

COMP 3111

SOFTWARE ENGINEERING

TESTING

LEARNING OBJECTIVES

1. Understand the **purpose** of **testing**.
2. Know **three basic types of test cases** and **how they can be generated**.
3. Understand the **principles of component-based** and **system-based testing**.
4. Learn different **strategies** for performing the different types of **test cases**.



TESTING OUTLINE

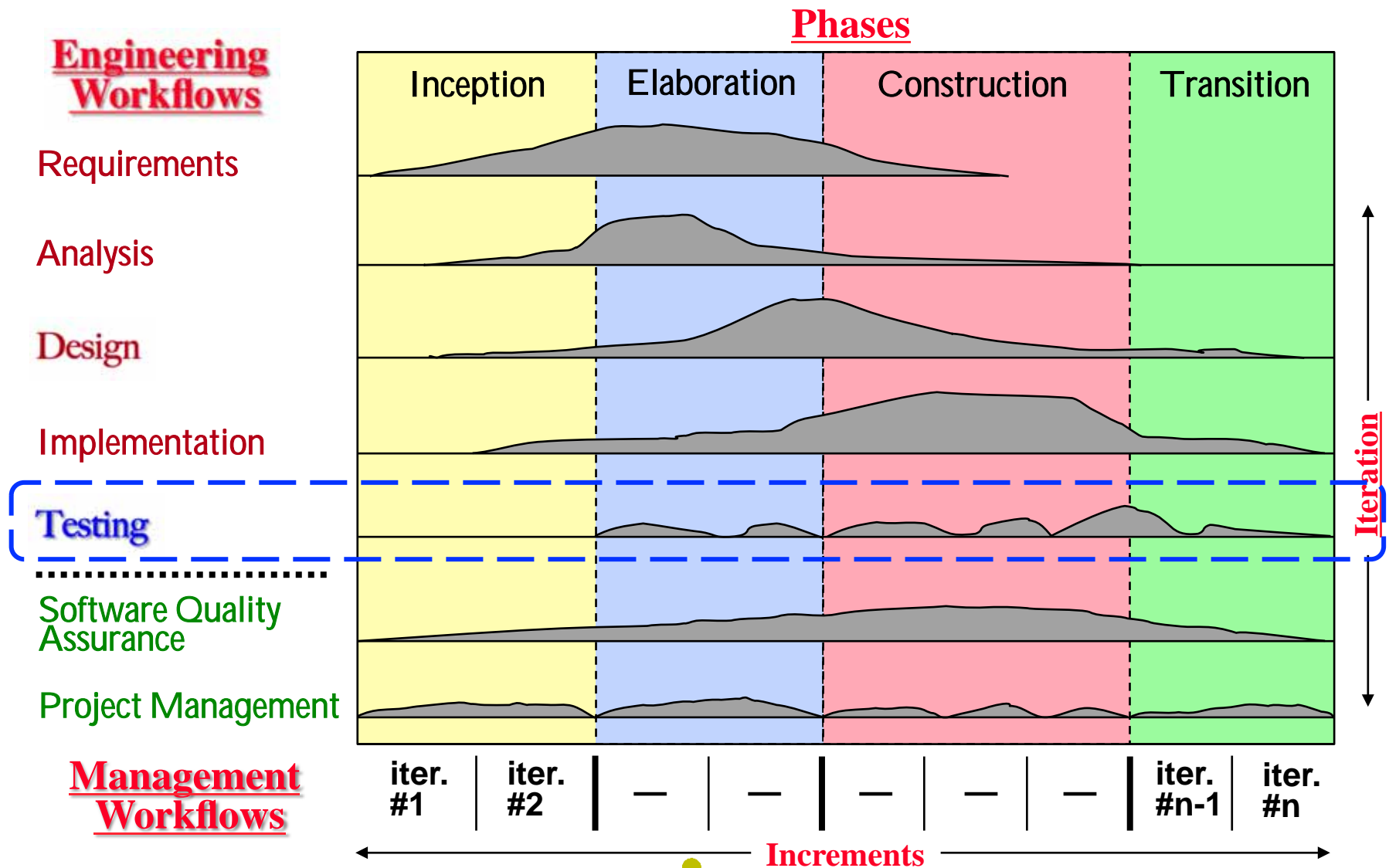
Testing Overview

- Life Cycle Role
- The Purpose of Testing
- The Reality of Testing

Testing Activities

- Plan Tests
- Design Tests: *White Box/Glass; Black Box; Regression*
- Implement Tests
- Perform Tests: *Unit; Integration; System; Acceptance*
- Evaluate Tests

TESTING LIFE CYCLE ROLE



TESTING LIFE CYCLE ROLE

- In the Unified Process, testing is primarily in focus both during **elaboration** (when the executable architectural baseline is tested) and during **construction** (when the bulk of the system is implemented).
- During **inception**, **elaboration** and **construction**, the focus is primarily on **integrating** and **system testing** each build.
- During **transition**, the focus is on **fixing defects (bugs)** and doing **regression testing**.
- Previously developed test cases are used later as **regression test cases**.
- The test model is **maintained throughout the life cycle**, but it evolves:
 - obsolete test cases are removed.
 - some test cases are refined into regression test cases.
 - new test cases are created for each new build.

ARIANE 5 ROCKET



- The rocket self-destructed 37 seconds after launch

Reason: A control software bug that went undetected.

- Conversion from 64-bit floating point to 16-bit signed integer value caused an exception.
 - The floating point number was larger than 32,767 (max 16-bit signed integer).
- Efficiency considerations had led to the disabling of the exception handler.
- The program crashed → rocket crashed.

Total cost: over US\$1 billion.

MARS POLAR LANDER



- A sensor signal falsely indicated that the lander had touched down when it was still 130 feet above the surface.
- The software—intended to ignore touchdown indications prior to the enabling of the touchdown sensing logic—was not properly implemented, and the spurious touchdown indication was retained.
 - Consequently, the descent engines shut down prematurely → lander crashed.
- The error was traced to *a single bad line of code*.
- NASA investigation panel's blame for the landers failure “are well known as difficult parts of the software engineering process.”

TESTING IS FOR EVERY SYSTEM

- These examples show particularly costly errors.
- However, every little error adds up!
 - In 1982 the Vancouver Stock Exchange instituted a new index initialized to a value of 1000.000. The index was updated after each transaction. Twenty two months later it had fallen to 520. The cause was that the updated value was truncated rather than rounded. The rounded calculation gave a value of 1098.892!
- Insufficient software testing costs \$22-60 billion per year in the U.S.

 If your software is worth writing,
it's worth testing to make sure it is correct!

THE PURPOSE OF TESTING

Testing tries to find differences between the specified (*expected*) and the observed (*actual*) system behavior.

- **Validation** makes sure that we have built the right product.
We check that the system meets its stated requirements and each system function derives from some requirement.

☞ Acceptance tests deal mainly with validation.

- **Verification** makes sure that we have built the product right.
We check the quality of the implementation by ensuring that each function works correctly and has no defects.

☞ Most testing is targeted at doing verification.

Goal: To design tests that will systematically find defects.

THE REALITY OF TESTING

- ✎ Testing **cannot** show the **absence** of software errors;
it **can only show** that software **errors are present!**
- ✎ It is **impossible** to **completely test** a **nontrivial system**.
(Most systems will have bugs in them that either users uncover
or that are *never uncovered!*)
- ✎ Testing is a **destructive** activity.
- ✎ It is often **difficult** for **software engineers**
to effectively test their own software.
(Since they have no incentive to make their software fail!)

IS EXHAUSTIVE TESTING FEASIBLE?

- Why do I need a testing plan?
- Why not just try my program with all inputs and see if it works?

```
int procl (int x, int y, int z)
    // requires:  $1 \leq x, y, z \leq 1000$ 
    // effects: computes some  $f(x, y, z)$ 
```

How many runs are needed to *exhaustively* test this program?

Conclusion

It is imperative to have a plan for testing if we want to uncover as many defects as possible in the shortest possible time.

PLAN TESTS

Goal: To design a set of tests that has the **highest likelihood** of uncovering defects with the **minimum amount of time and effort**.

WHY?

A test plan specifies:

1. A testing strategy → the **criteria** and **goals** of testing.
 - **What** kinds of tests to perform and **how** to perform them.
 - The required level of test and code **coverage**.
 - The **percentage** of tests that should execute with a specific result.
2. A schedule for the testing: when to run which tests.
3. An estimate of resources required: human/system.

DESIGN TESTS: TEST CASE

A test case is one way of testing the system.

Specifies: *what* to test, *under what conditions*, *how* to test and *expected result*.

White Box Tests: “testing-in-the-small”

Verify component logic based on data or control structures.

Test cases based on *knowledge of the internal workings* of a component.

👉 **Availability of source code is required.**

Black Box Tests: “testing-in-the-large”

Verify component functionality based on the inputs and outputs.

Test cases based on *knowledge of specified functionality* of a component.

👉 **Availability of source code is not required!**

Regression Tests: *selective* White Box and Black Box re-testing to ensure that no new defects are introduced *after a change is made*.

DESIGN TESTS: TEST CASE DOCUMENTATION

Test case name → name of the test case

Description of the test → what the test is designed to do

Target class/component/subsystem name → name of the thing to be tested

Target class/component/subsystem operation → name of the operation to be tested

Test type → black box/white box; valid/invalid input or output

Test value(s) → inputs to be used for the test

Verification → expected result

It requires a lot of time and effort to generate all this information!

DESIGN TESTS: WHITE/GLASS BOX TESTING

Goal: Design *selective tests* to ensure that we have executed all:

1. **independent paths** in the code **at least once**.

👉 **Basis Path Testing**

2. **logical decisions** on their **true** and **false** sides.

👉 **Condition Testing**

3. **loops** at their boundaries and within their bounds.

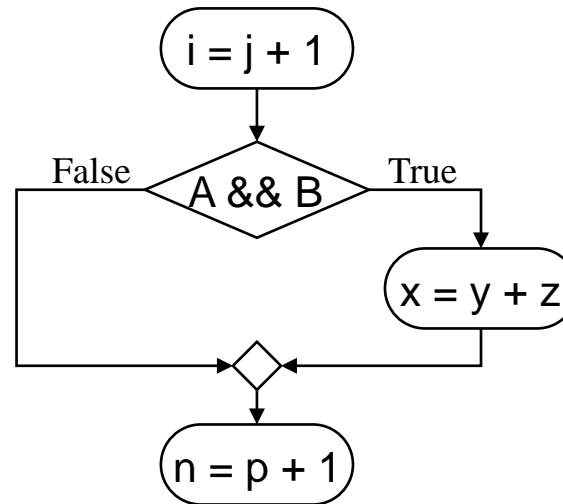
👉 **Loop Testing**

4. **internal data structures** to ensure their validity.

👉 **Data Flow Testing**

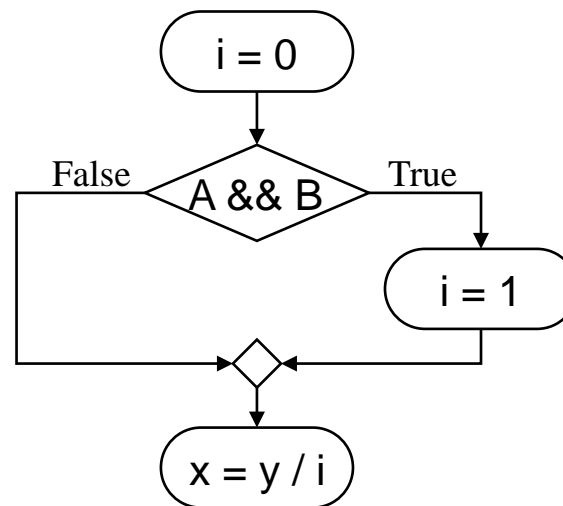
IS STATEMENT TESTING SUFFICIENT?

```
i = j + 1
if (A && B) {
    x = y + z;
}
n = p + 1;
```



Suppose we modify the program to:

```
i = 0
if (A && B) {
    i = 1;
}
x = y / i;
```



Do you see any problems with the modified program?

WHITE BOX TESTING: BASIS PATH TESTING

Goal: To execute each **independent path** at least once.

We derive a **logical complexity** measure for the code and then use this measure to define a **basis set** of execution paths.

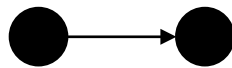
Overview

1. From the code, draw a corresponding **flow graph**.
2. Determine the **cyclomatic complexity** of the flow graph.
3. Determine a **basis set of linearly independent paths** based on the cyclomatic complexity.
4. Prepare **test cases** that **force** the **execution of each path** in the basis set.

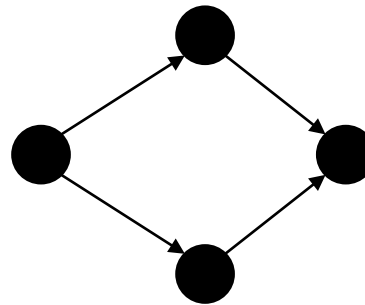
WHITE BOX TESTING: BASIS PATH TESTING (cont'd)

1. From the code, draw a corresponding flow graph.

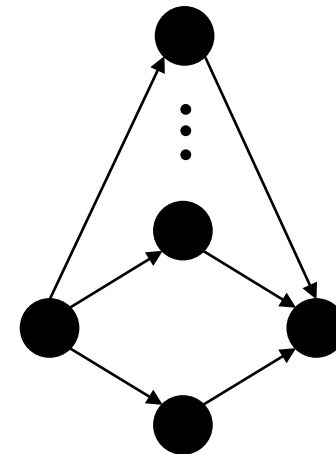
- Each circle represents **one or more** non-branching source code **statements**.



