Strategies for Effective Software Testing

ALLIANT
Technology for a better life

Thông tin giảng viên

H. Tran

Agenda

- Introduction (slides 1 – 17)
- The Purpose & Role of Testing (slides 18 -41)
- Different Types of Testing (slides 42 - 57)
- Unit Testing (slides 58 - 98)
- Software Integration (slides 99 – 116)
- System Testing (slides 117-131)
- Regression Testing (slides 132 – 146)
- Requirements Based Testing (slides 147 – 167)
- Observability of Test Results (slides 168 – 175)
- Independent Testing (176 – 189)
- Specific Test Techniques (slides 191 – 288)
  - Equivalence class partitioning (slides 192 – 207)
  - Control flow testing (slides 208 – 220)
  - Data flow testing (slides 221 – 235)
  - Transaction testing (slides 236 – 245)
  - Domain testing (slides 246 – 257)
- Specific Test Techniques (cont’d.)
  - Loop testing (slides 258 – 264)
  - Syntax testing (slides 265 – 269)
  - State machine testing (slides 270 – 281)
  - Load & stress testing (slides 282 – 288)
- The Testing Process & Test Documentation (slides 289 – 305)
- Test Planning & Test Coverage (slides 306 - 323)
- Test Effectiveness (slides 324 – 338)
- Test Reporting (slides 339 – 344)
- Testing Object Oriented Systems (Slides 345 – 369)
- Testing Web Applications (slides 370 – 386)
- Testing Real Time Systems (slides 387- 389)
- Bug Reporting (slides 390 – 396)
- Test Improvement (slides 397 – 401)
- Test Automation (slides 402 – 411)

Schedule

Introduction

- The Purpose & Role of Testing
- Different Types of Testing
- Unit Testing
- Software Integration
- System Testing
- Regression Testing
- Requirements Based Testing

Observability of Test Results

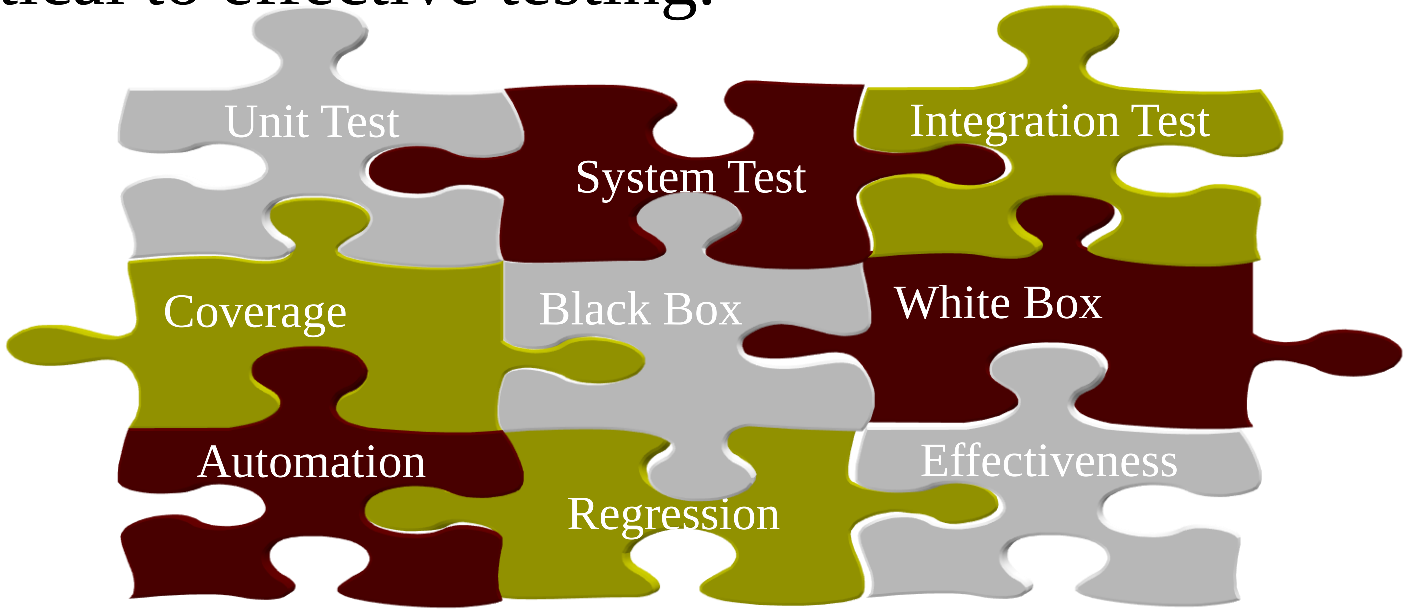
- Independent Testing
- Specific Test Techniques (Equivalence class partitioning, Control flow testing, Data flow testing, Transaction testing, Domain testing, Loop testing, Syntax testing)

Specific Test Techniques

  - State machine testing
  - Load & stress testing
- The Testing Process & Test Documentation
- Test Planning & Test Coverage
- Test Effectiveness
- Test Reporting
- Testing Object Oriented Systems
- Testing Web Applications
- Testing Real Time Systems
- Bug Reporting
- Test Improvement
- Test Automation

Introduction

- Software testing has its own technology, separate from software development.
- Testing software is a more challenging technical problem than building software.
- Understanding the nature and purpose of testing is critical to effective testing.

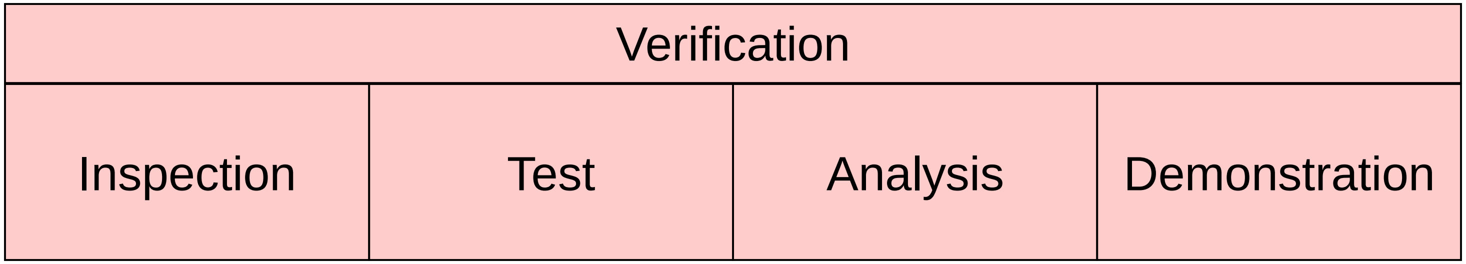


Definitions

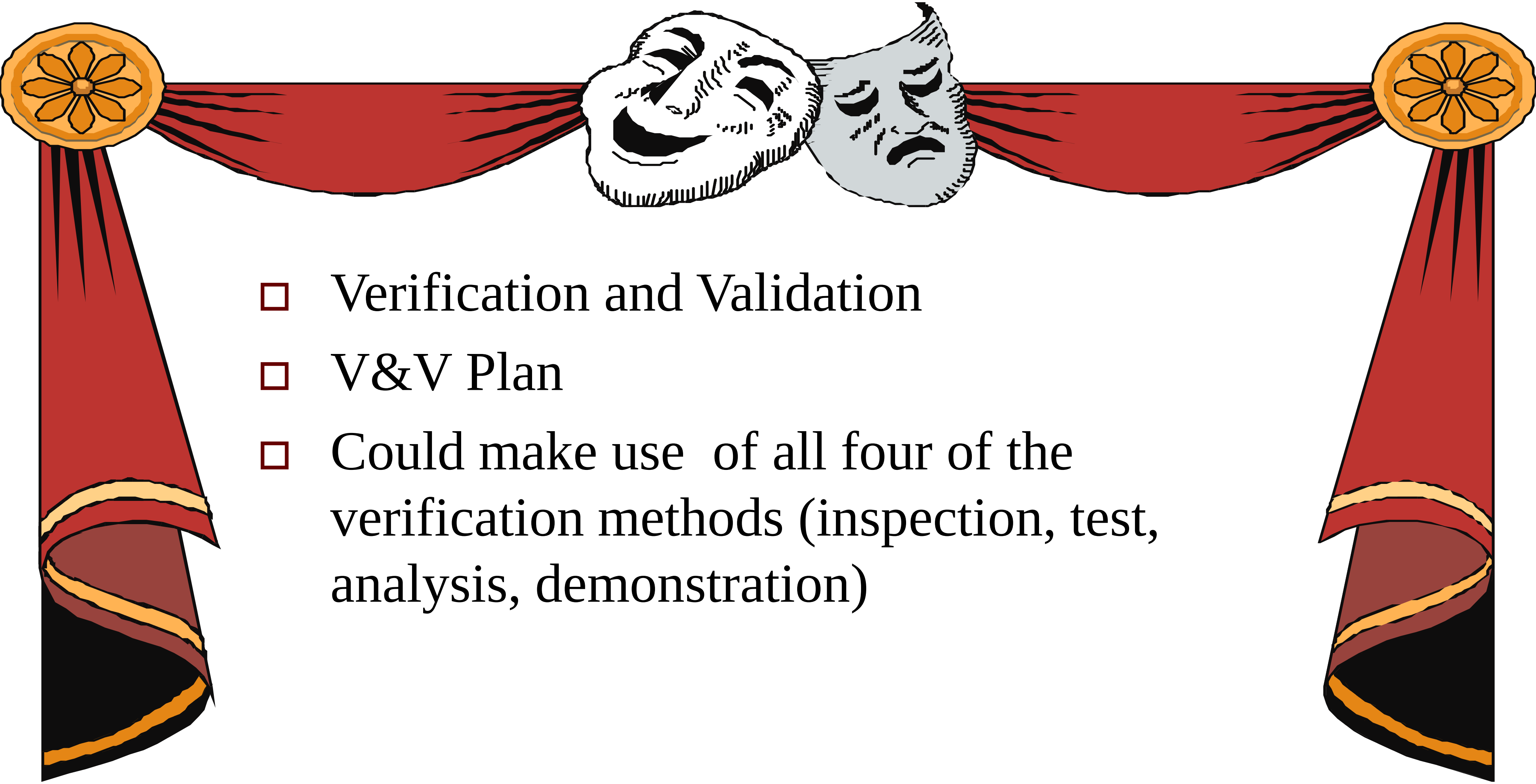
- ***Testing:*** The process of executing a program (or part of a program) with the intention of finding errors.
- ***Verification:*** The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. (IEE Std. 610.12-1990)
- ***Validation:*** The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements. (IEE Std. 610.12-1990)
- ***Debugging:*** To detect, locate, and correct faults in a computer program. Techniques include use of breakpoints, desk checking, dumps, inspection, single-step operation, and traces.

Verification Methods

- *Inspection* – Static observation of the unit in question.
- *Test* – Use instrumentation to observe values that will show correct/incorrect operation.
- *Analysis* – Collect data and do some analysis of it to determine if the unit is operating correctly or not.
- *Demonstration* – Operate the unit to show correctness.

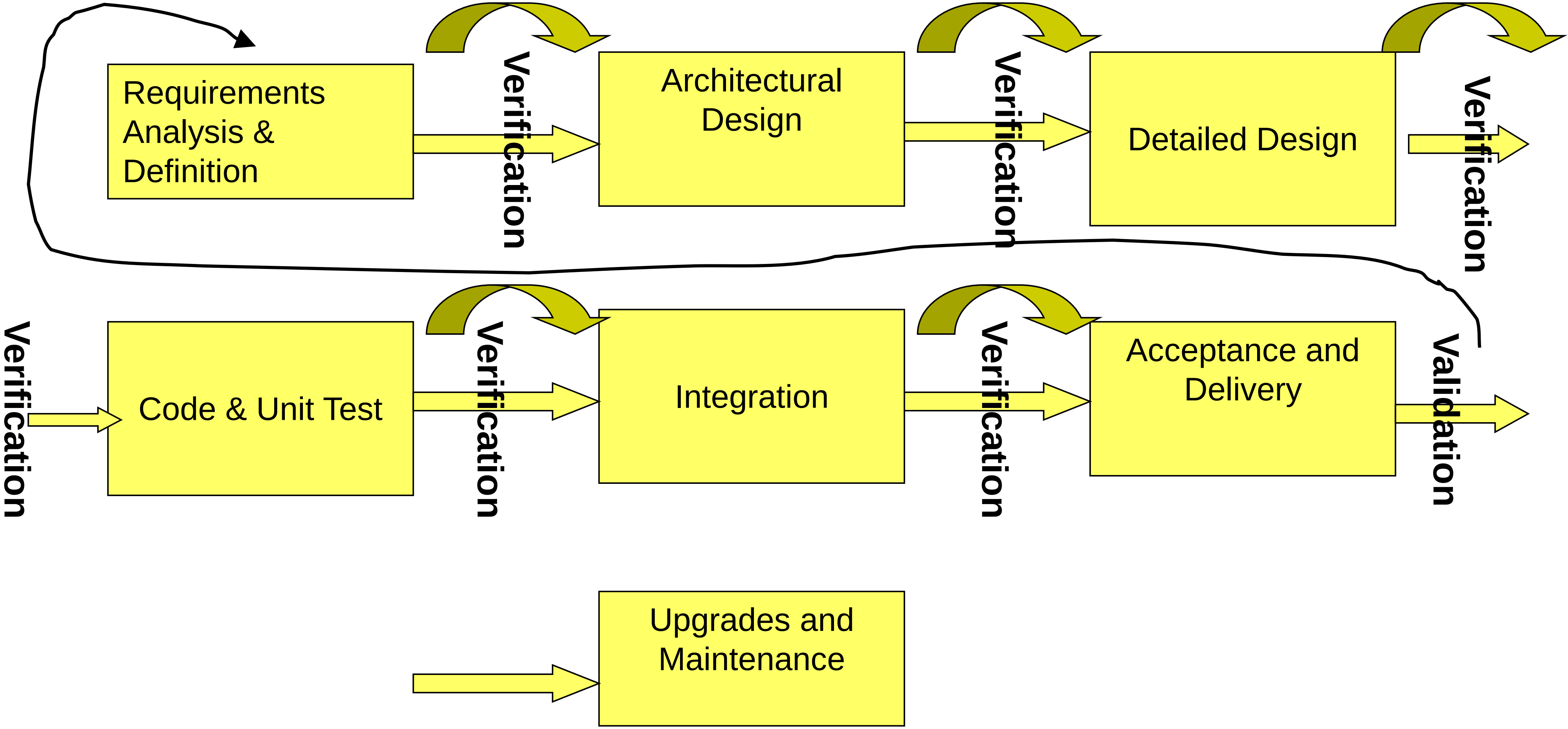


V & V

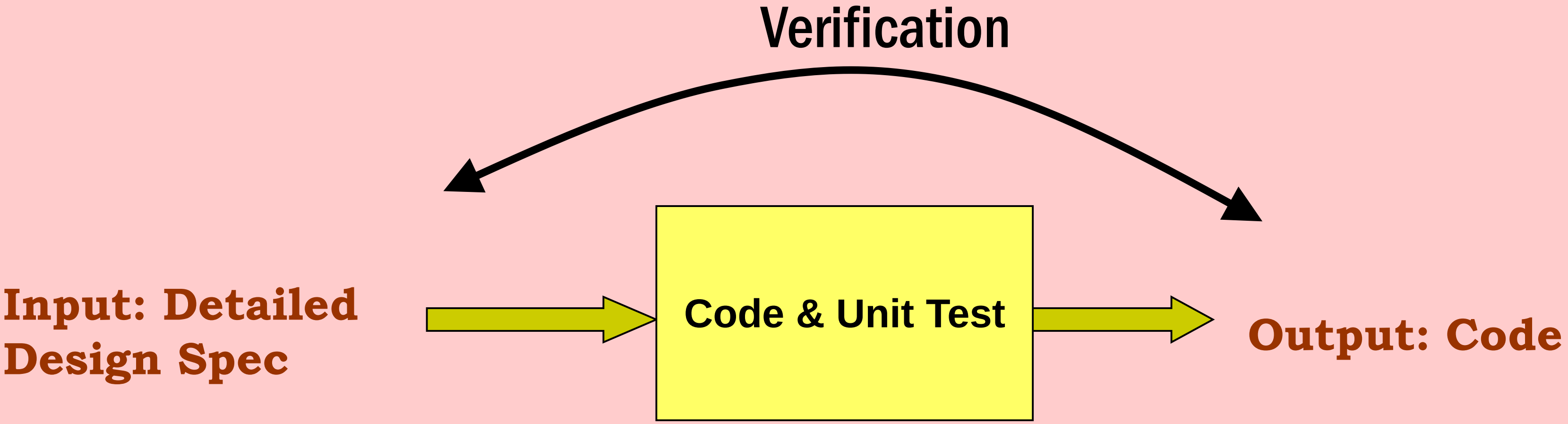


- Verification and Validation
- V&V Plan
- Could make use of all four of the verification methods (inspection, test, analysis, demonstration)

V & V


Requirements Analysis & Definition
Verification
Architectural Design
Verification
Detailed Design
Verification
Verification
Code & Unit Test
Verification
Integration
Verification
Acceptance and Delivery
Validation
Upgrades and Maintenance

Verification - Example



Verification

Input: Detailed Design Spec

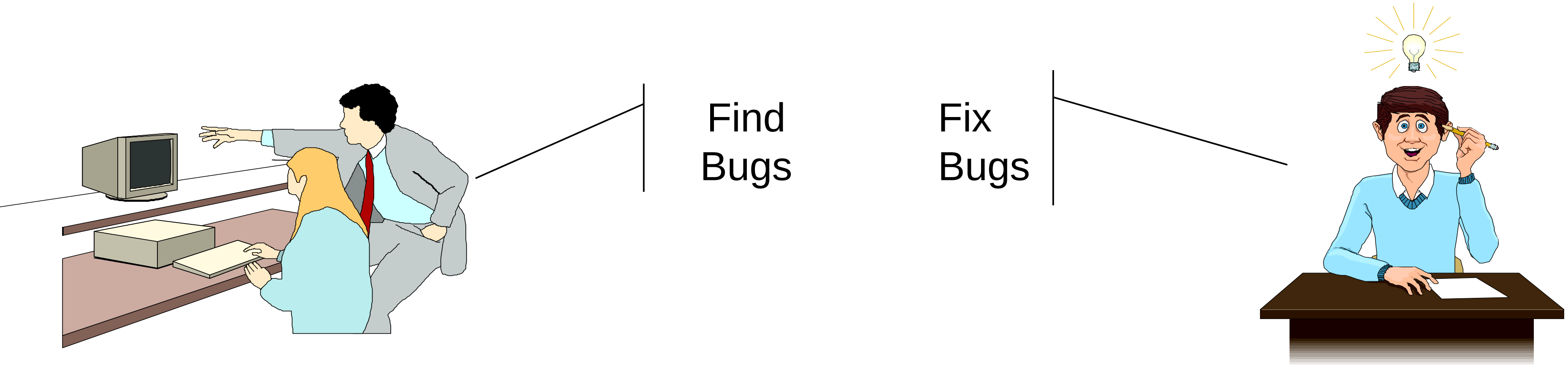
Code & Unit Test

Output: Code

"Verification" would involve activities to determine if the code matches the design.

Testing vs. Debugging

- "***Testing***" finds bugs, and records information about them, but does not fix them
- "***Debugging***" determines the location of the error or misconception that caused the program to fail and designs and implements the program changes needed to correct the error.



Find Bugs

Fix Bugs

Levels of Testing Awareness

Phase 0: There is no difference between testing and debugging.

Phase 1: The purpose of testing is to show that the software works.

Phase 2: The purpose of testing is to show that the software does not work.

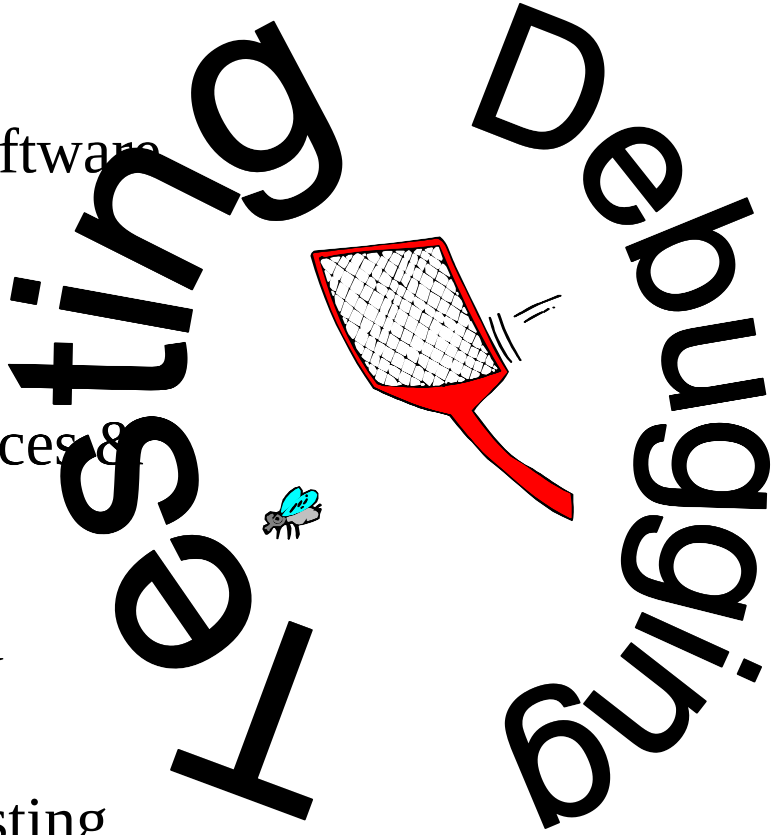
Phase 3: The purpose of testing is not to prove anything, but rather to reduce the perceived risk of not working to an acceptable level.

Phase 4: Testing is not an act. It is a mental discipline that results in low-risk software without much testing effort.

Source: Software Testing Techniques, Boris Beizer

Phase 0 Thinking

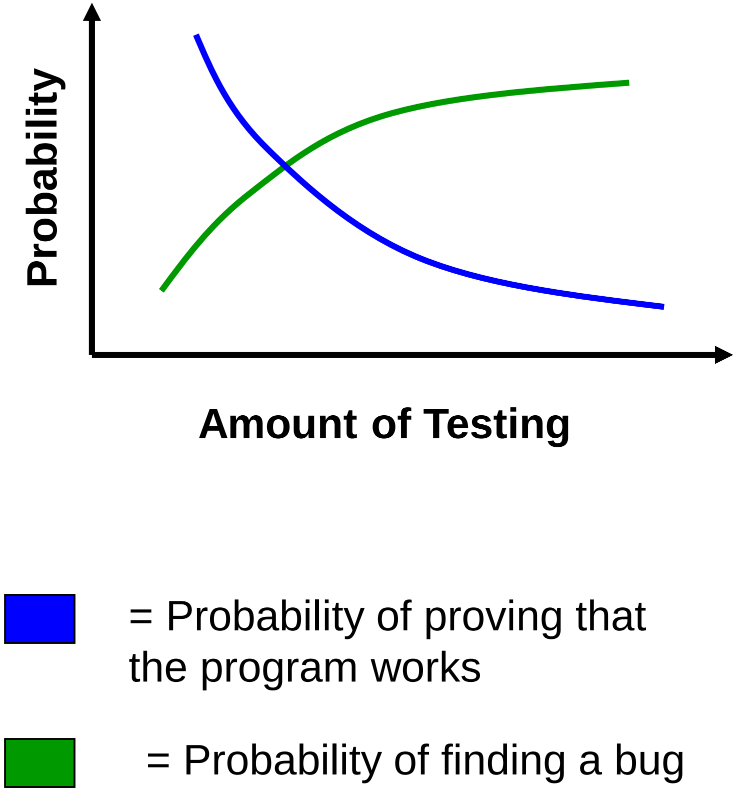
- Testing = Debugging
- Denies that testing matters.
- Was the norm in the very early days of software development, until testing emerged as a discipline.
- Was appropriate for an environment characterized by scarce computing resources & expensive hardware.
- Relatively low-cost software, single programmers, small projects, throw-away software.
- Today, it is the greatest barrier to good testing and quality software.



Source: Software Testing Techniques, Boris Beizer

Phase 1 Thinking – The Software Works

- □ Recognizes the distinction between debugging and testing.
- □ Dominated thinking until the late 1970’s
- □ Fallacy: It can only takes one test to prove that the software doesn’t work, but an infinite number of tests won’t prove that it does work.
- □ The probability of showing that the software works *decreases* as more tests are performed.
- □ So if you want to prove that a program works, test less!



Source: Software Testing Techniques, Boris Beizer

Phase 2 Thinking - The Software Doesn’t Work

- Trying to find bugs.
- Leads to an independent test group.
- An effective test is one that has a high probability of finding a bug.
- More testing will always find more bugs.
- Problem: We don’t know when to stop testing.

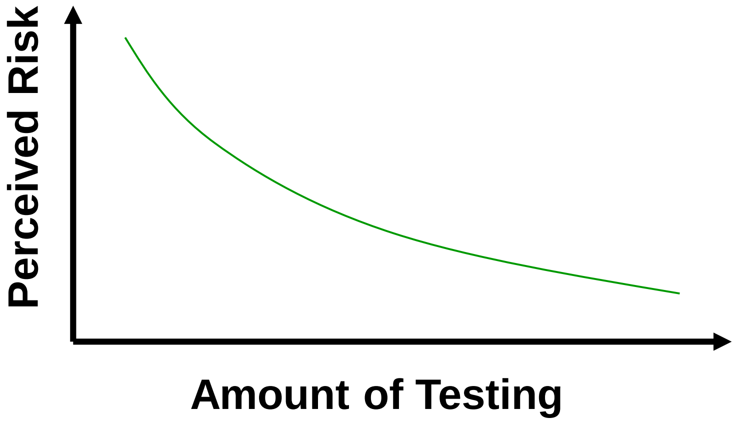
Developers

Testers

Source: Software Testing Techniques, Boris Beizer

Phase 3 Thinking – Test for Risk Reduction

- If the bugs found in testing are fixed, the product’s quality/reliability is improved.
- If extensive testing finds no bugs, the perceived quality/reliability of the product goes up.
- The more we test with effective tests, the higher our confidence in the software.
- Testing is a risk reduction activity.
- Test until the risk is low enough.



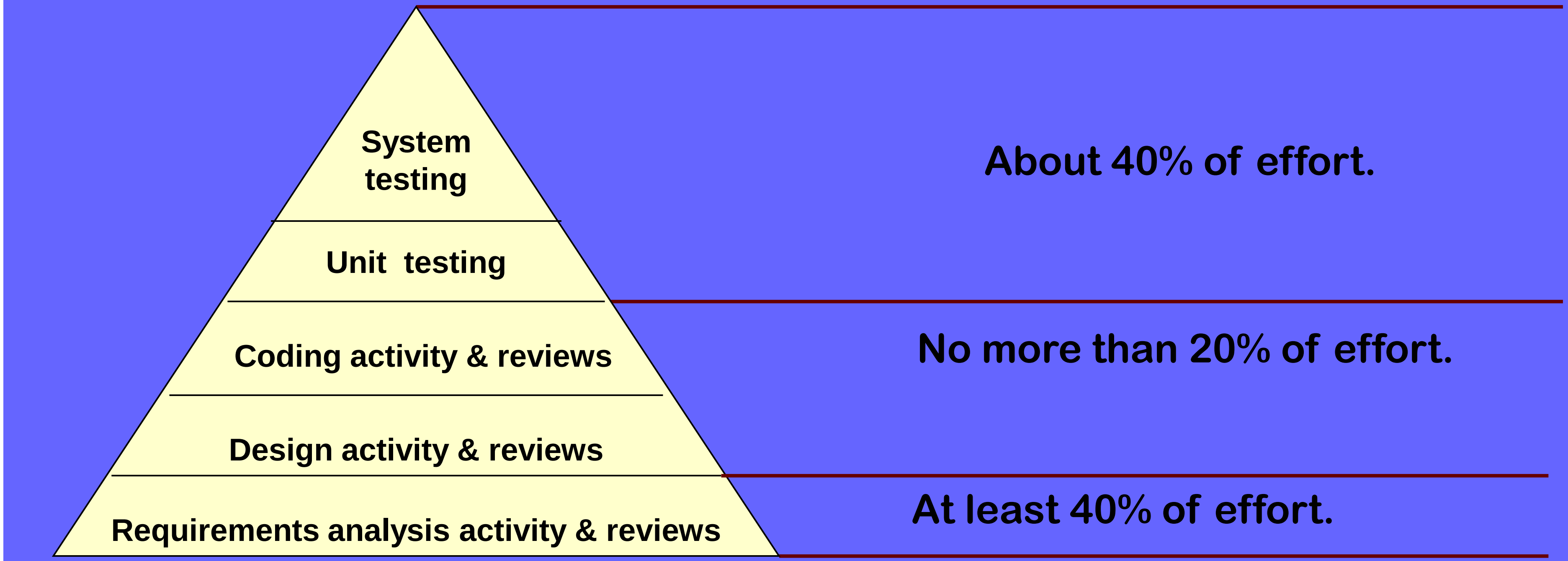
Source: Software Testing Techniques, Boris Beizer

Phase 4 Thinking – A State of Mind

- Driven by a knowledge of what testing can and can’t do.
- “Testability” is designed into the software.
- Why “testability”:
  - Reduces the labor of testing
  - Testable code has fewer bugs.

Source: Software Testing Techniques, Boris Beizer

Relative Percent of Total Effort



System testing

Unit testing

Coding activity & reviews

Design activity & reviews

Requirements analysis activity & reviews

About 40% of effort.

No more than 20% of effort.

At least 40% of effort.

Source: *Tactical Software Reliability*, SEMATECH, 1995

Getting Started

What is the purpose of testing?

The Purpose of Testing

Question: What is the purpose of software testing?
Answer: To find faults.

Formal definition: Software testing is defined as the execution of software to find its faults.

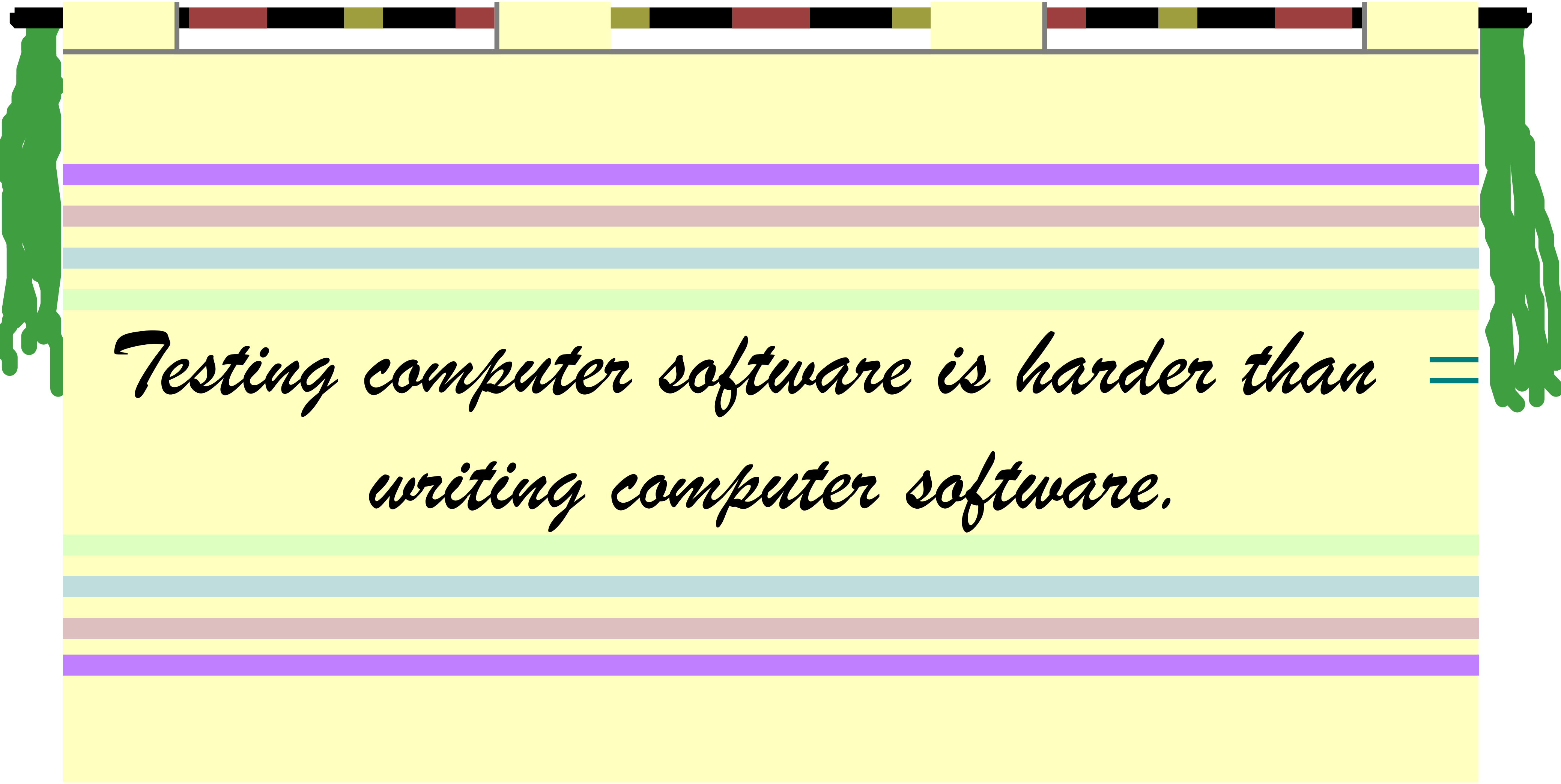
“Program testing can be used to show the presence of bugs, but never their absence”

Edsgar Dijkstra, 1969

How Is Testing Different

- Need a different mind-set. Must assume that there are bugs in the code.
- The goal is to break the software, not to show that it works properly.
- Testing does not prove the absence of errors.
- Testing by itself does not improve the quality of the software. To improve software, don’t test more; develop better.

Relative Difficulty



Testing computer software is harder than writing computer software.

Software Testing - Considerations

- With large systems, it is always true that more testing will find more bugs.
- The question is not whether all the bugs have been found, but whether the software is sufficiently “good” to stop testing.
- Software testing presents a problem in economics.
- One of the most difficult problems in software testing is knowing when to stop.

The Proper Role of Testing

Example program:

```
Begin
Read
   (AAAAAAAAAA)
Print
End
```

Number of Input Conditions:
26^{10}

Test time:

- Assume automated testing
- 1 micro-second per test case
- 4.5 million years
- Doesn’t include error conditions.
- Loops make it even worse.

Source: Managing the Software Process, Watts Humphrey

Example

- For the previously defined program, suppose 1200 test cases have been run.
- No bugs have been found.
- How good is the software?

Example - continued

- □ Here are the test cases that were run:
  - AAAAAAAAAA through AAAAAAAAZZ
  - AAAAAAABAA through AAAAAAAAZZ

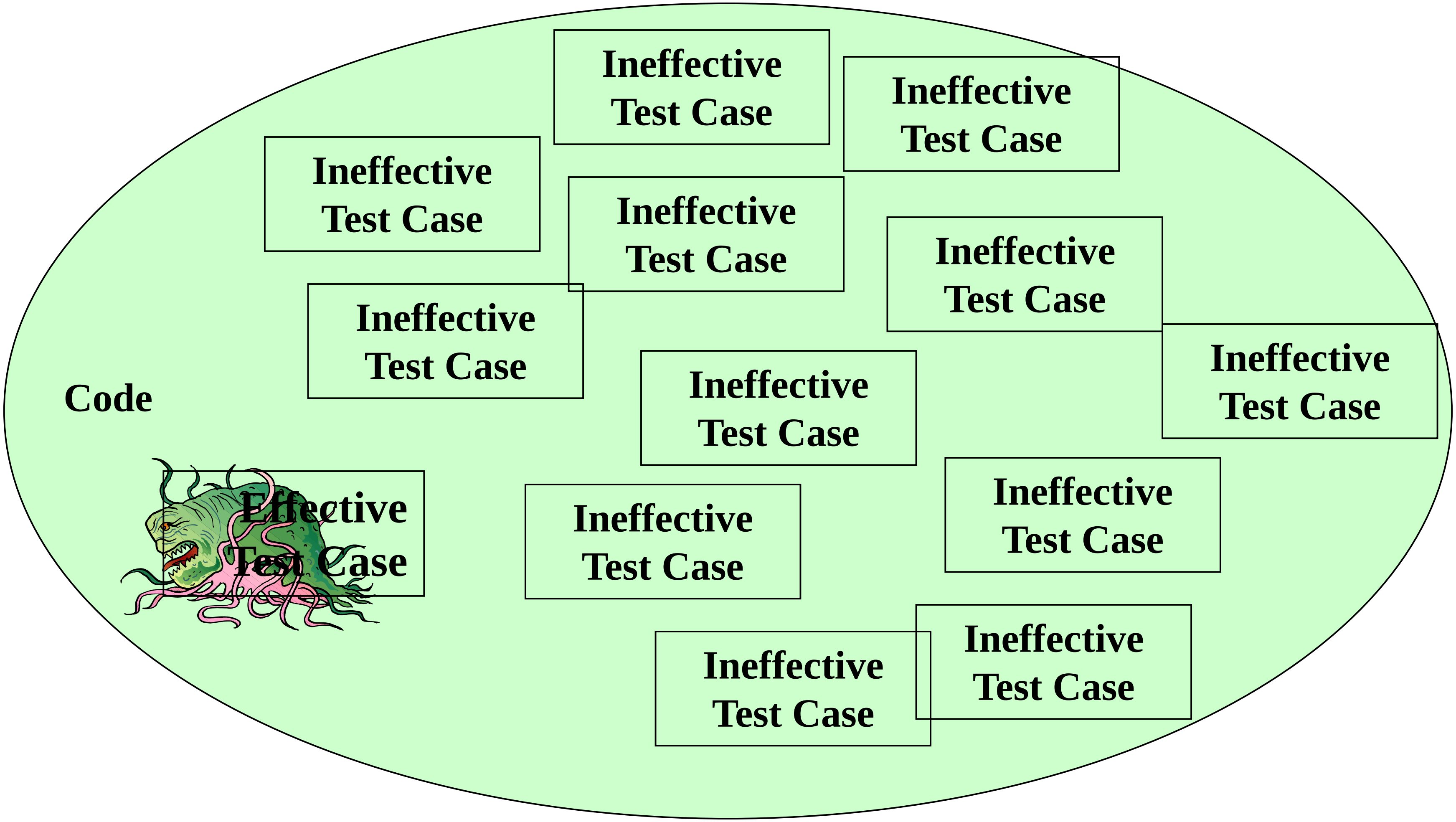
- □ Remember: No bugs have been found by these test cases.

- □ *What is your confidence level in this software?*

The Testing Challenge

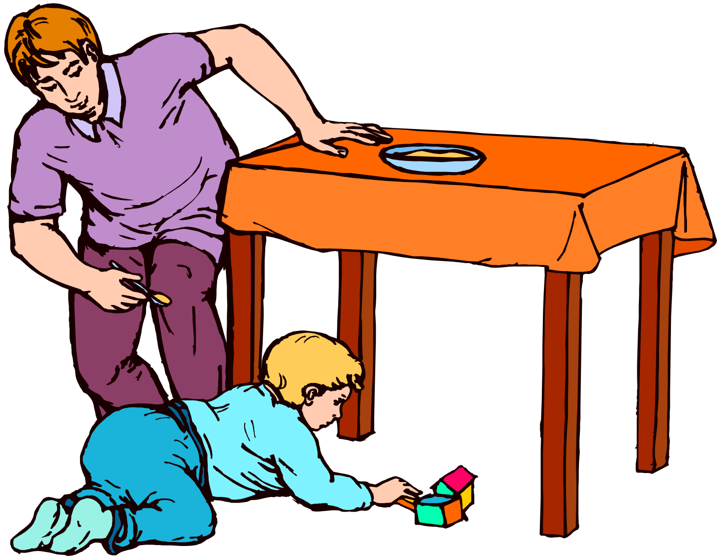
- The #1 Issue in software testing, by far, is to decide which test cases will be run such that the testing is effective.
- What is an effective test: One that finds a bug.
- Repeatedly running a test that does not find a bug can be wasted effort. (Possible exception: Regression Testing.)
- The science of testing is picking the test cases most likely to find errors.

An Effective Test Case



More on Effective Testing

- The key to effective testing is to design the test coverage such that bugs are found.
- Bugs are not distributed evenly throughout the code.
- Use testing strategies that will direct the test cases to the areas of the software where it is likely that bugs will be found.
- Think of testing as a “bug hunt”.
- That is what this class is all about.



Mean Fault Densities

Phase	**Faults/KLOC**
Coding (after compilation)	**99.5**
Unit Test	**19.7**
System Test	**6.01**
Operation	**1.48**

(Microsoft: O. 5 defects/kloc)

Sources: Musa, John d., et al, Software Reliability, McGraw Hill, 1990, p. 118) & McConnel, Steve, Code Complete, Microsoft Press, 1993

The Testing Dilemma

- More testing will always find more bugs.
- How much testing is enough?
- How do I know when to stop testing:
- When all the bugs have been found?
- When we run out of time.
- When we run out of money.
- Management says, “Stop”.
- The customer says, “Ship it”.
- We get tired.
- Etc.

When To Stop Testing

- The question is not whether all the bugs have been found, but whether the software is sufficiently “good” to stop testing.
- The trade-off should consider:
  - The probability of finding more bugs in test,
  - The marginal cost of doing so,
  - The probability of the users encountering the remaining bugs,
  - The resulting impact of these bugs on the user.

Source: Managing the Software Process, Watts Humphrey

When To Stop Testing

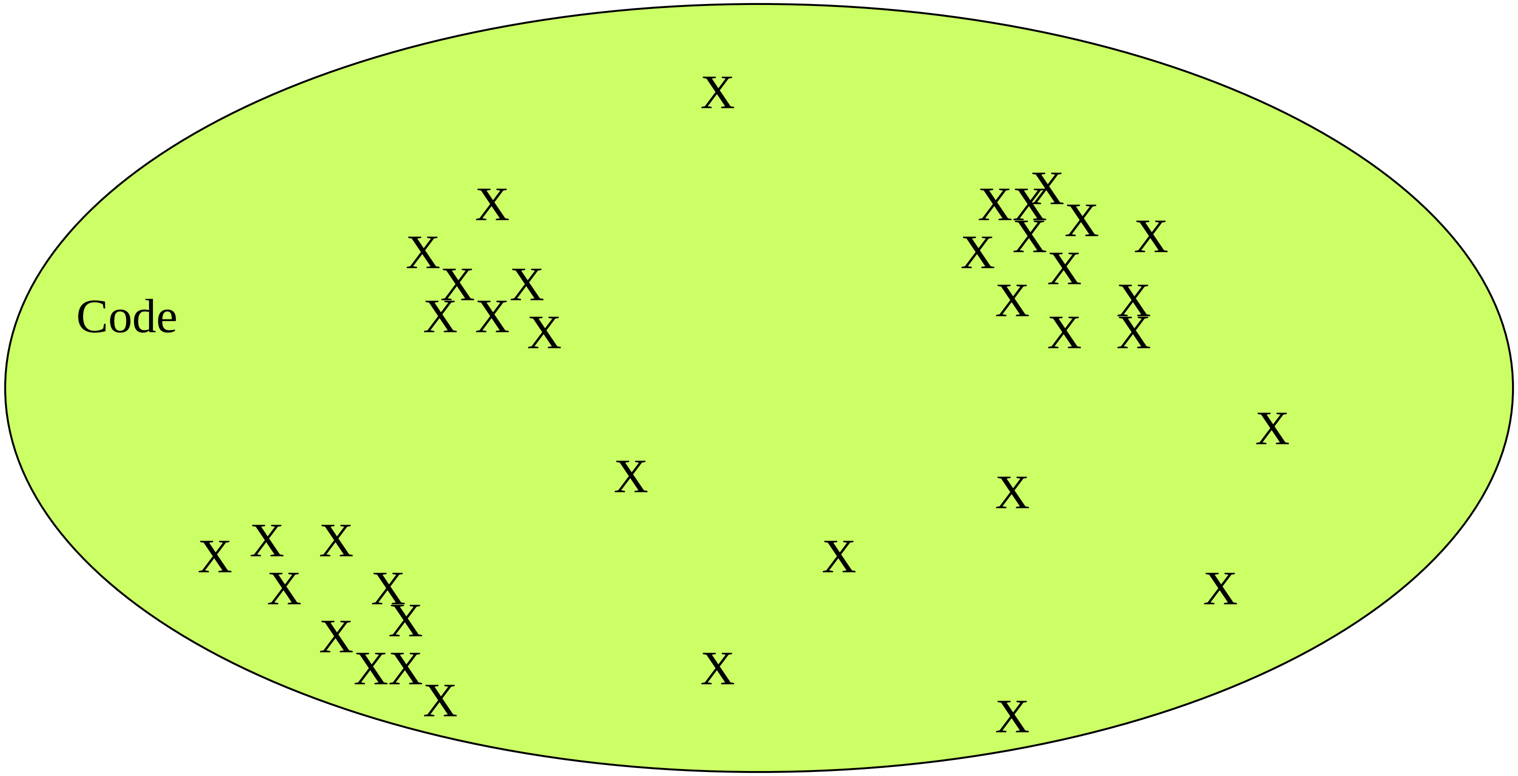
- Lack of Test Data
  - The general lack of data on the software process inhibits our ability to make this trade-off intelligently.
  - Usually, testing is stopped when testing time is used up, even when there is ample evidence that many more bugs remain to be found.
  - The purpose of this course is to enable a better determination of what is an adequate amount of testing and how to write effective test cases.

Source: Managing the Software Process, Watts Humphrey

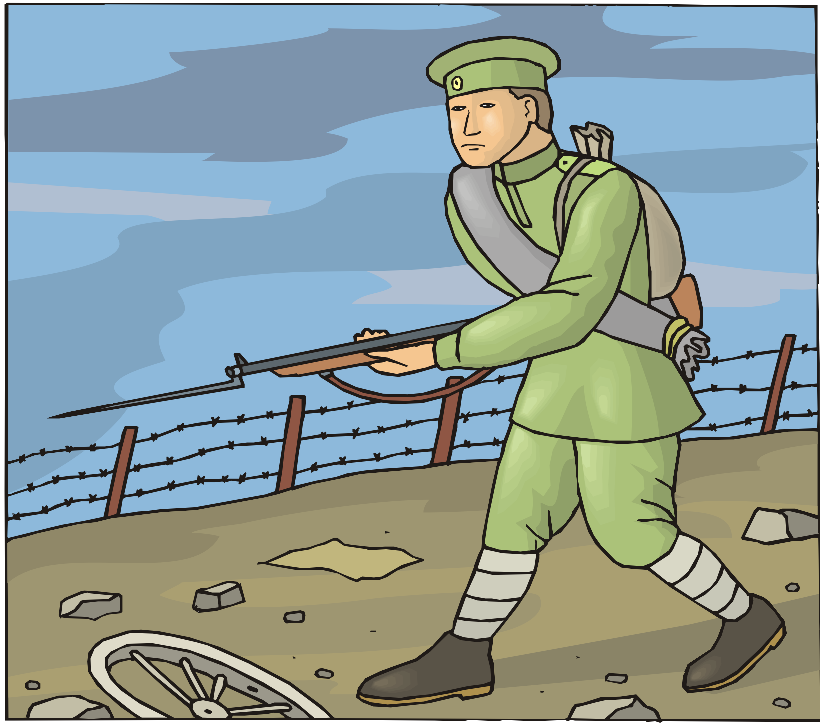
Distribution of Bugs in Software

- A common view is that all untested code has a roughly equal probability of containing defects, but this is usually not true.
- The incidence of bugs in untested code varies widely.
- Bugs are not evenly distributed in the code.
- Once again: Testing is a bug hunt.

Distribution of Bugs in Software



Testing Is A Bug Hunt



Error-prone Modules

- A very common phenomenon.
- Will occur in all large systems unless steps are taken to prevent it.
- IBM OS/360: 4% of the modules contained 38% of the defects.
- IBM PARC (Database Products): 57% of defects in 31% of the modules.
- Confirmed at AT&T, ITT, HP, etc.

Source: Applied Software Measurement, Capers Jones

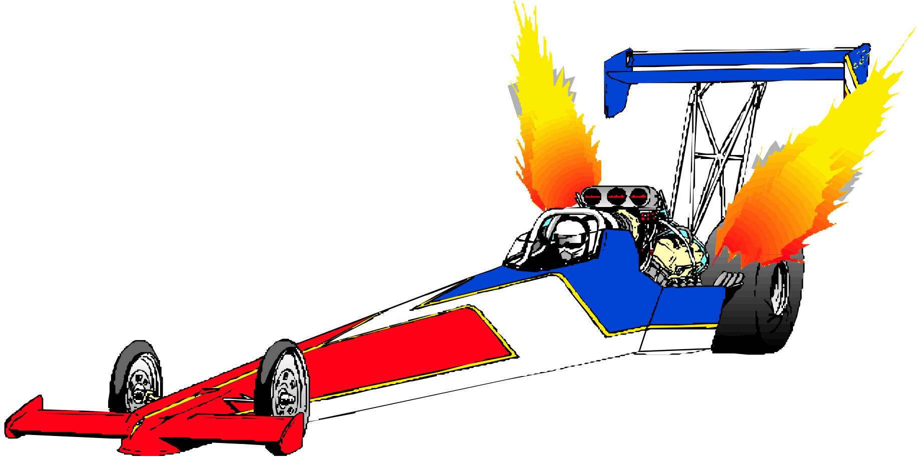
Error-prone Modules – Causes

- Excessive schedule pressure.
- Excessive complexity:
  - Failure to use structured techniques
  - Intrinsic nature of the problem to be encoded
- Excessive size of individual modules (>500 statements)
- Failure to test the module after the code was complete.

Source: Applied Software Measurement, Capers Jones

Error-prone Modules – Lack of Testing

- Created very late in the development life cycle.
- Rushed into production.
- Specifications and test case libraries not updated.



Source: Applied Software Measurement, Capers Jones

Axioms of Software Testing

- A good test case is one that has a high probability of detecting a previously undiscovered bug.
- One of the most difficult problems in testing is knowing when to stop.
- It is impossible to test your own code.
- A necessary part of every test case is a description of the expected output.
- Avoid non–reproducible or “on-the-fly” testing.
- Write test cases for invalid as well as valid input conditions.
- Thoroughly inspect the results of each test.
- As the number of detected defects in a piece of software increases, the probability of the existence of more undetected defects also increases.
- Assign your best software engineers to testing.
- Ensure that testability is a key objective in software design.
- Testing, like almost every other activity, must start with objectives.

Source: Managing the Software Process, Watts Humphrey

Is Testing Easy?

- Glenford Myers had a group of experienced programmers test a program with 15 known defects.
- The average programmer found 5 of the 15.
- The best found 9 of the 15.

How Much Benefit Do We Get From Testing?

	Defect Removal Efficiency - %		
	Lowest	Median	Highest
1. No design inspections No code inspections No quality assurance No formal testing	30	40	50
2. No design inspections No code inspections No quality assurance *Formal testing*	37	53	60
3. Formal design inspections *Formal code inspections* *Formal quality assurance* *Formal testing*	95	99	99

Source: Applied Software Measurement, Capers Jones

Two Fundamental Approaches

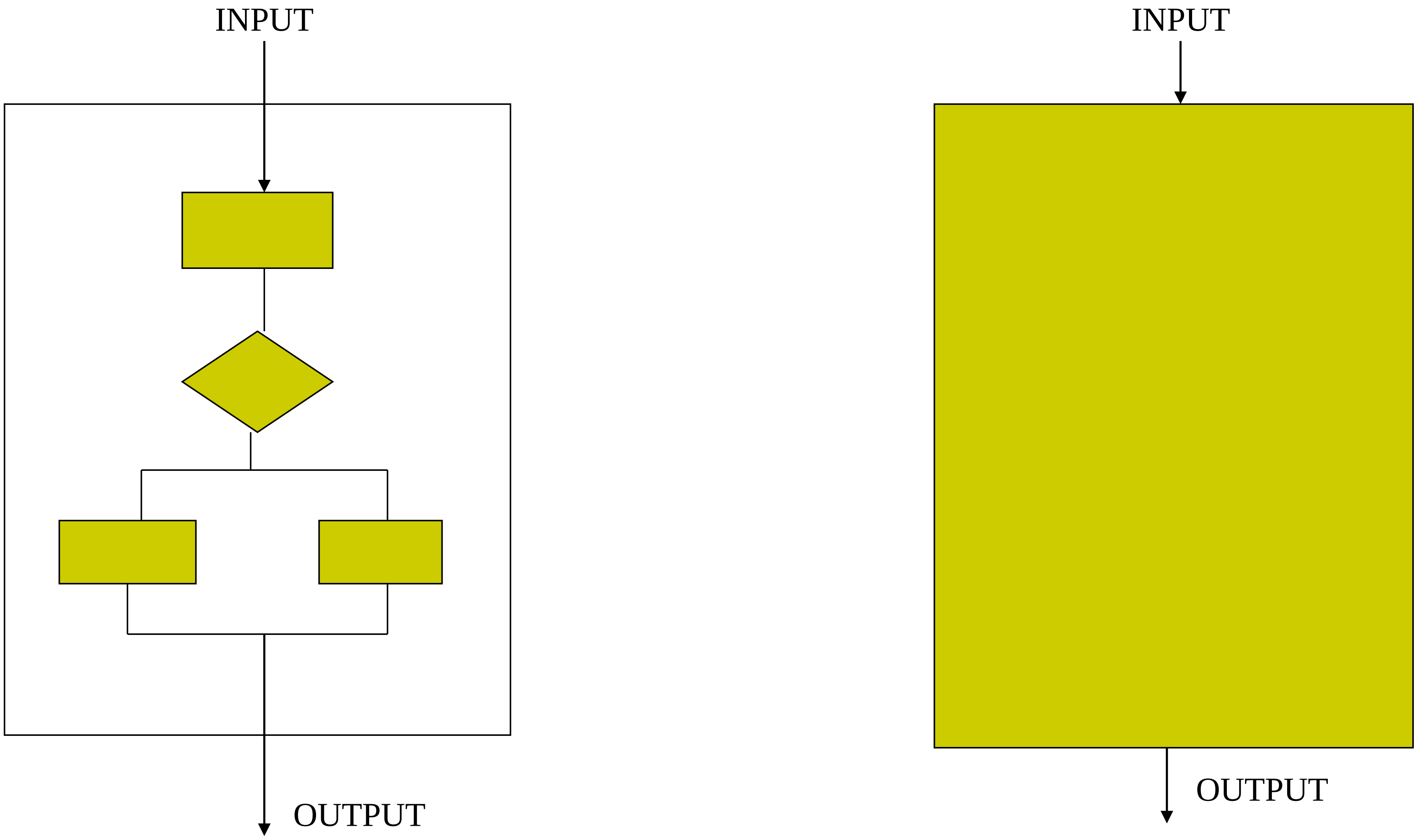
Structural Testing

- Also known as: white box testing.
- Test is based on the structure of the code.

Functional Testing

- Also known as: black box testing, functional testing, behavioral testing.
- Test is based on the behavior of the software. The code itself is not looked at.

White Box vs. Black Box Testing



White Box (Structural) Testing

- Examines internal software design.
- Requires the tester to have detailed knowledge of the software structure.
- Static structural analysis
  - Complexity
  - Code coverage
  - Paths
- Dynamic structural analysis
  - Call pairs
  - Control Flow
  - Data flow
  - Memory leaks

White Box Testing - Definition

A test case design method that uses the control structure of the procedural design to derive test cases.

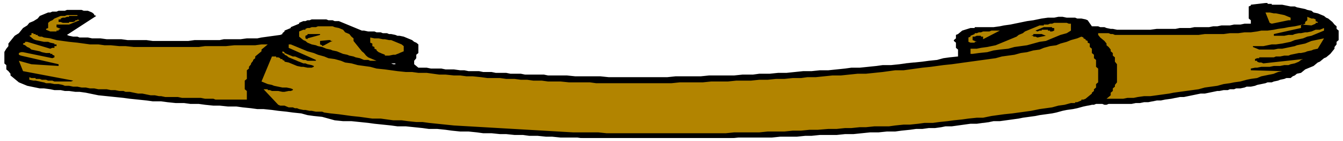
Source: Software Engineering, Roger Pressman

White Box Testing

- Driven by program structure
- Looks at the implementation details.
- Concerned with:
  - Programming style
  - Control method
  - Language
  - Database design
  - Coding details

Black Box (Functional) Testing

- Based upon functional operation, does not require knowledge of the code or software structure.
- Functional test coverage (requirements tracing).
- Examples:
  - Requirements based testing
  - Use case testing
  - State machine testing
  - Boundary value testing (domain testing)
  - Equivalence class partitioning
  - Syntax testing
  - Data flow testing



Example – Requirements Based Testing

Software Requirements Specification

The software shall recognize three types of triangles: Isosceles, Equilateral, Scalene.

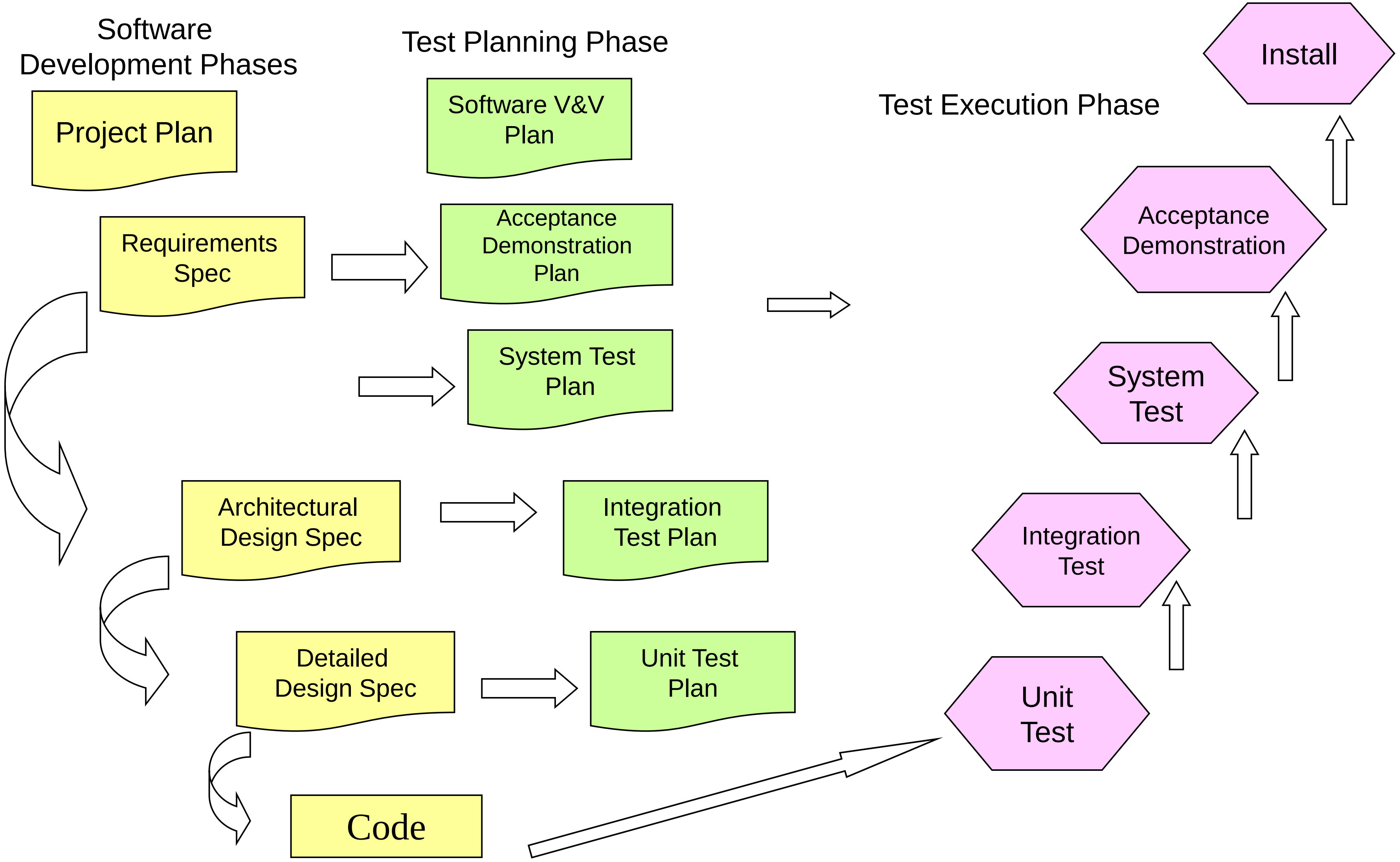
Test cases:

TS1 - Input: Side 1 = a, side 2 = a, side 3 = b
Expected result: Triangle identified as isosceles.

TS2 – Input: Side 1 = a. side 2 = a, side 3 = a
Expected result: Triangle is identified as equilateral.

TS3 – Input: Side 1 = a, side 2 = b, side 3 = c.
Expected result: Triangle is identified as scalene.

