CMPS 411

Software Testing

Please read Chapter 16, 17 & 18

Dr. Abdelkarim Erradi

Dept. of Computer Science & Engineering

QU

Outline

- Types of testing
- Black-box testing & White-box testing
- Unit Testing using JUnit

Types of testing

Back

Definition

- Testing is the process of executing a program with the intent of finding errors prior to delivery to the end user.
- We typically begin by ‘testing-in-the-small’ and move toward ‘testing-in-the-large’
  - E.g., OO software we start with testing class methods then integration of classes to deliver a use case

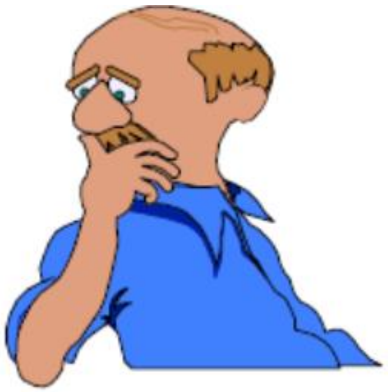
Testing Strategy

- To perform effective testing, you should conduct effective technical reviews. By doing this, many errors will be eliminated before testing commences.
- Testing begins at the component level and works "outward" toward the integration of the entire computer-based system.
- Testing is conducted by the developer of the software and (for large projects) an independent test group.
- Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.

Who tests the software

developer

Understands the system but, will test "gently" and, is driven by "delivery"