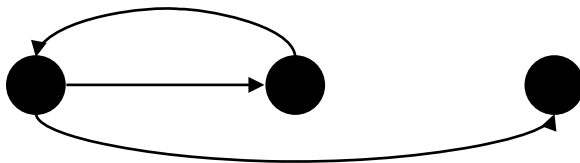
Sequence



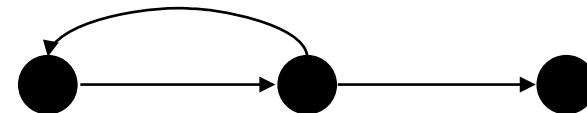
If-then-else



Case/Switch



Do-while loop



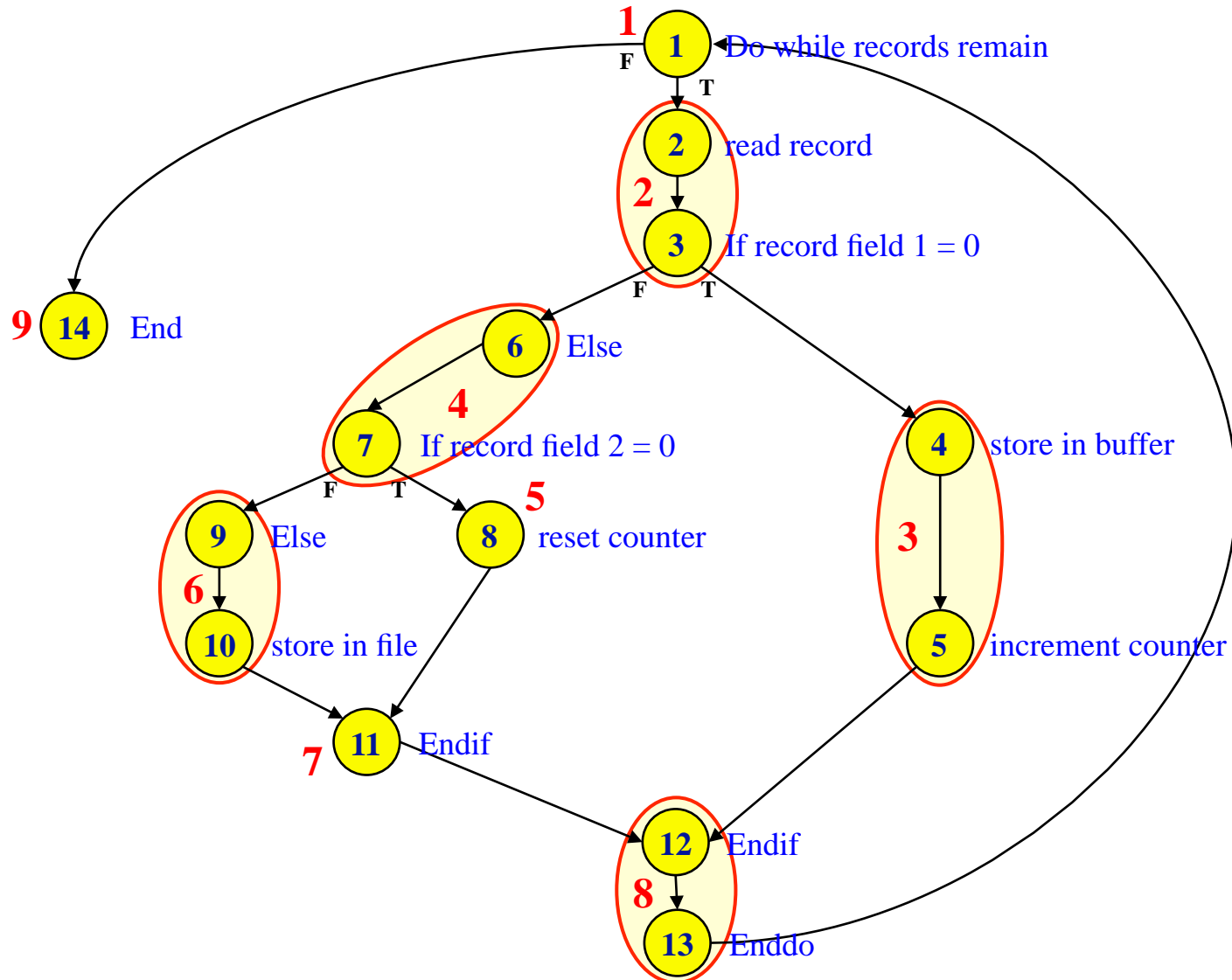
Do-until loop

EXAMPLE BASIS PATH TESTING: FLOW GRAPH

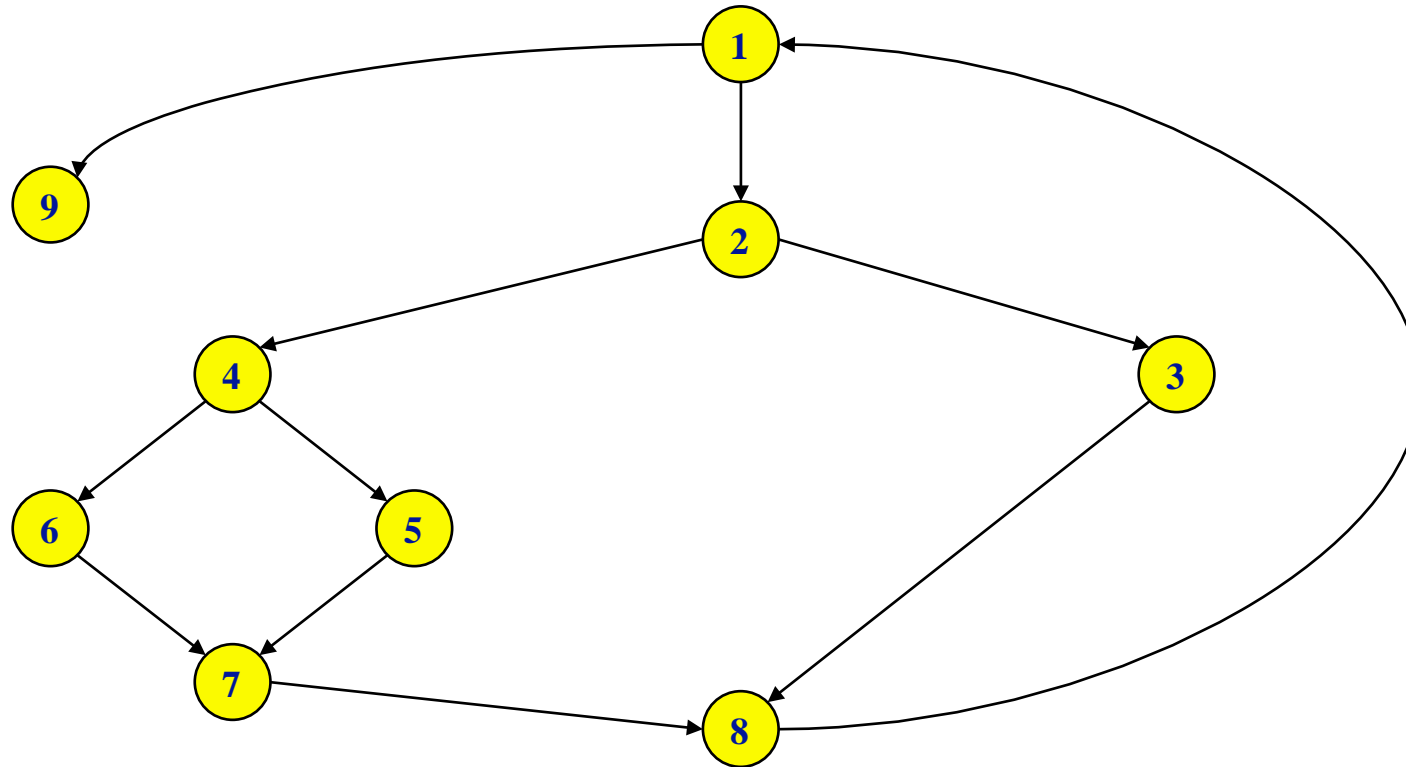
Procedure: process records

1. Do while records remain
2. read record
3. If record field 1 = 0
4. store in buffer
5. increment counter
6. Else
7. If record field 2 = 0
8. reset counter
9. Else
10. store in file
11. Endif
12. Endif
13. Enddo
14. End

EXAMPLE BASIS PATH TESTING: FLOW GRAPH



EXAMPLE BASIS PATH TESTING: FLOW GRAPH



Flow graph node to program statement mapping:

<u>Node</u>	<u>Statement</u>	<u>Node</u>	<u>Statement</u>	<u>Node</u>	<u>Statement</u>
1.	1	4.	6, 7	7.	11
2.	2, 3	5.	8	8.	12, 13
3.	4, 5	6.	9, 10	9.	14

WHITE BOX TESTING: BASIS PATH TESTING (cont'd)

2. Determine the cyclomatic complexity of the flow graph.

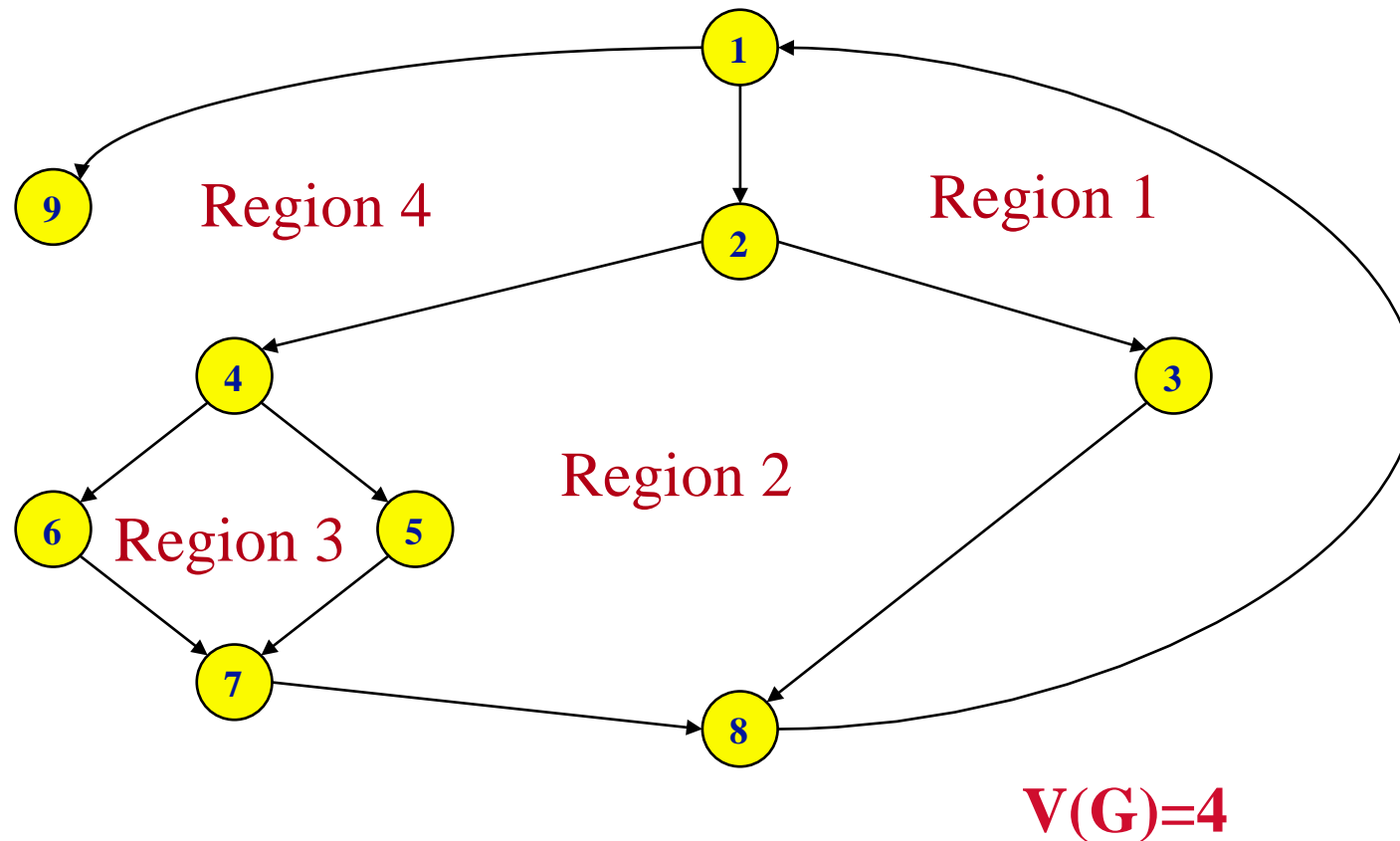
cyclomatic complexity $V(G)$: A quantitative measure of the logical complexity of the code.

Cyclomatic complexity provides an upper bound on the number of paths that need to be tested in the code.

Ways to compute cyclomatic complexity $V(G)$:

- 👉 $V(G)$ = the number of regions (areas bounded by nodes and edges—area outside the graph is also a region)
- 👉 $V(G)$ = the number of edges - the number of nodes + 2
- 👉 $V(G)$ = the number of (simple) predicate nodes + 1

EXAMPLE BASIS PATH TESTING: CYCLOMATIC COMPLEXITY



WHITE BOX TESTING: BASIS PATH TESTING (cont'd)

3. Determine a **basis set of linearly independent paths** based on the cyclomatic complexity.

Independent path: a path that introduces at least one new set of processing statements or a new condition.

☞ An independent path must traverse at least one edge in the flow graph that has not been traversed before the path is defined.

Basis set: the set of linearly independent paths through the code.

☞ A basis set is not unique.

