvement over the existing software once users get comfortable with it. However, they decide to gray out sections of the screen and put help hints on the bottom bar to prompt the users to follow the correct workflow. The confusing menu items are changed.

As the software is approaching alpha test phase, it is time for some informal usability tests on the working code. The graphic interface has undergone some changes from the original paper prototypes, brought about by the iterative process of exploring hopefully improved ways of accomplishing the task. In the first informal tests, the test subject is given a software installation, but no documentation. They are given only a single sheet of paper listing some things that the software can do. The user works independently to try doing those things based on the graphic interface alone. At the end of a day of working with the software, the tester submits a short report stating which things they figured out easily, which they couldn't figure out how to do at all, etc. The design team makes some minor changes to the interface and adds balloon help hints.

The next round of usability testing is a formal test. Users that have never seen the software are asked to do tasks as they were in the formal paper prototyping sessions. A moderator is there to encourage the test user to say why they are choosing certain options, then help the user along if the test stalls out so that later steps can be tested. The user's words and screen display are recorded. The developers can later watch these videos or watch in real time on a screen in a different room.

The last round of in-house testing is usually an informal test. The new software package along with complete documentation is distributed to the sales and support staff. These staff members are asked to become familiar with using the software, and to contact the development staff if there are still things that they can't figure out how to do from the documentation. After cleaning up any issues found at this stage, the software is sent out to selected customers in a beta test trial. This format of informal testing with full documentation will work for both graphical and non-graphical software packages.

Security testing

There are both unit tests and functional tests specifically aimed at identifying security flaws. These tend to be specific variations on invalid data and load tests. Some of the security issues typically examined are; buffer overflows, man in the middle attacks, trying common passwords, blank inputs, invalid inputs, using an incorrect data structure or file format, SQL injection, and denial of service attacks.

Security testing is particularly important if the software will talk to the internet or run as root (with administrator privileges on Windows). Due to the large number of security threats with new ones rapidly emerging, many organizations use a testing tool specifically designed for security testing.

Static analysis

Another type of testing is static testing, also called static analysis. This means looking at the code without actually compiling or executing it. There are automated static testing programs such as lint that do a very thorough job of this. The downside of some of these programs is that they kick out piles of irrelevant warnings. Today some compilers give excellent static analysis simply by turning on all available warning messages.

Code reviews

The manual form of static testing is the code review process. Many companies have some manner of code review process to look over code to ensure that it is meeting their coding standards, such as having adequate comments, utilizing variable naming conventions, and having error traps. This helps teach entry-level programmers what is expected of them, and acts as a deterrent against senior programmers getting sloppy in their work. Some companies utilize multiple types of code reviews with different objectives. These types of code reviews are usually given different names, such as; informal review, walk-through, technical review, peer review, test design review, management review, inspection, or audit.

Note that conducting effective code reviews is more difficult that it might seem. The difficult part is keeping a format that focuses on improving the product quality, without individuals feeling that they are being accused or disciplined. Some code reviews incorporate code metrics, such as the percentage of lines containing comments, or McCabes Cyclomatic Complexity, in order to make the code examination feel less personal.

Maintainability

One metric that may be generated by a code review is a maintainability metric. This is an indication of how easy it will be to maintain the code base long into the future. Maintainability looks at coding practices to answer a number of questions; Is the code analyzable (how easily debugged)? Is it changeable (i.e. are there hooks for readily adding new data types or functionality)? Is it stable (how much is the system being re-architected from one version to the next)? Is it easy to test that the code is working properly?

Part of maintainability is being able to install an updated version of the software. Sometimes this will be done via the operating system's package manager. If software updates are frequently security related, the software may be designed to check for available updates and install them once the user approves it.

Portability analysis

Another type of test incorporated in some code reviews is a portability analysis. This is a check on whether the code is intrinsically tied to a specific hardware platform or operating system. Portability analysis typically includes an examination of the installation process, how well the application can coexist with other applications, and how readily it can be adapted to run on a different operating system.

