TESTING (PART A)

Chapter 9
Block a. Testing

Software Engineering
Computer Science School
DSIC – UPV

Goals

- Understand difficulties associated to software validation and verification.
- Understand the basic techniques for software testing
- Design test cases for a module or function using the **Basis Path** and **Equivalence Partitioning** testing techniques.

Contents

- Introduction to software testing
- Techniques to design test cases
- White box testing: Basis Path Testing.
- Black box testing: Equivalence Partitioning.
- Tools for automated testing

References

- SOMMERVILLE, I., Software Engineering, 7ª Edición. Addison Wesley, 2005
- PFLEEGER, S. L., Ingeniería del Software: Teoría y Práctica. Prentice Hall, 2002.
- PRESSMAN, R. Ingeniería del software. Un enfoque práctico. 6ª Edición, McGraw-Hill, 2006.
- COLLARD, J.F, BURNSTEIN, I, Practical Software Testing: A Process-Oriented Approach, Springer. 2003
- EVERETT, D., McLEOD, R. Software Testing. Testing Across the Entire Software Development Life Cycle, IEEE Press
- BEIZIER, B., Testing and quality assurance, von Nostrand Reinhold, New York, 1984

Introduction

- Testing: critical factor to determine the quality of a software system

- Software testing may be defined as an activity in which a system or one of its components is executed under previously defined conditions, the results are observed and recorded and the evaluation of some aspect is performed: correctness, robustness, efficiency, etc.

- Test case: «a set of inputs, execution conditions and expected results developed for a given goal»

Basic principles

- **Principle 1**: Testing is the process of executing a software component using a basic set of test cases with the intention of (i) revealing bugs, and (ii) evaluating the quality
- **Principle 2**: If the goal of the test is to reveal bugs then a good test case is that with a higher probability of detecting undetected bugs
- **Principle 3:** The results of a test must be inspected in detail
- **Principle 4:** A test case must include the expected results
- **Principle 5:** Test cases must be defined for both valid and invalid input conditions

Basic principles

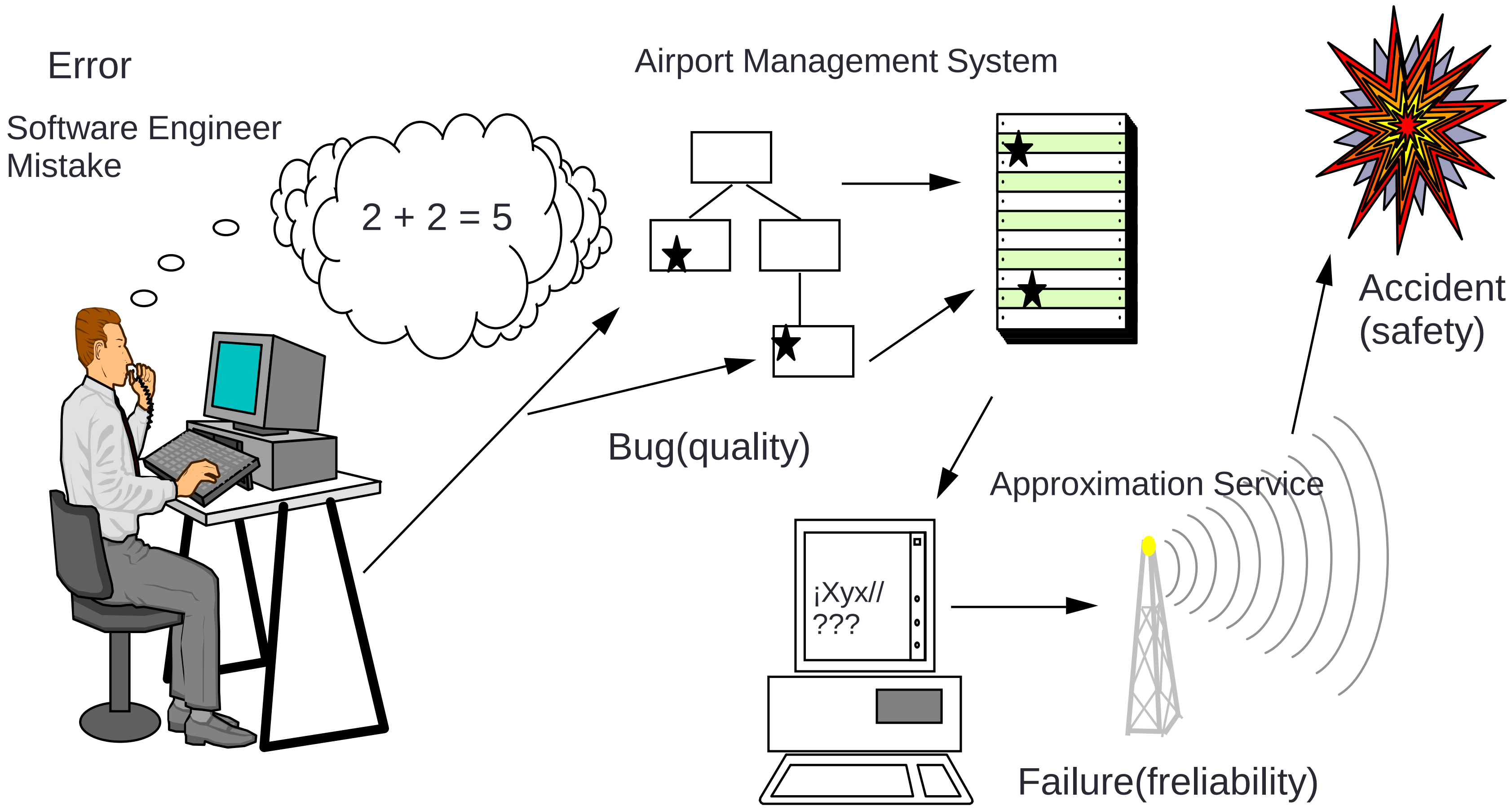
- **Principle 6**: The probability of existing additional bugs is proportional to the number of bugs already detected for a component
- **Principle 7**: Tests must be carried out by an independent group (different from the development team)
- **Principle 8:** Test must be reproducible and reusable
- **Principle 9:** Tests must be planned
- **Principle 10:** Testing activities should be integrated with the other ones of the lifecycle
- **Principle 11:** Testing is a creative and defiant activity