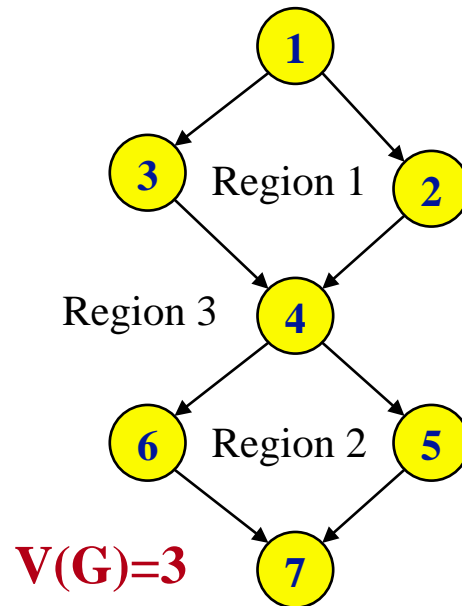
Test cases derived from a **basis set** are guaranteed to execute every statement at least one time during testing.

☞ This is only the minimum number of test cases required.

BASIS PATH TESTING: INDEPENDENT PATHS

Procedure: example()

1. If c1
2. f1()
3. Else
4. f2()
5. Endif
6. If c2
7. f3()
8. Else
9. f4()
10. Endif
11. End

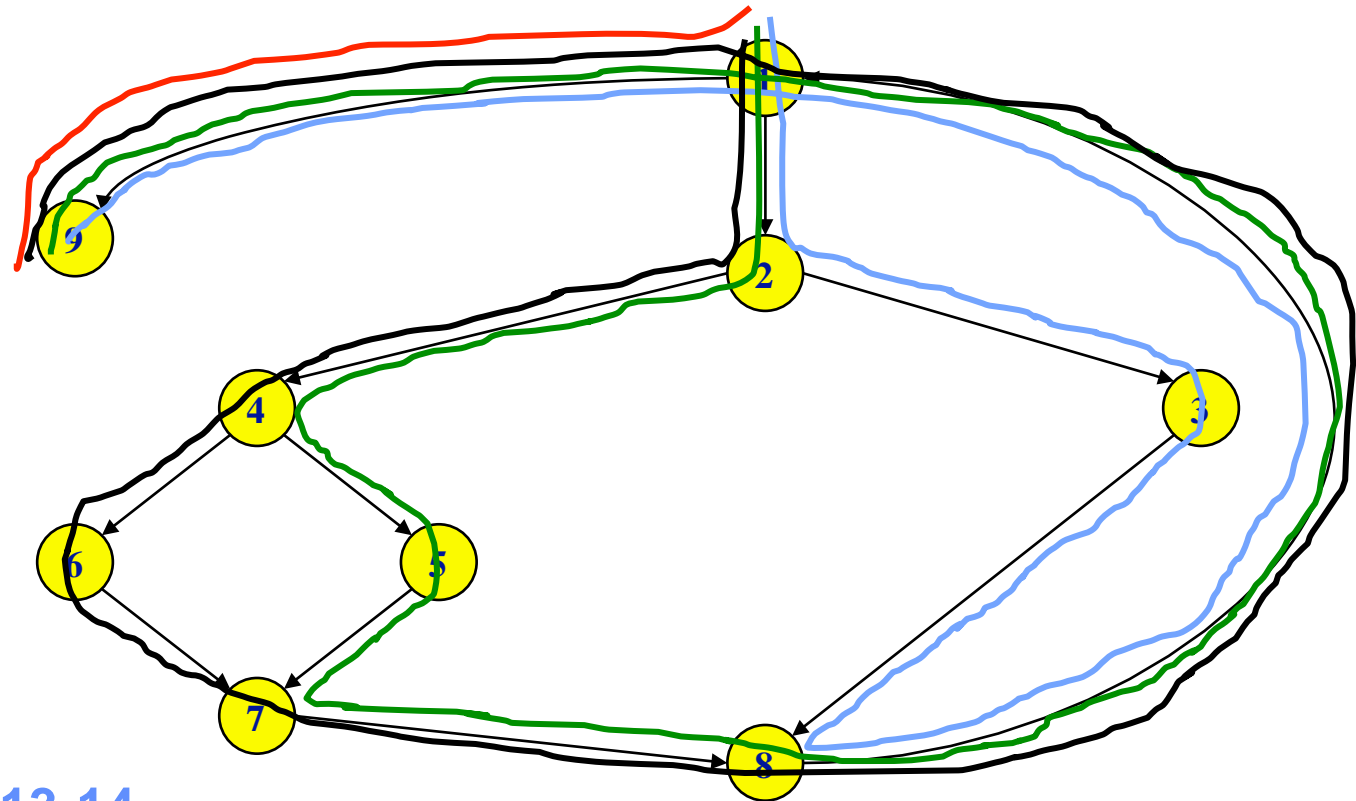


How many independent paths are there in the basis set?

Recall: An independent path introduces at least one new set of statements or a new condition (i.e., it traverses at least one new edge).

👉 $V(G)$ is just an **upper bound** on the number of independent paths.

EXAMPLE BASIS PATH TESTING: INDEPENDENT PATHS



1-14

1-2-3-4-5-12-13-14

1-2-3-6-7-8-11-12-13-14

1-2-3-6-7-9-10-11-12-13-14

WHITE BOX TESTING: BASIS PATH TESTING (cont'd)

4. Prepare test cases that force the execution of each path in the basis set.

Notes:

- The basis set refers to the statement numbers in the program.
- Count each logical test—compound tests count as the number of Boolean operators + 1 (i.e., count each simple predicate).
- Basis path testing should be applied to all components, if possible, and to critical components always.

Basis path testing does not test all possible combinations of all paths through the code; *it just tests every path at least once.*

EXAMPLE ASU WHITE BOX TESTING: BASIS PATH TESTING

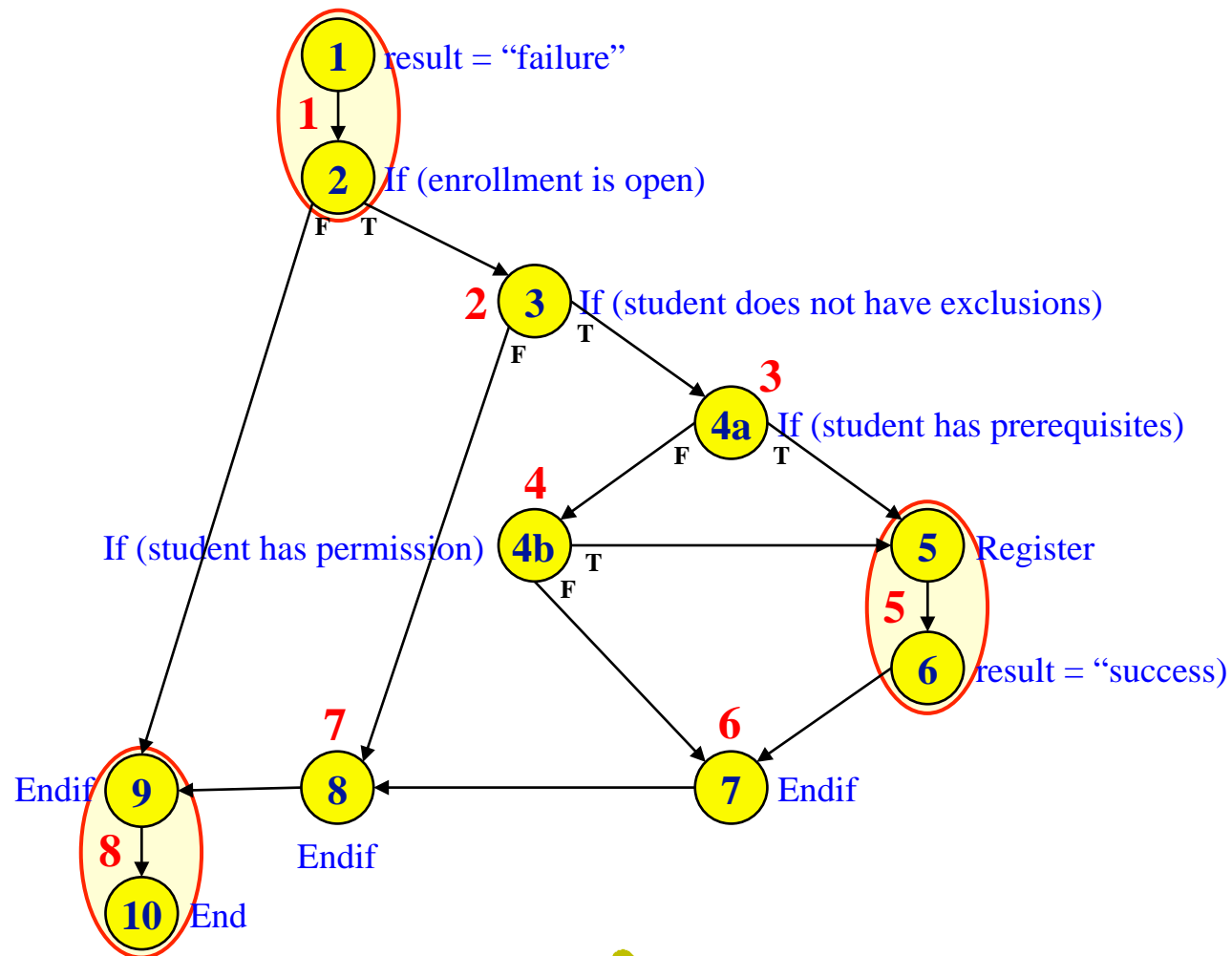
checkRegistration Procedure

A student can register for a course if he has taken the prerequisites. However, he may not register for a course if he has already taken a course that is an exclusion. A student who does not have the prerequisites can register for a course if he is currently registered in the prerequisite course and has the permission of the instructor. Notwithstanding the preceding, the instructor may waive the prerequisite for a student.

```
Procedure: checkRegistration return(result)
1.  result = "failure"
2.  If (enrollment is open)
3.      If (student does not have exclusions)
4.          If (student has prerequisites) OR (student has permission)
5.              Register
6.              result = "success"
7.          Endif
8.      Endif
9.  Endif
10. End
```

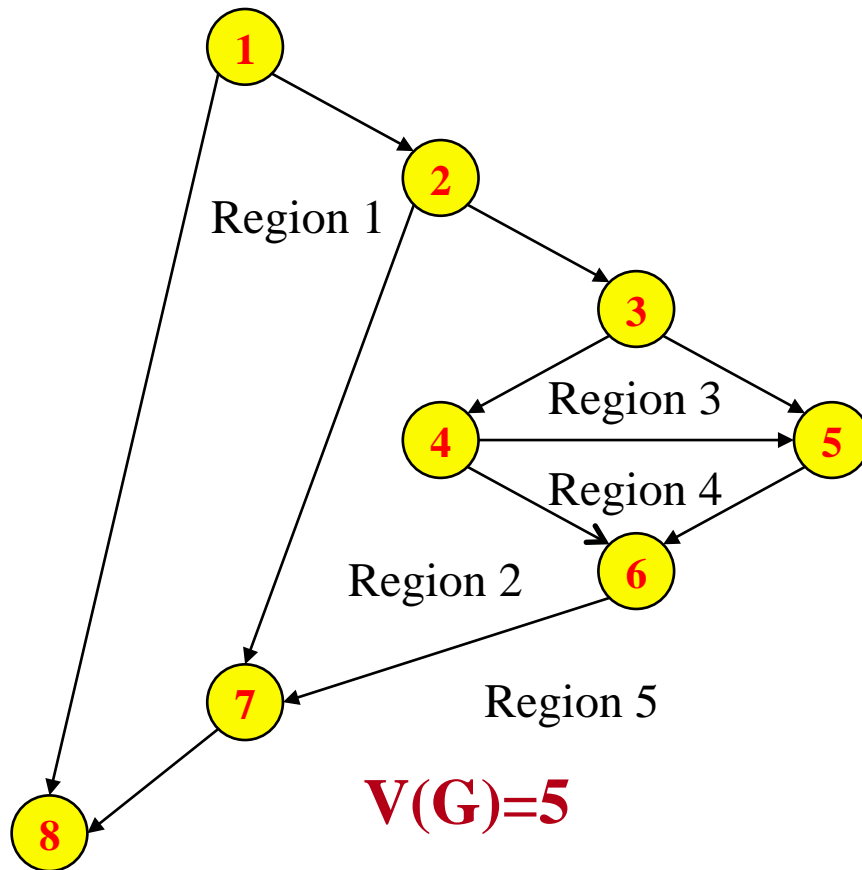
EXAMPLE ASU BASIS PATH TESTING: FLOW GRAPH

Procedure: checkRegistration



EXAMPLE ASU BASIS PATH TESTING: FLOW GRAPH

Procedure: checkRegistration



Flow graph node to program statement mapping:

Node Statement

1. 1, 2
2. 3
3. 4a
4. 4b
5. 5, 6
6. 7
7. 8
8. 9, 10

Basis set (statement numbers):

1. 1 2 3 4a 5 6 7 8 9 10
2. 1 2 3 4a 4b 5 6 7 8 9 10
3. 1 2 3 4a 4b 7 8 9 10
4. 1 2 3 8 9 10
5. 1 2 9 10

EXAMPLE ASU BASIS PATH TESTING: TEST DATA

checkRegistration Procedure

- Basis set path 1:** enrollment is open
student does not have exclusions
student has prerequisites
- Basis set path 2:** enrollment is open
student does not have exclusions
student does not have prerequisites
student has permission
- Basis set path 3:** enrollment is open
student does not have exclusions
student does not have prerequisites
student does not have permission
- Basis set path 4:** enrollment is open
student has exclusions
- Basis set path 5:** enrollment is closed

BASIS PATH TESTING EXAMPLE: VIDEO SALES

The following procedure is used for reserving a copy of a video.

```
Procedure: reserveVideoCopy return(result)
1.   If status = "available"
2.       reserve video copy
3.       status = "reserved"
4.       result = "success"
5.   Else
6.       If (status = "rented") AND (returnDate ≤ requestDate)
7.           reserve video copy
8.           status = "reserved"
9.           result = "success"
10.      Else
11.          result = "failure"
12.      Endif
13.  Endif
14.  End
```

1. Draw the flow graph for the reserveVideoCopy procedure.
2. Show the flow graph node to program statement mapping.
3. Clearly label the regions of the flow graph.

