Introduction to Software Testing

Instructor : Ali Sharifara, PhD

CSE 4321/5321

Summer 2017

Agenda

- Grader Info
- Quick Review
- Introduction to Software Testing
- Input Space Partitioning

Outline

- Introduction
- Basic Concepts
- The Testing Process
- Types of Testing
- Testing Philosophy
- Summary

Introduction

What is software testing ?

Software testing is a check activity to validate whether the actual results matches with expected results and to ensure that the software system is bug free.

Why it is needed ?

- Field validation issues (e.g. check date of birth, or age, etc.)
- Sites responsiveness under specific load (e.g. during a registration …)
- Payment related issues (e.g. payment gateways)
- Security issues (payment methods)
- Verifying store policies and customer support (return policy, etc.)
- Lack of device and browser compatibility (different browsers.)
- etc.

Lack of testing may lead to

- Loss of money
- Loss of time
- Loss of business reputation

SQA, SQC, and Testing

- **Software Quality Assurance (SQA)**
  - Is **error preventive** & **verification** activity**.**
  - **SQA role** is to observe that documented standards, processes, and procedures are followed as enforced by development
- **Software Quality Control (SQC)**
  - is defect **detection** and **validation** activity.
  - **SQC role** (usually testers) is to **validate** the quality of a system and to check whether this application adheres the defined quality standards or not.
- **Testing**
  - It includes activities that ensure that identification of bugs/error/defects in a software

Software Development Life Cycle

- Software Development Life Cycle (SDLC)
  - A series of steps, or phases, that provides a model for the development and lifecycle management of an application or software

- **SDLC Steps**
  - Requirement Analysis
  - Project Planning
  - Project Design
  - Development
  - **Testing**
  - Implementation
  - Maintenance

SDLC methods

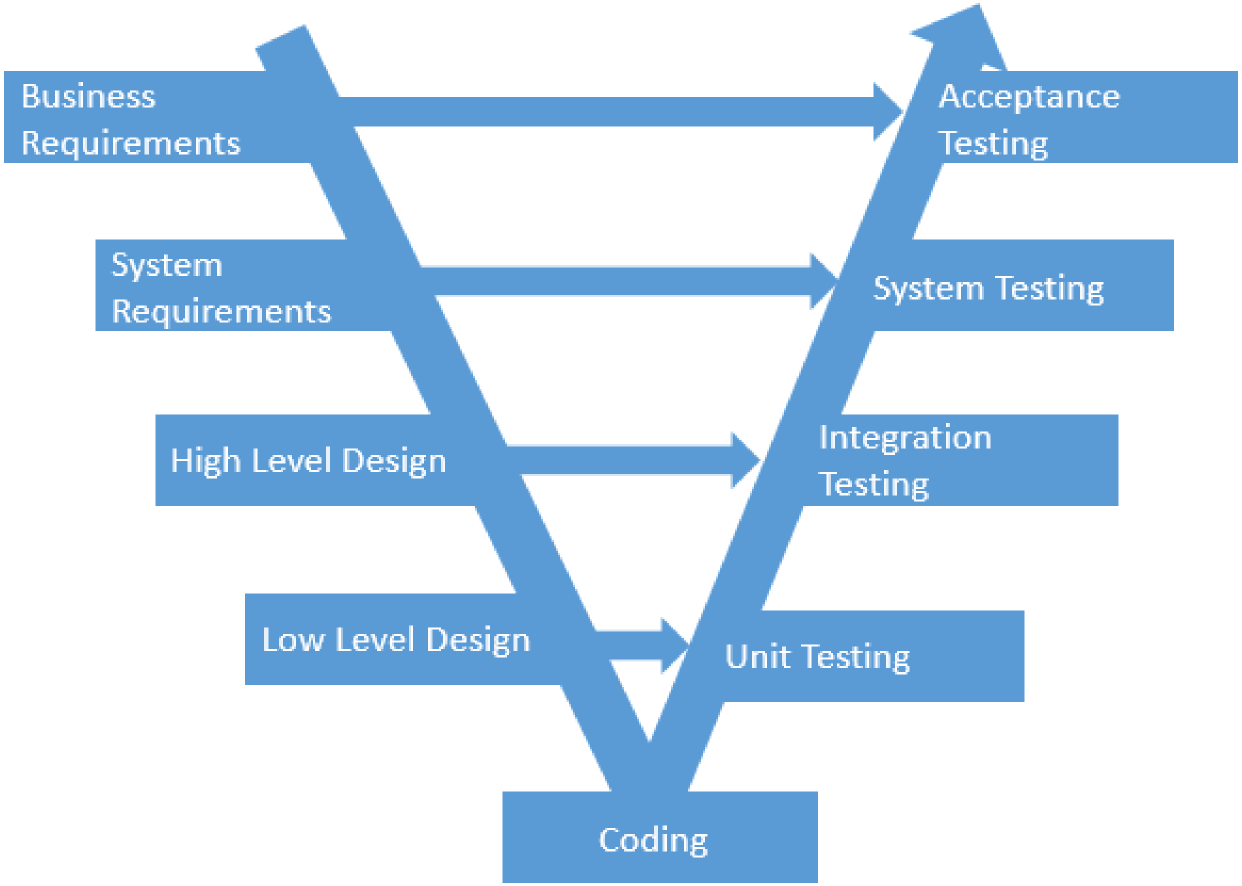
1. Waterfall model
2. Spiral Model
3. Prototype Model
4. Agile Model
5. V-Model
6. W-Model

V-model vs W-model

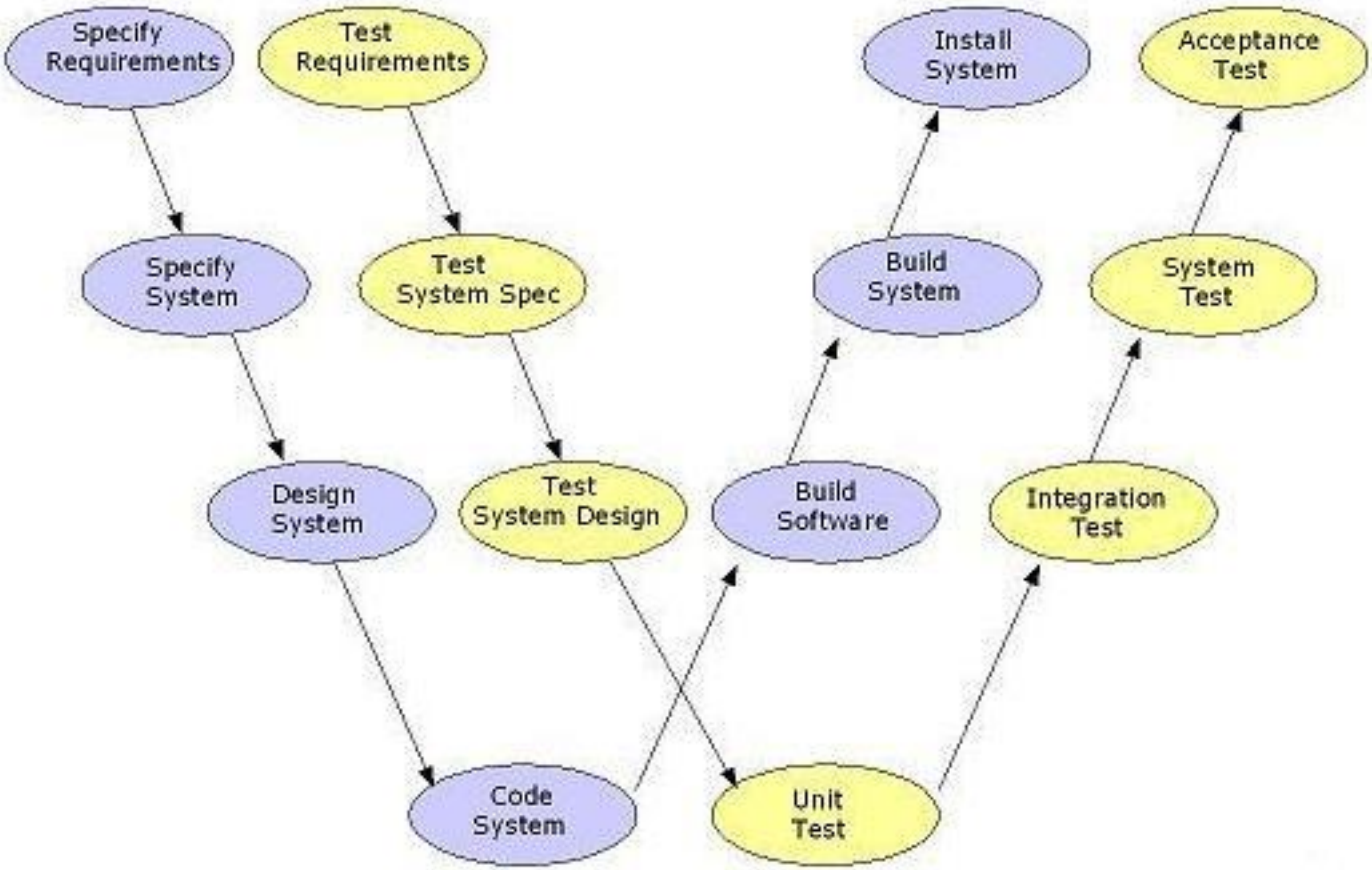
- **V-model :**
  - v model is the most used software development model in companies where we plan about testing at the same time of system development.

- **W-model**
  - This model is the most recent software development model where we start real testing activity concurrently with the software development process (from the beginning)

V-model


Business Requirements
Acceptance Testing
System Requirements
System Testing
High Level Design
Integration Testing
Low Level Design
Unit Testing
Coding

W-model



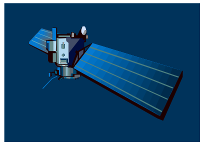
Specify Requirements
Test Requirements
Install System
Acceptance Test
Specify System
Test System Spec
Build System
System Test
Design System
Test System Design
Build Software
Integration Test
Code System
Unit Test

Testing in the 21st Century

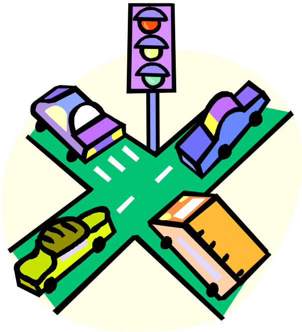
- Software defines behavior
- network routers, finance, switching networks, other infrastructure
- Today's software market :
  - is much bigger
  - is more competitive
  - has more users
- Embedded Control Applications
  - airplanes, air traffic control
  - spaceships
  - watches
  - ovens
  - remote controllers
- Agile processes put increased pressure on testers
  - Programmers must unit test – with no training or education!
  - Tests are key to functional requirements – but who builds those tests ?

Software is a skin that Surrounds our civilization

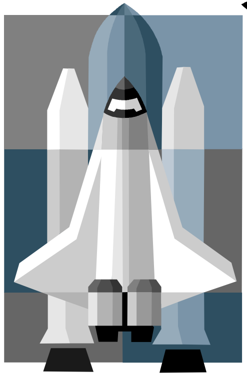








Software Quality

- The priority concern in software engineering
  - No quality, no engineering!
  - Software malfunctions can cause severe consequences including environmental damages, and even loss of human life.
- An important factor that distinguishes a software product from its competition
  - The feature set tends to converge between similar products

Software Testing

- A dynamic approach to ensuring software correctness
- Involves sampling the input space, running the test object, and observing the runtime behavior
- Among the most widely used approaches in practice
  - Labor intensive, and often consumes more than 50% of development cost

Static Analysis

- Reason about the behavior of a program based on the source code, i.e., without executing the program

- Question: How do you compare the two approaches?