Different Types of Testing – V-Model



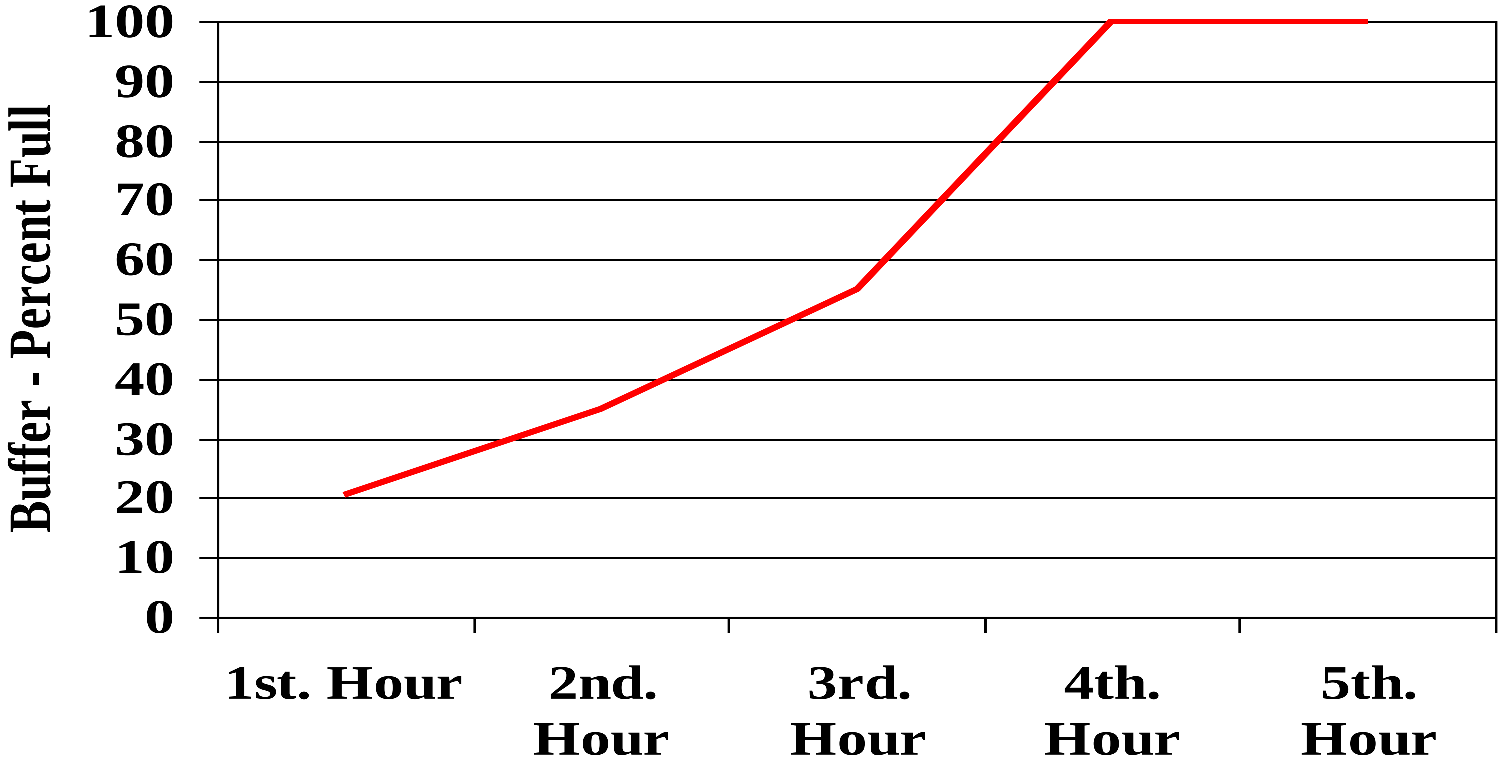
Types of Software Testing

- **Unit or Module Tests**
  - Examines single modules or “units”
  - Unit: Lowest level of individually compilable code.
  - Conducted in isolated or special test environments.
  - Makes use of “test ware” or stubs and drivers
- **Integration Testing**
  - Examines the interfaces between previously tested units.
- **System or Qualification Testing**
  - System Testing: Examines the total system as a whole.
  - Qualification Test: Validates the system to its initial requirements spec
- **Acceptance (Test) Demonstration**
  - Shows that the system is ready to be shipped to the customer.
  - Conducted on the complete system after all other testing has been done.
- **Installation Testing**
  - Examines installability and operability aspects of the system.
- **Regression Testing**
  - Testing conducted on the whole system after some code changes have been made.
  - Looks for new bugs in the unchanged part of the system.

Another Type - Continuous Run Testing

- Performed on the whole system so it is a type of system test.
- Introduces the time factor.
- Some bugs don’t show up until the system has been in continuous operation for some amount of time:
  - Buffers overflow,
  - Queues fill-up,
  - Latency,
  - Corrupt data is propagated throughout the system,
  - Etc.
- Reliability calculations:
  - Mean Time Between Failure (MTBF)
  - True “reliability” (probability of failure)
  - Particularly pertinent in real time systems or embedded control applications.

Continuous Run Testing



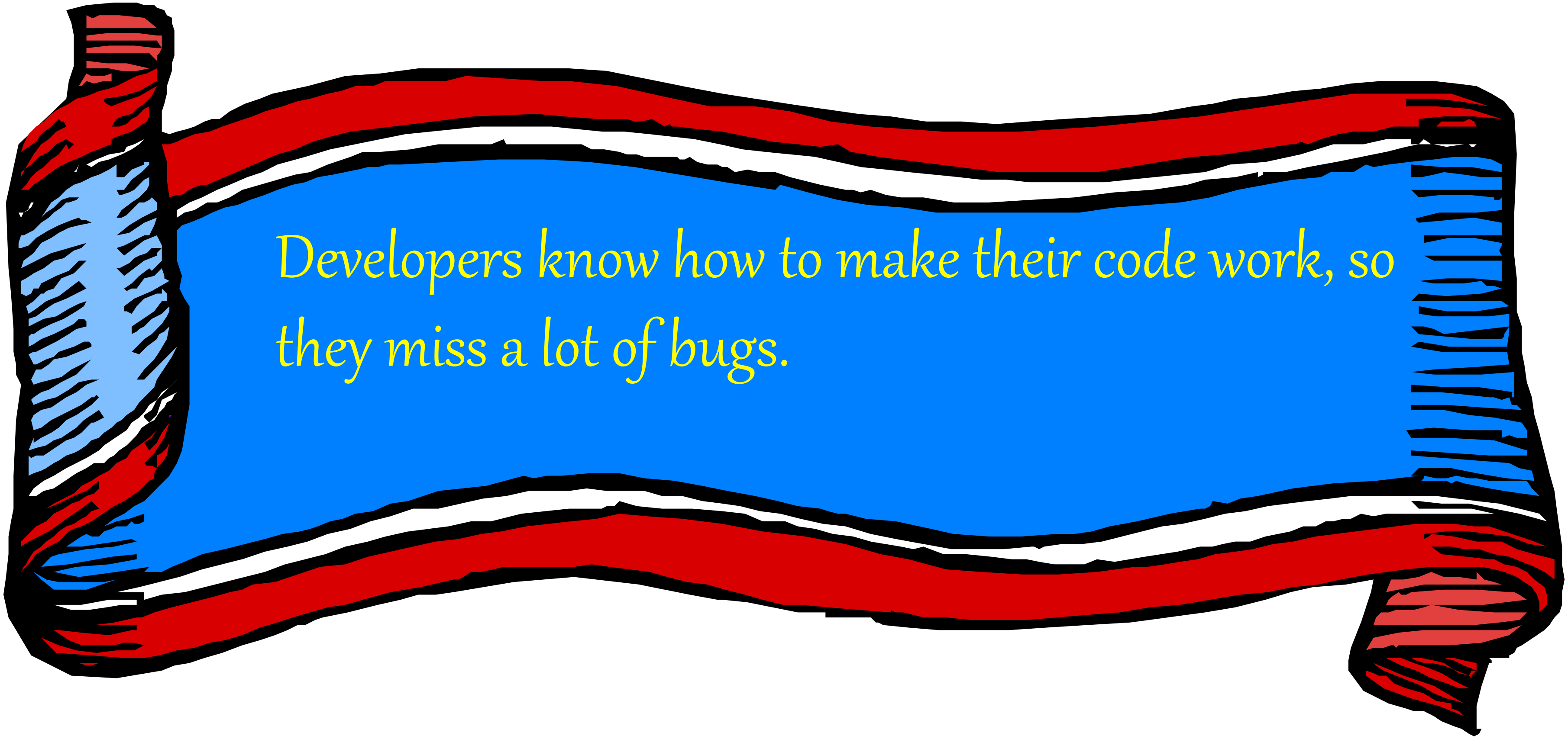
Developer Testing vs. Independent Testing

- SSome testing is done by the code developers.
- SSome testing is done by an independent testing group.
- TThe presence of an independent test group does not mean that the developers stop testing.
- NNeed both.

Trying to Proof Read Your Own Work

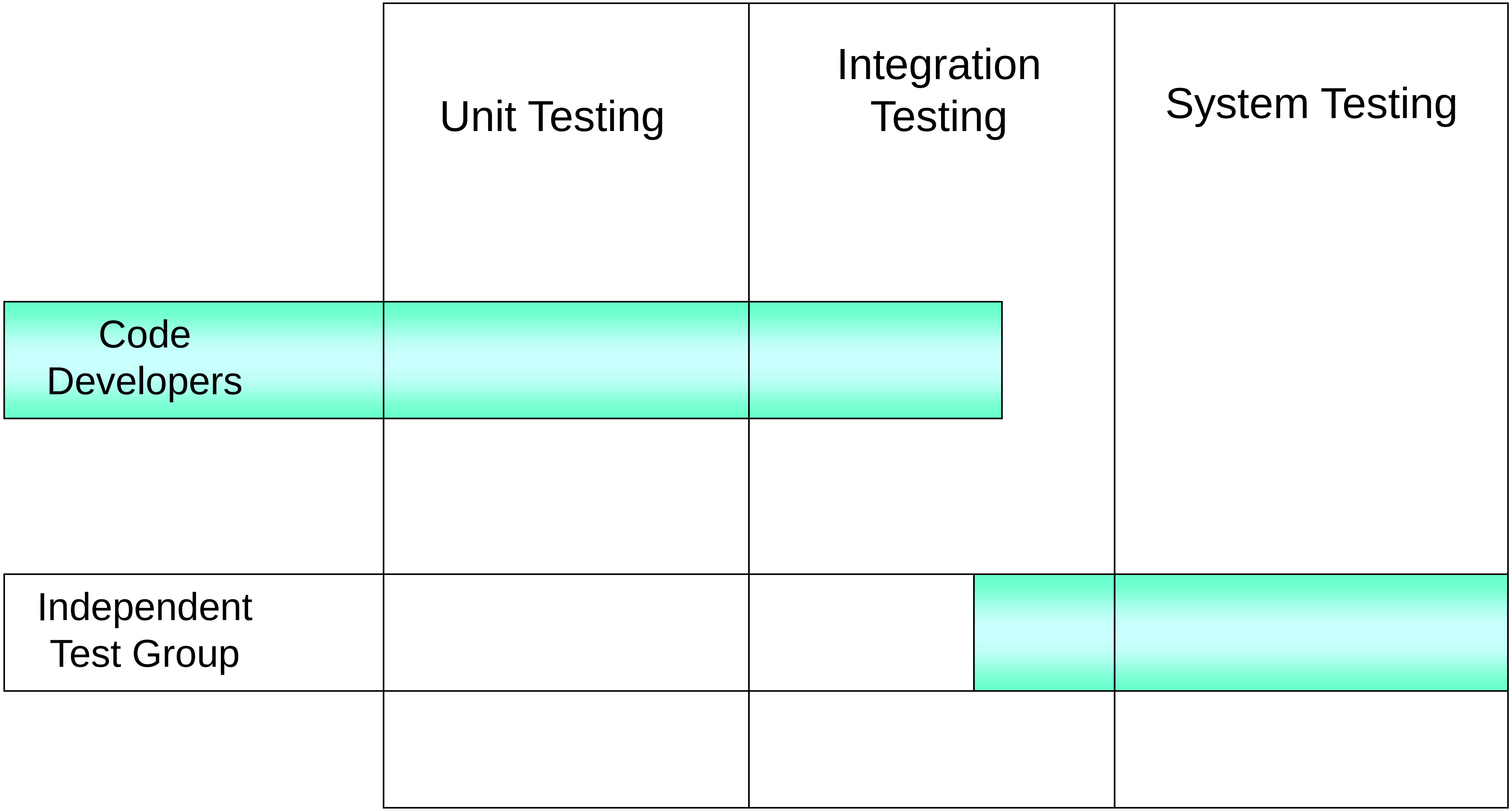
- Developers are usually inherently incapable of effectively their own code:
  - Bug guilt
  - Mind set
  - Proof reading your own work
  - Separate testing from program design and implementation.
- Usually advisable after unit test to have an independent test group take over the responsibility for testing.
- The role of the independent test group is to find as many bugs as possible.

The Need for Independent Testing



Developers know how to make their code work, so they miss a lot of bugs.

Organizational Roles - Testing



Unit Testing

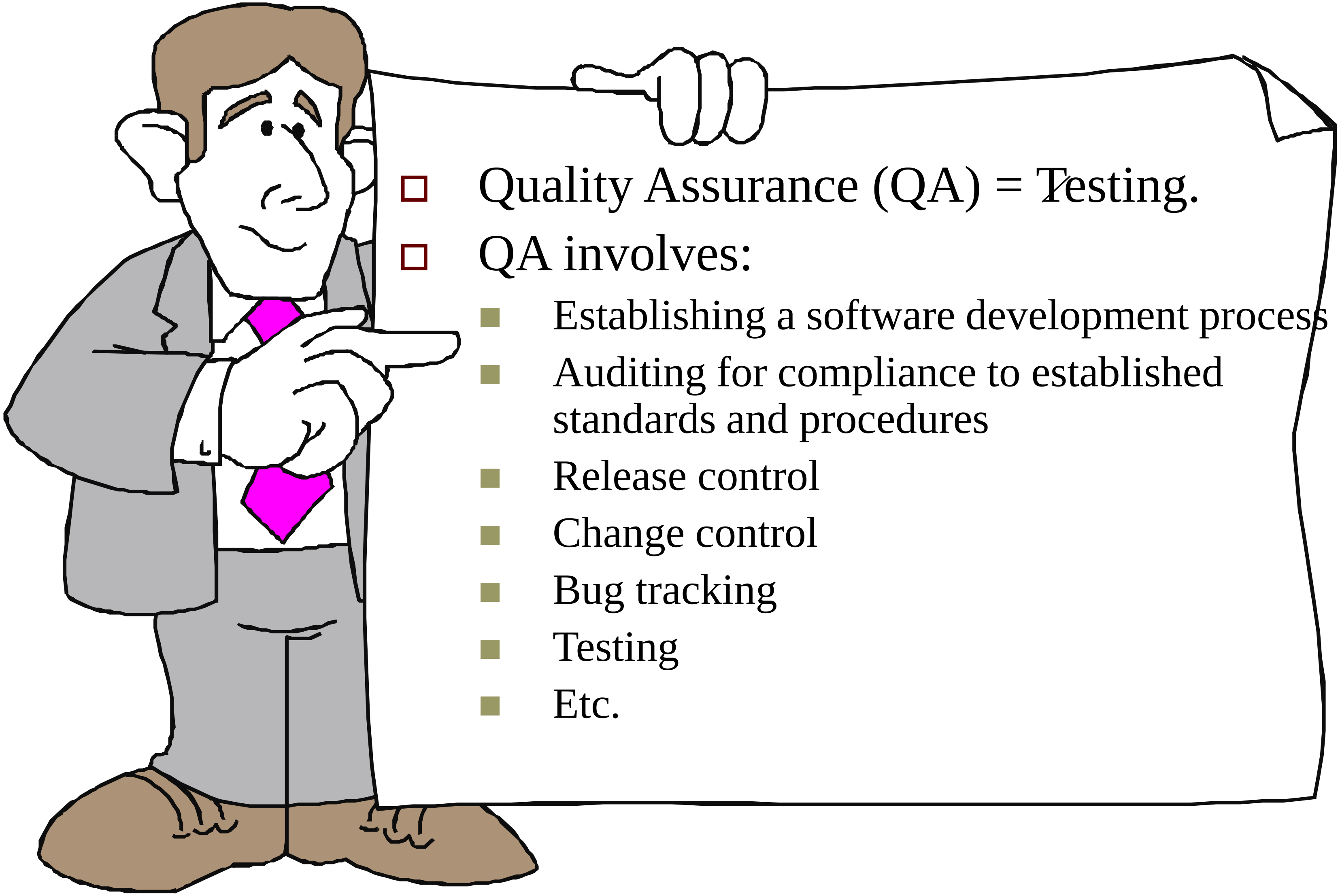
Integration Testing

System Testing

Code Developers

Independent Test Group

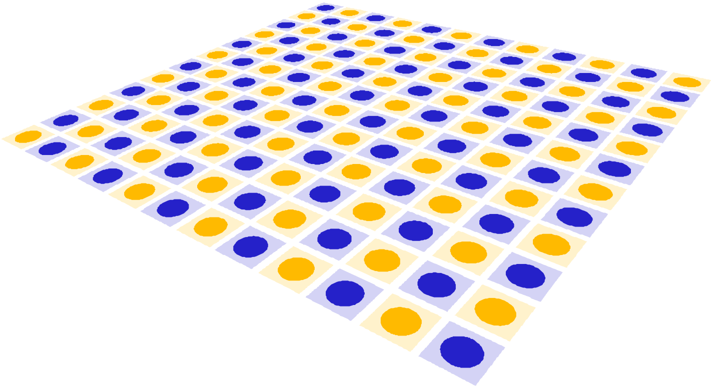
Role Confusion – Testing & QA



- □ Quality Assurance (QA) = Testing.
- □ QA involves:
  - Establishing a software development process
  - Auditing for compliance to established standards and procedures
  - Release control
  - Change control
  - Bug tracking
  - Testing
  - Etc.

Unit Testing

- What is a "unit"?
  - The smallest piece of software that can be complied, linked, and loaded.
  - Can be put under the control of a *driver* or *test harness*.
  - Usually the work of one software engineer.
  - Usually consists of a small number of lines of code (Several hundred of fewer).



Unit Testing - Characteristics

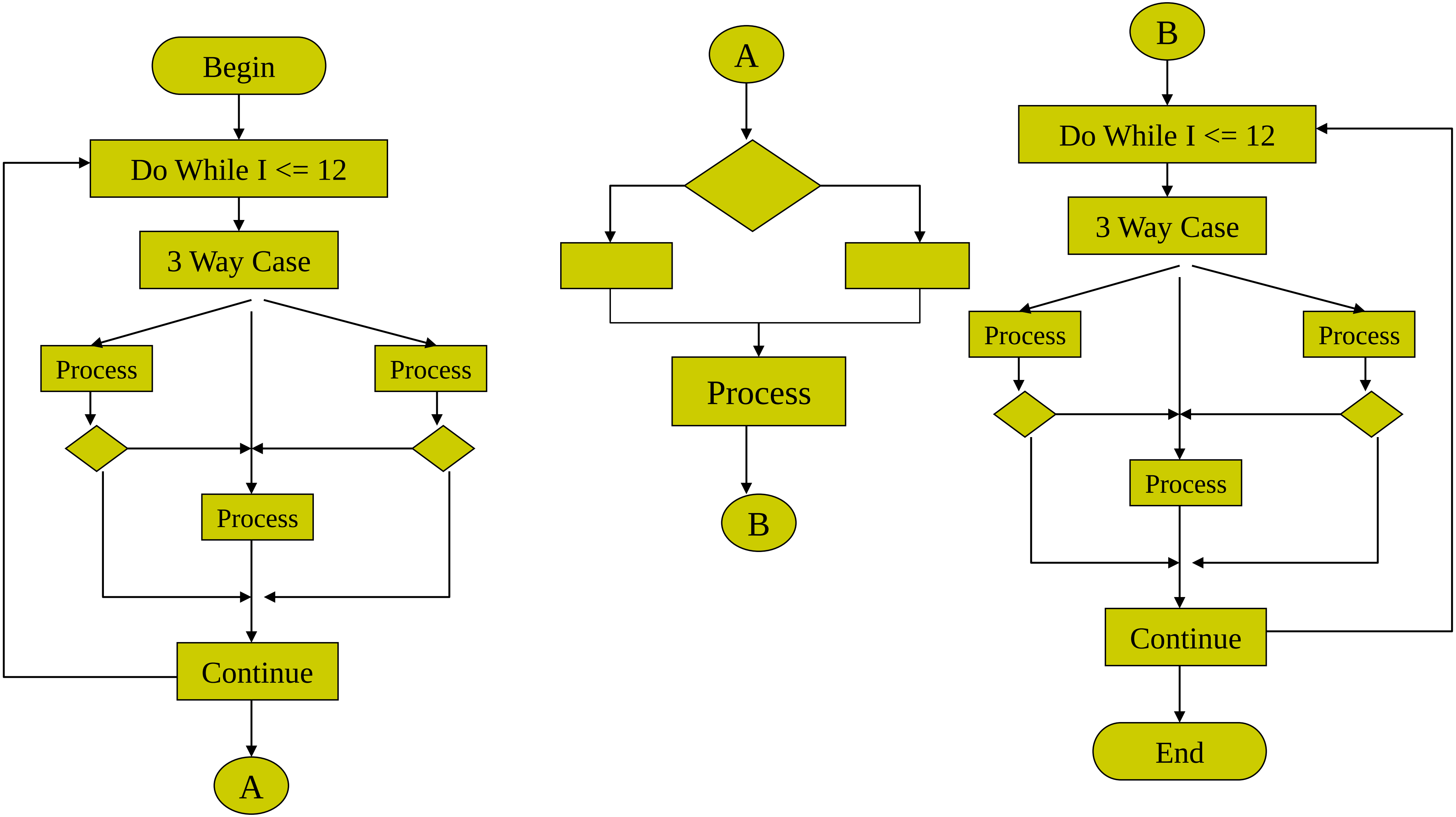
- Done by the developers.
- White box testing.
- Test cases are defined by specifying paths.
- Focus on a relatively small segment of code.
- A path is an instruction sequence that threads through the entire program form initial entry to final exit.
- Simplest approach is to ensure that every statement is exercised once.
- More stringent: Require coverage of every path. Usually not practical.

Source: Managing the Software Process, Watts Humphrey

Unit Testing - Paths

- Loops are problematic in testing.
- Each traverse of a loop is a path.
- For even small programs, there are a very large number of paths.
- Not practical to try to cover them all.
- Even if you could, it still would not ensure that all problems were detected.

An Example - Issues With Loops

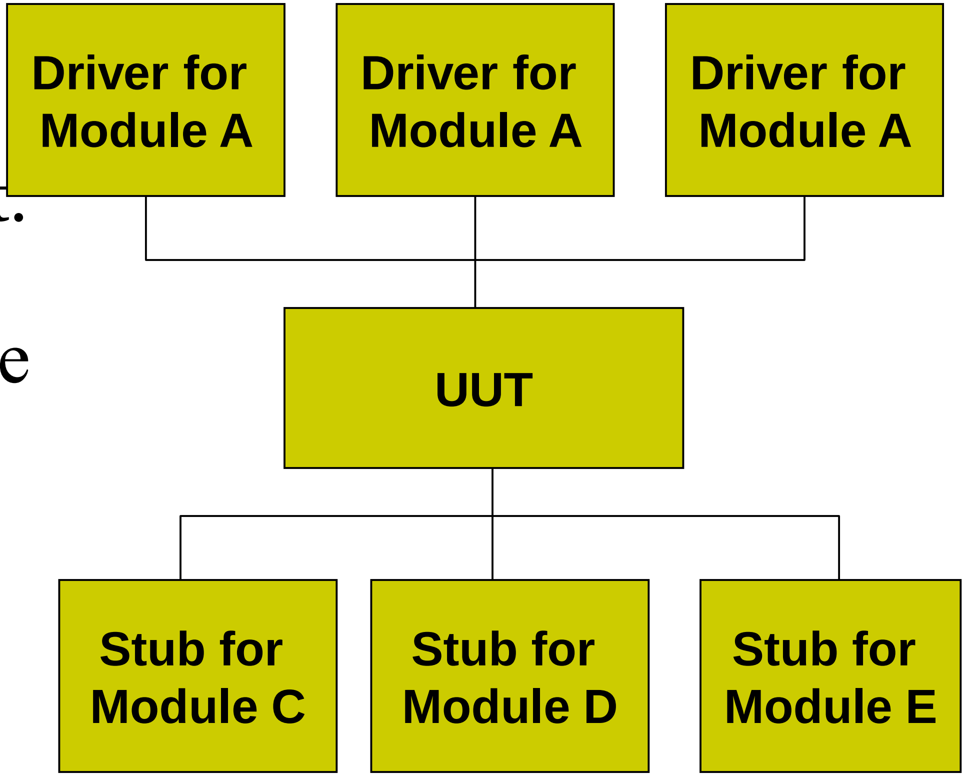


Example - continued

- How many paths are there through this program?
- Answer: 10^{23}
- If you checked 10 million paths per second, it would take approximately 32 million years to check all paths
- Add to that all paths for all possible data inputs, and error conditions --- ?

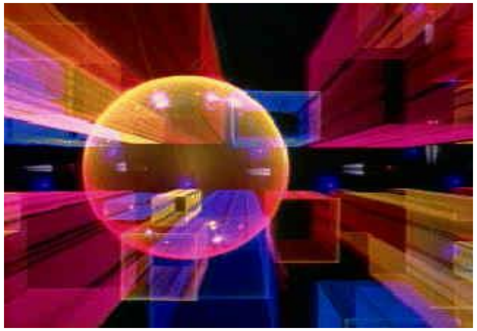
Unit Testing - Stubs and Drivers

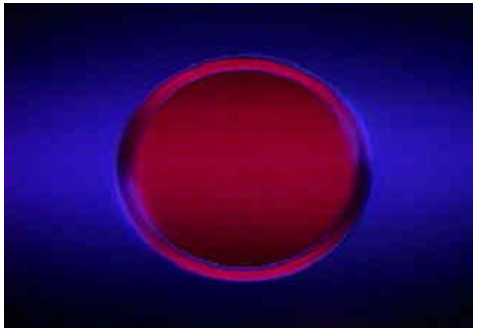
- Unit testing is done in an isolated, or “stand-alone” environment.
- Other modules are not ready yet.
- Must write some testware, or “scaffolding”, in order to be able to execute the unit under test (UUT).
- “Drivers” for higher level modules, and “stubs” for lower level modules.



Unit Testing –Stubs and Driver

- **Stubs & drivers are very simplified versions of the real modules.**
- **Drivers**
  - **Issues calls to the UUT with static parameters**
  - **Receives data from the UT, but does nothing with it.**
- **Stubs**
  - **Receives calls and data from the UUT.**
  - **On request, provides "canned" data to the UUT.**
  - **No further actions.**
- **Can also stub out database interfaces.**





Unit Testing Criteria

- Exercise each condition for each decision statement at least once.
- Ensure that all variables and parameters are exercised:
  - At & below minimums
  - At and above maximums
  - At intermediate values

Source: Managing the Software Process, Watts Humphrey

White Box Testing Techniques

- Control Flow Testing
  - Statement coverage
  - Branch coverage
  - Decision coverage
  - Basis path testing
  - Condition testing
  - Loop testing
- Data Flow Testing

Source: Software Engineering, Roger Pressman

Degrees of Module Coverage

- Statement Coverage – Execute every statement at least once.
- Decision (Branch) Coverage – Exercise each decision node such that each outcome is executed at least once.
- Multi-Condition Coverage – All combinations of conditions are tested.

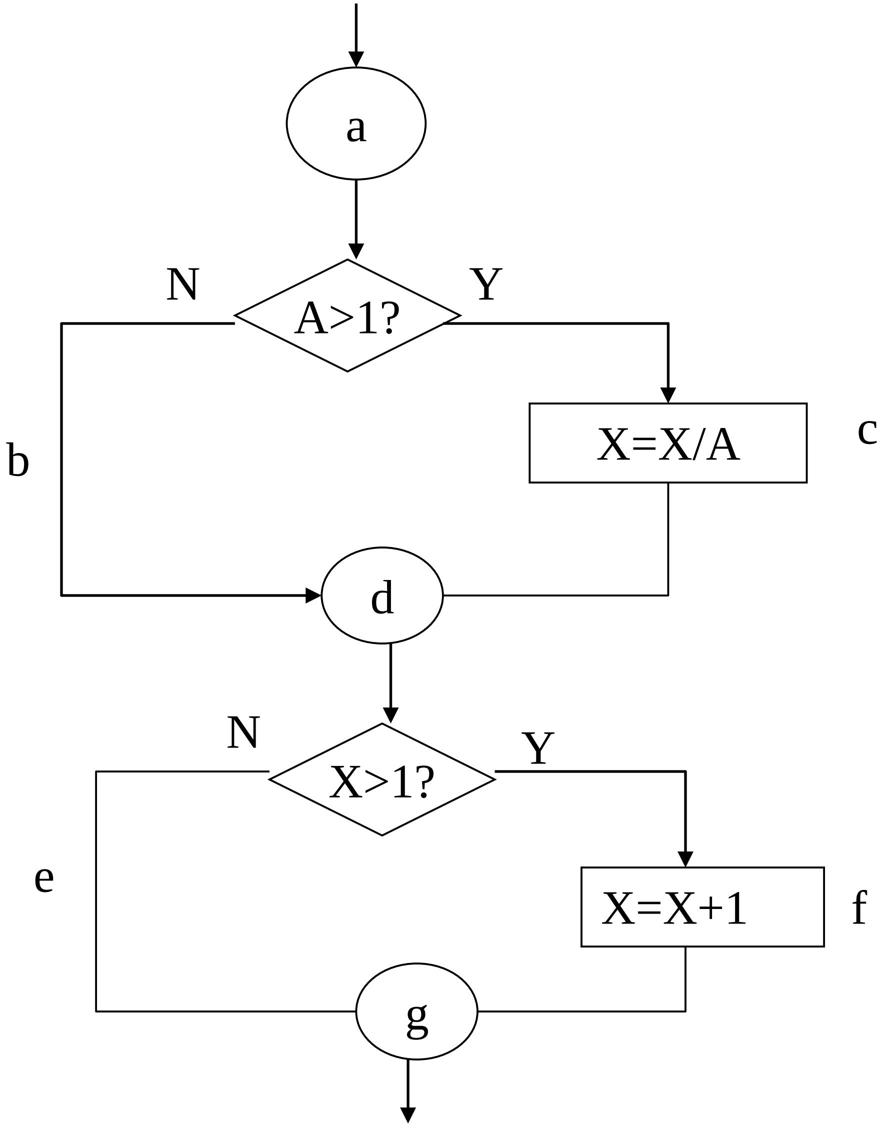
An Example Program

```
Begin

Read X, A
If (A>1) then
    X = X/A
Endif

If (X>1) then
X = X+1
Endif
Print X

End
```



An Example – Statement Coverage

Test Case

Input:

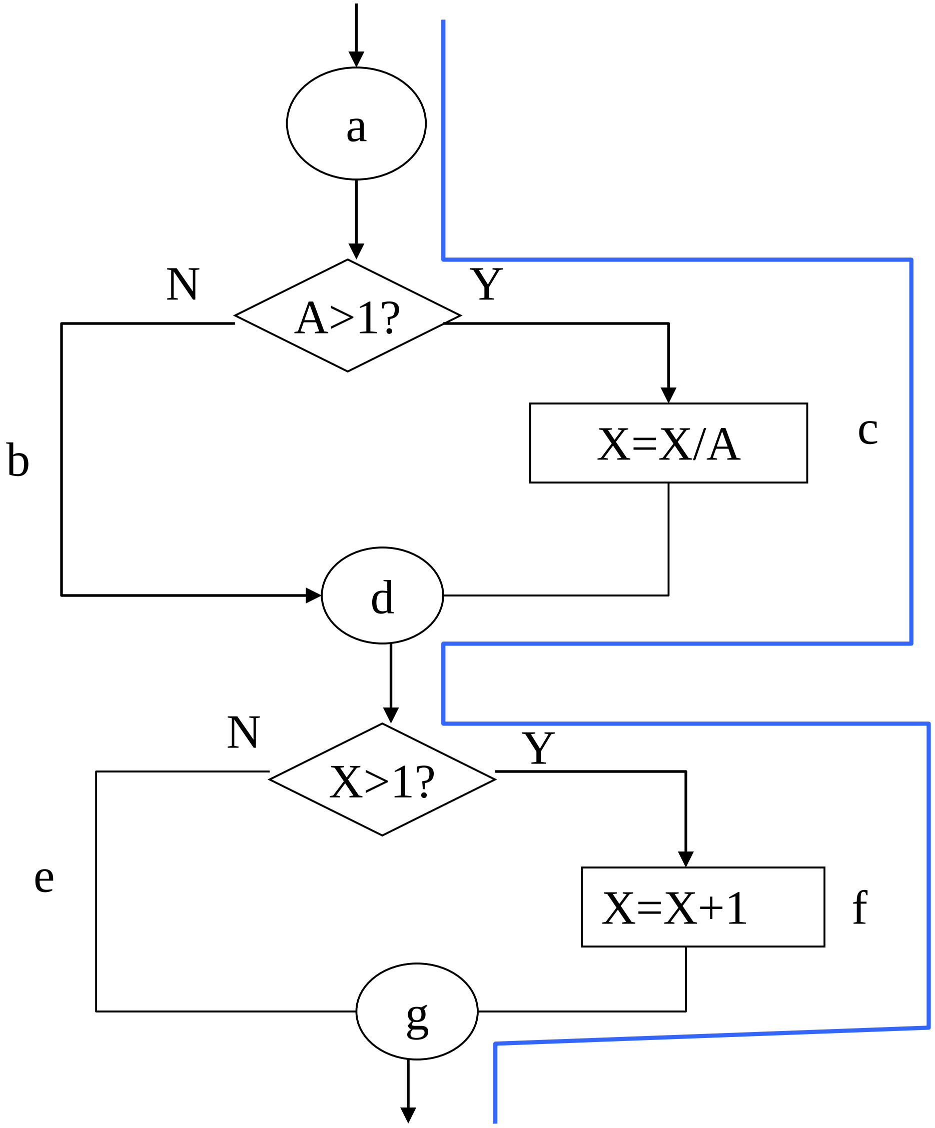
A = 2

X = 4

Path = acdfg

Output:

X = 4



An Example – Branch Coverage

Test Case 1

Input: A = 3; X = 3

Path = acdeg

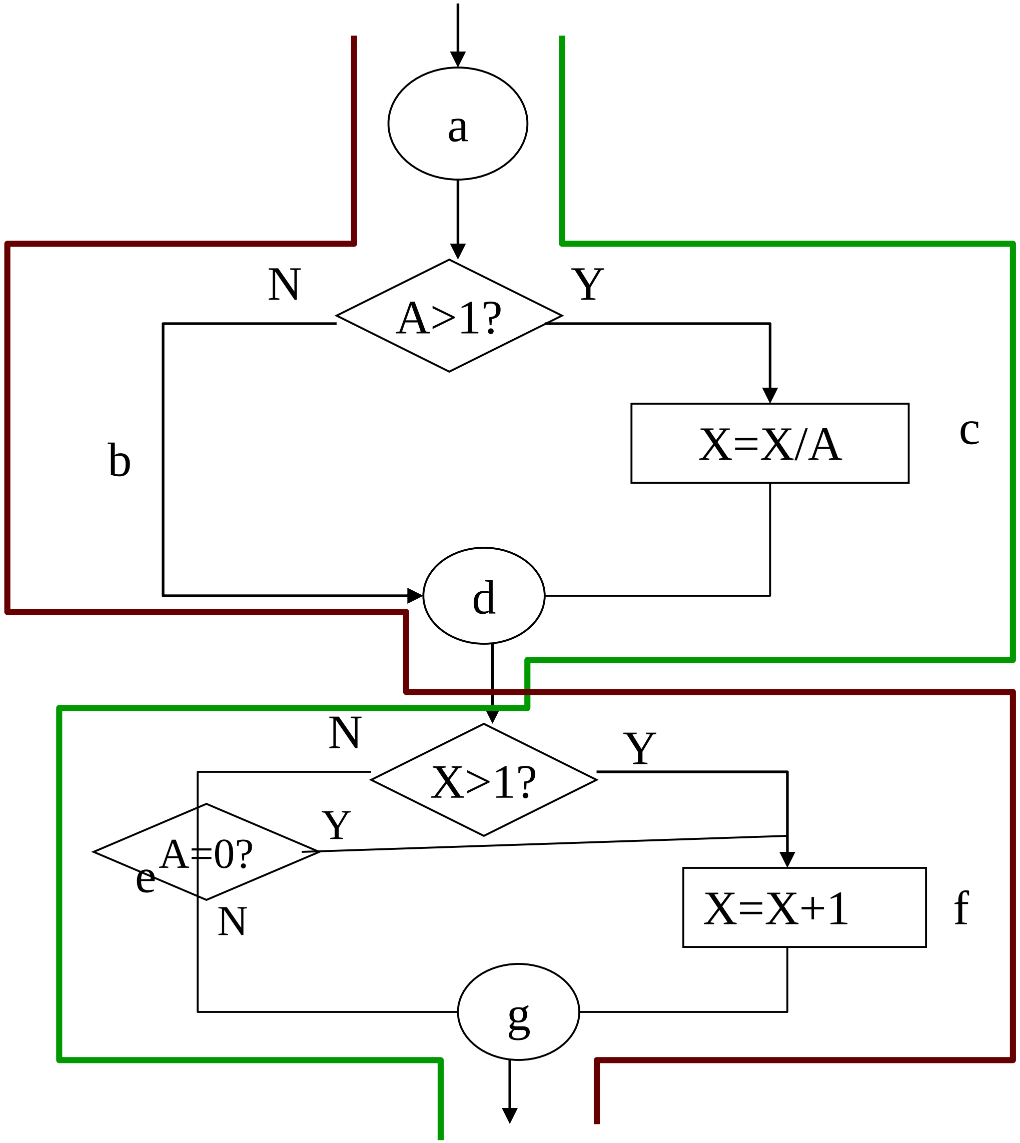
Output: X = 1

Test Case 2

Input: A = 0; X = 2

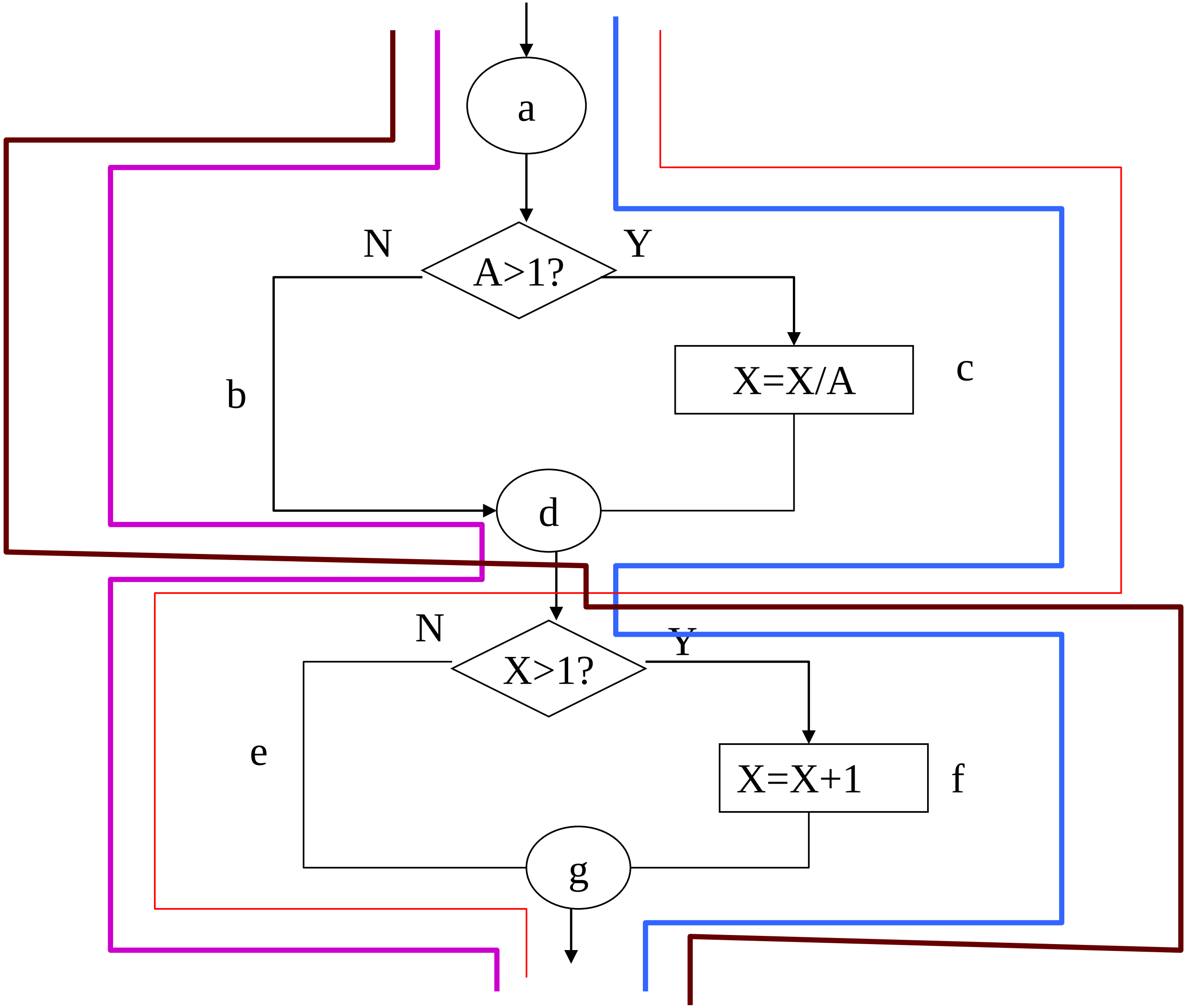
Path = abdfg

Output: X = 3



Example – Multi-condition or Path Coverage

Need 4 test cases.



Basis Path Testing

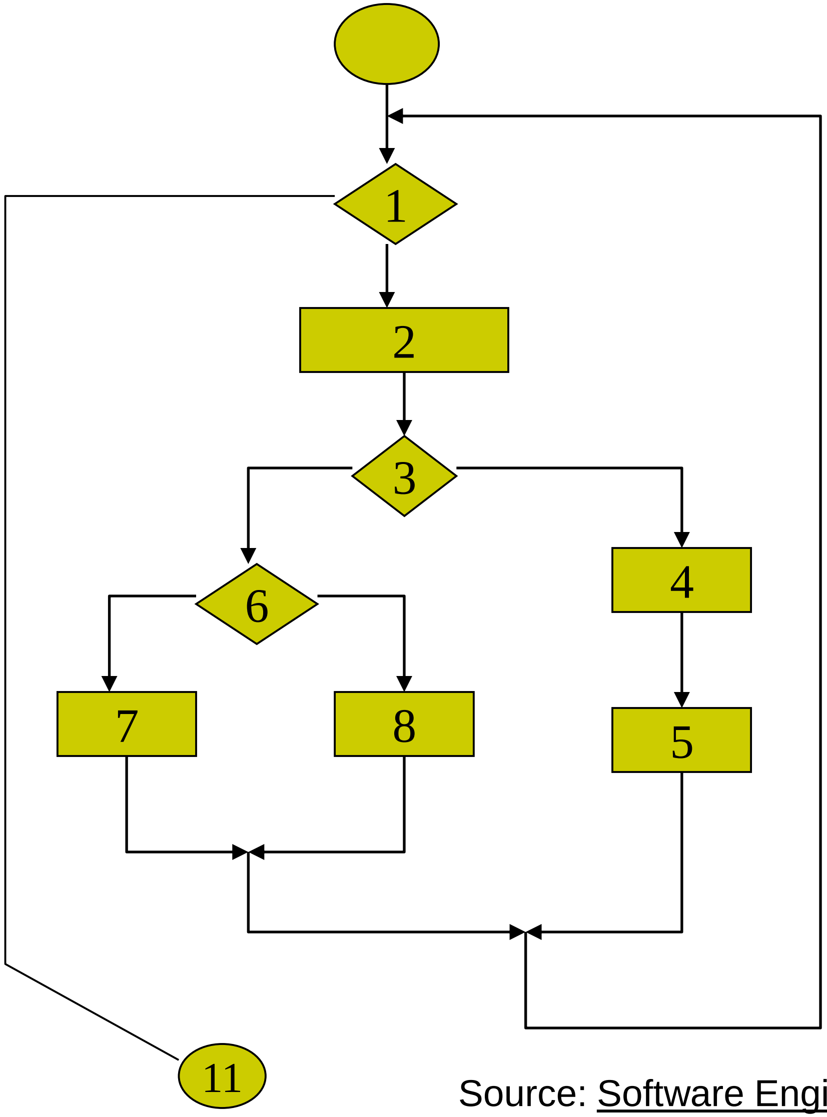
- A white box technique first proposed by Tom McCabe.
- Based upon the concept of program complexity (Cyclomatic complexity). Foundation in graph theory.
- Complexity is based upon the number of decision in a program (logical complexity).
- The premise is that highly complex programs are hard to test, unreliable, and hard to maintain.
- Can also use the complexity analysis as a guide for defining a *“basis set”* of execution paths through the program.
- A *basis set* of paths is a set from which all other paths can be obtained by linear combination of the basis paths. This is the minimum number of paths that ensure that all statements are executed at least once and all decisions are exercised in each direction.
- Derived from a flow graph of the software logic..
- Test cases derived from the *basis path set* are the minimum number of test cases that ensure statement coverage.

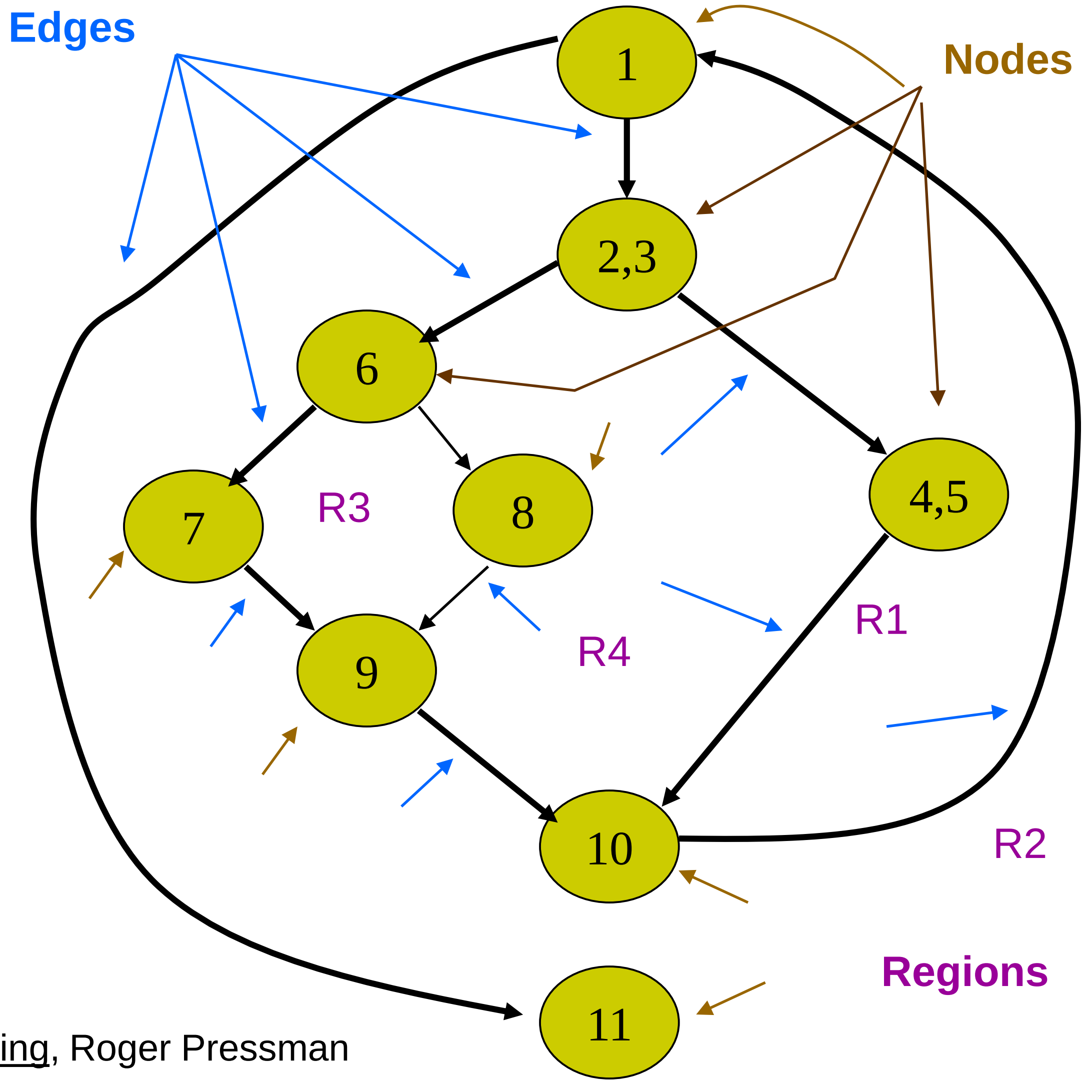
Example Program for Complexity Calculation

```
Procedure: Sort
1:   Do while records remain
         Read record;
2:       If record field 1 = 0
3:           then process record;
                 store in buffer;
                 increment counter;
4:           elseif record field 2 = 0
5:                then reset counter;
6:                else process record;
                      store in file;
7a:          endif
          endif
7b:  enddo
8. End
```

Source: Software Engineering, Roger Pressman

Flow Graph Notation - Example





Source: Software Engineering, Roger Pressman

Cyclomatic Complexity - Formulas

- $V(G) = E - N + 2$

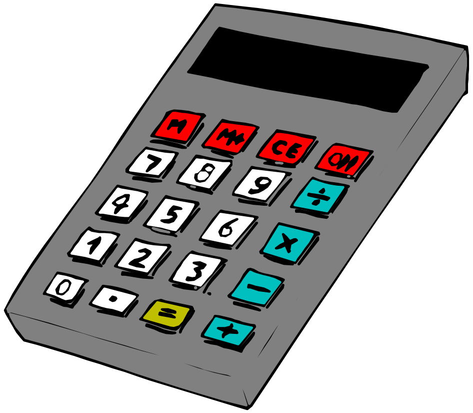
  where: E is the number of edges

  N is the number of nodes

- $V(G)$ = number of regions

- $V(G) = P + 1$

  where: P is the number of predicate nodes



Source: “A Software Complexity Metric”, Tom McCabe, IEEE Trans. Software Engineering, Dec., 1976

Cyclomatic Complexity – Example Calculations

- *GV(G)* = 11 edges – 9 nodes + 2 = 4
- *VV(G)* = number of regions = 4
- *VV(G)* = 3 predicate nodes + 1 = 4

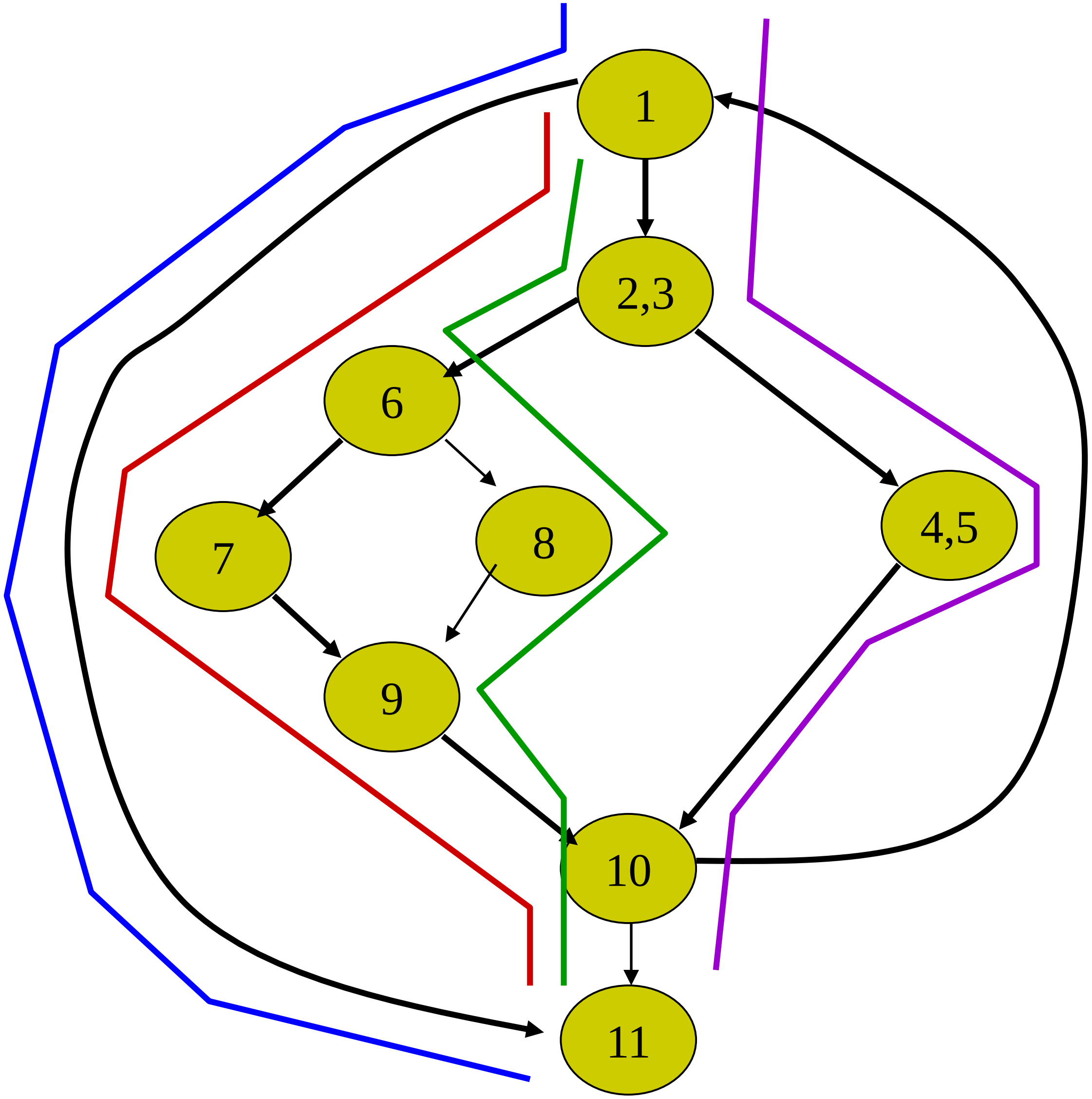
Deriving Test Cases From Basis Paths

- Calculate the complexity.
- Determine a set of “independent” paths through the flow graph.
- An “independent” path is one that introduces an edge not covered in another path in the set of independent paths.
- The number of independent paths is equal to the complexity.
- Each independent path becomes a test case.
- Specify input values that will cause each path to be executed. These are the test cases.

Example – Basis Path Definition

- IIn the previous example, the complexity was four, so we need a set of four basis paths.
    - I
- PPath 1: 1 – 11
- PPath 2: 1-2-3-6-7-9-10-1-11
- PPath 3: 1-2-3-6-8-9-10-1- 11
- PPath 4: 1-2-3-4-5-10-1-11

Example - Paths Shown



Example – Test Cases From Basis Paths

- Test Case 1: No records to be processed.
- Test Case 2: One record to be processed

  Record field 1 = 0
- Test Case 3: One record to be processed

  Record field 1 = 0⁄

  Record field 2 = 0
- Test Case 4: One record to be processed

  Record field 1 = 0⁄

  Record field 2 = 0⁄

Example – Limitation of Basis Path Testing

What test cases that probably need to be run have not been defined in the previous example?

Limitations - Continued

- Multiple records.
- Volume test (max. number of records)
- Records with erroneous data (numeric characters in the record fields instead of alphabetic).

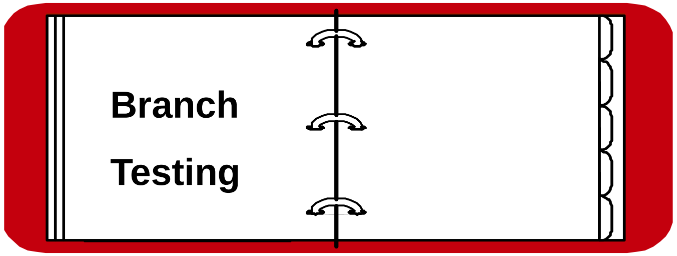
Condition Testing

- Focuses on testing each logical condition in the program
- A simple condition is a Boolean variable or a relational expression
  .
  - Boolean variable: Or, And, Not
  - Relation expression: *E_1 (relational-operator) E_2*

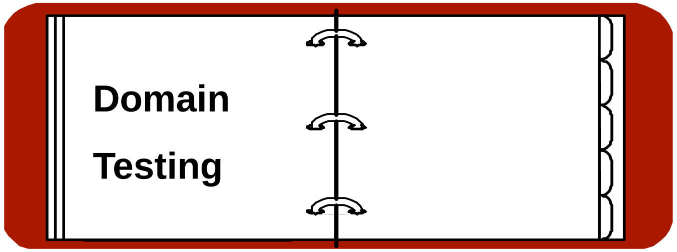
  Where E_1 and E_2 are arithmetic expressions and *(relational-operator)* is: <, <=, = =, >,>=/.
- A compound condition is composed two or more simple conditions.
- Premise: If tests are effective in finding errors in the program conditions, they are probably effective for finding other errors.

Condition Testing Strategies

- **Branch Testing:** The *true* and *false* branches of every condition are exercised.



- **Domain Testing**
- *E_1 (relational operator) E_2*
- Three tests are required to make E_1 greater than, equal to, or less than E_2.



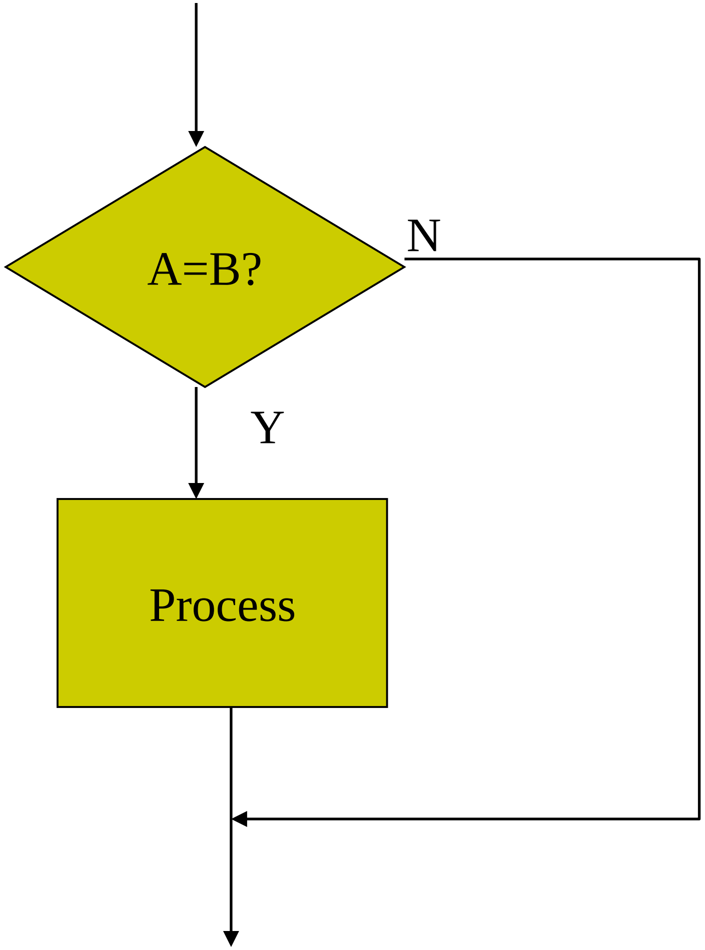
Example - Domain Testing

Test Cases

TS1: A=B

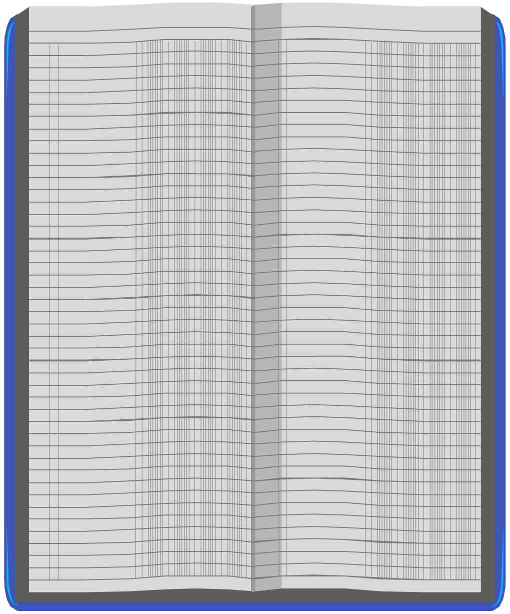
TS2: A>B

TS3: A<B



Data Flow Testing

- At least half of contemporary source code consists of data declaration statements – that is, statements that define data structures , individual objects, initial or default values, and attributes.
- Data bugs are at least as common as bugs in code.
- Code migrates to data
  - Low cost of memory
  - High cost of software
  - Table-driven software



Source: Software Testing Techniques, Boris Beizer

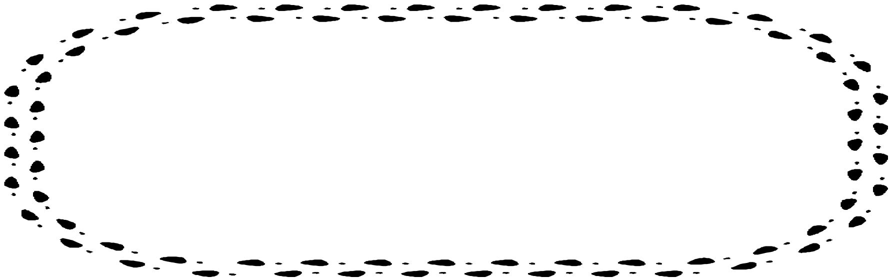
Data Flow Testing

- ***Data flow testing*** selects test paths of a program according to the definitions and uses of variables in the program.
- Definition Statement (X) → Contains a definition of X.
- Use Statement (X) → Contains a use of X .
- Basic Testing Strategy: Every *definition –use* path is covered.
- There are other, more complicated testing strategies.

