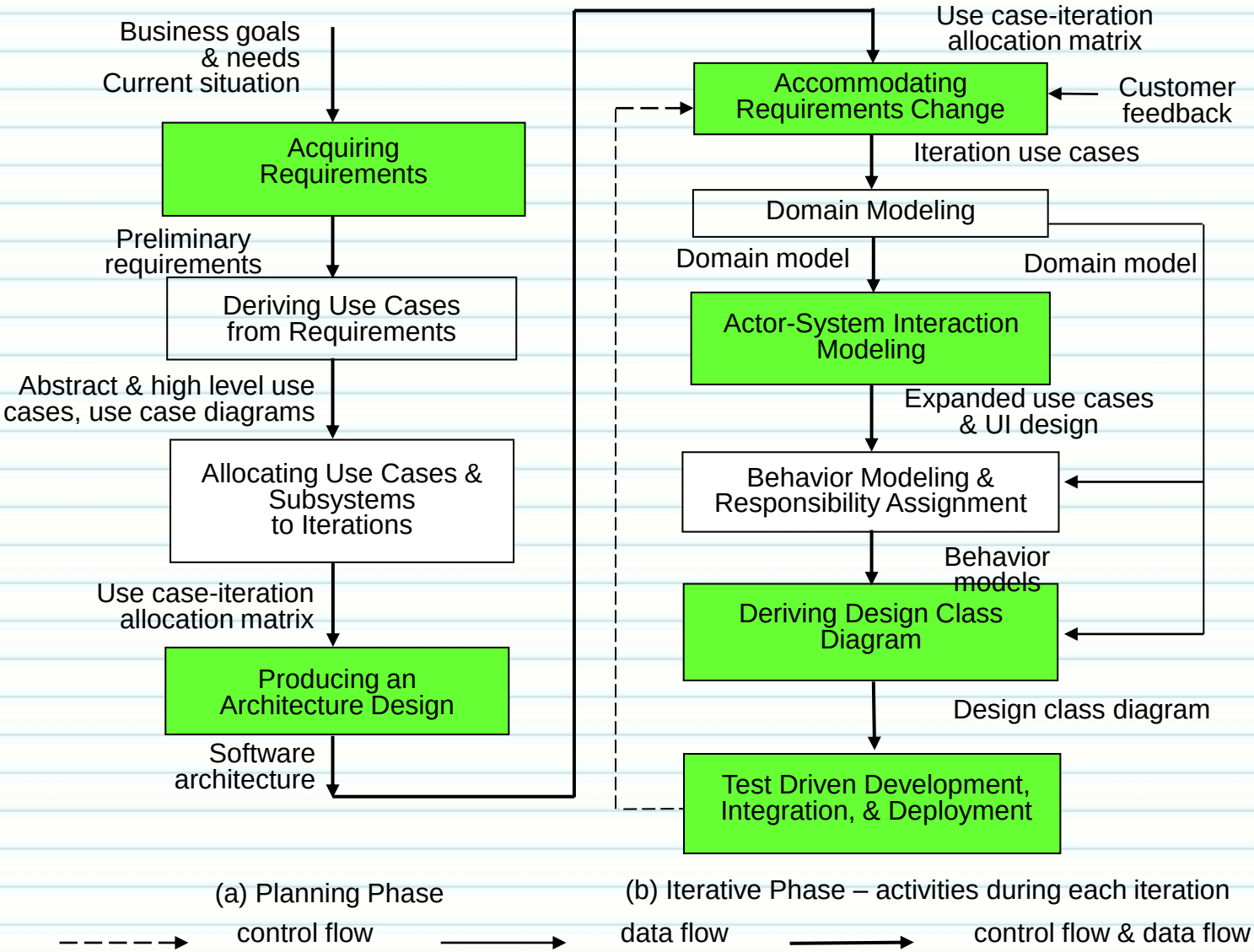
Chapter 20 – Software Testing

Dr. Michael F. Siok, PE, ESEP
UT Arlington
Computer Science and Engineering

Key Takeaway Points

- Software testing is a dynamic validation technique
- Software testing can detect errors in the software but cannot prove that the software is free of errors
- Software testing increases the development team's confidence in the software product

Testing in the Methodology Context



Software is Ubiquitous

- Everywhere!
  - Automotive, phones, industrial control, power distribution, embedded devices, aerospace, military, IT systems
- Software is used in high security and/or safety applications
  - Medical devices, nuclear reactors, aerospace (commercial and military)
  - Banking Systems, identification systems, surveillance systems
- The user expectation that software will work reliably places important emphasis on software test

Other Motivations for Software Test

- Agile processes emphasize test
  - Test driven requirements (user stories)
  - Continual unit testing
- The Industry has created a number of Software Test Certification organizations
  - Some companies highly value these certifications $$
  - Much of the class material this semester will be aligned to the ISTQB (International Software Testing Qualifications Board) foundation level material and test (more later)
- As good as the CMMI is, it is particularly weak in the software test function
  - It does address many verification techniques but does not address common methods of software test, estimation, management or planning
- Most CS degree holders graduate without a single class in software test!
- Spectacular software failures described at: http://en.wikipedia.org/wiki/List_of_software_bugs

A Concrete Example

Fault: Should start searching at 0, not 1

```
public static int numZero (int [ ] arr)
{  // Effects: If arr is null throw NullPointerException
   // else return the number of occurrences of 0 in arr
   int count = 0;
   for (int i = 1; i < arr.length; i++)
   {
      if (arr [ i ] == 0)
      {
         count++;
      }
   }
   return count;
}
```

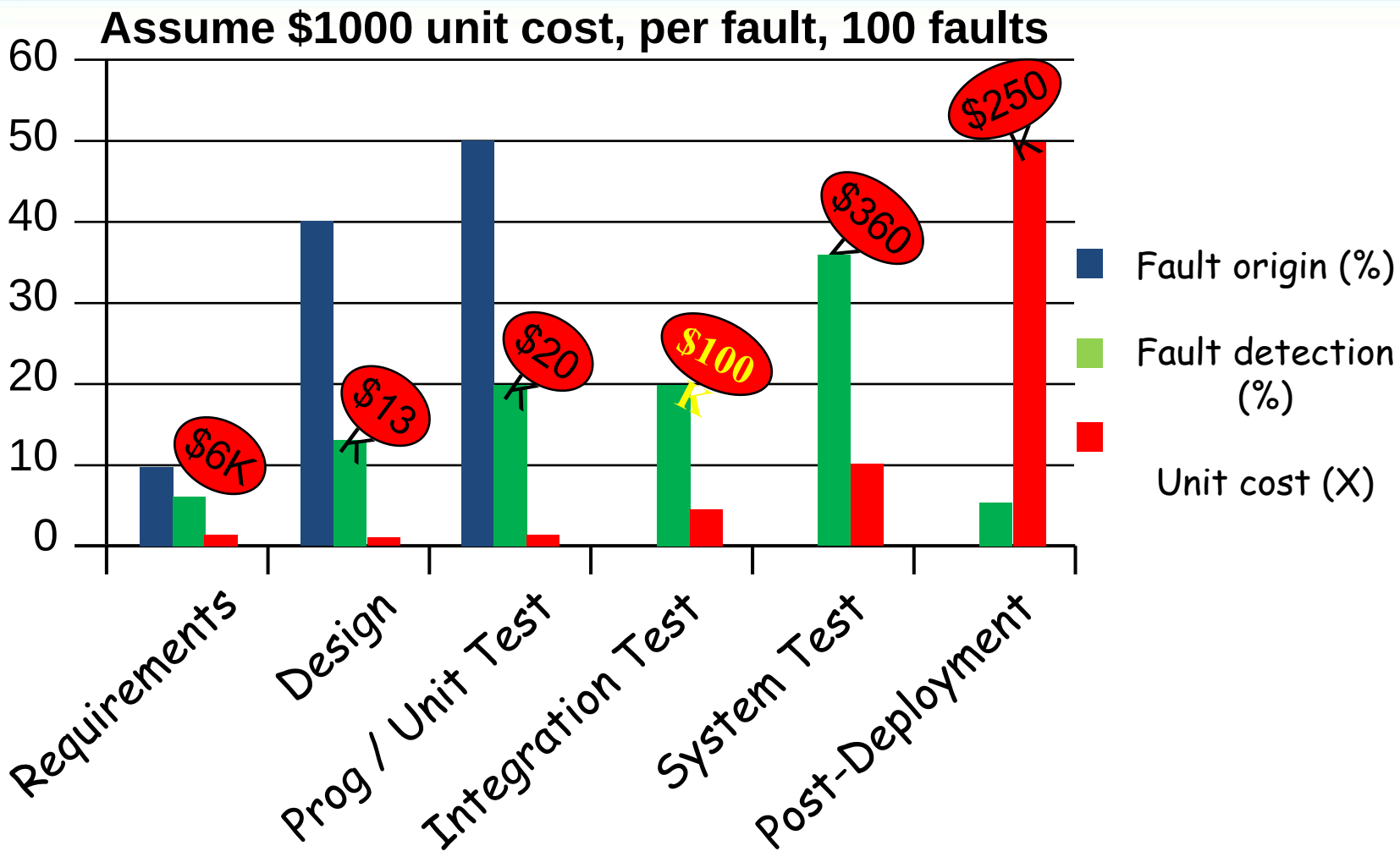
Test 1
[2, 7, 0]
Expected: 1
Actual: 1

Error: i is 1, not 0, on the first iteration
Failure: none

Test 2
[0, 2, 7]
Expected: 1
Actual: 0

Error: i is 1, not 0
Error propagates to the variable count
Failure: count is 0 at the return statement

