ESSENTIALS OF

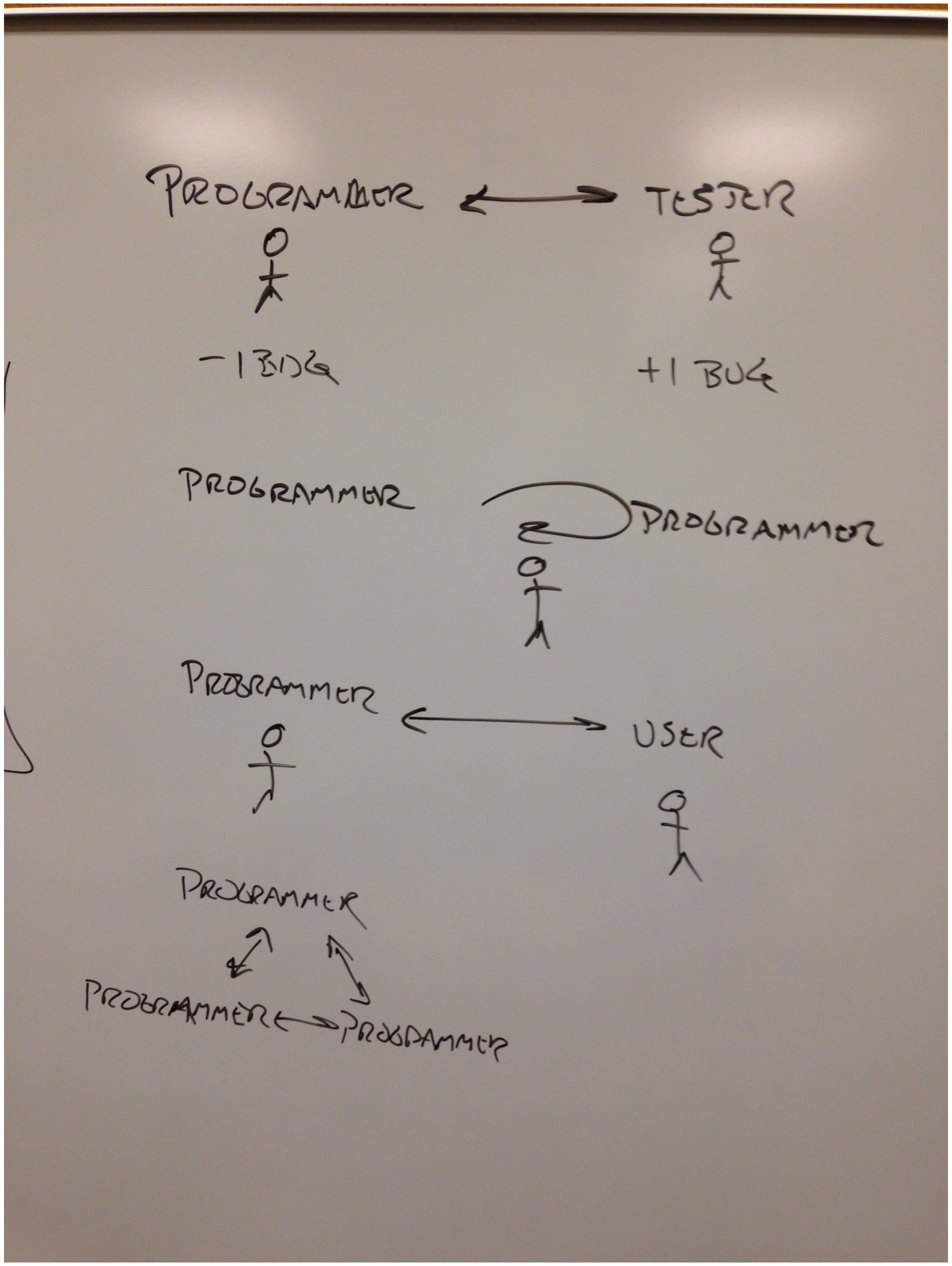
software engineering

THIRD EDITION

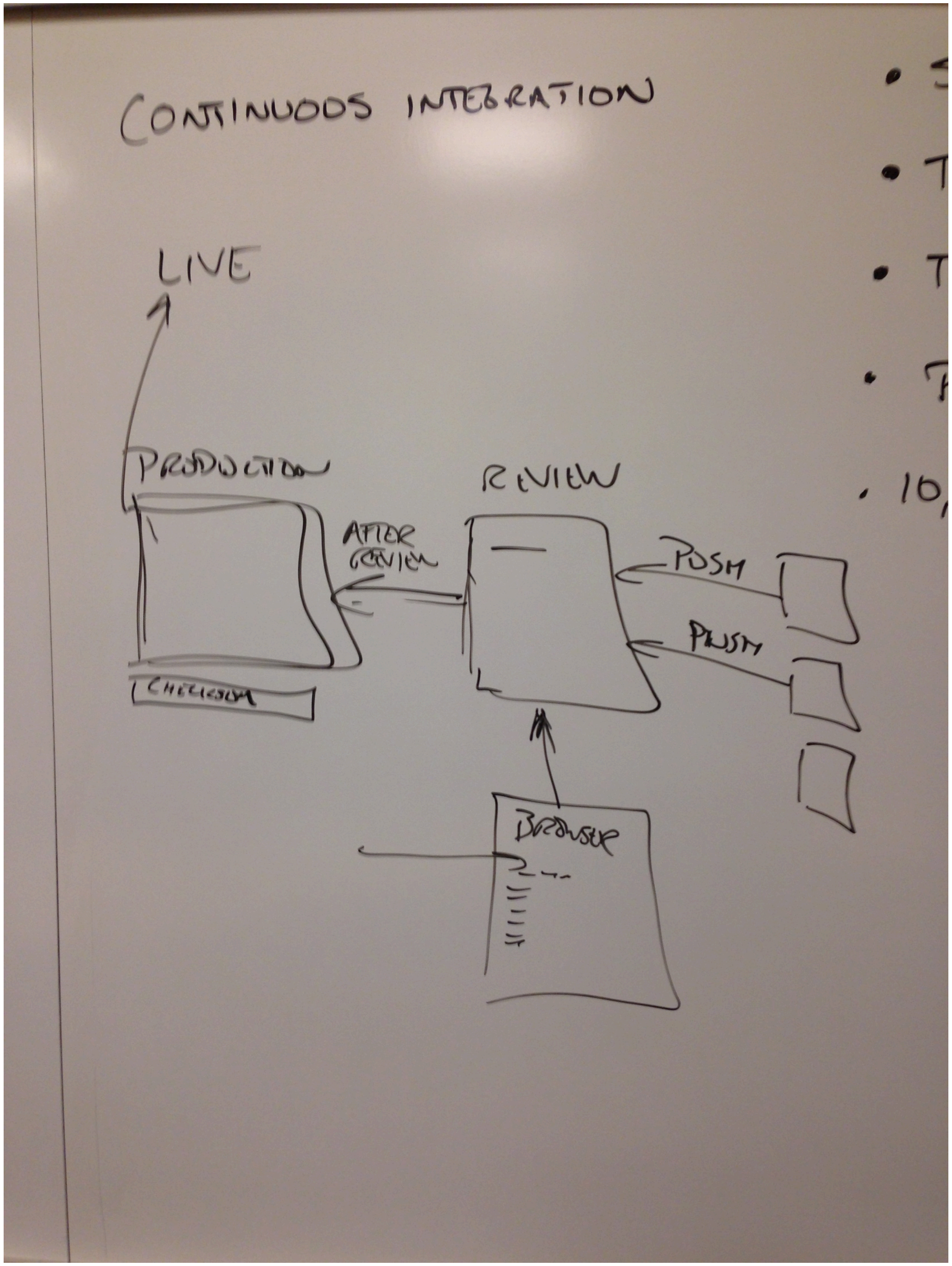
Frank Tsui
Orlando Karam
Barbara Bernal

Chapter 10

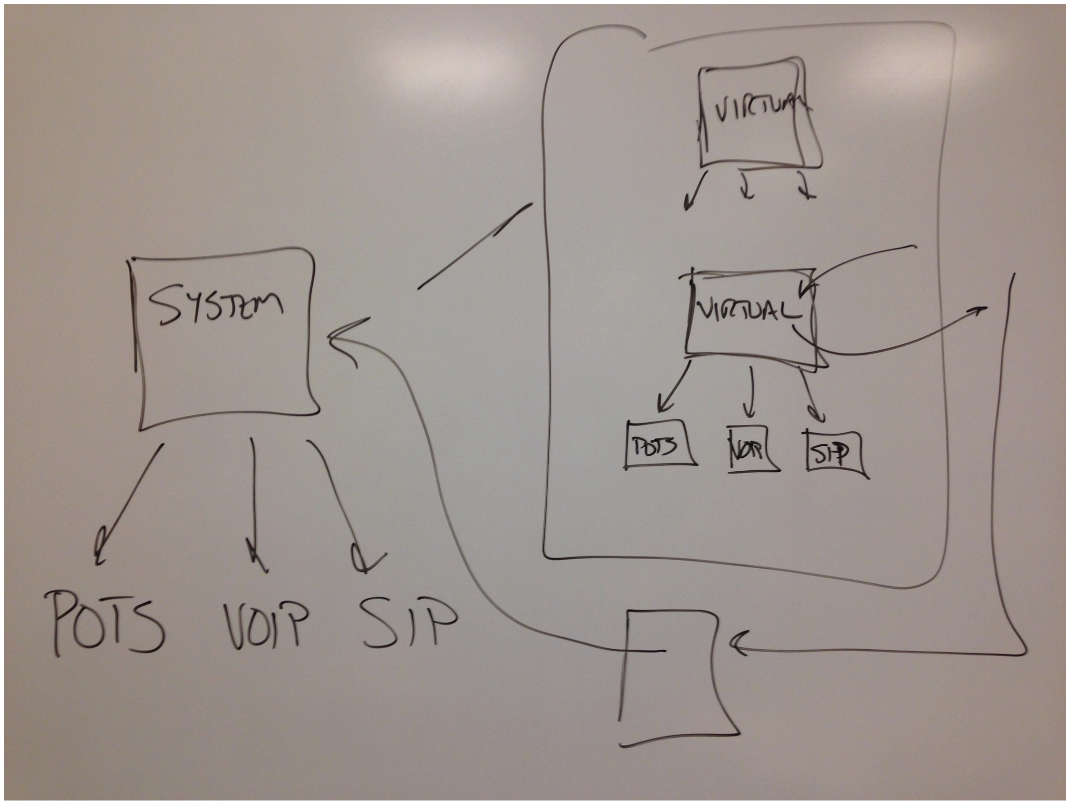
Testing and Quality Assurance



Different styles of doing code review



Human Reviewer Code Inspection with continuous integration infrastructure



Pinger’s testing set up

Testing Related topics

1. Understand basic techniques for software verification and validation

2. Analyze basics of software testing and testing techniques

3. Discuss the concept of “inspection” process

Introduction

- **Quality Assurance (QA): activities designed to *measure* and *improve quality* in a product --- and process**

similar

- **Quality control (QC): activities designed to *validate & verify the quality* of the product through *detecting faults and “fixing” the defects***

- **Need good *techniques*, *process*, *tools* and *team***

What is "Quality?"

- **Two traditional definitions**:
  - ***Conforms to requirements***
  - ***Fit to use***

- ***Verification***: checking the software ***conforms to its requirements*** (***did the software evolve from the requirements properly***)

- ***Validation:*** checking software meets user requirements (***fit to use***)

Some “Error Detection” Techniques
(finding errors)

- **Testing**: executing program in a controlled environment and “verifying/validating” output
- **Inspections and Reviews**
- **Formal methods** (proving software correct)
- **Static analysis** detects “**error-prone conditions**”

Faults and Failures

- **Error**: a mistake made by a programmer or software engineer which caused the fault, which in turn may cause a failure
- **Fault** (**defect, bug**): condition that *may* cause a failure in the system
- **Failure** (**problem**): inability of system to perform a function according to its spec due to some fault

