A Software Testing Methodology
Request For Comments (RFC)

Michael P. Zeleznik, Ph.D.

1. Introduction

Following is a strawman write up of the issues involved in creating a more structured, comprehensive, documented software testing process, and a methodology to achieve that goal. This is an RFC, open for discussion. Please comment on agreements, disagreements, or what is missing.

This does not address quality software development concepts (that should be another discussion). Those are important, but no matter what is done there, the bottom line is still end point testing. Does the system work correctly?

To get the picture, please read the first 2 short sections: Current Situation in Many Places, and Strawman Proposal. The remainder expands on those issues.

Table of Contents

2. Current Situation in Many Places

In many environments, each software developer tests their own segments of the system in ways they feel are appropriate, and overall system (end point) testing is carried out by others, and at alpha/beta sites. Although such testing might be extensive, it has the following limitations.

1. The end point testing does not necessarily reflect any code-specific testing needs from each of us. Much thought goes into developing robust software that handles special cases or that performs various sanity/integrity/safety checks. These are tested at development time, but we must insure they work in the future as well. End point testing will likely not exercise these, unless designed to do so.

2. No comprehensive record exists of either what needs to be tested, or what is currently tested, making it very difficult to assess where we are, or where we need to be going.

3. Since testing is so difficult and time consuming, it is not done as often or as comprehensively as it could be. Bugs (especially side effect ones) introduced today might not be found until well into the future. This is especially an issue with the addition of substantial new functionality (e.g., 3D planning or visualization), or with changes that touch large amounts of code (e.g., the multiuser database).

4. The testing processes might not be carried out as completely or consistently as they could. For example, specific procedures might not be documented, required data might be unavailable or inappropriate at a later time, or only a subset of tests is run to reduce time or avoid monotony.

5. If responsibility for some function is moved from one person to another, how are the required testing procedures and issues passed along?

Any of these can lead to reduced effectiveness of testing, which is not only dangerous, but it can increase the cost of fixing problems (when discovered later rather than sooner). We need a plan that can evolve as we learn, which addresses these issues. I realize we are obtaining standard CT data sets, but that is orthogonal to the problem of creating the process in which they are effectively used.

3. Strawman Proposal

Each of these is discussed in the noted section(s). Especially important, but possibly not obvious, are 3, 4, 10, and 11.

1. Appropriate software development concepts are important in producing error free code, but the bottom line is always end point testing. Does the resulting system produce correct results (Sec 3)?

2. Development of our testing process must be iterative and evolutionary. Given our limited resources, we must strive for the biggest bang for the smallest buck; learn-as-we-go, execution-based testing (Sec 3, 5).

3. Testing must cover both black box (testing to specifications) and glass box (testing to code) aspects. This is very important (Sec 4, 5).

4. As much as possible, both black box and glass box tests should utilize the highest level inputs and outputs (end point testing). Only when absolutely necessary should module level tests be required (Sec 4, 5).

5. When possible, provide for stand-alone subsystem testing (e.g., CT interfaces, DICOM, visualization), observing above goal of using high level inputs and outputs (Sec 5).

6. Automate as much testing as possible, and clearly document the manual parts, to facilitate testing as much of the system as possible when any change is made (Sec 5).

7. Automating the test process requires answers to unknowns, such as how to drive the GUIs, and to obtain and analyze graphics outputs (Sec 5).

8. Minimize the amount and complexity of test data. All test requirements should be served by a small number of shared test data sets (Sec 5).

9. A record of the test purposes, procedures, data, how to interpret results, and results must be maintained in the "test database" (Sec 5).

10. This test database must be specially structured. This is critical to maintain integrity, and document where we are and should be going (Sec 5).

11. All test database elements, including test data, must live under RCS, with version names based on system version names (Sec 5).

12. Whatever plan we have, someone must make sure that it is carried out!

I agree, this is a big effort. But I see no alternative as the the number of developers grows or changes, and the amount of altered or new functionality increases.