Cost of Late Testing



Software Engineering Institute; Carnegie Mellon University; Handbook CMU/SEI-96-HB-002

Definition of Terms

- Definitions are from the IEEE 24765 and the ISTQB definitions when they agree no color is used - These terms are used throughout the software literature

- Bug: see Defect
- Defect: see Fault see Fault
- Error: "A human action that produces an incorrect result." This a mistake that a human makes
- Fault: "An incorrect step, process, or data definition in a computer program" "A flaw in a component or system that can cause the component or system to fail to perform its required function"
- Failure: "Termination of the ability of a product to perform a required function or its inability to perform within previously specified limits." "Deviation of a component or system from its expected delivery, service, or result"
- Mistake: see Error
- Latent Defect: An undiscovered defect in delivered software
- Test Oracle: "A source to determine expected results to compare with the actual result of the software under test."

A Better Understanding of the Terms

- Failure is what is observed "from the outside" – by the customer or other outside viewer
- A Failure is caused by one or more faults (an incorrect step or flaw) in the software – a fault is also known as a defect or bug – this is what we detect through test or inspection
- An Error – this is the mistake

- A good analogy is the following event
  - A human has a heart attack
  - The failure - the heart's ability to supply oxygen rich blood to the body - this can be observed
  - The fault - the heart muscle stops working because of oxygen starvation - one or more coronary arteries have become blocked
  - The error (mistake) – (probably) Poor diet - how do you spell "Supersize Me"?
  - Not all heart attacks are caused by the same faults or errors
  - Poor diet does not always cause a heart attack – but it is still there with the potential to cause a fault . . . which could cause the failure
  - Like defect detection above (test/inspection), the fault is fixed with medicine but the error (poor diet) may still be there

Defect Detection and Removal

- V&V is a defect (fault) detection activity - it is not a failure detection activity.
- The term ‘failure’ is an affect not a cause. One or more defects are the cause of the failure. This means that the same failure may still result when one of its contributing faults is fixed.
- An understanding of defects and removal is important
  - Defects are detected -> some or all detected defects are removed – the removal only occurs when the product is corrected
  - Defect removal is a function of both detection and removal - most software has more defects detected than removed – why?
  - Un-removed defects are called ‘restrictions’
  - Un-detected defects are ‘latent defects’
  - Almost all software has restrictions and latent defects
- We can predict both faults and failures in the software – the latter requires operational history

Verification/Validation and the Software Life Cycle

- The following are definitions from the IEEE SWEBOK V3 section 10.2.2
- Verification : "… ensure[s] that the product is built correctly." Is the product built to its requirements?
- Validation : "… ensure[s] that the right product is built—that is, the product fulfills its specific intended purpose." But then we need to ask, "Are the requirements correct?"
- Which of these activities detect defects?
- Common Software Development Life Cycles (SDLCs) include the V-model, Prototyping, Iterative and Incremental development, Spiral development, Rapid Application Development, Extreme Programming and Agile.
- In terms of V&V (which is a defect detection activity), most of what we care about from these SDLCs are
  - what activities detect defects, and
  - when these activities occur

Verification/Validation Activities and the SDLC

Note: this table is not meant to imply a specific SDLC

Activity	Verification	Validation
Software Requirements	•Requirements Technical Reviews •Requirements Based Testing	Customer Review, Expert Review, Modeling, Prototyping
High Level Design	•High Level Design Technical Reviews •Integration Level Testing	Modeling, Prototyping
Detailed Design	•Detailed Design Technical Reviews •Integration Level Testing	Modeling, Prototyping
Coding	•Code Technical Reviews •Unit Level Testing	Modeling, Prototyping

- Technical Reviews (SWEBOK) can be Formal Inspections, Walk-throughs, or Peer Reviews
  - For high maturity teams, the Technical Review activity can detect up to 90 percent of the software defects!

Software Testing and Techniques Are Essential

- *Technical reviews* can detect a significant percentage of defects, but it is important to understand what kinds of defects they detect
  - Defects are typically limited to the scope of the product being reviewed or possibly close neighbors
  - Defects are typically limited to those found statically (as opposed to dynamically such as execution)
- Example: for a "group" of requirements we would normally find the following during a technical review
  - Defects related to the requirements being reviewed (or possibly closely related requirements)
  - Issues with the completeness or correctness of specification (missing/incorrect thresholds, logic, sequencing, etc.)
- Defects that are related to dynamic or more global functionality are best suited for testing – the need for software testing will never be eliminated!
- Software test design
  - Where most defects are detected – not during test execution
  - Techniques are very useful for technical reviews also

A Few More Introductory Terms

- Quality: "The degree to which a component, system or process meets specified requirements and/or user/customer needs and expectations."

- Defect prevention: "A structured problem-solving methodology to identify, analyze and prevent the occurrence of defects." Defect prevention changes the process to prevent occurrence.
  - How does this compare with defect detection and removal?
  - What is quality control vs. quality assurance (defect prevention)?

- Debugging: "The process of finding, analyzing and removing the causes of failures in software."
  - How does debugging differ from defect detection?
  - From troubleshooting?

Seven Testing Principles

1) *Testing shows presence of defects*: Testing can show the defects are present, but cannot prove that there are no defects.

2) *Exhaustive testing* is not possible.

3) *Early testing*: In the software development life cycle testing activities should start as early as possible to reduce cost.

4) *Defect clustering*: A small number of modules contains most of the defects discovered. Sometimes referred to as the 80/20 rule.

5) *Pesticide paradox*: If the same kinds of tests are repeated again and again, eventually the same set of test cases will no longer be able to find any new bugs. (Does not apply to regression testing.)

6) *Testing is context-dependent*: Different software products have varying requirements, functions, and purposes.

   For example, safety–critical software is tested differently from an e-commerce site

7) *Absence-of-errors fallacy*: Declaring that "*a test has unearthed no errors*" is not the same as declaring the software "*error-free*".

Test Case Generation – White Box

- Knowing the internal workings of a product
  - focus on the program's structure and internal logic
  - test cases are designed according to the structure and internal logic

- Well-known techniques
  - Basis path testing: test cases are designed to exercise the control flow paths of a program
  - Condition testing: test cases are designed to exercise each outcome of a condition
  - Data flow testing: test cases are designed to test data elements' define and use relationships