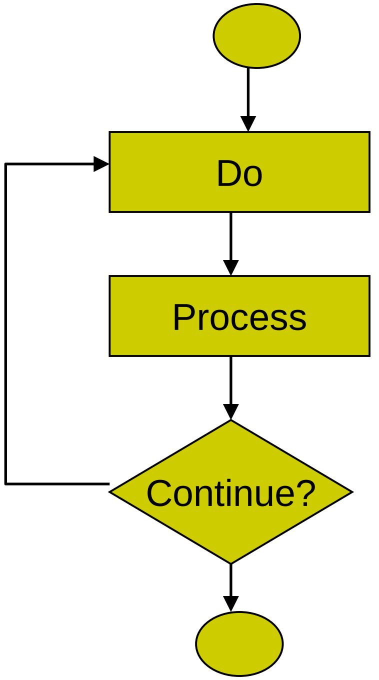
Source: Software Testing Techniques, Boris Beizer

Loop Testing

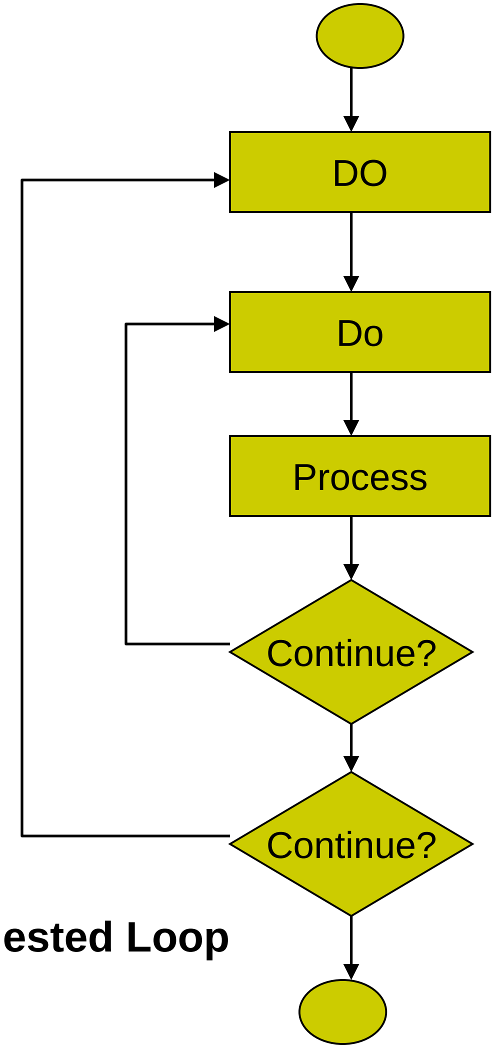
- Loops are the foundation of most algorithms.
- Loops tend to be buggy.
- Often neglected in testing.



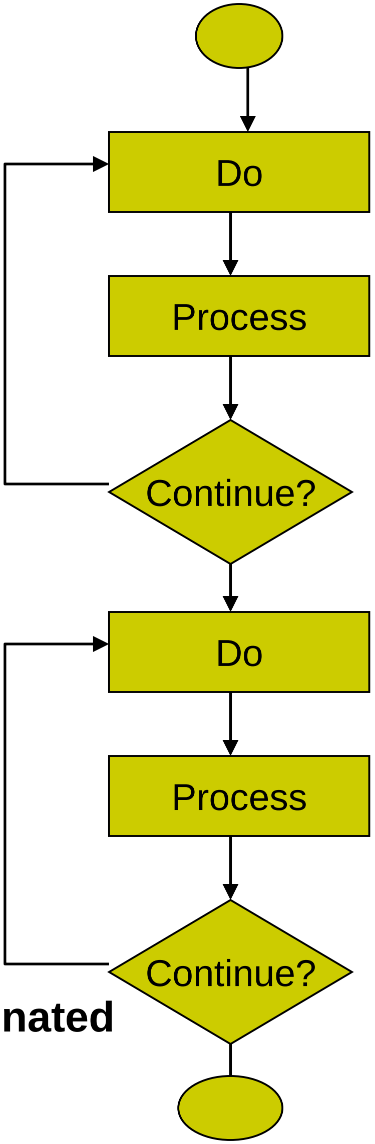
Types of Loops



Simple Loop



Nested Loop



Concatenated Loop

Unstructured Loops: Loops that jump into other loops. Spaghetti code.

Criteria for Testing of Simple Loops

- Zero times through the loop.
- Once through.
- Twice through.
- Typical number of time through.
- (Maximum – 1) number of times through.
- Maximum number of times through.
- (Maximum + 1) number of times through.

Sources: Black Box Testing, Boris Beizer; Software Engineering, Roger Pressman

Criteria for Testing Concatenated Loops

- Use the approach for simple loops on each loop independently.

Sources: Black Box Testing, Boris Beizer; Software Engineering, Roger Pressman

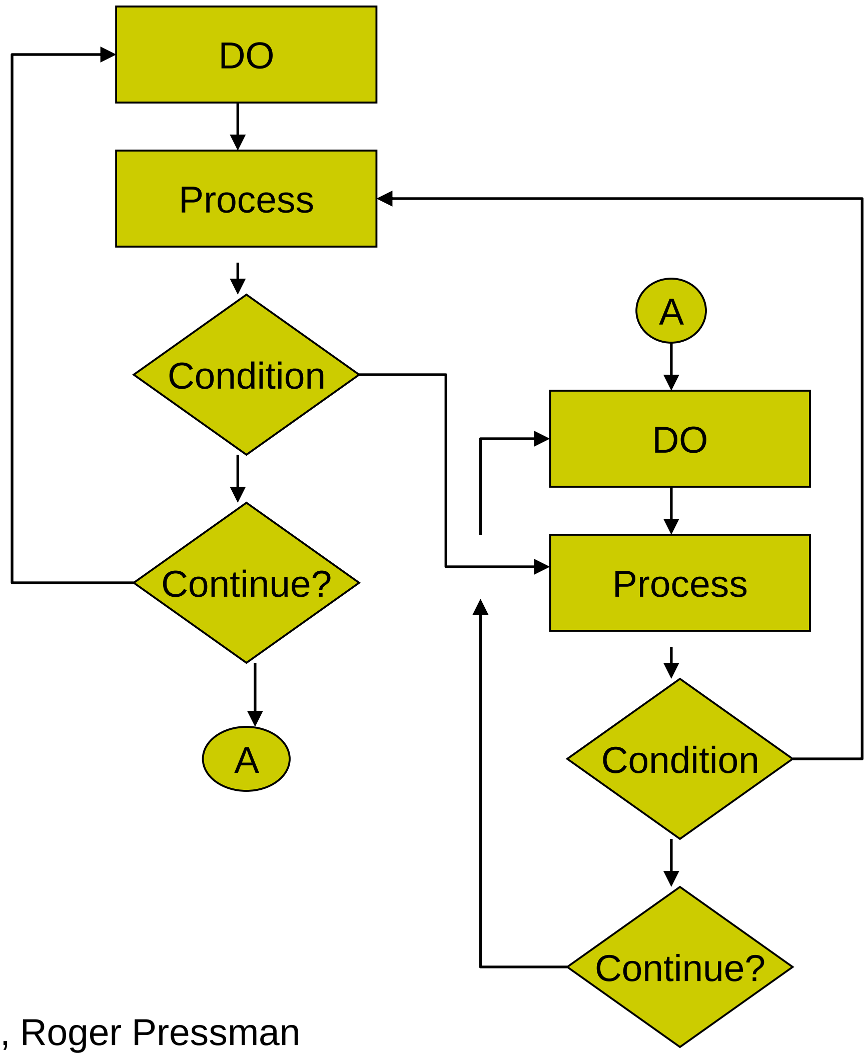
Criteria for Testing Nested Loops

- Set outer loops to typical values; conduct the critical cases for the innermost loop.
- Go to the next outer loop: Conduct tests of critical values, with inner loops and outer loops set to typical values.
- Continue working outward until all loops are tested.
- Test all of the combinations of bypass, one, two, max for all of the loops. For two nested loops , this is 16 additional tests. For three nested loops, it is 64 additional tests, etc.

Sources: Black Box Testing, Boris Beizer; Software Engineering, Roger Pressman

Criteria for Testing Unstructured Loops

- Very difficult to test.
- But also very buggy.
- The only effective approach is to recode them using structured constructs.



Sources: Black Box Testing, Boris Beizer; Software Engineering, Roger Pressman

Loop Testing - An Example

- **Payroll System that will handle up to 10,000 employees.**
- **Test Cases:**

  TC1 Input: Data for no employees
  Output: No action; continued correct operation.
  TC2 Input: Data for one employee.
  Output: Correct payroll processing for one employee.
  TC3: Input: Data for two employees.
  Output: Correct payroll processing for two employees.
  TC4 Input: Data for 500 employees.
  Output: Correct payroll processing for 500 employees.
  TC5 Input: Data for 9999 employees.
  Output: Correct payroll processing for 9999 employees.
  TC6 Input: Data for 10,000 employees.
  Output: Correct payroll processing for 10,000 employees.
  TC7 Input; Data for 10,001 employees.
  Output: Correct processing for 10,000 employees; correct operation next time.

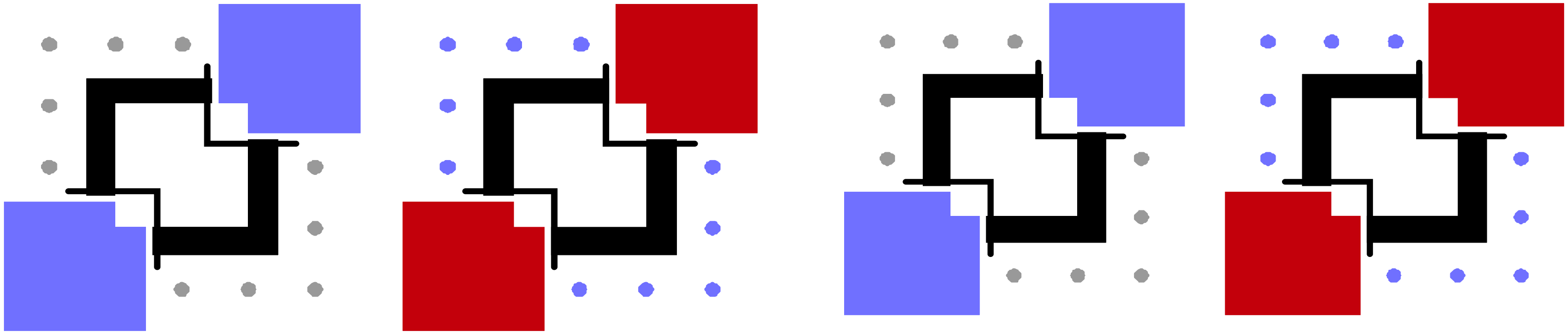
Unit Testing Guidelines & Checklists

See separate handouts.

These are additional suggestions for unit testing. Some we have discussed; some are in addition to the material presents in this class.

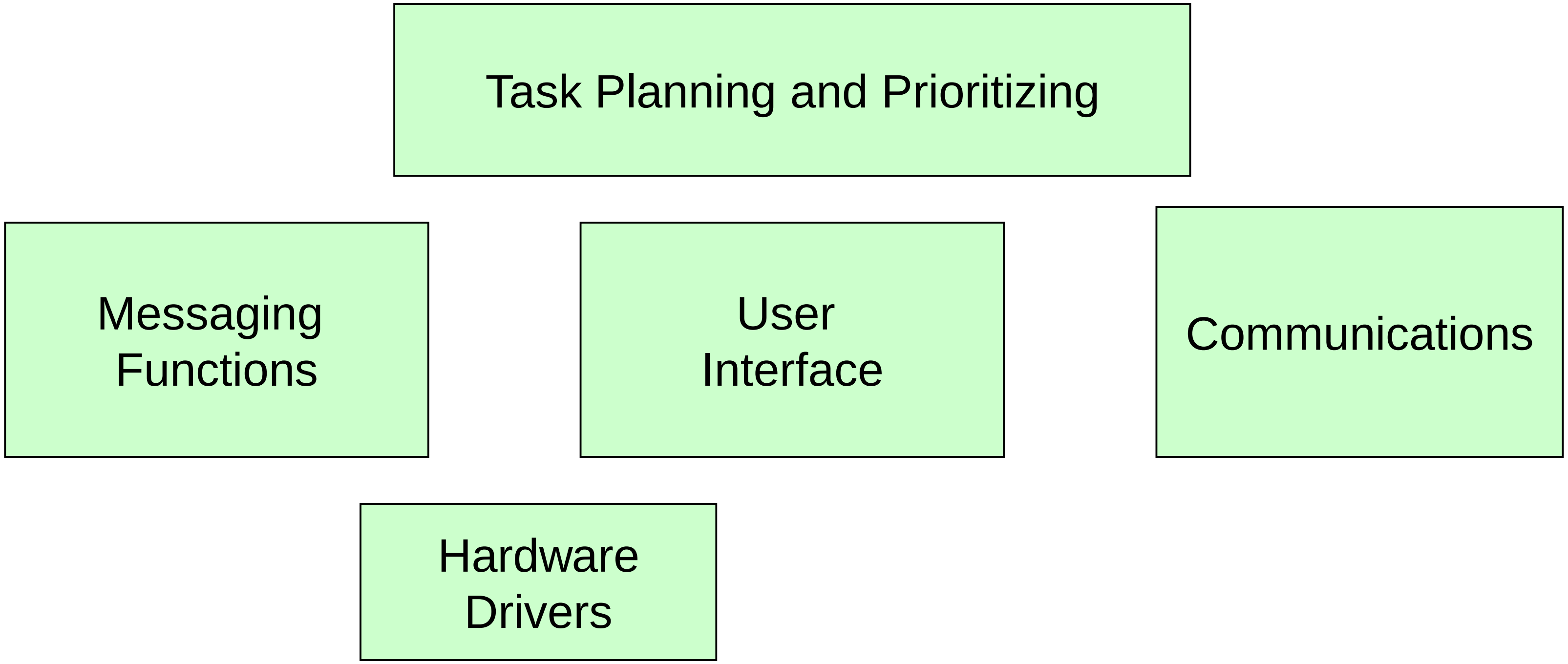
Software Integration

- *Software Integration* is the combining of previously tested units into aggregates until the full system is there.
- Integration is a “building block” process.
- *Integration Testing* is the testing of interfaces between previously tested units.



Subsystems

- Software integration is frequently done on a subsystem by subsystem basis.
- The modules in each subsystem are "integrated" to form that subsystem. Testing focuses on the interfaces between modules in the one subsystem.
- The subsystems would then be integrated with each other. Testing focuses on the interfaces between subsystems.



Approaches to Integration

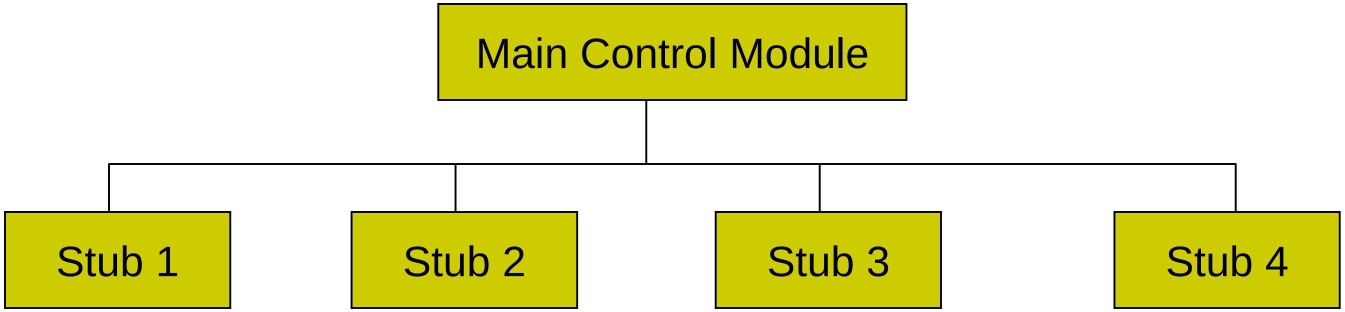
- Top-down Integration
- Bottom-up Integration
- “Big Bang” Integration

Top-down Integration - Approach

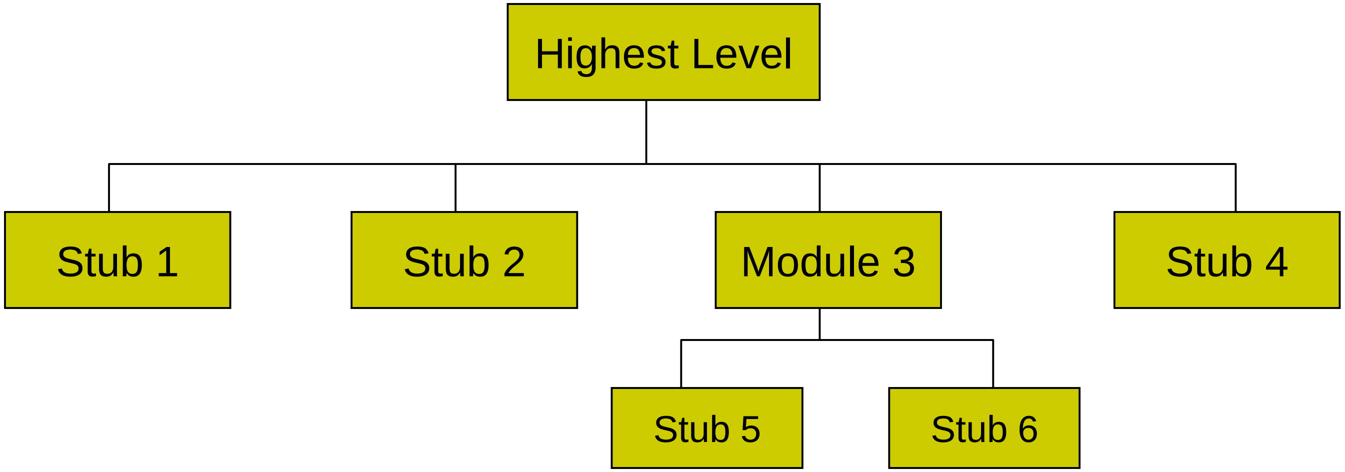
- Start building the system with the highest level modules in the control hierarchy.
- Use “stubs” to represent lower level modules.
- The integration process consists of replacing the stubs with the actual modules.
- When all stubs are replaced, the system in “integrated”.

Top-down Integration - Steps

Step 1

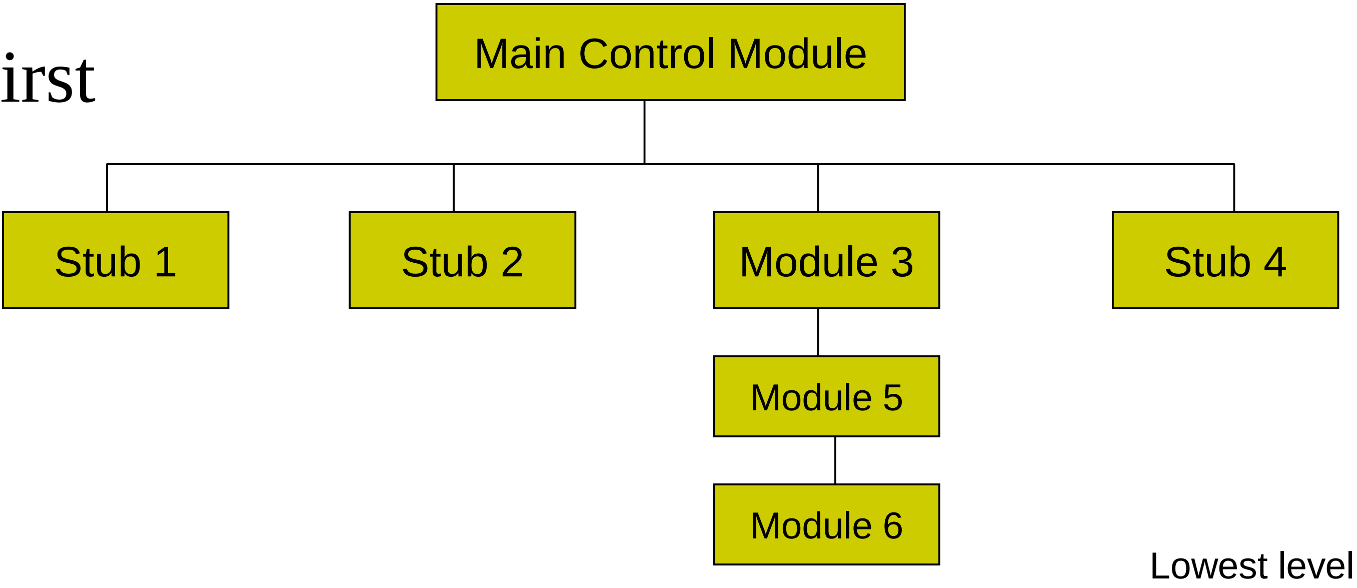


Step 2



Top-down Integration – Two Methods

- Depth first



Main Control Module

Stub 1

Stub 2

Module 3

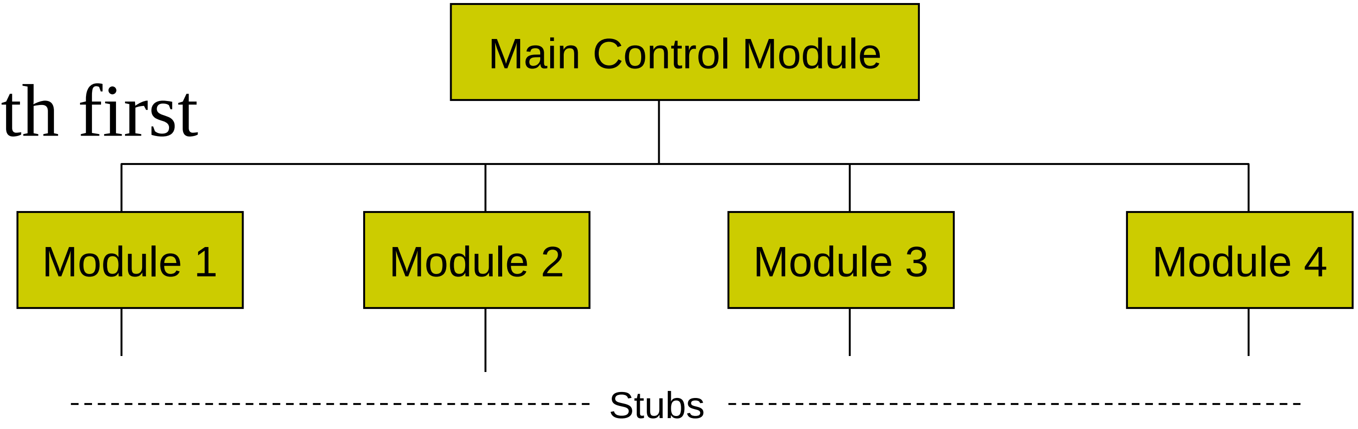
Stub 4

Module 5

Module 6

Lowest level

- Breadth first



Main Control Module

Module 1

Module 2

Module 3

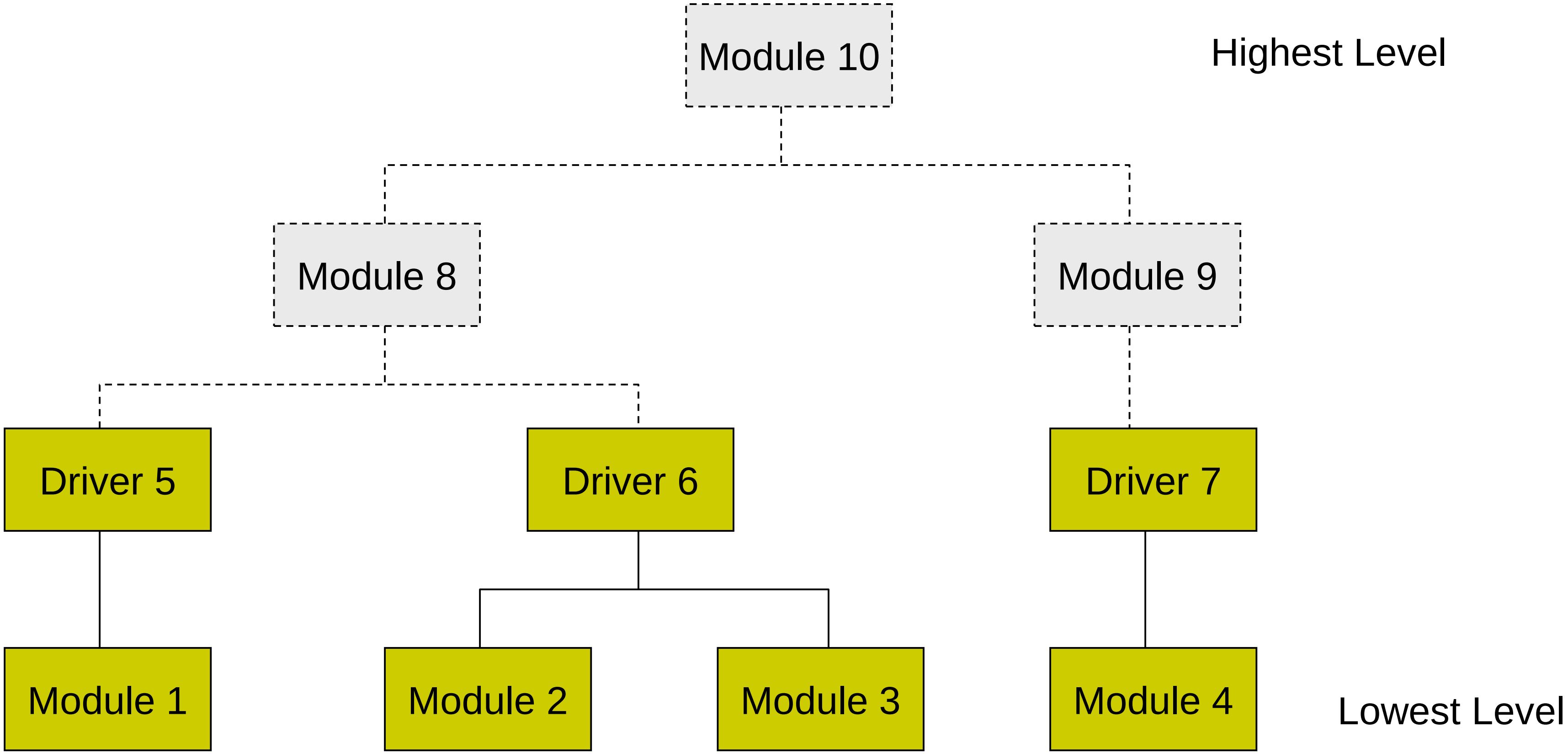
Module 4

------------------------------------ Stubs --

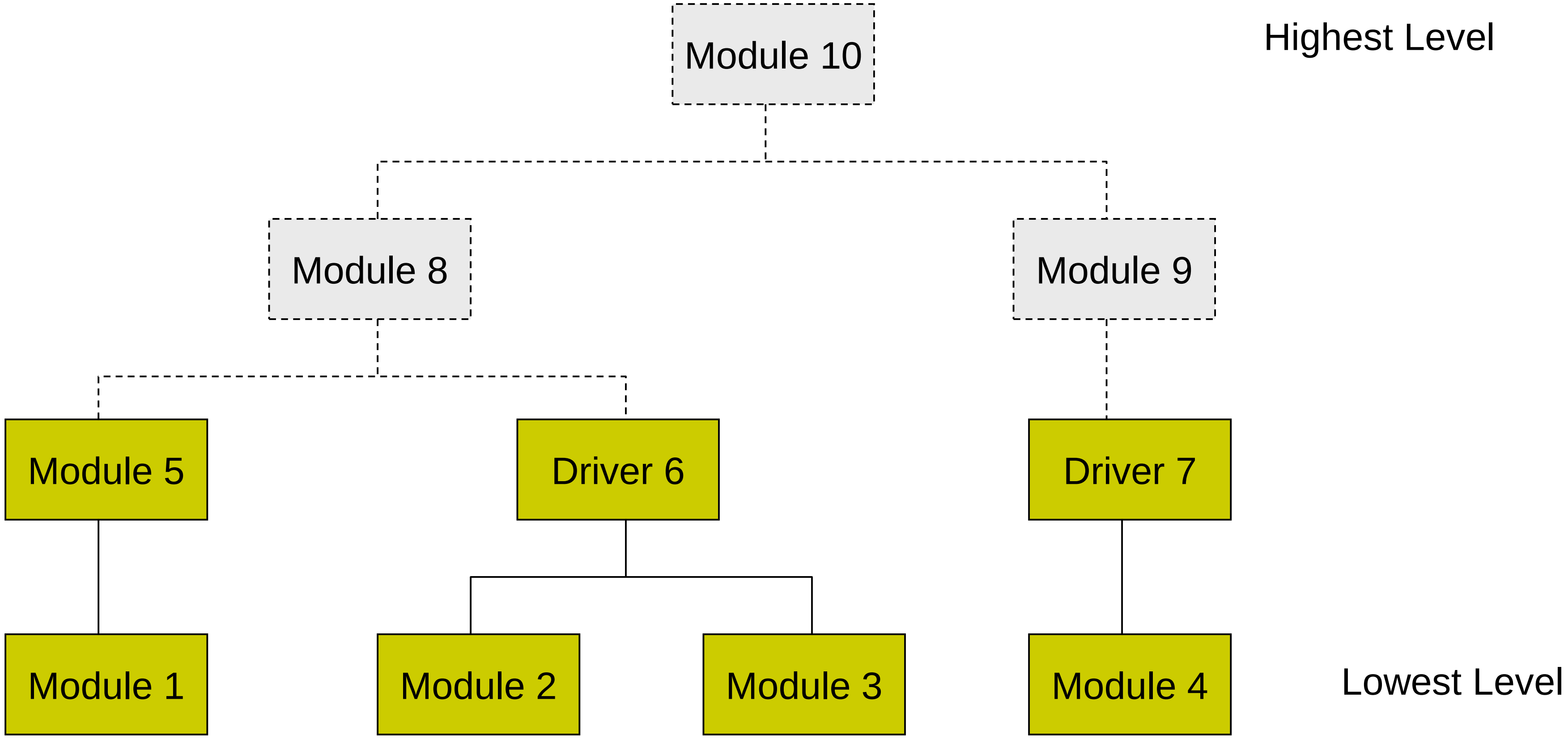
Bottom-up Integration - Approach

- Start building the system with the lowest level modules.
- Use “drivers” to represent higher level modules.
- The integration process consists of replacing the drivers with the actual modules.
- When all drivers are replaced, the system in “integrated”.

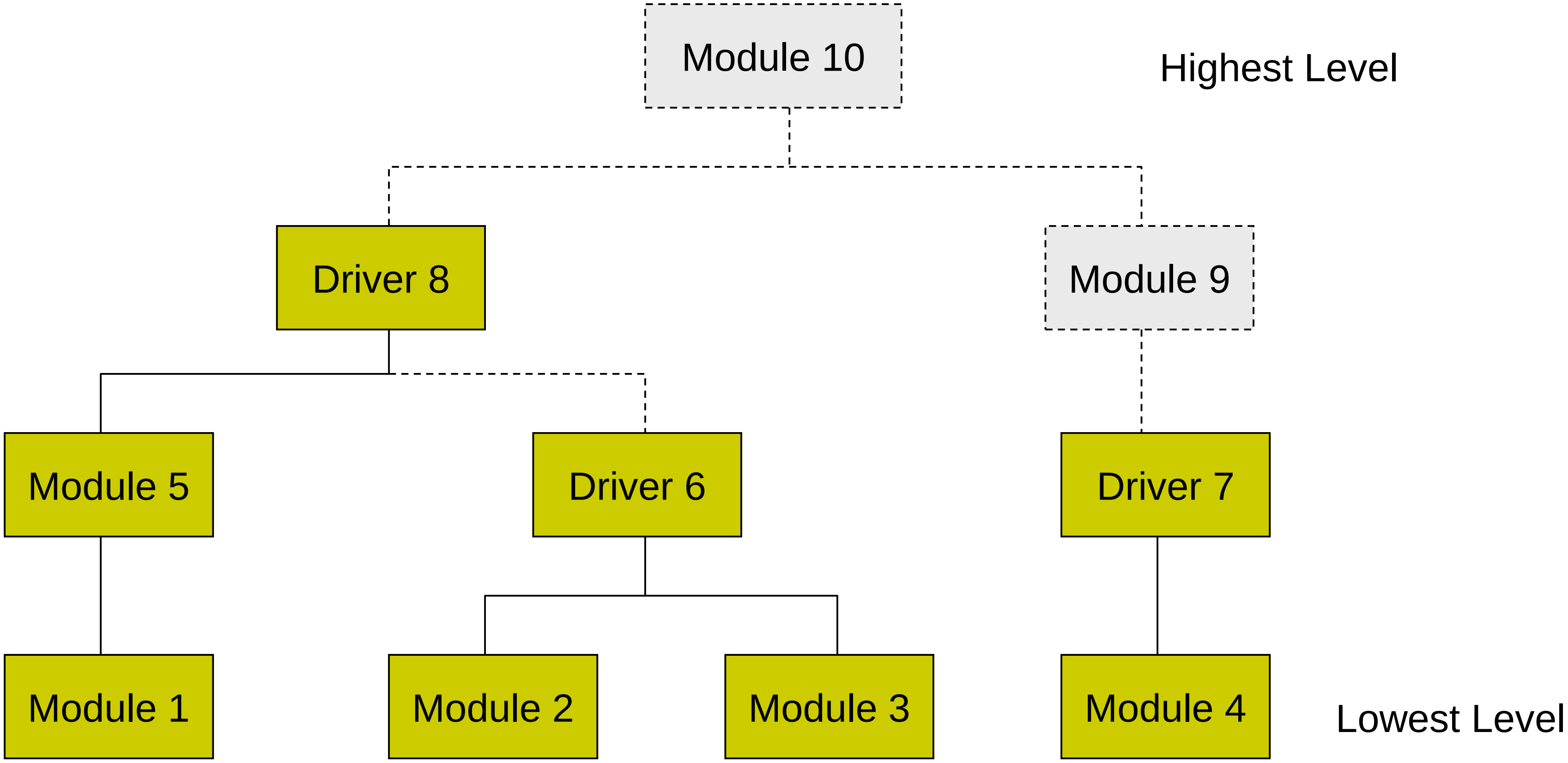
Bottom-up Integration - Steps



Bottom-up Integration - Steps

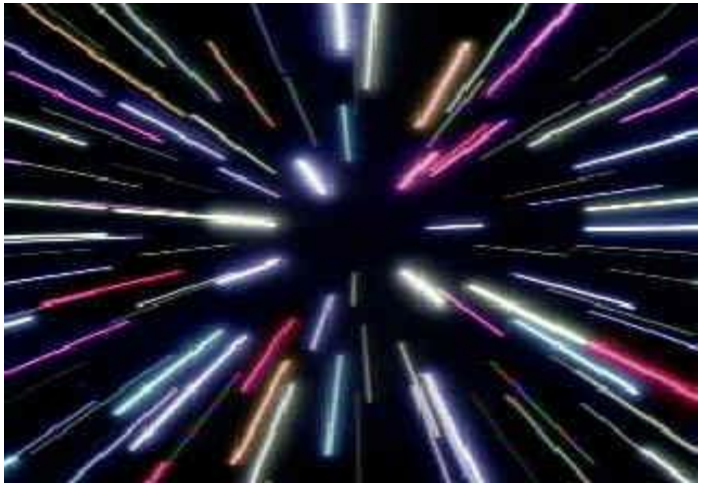


Bottom-up Integration - Steps



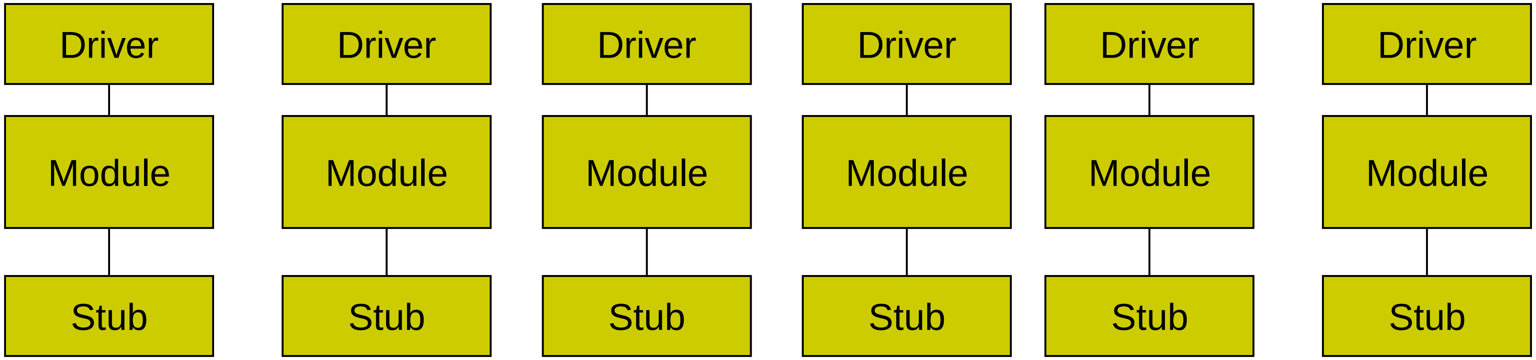
Big Bang Integration - Approach

- All modules are combined in one step.
- Most common integration approach.
- Usually is the least effective approach.

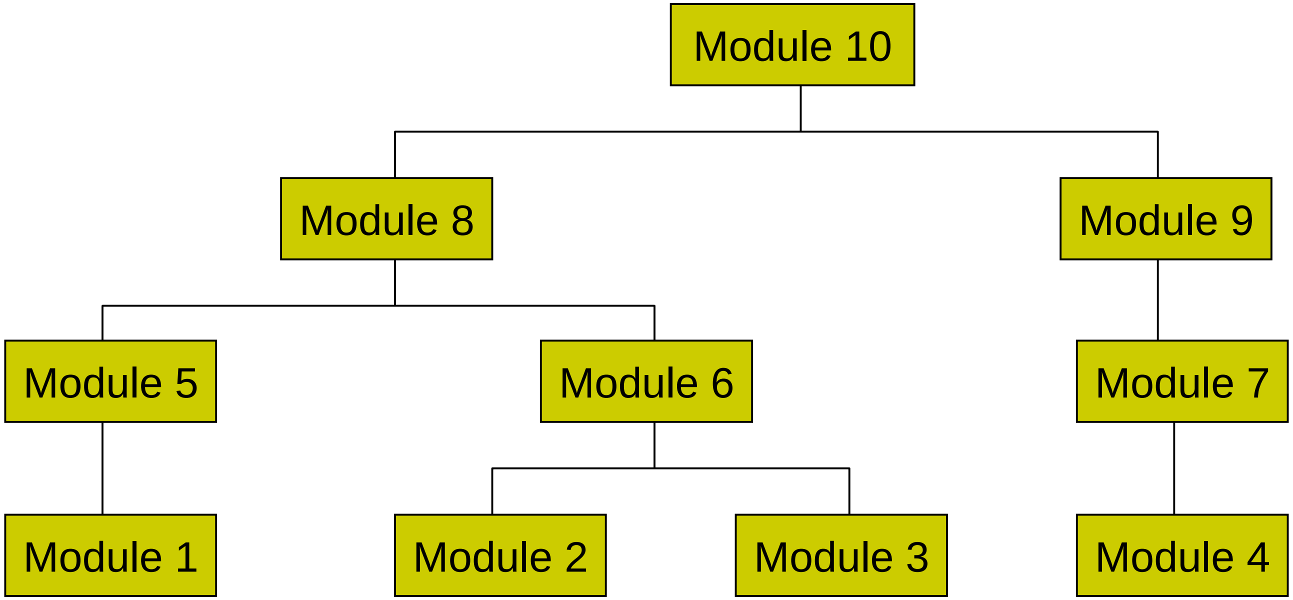


Big Bang Integration - Steps

Unit Testing


Driver
Module
Stub
Driver
Module
Stub
Driver
Module
Stub
Driver
Module
Stub
Driver
Module
Stub
Driver
Module
Stub

Step 1


Module 10
Module 8
Module 9
Module 5
Module 6
Module 7
Module 1
Module 2
Module 3
Module 4

Problems With “Big Bang” Integration

- It can be very difficult to locate the source of problems.
- Don’t know where to look.
- No “divide and conquer” effect.
- Not recommended except for very small systems.

Comparison of Integration Approaches

Bottom-up

Major Features: Allows early testing aimed at proving feasibility and practicality of particular modules.

Modules can be integrated in various clusters, as desired.

Gives more emphasis to module functionality and performance.

Advantages: No test stubs are needed.

It is easier to adjust manpower needs.

Errors in critical modules are found early.

Disadvantages: Test drivers are needed.

Many modules must be integrated before a working program is available.

Interface errors are discovered late.

Comments: At any given point, more code has been written and tested than with top-down integration.

Source: Software Reliability, Principles and Practices, Glenford Myers

Comparison of Integration Approaches

Top-down

Major Features: The control module is tested first.

Module are integrated one at a time.

More emphasis is placed on interface testing.

Advantages: No test drivers are needed.

The control module plus a few other modules constitute a basic early working version

Interface errors are discovered early.

Modular features aid debugging.

Disadvantages: Test stubs are needed.

The extended early phases dictate a slow manpower build-up.

Error in critical modules at low levels are found late.

Comments: An early working system raises morale and helps to demonstrate that progress is being made.

It is hard to maintain a pure top-down approach in practice.

Source: Software Reliability, Principles and Practices, Glenford Myers

Comparison of Integration Approaches

Big Bang

Major Features: All modules are combined at once.

Advantages: No test stubs or drivers are needed.

Gives the *appearance* of making much progress.

Disadvantages: Module interfaces are not tested except in a system environment.

Problems can be very difficult to “troubleshoot”.

Can be very frustrating to the software engineers.

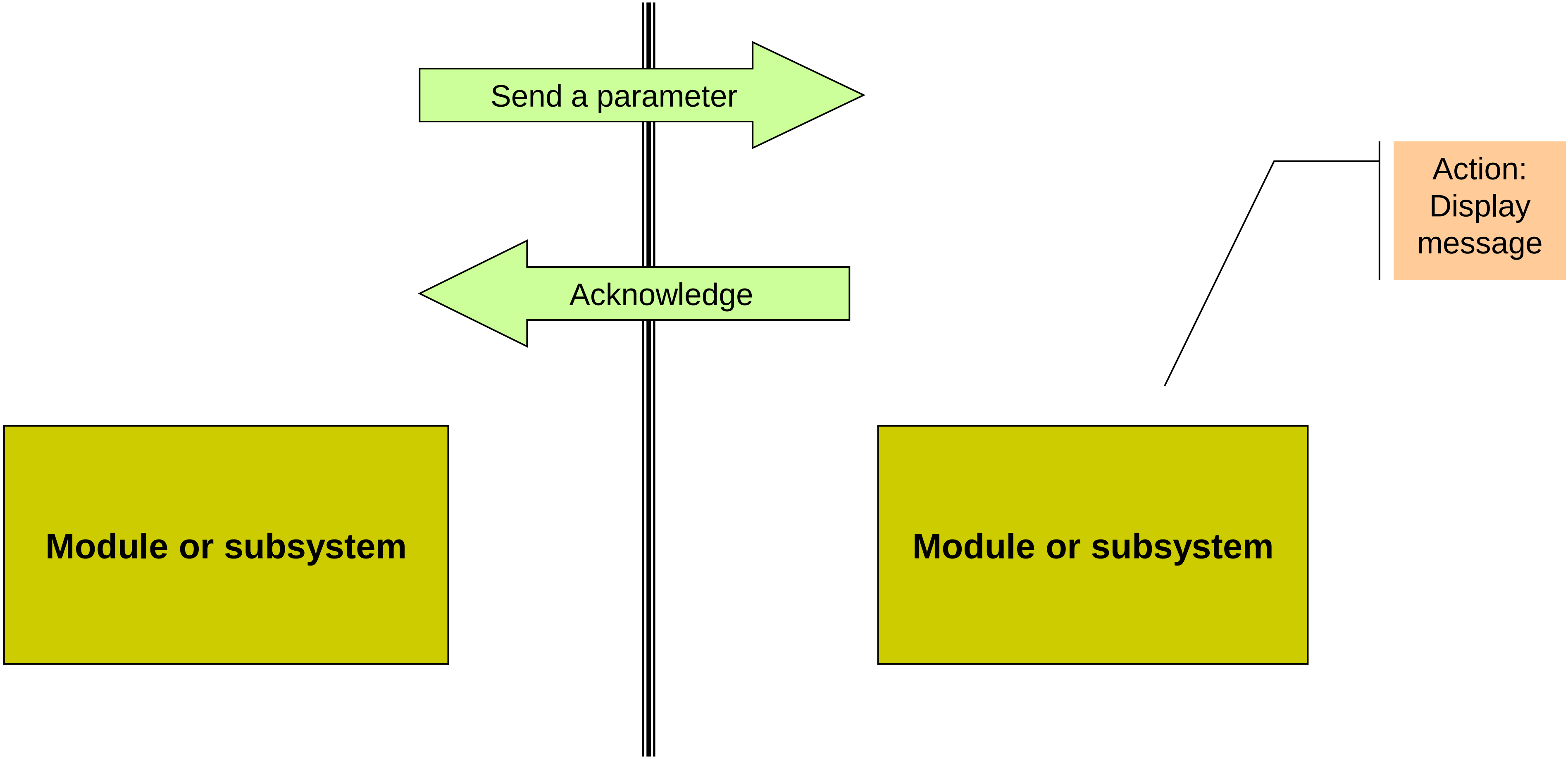
Management may not understand why it takes so long to fix an integration problem.

Comments: Not recommended except for very small systems.

Integration Testing

- Require that modules have been unit tested.
- Ask to see the unit test results.
- If a lot of bugs are found in integration testing that should have been found in unit test, send the module back to the owner.
- Integration testing should focus on interfaces.
- Interfaces are buggy.
- Parameter passing, data flow, call sequences, etc.
- If there is an interface spec, use it to derive test cases.

Integration Testing - Interfaces

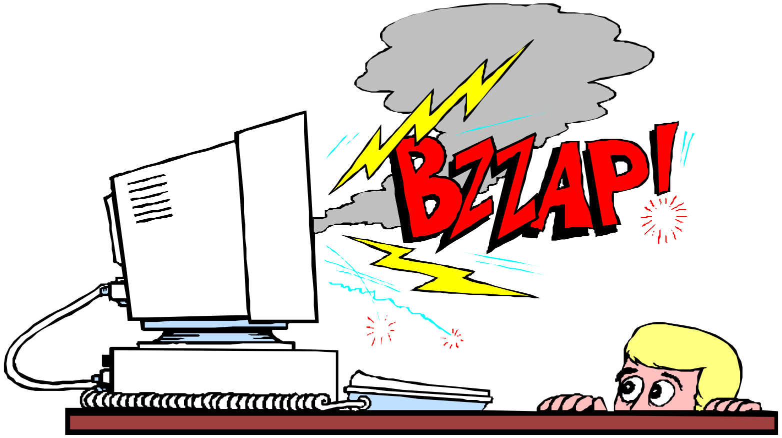


System Testing

- Integration is complete and a build is available.
- System testing is black box testing.
- Implementation doesn’t matter.
- This is where an independent test group comes into play.
- Requirements-based testing.
- Need to have a mechanism for providing inputs to the system and observing responses:
  - GUI
  - Printouts
  - Control actions
  - Instrumentation

Sources of System Requirements

- System Requirements Specification
- Interface Specifications
- Users Guide
- Use Cases
- Customers
- Domain Experts
- Bug Data
- Re-engineering



Derived Requirements

- Some requirements may not be stated.
- There is a concept of “fitness for use”.
- ”The system shall not crash.”
- What about safety?

System Test Categories

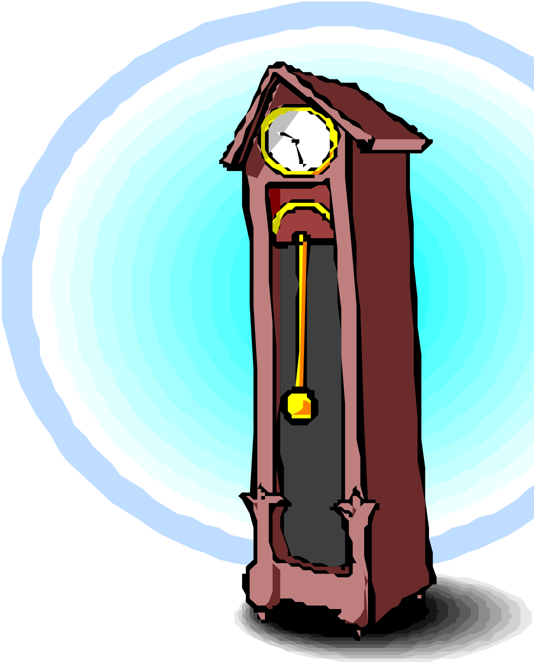
- **Functionality – To find problems in the functions and features.**
- **Reliability/Availability – To find problems based upon continuous running of the system.**
- **Load/Stress – To identify problems caused by peak load conditions.**
- **Volume – To find problems in the system's ability to process a heavy load continuously.**
- **Performance – To determine the actual response time and CPU loading conditions of the system.**
- **Installability – To identify problems in the installation procedures.**
- **Recovery – Force the system to fail and then find problems in the recovery processing of the system. .Particularly data.**
- **Security – To find holes in the system's security provision.**
- **Serviceability – Maintenance and repair procedures.**

Acceptance (Test) Demonstration

- Should be called a “demonstration, not a “test”.
- It’s purpose is to show that the system is ready to be shipped to the customer.
- Conducted on the complete system after all other testing (including system testing) has been done.
- This is one situation where the goal is not to find problems!
- Must have some basis for the testing. This will usually be the system requirements or some subset of them.
- Demonstration procedures (tests) to be performed must be agreed to by the customer.

Reliability Testing

- Consists of a continuous run under some approximation of normal load or operation.
- Many problems don’t appear until after some time of normal operation of the system.
- Intended to find problems with buffers overflowing, memory leaks, etc.
- Test tools will help a lot. Difficult to do manually. (Capture/Playback).



Random Events

Some problems only occur when:

- A certain sequence of events takes place.
- Events happen in a certain time frame.
- Particularly true for real time systems.

Software Reliability - Definition

- **Definition:** The probability of failure-free operation of the software for a specified period of time in a specified environment.
- **Key** aspects:
  - Given time period
  - Specified set of operating conditions
- **Range of values:** 0.000 to 1.000
- **Example:** A software application has a reliability of 0.93 for 24 hours when used in a typical manner. This means that the software would operate without failure over a 24 hour period for 93 out of 100 of those periods.

Source: Software Reliability, Musa et al, p.15

Assumptions – Reliability Calculation

- Released software is in use continuously.
- Each new version sees about the same number of users and about the same overall use profile.
- Examples: 1) Web site, 2) A single system in extended, continuous use

Software Reliability – Data Gathering

- Search the bug data base for bugs reported from field use.
- Look at a time period immediately after release of a new version.
- Must judiciously select the time period.
- Could also do an extended run in the lab.
- Count bugs that cause a system failure.
  - Must establish some criteria here
  - May want to categorize them by severity

Field Data - Example

Version 1.6 release date: Sept. 1

Number	Date	Severity	Version	Description
111	Aug. 15	5	1.5	Screen lay-out
112	Aug. 31	4	1.5	Screen lay-out
113	Sept 1	3	1.4	Menu tree problem.
114	Sept. 3	2	1.6	Incorrect temperature calculated.
115	Sept. 3	2	1.6	Wrong data displayed.
116	Sept. 3	3	1.6	Menu missing a selection.
117	Sept. 4	5	1.5	Wording is poor.
118	Sept. 4	1	1.6	Report look-up causes crash.
119	Sept. 5	3	1.6	Entry is lost.
120	Sept. 7	4	1.6	Screen lay-out poor.
121	Sept. 10	5	1.6	Spelling error

Software Reliability Formula

Formula (Source: Software Reliability, Musa et al, P.91)

$$\boldsymbol{R = \exp(-\lambda_t t)}$$

where R = reliability

λ_t = the number of failures/hour

t = the time period for which the reliability is to be calculated

Software Reliability Calculation

Example (using the data from the previous table)

- 55 failures in a 7 day (168 hour) period. The reliability for periods of usage of 24 hours in length is desired.

We have: $\lambda_t = 5/168 = 0.0298$ $\quad t = 24$

Therefore:

$R = \exp\ (-\lambda_t\, t) = \exp\ (-0.0298)\ (24) = 0.489$

What this tells us is that in 100 periods of time that are each 24 hours in length, this software will run failure-free (for all users) in 48.9 of those 24 hour periods.

Alternate Reliability Metric

- If the software versions are not in continuous use, a different reliability measure must be used.
- Mean Time Between Failure (MTBF)
- Concept of “production time” and “non-production time”
- Count failures, as before, but must also determine the number of hours that the software was in production when it incurred those failures.
- MTBF = total number of production hours divided by the total number of failures.
- Works best when data from multiple installations is aggregated.

System Testing Guidelines & Checklist

See separate handouts.

These are additional suggestions for system testing. Some we have discussed; some are in addition to the material presents in this class.

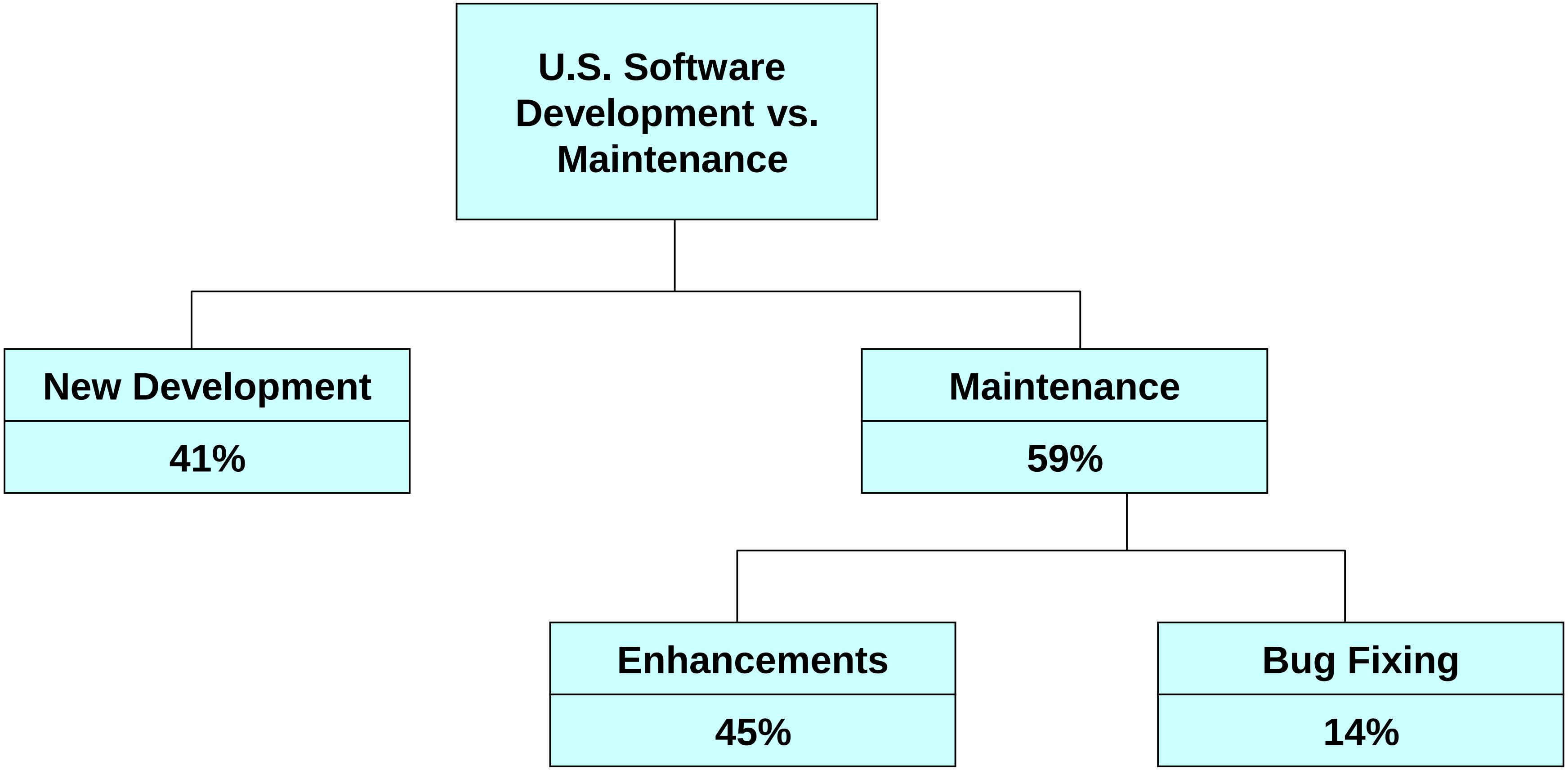
Regression Testing

- Looks for bugs in the unchanged portions of the software due to side effects from the changed portion.
- Comes into play when a series of new versions are being issued to a fielded software system (product upgrade mode).
- Software maintenance

Regression Testing – When Is It Done?

- During integration
  - On each system build.
- After the product has been fielded, and upgrade versions are being released.
  - On each new version before formal release.

Software Maintenance



U.S. Software Development vs. Maintenance

New Development 41%

Maintenance 59%

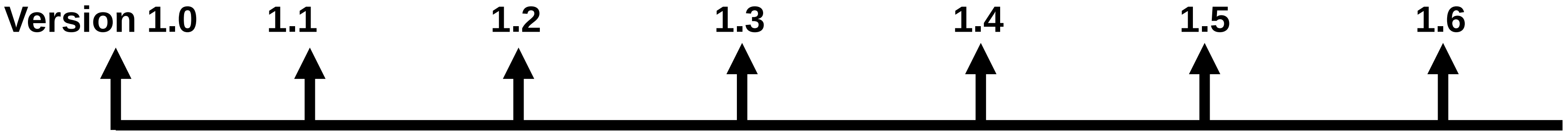
Enhancements 45%

Bug Fixing 14%

Source: Applied Software Measurement, Capers Jones

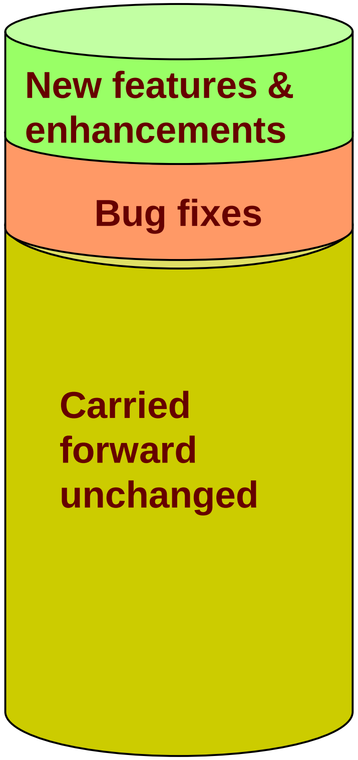
Product Upgrades

- There is a fielded software product (stand-alone or embedded).
- New versions are being issued on a regular, periodic basis.
- The new versions are incremental upgrades and enhancements to the previous version (not next generation).

