6.005 Elements of Software Construction
Fall 2008

6.005 elements of software construction

Testing and Coverage

Rob Miller
Fall 2008

Today's Topics

why testing is important and hard

choosing inputs
- input space partitioning
- boundary testing

how to know if you've done enough
- coverage

testing pragmatics
- stubs, drivers, oracles
- test-first development
- regression testing

WHY TESTING MATTERS

Real Programmers Don't Test!(?)

top 5 reasons not to test

5) I want to get this done fast – testing is going to slow me down.
4) I started programming when I was 2. Don't insult me by testing my perfect code!
3) testing is for incompetent programmers who cannot hack.
2) we're not Harvard students – our code actually works!
1) "Most of the functions in Graph.java, as implemented, are one or two line functions that rely solely upon functions in HashMap or HashSet. I am assuming that these functions work perfectly, and thus there is really no need to test them." – an excerpt from a 6.170 student's e-mail

Who Says Software is Buggy?

Ariane 5 self-destructed 37 seconds after launch

Photographs of the Ariane 5 rocket
removed due to copyright restrictions.

reason: a control software bug that went undetected

- conversion from 64-bit floating point to 16-bit signed integer caused an exception
  - because the value was larger than 32767 (max 16-bit signed integer)
- but the exception handler had been disabled for efficiency reasons
- software crashed ... rocket crashed ... total cost over $1 billion

Another Prominent Software Bug

Mars Polar Lander crashed

Diagrams of the Mars Polar Lander
removed due to copyright restrictions.

- sensor signal falsely indicated that the craft had touched down when it was still 130 feet above the surface.
- the descent engines shut down prematurely... and it was never heard from again

the error was traced to a single bad line of code

- Prof. Nancy Leveson: these problems "are well known as difficult parts of the software-engineering process"... and yet we still can't get them right

The Challenge

we want to

- know when product is stable enough to launch
- deliver product with known failure rate (preferably low)
- offer warranty?

but

- it's very hard to measure or ensure quality in software
- residual defect rate after shipping:
  - 1 - 10 defects/kloc (typical)
  - 0.1 - 1 defects/kloc (high quality: Java libraries?)
  - 0.01 - 0.1 defects/kloc (very best: Praxis, NASA)
- example: 1Mloc with 1 defect/kloc means you missed 1000 bugs!

Testing Strategies That Don't Work

exhaustive testing is infeasible

- space is generally too big to cover exhaustively
- imagine exhaustively testing a 32-bit floating-point multiply operation, a*b
  - there are 2^64 test cases!

statistical testing doesn't work for software

- other engineering disciplines can test small random samples (e.g. 1% of hard drives manufactured) and infer defect rate for whole lot
- many tricks to speed up time (e.g. opening a refrigerator 1000 times in 24 hours instead of 10 years)
- gives known failure rates (e.g. mean lifetime of a hard drive)
- but assumes continuity or uniformity across the space of defects, which is true for physical artifacts
- **this is not true** for software
  - overflow bugs (like Ariane 5) happen abruptly
  - Pentium division bug affected approximately 1 in 9 billion divisions

Two Problems

often confused, but very different

(a) problem of **finding** bugs in defective code

(b) problem of showing **absence** of bugs in good code

approaches

- testing: good for (a), occasionally (b)
- reasoning: good for (a), also (b)

theory and practice

- for both, you need grasp of basic theory
- good engineering judgment essential too

Aims of Testing

what are we trying to do?

- find bugs as cheaply and quickly as possible

reality vs. ideal

- ideally, choose one test case that exposes a bug and run it
- in practice, have to run many test cases that "fail" (because they don't expose any bugs)

in practice, conflicting desiderata

- increase chance of finding bug
- decrease cost of test suite (cost to generate, cost to run)

Practical Strategies

design testing strategy carefully

- know what it's good for (finding egregious bugs) and not good for (security)
- complement with other methods: code review, reasoning, static analysis
- exploit automation (e.g. JUnit) to increase coverage and frequency of testing
- do it early and often

Basic Notions

what's being tested?