The experimental side of portability analysis is a configuration test. This consists of simply compiling and running the software (usually smoke tests) on various versions of operating systems or hardware platforms.

Testing on multiple operating systems is much easier today than in the past because of the availability of high-quality virtualization software. It is now possible to install a couple dozen operating systems on a single physical computer and run them inside of virtual machines. Another option is to configure a computer to have multiple hard drive partitions to choose from at boot time. Some virtualization software can mimic multiple types of video card and other hardware components. However, testing hardware dependent software still relies on a good collection of physical computers to run tests.

It is prudent to test where problems frequently occur. For example, the Python ecosystem is notorious for coexistence issues. These arise when one Python program requires version 2 of a given prerequisite, and another Python program requires version 3 of the same prerequisite. This is made worse by the Python development culture of getting as much functionality as possible from other tools that you import into your program, thus resulting in a large number of dependencies between Python programs and modules. A number of bandaids have been developed to combat this problem, including anaconda, pip, and virtual environments. Some software developers bundle a compatible Python version with their software, which gets their software working but may break other Python-based software. Some large computing systems use an environment module system, such as LMOD, which allows installing and managing multiple versions of Python, compilers, math libraries, and other software. Some software developers eschew Python to avoid these problems.

Another common coexistence issue arises from the use of dynamic linked software. This is compiled software that accesses other library files on the system at run time. Installing an update to the operating system might change the library file that the software is calling. In theory, this can push out small bug fixes that the software will automatically see the next time it is run. In practice, this serves as a mechanism to break software much more often than it fixes software. An alternative is static linking, which creates a larger executable file that has all of the prerequisite software bundled into it. A second alternative is bundling all of the needed libraries and utilities with the software.

Portability should be considered early in the project. One project may set a criterion of writing nongraphical software that works on a handful of popular Linux distributions. Another project may define success as having a graphic interface that works on Linux, Windows, and Macintosh. These marketing decisions can and should affect code design that is done before the first line of code is written.

Documentation testing

Documentation testing (not to be confused with test documentation, which is discussed in the next section) is a test of whether the program documentation has been well written.

The first step in testing documentation is to have a clear picture of what good documentation looks like. Too often, documentation departments are overly focused on document format, grammar, and identifying copyrighted or trademarked material. It is apparently too easy to miss the bigger picture of writing good documentation. Here is a working definition of documentation quality.

- Bad documentation incompletely documents installation and functionality of the software. Lack of an index or search feature in the documentation falls under the “bad” category as well.
- Mediocre documentation fully documents installation and functionality.
- Good documentation documents not only installation and functionality, but also discusses when and why you would choose to use that functionality. This may take the form of a tutorial, theory manual, or many examples of how the software is used.
- Great documentation does all of this in such a clear, understandable, and interesting manner that the new owner of the software enjoys going through the manual and learning to use it.

Some of the best testers of documentation are people with knowledge in the field, but who have never used the software. We have seen multiple companies that ask new employees to work through the documentation both to learn the company’s products, and to make notes about things that were difficult to understand from the documentation.

Test documentation

You can have the best software testing in the world, but it is useless if no one is aware of the results. Too often test results go into a file, which no one looks at. Some companies have very formalized test documentation systems where detailed reports are created, routed, and discussed in meetings. Some have reports automatically generated and emailed to the developers and product manager. Some testing systems are so well automated that the test harness will put an entry in the bug-tracking database.

The interaction between the software tester and the software developer is usually through tickets in a bug-tracking database. There are many bug tracking products available, both commercial and open source. Many are accessed via a web browser. These are often configurable to the needs of the organization. Most often the information in the ticket includes bug severity, how to reproduce it, who filed the bug, who is assigned to fix it, what version of the software the bug was seen in, and when the bug was resolved. Feature requests will get filed as bugs, so there should be a way to identify those cases. The user support department will search that database for issues users are seeing, so there should be a mechanism to allow them to search for only bugs known to be in versions that were released to users.