Outline

- Introduction
- Basic Concepts
- The Testing Process
- Types of Testing
- Testing Philosophy
- Summary

Software Faults, Errors & Failures

- Fault : a static defect in the software
  - Incorrect instructions, missing instructions, extra instructions
  - What we often call a “bug”

- Failure : An incorrect output/behavior that is caused by executing a fault (external).
  - The failure may occur immediately (crash!) or much, much later in the execution

- Error : An incorrect internal state that is the manifestation of some fault (a mistake made by an engineer)
  - Often misunderstanding of a requirement or design specification

An Example, Fault and Failure

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

Testing and Debugging

- Testing attempts to surface failures in our software systems
- Debugging : attempts to associate failures with faults so they can be removed from the system.

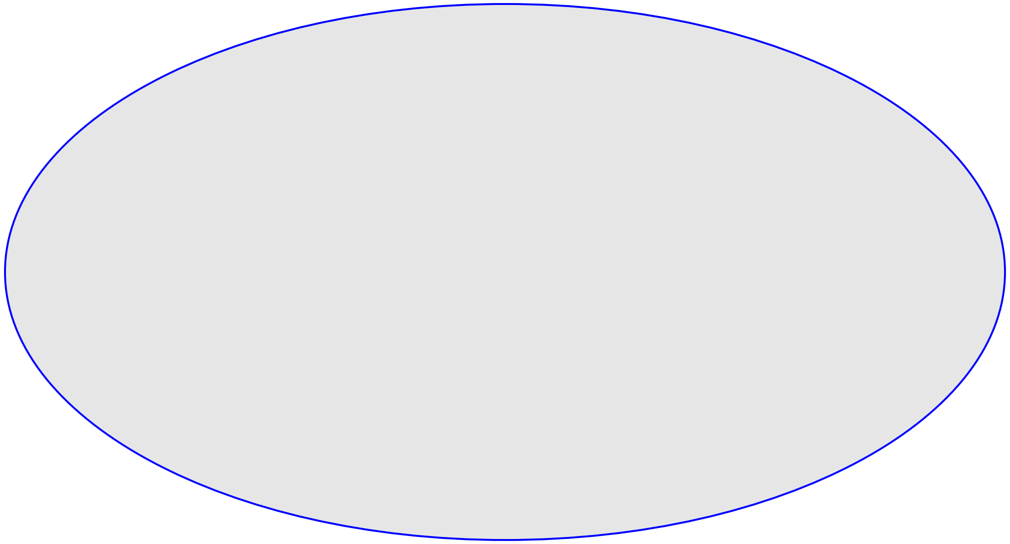
Question ?

If a system passes all of its tests, is it free of all faults ?

No!

- Faults may be hiding in portions of code that only rarely get executed.
- "Testing can only be used to prove the existence of faults not their absence" or "not all faults have failures"
- Sometimes faults mask each other resulting in no visible failures.
- However, if we do a good job in creating a test set that
  - Covers all functional capabilities of a system
  - And covers all code using a metric such as "branch coverage"
- Then, having all test pass increase our confidence that our system has high quality and can be deployed

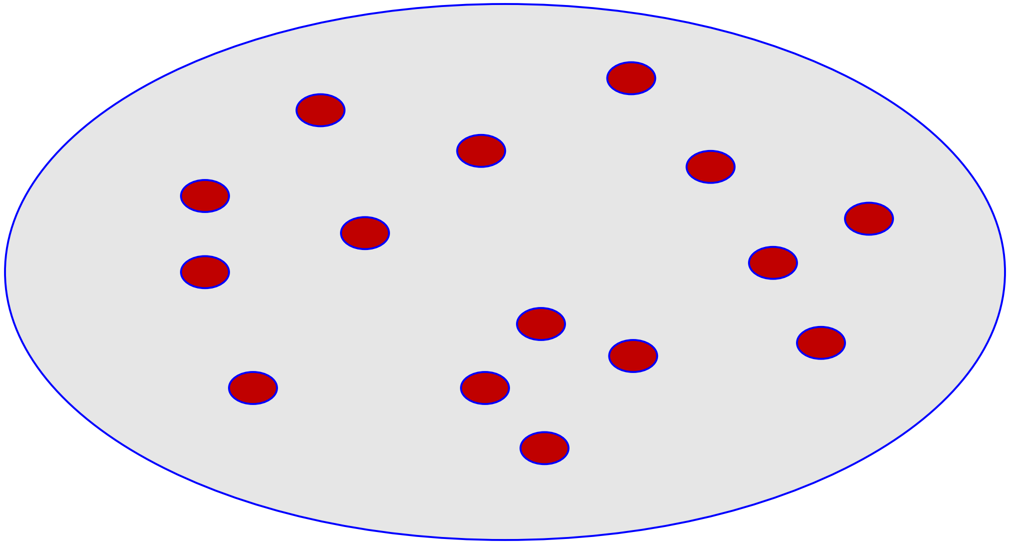
Looking for Faults



All possible states/ behaviors of a system

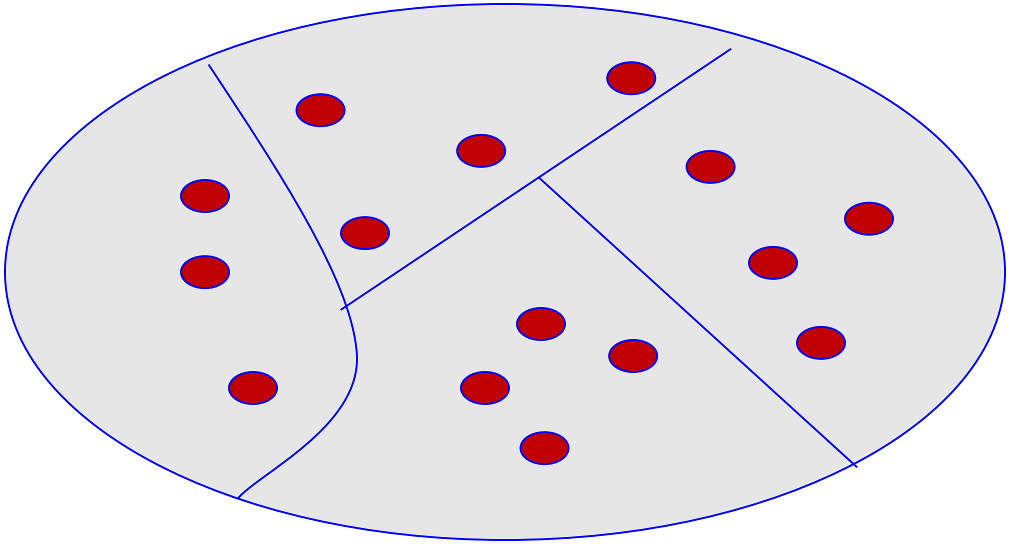
Looking for Faults

As you can see, it is not very comprehensive



Tests are a way of sampling the behaviors of a system, looking for failures

One way forward ? Fold



The testing literature advocates folding the space into equivalent behaviors and then sampling each portion

Fold ? What does that mean ?

Consider a simple example like the Greatest Common Divisor function:

int gcd(int x, int y)

At first glance, this function has an infinite number of test cases

But, let's fold the space

x= 6, y=9, returns 3, tests common case

x= 2 y=4, returns 2, tests when x is the GCD

x= 3 y=5, returns 1, tests two primes

x= 9 y=0, returns ? , tests zero

x= -3 y=9, returns ?, tests negative

Completeness

From this discussion, it should be clear that **"completely"** testing a system is impossible

- So, we settle for heuristics
  - Attempt to fold the input space into different functional categories
  - Then, create tests that sample the behavior/output for each functional partition

As we will see, we also look at our coverage of the underlying code; are we hitting all statements, all branches, all loops ?

Continuous Testing

- Testing is a continuous process that should be performed at every stage of a software development process
  - During requirement gathering, for instance, we must continually query the user “Did we get this right?”

- Facilitated by an emphasis on iteration throughout a life cycle
  - At the end of each iteration
    - We check our results to see if what we built is meeting our requirements (specification)

Fault, Error, and Failure

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

Fault: Should start searching at 0, not 1

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

Fault, Error, and Failure

- The state of numZero consists of the values of the variables x, count, i, and the program counter.
- Consider what happens with numZero ([2, 7, 0]) and numZero ([0, 7, 2])?

The Term Bug

- *Bug* is used informally
- Sometimes speakers mean fault, sometimes error, sometimes failure … often the speaker doesn't know what it means !
- This class will try to use words that have precise, defined, and unambiguous meanings

Fault & Failure Model

- Three conditions must be satisfied for a failure to be observed
  - Reachability : The location or locations in the program that contain the fault must be reached
  - Infection : The state of the program must be incorrect
  - Propagation : The infected state must propagate to cause some output of the program to be incorrect

Static Analysis & Dynamic Testing

- Static Analysis: Testing without executing the program.
  - Code walkthrough & inspection, and various static analysis techniques.
- Dynamic Testing: Testing by executing the program with real inputs
  - Static information can often be used to make dynamic testing more efficient.

Test Case

- Test data: data values to be input to the program under test

- Expected result: the outcome expected to be produced by the program under test

Testing the System(I)

- Unit Tests
  - –Tests that cover low-level aspects of a system
  - –For each module, does each operation perform as expected
  - –For Method foo(), we would like to see another method testFoo()
- Integration Tests
  - –Tests that check that modules work together in combination
  - –Most projects are on schedule until they hit this point (Brookes)
    - All sorts of hidden assumptions are surfaced when code written by different developers.
  - –Lack of integration testing has led to performance failures (Mars Polar Lander)

Testing the System(II)

- System Tests
  - Tests performed by the *developer* to ensure that all major functionality has been implemented
    - Have all user stories been implemented and function correctly ?
- Acceptance Tests
  - Tests performed by the *user* to check that the delivered system meets their needs
    - In Large, custom projects, developers will be on-site to install system and then respond to problems as they arise.

Verification & Validation

- Validation: Ensure compliance of a software product with *intended usage* (Are we building the right product ?)
- Verification: Ensure compliance of a software product with its *design phase* (does X meets its specification ?)
  - Where “X”, can be code, a model, a design diagram, a requirement, …
- Question: Which task, validation or verification, is more difficult to perform?