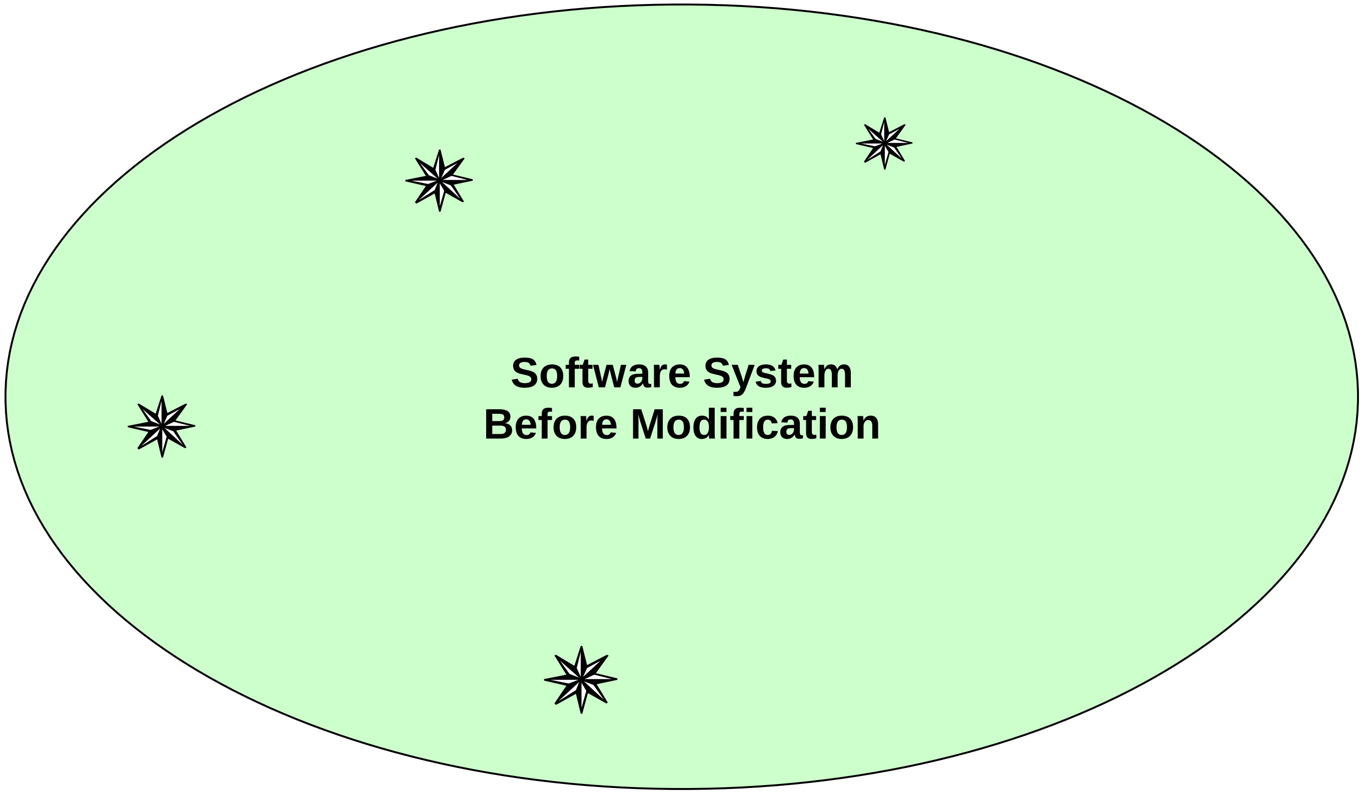


Assumptions Regarding Regression Testing

- **Each new version is an incremental change to the previous version.**
- **The amount of change is roughly the same in each version.**
- **The proportion of new features and bug fixes in each release is approximately constant.**
- **Each new version sees a similar usage profile and degree of usage.**

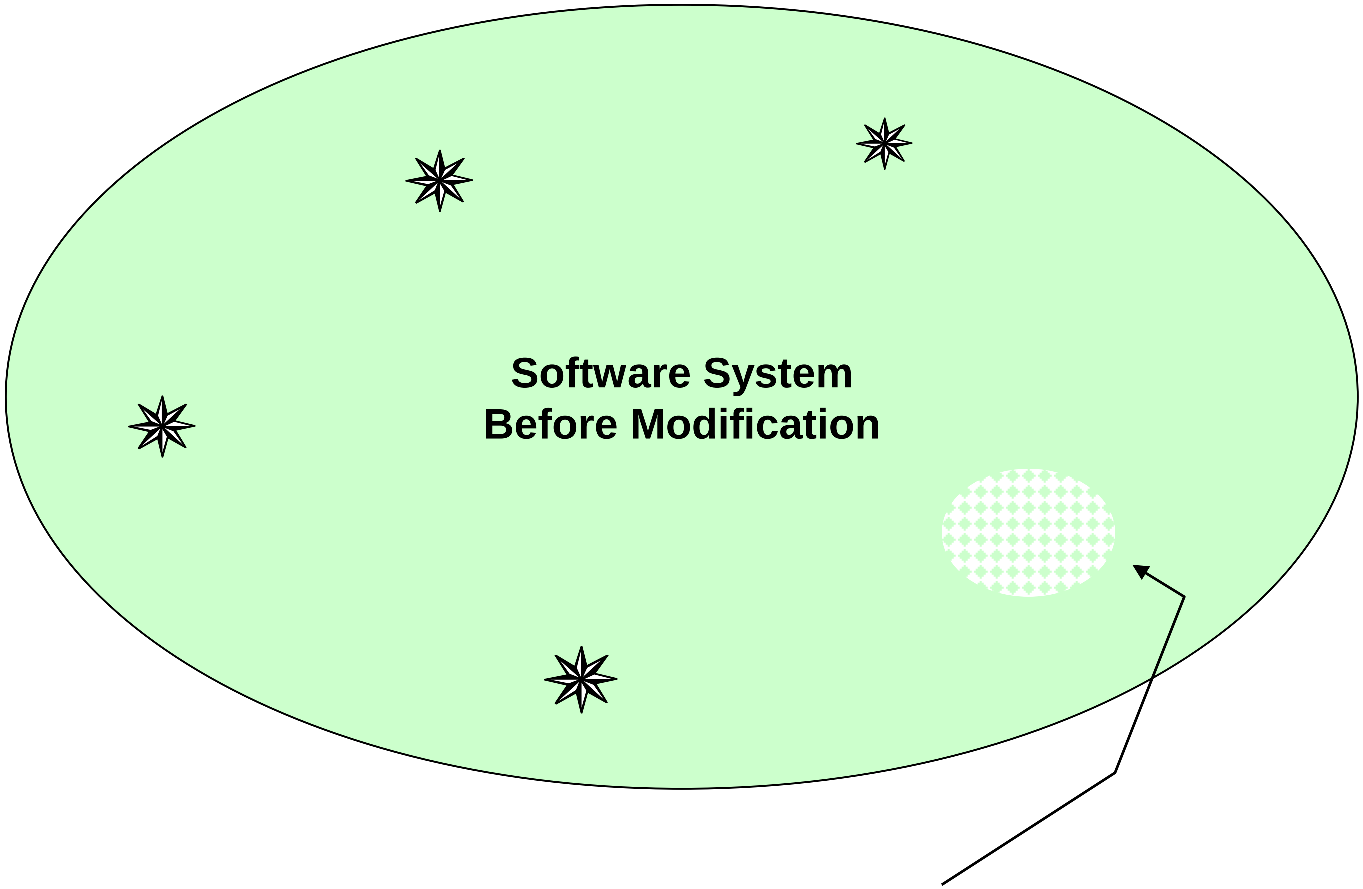


The Regression Problem - 1



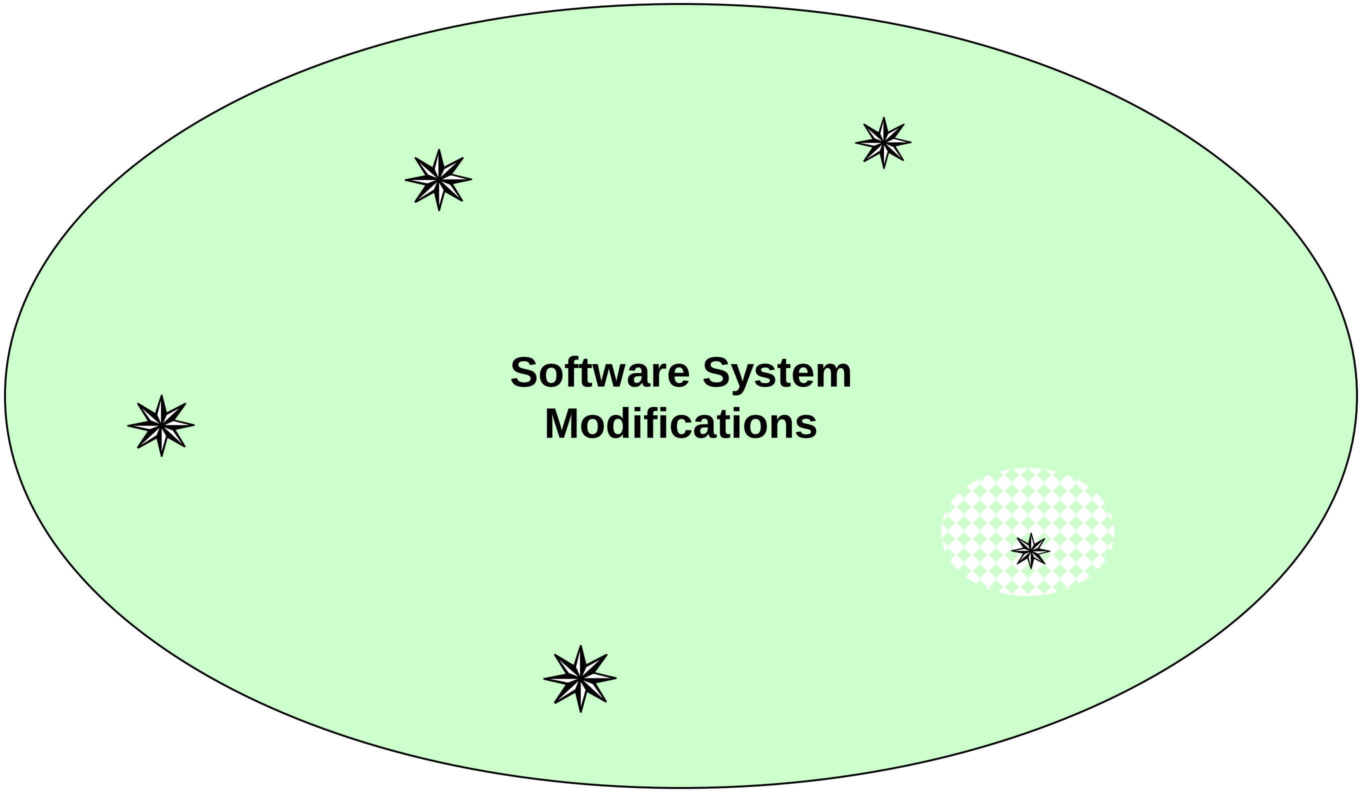
Distribution of Bugs

The Regression Problem - 2



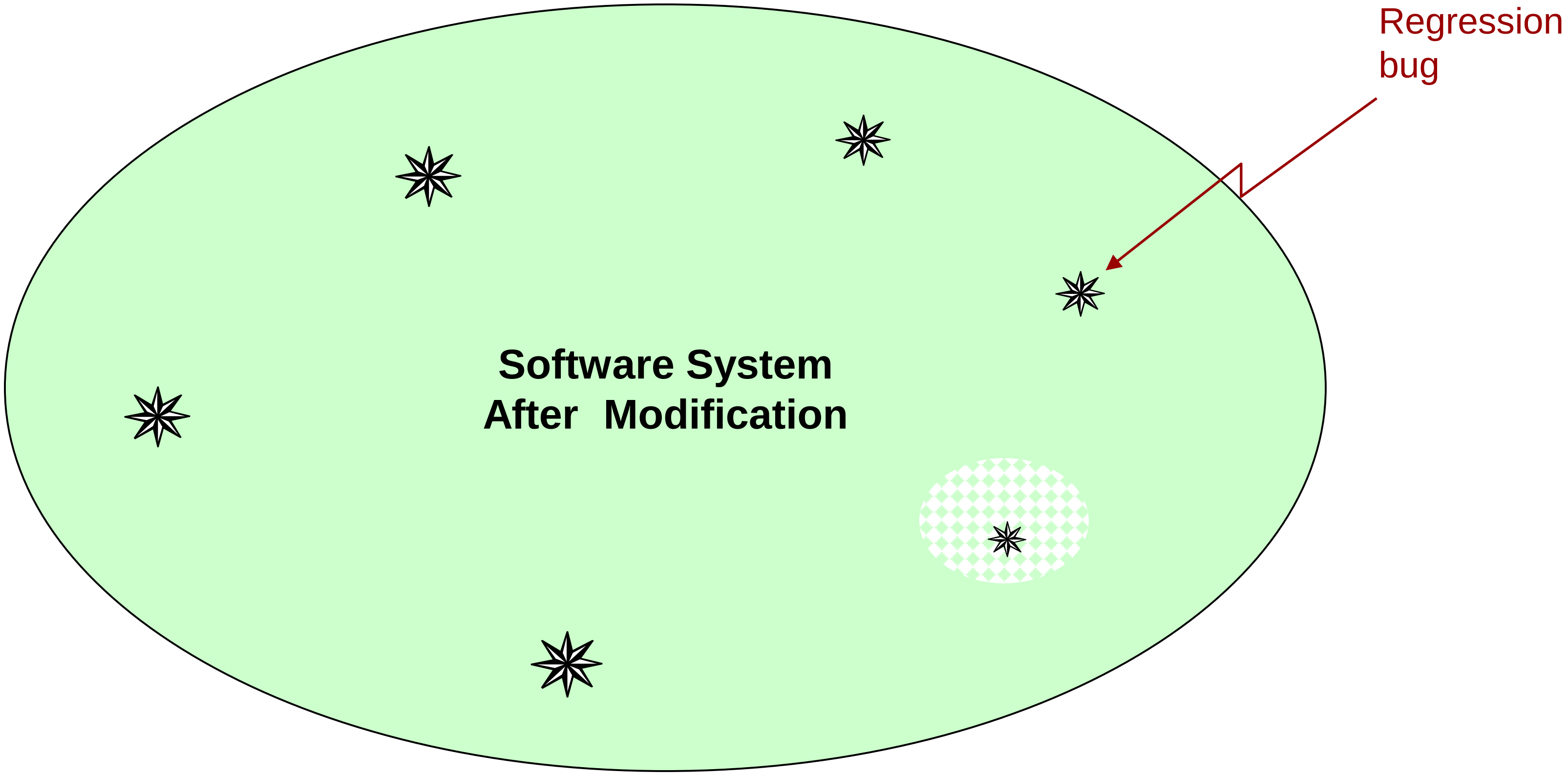
Modifications are made.

The Regression Problem - 3



Expected bug distribution after modifications are made.

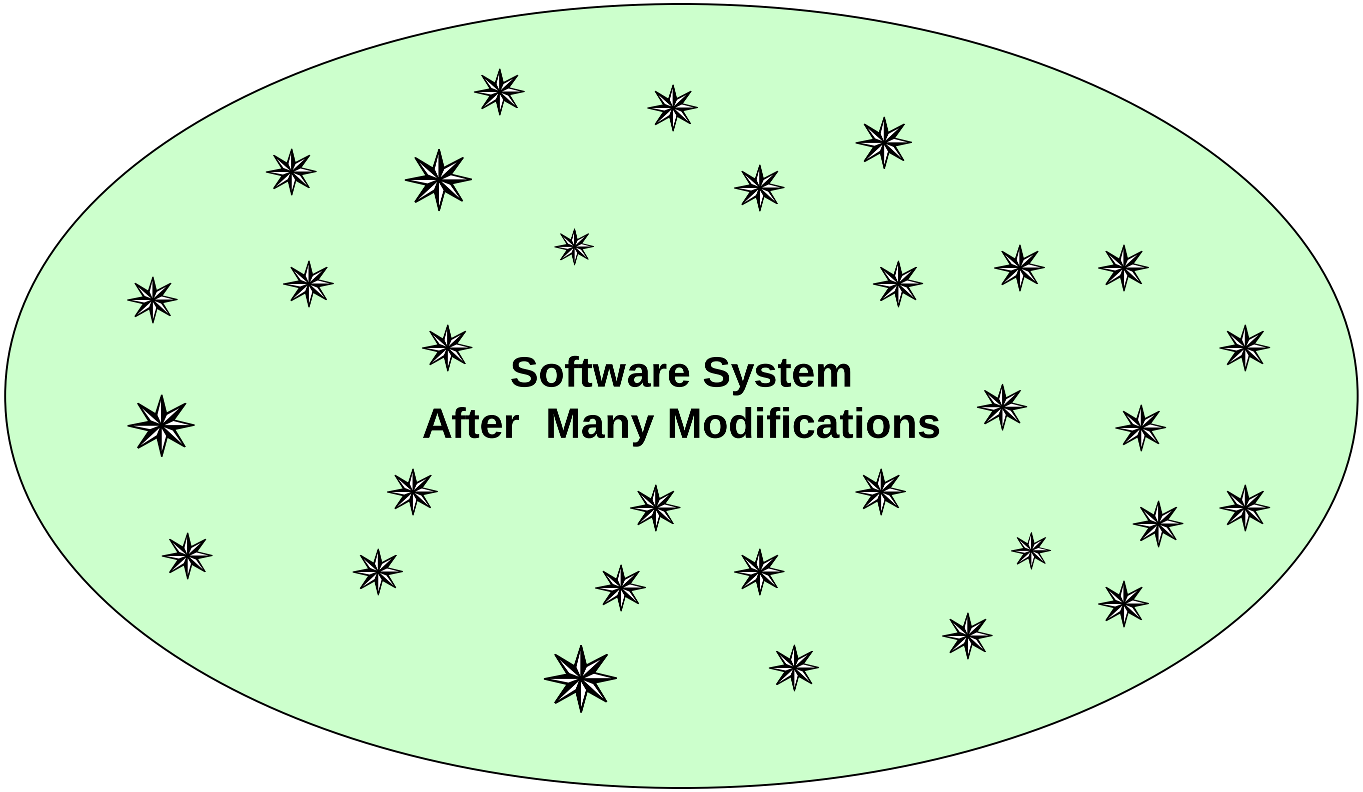
The Regression Problem - 4



Actual bug distribution after modifications are made.

Regression – The Danger

- If we only test the new and/or changed portions of the code, we will miss the regression bugs.
- Over time, they accumulate, and soon we have a monster!!



Test Planning for an Upgrade Version

Progressive

- Design test cases for the new (or enhanced) features. Use the defined requirements as a guide.
- Test each bug fix. Use the bug report(s) as a guide.

Regressive

- Design test cases to find new bugs in the unchanged portions of the code (regression testing).

Planning Regression Testing

- Use bug data ✓
  - Most common types of bugs
  - Error prone modules
- Customer usage scenarios ✓
- Module complexity ✓
- Sub-system or system interfaces ✓
- Most critical functions ✓
- Use test cases that have been shown to be effective. ✓

Regression Test Suite

- Put selected test cases into a regression test suite.
  - Add to it as new features are added to the software (from the test cases defined for those new features).
  - Add & revise regression test suite based upon results from the field.
- May want to have a “full” regression test suite and a “partial” suite.
  - Run the partial suite frequently
  - Run the full suite less frequently.

Regression Testing – How Much To Do?

- Once again – the issue of “when to stop”.
- Based upon experience.

Regression Testing Guidelines

See separate handouts.

These are additional suggestions for regression testing. Some we have discussed; some are in addition to the material presents in this class.

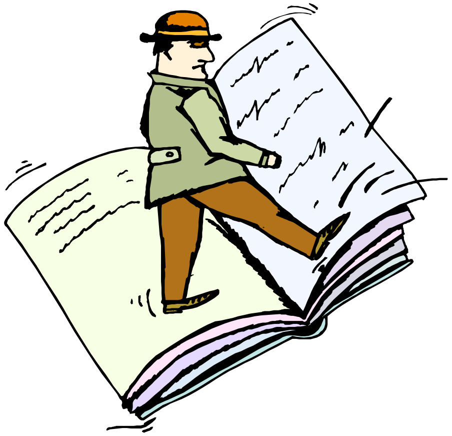
Requirements Based Testing

Basic Approach to System Testing

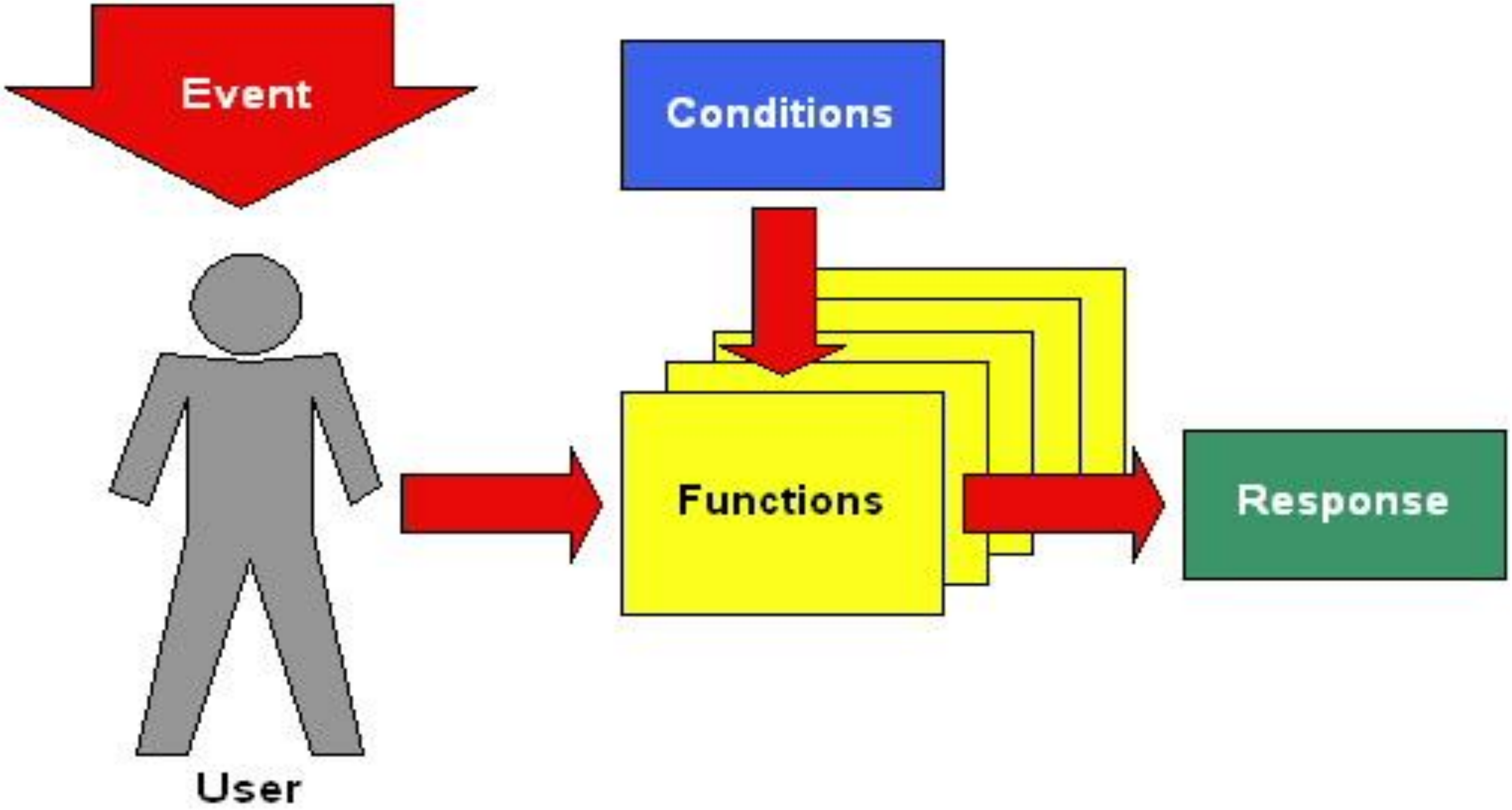
- One of the fundamental approaches to system testing.
- Use the system requirements to derive test cases:
  - Software Requirements Specification (SRS)
  - Marketing Requirements Specification (MRS)
  - Product Specification
  - Users Guide
  - Etc.

Users Guide

- A very effective way of doing system level testing.
- Especially when there is no SRS.
- Added benefit: It debugs the Users Guide.
- Go through it paragraph-by-paragraph



Model of a Requirement



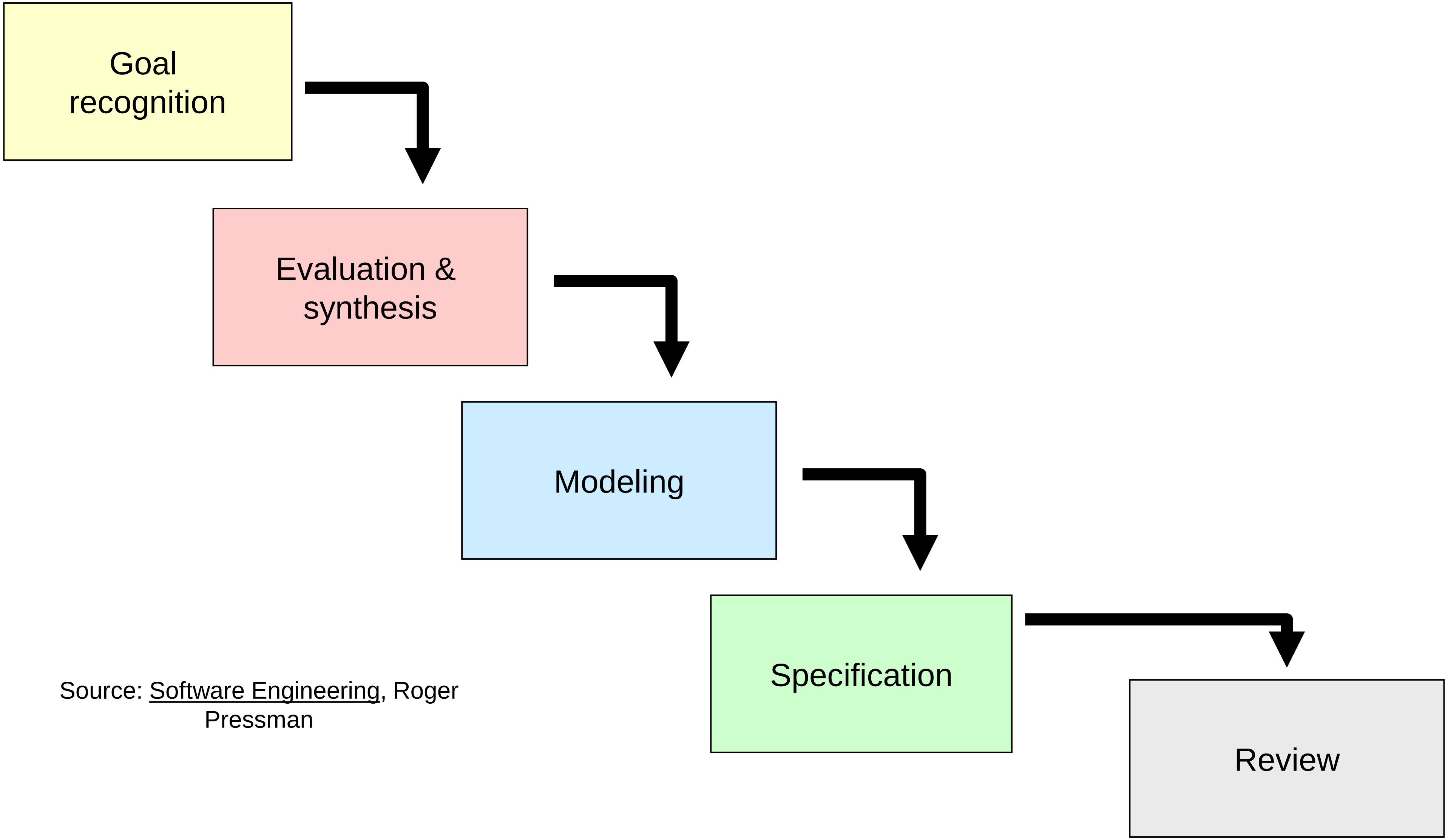
Source - SEMATECH Semiconductor Industry Standards Conformance Guidelines: Assessment Criteria and Processes

Aspects of Requirements Analysis

- A complete understanding of requirements is essential to the success of a software development project.
- Excellent coding and design cannot make-up for poor requirements analysis and specification.
- The requirements drive the design, coding, and testing.
- Appears to be simple, but it isn’t.

Source: Software Engineering, Roger Pressman

Requirements Analysis Process

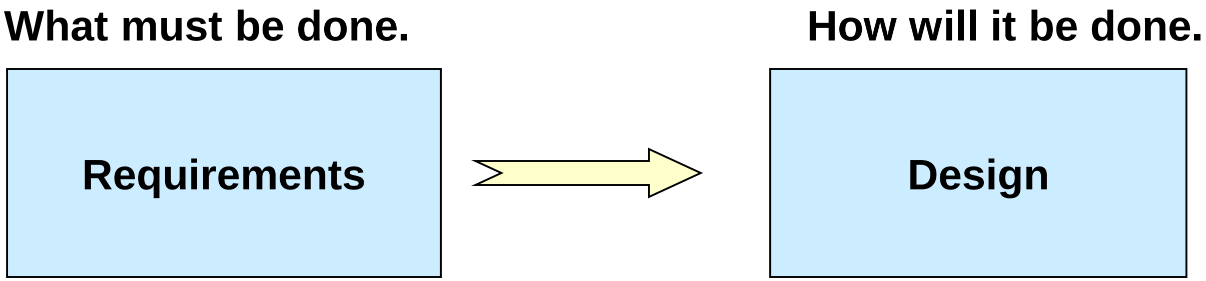


Source: Software Engineering, Roger Pressman

The Nature of Requirements

Requirements should specify the “*what*”, not *“how”*.

- *What* data needs to be produced?
- In *what* format.
- *What* calculations must be performed.
- *What* are the interfaces that must be accommodated.
- *What* actions will the user perform.
- *What* features and functions are needed.



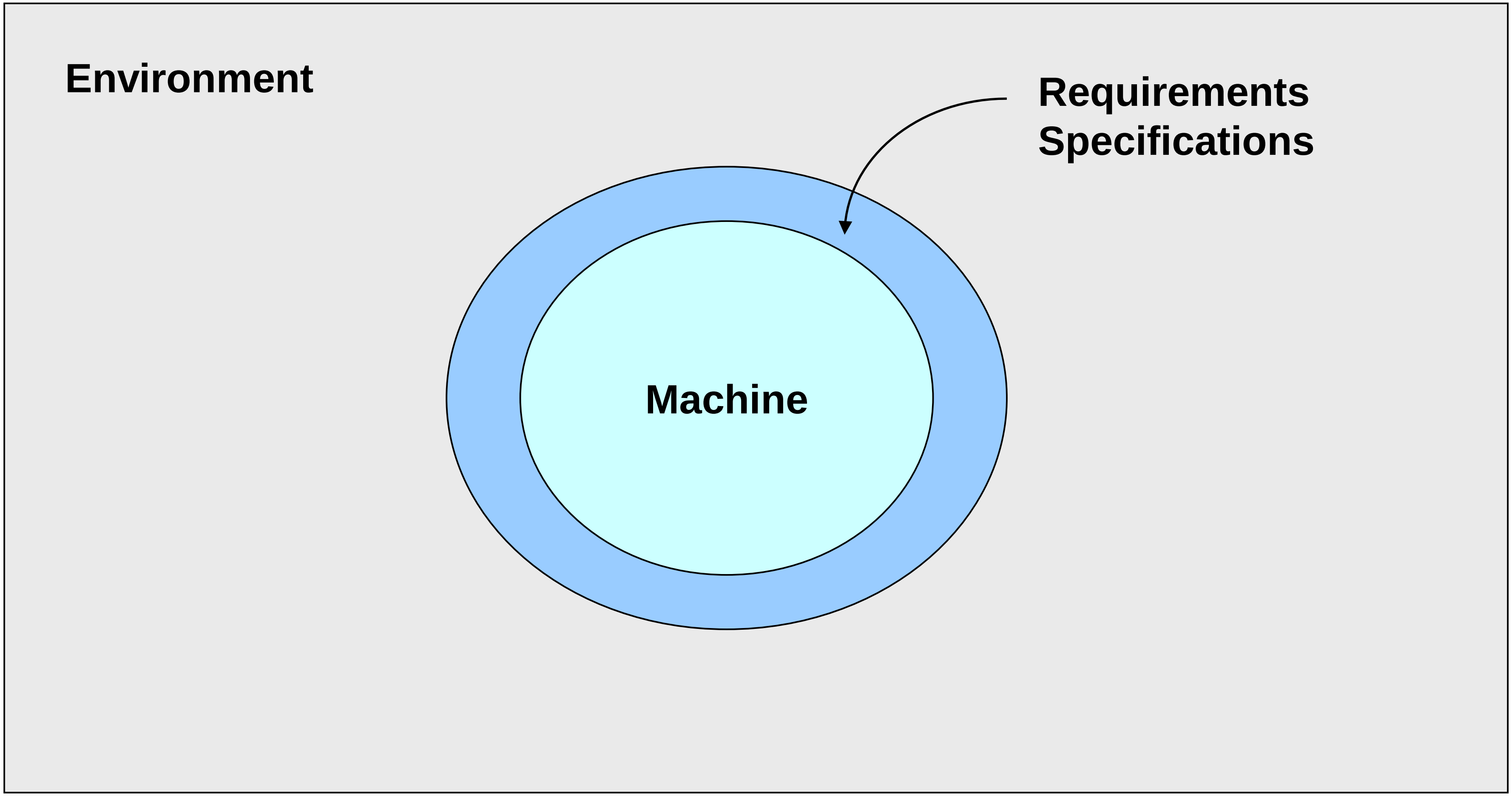
What must be done.

How will it be done.

Requirements

Design

Requirements, Specifications, Machines



Source: Software Requirements and Specifications, M. Jackson

Requirements

- **Stated in the language of the problem domain**
  - Standard problem frames
- **Describe the “givens”**
  - Components and shared phenomenon
  - Cause-effect dependencies
  - Equations of state, relations
  - Physical laws, expectations (safety, reliability)
  - Economic constraints
  - Legal constraints
- **Express the “to be’s”**
  - Transformations
  - Relation sot be established, conditions to be met
  - Historical references

Source: “Tutorial on Software Testing”, Dr. Dwayne Knirk, Sandia National Laboratories, Jan., 1997

Specifications

- Stated in the language of shared phenomena.
  - Standard interaction patterns.
- Describes the interactions between the environment and the machine.
  - Direct effect: input, output
  - Representation: digital, analog
  - Presence: continuous, discreet
  - Values: data symbols, event times
- Expresses interaction sequences and coordination.
  - Stimulus – response interactions (cause – effect)
  - Internal “real world” model.
  - Serialization and concurrency.

Source: “Tutorial on Software Testing”, Dr. Dwayne Knirk, Sandia National Laboratories, Jan., 1997

Recording Requirements & Specifications

- The Software Requirements Specification (SRS)
- IEEE Std 830-1998, “IEEE Recommended Practice for Software Requirements Specifications.

Characteristics of a Good SRS

An SRS should be:

- □ Correct
- □ Unambiguous
- □ Complete
- □ Consistent
- □ Ranked for importance and/or stability
- □ Verifiable
- □ Modifiable
- □ Traceable

Source: “IEEE Std 830-1998 Recommended Practice for Software Requirements Specifications”

Contents of a Good SRS

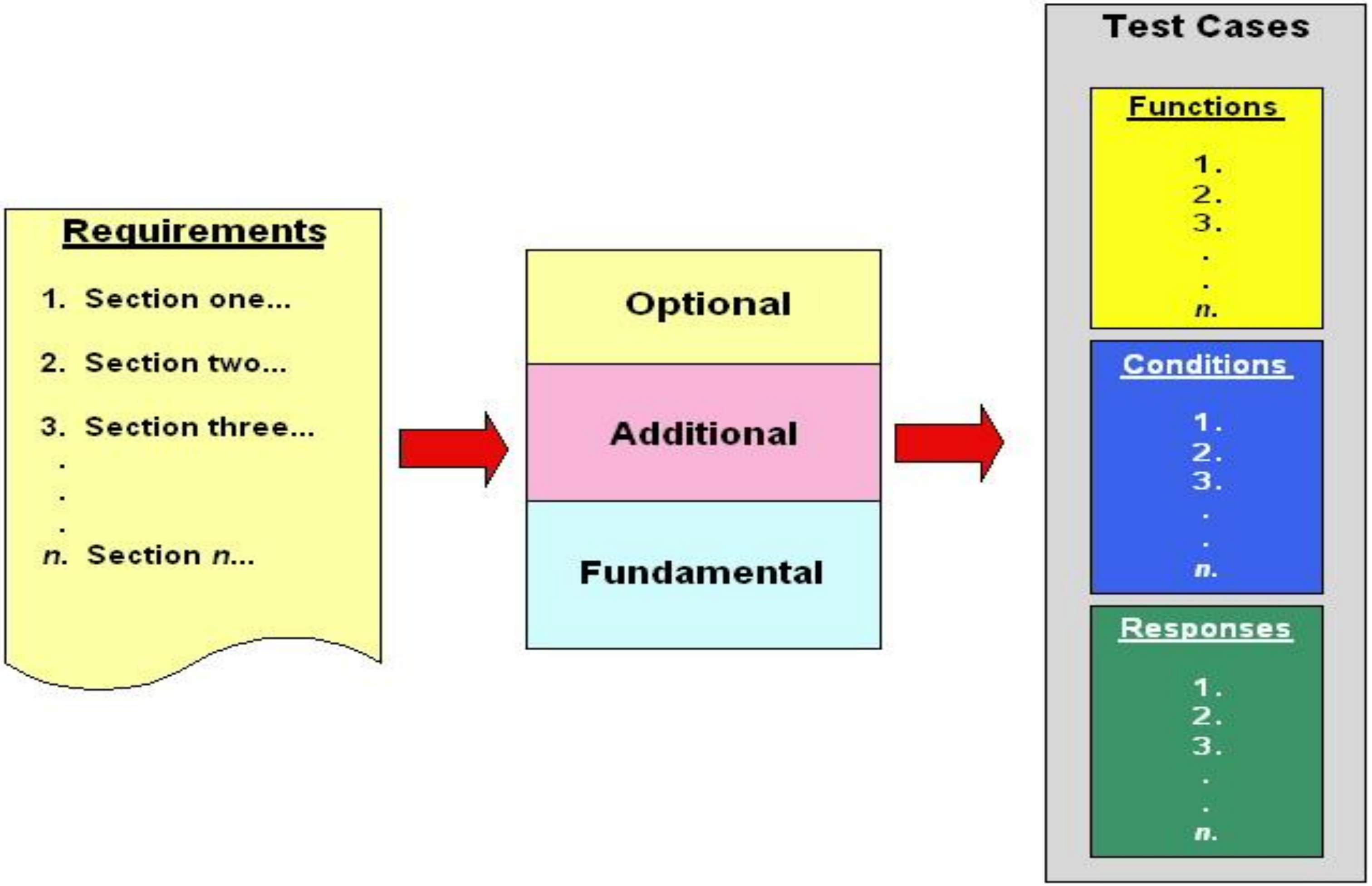
1. **Introduction**
    - 1.1 Purpose
    - 1.2 Scope
    - 1.3 Definitions, Acronyms, & Abbreviations

2. **Overall Description**
    - 2.1 Product Perspective
    - 2.2 Product Functions
    - 2.3 User Characteristics
    - 2.4 Constraints
    - 2.5 Assumptions & Dependencies

3. **Specific Requirements**
    - 3.1 External Interfaces
    - 3.2 Functions
    - 3.3 Performance Requirements
    - 3.4 Logical Database Requirements
    - 3.5 Design Constraints
    - 3.6 Software System Attributes
        - 3.6.1 Reliability
        - 3.6.2 Availability
        - 3.6.3 Security
        - 3.6.4 Maintainability
        - 3.6.5 Portability
        - Others

Source: “IEEE Std 830-1998 Recommended Practice for Software Requirements Specifications”

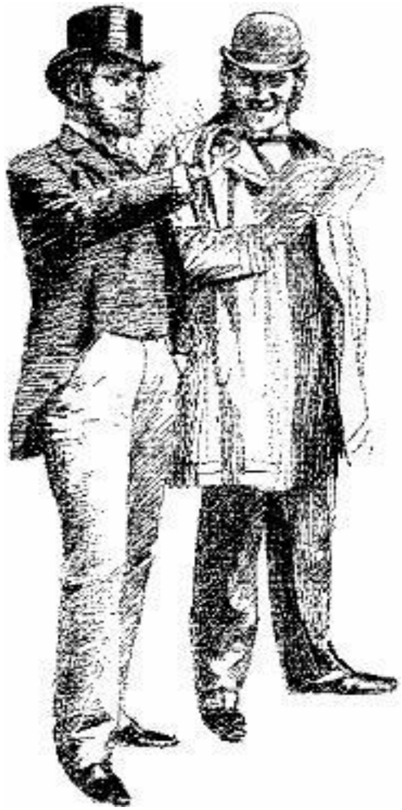
Deriving Test Cases From Requirements



Source - SEMATECH Semiconductor Industry Standards Conformance Guidelines: Assessment Criteria and Processes

Requirements Communication Difficulties

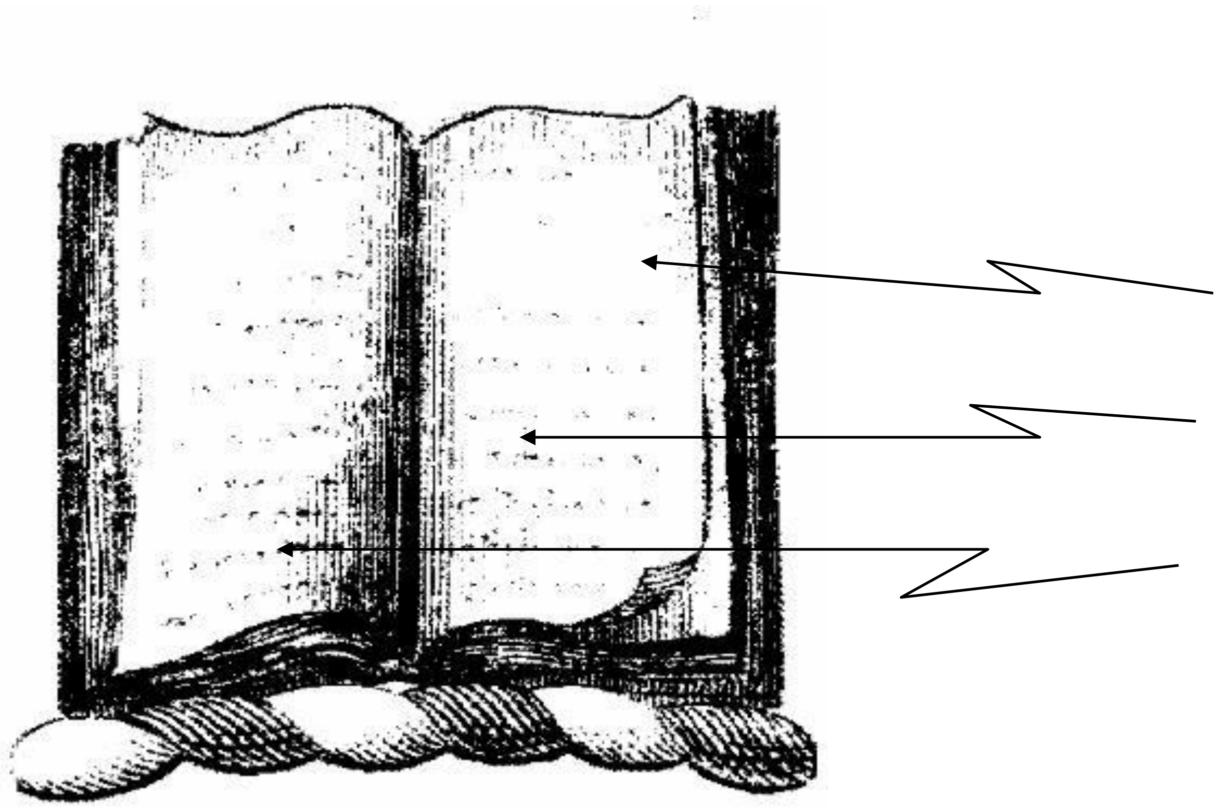
- Requirements analysis and definition is a communication intensive activity.
- Every communications activity must have a sender and a receiver.
- With any communications, there is a signal-to-noise ratio.
- Communication errors occur on both ends.



Problem – Identifying Requirements in the Spec

- A requirements document is a communication vehicle.
- Somewhere in there is the message.
- It is not always easy to find the specific, individual requirements.

What Form Do Requirements Take



Requirements

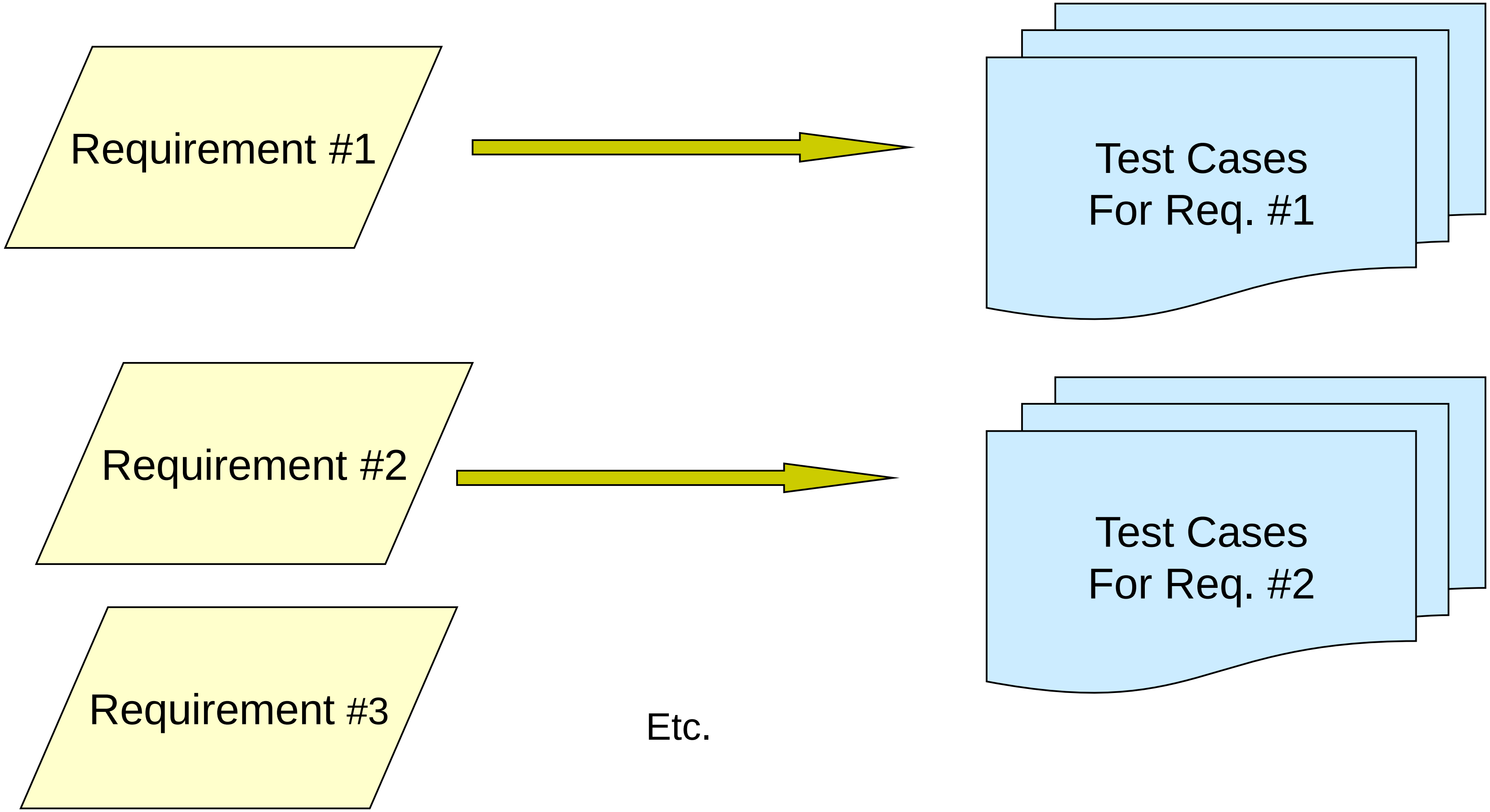
- Text
- Tables
- Diagrams

Requirements Extraction

- Must perform an analysis on the requirements specification.
- Read through it and make a list of all of the specific requirements:
- Two methods:
  - Annotate in the requirements document…
  - Put into a separate list.

Test Case Design

For each identified requirement; define test cases.



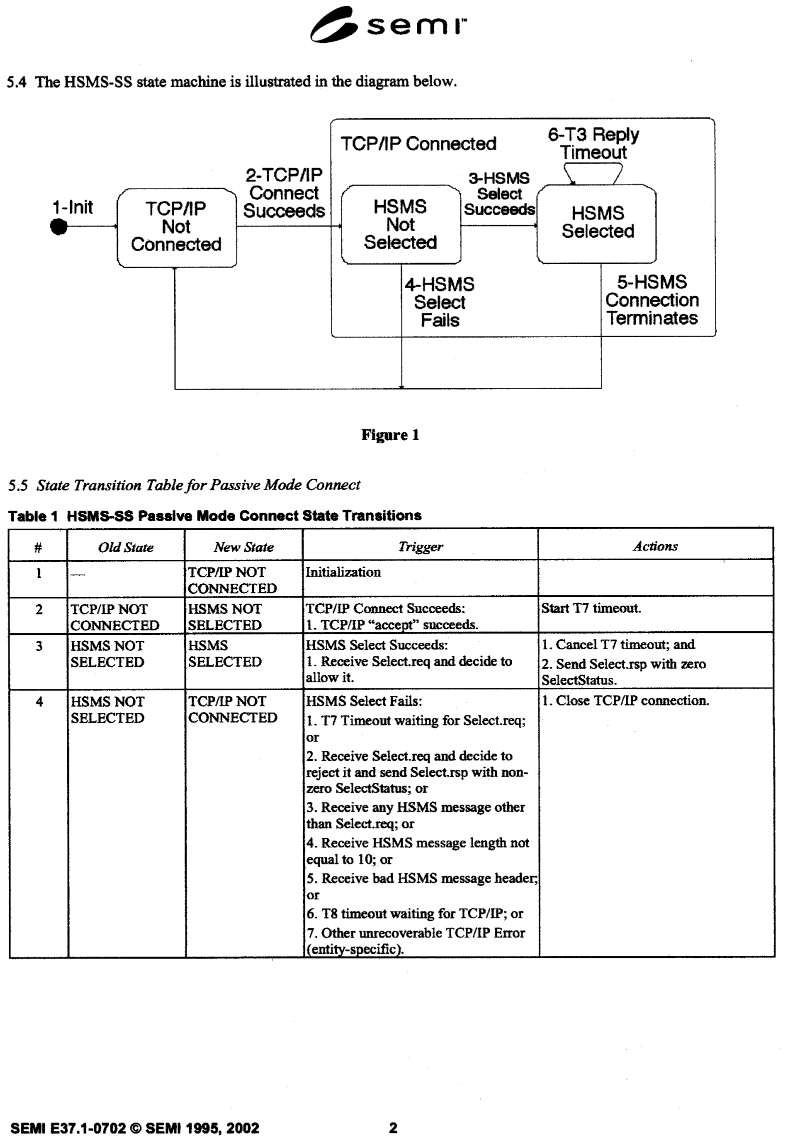
Requirements Extraction – Class Exercise

A Standard

SEMI E37.1-0702

High –speed SECS Message Service Single Selected-session Mode (HSMS-SS)

E37.1 – Extracted Requirements



5.4 The HSMS-SS state machine is illustrated in the diagram below.

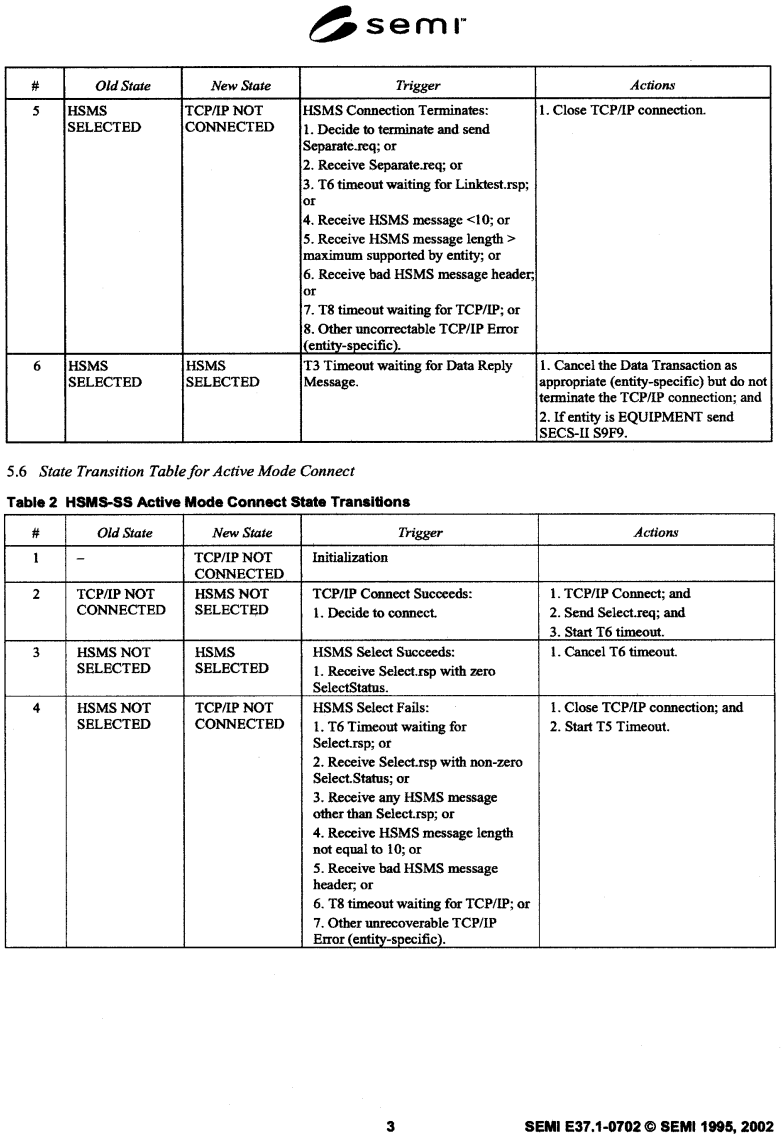
TCP/IP Connected
6-T3 Reply Timeout
1-Init
TCP/IP Not Connected
2-TCP/IP Connect Succeeds
HSMS Not Selected
3-HSMS Select Succeeds
HSMS Selected
4-HSMS Select Fails
5-HSMS Connection Terminates

Figure 1

5.5 *State Transition Table for Passive Mode Connect*

Table 1 HSMS-SS Passive Mode Connect State Transitions

#	*Old State*	*New State*	*Trigger*	*Actions*
1	—	TCP/IP NOT CONNECTED	Initialization	
2	TCP/IP NOT CONNECTED	HSMS NOT SELECTED	TCP/IP Connect Succeeds: 1. TCP/IP "accept" succeeds.	Start T7 timeout.
3	HSMS NOT SELECTED	HSMS SELECTED	HSMS Select Succeeds: 1. Receive Select.req and decide to allow it.	1. Cancel T7 timeout; and 2. Send Select.rsp with zero SelectStatus.
4	HSMS NOT SELECTED	TCP/IP NOT CONNECTED	HSMS Select Fails: 1. T7 Timeout waiting for Select.req; or 2. Receive Select.req and decide to reject it and send Select.rsp with non-zero SelectStatus; or 3. Receive any HSMS message other than Select.req; or 4. Receive HSMS message length not equal to 10; or 5. Receive bad HSMS message header; or 6. T8 timeout waiting for TCP/IP; or 7. Other unrecoverable TCP/IP Error (entity-specific).	1. Close TCP/IP connection.



#	*Old State*	*New State*	*Trigger*	*Actions*
5	HSMS SELECTED	TCP/IP NOT CONNECTED	HSMS Connection Terminates: 1. Decide to terminate and send Separate.req; or 2. Receive Separate.req; or 3. T6 timeout waiting for Linktest.rsp; or 4. Receive HSMS message <10; or 5. Receive HSMS message length > maximum supported by entity; or 6. Receive bad HSMS message header; or 7. T8 timeout waiting for TCP/IP; or 8. Other uncorrectable TCP/IP Error (entity-specific).	1. Close TCP/IP connection.
6	HSMS SELECTED	HSMS SELECTED	T3 Timeout waiting for Data Reply Message.	1. Cancel the Data Transaction as appropriate (entity-specific) but do not terminate the TCP/IP connection; and 2. If entity is EQUIPMENT send SECS-II S9F9.

5.6 *State Transition Table for Active Mode Connect*

Table 2 HSMS-SS Active Mode Connect State Transitions

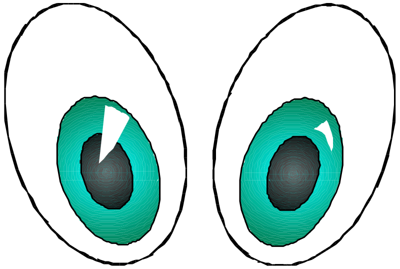
#	*Old State*	*New State*	*Trigger*	*Actions*
1	–	TCP/IP NOT CONNECTED	Initialization	
2	TCP/IP NOT CONNECTED	HSMS NOT SELECTED	TCP/IP Connect Succeeds: 1. Decide to connect.	1. TCP/IP Connect; and 2. Send Select.req; and 3. Start T6 timeout.
3	HSMS NOT SELECTED	HSMS SELECTED	HSMS Select Succeeds: 1. Receive Select.rsp with zero SelectStatus.	1. Cancel T6 timeout.
4	HSMS NOT SELECTED	TCP/IP NOT CONNECTED	HSMS Select Fails: 1. T6 Timeout waiting for Select.rsp; or 2. Receive Select.rsp with non-zero Select.Status; or 3. Receive any HSMS message other than Select.rsp; or 4. Receive HSMS message length not equal to 10; or 5. Receive bad HSMS message header; or 6. T8 timeout waiting for TCP/IP; or 7. Other unrecoverable TCP/IP Error (entity-specific).	1. Close TCP/IP connection; and 2. Start T5 Timeout.