Google Vision about Testing

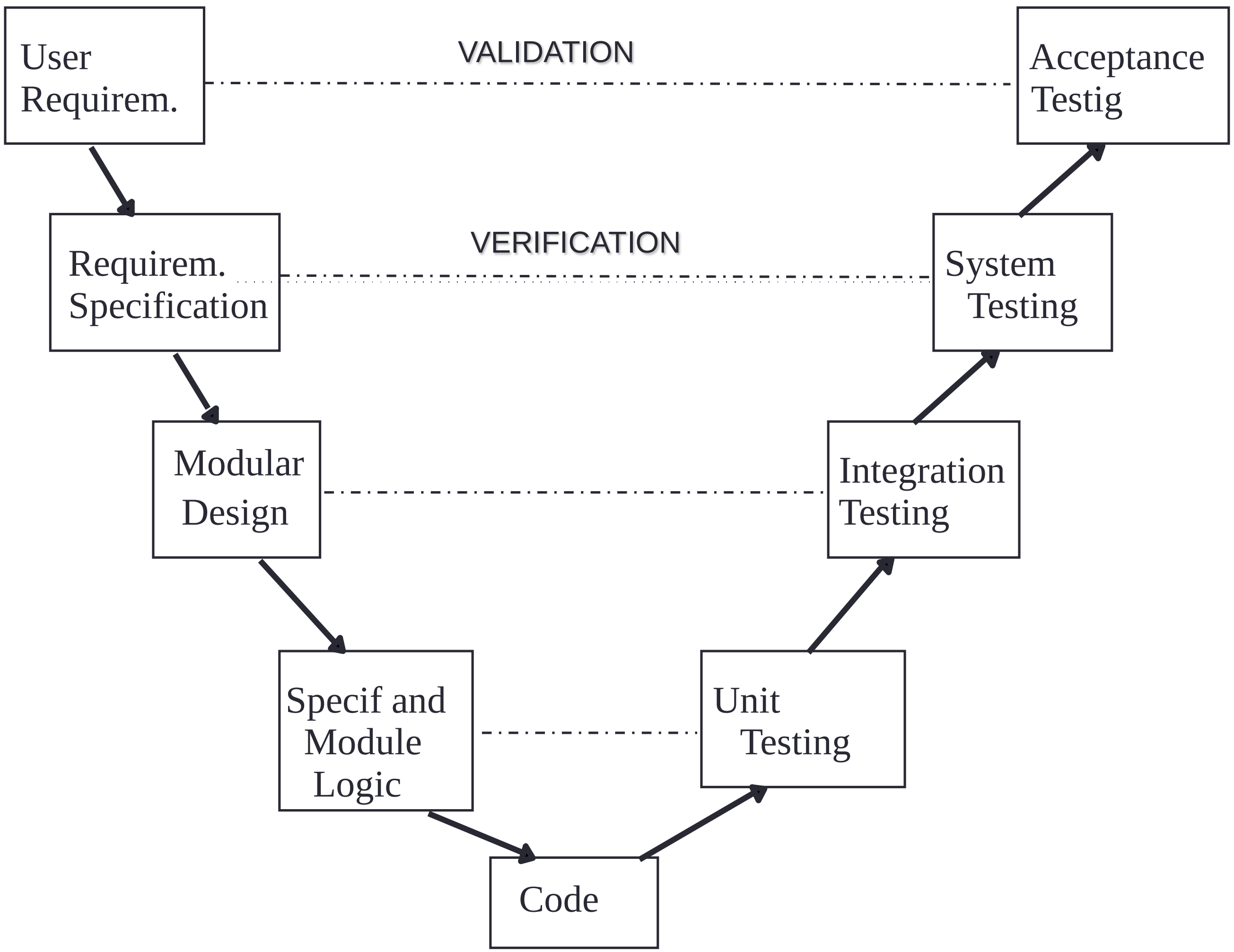
Definitions

- Verification: Are we correctly building the product? Has the product been built according to the specifications?
- Validation: Are we building the correct product? Does the software do what the user really wants?
- Bug: An anomaly in the software, e.g. An incorrect process, data definition or step in a program
- Failure: when the system is not capable of performing the required functions within the specified performance values
- Error: human action conducting to an incorrect result (e.g. Coding error)

