**🧪 White Box Testing**

**✅ Overview**

**White Box Testing**, also called **Structural Testing** or **Glass Box Testing**, is a method of software testing where the internal structure, design, and code of the program are known to the tester. This testing focuses on verifying the flow of inputs and outputs through the code, ensuring internal operations perform as expected.

**📍 Data Flow Testing**

**🔍 Definition**

**Data Flow Testing** is a white-box testing technique that:

* Focuses on the **points in a program where variables are defined and used**.
* Detects **data flow anomalies** (e.g., using a variable without defining it).
* Helps ensure variables are **properly initialized**, **used**, and **killed** (released or go out of scope).

**🎯 Purpose**

* Detect incorrect or unintended data manipulations.
* Catch logical errors like using undefined variables or overwriting variables without use.

**🧠 Why It Matters in Software Engineering**

Improper data use can lead to bugs, crashes, or incorrect output. Data flow testing adds robustness to the software by systematically tracking how data is handled across paths.

**⚠️ Define / Reference (Use) Anomalies**

**❌ Common Faults**

These errors were the focus of early data flow analysis:

1. **Defined but never used**

cpp

CopyEdit

int x = 5; // x is never used after this

1. **Used but never defined**

cpp

CopyEdit

cout << x; // x was never initialized

1. **Defined twice before use**

cpp

CopyEdit

int x = 5;

x = 10; // Previous value 5 was useless

**🔁 Static Variants of Data Flow Testing**

These terms are important in static analysis (analyzing code without running it):

* **Kill and again kill**: Variable is "killed" (released or goes out of scope) twice — suspicious.
* **Basic Blocks**: A sequence of code with no jumps or branches.
* **All-use coverage**: Covers all the uses of all variables.
* **c-use (Computation use)**: Variable used in a computation (e.g., x = y + 1).
* **p-use (Predicate use)**: Variable used in a decision/predicate (e.g., if (x > 0)).
* **All-defs**: Ensures every variable definition has been tested.
* **du-path (Definition-Use Path)**: A control flow path from a variable’s definition to its use without another redefinition in between.

**🧱 Steps to Perform Data Flow Coverage**

1. **Identify the Basic Blocks**  
   Basic block: A straight-line sequence of code without any branching.
2. **Identify All Definitions**  
   Where variables receive values.
3. **Identify All Uses**  
   Where variables are referenced:
   * **p-uses**: used in conditions
   * **c-uses**: used in calculations
4. **Draw paths** from each definition to each use that might be affected by that definition.

**🔍 Example: Analyzing Basic Blocks, Defs, and Uses**

cpp

CopyEdit

1. s = 0; // Def(s)

2. i = 1; // Def(i)

3. while (i <= n) { // p-use(i), p-use(n)

4. s += i; // c-use(s), c-use(i)

5. i++; // Def(i)

}

6. cout << s; // c-use(s)

7. cout << i; // c-use(i)

8. cout << n; // c-use(n)

* **Definitions**:
  + Line 1: s is defined.
  + Line 2 and 5: i is defined.
* **Uses**:
  + Line 3: i, n used in predicate (p-use).
  + Line 4: i, s used in computation (c-use).
  + Line 6, 7, 8: s, i, n used in computation (c-use).

**🔤 Notations and Their Meanings**

* **d**: Defined (created, initialized)
* **k**: Killed (variable goes out of scope or is no longer valid)
* **u**: Used (in any way)
  + **c**: computation use
  + **p**: predicate use

**~ (Tilde) Notations**

* **~d**: Variable doesn’t exist, then is defined — OK.
* **~u**: Variable doesn’t exist, then is used — ❌ Anomalous.
* **~k**: Variable doesn’t exist, then is killed — ❌ Anomalous.

**Ending ~ Means Nothing Happens After:**

* **d~**: Defined but never used — ⚠️ Possibly anomalous.
* **u~**: Used but not killed — ✅ Not anomalous.
* **k~**: Last action was killing — ✅ Not anomalous.

