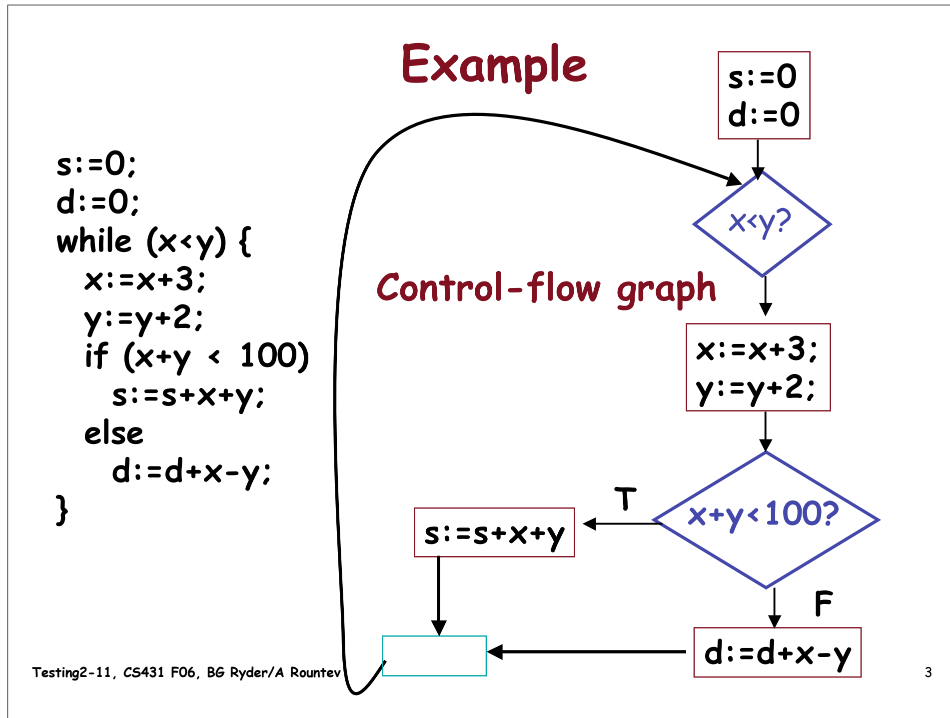
Testing2

- White box testing
  - Control-flow and dataflow metrics
  - Coverage metrics
- Black box testing
- Testing OO programs
  - Class testing
    - Testing polymorphism
    - Building call graphs using class hierarchy information

Control-flow-based Testing

- Traditional form of white-box testing
- Step 1: From the source code, create a graph describing the flow of control
  - Called the control flow graph
  - The graph is created (extracted from the source code) manually or automatically
- Step 2: Design test cases to cover certain elements of this graph
  - Nodes, edges, branches, paths

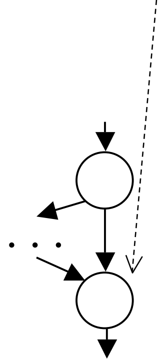


Elements of a Control Flow Graph

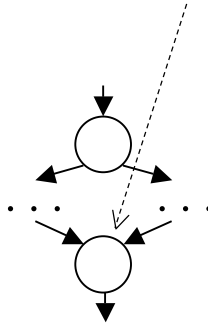
- Three kinds of nodes:
  - Statement nodes: single-entry-single-exit sequences of statements
  - Predicate (decision) nodes: conditions for branching
  - Auxiliary nodes: (optional) for easier understanding of conditional flow constructs (e.g. merge points for IF)
- Edges: show possible flow of control