- unit testing: individual module (method, class, interface)
- subsystem testing: entire subsystems
- integration, system, acceptance testing: whole system

how are inputs chosen?

- random: surprisingly effective (in defects found per test case), but not much use when most inputs are invalid (e.g. URLs)
- systematic: partitioning large input space into a few representatives
- arbitrary: *not* a good idea, and not the same as random!

how are outputs checked?

- automatic checking is preferable, but sometimes hard (how to check the display of a graphical user interface?)

Basic Notions

how good is the test suite?

- coverage: how much of the specification or code is exercised by tests?

when is testing done?

- test-driven development: tests are written first, before the code
- regression testing: a new test is added for every discovered bug, and tests are run after every change to the code

essential characteristics of tests

- modularity: no dependence of test driver on internals of unit being tested
- automation: must be able to run (and check results) without manual effort

CHOOSING TESTS

Example: Thermostat

specification

- user sets the desired temperature Td
- thermostat measures the ambient temperature Ta
- want heating if desired temp is higher than ambient temp
- want cooling if desired temp is lower than ambient temp

if Td > Ta, turn on heating
if Td < Ta, turn on air-conditioning
if Td = Ta, turn everything off

How Do We Test the Thermostat?

arbitrary testing is not convincing

- "just try it and see if it works" won't fly

exhaustive testing is not feasible

- would require millions of runs to test all possible (Td,Ta) pairs

key problem: choosing a test suite systematically

- small enough to run quickly
- large enough to validate the program convincingly

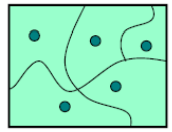
Key Idea: Partition the Input Space

input space is very large, but program is small

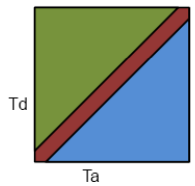
- so behavior must be the "same" for whole sets of inputs

ideal test suite

- identify sets of inputs with the same behavior
- try one input from each set



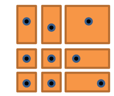
if Td > Ta, turn on heating
if Td < Ta, turn on air-conditioning
if Td = Ta, turn everything off



More Examples

java.math.BigInteger.multiply(BigInteger val)

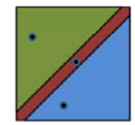
- has two arguments, this and val, drawn from BigInteger
- partition BigInteger into:
  - BigNeg, SmallNeg, -1, 0, 1, SmallPos, BigPos
- pick a value from each class
  - -265, -9 -1, 0, 1, 9, 265
- test the 7 × 7 = 49 combinations



approach 1: partition inputs separately, then form all combinations

static int java.Math.max(int a, int b)

- partition into:
  - a < b, a = b, a > b
- pick value from each class
  - (1, 2), (1, 1), (2, 1)



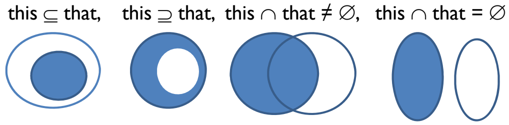
approach 2: partition the whole input space (useful when relationship between inputs matters)

More Examples

Set.intersect(Set that)

- partition Set into:
  - ∅, singleton, many
- partition whole input space into:
  - this = that, this ⊆ that, this ⊇ that, this ∩ that ≠ ∅, this ∩ that = ∅

use both approaches



- pick values that cover both partitions
  - {},{} {},{2} {},{2,3,4}
  - {5},{} {5},{2} {4},{2,3,4}
  - {2,3},{} {2,3},{2} {1,2},{2,3}

Boundary Testing

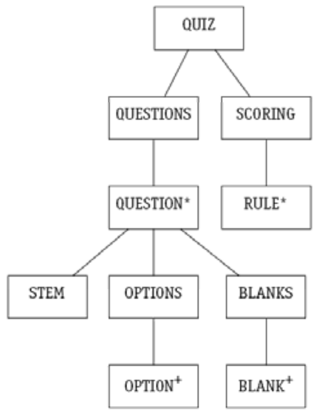
- include classes at **boundaries** of the input space
  - zero, min/max values, empty set, empty string, null
- why? because bugs often occur at boundaries
  - off-by-one errors
  - forget to handle empty container
  - overflow errors in arithmetic

