Introduction to Software Testing

Chapter 1

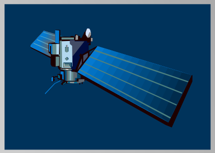
Intoduction

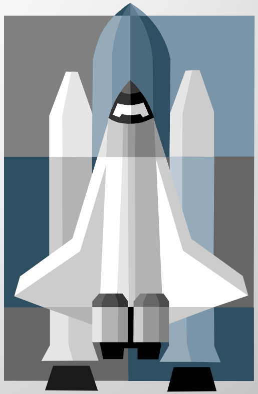
Paul Ammann & Jeff Offutt

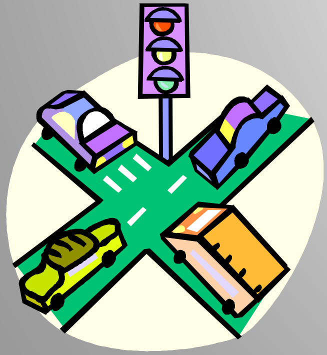
Updated by Sunae Shin

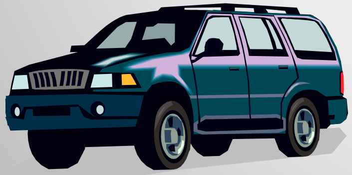
Why Testing?

Software is a Skin that Surrounds Our Civilization









Testing in the 21st Century

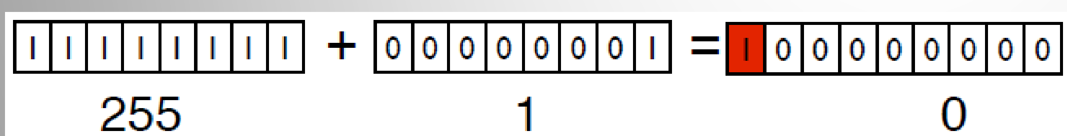
- Software defines behavior
  - network routers, finance, switching networks, other infrastructure
- Today's software market :
  - is much bigger
  - is more competitive
  - has more users

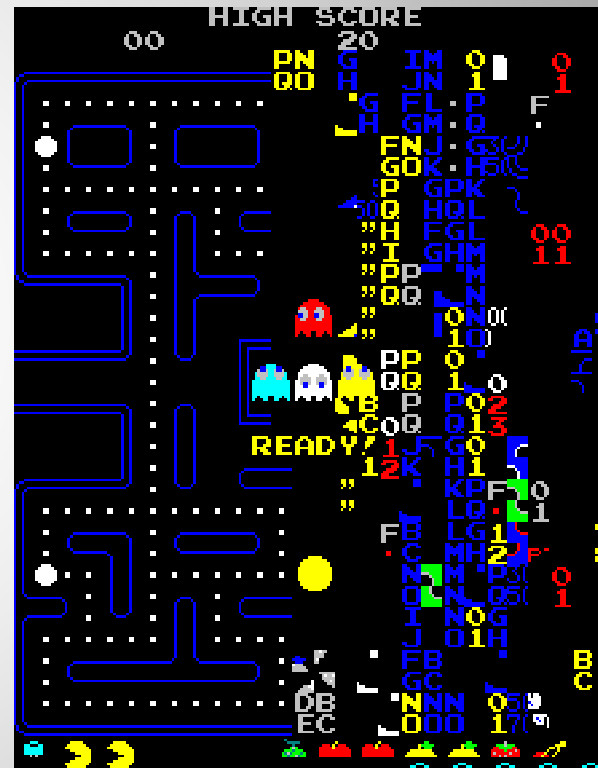
Industry is going through a revolution in what testing means to the success of software products

- Embedded Control Applications
  - airplanes, air traffic control
  - spaceships
  - watches
  - ovens
  - remote controllers
  - PDAs
  - memory seats
  - DVD players
  - garage door openers
  - cell phones

Spectacular Software Failures

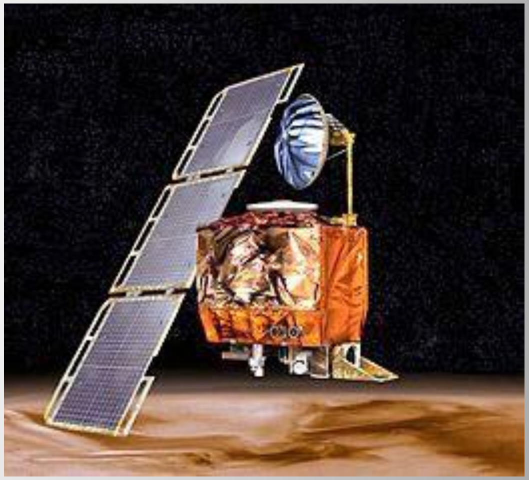
- Pac-Man (1980)
  - Should always have no ending
  - Has “Split Screen” at level 256
- Cause: Integer overflow
  - 8 bits: maximum representable value





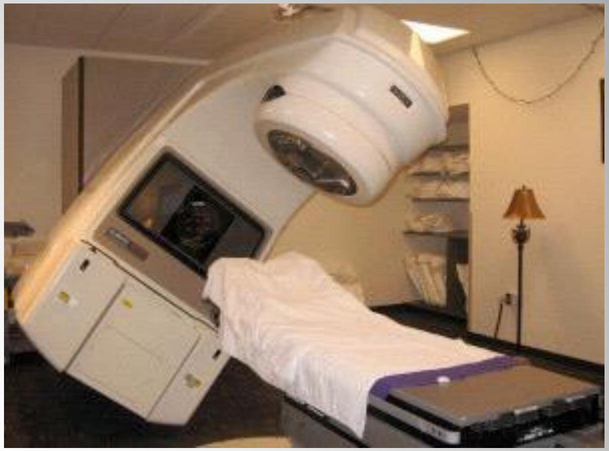
Spectacular Software Failures

- Mars Climate Orbiter (1998)
  - Sent to Mars to relay signal from Mars Lander
  - Smashed to the planet



- Cause: Failing to convert between different metric standards
  - Software that calculated the total impulse presented results in pound-seconds
  - The system using these results expected its inputs to be in newton-seconds

Spectacular Software Failures



- THERAC-25 radiation machine (1985)
  - Poor testing of safety-critical software can cost *lives* : 3 patients were killed

- Ariane 5 explosion (1996)
  - On June 4, 1996 an unmanned Ariane 5 rocket launched by the European Space Agency exploded just 37 seconds
    - Cost: $370 Millions
  - Cause: arithmetic overflow
    - Data conversion from a 64-bit floating point to 16-bit signed integer value caused an exception
    - The software from Ariane 4 was re-used for Ariane 5 without retesting

Northeast Blackout of 2003

508 generating units and 256 power plants shut down

Affected 10 million people in Ontario, Canada

Affected 40 million people in 8 US states

Financial losses of $6 Billion USD

The alarm system in the energy management system failed due to a software error and operators were not informed of the power overload in the system

Software Failures

- Only 32% of software projects are considered successful

(full featured, on time, on budget)

- On average, 1-5 bugs per KLOC (thousand lines of code)
  - In mature software (more than 10 bugs in prototypes)

- Software failures cost the US economy $59.5 billion dollars every year [NIST 2002 Report]

What Does This Mean?