IF-THEN, IF-THEN-ELSE, SWITCH

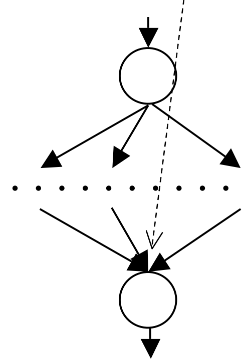
if (c)
  then …
join point



if (c)
  then …
  else …
join point

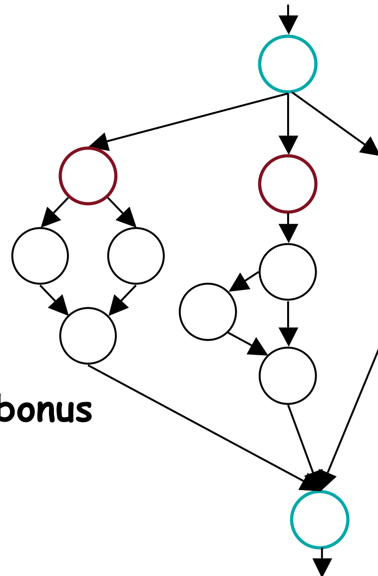


switch (c)
  case 1: …
  case 2: …
  . . .
join point



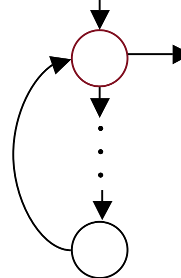
Example

```
switch (position)
  case CASHIER:
    if (empl_yrs > 5)
      bonus := 1;
    else
      bonus := 0.7;
    break;
  case MANAGER:
      bonus := 1.5;
      if (retiring_soon)
         bonus := 1.2 * bonus
      break;
  case ...
endswitch
```



Mapping for Loops

```
while (c) {
  …
}
```



Note: other loops (e.g. FOR, DO-WHILE, …) are mapped similarly

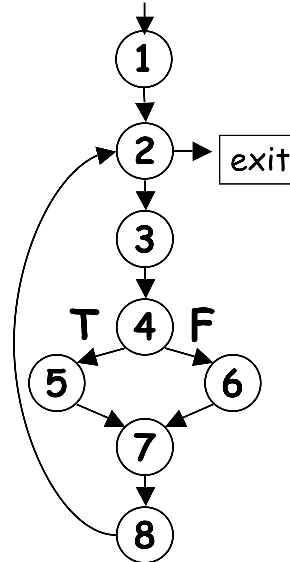
Mini-assignment: figure out how to map these other styles of loops

Statement Coverage

- Given the control flow graph, define a coverage "target" and write test cases to achieve it
- Traditional goal: statement coverage
  - Find a set of tests that cover all nodes
- *Hypothesis: Code that has not been executed during testing is more likely to contain errors*
  - Often this is the "low-probability" to be executed code

Example

- Suppose that we write and execute two test cases
- Test case #1: follows path 1-2-exit
- Test case #2: 1-2-3-4-5-7-8-2-3-4-5-7-8-2-exit (loop twice, and both times take the true branch)
- Problem: node 6 is never executed, so we don't have 100% statement coverage



Branch Coverage

Goal: write tests that cover all branches of the predicate nodes

- *True* and *false* branches of each IF
- The two branches corresponding to the condition of a loop
- All alternatives in a SWITCH

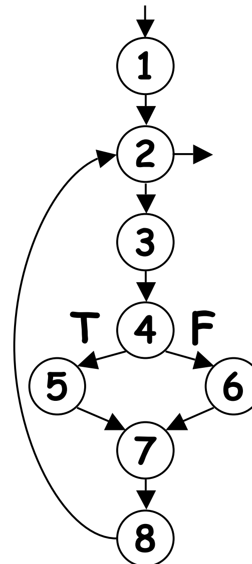
- In modern languages, branch coverage implies statement coverage
  - Because there are no goto's

Branch Coverage

- Statement coverage does not imply branch coverage
- Example: if (c) then s;
  - By executing only with c=true, we will achieve statement coverage, but not branch coverage
- Motivation: experience shows that many errors occur in "decision making"
  - Plus, it subsumes statement coverage

Example

- Same example as before: two test cases
  - Path 1-2-exit
  - Path 1-2-3-4-5-7-8-2-3-4-5-7-8-2-exit
- <u>Problem</u>: the "false" branch of 4 is never taken - don't have 100% branch coverage