**🧮 Valid and Invalid Combinations**

| **Combination** | **Meaning** | **Acceptability** |
| --- | --- | --- |
| **dd** | Defined twice | Suspicious |
| **du** | Defined then used | ✅ Perfect |
| **dk** | Defined then killed | ⚠️ Possibly erroneous |
| **ud** | Used then defined | ✅ Acceptable |
| **uu** | Used twice | ✅ Acceptable |
| **uk** | Used then killed | ✅ Acceptable |
| **kd** | Killed then defined | ✅ Acceptable |
| **ku** | Killed then used | ❌ Major defect |
| **kk** | Killed twice | ⚠️ Likely programming error |

**🔀 Control Flow Graph + Data Flow Graph**

A **Data Flow Graph** is a directed graph where:

* **Nodes (N)** represent **basic blocks**.
* **Edges (E)** represent the **flow of control**.
* Annotations include:
  + **d**: where variables are defined
  + **u**: where they're used
  + **k**: where they're killed

**✂️ Slice-Based Testing**

**🧩 What is a Program Slice?**

**Program Slice S(V, n)**: Given a program P and variable set V, the slice at statement n includes all statements that affect the value of V at n.

**✅ Why Use Slices?**

* Isolate parts of the program relevant to a specific computation or variable.
* Useful for debugging and understanding how data propagates.

**🧪 Example Slice on Variable i**

cpp

CopyEdit

1. s = 0;

2. i = 1;

3. while (i <= n) {

4. s += i;

5. i++;

}

6. cout << s;

7. cout << i;

8. cout << n;

**Slice on i**:

* Includes lines 2, 3, 4, 5, 7 — all affect or use i.

**🧱 Applicability & Limitations**

* **When to Use**: For modules where logic is complex and can't be fully reviewed manually.
* **Tester Requirements**: Strong programming knowledge needed.
* **Downside**: Time-consuming and may be overkill for simple modules.

Let me know when you're ready to continue, and we’ll dive into the **Integration Testing** part next.

Or if you want, we can test your understanding of this section with questions or diagrams.

**You said:**

in Dataflow Testing, what are the steps and what will be the question asked and how to solve it step by step

**ChatGPT said:**

Great question! Let’s break this down clearly for **Data Flow Testing**, focusing on:

1. ✅ **Steps involved**
2. ❓ **Types of questions you might be asked**
3. 🧩 **How to solve them step by step (with example)**

**✅ Steps in Data Flow Testing**

Here’s a structured approach for applying Data Flow Testing:

**🔢 Step-by-Step:**

1. **Identify Basic Blocks**
   * Divide the program/code into **basic blocks** (a straight-line code sequence with no branches).
2. **Construct Control Flow Graph (CFG)**
   * Create a **directed graph** showing control flow between blocks.
3. **Identify Variable Definitions**
   * For each block, identify where each variable is **defined/assigned**.
4. **Identify Variable Uses**
   * Find **c-use** (computation use) and **p-use** (predicate use) of variables in each block.
5. **Determine du-paths (Definition-Use paths)**
   * For each definition, trace **all paths** to **each use** without redefinition in between.
6. **Apply Coverage Criteria**
   * Test using one or more of the following:
     + **All-defs coverage**: Each definition reaches at least one use.
     + **All-uses coverage**: All uses (both c-use and p-use) of each definition are covered.
     + **All-du-paths coverage**: Every def-use pair is tested along every path where the definition reaches the use.

**❓ What Kind of Questions Can Be Asked?**

**🔸 Theoretical:**

* Define **c-use** and **p-use**.
* What is a **du-path**?
* Explain anomalies like **d-u-k**, **k-u**, etc.

**🔹 Practical / Numerical (More likely):**

You may be given:

* A **code snippet**
* Asked to:
  1. Identify definitions and uses
  2. Construct the **Control Flow Graph**
  3. Identify du-paths
  4. Apply **All-Defs** or **All-Uses** testing

**🧩 How to Solve a Question Step-by-Step (Example)**

**✅ Given Code:**

cpp

CopyEdit

1. int x = 0;

2. int y = 2;

3. if (y > 0) {

4. x = y + 1;

5. } else {

6. x = y - 1;

7. }

8. print(x);

**🧠 Step-by-Step Solution:**

**Step 1: Identify Basic Blocks**

* **B1**: Lines 1-2
* **B2**: Line 3
* **B3**: Line 4
* **B4**: Line 6
* **B5**: Line 8

