CompSci 230
Software Design and Construction

Software Quality 2015S1

White box testing

Lecture plan

Week 1: *No class - Anzac Day*
What is software quality?
Some key developer practices (version control, testing).

Week 2: Black box testing.
White-box testing.
Myers' testing principles.

Week 3: Traditional approach to testing (Waterfall).
Agile approach to testing (XP).
Famous failures.

Myers Ch. 2, pp. 10-12

Learning Goals for Today

- Develop a test suite for some code after looking at it. (White-box testing.)
  - What are **your** initial questions? (Have you written some already?)
  - Should I analyse the specification carefully, as in black-box testing, to discover "what the code should be doing"?
  - Does white-box testing have an advantage, over black-box testing, in testing "what the program is not intended to do"?
  - Should I carefully test interfaces, exceptions, or error returns, or should I concentrate on confirming the correctness of functional "internal" behaviour?

- Evaluate strengths and weaknesses of Black-box and White-box testing.

White-Box Testing

- "This strategy derives test data from an examination of the program's logic
  - "(and often, unfortunately, at the neglect of the specification)."
  - Hmmm… Myers is **not** encouraging white-box testers to "neglect" the specification.
    - He is pointing out that many white-box testers do not pay careful attention to the specification.
- Aim is to test what has been implemented

White-Box Testing

- What is the overall strategy or "gold standard" for white-box testing?
  - "Causing every statement in the program to execute at least once"?
  - No… this is "highly inadequate". (Can you see why?)

A Gold Standard for White-Box Testing

- **Exhaustive Path Testing:** a gold standard for White-Box Testing
  - "if you execute, via test cases, all possible paths of control flow through the program, then ***possibly*** the program has been completely tested."

- Testing all possible paths of control flow is not enough to prove correctness in a white-box test, so how can this be a gold standard?
  - Recall that testing all possible inputs is not enough to prove correctness in a black-box text, yet this is still the gold standard for black-box testing.
  - The justification is the other way around! If you ***don't*** exercise all control paths, then a test case for this path may reveal an obvious bug in this path. Someone might ask: why didn't you test that path? How would you respond.

A Gold Standard for White-Box Testing

- **Exhaustive Path Testing:**

- Testing all paths is often (almost always?) infeasible.
  - Recall that the gold-standard for black-box testing (of testing all possible inputs) is almost always infeasible.
  - Even if you know you can't possibly "reach the gold", you can still use this standard to measure your progress!

What is a "Possible Path of Control Flow"?

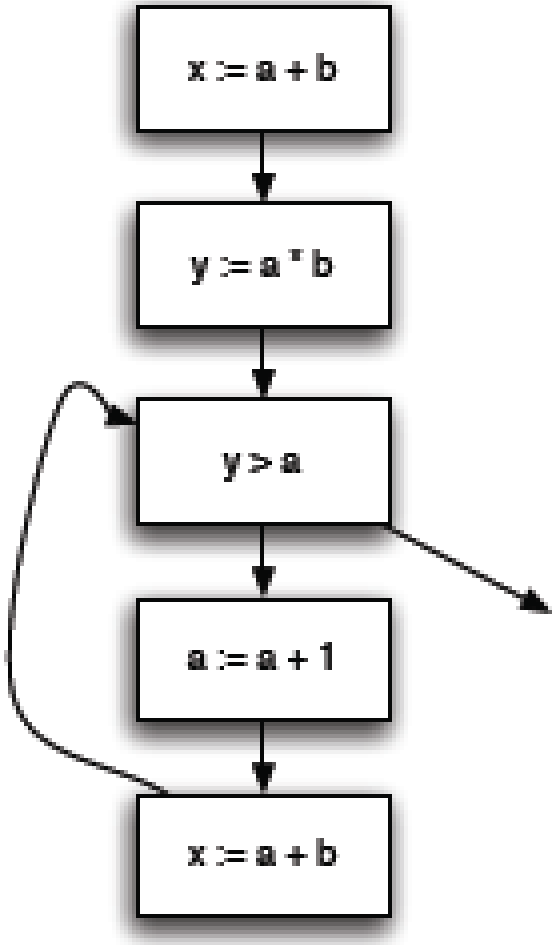
- Myers does not offer a sharp definition in this chapter – you'd have to read Chapter 4 (and even there you won't find one AFIK ;-).
  - In Chapter 2, he discusses the concept of a "unique logic path", where a "logic path" is **the sequence of instructions executed when a program is given a particular input**.
  - If the program has a conditional branch, then it has multiple logic paths (assuming that an input can be found which causes the program to branch, and another input which causes the program not to branch).

