Week 9

CS2212

Introduction to Software Engineering

Software Testing
Part 1: Component Testing

Ask Questions Live
cs1.ca/ask

Verification and Validation

Software testing is one element of Verification and Validation (V&V)

- **Verification:** refers to the set of tasks that ensure that software correctly implements a specific function.

"Are we building the product right?"

- **Validation:** refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements.

"Are we building the right product?"

Verification and Validation

V&V includes a wide verity of Software Quality Assurance (SQA) activities including:

Usability Testing

Simulation

Technical Reviews

Database Review

Development Testing

Algorithm Analysis

Quality and Configuration Audits

Installation Testing

Performance Monitoring

Feasibility Study

Acceptance Testing

Documentation Review

Qualification Testing

Who Does the Testing?

- **Software developers** are always **responsible for testing individual program components** and ensuring that each performs its designed function or behaviour.
- Only **after the software architecture is complete** does an **Independent Test Group (ITG)** become involved.
- The role of an **Independent Test Group (ITG)** is to remove the **conflict of interest** associated with letting the builder test the thing that has been built.
- **Developers** and ITG work closely throughout a software project to ensure that thorough tests will be conducted.

Types of Testing

- **Unit Testing:** individual units of source code (methods, classes, modules, components, etc.) are tested to determine whether they are fit for use.
- **Integration Testing:** individual software modules (collections of classes, components, etc.) are combined and tested as a group.
- **Validation Testing:** validates the requirements established as part of requirements modeling against the software that has been constructed.
- **System Testing:** software and other system elements are tested as a whole all together.

Software Testing Strategy

- Testing begins at the component level (**unit testing**) and works "outward" toward the integration of the entire system.
- **Integration testing** is done when components are combined and tested.

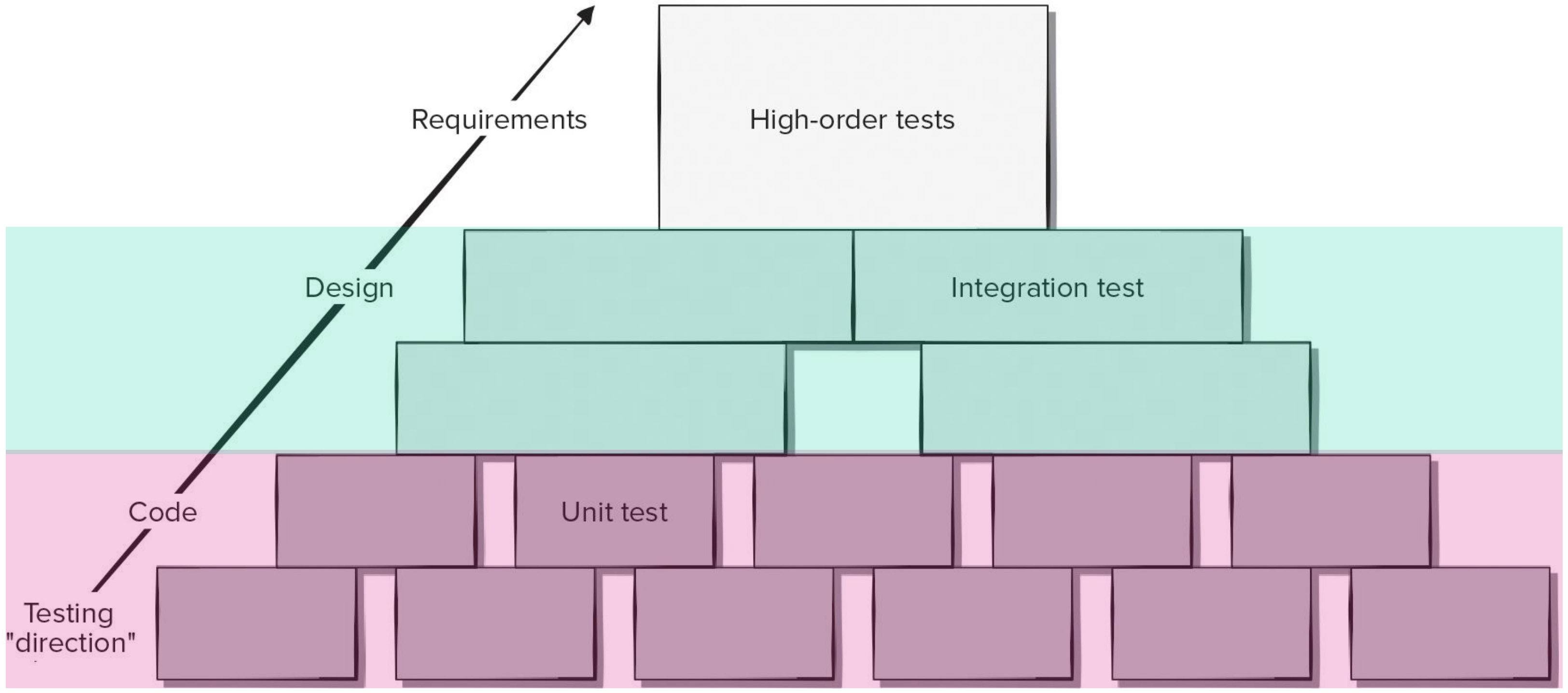


Fig. 19.2 from textbook.

Software Testing Strategy

- After software is integrated, **high-order tests** such as **validation** and **system testing** are conducted.

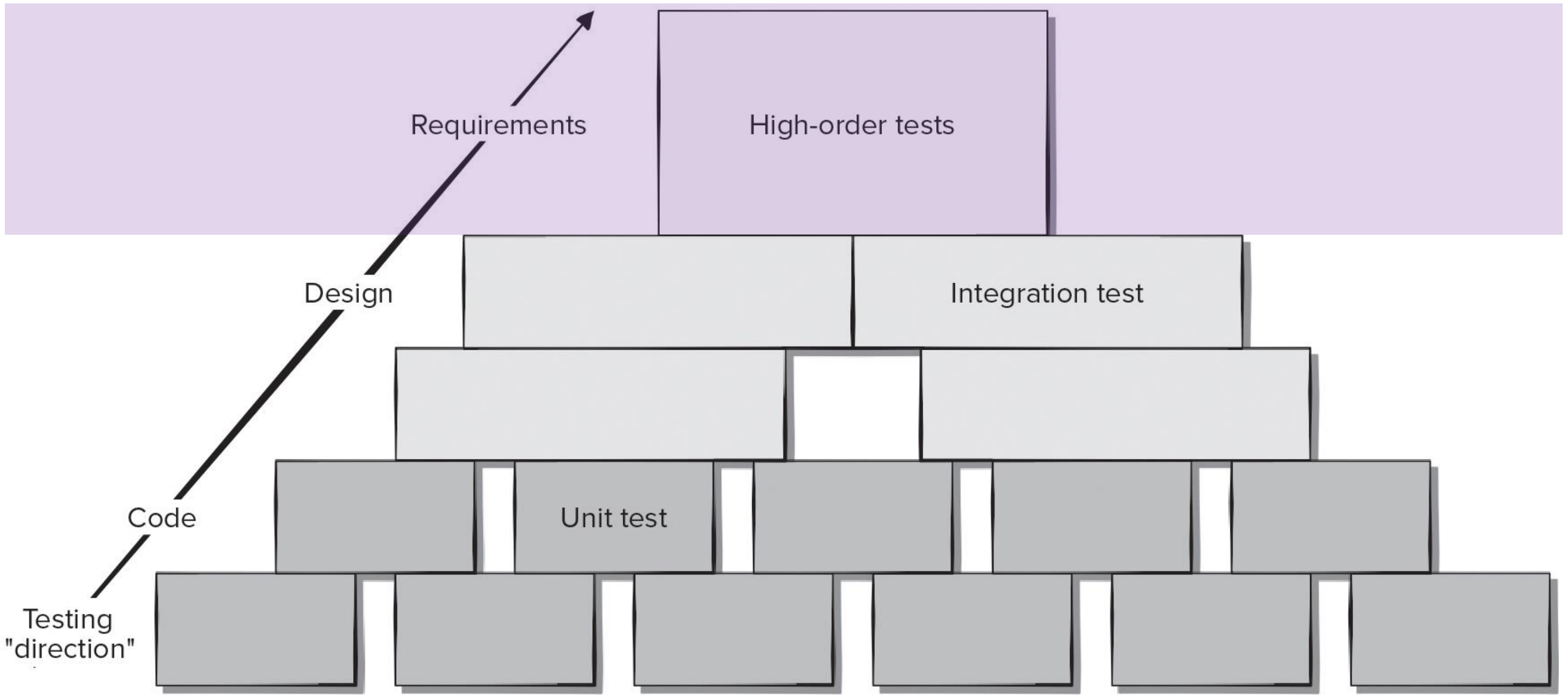


Fig. 19.2 from textbook.

Software Testing Strategy

Test Planning

- Can not wait until system is fully constructed to start testing, **need a plan**.
- Your **testing strategy** will only succeed if you:

1. Specify product requirements in a quantifiable manner.
2. State testing objectives explicitly.
3. Understand your end users and develop a profile for each user category.
4. Develop a testing plan that emphasizes *"rapid cycle testing"*.
5. Build *"robust"* software that is designed to test itself.
6. Use effective technical reviews as a filter prior to testing.
7. Conduct technical reviews on test strategy and cases themselves.
8. Develop a continuous improvement approach for the testing process.

Software Testing Strategy

Recordkeeping

- **Test cases** can be recorded in online collaborate and knowledge management systems such as **Confluence**.
- **Test case** documentation should contain:
  - **Brief description** of the **test case**.
  - A pointer to the **requirement** being tested.
  - **Expected output** from the **test case** data or the **criteria for success**.
  - Indicate whether the test was **passed or failed**.
  - **Dates** the **test case** was run.
  - Room for **comments about why a test may have failed** (aids in debugging).

Software Testing Strategy

Recordkeeping

- Typical test case format:

Test Case Name:	*This field uniquely identifies a test case.*
Test Case Description:	*This field describes the test case objective.*
Test Steps:	*In this field, the exact steps are mentioned for performing the test case.*
Pre-Requisites:	*This field specifies the conditions or steps that must be followed before the test steps executions.*
Expected Results:	*Expected output from the test case data or the criteria for success.*
Test Category:	*The type or category of the test (e.g. unit test, integration test, validation test, system test, etc.).*
Requirement:	*Reference to the requirement being tested/evaluated.*
Automation:	*Whether this test case is automated (e.g. using JUint) or not (e.g. manually run by a human).*
Date Run:	*The date/time this test was last run.*
Pass/Fail:	*If this test was passed or failed when last run.*
Test Results:	*The output or results of the test.*
Remarks:	*Remarks and comments regarding the last time the test was executed (e.g. explaining why the test failed).*

Software Testing Strategy

Recordkeeping

Project Note

For your group project, you should use a format like this to document your **manual** test cases. In particular, any integration tests, verification tests, and system tests should be detailed like this.

For **automated** tests (i.e. unit tests), only a general description of your testing strategy is needed (don't need to detail each unit test individually) as well as the JUint test files (in your repo).

Requirement:	*Reference to the requirement being tested/evaluated.*
Automation:	*Whether this test case is automated (e.g. using JUint) or not (e.g. manually run by a human).*
Date Run:	*The date/time this test was last run.*
Pass/Fail:	*If this test was passed or failed when last run.*
Test Results:	*The output or results of the test.*
Remarks:	*Remarks and comments regarding the last time the test was executed (e.g. explaining why the test failed).*

Software Testing Strategy

Criteria for Done

- How do we know we are done testing? How do you know that you’ve tested enough?
- Some common (but incorrect) views:
  - You’re never done testing; the burden simply shifts from the software engineer to the end user.
  - When our tests no longer show any errors.
  - You’re done testing when you run out of time or you run out of money.
- Better to have more rigorous criteria for sufficient testing:
  - Use **statistical techniques** to estimate the number and severity of issues remaining in the software and use these statistical models to help determine when testing is completed.

