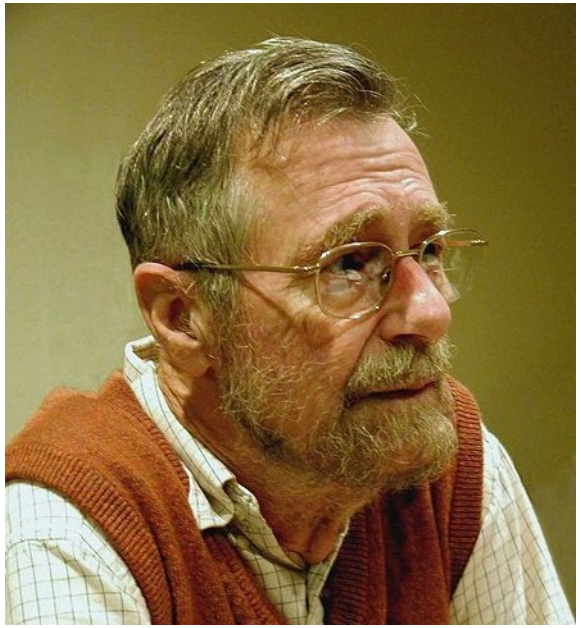
Software Testing (1)

September 22, 2022

Byung-Gon Chun

(Slide credits: George Candea, EPFL and Armando Fox, UCB)

Goals of Testing



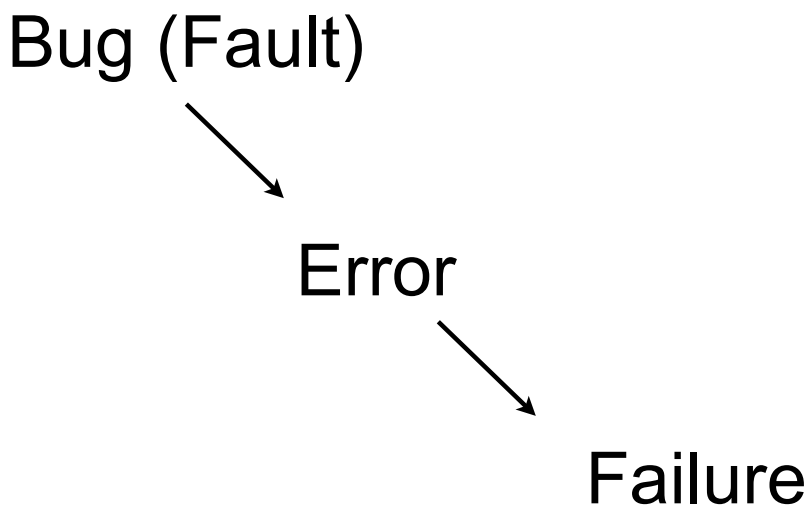
"Program testing can be used to show the presence of bugs, but never to show their absence."

— Edsger Dijkstra

- Cannot prove correctness
  - *Testing = exec a program in order to find bugs*
  - *Ideally, the found bugs are easy to reproduce*

- *Risk management*
  - *Use testing to gain some level of confidence*
  - *Write tests to catch bugs as early as we can*

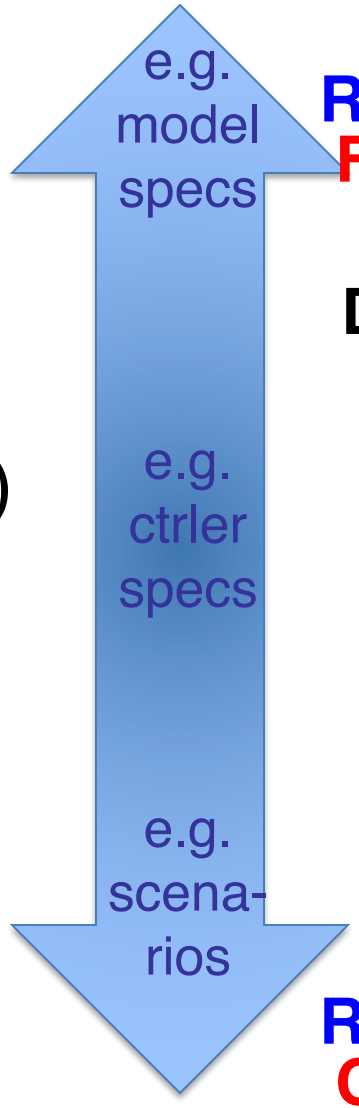
The Risk of Not Testing



- Industry-average bug density
  - 10-100 bugs / KLOC after coding
  - 0.5 – 5 bugs / KLOC not detected before delivery

What kinds of tests?