- Your goal, as a white-box tester, is to devise an input which will "force" the program to take each important path.
  - We define "path" informally in this course, with reference to a flowchart as on the following slide.

White-Box Testing: An Example

- Devise a test set for this code. (Inputs: a, b; Outputs: a, x, y.)

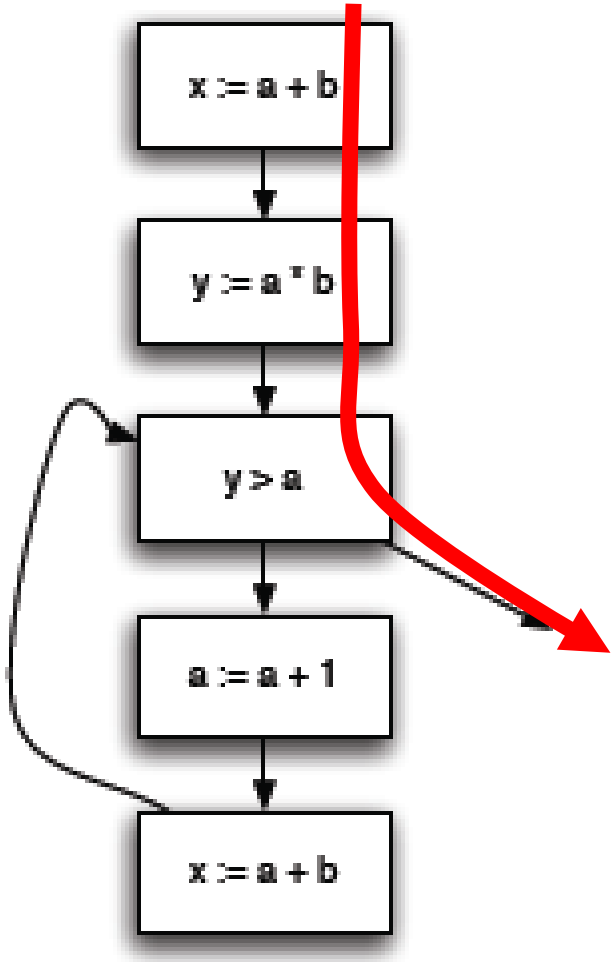
```
x := a + b;
y := a * b;
while (y > a) {
    a := a + 1;
    x := a + b
}
```



White-Box Testing: An Example

- a = 1, b = 1 ? Yes, this is "red" input... output: a = 1, x = 2, y = 1.