Software testing is getting more important

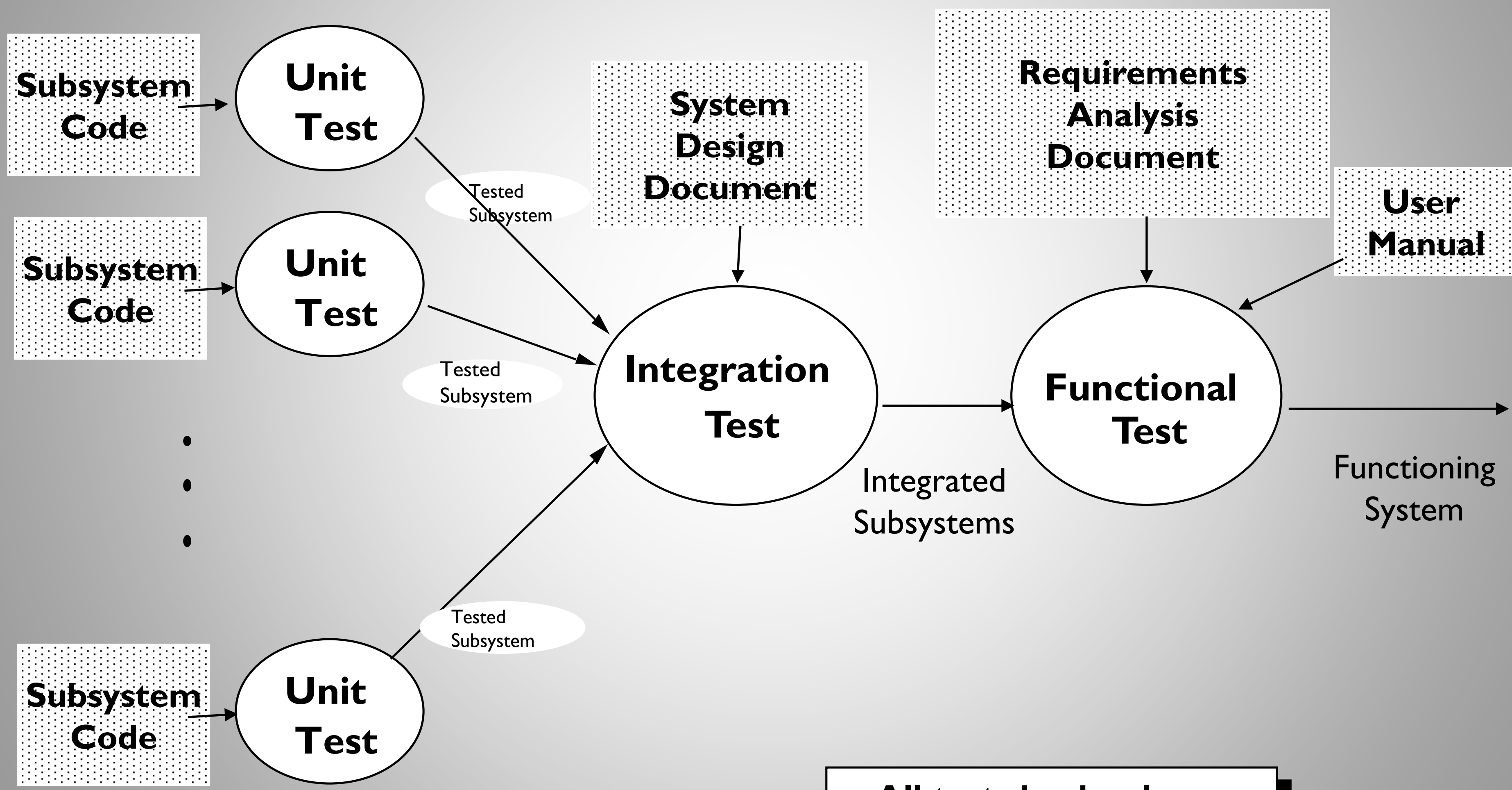
What are we trying to do when we test ?

What are our goals ?

Testing Levels

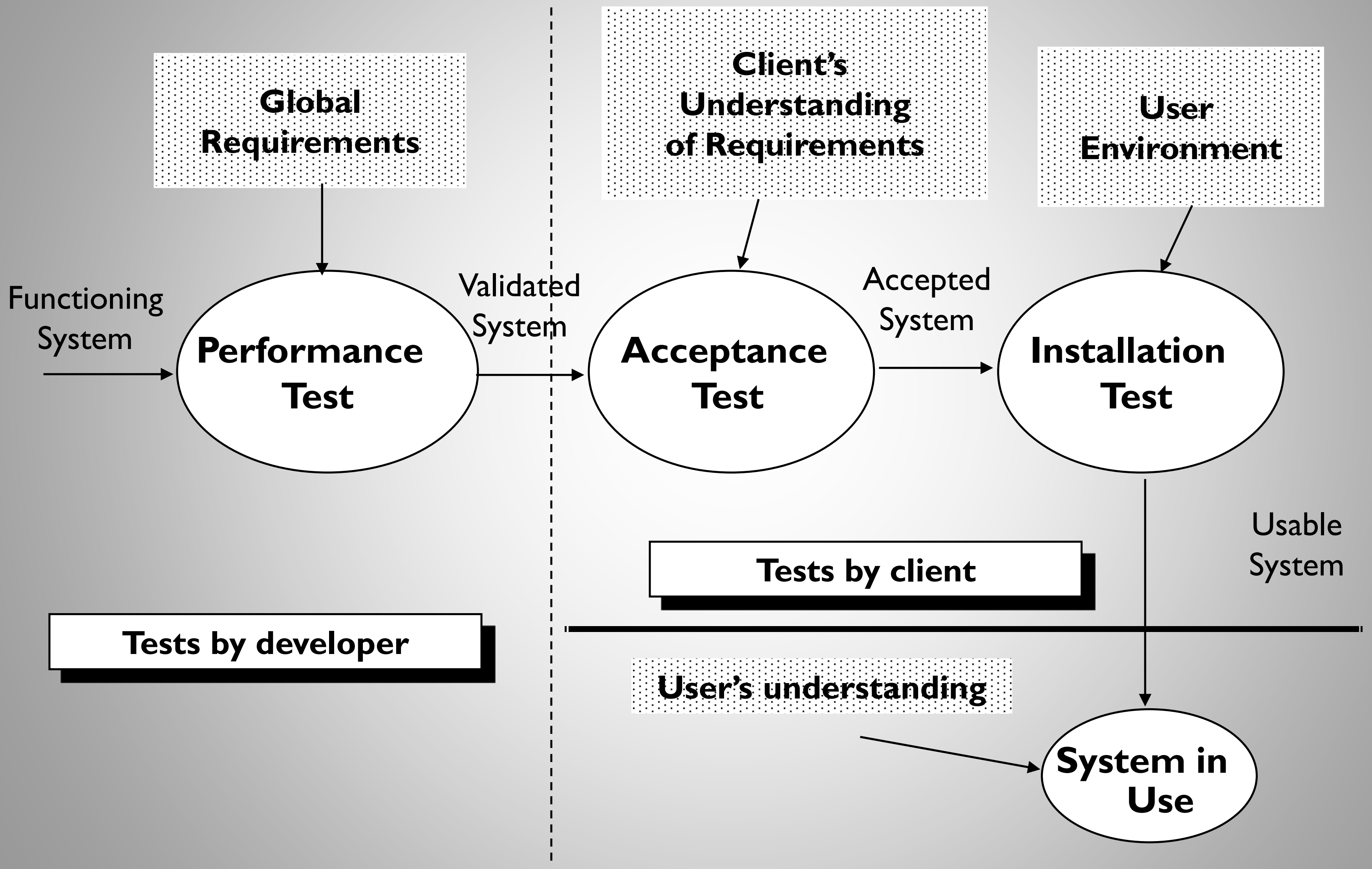
- Testing Levels based on Software Activities
- Testing Levels based on Test Process Maturity

