

# **COSC 310:**

# **Software Engineering**

Dr. Bowen Hui

University of British Columbia Okanagan

[bowen.hui@ubc.ca](mailto:bowen.hui@ubc.ca)

# Testing

- what's the purpose of testing?
- what is the process involved?
- why might things go wrong after you do testing?

# Testing

- what's the purpose of testing?
  - show that a program works as intended
  - find and fix all defects before deployment
- what is the process involved?
  - different levels of test cases using artificial data
- why might things go wrong after you do testing?

# Related Terminology

- Testing can only show the presence of errors, not their absence
- **validation**: are we building the right product?
  - ensures software meets customer expectations
- **verification**: are we building the product right?
  - ensures software meets all requirements
- V&V establishes software's fit for purpose

# V&V

- validation
  - e.g., can test user-friendliness, usability, often under real use environments
  - example activities:
    - usability testing
    - informal user feedback
- verification
  - e.g., can test timeliness, interoperability
  - example activities:
    - testing
    - inspections
    - static analysis

# Elevator Response Example

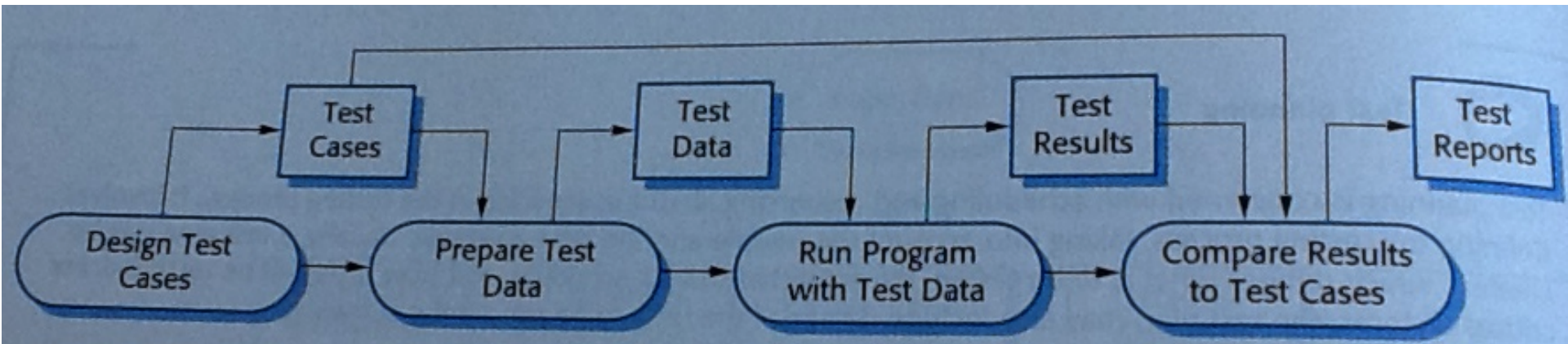
- "if user presses button for floor x, then elevator must arrive at floor x within 30 seconds"
  - requirement specifies a precise quantity that is measurable
  - requires verification
- "if user presses button for floor x, then elevator must arrive at floor x soon"
  - requirement specifies a subjective quantity
  - requires validation

# Why Test?

- software is never correct
  - 80% errors found often traced back to 20% of components
- ability to measure and improve the quality of product
- goals of testing and analysis
  - find faults
  - formal proof of software properties (e.g., flowgraphs)
  - provide confidence
  - compromise between software accuracy, generality, complexity

# Traditional Testing Process

- used in plan-driven development





# Software Test Plan

- document to explain testing approach
- exhaustive testing is not possible
- serves as link between development and testing
  - items to be tested
  - resource requirements (people, hardware, software)
  - test schedule
  - reporting requirements
  - outcomes (evaluation criteria, level of acceptable risk)
- all test cases should be traceable to requirements

# Elements of a Test Case

- reference number
- description
- preconditions
- input
- expected output
- outcome

# Example Test Case

TC#: S-221

TC Description: This test is checks the system response to an invalid input selection.

TC Precondition: Go to screen  
Flight\_Management

TC Input: Enter <F7>

TC Expected Output: Error message: "Invalid input, Enter "1" to "4" or ESC"

TC Outcome: Pass, Fail

# Development Testing

- **unit testing**
  - white box approaches
  - black box approaches
- **integration testing**
  - non-incremental (big bang)
  - incremental (top-down, bottom-up, sandwich)
- **system testing**
  - performed using system level input and outputs on target platform
  - answers: "can we ship the product yet?"
- **regression testing**
  - ensure changes don't cause unintended effects elsewhere

# Testing Order

- which testing approach first?

# Testing Order

- which testing approach first?
- from small to large:
  - individual units (white box testing)
  - units integrated as subsystems (black box testing)
  - subsystems integrated into system (integration testing)
  - system placed in real environment (system testing)
  - system tests are preserved to be re-run when changes are made (regression testing)

# Unit Testing Considerations

- Interface is tested to ensure information properly flows into and out of the unit
- Examines local data structure
- Tests boundary conditions
- Tests all independent paths
  - Ensures statement coverage
  - Ensures branch coverage
- Tests all error handling paths

# Black-Box Testing Example

- Code description:
  - Given two integers, return the larger one. Return the first integer if tie.
- Test case description and input:
  - Case  $n1 > n2$ :  $n1=2, n2=1$
  - Case  $n1 < n2$ :  $n1=1, n2=2$
  - Case  $n1 = n2$ :  $n1=2, n2=2$



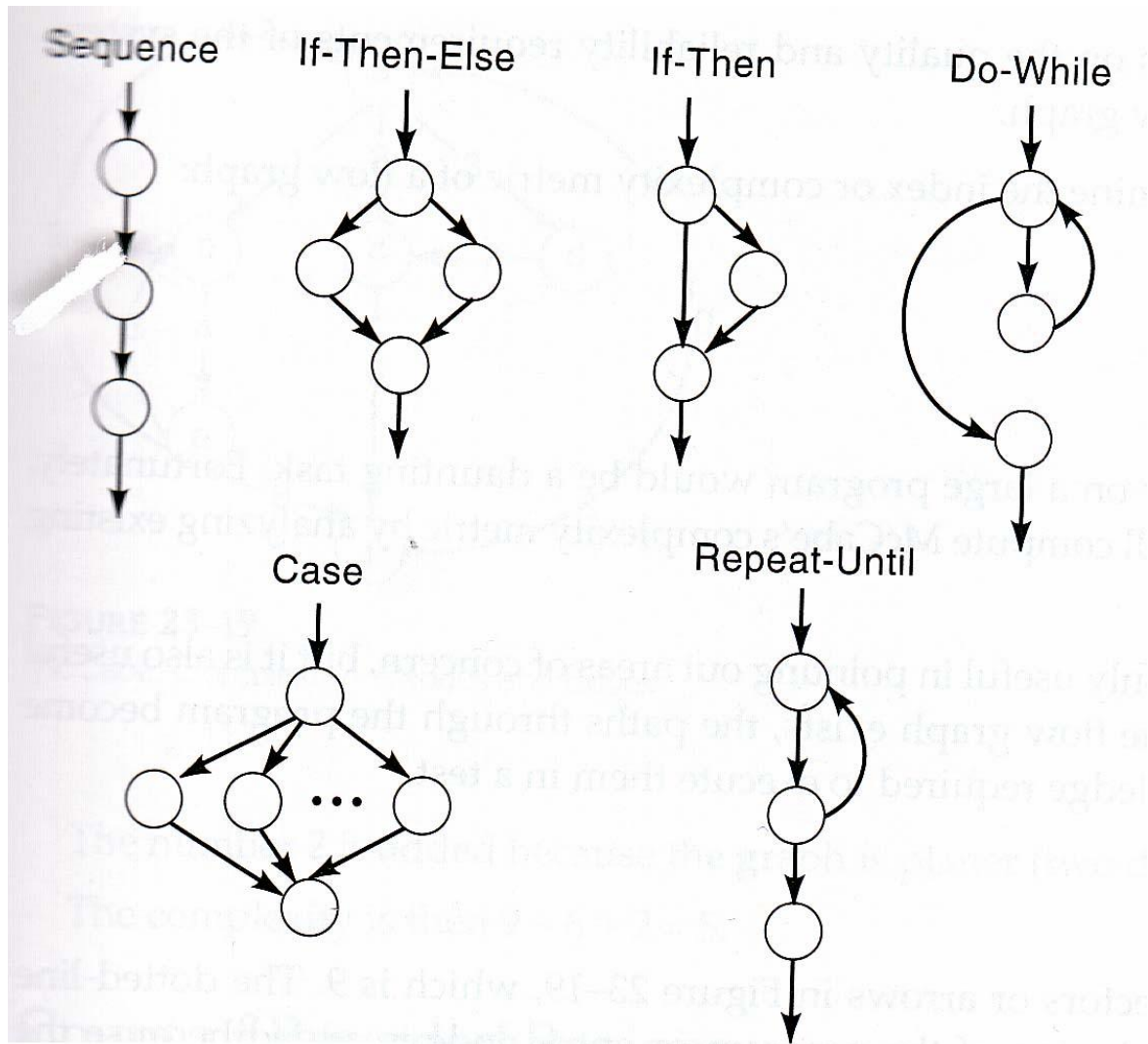
# Black-Box (Behaviour) Testing

- Subsystems can be subject to:
  - Boundary value testing
  - Equivalence partitioning
    - Useful for improving testing efficiency
    - Input domain partitioned into finite number of equivalence classes
    - A representative test case used for each class
    - E.g. condition  $x > y$  has two equivalence classes for inputs:
      1. Values of  $x$  and  $y$  such that  $x > y$
      2. Values of  $x$  and  $y$  such that  $x \leq y$

