一、实验名称

白盒测试—逻辑覆盖

二、实验目的

本实验旨在通过对白盒逻辑判定程序（登录安全等级决策）的测试实践，掌握逻辑覆盖类白盒测试方法，并能结合控制流与复杂度分析定位缺陷：

1. 掌握语句覆盖、分支覆盖、条件覆盖、条件/判定覆盖、条件组合覆盖与路径覆盖的概念与差异；
2. 能从被测程序中抽取基本条件与复合条件，系统设计覆盖用例集；
3. 能绘制控制流图并使用三种方法计算环形复杂度，基于独立路径设计用例并验证可行性；
4. 使用 pytest 实现自动化执行，记录用例结果，并对暴露的逻辑缺陷进行分析与提出修复建议。

三、实验内容

被测对象为登录安全验证程序 `security_decision(two_factor, strong_password, trusted_device)`：

- 输入：3 个布尔值
  - `two_factor`：是否通过双因素认证
  - `strong_password`：是否使用强密码
  - `trusted_device`：是否在可信设备上登录
- 输出：安全等级字符串（`"High Security"` / `"Medium Security"` / `"Low Security"` / `"Access Denied"`）

需求规则：

1. **高安全等级**：双因素认证通过 且（使用强密码 或 在可信设备上登录）
2. **中安全等级**：未通过双因素认证 但 使用强密码 且 在可信设备上登录
3. **低安全等级**：仅使用强密码 或 仅在可信设备上登录（但不同时满足）
4. **拒绝登录**：既不使用强密码，也不在可信设备上登录

实验任务：使用逻辑覆盖与路径覆盖等白盒方法设计用例，并用自动化测试发现潜在缺陷。

四、实验仪器

1. 操作系统：`Linux-6.18.2-2-cachyos-x86_64-with-glibc2.42`

2. Python：`3.13.11`
3. pytest：`9.0.2`

五、实验原理

白盒测试从程序内部结构出发，依据控制流与逻辑条件设计测试以达到一定覆盖标准。本实验重点覆盖：

- **语句覆盖**：保证每条可达语句至少执行一次；
- **分支覆盖**：保证每个判定的 True/False 分支至少执行一次；
- **条件覆盖**：保证每个基本条件都至少取到一次 True 与一次 False；
- **条件/判定覆盖**：同时满足判定结果 True/False 与各基本条件 True/False；
- **条件组合覆盖**：对多个条件的所有组合进行覆盖；
- **路径覆盖**：覆盖全部可行独立路径。

相比仅做语句/分支覆盖，条件组合覆盖与路径覆盖更适合发现"条件组合遗漏"类缺陷。

六、实验过程及实验结果

第一部分：逻辑覆盖测试用例设计

1. 条件识别

- 条件 C1：`two_factor` 为真
- 条件 C2：`strong_password` 为真
- 条件 C3：`trusted_device` 为真
- 复合条件 C4：`strong_password or trusted_device` 为真

2. 测试用例设计

本实验采用"**8 个核心组合用例 + 4 个冗余用例**"的方式：

- 核心用例 TC01–TC08：穷举 3 个布尔输入的全部 8 种组合，用于条件组合覆盖与路径覆盖；
- 冗余用例 TC09–TC12：重复关键路径，满足用例数量要求并增强缺陷复现稳定性。

测试用例如下：

用例编号	two_factor	strong_password	trusted_device	预期输出	设计意图
TC01	T	T	T	High Security	P1：高安全典型组合
TC02	T	T	F	High Security	P1：覆盖 C3=F 情况
TC03	T	F	T	High Security	P1：覆盖 C2=F 情况
TC04	T	F	F	Access Denied	P2：可疑组合（仅双因素），用于暴露缺陷
TC05	F	T	T	Medium Security	P3：中安全路径
TC06	F	T	F	Low Security	P4：仅强密码
TC07	F	F	T	Low Security	P4：仅可信设备
TC08	F	F	F	Access Denied	P5：拒绝登录
TC09	T	T	T	High Security	冗余：重复高安全路径
TC10	F	T	F	Low Security	冗余：重复低安全（仅强密码）
TC11	F	F	T	Low Security	冗余：重复低安全（仅可信设备）
TC12	F	T	T	Medium Security	冗余：重复中安全路径