- Unit (one method/class)
- Functional or module (a few methods/classes)
- Integration/system



Runs fast **High coverage**
Fine resolution
Many mocks;
Doesn't test interfaces

Few mocks;
tests interfaces

Runs slow **Low coverage**
Coarse resolution

Testing Today

- Far more automated
- Tests are self-checking
  - The test code itself can determine if the code being tested works or not
- What are good tests?
  Five principles for creating good tests

Unit tests should be FIRST

- **F**ast
- **I**ndependent
- **R**epeatable
- **S**elf-checking
- **T**imely

Unit tests should be FIRST

- **F**ast: run (subset of) tests quickly (since you'll be running them *all the time*)
- **I**ndependent: no tests depend on others, so can run *any subset* in *any order*
- **R**epeatable: run N times, get same result (to help isolate bugs and enable automation)
- **S**elf-checking: test can *automatically* detect if passed (*no human checking* of output)
- **T**imely: written about the same time as code under test (with TDD, written *first!*)

(Conventional) Unit Testing Tool (Used for unit/functional testing)

- xUnit: a framework to write repeatable tests
  - JUnit
  - NUnit
  - CUnit
  - Ruby Test::Unit
  - Python unittest
  - …
- Ways of creating tests: annotation, inheritance, DSL

xUnit

- Creating tests
  - Annotation: Java, C#, …
  - Inheritance: ruby, python unittest, …

- Automatic checking using assertions
  - Tests that exercise **happy** paths
  - Tests that exercise **sad** paths

Example: JUnit

(Ref. https://github.com/junit-team/junit/wiki/Getting-started)

Calculator.java

```
public class Calculator {
  public int evaluate(String expression) {
    int sum = 0;
    for (String summand: expression.split("\\+"))
      sum += Integer.valueOf(summand);
    return sum;
  }
}
```

CalculatorTest.java

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class CalculatorTest {
  @Test
  public void evaluateExpression() {
    Calculator calculator = new Calculator();
    int sum = calculator.evaluate("1+2+3");
    assertEquals(6, sum);
  }
}
```

Example: A Successful Test

```
java -cp .:junit-4.XX.jar:hamcrest-core-1.3.jar org.junit.runner.JUnitCore CalculatorTest
```

```
JUnit version 4.12
.
Time: 0,006

OK (1 test)
```

You will use a software tool, which simplifies running tests.
E.g., make, maven, rake, npm test, coverage

Example: A Failing Test

- Replace the line `sum += Integer.valueOf(summand);`
  with `sum -= Integer.valueOf(summand);`

```
java -cp .:junit-4.XX.jar:hamcrest-core-1.3.jar org.junit.runner.JUnitCore CalculatorTest

JUnit version 4.12
.E
Time: 0,007
There was 1 failure:
1) evaluatesExpression(CalculatorTest)
java.lang.AssertionError: expected:<6> but was:<-6>
  at org.junit.Assert.fail(Assert.java:88)
  ...

FAILURES!!!
Tests run: 1,  Failures: 1
```


Which test failed

What went wrong

Example. Python Unit Test

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main()
```