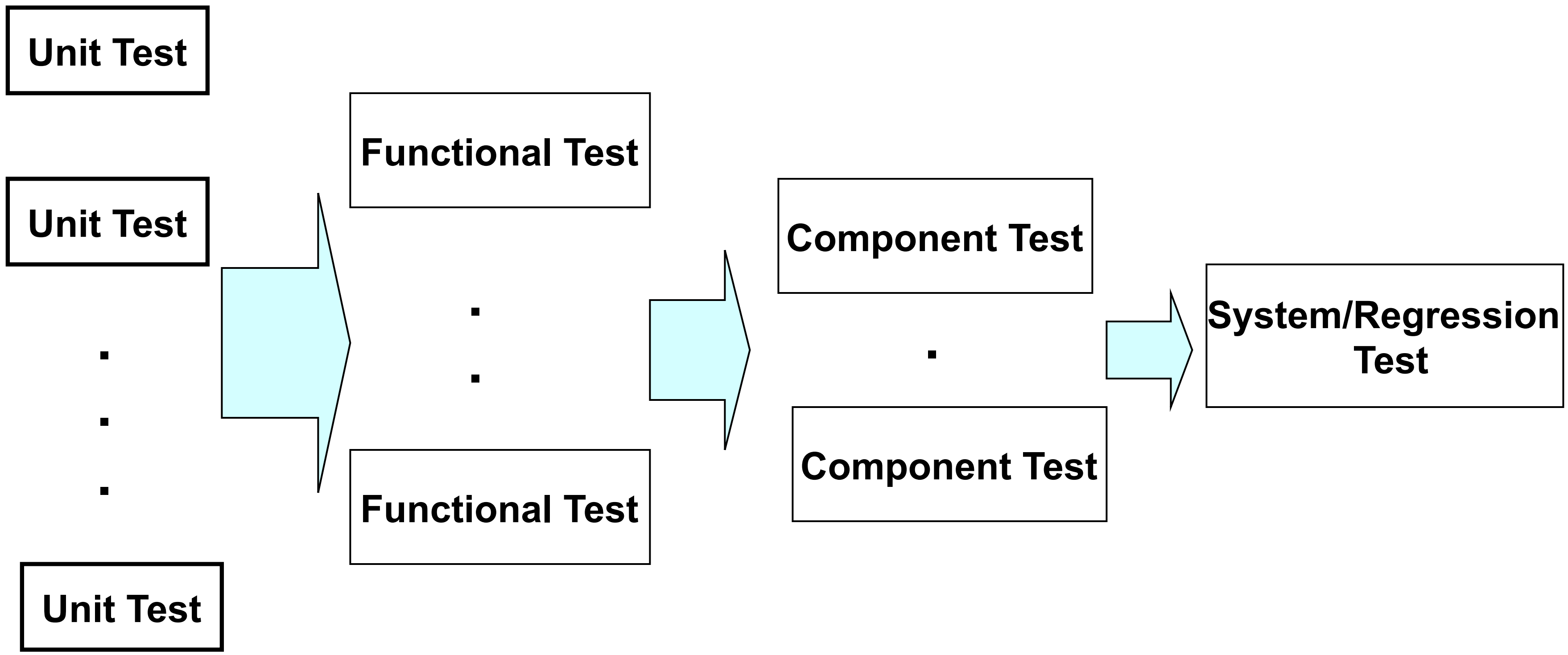
- **Fault or Problem severity (based on consequences)**
- **Fault or Problem priority (based on importance of developing a fix which is in turn based on severity)**

Testing

- <u>Activity</u> performed for
  - *Evaluating* product quality
  - *Improving products* by identifying defects and <u>having them fixed prior to software release</u>. ← Not always done !
- Dynamic (running-program) verification of program's behavior on a finite set of test cases selected from execution domain
- Testing can NOT prove product works 100%- -- even though we use testing to demonstrate that parts of the software works

Testing

- **Who tests**
  - *Programmers*
  - *Testers/Req. Analyst*
  - *Users*
- **What is tested**
  - **Unit Code** testing
  - **Functional Code** testing
  - Integration/**system** testing
  - **User interface** testing
- **Why test**
  - **Acceptance (customer)**
  - **Conformance (std, laws, etc)**
  - **Configuration (user .vs. dev.)**
  - **Performance, stress, security, etc.**
- **How (test cases designed)**
  - **Intuition**
  - **Specification based** (***black box)***
  - **Code based** (***white-box)***
  - **Existing cases** (***regression***)



Progression of Testing

Equivalence Class partitioning

- **Divide the input into several groups, deemed "equivalent" for purposes of finding errors.**
- **Pick one "representative" for each class used for testing.**
- **Equivalence classes determined by req./des. specifications and some intuition**

Example: pick "larger" of two integers and -------

Class	Representative
First > Second	10,7
Second > First	8,12
First = second	36, 36

1.Lessen duplication
2.Complete coverage

Simple Example of Equivalence Testing

- **Suppose we have <u>n distinct functional requirements</u>.**
  - **Suppose further that these n "functional" requirements are such that**
    - **r1 U r2 U ------ U rn = all n requirements and**
    - **ri $\cap$ rj = θ**
  - **We can devise a test scenario, ti, for each of the ri functionality to check if ri "works." Then:**
    - **t1 U t2 U --------- tn = all the test cases to cover the software functionalities.**
    - **Note that there may be more than one ti for ri. But <u>picking only one from the set of potential test cases</u> for ri, we form an equivalence class of test cases**

Boundary Value analysis
(A Black-Box technique)

- Past experiences show that "**Boundaries**" are error-prone
- **Do equivalence-class partitioning**, add test cases for boundaries (*at boundary*, *outside*, *inside*)
  - **Reduced cases**: consider boundary as falling between numbers
    - If boundary is at12, **normal**: 11,12,13; ***reduced***: 12,13 (boundary 12 and 13)
- **Large number of cases** (~3 per boundary)
- ***Good for "ordinal values"***

