Software Quality Assurance

Testing Preliminaries

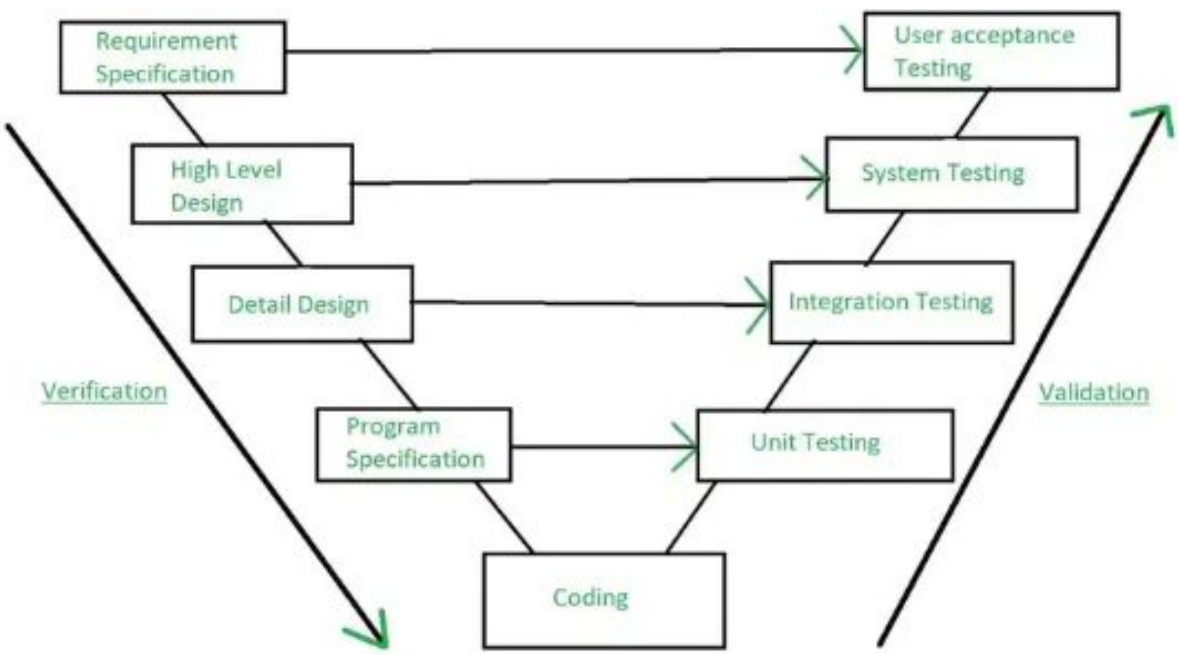
Lecture 2

Verification & Validation (*IEEE*)

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

IV&V stands for "independent verification and validation"

Verification: *Are we building the product right?*
Validation: *Are we building the right product?*



Verification and Validation

Verification:

- Inspections
- Reviews
- Walkthroughs
- Desk-checking

Validation:

- Black Box Testing
- White Box Testing
- Unit Testing
- Integration Testing

Testing & Debugging

- Testing : Evaluating software by observing its execution
- Test Failure : Execution of a test that results in a software failure
- Debugging : The process of finding a fault given a failure

Not all inputs will “trigger” a fault into causing a failure

Testing vs Debugging

TESTING	DEBUGGING
a) Finding and locating of a defect	**a)** Fixing that defect
b) Done by Testing Team	**b)** Done by Development team
c) Intention behind is to find as many defect as possible	**c)** Intention is to remove those defects *© ianswer4u.com*

Testing vs Debugging

Testing	Debugging
Testing finds cases where a program does not meet its specification.	The process of debugging involves analyzing and possibly extending (with debugging statemets) the given program that does not meet the specification in order to find a new program that is close to the original and does satisfy the specifications.
Its objective is to demonstrate that the program meets its design specifications.	Its objective is to detect the exact cause and remove known errors in the program.
Testing is complete when all desired verifications against specification have been performed.	Debugging is complete when all known errors in the program have been fixed.
Testing can begin in the early stahges of the software development.	Debugging can begin only after the program is coded.
Testing is the process of validating the correctness of the program.	Debugging is the process of eliminating the errors in a program.

Testing Goals 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 not distinguish between incorrect behavior and mistakes in the program
- Does not help to develop software that is reliable or safe