# White-box Testing Strategies

- Statement coverage
  - Tests each statement
- Branch coverage
  - Tests each possible outcome from a branch
- (Multiple) Condition coverage
  - Tests each combination of conditions in decision nodes
- Loop coverage
  - Executes each loop zero, one, and more than one time
- Path coverage
  - Simple paths (contains no repeated nodes, except start/end nodes may be the same)
  - All paths
  - Basis paths (calculated by cyclomatic complexity)

# Flow Graph Representation: Basic Constructs



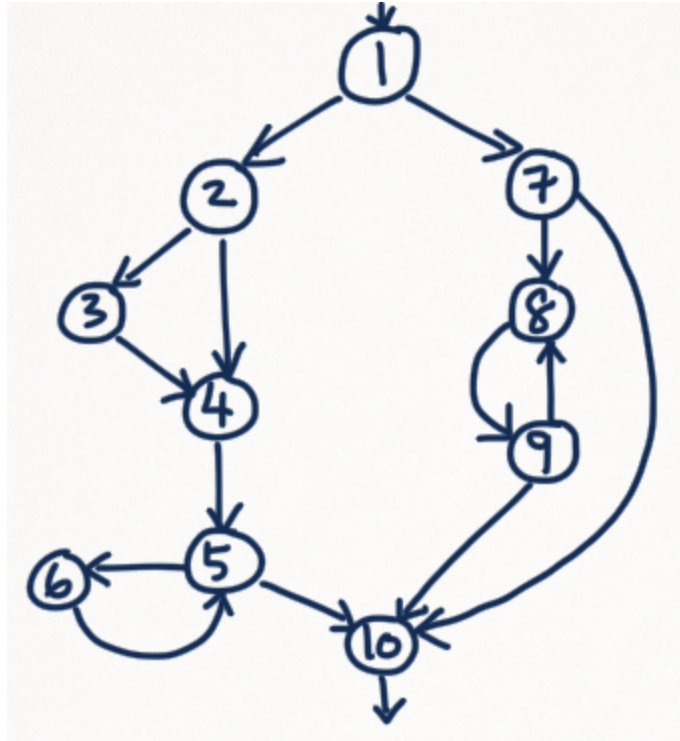
# Example Source Code and Corresponding Flow Graph

```
euclid( int m, int n )
{
    // assuming m and n are both greater than 0
    // return their gcd
    // enforce m >= n for efficiency
1.  int r;
2.  if( n > m )
    {
3.      r = m;
4.      m = n;
5.      n = r;
    }
6.  r = m % n;
7.  while( r != 0 )
    {
8.      m = n;
9.      n = r;
10.     r = m % n;
    }
11. return n;
}
```

Graph:

# Example

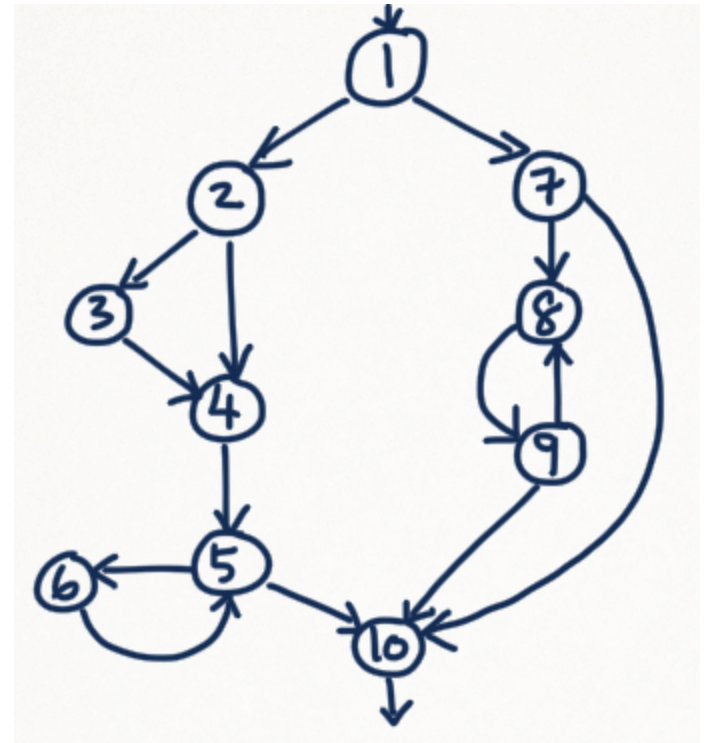
- Recover the code structure for this flowgraph:



- How many if statements are here?
- Which loops are shown here?

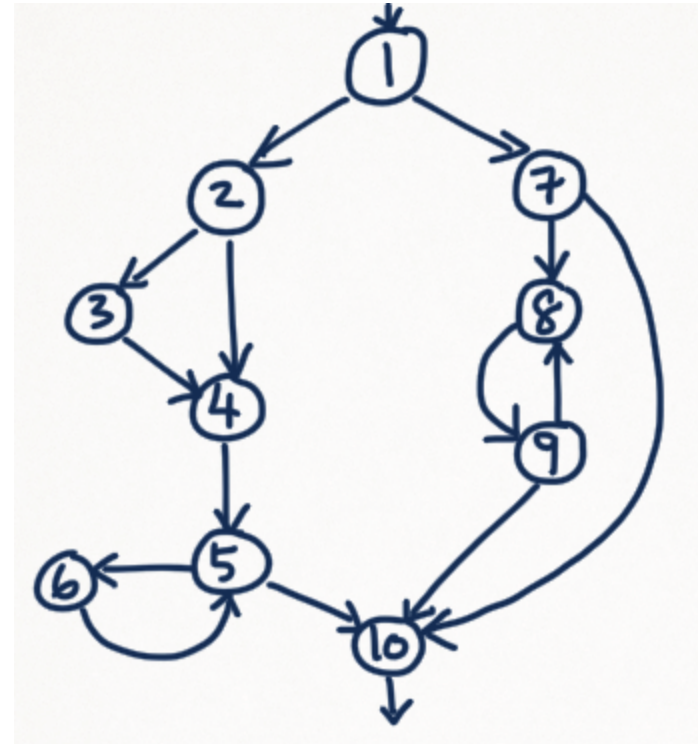
# Corresponding Code Structure

Code:



# Statement and Branch Coverage

- Statement coverage
  - Min # test cases = 2
  - Paths:
    - 
    -
- Branch coverage
  - Min # test cases = 4
  - Paths:
    - 
    - 
    - 
    -



# Multiple Condition Coverage

- Examples of complex conditions and test cases:

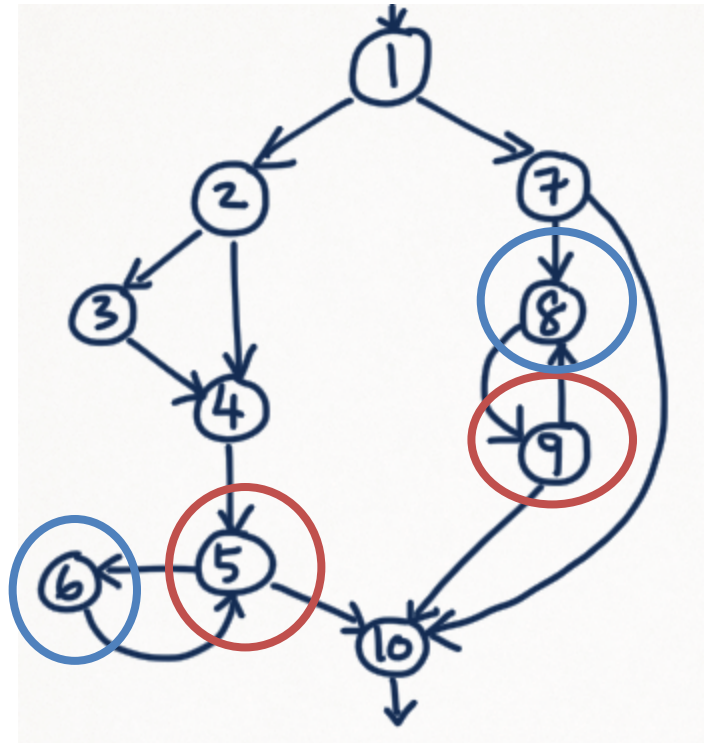
	a	&&	b	&&	(c		(d	&&	e)
1.	F		-		-		-		-
2.	T		F		-		-		-
3.	T		T		F		F		-
4.	T		T		F		T		F
5.	T		T		F		T		T
6.	T		T		T		-		-

	((a		b)	&&	(c		d)	&&	e
1.	F		F		-		-		-
2.	F		T		F		F		-
3.	F		T		F		T		F
4.	F		T		F		T		T
5.	F		T		T		-		F
6.	F		T		T		-		T
7.	T		-		F		F		-
8.	T		-		F		T		F
9.	T		-		F		T		T
10.	T		-		T		-		F
11.	T		-		T		-		T



# Loop Coverage

- Loops:
  - Do-while
    - **Condition** at node 5
    - **Body** at node 6
  - Repeat-until
    - **Condition** at node 9
    - **Body** at node 8



- Ensure loop bodies are executed zero, one, more than one time

# Loop Coverage

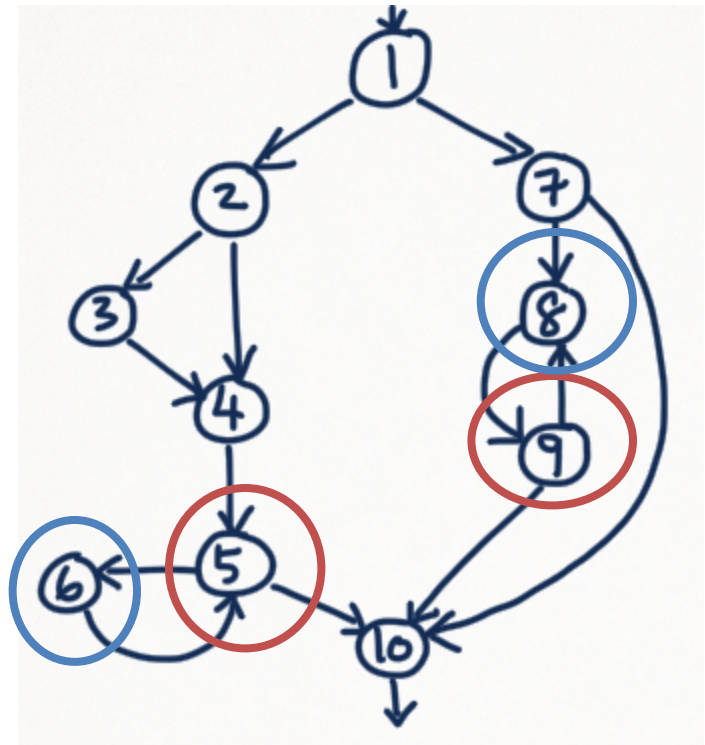
- Min # test cases = 6

- Do-while paths:

- Zero:
- One:
- Two+:

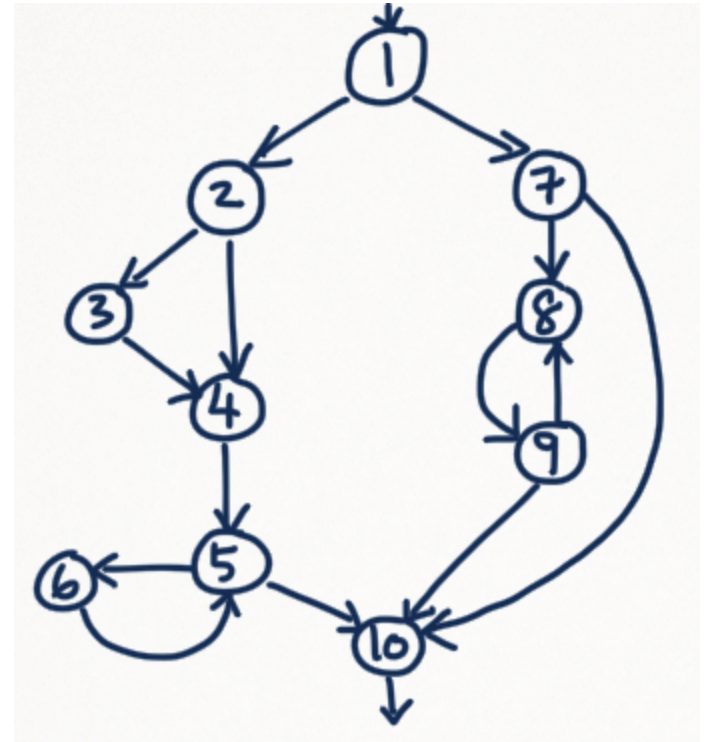
- Repeat-until paths:

- Zero:
- One:
- Two+:



# Simple Path Coverage

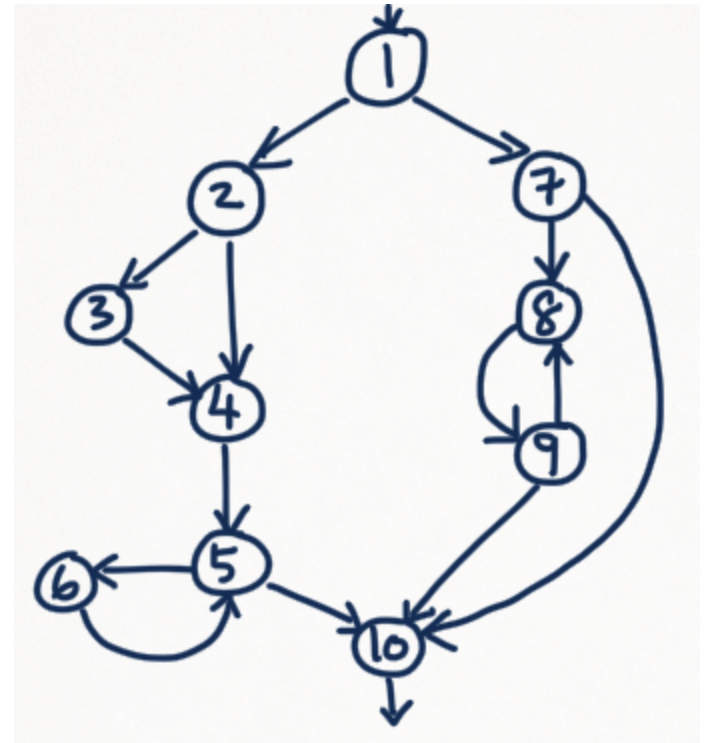
- Min # test cases = 4
- Paths:
  - (1,2,3,4,5,10)
  - (1,2,3,4,5,6,5,10)
  - (1,2,4,5,10)
  - (1,2,4,5,6,5,10)
  - (1,7,10)
  - (1,7,8,9,10)



- Which two paths are not *simple paths*?

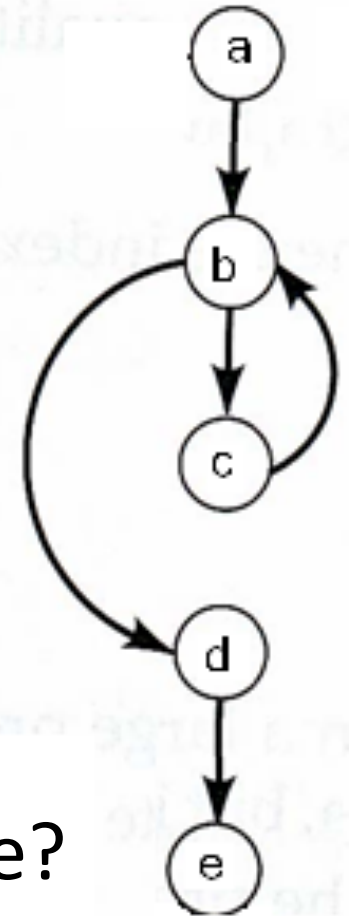
# All Paths Coverage

- Min # test cases = infinity
- Paths:
  - 
  - 
  - 
  -



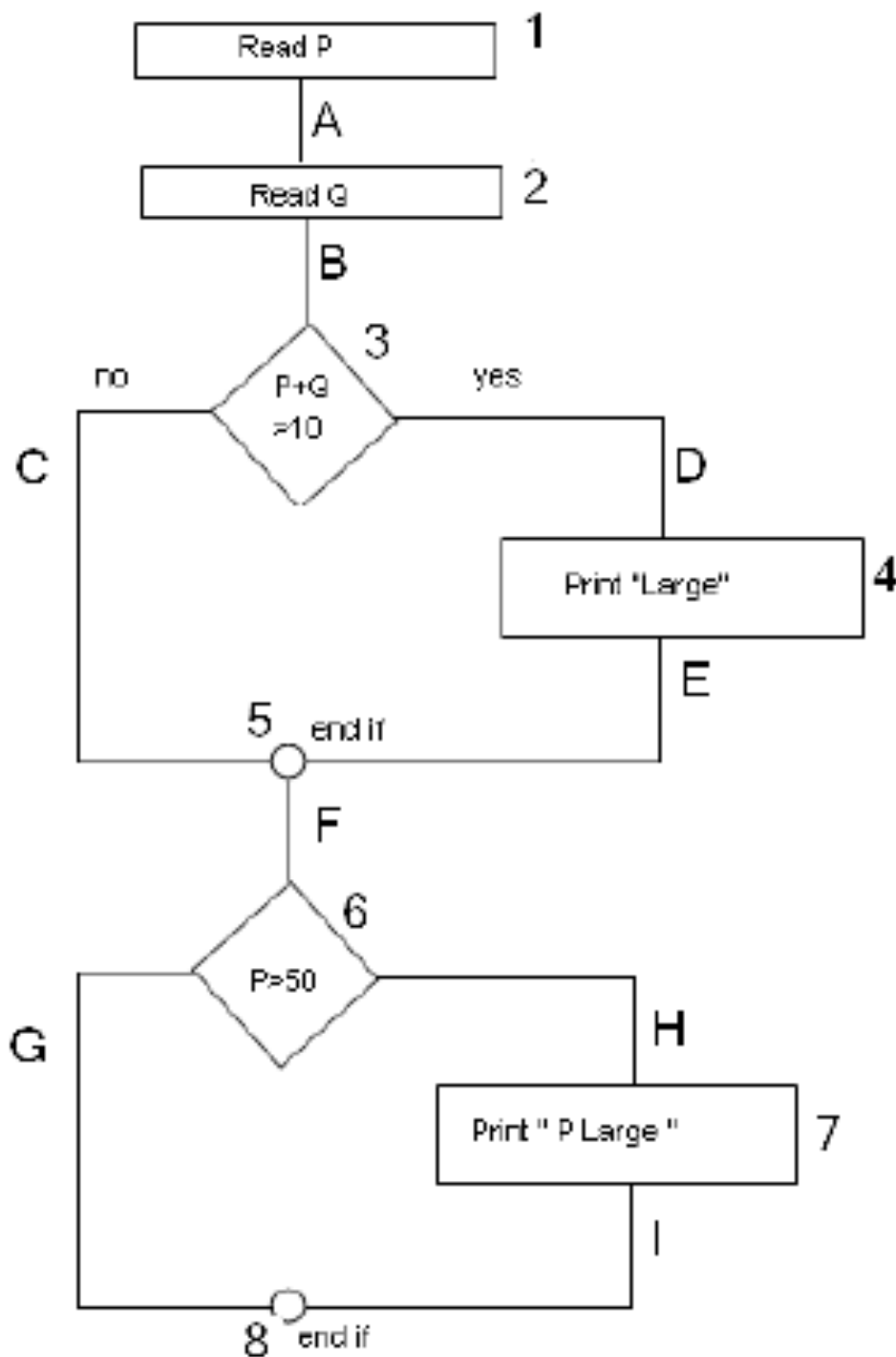
# Example

- Possible paths:
  - a,b,d,e
  - a,b,c,b,d,e
  - a,b,c,b,c,b,d,e
  - a,b,c,b,c,b,c,b,d,e
  - ...
- Which path gives full statement coverage?
- Which path gives full branch coverage?