Exercise

recall our quiz grammar

- partition the input space of quizzes
- devise a set of test quizzes

Option ::= Value? Text
Value ::= [digit+]
Text ::= char*
Rule ::= Range Message
Range ::= digit+ - digit+ :
Message ::= char*



- what important class of inputs are we leaving out?

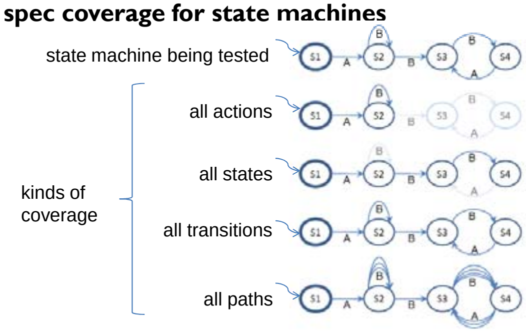
COVERAGE

Coverage

how good are my tests?

- measure extent to which tests 'cover' the spec or code

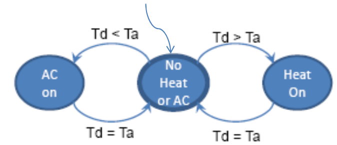
spec coverage for state machines



- all-actions, all-states ≤ all-transitions ≤ all-paths

State Diagram for Thermostat

if Td > Ta, turn on heating
if Td < Ta, turn on air-conditioning
if Td = Ta, turn everything off



- a test case is a trace of (Td,Ta) pairs
- all actions: (Td<Ta), (Td=Ta), (Td>Ta)
  - e.g., using actual temperatures: (67, 70), (67, 67), (70, 67)
- all states: the same trace would cover all states
- all transitions: (Td<Ta), (Td=Ta), (Td > Ta), (Td=Ta)
  - e.g. (67, 70), (67, 67), (70, 67), (70, 70)

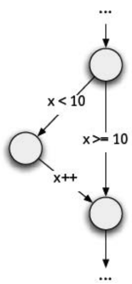
Code Coverage

view control flow graph as state machine

- and then apply state machine coverage notions

example

if (x < 10) x++;



state machine coverage notion	code coverage notion
all-states	all-statements
all-transitions	all-branches
all-paths	all-paths

How Far Should You Go?

for spec coverage

- all-actions: essential
- all-states, all-transitions: if possible
- all-paths: generally infeasible, even if finite

for code coverage

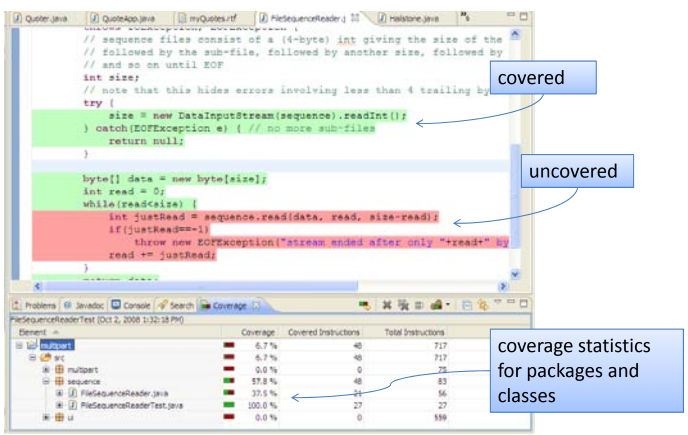
- all-statements, all-branches: if possible
- all-paths: infeasible

industry practice

- all-statements is common goal, rarely achieved (due to unreachable code)
- safety critical industry has more arduous criteria (eg, "MCDC", modified decision/condition coverage)

