一、实验名称

黑盒测试—边界值分析与等价类划分

二、实验目的

本实验旨在通过对典型逻辑程序（三角形判别系统）的测试实践，提升软件测试的理论认知与工程实践能力：

1. 深入掌握黑盒测试方法，包括边界值分析法与等价类划分法，并理解输入域探索的重要性。
2. 培养严密的测试设计思维，从需求规格说明书中提取测试需求，构建测试模型，以尽可能少的测试用例实现更大覆盖率。
3. 掌握自动化测试流程，包括使用 Python（Pytest 框架）实现自动化测试脚本、断言与测试夹具，以及利用回归测试验证缺陷修复并进行根因分析。

三、实验内容

实验对象为两个功能相同但实现不同的三角形判别程序（triangle_program1.py 与 triangle_program2.py）。两者模拟真实开发场景中常见的逻辑错误。

被测对象功能规范：

- 输入接口：接收三个整数变量 a, b, c。
- 业务逻辑：基于欧几里得几何原理判断三条边是否能构成三角形，并进一步识别等边、等腰或一般三角形。
- 输出定义：返回“等边三角形”“等腰三角形”“一般三角形”“非三角形”或“无效输入”。

实验核心任务：

1. Program 1：应用边界值分析法设计不少于 10 个测试用例，覆盖最小值、最大值及几何临界情况。
2. Program 2：应用等价类划分法设计不少于 8 个测试用例，覆盖所有有效与无效等价类。
3. 工程化实现与分析：构建 Python 测试工程，批量执行并比对测试结果，对缺陷进行逆向分析并提出重构建议。

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

黑盒测试基于用户视角，不关注程序内部结构。本实验涉及以下两种核心技术。

1. 边界值分析法

理论基础指出大量逻辑错误出现在输入输出边界区域。常见原因包括对比较运算符的使用失误。

操作原则：识别输入边界，采用"五点法"或"七点法"测试最小值、略大于最小值、正常值、略小于最大值、最大值等典型点。

在本实验中，关键边界包括数值边界（如 1）以及几何边界（如 $a+b=c$）。例如 $(1,2,3)$ 比 $(1,2,5)$ 更容易暴露错误使用 $\geq$ 的逻辑缺陷。

2. 等价类划分法

理论基础认为输入域可划分为若干互不交叠的子集（等价类），在同一等价类中选择的任意输入具有相同的测试效果。

分类策略：

- 有效等价类：满足需求的合法输入
- 无效等价类：不符合需求的输入，包括负数、零、浮点数、字符等

六、实验过程及实验结果

第一部分：测试用例设计

本阶段通过逻辑分析将需求转化为可执行的测试数据。

1. Program 1 的边界值分析

设计策略：确定边长范围为 $[1,\infty)$，基于单故障假设一次主要变动一个边界值，或关注几何临界点。

边界关注点：

- 数值边界：1
- 几何临界：$a+b=c$
- 逻辑组合：等腰三角形中的不同相等组合

测试用例如下：

测试类别	输入 (a, b, c)	预期输出	设计意图
最小值边界	$(1, 1, 1)$	等边三角形	最小正整数等边
最小等腰	$(2, 2, 1)$	等腰三角形	覆盖最小等腰组合
典型一般	$(3, 4, 5)$	一般三角形	典型一般三角形
临界非三角形	$(1, 1, 2)$	非三角形	覆盖 $a + b = c$ 边界
临界非三角形	$(2, 5, 3)$	非三角形	覆盖 $a + c = b$ 边界
最小一般	$(2, 3, 4)$	一般三角形	最小满足不等式的一般三角形
等腰（a=b）	$(5, 5, 8)$	等腰三角形	覆盖 $a = b$ 分支
等腰（a=c）	$(5, 4, 5)$	等腰三角形	覆盖 $a = c$ 分支
等腰（b=c）	$(3, 5, 5)$	等腰三角形	覆盖 $b = c$ 分支
大数等边	$(1000, 1000, 1000)$	等边三角形	大数值等边
大数等腰	$(1000, 1000, 1)$	等腰三角形	大数值等腰
非法：零值	$(0, 1, 1)$	无效输入	非正整数
非法：负值	$(-1, 2, 3)$	无效输入	非正整数
非法：浮点	$(1.5, 2, 3)$	无效输入	类型校验
非法：字符串	$(”a”, 1, 1)$	无效输入	类型校验