覆盖结论：

- 语句覆盖：所有可达 `return` 语句均被执行；末尾 `return "Error State"` 为不可达语句；

- 分支覆盖：`if two_factor`、内层 `if strong_password or trusted_device`、`if strong_password and trusted_device`、`elif strong_password or trusted_device` 的 True/False 分支均被覆盖；
- 条件覆盖：C1、C2、C3 各自均取到 True 与 False；
- 条件组合覆盖：TC01–TC08 覆盖全部 8 种布尔组合；
- 路径覆盖：覆盖 5 条独立路径（见第二部分）。

第二部分：路径测试与环形复杂度分析

1. 控制流图描述

将程序抽象为基本块节点：

- N1：入口
- N2：判定 `if two_factor`
- N3：判定 `if strong_password or trusted_device`（two_factor=True 内部）
- N4：`return "High Security"`
- N5：`return "Low Security"`（two_factor=True 且 C4=False）
- N6：判定 `if strong_password and trusted_device`（two_factor=False）
- N7：`return "Medium Security"`
- N8：判定 `elif strong_password or trusted_device`
- N9：`return "Low Security"`（仅强密码或仅可信设备）
- N10：`return "Access Denied"`
- Exit：统一出口（所有 return 指向 Exit）

2. 环形复杂度计算

1）判定节点数法：程序中判定节点 4 个，因此

- $V(G) = 4 + 1 = 5$

2）独立路径数法：基本独立路径数应为 5（见下节），因此

- $V(G) = 5$

3）边-点法：在控制流图中增加统一出口 Exit，统计可达子图：

- 节点数 N=11（N1–N10 + Exit）
- 边数 E=14（结构边 9 条 + 5 条 return→Exit）

- 连通子图 P=1

则

- $V(G) = E - N + 2P = 14 - 11 + 2 = 5$

三种方法结果一致，验证正确。

3. 基本路径集合与用例映射

选择 5 条独立路径作为基本路径集合：

- P1：N1→N2(T)→N3(T)→N4（High Security）
  - 对应用例：TC01/TC02/TC03/TC09
- P2：N1→N2(T)→N3(F)→N5（Low Security，two_factor=True 且其它为 False）
  - 对应用例：TC04（用于暴露缺陷）
- P3：N1→N2(F)→N6(T)→N7（Medium Security）
  - 对应用例：TC05/TC12
- P4：N1→N2(F)→N6(F)→N8(T)→N9（Low Security）
  - 对应用例：TC06/TC07/TC10/TC11
- P5：N1→N2(F)→N6(F)→N8(F)→N10（Access Denied）
  - 对应用例：TC08

基本路径—用例”关系图示：

第三部分：待测程序代码与 pytest 用例

被测程序 `security_decision_program.py` 完整代码如下：

```python
# security_decision_program.py - 白盒测试程序

def security_decision(two_factor, strong_password, trusted_device):
    """
    安全等级判定程序

    输入：三个布尔值
    two_factor: 是否通过双因素认证
    strong_password: 是否使用强密码
    trusted_device: 是否在可信设备上登录

    输出：安全等级字符串（"High Security", "Medium Security", "Low Security", "Access Denied"）
    """

    # 检查双因素认证
    if two_factor:
        if strong_password or trusted_device:
            return "High Security"
        else:
            return "Low Security"  # 疑似缺陷：有双因素但无其他验证应该返回什么？

    # 无双因素认证的情况
    if strong_password and trusted_device:
        return "Medium Security"

    elif strong_password or trusted_device:
        return "Low Security"

    else:
        return "Access Denied"

    # 逻辑上不可达的代码（用于增加复杂性）
    return "Error State"
```