Test Case Design

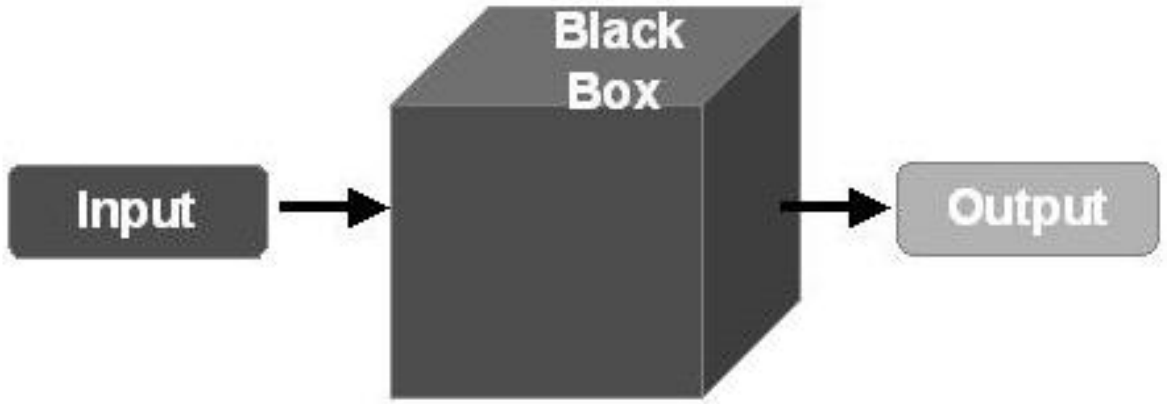
Attributes of a good test:

1. A good test has a **high probability of finding an error**.
2. A good test is **not redundant**.
3. A good test should be ***"best of breed"***.
4. A good test should be **neither too simple nor too complex**.

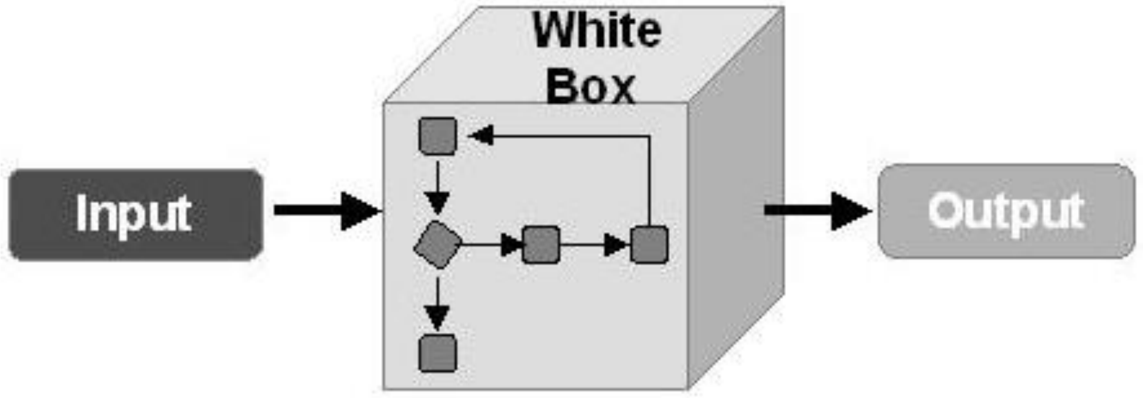
Images from softwaretestinggenius.com

Test Case Design

Two approaches to testing:



- Also known as **functional** or **behavioural** testing.
- Based on requirements from usage scenarios, rather than knowledge of internal control structures.
- Uses end user visible inputs and observable outputs.
- Focuses on functional requirements.
- Tends to be applied during later stages of testing.

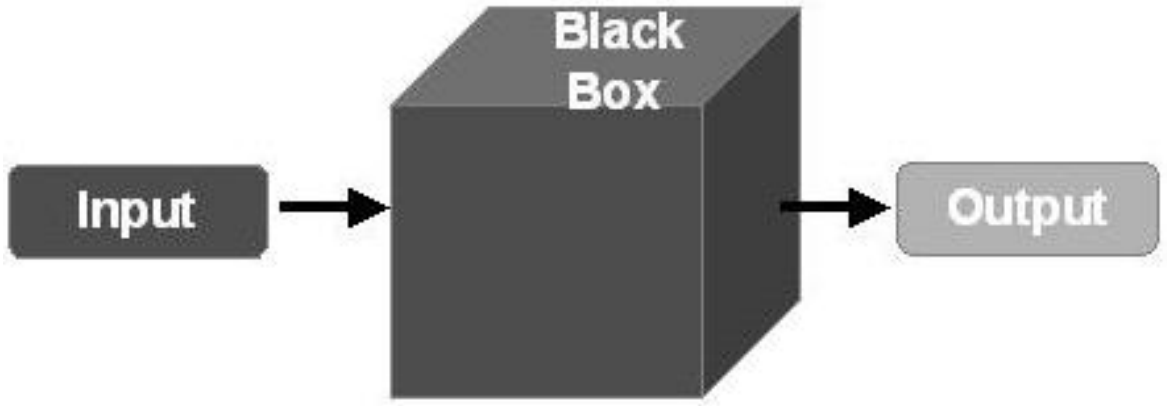


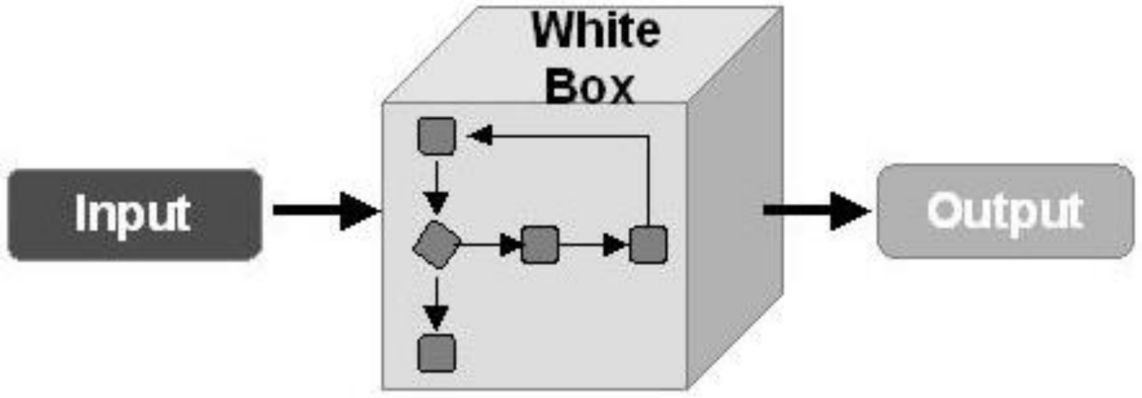
- Also known as **glass box** or **structural** testing.
- Uses implementation knowledge of control structures.
- Can only be done once source code exists.
- Aims to test all independent paths within a module at least once.
- Important data structures also the aim of testing.

Images from softwaretestinggenius.com

Test Case Design

Two approaches to testing:





Black Box and White Box testing are complimentary, meant to be used together, not alternatives to each other.

Unit Test Case Design

- Design unit test cases **before developing code** to ensure that code will pass the tests.
- Test cases **are designed to cover the following areas:**
  - The component's **interface** is tested to ensure that information properly flows into and out of the unit.
  - Local **data structures** are examined to ensure that stored data maintains its integrity.
  - **Paths through control structures** are exercised to ensure all statements are executed at least once.
  - **Boundary conditions** are tested to ensure the component operates properly at input/output boundaries.
  - All **error-handling** paths are tested and have **sensible error messages**.

Unit Test

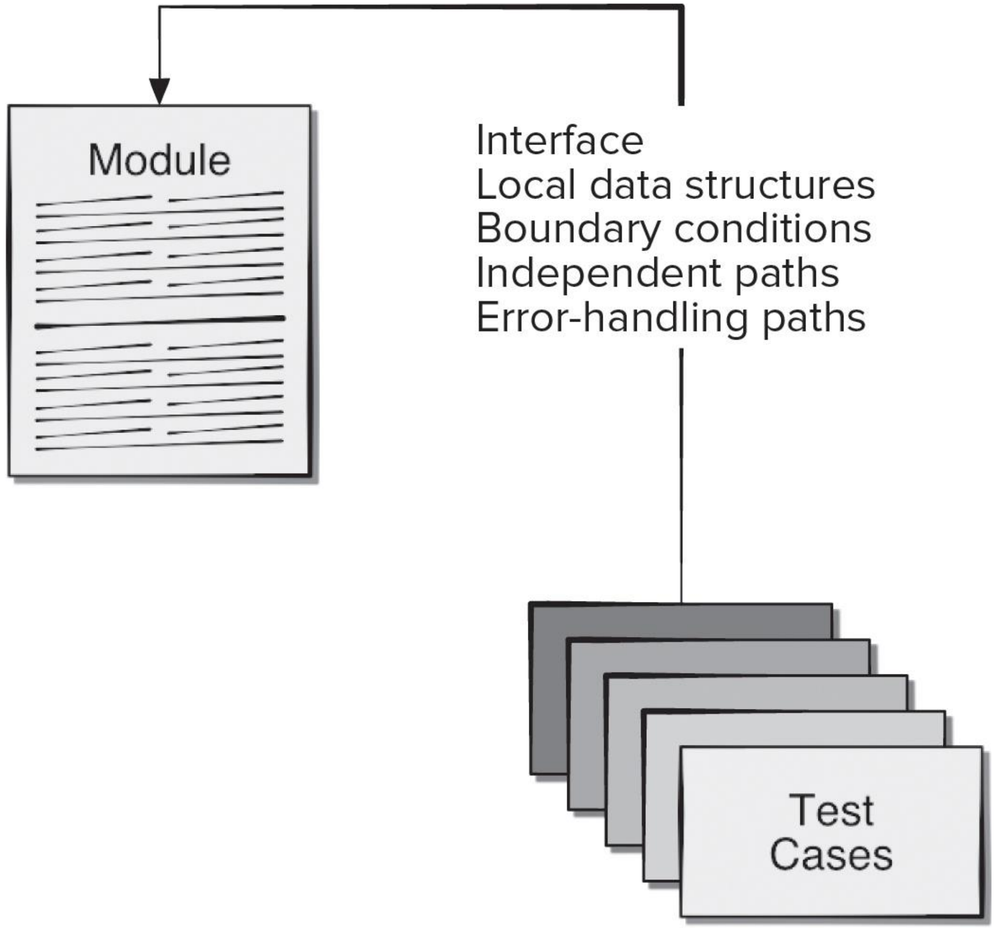


Fig. 19.4 from textbook.

Unit Test Scaffolding

- Because a component is not a stand-alone program, **driver** and/or **stub** software must often be developed for each **unit test**.
- **Driver**: a *"main program"* that accepts test-case data, passes this data to the component being tested, and records the results.
- **Stub**: serve to replace modules that are invoked by the component being tested. They replicate the other modules' interfaces, may do minimal data manipulation, record verification of entry, and return control to the component being tested.