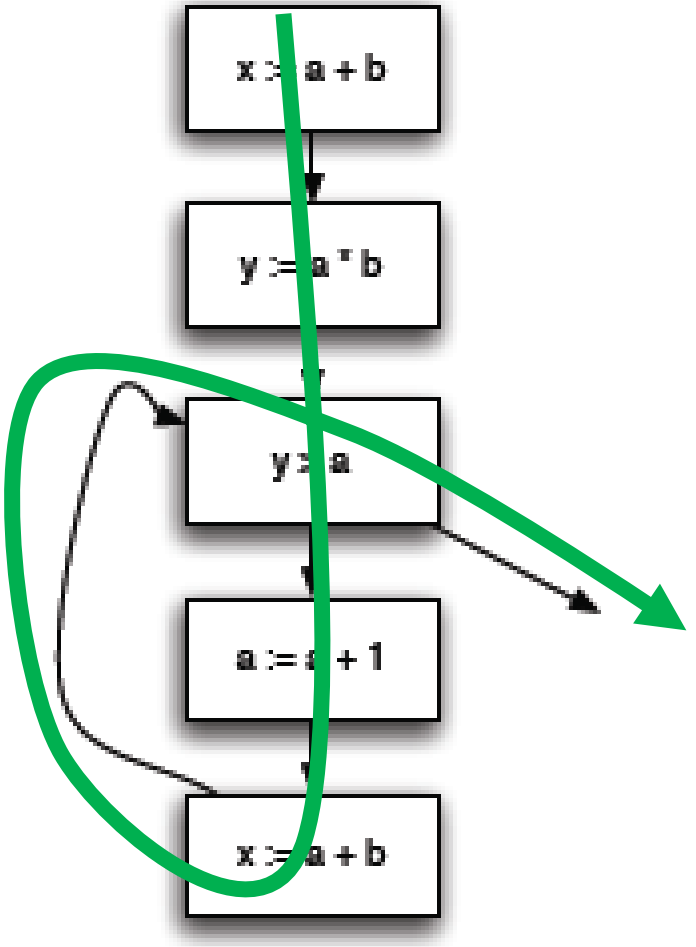
```
x := a + b;
y := a * b;
while (y > a) {
    a := a + 1;
    x := a + b
}
```



White-Box Testing: An Example

- a = 1, b = 2 ? Hmmm... x = 3, **y = 2**; then **a = 2**, **x = 4**.

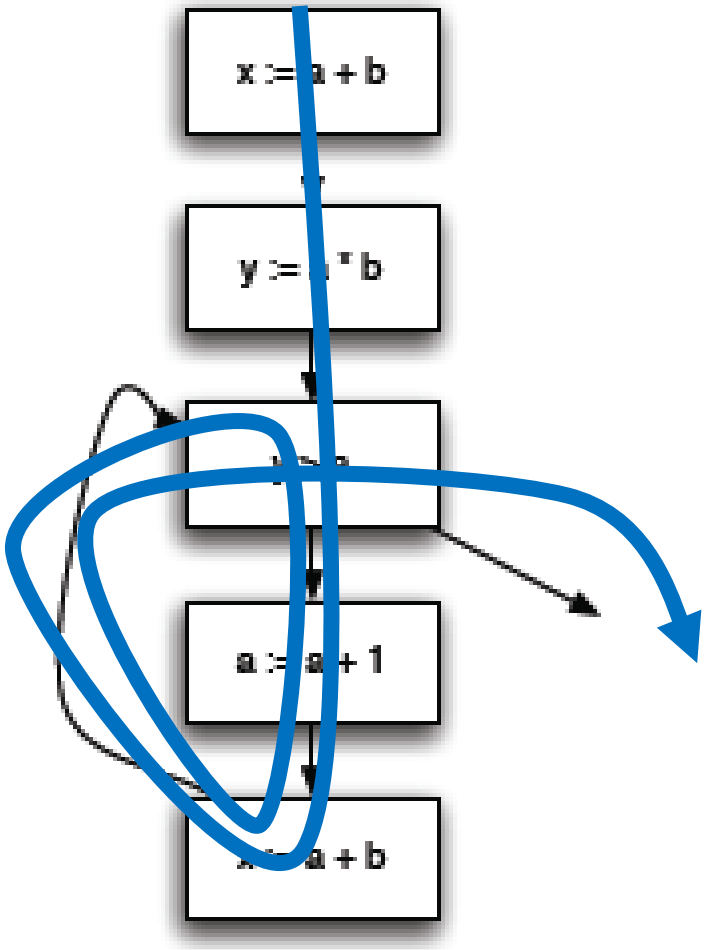
```
x := a + b;
y := a * b;
while (y > a) {
    a := a + 1;
    x := a + b
}
```



White-Box Testing: An Example

- a = 1, b = 3 ? Hmmm... x = 4, **y = 3**; a = 2, x = 7; **a = 3**, **x = 10**.

```
x := a + b;
y := a * b;
while (y > a) {
    a := a + 1;
    x := a + b
}
```



When should we stop?

- Don't add a test unless it has a reasonable chance of exposing a new bug.
  - A loop that "works correctly" on two iterations is not very likely to show an obvious problem on the third iteration, so we might stop after testing the green path.