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

This is what hardware engineers often expect

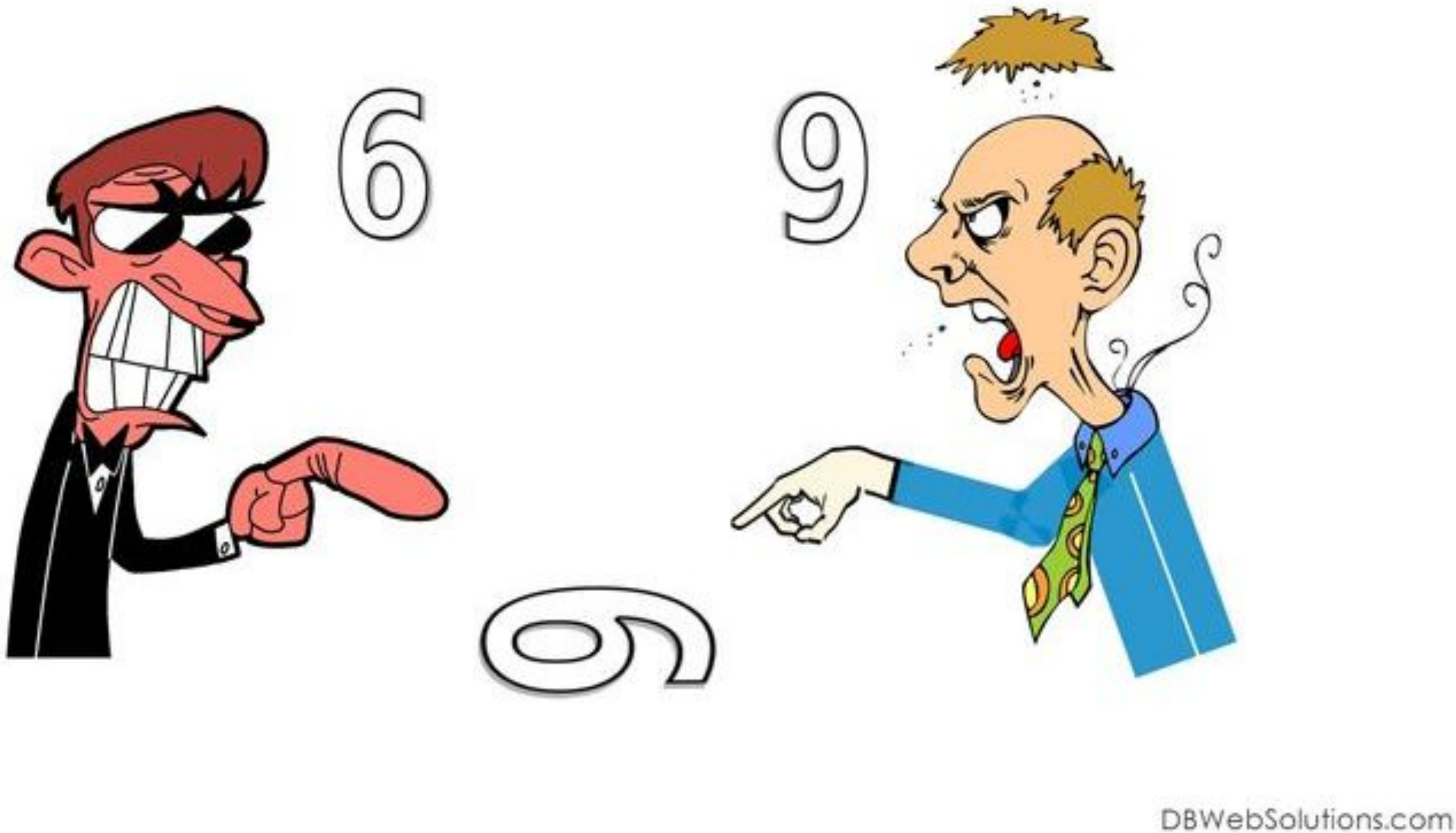
Level 2 Thinking

- 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.

How can we move to a *team approach* ??

Developer vs Tester



Level 3 Thinking

- Testing can only show the presence of failures
- 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

It 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 is to measure and improve software quality
- Their expertise should help the developers

This is the way "traditional" engineering works

Software Testing Activities

- Test Engineer : An IT professional who is in charge of one or more technical test activities
  - Designing test inputs
  - Producing test values
  - Running test scripts
  - Analyzing results
  - 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