pytest 用例文件 `test_security_decision_program.py` 完整代码如下：

```
"""
pytest 测试用例：用于对白盒测试示例程序 security_decision 进行逻辑覆盖分析。

设计思路：
- 使用 8 个核心用例穷举 three-factor 布尔组合，满足条件组合覆盖与路径覆盖；
- 补充 4 个冗余用例，突出可疑路径（仅 two_factor 为真）并满足“≥12 个用例”的实验要求。
"""

from __future__ import annotations

import pytest

from src.security_decision_program import security_decision

@pytest.mark.parametrize(
    "two_factor,strong_password,trusted_device,expected",
    [
        # P1: two_factor=True 且 (strong_password or trusted_device)=True → 高安全等级
        (True, True, True, "High Security"),
        (True, True, False, "High Security"),
        (True, False, True, "High Security"),
        # P2: 可疑路径——仅 two_factor=True, 其它为 False
        # 按题意“既不使用强密码，也不在可信设备上登录”应拒绝登录，
        # 因此这里期望 "Access Denied"，用于暴露程序缺陷。
        (True, False, False, "Access Denied"),
        # P3: 无双因素，但使用强密码且在可信设备上登录 → 中安全等级
        (False, True, True, "Medium Security"),
        # P4: 无双因素，且仅使用强密码或仅在可信设备上登录 → 低安全等级
        (False, True, False, "Low Security"),
        (False, False, True, "Low Security"),
        # P5: 不使用强密码，也不在可信设备上登录 → 拒绝登录
        (False, False, False, "Access Denied"),
    ],
)
def test_security_decision_core_paths(two_factor, strong_password, trusted_device,
expected):
    """核心用例集：覆盖全部布尔组合和独立路径。"""
    assert security_decision(two_factor, strong_password, trusted_device) == expected


@pytest.mark.parametrize(
    "two_factor,strong_password,trusted_device,expected",
    [
        # 再次覆盖高安全路径
        (True, True, True, "High Security"),
        # 再次覆盖仅强密码的低安全路径
        (False, True, False, "Low Security"),
        # 再次覆盖仅可信设备的低安全路径
```

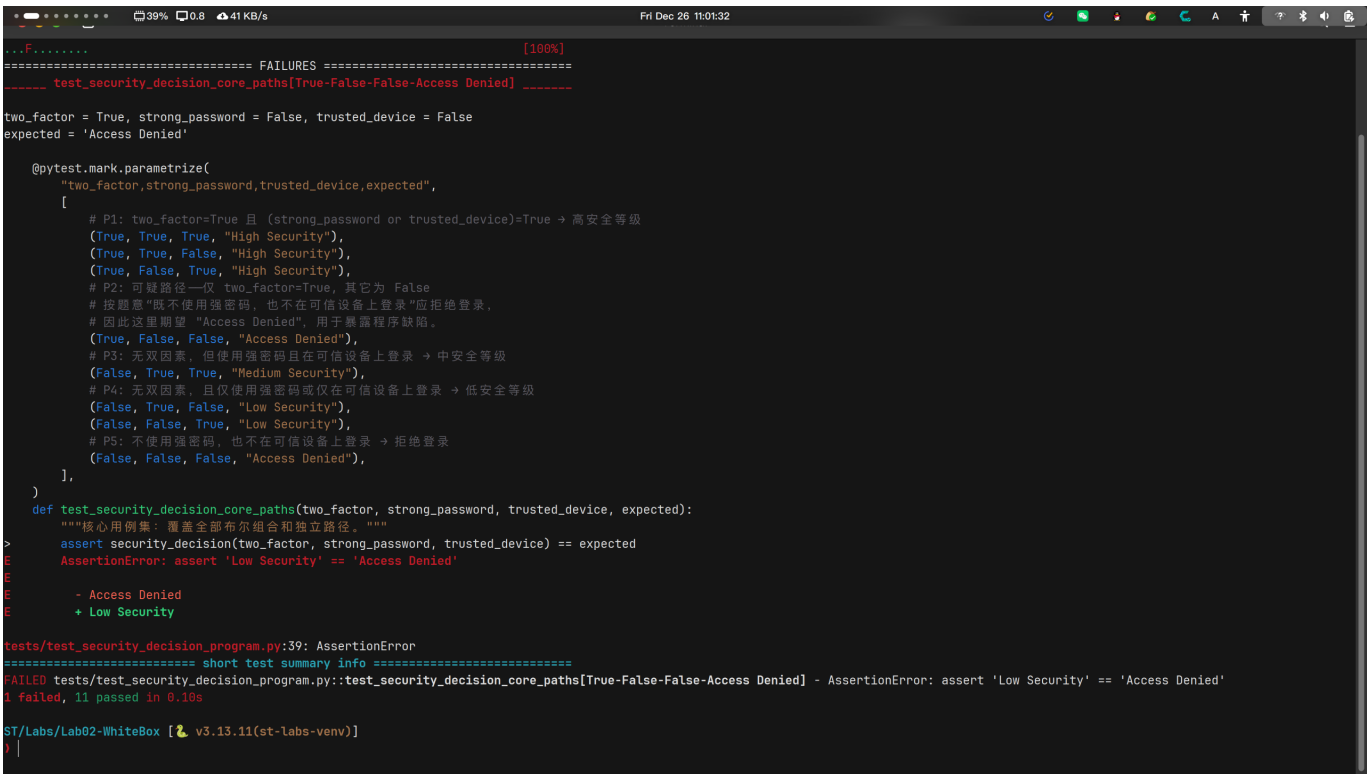
```
        (False, False, True, "Low Security"),
        # 再次覆盖中安全路径
        (False, True, True, "Medium Security"),
    ],
)
def test_security_decision_redundant_for_coverage(two_factor, strong_password,
trusted_device, expected):
    assert security_decision(two_factor, strong_password, trusted_device) == expected
```