Testing Levels based on Software Activities



All tests by developer

Testing Levels (contd)



Type of Testing

- **Unit** Testing:
  – Individual *subsystem*
  – Carried out by developers
  – Goal: Confirm that subsystems is correctly coded and carries out the intended functionality

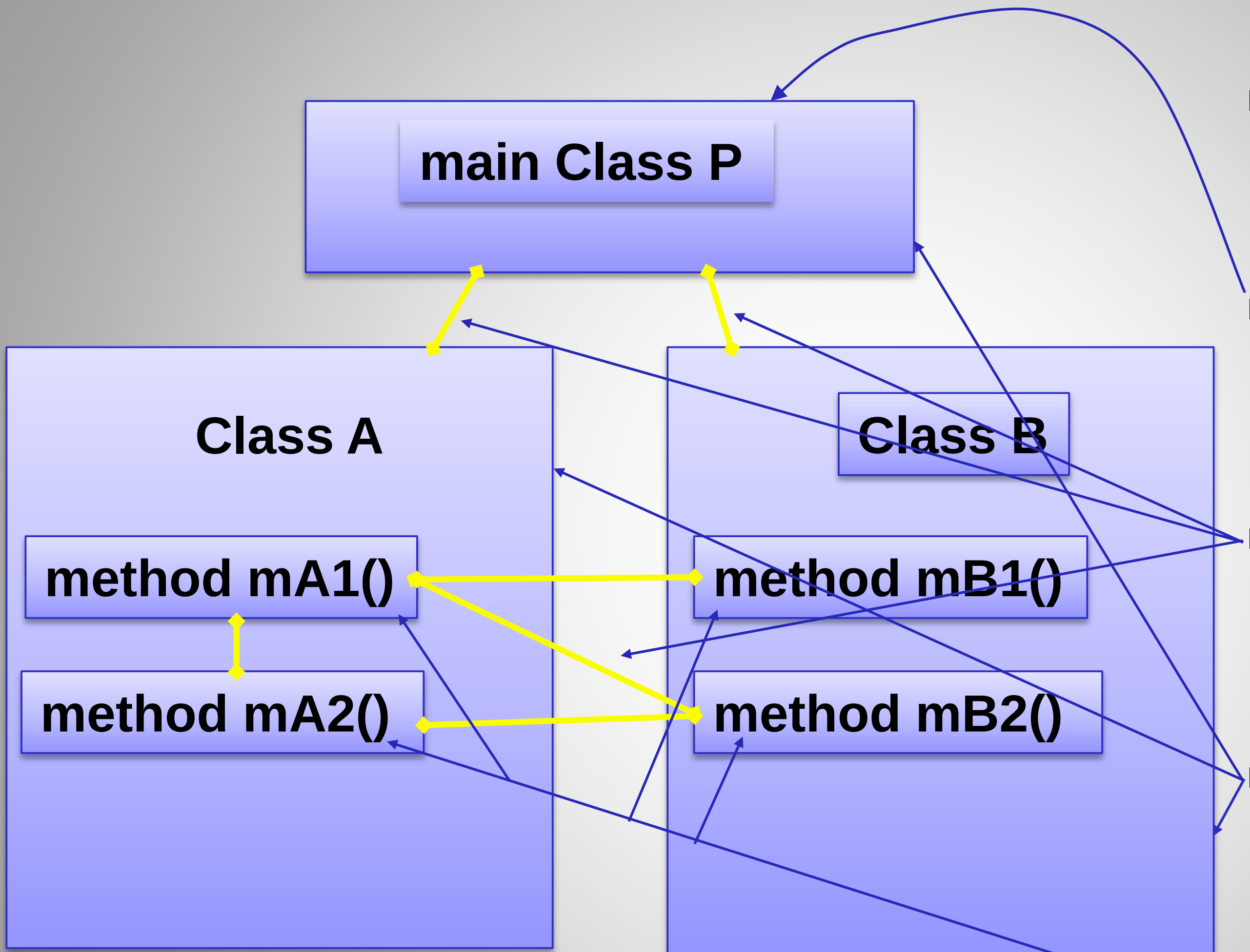
- **Integration** Testing:
  – Groups of subsystems (collection of classes) and eventually the entire system
  – Carried out by developers
  – Goal: Test the *interface* among the subsystem

Type of Testing

- **System** Testing:
  - The entire system
  - Carried out by developers
  - Goal: Determine if the system meets the *requirements* (functional and *global*)