Unit Testing Environment

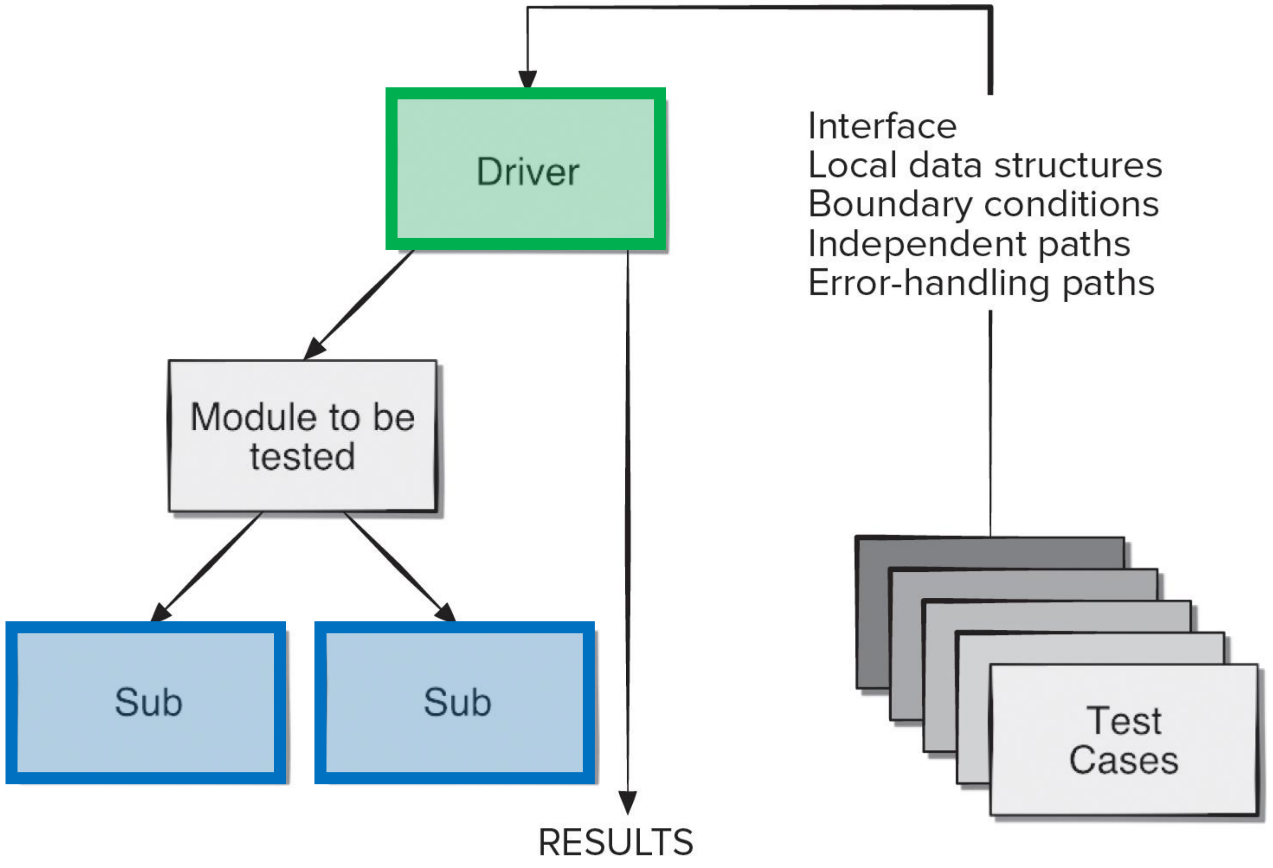


Fig. 19.3 from textbook.

White Box Unit Testing

Using white-box testing methods, you can derive test cases that:

1. Guarantee that **all independent paths** within a module have been **exercised at least once**.

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

3. **Execute all loops at their boundaries** and within their operational bounds.

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

White Box Unit Testing

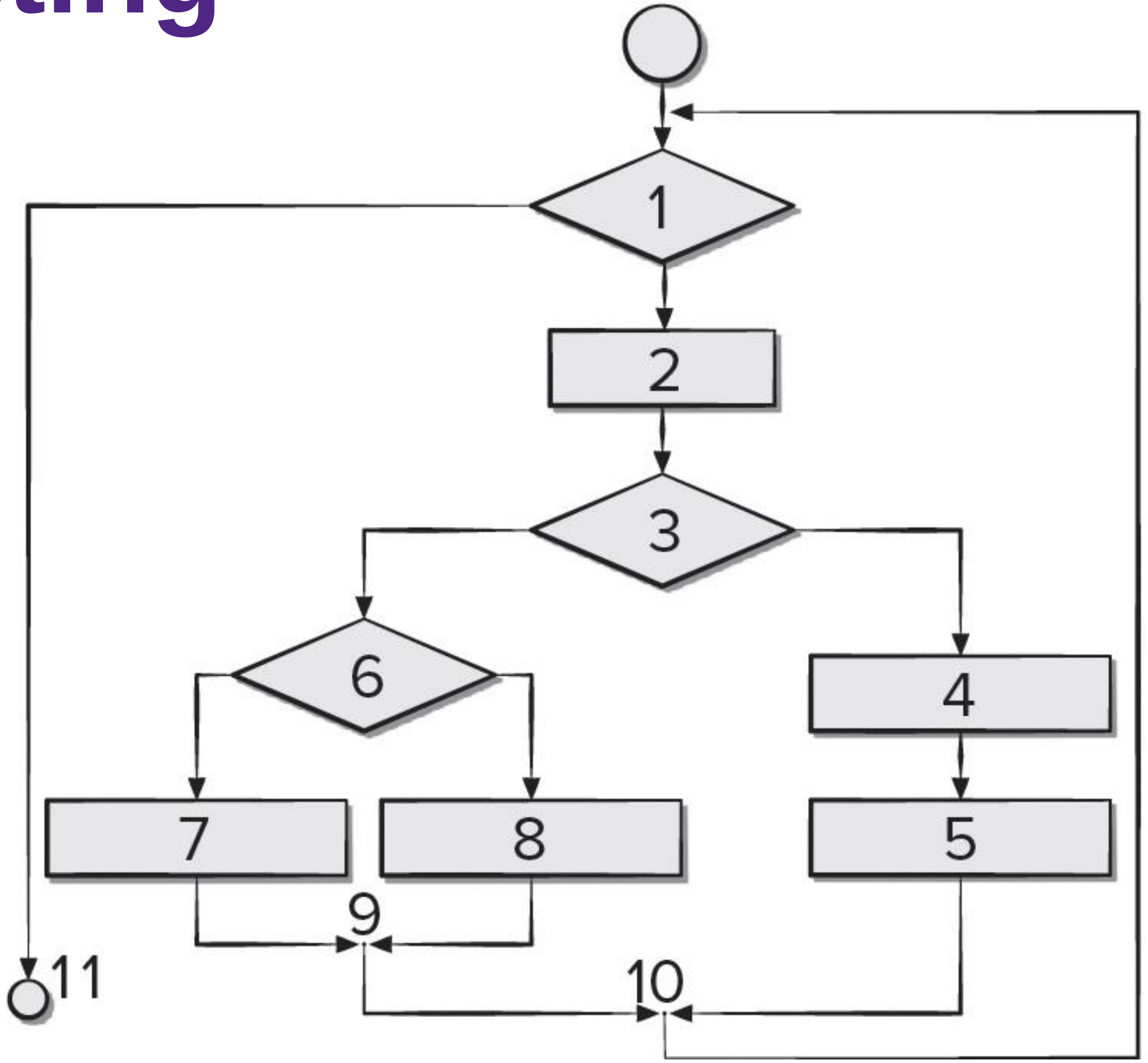
Independent Paths

- An **independent path** is any path through the program that introduces at least one new set of processing statements or a new condition.
- **Example:**
  - **Path 1:** 1-11
  - **Path 2:** 1-2-3-4-5-10-1-11
  - **Path 3:** 1-2-3-6-8-9-10-1-11
  - **Path 4:** 1-2-3-6-7-9-10-1-11

Not independent:

- 1-2-3-4-5-10-1-2-3-6-8-9-10-1-11

Flow Chart

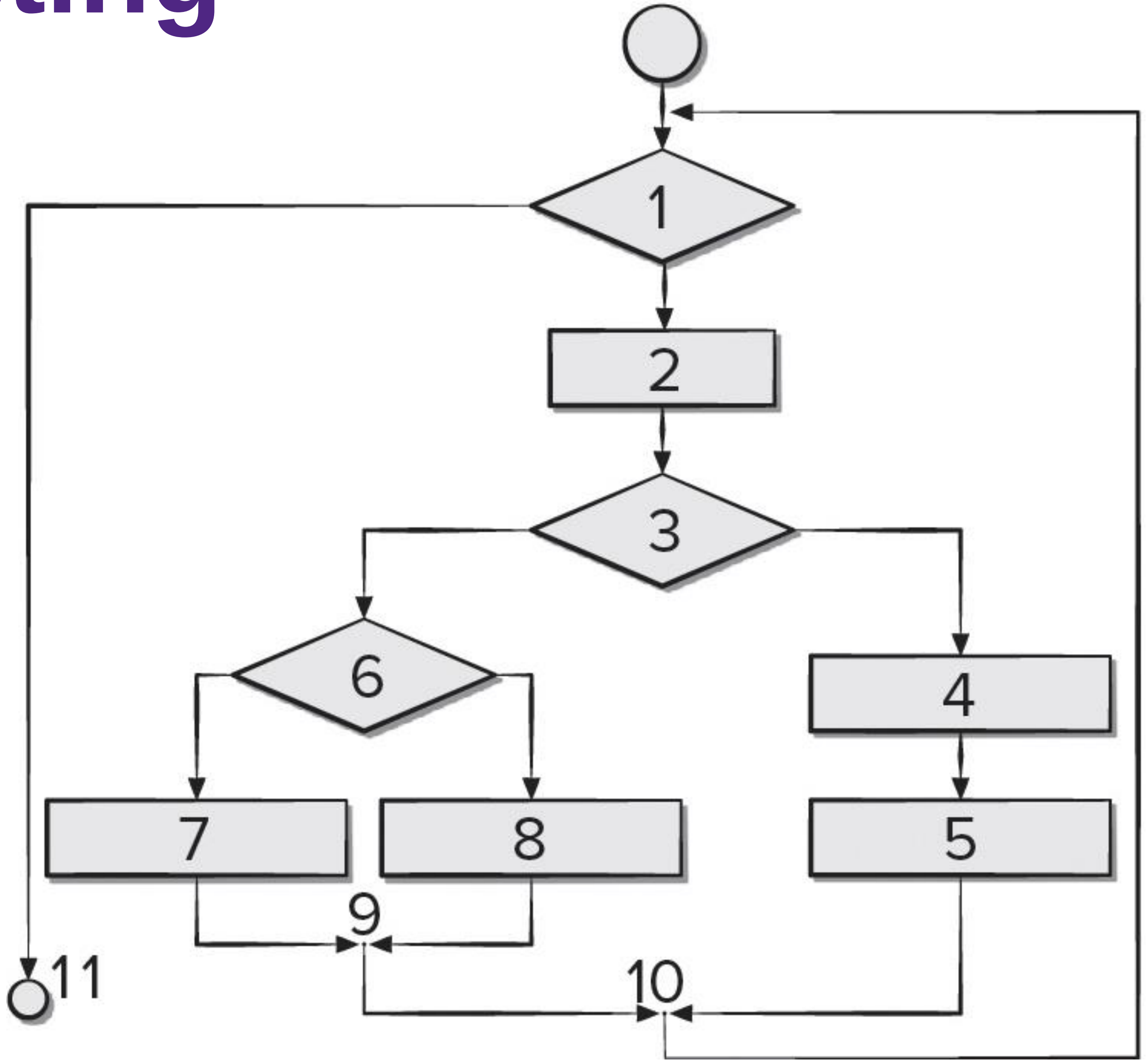


White Box Unit Testing

Independent Paths

- **Basis Set**: set of paths that will execute every program statement.
- **Basis set** is **not unique** (multiple basis sets are possible for each program).
- How to determine the upper number of paths in the **basis set**?
- Need to compute **Cyclomatic Complexity**.