IV&V stands for “*independent verification and validation*”

Quality Attributes

- Static attributes refer to the actual code and related documentation
  - Well-structured, maintainable, and testable code
  - Correct and complete documentation

- Dynamic attributes refer to the behavior of the application while in use
  - Reliability, correctness, completeness, consistency, usability, and performance

Testability

- The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met
- The more complex an application, the lower the testability, i.e., the more effort required to test it
- Design for testability: Software should be designed in a way such that it can be easily tested

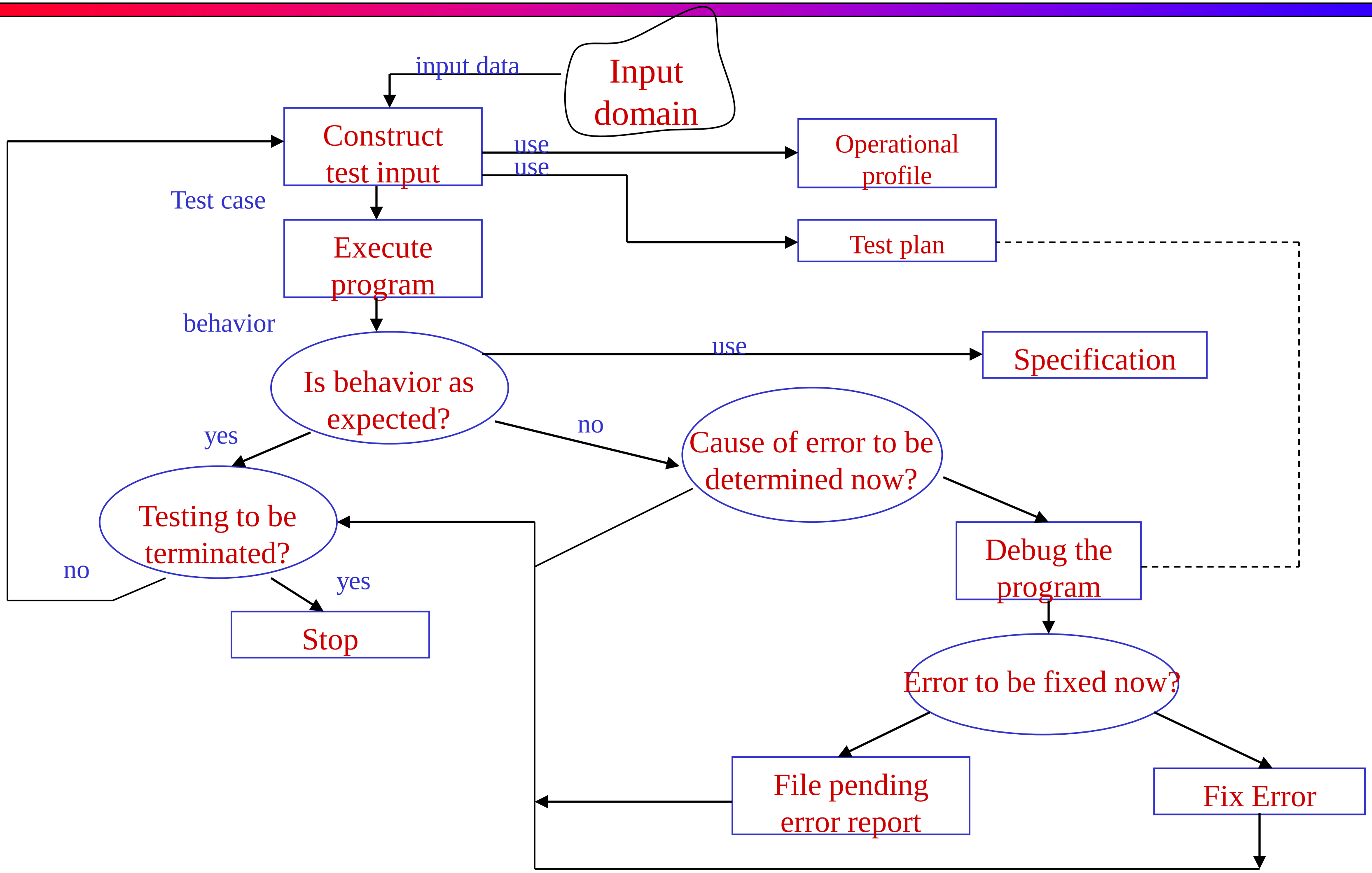
Outline

- Introduction
- Basic Concepts
- The Testing Process
- Types of Testing
- Testing Philosophy
- Summary

The Testing Process

- Preparing a test plan
- Constructing test data
- Specifying program behavior
- Executing the program
- Evaluating program behavior
- Construction of automated oracles

Test & Debug Cycle



An Example

- Program sort:
  - Given a sequence of integers, this program sorts the integers in either ascending or descending order.
  - The order is determined by an input request character “A” for ascending or “D” for descending.

Test plan

1. Execute the program on at least two input sequences, one with "A" and the other with "D" as request characters
2. Execute the program on an empty input sequence
3. Test the program for robustness against invalid inputs such as "R" typed in as the request character
4. All failures of the test program should be reported

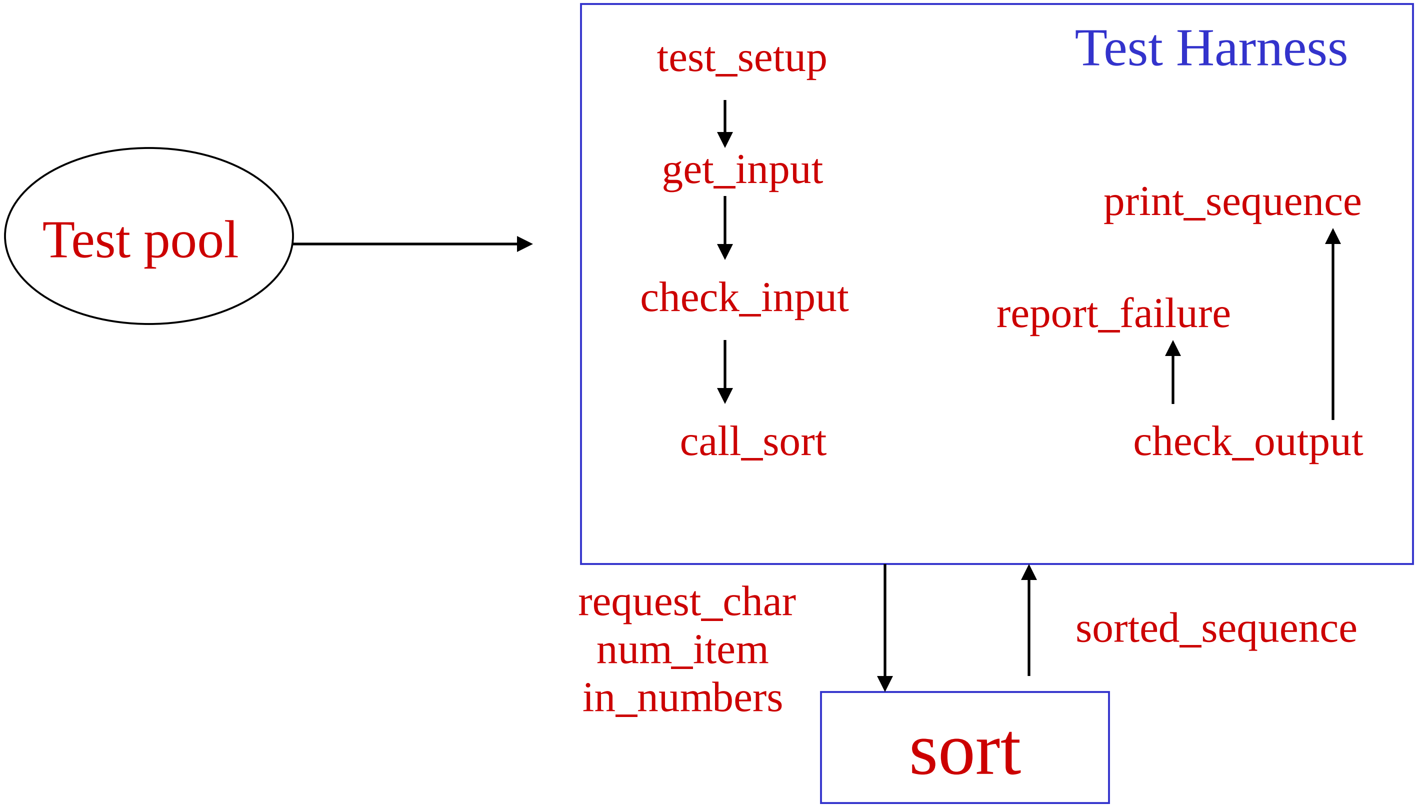
Test Data

- Test case 1:
  - Test data: <“A” 12 -29 32 .>
  - Expected output: -29 12 32
- Test case 2:
  - Test data: <“D” 12 -29 32 .>
  - Expected output: 32 12 -29
- Test case 3:
  - Test data: <“A” .>
  - Expected output: No input to be sorted in ascending order.
- Test case 4:
  - Test data: <“D” .>
  - Expected output: No input to be sorted in ascending order.
- Test case 5:
  - Test data: <“R” 3 17 .>
  - Expected output: Invalid request character
- Test case 6:
  - Test data: <“A” c 17.>
  - Expected output: Invalid number

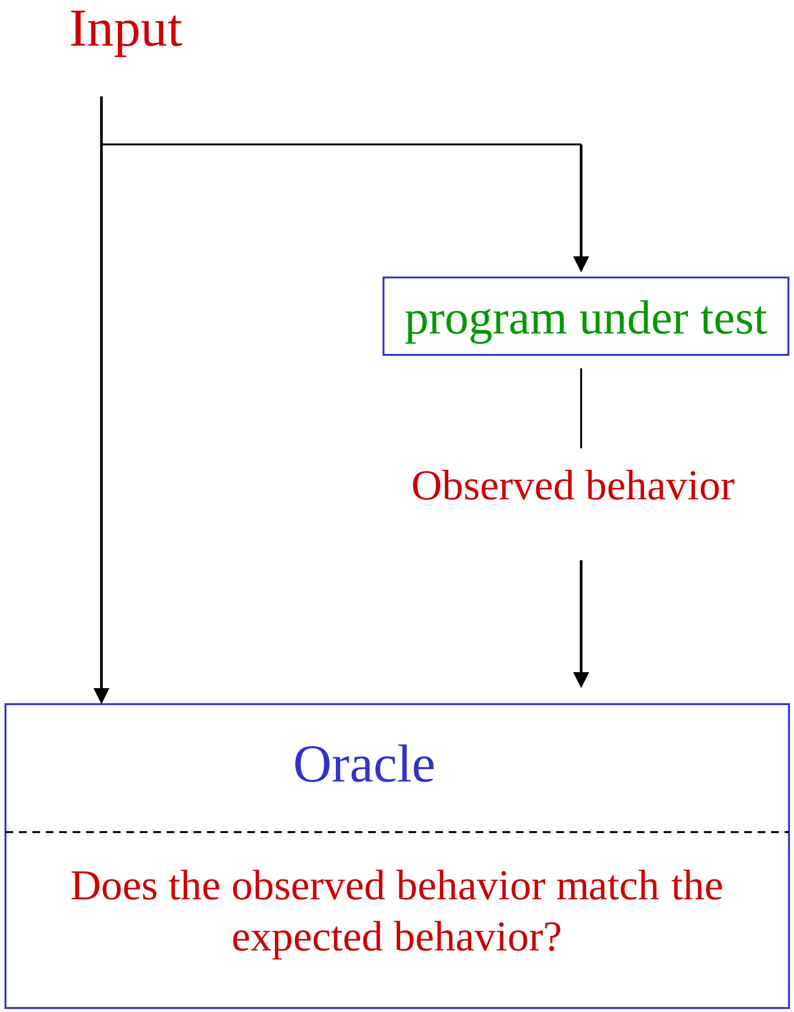
Test Harness

- In software testing, a Test harness or Automated test framework is :
  - A collection of software and test data configured to test a program unit by running it under different conditions and monitoring its behavior and outputs

Test Harness



Test Oracle


Input
program under test
Observed behavior
Oracle
Does the observed behavior match the expected behavior?