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s OK
```

Unit Tests

- Examine a single class
- Require just the source code of the application
- Are fast
- Are not affected by external systems, e.g. web services or databases
- Perform little or no I/O, e.g. no real database connections.

Unit Tests

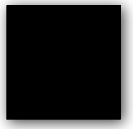
- A test that writes to a database or reads JSON from a web service is **NOT** a unit test.
- It can become a unit test if you ***mock*** that external web service.
- Unit tests and integration tests should be handled differently.

Testing techniques

Types of Testing

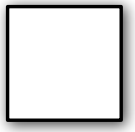
Testing

Black Box



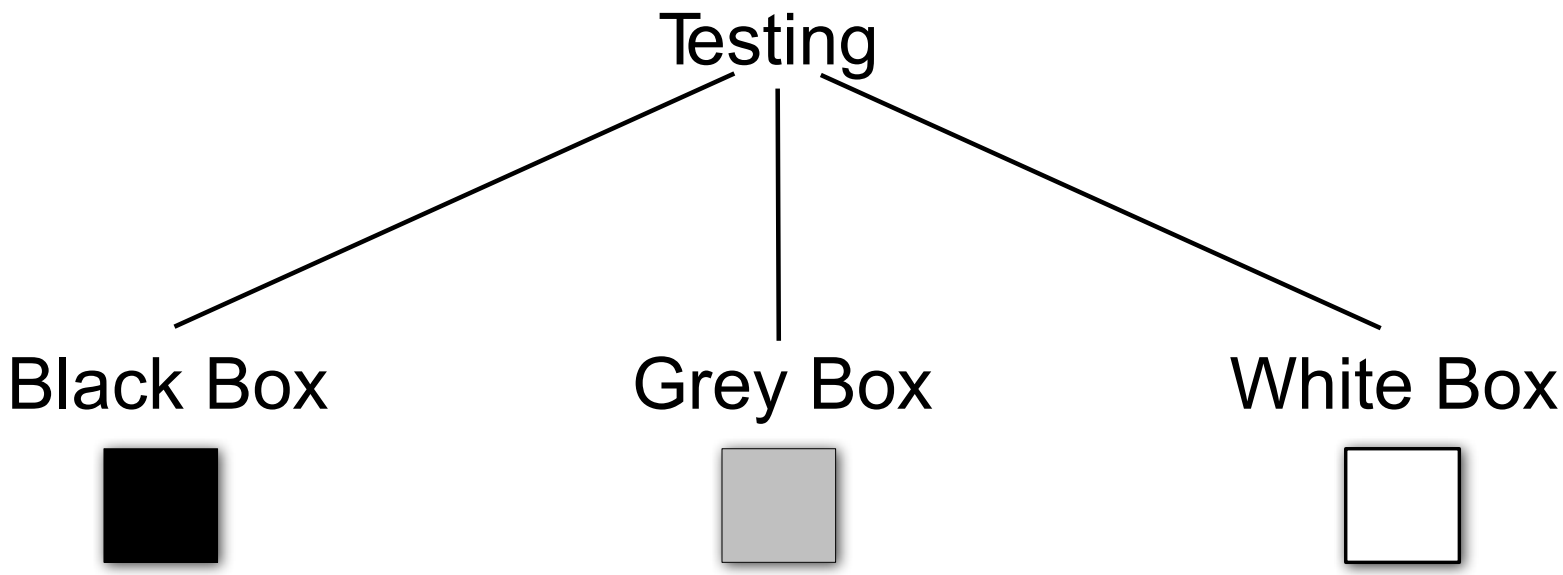
Test functionality

White Box

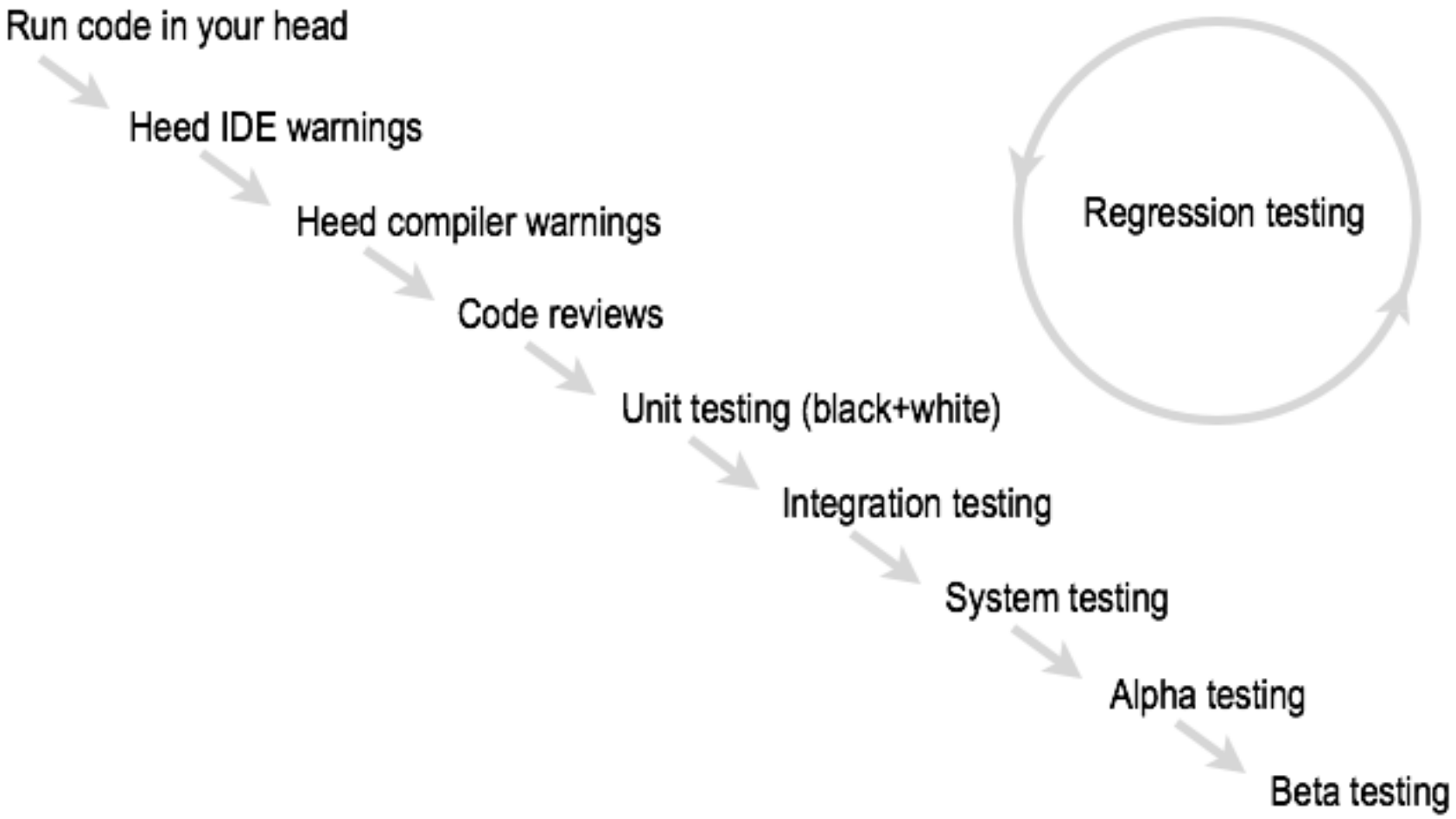


Test internal structures

Types of Testing



Traditional Quality Assurance



Boundaries & Equivalence Classes

- Equivalence classes
  - One check -> two classes
    - E.g., “for input to be valid, it must be < 10”
    - Equivalence class #1: x < 10
    - Equivalence class #2: x >= 10
  - Two checks => three classes
    - E.g., valid x is in range (0, 100)
    - Equivalence class #1: 0 < x < 100
    - Equivalence class #2: x <= 0
    - Equivalence class #3: x >= 100
  - **Test at least one value in each class**
- Boundary testing
  - Most programs fail at **input boundaries**
  - If valid input in [min…max], test with
    - x = min and x = max
    - x < min and x > max
  - Same for boundaries of data structures (e.g., arrays)

How much testing is enough?

- Bad: “Until time to ship”
- A bit better: (Lines of test) / (Lines of code)
  - 1.2–1.5 not unreasonable
  - often *much higher* for production systems
- Better question: “How thorough is my testing?”
  - Formal methods
  - Coverage measurement
  - We focus on the latter, though the former is gaining steady traction

Metrics

- **X is covered** if it is executed at least once by at least one test
- Coverage = % covered of total available

- Tension between quality vs. cost

- Popular metric = coverage
  - X = methods -> method/function coverage
  - X = statements -> statement/line/basic-block coverage
  - X = branches -> branch coverage
  - X = paths -> path coverage

Measuring Coverage—Basics