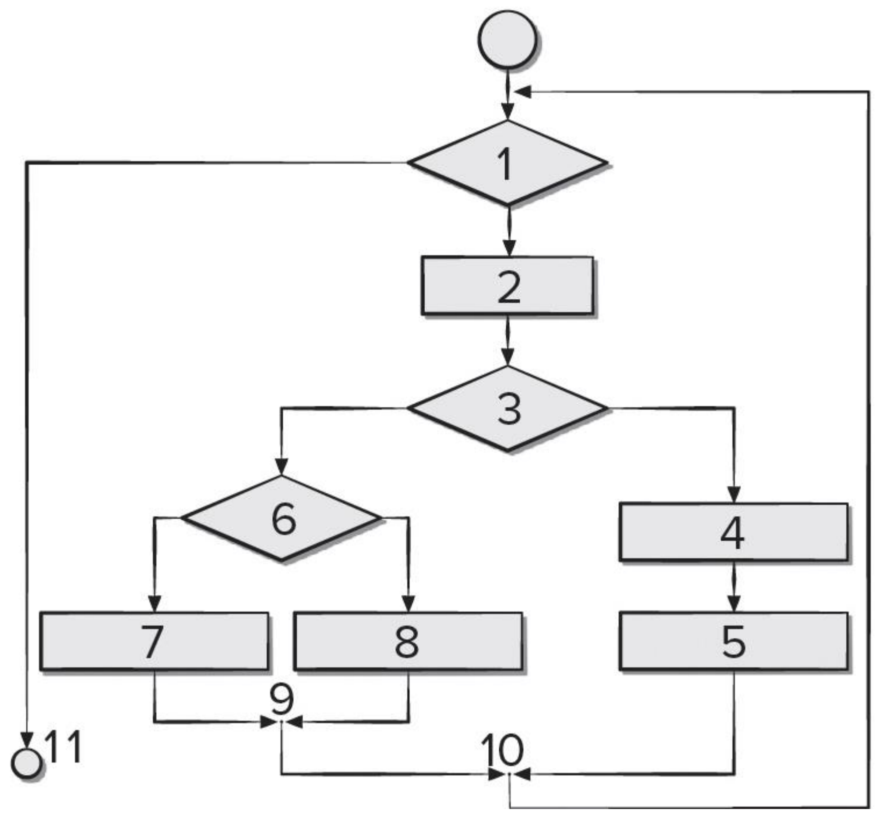
Flow Chart



White Box Unit Testing

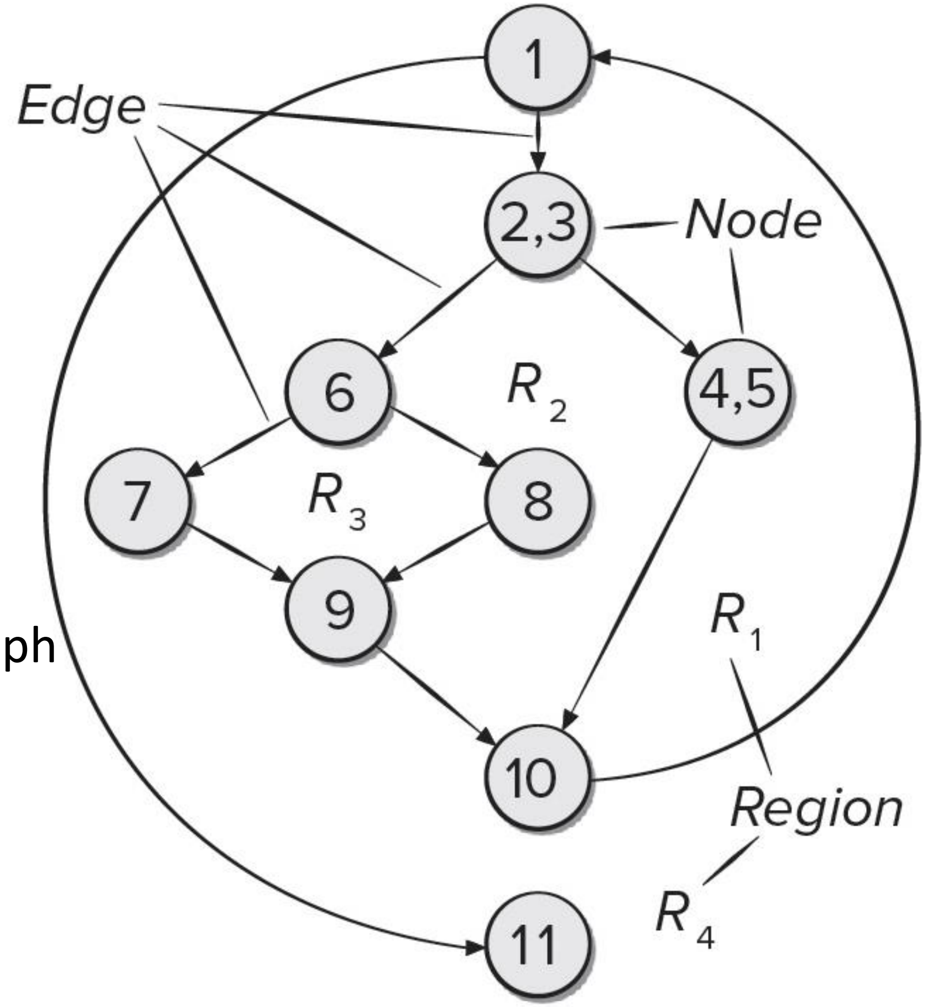
Cyclomatic Complexity

Flow Chart



Step 1: Create Flow Graph

Flow Graph

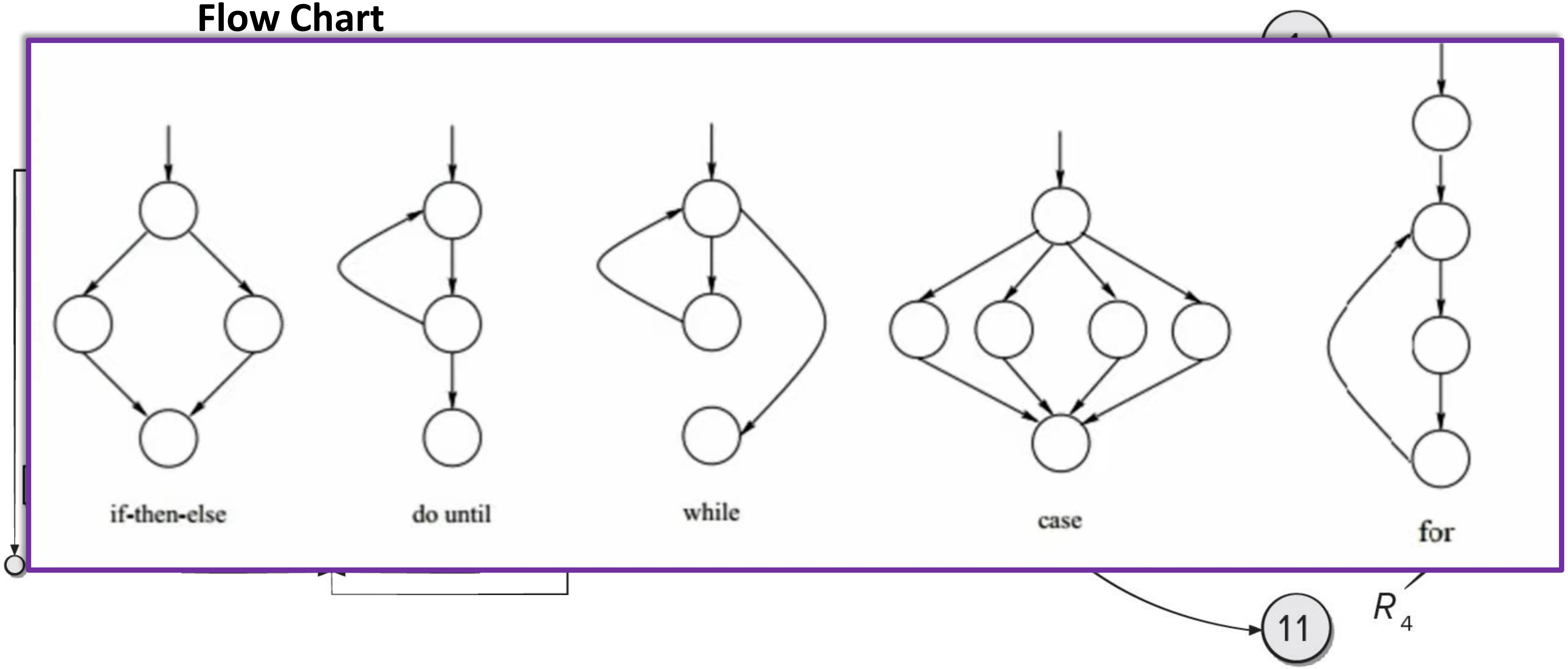


White Box Unit Testing

Cyclomatic Complexity

Flow Graph

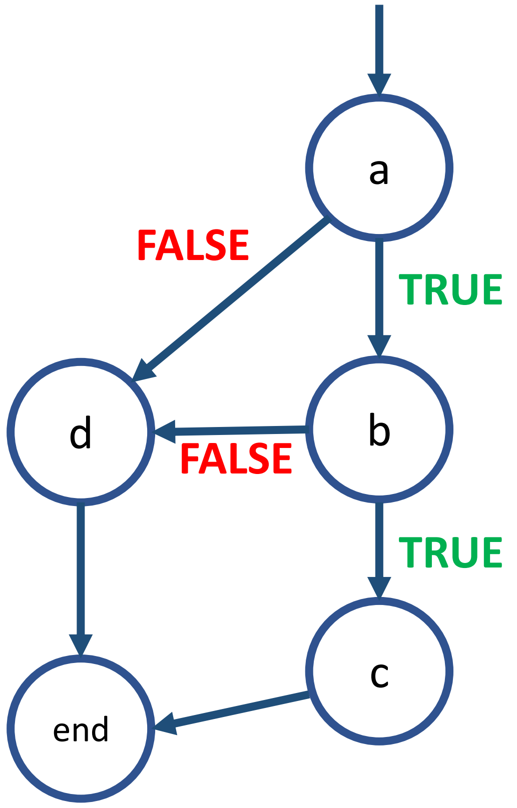
Flow Chart



White Box Unit Testing

Cyclomatic Complexity

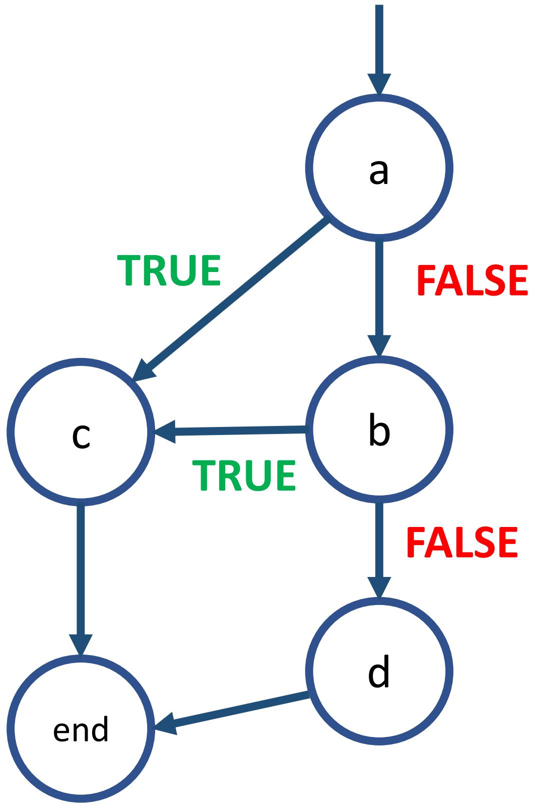
```
IF (a AND b) THEN
        c;
ELSE
        d;
```



White Box Unit Testing

Cyclomatic Complexity

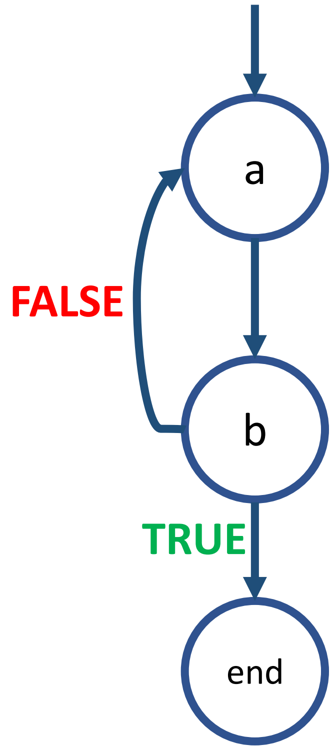
```
IF (a OR b) THEN
        c;
ELSE
        d;
```



