COSC345 Week 22

Test Plans and Testing

9 September 2014

Richard A. O'Keefe

Why test?

If it isn't tested it doesn't work.

Verification and Validation

Validation asks "Are we building the right product?" Did we understand the customer? Requirements analysis, prototyping, early customer feedback.

Verification asks "Are we building the product right?" *Given* a specification, does the code meet it? *Defect testing* looks for mismatches.

Statistical testing concerns performance measurement and the non-functional requirements. Very weak bug-finder. What *is* the distribution of typical uses?

Why Test?

1–3 bugs/100 SLOC of good code

Beizer's "five phases"

phase 0 Testing is just debugging

phase 1 Testing is to show code works

phase 2 Testing is to show it doesn't

phase 3 It's to reduce risk of poor quality

phase 4 It's a mental discipline that gives good quality.

Showing the software works

— one test shows program broken

— no number will show it's right

— statistical testing may mislead

— "conspire with" developers

— think "test failed" bad

Showing the software is broken

— it *is*, so more realistic

— deliberately try to break things

— *want* "test failed"

— don't throw tests away, bugs return

— haphazard testing ineffective

Test design preventing bugs?

— think about tests before coding

— design for testability

— test early and test often

— choose tools that support testing

Complementary techniques (Beizer)

Inspection Methods walkthroughs, desk checking, formal inspections, code reading. Find some bugs testing misses and *vice versa*

Design Style testability, openness, clarity

Static Analysis Methods strong types, type checking, data flow checking, *e.g.,* splint pointer ownership

Language languages can prevent (pointer) or reduce (initialisation) mistakes, *e.g.,* Java *vs* C++.

Development Process and Environment configuration management, documentation tools, test harnesses.

Exploratory Testing

See en.wikipedia.org/wiki/Exploratory_testing

You cannot preplan all testing

Repeating old tests just says you weren't *that* stupid

The aim of testing is learning

Your test suite grows and changes

After pre-planned test pass, what?

Keep on growing your test scripts!

Testing is not debugging 1

testing is to show program has errors.

Starts with known conditions. Can/must be planned, designed, scheduled, predictable, dull, constrained, rigid, inhuman. Automate it! Use scripts, record/playback... Don't need source code or design, only specification. Can/should be done by outsider. There's much theory of testing.

(Beizer 1990)

Testing is not debugging 2

debugging is to find cause of error and fix it

Starts with unknown conditions; don't know what we'll find. Duration and requirements not predictable. Is like science: examine data, form hypotheses, perform experiments to test them. Creative. "wolf fence" method. Needs source code, detailed design knowledge. Must be done by insider. Not much theory (but look up "algorithmic debugging" and "rational debugging"). Tools can help. Interactive debugging is a huge time waster, can't always be avoided but try!

(Beizer 1990)

The "Wolf Fence" algorithm for debugging

CACM Vol.25 No.11, November 1982, p 780

1 Let A be the area holding the wolf

2 Make a fence splitting A into B, C

3 Is the wolf howling in B or C?

4 Repeat until the area is small enough.

Assumes Wolf is fixed in area A

Assumes You can build a fence (print statement).

22.1 The testing process 1

1. Unit testing tests single components (functions, even data files)

2. Module testing tests encapsulated clusters of components (a class, or perhaps a package)

3. Subsystem testing tests bound groups of modules *e.g.*, a program) typically looking for interface errors

Problem: easy to test a single function in Lisp, not easy in C. Need a "test harness" that the component/module/subsystem can "plug into" so that tests can be fed to it and outcomes observed. *Plan for this!*

The testing process 2

4. System testing tests entire suite (*e.g.*, Java applet + browser + server + data base) looking for interface errors and checking against requirements, using designed test data

5. Acceptance/alpha testing tests with real data in realistic situation (maybe at customer's site)

6. Beta testing uses friendly customers to get realistic tests

Problem: beta test feedback is really too late. Need early customer feedback. Even with prototyping, shouldn't skip beta, but can be in-house.