Test Case Generation – Black Box

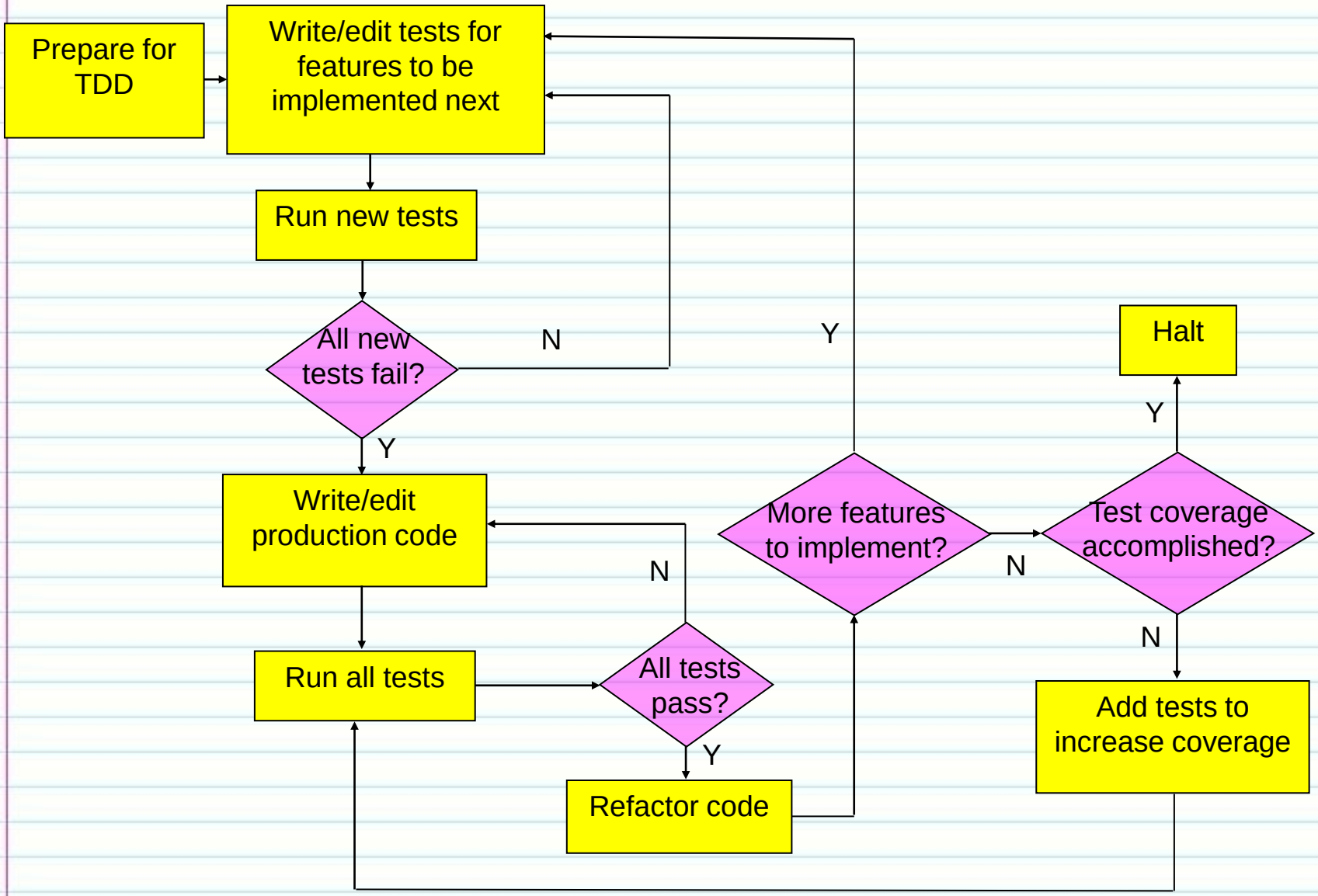
- Knowing the functional specification
  - Focus on functional requirements
  - Test cases are designed to test the functionality of the program

- Well-known techniques:
  - Boundary value analysis -- test cases are derived according to the boundary values of variables
  - Causal-effect analysis -- test cases are derived according to the stimuli and responses, and input output relationships
  - Equivalence partitioning -- test cases are derived from partitions of the input and output domains

Test Categories

- Unit Testing: testing at the individual module level, functions, modules, classes, etc.
- Integration Testing: testing the interfaces among the modules or components.
- Validation Testing: test according to customer's functional requirements.
- System Testing:
  - Recovery Testing
  - Security Testing
  - Stress Testing
  - Performance Testing

Test Driven Development



Merits of Test Driven Development

- TDD helps the team understand and improve requirements.
- TDD produces high-quality code.
- TDD improves program structure and readability through refactoring.
- TDD makes it easier to debug.

Black Box Testing Techniques

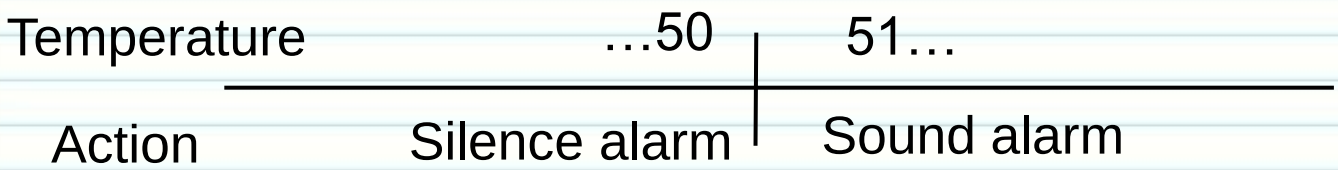
- Typical “black-box” test analysis and design techniques include:
  - Equivalence partitioning
  - Boundary value analysis
  - Decision table testing
  - State transition analysis
  - Use Case testing
  - Decision logic and Karnaugh maps

- Black box is a bit of a misnomer – we do not and cannot test strictly black box

- So these are correctly called *Specification-based* test analysis and design techniques, but we will use the term “black box” because it is commonly used

Equivalence Classes or Partitions

- This is a technique used to reduce a large input space down into a more manageable set
- I have a requirement to set an alarm when the temperature (assume integer) is greater than 50 degrees F, otherwise the alarm is not silenced

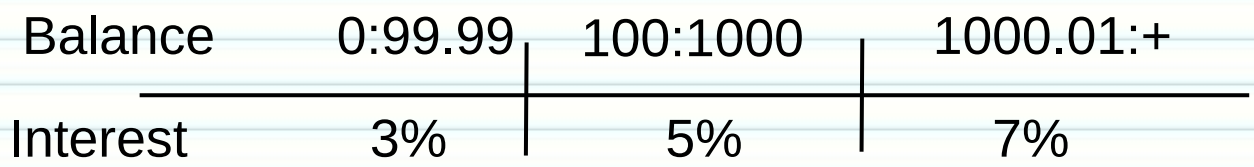


- I have two equivalence classes based on the actions needed – they are equivalent because the actions taken are equivalent within the same partition
- A domain partition is a partition of the input domain into a number of sub-domains
- A boundary is where two sub-domains meet

Equivalence Classes or Partitions (cont.)

- A savings account in a bank has a different rate of interest depending on the balance in the account.

1. 3% rate of interest is given if the balance in the account is less than $100
2. 5% rate of interest is given if the balance in the account is in the range of $100 to $1000
3. 7% rate of interest is given if the balance in the account is at least $1000

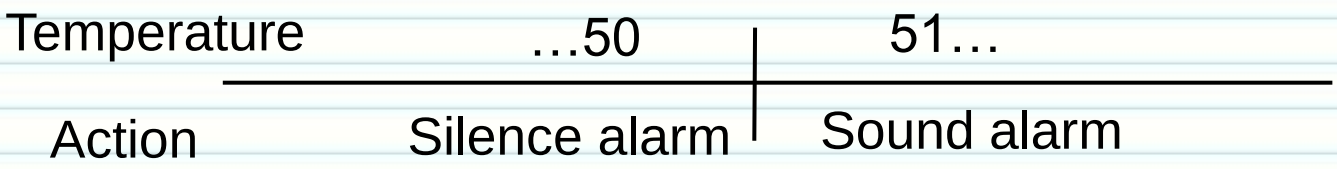


- We need to pay careful attention to the range of values
- What would happen to the ranges above if this was float but not dollars and cents? How would this change when we represent this in code?
- Here we have three partitions based on the set of actions

Boundary Value Analysis

- Boundary value testing does two things
  - It drastically reduces the number of test cases needed
  - It accounts/checks for the human tendency to have specification errors at boundary conditions

- We set inputs based on each "border" of the partition and check actual outputs to the expected condition



- We will have **a minimum of** two test cases

	Temperature	Alarm state
Test Case 1	50	Silence
Test Case 2	51	Sounding

Decision Tables

- If you are a new customer and you want to open a credit card account then there are three condition.
  1. You will get a 15% discount on all your purchases today.
  2. If you are an existing customer and you hold a loyalty card, you get a 10% discount.
  3. If you have a coupon, you can get 20% off today (but it can't be used with the 'new customer' discount). discount amounts are added, if applicable.

- Some interesting combinations…
  1. What happens if I am a new customer? Obvious!
  2. What happens if I am a new customer with a coupon?
  3. What happens if I am an existing customer with a coupon **and** a loyalty card?
  4. What happens if I am a new customer with a loyalty card and no coupon?
  5. What happens if I am a new customer with a loyalty card and a coupon?

Decision Tables (cont.)

- We might be tempted to code the logic as follows:

```
if (new_customer) {
  if (!loyalty_card) {
    if (coupon)
      discount = 0.2;
    else discount = 0.15; }
  else error;}
else {
  if (loyalty_card)
    discount += 0.1;
  if (coupon)
  discount += 0.2;}
```

- Is this technically correct? Is it understandable?

Decision Tables (cont.)

- We can make this much more concise and understandable.
- How many conditions do we have? How many possible combinations does that give us?
- How many actions do we have?

Conditions	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7	Rule 8
New Customer	F	F	F	F	T	T	T	T
Loyalty Card	F	F	T	T	F	F	T	T
Coupon	F	T	F	T	F	T	F	T
Actions								
Discount	0	20	10	30	15	20	X	X

- I have two cases where the requirements are not complete. I need to update the requirements to address what action I do with these.
- The best way to test this is to develop a test case for each column

Decision Tables (cont.)

- Loan Calculation

- The input has four fields and a Submit button, you can enter
  1. The amount of the total loan
  2. Interest rate
  3. The number of months you want to pay it off (term)

- Rules

  The monthly payment is calculated when the submit button is pressed if:

  1. All three are given
  2. Based on 5.5 % interest if the total loan and term is given
  3. Based on 48 months term if the total loan and interest are given

  An error condition is indicated:

  1. When the total loan amount has not been given

Decision Tables (cont.)