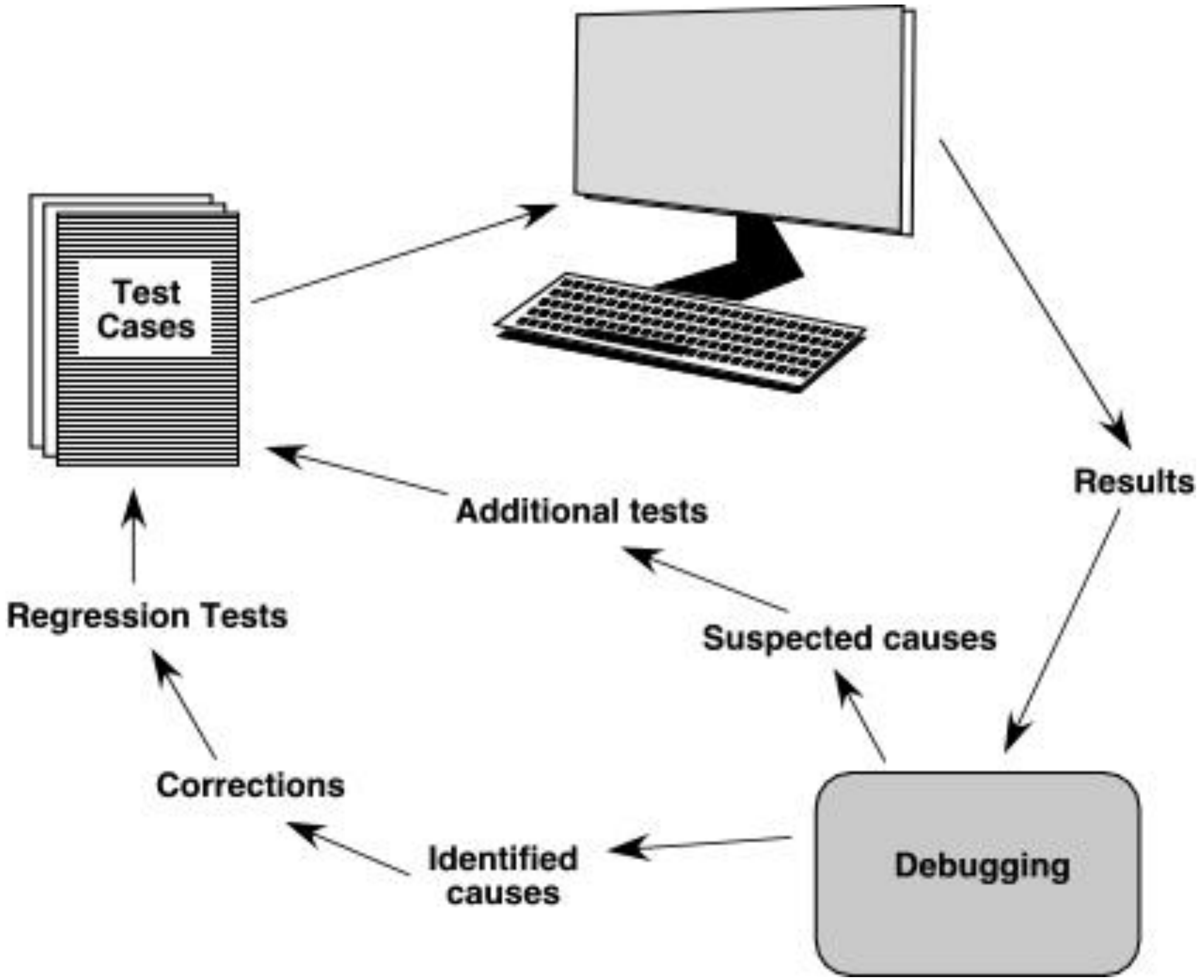
independent tester

Must learn about the system, but, will attempt to break it and, is driven by quality