第四部分：全量用例执行与原始结果

执行命令：

```
"/tmp/st-labs-venv/bin/python" -m pytest -q "tests/test_security_decision_program.py"
```

终端输出：



第四部分：缺陷分析报告

1. 缺陷定位

缺陷用例：TC04 `(True, False, False)`。

- 题意规则④："既不使用强密码，也不在可信设备上登录"应拒绝登录；该规则没有排除 two_factor 为真的情况。
- 实际代码：当 `two_factor=True` 且 `strong_password or trusted_device` 为假时直接返回 `"Low Security"`，导致该组合被错误分类。

因此，程序存在"条件组合遗漏/规则优先级实现不一致"的逻辑缺陷。

2. 缺陷原因分析

1. 规则优先级未显式表达：代码以 `two_factor` 为入口分派，导致规则④在 `two_factor=True` 的分支中无法生效。
2. 条件组合考虑不全：缺少"仅双因素认证"的边界组合处理策略。
3. 冗余代码：末尾 `return "Error State"` 不可达，会影响覆盖率评价并增加维护成本。

3. 覆盖标准有效性评估

- 若只做语句/分支覆盖，可能选取的用例不包含 TC04，从而无法发现该缺陷；
- 条件组合覆盖穷举了全部 8 种组合，路径覆盖覆盖了全部独立路径，因此必然触发 TC04，并能稳定定位缺陷。

建议策略：在类似"多条件逻辑判定"程序中，优先采用条件组合覆盖或至少做到条件/判定覆盖，再结合基本路径测试。

4. 修复建议

建议将"拒绝登录"的规则作为最高优先级，先行判断：

```
if not strong_password and not trusted_device:
    return "Access Denied"

if two_factor and (strong_password or trusted_device):
    return "High Security"

if (not two_factor) and strong_password and trusted_device:
    return "Medium Security"

return "Low Security"
```

同时删除不可达的 `return "Error State"`，并保留 TC01–TC08 的组合用例作为回归测试集。

七、讨论与结论

1. 本实验以条件组合覆盖 + 基本路径测试为主，覆盖了所有 8 种布尔组合与 5 条独立路径，覆盖效果系统且可复现。
2. 自动化测试成功发现并复现了关键缺陷：仅双因素认证但无强密码/可信设备时，程序与题意规则不一致。
3. 结论：对于多条件决策逻辑，单靠语句/分支覆盖可能产生“覆盖率高但仍漏缺陷”的情况；条件组合覆盖与路径覆盖能显著提升缺陷检出率。

一、实验名称

白盒测试—集成测试（自顶向下）

二、实验目的

本实验以一个简单计算器系统为对象，通过自顶向下集成测试验证模块协作的正确性，目标包括：

1. 掌握自顶向下集成测试策略：从主控模块开始，使用桩模块替代下层模块并逐步替换为真实模块；
2. 能将集成验证拆解为三类关注点：接口调用正确性、数据传递正确性、输出行为正确性；
3. 使用 pytest 构建可重复执行的集成测试用例集，并生成可留档的测试结果文件。

三、实验内容

被测系统为“三模块计算器”，包含：

- Main：主控模块，接收输入并协调 Calc 与 Display；
- Calc：计算模块，返回 `(result, error_flag)`；
- Display：显示模块，根据 `error_flag` 输出结果或错误信息。

模块层次结构：

```
          主控模块(Main)
            /      \
计算模块(Calc)    显示模块(Display)
```