- First we do not list the Submit button as an action – why?
- Second, the requirements are incomplete – a condition in the decision table has not been specified

		Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7	Rule 8
Conditions	The amount of the total loan	F	F	F	F	T	T	T	T
	The interest rate	F	F	T	T	F	F	T	T
	Number of months	F	T	F	T	F	T	F	T
Actions	Compute based on 3 inputs								x
	Compute based on 5.5% interest						x		
	Compute based on 48 months							x	
	Error	x	x	x	x	?			

Decision Tables (cont.)

- Loan Calculation

- The input has four fields and a Submit button, you can enter
  1. The amount of the total loan
  2. Interest rate
  3. The number of months you want to pay it off (term)

- Rules

  The monthly payment is calculated when the submit button is pressed if:
  1. All three are given
  2. The total loan and term is given then it is based on a 5.5 % interest
  3. The total loan and interest are given then it is based on a 48 months term
  4. Only the total loan amount is provided, then the calculation is based off of the 5.5% interest and a 48 months term

  An error condition is indicated:
  1. When the total loan amount has not been given

Decision Tables (cont.)

- Now the requirements are complete – all conditions in the decision table have been specified

		Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7	Rule 8
Conditions	The amount of the total loan	F	F	F	F	T	T	T	T
	The interest rate	F	F	T	T	F	F	T	T
	Number of months	F	T	F	T	F	T	F	T
Actions	Compute based on 3 inputs								x
	Compute based on 5.5% interest					x	x		
	Compute based on 48 months					x		x	
	Error	x	x	x	x				

- How many test cases do I need to test this?

Decision Tables (cont.)

- Incomplete and Over-specification of decision tables
- A decision table is complete if every possible set of conditions has a corresponding action prescribed (otherwise I have a case where an action is missing) – we have already looked at these
- An over-specified decision table results in a logic error – e.g., accidentally performing multiple actions where unintended

Applying These Techniques Elsewhere

- We can apply Boundary Value and Equivalence classes/partitions to other areas as well
- Suppose an internal phone system for a company with 200 telephones has a 3-digit extension numbers from 100 to 699, we can identify the following classes/partitions and boundaries (Example from ISTQB Foundations)
- Digits: Valid partition 0-9, Invalid partition otherwise
- Number of Digits: Valid partition number 3 digits, Invalid boundary values of 2 and 4 digits
- Range of extension numbers: Valid partition 100 to 699, Two invalid boundaries of 099 and 700
- Assigned extensions: at least two partitions – Valid=assigned and Invalid=not assigned
- Boundary value of assigned extensions: lowest and highest could also be used

Inspection and Testing

Inspection

- static
- a verification technique
- can check conformance to coding standards
- cannot check satisfiability of non-functional requirements

Testing

- dynamic
- a validation technique
- cannot check conformance to coding standards
- can check satisfiability of non-functional requirements

Logic Coverage

- We are still looking at expressions independent of source code but will use source code constructs as they might be specified in requirements
- A logical expression is a **decision**, and the decision consists of **conditions**, conjoined by Boolean operators (and, or, ...). A condition contains no Boolean operators.
- A decision is a Boolean expression composed of conditions and zero or more Boolean operators. A decision without a Boolean operator is a condition.

- Logic Coverage

```
If (( (a>b) || C) && p(x))
    X=1
else
    X=2
If (x>9)
    ...
```

Decision
Condition

Decision Coverage

- For each Decision:
  - Have at least one test where it evaluates to true
  - Have at least one test where it evaluates to false
- Also known as:
  - Predicate coverage
  - Statement coverage (when we get to code)
- From a testing standpoint it is the weakest logic coverage
- It does not test the individual Conditions only that the overall Decision is evaluated to either True or False

Select one of this set
and one of this set

a > c	C	p(x)	((a>b)||C)&&p
FALSE	FALSE	FALSE	FALSE
FALSE	FALSE	TRUE	FALSE
FALSE	TRUE	FALSE	FALSE
TRUE	FALSE	FALSE	FALSE
TRUE	TRUE	FALSE	FALSE
FALSE	TRUE	TRUE	TRUE
TRUE	FALSE	TRUE	TRUE
TRUE	TRUE	TRUE	TRUE

Condition Coverage

- A condition is a leaf-level Boolean expression (it cannot be broken down into a simpler Boolean expression).
- For each Condition in a Decision:
  - Evaluate to true
  - Evaluate to false
- Also known as Clause (or condition) coverage

Here each Condition is tested to both true and false but the Decision is not

a > c	C	p(x)	((a>b)||C)&&p
FALSE	FALSE	FALSE	FALSE
FALSE	FALSE	TRUE	FALSE
FALSE	TRUE	FALSE	FALSE
FALSE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	FALSE
TRUE	FALSE	TRUE	TRUE
TRUE	TRUE	FALSE	FALSE
TRUE	TRUE	TRUE	TRUE

- There are methods to gain additional compound conditions (like Condition/Decision Coverage)

Condition/Decision Coverage

- When applied to logical statements the requirement is…
- Every condition in a decision has taken all possible outcomes at least once, and every decision has taken all possible outcomes at least once.
- This is a requirement for DO-178 which is used world-wide for commercial aerospace applications. It is used for the second most stringent coverage criteria – applications that have some safety involvement
- This is also a criteria used for Security relevant applications

MCDC Coverage and Test Case Design

- Modified Condition/Decision Coverage is the highest (and most expensive level of test coverage) of logical statements

- Every condition in a decision has taken on all possible outcomes at least once, and each condition has been shown to affect that decision's outcome independently. A condition is shown to affect a decision's outcome independently by varying just that condition while holding fixed all other possible conditions.

- So, for A && B, here is the corresponding truth table

a	b	a && b
FALSE	FALSE	FALSE
FALSE	TRUE	FALSE
TRUE	FALSE	FALSE
TRUE	TRUE	TRUE

MCDC Coverage and Basic Logic

- So, for A && B

a	b	a && b
FALSE	FALSE	FALSE
FALSE	TRUE	FALSE
TRUE	FALSE	FALSE
TRUE	TRUE	TRUE

- I want to stimulate one input at a time and satisfy independence requirements
  1. TRUE,TRUE is the only test that returns a TRUE decision so I must chose this
  2. FALSE,TRUE is required as it is the only test that changes the value of only A which changes the decision to FALSE - establishing the independence of A – ANDs are sensitive to FALSE
  3. Similarly for B – since ANDs are sensitive to FALSE then I want to only set B to FALSE and A to TRUE, which will cause a FALSE decision
- We have shown above that a FALSE value for A alone causes the decision to be FALSE , similarly for B independently of A

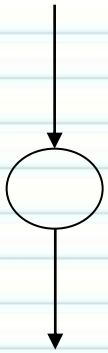
White-Box Testing Techniques

- Basis Path Testing
  - Flow Graph Notation
  - Cyclomatic Complexity
  - Deriving Test Cases
- Condition Testing
- Data Flow Testing
- Loop Testing
- Symbolic Execution

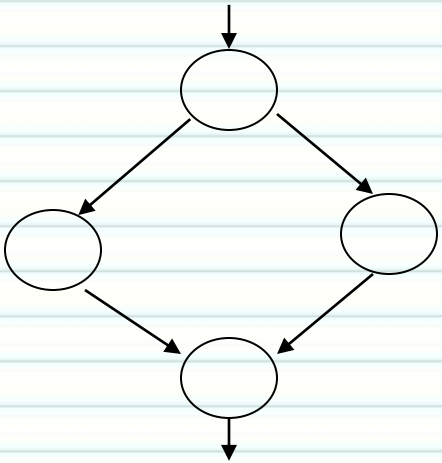
Basis Path Testing Steps

- Construct a flow graph.
- Compute cyclomatic complexity.
- Determine basis paths.
- Check to ensure that the number of basis paths equals to the cyclomatic complexity.
- Derive test cases to exercise the basis paths according to the required coverage criteria.