2. Program 2 的等价类划分

将输入划分为以下等价类：

- 有效：等边、等腰、一般
- 无效：非三角形、零值、负值、浮点、字符
- 需特别关注特殊非三角形（如 $(1, 1, 3)$）

测试用例如下：

等价类	输入 (a,b,c)	预期输出	设计意图
有效：等边	$(5,5,5)$	等边三角形	覆盖等边分支
有效：等腰（a=b）	$(5,5,3)$	等腰三角形	覆盖等腰分支
有效：等腰（b=c）	$(3,5,5)$	等腰三角形	覆盖等腰分支
有效：等腰（a=c）	$(5,3,5)$	等腰三角形	覆盖等腰分支
有效：一般	$(3,4,5)$	一般三角形	覆盖一般分支
无效：非三角形	$(1,2,5)$	非三角形	明显不满足不等式
无效：非三角形（等腰但不等式失败）	$(1,1,3)$	非三角形	验证“不等式优先级”
无效：非三角形（等腰但不等式失败）	$(2,5,2)$	非三角形	覆盖 a+c≤b
无效：非三角形（边界）	$(2,2,4)$	非三角形	覆盖 $a+b=c$ 边界
无效：零值	$(0,1,1)$	无效输入	非正整数
无效：负值	$(-1,2,3)$	无效输入	非正整数
无效：浮点	$(1.5,2,3)$	无效输入	类型校验
无效：字符串	$(”a”,1,1)$	无效输入	类型校验
缺陷探测：不等式校验不完整	$(10,5,2)$	非三角形	检查是否仅验证 `a+b>c`

第二部分：待测程序代码与测试实现

待测程序 `triangle_program1.py` 完整代码如下：

```
def classify_triangle1(a, b, c):
    if not isinstance(a, int) or not isinstance(b, int) or not isinstance(c, int):
        return "无效输入"

    if a <= 0 or b <= 0 or c <= 0:
        return "无效输入"

    if a + b <= c or a + c <= b or b + c <= a:
        return "非三角形"

    if a == b and b == c:
        return "等边三角形"
    elif a == b or b == c or a == c:
        return "等腰三角形"
    else:
        return "一般三角形"
```

待测程序 `triangle_program2.py` 完整代码如下：

```
def classify_triangle2(a, b, c):
    if not isinstance(a, int) or not isinstance(b, int) or not isinstance(c, int):
        return "无效输入"

    if a <= 0 or b <= 0 or c <= 0:
        return "无效输入"

    if a == b == c:
        return "等边三角形"
    elif a == b or b == c or a == c:
        return "等腰三角形"
    elif a + b > c:
        return "一般三角形"
    else:
        return "非三角形"
```