Achieving Branch Coverage

- Branch coverage: a necessary minimum
  - Pick a set of start-to-end paths that cover all branches, and write tests cases to execute these paths
- Basic strategy
  - Add a new path that covers at least one edge that is not covered by the current paths
  - Sometimes the set of paths chosen with this strategy is called the "basis set"
    - Cf PRE Ch 14.4.2

Testing Loops

- Simple loops
    - Skip loop entirely
    - Go once through the loop
    - Go twice through the loop
    - If loop has max passes=n, then go n-1,n, n+1 times through the loop
- Nested loops
    - Set all outer loops to their minimal value and test innermost loop
    - Add tests of out-of-range values
    - Work outward, at each stage holding all outer loops at their minimal value
    - Continue until all loops are tested

Dataflow-based Testing

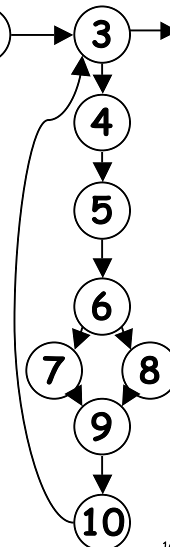
- **Test connections between variable definitions ("write") and variable uses ("read")**
- **Variation of the control flow graph**
  - A node represents a single statement, not a single-entry-single-exit chain of statements
- **Set** DEF(n) **contains variables that are** defined (written) **at node n**
- **Set** USE(n): **variables that are** read

Example

assume y is already initialized

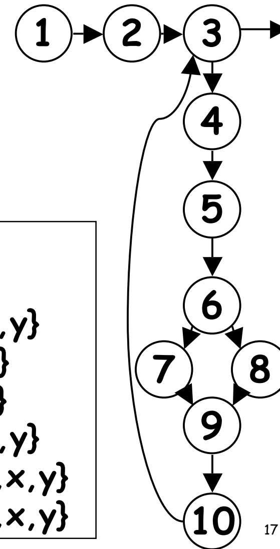
```
1 s:=0;
2 x:=0;
3 while (x<y) {
4   x:=x+3;
5   y:=y+2;
6   if (x+y<10)
7     s:=s+x+y;
    else
8     s:=s+x-y;
  }
```

DEF(1)={s} USE(1)=∅
DEF(2)={x} USE(2)=∅
DEF(3)=∅ USE(3)={x,y}
DEF(4)={x} USE(4)={x}
DEF(5)={y} USE(5)={y}
DEF(6)=∅ USE(6)={x,y}
DEF(7)={s} USE(7)={s,x,y}
DEF(8)={s} USE(8)={s,x,y}
DEF(9)=∅ USE(9)=∅
DEF(10)=∅ USE(10)=∅



Reaching Definitions

A definition of **x** at **n1** reaches **n2** if and only if there is a path between **n1** and **n2** that does not contain a definition of **x**



Reaches nodes 2, 3, 4, 5, 6, 7, 8, but not 9, 10

DEF(1)={s}	USE(1)=∅
DEF(2)={x}	USE(2)=∅
DEF(3)=∅	USE(3)={x,y}
DEF(4)={x}	USE(4)={x}
DEF(5)={y}	USE(5)={y}
DEF(6)=∅	USE(6)={x,y}
DEF(7)={s}	USE(7)={s,x,y}
DEF(8)={s}	USE(8)={s,x,y}

Def-Use Pairs

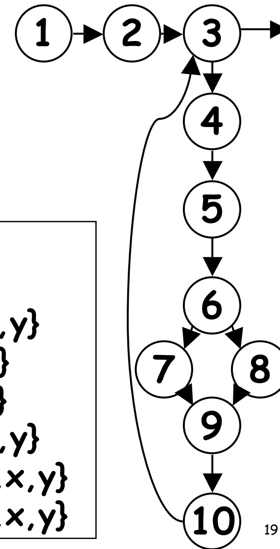
- A **def-use (DU) pair** for variable x is a pair of nodes (n1,n2) such that
  - x is in DEF(n1)
  - the definition of x at n1 reaches n2
  - x is in USE(n2)
- The value that is assigned to x at n1 is used at n2
  - Since the definition reaches n2, along some path n1…n2 the value is not "killed"