Outline

- Introduction
- Basic Concepts
- The Testing Process
- Types of Testing
- Testing Philosophy
- Summary

Multi-Level Testing

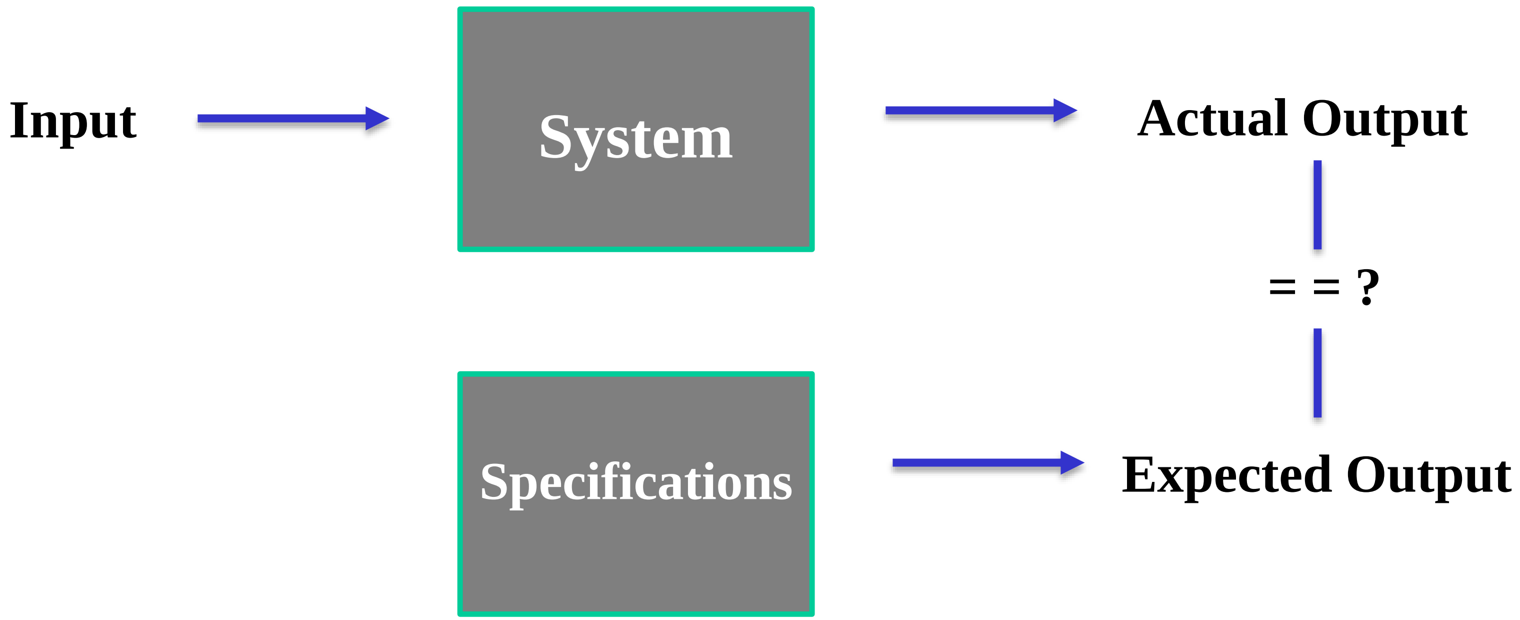
Once we have code, we can perform three types of tests :

- *Black Box Testing*
  - Does the system behave as predicted by its specification
- *Grey Box Testing*
  - Having a bit of insight into the *architecture* of the system, does it behave as predicted by its specification
- *White Box testing*
  - Since, we have access to most of the code, let's make sure we are covering all aspects of the code: Statements, branches, …

Source of Test Generation

- Black-box testing: Tests are generated from informally or formally specified requirements
  - Does not require access to source code
  - Boundary-value analysis, equivalence partitioning, random testing, pairwise testing
- White-box testing: Tests are generated from source code.
  - Must have access to source code
  - Structural testing, path testing, data flow testing

Black Box Testing



- A black box test passes input to a system, records the actual output and compares it to the expected output

Results

- If actual output == expected output
  - Test passed
- else
  - Test Failed

- Process
  - Write at least one test case per functional capability
  - Iterate on code until all tests pass
- Need to automate this process as much as possible

Black Box Categories

- Functionality
  - User input validation (based off specification)
  - Output results
  - State transitions
    - Are there clear states in the system in which the system is supposed to behave differently based on the state ?
  - Boundary cases and off-by-one errors

Grey Box Testing

- Use knowledge of system's architecture to create a more complete set of black box tests
  - Verifying auditing and logging information
    - For each function is the system really updating all internal state correctly
- Data destined for other systems
- System-added information (timestamp, checksum, etc.)
- "Looking for scarps"
  - Is the system currently cleaning up after itself
    - Temporary files, memory leaks, data duplication/ deletion

White boxing Testing

- Writing test cases with complete knowledge of code
  - Format is the same: input, expected output, actual output
- But, now we are looking at
  - Code coverage (more on this in a minute)
  - Proper error handing
  - Working as documented (is method "foo" thread safe?)
  - Proper handling of resources
    - How does the software behave when resources become constrained ?

Code coverage(I)

- A criteria for knowing white testing is "complete"
  - Statement coverage
    - Write tests until all statements have been executed
  - Branch Coverage (edge coverage)
    - Write tests until each edge in a program's control flow graph has been executed at least once (covers true/false conditions)
  - Condition coverage
    - Like branch coverage but with more attention paid to the conditionals (if compound conditionals, ensure that all combinations have been covered)

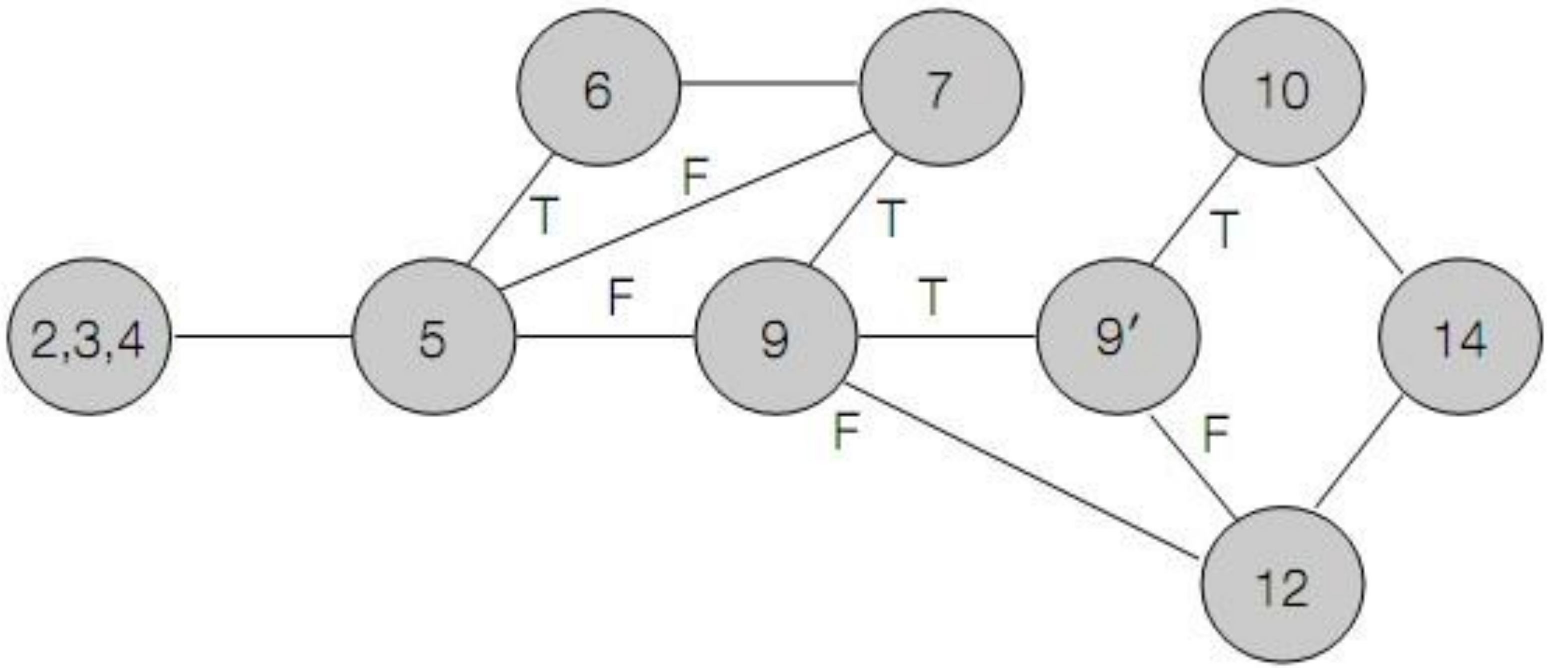
Code Coverage (II)

- A criteria for knowing white box testing is "complete"
- Path coverage
  - Write tests until all paths in program's control flow graph have been executed multiple times as dictated by heuristics e.g.
- For each loop, write a test case that executes the loop
  - Zero times (Skips the loop)
  - Exactly one time
  - More than once (exact number depends on context)

A sample Program

```
Public int P()
{
int x,y;
x = Console.Read(); y = Console.Read();
while(x>10){
   x = x-10;
   if (x==10) break;
}
if (y < 20 && ( x % 2) == 0){
   y = y +20;
}else{
y = y-20;
}
Return 2 * x + y;
}
```

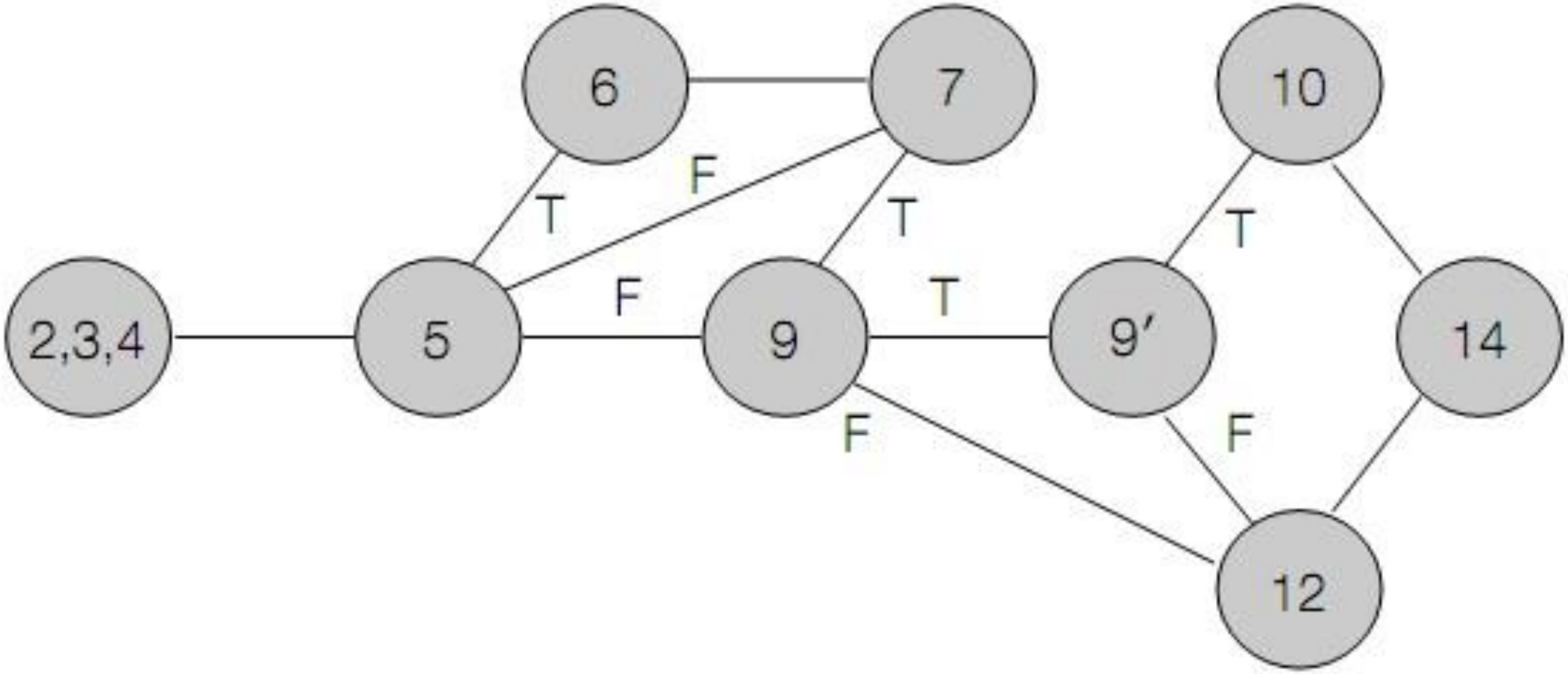
P's Control Flow Graph (CFG)