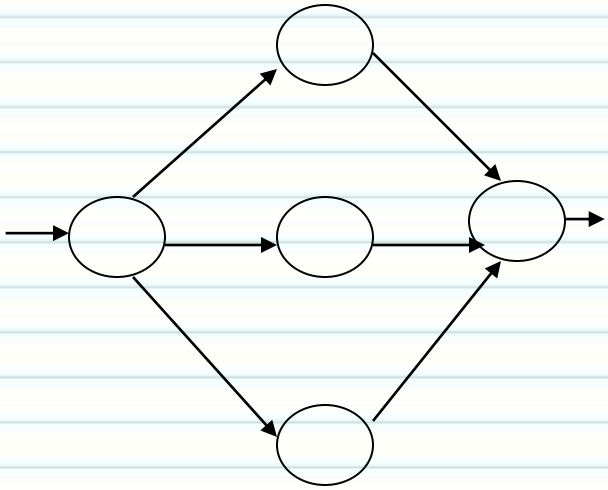
Flow Graph Notations



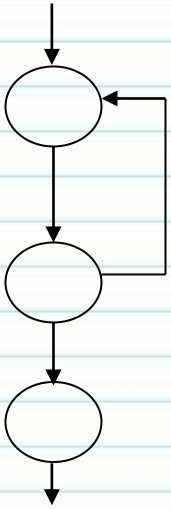
sequential



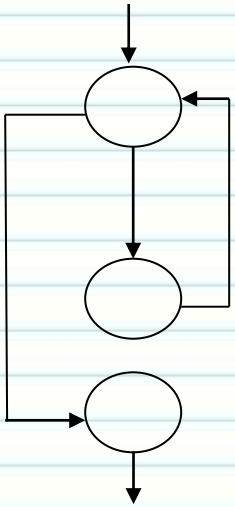
if-then-else



case

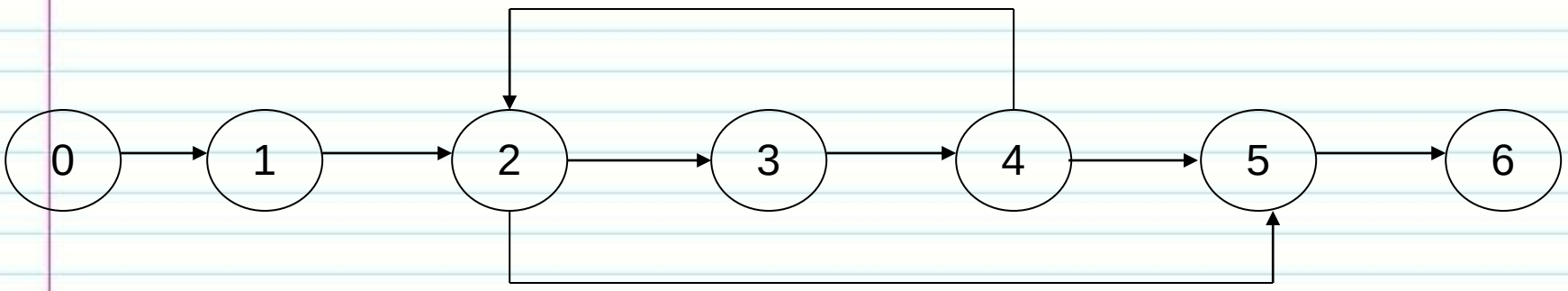


until

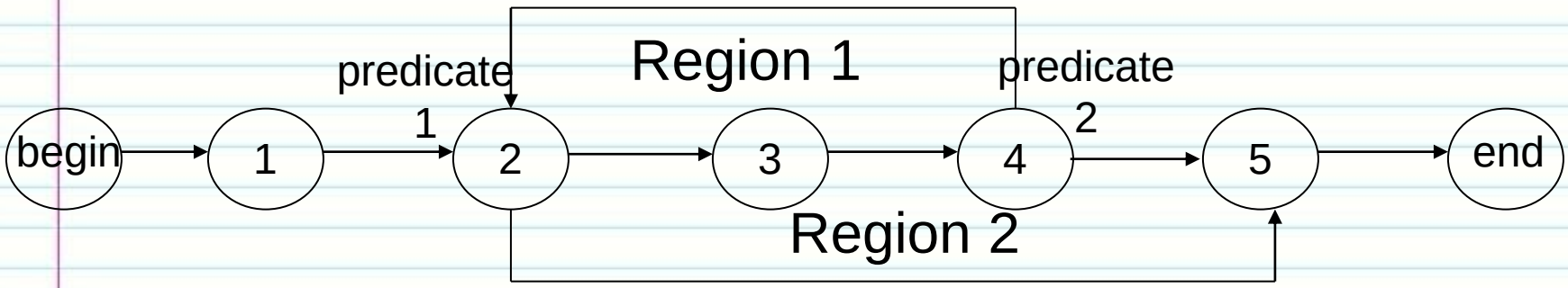


while

An Example Flow Graph



Cyclomatic Complexity



Three ways to compute cyclomatic complexity:

- Number of closed regions + 1
- Number of atomic binary predicates + 1
- Number of Edges - Number of Nodes + 2

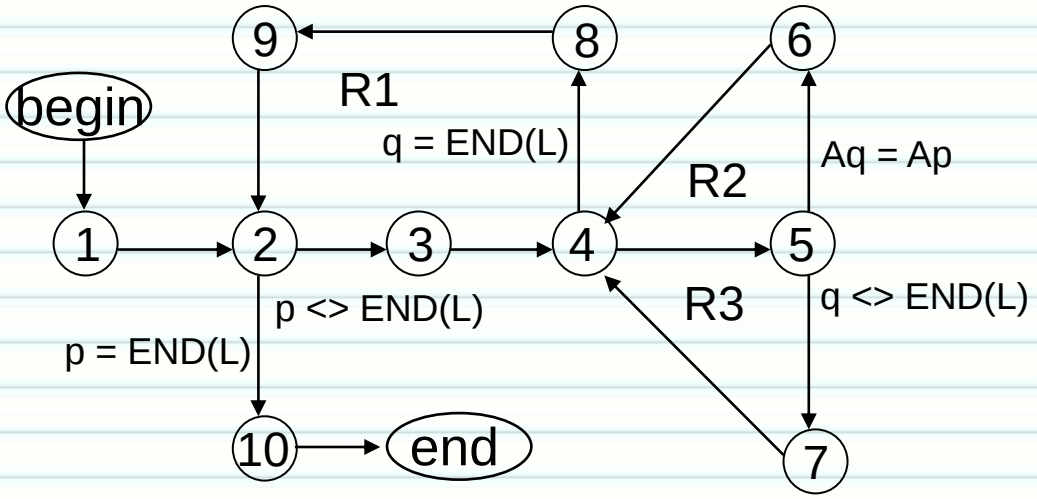
The cyclomatic complexity is 2+1=3.

Determining Basis Paths

- The total number of test cases needed to exercise all basis paths equals the cyclomatic complexity
- A basis path is
  - A path from the begin node to the end node AND
  - Traverses a cycle either zero times or exactly once.
- The basis paths are:
  - begin, 1, 2, 3, 4, 2, 5, end
  - begin, 1, 2, 3, 4, 5, end
  - begin, 1, 2, 5, end

Example

```
Procedure purge (var L:list)
      var p: …
(1)     begin p:= FIRST(L);
(2)     while P <> END(L) do
(3)     begin q:= next(p,L);
(4)       while q <> END(L) do
(5)           if Aq = Ap then
(6)              delete (Aq, L)
(7)           else q:= next(q,L);
(8)       end
(9)       p := next(p,L)
(10)    end;
```



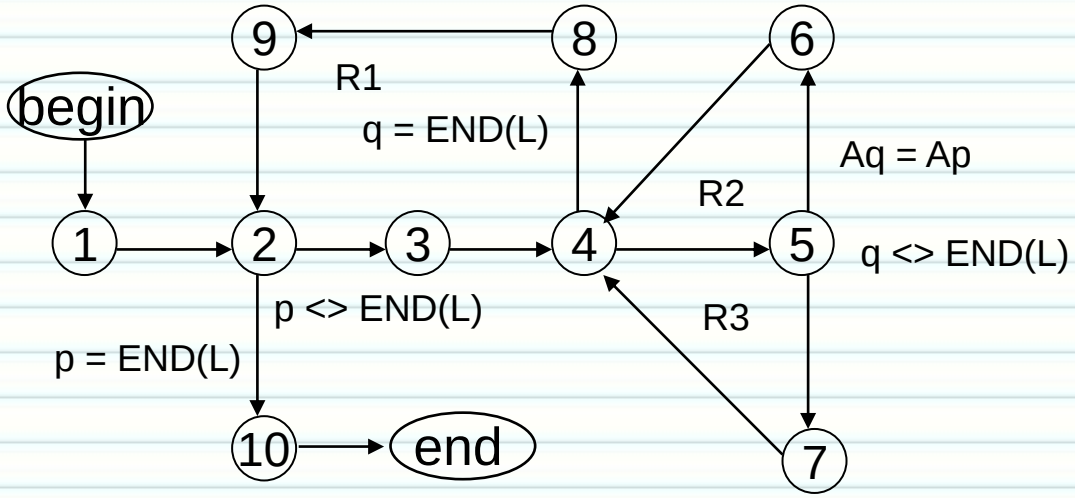
Path 1 (R1): 1-2-3-4-8-9-2-10

Path 2 (R2): 1-2-3-4-5-6-4-8-9-2-10

Path 3 (R3): 1-2-3-4-5-7-4-8-9-2-10

Path 4: 1-2-10

Example



Path 1 (R1):	1-2-3-4-8-9-2-10	L=(Ap)	
Path 2 (R2):	1-2-3-4-5-6-4-8-9-2-10	L=(Ap Aq)	Ap = Aq
Path 3 (R3):	1-2-3-4-5-7-4-8-9-2-10	L=(Ap Aq)	Ap <> Aq
Path 4:	1-2-10	L=( )	