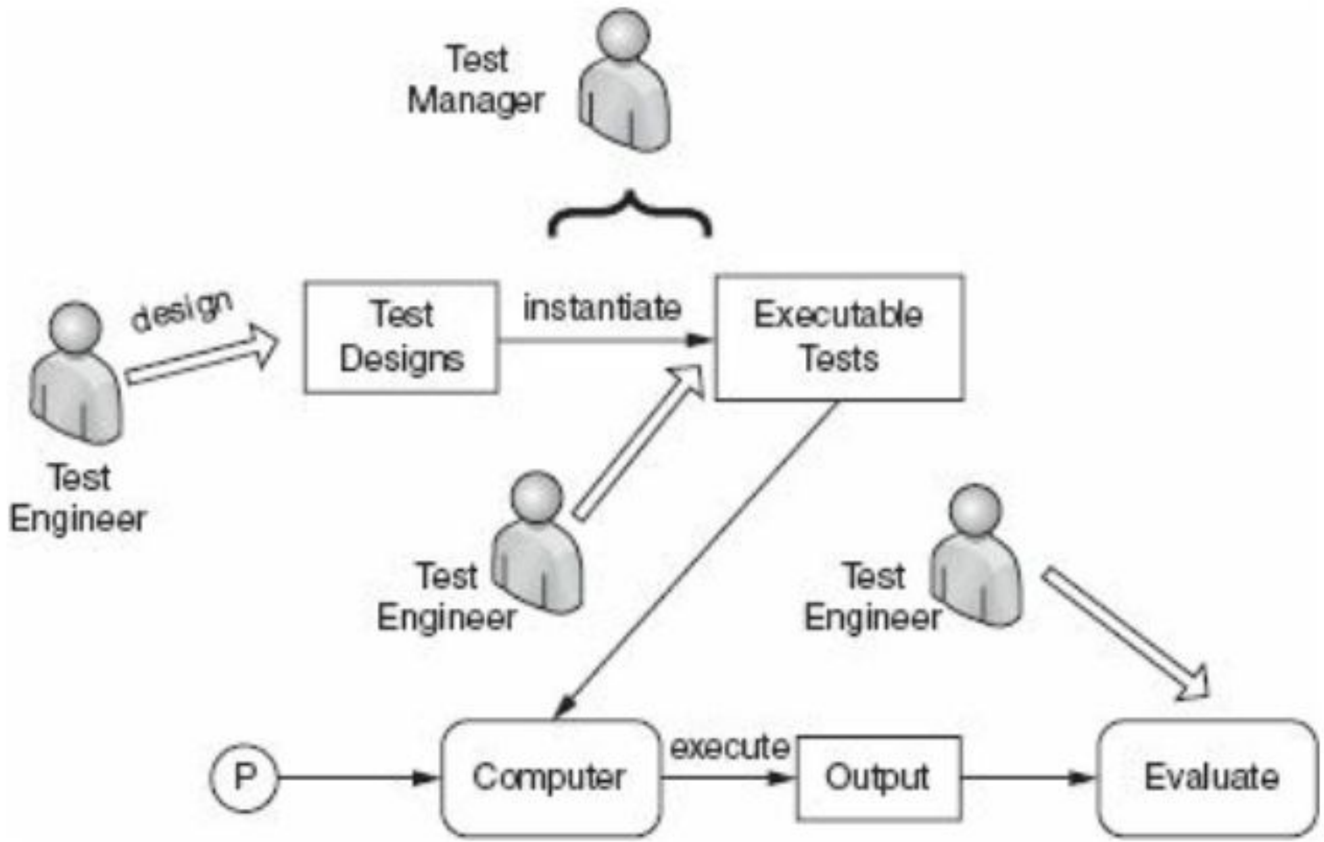


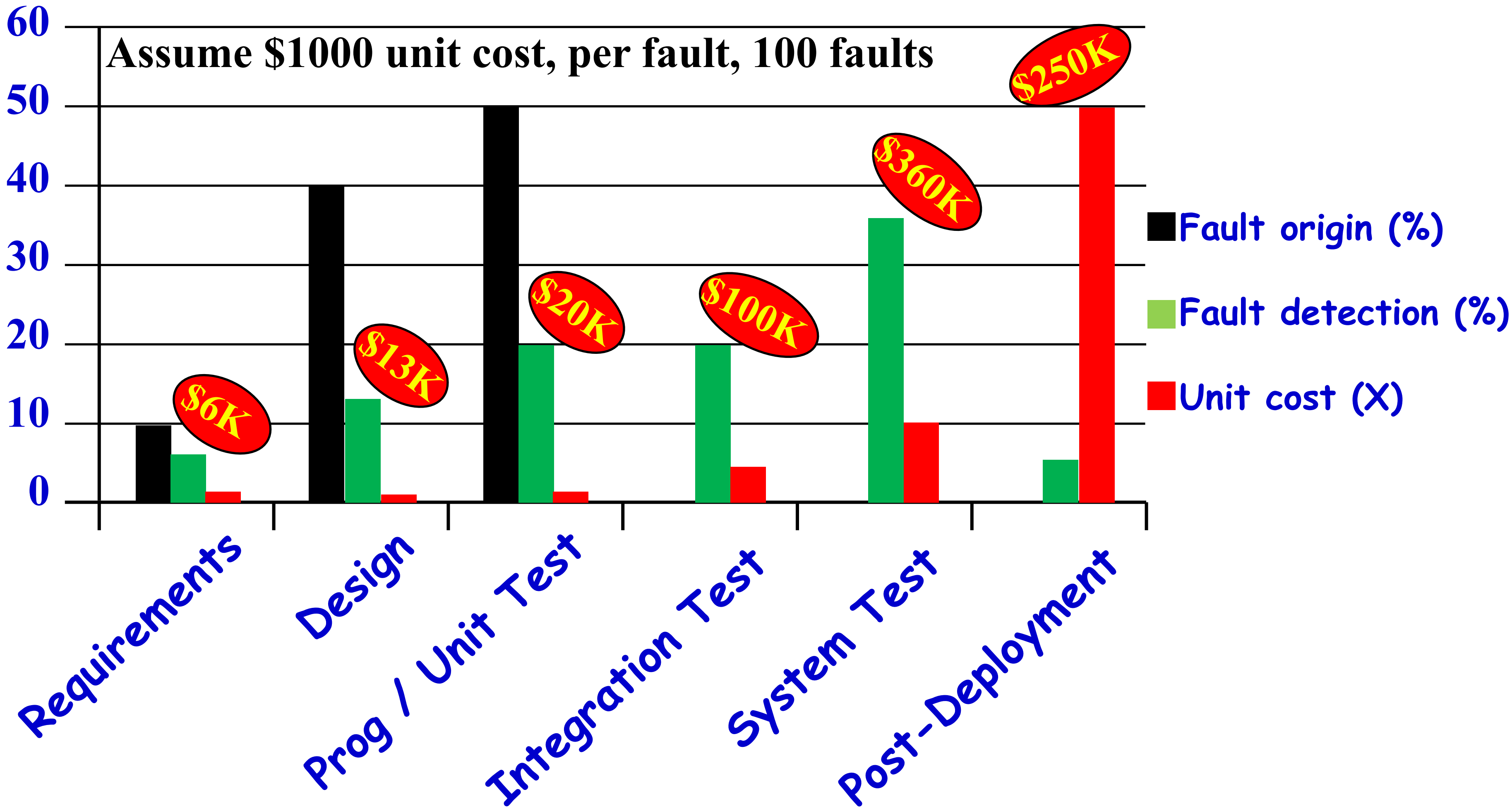
Figure: Activities of test engineer.