WHITE BOX TESTING: CONDITION TESTING

Condition testing executes the true and false value of each simple logical condition in a component.

Errors in a condition result from errors in:

- simple condition:** $(a \text{ rel-op } b)$ where $\text{rel-op} = \{<, \leq, =, \neq, \geq, >\}$
may be negated with NOT, e.g., $a \leq b$; NOT($a \leq b$)
- compound condition:** two or more simple conditions connected with AND, OR, e.g., $(a > b)$ AND $(c < d)$
- relational expression:** $(E_1 \text{ rel-op } E_2)$ where E_1 and E_2 are arithmetic expressions, e.g., $((a * b + c) > (a + b + c))$
- Boolean expression:** non relational expressions (e.g., NOT A)

Due to incorrect/missing/extra:

- Boolean operator
- Boolean variable
- parenthesis
- relational operator
- arithmetic expression

WHITE BOX TESTING: CONDITION TESTING (cont'd)

1. Branch testing

For a **compound condition C**, test true and false branches of C and every simple condition of C.

e.g., for $C = (a > b) \text{ AND } (c < d)$ we test for:

C	TRUE and FALSE
$a > b$	TRUE and FALSE
$c < d$	TRUE and FALSE

Basis path testing already covers these cases.

2. Domain testing

For an **expression E_1 rel-op E_2** , we test using values that make: E_1 greater than E_2 , E_1 equal to E_2 , and E_1 less than E_2 .

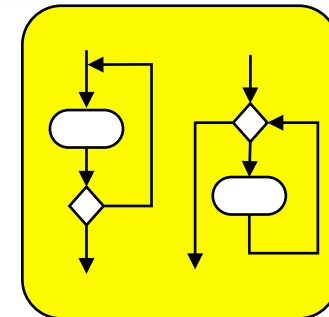
- This **guarantees detection of rel-op error** if E_1 and E_2 are correct.
- To detect errors in E_1 and E_2 , **the difference between E_1 and E_2** for the tests that make E_1 greater than E_2 and E_1 less than E_2 **should be as small as possible.**

WHITE BOX TESTING: LOOP TESTING

Loop testing executes loops at and within their bounds.

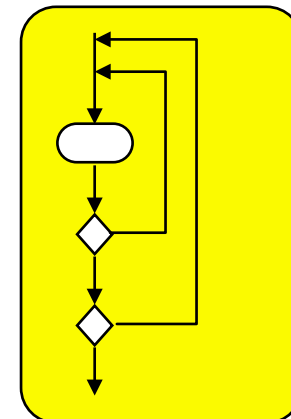
1. Simple Loops (n iterations)

1. 0, 1, 2 passes through the loop.
2. m passes through the loop where $m < n$.
3. $n-1$, n , $n+1$ passes through the loop.



2. Nested Loops

1. Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration values.
2. Work outward, conducting simple loop tests for the next innermost loop.
3. Continue until all the loops have been tested.

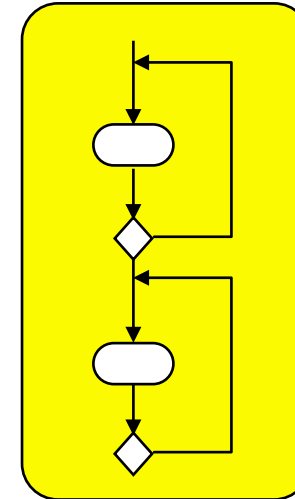


Tests grow geometrically with level of nesting!

WHITE BOX TESTING: LOOP TESTING (cont'd)

3. Concatenated Loops

- ✎ If loops are **independent**
→ use **simple loop testing**
- ✎ If loops are **dependent** (i.e., one loop depends on a variable set in the other loop)
→ use **nested loop testing**



4. Unstructured Loops

- ✎ **Refactor the code!!!**

These loop test have the **highest likelihood of uncovering errors** with the **minimum amount of effort and test overlap.**

WHITE BOX TESTING: DATA FLOW TESTING

Data flow testing ensures that the value of a variable is correct at certain points of execution in the code.

Select test paths according to the **locations** of **definitions** (S) and **uses** (S') of a variable (X).

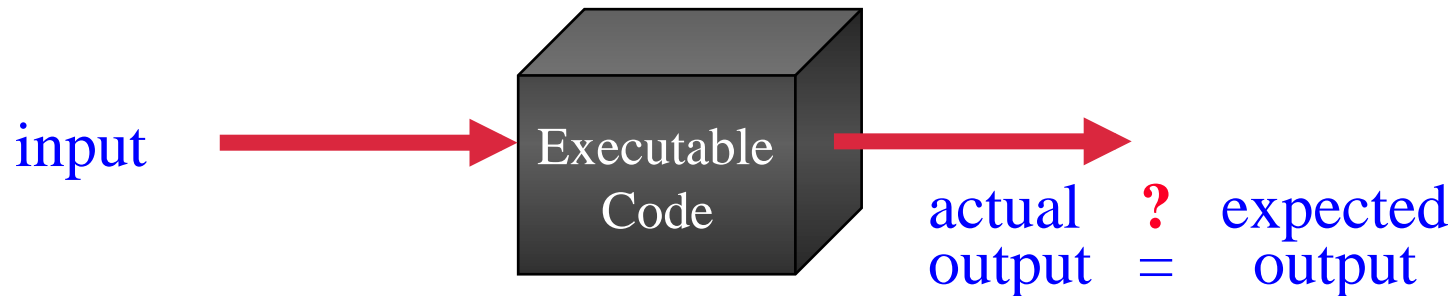
Definition Use (DU) Chain (X) is the set of $[X, S, S']$ where X is
S, S' are statement numbers not redefined between S and S'.

A testing strategy: Every DU chain must be covered once.

For every variable, do a test along the path from where the variable is defined to the statement(s) where the variable is used.

These tests can be combined with basis path testing.

DESIGN TESTS: BLACK BOX TESTING



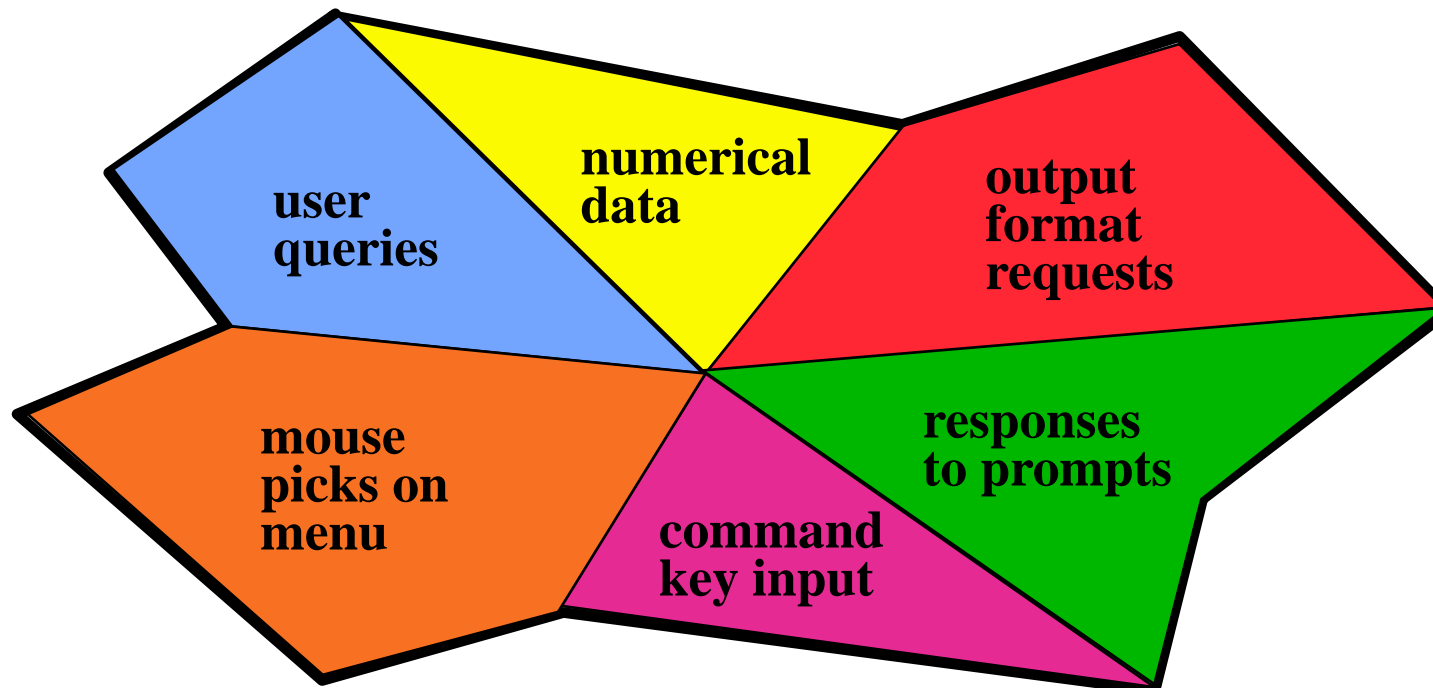
Black box tests attempt to find:

- incorrect or missing functions
- interface incompatibility errors
- data structure or external database access errors
- performance errors
- initialization and termination errors

To achieve reasonable testing, a black box test case should cover a range of input or output values, not just a single value. That is, it should tell us something about the presence or absence of a class of errors (e.g., all character data is correctly/incorrectly processed).

BLACK BOX TESTING: EQUIVALENCE PARTITIONING

Equivalence partitioning groups inputs and outputs by type to create thorough test coverage of a class of errors.



BLACK BOX TESTING: EQUIVALENCE PARTITIONING

IDEA: The input space is very large, the program is small.

A program's behaviour is the “same” for “equivalent” sets of inputs.

- Ideal test suite:
 - Identify sets of inputs with the same behaviour.
 - Test with one value from each set.
- Two problems
 1. The notion of the same behaviour is subtle.
 - Naïve approach: execution equivalence
 - Better approach: revealing subdomains
 2. Discovering the sets of inputs requires perfect knowledge.
 - Use heuristics to approximate the sets cheaply.

NAÏVE APPROACH: EXECUTION EQUIVALENCE

```
int abs (int x) {  
  // returns:   x < 0      => returns -x  
  //           otherwise   => returns x  
  
  if (x < 0)    return -x;  
  else         return x;  
}
```

- All $x < 0$ are execution equivalent.
 - The program *takes the same sequence of steps* for any $x < 0$.
- All $x \geq 0$ are execution equivalent.
 - The program *takes the same sequence of steps* for any $x \geq 0$.

👉 This suggests, for example, that $\{-3, 3\}$ is a good test set.

NAÏVE APPROACH: EXECUTION EQUIVALENCE

- However, consider the following buggy code.

```
int abs (int x) {  
  // returns:  x < 0      => returns -x  
  //           otherwise  => returns x  
  
  if (x < -2) return -x;  
  else       return x;  
}
```

- Two executions:

$x < -2$ $x \geq -2$

- Three behaviours:

$x < -2$ (**OK**) $x = -2$ or -1 (**BAD**) $x \geq 0$ (**OK**)

 Using $\{-3, 3\}$ as the test set does not reveal the error!

BETTER APPROACH: REVEALING SUBDOMAIN

- We say a program has the “same behaviour” for two inputs if it
 1. gives a correct result on both, or
 2. gives an incorrect result on both.
- A subdomain is a subset of the set of possible inputs.
- A subdomain is revealing for an error, E ,
 1. if each element in the subdomain has the same behaviour.
 2. if the program has error E , then it is revealed by the test.
- The trick is to divide the inputs into sets of revealing subdomains (i.e., equivalence partitions) for various errors.

REVEALING SUBDOMAIN EXAMPLE

- For buggy `abs`, what are the revealing subdomains?

```
int abs (int x) {  
  // returns:   x < 0      => returns -x  
  //           otherwise   => returns x  
  if (x < -2)   return -x;  
  else         return x;  
}
```

- Consider these input sets:

`{-1}` `{-2}` `{-3, -1}` `{-3, -2, -1}`

- Which is best to reveal the error?

Why?

REVEALING SUBDOMAIN: HEURISTICS

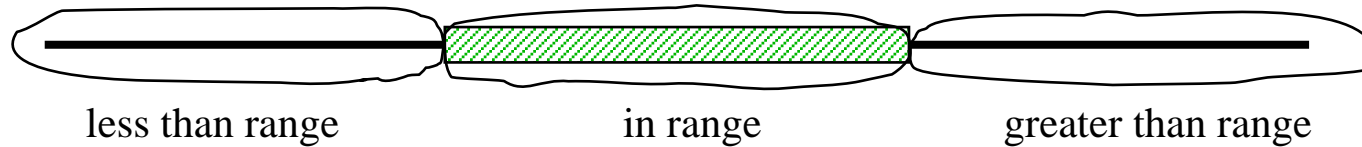
- A good heuristic gives:
 - few subdomains
 - for all errors in some class of errors **E**, high probability that some subdomain is revealing for **e**.
- Different heuristics target different classes of errors.
 - In practice, we usually combine multiple heuristics.