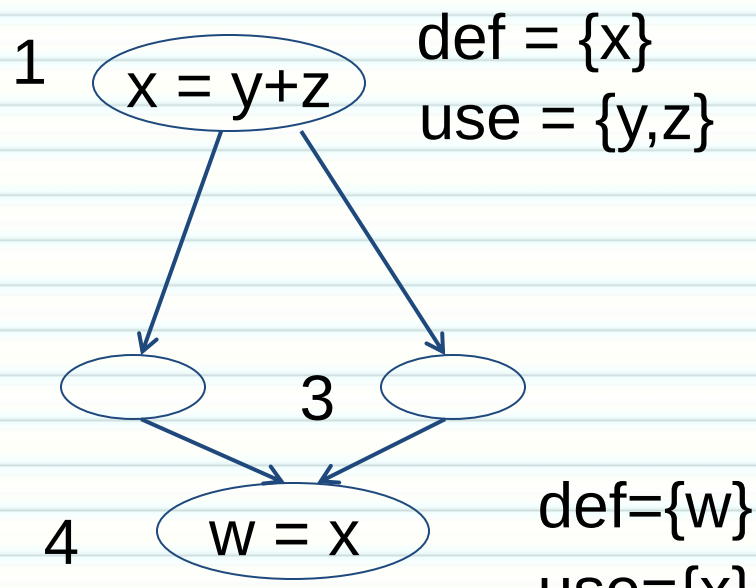
Data Flow Testing

- Select test paths of a program according to the locations of definitions and uses of variables in the program, then test these define-use chains.
- Steps
  - Determine define-use chains for each variables
  - Determine testing criteria, ex. all-defines, all-uses, …
  - Design test cases to satisfy the test criteria

Data Flow Testing

- The use of a variable can be either a c-use (computation use) or a p-use (predicate use)
- A def-use chain of a variable is a path from the definition to the use of the variable without any intervening redefinitions (i.e., definition-clear)
  - for c-uses, the def-use chains are paths from the statement containing the definition to the statement containing the computation use
  - for p-uses, the def-use chains are paths from the statement containing the definition to two successors of the statement containing the predicate use
- Test paths of a program are then selected based on the def-use chains of variables and test data adequacy criteria (e.g., all-use)

Data Flow Testing



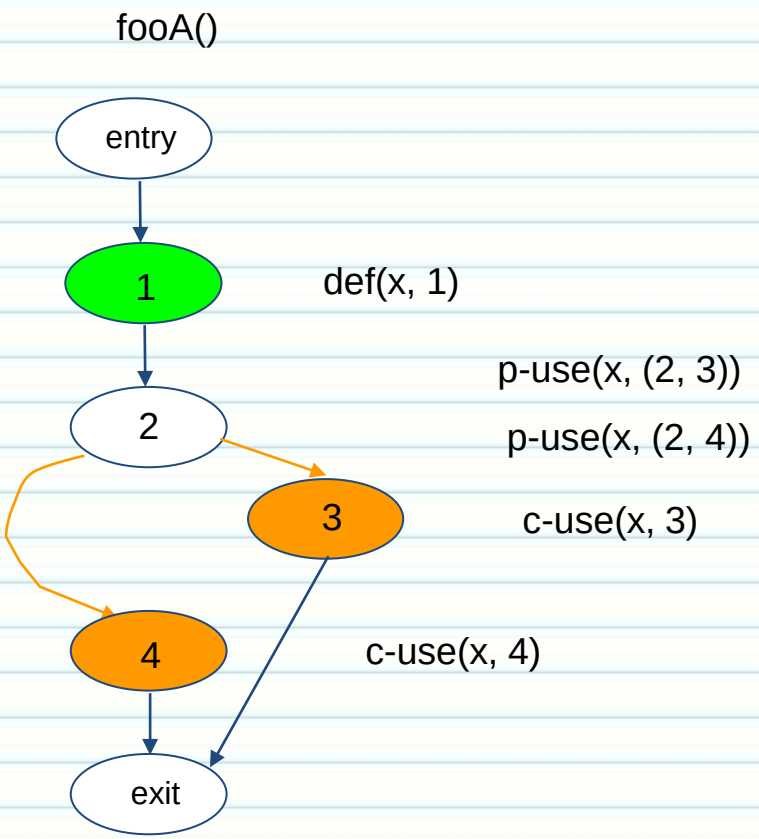
def-clear path
(with respect to variable *x*)
no redefinition of
x along the path

du-association (1, 4, x)

du-path (1, 2, 4) (1, 3, 4)
(with respect to variable *x*)

Data Flow Testing

```
 fooA(){
1  x = read();
2  if (x > 0)
3     print x-1;
   else
4     print x;
 }
```



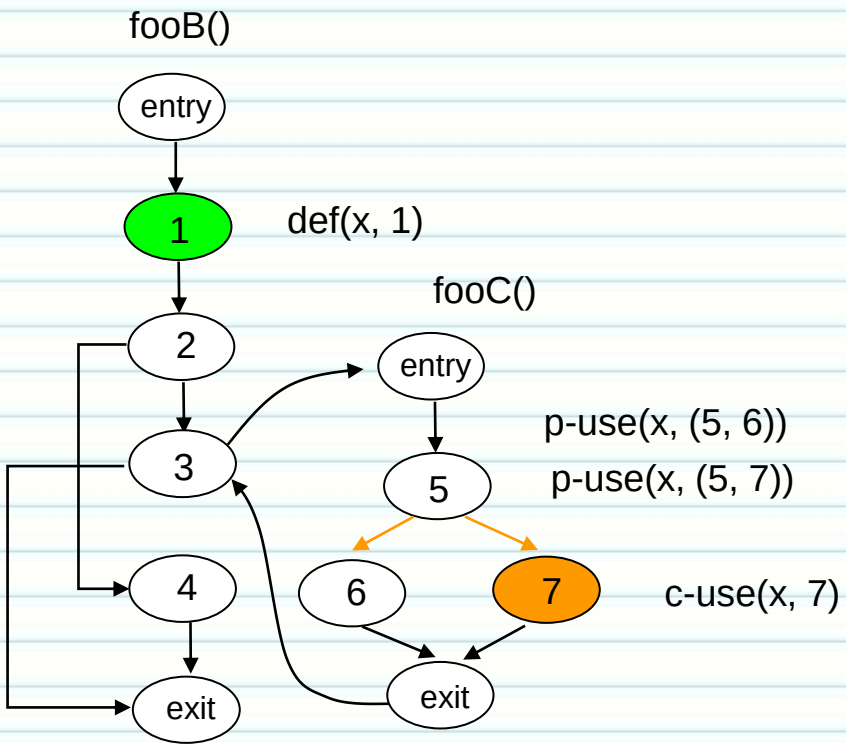
Intraprocedural def-use chains

<1, (2, 3)>, <1, (2, 4)>, <1, 3>, <1, 4>

Data Flow Testing

```
 fooB(){
1  x = read();
2  if (x > 0)
3     x=fooC(x);
   else
4     print x;
 }
```

```
  fooC(x){
5 if (x > 5)
6    return 5;
   else
7    return x;
 }
```



Interprocedural def-use chains

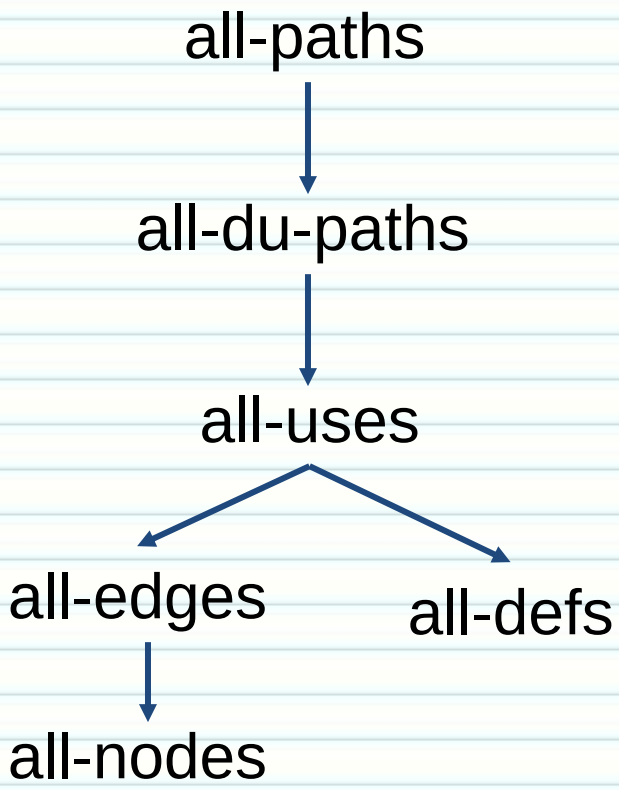
<1, (5, 6)>, <1, (5, 7)>, <1, 7>

Data Flow Testing Criteria

all-defs Cover each definition to some use

all-uses Cover each definition to each use

all-du-paths Cover all du-paths from each definition to each use



Data Flow Testing

- A variable can be in one of the following states:
  - d: defined, created, initialized, etc.
  - k: killed, undefined, released
  - u: used
    - c: computation use
    - p: predicate use

