Verification and Validation

CSE 403, Spring 2004

Software Engineering

http://www.cs.washington.edu/education/courses/403/04sp/

Readings and References

- References
  - » *If You Didn't Test It, It Doesn't Work*, Bob Colwell, IEEE Computer
    - http://www.computer.org/computer/homepage/0502/Random/
- Acknowledgment
  - » much of the content of this lecture is derived from a similar lecture by G. Kimura in an earlier instance of CSE 403

Verification and Validation

- Verification: “Did we build the system right?”
  - » Design and Implementation verification
  - » Does the system do specific tasks correctly?
  - » Developer / Tester has the knowledge
- Validation: “Did we build the right system?”
  - » Requirements validation
  - » Does the system do the required set of tasks?
  - » Customer / Integrator has the knowledge

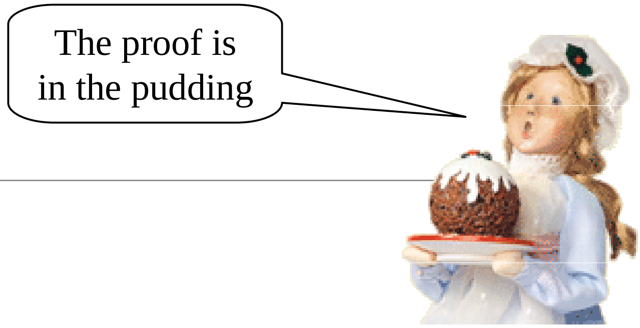
Some Approaches to Verification

- Process
  - » Improving the likelihood that code is correct
- Testing
  - » A dynamic approach
- Proof of correctness
  - » Use formal analysis to show an equivalence between a specification and a program

Process

- Process includes a broad set of ideas and approaches
  - » Software inspections, walkthroughs, reviews
  - » Capability maturity model, ISO 9000
  - » etc
- Software correctness depends on thousands and thousands of details being correct
  - » Good processes help you avoid making mistakes
  - » Processes are not magic

Testing vs. Proving



- Dynamic Testing
  - » Builds confidence (not certainty)
    - Can only show the presence of bugs, not their absence
  - » Used widely in practice
  - » Costly
- Static Proving
  - » Proofs are human processes - mistakes are possible!
  - » Applicability is limited in practice
  - » Extremely costly

Engineering: intelligent compromise

- Dynamic techniques are unattractive because they are unsound
  - » you can believe something is true when it's not
- Static techniques are unattractive because they are often very costly
  - » and can overlook fundamental problems
- The truth is that they should be considered to be complementary, not competitive

Testing

- Testing is by far the dominant approach to demonstrating that code does what it supposed to (whatever that means!)
- Testing is a lot like the weather
  - » everybody complains about it
  - » but nobody seems to do much about it
- However, unlike the weather, you can actually do something about it!

Terminology

- An *error*
  - » mistake the programmer made in design or implementation
- leads to a *defect*
  - » inappropriate code
- that leads to a *fault*
  - » when a program's internal state is inconsistent with what is expected
- that causes a *failure*.
  - » when the program doesn't satisfy its specification

Root cause analysis

- Track a failure back to an error
  - » Failures are precious information because an error has finally become visible
- Identifying errors is important because it can
  - » help identify and remove other related defects
    - other defects might not cause visible failures yet
  - » help a programmer (and perhaps a team) avoid making the same or a similar error again
    - If an error is made once, it is very likely made twice

Discreteness

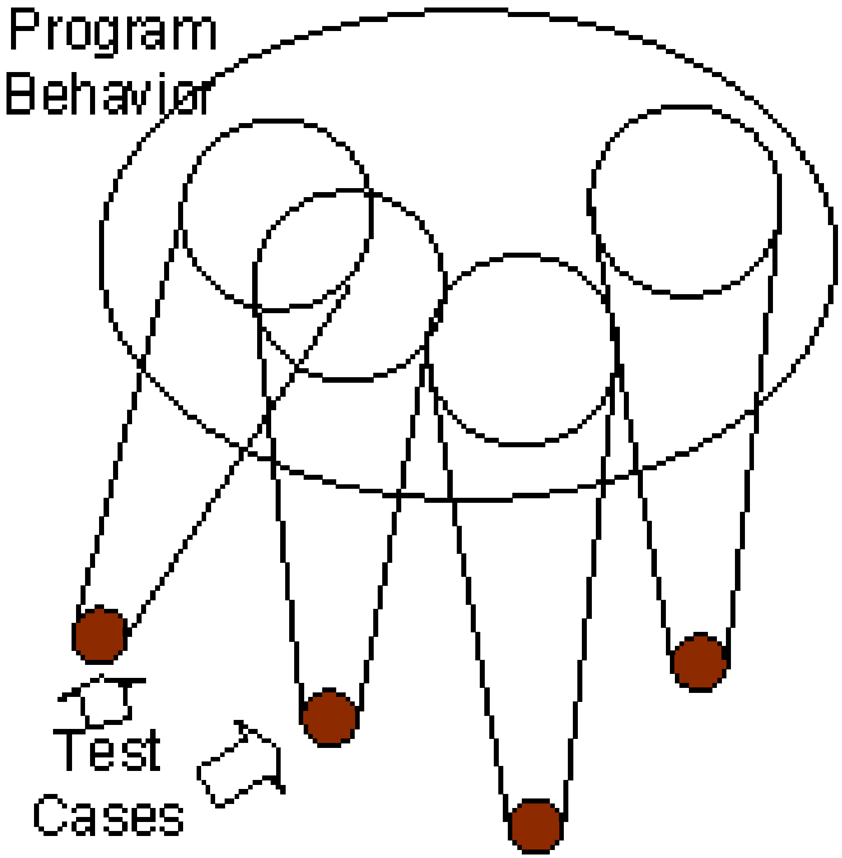
- Testing software is different from testing widgets
  - » In general, physical widgets can be analyzed in terms of continuous mathematics
  - » Software is based on discrete mathematics
- Why does this matter?
- In continuous math, a small change in an input corresponds to a small change in the output
  - » 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