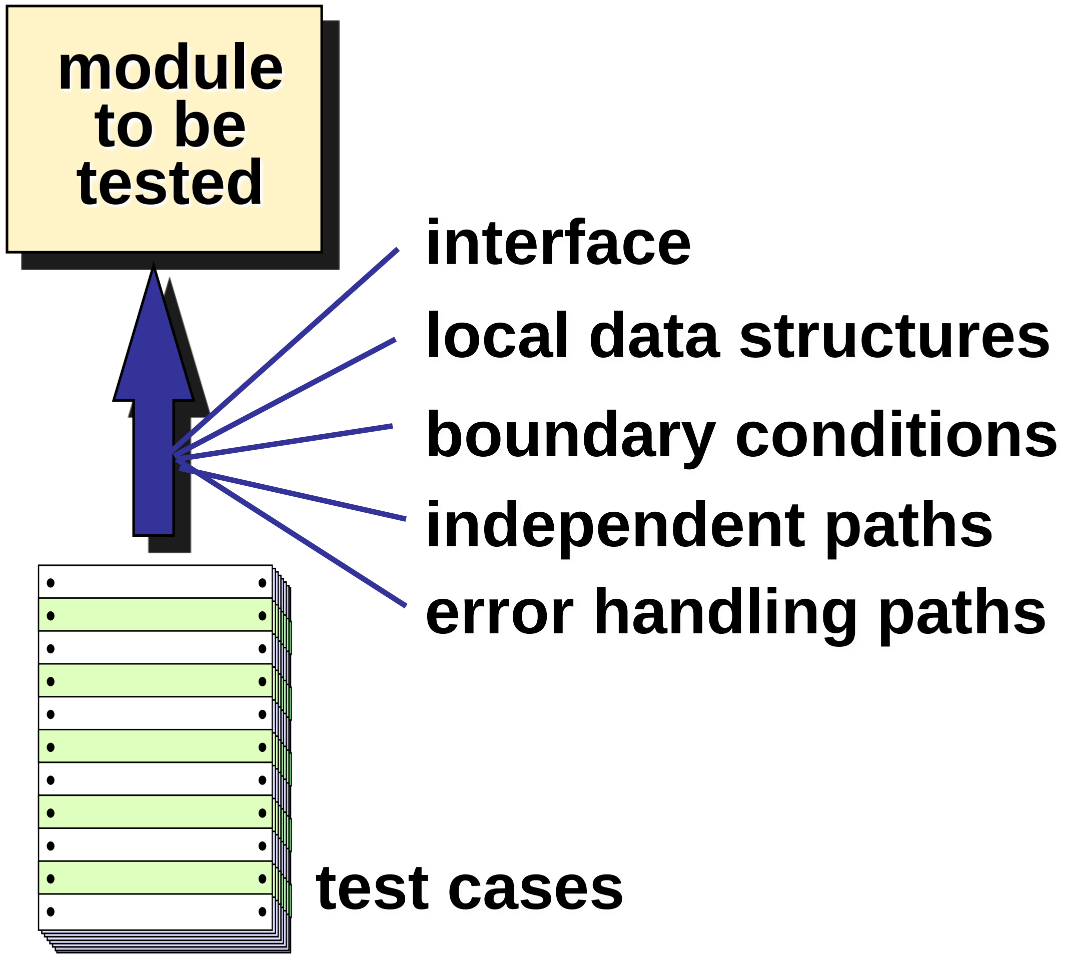
Types of testing

- **System testing**: focus is on system integration
- **Acceptance testing**: focus is on customer usage
- **Security testing**: verifies that protection mechanisms built into a system will, in fact, protect it from improper penetration
- **Stress testing**: executes a system in a manner that demands resources in abnormal quantity, frequency, or volume
- **Performance Testing**: test the run-time performance of software within the context of an integrated system
- **Regression testing**: is the re-execution of some subset of tests that have already been conducted to ensure that changes do not introduce unintended behavior or additional errors.

The Debugging Process: A Diagnostic Process



Unit Testing



OO Testing Strategy

- Class unit testing includes:
  - Methods within the class are tested
  - The state behavior of the class is examined
- Integration applied three different strategies
  - thread-based testing—integrates the set of classes required to respond to one input or event
  - use-based testing—integrates the set of classes required to respond to one use case
  - cluster testing—integrates the set of classes required to demonstrate one collaboration

What are the work products during testing?

- A set of test cases designed to test both internal logic and external requirements is designed and documented, and expected results are defined.

=> "Test Plan"

- Actual testing results are recorded

=> "Test Results"

- Bugs are reported back to the development team for their removal.

=> "Bug or defect" report and Bug "prioritization"

Test cases

- Number
- Name
- Function to be tested
- Description of system state before running the test case
- Input parameter values
- Expected outputs
- Actual output
- Short description (if needed)

Test case examples: ATM

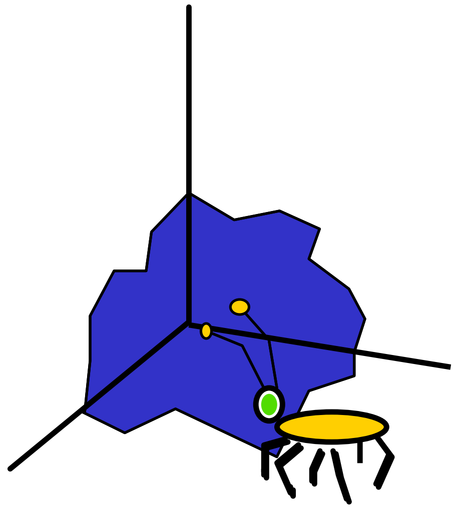
- Number: 1.3.2
- Name: verifyPinSuccess
- Function to be tested:
  - verify PIN
- System state before executing the test:
  - Card inserted
- Input parameter values:
  - correct PIN
- Expected outcome of the test:
  - authentication correct
  - display withdrawal screen

- Number: 1.3.3
- Name: verifyPinFailure
- Function to be tested:
  - verify PIN
- System state before executing the test:
  - Card inserted
- Input parameter values :
  - incorrect PIN
- Expected outcome of the test:
  - display warning message
  - ask for another attempt

Test Case Design

"Bugs lurk in corners
and congregate at
boundaries ..."