Example – Test Case Definition

Modes allowed by the standard	Section - Paragraph - Sentence	Test Case ID	Test Case	Category	Comments	Modes allowed by the standard	Section - Paragraph - Sentence
Passive connect mode: The passive mode is used when the local entity listens for and accepts a connect procedure initiated by the Remote entity.	5.5 (Table 1) - Item 1	TR-1		Passive Mode Connect - From *No State* to *TCP/IP Not Connected*	1) Table 5.5. defines triggers and expected responses. 2) What about state transitions not allowed? Are there some that are of particular concern? Are they testable?	Passive connect mode: The passive mode is used when the local entity listens for and accepts a connect procedure initiated by the Remote entity.	5.5 (Table 1) - Item 1
Passive connect mode	5.5 (Table 1) - Item 2	TR-2	Initially, no network connection is established. Local entity obtains a connection endpoint and binds it to a published port. Remote entity sends a "connect request". Expected Results: Local entity sends an "accept response" and starts T7 timer.	Passive Mode Connect - From *TCP/IP Not Connected* to *HSMS Not Selected*	Note: Local entity has now entered "connected" state.	Passive connect mode	5.5 (Table 1) - Item 2
Passive connect mode	5.5 (Table 1) - Item 3	TR-3	Local entity is in "connected" state. Remote entity sends a "select request". Expected Results: Local entity sends a "select response" message with zero as the Select Status. No T7 timeout occurs.	Passive mode Connect - From *HSMS Not Selected* to *HSMS Selected*		Passive connect mode	5.5 (Table 1) - Item 3

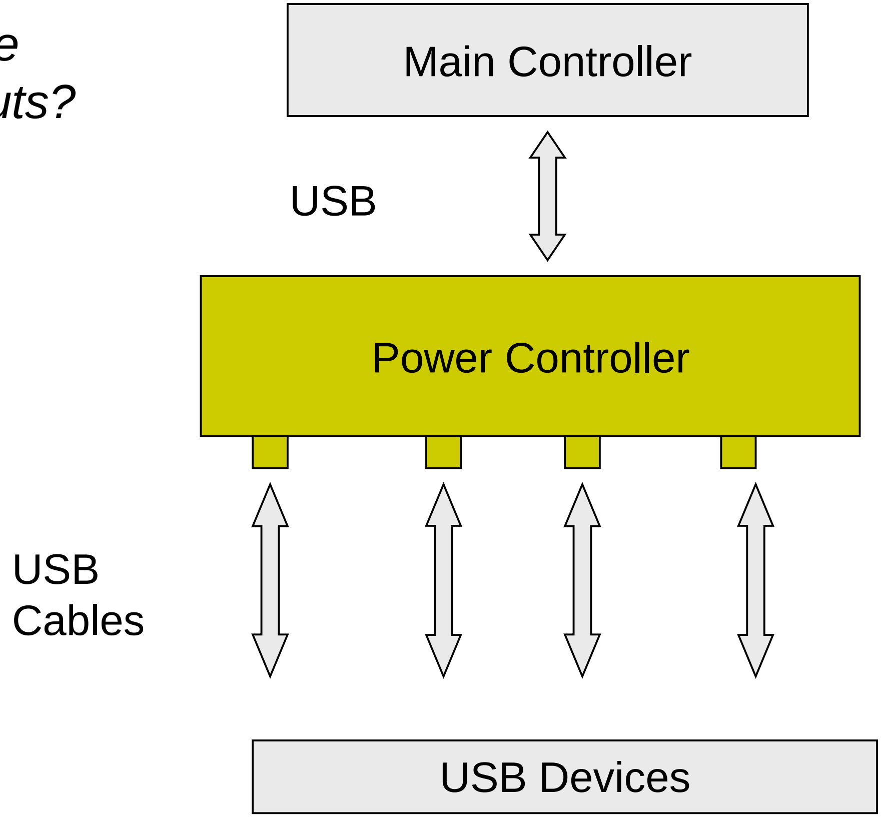
Observability of Test Results

- With a GUI
  - Use it
- Without a GUI
  - Depends on the nature of the system
    - Embedded control
    - Batch
    - Specialized applications
    - Others



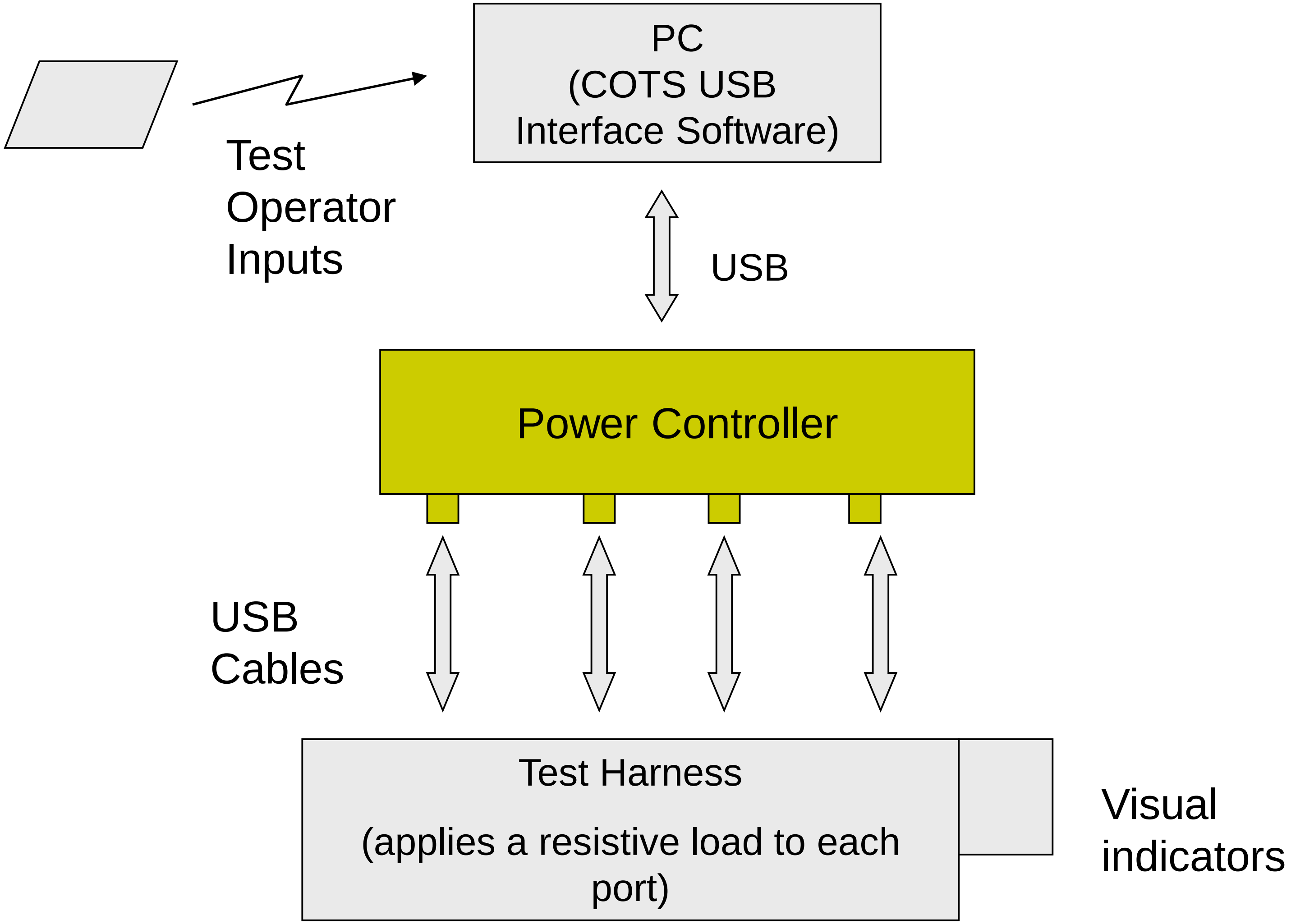
Example 1– Power Supply Controller

How to provide controlled inputs?

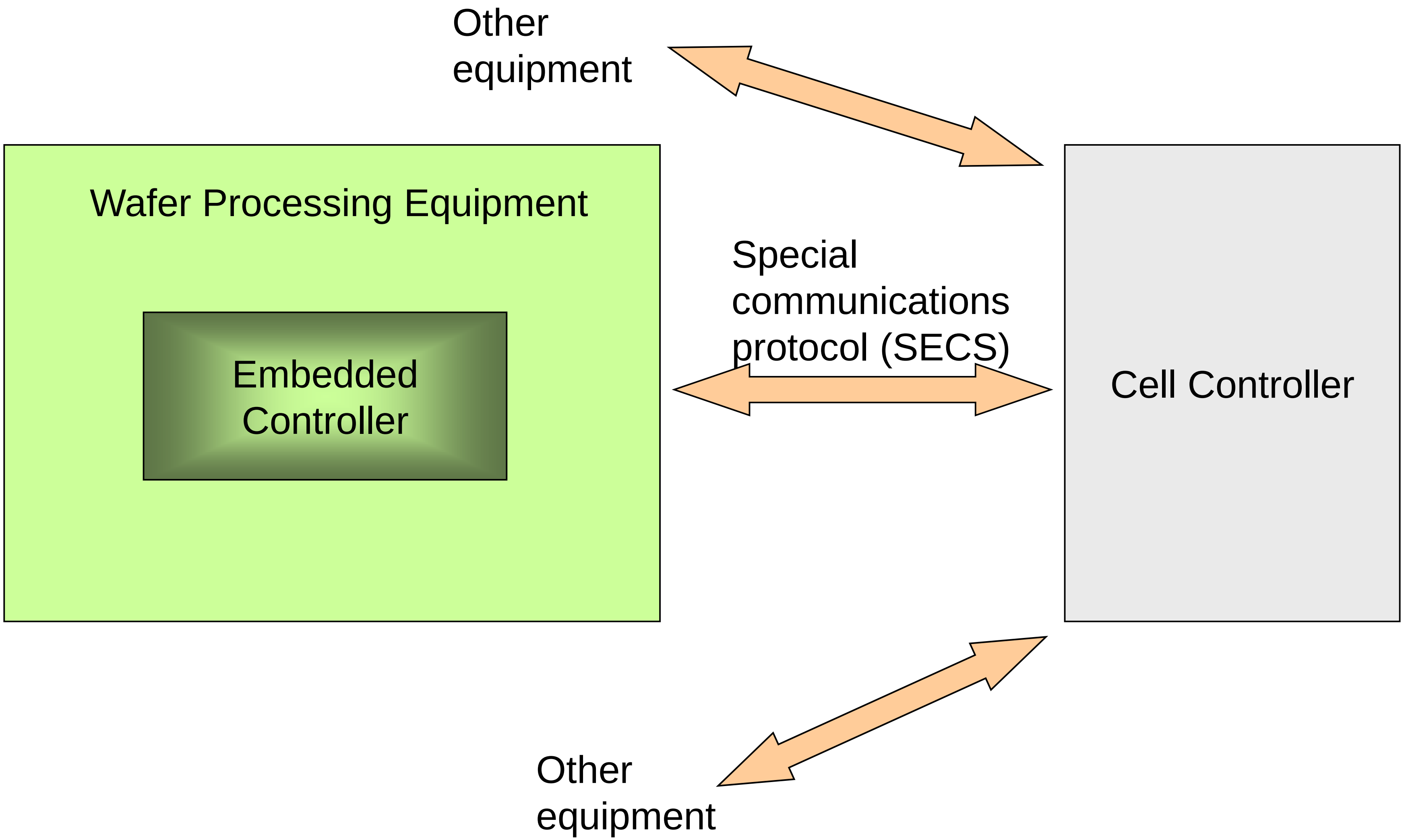


How to observe responses?

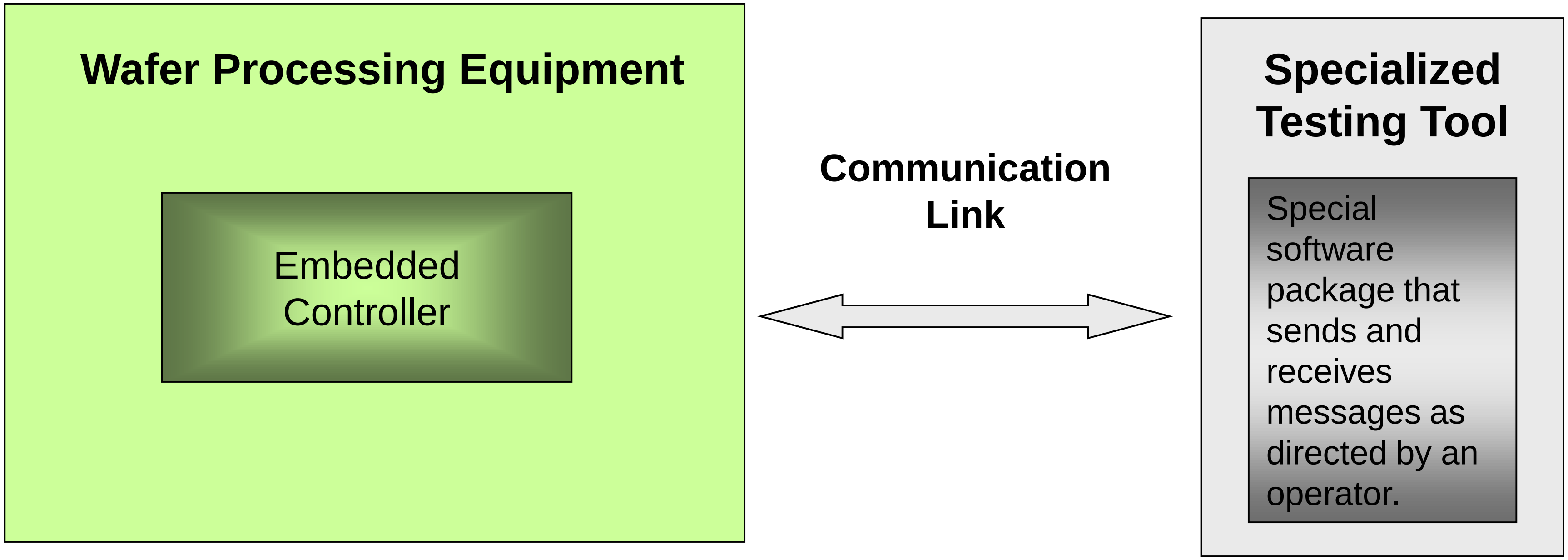
Example 1 - Solution



Example 2 – Communications Link


Other equipment
Wafer Processing Equipment
Embedded Controller
Special communications protocol (SECS)
Cell Controller
Other equipment

Example 2 - Solution

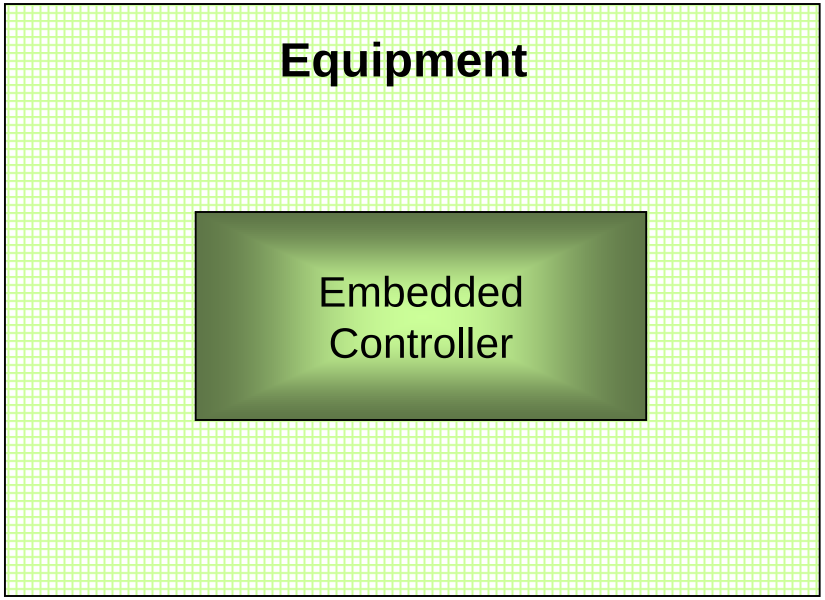

Wafer Processing Equipment
Embedded Controller
Communication Link
Specialized Testing Tool
Special software package that sends and receives messages as directed by an operator.

Test Cases:

Send designated messages.

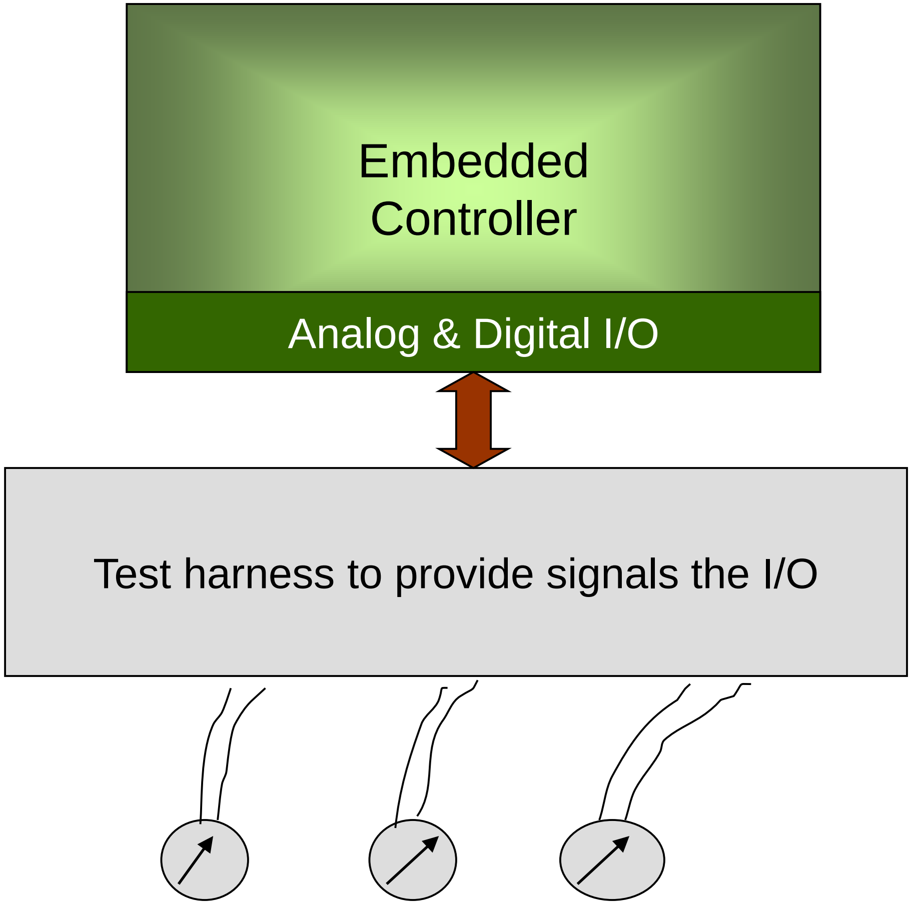
Look for proper messages back.

Example 3 – Embedded Control System



How do we test the controller if we don’t have the equipment available that it controls?

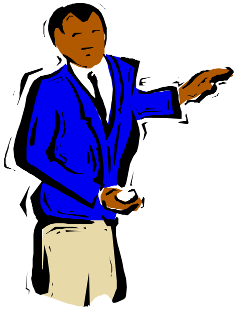
Example 3 – Solution 1



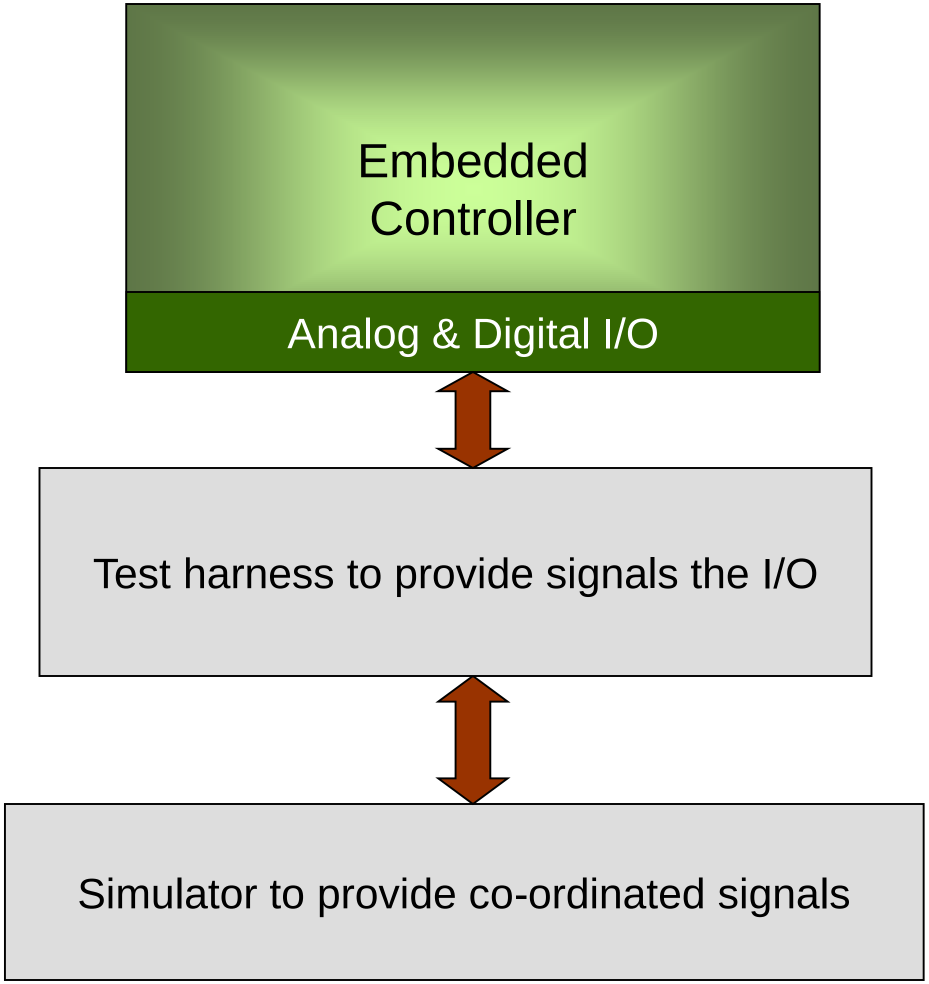
Embedded Controller

Analog & Digital I/O

Test harness to provide signals the I/O



Example 3 – Solution 2



Embedded Controller

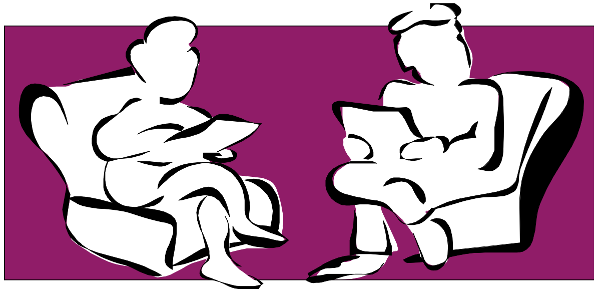
Analog & Digital I/O

Test harness to provide signals the I/O

Simulator to provide co-ordinated signals

Independent Test Group - Definition

Definition: Independent Testing – A software quality assurance function which is able to act as a customer advocate for the testing of software. This function may or may not be separate in an organizational sense; what is important is that it is able to effectively perform adequate testing without undue influence from project management. Software developers should not system-test their own software.



Source: Motorola Corporate Quality System Review Guidelines, November 1992

Close to the Customer

Watts Humphrey has said: “*System test planning should be done by a special test organization with a reasonably direct link to the end users, possibly through a users group or by close contact with selected key users.*”

Independent Testing - Considerations

a. The organization can demonstrate that independent testing of software is in place and contributing to improved quality.

b. The software testing organization participates in early life cycle reviews to assure testability of reviewed items.

c. The organization creates test plans early in software projects, in parallel with development activities.

d. Software project teams test software using customer-representative mechanisms, such as: a test lab or simulated environment.

e. The organization can show that software testing is analyzed to assure that it is customer-representative.

f. Testing effectiveness and efficiency are evaluated and tracked from multiple perspectives across the software development life cycle such as: SQA, field support, customer coverage, etc.

Source: Motorola Corporate Quality System Review Guidelines, November 1992

Independent Test Group - Evaluation

Poor: Testing is performed randomly by developers. Only system or product testing may be done independently and it is very dependent on the experience of the developers.

Weak: A few software projects have begun to address testing in a disciplined manner, with early planning and ties to anticipated customer usage. Testing is still mostly ineffective.

Fair: An independent testing approach has been defined that provides early tester participation and customer analysis, but it is only partially implemented. Measurement of test effectiveness has started.

Marginally Qualified: A well defined concurrent independent testing program has become institutionalized. Customer-representative testing and careful testing analysis assure that testing is effective at containing and preventing errors.

Outstanding: The organization has recognized software testing as a professional discipline and is a leader in the testing process and technology. Independent testing is used proactively to prevent the introduction of problems throughout the software life cycle. Innovative or world class leadership is demonstrated in this area.

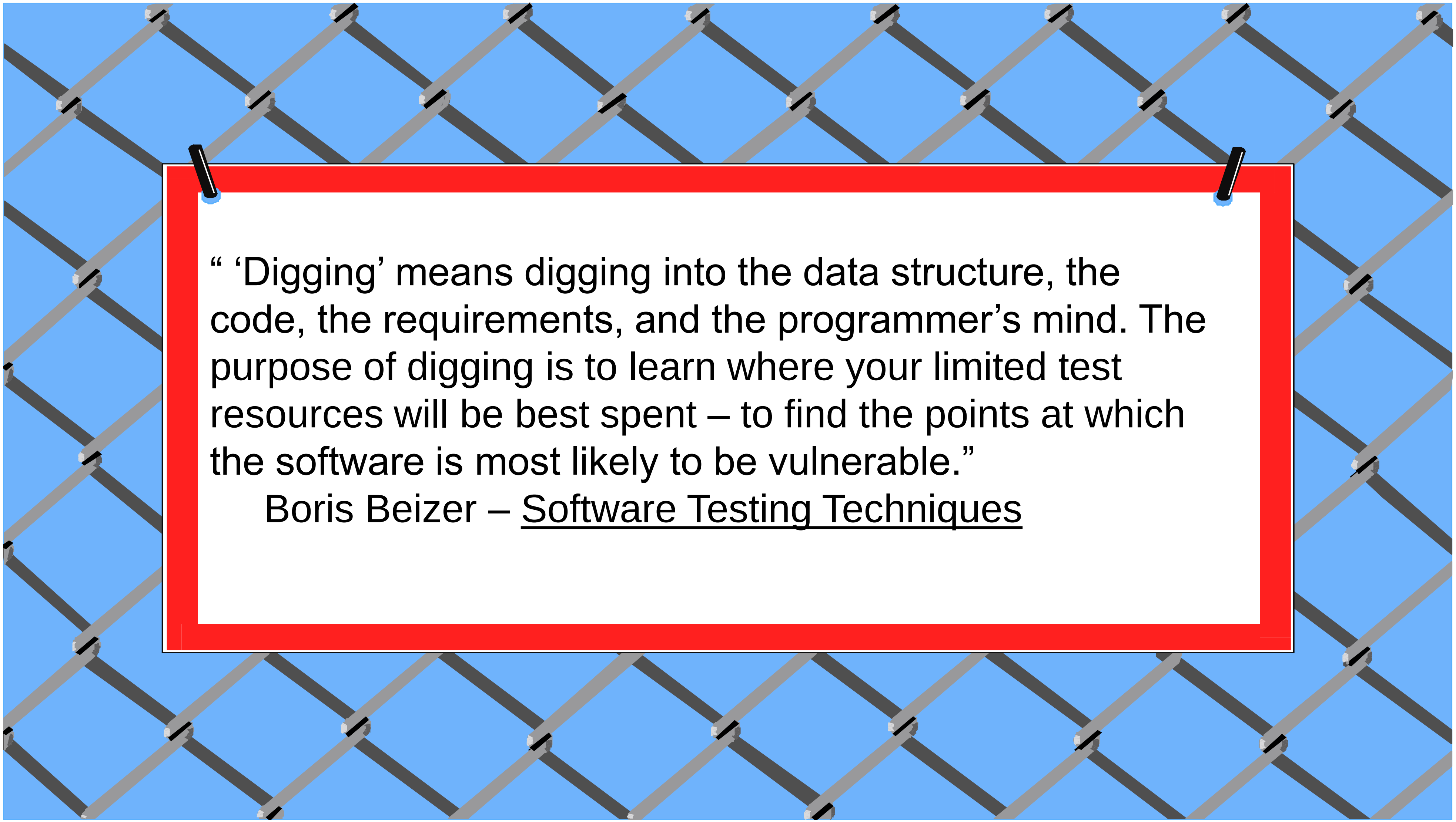
Source: Motorola Corporate Quality System Review Guidelines, November 1992

Independent Testing – Time Usage

Independent Test Group – Time Usage		
50% digging	25% test design & execution	25% Other

Source: Software Testing Techniques, Boris Beizer

Digging



" 'Digging' means digging into the data structure, the code, the requirements, and the programmer's mind. The purpose of digging is to learn where your limited test resources will be best spent – to find the points at which the software is most likely to be vulnerable."

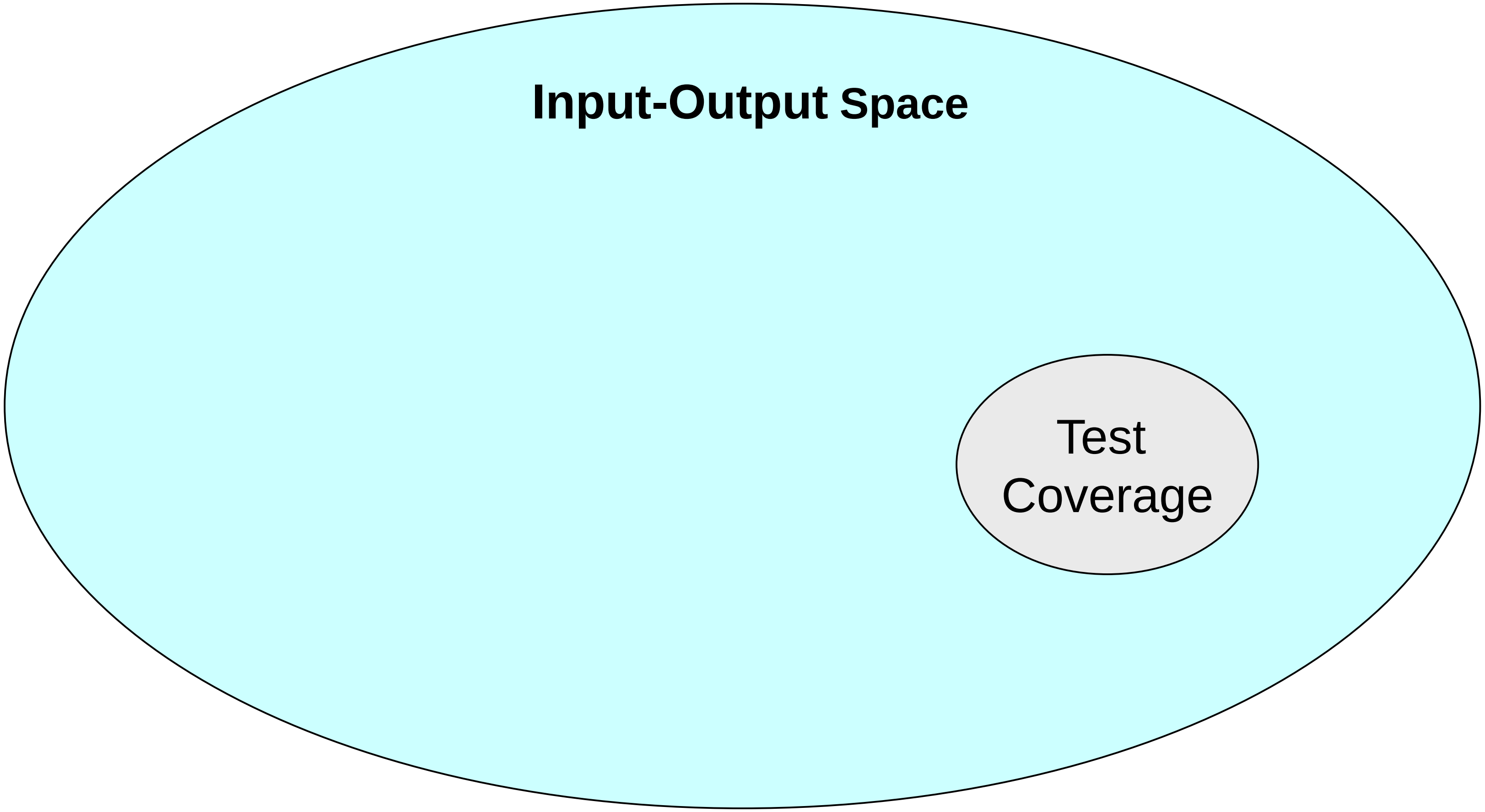
Boris Beizer – Software Testing Techniques

The Testing Dilemma

- Given
  - Finite number of requirements and behaviors
  - Infinite input and output domains
  - Infinite structure (path) possibilities
  - Infinite number of possible bugs
- To do testing with limited time, staff, and equipment, we must *sample* the problem space.
  - Which samples? → Test design methodologies
  - How many Samples → Test coverage tradeoffs.

Source: "Tutorial on Software Testing", Dr. Dwayne Knirk, Sandia National Laboratories, Jan., 1997

The Testing Dilemma



Positive and Negative Tests

- A positive, or “clean” test is based upon defined requirements.
  - Examines the basic functionality.
  - Requirements-based testing.
  - Bug verification
  - The coverage measure is requirements coverage.

- Negative, or “dirty” tests are based upon testing approaches that are likely to find bugs.
  - Stress, load. volume
  - Boundaries & interfaces
  - Anomalous conditions
  - Invalid input
  - Repetitive operations
  - Etc.

Example – Positive and Negative Tests

- Example program:
- Begin
- Read (AAAAAAAAAA)
- Print
- End

Positive Tests:

Input: AAAAAAAAAA

Input: BBBBBBBBBB

Input: CCCCCCCCCC

Negative Tests:

Input: AAAAAAAAA9

Input: 9AAAAAAAAA

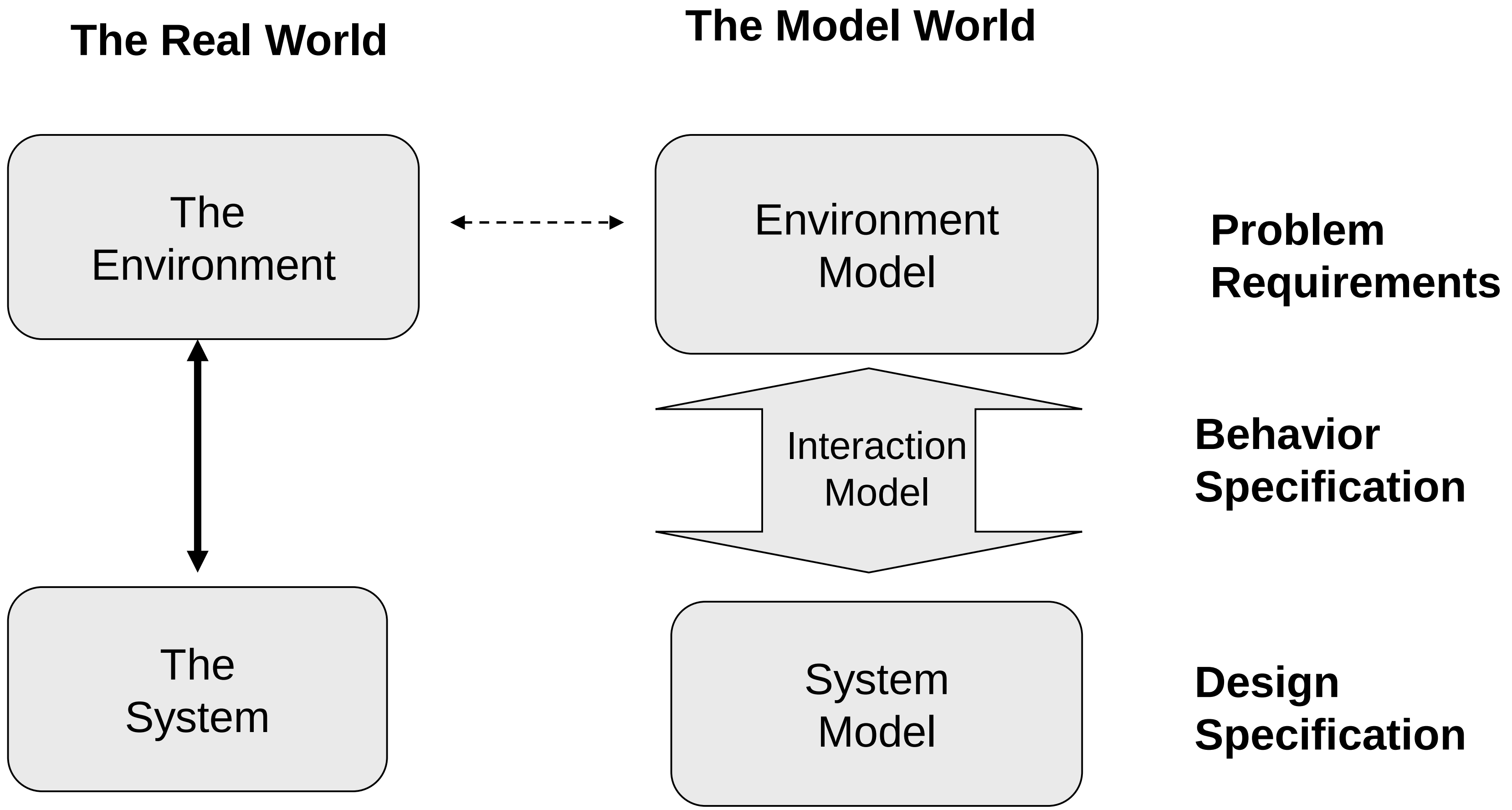
Input: 1234567890

Input: A

Input: aBcDeFgHiJ

Input: ABC   DEFGJI

Modeling The System's Behavior



Source: "Tutorial on Software Testing", Dr. Dwayne Knirk, Sandia National Laboratories, Jan., 1997

System Behavior

- **Behavior**
  - Observable activity when measurable in human terms of quantifiable effects on the environment whether arising form internal or external stimulus.
  - The peculiar reaction of a thing under given circumstances.
- **Behavior Specification**
  - Focuses on the functions required of the executing software
  - Expressed in terms of observables of software behavior.
  - Allows many possible software implementations.
  - Must be predictive to answer questions of the following sort: "What does the software do when P happens in situation Q?"

Source: "Tutorial on Software Testing", Dr. Dwayne Knirk, Sandia National Laboratories, Jan., 1997

Reason for Modeling

- By modeling the system’s behavior, we can apply structural testing techniques to the model.
- Graphs, Paths, Coverage
- Greatly expands our ability to design effective system tests, since we are not limited to requirements based testing.

Modeling Viewpoints

- Control flow
- Dataflow
- Transaction charts
- State transition models
- Decision trees
- Petri nets
- Use cases

System Testing

For the remainder of the class, we will be discussing system testing issues.

Specific Test Techniques

- Equivalence class partitioning
- Control flow testing
- Data flow testing
- Transaction testing
- Domain testing
- Loop testing
- Syntax testing
- Finite state machine testing
- Load and stress testing

Equivalence Class Partitioning

- Identify groups of system requirements, functions, behaviors.
- Select common classes of test case inputs.
- The premise is that a few test cases in each class is enough.
- It is more effective to test more classes than more test cases in the same class.

Example #1 – Sample Program

Example program:

- Begin
- Read (AAAAAAAAAA)
- Print
- End

What are the equivalence classes?

Example #1 - Solution

- Equivalence classes for “positive” tests:
  - All 10 inputs consist of the same character in upper case, repeated for each letter of the alphabet.
  - ALL 10 inputs consist of the same character in lower case, repeated for each letter of the alphabet.
  - All 10 inputs are different, mixed case.
- Test Cases:
  - TC01 - Input: AAAAAAAAAA
  - TC24 - Input: ZZZZZZZZZZ
  - TC25 - Input: aaaaaaaaaa
  - TC48 - Input: zzzzzzzzzz
  - TC49 - aBcDeFgHi
  - TC50 - IhGfEdCbA

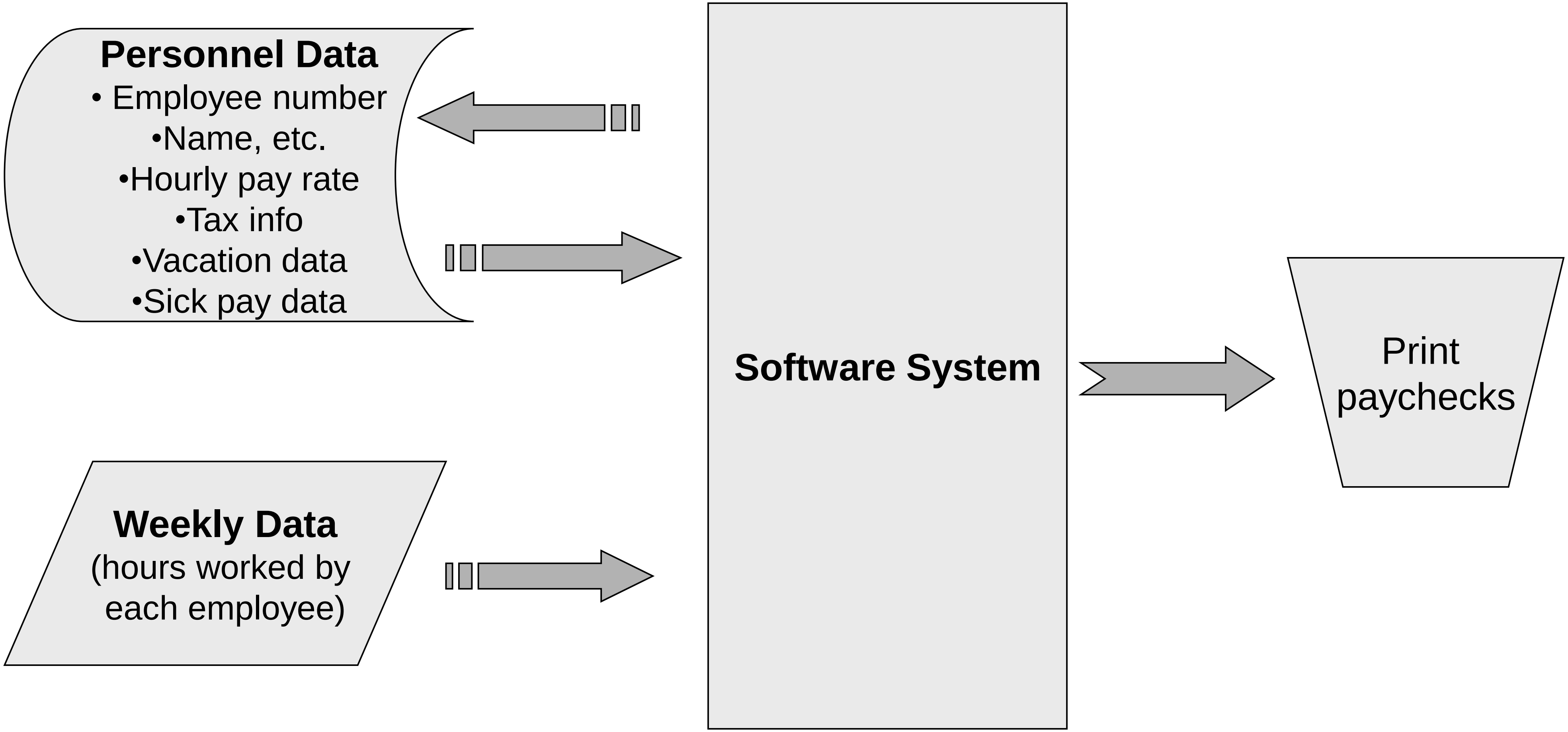
Example #1 - Solution (continued)

- Equivalence classes for “negative” tests:
  - All 10 inputs are numeric.
  - Mixed numeric and alphabetic inputs.
  - Embedded blanks
  - Input consists of one valid character.
  - Input consists of one invalid character.
  - Input includes special characters (*, & %, etc.)
  - Input consists of 11 characters.
- What would be a correct output for these cases?

Class Exercise – Payroll System

- A software package is used to calculate the weekly pay for employees and print the paychecks.
- You are assigned the job of testing this system.
- Your testing "budget" is 20 test cases.
- What are they?

Payroll System – Re-engineering



Personnel Data

- Employee number
- Name, etc.
- Hourly pay rate
- Tax info
- Vacation data
- Sick pay data

Software System

Print paychecks

Weekly Data
(hours worked by each employee)

Payroll System - Solution

- Equivalence classes – Positive tests
  - Employee works the standard number of hours.
  - Employee works more than the standard number of hours (overtime).
  - Employee has sick time.
  - Employee has vacation time.
  - The week includes a holiday.
  - Employee works less than the standard number of hours.
  - Holiday and vacation in the same week.
  - Holiday and sick time in the same week.
  - Holiday, sick time, and vacation in the same week.

Payroll System – Solution (continued)

- Equivalence classes – Negative tests
  - Employee works on Saturday.
  - Employee works on Sunday.
  - Employee works more than 24 hours in one day.
  - Employee works more than 168 hours in one week.
  - Employee works fractional hours.
  - Employee works one hour for the entire week.
  - Set up a test input file that shows the hourly pay rate to be $1000.
  - Loop tests
    - One employee.
    - Two employees.
    - Maximum number of employees -1
    - Maximum number of employees

Payroll System – Test Results

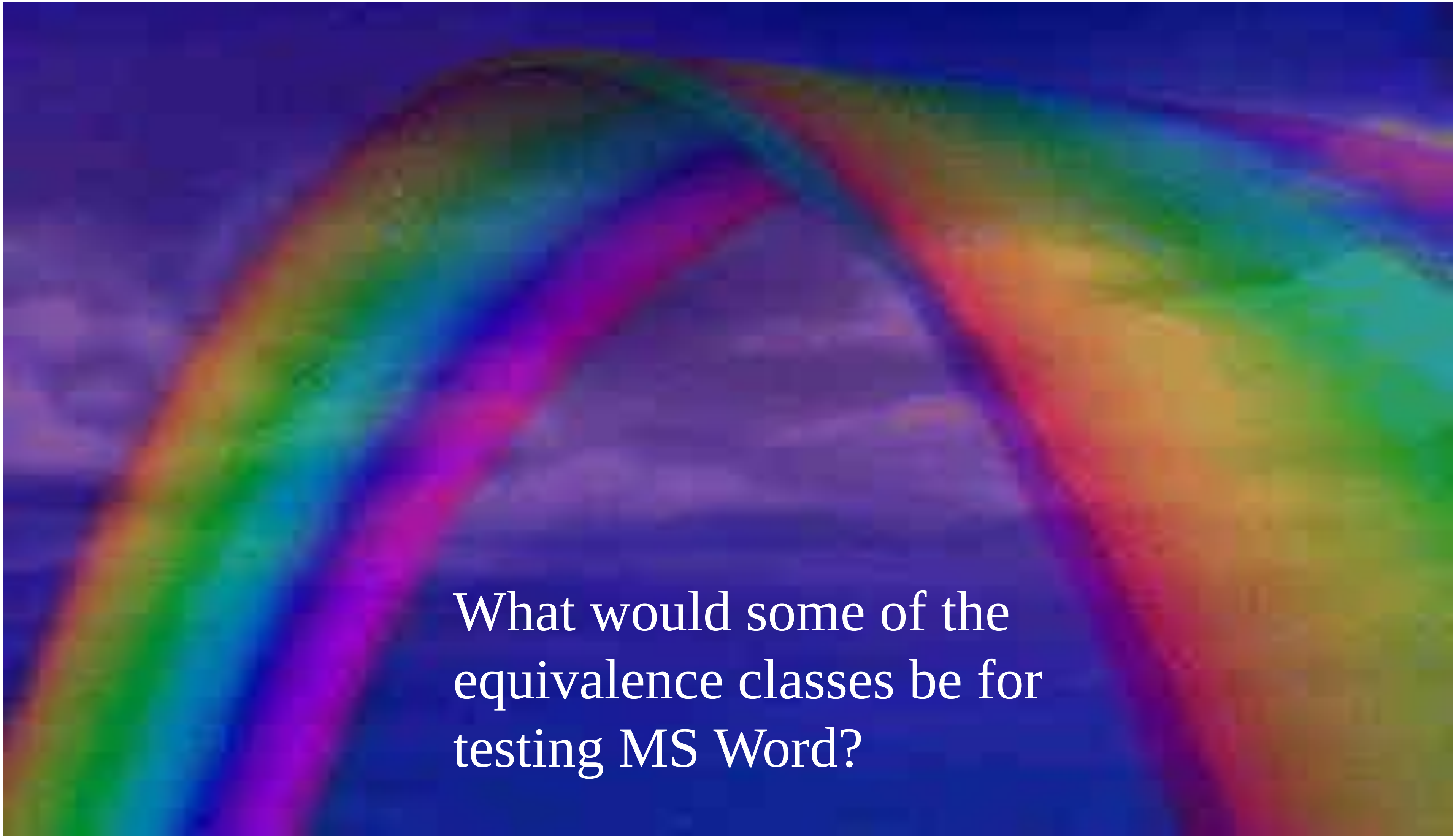
What would the expected outputs be for each employee?

Payroll System – Test Results

Expected results

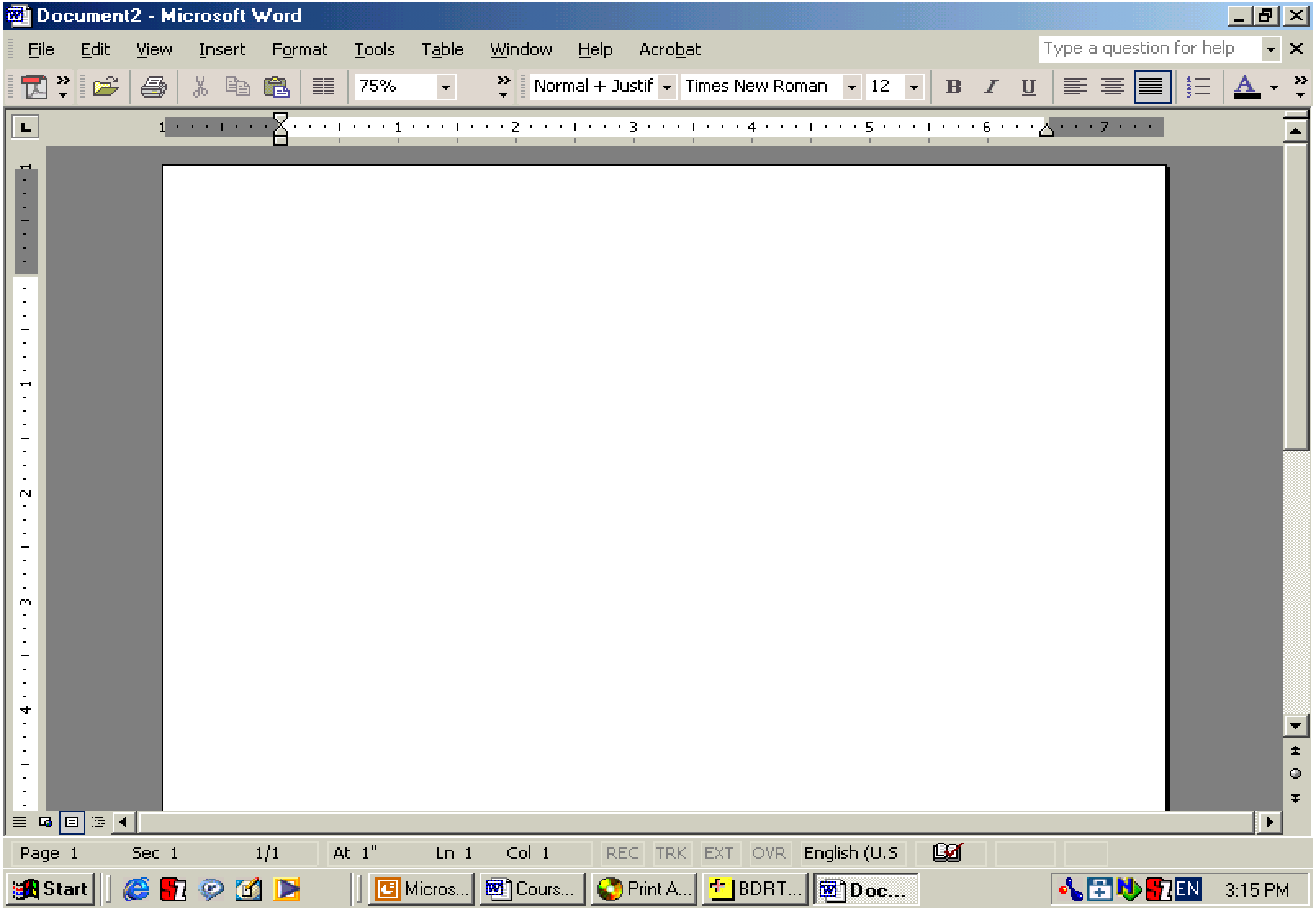
- Printed paycheck for the correct amount
- Update employee data:
  - Net pay for the week and total pay for the year.
  - Taxes withheld this week and total for year.
  - If sick time was used, update total sick time used.
  - If vacation was used, update total vacation time used.

Equivalence Class Partitioning – MS Word



What would some of the equivalence classes be for testing MS Word?

Example – MS Word



MS Word - Solution

Equivalence Classes – Positive Tests

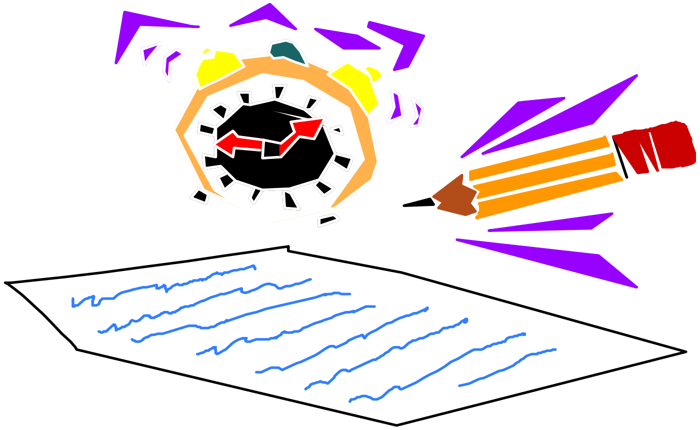
- A document consisting of pure text.
- A document that uses headings.
- A document that contains tables.
- A document that contains figures.
- Fonts.
- A document that uses columns
- Printing
- Numbered lists
- Bulleted lists
- Margins
- Indentation
- Tables
- Header & footers
- Etc.

Hierarchy of Equivalence Classes

- **Must decide on the level of granularity of the testing.**
- **Example:**
  - From the previous list, select “tables”.
  - Create equivalence classes for testing of “tables”:
    - Basic table (text only)
    - Text formatting
    - Number of rows and columns
    - Fill
    - Numbered list in a table
    - Bulleted list in a table
    - Borders
    - Add/deleted rows and columns.
    - Cut and paste into a table
    - Autoformat
    - Etc.

Issue – How Many Test Cases

- We are running into the testing dilemma.
- How many test cases do we create?

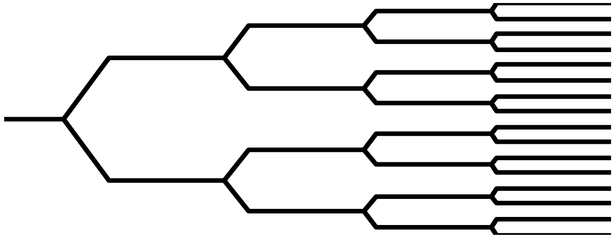


MS Word – Solution (continued)

- Equivalence classes – Negative Tests

Control Flow Testing

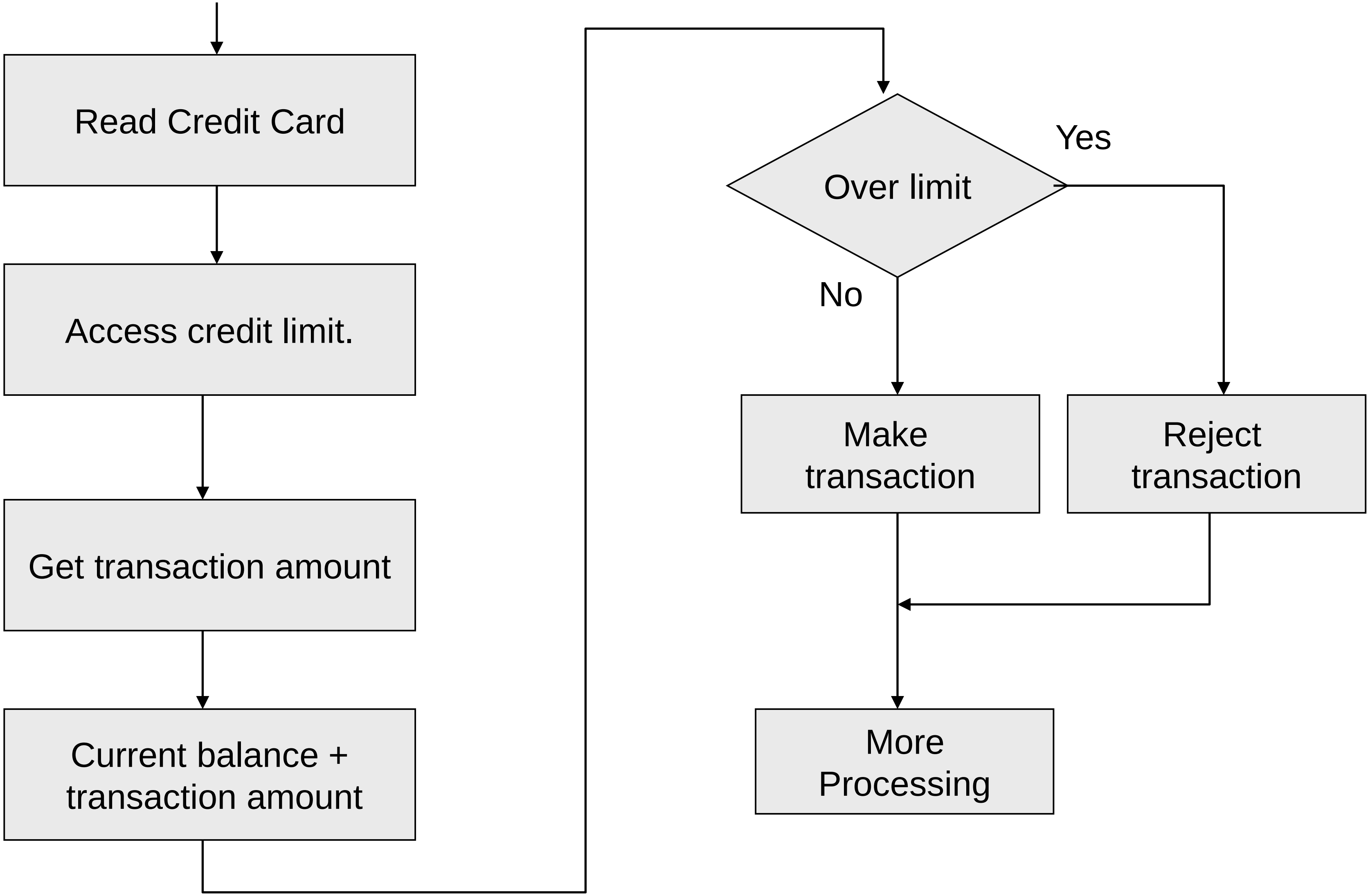
- □ Use of the program’s control flow is a fundamental testing method.
- □ A control flow graph is a basic model for testing.
- □ Applies to almost all software and is effective for most
- □ Applicable mostly to relatively small programs or segments of larger programs.
- □ Bugs are likely to be found in the control flow aspects of a program.
- □ Create a model of the control flow of the software.
- □ Flow graph



Flow Graph vs. Flow Chart

- Very similar in concept, but obtained in a much different manner.
- The flow graph does not need to show all of the processing details.
- No matter how many statements, a processing block will be shown as one block. In a flow chart, all steps will be shown.
- A flow graph is simpler than a flow chart.
- A flow graph focuses on decisions.
- For our purpose, flow graphs will be done on models of the software’s behavior, not on the actual code.

Sample Behavior Model



Read Credit Card

Access credit limit.

Get transaction amount

Current balance + transaction amount

Over limit

Yes

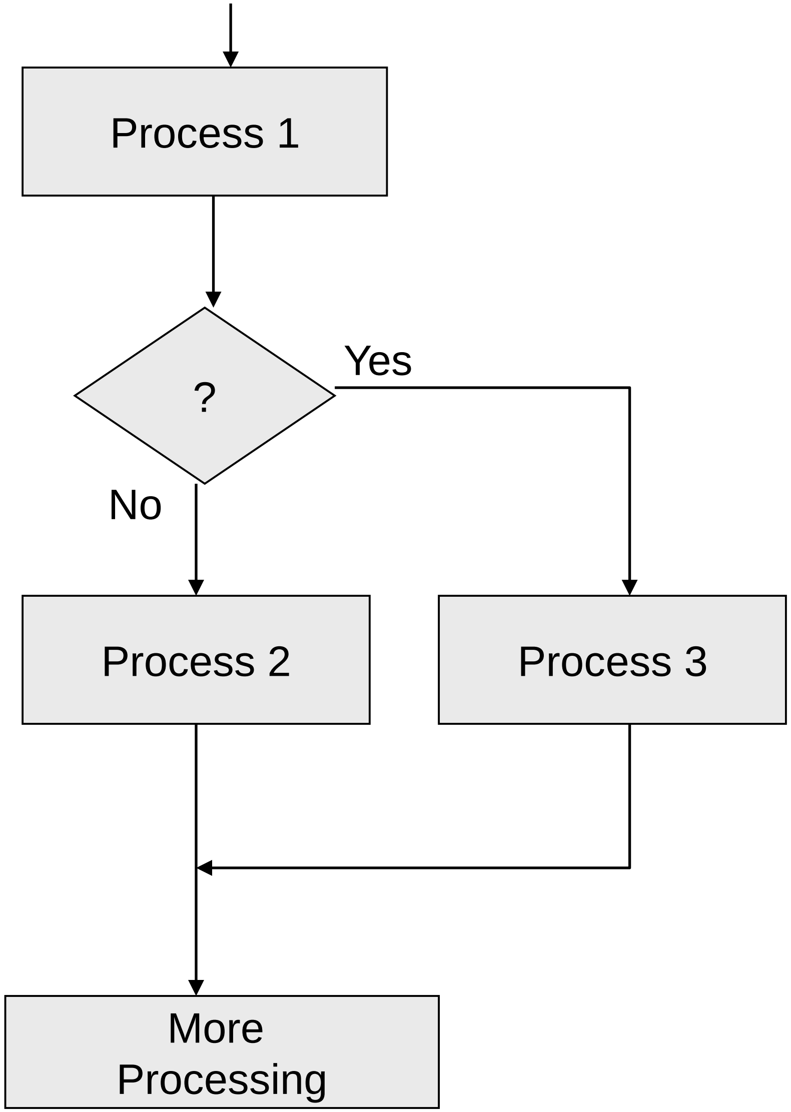
No

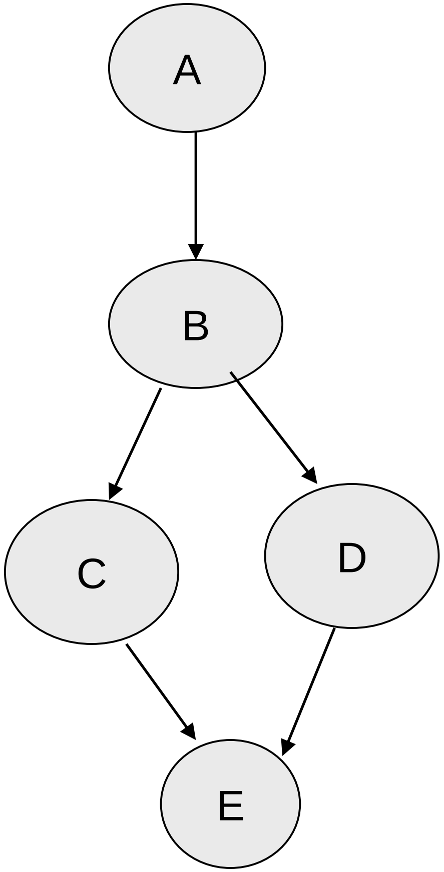
Make transaction

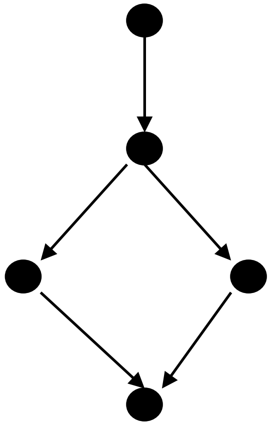
Reject transaction

More Processing

Flow Graph - Example





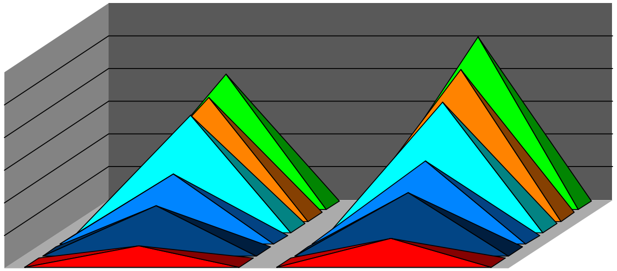


Divide and Conquer

- Model parts of the systems
- Sub-systems
- Specific functions
- Sequence of events.
- Customer usage scenarios.
- If a specification is available, use it.