Relationship between error, bug and failure



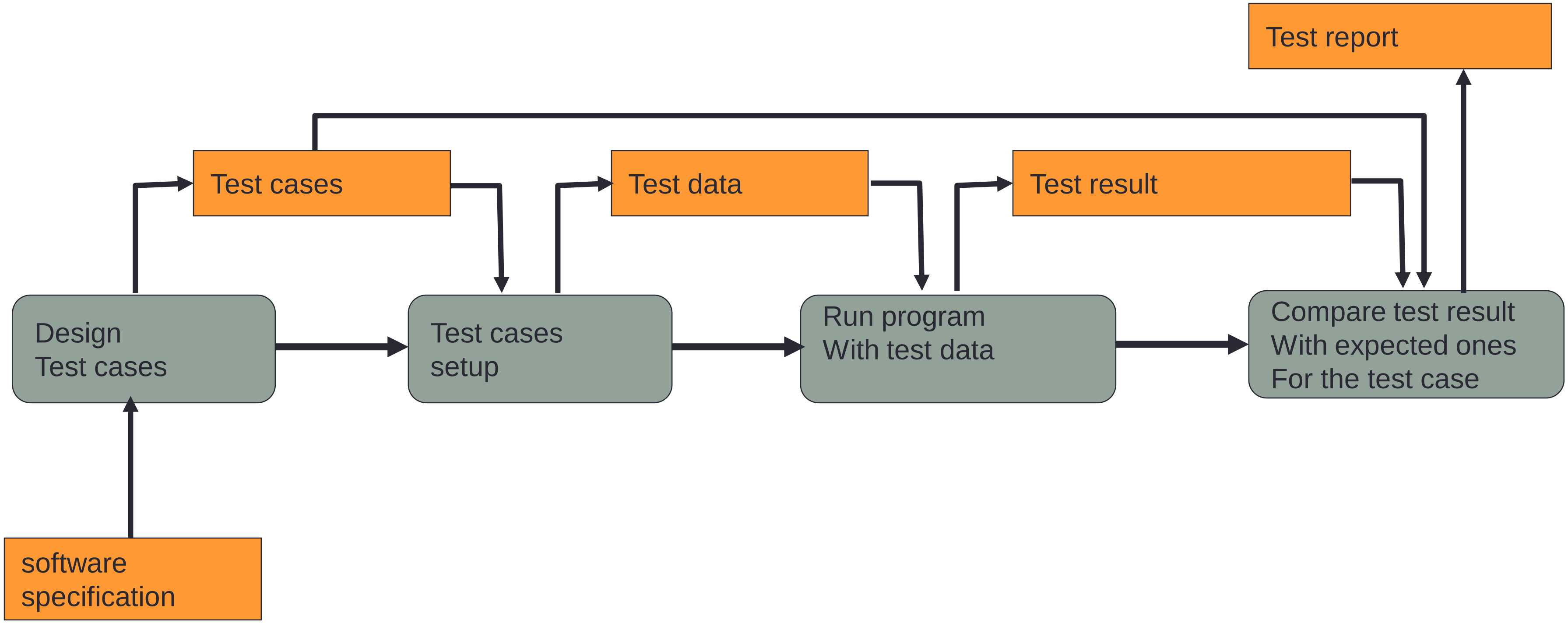
The testing process



The testing process

- Unit testing: each module is tested individually.
- Functional or Integration testing: the software completely assembled is tested as a whole to make sure that it is compliant with the functional and non-functional requirements: performance, security, etc.
- System testing: the software is validated with the rest of the system (e.g., mechanical elements, electronic interfaces, etc).
- Acceptance testing: the final product is tested by the final user in its own production environment to determine whether it is accepted.

Test information flow



Debugging process

- With respect to locating the bug in the source code
  - Analyze the information and think.
  - If a dead-end point is reached, swap to another task.
  - If a dead-end point is reached, describe the problem to another person..
  - Do not experiment by changing the code.
  - Bugs must be handled individually.
  - Pay attention to data.

Debugging process

- When correcting a bug:
  - Where a bug is found, there are usually more bugs.
  - The bug must be fixed, not its symptoms.
  - The probability of fixing a bug is not 100%.
  - New bugs may be inserted.
  - The correction must place ourselves at the design phase.