<u>Boundaries</u> of the input values

1 <= number of employees, **n** <= 1000000

1 ← **n** → **1000000**

1 <= employee **age** <= 150

1 ← **age** → **150**

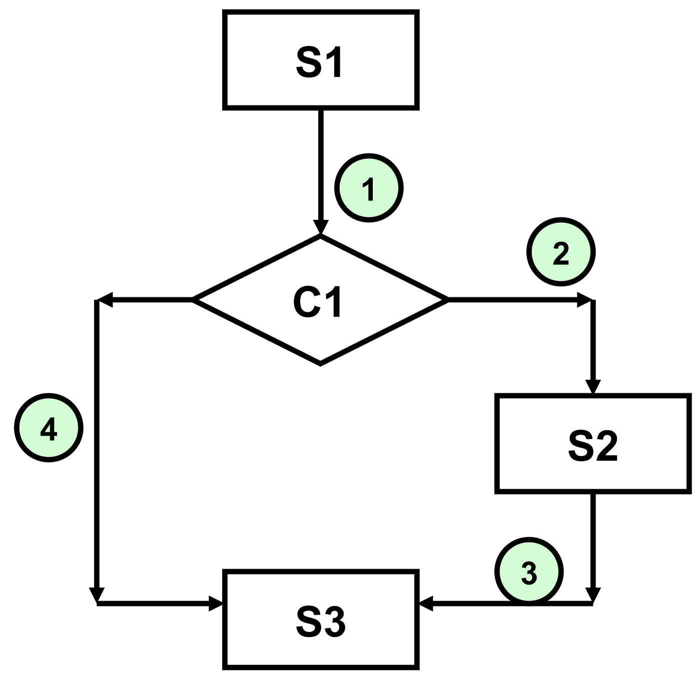
The "basic" boundary value testing for a value would include:

1. - at the "minimum" boundary
2. - immediately above minimum
3. - between minimum and maximum (nominal)
4. - immediately below maximum
5. - at the "maximum" boundary

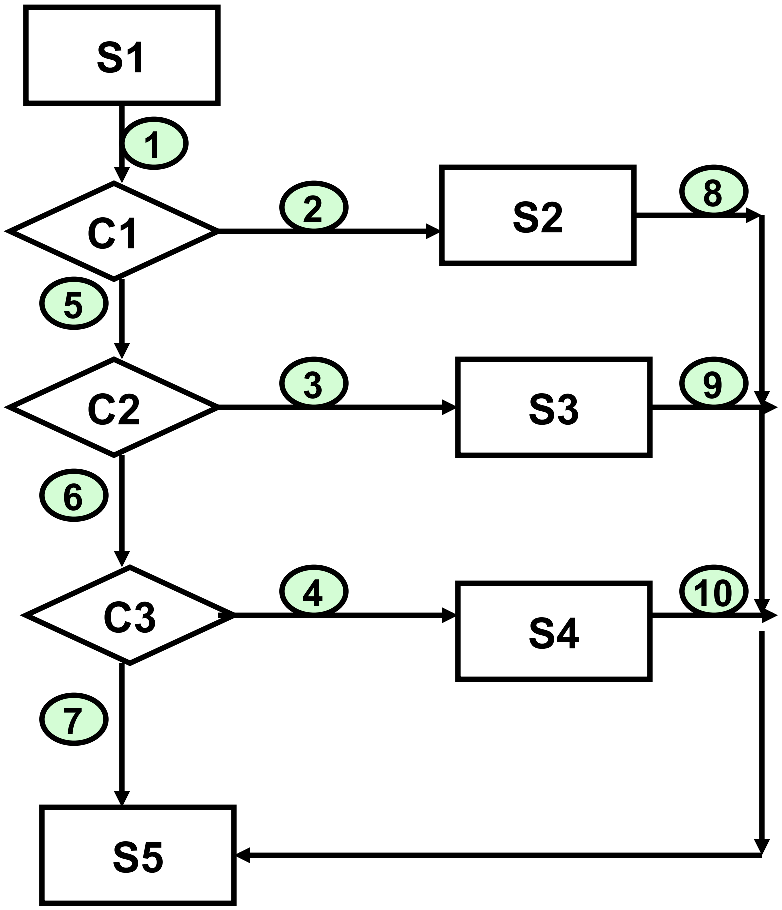
**** note that we did not include the <u>"outside" of the boundaries</u> here****

Path Analysis

- **White-Box** technique
- ***Two tasks***
  1. **Analyze number of paths in program**
  2. **Decide which ones to test**
- ***Decreasing coverage:***
  - **Logical paths**
  - **Independent paths**
  - **Branch coverage**
  - **Statement coverage**