4. Software quality assurance

Software quality assurance is a balancing act. Many software development concepts and tools exist to help reduce costs and produce code that functions correctly. These include formal and semi-formal specification and verification, structured walkthroughs, code assertions, compiler checks, runtime analyzers, debug walkthroughs, simulations, and both execution- and non-execution- based testing. We all know that it is best to detect problems as early in the development cycle as possible.

Ideally, a development project will assess these issues and define an appropriate quality assurance process covering the entire design life cycle. The options range from no methodology at all, to an extreme like IBM's Cleanroom (including formal techniques and non-execution based testing). Walt mentioned that Microsoft has about 1 tester for each software developer. That'd be nice.

Given our limited resources, we must strive for the biggest bang for the smallest buck, and find the fastest way to get there. Software development aspects are very important (and should be addressed in another discussion), but the bottom line is still the final end point testing. Does the system do what it is supposed to? For a given input, does it produce the correct output? For our purposes, execution-based testing is the only ticket.

We can certainly address the other quality assurance issues (we each employ our own already), and begin to create a plan, but the NUMBER ONE PRIORITY must be to resolve the testing issues and get that under our belt ASAP.

5. Execution-based testing: falls into following two extremes

5.1 Black box testing (testing to specifications)

Tests high level inputs and outputs of the system for correct functionality based on the system specification, but without regard to the underlying implementation.

For example, given a set of treatment plan parameters, is the resulting dose matrix correct and do the BEVs appear correctly? Or, given a set of beam parameters, is the profile correct? Such testing would not, for example, exercise known problem areas in the code, attempt to insure code coverage, or extensively exercise data extremes or boundary conditions.

5.2 Glass box testing (testing to code)

Also called white/open/clear box testing. Tests the actual implementation by exercising extremes, boundary conditions, known error conditions, coverage issues; any issues arising from knowledge of the actual code.

Hopefully, most of this will be done at the end point level with judiciously designed high-level inputs (see Sec 5), but some will require the use of lower level inputs or even modifications to the actual code (e.g., inserting code to cause conditions).

For example, at the end point level, one can create absurd beam parameters, degenerate structures, or inconsistent machine data to test those sanity and bounds checks, or create self-intersecting contours to test the visualization contour-intersection code. As another example, when critical bugs are discovered and fixed in support tools (e.g., IBM DX), it is important to construct tests to warn of their recurrence in the future. Reintroduction of previously fixed bugs is always a possibility (it happened routinely with cisco routers; with IBM it could occur in merging our special version with the next release). A number of critical DX bugs can be tested with the correct high level inputs.

Some tests will require lower level inputs. For example, the RTP software insures that beam block outlines do not self-intersect. However, the visualization beam model generator may include a sanity check for this. This can't be tested at the user-interface level, since the RTP system won't allow such blocks to be designed. However, they can be designed externally, and inserted into the high level test data set, and visualization tests can still be run at the high level.

However, some test conditions can not be handled by inputs alone, since they only occur if software bugs exist. For example, as a consistency check, the results of a new implementation might be compared with those from an old implementation. Ideally, this will never fail, but its ability to detect and report the problem should be tested. Otherwise, how can we know that it will work when needed (especially if the code is modified over time)?

In such cases, one simply must test by modifying the code to cause the condition to occur. This must be enabled for testing, then disabled for runtime. The developer must attempt to insure that when in test mode, it produces some noticeable problem or at least notification that this mode is enabled. The obvious concern is accidentally leaving the test mode enabled. This might be difficult, or dangerous, to include in an automated test procedure.

6. A plan of attack:

6.1 Development must be iterative and evolutionary

Development of the test process must be iterative and evolutionary, just like any other software development effort. We will never have the resources to undertake the type of ideal, comprehensive testing that any of us can imagine. Procedures must be implemented, even though they will be incomplete, and many unknowns and intractable issues will remain. We will learn as we go. If a new test is seen to be necessary, it must be integrated into the routine test suite.

6.2 Test as much as possible with each change

It is important to test as much of the system as possible, when any change is made. This may seem like overkill, but as the number of developers increase, along with the number and scale of software changes, it will be increasingly difficult to insure that changes are localized.