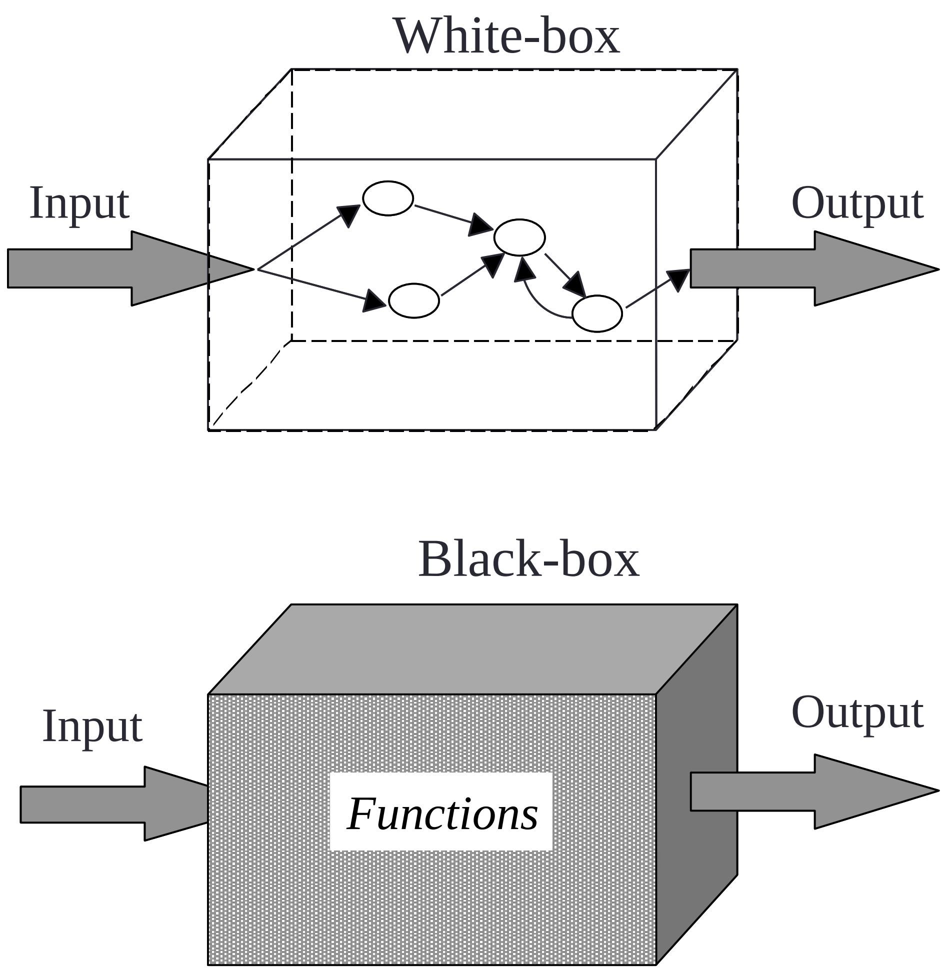
Designing test cases

- The design of test cases for software verification may result in a considerable effort (40% of the overall development time)

- There are mainly three approaches:
  - White-box testing
  - Black-box testing
  - Random testing

- Combining several approaches produces a better end result.

Designing test cases



Designing test cases

- White-box testing or structural testing is based on the meticulous study of all the operation of a part of the system considering procedural details.
  - Different execution paths are planned to observe the results and compare them with the expected ones.
  - It could be thought that all the possible execution paths of a procedure could be tested.
  - In practice this is not possible in most real systems due to the exponential growth in the number of possible combinations.

Designing test cases

- Black-box or functional testing analyzes the compatibility with respect to the interfaces of each module or software component.
- Random testing defines models that represent the possible inputs of the module and from these models the test cases are generated.
    - Statistical models that simulate the sequences and frequency of input data.
    - It is based on the assumption that the probability of finding bugs is the same no matter random tests or tests following coverage criteria are performed.
    - However, bugs may remain hidden that are only discovered with very concrete inputs.
    - This type of tests may be sufficient for non-critical software.

White-box Testing

- White-box testing uses the procedural control structure to derive the test cases.
- Idea: It is not possible to test all the different execution paths but test cases can be defined to execute all the paths called independent.
- For each independent path:
  - Test its two logical facets, i.e., when the path is executed and when it is not.
  - Execute all the loop at their operational limits
  - Use the internal data structures.

White-box Testing

- The logic bugs and the incorrect assumptions are inversely proportional to the probability of execution of a path.
- It is often assumed that a path is executed few times when it is in fact regularly executed.
- Typographic errors are random.
- As Beizer stated, “*Bugs lurk in corners and congregate at boundaries*”.

Code Coverage Types

- Statement coverage: each sentence or code instruction is executed at least once.
- Decision coverage: each decision has at least once a true and a false result.
- Condition coverage: each condition of a decision must adopt at least once a true and a false values
- Decision/condition coverage: when both types of coverage are required
- Multiple condition coverage: to guarantee that all possible combinations within a decision are tested.

Code Coverage Types

EXAMPLE 1:
 	
if (a>b)
  then a=a-b
  else no b=b-a
end_if
	EXAMPLE 2:
	
	if (a>b)
	  then a=a-b
	end_if

EXAMPLE 3:
 	
i=1
while (v[i]<>b)and(i<>5)
do
   i=i+1;
end_while
	EXAMPLE 4:
	
	if (a>b) and (b is prime)
	  then a=a-b
	end_if

Code Coverage Types

```
1 if (a>b) and (b is prime)
2   then a=a-b
  //without else
  end_if
```

Sentence coverage: 2 fragments

```
if (a>b) and (b is prime)     1
  then a=a-b                  2
//without else               3
end_if
```

Decision coverage: 3 fragments

```
      1, 2          3, 4
if  (a>b)  and  (b is prime)
5     then a=a-b
6   //without else
    end_if
```

Condition coverage: 6 fragments

2^2 = {TT, TF, FT, FF} True table

```
      1, 2          3, 4
if  (a>b)  and  (b is prime)
5     then a=a-b
6   //without else
    end_if
```