使用 pytest 编写自动化测试脚本

测试文件 `test_triangle_program1.py` 完整代码如下：

```python
import pytest
from src.triangle_program1 import classify_triangle1


@pytest.mark.boundary
class TestTriangleProgram1BoundaryValue:

    # 测试最小正整数等边三角形
    def test_min_equilateral(self):
        assert classify_triangle1(1, 1, 1) == "等边三角形"

    # 测试最小等腰三角形
    def test_min_isosceles(self):
        assert classify_triangle1(2, 2, 1) == "等腰三角形"

    # 测试一般三角形
    def test_normal_triangle(self):
        assert classify_triangle1(3, 4, 5) == "一般三角形"

    # 测试a+b=c边界情况
    def test_boundary_non_triangle_1(self):
        assert classify_triangle1(1, 1, 2) == "非三角形"

    # 测试a+c=b边界情况
    def test_boundary_non_triangle_2(self):
        assert classify_triangle1(2, 5, 3) == "非三角形"

    # 测试零值输入
    def test_zero_value(self):
        assert classify_triangle1(0, 1, 1) == "无效输入"

    # 测试负值输入
    def test_negative_value(self):
        assert classify_triangle1(-1, 2, 3) == "无效输入"

    # 测试大数值等边三角形
    def test_large_equilateral(self):
        assert classify_triangle1(1000, 1000, 1000) == "等边三角形"

    # 测试大数值等腰三角形
    def test_large_isosceles(self):
        assert classify_triangle1(1000, 1000, 1) == "等腰三角形"

    # 测试浮点数输入
    def test_float_input(self):
        assert classify_triangle1(1.5, 2, 3) == "无效输入"

    # 测试字符串输入
```

```
    def test_string_input(self):
        assert classify_triangle1("a", 1, 1) == "无效输入"

    # 测试等腰三角形(a==b)
    def test_isosceles_boundary(self):
        assert classify_triangle1(5, 5, 8) == "等腰三角形"

    # 测试等腰三角形(a==c)
    def test_isosceles_a_eq_c(self):
        assert classify_triangle1(5, 4, 5) == "等腰三角形"

    # 测试等腰三角形(b==c)
    def test_isosceles_b_eq_c(self):
        assert classify_triangle1(3, 5, 5) == "等腰三角形"

    # 测试大数值等腰三角形(a==c)
    def test_isosceles_large_a_eq_c(self):
        assert classify_triangle1(10000, 500, 10000) == "等腰三角形"

    # 额外增强：一般三角形的排列不变性（Permutation invariance）
    @pytest.mark.parametrize(
        "a,b,c",
        [
            (3, 4, 5),
            (4, 5, 3),
            (5, 3, 4),
        ],
        ids=["3-4-5", "4-5-3", "5-3-4"],
    )
    def test_permutation_invariance_general_triangle(self, a, b, c):
        assert classify_triangle1(a, b, c) == "一般三角形"

    # 额外边界：最小满足不等式的一般三角形（2,3,4）
    def test_min_general_triangle(self):
        assert classify_triangle1(2, 3, 4) == "一般三角形"

    # 仅依据题目文档的“正整数”与 src 行为进行测试，不引入额外规范假设
```

测试文件 `test_triangle_program2.py` 完整代码如下：

```python
import pytest
from src.triangle_program2 import classify_triangle2


@pytest.mark.equivalence
class TestTriangleProgram2EquivalenceClass:

    # 测试等边三角形
    def test_equilateral_class(self):
        assert classify_triangle2(5, 5, 5) == "等边三角形"

    # 测试等腰三角形(a==b)
    def test_isosceles_class_1(self):
        assert classify_triangle2(5, 5, 3) == "等腰三角形"

    # 测试等腰三角形(b==c)
    def test_isosceles_class_2(self):
        assert classify_triangle2(3, 5, 5) == "等腰三角形"

    # 测试等腰三角形(a==c)
    def test_isosceles_class_3(self):
        assert classify_triangle2(5, 3, 5) == "等腰三角形"

    # 测试一般三角形
    def test_normal_triangle_class(self):
        assert classify_triangle2(3, 4, 5) == "一般三角形"

    # 测试非三角形
    def test_non_triangle_class(self):
        assert classify_triangle2(1, 2, 5) == "非三角形"

    # 测试三角形不等式验证不完整
    def test_non_triangle_incomplete_check(self):
        assert classify_triangle2(10, 5, 2) == "非三角形"

    # 测试零值输入
    def test_zero_value_class(self):
        assert classify_triangle2(0, 1, 1) == "无效输入"

    # 测试负值输入
    def test_negative_value_class(self):
        assert classify_triangle2(-1, 2, 3) == "无效输入"

    # 测试浮点数输入
    def test_float_input_class(self):
        assert classify_triangle2(1.5, 2, 3) == "无效输入"

    # 测试字符串输入
```