```
class MyClass
 def foo(x,y,z)
  if x
    if (y && z) then bar(0) end
   else
    bar(1)
   end
 end
 def bar(x) ; @w = x ; end
end
```

- S0: every method called
- S1: every method *from every call site*
- C0: every statement
- C1: every branch in both directions
- C1+decision coverage: every *subexpression* in conditional
- C2: every path (difficult, and disagreement on how valuable)

Coverage Example

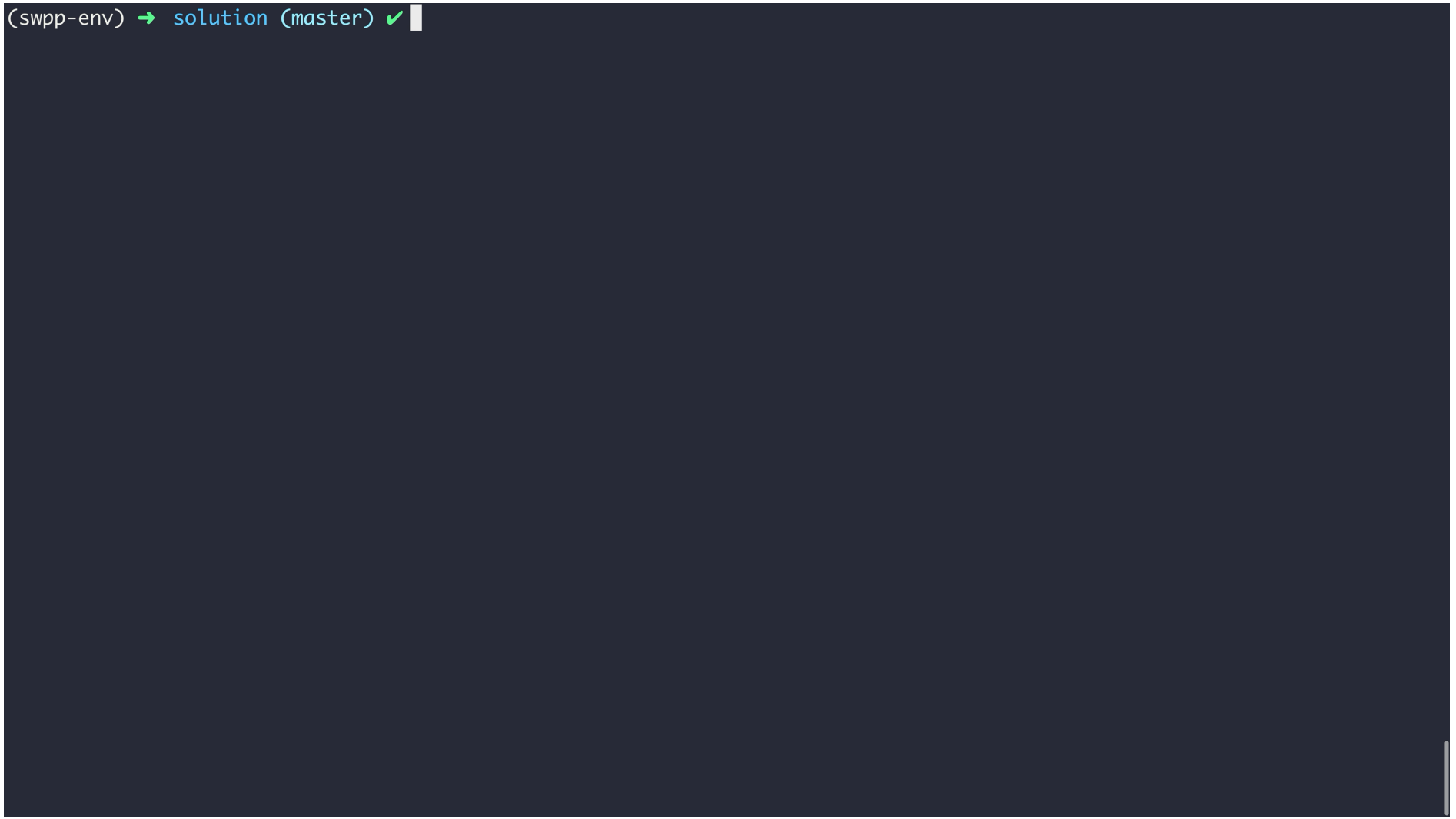
```
static final int MAXIMUM_CAPACITY = 1 << 30;
static final float DEFAULT_LOAD_FACTOR = 0.75f;
// ...
public HashMap(int initialCapacity, float loadFactor)
{
  if (initialCapacity < 0)
    throw new IllegalArgumentException("Illegal initial capacity: " +
                       initialCapacity);
  if (initialCapacity > MAXIMUM_CAPACITY)
    initialCapacity = MAXIMUM_CAPACITY;
  if (loadFactor <= 0 || Float.isNaN(loadFactor))
    throw new IllegalArgumentException("Illegal load factor: " +
                       loadFactor);
  int capacity = 1;
  while (capacity < initialCapacity)
    capacity <<= 1;