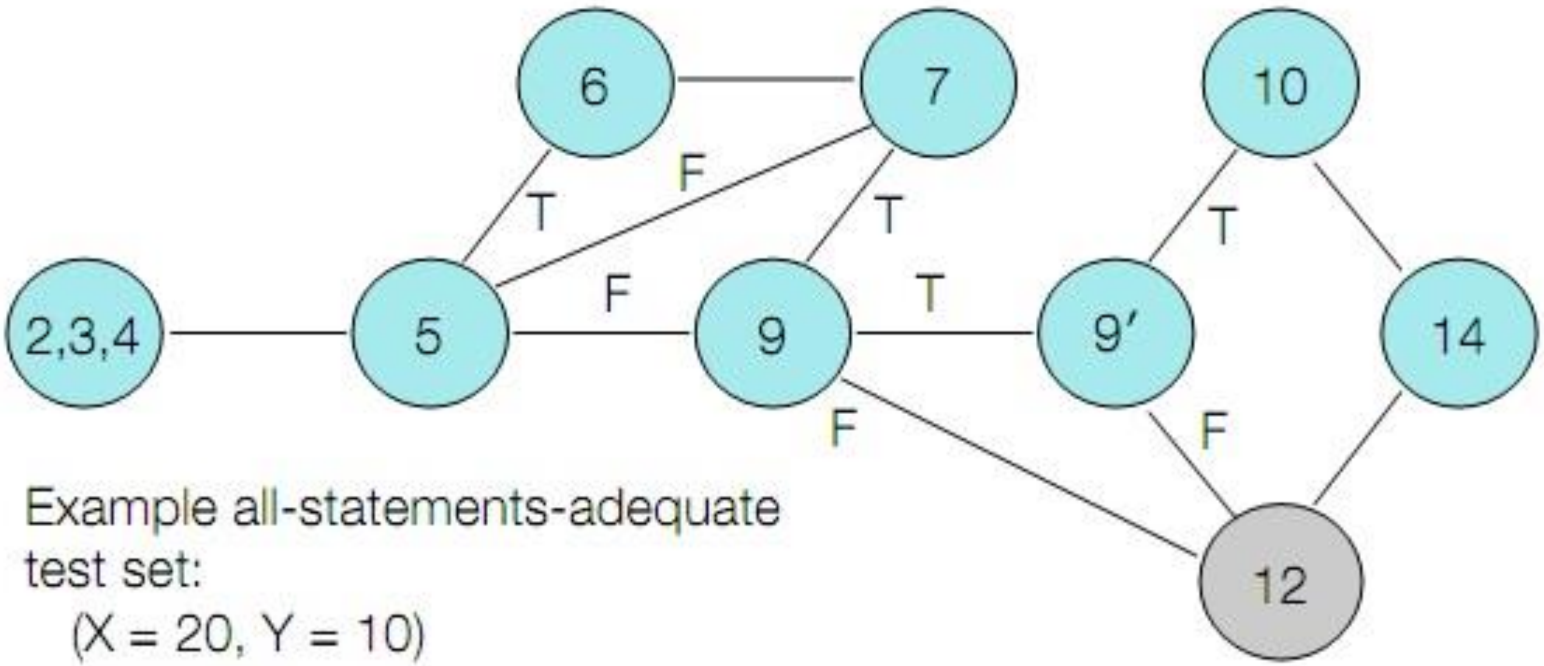
White boxing Criteria

- Statement Coverage
  - Create a test set T such that
    - By Executing P for each t in T
    - Each elementary statement of P is executed at least once

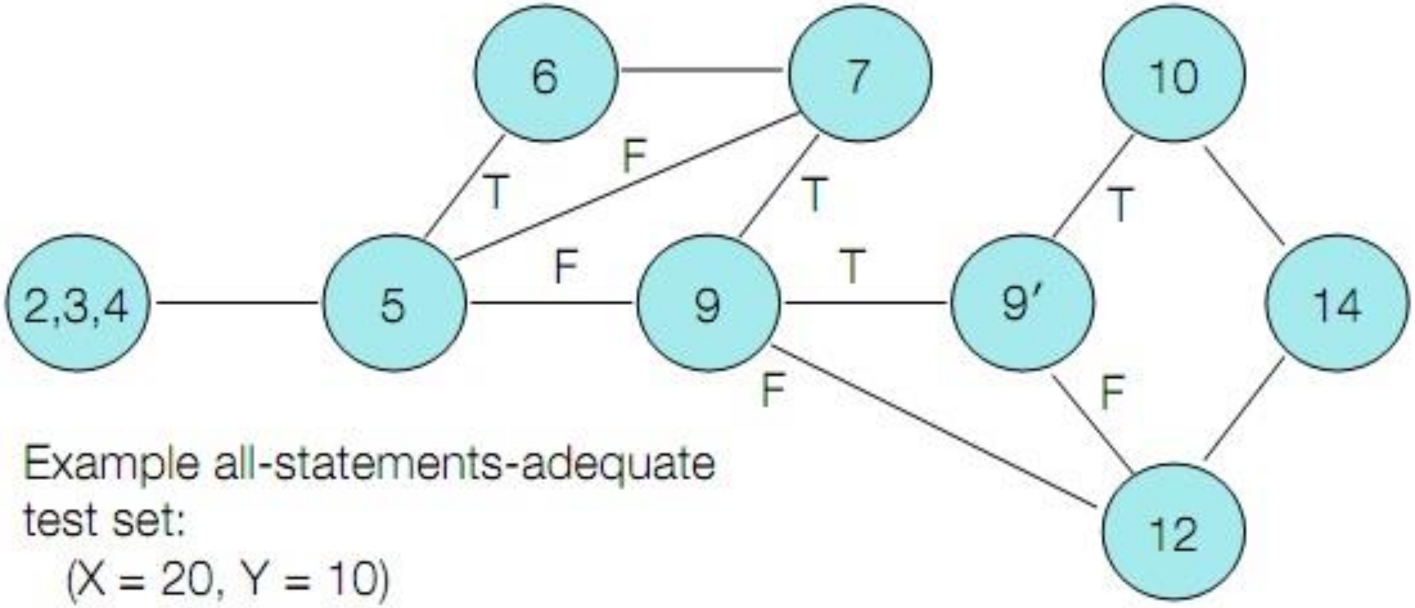
All statement coverage of P



All statement coverage of P



All statement coverage of P

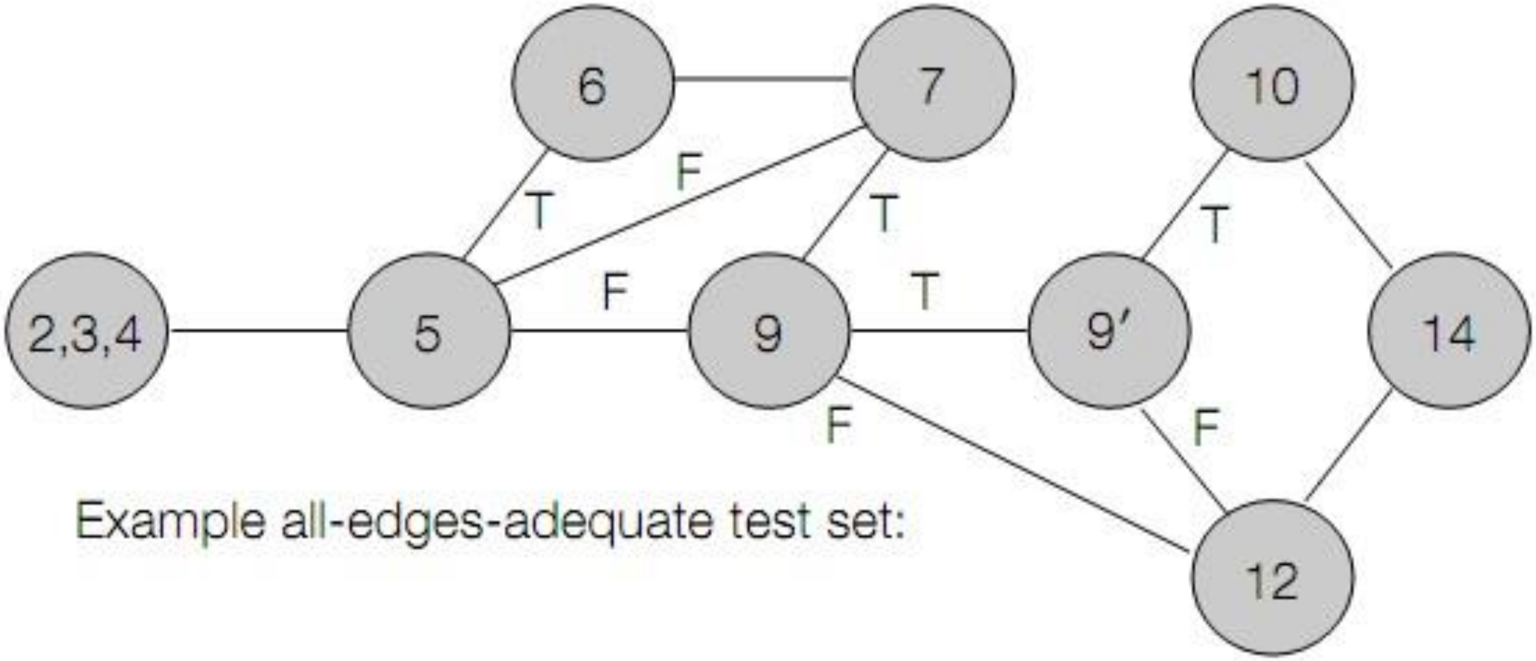


Example all-statements-adequate
test set:
(X = 20, Y = 10)
(X = 20, Y = 30)