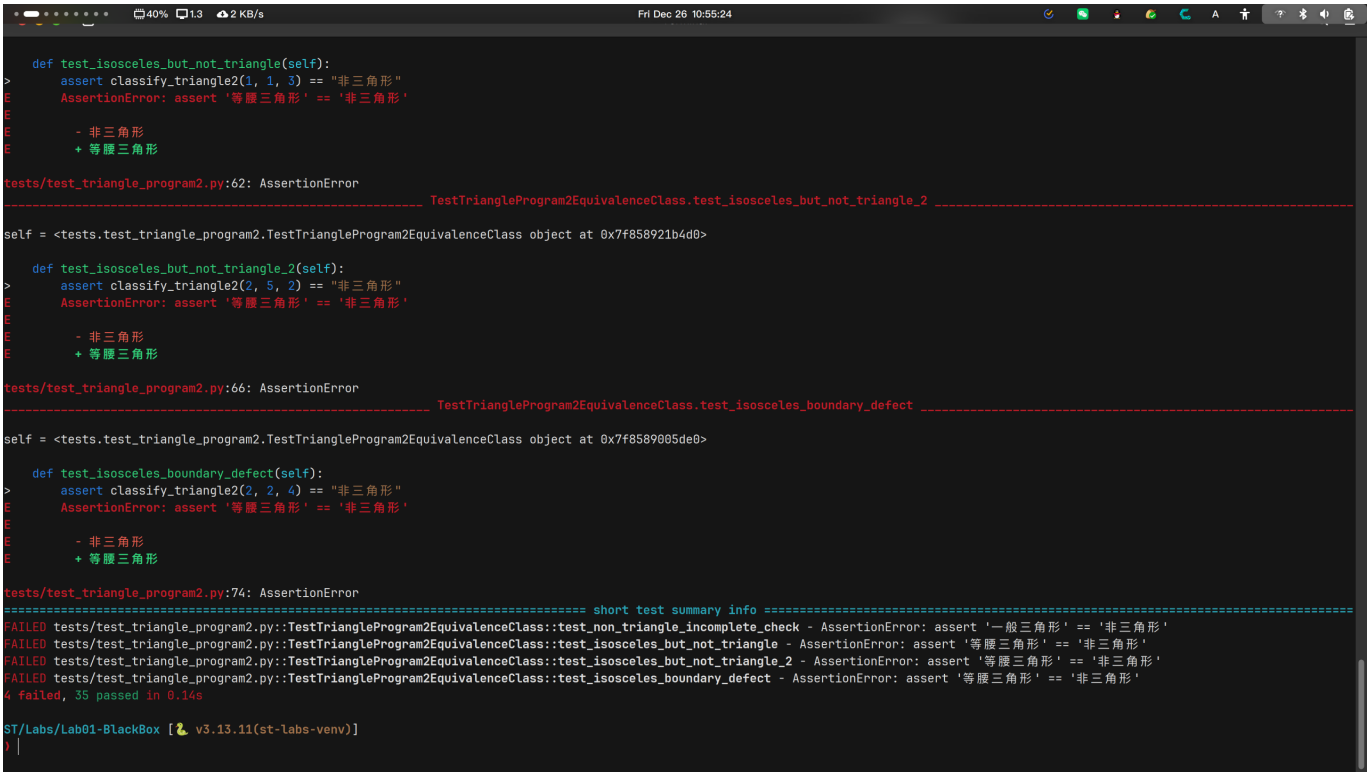
```
    def test_string_input_class(self):
        assert classify_triangle2("a", 1, 1) == "无效输入"

    # 测试超大数值等边
    def test_large_equilateral(self):
        assert classify_triangle2(9999, 9999, 9999) == "等边三角形"

    # 测试超大数值等腰
    def test_large_isosceles(self):
        assert classify_triangle2(9999, 9999, 1) == "等腰三角形"

    # 测试等腰但不满足三角形不等式
    def test_isosceles_but_not_triangle(self):
        assert classify_triangle2(1, 1, 3) == "非三角形"

    # 测试等腰但不满足三角形不等式(另一种情况)
    def test_isosceles_but_not_triangle_2(self):
        assert classify_triangle2(2, 5, 2) == "非三角形"

    # 测试大数值等边三角形
    def test_equilateral_edge_case(self):
        assert classify_triangle2(1000, 1000, 1000) == "等边三角形"

    # 测试等腰且a+b=c边界
    def test_isosceles_boundary_defect(self):
        assert classify_triangle2(2, 2, 4) == "非三角形"

    # 额外增强：一般三角形的排列不变性（Permutation invariance）
    @pytest.mark.parametrize(
        "a,b,c",
        [
            (3, 4, 5),
            (4, 5, 3),
            (5, 3, 4),
        ],
        ids=["3-4-5", "4-5-3", "5-3-4"],
    )
    def test_permutation_invariance_general_triangle(self, a, b, c):
        assert classify_triangle2(a, b, c) == "一般三角形"

    # 仅依据题目文档的“正整数”与 src 行为进行测试，不引入额外规范假设
```