6.3 Automate as much as possible

Thus, we must automate as much of the test suite as possible. As more comprehensive testing is defined, the process will become very tedious, especially trying to test as much of the system as possible. No person can repeatedly and accurately carry out such a mechanical task. Where manual testing is required, documentation must exist on how to carry it out and how to interpret the results.

6.4 How to drive inputs and obtain outputs

Automated testing requires a way for a program to drive the system inputs and obtain the system outputs. This is simple for terminal-based input/output but less simple for a GUI-based system with multiple graphics windows as outputs. There are tools for driving the GUI, but I am not familiar with them. Obtaining the window outputs could be done via screen captures (or perhaps tee-ing the windows to files?). Although some code is very amenable to automatic program control (e.g., Curtis' visualization control panel), it may be dangerous to test in ways that differ from what occurs in normal operation. I might rather test by mimicking the user pointing and clicking, than by assuming I can emulate this correctly with some modified version. We have to think about the pros and cons of this issue.

6.5 Test at end points

Ideally, all testing should be carried out using the highest level inputs and outputs (end point testing). The module developer should strive to develop end point tests that will adequately exercise the module internals, and see that these are incorporated into the test suite. Only when absolutely necessary should module level test data and routines be required, especially true for tests requiring code modification. Trying to keep module level test procedures and data up to date is difficult and time consuming as modules and their operating environment are changed (e.g., module interfaces change, data formats change, global requirements change, etc.). Further, the test environment and operating environment can differ in ways that reduce the value of those tests. By testing at the high level, such module changes are usually transparent; one might need to introduce additional test data and procedures for new or altered functionality, but likely will not have to change the data or procedures for previous tests (except when the RTP database or user interface are changed). Further, limiting the number of interfaces between the test procedures and the system might reduce the complexity of the testing tools.

6.6 Provide for isolated subsystem testing.

It is reasonable to provide for isolated subsystem tests (e.g., CT interfaces, the visualization tool, or DICOM tool), as long as they utilize high level end point inputs and outputs. For example, the visualization subsystem can be tested stand-alone, as long as an appropriate RTP database is provided. This does not preclude the need to fully test end-to-end, but allows module developers to test subsystems more often.

6.7 Minimize amount and complexity of test data.

In order to keep the test data tractable, utilize common data sets for multiple tests. For example, a single set of geometric object "structures" could be used in the visualization tests of structure orientations, BEVs, beam model orientations, and even dose volume histograms. Hopefully they will work in other xbeam tests as well. The same might be true for a single set of standard beams. Certainly, this will require cooperation among us all to get it right. I believe it would problematic if we each went off independently, creating endless special test data sets.

6.8 Documentation (see also Sec 9, 10)

All tests, from high-level black box testing down to module specific glass box tests, must be documented. A record must exist of all desirable tests and their purposes, whether or not implemented. For all implemented tests, the record must include the test programs, procedures, data, how to interpret results, and test results. This is mandatory to insure the integrity of the test process. It allows us to readily assess what we should be doing, what we are doing, and what we need to do, and provides a critical resource for maintaining the test suite (e.g., helping prevent a developer from accidentally removing or altering tests or data in ways that might negate the goals of other tests).

This is especially important for glass box testing. Whereas black box testing is relatively static and well-defined from the high-level system specifications, much of the glass box testing is dynamic, defined on-the-fly, and its purpose can be very non-intuitive. Without adequate records, this type of information is easily lost (e.g., the programmer forgets it, or it is not passed on to new programmers), and test processes might later be altered in ways that negate their original intent, because no one really understands why they are there anymore. For example, I will include tests to check for previously discovered subtle bugs in the IBM DX Render and Histogram functions (both critical for correct calculation of dose volume histograms). A slight change to these tests could render them useless.

The structure of this information is critical, and the next two sections go into this in detail. Skip sections 9 and 10 if you are not interested.

6.9 Logical structure of the test database information:

The information associated with the testing process can not be unstructured (e.g., just a set of doc files attached to the tests or test data). It must be structured to allow the following to be quickly and unambiguously determined:

1. Complete list of everything that needs to be tested and why.
2. For a given need, which test(s) handle it (if any).
3. For a given test, which needs it handles.
4. For a given need, which subsystem it is associated with (if any).
5. For a given code segment, which need it is associated with (if any).