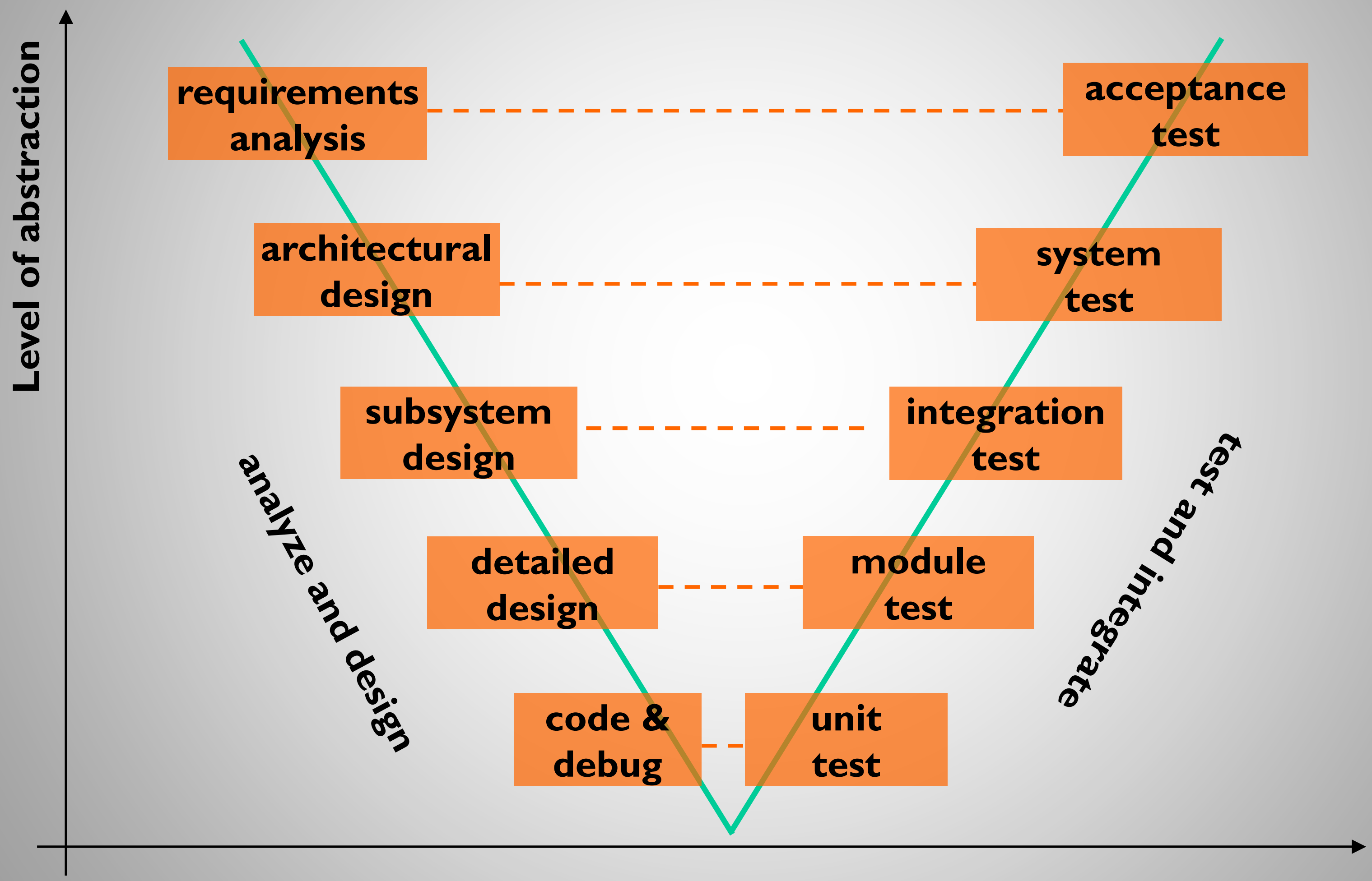
- **Acceptance** Testing:
  - Evaluates the system delivered by developers
  - Carried out by the *client*. May involve executing typical transactions on site on a trial basis
  - Goal: Demonstrate that the system meets customer *requirement*s and is ready to use

Testing Levels



- **Acceptance testing : Is the software acceptable to the user?**
- **System testing : Test the overall functionality of the system**
- **Integration testing : Test how modules interact with each other**
- **Module testing (developer testing) : Test each class, file, module, component**
- **Unit testing (developer testing) : Test each unit (method) individually**

Testing Levels – The "V Model"



Testing Levels Based on Test Process Maturity

- Level 0 : There's no difference between testing and debugging
- Level 1 : The purpose of testing is to show correctness
- Level 2 : The purpose of testing is to show that the software doesn't work
- Level 3 : The purpose of testing is not to prove anything specific, but to reduce the risk of using the software
- Level 4 : Testing is a mental discipline that helps all IT professionals develop higher quality software

Level 0 Thinking

- Testing is the same as debugging
- Does <u>not</u> distinguish between incorrect behavior and mistakes in the program
- Does <u>not</u> help develop software that is reliable or safe

This is what we teach undergraduate CS majors

Level 1 Thinking

- Purpose is to show correctness
- Correctness is impossible to achieve
- What do we know if no failures?
  - Good software or bad tests?
- Test engineers have no
  - Strict goal, Real stopping rule, Formal test technique
- Test managers are powerless
  - If a development manager asks how much testing remains to be done, the test manager has no way to answer the question

This is what hardware engineers often expect

Level 2 Thinking

- Purpose is to show failures
- Looking for failures is a negative activity
- Puts testers and developers into an adversarial relationship
- What if there are no failures?

This describes most software companies.

Level 3 Thinking

- Testing can only show the presence of failures
  - But, not the absence
- Whenever we use software, we incur some risk
- Risk may be small and consequences unimportant
- Risk may be great and consequences catastrophic
- Testers and developers cooperate to reduce risk

This describes a few "enlightened" software companies

Level 4 Thinking

A mental discipline that increases quality

- Testing is only one way to increase quality
- Test engineers can become technical leaders of the project
- Primary responsibility to measure and improve software quality
- Their expertise should help the developers

This is the way "traditional" engineering works

Where Are You?

Are you at level 0, 1, or 2 ?

Is your organization at work at level 0, 1, or 2 ? or 3?

We hope to teach you to become "change agents" in your workplace …

Advocates for level 4 thinking

Automation of Test Activities

- Software testing is expensive and labor intensive
  - Requires up to 50% of software development costs
  - Even more for safety-critical applications

- One of the goals of software testing is to automate as much as possible
  - Significantly reducing its cost, minimizing human error, and making regression testing easier

Terminologies

Test Case

- Test Case Values : The values that directly satisfy one test requirement

- Expected Results : The result that will be produced when executing the test if the program satisfies it intended behavior

White-box and Black-box Testing

- Black-box testing : Deriving tests from external descriptions of the software, including specifications, requirements, and design

- White-box testing : Deriving tests from the source code internals of the software, specifically including branches, individual conditions, and statements

Example of White-box Testing

- *Linear code sequences*

```
begin
  int x, y, p;
  input (x, y);
  if(x<0)
    p=g(y);
  else
    p=g(y*y);
end
```

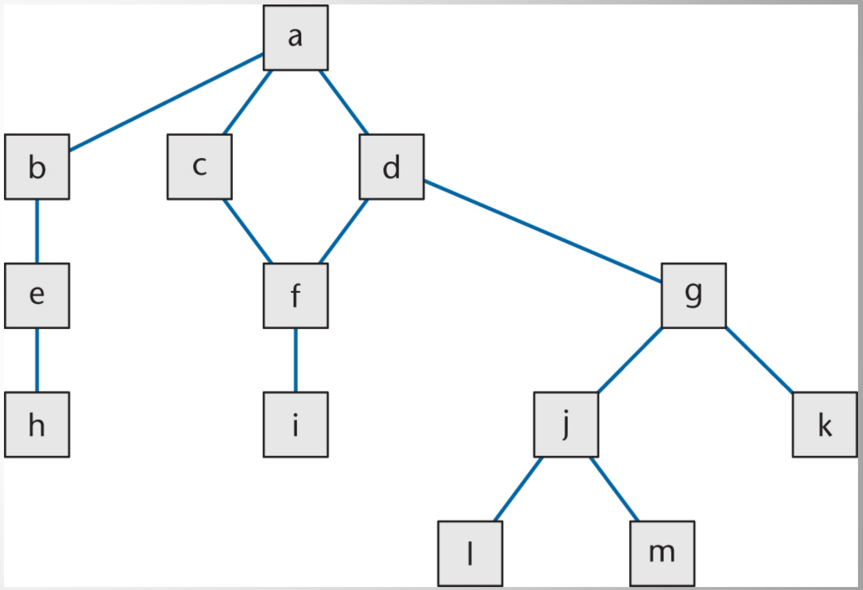
$$T = \left\{ \begin{array}{lll} t_1 : & < x = -5 & y = 2 > \\ t_2 : & < x = 9 & y = 2 > \end{array} \right\}$$