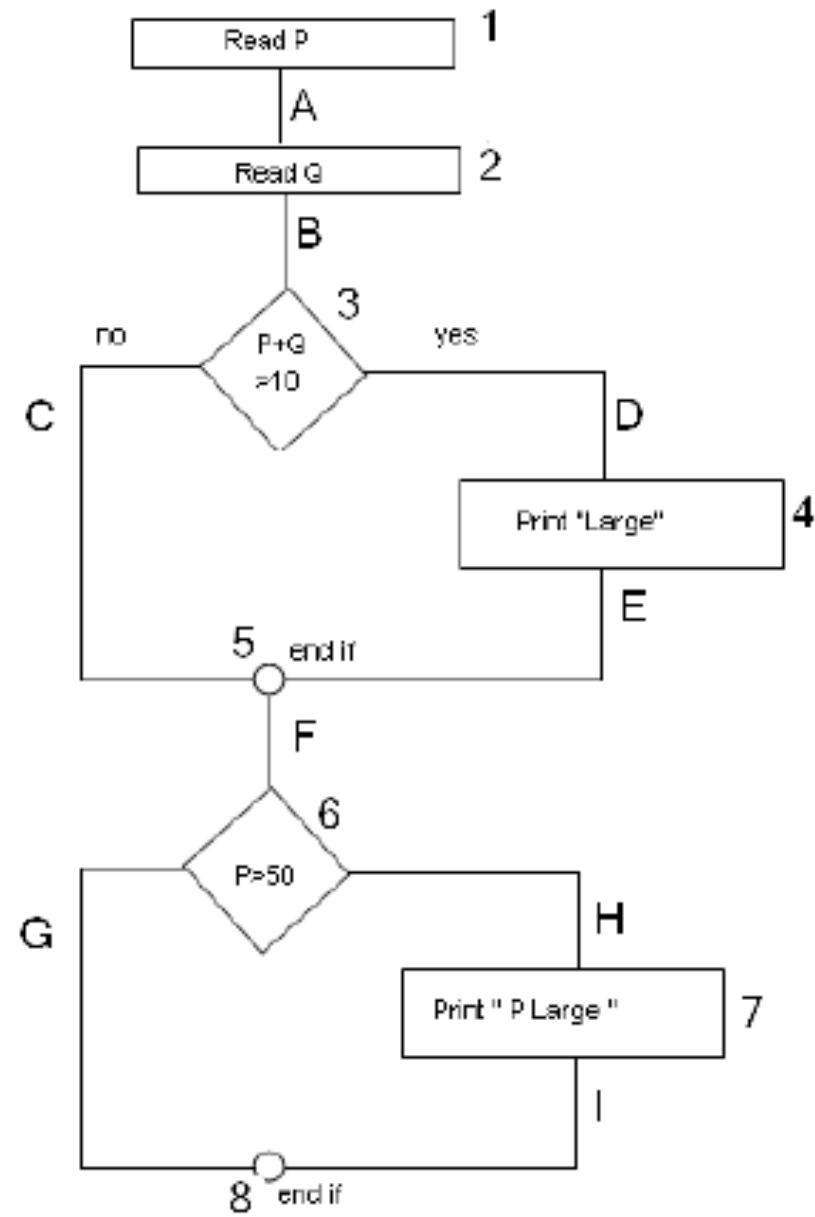
Example:

```
Read P  
Read Q  
IF P+Q > 100 THEN  
Print "Large"  
ENDIF  
If P > 50 THEN  
Print "P Large"  
ENDIF
```



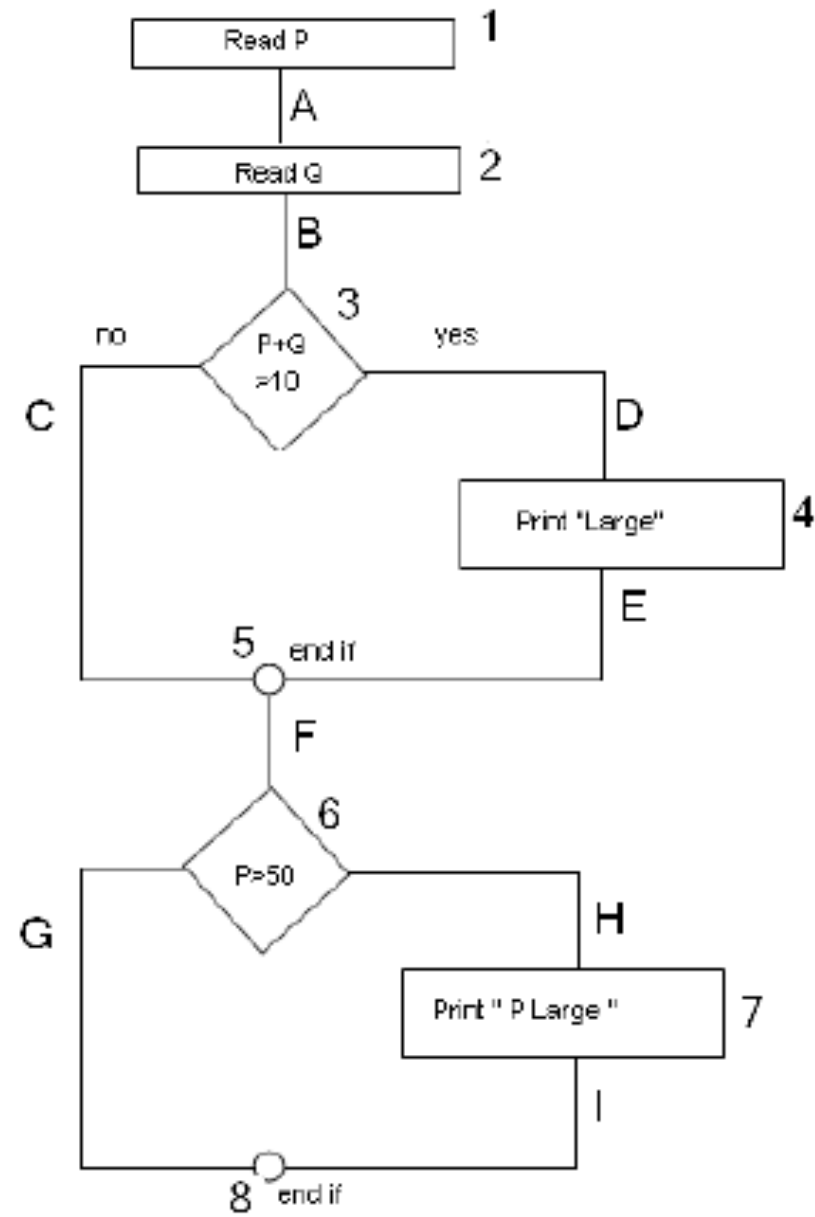
# Calculating Statement Coverage

- Define test case
- Identify traversed path correspond to test case
- Does test case cover all nodes?
- Example:
  - How many test cases are needed for full statement coverage?
- Goal: identify min # paths



# Calculating Branch Coverage

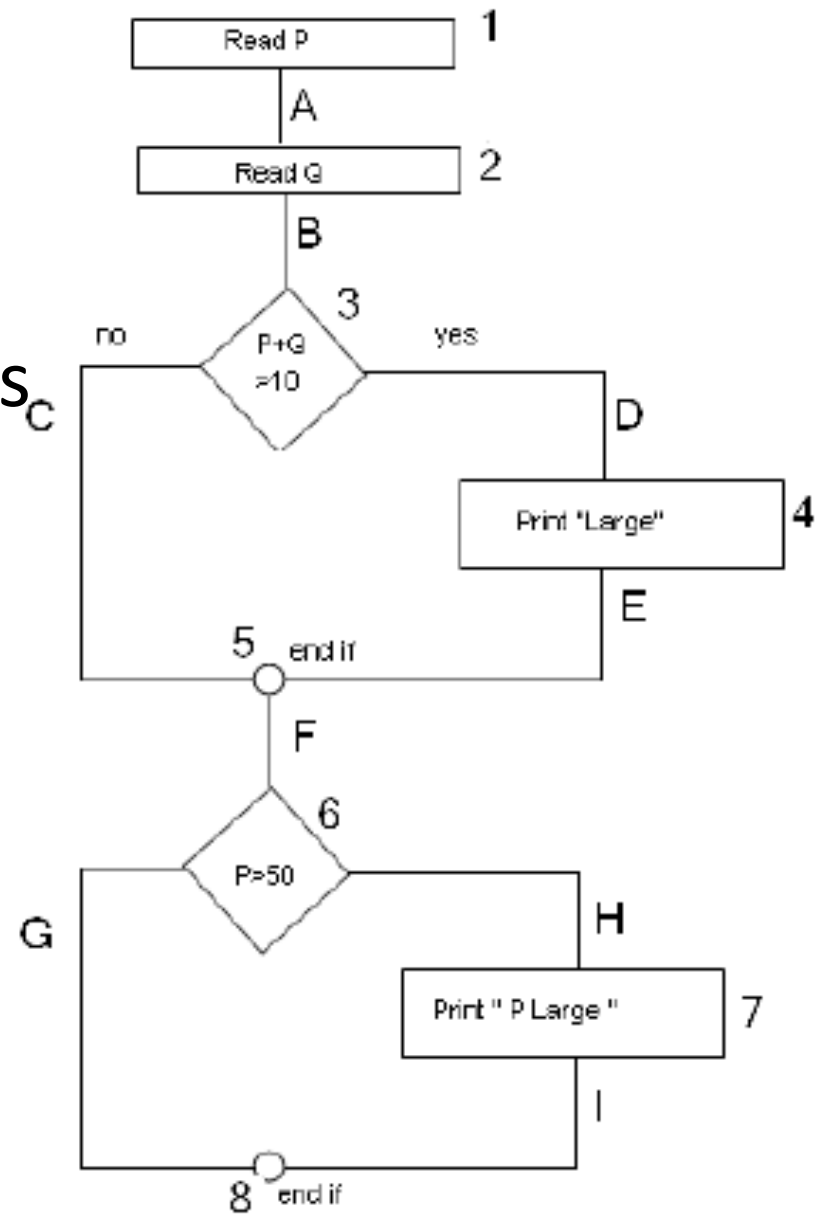
- Start with paths that cover most number of branches
- Identify new paths to cover remaining branches
- Are all branches covered?
- Example:
  - How many test cases are needed for full branch coverage?
- Goal: identify min # paths