第三部分：全量用例执行与原始结果

执行命令：

```
"/tmp/st-labs-venv/bin/python" -m pytest -q -o addopts=
"tests/test_triangle_program1.py" "tests/test_triangle_program2.py"
```

终端输出



第四部分：缺陷分析

1. Program 1（`triangle_program1.py`）：该部分测试用例全部通过，未观测到与测试用例期望不一致的返回结果。

2. Program 2（`triangle_program2.py`）：用例执行出现 4 个失败，均与“先判类型/等腰，再校验三角形不等式”以及“三角形不等式校验不完整”相关：

- 算法不完备：仅校验 `a + b > c`，未校验 `a + c > b` 与 `b + c > a`，导致 `(10, 5, 2)` 被误判为“一般三角形”。
- 控制流顺序问题：在不校验三角形存在性的情况下先判定等腰，导致 `(1, 1, 3)`、`(2, 5, 2)`、`(2, 2, 4)` 被误判为“等腰三角形”。

若仅从逻辑上给出修复建议，可将“不等式校验”前置且完整校验三条不等式，随后再做等边/等腰/一般分类，例如：

```
if a + b <= c or a + c <= b or b + c <= a:
    return "非三角形"

if a == b == c:
    return "等边三角形"
elif a == b or b == c or a == c:
    return "等腰三角形"
return "一般三角形"
```

七、讨论与结论

本实验的主要现象为：`triangle_program1.py` 的边界值用例集全部通过；`triangle_program2.py` 的等价类用例集出现 4 个失败，均集中在“三角形不等式校验”与“控制流顺序”两类问题上。

1. 失败用例与缺陷定位
   - `(10, 5, 2)`：仅校验 `a+b>c` 会把不满足其它不等式的组合误判为“一般三角形”。
   - `(1, 1, 3)`、`(2, 5, 2)`、`(2, 2, 4)`：在不做不等式校验的情况下先判定等腰，会把“等腰但非三角形”的输入误判为“等腰三角形”。
2. 方法有效性讨论
   - 边界值分析不仅覆盖数值边界（如 0、负数、大数），也覆盖几何边界（如 `a+b=c`），用于验证不等式判定的临界行为。
   - 等价类划分把输入域拆分为“有效三角形类别 / 非三角形 / 无效输入”等子集，并通过代表值触发不同分支；失败用例正是在“数值合法但逻辑非法”的等价类中暴露。
3. 结论
   本实验完成了边界值分析与等价类划分的用例设计与自动化执行，并在 `triangle_program2.py` 上稳定复现了逻辑缺陷：不等式校验不完整与分支顺序不当。整体结果表明，系统化的黑盒测试方法结合自动化断言能够有效发现多条件判定程序中的组合遗漏问题。

一、实验名称

黑盒测试2—判定表法和因果图法

二、实验目的

本实验在三角形判别问题的基础上，进一步练习基于模型的黑盒测试方法，具体目标为：

1. 掌握判定表法、因果图法的基本概念与建模步骤；
2. 能够从规格说明中抽取输入条件与输出结果，构建判定表 / 因果图并推导测试用例；
3. 使用 pytest 编写自动化测试，对被测程序进行路径覆盖和结果校验。