Boris Beizer

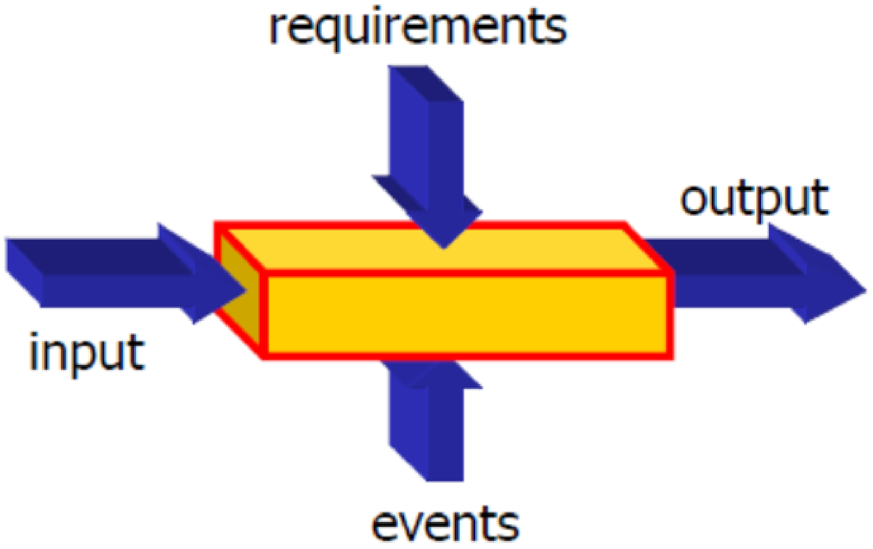


OBJECTIVE	to uncover errors
CRITERIA	in a complete manner
CONSTRAINT	with a minimum of effort and time

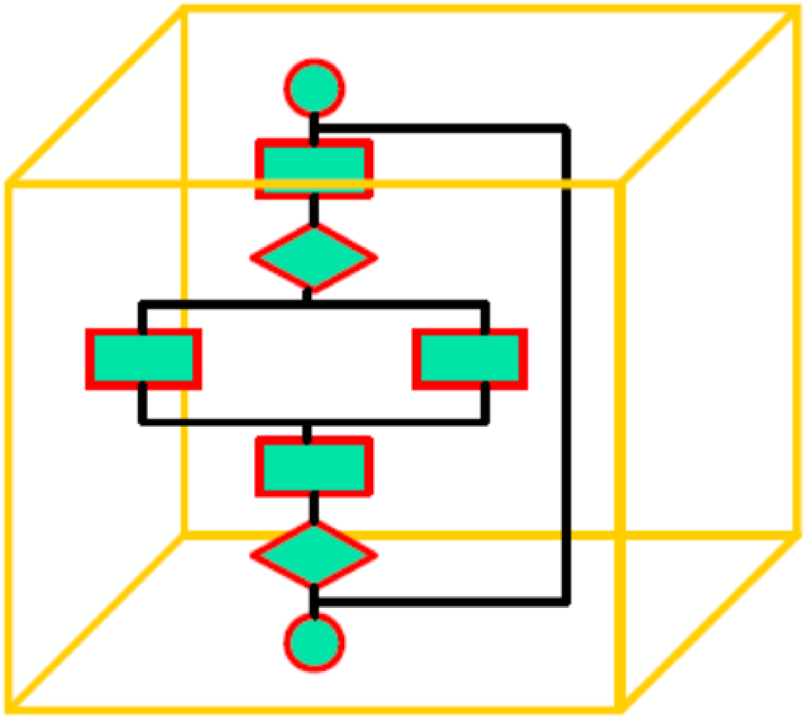
Black-box testing & White-box testing

Two Approaches to Testing

Black-box testing



White-box testing



Black-box testing

- Program is considered as a ‘black-box’
- Test cases are based on the system specification
- Test planning can begin early in the software process
- Two approaches:
  - Equivalence Partitioning
  - Boundary Value Analysis (BVA)

Equivalence Partitioning

- Input data and output results often fall into different classes where all members of a class are related
- Each of these classes is an equivalence partition where the program behaves in an equivalent way for each class member
- Test cases should be chosen from each partition

Example 1

- Specification of function f1: **char f1(char c)**
- f1 accepts a lower case alphabet character “c” as input parameter and returns its corresponding upper case character.
- Input Partitioning Classes:
  - Any valid lower case alphabet char (a-z)
  - Any invalid alphabet char (A-Z)
  - Any invalid non-alphabetical char
  - Numeric: (0-9)
  - Special chars: (=, *, $ etc.)