**Step 2: Construct Control Flow Graph**

markdown

CopyEdit

B1

↓

B2

/ \

B3 B4

\ /

B5

**Step 3: Identify Definitions and Uses**

| **Line** | **Variable** | **Type** |
| --- | --- | --- |
| 1 | x | def |
| 2 | y | def |
| 3 | y | p-use |
| 4 | y | c-use |
| 4 | x | def |
| 6 | y | c-use |
| 6 | x | def |
| 8 | x | c-use |

**Step 4: du-paths for x**

* **Definition at Line 1 (x = 0)**:
  + Overwritten at Line 4 or 6 before it is used, so **no valid path** to Line 8.
* **Definition at Line 4 or 6 (x = …)**:
  + **x** defined at Line 4 → used at Line 8 (path: 1 → 2 → 3 → 4 → 8)
  + **x** defined at Line 6 → used at Line 8 (path: 1 → 2 → 3 → 6 → 8)

These are the **du-paths** to test.

**Step 5: Apply All-Defs or All-Uses Coverage**

**All-Defs:**

* Line 1: x → no use (overwritten later) → ❌ not covered
* Line 4: x → used at Line 8 → ✅
* Line 6: x → used at Line 8 → ✅

**All-Uses:**

* For x: uses at Line 8
  + Ensure every definition (Line 4 & 6) is tested till Line 8 → ✅

**✅ Summary of How to Solve Data Flow Testing Question**

| **Step** | **What to Do** |
| --- | --- |
| 1️⃣ | Identify **basic blocks** |
| 2️⃣ | Draw the **Control Flow Graph** |
| 3️⃣ | Identify **definitions** and **uses** of variables |
| 4️⃣ | Find **du-paths**: from each definition to all uses |
| 5️⃣ | Apply **coverage criteria** (All-Defs, All-Uses, All-du-paths) |
| 6️⃣ | Write **test cases** that follow each valid du-path |

Would you like me to give you a **practice problem** based on this method to test your understanding?

**You said:**

yes

**ChatGPT said:**

Great! Here's a **Data Flow Testing practice problem** for you. Try to follow the steps we've discussed. After the problem, I'll guide you through solving it — but first try it on your own.

**🧪 Practice Problem**

**🔢 Code:**

cpp

CopyEdit

1. int a = 1;

2. int b = 2;

3. int c;

4. if (b > a) {

5. c = b + a;

6. } else {

7. c = b - a;

8. }

9. if (c > 0) {

10. b = c - a;

11. }

12. print(b);

**🎯 Tasks:**

1. **Identify the Basic Blocks**
2. **Draw the Control Flow Graph**
3. **Identify definitions (def) and uses (c-use/p-use) of variables**
4. **List all possible du-paths for variable b**
5. **Apply All-Defs Coverage** for variable b
6. **Suggest test inputs that satisfy All-Defs for b**

**Integration Testing**

**Objective: To ensure that code implemented the design properly**

**🔹 What It Means:**

* After individual units (functions, modules) of code are tested via **unit testing**, integration testing checks how these units work **together**.
* It ensures that **components integrate correctly** according to the design specifications and interact with each other as expected.

**Unit & Integration Testing Flow**

**🔹 Diagram Breakdown:**

Your notes show a flow like this:

kotlin

CopyEdit

Design Specification

↓

Unit Testing → Component Code → Unit Test

↓

Integration Test → Tested Components → Integrated Modules

**🔹 Explanation:**

* **Design Specification**: The blueprint outlining how components/modules should behave and interact.
* **Unit Testing**: Testing individual components (smallest units of code) for correct behavior.
* **Component Code**: The actual written code of these modules.
* **Integration Testing**: After unit tests pass, modules are **combined** and tested together.
* **Integrated Modules**: Group of tested components brought together to form a subsystem/system.

**Stubs and Drivers**

**🔹 Key Definitions:**

* **Stub**: A placeholder for a called function/module that **has not been developed yet**.
  + Example: If **Function A** calls **Function B**, and B is not ready, we use a stub to simulate B’s output.
* **Driver**: A placeholder that **calls** the module/function that is being tested.
  + Example: If you’ve completed **Function B**, but **Function A** (which calls B) isn’t ready, you use a driver to simulate a call to B.