Simplified picture (read on if this makes no sense):

```
SubSystem_A       List_of_Needs      List_of_Tests
 code_file_1
    need_1 <-------> need_1 <---------> test_1
    need_3 <-------> need_3 <----‘
 code_file_2
    need_4 <-------> need_4
    need_2 <-------> need_2 <---------> test_2
```

Without this structured information, it will be impossible over time to determine what the test suite really achieves, and to maintain its integrity. I saw this same situation in my dissertation in security. There it was "vulnerabilities" and "safeguards" to address them; here is is "test needs" and "tests" to address them.

In the following, the word "test" generally means everything associated with a test; test routines, data, documentation on how to run and interpret results, and the results.

When the need for a particular test (black or glass box) is first seen, the "need" must be recorded on a Need_List so that it is not forgotten. It may be a while before it is implemented, if ever. If a test to address this need is later implemented, it is recorded on a Test_List, and unambiguous links must exist between the need and the actual test, and from the test back to the need(s) it handles. A given test may handle many needs.

Without the link from need to test, someone might later introduce another test for the same goal, resulting in multiple tests that are probably out of sync. This can be dangerous since one test might no longer test for what it says it does, and it adds to the maintenance headache. Also, if the need ever changes, the link allows one to readily find and reassess the appropriate test.

Without the reverse link from test to need, one could not hope to understand all the purposes of a test (some might be very subtle). As such, test modifications would be dangerous, possibly negating some of the original intent; the needs list would say the need is addressed, but the test might no longer address it. By instead going back to the need, one can attempt to fully understand the test before modifying it.

Separate Need_Lists should exist for black box and glass box needs. The black box list would comprise the needs associated with the system high-level specifications. The glass box list would

comprise needs based on code specific issues and should be segmented according to subsystem (e.g., visualization, CT interface, DICOM, etc.). It will become much more detailed and involved than the black box list. We can have separate lists for some subsystems to keep things manageable, as long as there is ONE MASTER WAY of finding them all.

Having separate black box and glass box Test_Lists would not work in general, since a given test can address both types of needs. However, separate Test_Lists can exist for subsystems or other logical divisions, as long as there is ONE MASTER WAY of finding them all.

In the glass box Need_List if a need is code specific (as it probably is), it must have pointers back to the relevant subsystem (and code file(s) where possible). In those code files, the relevant code segments must have comments pointing to the need that addresses them. The comments would have a specified header format.

Without the link from code segment to need, there is higher risk of someone later looking at the code, seeing an issue that is already addressed by some need but not realizing this in scanning the Need_List, and then producing a new need and test for the same purpose. With the comments link, one is more likely to see the need.

Without the link from need to subsystem, test modifications would still be dangerous. It might be difficult to find the code segments that correspond to a given need, which would necessary to understand how it can be safely changed and still achieve its goals. With the link, the code segments can be found by a simple text search for the special comment format, once back at the subsystem or individual code files.

Aside from all of the above benefits, our testing issues are now fully documented by these lists. The Need_Lists are THE definition of what needs to be tested, and what is currently being tested. The Test_Lists are THE definition of how they are being tested.

Clearly, maintaining the integrity of these links is critical. We must discipline ourselves, or they will become worthless. With appropriate implementation, and sufficient structure on the lists and code segment comments, simple tools could automate some of this, and warn of some inconsistencies.

The following simple example (glass box) shows 5 needs (1,2,3,4,5) for 2 subsystems (VIS,CT) handled by 3 tests (A,B,C). Each need points to the test(s) that address it, each test points back to the need(s) it addresses. Each need also points to the subsystem(s)/file(s) it addresses, and those files have comments pointing back to the needs. In reality, each need would also have comments on its purpose, and each test would also point to everything associated with the test (data, procedures, etc.).