Cost of Not Testing

- 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

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

Cost of Late Testing



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

Complexity of Testing Software

- No other engineering field builds products as complicated as software
- The term correctness has no meaning
  - Is a building correct?
  - Is a car correct?
  - Is a subway system correct?
- Instead of looking for "correctness," wise software engineers try to evaluate software's "behavior" to decide if the behavior is acceptable within consideration of a large number of factors including (but not limited to) reliability, safety, maintainability, security, and efficiency.
- Obviously this is more complex than the naive desire to show the software is correct.

Complexity of Testing Software

- Like other engineers, we must use abstraction to manage complexity
  - This is the purpose of the model-driven test design process
  - The "model" is an abstract structure
- The Model-Driven Test Design (MDTD) process breaks testing into a series of small tasks that simplify test generation.
  - Then test designers isolate their task, and work at a higher level of abstraction by using mathematical engineering structures to design test values independently of the details of software or design artifacts, test automation, and test execution.

Software Testing Foundations

Testing can only show the presence of failures

Not their absence

Fault & Failure Model (RIPR)

Four conditions necessary for a failure to be observed

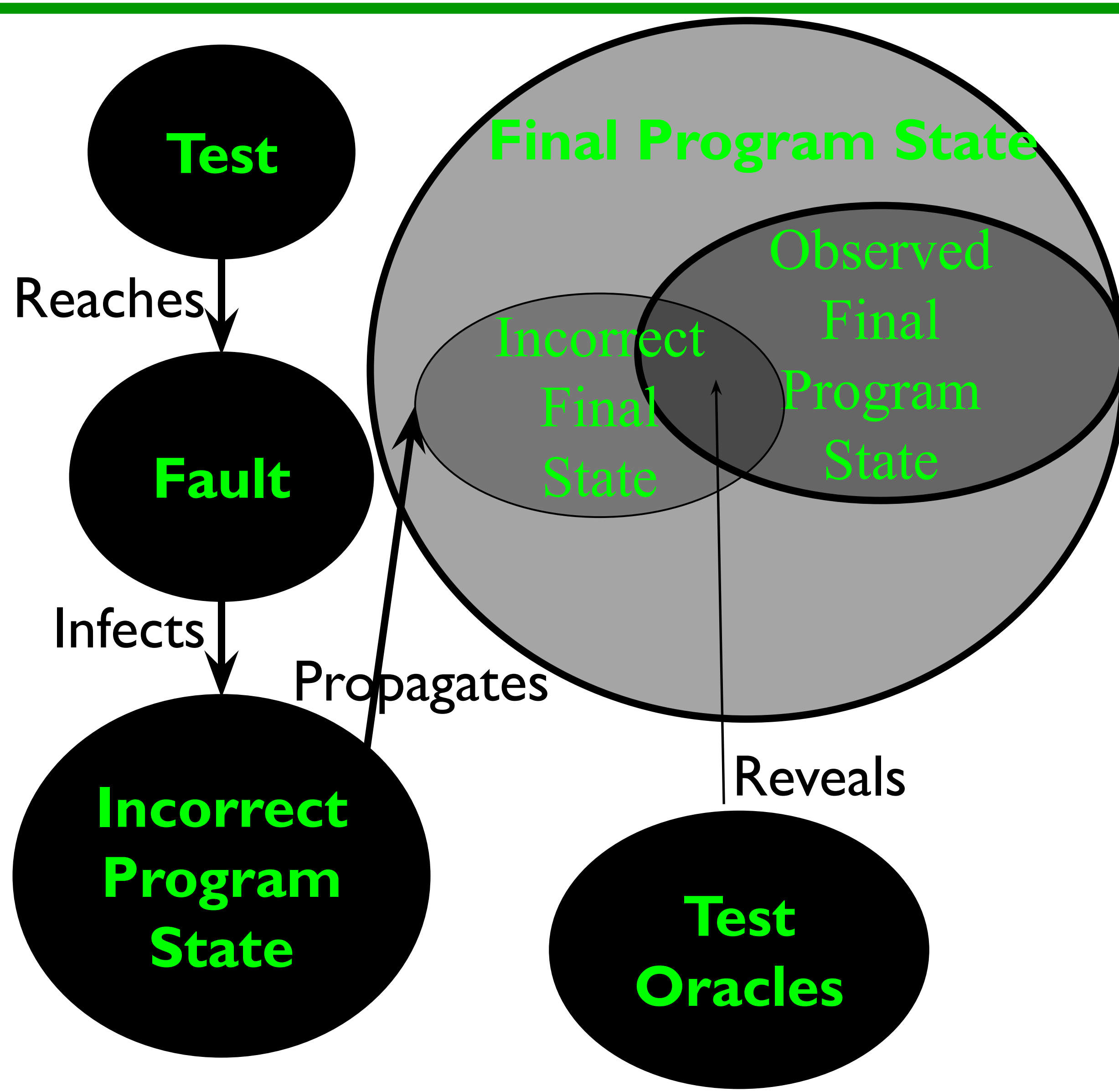
1. **Reachability** : The location or locations in the program that contain the fault must be reached
2. **Infection** : The state of the program must be incorrect
3. **Propagation** : The infected state must cause some incorrect output or final state
4. **Reveal** : The tester must observe part of the incorrect portion of the program state

```c
#include <stdio.h>

int calculate_total(int price, int quantity) {

    return price + quantity;
}

int main() {
    int price = 10;
    int quantity = 5;

    int total = calculate_total(price, quantity);

    printf("Total: %d\n", total);
    return 0;
}
```

```c
#include <stdio.h>

// Function with a fault
int calculate_total(int price, int quantity) {
    // Fault: Incorrect calculation (missing multiplication)
    return price + quantity; // Faulty calculation
}

int main() {
    int price = 10;
    int quantity = 5;

    // 1. Reachability: Fault location must be reached
    int total = calculate_total(price, quantity); // Fault location reached

    // 2. Infection: Program state becomes incorrect
    // In this case, 'total' is incorrect due to the fault in 'calculate_total'

    // 3. Propagation: Infected state leads to incorrect output
    printf("Total: %d\n", total); // Incorrect output due to fault

    // 4. Reveal: Tester observes incorrect portion of program state
    // Tester observes the incorrect total printed to the console

    return 0;
}
```