# Identifying All Paths

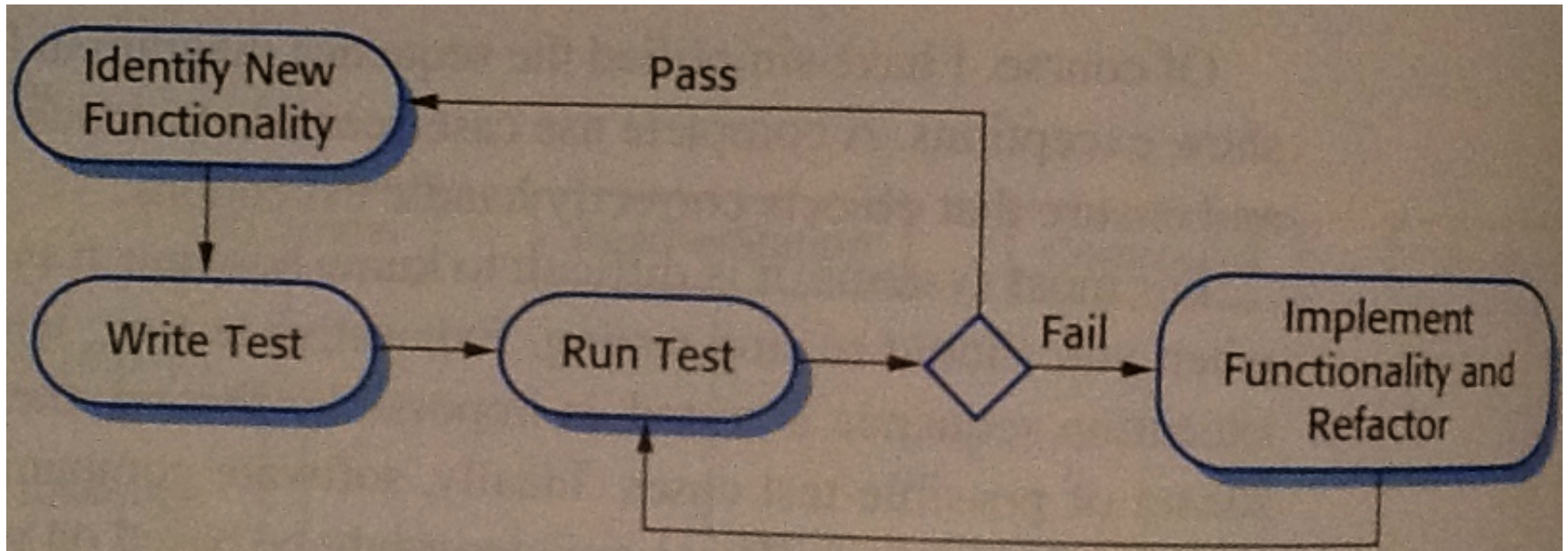
- Identify all unique paths from start to end
- Considers all combinations of branches
- Example:
  - How many test cases are needed for full path coverage?
- What are all the paths?



# Test-Driven Development

- interleave testing and coding
- develop code iteratively with test case
- originally introduced as part of XP
- can be incorporated into plan driven development as well

# TDD Process



- write a new failing test
- run tests  
(expectations?)
- get it to pass
- refactor
- repeat

# TDD Example: Ruby on Rails

- RSpec - language for testing

**Listing 3.9.** Code to test the contents of the Home page.

`spec/requests/static_pages_spec.rb`

```
require 'spec_helper'

describe "Static pages" do

  describe "Home page" do

    it "should have the content 'Sample App'" do
      visit '/static_pages/home'
      page.should have_content('Sample App')
    end
  end
end
```

# Example cont.

## Test fails

```
1. ~/rails_projects/sample_app (bash)
[sample_app (master)]$ bundle exec rspec spec/requests/static_pages_spec.rb
F

Failures:

  1) StaticPages Home page should have the content 'Sample App'
     Failure/Error: page.should have_content('Sample App')
       expected there to be content "Sample App" in "SampleApp\n\nStaticPages#home\nFind me in app/views/static_pages/home.html.erb\n\n"
     # ./spec/requests/static_pages_spec.rb:9:in `block (3 levels) in <top (required)>'

Finished in 6.69 seconds
1 example, 1 failure

Failed examples:

rspec ./spec/requests/static_pages_spec.rb:7 # StaticPages Home page should have the content 'Sample App'
[sample_app (master)]$
```

# Example cont.

## Write code to pass test

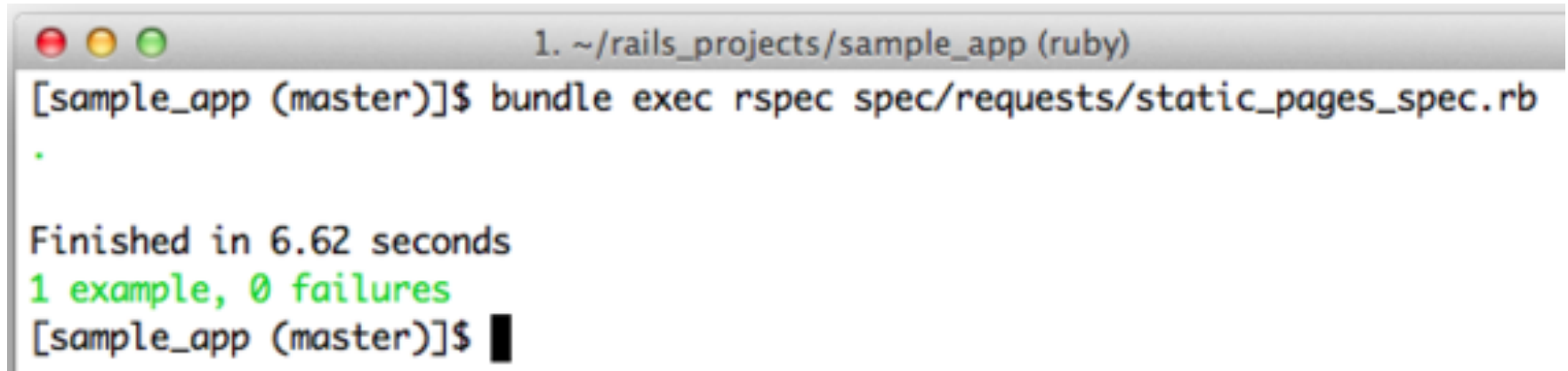
**Listing 3.10.** Code to get a passing test for the Home page.

`app/views/static_pages/home.html.erb`

```
<h1>Sample App</h1>
<p>
  This is the home page for the
  <a href="http://railstutorial.org/">Ruby on Rails Tutorial</a>
  sample application.
</p>
```

# Example cont.

## Pass test



```
1. ~/rails_projects/sample_app (ruby)
[sample_app (master)]$ bundle exec rspec spec/requests/static_pages_spec.rb
.

Finished in 6.62 seconds
1 example, 0 failures
[sample_app (master)]$
```

See <http://ruby.railstutorial.org/>

# TDD Benefits

- why is TDD helpful in the development process?



# TDD Benefits

- **clarity**
  - helps programmers clarify what the code is supposed to do
- **code coverage**
  - at least one test per code segment
- **regression testing**
  - incremental development process enables regression tests to be run any time
- **simplified debugging**
  - when a test fails, it should be obvious where the problem lies
- **system documentation**
  - use tests as a form of documentation

# Writing Tests

- what test(s) would you write if you have to...
  - write a feature that sums up the prices of the items in a shopping cart
  - write a feature that shows the time left in a user's turn (max 30 sec) in Tic-Tac-Toe
  - write a feature that shows who's turn it is currently in a two player game
  - write a feature that lets users delete their own entries in a blog

# Stress Testing

- a type of performance testing at system level
- stress the system by making demands that are outside the design limits of the software
- e.g.: process 300 transactions per second
  - start by testing system with  $< 300$  trans/sec
  - gradually increase load beyond 300 trans/sec
  - stop when load is well beyond 300 or when system fails
- tests system's failure behaviour
  - ensure no data corruption or unexpected loss of user services
- may cause unusual defects to be discovered

# User Testing

- users provide advice on system testing
- alpha testing
  - users work with development team to test software
  - at developer's site
- beta testing
  - software release made available to users
  - allows users to experiment and raise problems that they discovered
- acceptance testing
  - users test a system to decide whether or not it is ready to be accepted for deployment in user's environment