White boxing Testing Criteria

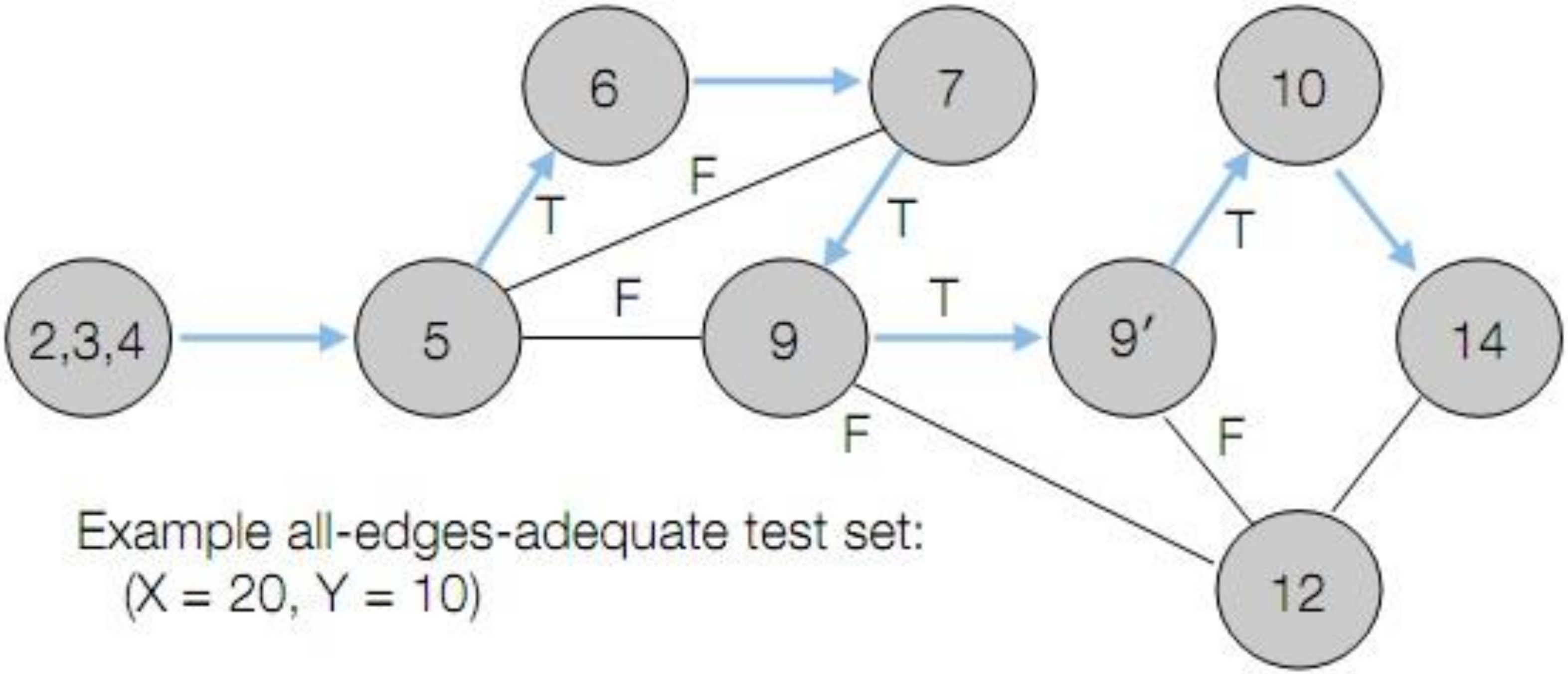
- Edge Coverage
- Select a test set T such that
  - By executing P for each t in T
  - Each edge of P's control flow graph is traversed at least once

All-Edge Coverage of P



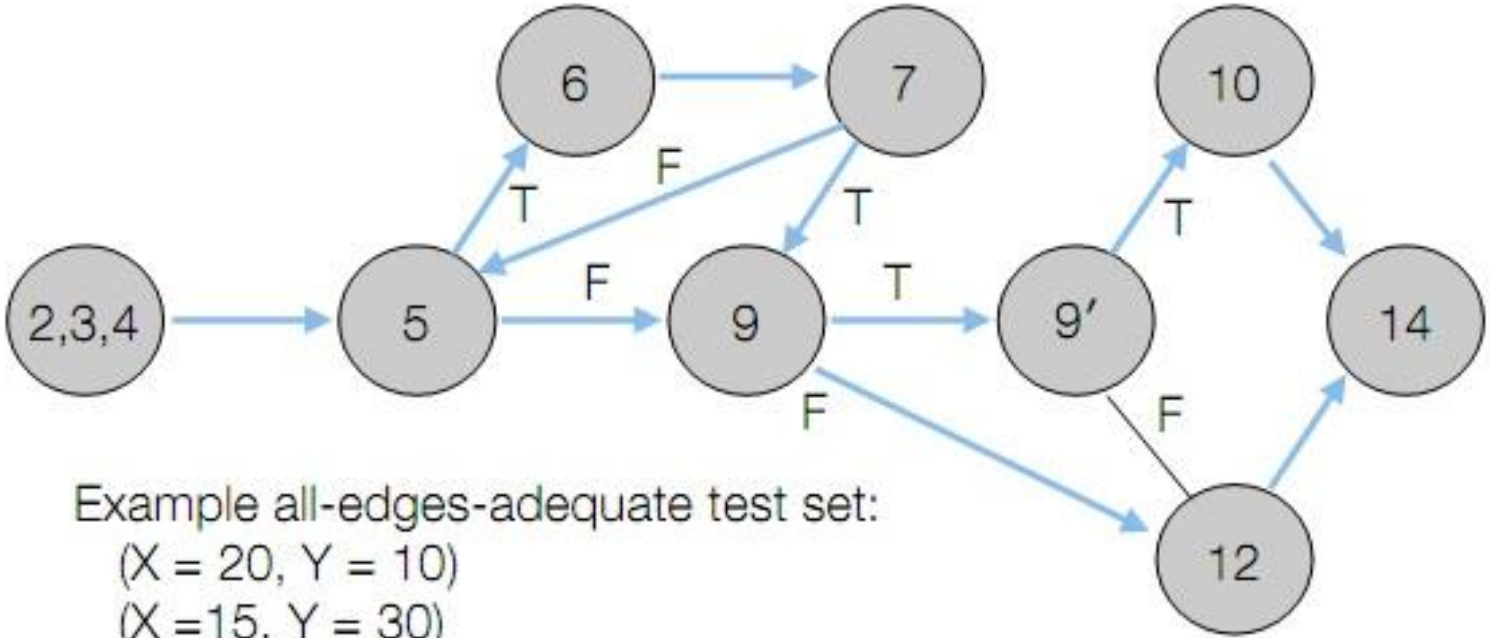
Example all-edges-adequate test set:

All-Edge Coverage of P



Example all-edges-adequate test set:
(X = 20, Y = 10)

All-Edge Coverage of P

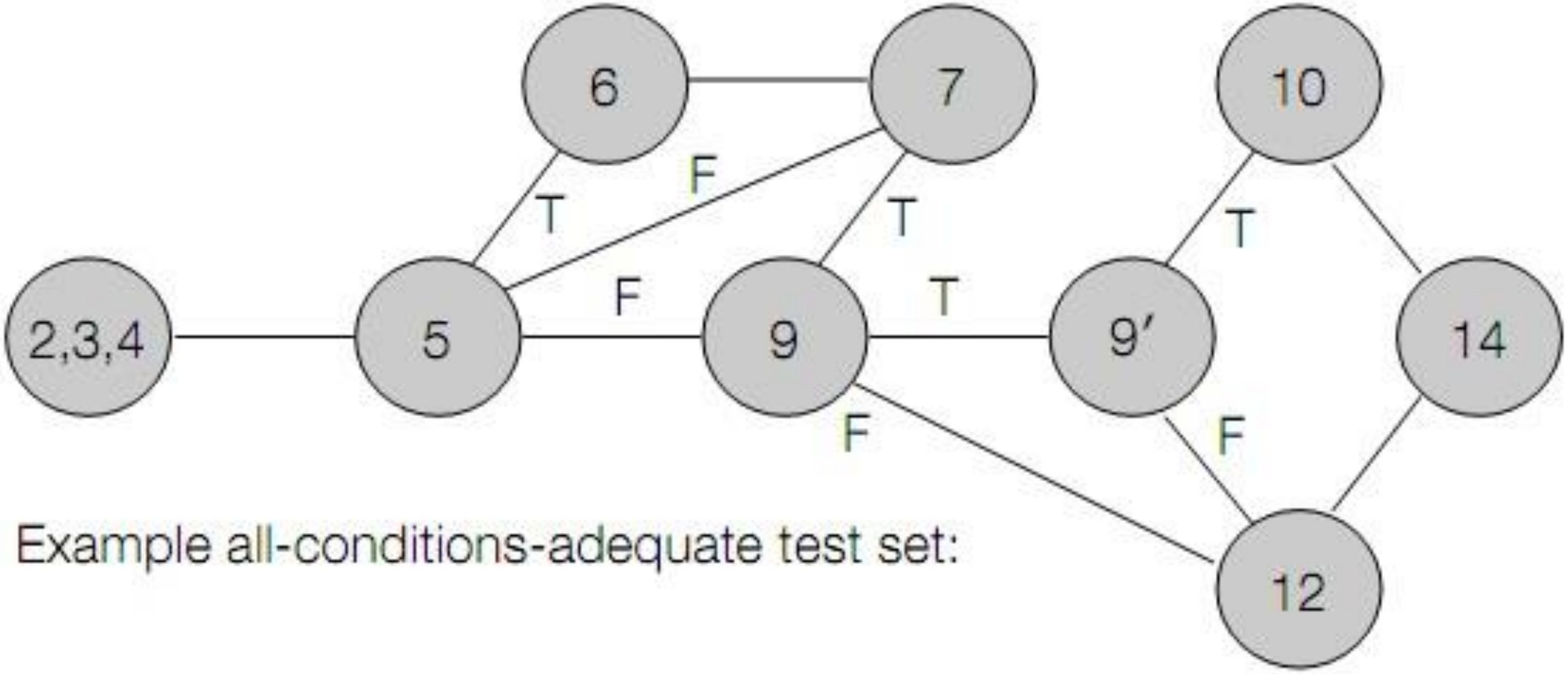


Example all-edges-adequate test set:
(X = 20, Y = 10)
(X =15, Y = 30)