Multiple Condition coverage: 6 fragments

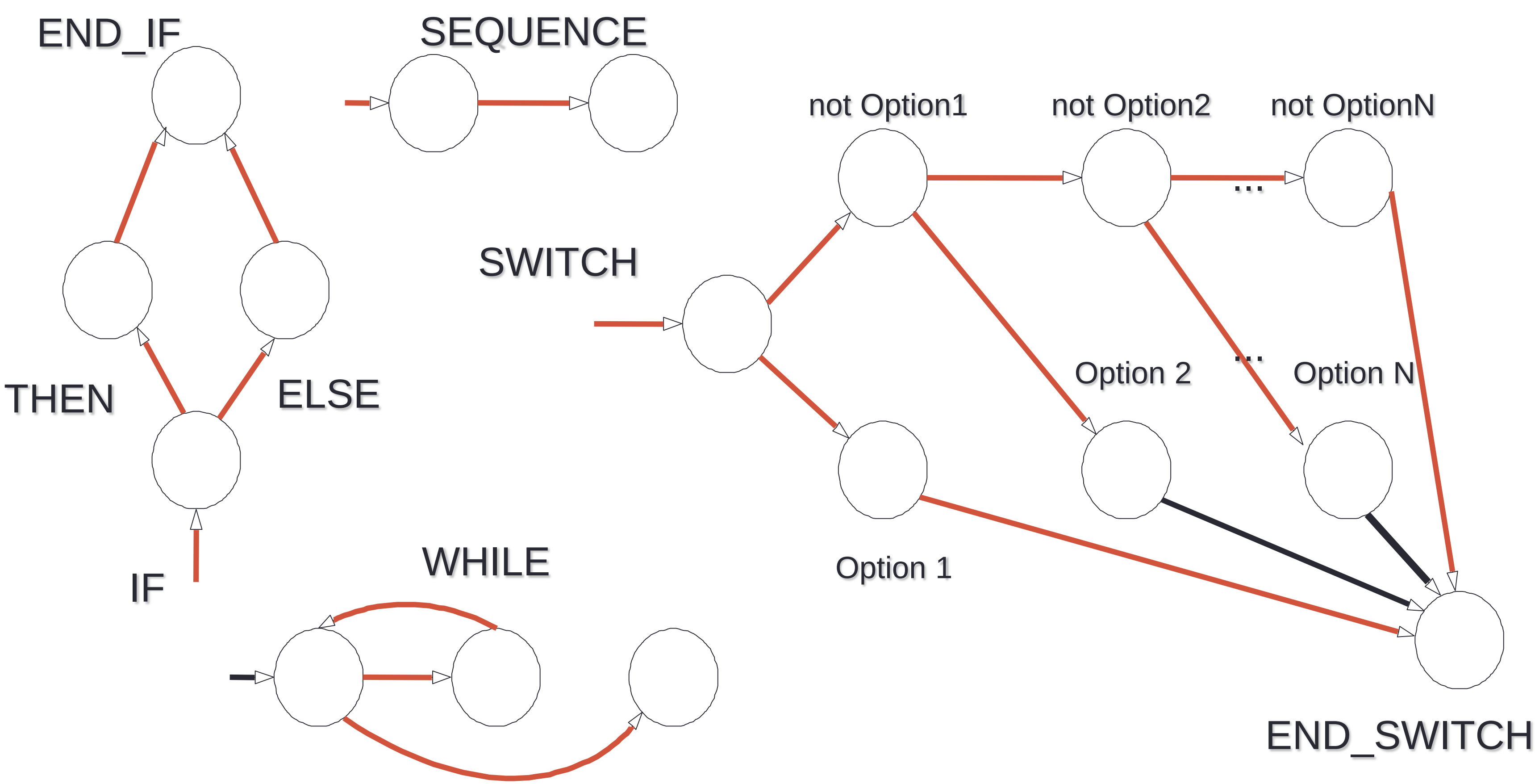
Basis Path Testing

- Basis path testing is a white-box testing technique proposed by Tom McCabe.
- The idea is to derive test cases from a set of independent paths representing different flow control executions.
- Independent path is the one adding a processing statement (or condition) that was not previously considered in the current set of independent paths.
- To obtain the set of independent paths we will build the associated control flow graph and we will calculate the cyclomatic complexity.

Basis Path Testing: Control flow graph

- The control flow of a program may be represented by means of a graph.
- Each node correspond to one or more statements of the source code
- Each node representing a condition is called predicate node.
- Any procedural representation may be transformed into a control flow graph.
- An independent path in the graph is one adding a new edge, i.e., an edge that was not present in the previous considered paths.

Basis Path Testing: Control flow graph



Basis Path Testing: Control flow graph

- If test cases are designed to cover the basis paths it is guaranteed the execution of each statement once and each condition in its two possible values (true and false).
- The basis set may not be unique for a given graph and it depends in the order in which new paths are defined.
- When combined logical conditions are considered the graph is more complex.

Basis Path Testing: Control flow graph

- Example:

IF a OR b

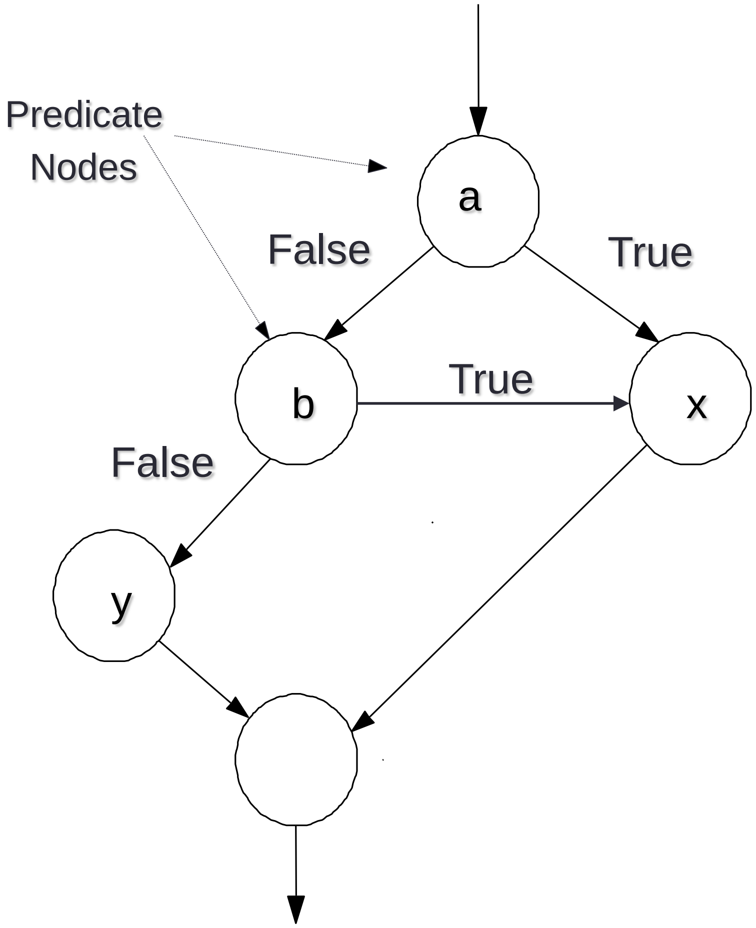
THEN

do x

ELSE

do y

ENDIF

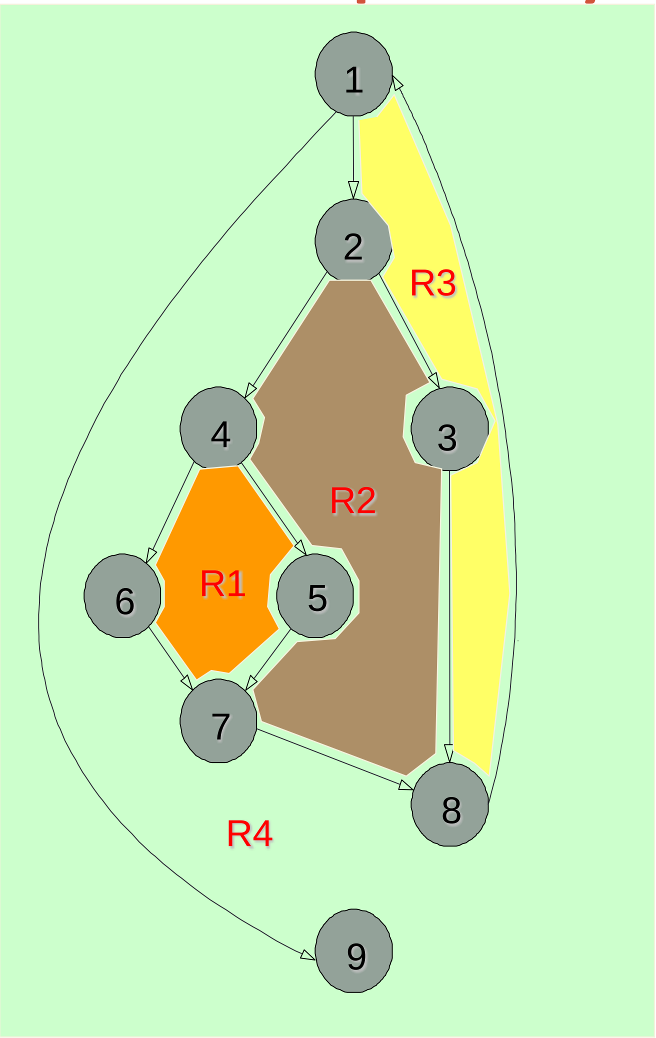


Basis Path Testing: Cyclomatic complexity

- Cyclomatic complexity of a control flow graph, V(G), indicates the maximum number of independent paths.
- It may be calculated in three different ways:
  - The **number of regions** in which the graph divides the plane.
  - **V(G) = E - N + 2**, whare E is the number of edges and N the number of nodes
  - **V(G) = P + 1**, where P is the number of predicate nodes.

Basis Path Testing: Cyclomatic complexity

- V(G) = 4
  - graph creates 4 regions
  - 11 edges - 9 nodes + 2 = 4
  - 3 predicate nodes + 1 = 4



Basis Path Testing: Cyclomatic complexity

- The set of independent paths will be at most 4.
  - Path 1: 1-9
  - Path 2: <u>1-2</u>-4-5-7-8-1-9
  - Path 3: 1-2-<u>4-6</u>-7-8-1-9
  - Path 4: 1-<u>2-3</u>-8-1-9

- Any other path will not be an independent path, e.g., 1-2-4-5-7-8-1-2-3-8-1-2-4-6-7-8-1-9

  because it is a combination of already added paths (no new edges)