# Software Quality

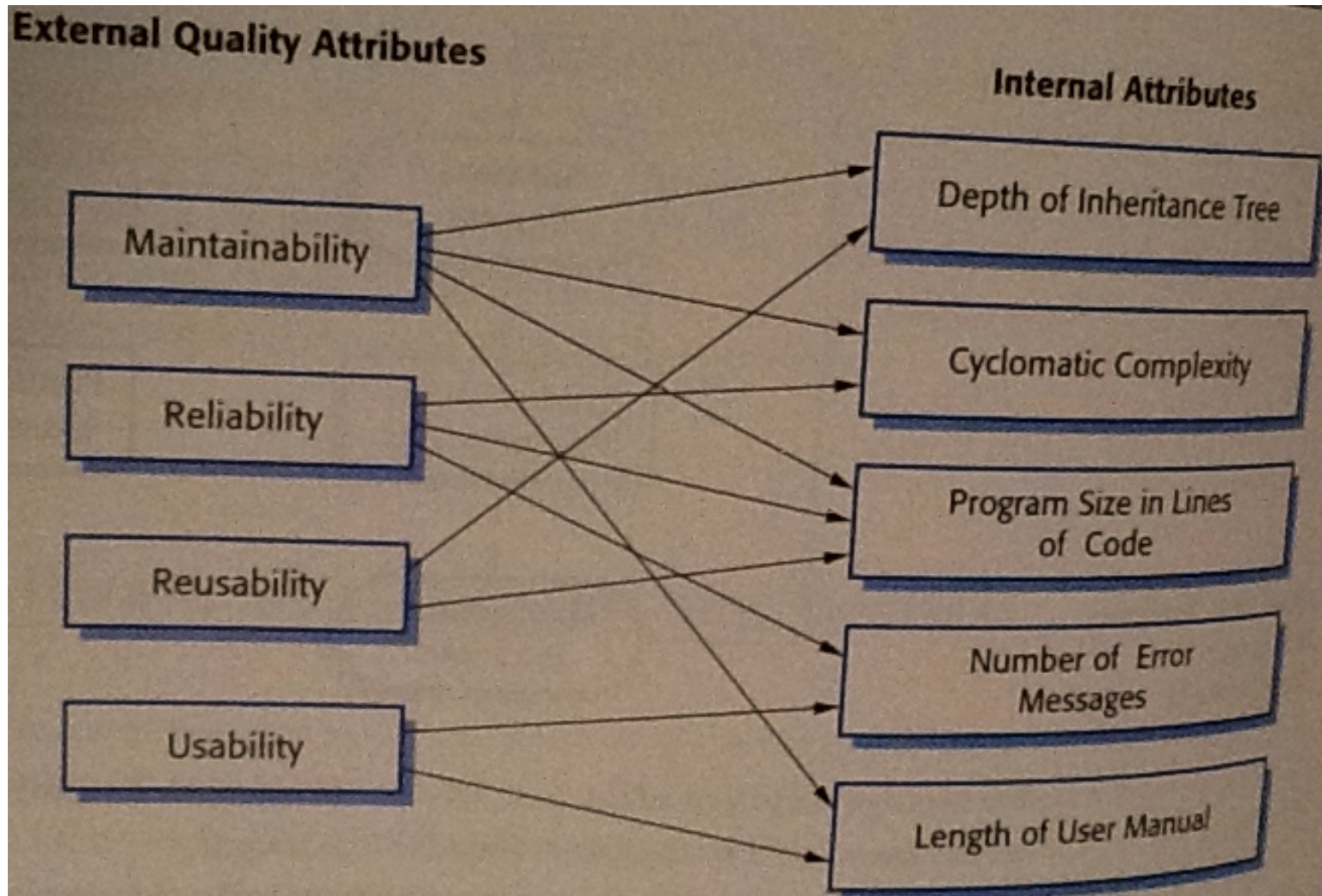
- what are some desirable software qualities?
  - from the correctness perspective?
  - from the development perspective?
  - from the users' perspective?

# Software Quality Attributes (Boehm 1978)

Safety	Understandability	Portability
Security	Testability	Usability
Reliability	Adaptability	Reusability
Resilience	Modularity	Efficiency
Robustness	Complexity	Learnability

- difficult to take direct measurements of these

# Measuring Internal Attributes



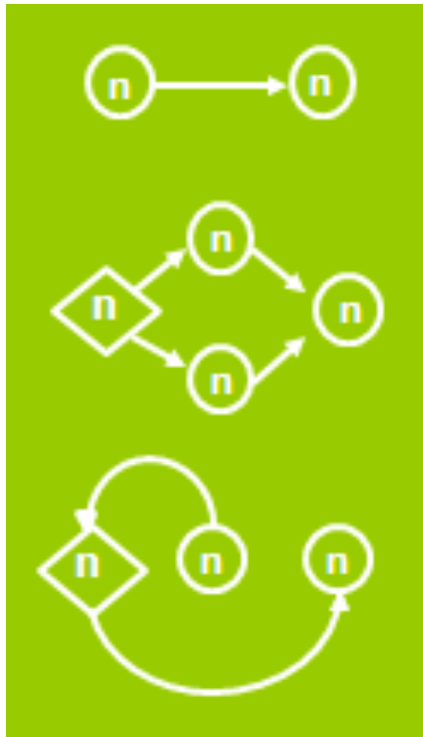
# Cyclomatic complexity $v(G)$

- Measures logical complexity of program
- $v(G) = |E| - |N| + 2$ 
  - $G$  = flow graph
  - $E$  = set of edges
  - $N$  = set of nodes
- This value gives the number of linearly independent paths in  $G$
- Indicates minimum effort to code and test module



# Examples

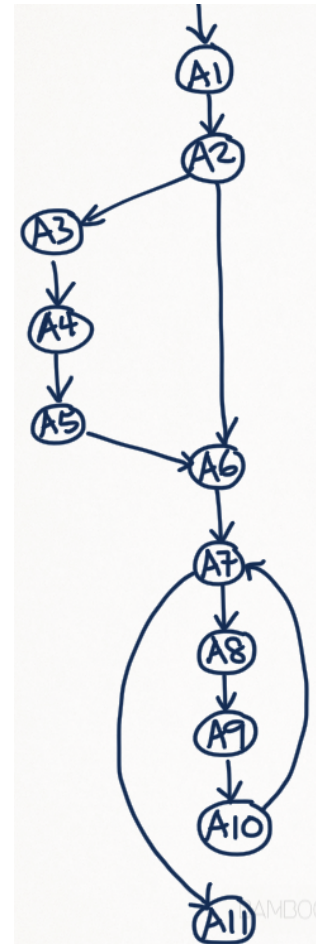
- $v(G) = |E| - |N| + 2$
- small examples:



$ E $	$ N $	$v(G)$
?	?	?
?	?	?
?	?	?

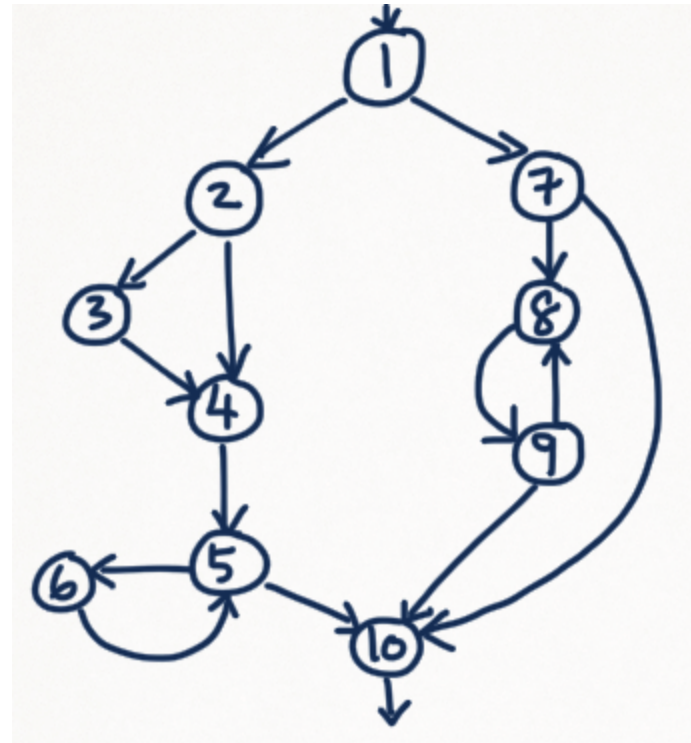
# Example

- $|E| = ?$
- $|N| = ?$
- $v(G) = ?$



# Example

- $|E| = ?$
- $|N| = ?$
- $v(G) = ?$



# Guidelines for V(G)

- High complexity ~ more bugs and security flaws

Complexity No.	Corresponding Meaning of V(G)
1-10	1) Well-written code, 2) Testability is high, 3) Cost / effort to maintain is low
10-20	1) Moderately complex code, 2) Testability is medium, 3) Cost / effort to maintain is medium.
20-40	1) Very complex code, 2) Testability is low, 3) Cost / effort to maintain is high.
> 40	1) Not testable, 2) Any amount of money / effort to maintain may not be enough.

# Something Simple

```
Program: cache
cache WriteAddress (F)
Cyclomatic Graph
Cyclomatic 3
Essential 1
Design 2
```

01/24/08  
Superimposed  
Upward Flows  
Loop Exits  
Plain Edges

```
Annotated Source Listing
Program : cache
File    : cache.c
Language: c_npp
Module  Module
Letter  Name
-----
F cache`WriteAddress
v(G)   ev(G)  iv(G)
-----
3     1     2

152   F0      static int
153   WriteAddress(addr, cache)
154   Address addr;
155   Cache *cache;
156   {
157   /* write addr to cache.  Observe write po
158
159   int rv = 0;
160
161   F1* F2    if (fAddressInCache(addr, cache))
162   F3        rv = HIT;
163   else
164   F4        if (WRITE_POLICY == WRITE_BACK) /*
165   F5*       PlaceAddressInCache(addr, cach
```

# Something Not So Simple

Graph/Listing for 'crackaddr'

Zoom In Zoom Out Print... Save As... Save Text... Close Help... Graph (45)%:

Magnification Level: 5

Page 1 of 3 crackaddr

Program: Security

crackaddr (AIL)

Cyclomatic Graph

Cyclomatic 89

Essential 30

Design 19

Annotated Source Listing

Program : Security 04/17

File : C:\Documents and Settings\tmccabe\Desktop\allc\crackaddr-bad.c

Language: cw\_C\_inst

Module	Letter Name	v(G)	ev(G)	iv(G)	Start Line	Num Lin
AIL	crackaddr	89	30	19	130	

```
130  AIL0      char *
131          crackaddr(addr)
132          register char *addr;
133          {
134              register char *p;
135              register char c;
136              int cmtlev;
137              int realcmtlev;
138              int anglelev, realanglelev;
139              int copylev;
140              int bracklev;
141              enum bool qmode;
142              enum bool realqmode;
143              enum bool skipping;
144                  enum bool putgmac = false;
145              enum bool quoteit = false;
146              enum bool gotangle = false;
147              enum bool gotcolon = false;
148              register char *bp;
149              char *obp;
150              char *buflim;
151              char *bufhead;
152              char *addrhead;
153              static char buf[MAXNAME + 1];
154              static char test_buf[10]; /* will use as a canary
155                                      /* of buf[] */
156
157  AIL1* AIL2      strcpy(test_buf, "GOOD");
158
159  AIL3* AIL4      printf("Inside crackaddr!\n");
160
161
```

# Relation to Testing and Quality

- high cyclomatic complexity and white box testing?
- how to reduce code's cyclomatic complexity?

