black box versus white box

**Black box testing** verifies that **requirements are met**.
**White box testing** looks for **common defects in code**.
Both are necessary.
Logic behind black-box testing:
Defects in the requirements (e.g., the preconditions and postconditions) often occur at **behavioral boundaries**.

Logic behind white-box testing:
Defects in the code often fit into a **taxonomy of common defects**, and manifest as

- **statements that crash when executed**,
- variables that remain **uninitialized when used**, or, more generally,
- **paths through the code** that do not work as expected.
Don't "bug" me

- History of the word "bug": in the time of Grace Hopper, logic elements were large enough that a real physical bug (of the insectoid variety) could get trapped between the relays and crash a program.
- There is no such thing as a "bug" in a modern program.
- Instead, programs have "defects", which means that in some way or other, they don't meet requirements.
- A "bug" is a semantic description of a "defect", e.g., what preconditions didn't assure what postconditions.

"Defects are in the code, bugs are in the comments!"
George Berkeley's question
Wednesday, November 11, 2009
3:25 PM

If a tree falls in the forest, and no one hears it, does it make a sound?

If a defect in code never manifests, is it still a defect?

Consider

```c
if (0) { *(char *)(0)='h'; }
```

(guaranteed segmentation fault, that can't happen!)
White box methodologies

Execute every statement.
Execute a representative set of paths.
Constructing a white-box test

Determine a path you want execution to take. Reverse-engineer an input that will force execution to take that path. Compute the expected output for that input from postconditions.
Example of white-box testing

```c
void foo(int i) {
    if (i>10) printf("hi");
    else printf("\n\n","ho\n"); // a bug
}
```

Two mutually exclusive cases: i>10 and !(i>10).

Possible tests:
- i=5
- i=15

will execute every statement at least once.
Problems with white-box testing

Constructing an input that will exercise a statement can be a complicated exercise in logic. There may not be an input that exercises a statement.

Consider:

```c
if (i>10) {
    printf("hi\n");
    if (i>9) {
        printf("ho\n");
    } else {
        printf("foo\n"); // no input will execute
    }
}
```
Some complications of logic:
  o Consider the general problem of choosing an input to a function so that a particular statement is executed.
  o This is less complicated than -- but related to -- computing the weakest preconditions to the function that exercise the function.
  o Difference is that you only need to compute "sufficient" -- and not "weakest" -- preconditions.
  o The usual method is to "try examples at random" until one finds one that exercises the line.
Cyclomatic basis testing
  o Some defects don't manifest until one takes a particular **path** through execution.
    E.g., consider
    ```c
    int *j=NULL;
    if (i>10) {
      j=&i;
    }
    if (i<20) {
      if (i>5) {
        *j=1; // potential "oops"
      }
    }
    ```
  o The only way you'll even see the defect is if execution takes a **particular path** in which the first, second, and third "if" statements are executed.
  o So, you need **5<i<10** to see the defect.
  o Exercising every line of code may **not** detect it!
Cyclomatic complexity

Depict program flow possibilities via a program graph. Start and end nodes represent beginning and end of program (double-circled). Arrows depict possible program flows. Branches depict binary "if" statements. Depict "while" as equivalent if statements.

Define the **cyclomatic complexity** of a program as the number of branches + 1

**Better metric** of program complexity than LOC

Has an **algebraic** meaning.

Some things to watch out for:

- implicit "if" statements, e.g., "while", &&, ||
- multiple "return" statements lead to one end node.
Computing white-box complexity

Transform code to equivalent (binary) "if" statements. Draw graph where "if"s are nodes. Exclude other nodes from graph.

Some common transformations:

```c
while (A) { B }
becomes
loop:
    if (!(A)) goto out;
    B;
    goto loop;
out:

switch (A) {
case B1:
    C1;
    break;

case B2:
    C2;
    break;
default:
    D;
}
becomes
if (A==B1) {
    C1;
} else {
```
if (A == B) {
    C2;
} else {
    D;
}

if (A && B) { C }
else { D }

becomes
if (A) {
    if (B) { C }
    else { D }
} else { D }

x=(a?b:c);

becomes
if (a) {
    x=b;
} else {
    x=c;
}
Example of cyclomatic complexity

void foo(int i) {
    int j=0;
    for (j=i; j>=0; j--) {
        printf("%d\n",j);
    }
}

becomes

void foo(int i) {
    int j=0;
{
    j=i
      while (j>=0) {
        printf("%d\n",j);
        j--;
      }
}

which becomes

void foo(int i) {
    int j=0;
    j=i;
    loop:
      if (!j>=0)) goto out;
}
printf("%d\n", j);
j--;
goto loop;
out:
}

which has program graph:
A path through a program graph is represented as a **labeling of the graph** with numbers representing the times the edge is taken.
Can "add" paths
"subtracting" one path from another

Can "subtract" paths
Paths form a vector space:

given paths $P$, $Q$, it is reasonable to compute $aP + bQ$, where $a$ and $b$ are integers!
McCabe's Theorem: the cyclomatic complexity of a program is the dimension of the paths through the program, viewed as a vector space.
Cyclomatic complexity testing

Compute a basis of the vector space of paths. Make one test per basis element. Then "you have tested representatives of all paths".

(Note that this is in fact an equivalence partitioning argument; the basis paths are representatives of equivalence classes)
Not all paths correspond to program flows, e.g.,

It is necessary to pick paths for the basis that can be tested.
You know you have a complete set of basis paths when they are linearly independent, i.e., you can't make one from the others by adding or multiplying. you have n of them where n is cyclomatic complexity
Example of basis formation

Monday, November 16, 2009
1:14 PM

Basis: all arrows labeled with "1"
\[ B = A + D - C \]
McCabe's conjecture

A function with a cyclomatic complexity > 10 cannot be tested via human means.

So how complex can we make a single function?
Depends upon whether you want to be able to test it.