White Box Unit Testing

Cyclomatic Complexity

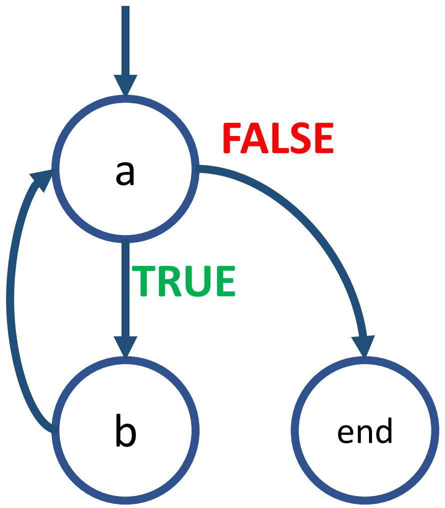
```
Do
        a;
Until b;
```



White Box Unit Testing

Cyclomatic Complexity

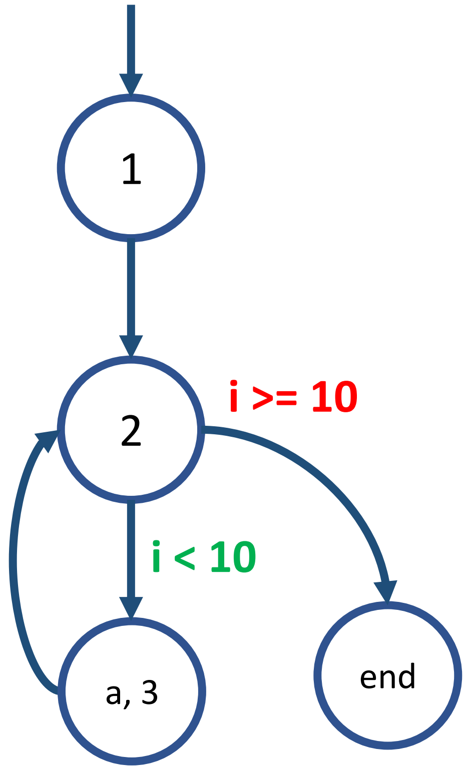
```
While a;
        b;
```



White Box Unit Testing

Cyclomatic Complexity

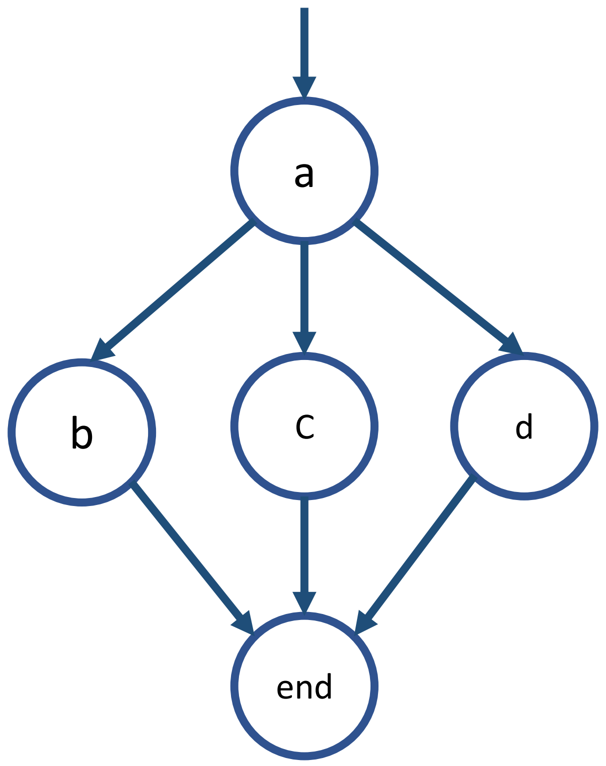
```
           1         2          3
For(i = 0; i < 10; i++)
         a;
```



White Box Unit Testing

Cyclomatic Complexity

```
Switch(a):
        case 1: b;
        case 2: c;
        default: d;
```


$$\begin{tikzcd}
 & \arrow[d] & \\
 & a \arrow[dl] \arrow[d] \arrow[dr] & \\
b \arrow[dr] & c \arrow[d] & d \arrow[dl] \\
 & \text{end} &
\end{tikzcd}$$

White Box Unit Testing

Cyclomatic Complexity

Three ways to compute cyclomatic complexity:

1. The **number of regions** of the flow graph corresponds to the cyclomatic complexity.

2. Cyclomatic complexity V(G) for a flow graph G is defined as:

   V(G) = E − N + 2

   E is the number of flow graph edges

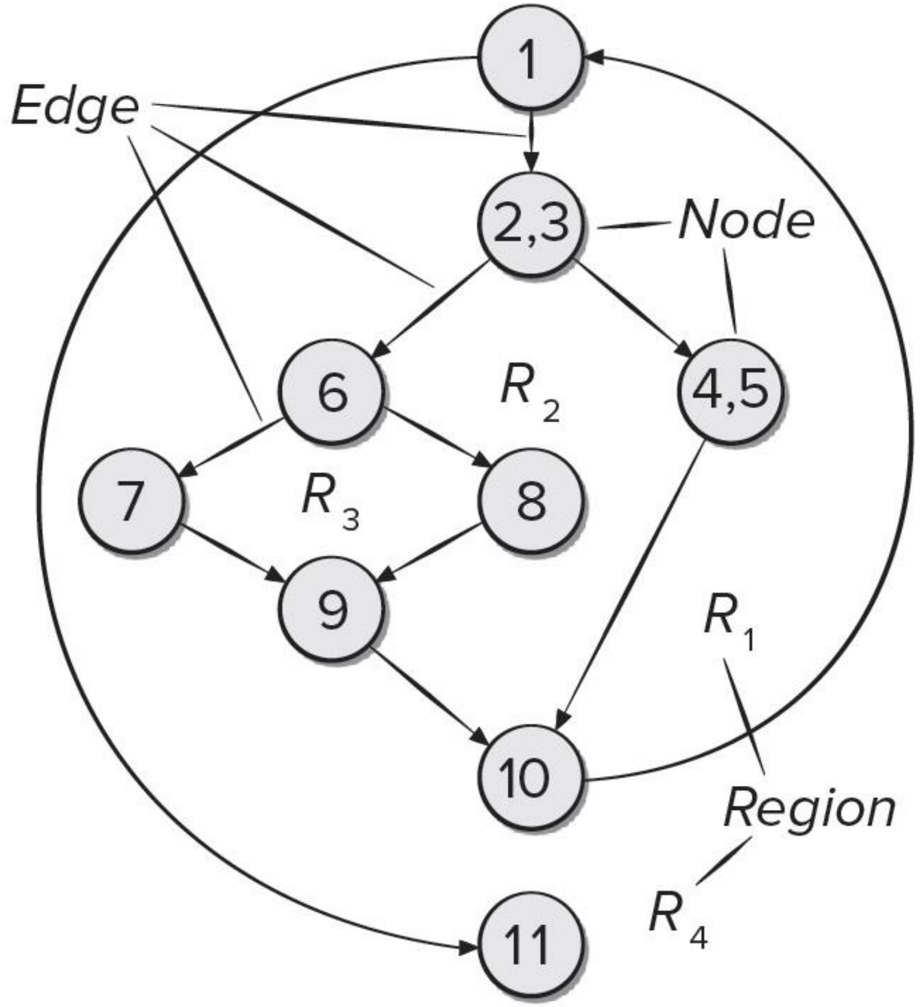
   N is the number of nodes.

3. Cyclomatic complexity V(G) for a flow graph G is also defined as:

   V(G) = P + 1

   P is number of predicate nodes contained in the flow graph G.

Flow Graph



White Box Unit Testing

Cyclomatic Complexity

Three ways to compute cyclomatic complexity:

1. The **number of regions** of the flow graph corresponds to the cyclomatic complexity. = 4
2. Cyclomatic complexity V(G) for a flow graph G is defined as:

   V(G) = E − N + 2

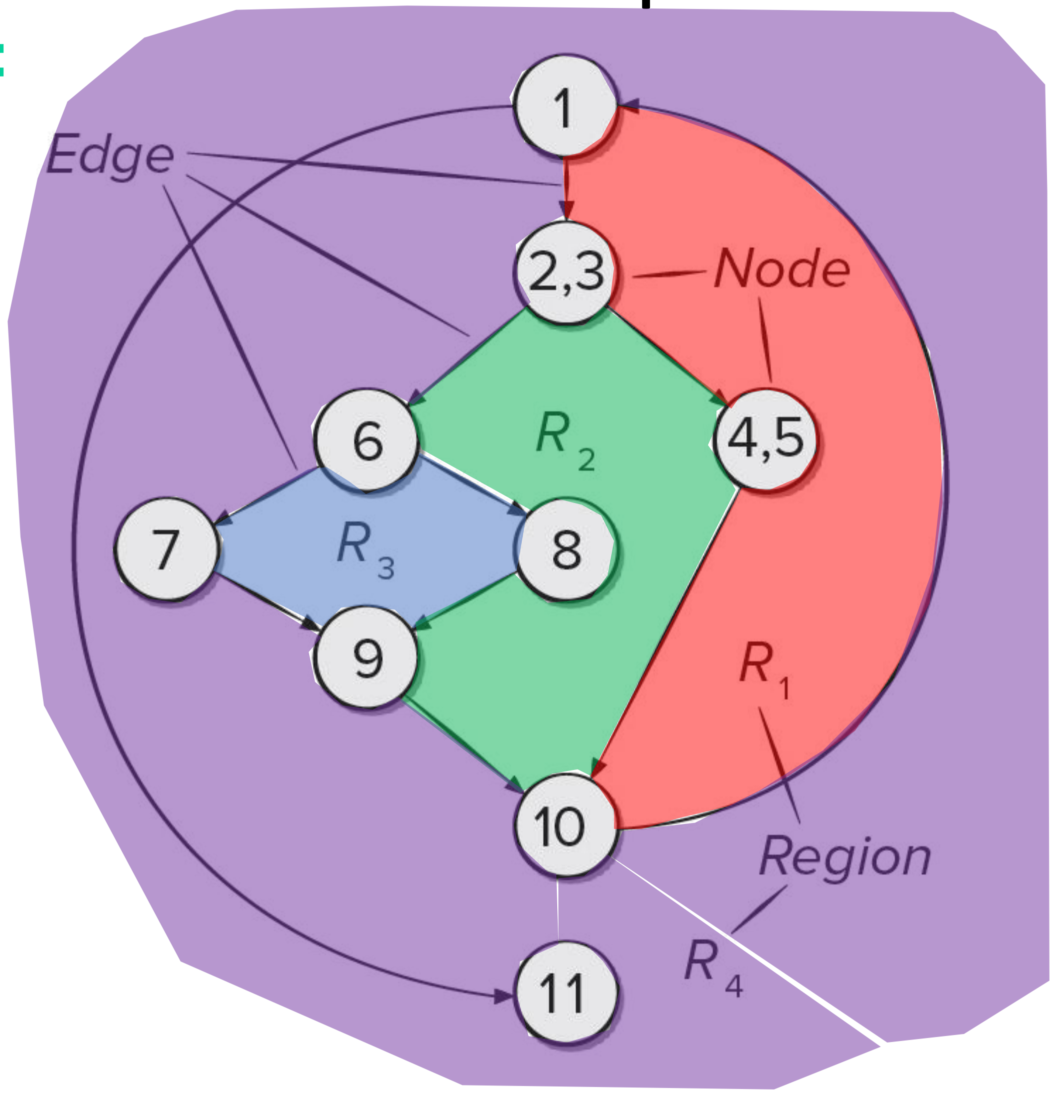
   E is the number of flow graph edges

   N is the number of nodes.
3. Cyclomatic complexity V(G) for a flow graph G is also defined as:

   V(G) = P + 1

   P is number of predicate nodes contained in the flow graph G.