三、实验内容

实验对象为题目给出的两个三角形判别程序：

- Program 3：`triangle_program3.py`，采用判定表法进行建模与测试；
- Program 4：`triangle_program4.py`，采用因果图法进行建模与测试。

被测程序的功能需求与黑盒测试1部分一致：
输入三个边长 a, b, c，先检查输入有效性，再检查三角形不等式，最后根据边长相等关系输出“等边三角形”“等腰三角形”“一般三角形”“非三角形”或“无效输入”。

四、实验仪器

本实验使用的软件与运行环境如下（来自实际运行时信息）：

1. 操作系统：`Linux-6.18.2-2-cachyos-x86_64-with-glibc2.42`
2. Python：`3.13.11`（解释器：`/tmp/st-labs-venv/bin/python`）
3. pytest：`9.0.2`（pluggy：`1.6.0`）

五、实验原理

1. 判定表法原理

判定表法适用于输入条件较多、存在组合与优先级关系的场景。基本思路是：

- 从需求中抽取一组布尔条件（C1, C2, …）；
- 抽取与之对应的输出动作（A1, A2, …）；
- 列出“条件 × 规则”矩阵，每一列代表一条可能的输入组合（规则）；
- 为每条规则指定唯一的输出动作，并合并不必要或不可能的规则；
- 从每条规则中选取代表性输入数据作为测试用例。

在本实验中，判定表法用于对 `triangle_program3` 的输入条件进行系统枚举，覆盖全部输出分支。

2. 因果图法原理

因果图法在判定表的基础上，用图形化方式刻画“输入条件（原因）”与“输出结果（效果）”之间的逻辑关系，并显式引入各种约束（互斥、仅能选一、至少选一等）。
典型步骤为：识别原因和结果 → 用逻辑连接线构造因果图 → 添加约束 → 将因果图转换为判定表 → 由判定表导出测试用例。

在本实验中，因果图法用于分析 `triangle_program4` 中“输入合法性校验—三角形不等式—类型判定”之间的优先级关系，帮助设计覆盖主要因果路径的测试用例。

六、实验过程及实验结果

第一部分：测试用例设计

1. Program 3 的判定表法

对 `classify_triangle3(a,b,c)` 抽象出以下条件与动作：

- 条件：
  - C1：类型有效（a, b, c 均为 `int`）；
  - C2：数值有效（a, b, c 均为正数）；
  - C3：满足三角形不等式（a+b>c 且 a+c>b 且 b+c>a）；
  - C4：三边全相等（a==b==c）；
  - C5：任意两边相等（a==b 或 a==c 或 b==c）。
- 动作：
  - A1：返回“无效输入”；
  - A2：返回“非三角形”；
  - A3：返回“等边三角形”；
  - A4：返回“等腰三角形”；
  - A5：返回“一般三角形”。

根据条件组合和优先级，可将 2^5=32 种条件组合在约束 C4⇒C5 和动作优先级 A1>A2>A3>A4>A5 下归并为 6 条有意义的规则（R1–R6）：

Program 3 判定表：

规则	C1 类型有效	C2 数值为正	C3 满足不等式	C4 三边全等	C5 任意两边相等	动作
R1	F	-	-	-	-	A1 无效输入
R2	T	F	-	-	-	A1 无效输入
R3	T	T	F	-	-	A2 非三角形
R4	T	T	T	T	T	A3 等边三角形
R5	T	T	T	F	T	A4 等腰三角形
R6	T	T	T	F	F	A5 一般三角形

其中 `-` 表示对该条件取值不敏感；例如 C4 为真时 C5 必然为真，C4=真且 C5=假 属于不可能组合，已在表中省略，其余不可能或被优先级覆盖的组合也被归并到上述 6 条规则中。