- A problem: we don't know what this program is supposed to do, so how can we choose a "correct" set of output values for each of our test cases in this example??!
  - This example is an extreme form of white-box testing, in which there is no specification, aside from the code we're looking at.

If you test a path…

- Can you conclude that there are no bugs on a path you have tested? Of course not! … but let's list some reasons why we may miss some bugs, then think about whether we can write test cases to cover these…
  - The output we specify for our test case may be incorrect
    - because we were doing a regression test against a buggy version of our program,
    - because we interpreted the specification carelessly,
    - or because the specification wasn't clear.
  - What can we do to decrease the number of incorrect test cases we write?

If you test a path…

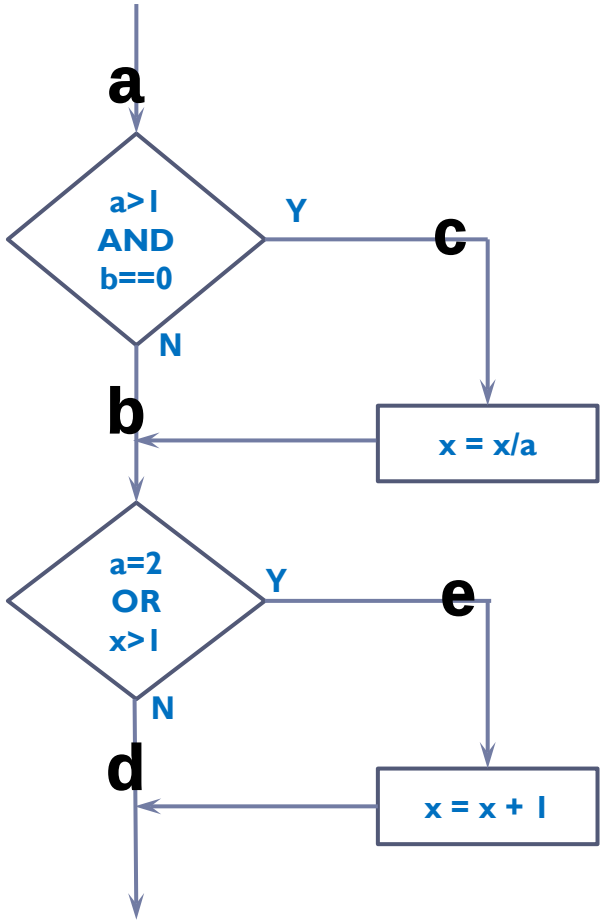
- There may be data-sensitive calculations in the path, and we're only testing a single point of what may be a very non-linear function.
  - If the path contains "x = a+b" but it should contain "x = a*b", our case won't reveal the error if our test input has a=0 or b=0.
  - Should we write more test cases for paths with data-sensitive computations?

- The computation on this path may be non-deterministic.
  - If a program is multi-threaded, then the value computed on one path may depend greatly on what path another thread is following, and on how far along that path the other thread has already moved. (Do **you** understand multi-threading?)