Top-Down and Bottom-Up Testing

- Top-Down Testing : Test the main procedure, then go down through procedures it calls, and so on
- Bottom-Up Testing : Test the leaves in the tree (procedures that make no calls), and move up to the root.
  - Each procedure is not tested until all of its children have been tested

Example: Top-down Integration

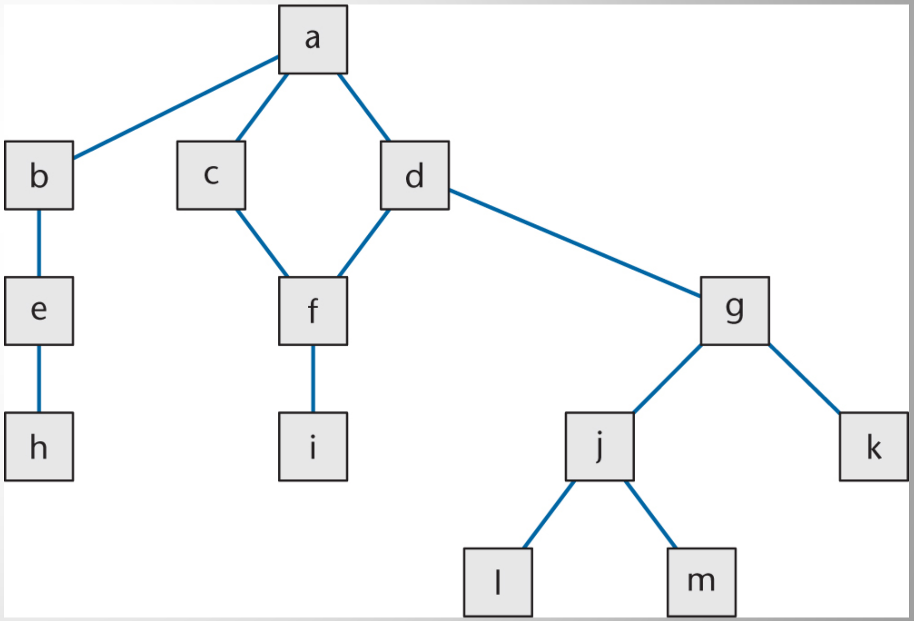
- If code artifact `mAbove` sends a message to artifact `mBelow`, then `mAbove` is implemented and integrated before `mBelow`
- One possible top-down ordering is
  - `a, b, c, d, e, f, g, h, i, j, k, l ,m`



Example: Bottom-up Integration

- If code artifact `mAbove` calls code artifact `mBelow`, then `mBelow` is implemented and integrated before `mAbove`
- One possible bottom-up ordering is

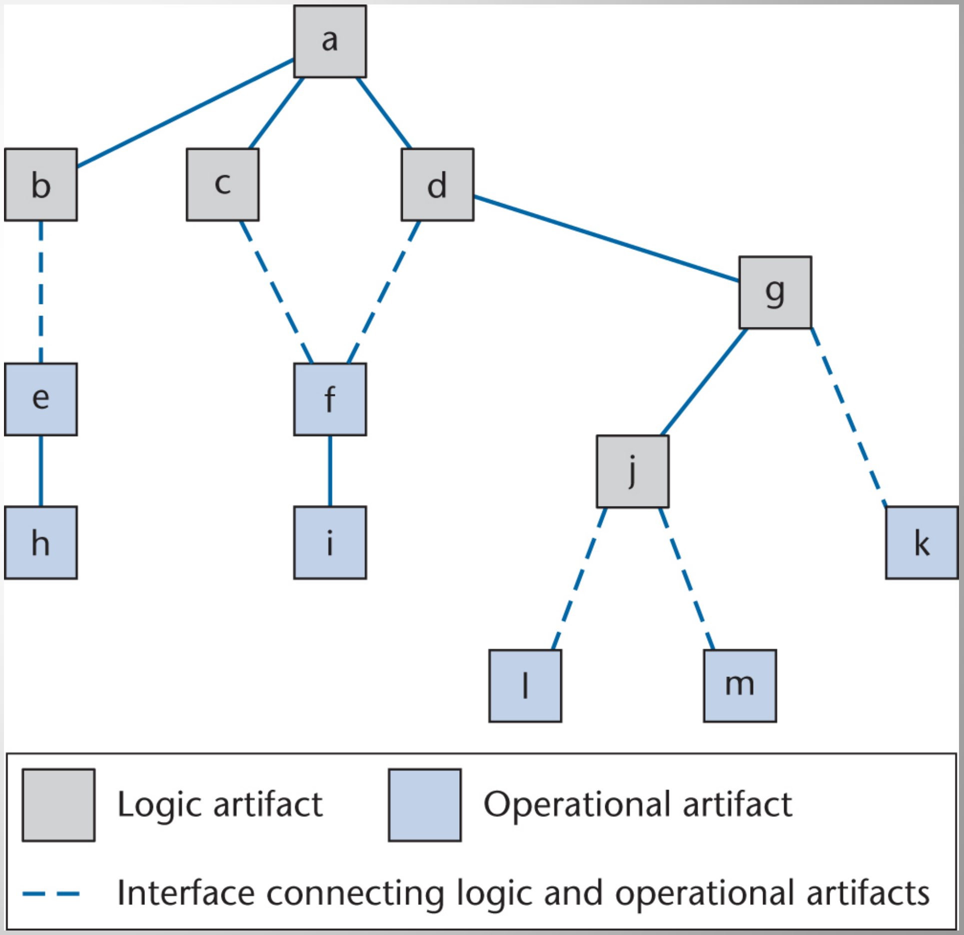
  `l, m, h, i, j, k, e, f, g, b, c, d, a`



Example: Sandwich Integration

- Logic artifacts: incorporate the decision-making flow of control aspects of the product
- Operational artifacts: perform the actual operations of the product

- Logic artifacts are integrated top-down
- Operational artifacts are integrated bottom-up
- Finally, the interfaces between the two groups are tested



Validation & Verification (*IEEE*)

- Validation : The process of evaluating software at the end of software development to ensure compliance with intended usage