RIPR Model

- **R**eachability
- **I**nfection
- **P**ropagation
- **R**evealability



Traditional Testing Levels

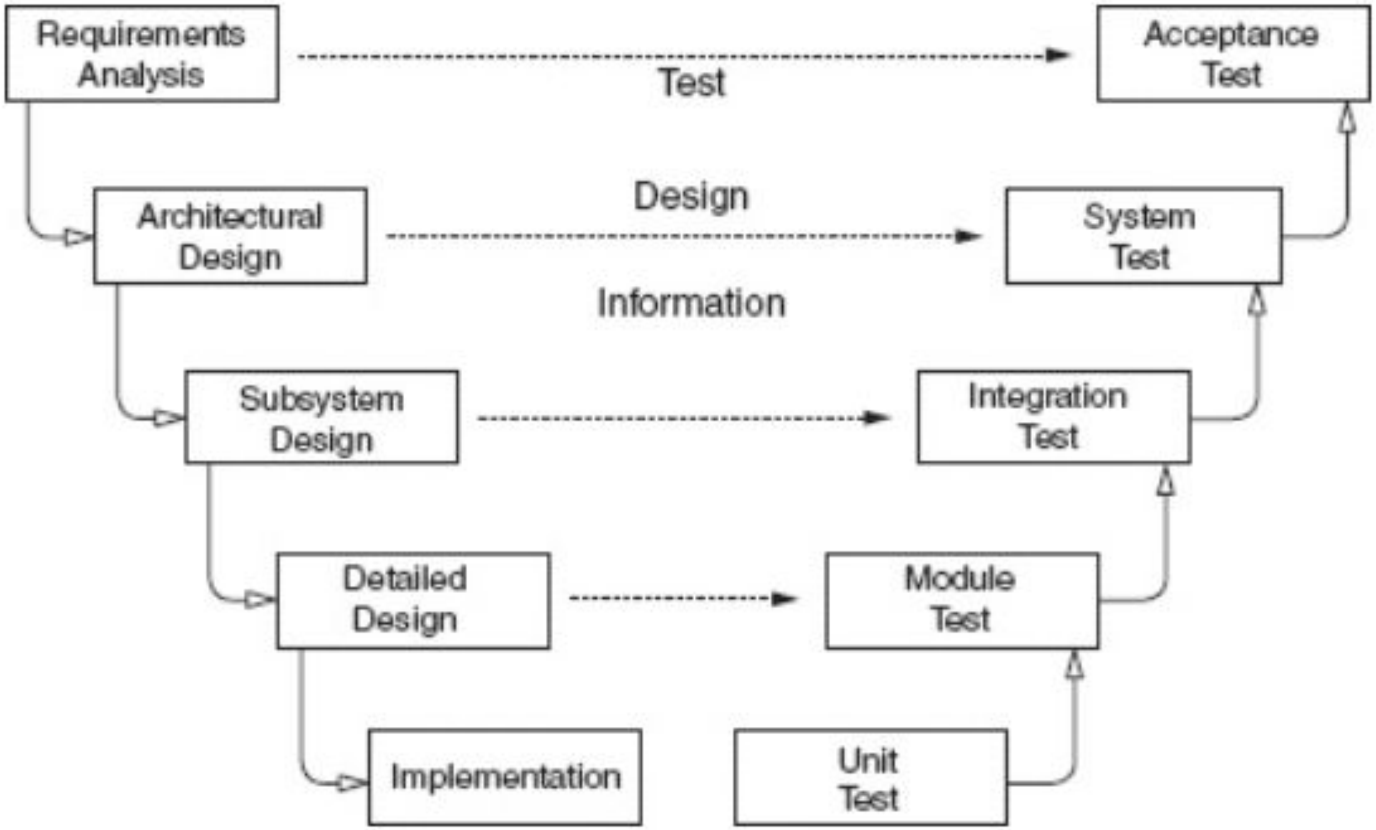


Figure: Software development activities and testing levels – the "V Model"

V-Model

- The *requirements analysis* phase of software development captures the customer's needs.