四、实验仪器

1. 操作系统：`Linux-6.18.2-2-cachyos-x86_64-with-glibc2.42`

2. Python：`3.13.11`
3. pytest：`9.0.2`

五、实验原理

1. 集成测试目标

集成测试关注模块组合后的协作行为是否符合接口契约与业务规格，常见缺陷包括：

- 接口参数/返回值约定不一致；
- 错误标志传递与错误处理策略不一致；
- 主控模块控制流遗漏（未调用/重复调用下层模块）；
- 输出行为与错误分支不一致。

2. 自顶向下集成策略

自顶向下策略以高层模块为入口，先通过桩模块模拟下层模块行为以验证接口与控制流，再逐步用真实模块替换桩模块，直到完整系统集成。

本实验采用深度优先方式：优先形成一条可执行主链路，逐步替换 Calc 与 Display。

六、实验过程及实验结果

1. 集成步骤设计

按自顶向下策略分 4 个阶段验证：

1. `Main + 桩Calc + 桩Display`
   - 断言重点：Main 是否正确调用 Calc（参数传递）并把返回值转交给 Display。
2. `Main + 真Calc + 桩Display`
   - 断言重点：Main 驱动下 Calc 计算结果/错误标志是否正确，并能正确转交给 Display。
3. `Main + 桩Calc + 真Display`
   - 断言重点：输出行为是否符合规格（stdout）。
4. `Main + 真Calc + 真Display`（全量集成）
   - 断言重点：端到端功能与异常处理符合规格。

桩模块注入方式：使用 pytest 的 `monkeypatch` 替换 `src.calculator.main` 中被引用的 `calculate` / `display`，实现“逐步替换”的集成过程。

2. 测试用例设计

全量集成用例覆盖基础功能与异常场景：

用例编号	operator	num1	num2	预期 stdout	设计意图
IT01	`+`	1	2	`Result: 3`	加法主链路
IT02	`-`	5	2	`Result: 3`	减法主链路
IT03	`*`	3	4	`Result: 12`	乘法主链路
IT04	`/`	8	4	`Result: 2.0`	正常除法
IT05	`/`	8	0	`Error occurred`	除数为 0
IT06	`?`	1	2	`Error occurred`	非法操作符

说明：Step 1–Step 3 复用同一组输入，但断言关注点分别为接口调用、数据传递、stdout 输出。

扩展负向用例（用于"找碴"，验证输入域约束与鲁棒性）：

用例编号	operator	num1	num2	期望 stdout	实际 stdout	缺陷现象
IT07	`+`	`"1"`	`"2"`	`Error occurred`	`Result: 12`	字符串被当作合法操作数发生拼接
IT08	`*`	`"a"`	3	`Error occurred`	`Result: aaa`	字符串被当作合法操作数发生重复

3. 待测程序代码与 pytest 用例

被测模块为 `calculator`，三层结构分别为 Main（主控）、Calc（计算）与 Display（输出）。对应代码如下。

`calculator/main.py` 完整代码如下：

```
from __future__ import annotations

from src.calculator.calc import calculate
from src.calculator.display import display


def main(operator, num1, num2):
    """
    主控模块：协调各模块工作
    """
    result, error = calculate(operator, num1, num2)
    display(result, error)
```

`calculator/calc.py` 完整代码如下：

```
def calculate(operator, num1, num2):
    """
    计算模块：执行数学计算
    返回: (计算结果, 是否错误)
    """
    if operator == '+':
        return num1 + num2, False
    elif operator == '-':
        return num1 - num2, False
    elif operator == '*':
        return num1 * num2, False
    elif operator == '/':
        if num2 == 0:
            return None, True
        return num1 / num2, False
    else:
        return None, True
```