Flow Graph



White Box Unit Testing

Cyclomatic Complexity

Three ways to compute cyclomatic complexity:

1. The **number of regions** of the flow graph corresponds to the cyclomatic complexity. = 4

2. Cyclomatic complexity V(G) for a flow graph G is defined as:

   V(G) = E − N + 2 **= 11 – 9 + 2 = 4**

   E is the number of flow graph edges

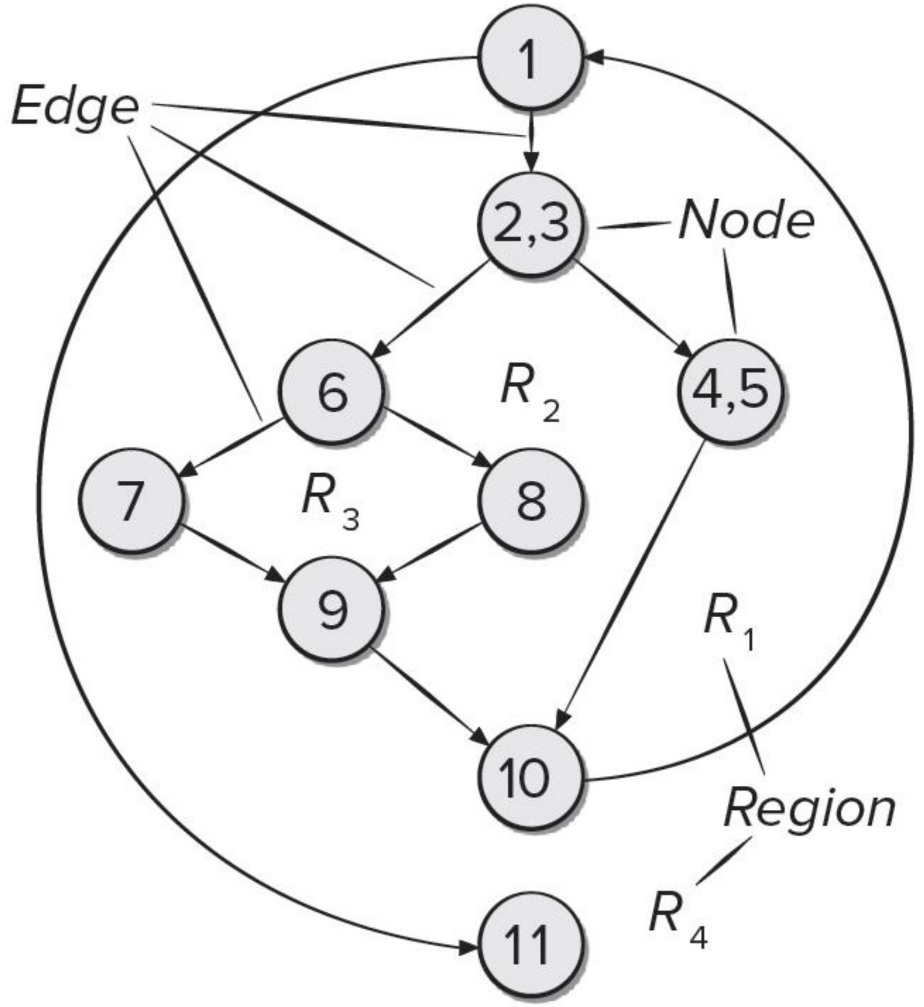
   N is the number of nodes.

3. Cyclomatic complexity V(G) for a flow graph G is also defined as:

   V(G) = P + 1

   P is number of predicate nodes contained in the flow graph G.

Flow Graph



White Box Unit Testing

Cyclomatic Complexity

Three ways to compute cyclomatic complexity:

1. The **number of regions** of the flow graph corresponds to the cyclomatic complexity. **= 4**
2. Cyclomatic complexity V(G) for a flow graph G is defined as:

   V(G) = E − N + 2 **= 11 – 9 + 2 = 4**

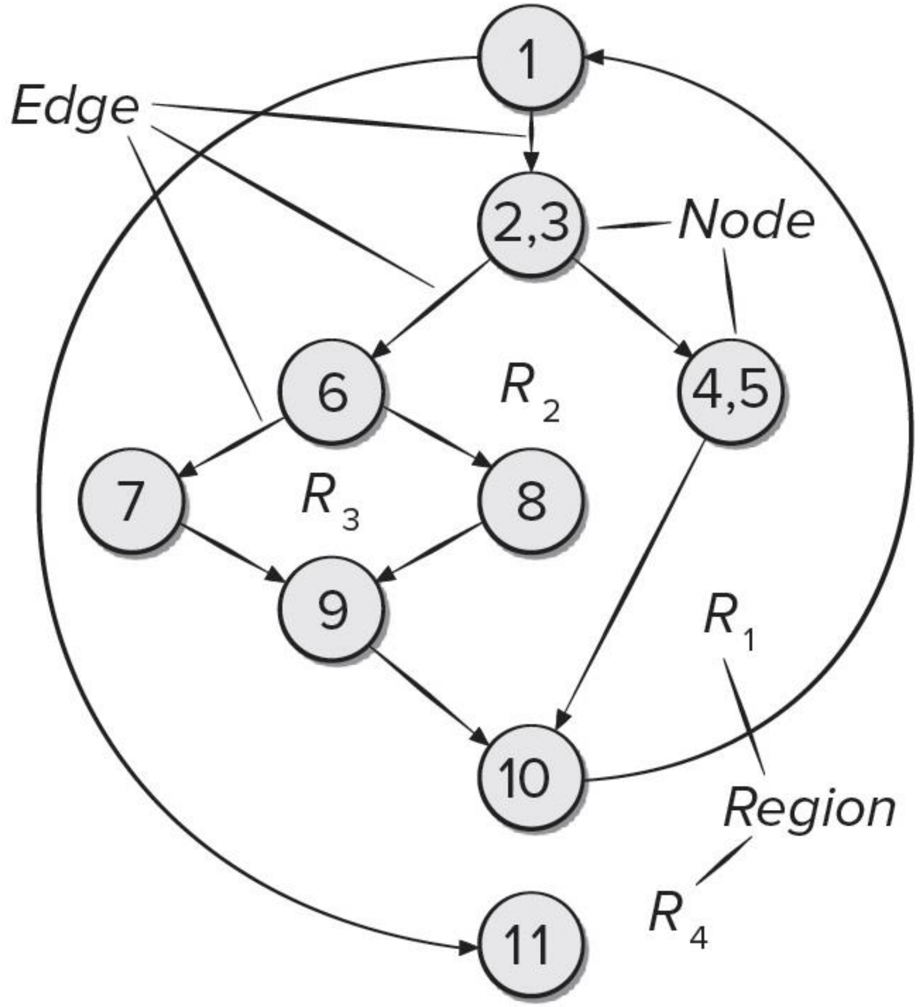
   E is the number of flow graph edges

   N is the number of nodes.
3. Cyclomatic complexity V(G) for a flow graph G is also defined as:

   V(G) = P + 1 **= 3 + 1 = 4**

   P is number of predicate nodes contained in the flow graph G.

Flow Graph



White Box Unit Testing

Cyclomatic Complexity

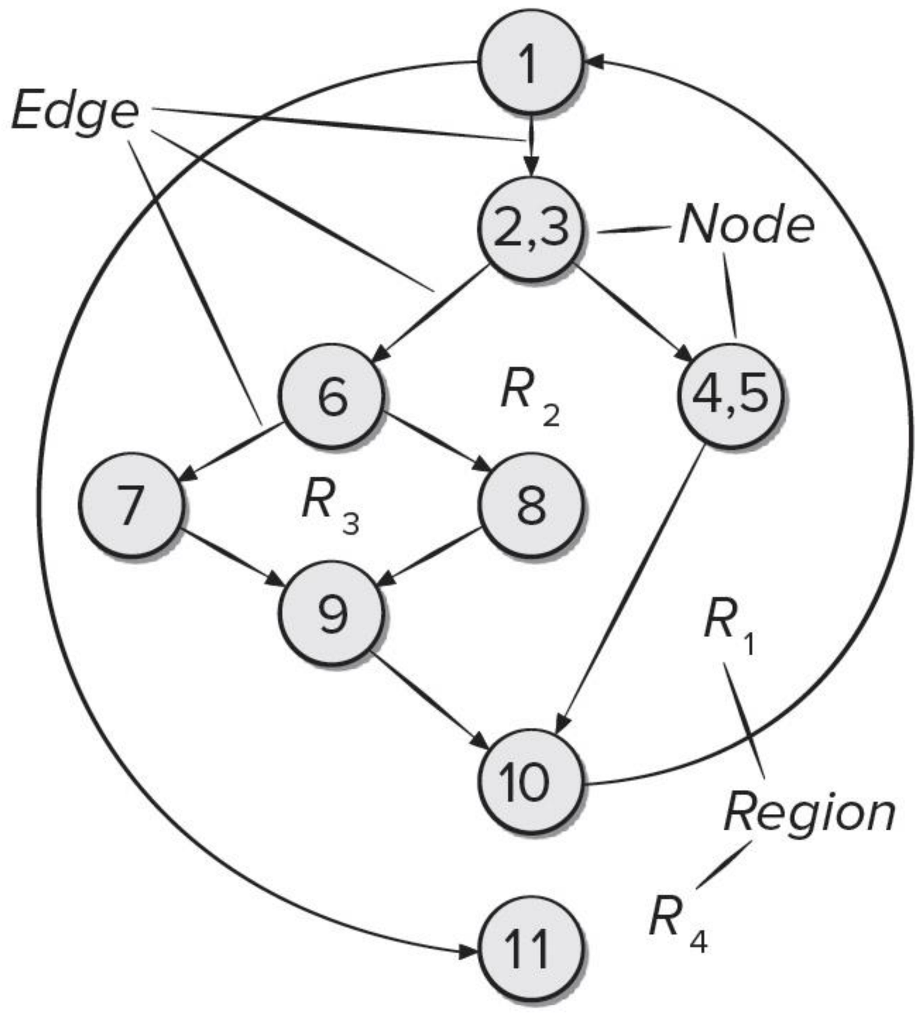
In this case the cyclomatic complexity is 4 meaning our basis set's upper bound is 4.

Possible Basis Set:

- **Path 1:** 1-11
- **Path 2:** 1-2-3-4-5-10-1-11
- **Path 3:** 1-2-3-6-8-9-10-1-11
- **Path 4:** 1-2-3-6-7-9-10-1-11

Therefore, for complete coverage of all **independent paths**, our unit tests would need to exercise these four paths.