- The four previous paths constitute a ***basis set*** for the graph

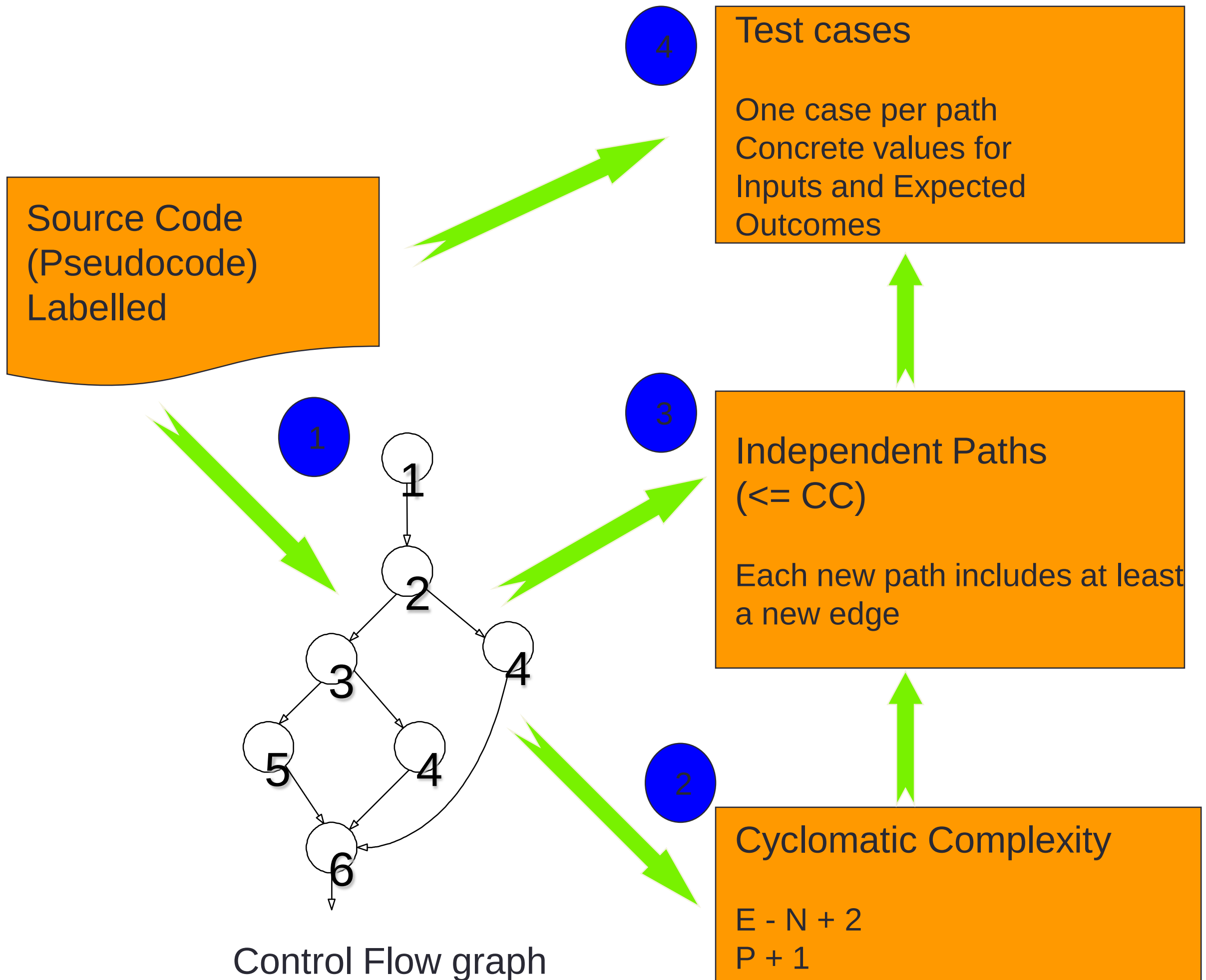
Basis Path Testing: Deriving test cases

- The method can be applied to a detailed procedural design (pseudocode) or to the application source code.

- Steps to design the test cases:

0. Label the source code giving a number to each statement (sometimes group of statements) and each simple condition.
1. Draw the associated control flow graph.
2. Calculate the cyclomatic complexity.
3. Obtain a basis path set.
4. Obtain a test case to execute each basis path.

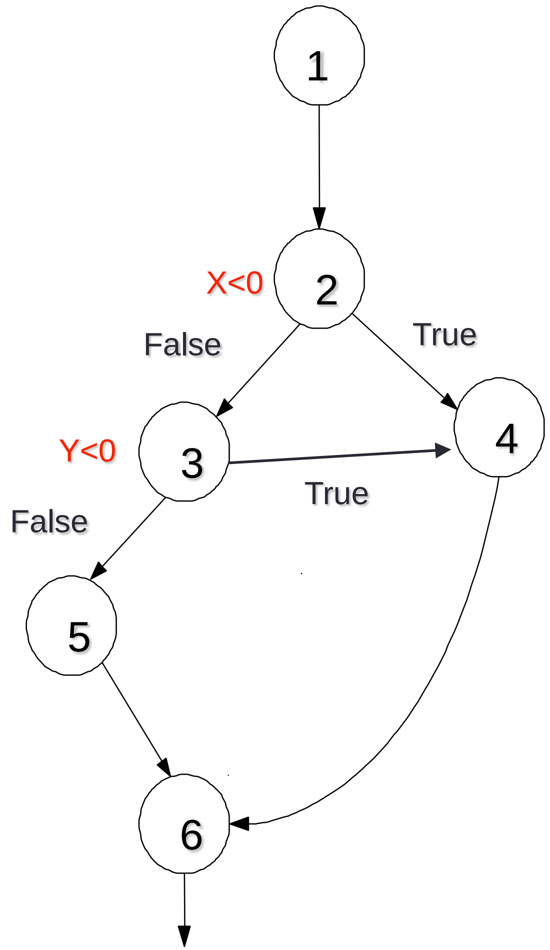
Basis Path Testing: Deriving test cases



Basis Path Testing: Example

```
     void calculate_and_print_average(float x, float y)
     {
 1         float result;
           result = 0.0;                       3
           if (x < 0 || y < 0)
       2         printf(“x and y must be positive”);   4
           else {
       5         resultado = (x + y)/2;
                 printf(“Average is %f\n”, result);
           }
 6   }
```

Basis Path Testing: Example



V(G) = 3 regions. Thus, at most three independent paths.