22.2 Test plans 1

test process describes phases of process

requirements traceability links requirements to tests

tested items lists which things are to be tested

test schedule says who is to test what when

test recording procedures say how to record results for audit

hardware and software requirements say how to set up for a test

Test plans 2

constraints time/budget/staff needs/limits

***test items** what the tests actually *are*

***outcomes and actions** what we expect and what to do next

good outcomes: what? how detected?

expected poor outcomes: what? how detected? what action?

bad outcomes: how recovered from? what action?

Test plans 3

— There's a test plan for each level

— There's a test plan for each module

— Develop each plan as soon as design complete enough

— Keep test plans under version control and revise 'em

— Keep test items under version control and revise 'em

— Word processors are evil.

22.3 Testing strategies

1 Top-down testing

2 Bottom-up testing

4 Stress testing

5 back-to-back testing

Top-down testing

— test top level before testing details

— don't trust details, use "stubs"

— stub handle few cases, or just print messages

— commonly used and useful for GUI testing

— also useful for compilers

— aim is to test as early as possible

Bottom-up testing

— test service provider before service client

— requires “test harnesses” that look like clients

— great for reusable components (libraries *etc*)

— distribute tests with reusable components

— easy in Lisp, also in Java with BlueJ & JUnit

— aim is to test as early as possible

Stress testing

— test system load or capacity

— *e.g.*, give Word a 4,000 page document

— *e.g.*, simulate everyone ringing OU at once

— it's testing: try to make the system *fail*

— tests failure behaviour: load shedding? crash?

— may flush out hard-to-catch bugs

— may have trouble with repeatability

paging, interrupts, fixed table sizes (readnews)

Back-to-back testing 1

— also known as using an oracle

— need two or more versions of program

— run tests against *both* versions

— result comparison non-trivial, see `tools/pcfpcmp.d`

Oracles provide right answers

— common sources of oracle:

→ old version of program

→ executable specification

→ prototype

→ N-version programming

— we do N-version programming for the Programming Contests and do back-to-back testing.

Test cases include

Scope —says what component is to be tested

Test data —the *input* for a test

— data may be generated automatically

Pass criterion —what counts as success?

— pass may be explicit data to match

— pass may be a programmed function

What next —what to do if test fails?

A sample test script

```
echo Test the 'foo' program.
failed=false
for i in foo-test/*; do
   foo <$i/in >tmp
   if [ $? -ne 0 ]; then
      failed=true; echo "foo $i failed (crash)."
   elif cmp tmp $i/out; then
      echo "foo $i passed."
   else
      failed=true; echo "foo $i failed (wrong output)."
   fi
od
exec $failed
```

Directory structure for example

Case	Input	Output	Notes
1	foo-test/1/in	foo-test/1/out	foo-test/1/notes
⋮	⋮	⋮	⋮
20	foo-test/20/in	foo-test/20/out	foo-test/20/notes

The ‘cmp’ command might be too strict, see tools/pcfpcmp.d for an alternative.

What doesn't that catch?

Doesn't catch file system changes

Doesn't catch unintended reads

Superuser can set up a "sandbox" file system for testing and run tests inside it using 'chroot'

Anyone can record file access times (find . -ls) before and after test and check for differences

Analogy to variable access/mutation inside a program; binary instrumentation can help

What if you aren't superuser?

It's worth having a testing machine anyway.

Use VirtualBox to set up testing environments where you *are* superuser.

NB VirtualBox and other VM systems are a *huge* benefit for testing.

Use an emulator like Bochs (x86) or Hercules (System/370).

Use interposition to fake an OS layer

Black box testing

— metaphor: component inside opaque box

— derive test cases from specification

— you must *have* a specification

— how would you test Compaq's C compiler?

— try typical inputs, but also

— try "boundary cases"

Fuzz testing

A form of black box testing

Feed random data to component

Look for crashes or hangs

Barton P. Miller and students

See ~ok/COSC345/fuzz-2001.d

Utility of fuzz testing

1990: 25-35% of UNIX utilities crashed

1995: 15-45% of utilities crashed; 26% GUIs

1995: only 6% of GNU and 9% of linux

2000: at least 45% of Win NT 4 and Win 2k

2006: 7% of MacOS X utilities crashed

2006: 73% of GUI programs crashed/hung

2013: 2 of 48 MacOS utilities (4%) crashed (me)

Why does fuzz testing pay off?

failure to check error return values

non-validating input functions

broken error handling

pointer/array errors (buffer overflow *etc*)

signed characters

races

“interaction effects” (XSS)

delegating to broken components

Equivalence partitioning

Partition input space into things that should be treated the same. Expect *lots* of partitions.