Example 1:Test Case Design

Class	Input parameter “c”	Expected Return Value/ Output
Valid	h	H
Invalid	G	Error Message
Invalid	%	Error Message

Example 2

- Specification of function f2:

int f2(int x, int y)

- Function takes two ints x, y, and returns the max of the two.

- Input Partition Classes:

  1) x > y (Example test case: x= 10, y=4)
  2) x < y
  3) x == y

Example 3

- Specification of function **int f3(int x)**
- Function returns the absolute value of x.
- Input Partition Classes:

  1) x > 0 (Example test case: x= 10)

  2) x < 0

  3) x = 0

Boundary Value Analysis (BVA)

- Errors tend to occur at the boundaries of the input domain rather than the center.
- BVA leads to a selection of test cases that exercise the boundary values (or "edges") of equivalence classes.
- Used together with Equivalence Class (EC) Partitioning

BVA

- Generate test cases as specified in equivalence classes (ECs), but choose boundaries for test cases.
- Example: Let x be an integer, such that 1<= x <= 100.
- Use 1, 2, 99, and 100 for valid EC
- Use 0 and 101 as invalid EC

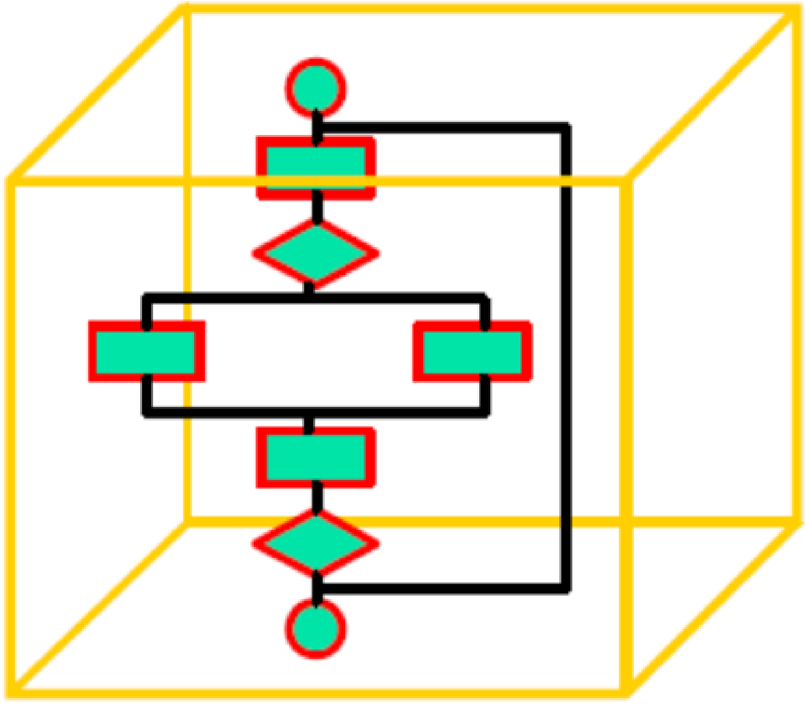
Example

- A banking program accepts a PIN number x between 0 and 100 (inclusive).
- Three equivalence classes can be identified:
  - Class I: x < 0 (Invalid EC)
  - Class II: 0 <= x <=100 (Valid EC)
  - Class III: x > 100 (Invalid EC)
- Test cases:
  - For class I: x = -1
  - For class II: x = 0, +1, 100 (should have some more test cases; e.g., x = 50)
  - For class III: x = 101