- *Acceptance testing* is designed to determine whether the completed software in fact meets these needs. In other words, acceptance testing probes whether the software does what the users want.
  - Acceptance testing must involve users or other individuals who have strong domain knowledge.

V-Model Cont.

- The *architectural design* phase of software development chooses *components and connectors* that together realize a system whose specification is intended to meet the previously identified requirements.
- *System testing* is designed to determine whether the assembled system meets its specifications. It assumes that the pieces work individually, and asks if the system works as a whole.
  - This level of testing usually looks for design and specification problems.
  - It is a very expensive place to find lower-level faults and is usually not done by the programmers, but by a separate testing team

V-Model Cont.

- The *subsystem design* phase of software development specifies the structure and behavior of **subsystems**, each of which is intended to satisfy some function in the overall architecture. Often, the subsystems are adaptations of previously developed software.
- *Integration testing* is designed to assess whether the interfaces between modules (defined below) in a subsystem have consistent assumptions and communicate correctly.
  - Integration testing must assume that modules work correctly.
  - Integration testing is usually the responsibility of members of the development team

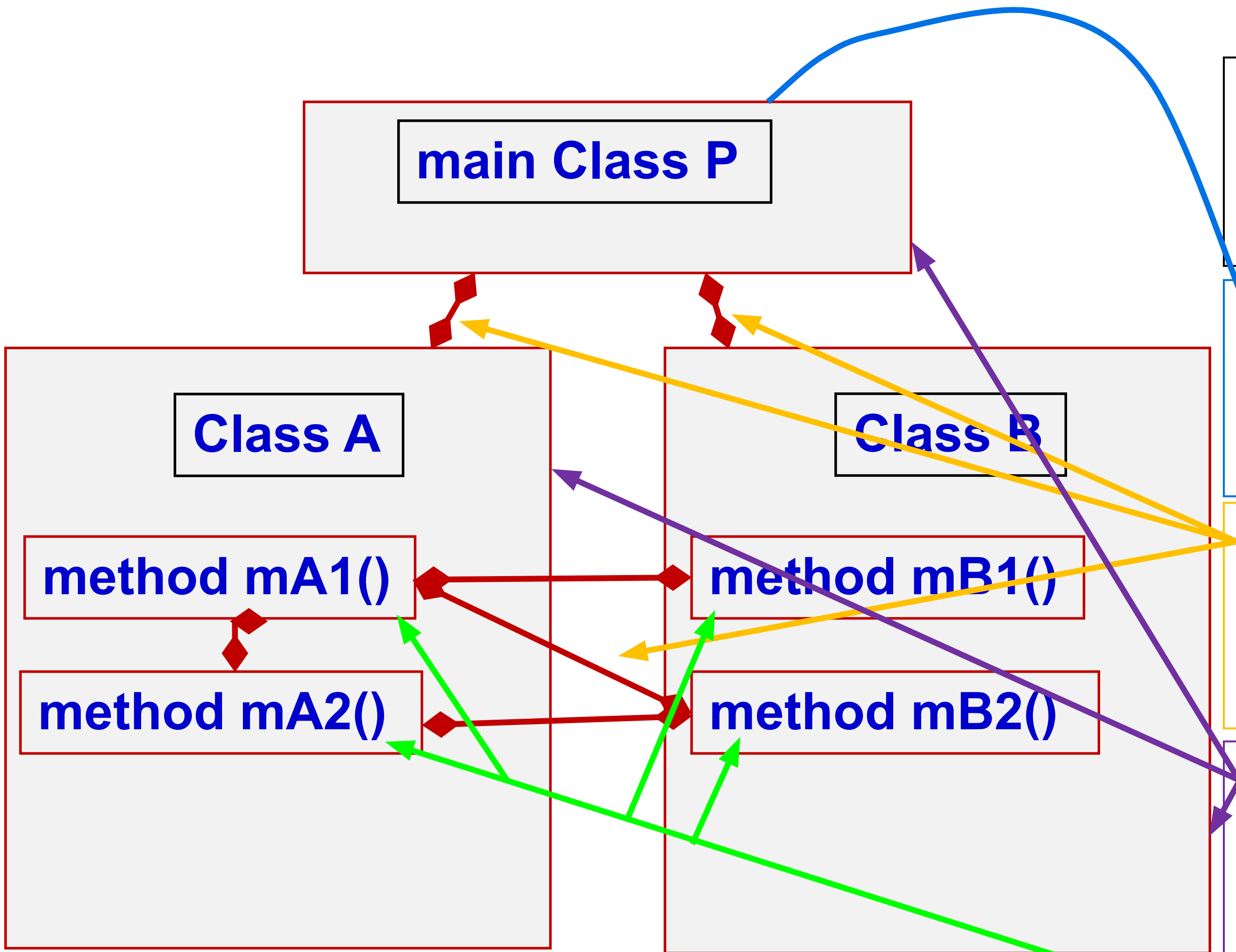
V-Model Cont.

- The *detailed design* phase of software development determines the structure and behavior of **individual modules**. A *module* is a collection of related units that are assembled in a file, package, or class.
  - This corresponds to a file in C, a package in Ada, and a class in C++ and Java.
- *Module testing* is designed to assess individual modules in isolation, including how the component units interact with each other and their associated data structures.
  - Most software development organizations make module testing the responsibility of the programmer; hence the common term *developer testing*.