This is a Behavior Model

- We are modeling the *behavior* of the system.
- It is not a structure chart of the code.
- There may or may not be corresponding paths in the code.
- It doesn't matter for testing purposes.



Modeling Behavior Using A Spec

- Rewrite the specification as a sequence of short sentences.
- Pay special attention to decisions.
- Number the sentences sequentially.
- Build the model.

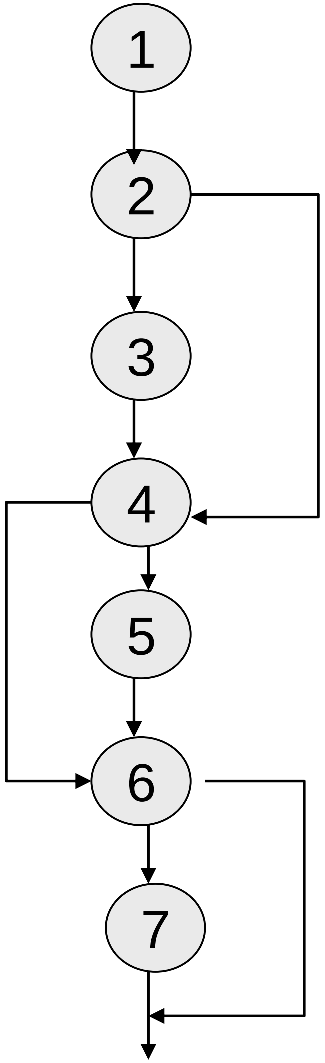
Source: Black-Box Testing, Boris Beizer

Example – Specification Excerpt

Adding the date or time From the Insert menu, choose Date And Time, and then select a format for the date and time. If you want to be able to update the date or time, select the “Insert As” check box, and then choose the OK button. To update the date or time, click in it and then press F9. You can also update the date or time each time you print. To do this, choose “Options” from the “Tools” menu, select the Print” tab, and then select the “Update Fields” check box.

Rewrite & Model

1*. Insert,* D&T, format.
2. Want D&T to be updateable?
3. Check “Insert As” box. OK.
4. Want to update D&T in doc?
5. F9.
6. Want to update D&T at print time?
7. Tools, Options, Print Tab, Update Fields.

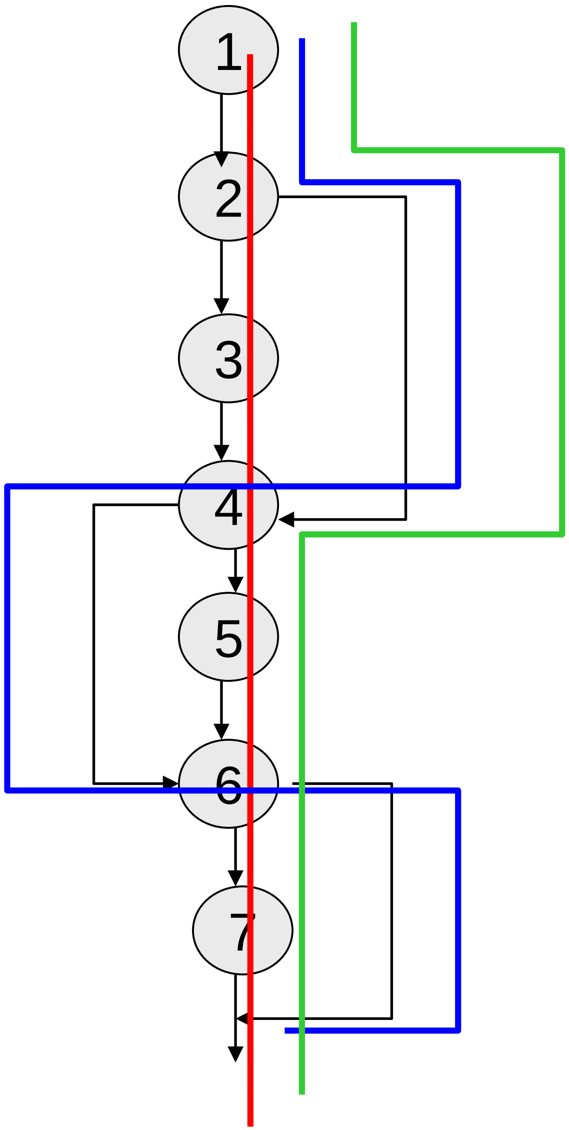


Select Paths for Test Cases

🟩 = Test Case 1

🟦 = Test Case 2

🟥 = Test Case 1



A More Rigorous Approach

- Can give weights to links.
- Add up the weights in a path.
- The higher the total, the more critical the path.

Example - What about Negative Tests?

- For example: Don’t make date & time fields updateable, then try to update them.
- Update field multiple times.
- Others?

Control Flow Testing Strategy - Summary

- Model the system or sub-system to be tested.
- Identify the objects.
- Identify the relationships
- Identify the weights.
- Identify paths through the model to cover objects.
- Identify paths through the model to cover links
- Each path is a test case.
- Specify input conditions and expected results for each test case.

Data Flow Testing

- All software requires some data in order to operate (to varying degrees).
- Control flow concepts do not pay attention to the data aspects of a system.
- For a software system that is data-intensive:

  We need to look at the data that is input to the system and that it produces.

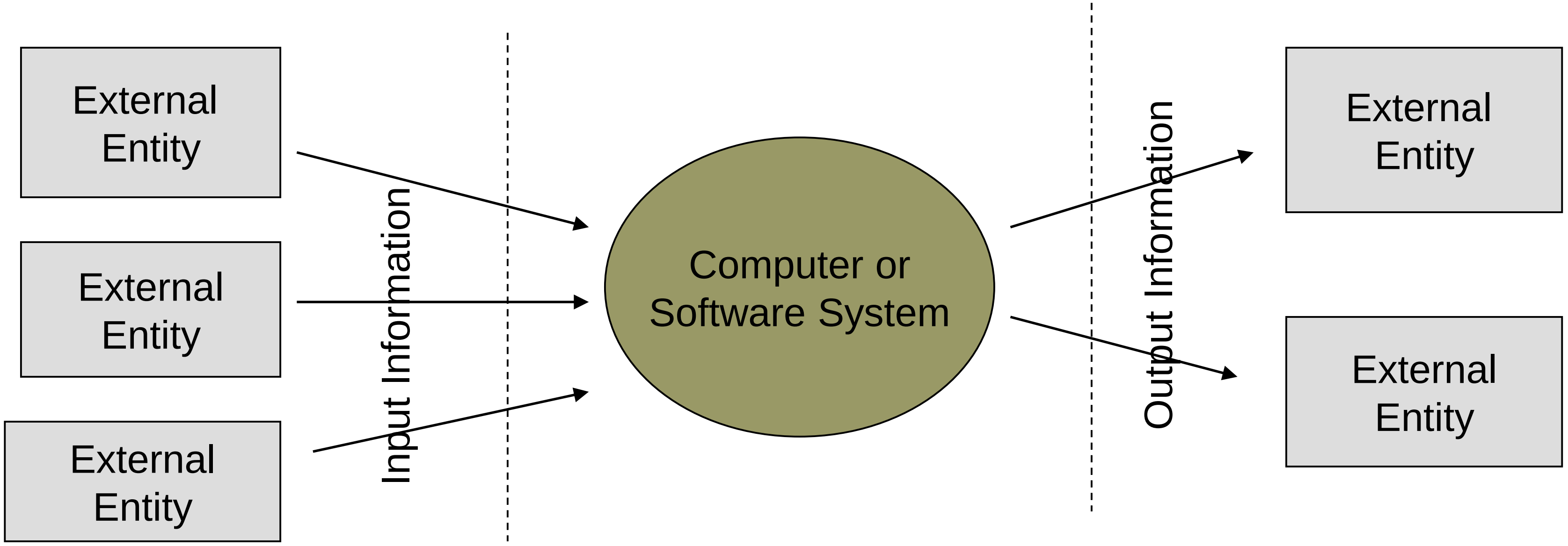
Structured Analysis

- Popularized by Tom DeMarco in the 1970’s.
- IT is another modeling methodology.
- Focuses on the information transformation of a system.
- Looks at the *flow* of data through the system, and the various forms it takes along the way.
- Combines both control flow and data aspects of a system into one model.
- As a design tool, there are problem using this methodology for real time systems. Ward and Mellor attempted to address this with real time “extensions” to the structured analysis methodology.
- We are not designing a system, but testing it, so we do not need to be concerned about the real time design limitations of the data flow paradigm.

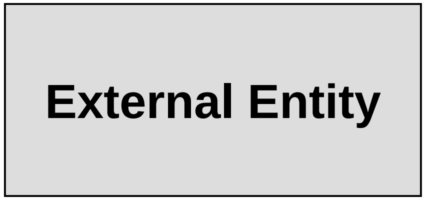
References: Structured Analysis and System Specification, T. DeMarco; Structured Development for Real Time Systems, Ward & Mellor

Basic Tool – Data Flow Diagram.

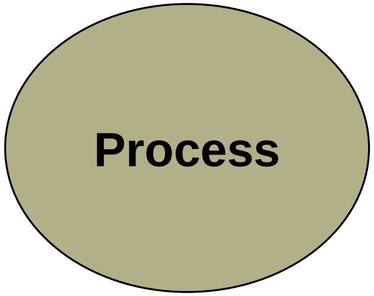
- Depicts information (data) flow and the transformations that are applied as data moves through a system from *input* to *output*.
- Information flow and content modeling.



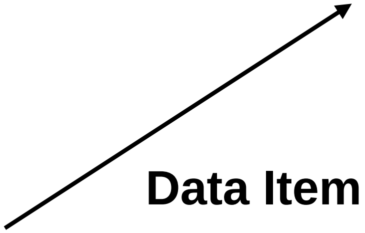
DFD Notation


External Entity

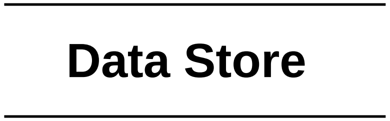
A producer or user of information to/from the System. Not included in the system being modeled.


Process

A transformation of information. It is within the system being modeled.

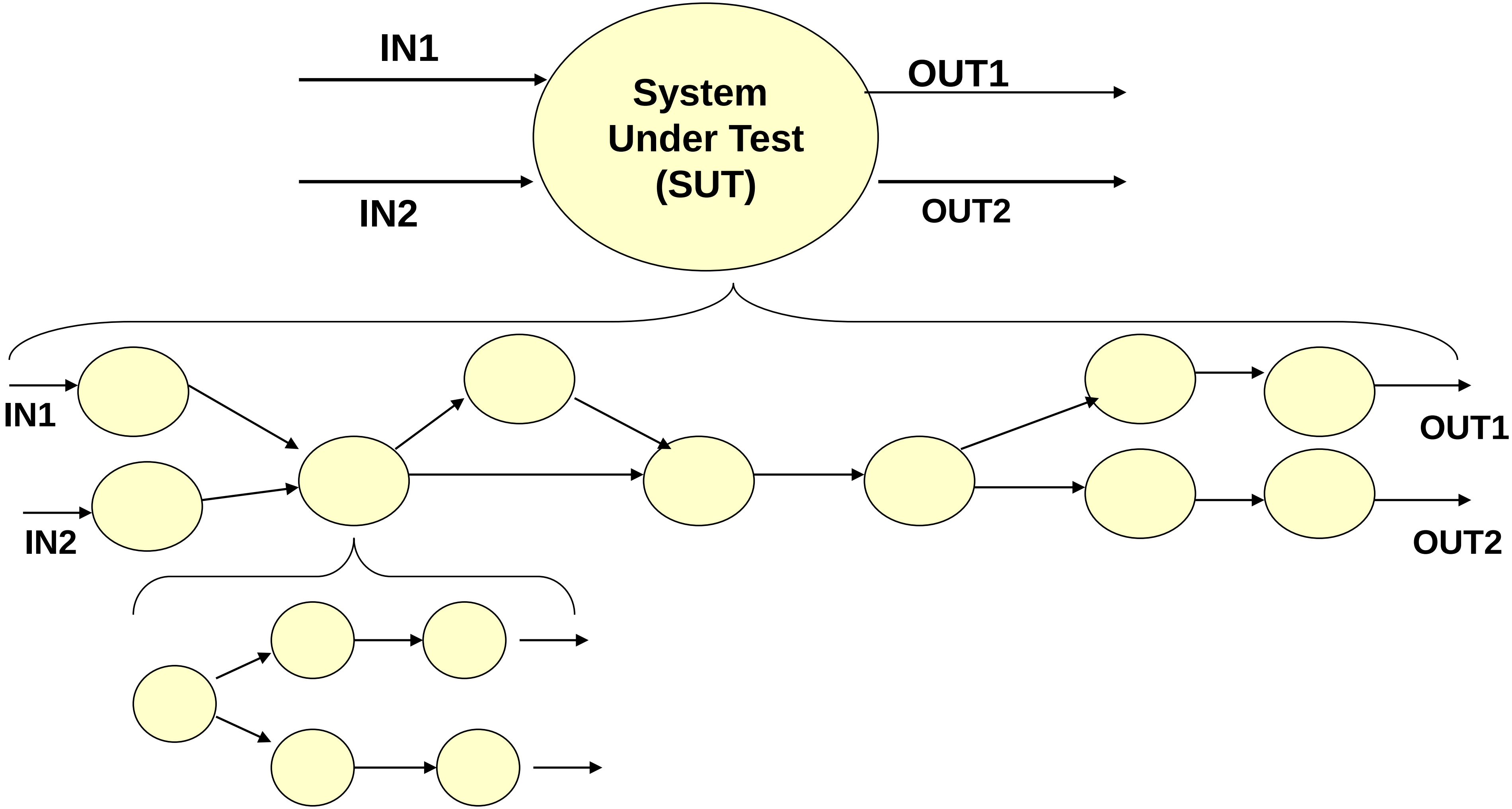

Data Item

A data entity flowing in the direction of the arrow.


Data Store

A data repository. Examples: Buffers, queues, flags, tables, RDBMS

DFD Layers of Abstraction



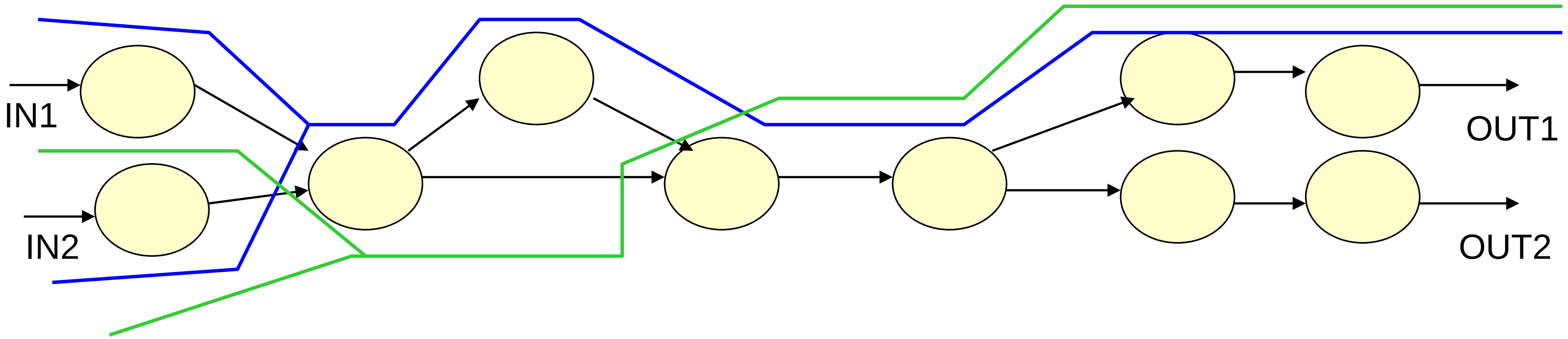
Data Flow Test Method

- Path ("slice") selection through the DFD.
- Start at an output node.
- Trace backward from the output node to all nodes that connect it. Keep going until you get to input nodes.
- This is a test case.

Source: Black Box Testing, Boris Beizer

Data Flow Testing – Example 1

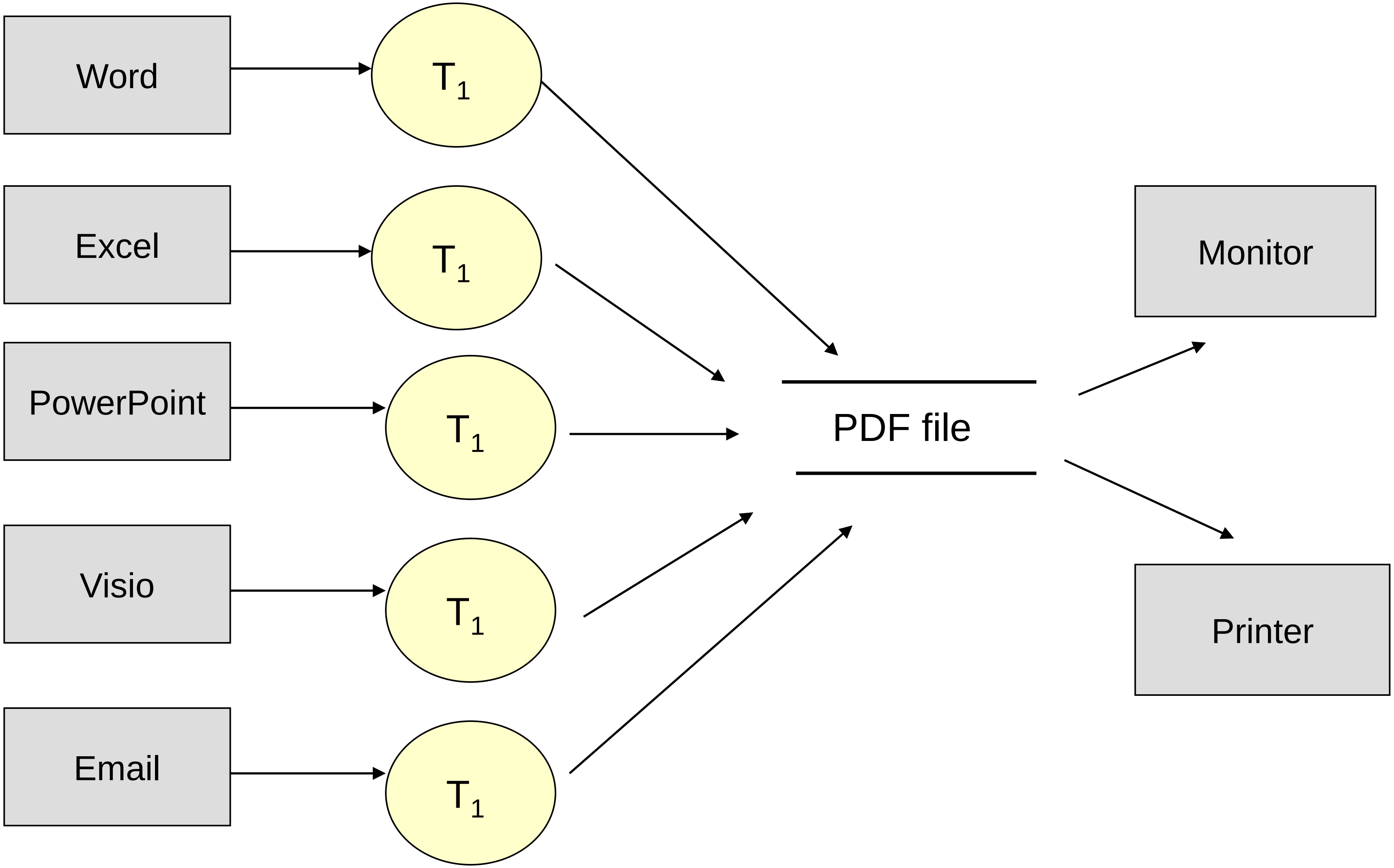
Previous DFD Example



🟦 = Test Case 1

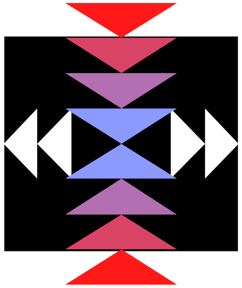
🟩 = Test Case 2

Data Flow Testing – Example 2



Data Flow Testing Example 2 (continued)

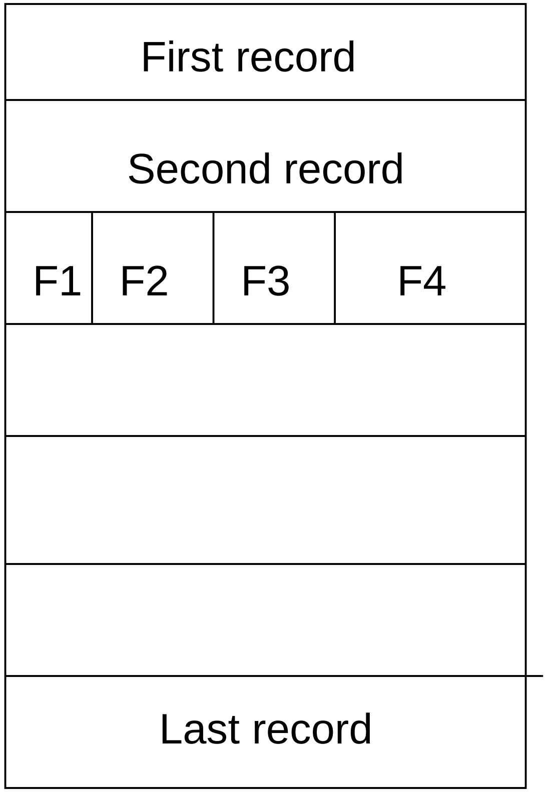
- Levels of abstraction
- Consider Word.
- Would want to do some “equivalence class partitioning” to identify different Word files to test:
  - .doc
    - Embedded tables
    - Headers and footers
    - Document sections
    - Embedded graphics
    - Etc.
  - .txt
  - .rtf



Data Flow Testing – Example 3

- In a RDBMS, consider:
- Table level
- Record level
- Field level

Table xyz



First record

Second record

F1

F2

F3

F4

Last record

Table Level

- Creating Tables.
- Deleting tables
- Copying tables
- Relations
- Printing

Record Level

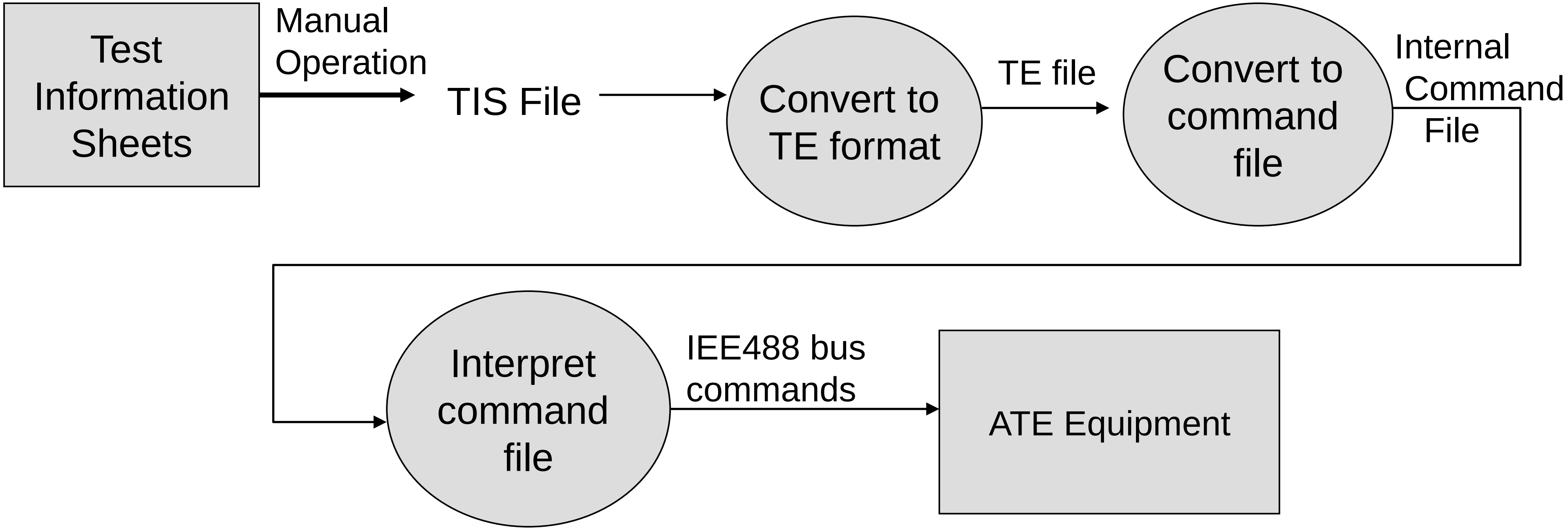
- Adding, deleting, modifying records.
- Reading records.

Field Level

- Modifying the contents of a field in a record.
- Reading fields.

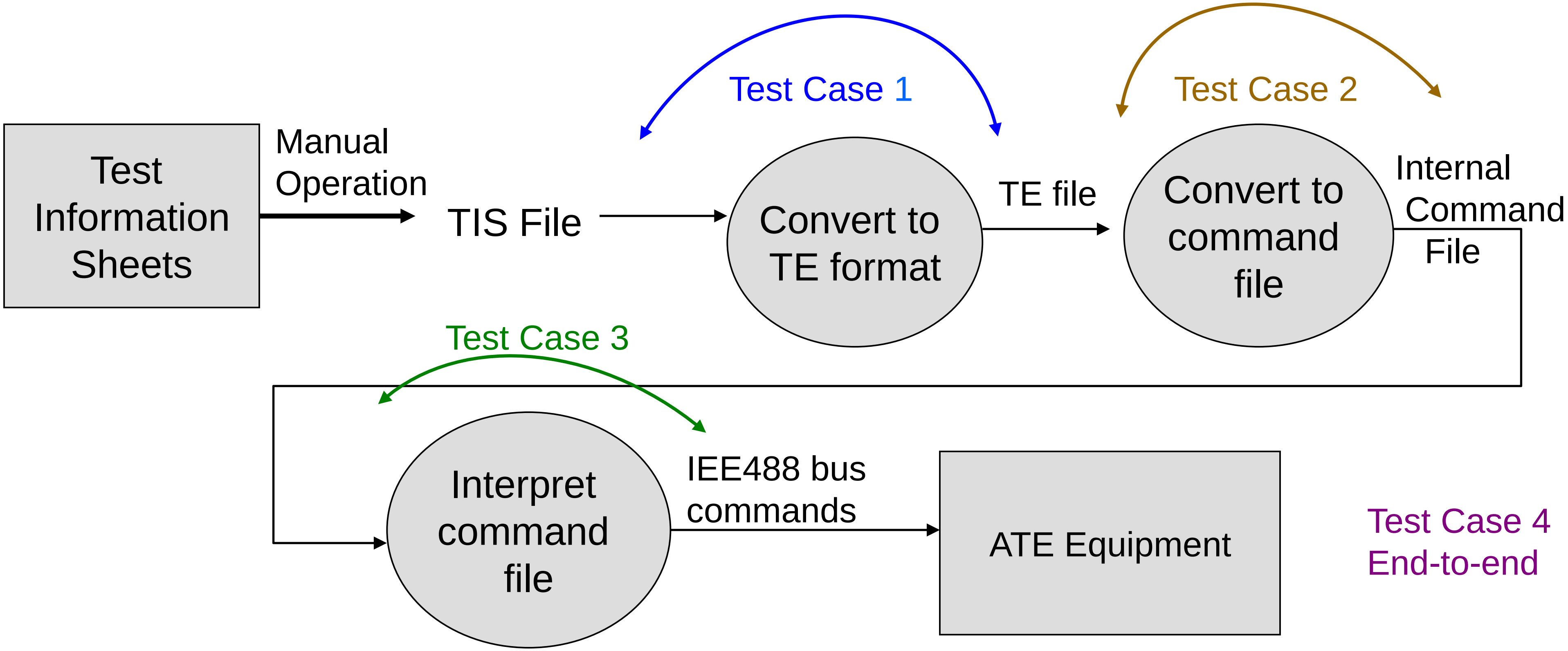
Data Flow Testing – Example 4

Automatic Test Equipment (ATE) Software



Data Flow Testing – Example 4 (continued)

- In this case, we may want to do more than just an “end-to-end” test.
- Test each data conversion.



Transaction Testing

- Another modeling technique
- For systems that handle “transactions”.
- Definition – Transaction: An instance of buying or selling something
- In computer systems, a “transaction” is an input message that must be dealt with as a single unit of work.
- Similar to data flow, but includes the concepts of:
  - Birth (of a transaction) – How does it get initiated?
  - Death (of a transaction) – When is it complete?
  - Queues – A number of similar transactions waiting to be processed.

Types of Transactions

- For what types of systems would we use the transaction testing approach?
  - Airline reservation system
    - Make a reservation
    - Cancel an existing reservation
    - Confirm a reservation
  - Banking
    - Deposit
    - Withdrawal
    - Open account
    - Change account data
  - Hotel reservation system
    - Make a reservation
    - Cancel an existing reservation
    - Confirm a reservation

Levels of Abstraction

- Must decide at what level the modeling and testing is to be done:
  - Complete end-to-end
  - Sub-components of the transaction processing.
- Usually done end-to-end for the entire transaction

Transaction Flow Testing Strategy

- Identify all transaction types.
- Identify origin and exit points for each transaction type.
- Identify queues (places where transactions may wait to be processed.
- Identify processing components (these do not necessarily correspond to software components).
- Construct the model.
- Identify paths.
- Identify input and output conditions that will cause these paths to be traversed.

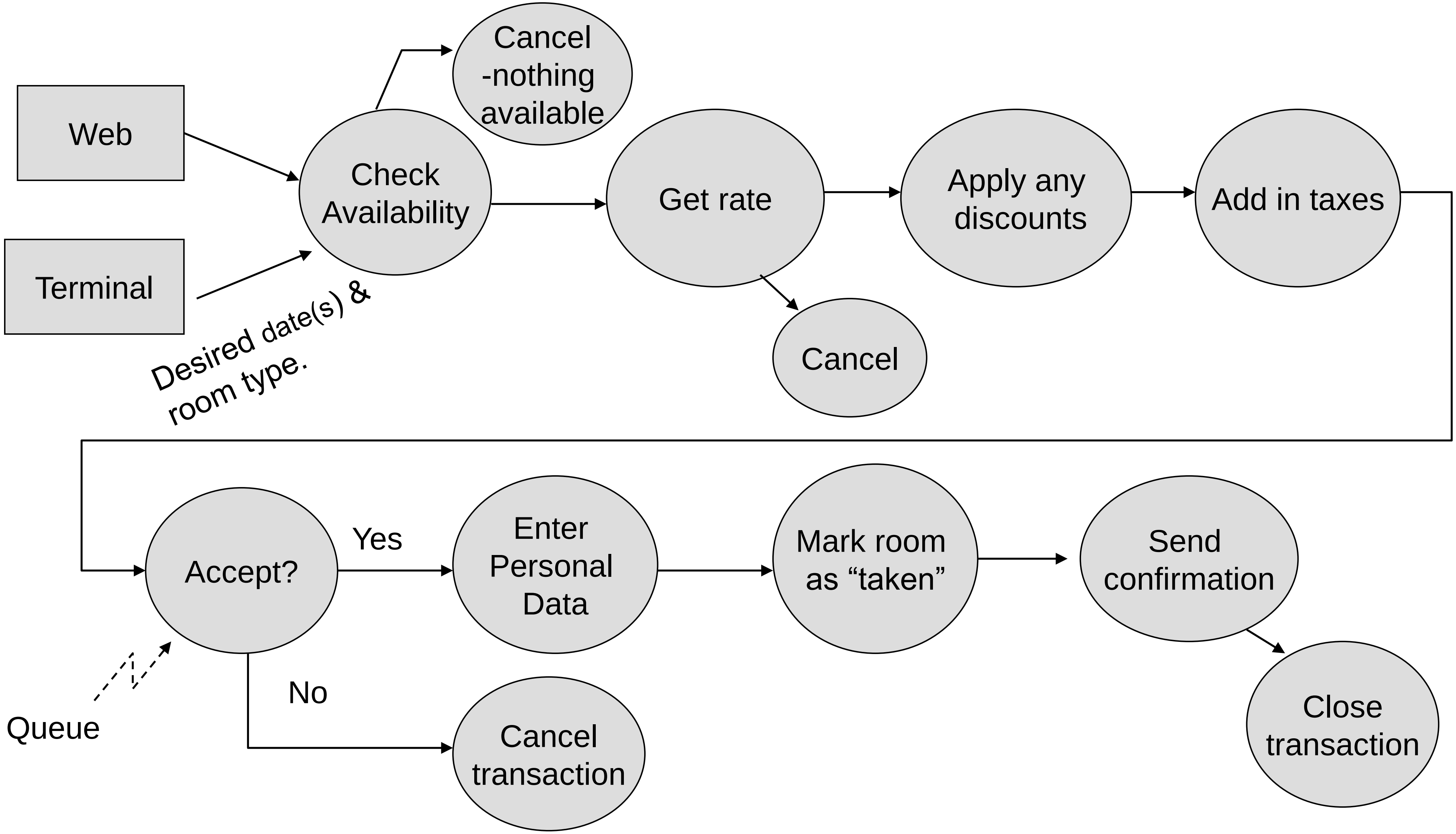
Source: Black Box Testing, Boris Beizer

Example – Hotel Reservation System

- Transaction types:
- Make a reservation.
- Cancel an existing reservation.
- Confirm a reservation.

- There is a queue for each one of these.

Example – Make A Reservation



Transaction Testing Methodology

- Origin/Exit Coverage
  - For each transaction type, test each combination of origin and exit.
- Could also apply “slicing”.
- Add “queue” tests.
- Add synchronization tests.

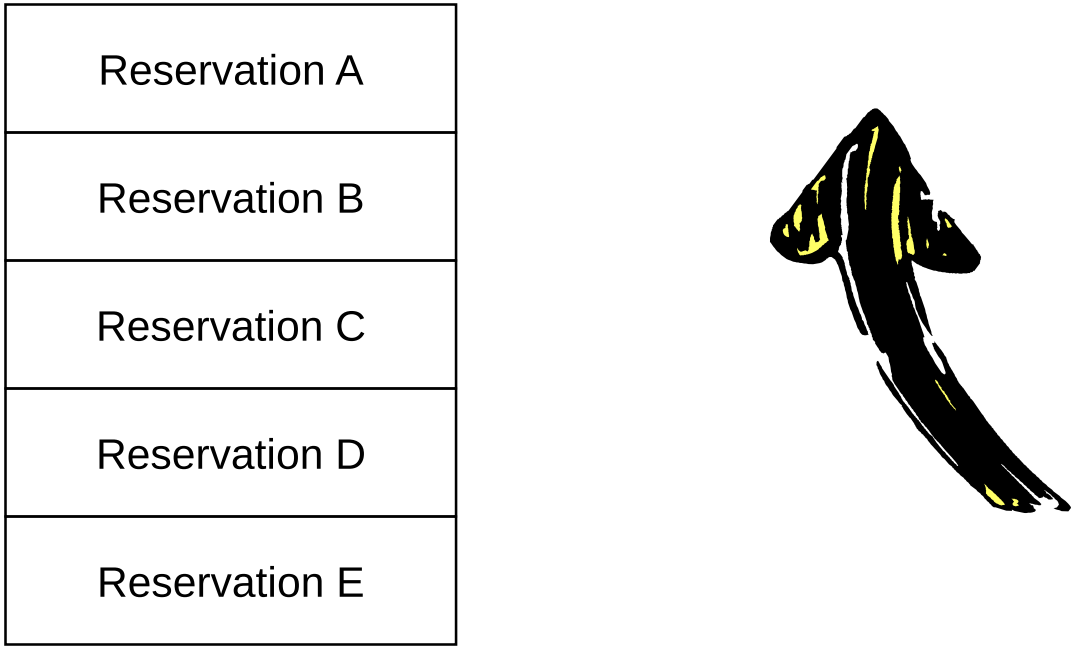
Source: Black Box Testing, Boris Beizer

Queue Tests

- Queue capacity tests.
  - Minimum
  - Maximum
- Selecting items from the queue:
  - FIFO (first-in-last0out)
  - LIFO (last-in-first-out)
  - Oldest
  - Random
  - Some assigned priority

Reservation Queue

Reservations waiting to be accepted.



Reservation A

Reservation B

Reservation C

Reservation D

Reservation E

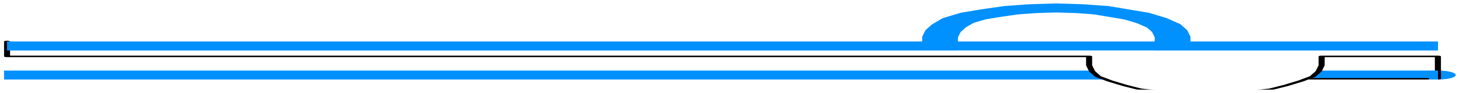
Synchronization Tests

- When transactions merge, need to do synchronizations test.
- Which one gets there first?
- Example - Banking system

* Check balance is a transaction.

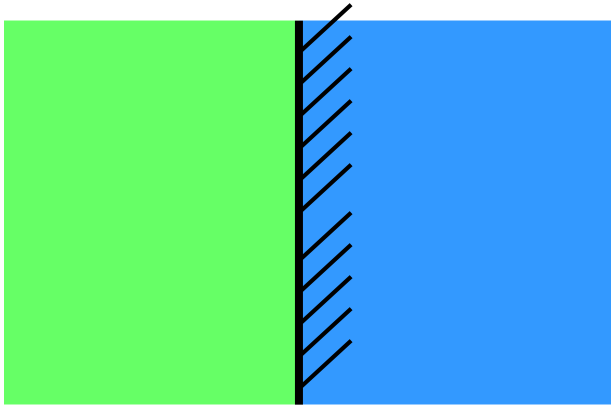
* Withdraw is a transaction.

What if I am withdrawing at the same time that another system is checking the balance for a credit card transaction for something I ordered by mail?



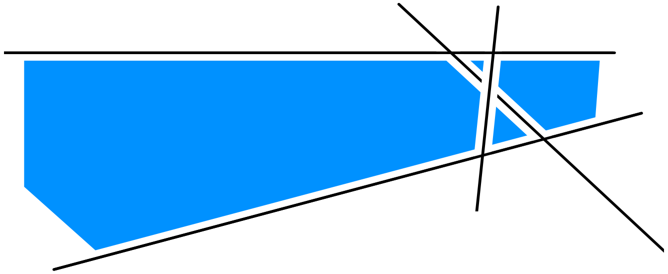
Domain Testing

- Applicable to software that deals with ranges of values of variables.
- Bugs cluster around boundaries, so they deserve special attention in testing.
- Domain testing examines the boundaries of specified ranges of values.
- Look for places where the software does different processing based upon the value of some variable.



There Are Boundaries Everywhere

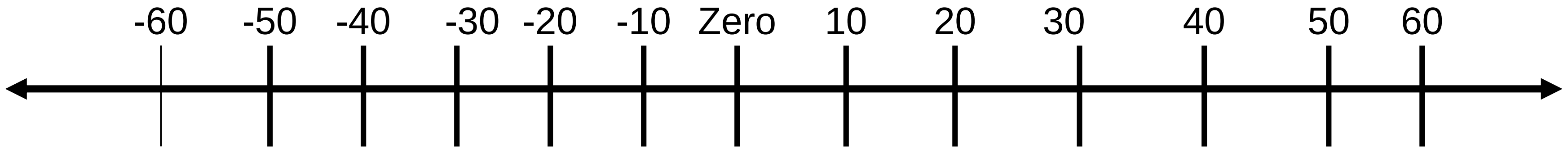
- Alarm conditions
- Flags
- Decisions
- Validity checks
- Error processing
- Case statements



Example 1

Software To Be Tested

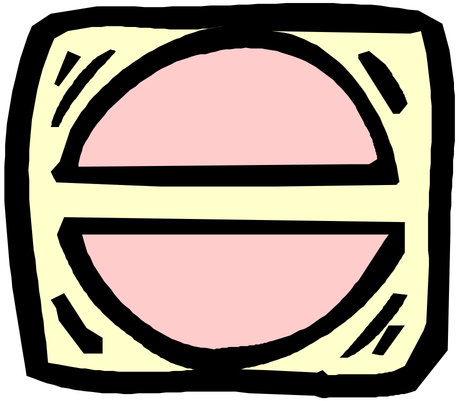
If x > 0 then perform processing;



How many test cases are needed?

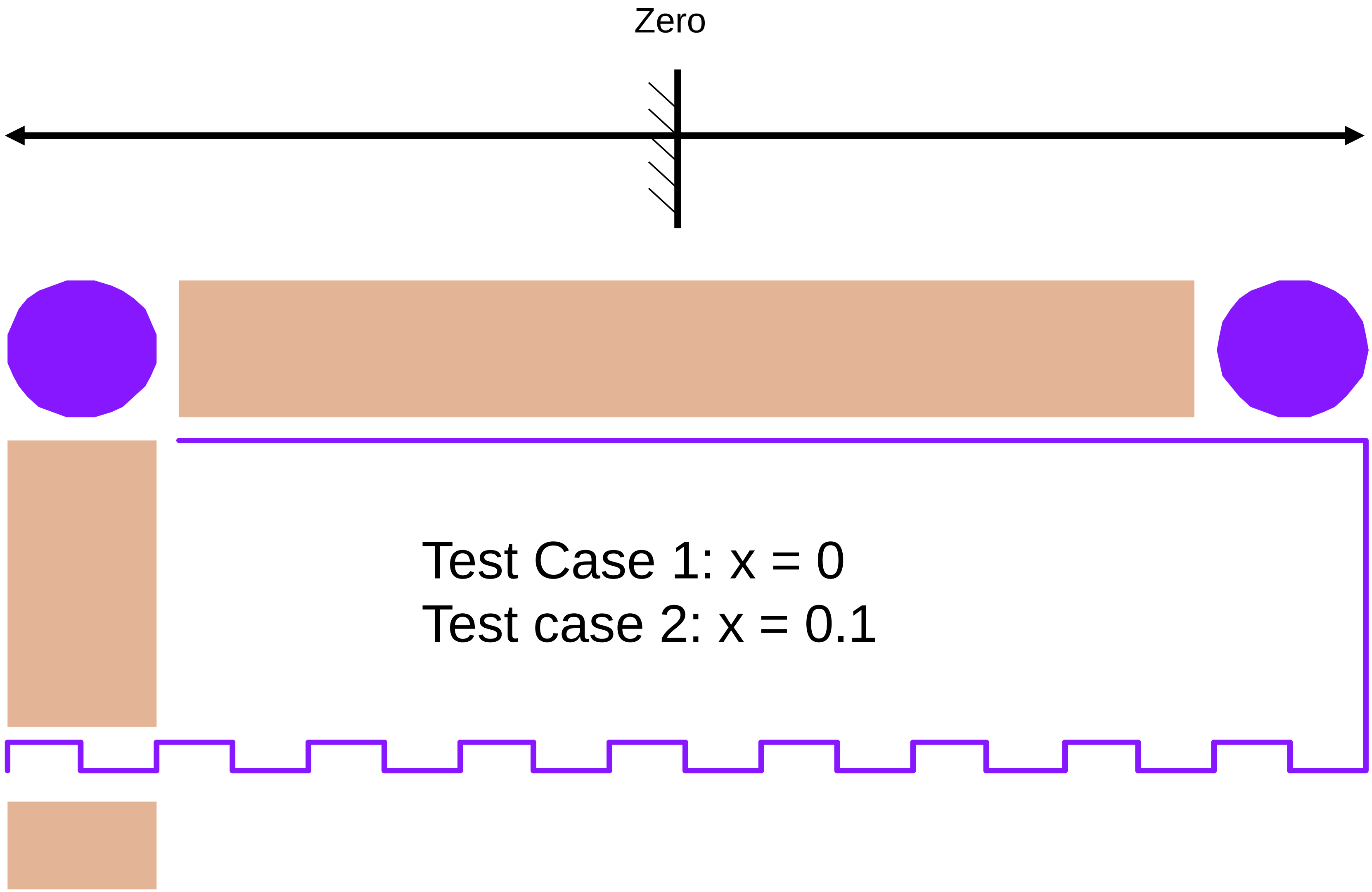
Testing Technique

- Look at each boundary value independently.
- Test for one point on the boundary and one point off.
- Choose as the "off point" a value close to the boundary value



Source: Black Box Testing, Boris Beizer

Example 1 - Solution



Test Case 1: x = 0
Test case 2: x = 0.1

Example 2

Specification for a temperature monitoring system:

If *temp* < 100, OK;

If *temp* >= 100 and < 110, yellow alarm.

If *temp* >= 110 and < 125, red alarm.

If *temp* >= 125, shut down.

Example 2 - continued

Specification for a temperature monitoring system:
If *temp* < 100, OK;
If *temp* >= 100 and < 110, yellow alarm.
If *temp* >= 110 and < 125, red alarm.
If *temp* >= 125, shut down.

How many test cases are needed?

Example 2 - continued

First, identify the boundaries:

1) *temp* < 100
2) *temp* >= 100
3) *temp* < 110
4) *temp* >= 110
5) *temp* < 125
6) *temp* >= 125

Example 2 - Solution

	Boundaries	Test Cases	
1.	*temp* < 100	On = 100	Off = 100.01
2.	*temp* >= 100	~~On = 100~~	~~Off = 100.01~~
3.	*temp* < 110	On = 110	Off = 110.01
4.	*temp* >= 110	~~On = 110~~	~~Off = 110.1~~
5.	*temp* < 125	On = 125	Off = 125.01
6.	*temp* >= 125	~~On = 125~~	~~Off = 125.01~~

Example 2 – Final Comments

Would also want to test:

1. ***temp*** = 0
2. ***temp*** = negative value

Domain Testing – Final Comments

- Very effective.
- Look for boundary values and transitions.
- Test them.
- In addition to the domain test case, do negative tests.

Loop Testing

- We talked about loop testing in structural, or white box, testing.
- The concepts are similar for black box testing, except that here we are talking about loops through a model of the software’s behavior, rather than loops through the code.

Looping Behavior – Examples

- Payroll System - Processing all of the employees.
- Searching a file for a record that meets predefined criteria.
- Sending an email to an email group.
- Outlook Express downloading emails from an ISP server.
- Semiconductor manufacturing – Control system that processes “jobs”.
- Semiconductor wafer fabrication: “Recipe” execution (step 1, step 2, step 3, --- step n).
- Mail merge.
- Printing multiple copies of a document.

Loops – Test Cases To Use

- ZZero times through the loop.
- OOnce through.
- TTwice through.
- TTypical number of time through.
- (Maximum – 1) number of times through.
- MMaximum number of times through.
- (Maximum + 1) number of times through.

Sources: Black Box Testing, Boris Beizer; Software Engineering, Roger Pressman

Example 1 – Payroll System

The system is to process weekly data on hours worked, and produce a paycheck for every employee. The system is specified to be able to handle up to 10,000 employees.

Test cases:

TC1 – Zero employees.
TC2 - 1 employee.
TC3 – 2 employees.
TC4 – 200 employees.
TC5 – 9999 employees
TC6 – 10,000 employees
TC7 – 10,001 employees.

Note: this is not all of the test case you would want to perform on this system. See also “equivalence class partitioning”.

Example 2 – MS Word – Mail Merge

- Explanation of how it works.
- Behavioral loop – Processing address data records from a data source .
- The issue here is that the maximum number of allowable times through the loop is not known. What to do?

Mail Merge – Loop Testing

Test case input data is contained in separate Excel files.

Mail Merge – Test Results

Test Case	Test Results
TC1: Zero input records	No bugs found.
TC2: One input record	No bugs found.
TC3: Two input records	No bugs found.
TC4: 100 input records	No bugs found.
TC5: 10,000 input records	Erratic behavior starting at record 5990. There are definitely bugs
TC6: 48,000 input records.	Accepts the data, but crashes the system.

Syntax Testing

- Very useful for testing:
  - Commands
  - Operator entry fields that must be in a certain format.
- Examples of syntax:
  - Dates
  - Email addresses
  - Telephone numbers
  - Mailing addresses

Syntax Testing - Technique

1. Analyze and understand the syntax definition.
   - □ This is usually difficult, since it is not written anywhere.
   - □ Will have to figure it out.
2. Design positive test cases using equivalence class partitioning.
3. Design negative tests.
   - □ Make one parameter wrong at a time.
4. Run the tests, etc.

Syntax Testing - Example

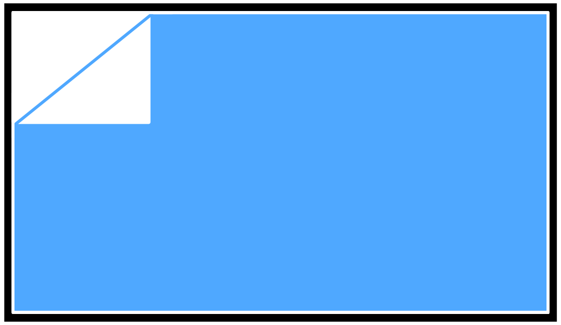
- MS Excel has a rich syntax for spreadsheet functions.
- Example: COUNTIF( )

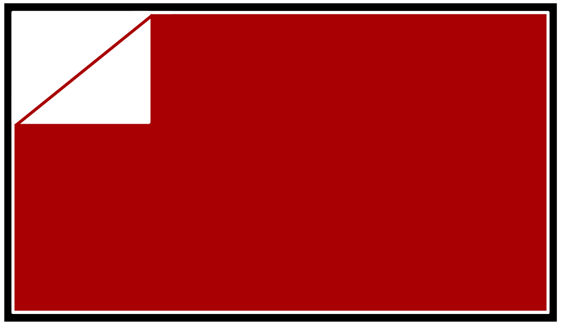
Analyze the Syntax

- COUNTIF(alpha-alpha-numeric-numeric: alpha-alpha-numeric-numeric, “condition”)
- You figure this out by looking at an Excel spreadsheet and experimenting.
- Question: Can the columns go beyond IV?

Design the Test Cases & Run Them

- Positive
  - Cells are in one column.
  - Cells are in one row.
  - Cells span multiple columns.
  - Columns are “AA”
  - Does A1:B1 give same results as B1:A1?
  - Various cell contents
  - Others?
- Negative
  - Column is AAA
  - Columns are specified as numeric
  - Cells are specified as 1A.
  - Symbols in cells.
  - Other?

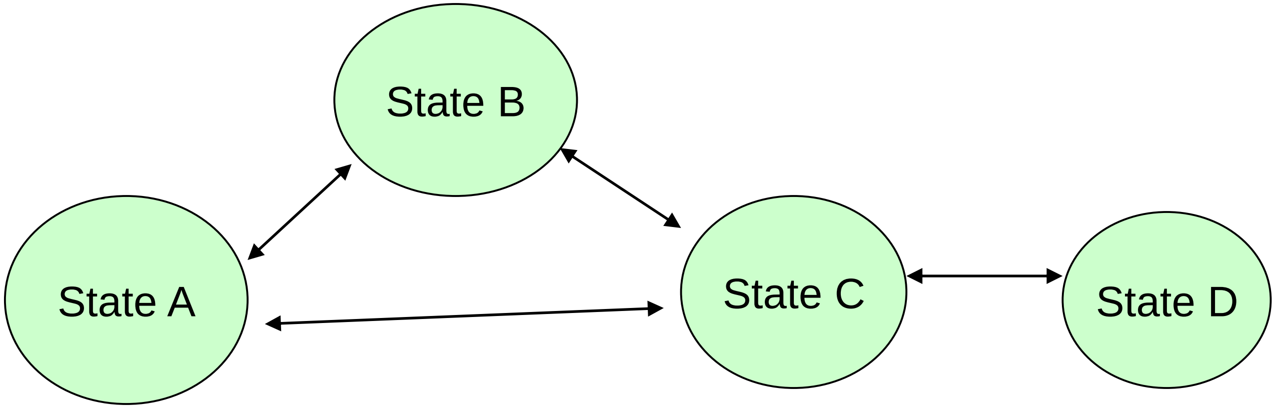




State Machine Testing

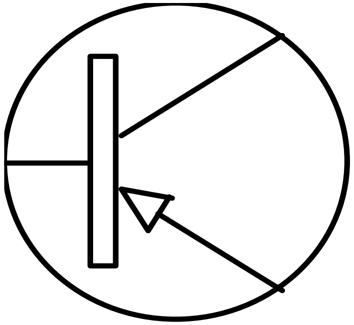
An excellent testing strategy for:

- Menu driven applications
- Systems designed using OO methods
- Any software that has a state-transition graph.



Examples of States

- Bank account
  - Active
    - In good standing
    - Overdrawn
  - Inactive
- An electric power generator
  - On-line
  - Off-line
    - Available
    - Unavailable
      - Scheduled maintenance
      - Unscheduled downtime



Significance of “States”

When the system is in a given “state”, some actions are allowed and others are not.

- If a bank account is in good standing (i.e., not overdrawn), cash withdrawals can be made, but if it is “overdrawn”, not cash withdrawals are allowed.
- If a generator is off line and “available”, it can be put on line, but if it is off line and “unavailable”, it cannot be put on line.

Comments on Bank Account Example

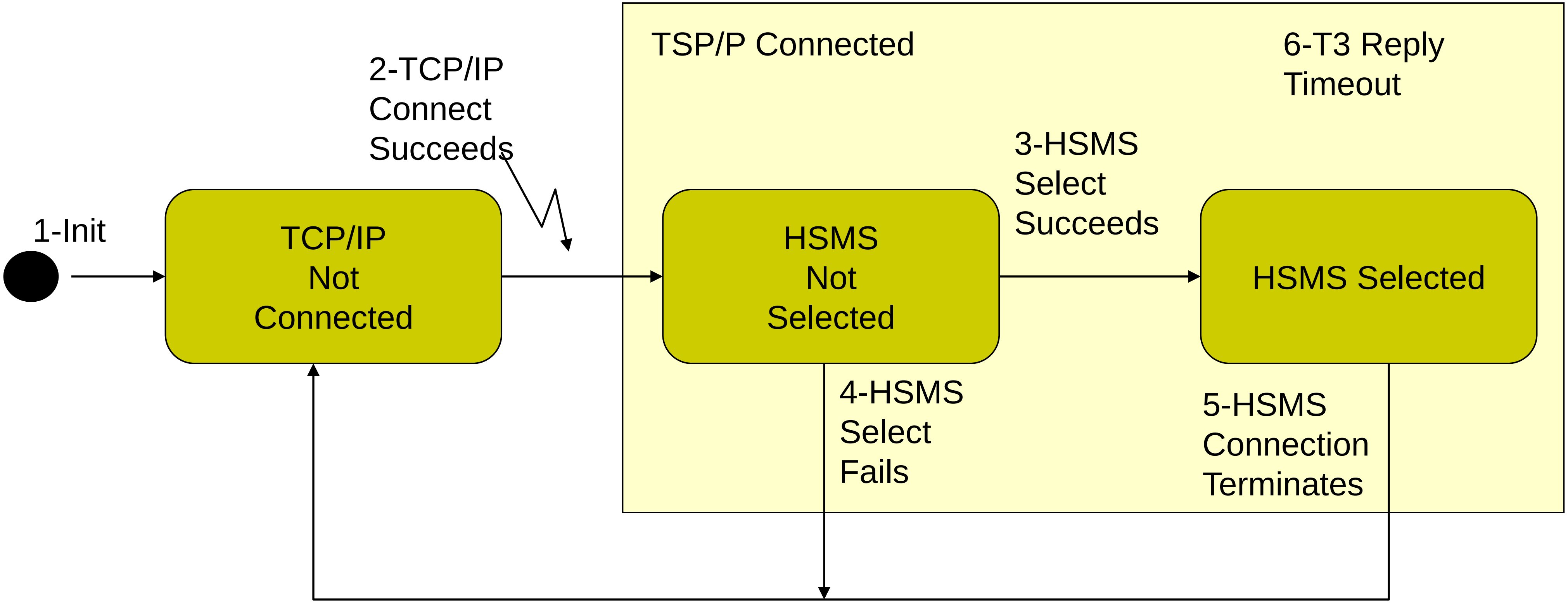
- Banking software would be a prime candidate for application of transaction testing techniques:
  - Deposit
  - Withdrawal
  - Check balance
  - Open a new account
  - Close account
- These are all “transactions”.
- But accounts go into different states, so “state machine testing” is also very applicable.
- For most systems, a combination of test techniques is needed.
- Deciding the techniques to use for a given system is what test design is all about.

The Method

- Obtain or create a state transition diagram for the system of sub-system to be tested.
- Positive tests: Define test cases for each state transition .
- Negative tests: Define test cases that try to force illegal state transitions.

Example 1 – Communication Protocol

State Diagram For a Communication Protocol



HSMS: High Speed Message Service

Example 1 – State Transition Table

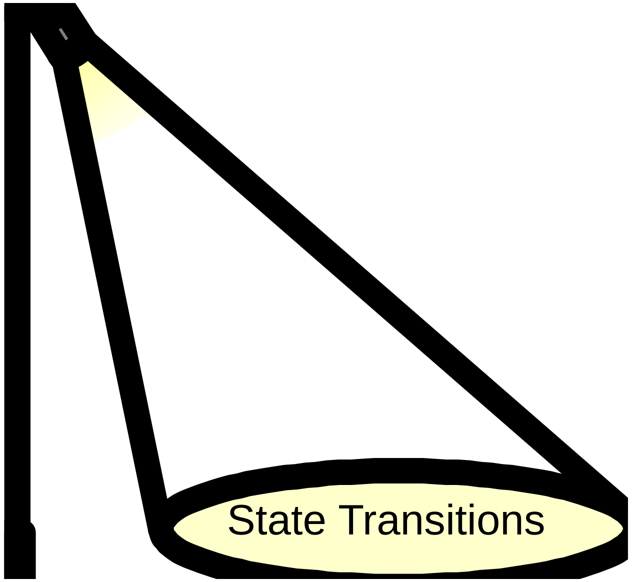
#	*Old State*	*New State*	*Trigger*	*Actions*
1	------	TCP/IP Not Connected	Initialization	Start T7 timeout.
2	TCP/IP Not Connected	HSMS Not Selected	TCP/IP connect succeeds: 1. TCP/IP “accept” succeeds.	1. Cancel T7 timeout; 2. Send Select.rsp with zero SelectStatus
3	HSMS Not Selected	TCP/IP Not Connected	HSMS Select fails: 1. T7 Timeout waiting for Select.req; or 2. Receive Select.req and decide to reject it and send Select.rsp with non-zero SelectStatus 3. Receive any HSMS message other than Select.req; or 4. Receive HSMS message length not equal to 10, or 5. Receive bad HSMS message header; or 6. T8 timeout waiting for TCP/IP; or 7. Other unrecoverable TCP/IP error.	1. Close TCP/IP connection.