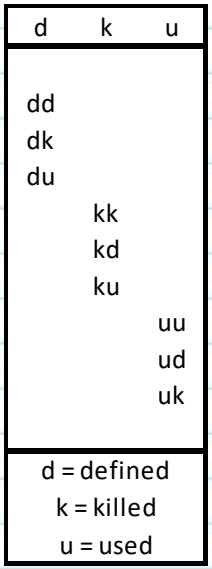
Examples

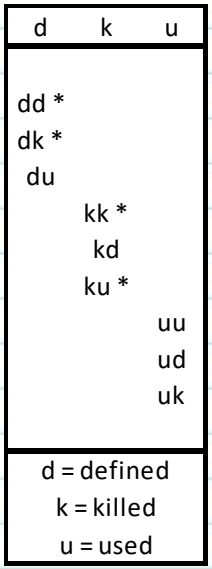
- x=y+1; x is defined, y is c-use
- x=new File(); x is defined
- x=null; x is killed
- if (x>=0) … x is p-use

Data Flow Anomalies

- There are 9 possible two letter combinations of d, k and u:
  - buggy combination
    1. ku: using an undefined variable, a bug
  - possibly anomaly combinations
    2. dd: why define the variable twice? . . . suspicious
    3. dk: why define the variable without using it . . . probably a bug
    4. kk: why kill a variable twice? . . . suspicious
  - normal cases

    5-9. du, kd, ud, uk and uu





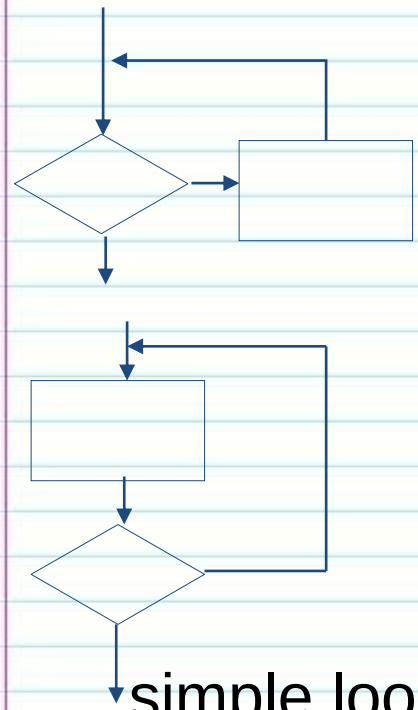
Data Flow Anomalies

- Let "-" denote nothing of interest (d,k,u) occurs prior or after an action. There are 6 single letter situations:
  - possibly anomalous situations
    - -k: why kill a variable without doing anything with it?
    - -u: possibly anomalous unless the variable is global and has been previously defined
    - d-: possibly anomalous unless it is global definition or defined for other routines
    - u-: may signify that dynamically allocated storage is not returned (memory leak)
  - normal situations: -d, k-

Loop Testing

- Focuses on the validity of loop constructs.
- Four different classes of loops:
  - Simple loop: a single loop
  - Nested loops: a loop within another loop
  - Concatenated loops: one loop after another loop
  - Unstructured loops: complex nested and concatenated loops.

Testing Simple Loops

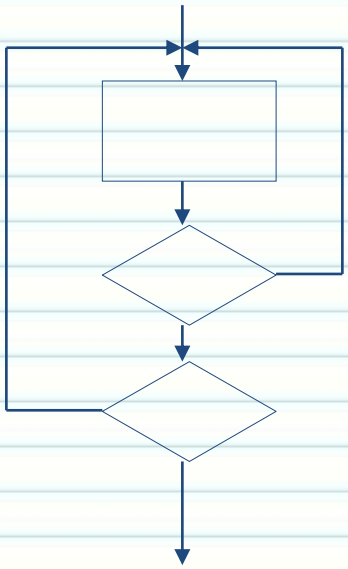


simple loops

Boundary value analysis applied to test:

1. Skip the loop entirely.
2. Only one pass through the loop.
3. Two passes through the loop.
4. *m* passes through the loop where $m < n$.
5. n -1, n, n + 1 passes through the loop.

Testing Nested Loops

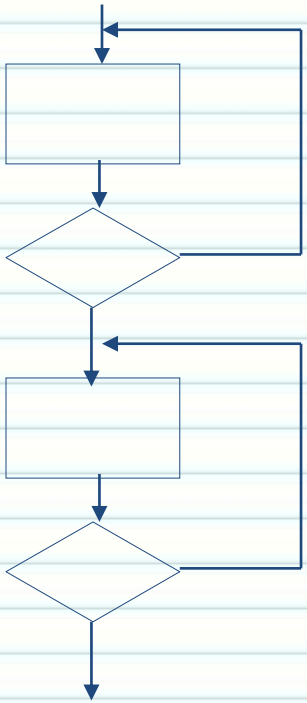


Nested Loops

Beizer's approach reduces the number of tests:

1. Start at the innermost loop. Set all other loops to minimum values.
2. Conduct simple loop tests for the innermost loop:
   - Holding the outer loops at their minimum iteration parameter.
   - Add other tests for out-of-range or excluded values.
3. Work outward, conducting tests for the next loop:
   - Keeping all other outer loops at minimum values, and
   - Other nested loops to "typical" values.
4. Continue until all loops have been tested.

Testing Concatenated Loops



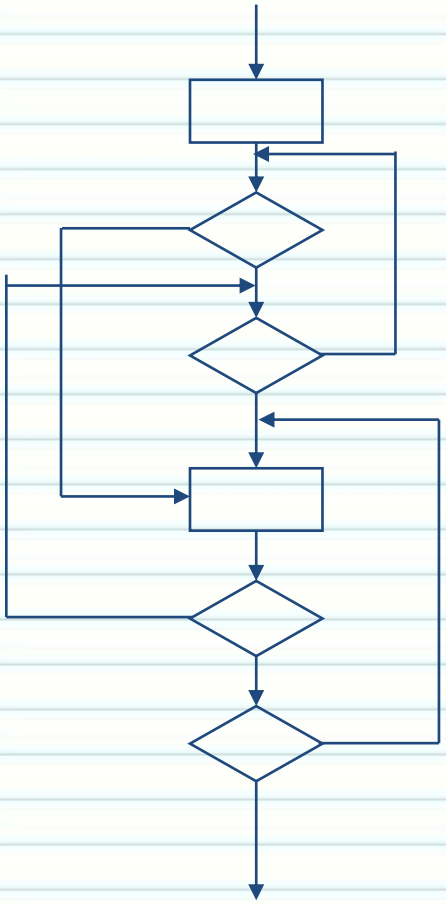
Concatenated Loops

Consider two cases:

- If each loop is independent of the other, then test them as simple loops.

- If two loops are concatenated and the loop counter for loop 1 is used as the initial value for loop 2, then test them as nested loops.