ONE COMMON HEURISTIC

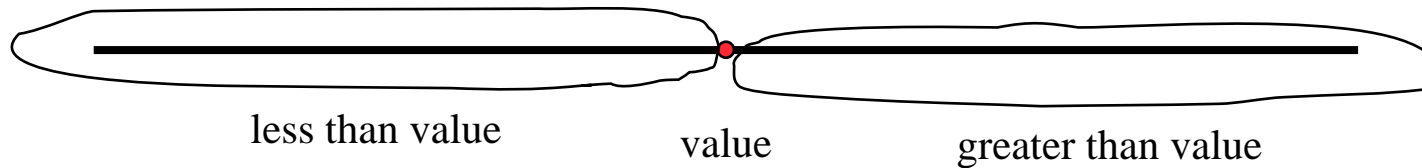
Select **subdomains** based on **valid** and **invalid** inputs.

EQUIVALENCE PARTITIONING: SELECTION GUIDELINES

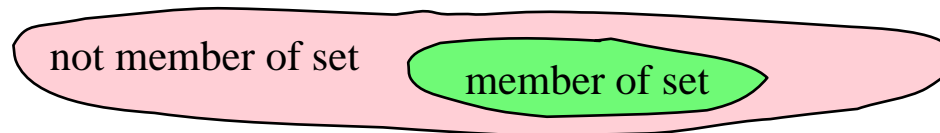
1. If the input is a **range** → **one valid** and **two invalid** subdomains:



2. If the input is a **specific value** → **one valid** and **two invalid** subdomains:



3. If the input is a **set of related values** → **one valid** and **one invalid** subdomain:



4. If the input is **Boolean** → **one valid** and **one invalid** subdomain:



BOUNDARY TESTING

More errors occur at the “boundaries” of a subdomain than in the “center”.

Why?

- off-by-one bugs
 - forget to handle empty container/null object
 - overflow errors in arithmetic
 - program does not handle aliasing of objects
- Small subdomains at the “boundaries” of the main subdomains have a high probability of revealing these common errors.

For testing we select: 1. “typical” values “inside” a subdomain.
2. values at the “boundaries” of a subdomain.



BOUNDARY TESTING EXAMPLE

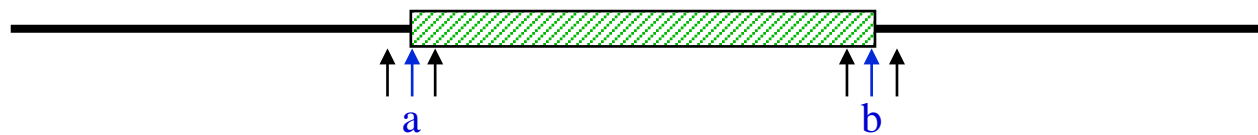
```
public int abs (int x)  
// returns: |x|
```

- Tests for `abs`

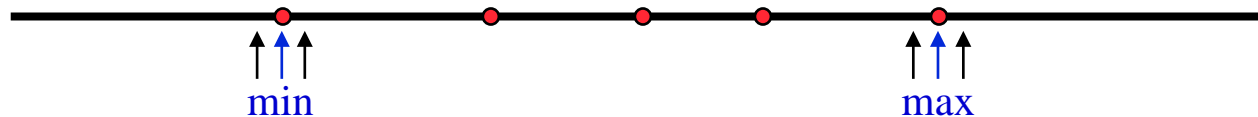
What are some values or ranges of `x` that might be worth testing?

EQUIVALENCE PARTITIONING: VALUE SELECTION GUIDELINES

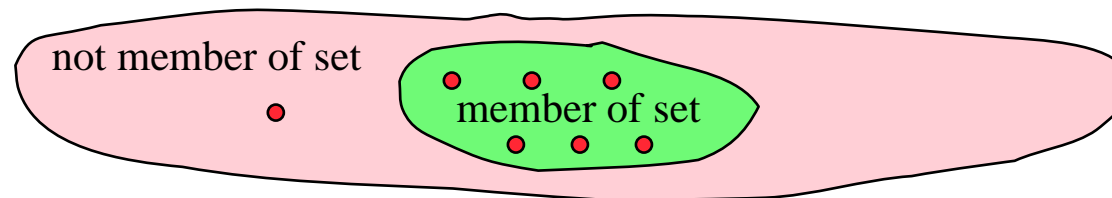
1. If the **input is a range** and is bounded by **a** and **b**, then use **a**, **b**, and values just above and just below **a** and **b**, respectively.



2. If the **input is a number of discrete values**, use the **minimum** and the **maximum** of the values and values just above and just below them, respectively. (Can also be applied to a single specific input value.)

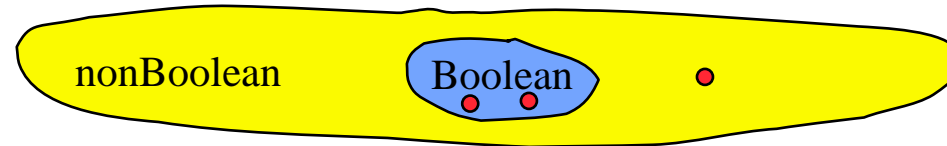


3. If the **input is a set of values**, test **all values in the set** (if possible) and **one value outside the set**.



BOUNDARY TESTING: VALUE SELECTION GUIDELINES

4. If the **input is a Boolean**, test for **both Boolean values** (T, F) and for a **non-Boolean value**.

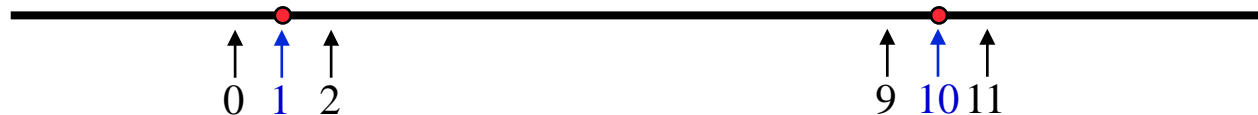


5. Apply guidelines 1 and 2 to create **output values** at the **minimum** and **maximum** expected values.

E.g., if the output is a table, create a minimum size table (1 row) and a maximum size table.

6. If **data structures have boundaries**, test these **boundary values** and values **just above** and **just below** them, respectively.

E.g., for an array with bounds 1 to 10 → test array index = 0, 1, 2, 9, 10, 11.



EXAMPLE ASU BLACKBOX TESTING

Procedure: Student Record Search

In the ASU System, the *Maintain Student Information* use case uses a procedure that searches the database for the record of a given student. The input to the search procedure is a student ID. The output is either a success indication and the student record containing the given student ID or a failure indication and a message indicating the nature of the failure. The valid range of student IDs is from 1000000 to 9999999.

For the following test categories:

- (a) boundary**
- (b) typical**
- (c) other**

what specific test values and test cases should be used to test whether the search procedure works correctly and accepts only valid student IDs?

EXAMPLE ASU BLACKBOX TESTING: SELECTING BOUNDARY VALUES

Boundary values

<u>Test Case</u>	<u>Test Value</u>
1. Minimum range—valid student ID	1000000
2. Minimum range—valid student ID	1000001
3. Minimum range—invalid student ID	999999
4. Maximum range—valid student ID	9999999
5. Maximum range—valid student ID	9999998
6. Maximum range—invalid student ID	10000000

Typical values

<u>Test Case</u>	<u>Test Value</u>
7. Mid-range—valid student ID; existing student record	5000000
8. Mid-range—valid student ID; non-existent student record	5000001
9. Below lower range—invalid student ID	500000
10. Above upper range—invalid student ID	50000000

Other values

<u>Test Case</u>	<u>Test Value</u>
11. Nonnumeric value—invalid student ID	5647ABC

EXAMPLE ASU BLACKBOX TESTING: DESIGNING TEST CASES

Test case procedures: boundary values

Test case 1: Minimum range—valid student ID
Setup: Ensure that a record with student ID 1000000 exists in the data base
Test value: 1000000
Verification: Successful retrieval of student record with ID 1000000

Test case 2: Minimum range—valid student ID
Setup: Ensure that a record with student ID 1000001 exists in the data base
Test value: 1000001
Verification: Successful retrieval of student record with ID 1000000

Test case 3: Minimum range—invalid student ID
Setup: None
Test value: 999999
Verification: Invalid student ID error message

Test case 4: Maximum range—valid student ID
Setup: Ensure that a record with student ID 9999999 exists in the data base
Test value: 9999999
Verification: Successful retrieval of student record with ID 9999999

EXAMPLE ASU BLACKBOX TESTING: DESIGNING TEST CASES

Test case 5: Maximum range—valid student ID
Setup: Ensure that a record with student ID 9999998 exists in the data base
Test value: 9999998
Verification: Successful retrieval of student record with ID 9999998

Test case 6: Maximum range—invalid student ID
Setup: None
Test value: 10000000
Verification: Invalid student ID error message

Test case procedures: typical values

Test case 7: Mid-range—valid student ID; existing student record
Setup: Ensure that a record with student ID 5000000 exists in the data base
Test value: 5000000
Verification: Successful retrieval of student record with ID 5000000

Test case 8: Mid-range—valid student ID; non-existent student record
Setup: Ensure that no record with student ID 5000001 exists in the data base
Test value: 5000001
Verification: Unsuccessful retrieval and no such record error message



EXAMPLE ASU BLACKBOX TESTING: DESIGNING TEST CASES

Test case 9: Below lower range—invalid student ID
Setup: None
Test value: 500000
Verification: Invalid student ID error message

Test case 10: Above upper range—invalid student ID
Setup: None
Test value: 50000000
Verification: Invalid student ID error message

Test case procedures: other values

Test case 11: Nonnumeric value—invalid student ID
Setup: None
Test value: 5647ABC
Verification: Invalid student ID error message

BLACKBOX TESTING EXAMPLE: VIDEO SALES

Procedure: validateRating

In the video sale and rental shop system, there is a procedure to validate the review rating input by the user. Recall that a review can be given a rating, which is an integer value, from 1 to 5.

For each of the following test categories:

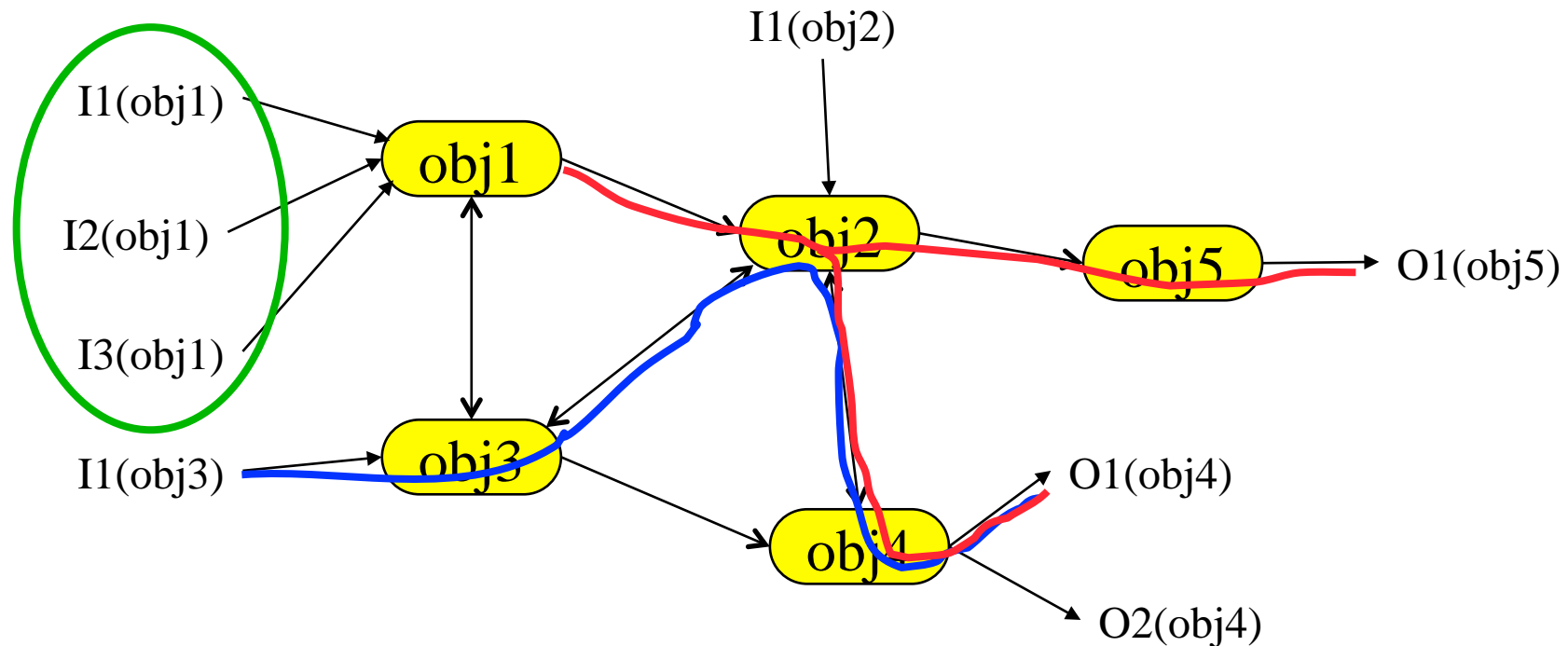
- (a) boundary
- (b) typical
- (c) other

what **specific test values** should be used to test whether the validateRating procedure works correctly and accepts only valid rating values?