在此基础上，为每条有意义的规则至少设计一个具体测试用例，得到如下 13 个用例：

Program 3 判定表法测试用例（13 个）：

输入 (a, b, c)	预期结果	测试类型说明
`("1", 2, 3)`	无效输入	C1 为假：非整数类型
`(1.0, 2, 3)`	无效输入	C1 为假：浮点数类型
`(0, 1, 1)`	无效输入	C2 为假：含 0
`(-1, 2, 3)`	无效输入	C2 为假：含负数
`(1, 2, 3)`	非三角形	C3 为假：a+b=c 边界
`(2, 2, 5)`	非三角形	C3 为假：明显不满足不等式
`(5, 5, 5)`	等边三角形	C3、C4 为真：等边三角形
`(1000, 1000, 1000)`	等边三角形	等边大数值覆盖
`(4, 4, 5)`	等腰三角形	C5 为真：a=b
`(5, 3, 5)`	等腰三角形	C5 为真：a=c
`(3, 5, 5)`	等腰三角形	C5 为真：b=c
`(3, 4, 5)`	一般三角形	C3 为真且无边相等：典型一般三角形
`(999, 1000, 1001)`	一般三角形	一般三角形大数值覆盖

2. Program 4 的因果图法

在 `classify_triangle4(a,b,c)` 上，将输入条件建模为原因 C1~~C5、输出建模为结果 E1~~E5：

- 原因：
  - C1：a,b,c 均为整数；
  - C2：a,b,c 均为正数；
  - C3：满足三角形不等式；
  - C4：三边全相等；
  - C5：任意两边相等。
- 结果：
  - E1：无效输入；E2：非三角形；E3：等边三角形；E4：等腰三角形；E5：一般三角形。

在因果图中加入约束：E1 > E2 > E3 > E4 > E5 的优先级；C4 ⇒ C5；`not C3` 时不能出现 E3/E4/E5。

结合因果图和约束，可得到以下简化判定表（R1–R6）：

Program 4 因果图转换判定表：

规则	C1 数值类型	C2 为正	C3 满足不等式	C4 三边全等	C5 任意两边相等	结果
R1	F	-	-	-	-	E1 无效输入
R2	T	F	-	-	-	E1 无效输入
R3	T	T	F	-	-	E2 非三角形
R4	T	T	T	T	T	E3 等边三角形
R5	T	T	T	F	T	E4 等腰三角形
R6	T	T	T	F	F	E5 一般三角形

其中 `-` 表示对该条件不敏感；例如 C4 为真时 C5 必然为真，C4=真且 C5=假 属于不可能组合已被排除，其余被约束或优先级覆盖的组合也归并到上述 6 条规则中。

在此基础上，为每条规则设计 1~2 个代表性测试用例，得到如下 11 个用例：

Program 4 因果图法测试用例（11 个）：

输入 (a, b, c)	预期结果	测试类型说明
`(1.2, 2, 3)`	无效输入	not C1：浮点数输入
`("x", 2, 3)`	无效输入	not C1：字符串输入
`(0, 2, 3)`	无效输入	not C2：含 0
`(1, -2, 3)`	无效输入	not C2：含负数
`(1, 2, 3)`	非三角形	not C3：a+b=c 边界
`(3, 1, 5)`	非三角形	not C3：明显不满足不等式
`(7, 7, 7)`	等边三角形	C3、C4 为真：等边三角形
`(6, 6, 7)`	等腰三角形	C3 为真且 a=b
`(6, 7, 6)`	等腰三角形	C3 为真且 a=c
`(7, 6, 6)`	等腰三角形	C3 为真且 b=c
`(5, 6, 7)`	一般三角形	C3 为真且无边相等：一般三角形