- Verification : The process of determining whether the products of a given phase of the software development process fulfill the requirements established during the previous phase

IV&V stands for "*independent verification and validation*"

Software Faults, Errors & Failures

- Software Fault : A static defect in the software

- Software Failure : <u>External, incorrect behavior</u> with respect to the requirements or other description of the expected behavior

- Software Error : An <u>incorrect internal state</u> that is the manifestation of some fault

Fault and Failure Example

- A patient gives a doctor a list of symptoms
  - Failures
- The doctor tries to diagnose the root cause, the ailment
  - Fault
- The doctor may look for anomalous internal conditions (high blood pressure, irregular heartbeat, bacteria in the blood stream)
  - Errors

A Concrete Example

Fault: Should start searching at 0, not 1

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

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

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

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

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

Software Faults, Errors & Failures

Example

```
public int findLast (int[] x, int y) {
//Effects: If x==null throw NullPointerException
//   else return the index of the last element
//   in x that equals y.
//   If no such element exists, return -1
   for (int i=x.length-1; i > 0; i--)
   {
      if (x[i] == y)
      {
         return i;
      }
   }
   return -1;
}
```

(a) Identify the fault.
(b) If possible identify a test case that results in an error, but not a failure.

The for-loop should include the 0 index:

```
for (int i=x.length-1; i >= 0; i--) {
```

For an input where `y` *is not in* `x`*, the missing path (i.e. an incorrect PC on the final loop that is not taken) is an error, but there is no failure.*

Input:	*x = [2, 3, 5]; y = 7;*
Expected Output:	*-1*
Actual Output:	*-1*

Test Engineer

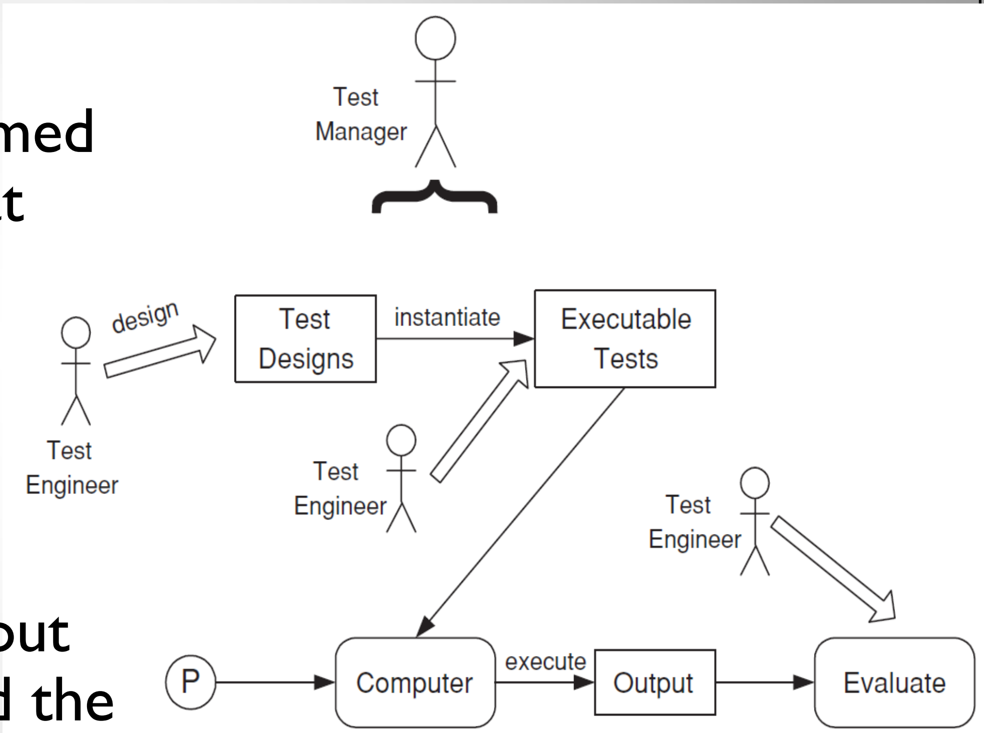
Activities of a Test Engineer

- Test engineer
  - An information technology professional who is in charge of one or more technical test activities including
    - designing test inputs,
    - Producing test case values,
    - Running test scripts,
    - Analyzing results, and
    - Reporting results to developers and managers

- Test Manager : In charge of one or more test engineers
  - Sets test policies and processes
  - Interacts with other managers on the project
  - Otherwise supports the engineers

Activities of a Test Engineer

- Test engineer must design tests by creating test requirements
- Requirements are then transformed into actual values and scripts that are ready for execution
  - 'P' in the figure – the results are evaluated to determine if the tests reveal a fault
- These activities may be carried out by one person or by several, and the process is monitored by a test manager



Types of Test Activities

- Testing can be broken up into four general types of activities
  1. Test Design → **1.a) Criteria-based** **1.b) Human-based**
  2. Test Automation
  3. Test Execution
  4. Test Evaluation

- Each type of activity requires different skills, background knowledge, education and training

1. Test Design – (a) Criteria-Based

- This is the most technical job in software testing
- Requires knowledge of :
  - Discrete math
  - Programming
  - Testing
- Requires much of a traditional CS degree
- Using people who are not qualified to design tests is a sure way to get ineffective tests

1. Test Design – (b) Human-Based

Design test values based on domain knowledge of the program and human knowledge of testing

- This is much harder than it may seem to developers
- Criteria-based approaches can be blind to special situations
- Requires knowledge of :
  - Domain, testing, and user interfaces
- Requires almost no traditional CS
  - A background in the domain of the software is essential
  - An empirical background is very helpful (biology, psychology, …)
  - A logic background is very helpful (law, philosophy, math, …)

2. Test Automation

Embed test values into executable scripts

- This is slightly less technical
- Requires knowledge of programming
  - Fairly straightforward programming – small pieces and simple algorithms
- Requires very little theory
- Very boring for test designers
- Programming is out of reach for many domain experts

3. Test Execution

Run tests on the software and record the results

- This is easy – and trivial if the tests are well automated
- Requires basic computer skills
  - Interns
  - Employees with no technical background
- If, for example, GUI tests are not well automated, this requires a lot of manual labor
- Test executors have to be very careful and meticulous with bookkeeping

4. Test Evaluation

Evaluate results of testing, report to developers

- This is much harder than it may seem
- Requires knowledge of :
  - Domain
  - Testing
  - User interfaces and psychology
- Usually requires almost no traditional CS
  - A background in the domain of the software is essential
  - An empirical background is very helpful (biology, psychology, …)
  - A logic background is very helpful (law, philosophy, math, …)
- This is intellectually stimulating, rewarding, and challenging
  - But not to typical CS majors – they want to solve problems and build things

Coverage Criteria for Testing

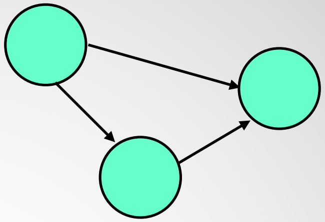
Changing Notions of Testing

- Old view focused on testing at each software development phase as being very different from other phases
  - Unit, module, integration, system …
- New view is in terms of structures and criteria
  - Graphs, logical expressions, syntax, input space