Bugs will be filed by users, usually through the support department. The support department will catch cases where it's really a user training issue, not a bug. If it is a bug, it is good business etiquette to respond to the user acknowledging the receipt of the information. Some organizations respond to users again to let them know what version of the software will have the bug fix incorporated.

If the support staff file it as a bug and it is later determined not to be, don't just delete the entry. Take a minute to determine why someone thought it was a bug and correct the problem. Incomplete documentation and confusing user interfaces can both make it look like there is a problem with the software when there isn't.

Some organizations make their bug databases visible to users, and others don't. The downside of transparency is that the sales staff for competing products will cite items out of your bug database, usually out of context, as a reason why people should not purchase your products.

A testing system can be useless if the test documentation is badly designed. If automated tests flag many errors that are not really errors, developers will stop looking at test results. If tests are made so lenient that errors are not caught, then the testing failed. If test outputs give pages of detailed information but lack a succinct, clear description of what is wrong, developers will not want to look at them. A good testing system is an easily used convenience for the software development staff.

Some software testing systems kick out piles of statistics such as a percentage of failed regression tests, or ratios of severe bugs to minor bugs. It is easy to get focused on these numbers and lose focus on what is important… making good software. It is usually best to keep statistical analysis of testing results to a minimum.

There isn't a right or wrong way to document testing results. The important thing is that the format works with your company culture to ensure that test results get to the correct people and that those test results subsequently guide the development team in improving the software.

From a business management perspective, keep in mind the adage “you become what you report”. If your system reports how many new lines of code were generated each week, software developers will feel pressured into developing software quickly even if it is full of bugs. Thus this metric should only be recorded if the most important requirement of the project is to quickly develop marginally functional software. A more useful tactic is to generate reports of metrics that encourage good coding practices. Some of these might include the percentage of code lines that have comments, whether variables are initialized, whether the variable naming system was utilized, and how many bugs were tracked back to a given section of code.

White box and black box testing

White box and black box testing are not type of tests, but rather the conditions under which testing takes place. White box testing means that the person creating the test has access to the source code. Black box testing means that the person creating the test has no knowledge of the source code.

White box testing always occurs since it is impossible for a programmer to write software without testing it. Admittedly quick and dirty software development work may incorporate manual testing only once as it is developed, without writing down or automating those tests to be run again in the future. If the program is very small, not having automated repeatable tests may work fine, but it’s a recipe for eventual disaster on larger projects. White box testing brings with it an advantage in generating tests with good code coverage because the person writing the test knows which algorithms are in the software and can thus generate tests designed to exercise them. The disadvantage of white box testing is that the test developer tends to get focused in on the things they designed the function to do and things that are indicated by the algorithms in the function.

Black box testing frees the tester to think about the bigger picture of ways the software might be used. They can better think out of the box in trying to come up with unusual or invalid input data, or unusual sequences in which to call the program functionality.

There is indeed a different mindset to software testing. Software developers spend most of their time focused on how to make things work. Software testers spend most of their time thinking about ways to break software. In the book “Herding Cats” on managing software development teams, the author suggests handling developers who chronically do dumb things by moving them to the testing group where that mindset is an advantage.

Another type of black box testing is having the software tested by someone who is a subject matter expert in the type of task that the software performs. For example, software for doing drug design should be tested by a scientist with drug design experience.

The bigger picture is that a range of skills are needed to create top quality software. If everyone doing the software development on a piece of scientific software is a scientist, the scientific functionality might be great but the code almost always comes out poorly engineered and thus difficult to test and maintain. If the development team contains all computer scientists but not a subject matter expert, the software is often not very usable for the end user. This latter situation happens all too often when the software development is assigned to a different department or an outside contractor who doesn't budget for a subject matter expert. Likewise, a good test engineer not only tests software, but helps the team to build a testable piece of software and develop a culture of testing and fixing bugs as the code is written. No matter how good the technical team members are, there is nothing so good that bad management can't screw it up, so the manager has to do a good job as well.