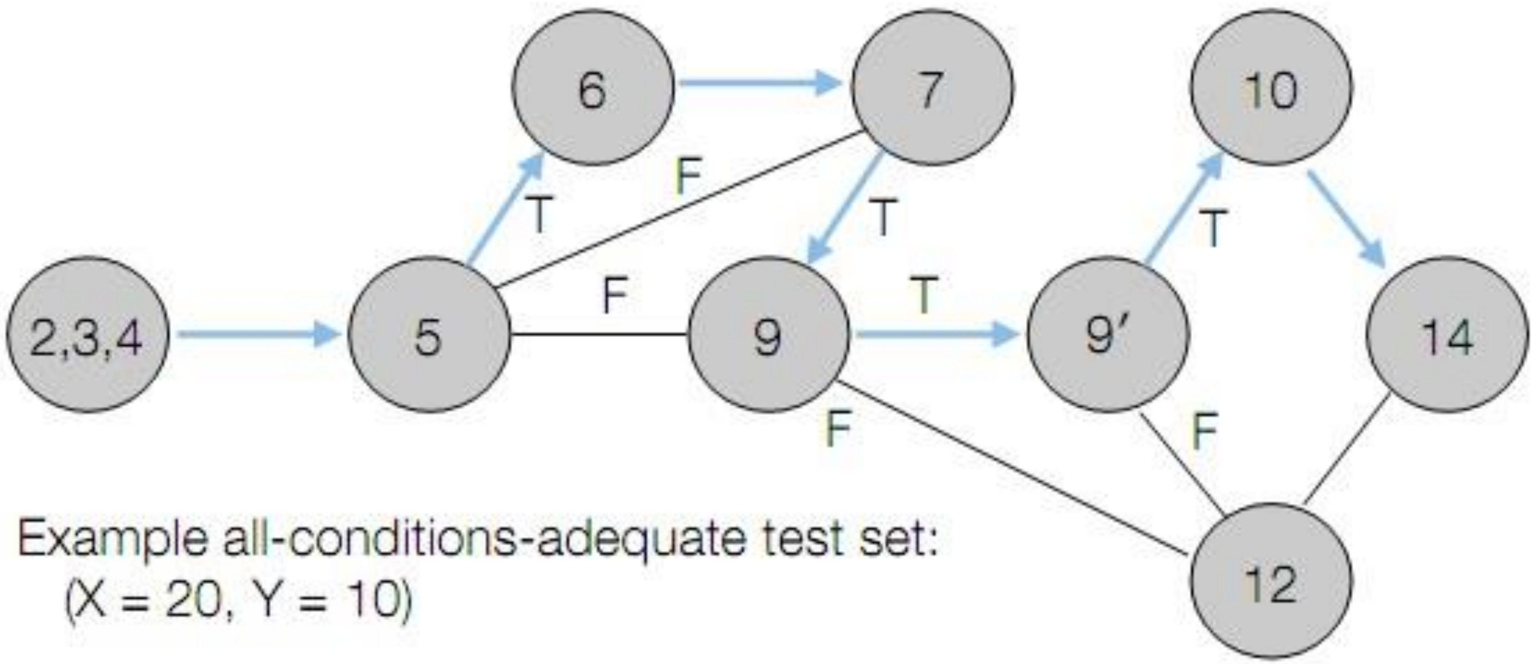
White-box Testing Criteria

- Edge Coverage
- Select a test set T such that
  - By executing P for each t in T
  - Each edge of P's control flow graph is traversed at least once
  - And all possible values of the constituents of compound conditions are exercised at least once

All- Conditions Coverage of P

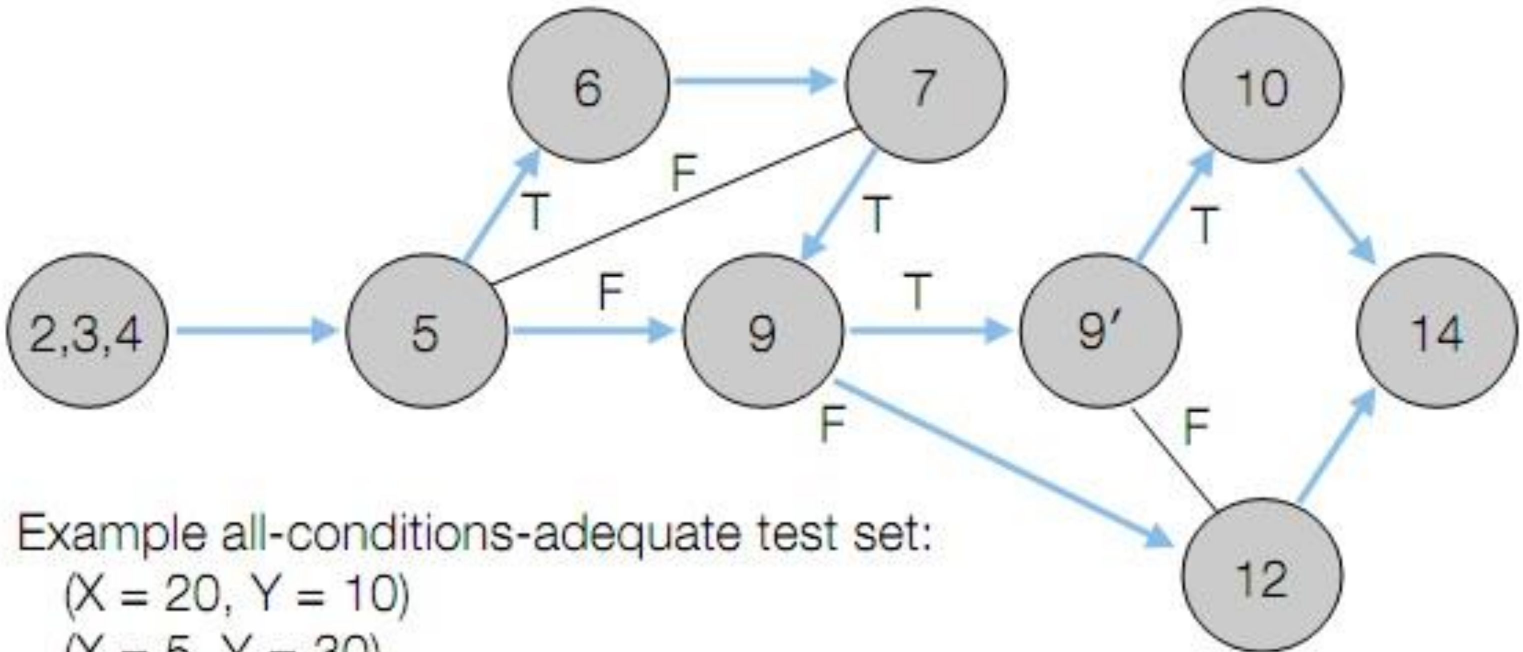


All- Conditions Coverage of P



Example all-conditions-adequate test set:
(X = 20, Y = 10)

All- Conditions Coverage of P

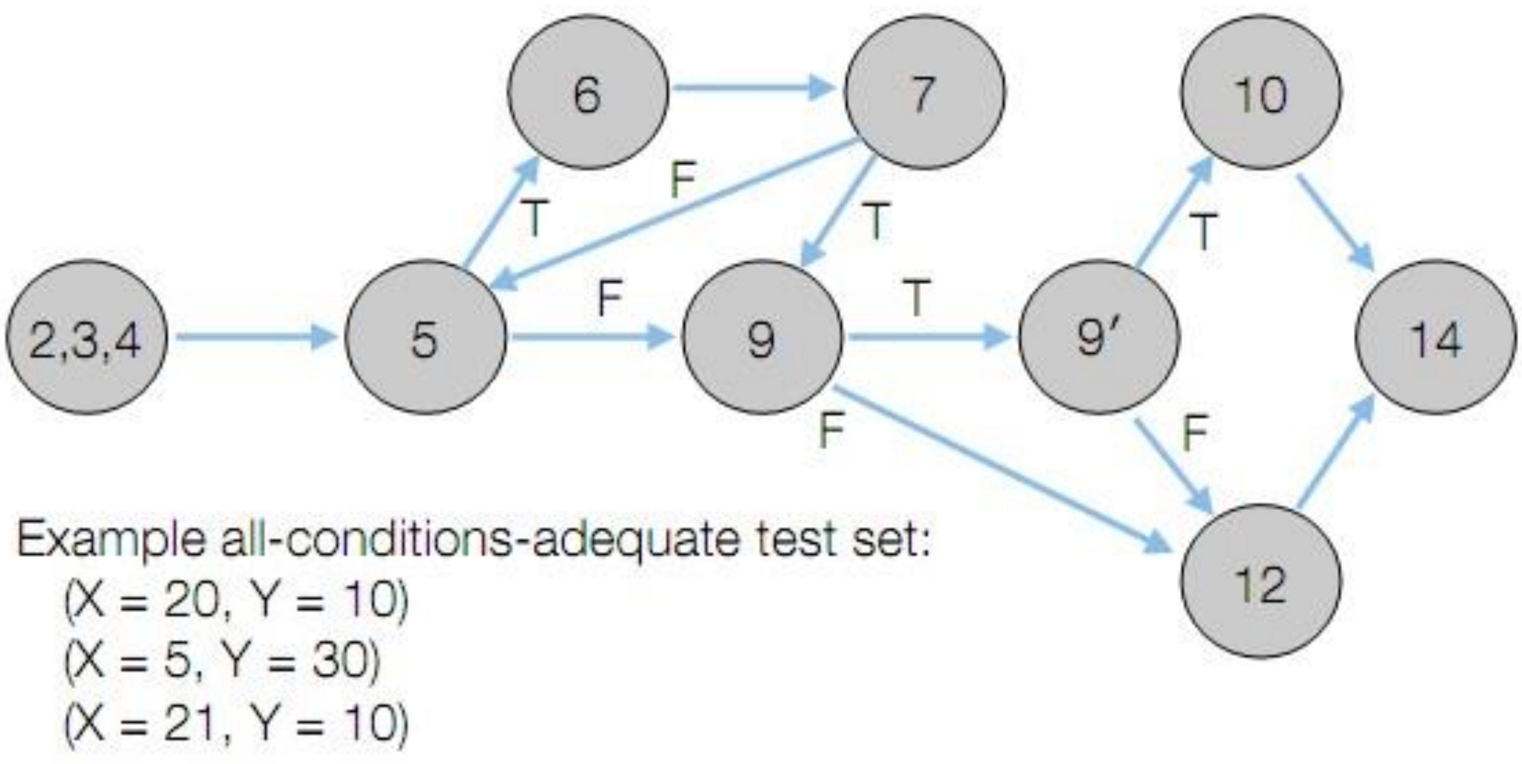


Example all-conditions-adequate test set:

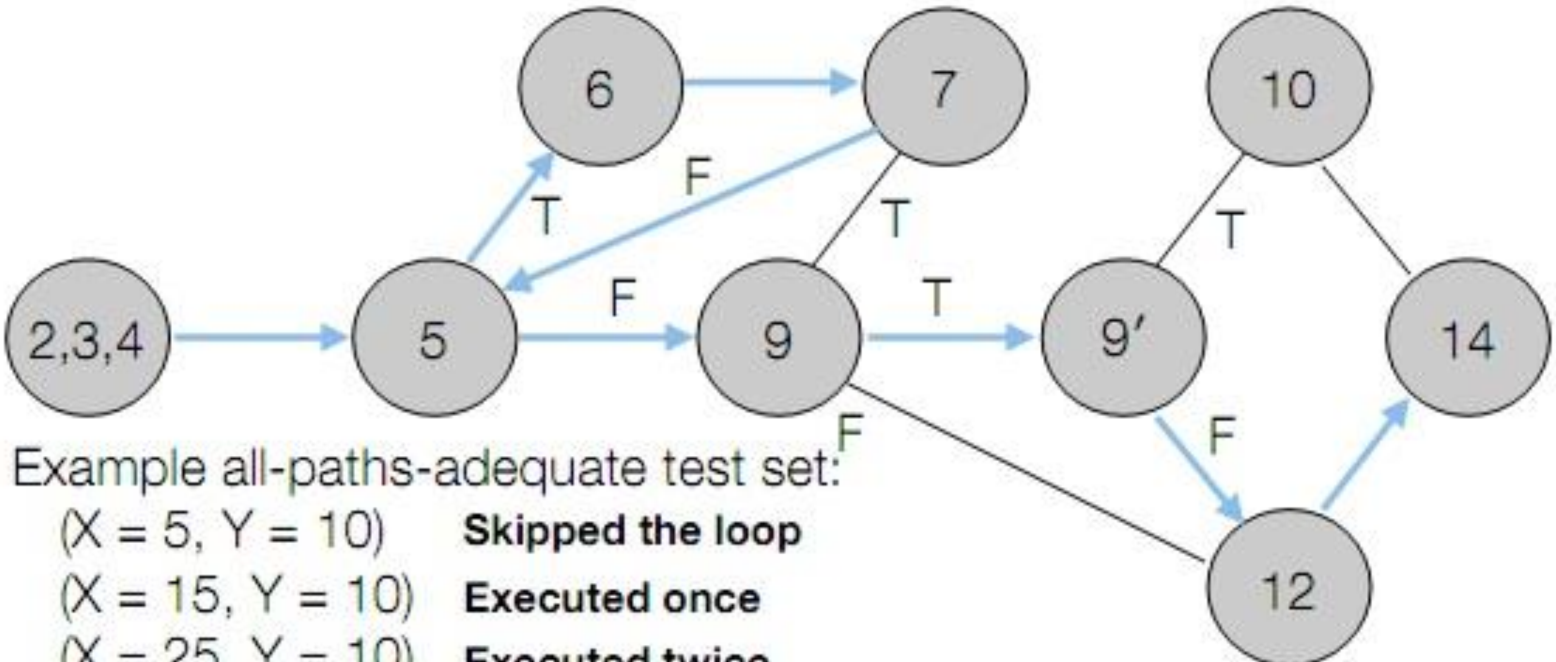
(X = 20, Y = 10)