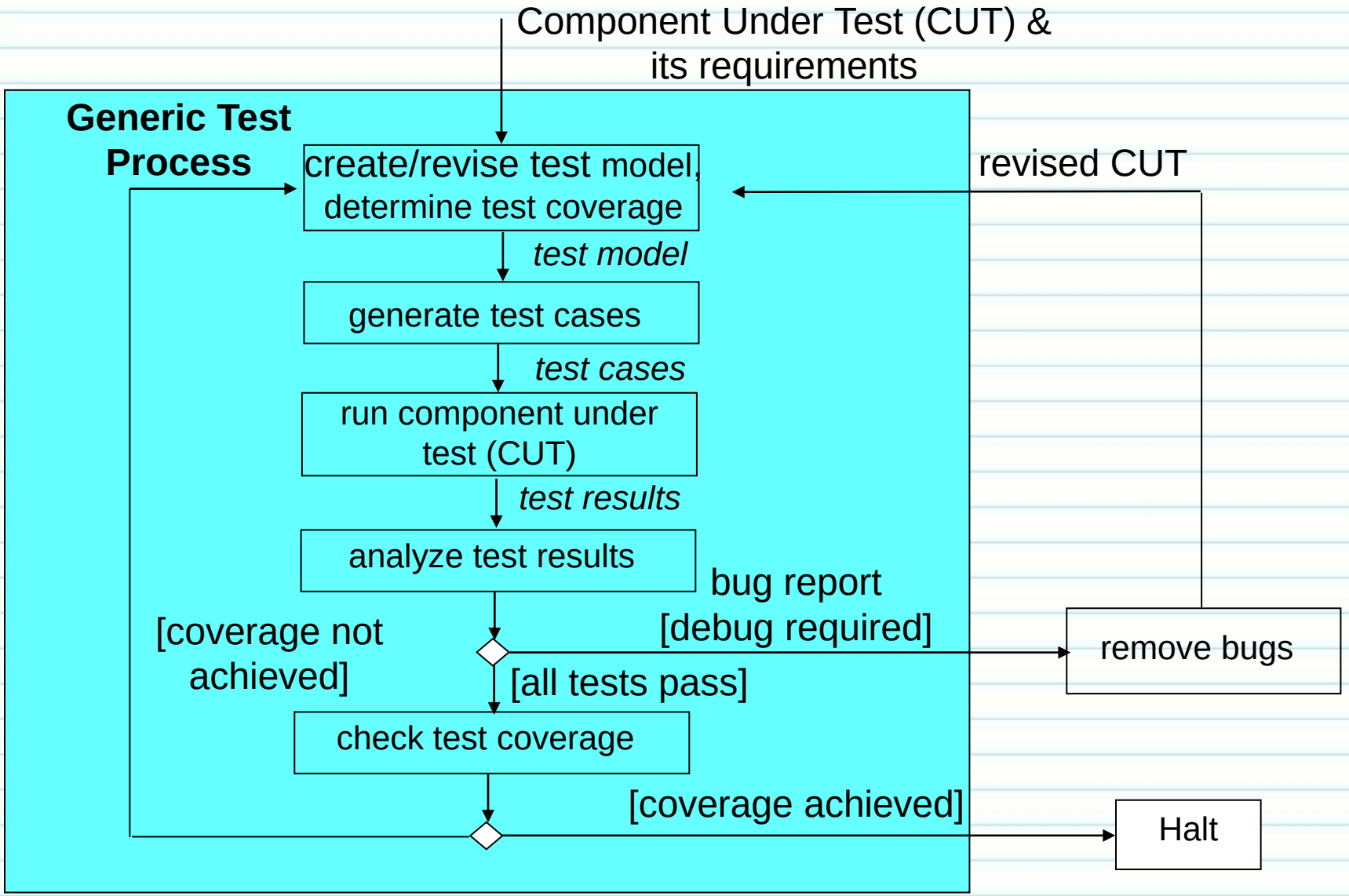
Testing Unstructured Loops



Unstructured Loop

Redesign the algorithm to remove loops.

A Generic Testing Process



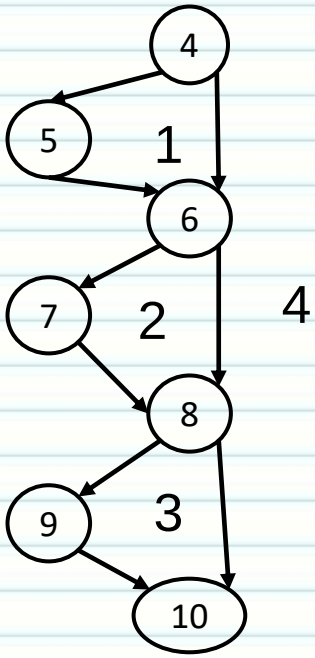
Demonstrate Code Based Coverage

```
1   public static int returnInput(boolean conditiona, boolean conditionb, boolean conditionc) {
2           int x=0;
3
4           if (conditiona)
5            x++;
6           if (conditionb)
7            x++;
8           if (conditionc)
9            x++;
10
11          return x;
12  }
```

1. Describe what the code seems to be doing
2. Draw the reduced CFG
3. Determine the Cyclomatic complexity
4. Develop the basis paths using the Cyclomatic complexity

Demonstrate Code Based Coverage (Con't)

1. The code is counting the number of the three input conditions that have the boolean value true
2. Basis paths are:
   a) 4, 5, 6, 7, 8, 9, 10
   b) 4, 6, 7, 8, 9, 10
   c) 4, 6, 8, 9, 10
   d) 4, 6, 8, 10



Coverage goal	Test Inputs	Number of combinations	What's measured?
Statement	TTT	1	Each node is reached
Branch	TTT,FFF	2	Each edge is reached
Decision	TTT,FFF	2	Each edge is reached
Condition	TTT,FFF	2	Decisions and Conditions are identical here
Basis Path	TTT,FTT,FFT,FFF	4	Each edge is reached by changing one decision at a time
Path	Truth table for 3 inputs	8	All possible paths

Note:
100 decision coverage implies 100% statement coverage.
100 % statement coverage does not imply 100 % decision coverage.

Use Case Based Testing

- Agile Approach:

  Step 1. Prioritize the use cases in the requirement-use case traceability matrix

  Step 2. Generate test scenarios from the expanded use cases

  Step 3. Identify test cases

  Step 4. Generate test data (values)

  Step 5. Generate test scripts

- Why do we prioritize use cases?
  - We may not have needed resources or time
  - Prioritizing ensures that high-priority use cases are tested
- The Requirement-Use Case Traceability Matrix (RUCTM)
  - Rows are requirements and columns are use cases
  - An entry is checked if the use case realizes the requirement
  - The bottom row sums up the priority of each use case

Requirement-Use Case Traceability Matrix

	Priority Weight	UC1	UC2	UC3	UC4	UC5	UC6
R1	3	X	X				
R2	2					X	
R3	2	X					
R4	1		X	X			
R5	1				X		X
R6	1		X			X	
Score		5	5	1	1	3	1

Generating Use Case Scenarios

- A scenario is an instance of a use case
- A scenario is a concrete example showing how a user would use the system
- A use case has a primary scenario (the normal case) and a number of secondary scenarios (the rare or exceptional cases)
- This step should generate a sufficient set of scenarios for testing the use case
  - Normal use cases
  - Abnormal use cases

Example: Register for Company Events

ID: UC5

Name: Register for Event

Actor: Event Participant

Precondition:

Participant has an account on the system and participant is not already registered for the event.

Primary Scenario: (see next slide)

Register for Event: Primary Scenario

Actor: Participant	System: Web App
	0. System Displays the home page
1. TUCBW the participant clicks the "Register" button on homepage.	2. Systems ask the participant to enter email and password.
3. Participant enter email and password.	4. System checks that login is correct and displays list of events.
5. Participant selects an event.	6. System asks for event related info.
7. Participant enters event related info.	8. System verifies info and displays a confirmation.
9. TUCEW Participant acknowledges by clicking the OK button (confirmation dialog)	

Identifying Secondary Scenarios/Exceptions

Base test cases on user input (we see some partitions here):

User Input	Normal Case	Abnormal Cases
email	valid	invalid
password	valid	invalid
selected event	still open	• registration closed • duplicate registration
event related info	valid	invalid

Other exceptions:

- The user quit (or abandoned the web page) before completing registration

Identifying Test Cases

- Identify test cases from the scenarios using a test case matrix:
  - The rows list the test cases identified
  - The columns list the scenarios, the user input variables and system state variables, and the expected outcome for each scenario
  - We're going to apply techniques as before to identify test cases such as:
    - Boundary Value Analysis
    - Equivalence classes
    - Decision tables

Use Case Based Test Case Generation

		Inputs					
Test Case ID	**Scenario**	**Email ID**	**Password**	**Registered**	**Event Info**	**Event Open**	**Expected Result**
TC1	Successful Registration	V	V	V	V	V	Display confirma-tion
TC2	User Not Found	I	NA	NA	NA	NA	Error msg
TC3	Invalid Info	V	V	NA	I	NA	Error msg
TC4	User Quits	V	V	NA	NA	NA	Back to login
TC5	Registration Closed	V	V	NA	NA	I	Error msg
TC6	Duplicate Registration	V	V	I	NA	NA	Error msg

Identifying Test Data Values

Test Case ID	Scenario	Email ID	Pass-Word	Registered	Event Info	Event Open	Expected Result
TC1	Successful Registration	lee@ca.com	Lee123	No	Yes	Yes	Display confirma-tion
TC2	User Not Found	unknow@ca.com	NA	NA	NA	NA	Error msg
TC3	Invalid Info	lee@ca.com	Lee123	NA	I	NA	Error msg
TC4	User Quits	lee@ca.com	Lee123	NA	NA	NA	Back to login
TC5	System Unavailable	lee@ca.com	Lee123	NA	NA	NA	Error msg
TC6	Registration Closed	lee@ca.com	Lee123	NA	NA	No	Error msg
TC7	Duplicate Registration	lee@ca.com	Lee123	Yes	NA	NA	Error msg