Summary

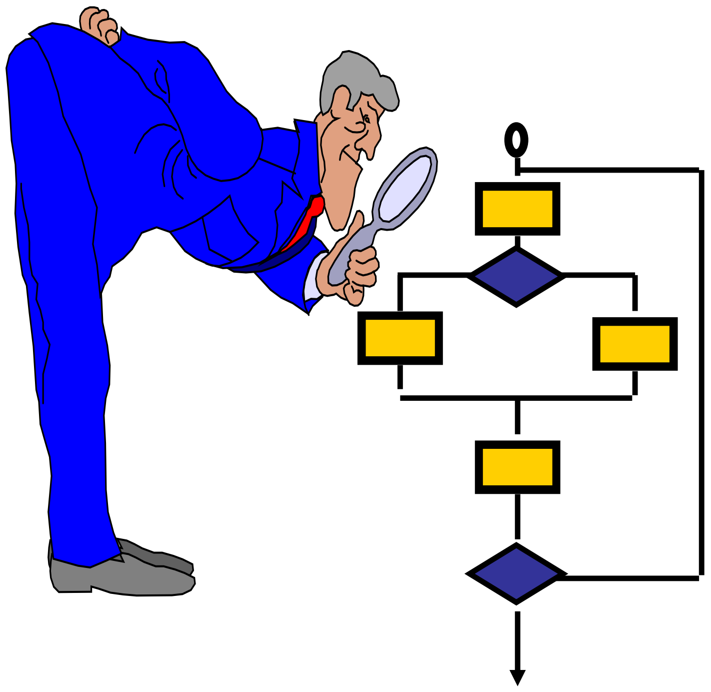
- Black box test planning depends only on the specification, not the implementation
- Need to determine “equivalence” classes
- Need to select test cases for each class
  - typical cases
  - boundary cases
- Black box test cases should be designed in parallel with requirements, design and coding

White box testing



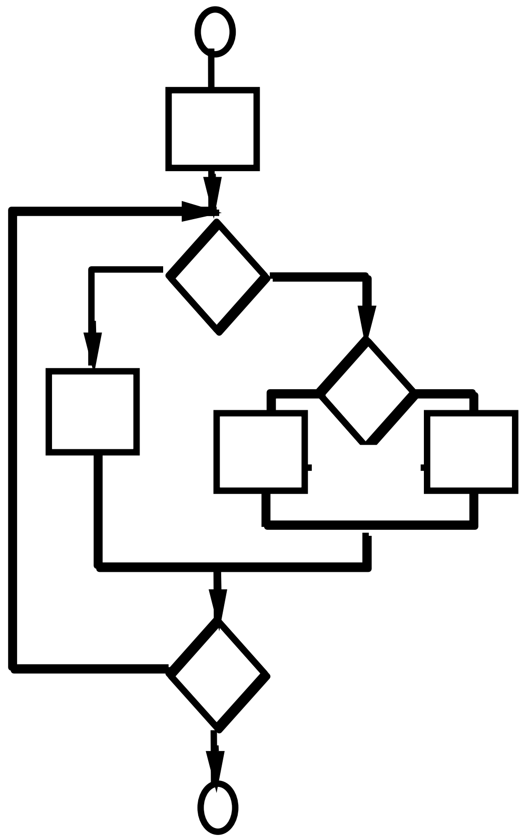
- Exercise all independent paths within a module at least once
- Exercise all logical decisions on their true and false sides
- Exercise all loops at their boundaries and within their operational bounds
- Exercise all internal data structures to assure their validity

White-Box Testing



... our goal is to ensure that all statements and conditions have been executed at least once ...