BLACK BOX TESTING: THREAD TESTING

- An **event-based approach** where tests are based on **events** which **trigger system actions**.
✎ **Particularly appropriate for object oriented systems.**
- It is used after classes have been **unit tested** and **integrated into subsystems**.
- Need to **identify** and **execute** each possible **processing thread**.
✎ **Examine the use cases.**
- However, it may not be possible to do complete thread testing.
✎ **Since there are usually too many input/output combinations.**
- Therefore, we focus on the **most commonly executed threads**.
✎ **These are the basic flow of events of use cases.**

BLACK BOX TESTING: THREAD TESTING (cont'd)



Types of thread tests include:

- **single thread:** obj3→obj2→obj4
- **multiple thread:** obj1→obj2→{obj4; obj5}
- **multi-input thread:** into obj1

BLACK BOX TESTING: STATE-BASED TESTING


- State-based testing focuses on **comparing** the **resulting state** of **a class** with the **expected state**.

👉 **Derive test cases using state machine diagram for a class.**

- Derive a **representative set of “stimuli”** (events) for each **transition** (similar to equivalence testing).
- **Check the attributes or links** of the class after applying a “stimuli” to determine if the specified state has been reached.

👉 **We first need to put the class in the desired state before applying the “stimuli”.**

REGRESSION TESTING

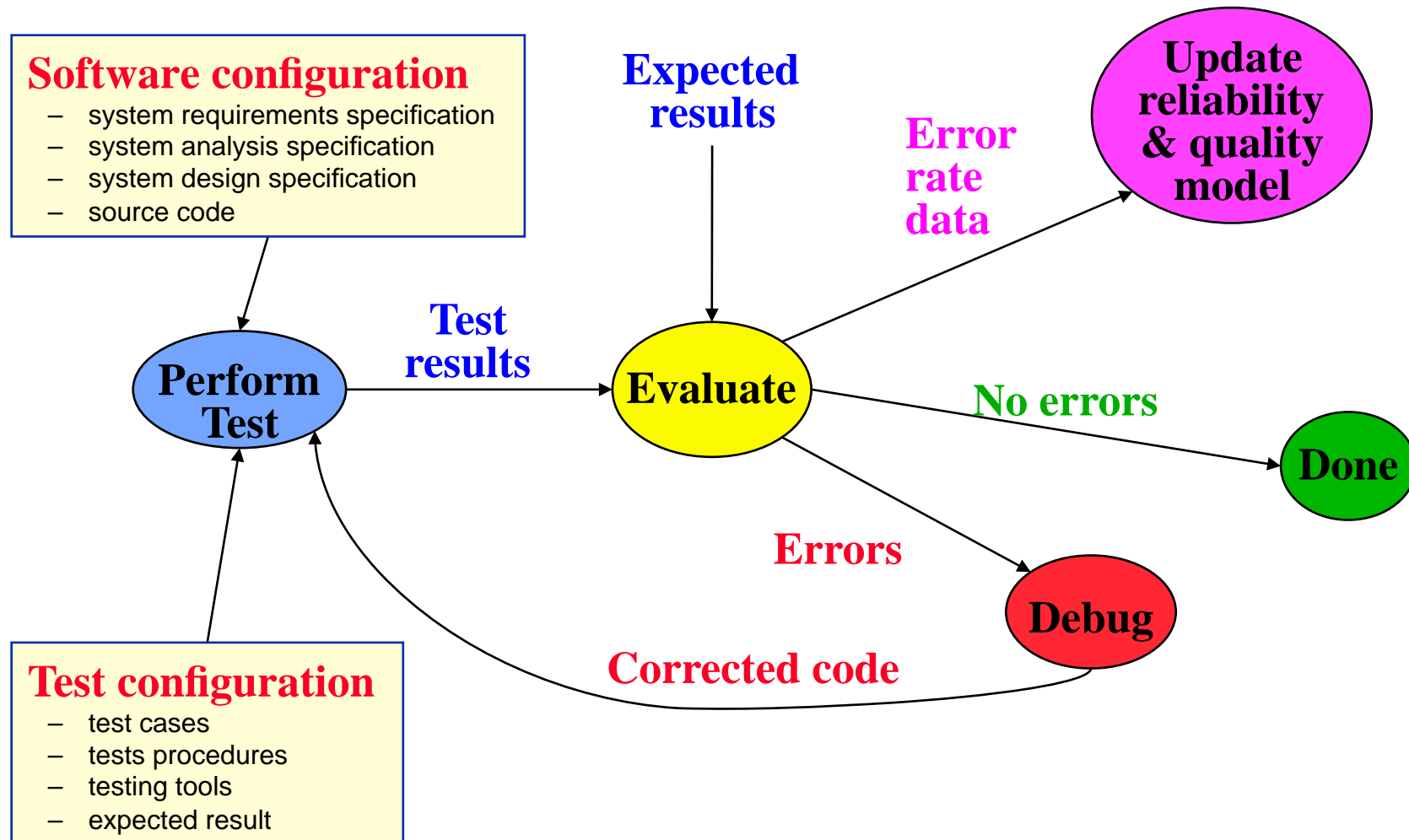
- **Whenever you find a bug:**
 - Reproduce it (before you fix it).
 - Record the input that produced that bug.
 - Record the correct output.
 - Put everything into the test suite.
 - Then, fix the bug and verify the fix.
 - **Why is this a good idea?**
 - It helps to populate the test suite with **good tests**.
 - It protects against changes that **reintroduce the bug**.
-  **It happened once, so it might happen again!**

IMPLEMENT TESTS

Goal: To automate test procedures as much as possible.

- Running test cases can be very **tedious** and **time consuming**!
 - There are many possible input values and system states to test.
 - A **test component** is a program that automates one or several test procedures or parts of them.
 - There are tools available to help write test components that:
 - **record the actions** for a test case as the user performs the actions.
 - **parameterize the recorded script** to accept a variety of input values.
- 👉 **Spreadsheets and/or database applications can be used to store the required input data and the results of each test.**

PERFORM TESTS



PERFORM TESTS: TESTING STRATEGY

A **testing strategy** specifies which testing techniques (white box, black box, thread, etc.) are appropriate at which point in time.

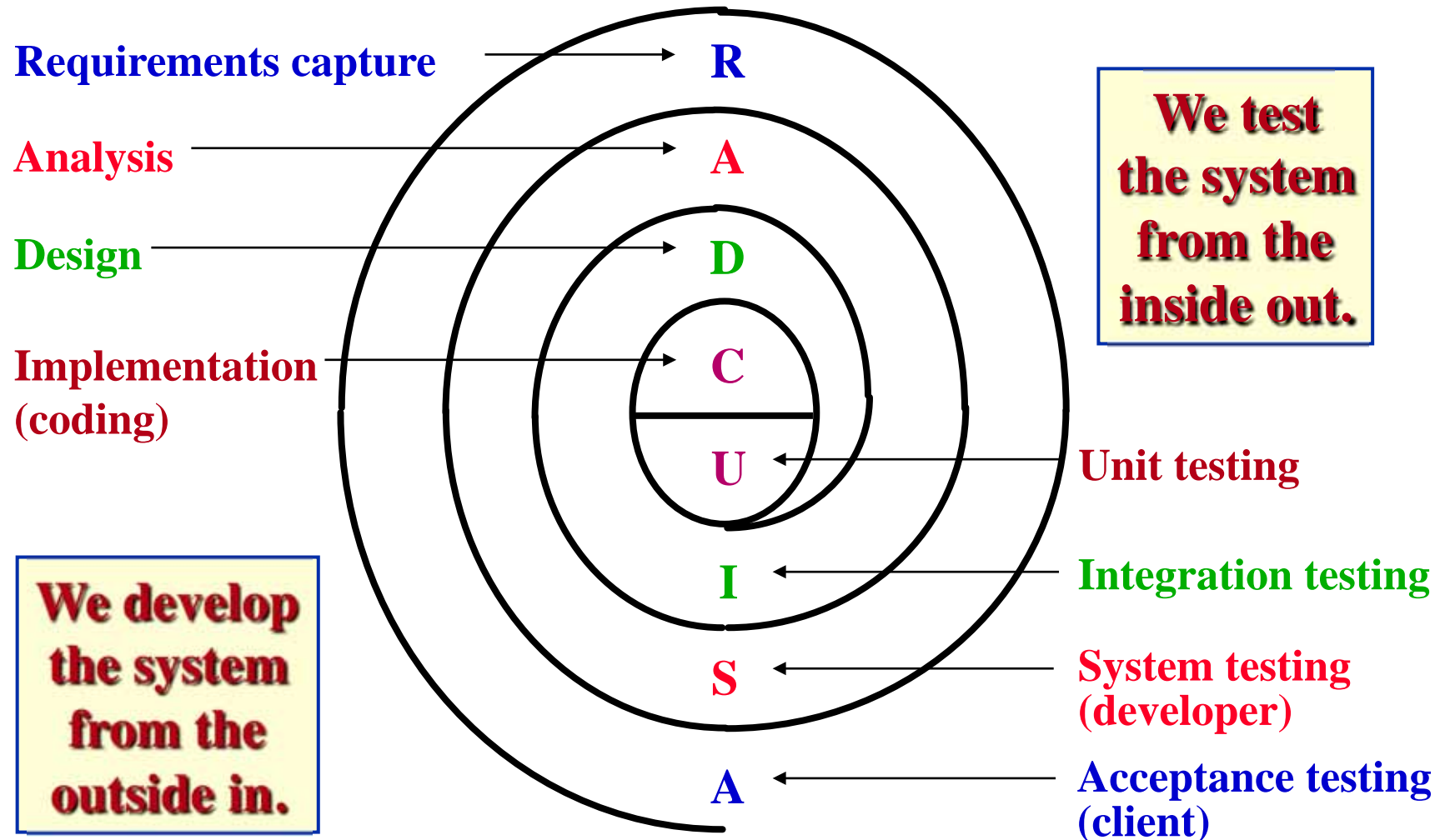
- A testing strategy integrates test cases into a well-planned series of steps that test a component/subsystem by specifying:
 - What are the steps that need to be conducted to test a component?
 - When are the steps planned and undertaken?
 - How much effort, time and resources will be required to do the steps?

However, test planning is difficult due to the time uncertainty in the debug part!

👉 Consequently, a testing strategy needs to balance:
flexibility & creativity with planning & management

Testing often occurs when deadline pressures are most severe.
Progress must be measurable and problems identified as early as possible.

A TESTING STRATEGY



A TESTING STRATEGY

Who should do the testing?

Unit Testing (using White Box testing)

- Verifies that each component/subsystem functions correctly.
 - Done by component engineer who develops the code.

Integration Testing (using Black Box & White Box testing)

- Verifies that the components/subsystems interact correctly.
 - Done by component engineer and/or independent test group (integration/system tester).

System Testing (using Black Box testing)

- Verifies that the system functions correctly as a whole.
 - Done by independent test group (integration/system tester).

Acceptance Testing (using Black Box testing)

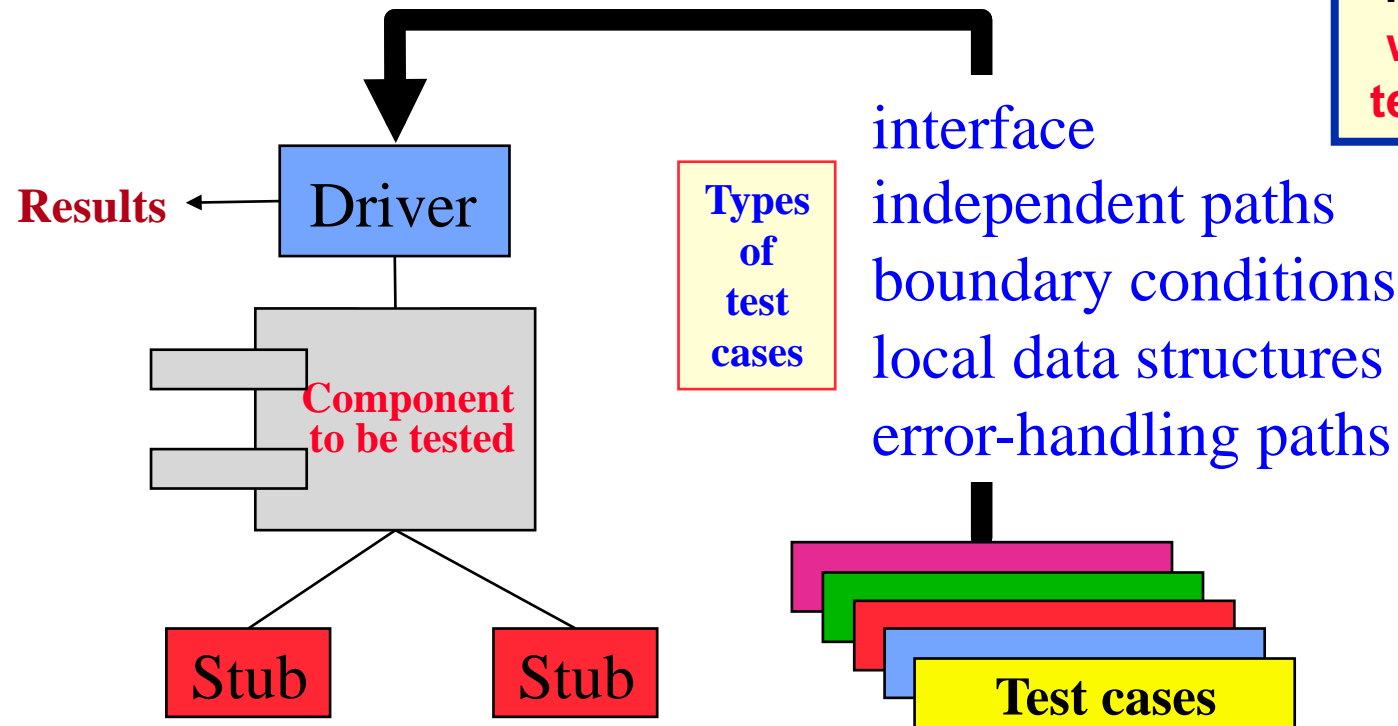
- Validates the software against its requirements.
 - Done by client/user.