Criteria Based on Structures

<u>Structures</u> : Four ways to model software

1. Graphs



2. Logical Expressions

   (not X or not Y) and A and B

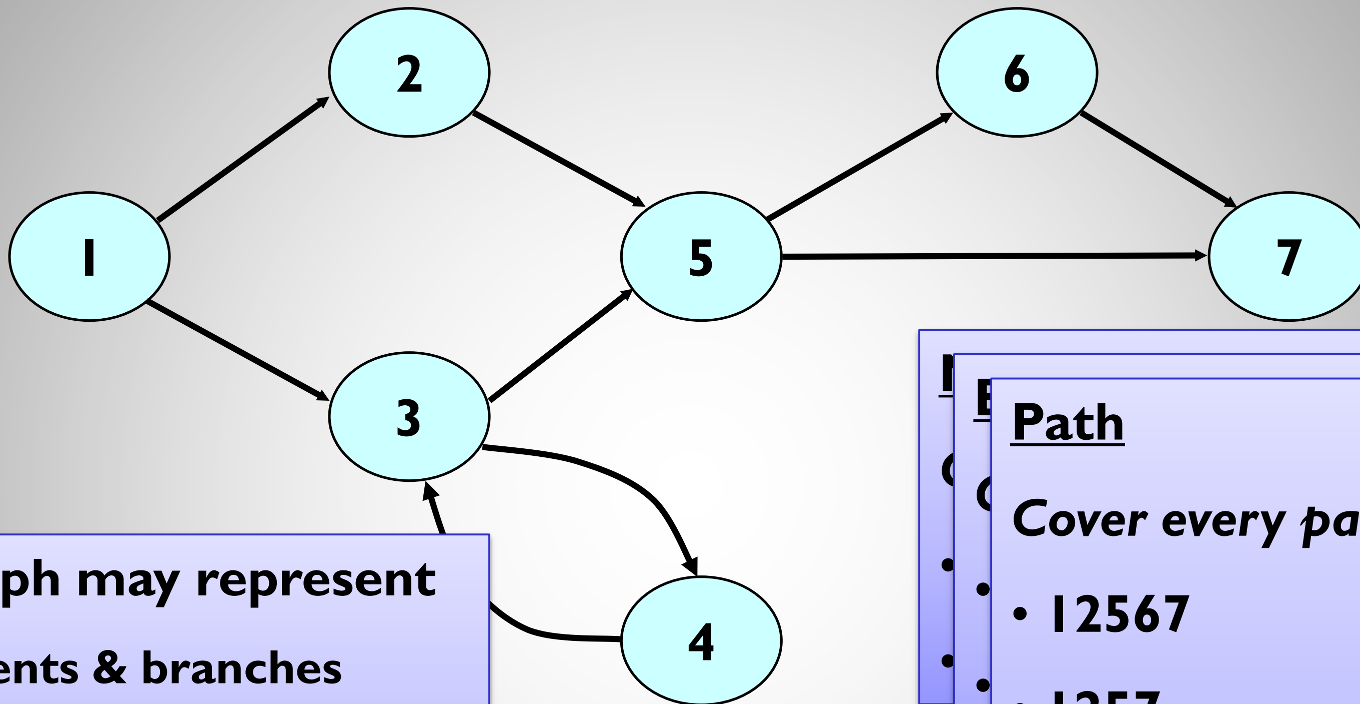
3. Input Domain Characterization

   A: {0, 1, >1}
   B: {600, 700, 800}
   C: {swe, cs, isa, infs}

4. Syntactic Structures

```
if (x > y)
    z = x - y;
else
   z = 2 * x;
```

1. Graph Coverage – Structural



This graph may represent

- **statements & branches**
- **methods & calls**
- **components & signals**
- **states and transitions**

⋮

Path

Cover every path

- **12567**
- **1257**
- **13567**
- **1357**
- **1343567**
- **134357 …**

2. Logical Expressions

$$((a > b) \text{ or } G) \text{ and } (x < y)$$

Transitions

Program Decision Statements → **Logical Expressions**

Software Specifications

2. Logical Expressions

((a > b) or G) and (x < y)

- Predicate Coverage : Each predicate must be true and false
  - *((a>b) or G) and (x < y)* = True, False
- Clause Coverage : Each clause must be true and false
  - *(a > b)* = True, False
  - *G* = True, False
  - *(x < y)* = True, False
- Combinatorial Coverage : Various combinations of clauses
  - *Active Clause Coverage*: Each clause must determine the predicate's result

3. Input Domain Characterization

- Describe the input domain of the software
  - Identify inputs, parameters, or other categorization
  - Partition each input into finite sets of representative values
  - Choose combinations of values
- System level
  - Number of students { *0, 1, >1* }
  - Level of course { *600, 700, 800* }
  - Major { *swe, cs, isa, infs* }
- Unit level
  - Parameters *F (int X, int Y)*
  - Possible values X*: { <0, 0, 1, 2, >2 }, Y : { 10, 20, 30 }*
  - Tests *F (-5, 10), F (0, 20), F (1, 30), F (2, 10), F (5, 20)*