User testing

No matter how good your developers and software testers are, a few bugs are going to get out to the user community. It's inevitable that amongst thousands of users, someone will try a permutation of features and input data that was never considered by your handful of developers and software testers. Some organizations just accept a number of bugs discovered by the user community as an expected part of software development, while others utilize the user community as part of their testing process.

Using users for testing is most often done by putting out a prerelease version. This is often called a "beta version" or "release candidate". The advantage of giving this version a special name is that the users don't expect it to be a stable release. The understanding that they are choosing to try out something that probably has a few bugs makes users more tolerant of flaws in this version. It is best for the prerelease version to come with more obvious requests for beta testers to report bugs they find. This is also a form of advertising because the early adopters who chose to try out the upcoming product start talking to others about the features they like.

One problem with prerelease versions is that some users may continue using this imperfect version. Over time, they forget that it is a testing version and just view your organization as putting out buggy software. Some organizations mitigate this by waiting until the product is extremely stable before putting out a beta version. Other organizations time bomb the prerelease to stop working at some future date that they are certain will be well after the stable release is available.

Some organizations offer incentives to users who submit bugs. For example, the first user to submit a previously undocumented bug may receive a free t-shirt with the company logo. Of course, if you misestimate the stability of the software, you may spend a lot more money than expected on mailing out t-shirts.

Software testing certifications

Certifications are an increasingly important part of the information technology field. Software testing is no exception. Earning a software testing certification can be a valuable item on your resume. It may also be required by some employers, or for some jobs. A software testing certification can be a plus on the resume of a software developer as well. There are a number of organizations that issue certifications in software testing.

The most active and well-known software testing certification organization is ISTQB, the International Software Testing Qualifications Board. ISTQB is an international organization, which is recognized by national organizations in multiple countries. ISTQB has a slowly expanding array of software testing certifications. Their certifications are organized into tiers, called foundation, advanced, expert, and specialist. These are given acronyms such as CTFL (Certified Tester Foundation Level). When this paper was written, twenty two certifications were available. ISTQB examinations can be taken at testing centers in most cities. For further information, see http://www.istqb.org/

There are a number of software testing certifications offered by Software Certifications, also called the International Software Certification Board. They also have three tiers, called associate, tester, and manager. These examinations are administered through Pearson VUE testing centers. For further information, see http://www.softwarecertifications.org/

There is an array of software testing certifications offered by International Institute for Software Testing. Unlike the previous two organizations that administer certification tests, this company is focused on teaching classes and seminars that lead to certifications from the same organization. Unlike ISTQB, which is non-profit, this is a for profit company. For further information, see http://www.testinginstitute.com/

There are also vendors of software testing tools that offer certifications for their tools, such as Mercury, Segue, Rational, and Empirix. A small number of questions on software testing are often included in the exams to earn certifications in various programming languages as well.

Recommendations

Anyone involved in software development should have some plan as to how they will ensure the proper functioning of their code. Quality should be part of the design of the code and the organizations coding practices. Testing helps to find bugs that inevitably arise in any project, but no amount of testing will fix a badly architected program.

Ideally, software quality and testing is a priority for everyone involved from interns up to the company CEO. Some companies have internal competitions or ratings for which development team has the best testing implemented. This technique of establishing a corporate culture can be much easier for managers than having to nag developers to write higher quality software.

In practice, company financial concerns sometimes push software versions to release before the software development team feels it is ready. This problem is mitigated by doing as much testing and debugging as possible at every stage of the software development so that it is always as close to release quality as practical. It’s better to have fewer features implemented well than to have a bunch of features that don’t work correctly.

Here are some best practices to consider.

- Consider the risks. What types of failures would have the most severe impact on the software users or your business?
- Plan out how testing and fixing bugs will be done before starting to write the software.
- Have testability as one of the design criteria for the software package. A text file output is easily comparable with a saved, correct output. Even better is a text file designed to be exactly the same every time (no dates or numbers printed to the full precision that will show unit round differences).
- Get your test automation tools in place very early in the project.
- Test and fix problems as the software is written. Many development organizations have nightly builds with automated regression testing. Some have continual testing, every time a piece of code is checked in or as close to it as possible.
- Use multiple types of software tests.
- If software development will take more than thirty days, create unit tests as each function is written, starting from the first line of code.
- Make software quality a part of the job for everyone involved.
- Have software developers, testing experts, subject matter experts, and management all involved in every step of the software development process.
- Adapt software testing tools and techniques to be well fitted to your project.