Basis Path Testing



First, we compute the cyclomatic complexity:

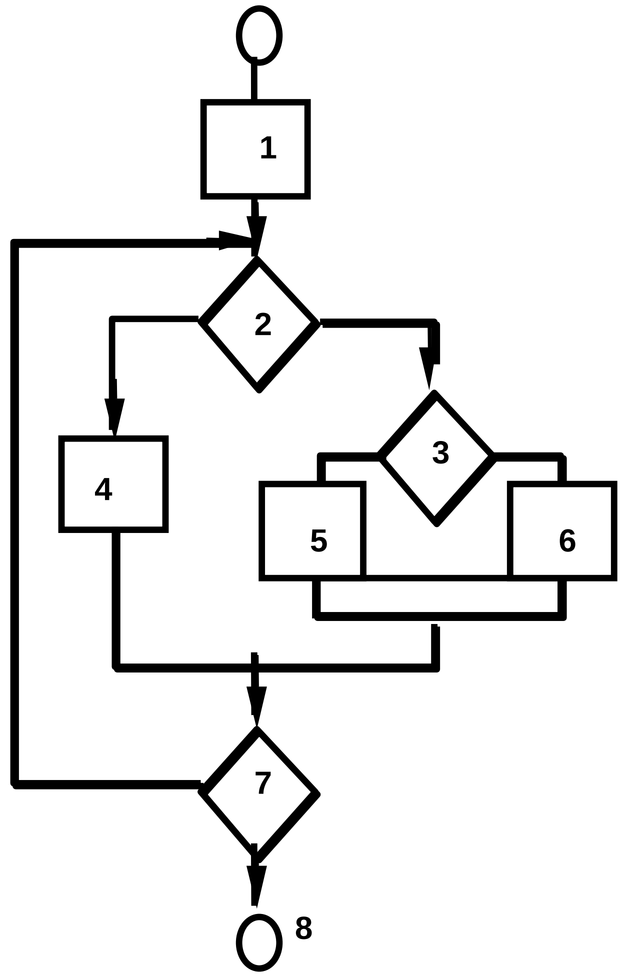
number of simple decisions + 1

or

number of enclosed areas + 1

In this case, V(G) = 4

Basis Path Testing



Next, we derive the independent paths:

Since V(G) = 4, there are four paths

Path 1: 1,2,3,6,7,8

Path 2: 1,2,3,5,7,8

Path 3: 1,2,4,7,8

Path 4: 1,2,4,7,2,4,...7,8

Finally, we derive test cases to exercise these paths.

basis path testing should be applied to critical modules

Deriving Test Cases

- *Summarizing:*
  - Using the design or code as a foundation, draw a corresponding flow graph.
  - Determine the cyclomatic complexity of the resultant flow graph.
  - Determine a basis set of linearly independent paths.
  - Prepare test cases that will force execution of each path in the basis set.

Control Structure Testing

- Condition testing — a test case design method that exercises the logical conditions contained in a program module
- Data flow testing — selects test paths of a program according to the locations of definitions and uses of variables in the program

Loop Testing: Simple Loops

Minimum conditions—Simple Loops

1. skip the loop entirely
2. only one pass through the loop
3. two passes through the loop
4. m passes through the loop $m < n$
5. (n-1), n, and (n+1) passes through the loop

where n is the maximum number of allowable passes

Unit Testing using JUnit

Automated Unit Tests with JUnit

- A procedure to validate individual units of Source Code such as a method
  - Validating each individual piece reduces errors when integrating the pieces together later
- JUnit is a simple, open source unit testing framework for Java
- Automated testing enables running and rerunning tests very easily and quickly
- Open source project www.junit.org

JUnit Example

```
public class Calc
{
   static public int add (int a, int b)
   {
      return a + b;
   }
}
```

```
import org.junit.Test
import static org.junit.Assert.*;

public class CalcTest
{
   @Test
   public void testAdd()
       {
       int result = Calc.add(2,3);
       assertEquals( 5, result);
       }
}
```