State Transition Table - continued

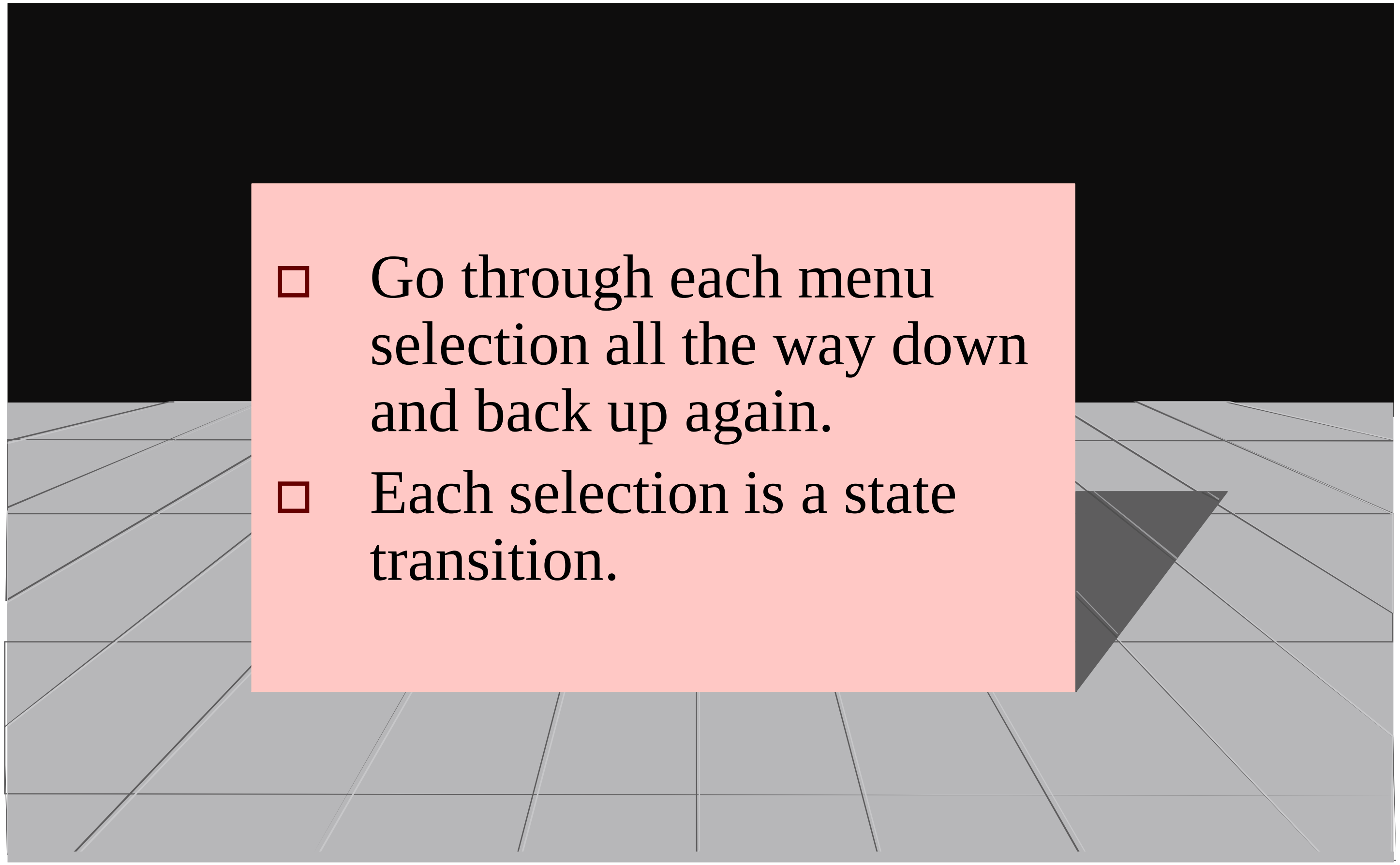
#	*Old State*	*New State*	*Trigger*	*Actions*
5	HSMS Selected	TCP/IP Not Connected	HSMS connection terminates: 1. Decide to terminate and send Select.req; or 2. Receive Separate.req; or 3.T6 timeout waiting for Linktest.rsp; or 4.Receive HSMS message<10; or 5.Receive HSMS> max; or 6.Receive bad HSM message header; or 7.T8 timeout waiting for TCP/IP; or 8. Other unrecoverable TCP/IP error.	1. Close TCP/IP connection.
6	HSMS Selected	HSMS Selected	T3 timeout waiting for data reply message.	1. Cancel Data Transaction as appropriate, but do not terminate the TCP/IP connection; 2. If entity is "Equipment", send SECS-II S9F9

Example 1 – Test Cases

- Each state transition becomes a test case.
- “Triggers” are the test case inputs, and “Actions” are the test case outputs.
- Look for tables of state transitions in the specs. Or ask the designers for one. If you can find one, most of your test design work is already done.



Example 2 – MS Word - “View” Menu



- □ Go through each menu selection all the way down and back up again.
- □ Each selection is a state transition.

Example 2 – MS Word – “View” Menu Test Results

- Bug - ?: Start in “Print Layout” view. Select “View” – “Document Map”. Can’t return.
- If you go to “Normal” view and back to “Print Layout” view to try to clear it, the document map is still there.
- Can only get rid of it by dragging the edge to the left side of the screen.
- Compare with behavior of “ View” – “Outline”.
- Bug - ?: Start in “Print Layout” view. Select “View” – “Mark-up”. Get the Message Window sometimes, but not always.
- Bug - ?: “View” - “Mark-up”. Can’t return. The only way to return is to go to “View” – “Tool Bars” and deselect “Reviewing”.

Menu Testing – Final Comments

- The problem here is determining what are correct state transitions and what are not.
- Must do a lot of inferring.
- Users Guide may help, but a lot of menus are usually not defined there, or not completely.
- Then you get into discussions of "bugs" verses "undocumented features".

Load and Stress Testing

- **Load Testing** – Forcing the system to do a large amount of processing.
- **Tress testing** – Operating the system in abnormal conditions.
- Sometimes it is not clear if a given test falls into the category of “load” or “stress”, but it doesn’t matter.

Load Testing

- Depends on the type of system.
- Large number of transactions.
- Large files.
- Large number of files.
- Large number of clients.
- Repeated operations.
- May require automated tools. Sometime “copy” and “Paste” can be used effectively to create conditions of heavy load.

Load Testing - Large Files

- My favorite technique.
- It frequently finds bugs.
- Mail Merge Example; 48,000 address records.
- To test email software, try attaching a large file. If the software doesn’t reject the file, then it should be expected to handle it properly, although it probably won’t.

Load Testing - Example

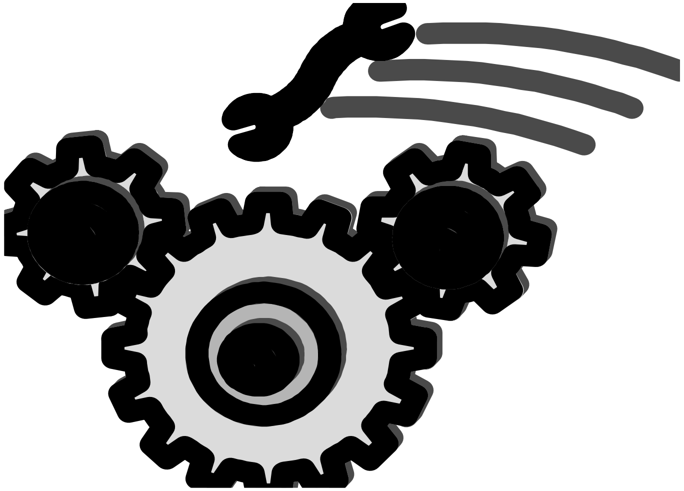
- MS Word – Large File (18 mbyte)
  - No bugs found.
- Second large file: 26622 pages.
  - No bugs found.
  - I wonder what would happen if you tried to print this? I haven’t tried.

Load Testing Example (continued)

Try opening both large files at once.

Stress Testing

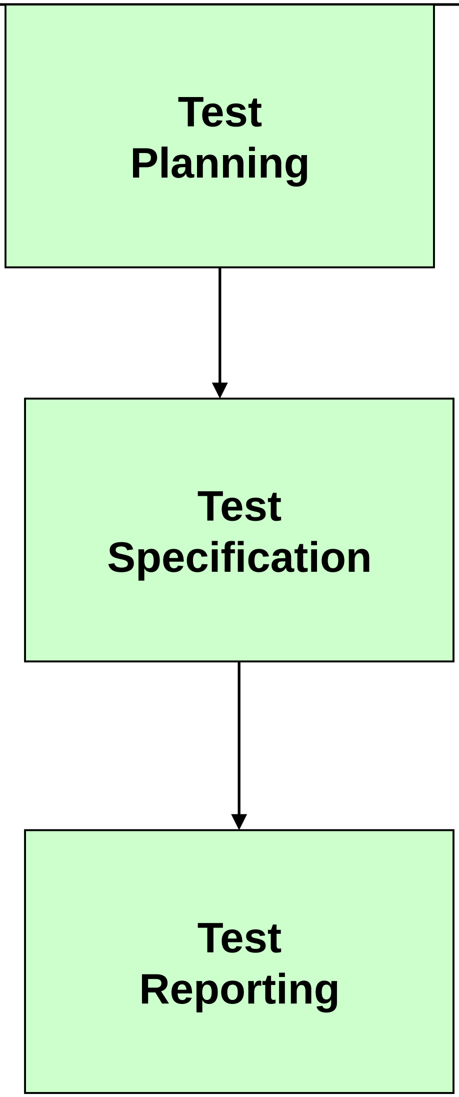
- Artificially restrict memory size.
- Networked system: Operate with a small number of nodes.
- Cause communications to be interrupted.
- Cause hardware to fail while in use.



Example – Printer

- My favorite example: From your PC, print a document and while it is printing, disconnect the communications cable to the printer.
  - Guaranteed to cause irrecoverable errors in the software.
- You don’t even have to be this perverse. Usually, just canceling the print job from the Windows control panel will do it. This finds bugs in the printer driver software for every HP printer I have ever used (and that’s a lot of different models over the years).
  - And it is such an annoying bug!

The Testing Process



Test Planning

Test Specification

Test Reporting

Test Planning

- Scope of the testing
- Approach to the testing
- Resources needed
- Schedule for the testing
- Items To Be Tested
- Features To Be Tested
- Testing Tasks To Be Performed
- Personnel Responsible For Each Task
- Risks Associated With The Test Plan

Source: IEEE Standard for Software Test Documentation; IEEE Std 829-1998

Test Specification

- **Test Design Specification**
  - Refines the test approach
  - Identifies the features to be covered by the testing
  - Identifies test cases
  - Specifies pass/fail criteria for the features
- **Test Case Specification**
  - Documents the actual input values and expected output for each test case
  - Identifies constraints on test procedures
  - Test case definitions are separated from test design to facilitate reuse
- **Test Procedure Specification**
  - Identifies all the steps:
    - Operate the system
    - Exercise the test cases
  - A cook-book
  - Separated from test design since they are intended to be followed step-by-step

Source: IEEE Standard for Software Test Documentation; IEEE Std 829-1998

Test Reporting

- **TTest Item Transmittal Report**
  - IIdentifies the software being turned over to the independent test group.
  - UUsed in the event that a formal beginning of testing is desired.
- **TTest Log**
  - UUsed by the people conducting the tests.
  - PPurpose is to record what occurred during the testing.
- **TTest Incident Report**
  - DDescribes any event that occurs during the testing which requires further investigation.
  - SSuch things as:
    - EEquipment failure
    - UEnexplained events
    - AAnomalies

Source: IEEE Standard for Software Test Documentation; IEEE Std 829-1998

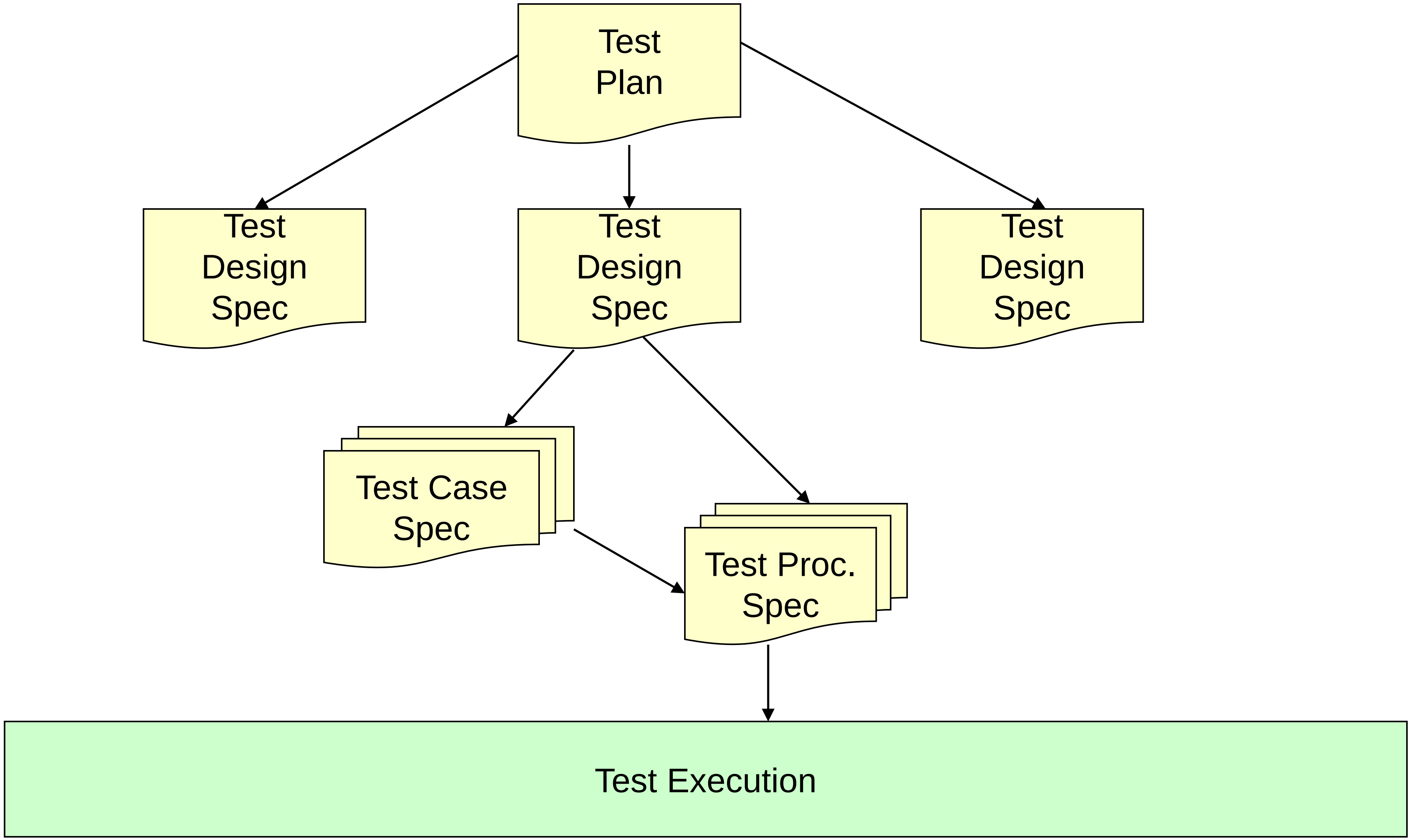
Test Reporting (continued)

Test Summary Report

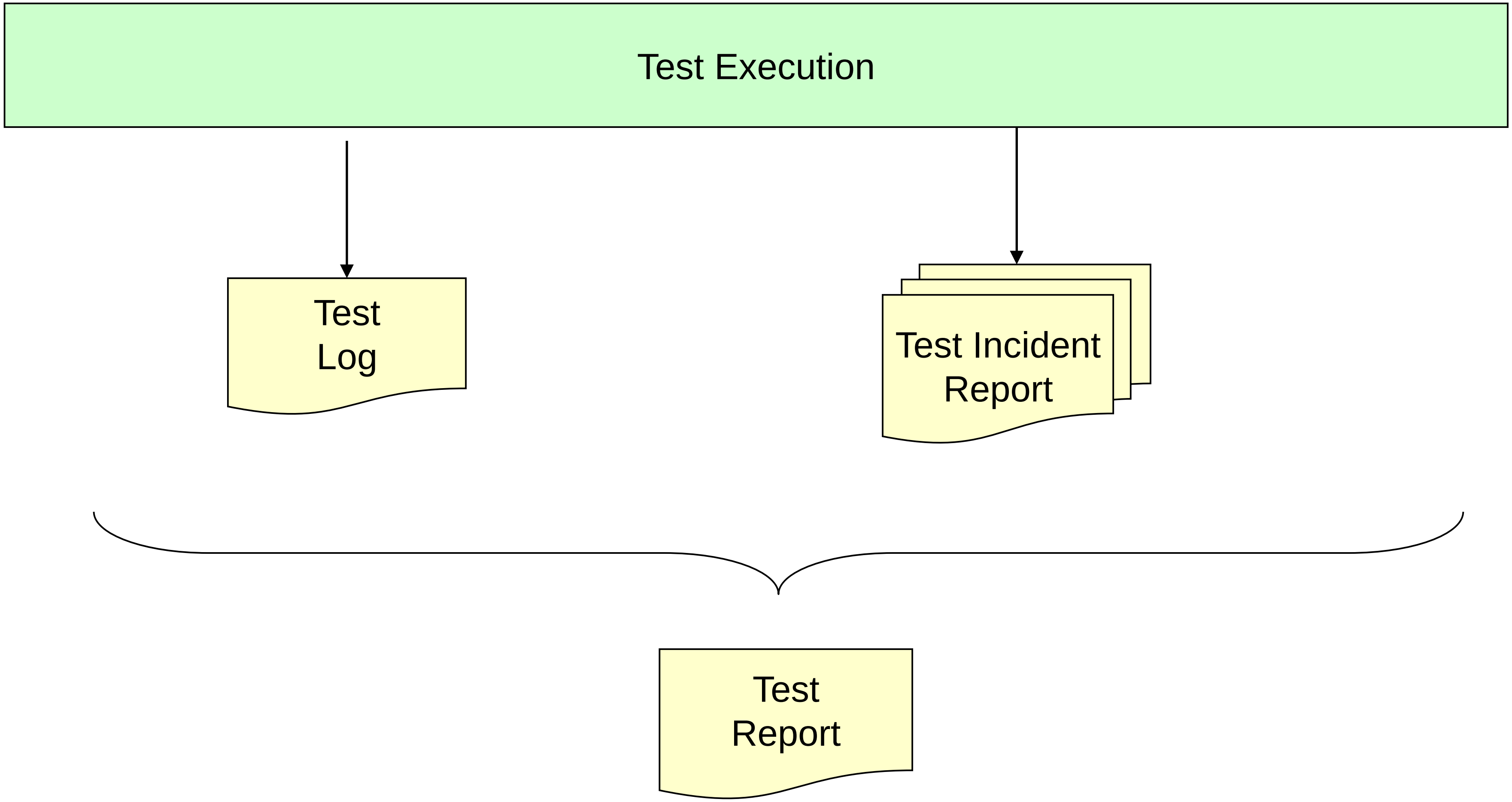
- TThe second most important test document.
- SSummarizes the results of the test cases.
- IIdentifies any variances form the tests plan.
- GGives an overall assessment of the test results

Source: IEEE Standard for Software Test Documentation; IEEE Std 829-1998

Test Documentation



Test Documentation (continued)

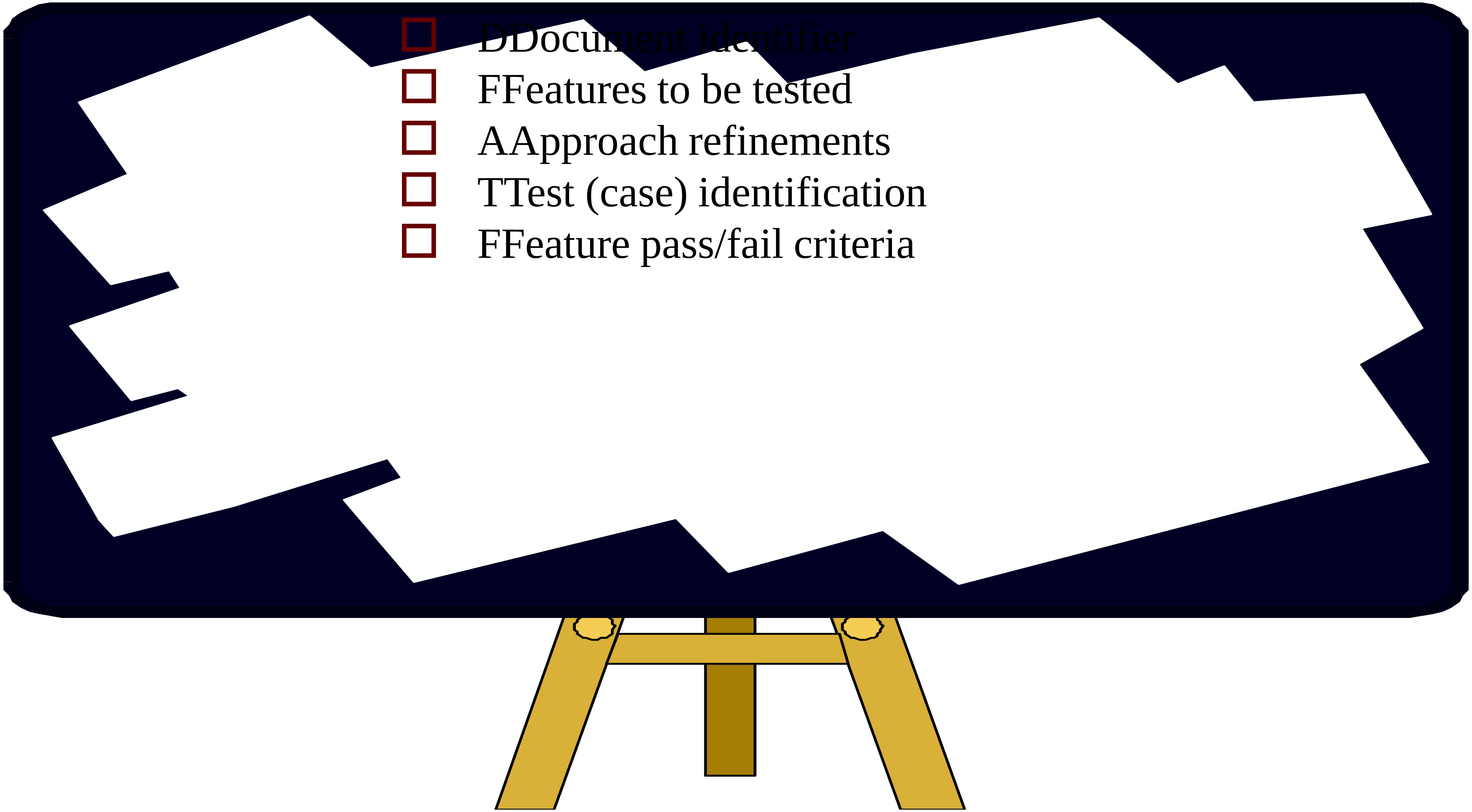


Contents of a Test Plan

1. Document identifier
2. Introduction
3. Test items
4. Features to be tested
5. Features not to be tested
6. Approach
7. Item pass/fail criteria
8. Suspension criteria and resumption requirements
9. Test deliverables
10. Testing tasks
11. Environmental needs
12. Responsibilities
13. Staffing and training needs
14. Schedule
15. Risks and contingencies
16. Approvals

Source: IEEE Standard for Software Test Documentation; IEEE Std 829-1998

Contents of a Test Design Specification



- □ DDocument identifier
- □ FFeatures to be tested
- □ AApproach refinements
- □ TTest (case) identification
- □ FFeature pass/fail criteria

Source: IEEE Standard for Software Test Documentation; IEEE Std 829-1998

Contents of a Test Case Specification

- DDocument identifier
- TTest items
- IInput specifications
- OOutput specifications
- EEnvironmental needs
- SSpecial procedural requirements
- IIntercase dependencies

Source: IEEE Standard for Software Test Documentation; IEEE Std 829-1998

Contents of a Test Procedure Specification

- □ DDocument identification
- □ PPurpose
- □ SSpecial requirements
- □ PProcedure steps

Source: IEEE Standard for Software Test Documentation; IEEE Std 829-1998

Content of a Test Log

- DDocument identifier
- DDescription
- AActivity and event entries

Source: IEEE Standard for Software Test Documentation; IEEE Std 829-1998

Contents of a Test Incident Report

- □ Document identifier
- □ Summary
- □ Incident description
- □ Impact

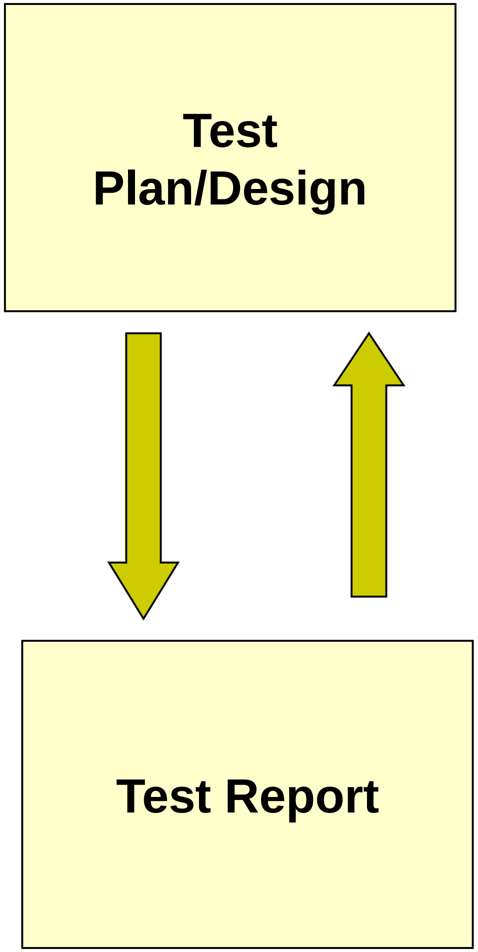
Source: IEEE Standard for Software Test Documentation; IEEE Std 829-1998

Contents of a Test Report

- □ Document identifier
- □ Summary
- □ Variances
- □ Comprehensive assessment
- □ Summary of results
- □ Evaluation\Summary of activities
- □ Approvals

Source: IEEE Standard for Software Test Documentation; IEEE Std 829-1998

Test Plan – Test Report



- *Defines the test environment and test cases to be performed.*
- *Specifies that amount of testing to be done.*
- *Provides a definitive basis for knowing when the testing is done.*
- *Answers the question of how much testing is enough.*

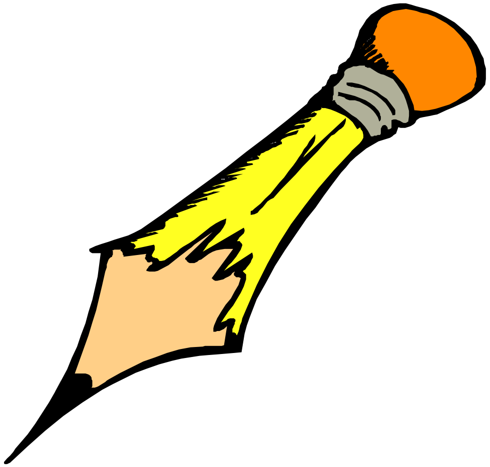
- *Reports the actual test results relative to plans.*
- *Tells us whether all of the planed testing was actually done or not.*

Use IEE Std 829 as Guidelines

- These test documents as defined by IEEE-829 are very comprehensive.
- Use them as guidelines.
- Make modifications as needed by the projects being tested and the environment.
- Small projects, especially, will need to streamline these document standards.

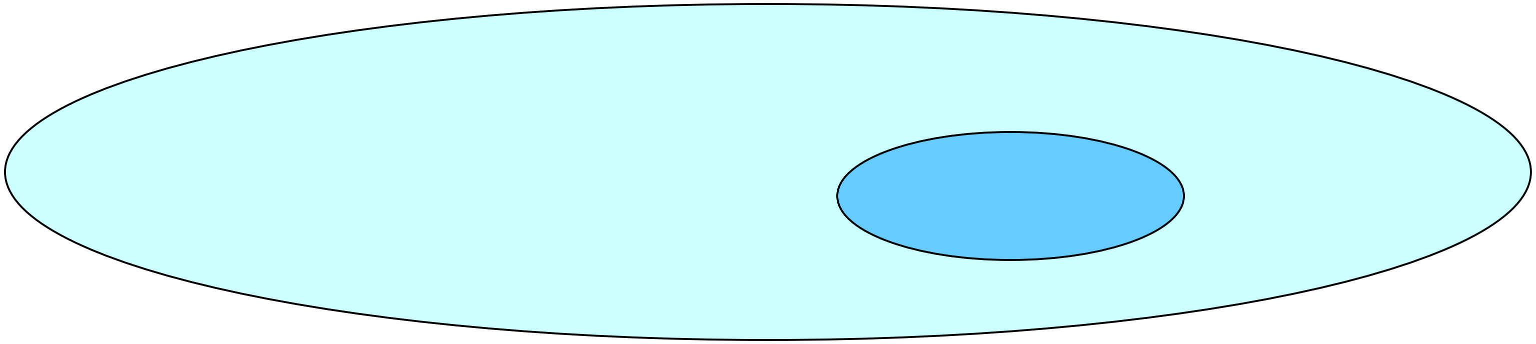
Suggested Modifications

- May want to combine the test plan and test design specification.
- May want to combine the test case and test procedure specifications.
- May want to add “Prerequisites” to the test case specification.
- Add a test coverage matrix in the test plan or design.
- In the test report, include data on the amount of man-hours spent in the testing.



Test Planning

- The major issue is “coverage”.
- How much is needed?
- Remember, exhaustive testing is not possible.
- Must address this issue in the test plan.



The Essence of Test Planning

“Test design is the judicious selection of a small subset of conditions that will reveal the characteristics of the software.”

Watts Humphrey

Elements of Test Planning

- **Establish objectives for each test phase**
- **Establish schedules and responsibilities for each test activity.**
- **Determine the availability of tools, facilities, and test libraries.**
- **Establish the procedures and standards to be used:**
  - Planning
  - Test execution
  - Reporting
- **Set the criteria for:**
  - Test completion
  - Success of each test

Importance of Test Planning

- Asked many integration testers from many projects what they would do differently next time.
- Response: Do a better job of test planning.
- It’s too late to begin test planning when testing actually begins.
- Test planning should be done in conjunction with requirements analysis/definition

Source: Managing a Programming Project, P.W. Metzger

Added Benefit of Early Test Planning

- Added benefit: Many requirements & design problems will be identified by the test planning & design activities.
- Some experts claim that when test planning and design take place early in the project, more problems are found due to the test planning /design than in actual test.

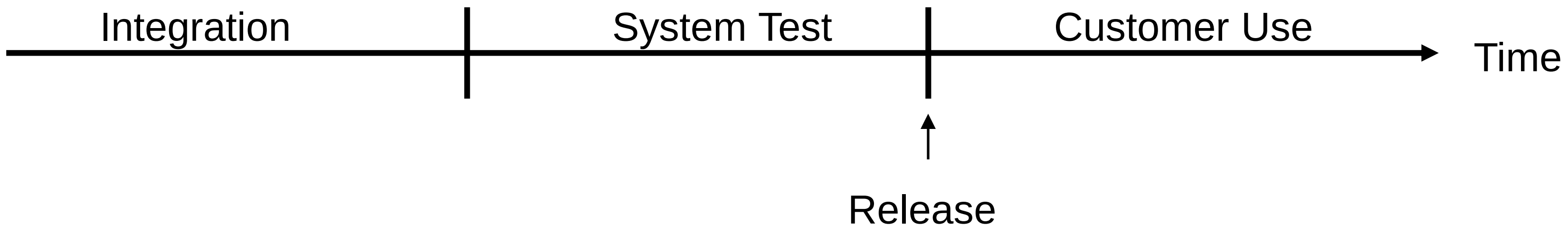
Where to Focus Testing

- □ Most likely errors.
- □ Most visible errors.
- □ Most often used program areas.
- □ Most critical areas of the program.
- □ Distinguishing areas of the program.
- □ Hardest to fix areas.
- □ Most understood by the tester.
- □ Least understood by the tester.

Source: Testing Computer Software, Kaner et. al

Test Scheduling

- Allow time for fixing bugs.
- Does this schedule make any sense:



Integration

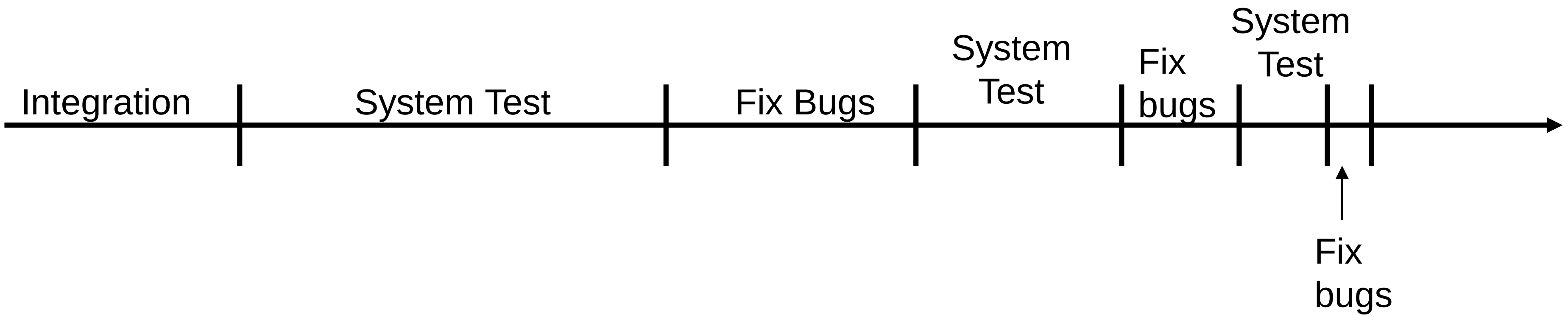
System Test

Customer Use

Time

Release

- The schedule should look like this:



Integration

System Test

Fix Bugs

System Test

Fix bugs

System Test

Fix bugs

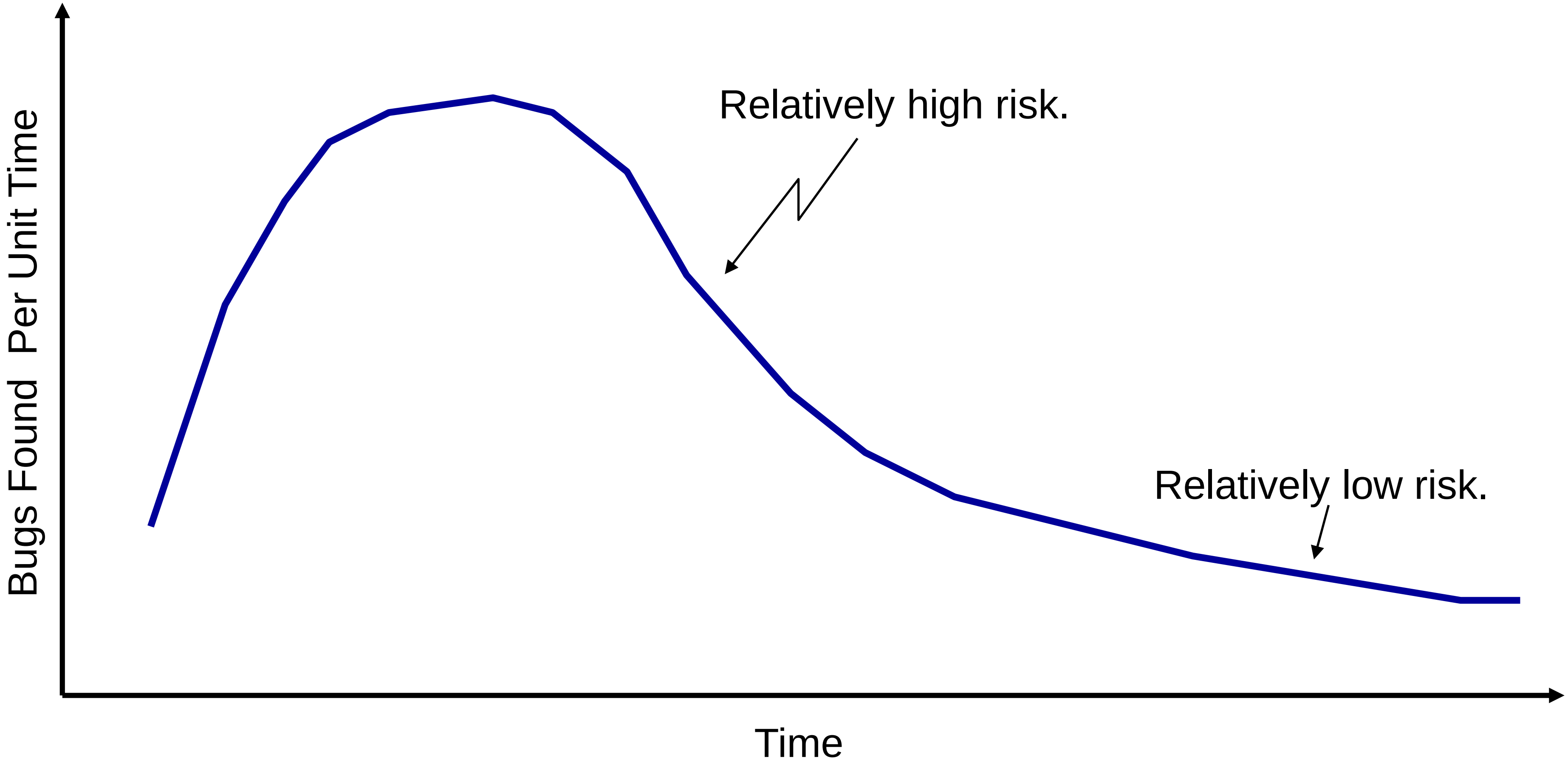
Selecting Test Cases

- □ The art of testing is that of picking the test cases most likely to find errors.
- □ Of the 26^{10} possible test cases, only a few are likely to find errors.
- □ Concentrate on picking a few that tell you different things, rather than ones that tell you the same thing over and over.
- □ Use the strategies covered in this class.

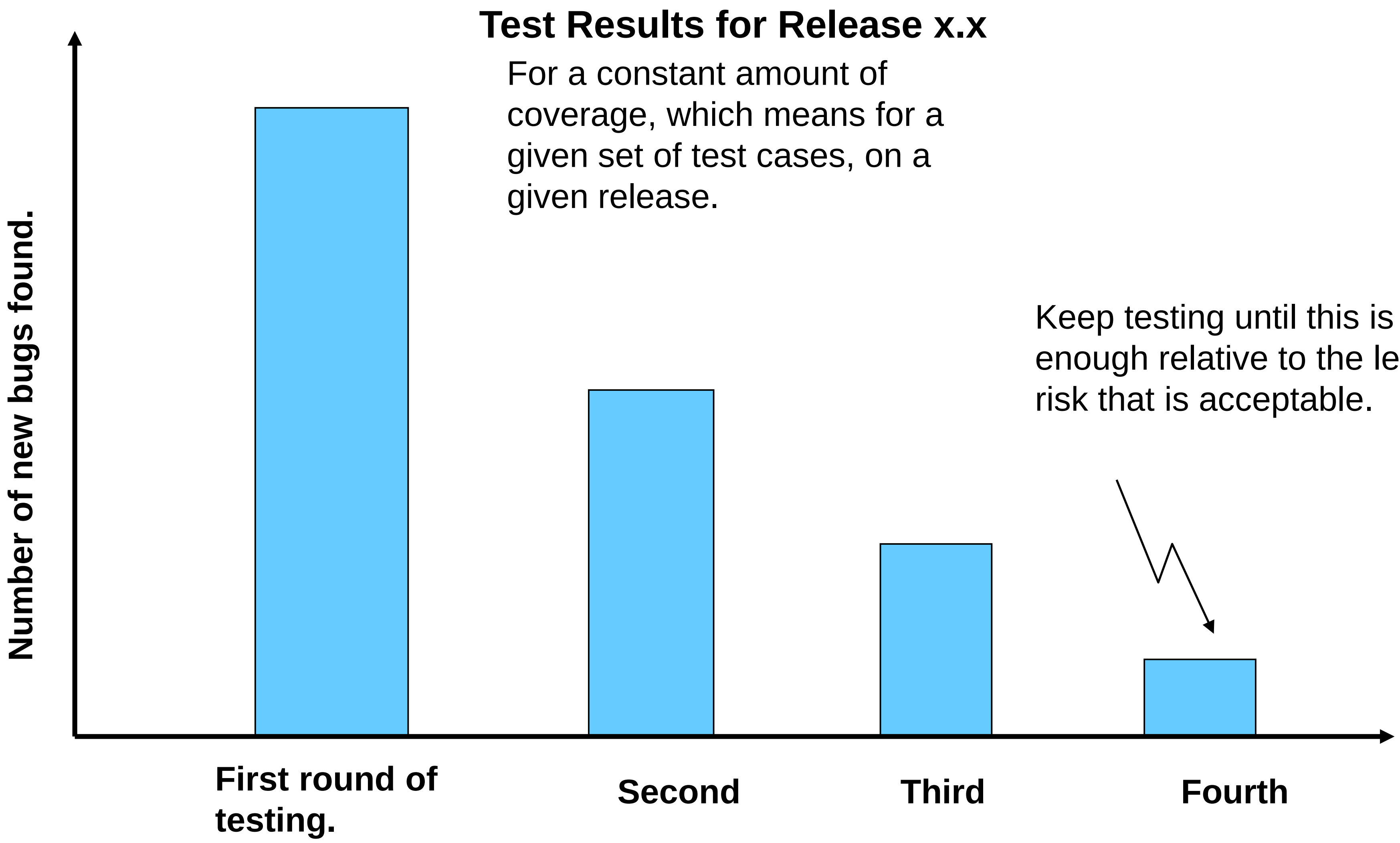
How Much Coverage Is Needed?

- The answer depends upon the level of risk that can be tolerated.
- The risk we are talking about is the risk of surprises in the field (bugs that weren’t found in the pre-release testing).
- If high risk can be accepted, less testing can be done.
- If there must be a low risk of previously-undetected bugs showing up in the field, must do more testing.
- This is a management decision, not a technical decision.

Release Readiness



Testing a Release



For a constant amount of coverage, which means for a given set of test cases, on a given release.

Keep testing until this is small enough relative to the level of risk that is acceptable.

Test Coverage Strategy

- Test all new features, and surrounding.
- Test all bug fixes, and surrounding.
- Test other areas of the software based upon:
  - Customer scenarios.
  - Error prone modules.
  - Critical functions.
  - Other.
- Use test techniques discussed previously in this class based upon the type of software being tested.
- Design both positive and negative tests.

Test Coverage Matrix

- A very effective test planning tool.
- Maps software features into test cases.
- Shows any gaps in the testing and redundant testing.
- Also gives insight into which test cases are most "productive".
- When software is modified, indicates which test cases need to possibly be modified.
- Visual presentation of the test space

How Does It Work

Test Coverage Matrix
Version x.x, Test Plan xyz

	TC #1	TC #2	TC #3	---	---	---	TC #n
Feature #1		X					
Feature #2	X	X	X				
Feature #3			X				
Feature #4						X	
---				X	X		
---			X				
Feature #n							

What Are the “Features”

- For requirements-based testing, they consist of each of the individually identifiable requirement.
- For a new release of an existing software system, they are the new features, enhanced features, and bug fixes.
- Equivalence classes.
- Error conditions.
- Negative tests to be conducted

Example – New Release

- A new release contains:
- Five new features.
- Ten enhancements to existing features.
- Ten bug fixes
- Some code that was rewritten for ease of maintenance

Example – New Release

	TC #1	TC #2	TC #3	TC #4	TC #5	TC #6	TC #n
New Feature #1							X
New Feature #5		X					
Enhancement #1			X	X	X		
Enhancement #10			X				
Bug #1			X				
Bug #10						X	
Rewritten code							

Example – Error Messages

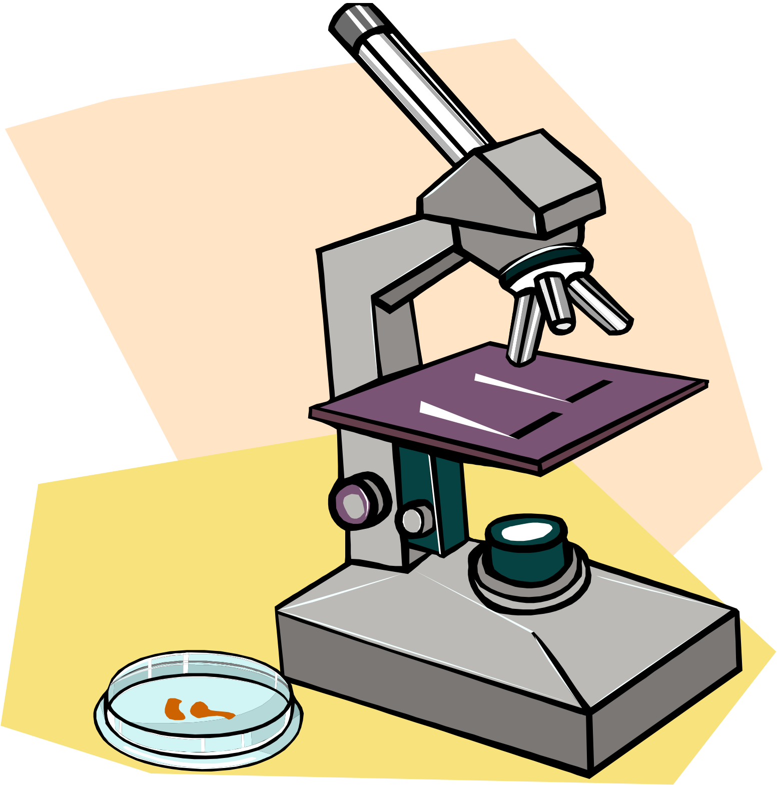
	TC #1	TC #2	TC #3	TC #4	TC #5	TC #6	TC #7
Error Msg. #1	X						
Error Msg. #1		X					
Error Msg. #3			X				
Error Msg. #4				X			
Error Msg. #5					X		
Error Msg. #6						X	
Error Msg. #7							X

Test Effectiveness

- What is an “effective” test?
- How do we know if the testing is accomplishing what we want?
- Testing is expensive: How do we know if we are getting our money’s worth?
- Can it be quantified?

Definition

- The purpose of testing is to find bugs.
- An effective test process will do that completely.
- A measure of test effectiveness: escaped bugs.



Escaped Bugs

Versions Released To The Field

Quantifying Test Effectiveness

- Count the number of newly reported defects from the field after release of the version.
- Make sure they are not duplicates of defects previously reported (prior to release)
- This is a “zero defects” type of metric (down is better).
- Trend it from version to version.

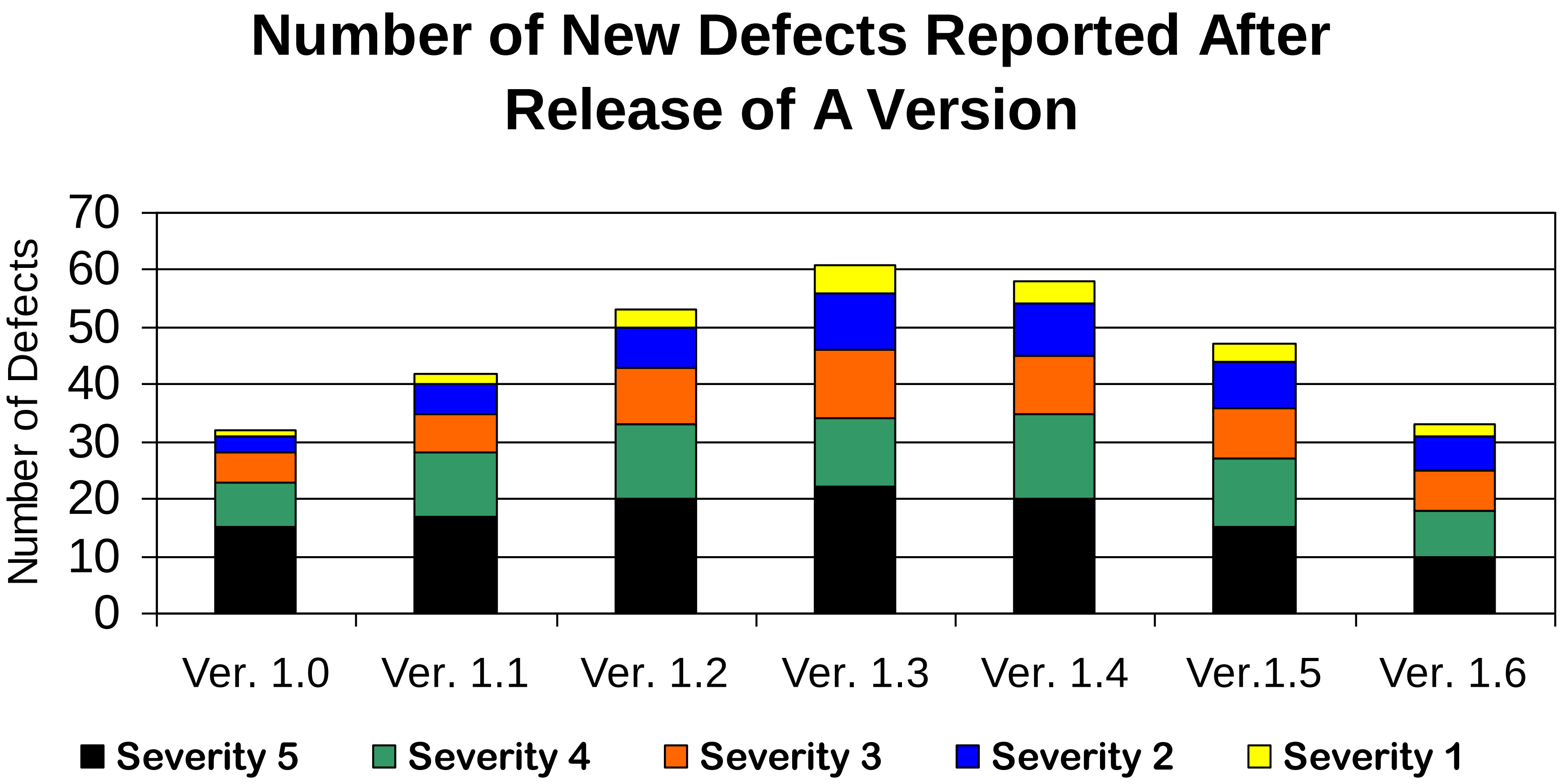
Test Effectiveness - Example

Version 1.6 release date: Sept. 1

Test effectiveness = 7

Number	Date	Severity	Version	Description
111	Aug. 15	5	1.5	Screen lay-out
112	Aug. 31	4	1.5	Screen lay-out
113	Sept 1	3	1.4	Menu tree problem.
114	Sept. 3	2	1.6	Incorrect temperature calculated.
115	Sept. 3	2	1.6	Wrong data displayed.
116	Sept. 3	3	1.6	Menu missing a selection.
117	Sept. 4	5	1.5	Wording is poor.
118	Sept. 4	1	1.6	Report look-up causes crash.
119	Sept. 5	3	1.6	Entry is lost.
120	Sept. 7	4	1.6	Screen lay-out poor.
121	Sept. 10	5	1.6	Spelling error

Test Effectiveness Trend Chart



Test Effectiveness For Test Cases

- An effective test case is one that finds a bug.
- Bug yield.
- What is the point of running tests that find no problems.
- Keep statistics.
- Track bugs to the test cases that found them.

Example – Bug Yield.

Bugs found per test case.

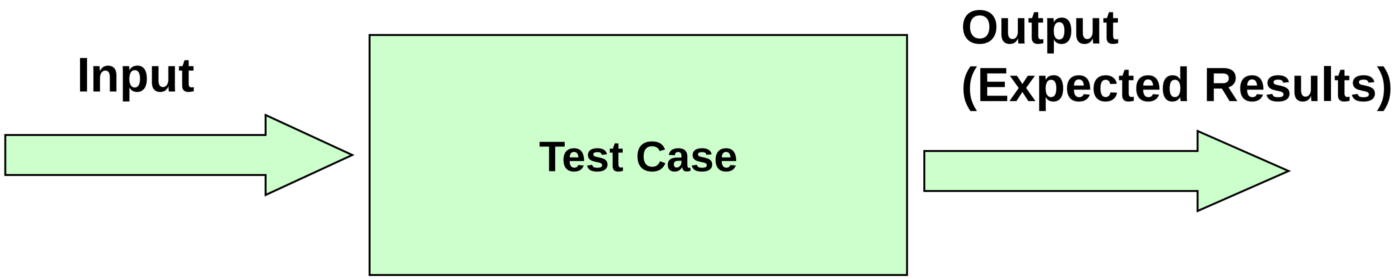
	Version 1.0	V1.1	V1.2	V1.3	V1.4	V1.5	V1.6
TC1	1	0	0	0	0	0	0
TC2	2	3	3	2	1	2	1
TC3	0	0	0	0	0	0	0
TC4	1	1	1	1	1	1	1
TC5	3	4	1	2	3	2	3
TC6	2	0	1	1	0	0	
TC7	0	0	0	1	0	0	0

Test Cases Wear Out

- A given test case can only be effective in finding a certain type of bug (boundary values, control flow, syntax, etc.).
- As those bugs are found *and fixed*, that test case will no longer be effective.
- Unless the software is unstable, which is another problem altogether.
- The test group must be continually retiring test cases and designing new ones:
  - Based up actual testing experience on what is effective in finding bugs.
  - To adapt to evolving software.

More on Test Cases

Test cases are basically input/output specifications.



Input

Test Case

Output
(Expected Results)

Example – USB Device

Testing a USB device.

Test Case #nnn

1. Prerequisites:
   - The device is powered off.
   - The signal source is disabled.
2. Inputs:
   - The device is powered on.
3. Outputs:
   - The device sends the following enumeration data to the main computer, device descriptor, configuration descriptor, interface descriptor.
   - The signal source is enabled.

Example – Communications System

Test Case # nnn

Prerequisite: Local entity is in "HSMS Selected" state.
Input: Local entity sends "link test request". Remote entity does not respond.
Expected Results: After T6 length of time, local entity closes TCP/IP connection.

Test Case – Test Procedure

- Sometimes, it is best to combine the test case (input/output) specification with the test case procedure.
- Sometimes, if the procedure for several test cases is the same, it is better to keep them separate, and only define the procedure once.

Test Case Specification – Table Format

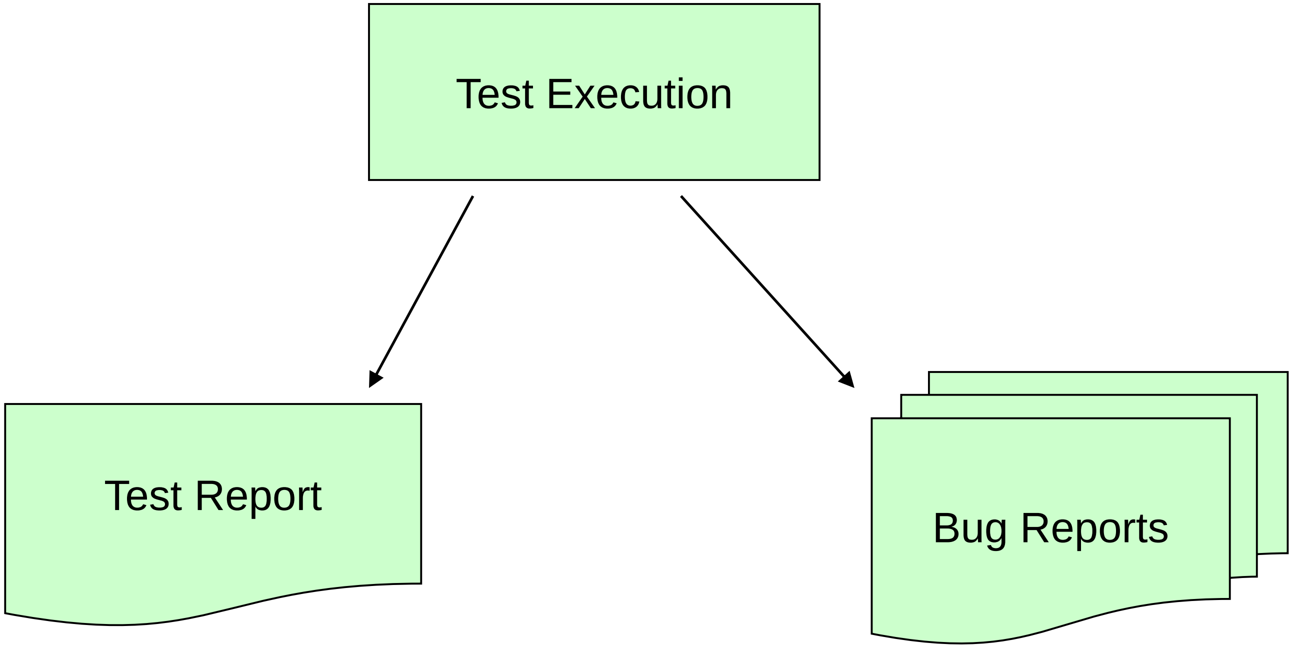
Test Case	Input	Output
TC1	01Jan04	Date field in header record is updated to 01Jan04.
TC2	01Feb04	Date field in header record is updated to 01JFeb04
TC3	01Dec04	Date field in header record is updated to 01Dec04
TC4	30Jan05	Date field in header record is updated to 30Jan05
TC5	32Feb05	Entry is rejected; date field is not updated.
TC6	15Abc05	Entry is rejected; date field is not updated.
TC7	99mar04	Entry is rejected; date field is not updated.

Using Bug Data in Test Planning

- Bug reports are a tester’s gold mine.
- If there is a bug tracking system, mine it.
- Use bug data to identify:
  - Error prone modules.
  - Error prone interfaces
  - Types of testing that has been effective.
  - Test cases that have been effective.
- Remember that test cases wear out, though.

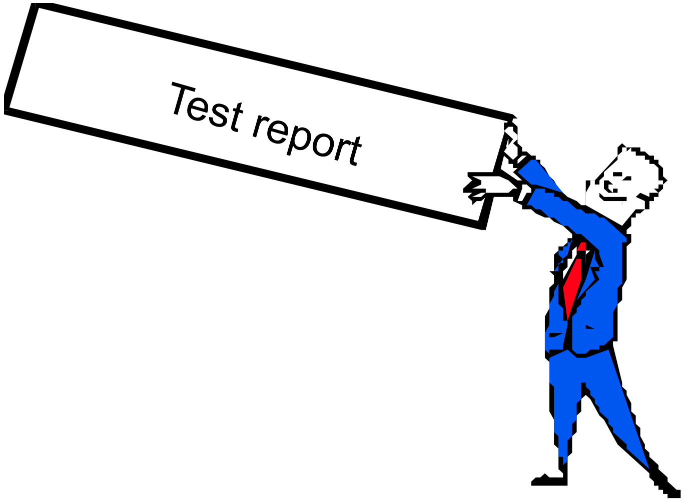
Test Results Reporting

Involves much more than reporting the bugs found.



Contents of a Test Report (Again)

- Document identifier
- Summary
- Variances
- Comprehensive assessment
- Summary of results
- Evaluation\Summary of activities
- Approvals



Source: IEEE Standard fro Software Test Documentation, IEEE Std 829-1998

Need a Test Report Every Time.

- Every round of testing should have a test report written.
- Every test report should be linked to a test plan.
- For a new product release, if there are several builds tested before one is released, write a test plan and report for each build.

Tips on Test Results Reporting

- Include data on the man-hours spent in testing.
- Be sure to identify ant deviations from the test plan, especially any planned tests that were not performed.
- Identify all bugs found (may be by reference to bug report numbers).
- Identify the most serious bugs found.
- May want to include an overall evaluation of the "soundness"" of the software under test:
  - It was hard to break it.
  - Many serious bugs were fond.
  - Etc.

Tips on Test Reporting (continued)

- **The test groups responsibility is to provide data:**
  - How much testing was done.
  - What type of testing was done.
  - Was all of the planned testing done.
  - How many bugs were found.
  - Were the bugs serious?
  - Etc.

- **Management's responsibility is to decide whether or not to release the software.**
  - This decision should be based upon the level of risk that management is willing to assume.
  - This acceptable risk level is determined by many factors, and is outside of the test group's purview.

Tips on Test Reporting (continued)

- Finally, be sure to archive all test reports.
- Never delete them or throw them away.
- They contain very valuable information.

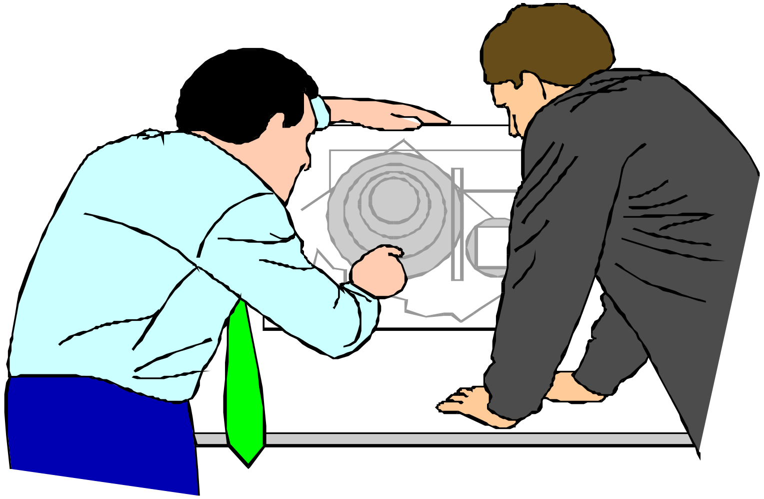
Object Oriented (OO) Systems

Object-oriented Vocabulary

- **Object:** A software packet containing a collection of related data (in the form of variables) and methods (procedures) for operating on the data.
- **Method:** A procedure contained within an object that is made available to other objects for the purpose of requesting services of that object.
- **Message:** A signal from one object to another that requests the receiving object to carry out one of its methods.
- **Class:** A set of objects that share a common structure and behavior (manifest by the set of methods). A template from which objects can be created.

OO Testing Considerations

- Class testing corresponds to unit testing.
- Testing collections of classes corresponds to integration testing
- Use case testing corresponds to system testing



Classes

- □ The fundamental unit of an object-oriented system.
- □ Contains both interface and implementation.
- □ Much OO testing is centered around classes.