Path1 : S1 – C1 – S3
Path2 : S1 – C1 – S2 – S3
OR
Path1: segments (1,4)
Path2: segments (1,2,3)

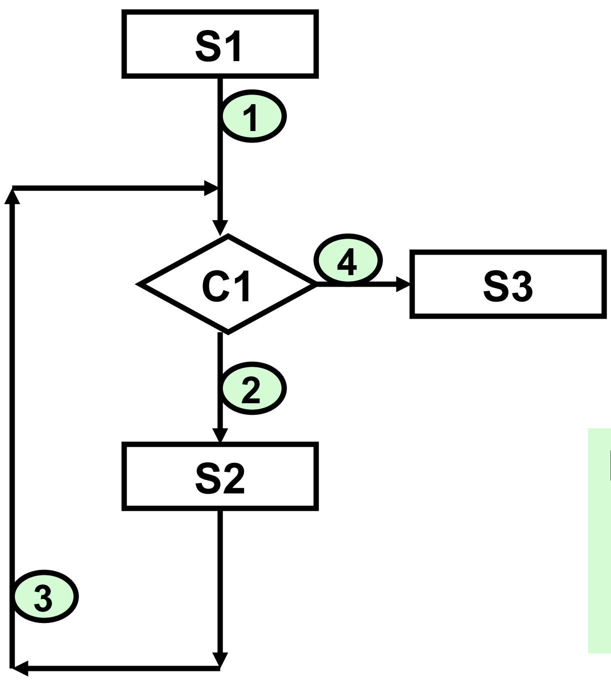


The 4 Independent Paths Covers:

Path1: includes S1-C1-S2-S5
Path2: includes S1-C1-C2-S3-S5
Path3: includes S1-C1-C2-C3-S4-S5
Path4: includes S1-C1-C2-C3-S5

A "CASE" Structure

Example with a Loop



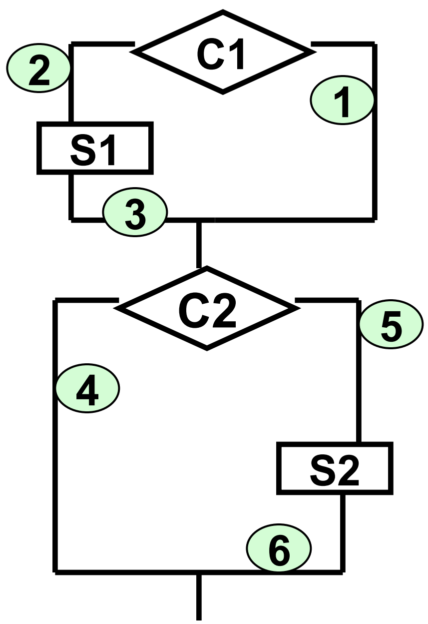
Linearly Independent Paths are:

path1 : S1-C1-S3 (segments 1,4)
path2 : S1-C1-S2-C1-S3 (segments 1,2,3,4)

A Simple Loop Structure

Linearly Independent Set of Paths

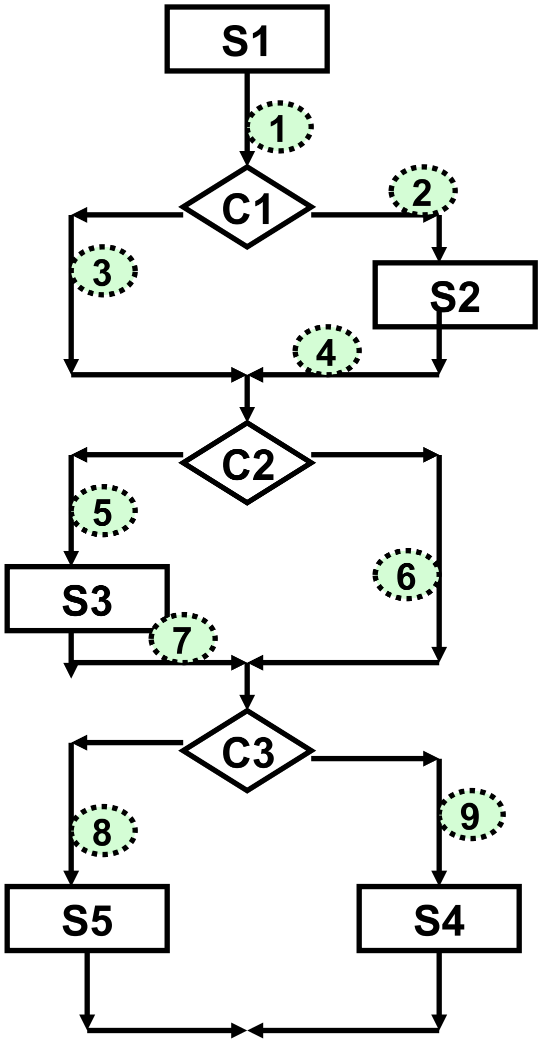
Consider path1, path2 and path3 as the <u>Linearly Independent Set</u>