Need_List		**Test_List**	**SubSystem/CodeFile**
Need: Tests:	*SubSys/file*	*Test: Needs*	*SubSys/File: Comments through code*
1: undone:	VIS/file1	A: 5	VIS/file1: /* NEED 1, details... */
2: B:	VIS/file1	B: 2,4	VIS/file2: /* NEED 4, details... */
3: C:	CT/file1	C: 3	CT/file1: /* NEED 3, details... */
4: B:	VIS/file2		CT/file2: /* NEED 5, details... */
5: A:	CT/file2		

6.10 Physical structure of the test database information

THIS NEEDS WORK. These are just first thoughts, and I'm sure it can be done better, especially considering tools to automate some of it. Conceptually, a matrix approach makes sense (e.g., a spreadsheet), but in practice I think this would become unwieldy due to size. Any better approach is welcome.

DEFINITIONS:

need_uid == unique ID for each need in any Need_List

test_uid == unique ID for each test in Test_List

subsys_uname == unique ID name for each subsystem (vis, CT, DICOM, etc.).

uids might be just be: "ethernet_id.date.time".

FILE FORMATS (all are text files):

In a code file, comments are as shown here (key word is "TEST_NEED"):

/* TEST_NEED: need_uid: blah, blah,... */

The "Need_List" format is:

need_uid: test_uid: subsys_uname: (optional file name) (use separate lines for each test/subsys/file that applies to same need)

need_uid: comment: blah, blah, blah, ... (define the need) (to extract everything for a given need_uid, use awk/perl script)

The "Test_List" format is:

test_uid: need_uid

(use separate line for each need handled by the same test)

(to extract everything for a given test_uid, just grep the file)

Test_Procedure_Information file(s) will have following format

TEST_INFO: test_uid:

information on how to run test, interpret results.

where test programs live, what data they want, etc.

TEST_INFO: test_uid:

information on how to run test, interpret results.

where test programs live, what data they want, etc.

...

(if this is too messy, perhaps one file per test is better)

6.11 All test programs, procedures, and data must live under RCS

All test database elements, including documentation, test programs, procedures, data sets, Need_Lists, Test_Lists, and test results, must live under RCS. When a subsystem or entire system is tested, the test database versions get the same name as the subsystem or system versions.

The reasons are simple. Any version of the system might have tests or test data that is incompatible with a previous version. Further, if tests that worked at one time begin to fail at a later time, one must be able to determine if the test data or programs have been changed. It is easy to imagine someone modifying a test program or data set for their purposes, which later breaks your test the next time it is run. With RCS, each test that works on a given date will have a unique name assigned to that version of the test programs, data, and system (perhaps just the system version name). Then it will be simple to see what changed and how. Otherwise this will be a mess to sort out.

6.12 Someone must make it happen

Whatever plan we have, someone must make sure it is carried out consistently over time. Tests must be run, the test database must be kept current, the validity of the test suite must be insured (are we still testing what needs to be tested as the system evolves?), etc. Otherwise, it is guaranteed to fall apart.

7. Examples of testing scenarios:

The following are all specific to the visualization subsystem. The test data sets must be chosen judiciously to exercise as many important conditions, extremes, boundaries as possible. Obviously we hit only a small fraction of all possibilities.

7.1 Automated tests

In the following, "compare resulting display" implies a function that does a graphical comparison with the expected display (might have to be a fuzzy comparison). The displays are used rather than some numeric output since they are the normal high level outputs. Even if numeric output were used, we would still need to show that the displays correctly reflected that data.

Dose Volume Histogram (DVH):

Black Box Test 1: DVH is performed on a test structure and dose matrix. Compare resulting displayed graphs. Do for set of structures and matrices.

Glass Box Test 1: The DX Render function was found to have a 1/2 pixel error in dimensions with an even number of pixels. I must detect if this bug returns. A specific structure (perhaps just one contour) is selected that is known to demonstrate this problem. Do DVH and compare resulting display.

Glass Box Test 2: The DX Histogram function was found to ignore "invalid data" when all data values were exactly the same. I must detect if this bug returns. Same process as for Glass Box Test 2.

Glass Box Test 3: Test data extremes by doing DVHs on structures with no contours, contours too small to render, dose matrices that are both disjoint from or fully contained within structures, dose matrices with no points, etc. Compare resulting displays.