V-Model Cont.

- *Implementation* is the phase of software development that actually produces code. A program *unit*, or procedure, is one or more contiguous program statements, with a name that other parts of the software use to call it.
  – Units are called functions in C and C++, procedures or functions in Ada, methods in Java, and subroutines in Fortran.
- *Unit testing* is designed to assess the units produced by the implementation phase and is the "lowest" level of testing.
  – As with module testing, most software development organizations make unit testing the responsibility of the programmer, again, often called developer testing.

Traditional Testing Levels (2.3)

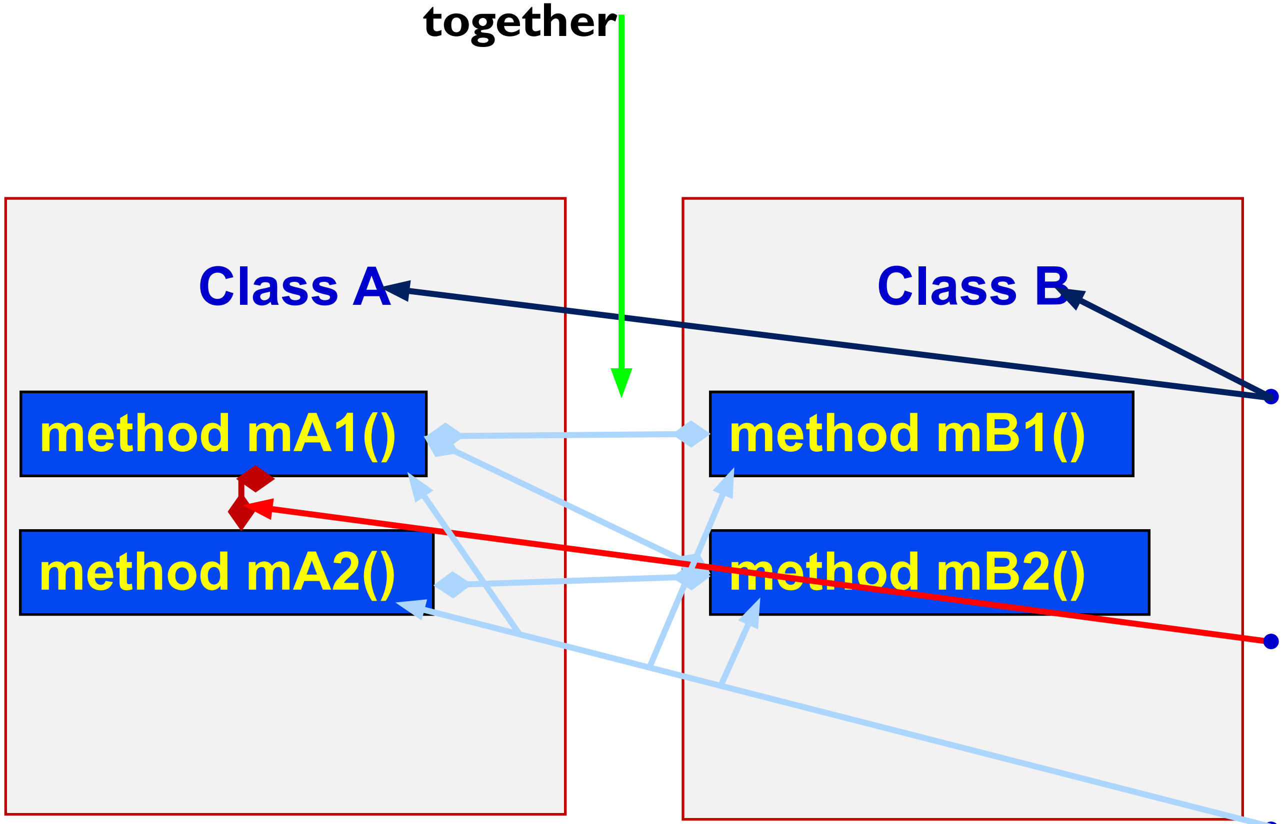


- **Acceptance testing** : Is the software acceptable to the user?
- **System testing** : Test the overall functionality of the system
- **Integration testing** : Test how modules interact with each other
- **Module testing (developer testing)** : Test each class, file, module, component
- **Unit testing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

This view obscures underlying similarities

Object-Oriented Testing Levels

- **Inter-class testing** : Test multiple classes together



- **Intra-class testing** : Test an entire class as sequences of calls
- **Inter-method testing** : Test pairs of methods in the same class
- **Intra-method testing** : Test each method individually

Old View : Colored Boxes

- Black-box testing(BBT) : Derive tests from external descriptions of the software, including specifications, requirements, and design
- White-box testing(WBT) : Derive tests from the source code internals of the software, specifically including branches, individual conditions, and statements
- Model-based testing(MBT) : Derive tests from a model of the software (such as a UML diagram)

MDTD makes these distinctions less important.

The more general question is:

from what abstraction level do we derive tests?