D
E
V
E
L
O
P
E
R
S

U
S
E
R
S



UNIT TESTING



Emphasis is mainly on **white box techniques**.

driver: a component that calls the component to be tested
stub: a component called by the component to be tested

Driver and/or stubs may need to be developed for each unit test.

UNIT TESTING: TEST CASES

- **interface**
 - These tests ensure that information properly flows in and out of the component.
- **independent paths**
 - These tests ensure that all paths in a component have been executed at least once.
- **boundary conditions**
 - These tests ensure that the component operates properly at boundaries established to limit or restrict processing.
- **local data structures**
 - These tests ensure that data stored temporarily maintains its integrity during execution.
- **error-handling paths**
 - These tests ensure that errors are correctly handled, but they are *often overlooked or ignored!*

UNIT TESTING: OBJECT-ORIENTED TESTING

What to unit test?

- A unit test has to be **at least a class** (i.e., we need to check an **object's behaviour**), but object state makes testing difficult.

 **A class must be tested in every state it can ever enter (i.e., use state-based testing).**

How to deal with inheritance and polymorphism?


- If a subclass **overrides methods** of an already tested superclass, what needs to be tested — only the overridden methods?

 **No! All of a subclass's methods need to be tested again due to dynamic binding and substitutability.**

UNIT TESTING: OBJECT-ORIENTED TESTING

How to deal with encapsulation?

- Encapsulation **hides what is inside an object** → hard to know its state.

 We need to provide a method, for testing only, that **reports all of an object's state**.

Example: Testing a stack (push, pop, *peek*)

Suppose popping an empty stack fails somehow.

- Peeking at an empty stack should show nothing.
- Push an element onto the stack, then **peek at the stack** expecting the same element you pushed.
- Push an element onto the stack, then pop it, expecting the same element you pushed and **peek at the stack** expecting the stack to be empty.

INTEGRATION TESTING

If components all work individually, **why more testing?**

Interaction errors cannot be uncovered by unit testing!
(e.g., interface misuse, interface misunderstanding, timing errors, etc.)

Integration approaches



versus

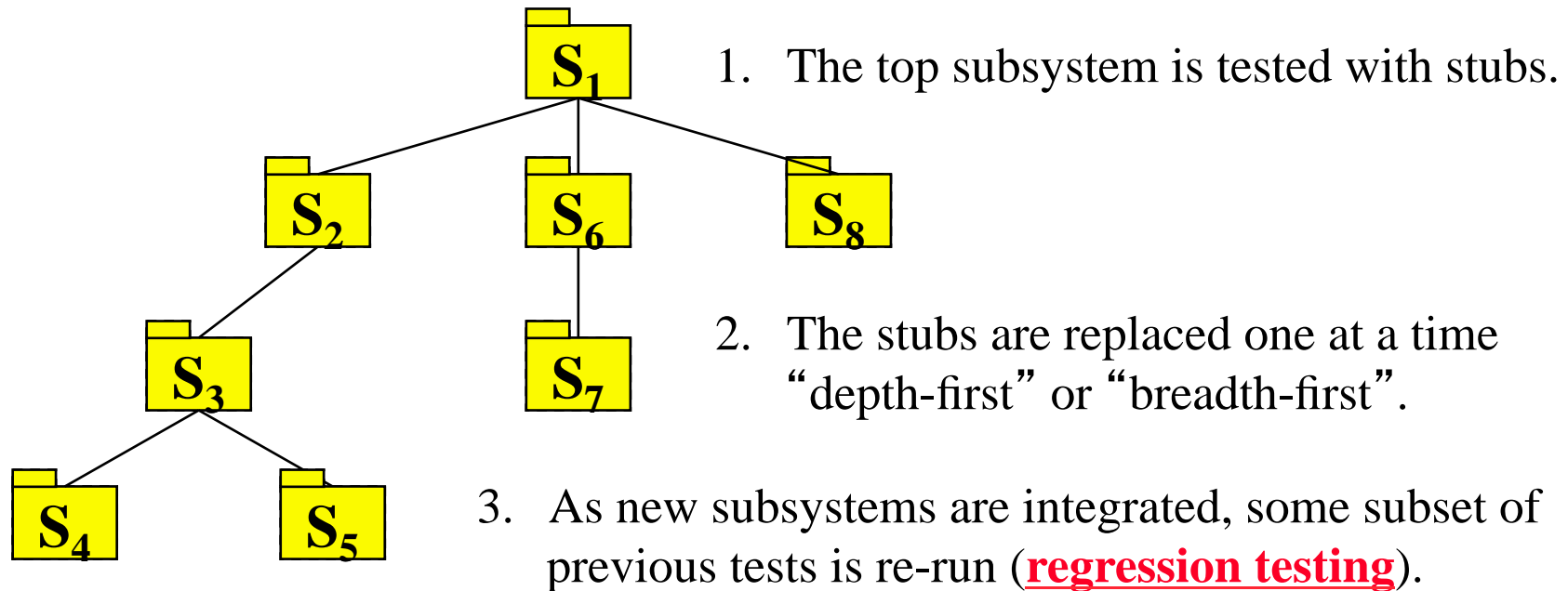
Incremental
construction
strategy

Incremental
“builds”

Regression
testing



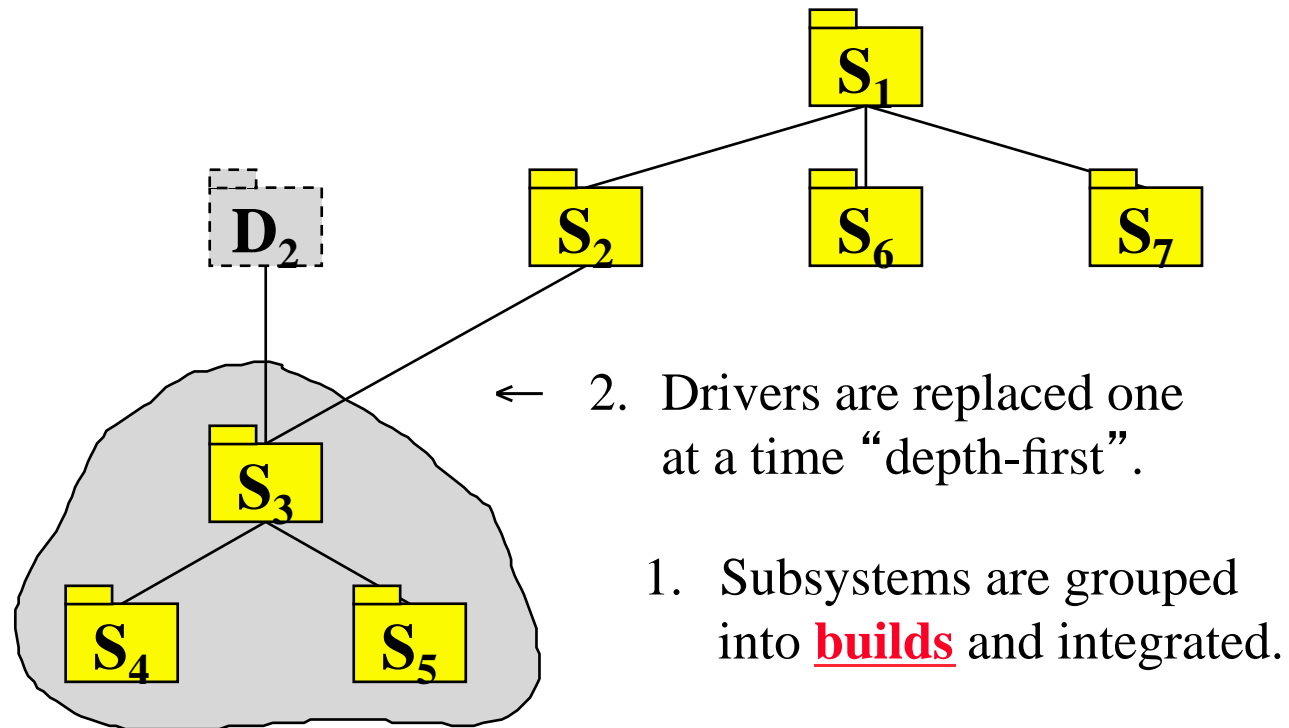
INTEGRATION TESTING: TOP DOWN



Pro: Early testing and error detection of user interface components; can demonstrate a complete function of the system early.

Con: Cannot do significant low-level processing until late in the testing; need to write and test stubs.

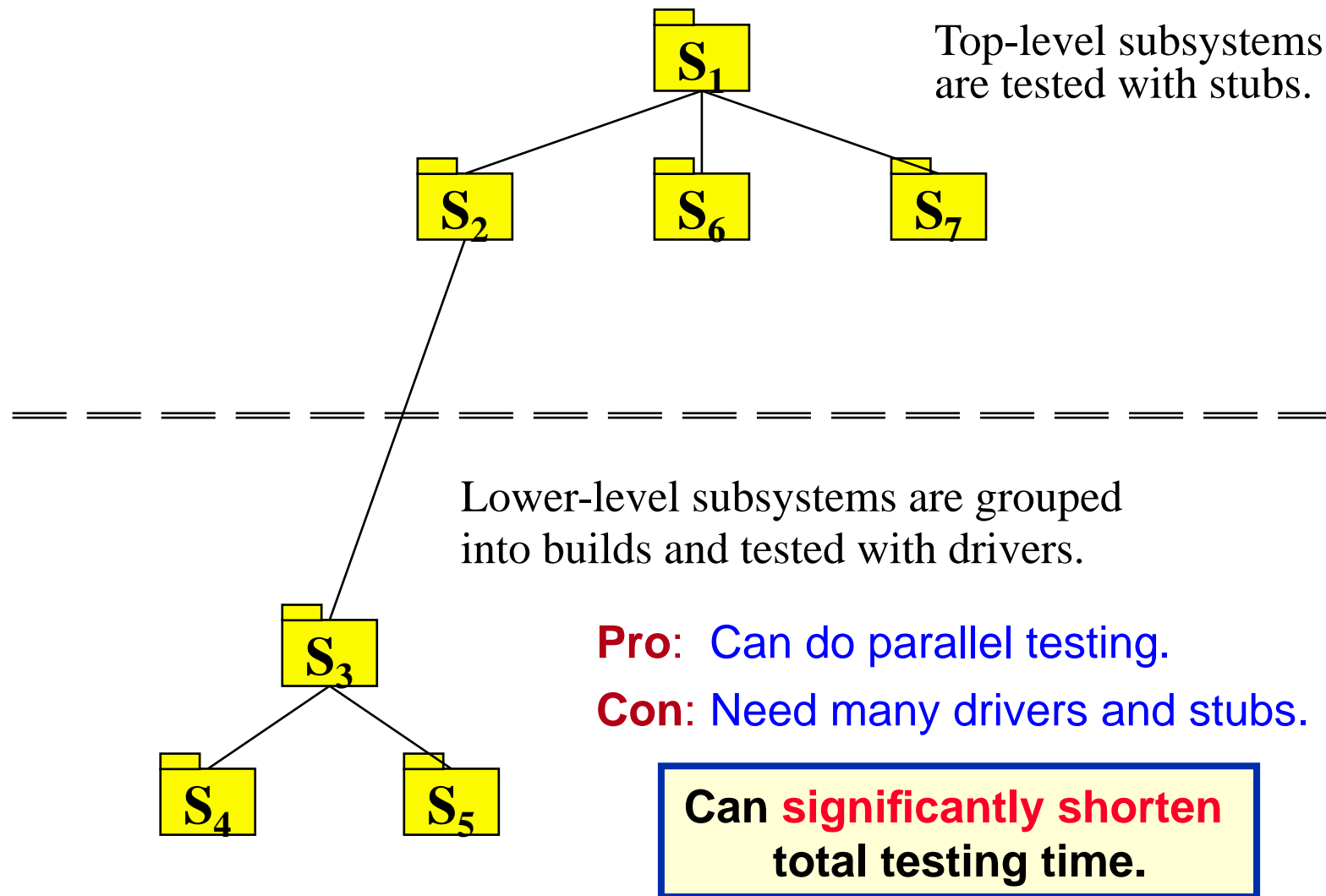
INTEGRATION TESTING: BOTTOM UP



Pro: Interaction faults are more easily found; easier test case design and no need for stubs.

Con: User interface components are tested last.

INTEGRATION TESTING: SANDWICH



INTEGRATION TESTING: CRITICAL SUBSYSTEMS

Critical subsystems should be tested as early as possible!

Critical subsystems are those that:

1. have **high risk**.
2. **address several software requirements** (i.e., they implement several use cases).
3. have a **high level of control**.
4. are **complex** or **error prone** → **high cyclomatic complexity**.
5. have **specific performance requirements**.

Regression testing is required for critical subsystems!





SYSTEM TESTING

System testing is testing of the entire system to be sure the system functions properly when integrated.

Some specific types of system tests:

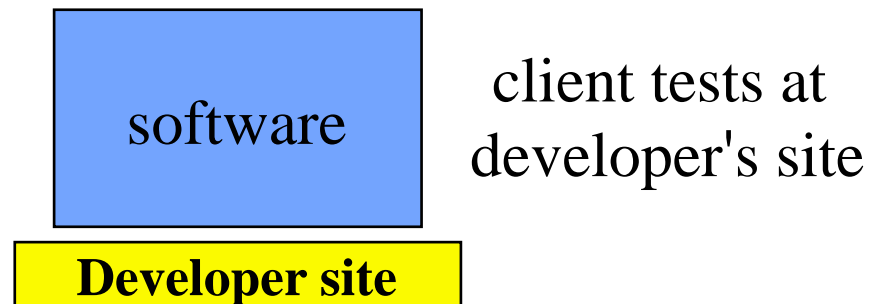
- Functional** → The **developers** **verify** that all user functions work as specified in the **system requirements specification**.
- ➔ **Performance** → The **developers** **verify** that the **design goals** (nonfunctional requirements) are met.
- ➔ **Pilot** → A **selected group of end users** **verifies** common **functionality** in the target environment.
- ➔ **Acceptance** → The **client/user** **verifies** **usability** and **validates** functional and nonfunctional requirements against the system requirements specification.
- Installation** → The **client/user** **verifies** **usability** and **validates** functional and nonfunctional requirements in real use.