A Typical Statement Coverage Tool

- EclEmma Eclipse plugin



Black Box vs. Glass Box Testing

black box testing

- choosing test data only from spec, without looking at implementation

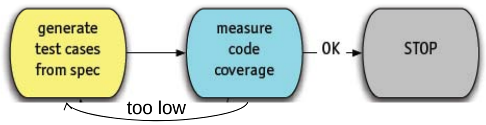
glass box (white box) testing

- choosing test data with knowledge of implementation
  - e.g. if implementation does caching, then should test repeated inputs
  - if implementation selects different algorithms depending on the input, should choose inputs that exercise all the algorithms
- must take care that tests don't *depend* on implementation details
  - e.g. if spec says "throws Exception if the input is poorly formatted", your test shouldn't check specifically for a NullPtrException just because that's what the current implementation does
- good tests should be **modular** -- depending only on the spec, not on the implementation

Black Box vs. Glass Box Testing

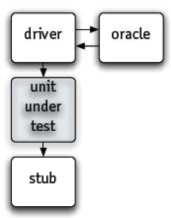
best practice

- generate black-box test cases until code coverage is sufficient



PRAGMATICS

Testing Framework



driver

- just runs the tests
- must design unit to be drivable!
- eg: program with GUI should have API

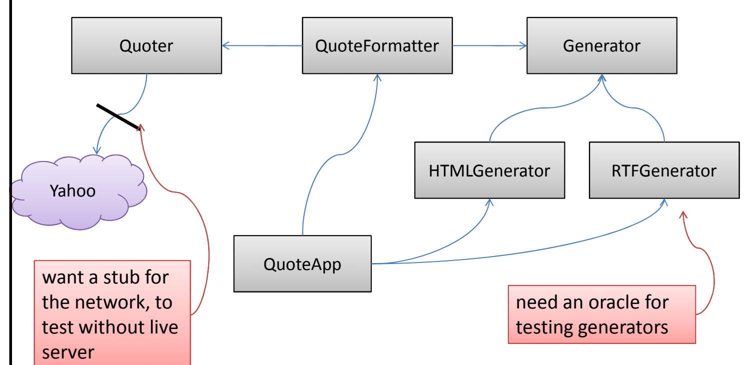
stub

- replaces other system components
- allows reproducible behaviours (esp. failures)

oracle

- determines if result meets spec
- preferably automatic and fast
- varieties: computable predicate (e.g. is the result odd?), comparison with literal (e.g. must be 5), manual examination (by a human)
- in regression testing, can use previous results as "gold standard"

Example: the Quote Generator



Test-First Development

write tests before coding

- specifically, for every method or class:
  1) write specification
  2) write test cases that cover the spec
  3) implement the method or class
  4) once the tests pass (and code coverage is sufficient), you're done

writing tests first is a good way to understand the spec

- think about partitioning and boundary cases
- if the spec is confusing, write more tests
- spec can be buggy too
  - incorrect, incomplete, ambiguous, missing corner cases
  - trying to write tests can uncover these problems

Regression Testing

whenever you find and fix a bug

- store the input that elicited the bug
- store the correct output
- add it to your test suite

why regression tests help

- helps to populate test suite with good test cases
  - remember that a test is good if it elicits a bug – and every regression test did in one version of your code
- protects against reversions that reintroduce bug
- the bug may be an easy error to make (since it happened once already)

test-first debugging

- when a bug arises, immediately write a test case for it that elicits it
- once you find and fix the bug, the test case will pass, and you'll be done

Summary

testing matters

- you need to convince others that your code works
- testing generally can't prove absence of bugs, but can increase quality by reducing bugs

test early and often

- unit testing catches bugs before they have a chance to hide
- automate the process so you can run it frequently
- regression testing will save time in the long run

be systematic

- use input partitioning, boundary testing, and coverage
- regard testing as a creative design problem

use tools and build your own

- automated testing frameworks (JUnit) and coverage tools (EclEmma)
- design modules to be driven, and use stubs for repeatable behavior