`calculator/display.py` 完整代码如下：

```
def display(result, error):
    """
    显示模块：输出结果
    """
    if error:
        print("Error occurred")
    else:
        print("Result:", result)
```

```python
from __future__ import annotations

import pytest

from src.calculator.main import main


def test_top_down_step1_main_with_calc_and_display_stubs(monkeypatch):
    """
    Step 1（自顶向下）：仅集成主控模块，计算/显示使用桩模块。
    目标：验证 Main 与下层模块接口调用正确性（参数传递 + 返回值转交）。
    """
    events: list[tuple[str, tuple[object, ...]]] = []

    def stub_calculate(operator, num1, num2):
        events.append(("calculate", (operator, num1, num2)))
        return 42, False

    def stub_display(result, error):
        events.append(("display", (result, error)))

    import src.calculator.main as main_module

    monkeypatch.setattr(main_module, "calculate", stub_calculate)
    monkeypatch.setattr(main_module, "display", stub_display)

    main("+", 1, 2)

    assert events == [
        ("calculate", ("+", 1, 2)),
        ("display", (42, False)),
    ]


@pytest.mark.parametrize(
    "operator,num1,num2,expected_result,expected_error",
    [
        ("+", 1, 2, 3, False),
        ("-", 5, 2, 3, False),
        ("*", 3, 4, 12, False),
        ("/", 8, 4, 2.0, False),
        ("/", 8, 0, None, True),
        ("?", 1, 2, None, True),
    ],
)
def test_top_down_step2_main_with_real_calc_stub_display(
    operator, num1, num2, expected_result, expected_error, monkeypatch
):
```

```
    """
    Step 2（自顶向下）：替换真实 Calc，Display 仍使用桩模块。
    目标：验证 Main→Calc 计算结果/错误标志正确，且能正确转交给 Display。
    """
    import src.calculator.main as main_module

    captured: list[tuple[object, object]] = []

    def stub_display(result, error):
        captured.append((result, error))

    monkeypatch.setattr(main_module, "display", stub_display)

    main(operator, num1, num2)

    assert captured == [(expected_result, expected_error)]


@pytest.mark.parametrize(
    "stub_result,stub_error,expected_stdout",
    [
        (3, False, "Result: 3\n"),
        (2.0, False, "Result: 2.0\n"),
        (None, True, "Error occurred\n"),
    ],
)
def test_top_down_step3_main_with_stub_calc_real_display(
    stub_result, stub_error, expected_stdout, monkeypatch, capsys
):
    """
    Step 3（自顶向下）：Display 替换为真实模块，Calc 使用桩模块。
    目标：验证 Main→Display 的输出行为（stdout）符合规格。
    """
    import src.calculator.main as main_module

    calc_calls: list[tuple[object, object, object]] = []

    def stub_calculate(operator, num1, num2):
        calc_calls.append((operator, num1, num2))
        return stub_result, stub_error

    monkeypatch.setattr(main_module, "calculate", stub_calculate)

    main("+", 1, 2)
    out = capsys.readouterr().out

    assert calc_calls == [("+", 1, 2)]
    assert out == expected_stdout
```