SYSTEM TESTING: PERFORMANCE TESTING

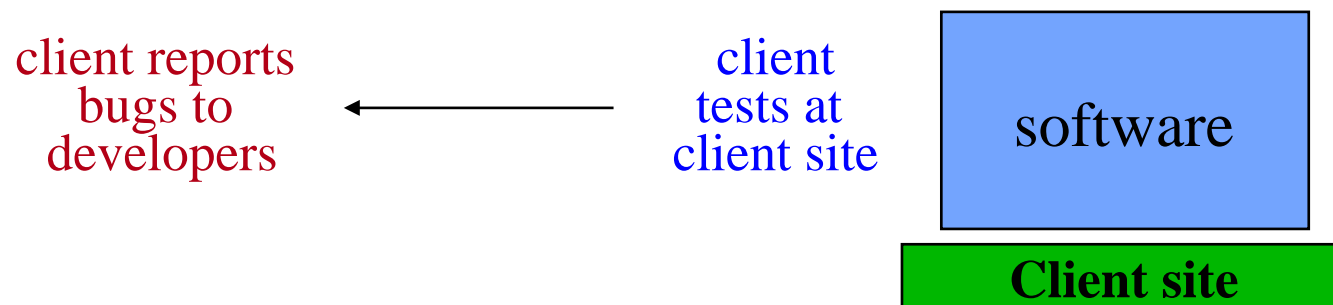
- stress** Verify that the system can continue functioning when confronted with **many simultaneous requests**.
 How high can we go? Do we fail-soft or collapse?
- volume** Verify that the system can handle **large amounts of data, high complexity algorithms, or high disk fragmentation**.
- security** Verify that **access protection mechanisms** work.
 Make penetration cost more than value of entry.
- timing** Verify that the system meets **timing constraints**.
 Usually for real-time and embedded systems.
- recovery** Verify that the system can **recover when forced to fail** in various ways.
 Database recovery is particularly important.

SYSTEM TESTING: PILOT TESTING

Alpha test: A test in a **controlled environment** so that developers can **observe users**.



Beta test: A test in a **real environment** so that bugs are uncovered from **regular usage patterns**.



SYSTEM TESTING: ACCEPTANCE TESTING

An acceptance test demonstrates to the client that a function or constraint of the system is fully operational.

Functional validity – Does the system provide the required functionality?

☞ **ASU:** Show that a professor can select a course offering to teach.

Interface validity – Do interfaces perform desired functions (accept desired input/provide desired output) and follow required design standards?

☞ **ASU:** Show that all data for course registration can be input.

Information content – Are the stored data correct (i.e., in the required format and obey the required constraints)?

☞ **ASU:** Show that all information of a student's course schedule is correct.

☞ **ASU:** Show that a student cannot register for more than four courses.

Performance – Does the system meet specified performance criteria?

☞ **ASU:** Show the response time to register for a course is less than 1 second.

SYSTEM TESTING: DERIVING ACCEPTANCE TESTS

- **Restate** written requirements in a **concise, precise** and **testable** way by:
 - grouping related requirements.
 - removing any requirements that cannot be tested.
- **Add** any additional requirements gathered from users by:
 - looking at **use cases** for **functional** and **interface** requirements.
 - looking at **domain model** for **information** content requirements.
 - looking at **nonfunctional requirements** for **performance** requirements.
- **Construct**, for each requirement, an **evaluation scenario** that will demonstrate to the client that the requirement is met.
(Since most evaluation scenarios depend on the user interface, they can not be completed until the user interface is designed.)

ASU COURSE REGISTRATION REQUIREMENTS

At the beginning of each term, students may request a course catalogue containing a list of course offerings needed for the term. Information about each course, such as instructor, department, and prerequisites are included to help students make informed decisions.

The new system will allow students to select four course offerings for the coming term. In addition, each student will indicate two alternative choices in case a course offering becomes filled or is canceled. No course offering will have more than forty students or fewer than ten students. A course offering with fewer than ten students will be canceled. Once the registration process is completed for a student, the registration system sends information to the billing system so the student can be billed for the term.

Professors must be able to access the online system to indicate which courses they will be teaching, and to see which students signed up for their course offerings.

For each term, there is a period of time that students can change their schedule. Students must be able to access the system during this time to add or drop courses.

ASU: REQUIREMENTS PRECISELY STATED & GROUPED

1. Students can request course catalogues containing course offerings for the term.
2. Students decide which courses to take.
3. Course registration:
 - 3.1. Students can select up to four course offerings.
 - 3.2. Students can select up to two alternate course offerings.
 - 3.3. Students can add course offerings to their schedule.
 - 3.4. Students can drop course offerings from their schedule.
 - 3.5. A course offering cannot have more than 40 students enrolled.
 - 3.6. A course offering with less than 10 students enrolled will be canceled.
4. The Registration System sends information to the Billing System.
5. Courses to teach:
 - 5.1. Professors can choose courses to teach for the coming term.
 - 5.2. Professors can request class lists for the courses they are teaching.

ASU: REQUIREMENTS RESTATED IN A TESTABLE WAY

1. The system must be able to produce a course catalogue containing course offerings for a term.
2. Deleted since it is not testable!
3. Course registration:
 - 3.1. Students can select up to four course offerings.
 - 3.2. Students can select up to two alternate course offerings.
 - 3.3. Students can add course offerings to their schedule.
 - 3.4. Students can drop course offerings from their schedule.
 - 3.5. A course offering cannot have more than 40 students enrolled.
 - 3.6. A course offering with less than ten students enrolled will be canceled.

ASU: REQUIREMENTS RESTATED IN A TESTABLE WAY

4. The Registration System sends the following information to the Billing System for each student:
 - 4.1. The student's id, surname, otherNames, address, modeOfStudy, yearOfStudy
 - 4.2. The code, title and credits for each course offering in which the student is enrolled.
5. Courses to teach:
 - 5.1. Professors can choose courses to teach for the coming term.
 - 5.2. Professors can request class lists for the courses they are teaching.

There will be other requirements that have been gathered from the users.

ASU: ACCEPTANCE TESTS

1. Demonstrate that the system can produce a course catalogue containing course offerings for a term.
2. Course registration:
 - 2.1. Demonstrate that course offerings can be selected up to a maximum of four.
 - 2.2. Demonstrate that alternate course offerings can be selected up to a maximum of two.
 - 2.3. Demonstrate that a course offering can be created and that a course offering can be added to an existing (i.e., non-empty) schedule.
 - 2.4. Demonstrate that students can drop course offerings from their schedule.
 - 2.5. Enroll students in course offerings and show that no more than 40 students can be enrolled in any one course offering.
 - 2.6. Enroll less than 10 students in a course offering and show that it is canceled at the end of the registration period.

ASU: ACCEPTANCE TESTS (cont'd)

3. Demonstrate that the Registration System can send the required information to the Billing System for each student.
4. Courses to teach:
 - 4.1. Demonstrate that professors can choose courses to teach for the coming term.
 - 4.2. Demonstrate that professors can display class lists for the courses they are teaching.

These tests must be made “operational” by devising test cases for use by the client.

ACCEPTANCE TESTS EXAMPLE: VIDEO SALES

The following statements are extracted from the video sales and rental problem statement.

- The system must be able to handle both physical and digital videos.
- It must be able to record which videos are sold and rented and by whom.
- Members ... can buy and rent digital videos via the Web.
- A sales clerk should be able to sell and rent physical videos and process the return of rented physical videos.

Restate these requirements in a testable way and group them by functionality.

TEST DRIVEN DEVELOPMENT

- Incremental, interleaved code development and testing.
 1. Identify increment of functionality to implement.
 2. Write test for functionality and implement as an automated test.
 3. Run the test with previous tests.
 4. Implement the functionality and re-run all tests.
- Requires automated testing (e.g., Junit, Nunit).
- Fits well with Agile development process.

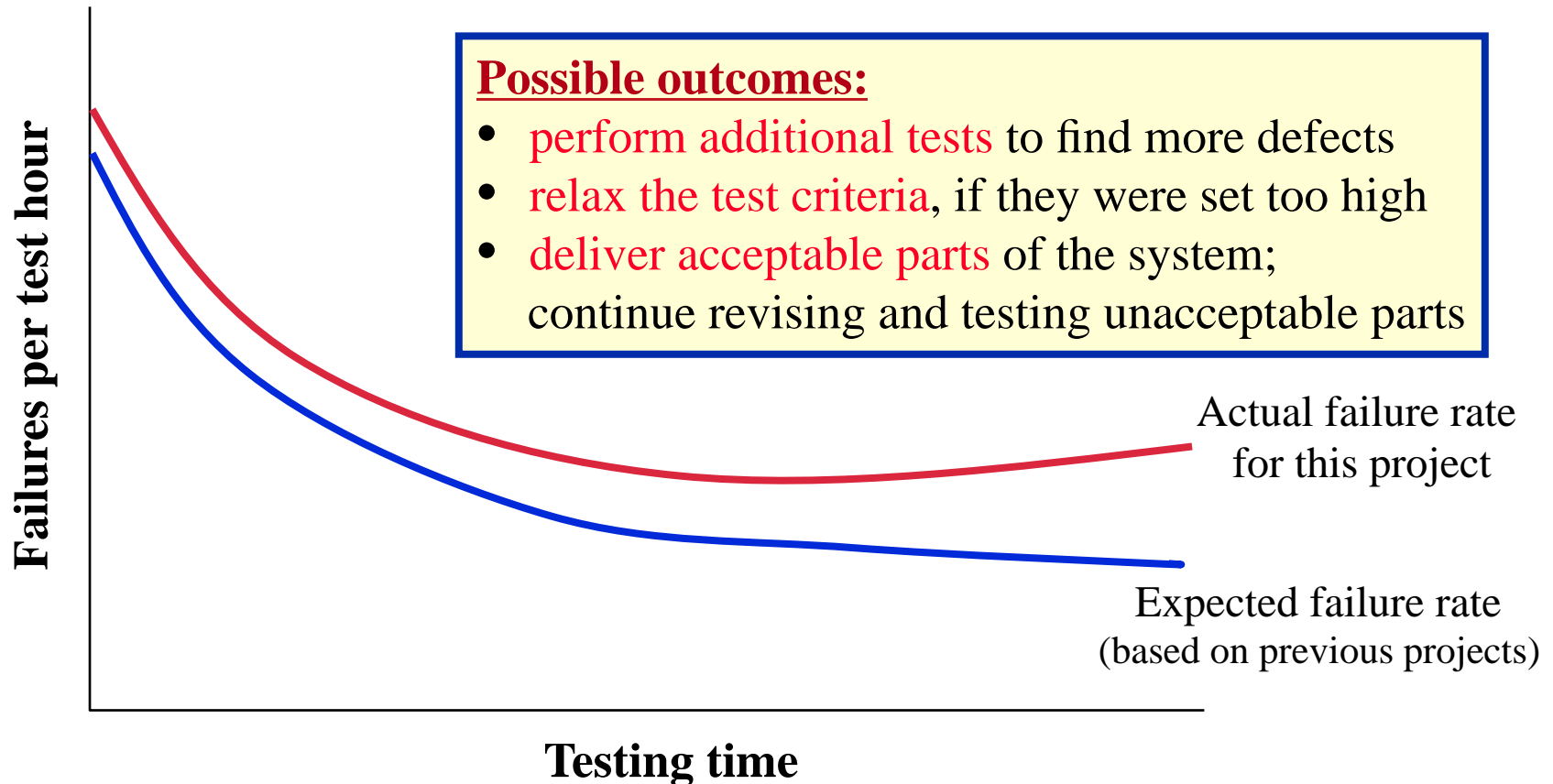
EVALUATE TESTS

- The **test engineer** needs to **evaluate the results** of the testing by:
 - **comparing the results** with the **goals outlined in the test plan**.
 - **preparing metrics** to determine the current quality of the software.

 **How do we know when to stop testing?**

- We can consider the system's:
 1. **testing completeness/coverage**: the % of test cases that have been run and the % of code that has been tested.
 2. **reliability**: based on trends in **testing error rate** when compared to previous projects.

EVALUATE TESTS: TESTING ERROR RATE



👉 Past testing history can be used to plot **expected failure rate**.
We compare this with **actual failure rate for this project**.

TESTING UNDER PRESSURE

All tests are important, but if you need to restrict testing ...

- Testing a **system's capabilities (functionality)** is **more important than testing its components**.
 - It is important to identify things that will prevent users from using the system (e.g., doing their work).
- Testing **old capabilities** is **more important than testing new capabilities**.
 - It is important that previous functions keep working!
- Testing **typical situations** is **more important than testing boundary cases**.
 - It is important that the system work under normal usage patterns.

TESTING: SUMMARY

Effective testing requires:

- **Good planning.**
 - Know for what you are trying to test.
- **Use of the right test in the right situation.**
 - Choose the appropriate test type (white box, black box, regression).
- **Early and frequent testing**
 - Catch bugs early, before they have a chance to hide.
- **Systematic and thorough tests!**
 - Have a testing strategy (e.g., unit, integration, system, acceptance).
- **Clear criterion for stopping.**
 - Decide beforehand how much testing will be enough.