4. Syntactic Structures

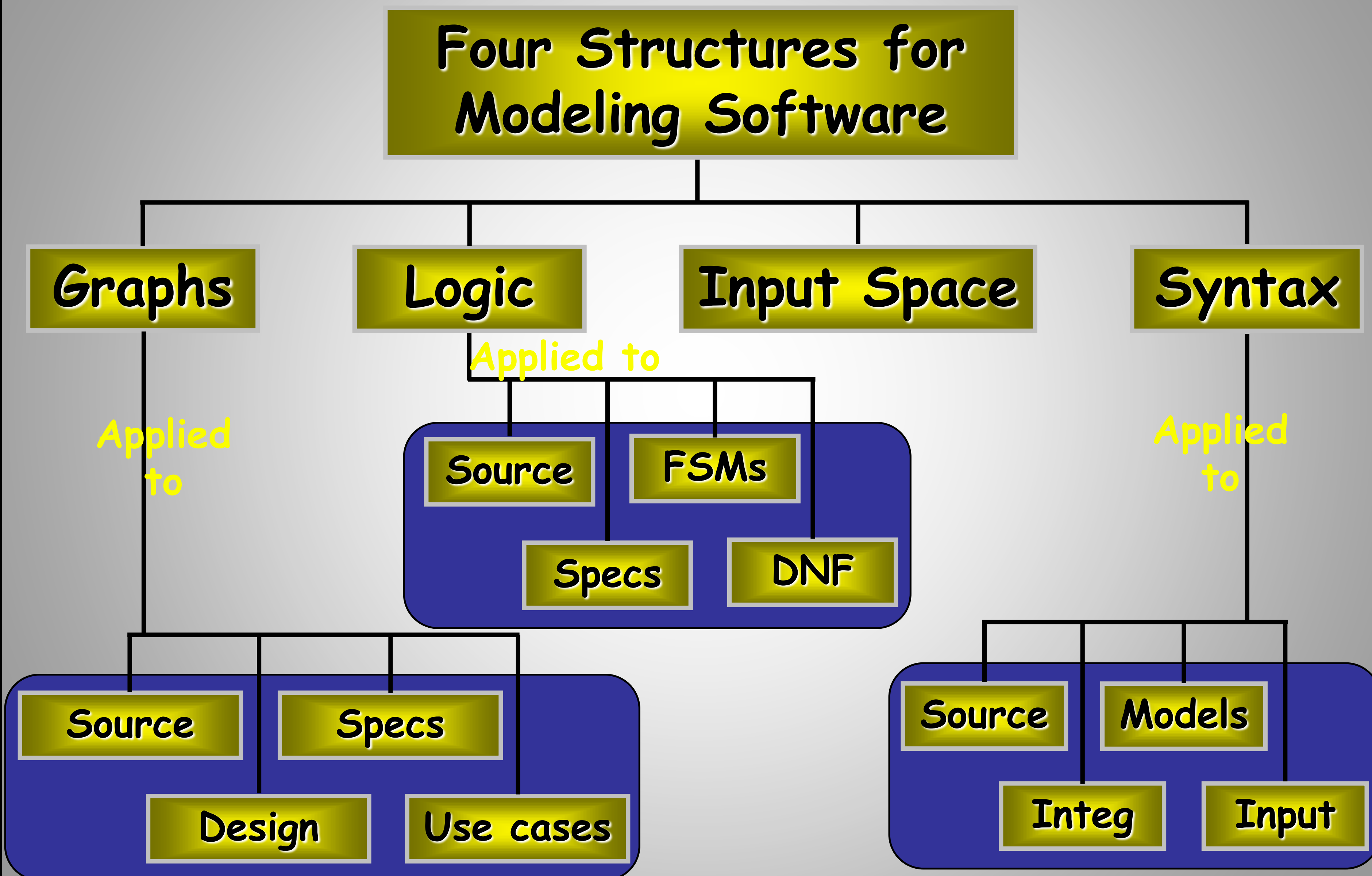
- Based on a grammar, or other syntactic definition
- Primary example is mutation testing
  1. Induce small changes to the program: mutants
  2. Find tests that cause mutant programs to fail: killing mutants
  3. Failure is defined as different output from the original program
  4. Check the output of useful tests on the original program

- Example program and mutants

```
if (x > y)
    z = x - y;
else
    z = 2 * x;
```

```
if (x > y)
Δif (x >= y)
    z = x - y;
    Δ z = x + y;
    Δ z = x – m;
else
    z = 2 * x;
```

Coverage Overview

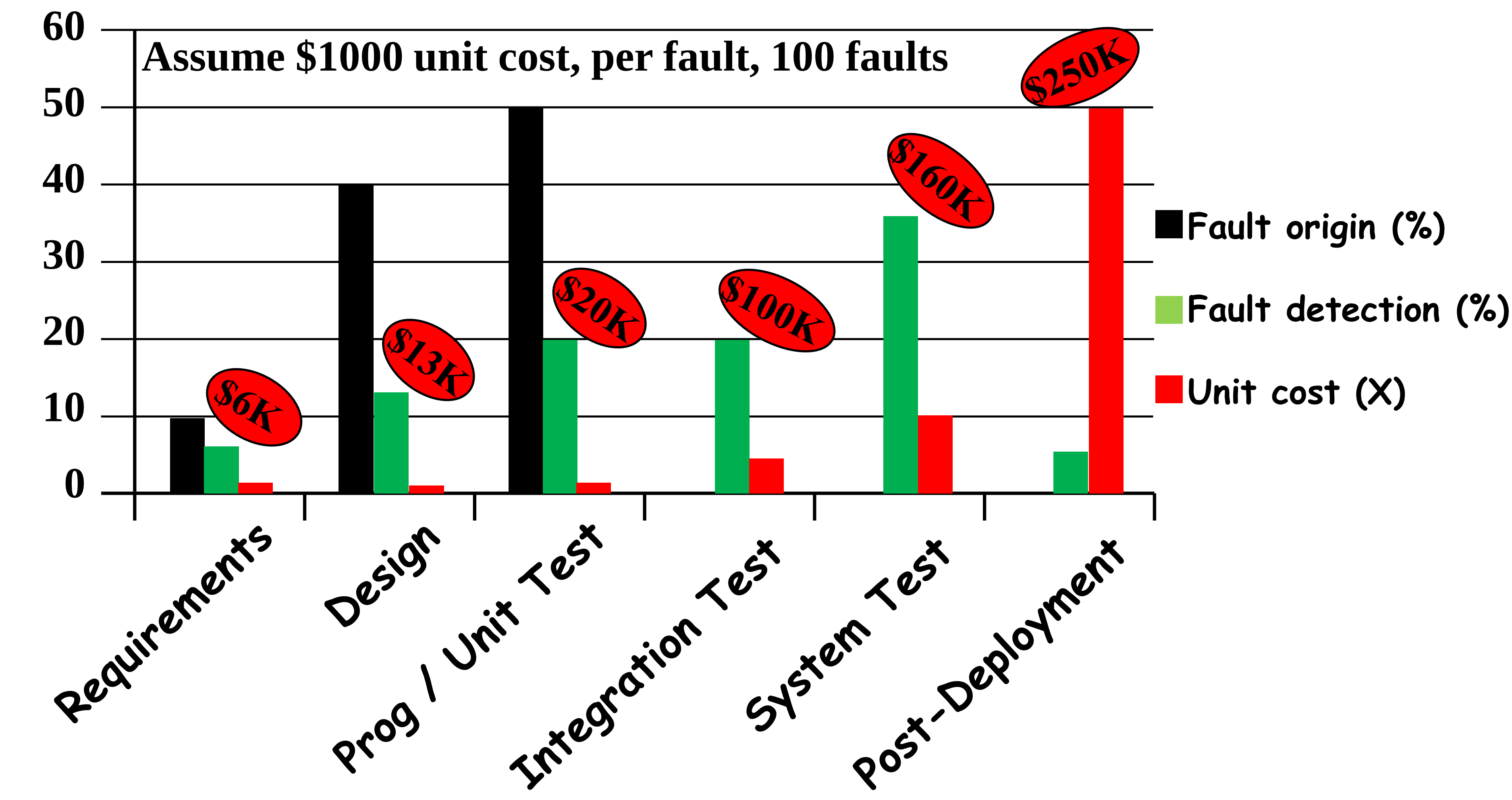


Cost of Not Testing

Poor Program Managers might say:
"Testing is too expensive."

- Testing is the most time consuming and expensive part of software development
- Not testing is even more expensive
- If we have too little testing effort early, the cost of testing increases
- Planning for testing after development is prohibitively expensive

Cost of Late Testing



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