	1	2	3	4	5	6
path1	1				1	1
path2	1			1		
path3		1	1	1		
path4		1	1		1	1

Remember McCabe's Cyclomatic number ?
It is the same as linearly independent set of paths

Total # of Paths and Linearly Independent Paths



Since for each binary decision, there are 2 paths and there are 3 in sequence, there are 2^3 = <u>8 total "logical" paths</u>

path1 : S1-C1-S2-C2-C3-S4
path2 : S1-C1-S2-C2-C3-S5
path3 : S1-C1-S2-C2-S3-C3-S4
path4 : S1-C1-S2-C2-S3-C3-S5

path5 : S1-C1-C2-C3-S4
path6 : S1-C1-C2-C3-S5
path7 : S1-C1-C2-S3-C3-S4
path8 : S1-C1-C2-S3-C3-S5

How many <u>Linearly Independent</u> paths are there?
Using Cyclomatic number = 3 decisions +1 = <u>4</u>

***<u>One set</u>* would be:**
path1 : includes segments (1,2,4,6,9)
path2 : includes segments (1,2,4,6,8)
path3 : includes segments (1,2,4,5,7,9)
path5 : includes segments (1,3,6,9)

Combinations of Conditions

- **Function of several <u>related variables</u>**
- **To fully test, we need all possible combinations (of equivalence classes)**
- **How to reduce testing:**
  - **Coverage analysis**
  - **Assess "important" (e.g. main functionalities) cases**
  - **Test all pairs of relations** (but not all combinations)

Unit Testing

- **Unit Testing**: Test each individual unit
- **Usually done by the programmer**
- **Test each unit as it is developed** (small chunks)
- **Keep test cases/results around** (use Junit or xxxUnit)
  - **Allows for regression testing**
  - **Facilitates refactoring**
  - **Tests become documentation !!**

Test-Driven development

- Write unit-test cases **BEFORE** the code !
- Tests cases “are” / “becomes” requirements
- Forces development in small steps
- **Steps**:

1. **Write test case & code**
2. **Verify (it fails or runs)**
3. **Modify code so it succeeds**
4. **Rerun test case, previous tests**
5. **Refactor until (success and satisfaction)**

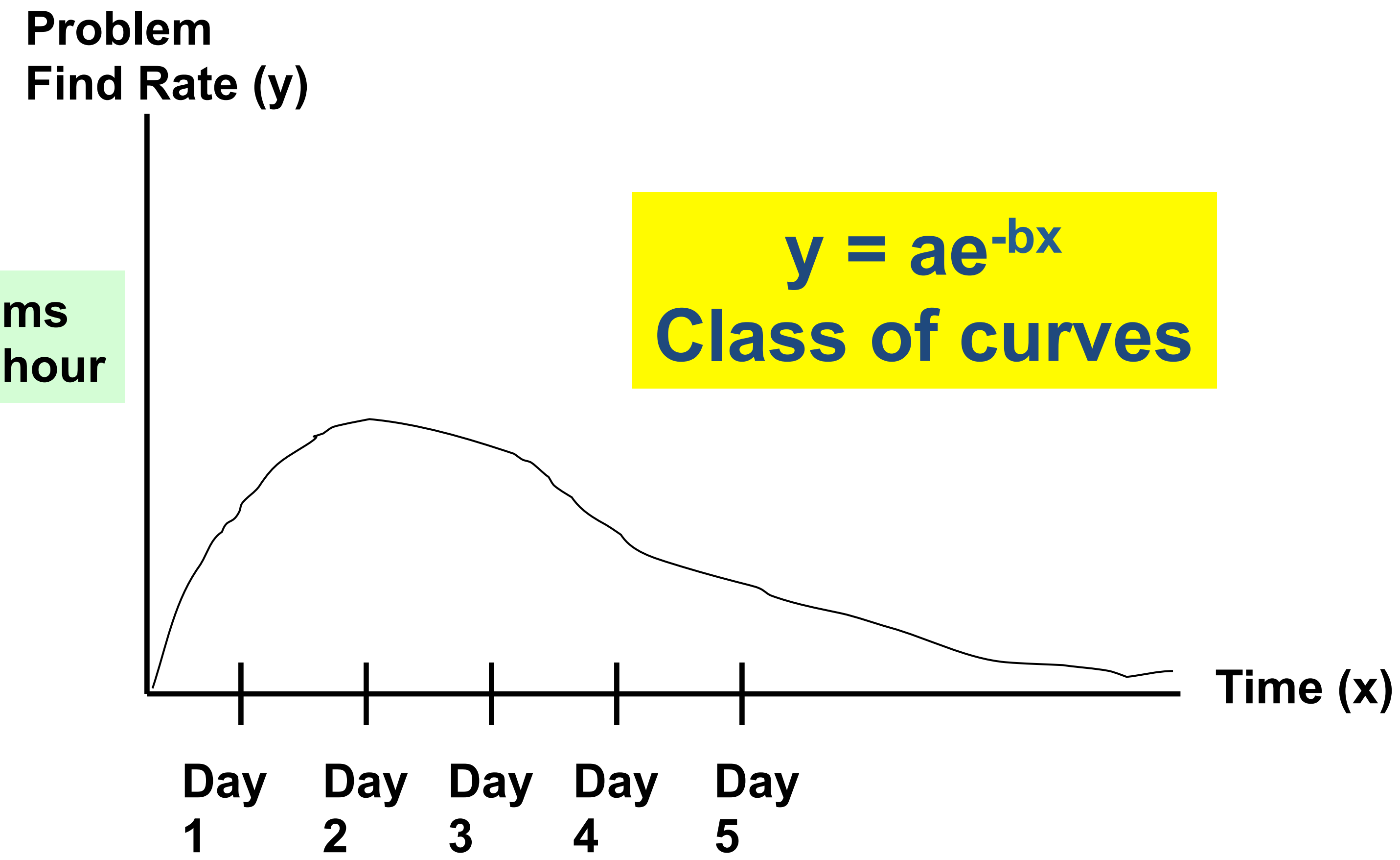
When to stop testing ?