Class Testing

- For each class, must decide whether to test it independently or as a component of a larger part of the system.
- Determined by:
  - Methods that interact.
  - Role of the class (risk involved with it contending bugs)
  - Complexity of the class
  - Amount of effort to develop a test driver.

Source: Testing Object-oriented Systems, Robert Binder

Class Testing (continued)

Two aspects of test planning/design:

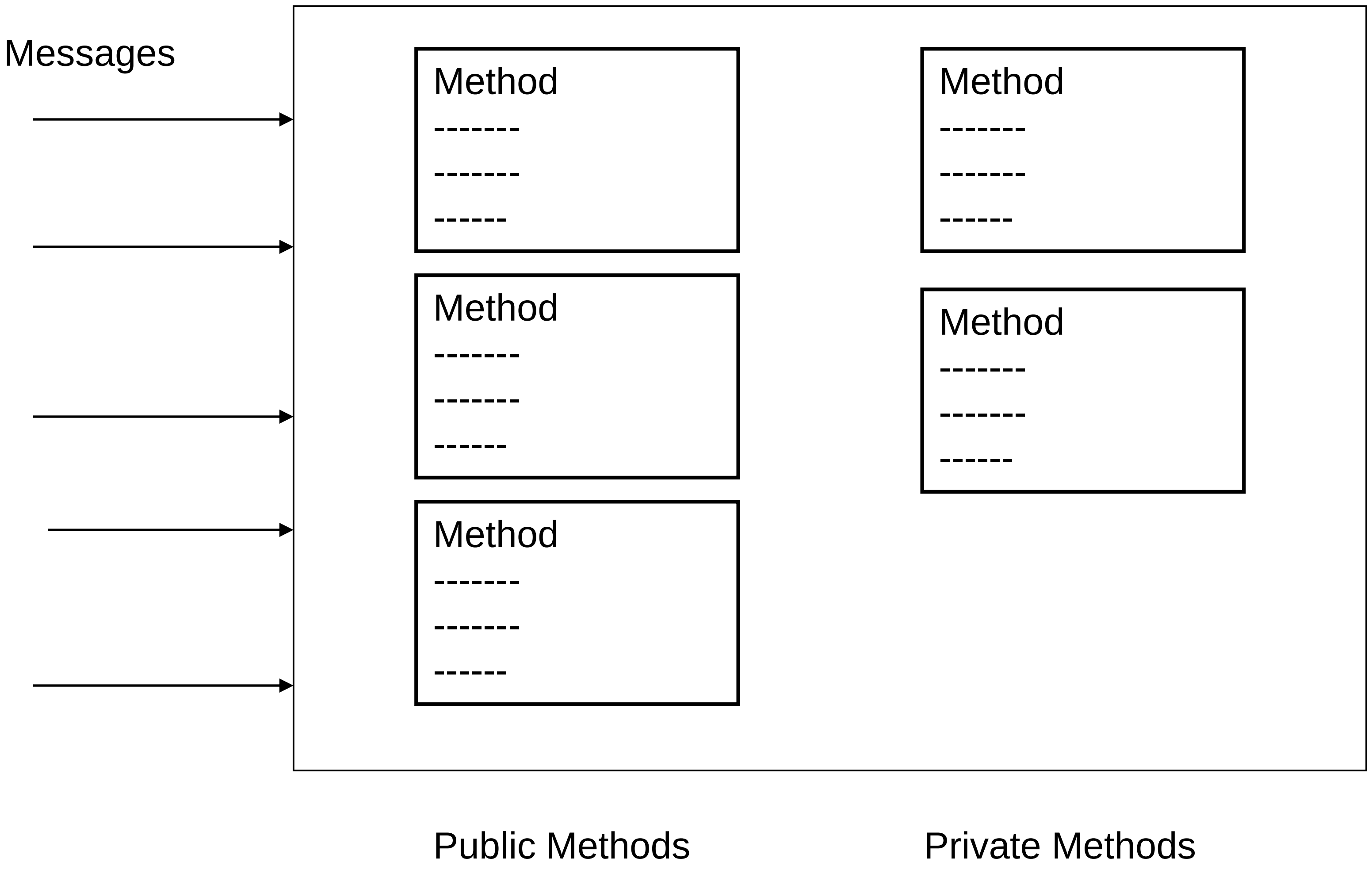
- Identification of test cases
- Development of a test driver
  - Creates instances of the class to run a test case,
  - Invokes class’s methods,
  - Report results.

Class Testing (continued)

- □ Exercising methods in various sequences is necessary to reveal class bugs.
- □ Test classes by sending messages to methods one at a time.
- □ Do private methods first.
- □ Class testing must exercise the cooperation of all methods that interact.

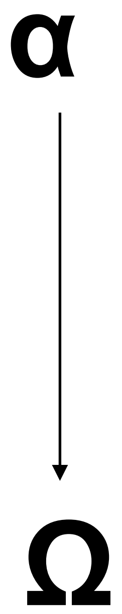
Source: Testing Object-oriented Systems, Robert Binder

Class Testing (continued)



Class Testing (continued)

- Alpha-Omega Cycle
  - Alpha state: The object declaration before it is constructed.
  - Omega state: The “remains” of an object after it has been deleted or destructed.
- Alpha-Omega cycle takes the objects from the alpha state to the omega state .
  - Send one message to every method at least once.
- Represents a minimum level of class testing.


α

Ω

Source: Testing Object-oriented Systems, Robert Binder

Class Testing (continued)

Alpha-Omega Test Suite

- Six basic steps
- Test driver sends one message to each of the types of methods in this order:
  - New or constructor methods
  - Accessor (get) methods)
  - Boolean (predicate) methods
  - Modifier (set) methods
  - Iterator method
  - Delete or destructor methods.
- Do private methods first, then public methods.

Source: Testing Object-oriented Systems, Robert Binder

Class Testing (continued)

- Abstract class: A class that has no instances.
  - In ADA: Procedures and functions declared as “abstract”.
  - IN C++: Any class that contains at least one pure virtual function.
  - In Eiffel: Includes at least on deferred feature.
  - In Java: Class is designated as abstract or has at least one method designated as abstract.
- Has operation declarations, but no methods or bodies.
- Interfaces are often declared through abstract classes.

Source: Testing Object-oriented Systems, Robert Binder

Class Testing (continued)

- Since an abstract class can’t be instantiated, it can’t be tested as written.
- Tested by developing a subclass that implements all of the abstract methods in the abstract class.
- A test suite is designed for the entire hierarchy.
- Write test cases for each subclass method that implements an abstract superclass method.

Source: Testing Object-oriented Systems, Robert Binder

Class Testing – State Transitions

- State transition diagram s show the behavior associated with instances of a class graphically.
- Use the state transition diagram.
- Write a test case for each state transition.

State	Events	State	Exceptions Thrown

Source: A Practical Guide to Testing Object-Oriented Systems, McGregor & Sykes

Class Testing – Test Driver

- After test cases are identified, must implement a driver to run each and report results.
- Creates one or more instances of a class.
- Classes are tested by creating instances and testing the behavior of those instances.
- Considerable effort can be required:
  - Identification of test cases
  - Writing test drivers

OO Integration Testing

Why is it different from integration testing in traditional systems?

- Declarative language, instead of imperative (sequential).
- Declarative languages suppress sequentiality. Source statement order has little to do with execution order.
- Event driven in nature.
- No functional decomposition. Concepts of “top-down” and “bottom up” do not apply.

OO Integration Testing (continued)

Need a appropriate construct. It should be:

- Compatible with composition,
- Avoid the structure-based goals of traditional integration testing,
- Support the declarative aspect of object integration,
- Be clearly distinct from unit- and system-level OO testing.

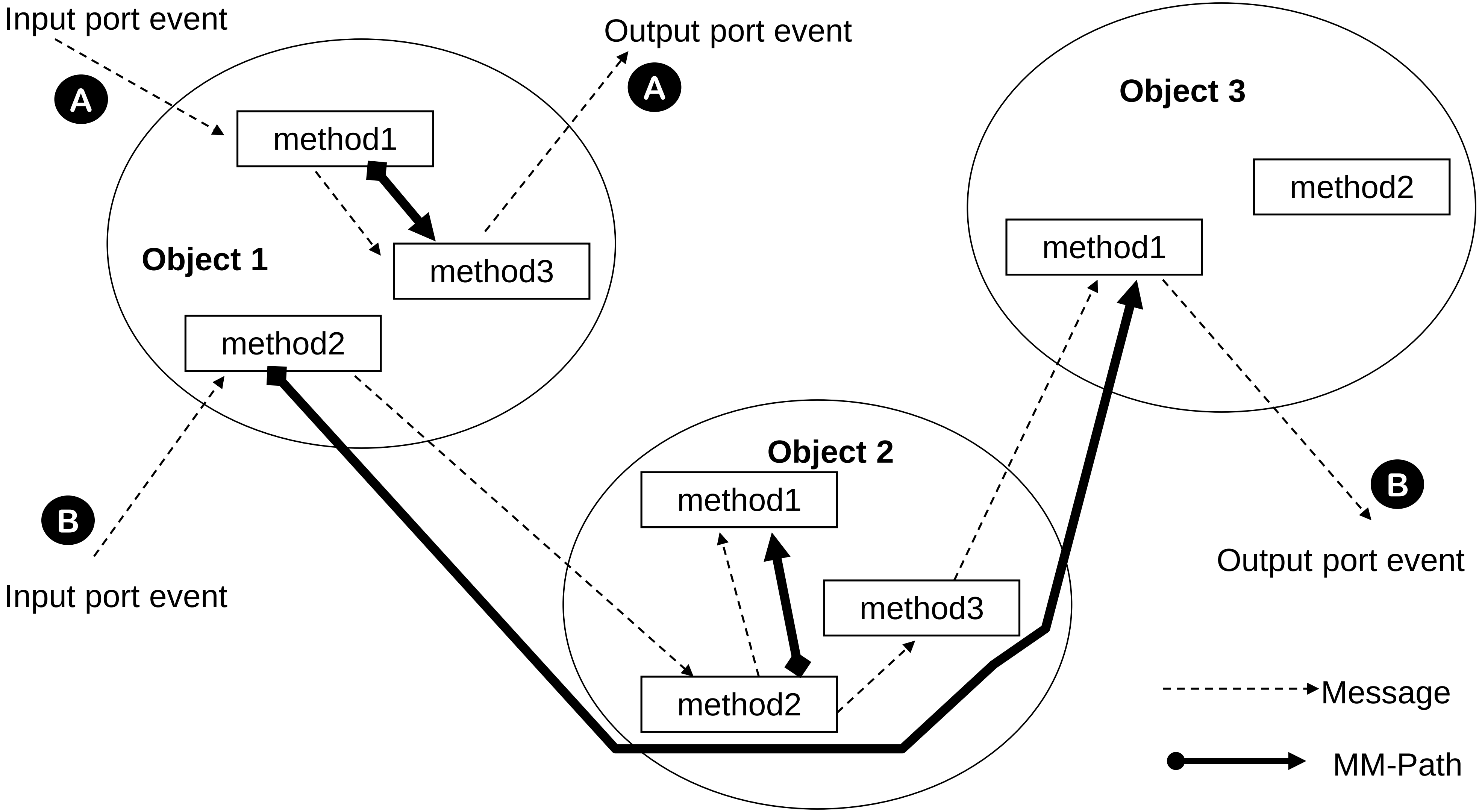
Source: ”Object-Oriented Integration Testing”, P.C. Jorgensen & C. Erickson, *Comm. ACM,* Sept. 1994

MM Path Definition

- A Method-Message Path (MM-Path) is a sequence of method executions linked by messages.
- An MM-Path starts with a method and ends when it reaches a method that does not issue any messages of it's own.
- This is called "*message quiescence*".
- MM-Paths are composed of linked method-message pairs.
- Examples: Paths A & B in the following diagram.

Source: "Object-Oriented Integration Testing", P.C. Jorgensen & C. Erickson, *Comm. ACM,* Sept. 1994

Method-Message (MM) Paths



Port Events

- Input port event: A system-level input event that causes the execution of OO software to begin.
- Output port event: A system level response to an input port event. When the system reaches this state, it is “ quiet” and waiting for another input port event.

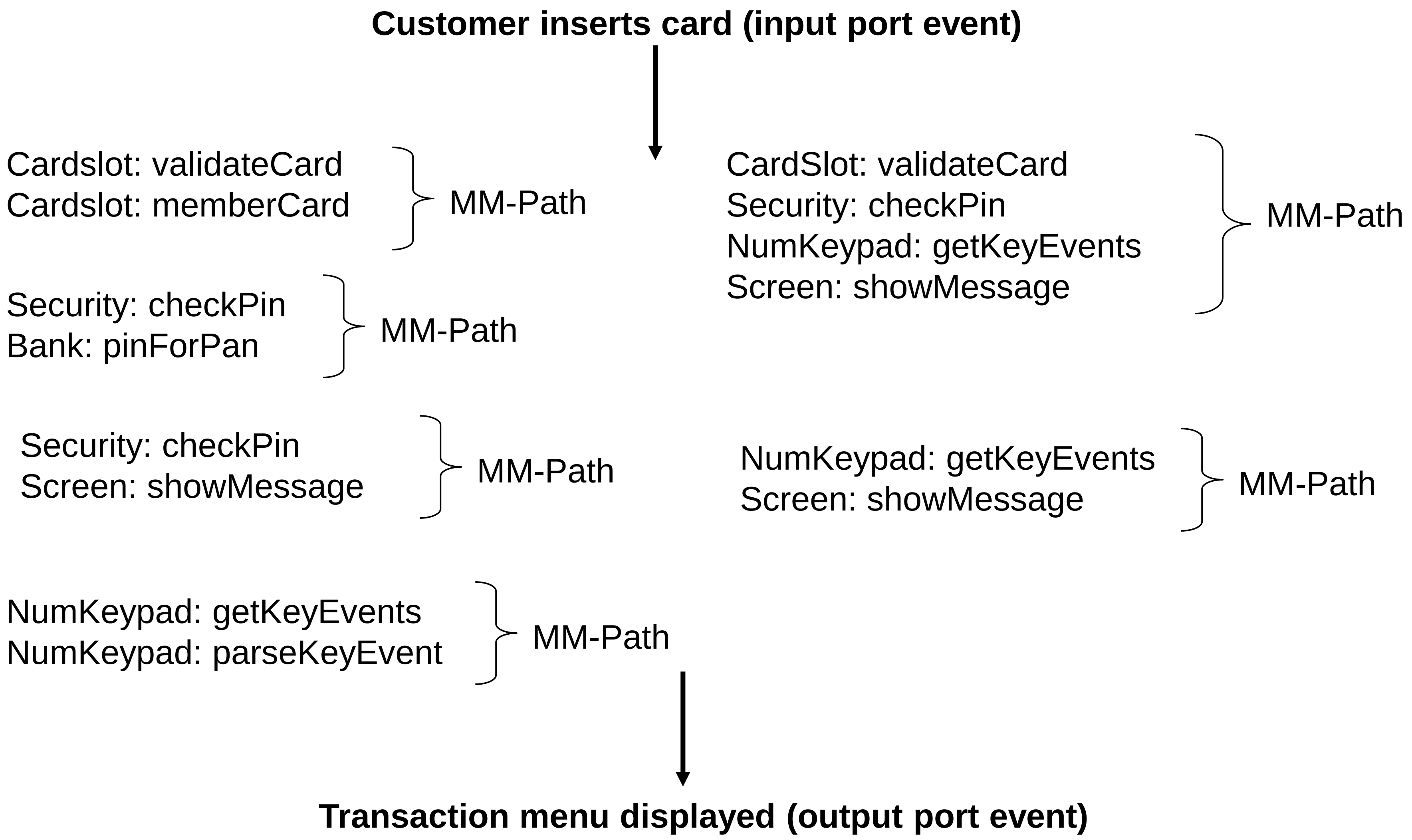
Source: ”Object-Oriented Integration Testing”, P.C. Jorgensen & C. Erickson, *Comm. ACM,* Sept. 1994

OO Integration Strategy

- Form groups of classes based on functionality.
- Initiate selected input port events .
- Track the resulting MM-Path through to message quiescence and an output port event.

Source: ”Object-Oriented Integration Testing”, P.C. Jorgensen & C. Erickson, *Comm. ACM,* Sept. 1994

OO Integration Example



Customer inserts card (input port event)

Cardslot: validateCard
Cardslot: memberCard

MM-Path

Security: checkPin
Bank: pinForPan

MM-Path

Security: checkPin
Screen: showMessage

MM-Path

NumKeypad: getKeyEvents
NumKeypad: parseKeyEvent

MM-Path

CardSlot: validateCard
Security: checkPin
NumKeypad: getKeyEvents
Screen: showMessage

MM-Path

NumKeypad: getKeyEvents
Screen: showMessage

MM-Path

Transaction menu displayed (output port event)

OO System Testing

Base it on use cases.

- Where to get them:
  - Get them from the designers.
  - Write your own.
- They are very valuable in testing.
- Each use case becomes one or more test cases.

Use Cases

- For a long time, in both object-oriented and traditional software development, people have used typical interactions between a user and the system t help them understand requirements.
- In object-oriented systems, the visibility of “use cases” has been raised .
- They have become a primary element in object oriented project development.

Use Cases (continued)

- Definition: A set of scenarios tied together by a common goal.
- Scenario: A sequence of steps describing an interaction between a user an a system.
- Caution: Use case methodology involves the term “actor”, which just means a “user with a specified role”.

Source: UML Distilled, Martin Fowler

Use Case - Example

Buy a Product

1. Customer looks through catalog and selects item(s) to buy.
2. Customer goes to “checkout”.
3. Customer fills in shipping information.
4. System presents price (including shipping).
5. Customer fills in credit card information.
6. System checks credit card information.
7. System authorizes transaction.
8. System confirms sale.
9. System sends confirmation email to customer.

Alternative 1: Authorization failure.
At step 7, system fails to authorize credit card purchase.
Customer is allowed to re-enter credit card information.

Alternative 2: Regular Customer
3a. System displays current shipping information, email, and last four digits of credit card information.
3b. Customer may accept or override default information.

Source: UML Distilled, Martin Fowler

Use Case to Test Cases

Test cases come directly from the use case:

□ **TC 1:** New customer; straight through steps 1 through 9.

□ **TC2:** Alternative 1 – Re-entry successful first retry.

□ **TC3:** Alternative 1 – Re-entry unsuccessful more than maximum number of allowable times.

□ **TC4:** Alternative 2 – Customer accepts default information.

□ **TC5:** Alternative 2 - Customer overrides default information.

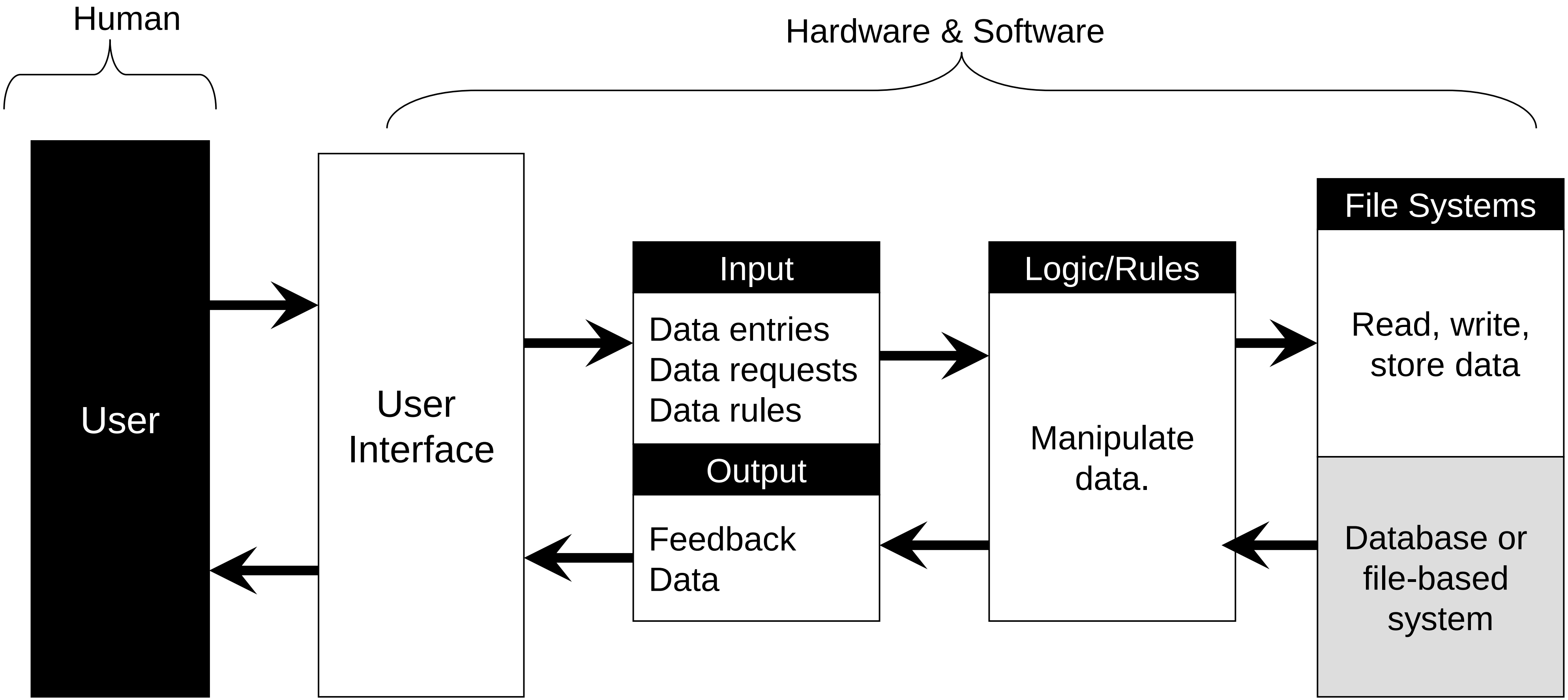
Scenarios & alternatives

Web Testing Considerations

- Many traditional software testing practices can be applied
- Technical issues that are specific to Web applications need to be considered.

The Generic Model

How Humans Interact With Computers

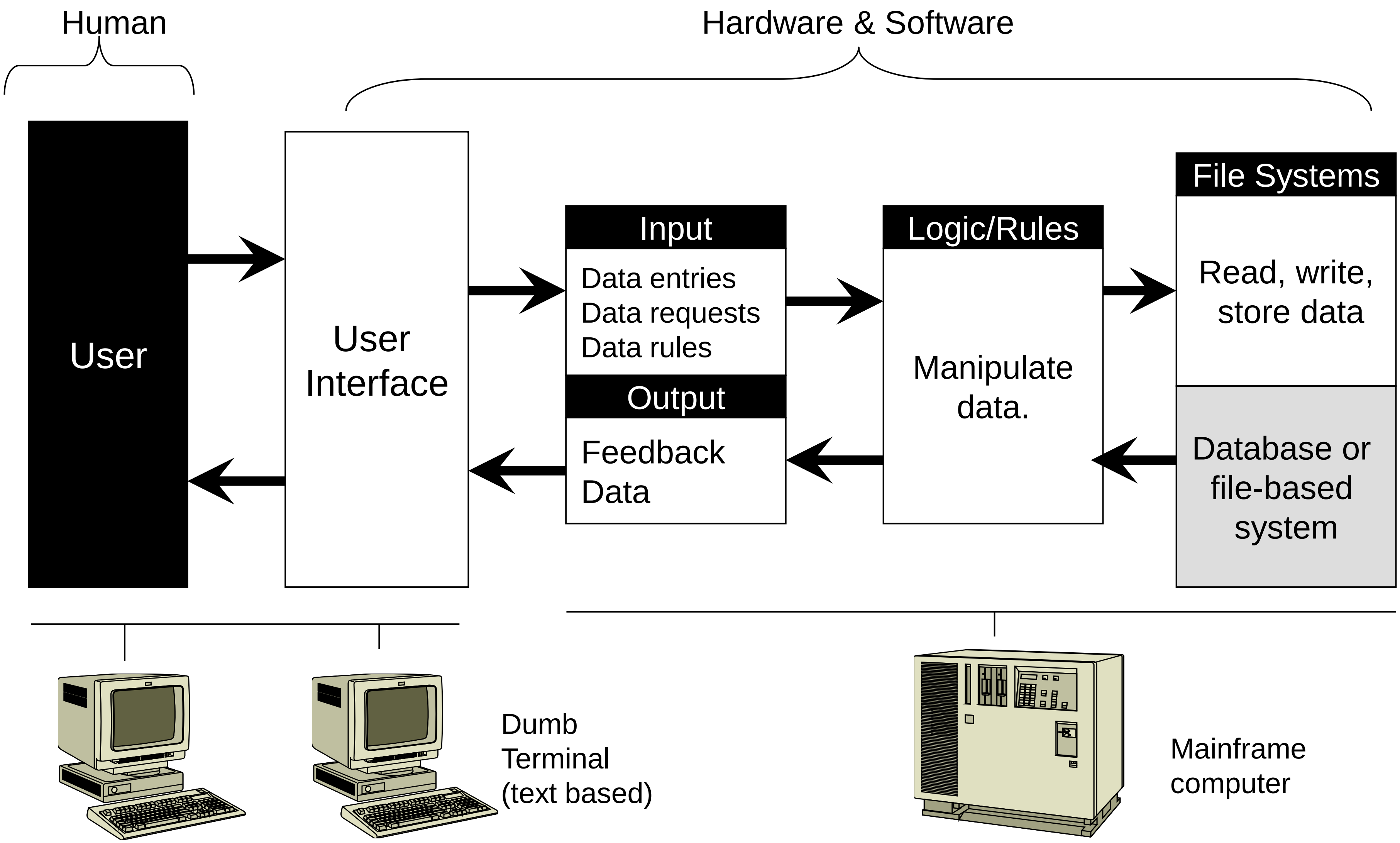


Source: Testing Application on the Web, Hung Nguyen et al

Applications

- Mainframe systems
- Desktop PC
- Client-server systems
- Web-based System

Mainframe System Model



Human

Hardware & Software

User

User Interface

Input

Data entries
Data requests
Data rules

Output

Feedback Data

Logic/Rules

Manipulate data.

File Systems

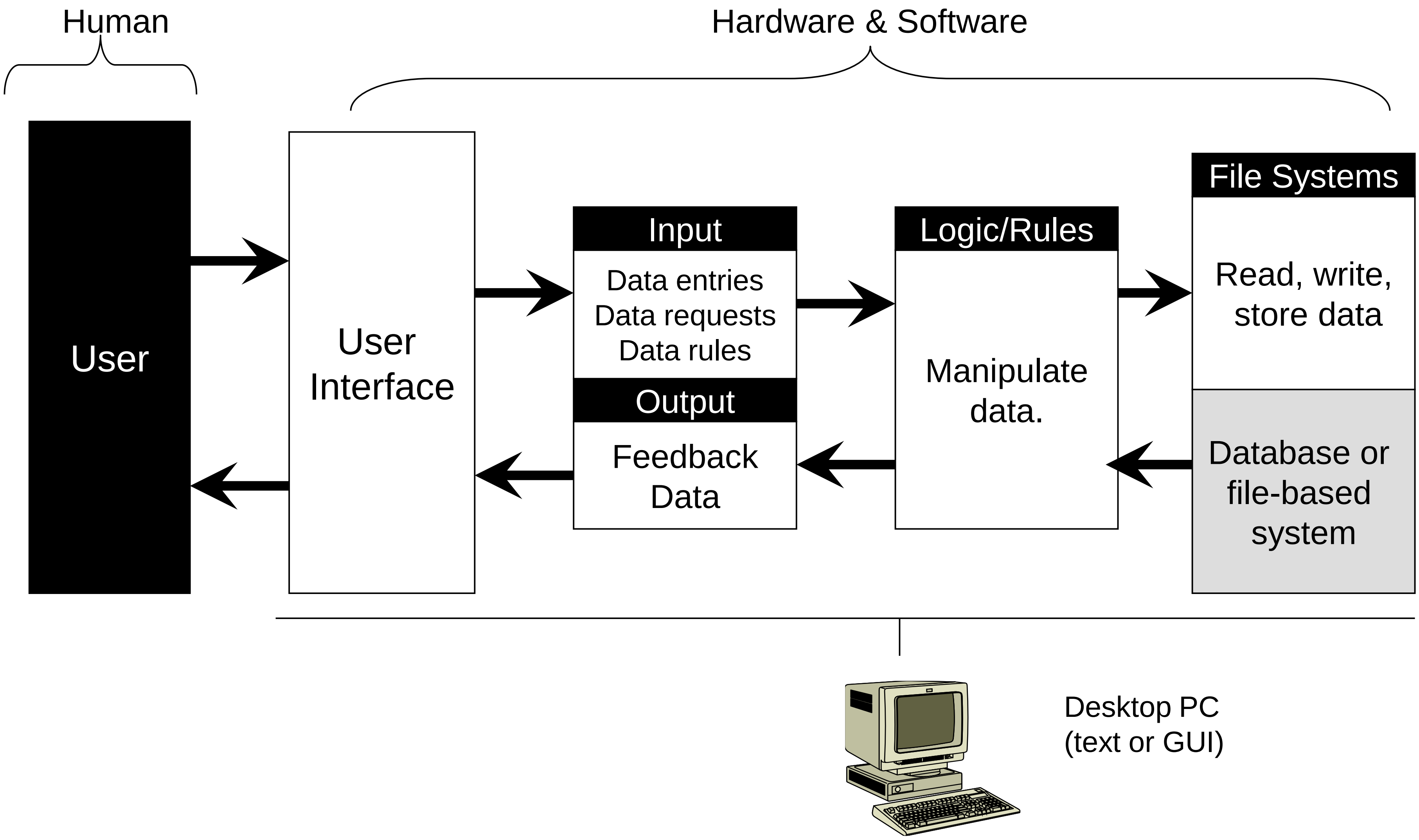
Read, write, store data

Database or file-based system

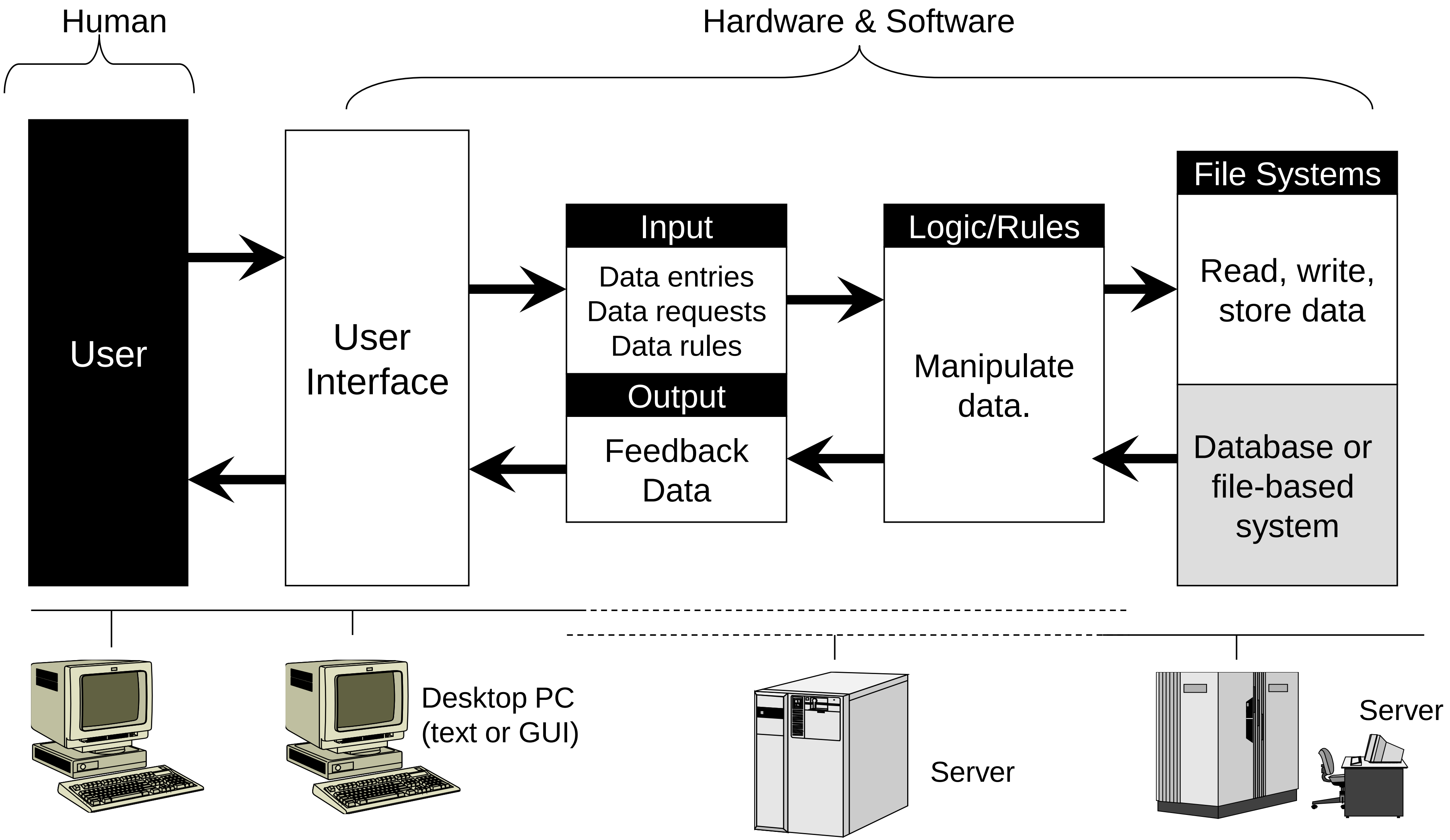
Dumb Terminal (text based)

Mainframe computer

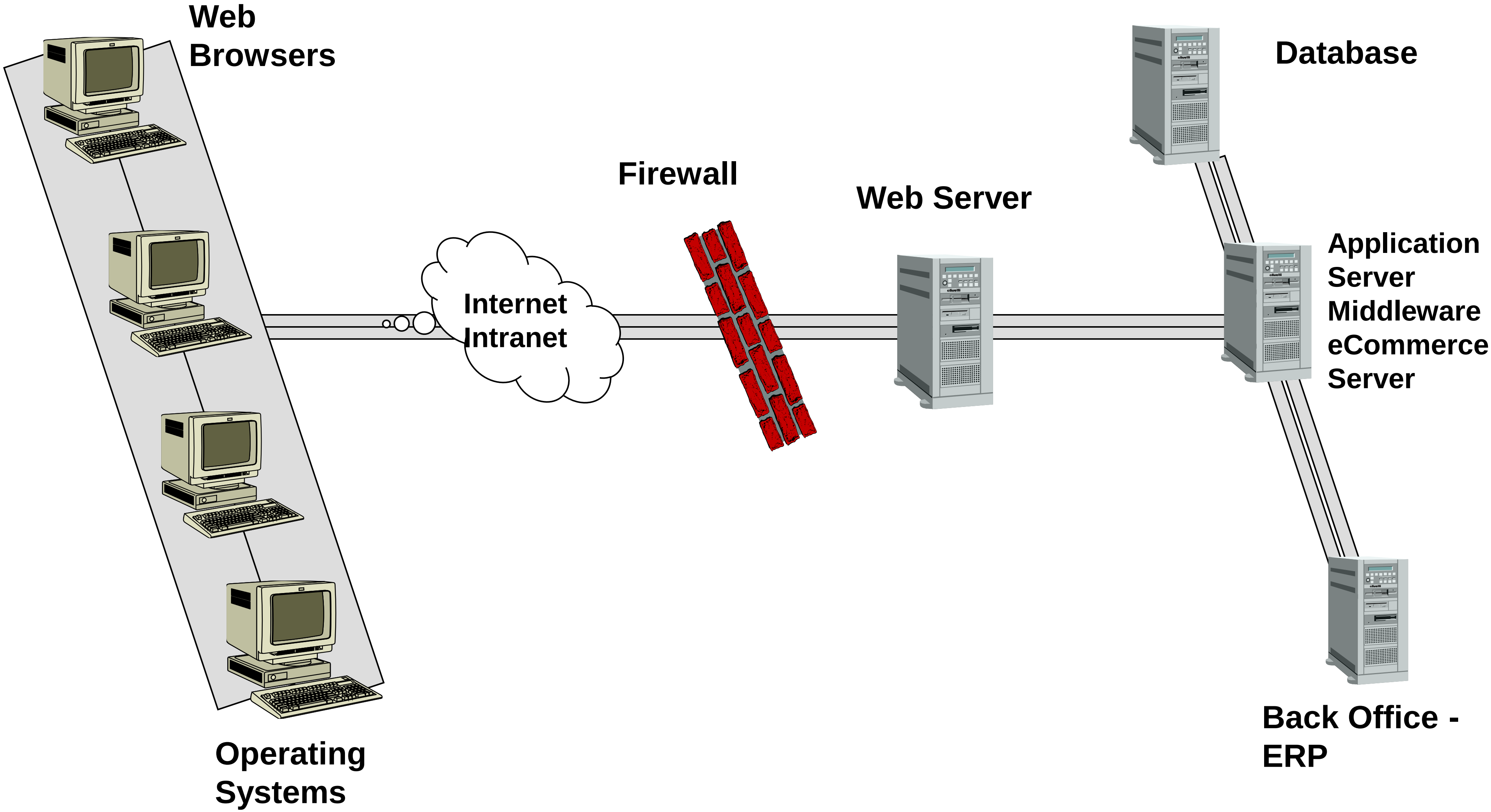
Desktop PC System Model



Client-Server System Model



Web-Based System Model



Client-Server Model

- Require a network and at least two machines:
  - Network
  - A *client* computer
  - A *server* computer
- Client
  - User interface (UI)
  - Request services from other programs
- Server
  - Receives requests from the client
  - Manipulates data
  - Sends it to client
- Web systems are built on a client-server architecture

Client-Server Model (continued)

- Not as neatly segmented as a mainframe or desktop.
- Either the client or the server can handle some of the processing.
- Server-side processing can be divided between multiple physical boxes.

Client-side Applications

- **Data-access driven**
- **Enable users to:**
  - Send input data
  - Receive output data
  - Interact with the back-end

- **Applications are platform specific**
  - Win-16
  - Win-32
  - Solaris
  - Mac
  - Unix
  - Linux

Web-based Systems

- Also data access driven
- Web-based client is operating within the Web browser’s environment
- Browsers consist of operating system-specific software running on a client computer
- Renders HTML & active contents to display web pages.
- Rendering engines and interpreters to translate and format HTML content.
  - Incompatibility issues

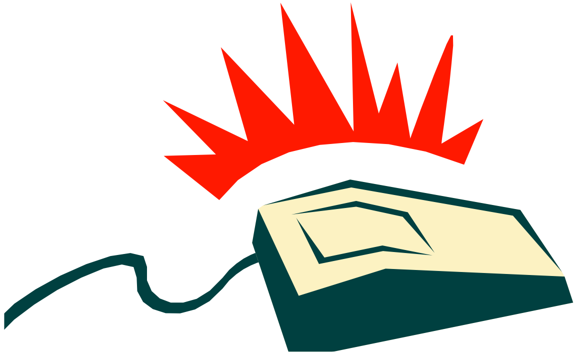
Source: Testing Applications on the Web, Hung Nguyen et al.

New Types of Clients

- Smaller than desktop versions.
- May be battery powered.
- Mobile devices
  - PDA
  - Smart phones
  - Picture phones
- Another class of client computers

Event Driven

- Inputs are driven by events
  - Clicks
    - Single
    - Double
  - Mouse movements
  - Keys
  - Input of data
- Depending on the type of event, certain procedures or functions in an application will be executed.
- Event-handling code.



Gray Box Testing

- Incorporates elements of both black box and white box testing.
- Uses both structural and functional approaches to testing.
- In test design, considers:
  - Interoperability of system components
- Web Systems: Numerous components
- Must be tested in the context of system design to evaluate:
  - Functionality
  - Compatibility

Source: Testing Applications on the Web, Hung Nguyen et al.

Gray Box Testing (continued)

- Methods and tools derived from knowledge of:
  - Application internals
  - The environment with which it interacts
- Knowledge of the designer’s intent used in;
  - Test design
  - Bug analysis
- Improve probability of finding errors.

Source: Testing Applications on the Web, Hung Nguyen et al.

Web Testing Challenge

- Main challenge: Learn the associated technologies in order to have a better command over the environment.
  - Web technologies
  - Interoperability
  - Web systems as a whole
- Web tester must be familiar with:
  - Test types
  - Testing issues
  - Common software issues
  - Quality issues specific to Web applications

Source: Testing Applications on the Web, Hung Nguyen et al.

Web Testing Considerations

Key areas beyond traditional testing:

- Web UI implementation
- Server & client installation
- Web-based help
- Configuration and compatibility
- Database
- Security
- Performance, load, and stress

Source: Testing Applications on the Web, Hung Nguyen et al.

Real Time Systems

- One in which correctness depends upon when processing is done as well as correctness of algorithms and coding.
- Not necessarily the same as “fast”
  - Some real time systems can be rather “slow”.

Real Time Systems - Example

- A control system sends out control signals to hardware every two seconds.
- If the process crosses two second boundaries, incorrect results will be obtained in the controlled hardware.
- Real time systems are often periodic:
  - Do something every two seconds
  - Do something every hour on the hour
  - Do something everyday at midnight

Guidelines for Testing Real Time Systems

- Do proper unit and integration testing.
- Don’t do coding “tricks” in the name of performance and real-time.
- Modularize the real-time code.
- Do static behavior testing of all functionality before any dynamic tests.
- Do early stress testing to find the easy synchronization and timing bugs
- Use external environmental simulators for rigorous testing of synchronization and timing.
- Do an inverse user profile to test low probability paths.

Source: Quality Techniques Newsletter, Sept., 2003

Reporting Bugs

- All bugs found in testing must be reported in writing.
- The purpose of a bug report is to provide enough information to allow the bug to be fixed.
- Write problem reports immediately.

An Effective Bug Report

- Explain how to reproduce the problem.
- Analyze the error so you can describe it in a minimum number of steps.
- Problem reports should be:
  - Accurate
  - Complete
    - Include as much information as possible
    - Attachments
  - Easy to understand
    - Simple
  - Non-antagonistic.

Content of a Problem Report

- Problem report number
- Software version identifier.
- Module or functional area.
- Summary description of problem.
- Detailed description of problem.
- Severity
- Priority
- Status of the problem
- Problem type
- Date; name of person reporting the problem.
- Attachments

Content of a Problem Report

- Problem report number
- Software version identifier.
- Module or functional area.
- Summary description of problem.
- Detailed description of problem.
- Severity
- Priority
- Status of the problem
- Problem type
- Date; name of person reporting the problem.
- Attachments

Severity vs. Priority

- Classification of severity
  - Determined by the effects of the bug.
    - Crashes the system every time
    - Cosmetic on a screen (misspelling)
  - Not determined by how hard it is to fix.
- Priority
  - Assigned by management
    - Customer needs
    - Management goals
    - Product plans

Severity

Priority

Data to Add When the Problem Is Fixed

- Description of what was done to fix the problem.
- By whom.
- When was the fix implemented.
- What version is the fix in.
- At what stage in the software development process was the problem introduced.
- Verified by whom & when.

Bug Tracking Tool

- Use one!
- It is absolutely essential.
- TRACK by Softool
- ClearQuest by Rational
- DDTS
- Seapine Software
- Bugzilla
- Borland StarTeam

Tests Are Software, Too

- Creating testware is a very similar to process to creating code:
  - It’s planned
  - It’s designed
  - It’s implemented
  - It’s maintained
- Is there any reason to assume that testware is bug free?
- Must debug the test plans, cases, and procedures.

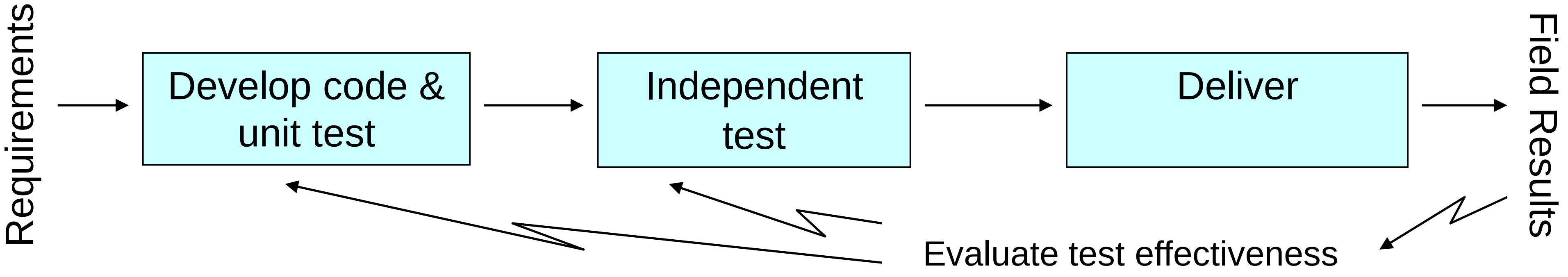
Test Improvement

Structural Testing

- By developers
- Test design
- Analyze code coverage
- Analyze complexity

Functional Testing

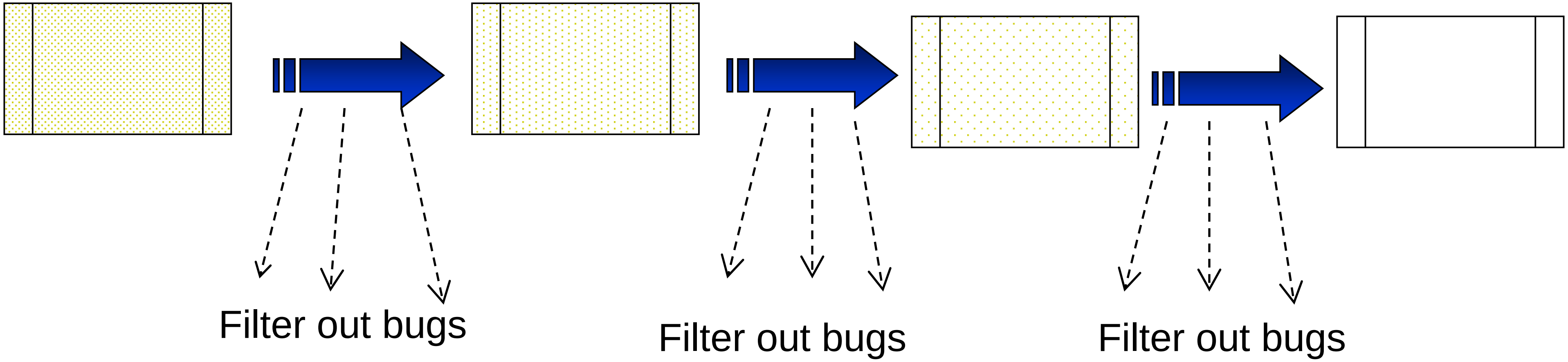
- By independent test group
- Analyze requirements coverage
- Testing for stress, boundary values, exception handling



Test Improvement (continued)

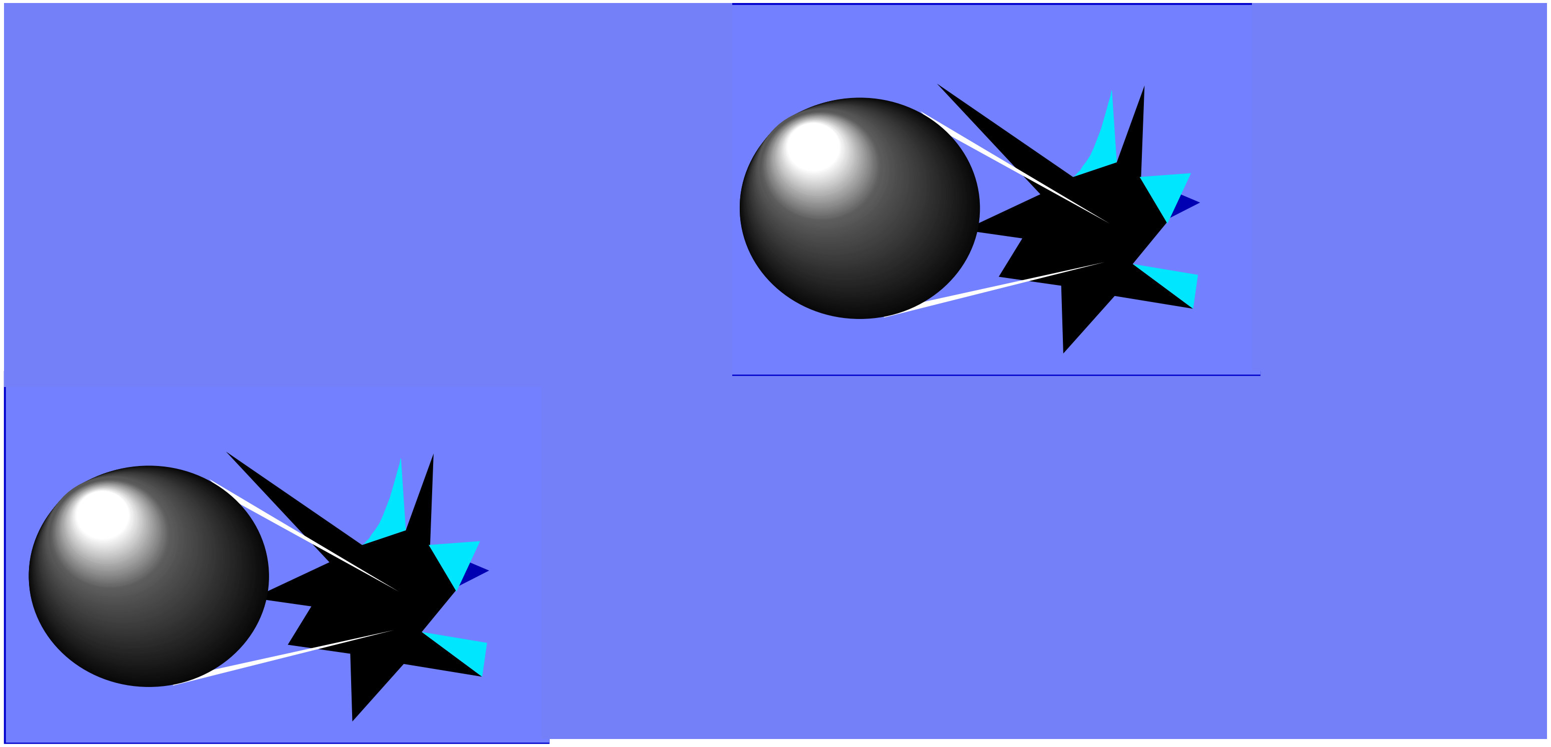
Look At All Stages of Testing

- Inspections (peer reviews)
- Unit testing
- Integration testing
- System Testing
- Acceptance testing



Escaped Bugs

Plugging The Holes



Test Improvement (continued)

- For every escaped bug, do a root cause analysis.
- Determine why it was not caught by the testing.
- Write a test case(s) to catch that and similar bugs, and include the test case(s) in the test suite.
- Net effect: Testing becomes more “ bullet-proof”, and fewer defects are passed to the next life cycle phase.

Test Automation

Test Design

Test Execution

Automation - Test Design

- Requires a tool that allows modeling the software’s behavior.
- Large investment in building the behavior model.
- Very powerful once it’s done.

Automation – Test Execution

- Most applicable in regression testing.
  - Run the set of tests over and over on new versions.
  - Repetitive tasks.
- Must buy a tool.
- Requires tool set up effort and a tool administrator.
- Large productivity gains are possible.

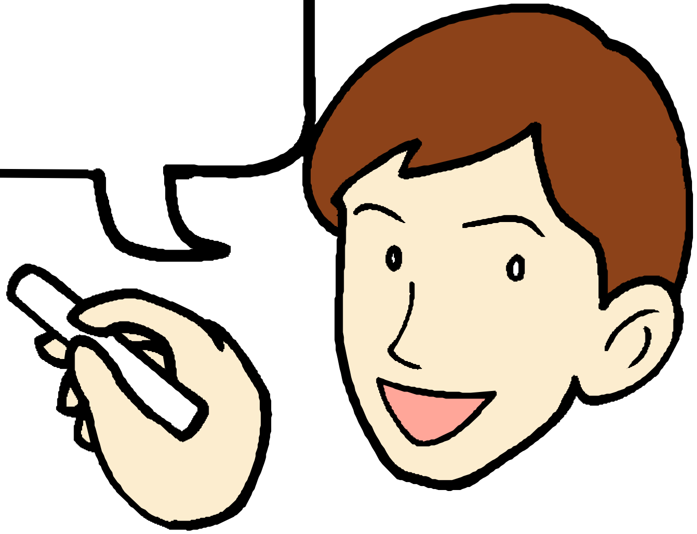
Test Automation Tools

In a Windows PC client – Unix server environment: Many tools are available.

In an embedded systems environment: Very few (almost none) tools are available.

Tool Availability

- Gartner / Dataquest says:
  - 45 Vendors
  - 104 Tools
- Stickyminds count:
  - 62 Vendors
  - 108 Tools



Can You Automate?

Do you:

□ HHave a testing process?
□ FFollow the testing process?
□ RRepeat the testing process?
□ HHave a training process?
□ WWork with a stable product with upgrades?

□ HHave a tool administrator available?
□ HHave realistic schedules?
□ HHave executive and grass roots commitment?
□ HHave a tool selection plan?
□ HHave a tool roll out plan?

Source: “The Past, Present, and Future of Test Automation Tools”, Greg Pope, Software Test Automation Conference, 9/27/02

Tool Selection Plan

- □ Create prioritized list of requirements.
- □ Agree on “Must Haves”.
- □ Identify and prioritize risks.
- □ Get input from all areas.
- □ Do an analytic evaluation of all products against selection criteria.
- □ Update the selection criteria as required.
- □ Do an in-house evaluation of “short list” products.
- □ Score short list contenders against selection criteria.

Source: “The Past, Present, and Future of Test Automation Tools”, Greg Pope, Software Test Automation Conference, 9/27/02

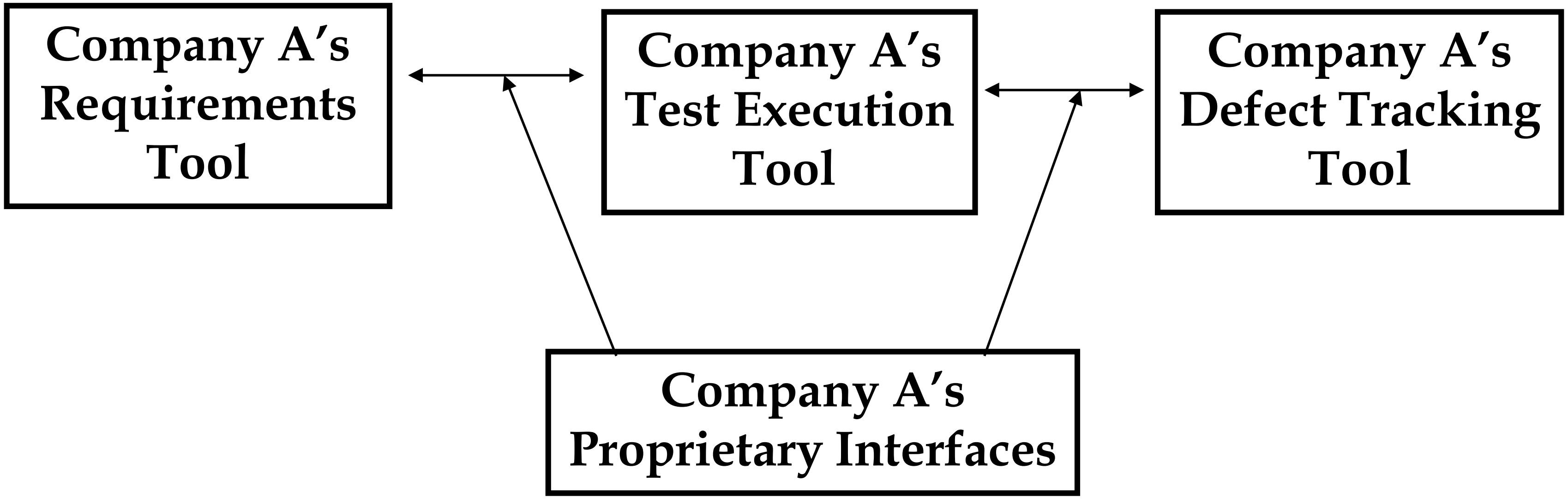
Tool Roll Out Plan

- Feasibility demonstration successfully completed
- Training plan
- Pilot project selected (small, non-critical)
- Success criteria agreed upon
- Implement training
- Implement pilot
- Evaluate pilot against success criteria
- Post mortem, tailor as required
- Go/No Go for general roll out

Source: “The Past, Present, and Future of Test Automation Tools”, Greg Pope, Software Test Automation Conference, 9/27/02

Tool Integration - Today

What we have today: Tool suites form a single vendor



Company A's Requirements Tool

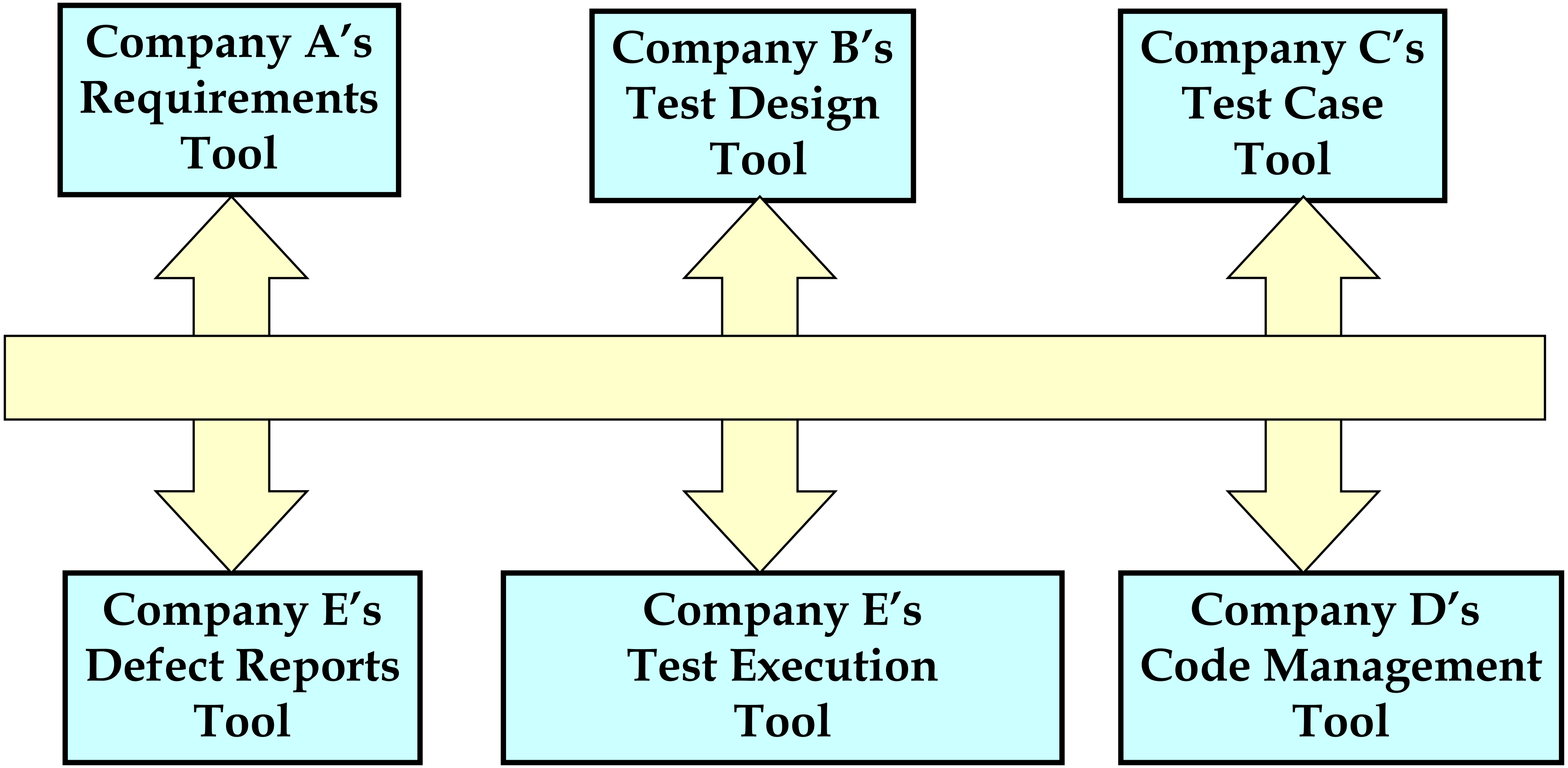
Company A's Test Execution Tool

Company A's Defect Tracking Tool

Company A's Proprietary Interfaces

Source: "The Past, Present, and Future of Test Automation Tools", Greg Pope, Software Test Automation Conference, 9/27/02

Tool Integration - Needed


Company A's Requirements Tool
Company B's Test Design Tool
Company C's Test Case Tool
Company E's Defect Reports Tool
Company E's Test Execution Tool
Company D's Code Management Tool

Source: "The Past, Present, and Future of Test Automation Tools", Greg Pope, Software Test Automation Conference, 9/27/02

The End