Coverage techniques

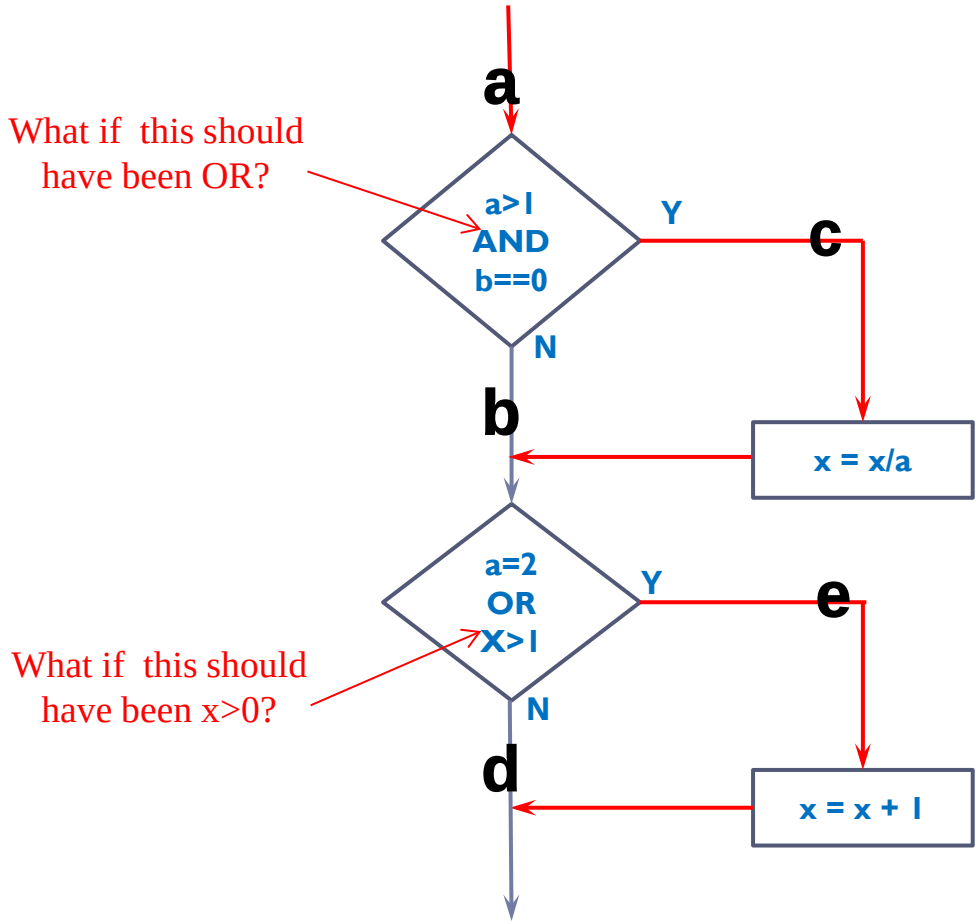
- Consider

```
public void foo( int a, int b, int x) {
    if (a>1  &&  b==0) {
        x = x/a;
    }
    if (a==2  || x>1) {
        x = x+1;
    }
}
```



Myers Ch. 4, pp. 3-9

Statement coverage



Test input

a=2, b=0, x=3

```
public void foo( int a, int b, int x) {
    if (a>1  &&  b==0) {
        x = x/a;
    }
    if (a==2  || x>1) {
        x = x+1;
    }
}
```

Myers Ch. 4, pp. 3-9

Decision coverage

Each decision outcome covered by a test case (cover every branch)