Coverage Criteria (2.4)

- Even small programs have too many inputs to fully test them all
  - **private static double computeAverage (int A, int B, int C)**
  - On a 32-bit machine, each variable has over 4 billion possible values
  - Over 80 octillion possible tests!!
  - Input space might as well be infinite
- This is the source of two key problems in testing:
  - (1) how do we search? and
  - (2)when do we stop?
- Testers search a huge input space
  - Trying to find the fewest inputs that will find the most problems

Coverage Criteria (2.4)

- Coverage criteria give structured, practical ways to search the input space
- Satisfying a coverage criterion gives a tester some amount of confidence in two crucial goals:
  - Search the input space thoroughly
  - Not much overlap in the tests
- Coverage criteria have many advantages for improving the quality and reducing the cost of test data generation.

Advantages of Coverage Criteria

1. Maximize the "bang for the buck"
   – with fewer tests that are effective at finding more faults
2. Provide traceability from software artifacts to tests
   – Source, requirements, design models, …
3. Make regression testing easier
4. Gives testers a "stopping rule" … when testing is finished
5. Can be well supported with powerful tools

Test Requirements and Criteria

- Test Criterion : A collection of rules and a process that define test requirements
  - Cover every statement
  - Cover every functional requirement
- Test Requirements : Specific things that must be satisfied or covered during testing
  - *Each statement might be a test requirement*
  - *Each functional requirement might be a test requirement*

Testing researchers have defined dozens of criteria, but they are all really just a few criteria on four types of structures …

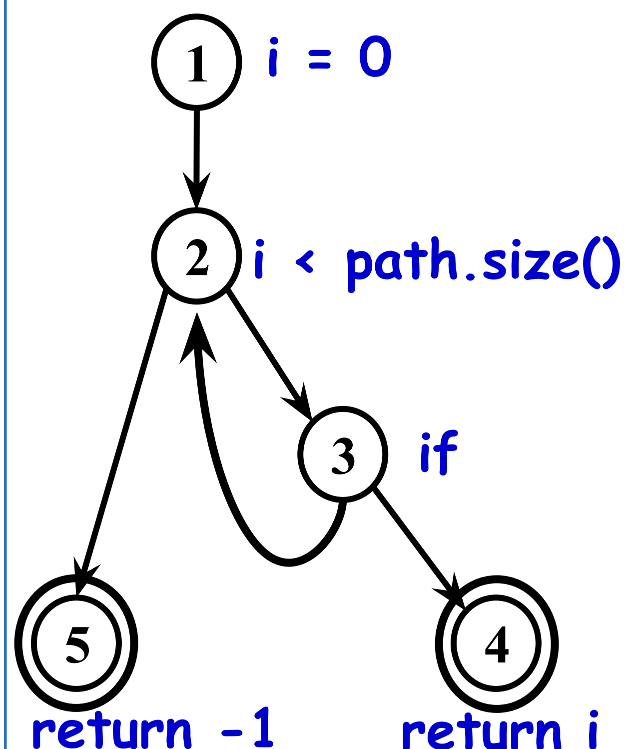
1. Input domains
2. Graphs
3. Logic expressions
4. Syntax descriptions

Small Illustrative Example

Software Artifact : Java Method

```
/**
   * Return index of node n at the
   * first position it appears,
   * -1 if it is not present
*/
public int indexOf (Node n)
{
   for (int i=0; i < path.size(); i++)
      if (path.get(i).equals(n))
         return i;
   return -1;
}
```

Control Flow Graph

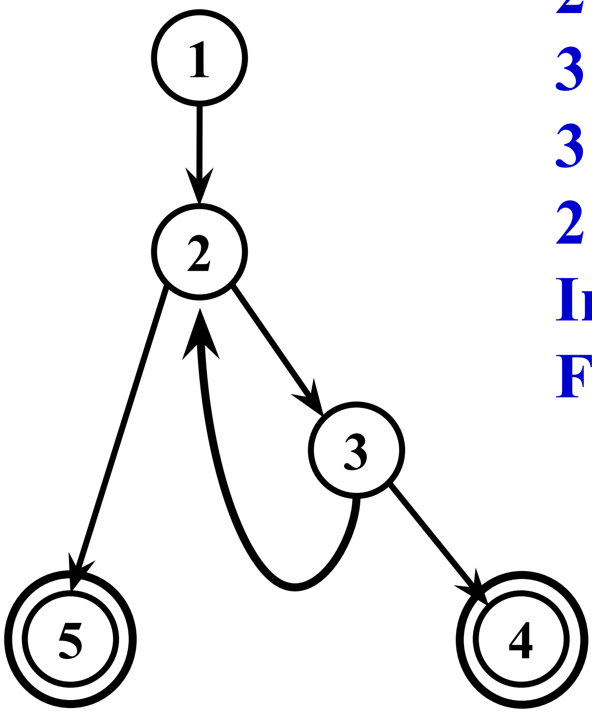


Example (2)

Support tool for graph coverage

http://www.cs.gmu.edu/~offutt/softwaretest/

Graph
Abstract version



Edges
1 2
2 3
3 2
3 4
2 5
Initial Node: 1
Final Nodes: 4, 5

6 requirements for Edge-Pair Coverage

1. [1, 2, 3]
2. [1, 2, 5]
3. [2, 3, 4]
4. [2, 3, 2]
5. [3, 2, 3]
6. [3, 2, 5]

Test Paths
[1, 2, 5]
[1, 2, 3, 2, 5]
[1, 2, 3, 2, 3, 4]

Find values …