# Changing views of quality

Past	Present
Quality is responsibility of blue collar workers and direct labour employees working on the product	Quality is everyone's responsibility including white collar workers, the indirect labour force and the overhead staff
Quality defects should be hidden from the customer and management	Defects should be highlighted and brought to surface for corrective action
Quality problems lead to blame faulty justifications and excuses	Quality problems lead to cooperative solutions



# Changing views of quality

Past	Present
Corrections-quality problems should be accompanied with minimum documentation	Documentation is essential for “lessons learnt” report
Increase quality will increase project cost	Improved quality saves money and increase business
Will not occur without close supervision of people	People want to produce quality products

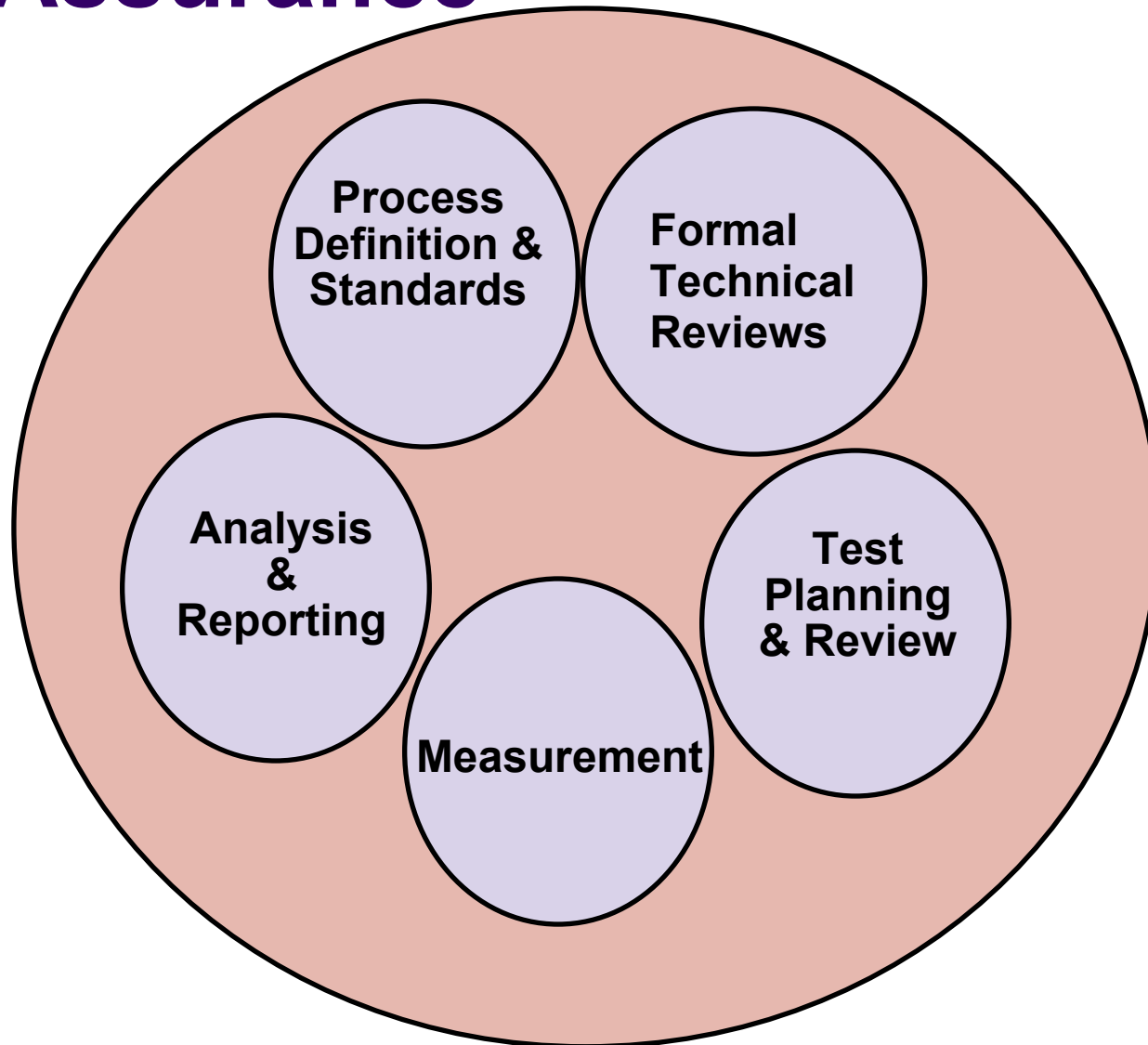
# What is Software Quality Assurance?

- reviews all software products and activities
- auditing can be performed to ensure products and processes conform to organizational standards
- must define:
  - product standards
  - process standards
- what might be some examples of these standards?

# Example Standards

<b>Product Standards</b>	<b>Process Standards</b>
Design review form	Design review conduct
Requirements document structure	Submission of new code to configuration management
Method header format	Version release process
Java programming style	Project plan approval process
Project plan format	Change control process
Change request form	Test recording process

# Software Quality Assurance



# A Detailed Look at SQA

- SQA plan - created to define team's strategy in achieving quality
- explicitly defines what is meant by "software quality"
- create activities to ensure every work product exhibits high quality
- use metrics to develop strategies for improving software process and product quality
- when is the SQA plan created?

# SQA Responsibilities

- prepares SQA plan
- participates in development of software process description
- review all activities to verify compliance
- audits designated software product
- ensures deviations on software work and work products are documented and handled according to procedure
- records any non-compliance

# Software Inspection

- static V&V activity
- use knowledge of system, application domain, programming experience to discover errors
- a meeting conducted by technical people for technical people
- inspects any readable representation of software
  - source code (most emphasis)
  - requirements
  - design models
  - test plans

# Who are the Practitioners?

Software Assurance practitioners include a wide range of personnel employed throughout SDLC

Software Managers

Safety Engineers

Software Engineers



Systems Engineers

Software Quality Personnel

V&V Personnel

*Software Assurance personnel aren't just those Software Quality folks!!*



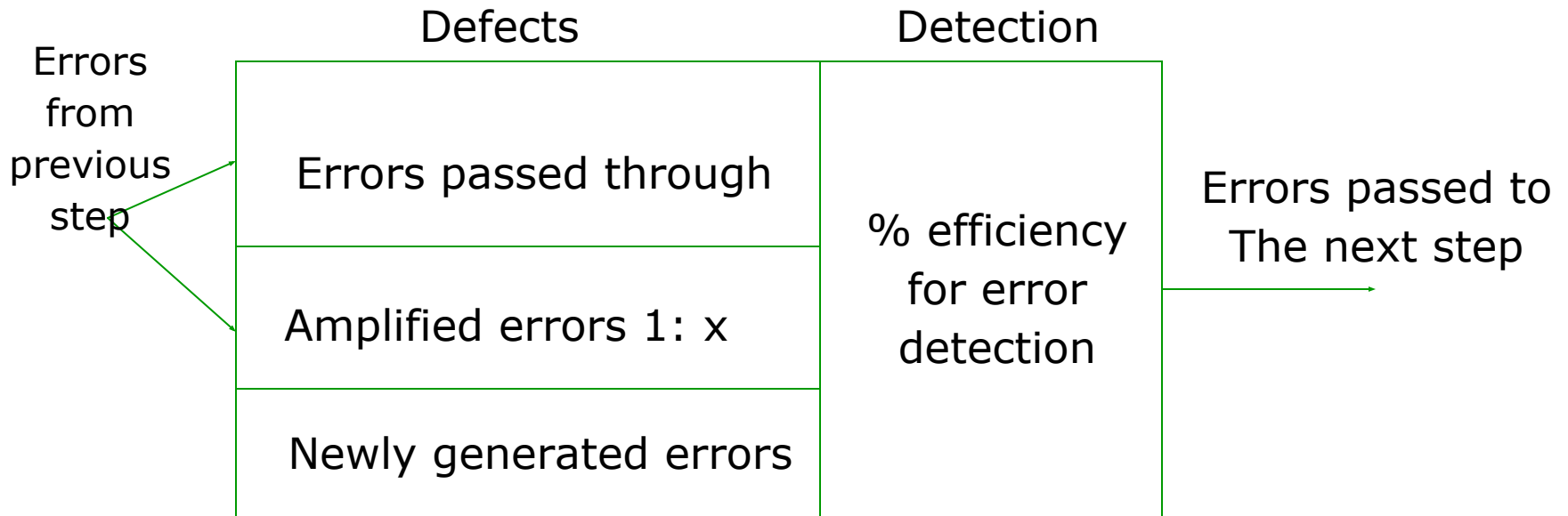
# Advantages of Inspections over Testing

- do not have to consider system interactions
  - errors may mask other errors - difficult to identify source of error
- able to inspect incomplete versions of system
- able to consider broader quality attributes:
  - compliance with standards, portability, maintainability
  - look for inefficiencies, inappropriate algorithms, poor programming style

# Software Reviews

- less structured static V&V activity
- aim to improve software quality by detecting defects
- creates technical assessment of a work product
- may also serve as training ground
- example activities:
  - pair programming
  - system walkthroughs
  - formal inspections

# Cost Impact- Defect Amplification model



How to convince the management that reviews are cost-effective!