Kinds of testing

- Unit
- White-box
- Black-box
- Gray-box
- Bottom-up
- Top-down
- Boundary condition
- Syntax-driven
- Big bang
- Integration
- Acceptance
- Stress
- Regression
- Alpha
- Beta
- etc

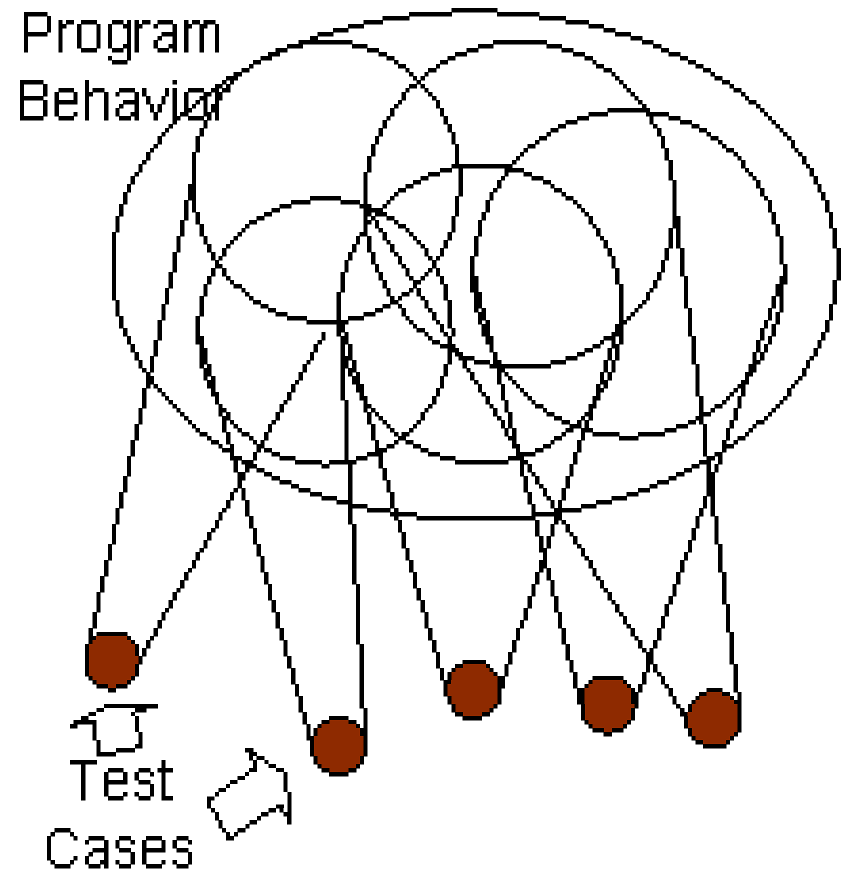
Picking Test Cases

- 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



Cover the behavior space

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



Black box testing

- Treat the unit under test as a black box
  - » You can hypothesize about the way it is built, but you can't see inside 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?
  - » equivalence partitioning, boundary values, etc
  - » independent testers

Equivalence partitioning

- Based on input conditions
  - » If input conditions are specified as a 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