Terminologies

- **Test Case**
  - Part of code which ensures that the another part of code (method) works as expected
  - Characterized by a known input (precondition) and by an expected output (postcondition)
  - Positive test case:
    - verifies a correct functionality
  - Negative test case:
    - verifies that a function will fail or throw an exception
- **Test Suite**
  - Contains test cases and run them together
  - Can contain other test suites

Assert statements

Statement	**Description**
assertTrue([message], boolean condition)	Checks that the boolean condition is true.
assertFalse([message], boolean condition)	Checks that the boolean condition is false.
assertEquals([String message], expected, actual)	Tests that two values are the same. Note: for arrays the reference is checked not the content of the arrays.
assertNull([message], object)	Checks that the object is null.
assertNotNull([message], object)	Checks that the object is not null.
assertSame([String], expected, actual)	Checks that both variables refer to the same object.
assertNotSame([String], expected, actual)	Checks that both variables refer to different objects.

JUnit Annotations

Annotation	Description
@Test public void method()	The @Test annotation identifies a method as a test method.
@Test (expected = Exception.class)	Fails, if the method does not throw the named exception.
@Test(timeout=100)	Fails, if the method takes longer than 100 milliseconds.
@Before public void method()	This method is executed before each test. It is used to can prepare the test environment (e.g. read input data, initialize the class).
@After public void method()	This method is executed after each test. It is used to cleanup the test environment (e.g. delete temporary data, restore defaults). It can also save memory by cleaning up expensive memory structures.
@BeforeClass public static void method()	This method is executed once, before the start of all tests. It is used to perform time intensive activities, for example to connect to a database. Methods annotated with this annotation need to be defined as static to work with JUnit.
@AfterClass public static void method()	This method is executed once, after all tests have been finished. It is used to perform clean-up activities, for example to disconnect from a database. Methods annotated with this annotation need to be defined as static to work with JUnit.
@Ignore	Ignores the test method. This is useful when the underlying code has been changed and the test case has not yet been adapted. Or if the execution time of this test is too long to be included.

Steps to perform unit tests (Junit)

1. **Prepare** (or setUp()) environment conditions that must be met, according to the test plan. At this stage, define and set prefix values. E.g. instantiate objects, initialize fields, turn on logging, etc.
2. **Execute** the test case. This means, executing (exercising) the part of the code to be tested. For that we use some test inputs (test case values), according to the test plan.
3. **Evaluate** (or **assert*()**) the results, or side effects generated by the execution of the test case, against an expected value as defined in the test plan.
4. **Clean up** (or **tearDown()**) the test environment if needed so that further testing activities can be done, without being influenced by the previous test cases. We deal here with postfix values.

Step 1: Unit Testing with JUnit

1. **Prepare** (or setUp()) the test environment:

   - Annotate with @Before: Those methods are executed before each test case (test method).

```
@Before
public void setUp() {
    s = new Sample();
}
```

Step 2&3: Unit Testing with JUnit

2. Execute the test case.

3. Evaluate the results (using assertion).

```
@Test
public void testAddition( ) {
    int a=3 , b=6;
    int expectedOutput = (a+b);
    int res = s.Addition(a, b);
    assertEquals(expectedOutput, res);
}
```

Step 4: Unit Testing with JUnit

4. Clean up (or tearDown()) the test environment is done in one or several methods that are run after execution of each test method.

   - A method has to be annotated with @After.
   - If you allocate external resources in a @Before method, you need to release them after the test runs.

```
@After
public void tearDown() {
    s = null;
}
```

Status of a Test

- A test is a single run of a test method.
- Success
  - A test succeeds in time when No assert is violated; No fail statement is reached; No unexpected exception is thrown.
- Failure
  - A test fails when an assert is violated or a fail statement is reached.
- Error
  - An unexpected exception is thrown or timeout happens.

Junit with Netbeans

1. Create the Java Project
2. Create the Java Class
3. Create a Test Class for Java Class
4. Write Test Methods for Test Class
5. Run the Test
6. Create Test Suit (optional)