Either

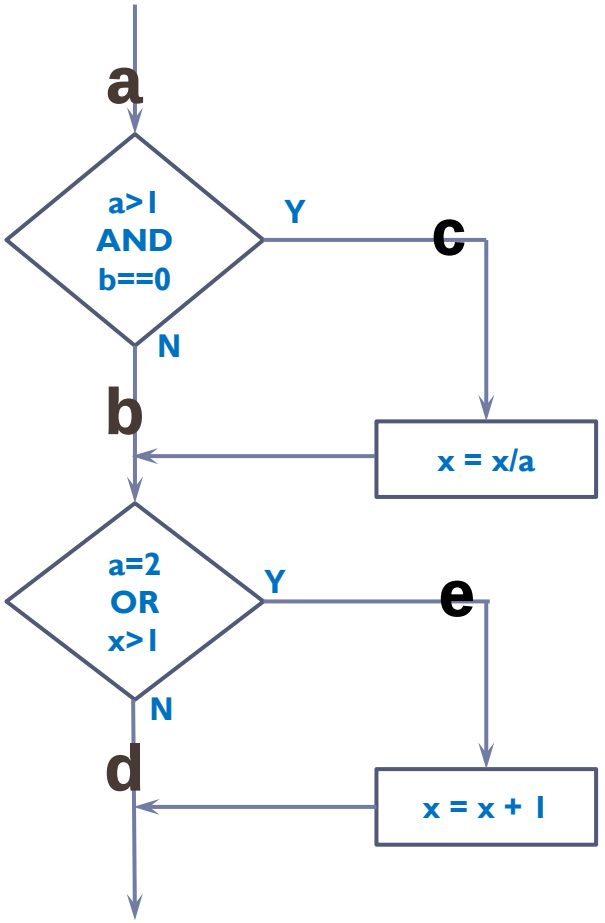
- paths ace and abd

Or

- paths acd and abe

BUT

- Only the former explores the path where x is not changed

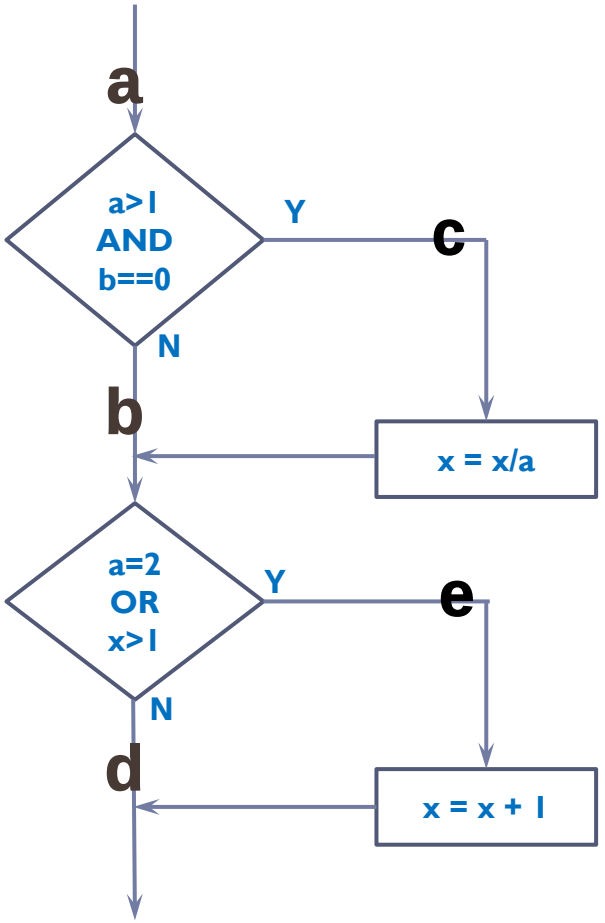


Myers Ch. 4, pp. 3-9

Decision/condition coverage

Each condition in a decision outcome covered by a test case

- Test cases cover 8 combinations:
  1. a>1, b=0
  2. a>1, b!=0
  3. a<=1, b=0
  4. a<=1, b!=0
  5. a=2, x>1
  6. a=2, x<=1
  7. a!=2, x>1
  8. a!=2, x<=1

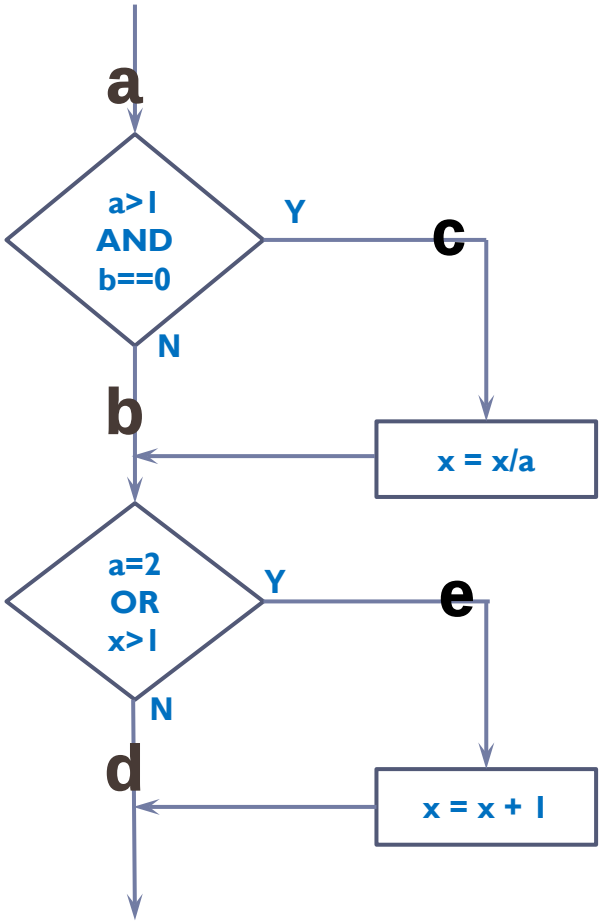


Myers Ch. 4, pp. 3-9

Decision/condition coverage

All combinations of conditions

1. a>1, b=0
2. a>1, b!=0
3. a<=1, b=0
4. a<=1, b!=0
5. a=2, x>1
6. a=2, x<=1
7. a!=2, x>1
8. a!=2, x<=1