Dose Isosurface:

Black Box Test 1: Generate isosurface for test dose matrix. Compare resulting display for multiple views (caudad, cephalad, etc.). Do for N isosurfaces and N dose matrices.

Dose:

Glass Box Test 1: Generate dose matrix with attributes known to be invalid, that should be detected by visualization import routines (e.g., "space" is not minimum of X and Y size, "space" or "start positions" vary from plane to plane, values out of range, etc.). Verify expected results (e.g., no dose data results, and/or error is reported).

BEV (beam orientation and displayed beam parameters):

Black Box Test 1: Generate a BEV for test beam and set of standard structures (perhaps cubes, pyramids, etc.) arranged to be unambiguous. Compare resulting BEV display. Do for N test beams (different orientations and machine parameters).

Beam Solid Modeling:

Black Box Test 1: Display a test beam in perspective window. For each of N different views (caudad, cephalad, right diagonal, etc.), compare resulting display. Do for N test beams, attempting to create odd shaped collimator and block designs. Can display as surface, edges, etc.

Glass Box Test 1: Create extreme beams (e.g., blocks inside blocks, all blocks outside or inside collimator, etc.) and compare resulting displays.

Structure Solid Modeling:

Black Box Test 1: Display a test structure in perspective window. For each of N different views (caudad, cephalad, right diagonal, etc.), compare resulting display. Do for N test structures, attempting to create odd shapes. Display as surfaces, points, mesh, etc.

Glass Box Test 1: Create extreme structures (e.g., self intersecting, intersecting with vertical and horizontal lines, missing planes, degenerate contours, etc.), and compare resulting displays.

7.2 Manual tests

No examples at this time. Hopefully no manual tests.

8. Issues to Sort Out:

Just a terse list off top of my head. Many more exist.

1. how to drive GUIs.
2. how to obtain and handle output windows.
3. how to compare graphics windows.
4. physical structure of Need_Lists and Test_Lists and linkages.
5. how to save test results into test database.
6. simple tools to help handle test database.

7. how to specify just what was tested at a given time.
8. how to handle tests that require code modification.
9. can we just name test database versions with system version names, or is more needed?

9. References:

Just a representative list here. There are MANY books, journals, conferences, and papers on this subject. I've seen an entire book on just software inspection. Trick is finding/using only what we really need.

9.1 WEB Sites

Quality Information Center: http://www.nicom.com/~qadude/myquality.html Lots of pertinent information; I've only begun to explore it. Many pointers to books, periodicals, newsgroups, etc.

Online Quality Information: http://www.nicom.com/~qadude/qualitylinks.html Tons of stuff, but very wide reaching.

Software QA (EPA): http://www.epa.gov:80/docs/emap/html/qc/softwrqa.html. Never saw this; slow link.

9.2 Books

Software Quality Assurance and Measurement: A Worldwide Perspective, Fenton, Whitty, Iizuka, 1995.

The Mythical Man Month, Fred Brooks Jr., 1995 anniv ed. Classic on the whole development process; anniversary ed. has new chaps.

Writing Solid Code, Steve Maguire, 1993. Walt found this one; lots of fundamentally good concepts on developing error-free code.

Software System Safety and Computers, Nancy Leveson, 1995. Haven't read, but see articles below about a 1986 paper of hers that I liked.

ISO 9000 for Software Developers, Charles Schmauch, 1994. High-level introduction to these standards.

9.3 Journals

ACM Transactions on Software Engineering and Methodology IEEE Transactions on Software Engineering IEEE Software Magazine

9.4 Conferences

IEEE COMPASS (Computer Assurance) (annual). IEEE International Symposium on Software Reliability Engineering (annual).

9.5 Articles

(I haven't kept up so don't have newer ones):

Software Safety: Why, What, and How, Nancy Leveson, ACM Computing Surveys, June 1986, p125. Possibly dated, but I found it very useful years ago in presenting a complete framework, distinguishing among safety, reliability, and security, in what was then a relatively new research area.

A Spiral Model of Software Development and Enhancement, IEEE Computer, May 1988. One of the more rational definitions of software life cycle; I'm sure there are newer papers or books on this by now.