Typical cases inside each partition

Valid boundary cases should be tried (sqrt(-0))

Invalid boundary cases should be tried (sqrt(-MINDOUBLE))

Zero is a number: try empty inputs

One is a number: try 1 input, all inputs equal, *etc*

Search problems: found at once, found at end, obviously missing, missing but similar to present thing.

Assumption/Common Causes 1 (Beizer)

Domain testing/equivalence partitioning assumes "that processing is OK but the domain definition is wrong".

Zero(a) $+0.0$ and -0.0 are distinct

Zero(b) comparing floats against 0 is usually wrong

Inconsistent spec domains in specification overlap;
programmer separated them

Incomplete spec domains in specification don't cover input space;
programmer had to guess

Common Causes 2 (Beizer)

Overspecification so many constraints on domain that it has no values; code never executed

Closure reversal Using $\geq$ instead of $\leq$ or $>$ instead of $<$ or vice versa; *e.g.*, `strcmp`

Faulty logic *e.g.*, making !($x < 0$) be ($x > 0$) instead of ($x \geq 0$).

Boundary errors —extra test, missing test, coefficients wrong. Bugs breed in boundaries; don't skimp boundary testing.

White box testing

— also known as “glass box”, “structural”.

— derive (more) test cases from source code

— decisions in code may split equivalence partitions

— you must *have* the source code

— read Sommerville about “path testing”

— tends to overlook *missing* code

Glass methods for black boxes

You can use white-box methods for black-box testing, by constructing tests for a *model* of the program.

The model might be a flowchart, a finite state automaton, pseudocode, a formal specification, or a working prototype.

Reducing the size of a test case

Big test cases are hard to debug with.

Use the Delta Debugging algorithm to shrink them.

Original work at http://www.st.cs.uni-saarland.de/dd/

DD.py can be got there.

Open source code at http://delta.tigris.org

A divide-and-conquer algorithm for minimising a change set.

Like "wolf fence" but on *input*, not *code*.

Levels of Coverage

Statement coverage: every statement is executed in some test. This is the minimum acceptable level.

Branch coverage: every branch alternative is executed in some test. Strictly speaking, it's impossible in a language with exception handling like Lisp, Ada, C++, Java. At least cover the normal branches.

Path coverage: every path through the component is executed in some test. Obviously not achievable with loops, but try to cover 0 trips, 1 trip, a few trips, lots of trips.

Beizer books in bibliography belong on your shelf.

tcov lets you check for statement coverage (gcov)

Interface testing 1

Parameters: what parameter properties does your language's type system *not* express/enforce? Is a Pascal Integer the same as a C int?

Shared memory: do all parts agree about format of data? What if it starts at different addresses in different components?

Procedures: what postconditions are assumed but not checked?

Message passing: low level interfaces (UNIX message queues, UNIX and Windows named pipes) make it hard to specify type, grammar, protocol; are they consistent?

Interface testing 2

Version skew client correct according to server version N interface but connected to version N+1 implementation.

Precondition violation: type error, value out of range, arguments in wrong order, object in wrong state (*e.g.*, file not open).

Postcondition failure: server does not provide what client expects.

Side effects server changes things client thought were safe

Timing information out of date, overwritten ($\geq$ `tmpnam(NULL)`), not yet updated

Interface testing 3

Strong type checking eliminates a lot of problems *if* combined with good version control and configuration management.

Consider using the XDR library (for memory or IPC) or 'rpcgen', or ASN.1, or even CORBA to describe interfaces between programs rather than raw/home-brew formats. Better still: UBF contracts.

```
struct person {char *famname, *ownname; int age;} fred;
fread(&fred, 1, sizeof fred, binfile); /* why wrong? */
```

See Also

lecs/lec22/testing.pdf (handout)

lecs/lec22/testplan.htm (handout)

lecs/lec22/uiuc-testing (directory),

JUnit, a nice unit testing framework for Java, at http://www.junit.org

http://www.xprogramming.com/software.htm for testing frameworks for other languages.