Off-the-wall testing

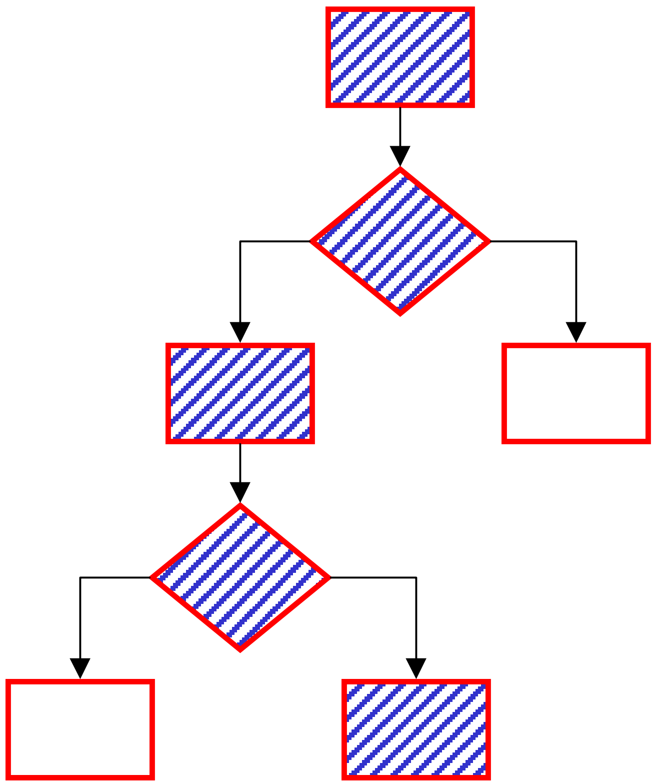
- Real life and real people are not interested in what you thought the specification said
  - » Life takes strange turns
  - » Users are not focused on treating your program with kid gloves
- When your program is released in the wild, it *will* get knocked around
  - » welcome the comments of the tester who pushes your program to its limits, don’t shout them down

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
  - » The focus tends to shift from how the system behaves to what parts of the code are exercised
    - this can be very useful, and very misleading
- The tester's challenge: Can you find a *defect* that leads to a *fault* that causes a *failure*?

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;
```

More Coverage

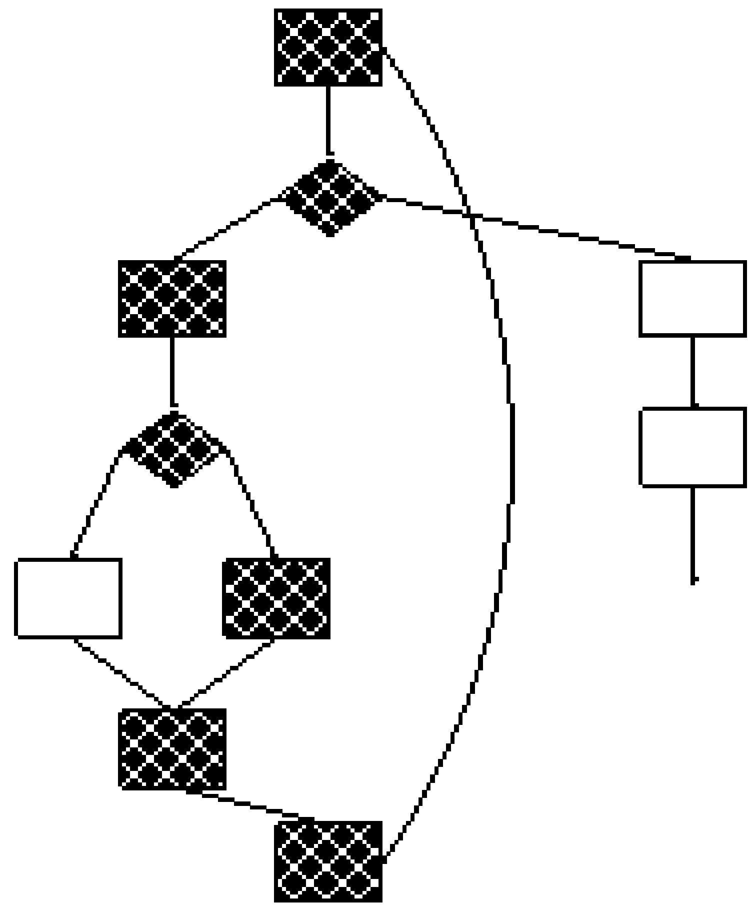
- Edge coverage
  - » Use control flow graph (CFG) representation of a program
  - » Ensure that the suite covers all edges in the CFG
- Condition coverage
  - » Complex conditions can confound edge coverage

```
if ((p != NULL) && (p->left < p->right)) …
```

    - Is this a single conditional statement in the CFG?
    - How are short-circuit conditionals handled?
- Path coverage
  - » Edge coverage is in some sense very static
  - » Edges can be covered without covering paths (sequences of edges)
  - » Paths are better models of the actual execution

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
  - » We resort to heuristics like those from black box testing to exercise these loops



Some more practical aspects

- Who tests the tests, especially a large complicated test?
  - » If your test program generates random data, who confirms the results?
  - » Another example is testing trig functions.
- Testing the error cases can be a wider set of inputs. You have two problems
  - » Making sure you have proper test coverage and
  - » Making sure the results are correct.
- Fault injection is another way of testing systems.
  - » For example, injecting I/O failures in a disk controller can test the error cases for the disk driver and file system.
  - » Another example is injecting memory allocation errors, to see how programs behave when they run out of memory.

Final note on testing

- It’s unsound and based on heuristics
- It’s extremely useful and important

- Good testing requires a special mindset
  - » “I’m going to find a way to make that system fail!”
  - » “My test case is a success - it found a system problem.”
- Good coding requires a special mindset
  - » “Nobody’s going to break *my* code!”
  - » “Good thing we found the failure now, not in real life.”