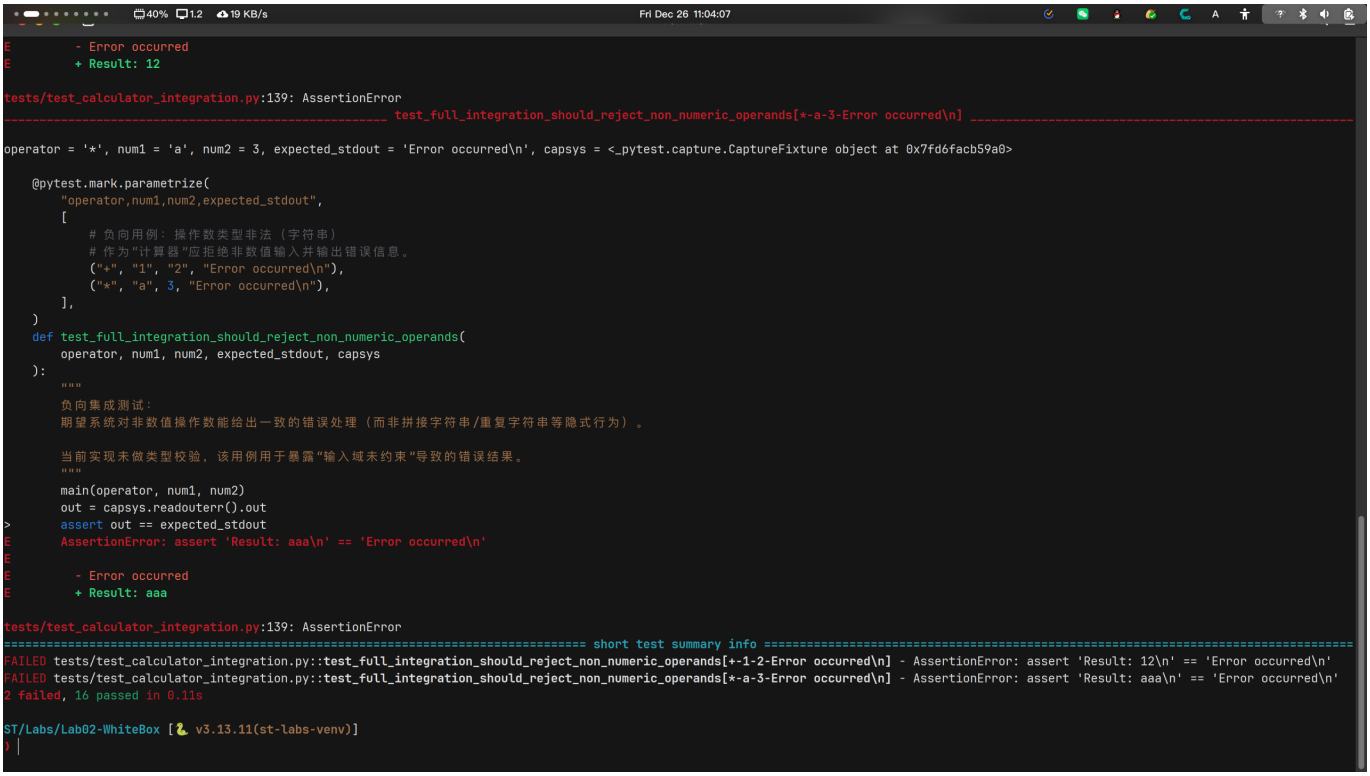
```
@pytest.mark.parametrize(
    "operator,num1,num2,expected_stdout",
    [
        ("+", 1, 2, "Result: 3\n"),
        ("-", 5, 2, "Result: 3\n"),
        ("*", 3, 4, "Result: 12\n"),
        ("/", 8, 4, "Result: 2.0\n"),
        ("/", 8, 0, "Error occurred\n"),
        ("?", 1, 2, "Error occurred\n"),
    ],
)
def test_top_down_step4_full_integration(operator, num1, num2, expected_stdout,
capsys):
    """
    Step 4（自顶向下）：Main + Calc + Display 全部集成验证。
    """
    main(operator, num1, num2)
    out = capsys.readouterr().out
    assert out == expected_stdout


@pytest.mark.parametrize(
    "operator,num1,num2,expected_stdout",
    [
        # 负向用例：操作数类型非法（字符串）
        # 作为“计算器”应拒绝非数值输入并输出错误信息。
        ("+", "1", "2", "Error occurred\n"),
        ("*", "a", 3, "Error occurred\n"),
    ],
)
def test_full_integration_should_reject_non_numeric_operands(
    operator, num1, num2, expected_stdout, capsys
):
    """
    负向集成测试：
    期望系统对非数值操作数能给出一致的错误处理（而非拼接字符串/重复字符串等隐式行为）。
    """
    main(operator, num1, num2)
    out = capsys.readouterr().out
    assert out == expected_stdout
```

4. 全量用例执行与原始结果

执行命令：

```
"/tmp/st-labs-venv/bin/python" -m pytest -q "tests/test_calculator_integration.py"
```

终端输出：



七、讨论与结论

本实验在 IT01–IT06 的正常输入场景下均得到预期输出；在 IT07–IT08 的负向输入（字符串操作数）场景下出现 2 个失败，用例的期望为输出 `Error occurred`，而实际输出为字符串拼接/重复结果（`Result: 12`、`Result: aaa`），该现象反映出被测实现未对输入域进行类型约束。

1. 缺陷现象与根因

   - 现象：`calc.calculate()` 对非数值类型仍执行 `+/*` 的 Python 动态语义，返回值被当作"正常结果"传递到 Display。
   - 根因：计算模块缺少对操作数类型的校验，主控模块也未对返回值语义进行约束（仅依赖 `error` 标志）。

2. 处理方法与修复建议（不修改 src，仅给出建议）

   - 在 `calculate()` 入口增加类型校验，仅允许 `int/float`；不满足则返回 `(None, True)`。
   - 或在 `main()` 中捕获异常并统一转换为错误输出，保证显示层行为一致。

3. 集成策略讨论

   - 深度优先（本实验采用）更快形成端到端可运行链路，便于尽早验证主流程与接口调用顺序。
   - 广度优先更早验证主控模块与多个下层模块的接口一致性，对多分支接口契约偏差更敏感。

4. 结论

   自顶向下集成测试通过“桩→逐步替换→全量集成”的过程，使定位粒度从端到端失败逐步收敛到具体模块（Calc/Display）接口与输入域约束问题；同时，结合负向用例能够揭示“看似成功但语义错误”的隐蔽缺陷。