Studies on software development have shown that it is cheapest to fix bugs as early as possible. If a bug never occurred because the problem was addressed at the requirements gathering and code design phase, very little labor and cost went into dealing with that problem. If a developer finds an issue with a unit test and fixes it immediately, the cost was a couple hours of the developer's labor. If functional testing catches the bug when the version is nearly complete, more time and thus labor cost went into finding the problem and it takes developers more time to work through the whole code to find and fix the source of the problem. The most harm is done when users find the bug after the stable release has shipped. In this case, some customers will switch away from your products or not buy your product because of these issues, thus potentially losing millions of dollars in future revenue.

Software developers need to occasionally stand back and look at the larger picture related to software quality and testing. It isn't just about earning a paycheck and having the company make money. A well-written piece of software can be a source of pride for the development team. A badly written piece of software can be an embarrassment that comes back to haunt you in future job promotions and employment interviews.

Managers need to stand back from current financial goals to see the bigger picture. This is perhaps best summarized by Peters & Waterman in the business management classic "In Search of Excellence". The key to the most successful companies in history is ultimately very simple… customer satisfaction and repeat sales.

For further information

AMJ Hass “Guide to Advanced Software Testing” Artech House, 2008.

J Whittaker, J Arbon, J Carollo “How Google Tests Software” Addison-Wesley, 2012

D Graham, E Van Veenendaal, I Evans, R Black “Foundations of Software Testing: ISTQB Certification Intl” Thomson Business, 2008

A Spillner T Linz; H Schaefer “Software Testing Foundations: A Study Guide for the Certified Tester Exam, 2nd Edition” Rocky Nook, 2007

R Black “Advanced Software Testing - Vol. 1: Guide to the ISTQB Advanced Certification as an Advanced Test Analyst” Rocky Nook, 2008

R Black “Advanced Software Testing, Volume 2: Guide to the ISTQB Advanced Certification as an Advanced Test Manager” Rocky Nook, 2008

J Mitchell, R Black “Advanced Software Testing Vol. 3 Guide to the ISTQB Advanced Certification as an Advanced Technical Test Analyst” Rocky Nook, 2015

A Spillner, T Linz, T Rossner, M Winter “Software Testing Practice: Test Management: A Study Guide for the Certified Tester Exam ISTQB Advanced Level” Rocky Nook, 2007

B Hambling, P Morgan, A Samaroo, G Thompson, P Williams “Software Testing - An ISTQB-ISEB Foundation Guide Revised Second Edition” British Informatics Society Limited, 2010

R Black “Pragmatic Software Testing: Becoming an Effective and Efficient Test Professional” Wiley, 2007

WE Lewis “Software Testing and Continuous Quality Improvement, Third Edition” AUERBACH, 2008

R Black “Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing” Wiley, 2009

P Hamill “Unit Test Frameworks” O'Reilly, 2005

C Kaner, J Bach, B Pettichord “Lessons Learned in Software Testing” Wiley, 2002

M Pezze, M Young “Software Testing and Analysis: Process, Principles and Techniques” Wiley, 2008

J. McCaffrey “Software Testing: Fundamental Principles and Essential Knowledge” James McCaffrey, 2009

JH Rainwater “Herding Cats: A Primer for Programmers Who Lead Programmers” Apress, 2002

TJ Peters & RH Waterman, Jr. “In Search of Excellence, Lessons from America's Best-Run Companies” Harper & Row, 1982

http://www.ibm.com/developerworks/library/j-test/index.html
This is a nice, short introduction to using unit and functional testing by J Canna

http://www.scribd.com/doc/6610947/Unit-Testing
This is an introduction to unit testing by R Parkin