(X = 5, Y = 30)

All- Conditions Coverage of P



All- Conditions Coverage of P



Example all-paths-adequate test set:

(X = 5, Y = 10) **Skipped the loop**

(X = 15, Y = 10) **Executed once**

(X = 25, Y = 10) **Executed twice**

And so on… you would also want permutations that exit the loop early

Black-box testing

- Boundary Value Analysis : a software testing technique in which tests are designed to include representatives of boundary values in a range.
- Random testing : a black-box software testing technique where programs are tested by generating random, independent inputs.
- Pairwise testing: combinational method of software testing for each pair of input parameters to a system. Other words, tests all possible discrete combinations of those parameters
- Equivalence partitioning : next slides

Pairwise testing- Example

Parameter name	Value 1	Value 2	Value 3	Value 4
Enabled	True	False	*	*
Choice type	1	2	3	*
Category	a	b	c	d

'Enabled', 'Choice Type' and 'Category' have a choice range of 2, 3 and 4, respectively. An exhaustive test would involve 24 tests (2 x 3 x 4).
Multiplying the two largest values (3 and 4) indicates that a pair-wise tests would involve 12 tests. The pict (Pairwise Independent Combinatorial Tool) tool generated pairwise test cases and test configurations.

Enabled	Choice type	Category
True	3	a
True	1	d
False	1	c
False	2	d
True	2	c
False	2	a
False	1	a
False	3	b
True	2	b
True	3	d
False	3	c
True	1	b

Boundary Value Analysis (BVA) - Example

Date 1-31

0	1	31	32
Boundary value just below the boundary	Boundary value just above the boundary	Boundary value just below the boundary	Boundary value just above the boundary
Invalid partition – Valid partition Lower Boundary		Invalid partition – Valid partition Upper Boundary	

Equivalence partitioning

- Divides the input data of a software unit into partitions of equivalent data from which test cases can be derived.
- This technique aims to define test cases that uncover classes of errors, thereby reducing the total number of test cases that must be developed.
- Equivalence partitioning is usually applied to the inputs of a tested component.
- Advantage :
  - Reduction in the time required for testing a software due to lesser number of test cases.

Equivalence Partitioning cont.

- Test Each Partition once (the assumption is that any input in a partition is equivalent)
- Example – Date (1-31) It takes one value from each partition to test.

Invalid Partition	Valid Partition	Invalid Partition
0	1	32
-2	2	33
-4	3	34
-6	..	…
…	31	

Equivalence Partitioning cont.

- Example – Age (18-80 Excepts 60 to 65 years)

Invalid Partition	**valid Partition**	Invalid Partition	**valid Partition**	Invalid Partition
0	18	60	66	81
1	…	…	…	82
…	59	65	80	83
17				…

White-box Testing

1. Structural Testing:
   – Tests are derived from the knowledge of the software's structure or internal implementation (source code)

- Structural Testing Techniques:
  – Statement Coverage
    - Exercising all programming statements with minimal tests
  – Branch Coverage
    - Running a series of tests to ensure that all branches are tested at least once
  – Path Coverage
    - all possible paths which means that each statement and branch are covered

White-box Testing Cont.

- Advantages of Structural Testing
  - Forces test developer to reason carefully about implementation
  - Reveals errors in "hidden" code
  - Spots the Dead Code or other issues with respect to best programming practices.

- Disadvantage of Structural Testing
  - Expensive as one has to spend both time and money to perform white box testing.
  - Every possibility that few lines of code is missed accidentally.
  - In-depth knowledge of programming language is necessary to perform white box testing.

White-box Testing Cont.

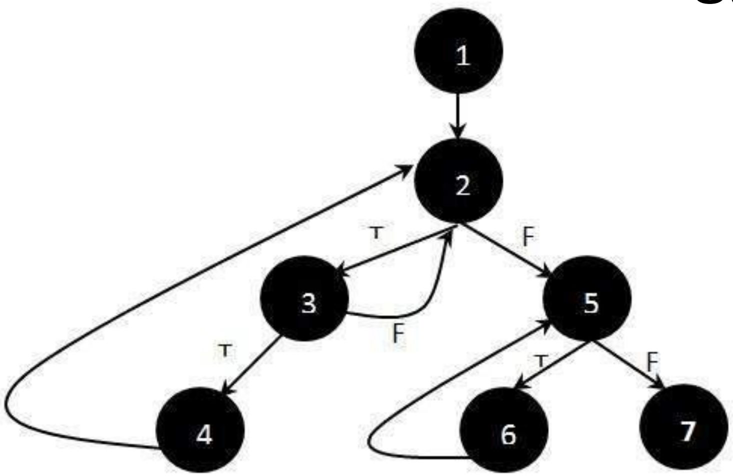
2. Path Testing:

- Designing test cases intended to examine all possible paths of execution at least once.

- Calculate the independent paths :
  - Step 1 : Draw the Flow Graph of the Function/Program
  - Step 2 : Determine the independent paths.

White-box Testing Cont.

Step 1 : Draw the Flow Graph of the Function/Program



Step 2 : Determine the independent paths.

```
Path 1:  1 - 2 - 5 - 7
Path 2:  1 - 2 - 5 - 6 - 7
Path 3:  1 - 2 - 3 - 2 - 5 - 6 - 7
Path 4:  1 - 2 - 3 - 4 - 2 - 5 - 6 - 7
```

White-box Testing Cont.

3. Data Flow Testing:

- Selecting paths through the program's control flow in order to explore sequences of events related to the status of variables or data objects

- Dataflow Testing focuses on the points at which variables receive values and the points at which these values are used.

White-box Testing Cont.

Advantage of Data Flow Testing :

Data Flow testing helps us to catch any of the following issues:

- A variable that is declared but never used within the program.
- A variable that is used but never declared.
- A variable that is defined multiple times before it is used.
- Deallocating a variable before it is used.

Test Automation

- It is important that your tests be automated
  - More likely to be run
  - More likely to catch problems as changes are made
- As the number of tests grow, it can take a long time to run the tests, so it is important that the running time of each individual test is as small as possible
  - If that's not possible to be achieve then
  - Segregate long running tests from short running tests
    - Execute the latter multiple times per day (short)
    - Execute the former at least once per day (they still need to be run!) (long)

Test Automation Cont.

- It is important that running tests be easy
  - Testing frameworks allow tests to be run with single command.
  - Often as part of the build management process

We will see examples of this later in the semester

Continuous Integration (CI)

- Since test automation is so critical, system known as continuous integration frameworks have emerged.
- Continuous integration (CI) systems wrap up version control, compilation, and testing into a single repeatable process.

- You creat/debug code as usual
- You then check your code and the CI system builds your code, test it, and reports back to you

Classifier C2: Life Cycle Phases

PHASE	TECHNIQUE
Coding	Unit Testing
Integration	Integration Testing
System Integration	System Testing
Maintenance	Regression Testing
Postsystem, pre-release	Beta Testing

Classifier C3: Goal Directed Testing

GOAL	TECHNIQUE
Features	Functional Testing
Security	Security Testing
Invalid inputs	Robustness Testing
Vulnerabilities	Penetration Testing
Performance	Performance Testing
Compatibility	Compatibility Testing

Classifier C4: Artifact Under Test

ARTIFACT	TECHNIQUE
OO Software	OO Testing
Web applications	Web Testing
Real-Time software	Real-time testing
Concurrent software	Concurrency testing
Database applications	Database testing

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 expensive (time)

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

Summary: Why Do We Test Software ?

- A tester’s goal is to eliminate faults (root of failure and errors) as early as possible.

- Improve quality
- Reduce cost
- Preserve customer satisfaction

Outline

- Introduction
- Basic Concepts
- The Testing Process
- Types of Testing
- Testing Philosophy
- Summary

Philosophy

- Level 0: Testing is the same as debugging.
- Level 1: Testing aims to show correctness
- Level 2: Testing aims to show the program under test doesn't work
- Level 3: Testing aims to reduce the risk of using the software
- Level 4: Testing is a mental discipline that helps develop higher quality software

Level 0 Thinking

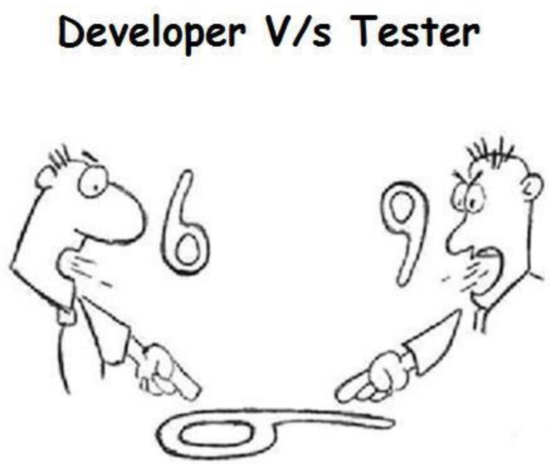
- Testing is the same as debugging
- Does not distinguish between incorrect behavior and mistakes in the program
- Does not help develop software that is reliable or safe

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

Level 2 Thinking

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



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 disastrous
- Testers and developers cooperate to reduce risk

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

Where Are you ?

Are you at level 0, 1, or 2 ?

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

Or 3?

Support for level 4 thinking

Make Testing fun

Outline

- Introduction
- Basic Concepts
- The Testing Process
- Types of Testing
- Testing Philosophy
- Summary

Summary

- Quality is the central concern of software engineering.
- Testing is the single most widely used approach to ensuring software quality.
  - Validation and verification can occur in any phase
- Testing consists of test generation, test execution, and test evaluation.
- Testing can show the presence of failures, but not their absence.
- Testing of code involves
  - Black box, Grey box, and white box tests
  - All require : input expected output, actual output
  - White box additionally looks for code coverage
- Testing of systems involves
  - Unit test, integration test, system test and acceptance tests