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

Test cases:

Path 1: x=-1, y=3, result=0, error

Path 2: x=3, y= 5, result=4

Path 3: x=4, y=-3, result=0, error

Basis Path Testing: Exercise

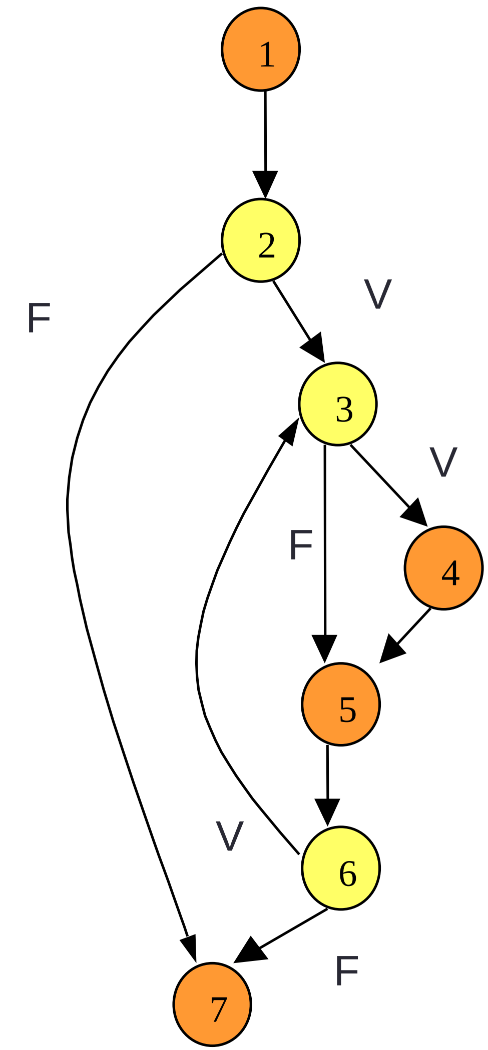
```
int count_character(char string[10], char c)
{
  int cont, n,  lon;
  n=0; cont=0;
  lon = strlen (string);
  if (lon > 0) {
   do {
     if (string[cont] == c)
             n++;
     cont++;
     lon--;
   } while (lon > 0);
  }
  return n;
}
```

Basis Path Testing: Exercise

```
int count_character(char string[10], char c)
{
  int cont, n,  lon;
  n=0; cont=0;
  lon = strlen (string);
  if (lon > 0) {
   do {
     if (string[cont] == c)
           n++;
     cont++;
     lon--;
   } while (lon > 0);
  }
  return n;
}
```

Basis Path Testing: Exercise

```
int count_character(char string[10], char c)
{
  int cont, n,  lon;          1
  n=0; cont=0;
  lon = strlen (string);
  if (lon > 0) {   2
    do {
      if (string[cont] == c)          3
            n++;   4
      cont++;          5
      lon--;
    } while (lon > 0);   6
  }
  return n;   7
}
```



Basis Path Testing: Exercise

- V(G) = 4;

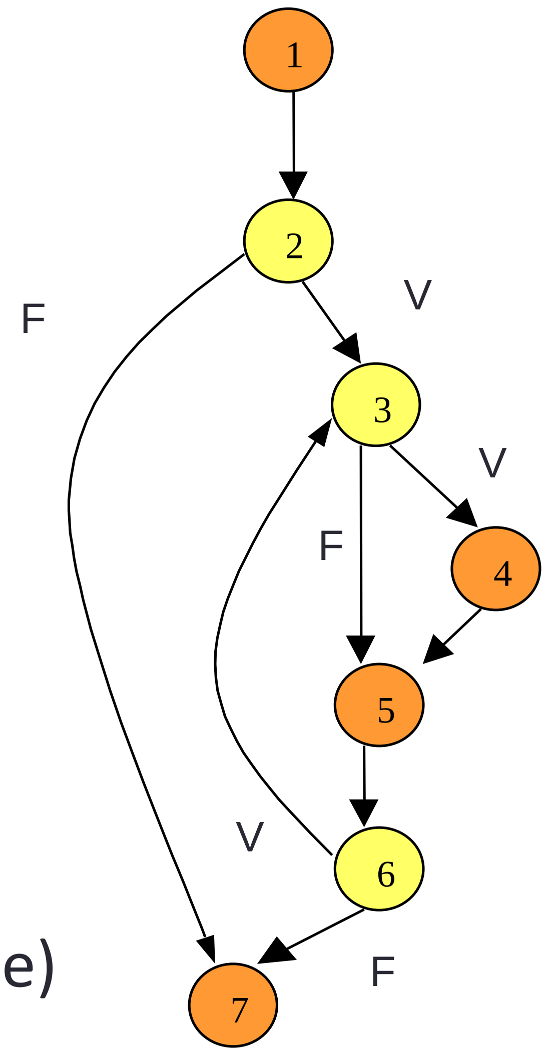
  Nodes=7; Edges=9;

  Predicate Nodes=3;

  Regions = 4

-

- Basis Path Set:
  1. 1-2-7
  2. 1-2-3-5-6-7
  3. 1-2-3-4-5-6-7
  4. 1-2-3-4-5-6-3-5-6-7 (Not unique)



Basis Path Testing: Exercise

1.1-2-7	string = ””	c = ‘a’	n = 0;
2.1-2-3-5-6-7	string = “b”	c = ‘a’	n = 0;
3.1-2-3-4-5-6-7	string = “a”	c = ‘a’	n = 1;
4.1-2-3-4-5-6-3-5-6-7	string = “ab”	c = ‘a’	n = 1;