# Example

Prelim. Design

0	0%
0	
10	

10

6

4

Detailed Design

6	0%
4 x 1.5	
25	

37

10

27

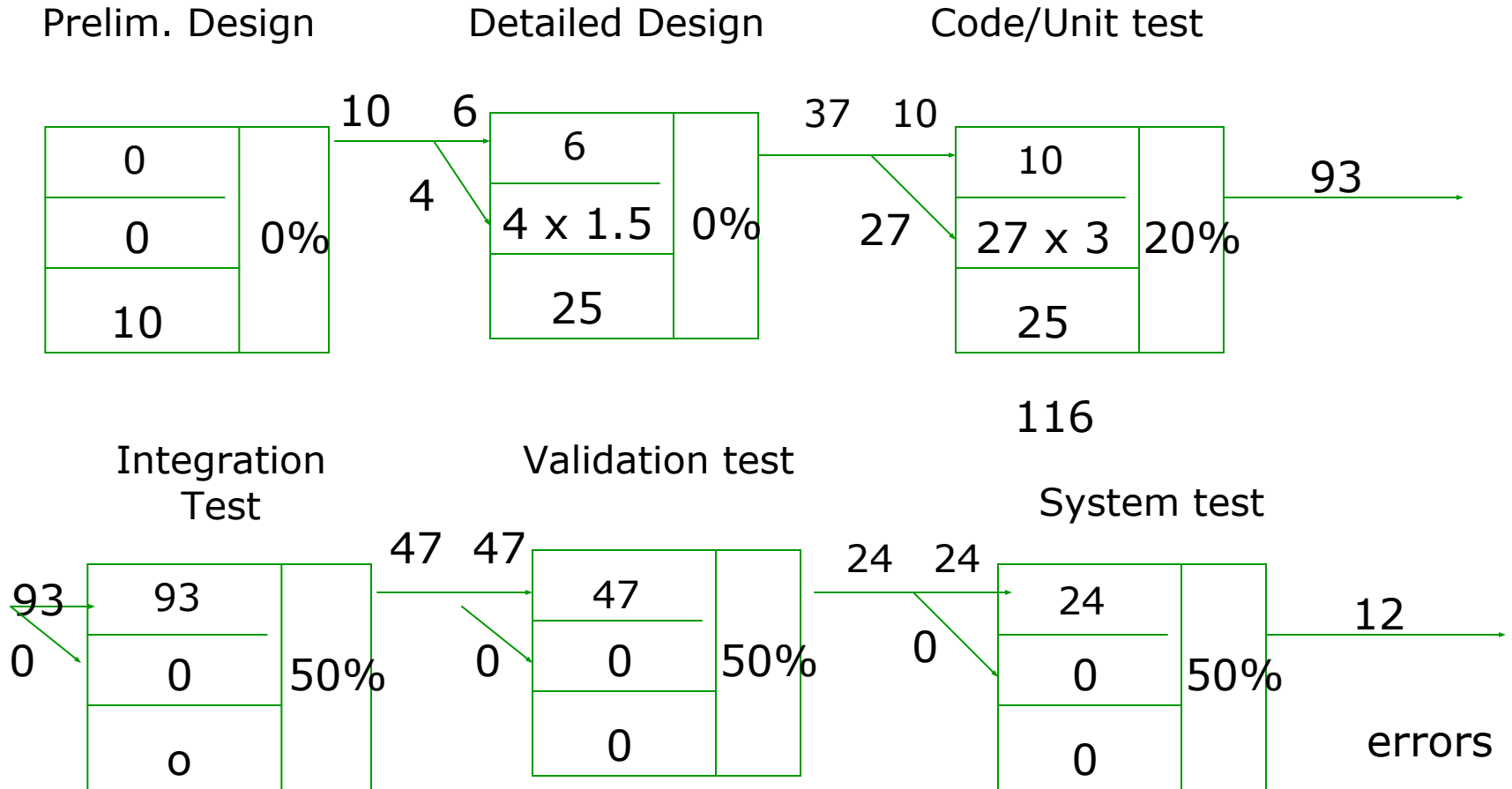
Code/Unit test

10	20%
27 x 3	
25	

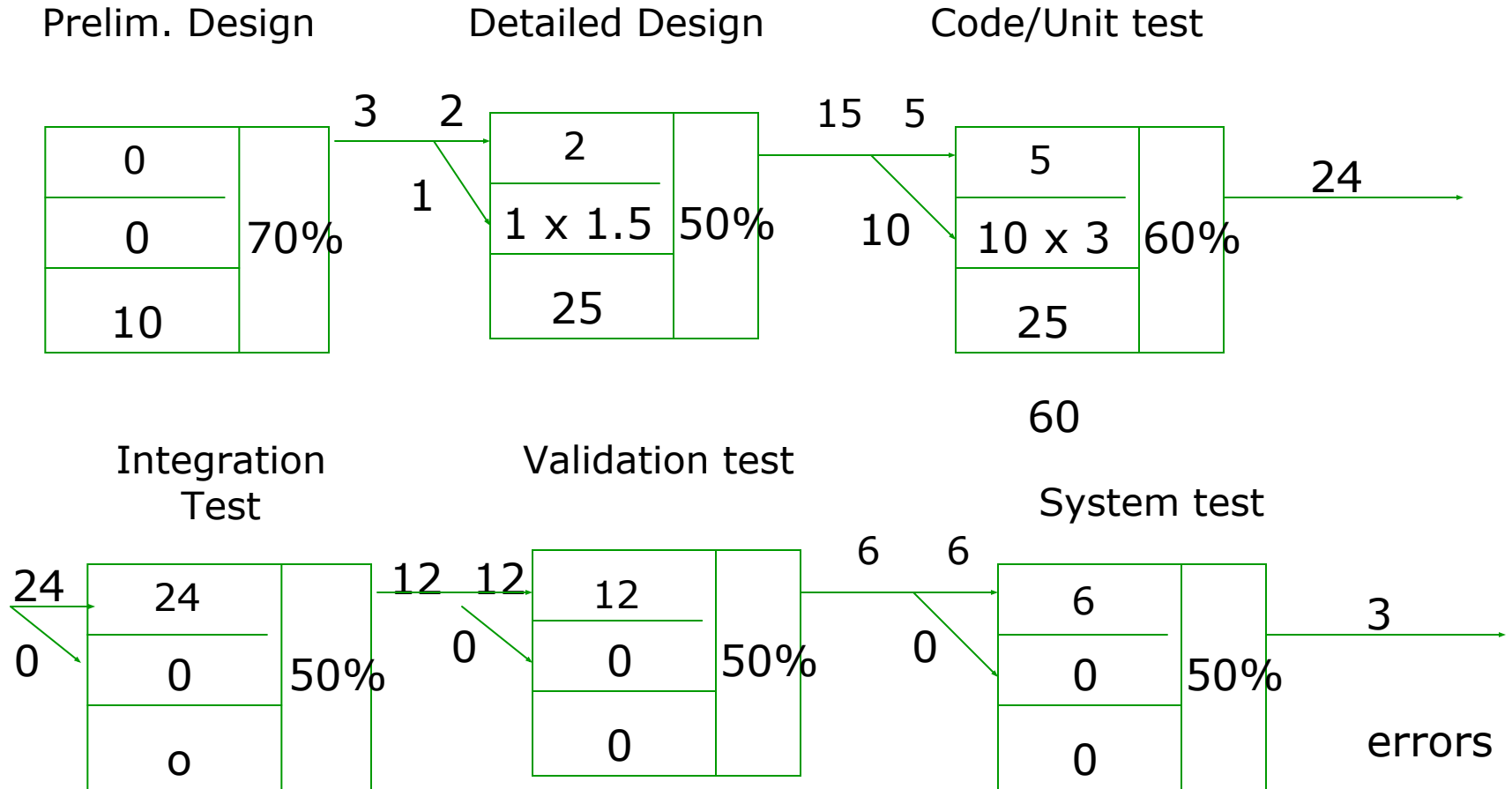
93



# Defect Amplification model –No review Model



# Defect Amplification model – Reviews conducted





How the customer explained it



How the Project Leader understood it



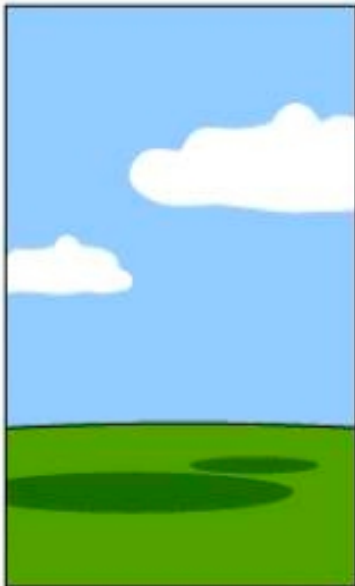
How the Analyst designed it



How the Programmer wrote it



How the Business Consultant described it



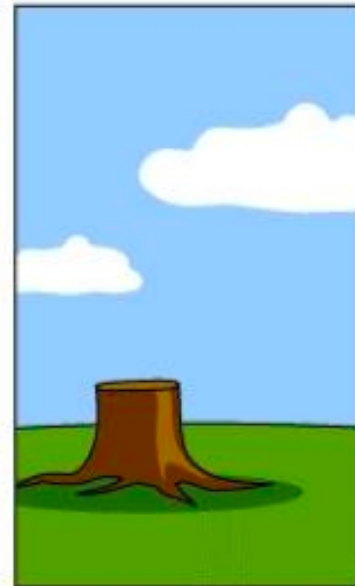
How the project was documented



What operations installed



How the customer was billed



How it was supported



What the customer really needed

# References

- <http://www.sa-depot.com/?p=203>