Example of Def-Use Pairs

Reaches nodes 2, 3, 4, 5, 6, 7, 8, but not 9, 10

For defn of s at 1, two DU pairs 1-7, 1-8

DEF(1)={s} USE(1)=∅
DEF(2)={x} USE(2)=∅
DEF(3)=∅ USE(3)={x,y}
DEF(4)={x} USE(4)={x}
DEF(5)={y} USE(5)={y}
DEF(6)=∅ USE(6)={x,y}
DEF(7)={s} USE(7)={s,x,y}
DEF(8)={s} USE(8)={s,x,y}



Dataflow-based Testing

- Identify all DU pairs and construct test cases that cover these pairs
  - Variations with different "strength"
- All-DU-paths: for each DU pair (n1,n2) for x, exercise all possible paths n1…n2 that are clear of definitions of x
- All-uses: for each DU pair (n1,n2) for x, exercise at least one path n1…n2 that is clear of definitions of x

Dataflow-based Testing

- All-defs: for each definition, cover at least one DU pair for that definition
  - i.e., if x is defined at n1, execute at least one path n1…n2 such that x is in USE(n2) and the path is clear of definitions of x
- All-defs <<(subsumes) all-uses << all DU-paths
- Motivation: see the effects of using the values produced by computations
  - Focuses on the data, while control-flow-based testing focuses on the control

Dataflow-based Testing

- Best criteria (?): *all-paths*
  - Select data that traverses all paths in a program, but possible problems:
    - Data causing execution to traverse a path, may not reveal an error on that path
    - There may be an infinite number of paths due to loops
- Rapps & Weyuker 1985 contribution
  - Designed a family of test data selection criteria so finite number of paths traversed
  - Systematic exploration of satisfying the criteria
  - Coverage criteria can be automatically checked

S. Rapps, E. Weyuker, "Selecting Software Test Data Using Data Flow Information, IEEE TSE, April 1985, pp 367-375.

Black-box Testing

- Unlike white-box testing: don't use any knowledge about the internals of the code
- Test cases are designed based on specifications
  - Test of expected behavior
- Example: search for a value in an array
  - Postcondition: return value is the index of some occurrence of the value, or -1 if the value does not occur in the array

Equivalence Partitioning

- Consider input/output domains and partition them into equivalence classes
  - For different values from the same class, the software should behave equivalently
- Test values from each class
  - For input range 2..5: "less than 2", "between 2 and 5", and "greater than 5"
- Testing with values from different classes is more likely to find errors than testing with values from the same class

Equivalence Classes

- Examples
  - Input x in range [a..b]: three classes "x<a", "a<=x<=b", "b<x"
  - Boolean: classes true and false
  - Some classes may represent invalid input
- Choosing test values
  - Choose a typical value in the middle of the class(es) that represent valid input
  - Choose values at the boundaries of classes
    - e.g. for [a..b], use a-1, a, a+1, b-1, b, b+1

Example

- Spec says that the code accepts between 4 and 24 inputs; each is a 3-digit integer
- One partition: number of inputs
  - Classes "x<4", "4<=x<=24", "24<x"
  - Chosen values: 3,4,5,14,23,24,25
- Another partition: integer values
  - Classes: "x<100", "100<=x<=999", "999<x"
  - Chosen values: 99,100,101,500,998,999,1000

Another Example

- Similarly for the output: exercise boundary values
- Spec: the output is between 3 and 6 integers, each in the range 1000-2500
- Try to design inputs that produce
  - 3 outputs with value 1000
  - 3 outputs with value 2500
  - 6 outputs with value 1000
  - 6 outputs with value 2500

