[kwol-i-tee] • 1250–1300; ME qualite < OF < L quālitās
[uh-shoor-uhns, -shur-] • 1325–75; ME ass(e)ura(u)nce < MF ass(e)urance
dictionary.com

CSE403: Software Engineering

David Notkin
Winter 2009

Upcoming lecture plan

MONDAY	WEDNESDAY	FRIDAY
	Testing	Design reviews, Midterm review
Presidents' Day	Midterm	SDS presentations (2)
SDS presentations (2)	Guest lecture TBA	

Terminology

- A *failure* occurs when a program doesn't satisfy its specification
- A *fault* occurs when a program's internal state is inconsistent with what is expected (usually an informal notion)
- A *defect* is the code that leads to a fault (and perhaps to a failure)
- An *error* is the mistake the programmer made in creating the defect

More terminology

- A test case is a specific set of data that exercises the program
- A test suite is a set of test cases
- Old terminology
  - A test case (suite) fails if it demonstrates a problem
- New terminology
  - A test case (suite) succeeds if it demonstrates a problem

Root cause analysis

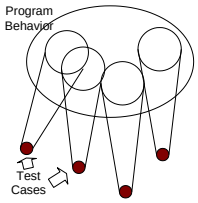
- Tries to track a failure to an error
- Identifying errors is important because it can
  - help identify and remove other related defects
  - help a programmer (and perhaps a team) avoid making the same or a similar error again

Discreteness

- It's important to remember that testing software is different from testing physical widgets
  - In general, physical widgets can be analyzed in terms of continuous mathematics
  - Software is based on discrete mathematics
- Why does this matter?
- In continuous math, a small change in an input corresponds to a small change in the output
  - This allows safety factors to be built in
- In discrete math, a small change in an input can correspond to a huge change in the output

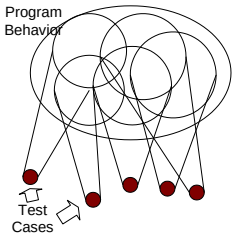
Characteristic tests

- A goal of picking a test case is that it be characteristic of a class of other tests
- That is, one case builds confidence in how other cases will perform



More characteristic tests

- The overall objective is to cover as much of the behavior space as possible
  - It's generally infinite
- In general, it's useful to distinguish the notions of common vs. unusual cases for testing



Black box testing

- Treat the unit (program, procedure, etc.) as a black box
  - You can hypothesize about the way it is built, but you can't see it
- Depend on a specification, formal or informal, for determining whether it behaves properly
- How to pick cases that cover the space of behaviors for the unit?
  - Use heuristics

Equivalence partitioning

- Based on input conditions
  - If input conditions are specified as an ordered range, you have one valid class (in the range) and two invalid classes (outside the range on each side)
  - If specified as a set, then you can be valid (in the set) or invalid (outside the set)
  - Etc.

Boundary values

- Problems tend to arise on the boundaries of input domains than in the middle
- So, extending equivalence partitioning, make sure to pick added test cases that exercise inputs near the boundaries of valid and invalid ranges

Others include

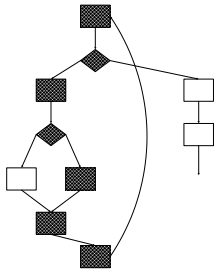
- Cause-effect graphing
- Data validation testing
- Syntax-direct testing
- …

White box testing

- In this approach, the tester has access to the actual software
- They needn't guess at the structure of the code, since they can see it
- In this approach, the focus often shifts from how the code behaves to what parts of the code are exercised

White box coverage

- In black box, the tests are usually intended to cover the space of behavior
- In white box, the tests are usually intended to cover the space of parts of the program



Statement coverage

- One approach is to cover all statements
  - Develop a test suite that exercises all of a program's statements
- What's a statement?
  - `max = (x > y) ? x : b;`

```
if x > y then
    max := x
else
    max :=y
endif
```

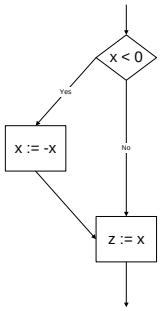

Weakness

- Coverage may miss some obvious issues
- In this example (due to Ghezzi et al.) a single test (any negative number for x) covers all statements
  - But it's not satisfying with respect to input condition coverage, for example

```
if x < 0 then
    x := -x;
endif;
z := x;
```


Edge coverage

- Another approach is to use a control flow graph (CFG) representation of a program
  - Essentially, a flowchart
- Then ensure that the suite covers all edges in the CFG



Condition coverage

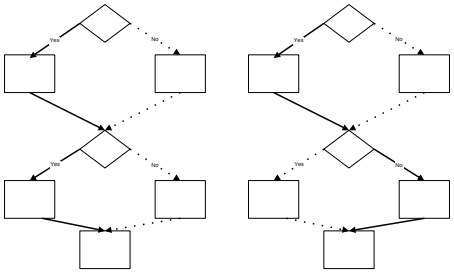
- Complex conditions can confound edge coverage
  - `if (p != NULL) and (p->left < p->right)` …
- Is this a single conditional statement in the CFG?
- How are short-circuit conditionals handled?
  - `andthen`, `orelse`

Path coverage

- Edge coverage is in some sense very static
- Edges can be covered without covering paths (sequences of edges)
  - These better model the actual execution

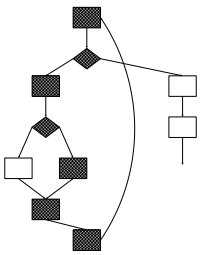
```
if x <> 0 then
   y := 5
else
   z := z-x
endif
if z > 1 then
   z := z/x
else
   z := 0
endif
```


Example



Path coverage and loops

- In general, we can't bound the number of times a loop executes
- So there are an unbounded number of paths in general



Testing

- It's unsound
- It's heuristic
  - Heuristic doesn't mean undisciplined
- It's extremely useful and important
- Good testing requires a special mindset
  - "I'm going to make that sucker fail!"
- Good coding requires a special mindset
  - "Nobody's going to break *my* code!"

Questions?