- **Simple answer**, stop when
  - **All planned test cases are executed**
  - **All those problems that are found are fixed**
- **Other techniques:**
  - *Stop when you are not finding any more errors*
  - ***Defect seeding** -- test until all (or % of)the seeded bugs found*
- **NOT -- when you ran out of time -- poor planning!**

Defect Seeding

- Seed the program (component)
  - Generate and scatter with "x" number of bugs &
  - *do not tell the testers.*
  - - set a % (e. g. 95%) of seed bugs found as stopping criteria
- Suppose "y" number of the "x" seed bugs are found
  - If (y/x) > (stopping percentage); stop testing
  - If (y/x) ≤ (stopping percentage), keep on testing
- Get a feel of how many bugs may still remain:
  - Suppose you discovered "u" non-seeded bugs through testing
  - Set   y/x = u/v ; v = (u * x)/y
  - Then there is most likely (v-u) bugs still left in the software.

Problem Find Rate



# of Problems Found per hour

$y = ae^{-bx}$
Class of curves

Decreasing Problem Find Rate

Inspections and **Reviews**

- **Review**: any process involving human testers reading and understanding a document and then analyzing it with the purpose of detecting errors
- **Walkthrough:** author explaining document to team of people
- **Software inspection**: detailed reviews of work in progress, following Fagan’s method.

Software Inspections

- ***Steps:***

  1. **Planning**
  2. **Overview**
  3. **Preparation**
  4. **Inspection**
  5. **Rework**
  6. **Follow-Up**

- **Focused** on finding defects
- **Output**: list of defects
- **Team of**:
  - 3-6 people
  - Author included
  - People working on related efforts
  - Moderator, reader, scribe

Inspections vs Testing

- **Inspections**
  - Partially Cost-effective
  - Can be applied to intermediate artifacts
  - Catches defects early
  - Helps disseminate knowledge about project and best practices

- **Testing**
  - Finds errors cheaper, but correcting them is expensive
  - Can only be applied to code
  - Catches defects late (after implementation)
  - Necessary to gauge quality

Formal Methods

- **Mathematical techniques** used to prove that a program works
- Used for requirements/design/algorithm specification
- Prove that implementation conforms to spec
- Pre and Post conditions
- **<u>Problems</u>:**
  - Require math training
  - Not applicable to all programs
  - Only verification, not validation
  - Not applicable to all aspects of program (e.g. UI or maintainability)

Static Analysis

- Examination of **static structures** of design/code for ***detecting error-prone conditions*** (cohesion --- coupling)
- Automatic program tools are more useful
- Can be applied to:
  - Intermediate documents (but in formal model)
  - Source code
  - Executable files
- Output needs to be checked by programmer