**🔹 Visual Representation:**

mathematica

CopyEdit

Component to be tested

↑

Driver

↓

Stub

Stub

**🔹 Example:**

* Suppose:
  + Function A (calculates total marks) calls Function B (calculates marks per subject).
* **Stub** Use Case:
  + If Function B is not ready, create a **stub** that returns dummy marks (e.g., 85) to test Function A.
* **Driver** Use Case:
  + If Function B is ready but Function A isn’t, create a **driver** that simulates a call to B with dummy data.

**Boundary Conditions, Independent Paths, Interface, Test Cases, System**

These are **elements** considered during integration testing.

**🔹 Definitions:**

* **Boundary Conditions**: Test the limits (e.g., 0 students, max marks, etc.).
* **Independent Paths**: Each distinct logical path in the code.
* **Interface**: The connection point between modules.
* **Test Cases**: Defined inputs and expected outputs to validate behavior.
* **System**: The full software application composed of all integrated modules.

**Presumes previously tested units**

* Integration Testing assumes that all **individual components have passed Unit Testing**.
* The focus shifts to testing how components **interact**.

**Tests functionality "between" unit and system levels**

* Integration Testing is **intermediate**:
  + More than **Unit Testing** (individual components).
  + Less than **System Testing** (entire application).

**Emphasis shifts from “how to test” to “what to test”**

* In Unit Testing, you test **how** each function works.
* In Integration Testing, focus is on **what happens when components interact**.

**Purpose of Integration Testing**

* To **validate interaction** between modules.
* To **verify interfaces**, data flow, and communication paths.
* To detect **integration errors** early.

**Approaches to Integration Testing**

**➤ Based on Functional Decomposition:**

**1. Top-Down Integration**

* Start from the top-level module and integrate downward.
* **Stubs** are used for lower-level modules not yet implemented.

**2. Bottom-Up Integration**

* Start from the bottom modules and integrate upward.
* **Drivers** are used for higher-level modules.

**3. Sandwich Integration**

* A **hybrid** of Top-Down and Bottom-Up.
* Middle layers tested first, with both stubs and drivers minimized.

**4. Big Bang Integration**

* All components are integrated at once.
* Major drawback: If a test fails, it’s **hard to isolate the cause**.

**Top-Down Integration – Step-by-Step**

**🔹 Steps:**

1. Start with the **top-most control module**.
2. Replace lower-level modules with **stubs**.
3. Gradually integrate and test each lower-level module.
4. Move down the hierarchy.

**🔹 Visual Flow:**

scss

CopyEdit

Top Level (Tested)

↓

Second Level (Add modules and test)

↓

Bottom Level (Final integration)

**Bottom-Up Integration – Step-by-Step**

**🔹 Steps:**

1. Start testing from the **lowest modules**.
2. Use **drivers** to simulate calls from higher-level modules.
3. As more modules integrate, remove drivers.
4. Build up until top-most control module is integrated.

**Sandwich Integration**

**🔹 Explanation:**

* Combine both Top-Down and Bottom-Up approaches.
* Middle layer integrated first.
* Reduces effort in writing many stubs/drivers.
* **Challenge**: Fault isolation is more difficult.

**Big Bang Integration**

**🔹 Explanation:**

* Integrate **all** modules in one go and test the complete system.
* **Disadvantage**: When a failure occurs, it's **hard to trace** the fault because everything is integrated at once.

**Call Graph-Based Integration**

**➤ Pair-Wise Integration**

**🔹 Concept:**

* Focuses on **testing pairs of modules** that call each other.
* **No need for stubs/drivers**—real components are used.

**🔹 Advantages:**

* Saves development time.
* Real interaction paths are tested.
* Good for systems with a well-defined **call graph** (function calling relationships).

**✅ Summary of Key Integration Strategies**

| **Strategy** | **Uses Stubs?** | **Uses Drivers?** | **Fault Isolation** | **Development Effort** |
| --- | --- | --- | --- | --- |
| Top-Down | Yes | No | Easier | High |
| Bottom-Up | No | Yes | Easier | High |
| Sandwich | Some | Some | Harder | Moderate |
| Big Bang | No | No | Very Hard | Low (but risky) |
| Pair-Wise | No | No | Moderate | Low |