Test cases

a=2, b=0, x=4	covers 1, 5 (ace)
a=2, b=1, x=1	covers 2, 6 (abe)
a=1, b=0, x=2	covers 3, 7 (abe)
a=1, b=1, x=1	covers 4, 8 (adb

We missed acd

Myers Ch. 4, pp. 3-9

Strengths of White-Box Testing

- It is a very natural approach when testing a GUI.
  - If you test set includes a path through every implementation of a MouseListener (in an AWT code) then your test coverage will include most of the GUI activity.
- It is a very efficient approach for strongly-OO code.
  - Unit testing: the process of testing each object (or method), to confirm that it works correctly. Usually combined with integration testing, to confirm that the modules "work correctly together".
  - If each unit is straight-line code or has just a single branch or loop, then we can easily write a "gold standard" collection of (white-box) unit tests.
  - If the integration code does not have complex paths, then it can be white-box tested with good coverage (but probably not to gold-standard).

Summary

- Black box
  - Design tests from the specifications only (no knowledge of code structure)
    - Tester must understand the user perspective
    - Independent tester? Or developer with domain knowledge?
  - Techniques
    - Equivalence partitioning (split the input into partitions, where values in each partition can be viewed as being 'similar')
    - Boundary value analysis (for each partition, choose values at the boundaries over those in the middle of the partition)

Summary

- Black box

Thoughts
Perhaps if the software-under-test is an application, someone who understands the users viewpoint will be more effective?
Is this technique really appropriate for within-development? What if the software-under-test is an API? Who is the user?
In a way, Black box testing can be viewed as testing interfaces – between human user and application, system interfaces, development modules, …
Can we use only for functionality? Or can we use to test other quality characteristics (efficiency, reliability, …)?

Summary

- White box
  - Design tests from a knowledge of code structure
    - Tester must be familiar with programming language
    - Developer ? (BEFORE submitting code)
  - Logic path techniques (in order of strength)
    - Statement coverage
    - Decision coverage
    - Decision/condition coverage

Summary

- White box

Thoughts
Developer unwillingness to find defects in his or her own work (Myers' psychology of testing).
What if the code is wrong in the first place (developer didn't understand the specification)? Tests will pass when carried out by developer, but QA may later reject code when testing from the specs.
It might be difficult to find inputs that will force a test to take a specific path. Even more difficult to be sure every path is taken (recognising 'dead code' is surprisingly difficult in a procedural programming language such as Java or C).
Aim to exercise the most likely usage paths? Is the developer the best person to know this? Is this where Black box testing comes in?

Myers' Principles

- "Continuing with the major premise of this chapter,
  - "That the most important considerations in software testing are issues of psychology,
- "We can identify a set of vital testing principles or guidelines."

- Wow – artistic guidelines for testers!
  - Myers is giving advice on how you can be a better tester.
  - More precisely, he's telling you what he thinks a tester should do.

- "Most of these principles may seem obvious, yet they are all too often overlooked."

Principle 1

- "A necessary part of a test case is a definition of the expected output or result."

Rationale:

- "If the expected result of a test case has not been predefined,
  - "chances are that a plausible, but erroneous, result will be interpreted as a correct result
  - "because of the phenomenon of 'the eye seeing what it wants to see'."

Prescription:

- "A test case must consist of two components:

1. "A description of the input data to the program.
2. "A precise description of the correct output of the program for that set of input data."

I think Myers' advice is sound. Do you? (Are you willing to give his prescription a go? Or do you think it might poison you?)