Flow Graph

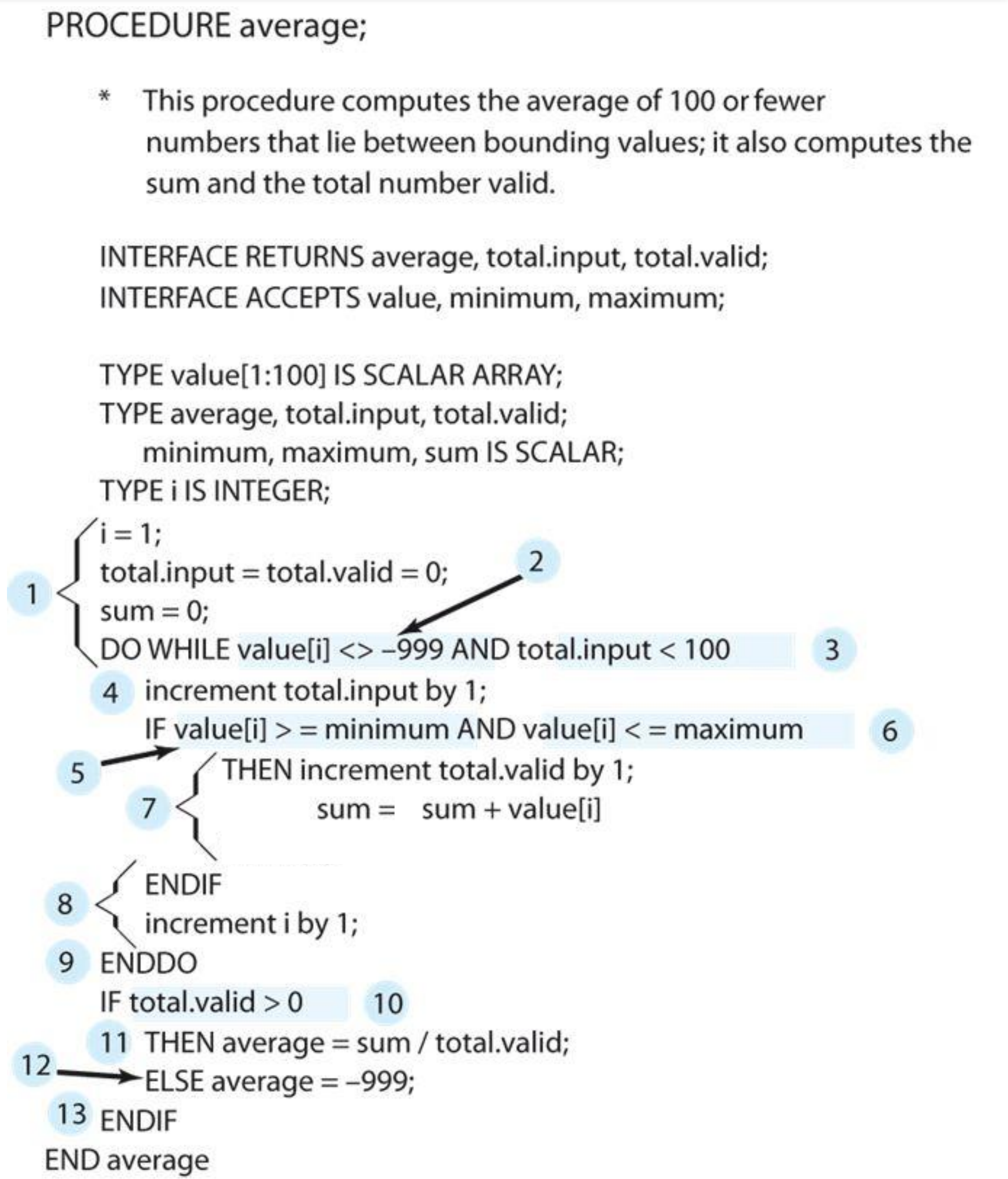


Activity: Calculate the Cyclomatic Complexity

Step 1: Based on the code to the right, create a **flow graph**.

Hint: Nodes are given for you (each number is it's own node).

Step 2: Calculate the cyclomatic complexity based on the flow graph.

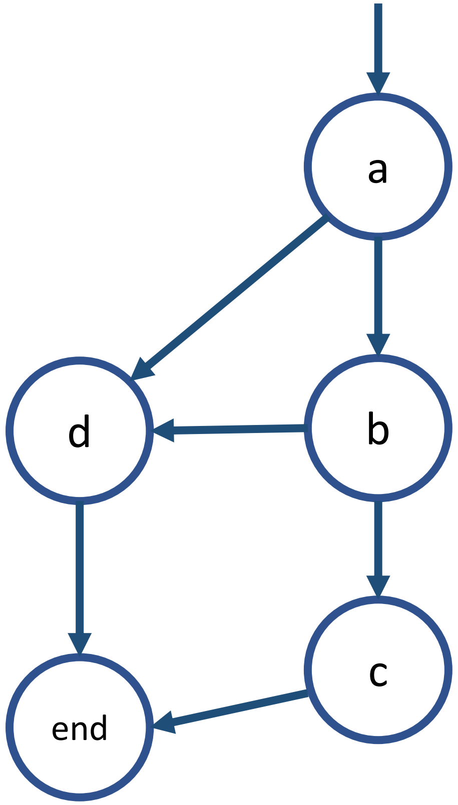


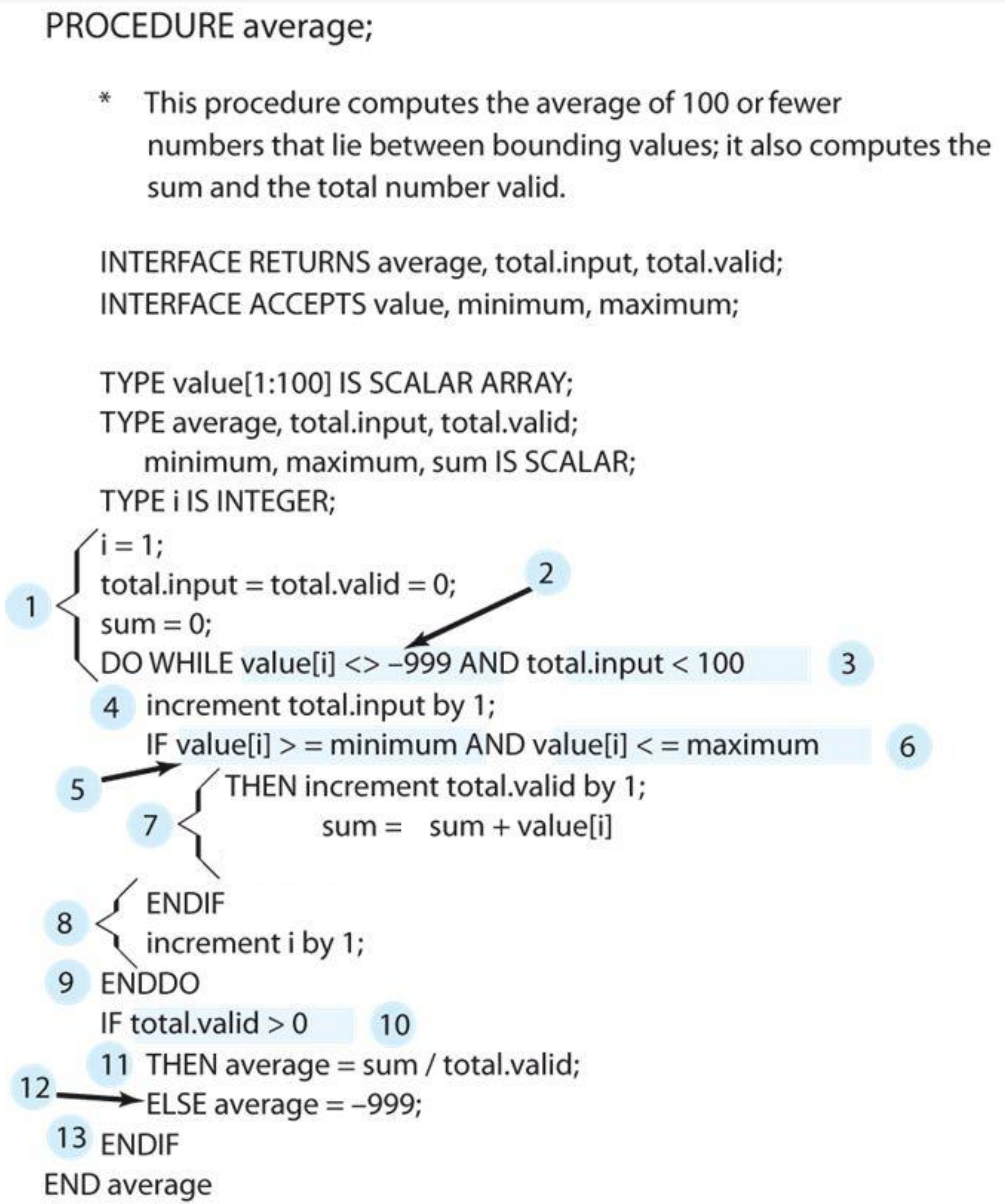
```
PROCEDURE average;

    *  This procedure computes the average of 100 or fewer
       numbers that lie between bounding values; it also computes the
       sum and the total number valid.

    INTERFACE RETURNS average, total.input, total.valid;
    INTERFACE ACCEPTS value, minimum, maximum;

    TYPE value[1:100] IS SCALAR ARRAY;
    TYPE average, total.input, total.valid;
        minimum, maximum, sum IS SCALAR;
    TYPE i IS INTEGER;
  ( i = 1;
1 { total.input = total.valid = 0;          2
  ( sum = 0;
    DO WHILE value[i] <> –999 AND total.input < 100        3
    4  increment total.input by 1;
       IF value[i] > = minimum AND value[i] < = maximum      6
5 →      ( THEN increment total.valid by 1;
      7 {        sum =   sum + value[i]
8 { ENDIF
  ( increment i by 1;
9   ENDDO
    IF total.valid > 0       10
    11 THEN average = sum / total.valid;
12 →   ELSE average = –999;
13  ENDIF
END average
```

Activity: Calculate the Cyclomatic Complexity

Hint: Flow graph for AND statements

```
IF (a AND b) THEN
    c;
ELSE
    d;
```





```
PROCEDURE average;

    *  This procedure computes the average of 100 or fewer
       numbers that lie between bounding values; it also computes the
       sum and the total number valid.

    INTERFACE RETURNS average, total.input, total.valid;
    INTERFACE ACCEPTS value, minimum, maximum;

    TYPE value[1:100] IS SCALAR ARRAY;
    TYPE average, total.input, total.valid;
        minimum, maximum, sum IS SCALAR;
    TYPE i IS INTEGER;
    i = 1;
1   total.input = total.valid = 0;          2
    sum = 0;
    DO WHILE value[i] <> –999 AND total.input < 100      3
    4  increment total.input by 1;
       IF value[i] > = minimum AND value[i] < = maximum      6
 5        THEN increment total.valid by 1;
    7             sum =    sum + value[i]
       ENDIF
 8     increment i by 1;
 9  ENDDO
    IF total.valid > 0    10
    11 THEN average = sum / total.valid;
12     ELSE average = –999;
13  ENDIF
END average
```

Basis Path Testing

Designing Test Cases

1. Using the design or code as a foundation, **draw a corresponding flow graph**.

2. Determine the **cyclomatic complexity** of the resultant flow graph.

3. Determine a **basis set** of linearly independent paths.

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

Control Structure Testing

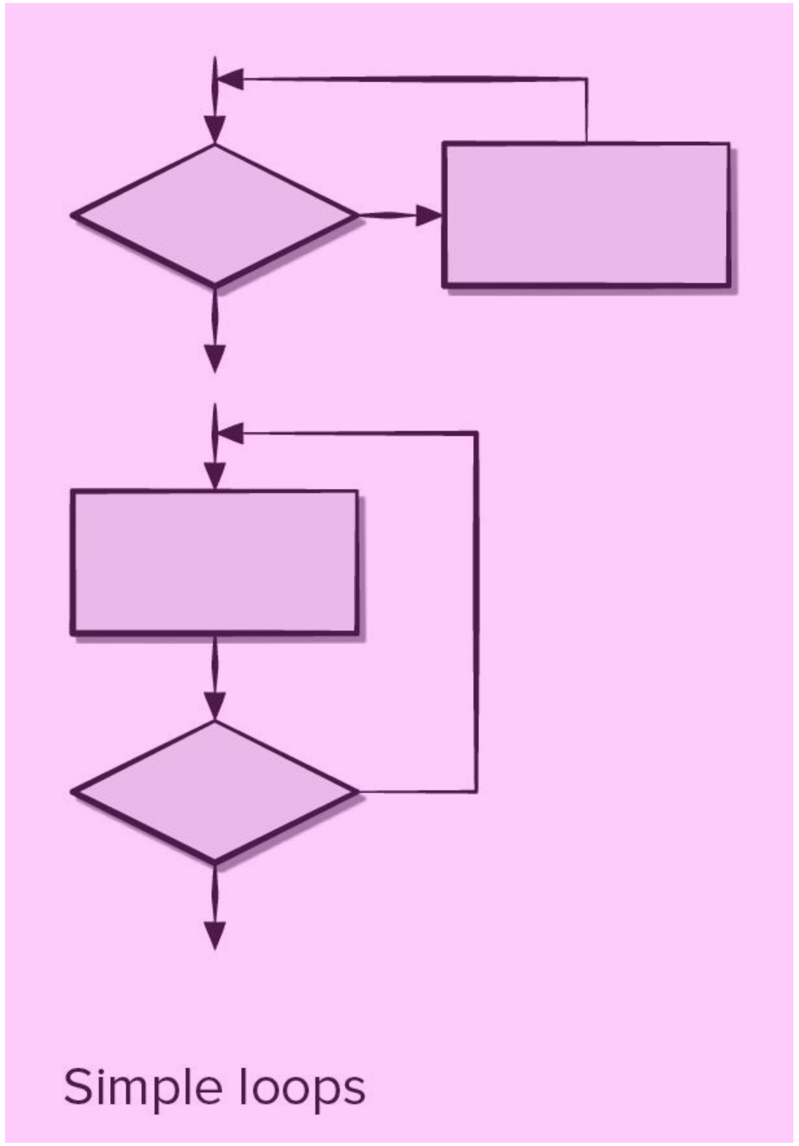
Basis path testing is simple and effective, but may not be sufficient on its own for white box testing; other methods can improve white-box testing:

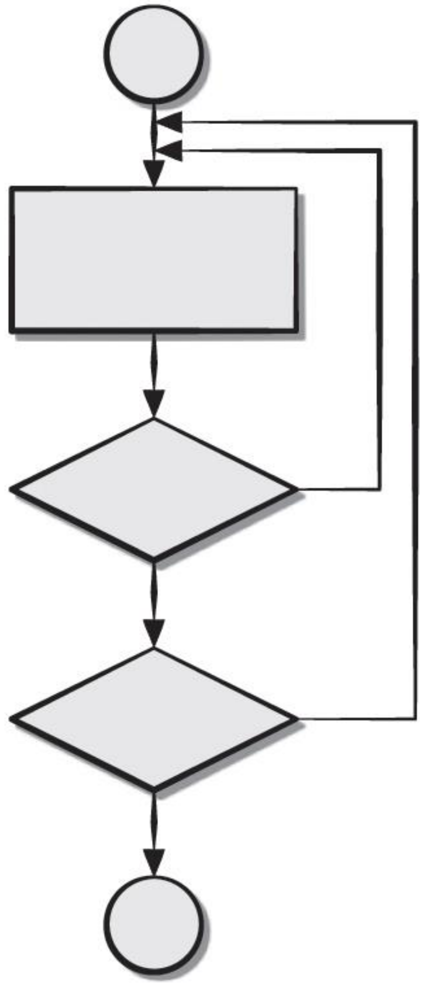
- **Condition Testing:** is a test-case design method that exercises the logical conditions contained in a program module.
- **Data Flow Testing:** selects test paths of a program according to the locations of definitions and uses of variables in the program.
- **Loop Testing:** is a white-box testing technique that focuses exclusively on the validity of loop constructs.

Loop Testing

Test cases for 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 where *m* < *n.*
5. *n* − 1, *n, n* + 1 passes through the loop.

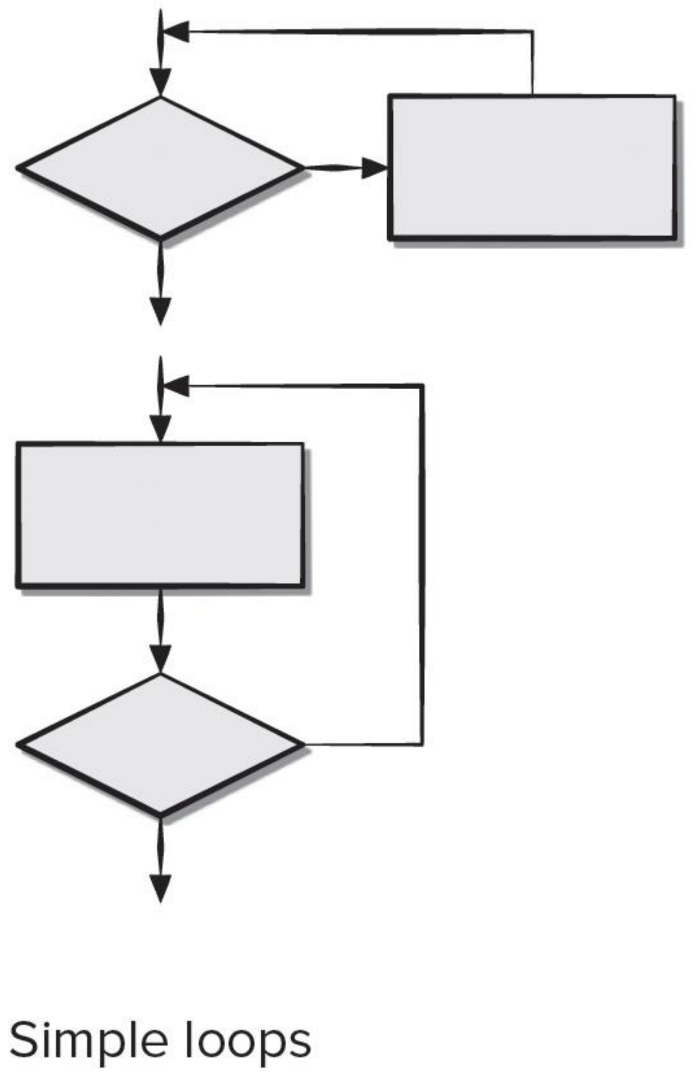

Simple loops

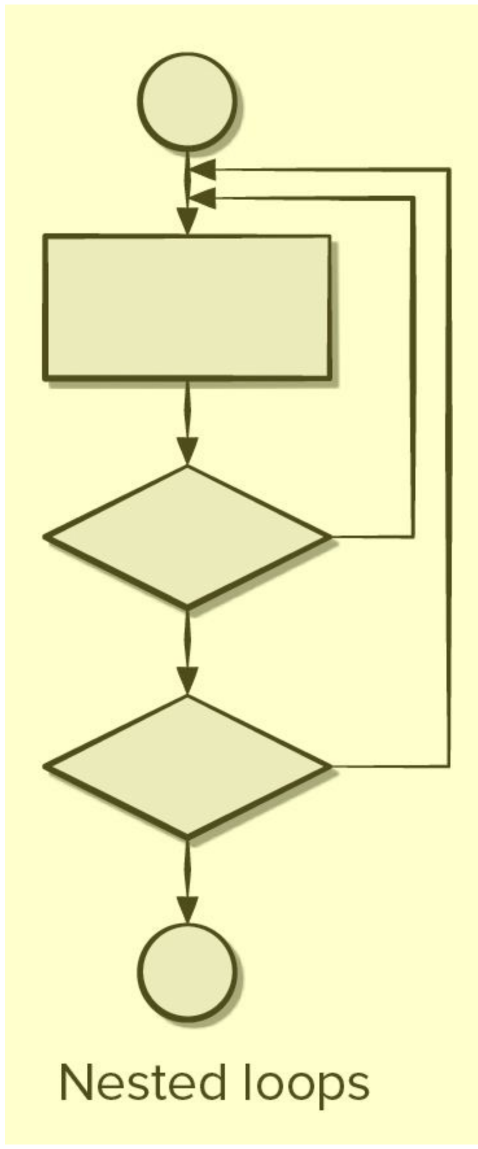

Nested loops

Loop Testing

Test cases for nested loops:

1. Start at the innermost loop. Set all other loops to minimum values.
2. Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter (for example, loop counter) values.
3. Add other tests for out-of-range or excluded values.
4. Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to “typical” values.
5. Continue until all loops have been tested.

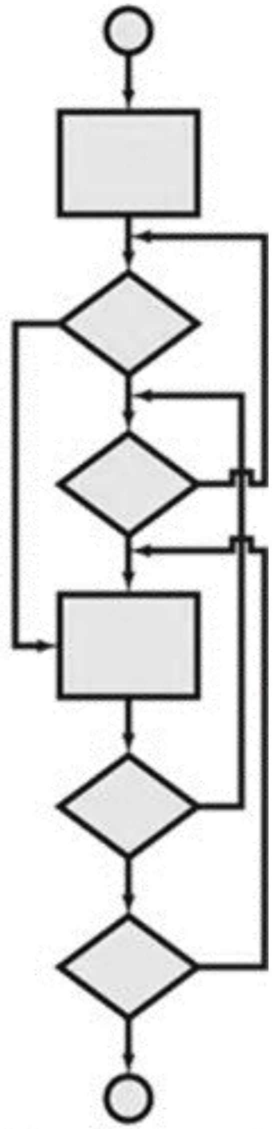




Loop Testing

What about unstructured loops?

- Likely means poor quality code was created (e.g. abuse of GOTO statements).
- Whenever possible, this class of loops should be restructured and refactored to reflect the use of more structured programming constructs.


Unstructured loops

White Box Unit Testing

- It is important to recognize **that exhaustive white-box testing is incredibly expensive** and is often infeasible in large programs; that said, this is **still a very useful and important approach to testing.**

Black Box Unit Testing

- Black-box (functional) testing attempts to find errors in the following categories:

  1. **Incorrect or missing functions**.
  2. **Interface errors.**
  3. **Errors in data structures or external database access.**
  4. **Behaviour or performance errors.**
  5. **Initialization and termination errors.**

- Black-box testing tends to be applied during later stages of testing.

Black Box Unit Testing

Common Black Box Tests:

- Interface Testing
- Equivalence Partitioning
- Boundary Value Analysis

Black Box Unit Testing

Common Black Box Tests:

- **Interface Testing**
- Equivalence Partitioning
- Boundary Value Analysis

Interface Testing

- Used to check that a program component **accepts information passed to it in the proper order** and **data types and returns information in proper order and data format**.
- Testing interfaces requires the use of **stubs** and **drivers**.

Black Box Unit Testing

Common Black Box Tests:

- Interface Testing
- **Equivalence Partitioning**
- Boundary Value Analysis

Equivalence Partitioning

- Method that **divides the input domain of a program into classes** of data from which test cases can be derived.
- **Uncovers a class of errors** that might otherwise require many test cases to be executed to correct.

Black Box Unit Testing

Common Black Box Tests:

- Interface Testing
- **Equivalence Partitioning**
- Boundary Value Analysis

Equivalence Class

Equivalence classes may be defined according to the following guidelines:

1. If an input condition specifies **a range**, one valid and two invalid equivalence classes are defined.
2. If an input condition requires a **specific value**, one valid and one invalid equivalence class are defined.
3. If an input condition specifies a **member of a set**, one valid and one invalid equivalence class are defined.
4. If an input condition is **Boolean**, one valid and one invalid class are defined.

Black Box Unit Testing

Common Black Box Tests:

- Interface Testing
- Equivalence Partitioning
- **Boundary Value Analysis**

Boundary Value Analysis (BVA)

- Errors tend of occur at the boundaries of the input domain rather than in the "middle".
- BVA selects test cases that exercise bounding values.
- Complements equivalence partitioning; select test cases at the "edges" of each class.

Black Box Unit Testing

Common Black Box Tests:

- Interface Testing
- Equivalence Partitioning
- **Boundary Value Analysis**

BVA Guidelines

1. If an input condition specifies a **range bounded by values *a* and *b***, test cases should be designed with **values *a* and *b* and just above and just below *a* and *b*.**
2. If an input condition specifies a **number of values**, test cases should be developed that **exercise the minimum and maximum values**; **values just above and below minimum and maximum** should also be tested
3. Apply guidelines 1 and 2 to output conditions; test cases should be designed to **produce the minimum and maximum outputs as well.**
4. If internal program data structures have prescribed limits, be certain to design test cases to **exercise the data structures at their boundaries.**

Comments on Object-Oriented Testing (OOT)

- Object-Oriented (OO) system can have **unique challenges** and test cases must account for the unique characteristics of OO software.
- Smallest unit of code in traditional programing is normally an operation/function.
- In OO systems, the **smallest unit is often the class** as all methods in a class may share common attributes and have side effects on each other.
- Tests should be designed to **exercise all possible states of a class** (a state diagram can be helpful for this).
- All **methods inherited from a parent class** may need to be **tested at least once for each child class**, as the child class may change their behaviour.