  this.loadFactor = loadFactor;
  threshold = (int)(capacity * loadFactor);
  table = new Entry[capacity];
  init();
```

- Test input
  x = new HashMap(64, 0.75);
  x = new HashMap(-1, 0.75);
  x = new HashMap(1+MAXIMUM_CAPACITY, -1);

- 100% statement coverage
  4.6% path coverage

Python Coverage Example



Identifying What's Wrong in Your Code

- How can you tell when code is less than beautiful, and how do you improve it?

- We can identify problems in two ways:
  - Quantitatively using *software metrics*
  - Qualitatively using *code smells*

Quantitative: Metrics

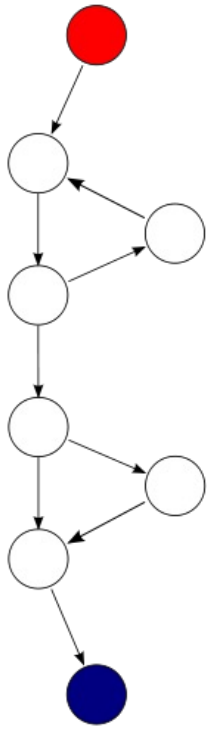
Metric	Target score
Code-to-test ratio	≤ 1:2
C0 (statement) coverage	90%+
Assignment-Branch-Condition score (ABC score)	< 20 per method
Cyclomatic complexity	< 10 per method (NIST)

- "Hotspots": places where *multiple metrics* raise red flags
- Take metrics with a grain of salt
  - Better for *identifying where improvement is needed* than for *signing off*

Cyclomatic complexity (McCabe, 1976)

- Cyclomatic complexity =
  # of linearly-independent* paths through code =
  E– N+2P (edges, nodes, connected components)
  * each path has at least one edge that is not in one of the other paths

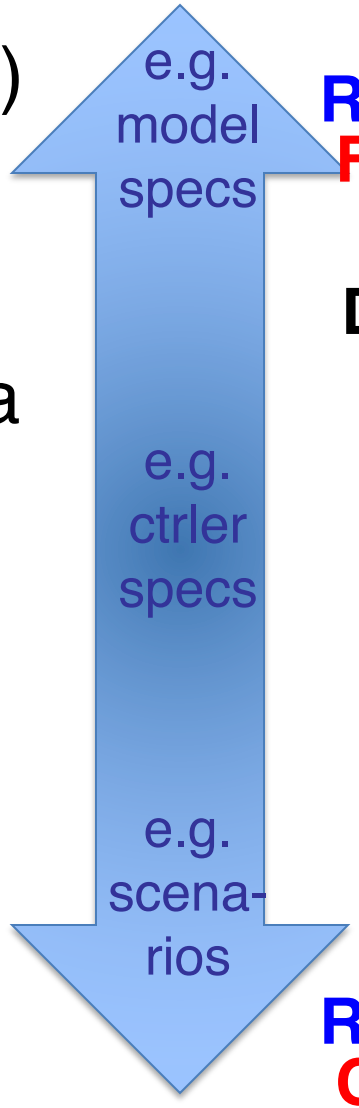
```
def mymeth
  while(...)
    ....
  end
  if (...)
    do_something
  end
end
```



- Here, E=9, N=8, P=1, so CC=3
- NIST (Natl. Inst. Stds. & Tech.): ≤10 /module

What kinds of tests?

- Unit (one method/class)
- Functional or module (a few methods/classes)
- Integration/system



Runs fast **High coverage**
Fine resolution
Many mocks;
Doesn't test interfaces

Few mocks;
tests interfaces

Runs slow **Low coverage**
Coarse resolution

Going to extremes

- ✗ “I kicked the tires, it works”
- ✗ “Don’t ship until 100% covered & green”
- ☑ use coverage to identify untested or undertested parts of code
- ✗ “Focus on unit tests, they’re more thorough”
- ✗ “Focus on integration tests, they’re more realistic”
- ☑ each finds bugs the other misses