Example: Searching

- Search for a value in an array
  - Return: index of some occurrence of the value, or -1 if the value does not occur
- One partition: size of the array
  - Programmer errors are often made for size 1: a separate equivalence class
  - Classes: "empty array", "array with one element", "array with many elements"
- Another partition: location of the value
  - "first element", "last element", "middle element", "not found"

Example: Searching

Array	Value	Output
empty	5	-1
[7]	7	0
[7]	2	-1
[1,6,4,7,2]	1	0
[1,6,4,7,2]	4	2
[1,6,4,7,2]	2	4
[1,6,4,7,2]	3	-1

Object-Oriented Software

- Initially hoped it would be easier to test OO software than procedural software
  - Soon became clear that this is not true
- Some of the older testing techniques are still useful
- New testing techniques are designed specifically for OO software

One Difference: Unit Testing

- Traditional view of "unit": a procedure
- In OO: a method is similar to a procedure
- But a method is part of a class, and is tightly coupled with other methods and fields in the class
- The smallest testable unit is a class
  - It doesn't make sense to test a method as a separate entity
- Unit testing in OO = class testing

Class Testing

- Traditional black-box and white-box techniques still apply
  - E.g. testing with boundary values
  - Inside each method:
    - Obtain at least 100% branch coverage;
    - Cover all DU-pairs inside a method (intra-method)
  - DU pairs that cross method boundaries (inter-method)
    - Example: inside method m1, field f is assigned a value; inside method m2, this value is read

Example: Inter-method DU Pairs

```
class A {
        private int index;
        public void m1() {
            index = …;
            …
           m2();
        }
        private void m2() { … x = index; … }
        public void m3() { … z = index; … }
}
```

test 1: call m1, which writes index and then calls m2 which reads the value of index
test 2: call m1, and then call m3

Possible Test Suite

```
public class MainDriver {
      public static void main(String[] args) {
            A a = new A();
            …
            a.m1();
            a.m3();
            …
}
```

Note: need to ensure that the actual execution exercises definition-free paths for each of the two DU pairs

Class Testing

- Also try to test all sequences of calls to public methods of A, that a client of A could invoke (intra-class)
  - Want to discover more DU edges to test, that can be setup by this sort of sequence of calls
- *Q: What about overriding subclass methods? How do they get tested?*

Polymorphism

- Example: class A with subclasses B and C
  - class A { … void m() {…} …}
  - class B extends A { … }
  - class C extends A { … void m() {…} … }
- Suppose inside class X there is call a.m(), where variable a is of type A
  - Could potentially send message m() to an instance of A, instance of B, or instance of C
  - The invoked method could be A.m or C.m

Testing of Polymorphism

- During class testing of X: "drive" call site a.m() through all possible bindings
- All-receiver-classes: execute with at least one receiver of class A, at least one receiver of class B, and at least one receiver of class C
- All-invoked-methods: need to execute with receivers that cover A.m and C.m
  - i.e. (A or B receiver) and (C receiver)
- *Q: How can we figure out the possible method targets?*

Compile-time Analysis

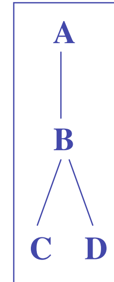
- Class Hierarchy Analysis (CHA)
- Use knowledge of type hierarchy to figure out possible method targets at a call site a.f()
  - Know all subclasses of a class A, when a declared to be an A object
  - Know all methods defined in those subclasses with same method signature f()
  - Refinement: also might collect info on which classes are actually instantiated (RTA) so don't over-expand call graph

Example

cf Frank Tip, OOPSLA'00

```
static void main(){
   B b1 = new B();
   A a1 = new A();
   f(b1);
   g(b1);
}
static void f(A a2){
   a2.foo();
}
static void g(B b2){
   B b3 = b2;
   b3 = new C();
   b3.foo();
}
```

```
class A {
   foo(){..}
}
class B extends A{
   foo() {…}
}
class C extends B{
   foo() {…}
}
class D extends B{
   foo(){…}
}
```

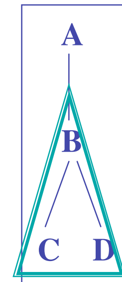


CHA Example

cf Frank Tip, OOPSLA'00

```
static void main(){
   B b1 = new B();
   A a1 = new A();
   f(b1);
   g(b1);
}
static void f(A a2){
   a2.foo();
}
static void g(B b2){
   B b3 = b2;
   b3 = new C();
   b3.foo();
}
```

```
class A {
   foo(){..}
}
class B extends A{
   foo() {…}
}
class C extends B{
   foo() {…}
}
class D extends B{
   foo(){…}
}
```



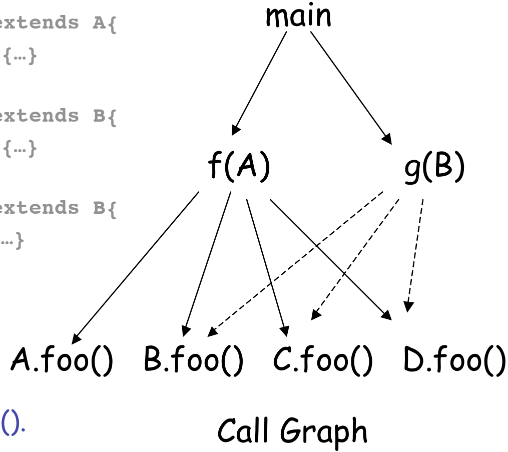
Cone(Declared_type(receiver))

CHA Example

```
static void main(){
   B b1 = new B();
   A a1 = new A();
   f(b1);
   g(b1);
}
static void f(A a2){
   a2.foo();
}
static void g(B b2){
   B b3 = b2;
   b3 = new C();
   b3.foo();
}
```

```
class A {
   foo(){..}
}
class B extends A{
   foo() {…}
}
class C extends B{
   foo() {…}
}
class D extends B{
   foo(){…}
}
```

All-Receiver-class coverage requires that we cover each possible receiver type at call in f().



Call Graph

All-invoked-method coverage requires that we cover each outgoing edge from the call in f().

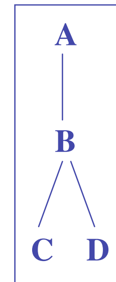
RTA Example

cf Frank Tip, OOPSLA'00

```
static void main(){
   B b1 = new B();
   A a1 = new A();
   f(b1);
   g(b1);
}
static void f(A a2){
   a2.foo();
}

static void g(B b2){
   B b3 = b2;
   b3 = new C();
   b3.foo();
}
```

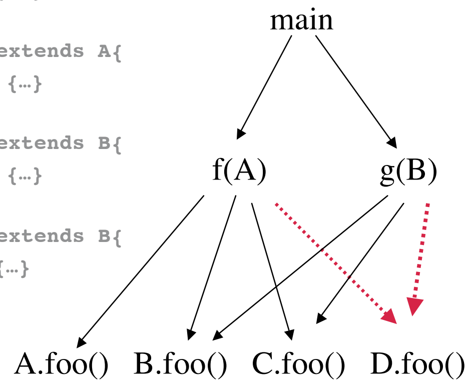
```
class A {
   foo(){..}
}
class B extends A{
   foo() {…}
}
class C extends B{
   foo() {…}
}
class D extends B{
   foo(){…}
}
```



RTA Example

```
static void main(){
   B b1 = new B();
   A a1 = new A();
   f(b1);
   g(b1);
}
static void f(A a2){
   a2.foo();
}
static void g(B b2){
   B b3 = b2;
   b3 = new C();
   b3.foo();
}
```

```
class A {
   foo(){..}
}
class B extends A{
   foo() {…}
}
class C extends B{
   foo() {…}
}
class D extends B{
   foo(){…}
}
```



Call Graph