第二部分：测试程序实现

Program 3/4 的实现代码如下（被测对象为 `triangle_program3.py` 与 `triangle_program4.py`）。

`triangle_program3.py` 完整代码如下：

```python
def classify_triangle3(a, b, c):
    # 类型与取值校验（与 program1 保持一致：仅 int）
    if not isinstance(a, int) or not isinstance(b, int) or not isinstance(c, int):
        return "无效输入"
    if a <= 0 or b <= 0 or c <= 0:
        return "无效输入"

    # 不等式校验（先于分类）
    if a + b <= c or a + c <= b or b + c <= a:
        return "非三角形"

    # 分类逻辑
    if a == b and b == c:
        return "等边三角形"
    elif a == b or a == c or b == c:
        return "等腰三角形"
    else:
        return "一般三角形"
```

`triangle_program4.py` 完整代码如下：

```python
def classify_triangle4(a, b, c):
    # 类型与取值校验（仅 int）
    if not isinstance(a, int) or not isinstance(b, int) or not isinstance(c, int):
        return "无效输入"
    if a <= 0 or b <= 0 or c <= 0:
        return "无效输入"

    # 不等式校验
    if not (a + b > c and a + c > b and b + c > a):
        return "非三角形"

    # 分类逻辑
    if a == b == c:
        return "等边三角形"
    elif a == b or b == c or a == c:
        return "等腰三角形"
    else:
        return "一般三角形"
```

第三部分：pytest 用例实现

测试文件 `test_triangle_program3.py` 完整代码如下：

```
import pytest
from src.triangle_program3 import classify_triangle3


@pytest.mark.decision_table
class TestDecisionTableTriangleProgram3:
    """
    判定表法测试集（被测程序：triangle_program3）

    判定条件：
    - C1: 类型有效（a,b,c 均为 int）
    - C2: 数值有效（a,b,c 均为正数）
    - C3: 满足三角形不等式（a+b>c 且 a+c>b 且 b+c>a）
    - C4: 三边全相等（a==b==c）
    - C5: 任意两边相等（a==b 或 a==c 或 b==c）

    动作/期望（由优先级决定）：
    - A1: 返回 "无效输入"          when not C1 or not C2
    - A2: 返回 "非三角形"          when C1 and C2 and not C3
    - A3: 返回 "等边三角形"        when C1 and C2 and C3 and C4
    - A4: 返回 "等腰三角形"        when C1 and C2 and C3 and not C4 and C5
    - A5: 返回 "一般三角形"        when C1 and C2 and C3 and not C4 and not C5

    依据上述简化判定表，设计覆盖性的测试用例（>=12）。
    """

    @pytest.mark.parametrize(
        "a,b,c,expected",
        [
            # A1: 类型无效 -> 无效输入
            ("1", 2, 3, "无效输入"),          # 非数值类型
            (1.0, 2, 3, "无效输入"),           # 浮点数在本实现中视为无效
            # A1: 数值无效 -> 无效输入
            (0, 1, 1, "无效输入"),            # 含零
            (-1, 2, 3, "无效输入"),           # 负数

            # A2: 非三角形（不满足三角形不等式）
            (1, 2, 3, "非三角形"),            # 边界：a+b=c
            (2, 2, 5, "非三角形"),            # 明显不满足

            # A3: 等边三角形
            (5, 5, 5, "等边三角形"),
            (1000, 1000, 1000, "等边三角形"),

            # A4: 等腰三角形（满足不等式）
            (4, 4, 5, "等腰三角形"),          # a==b
            (5, 3, 5, "等腰三角形"),          # a==c
            (3, 5, 5, "等腰三角形"),          # b==c
```

```
            # A5: 一般三角形
            (3, 4, 5, "一般三角形"),
            (999, 1000, 1001, "一般三角形"),  # 任意无相等且满足不等式
        ],
    )
    def test_decision_table_core(self, a, b, c, expected):
        assert classify_triangle3(a, b, c) == expected

    @pytest.mark.parametrize(
        "a,b,c",
        [
            (1, 1, 3),  # a+b<=c 且 a==b
            (2, 2, 4),  # 边界：a+b=c 且 a==b
            (2, 5, 2),  # a+c<=b 且 a==c
        ],
    )
    def test_decision_table_isosceles_but_not_triangle(self, a, b, c):
        # 确保不等式优先：即便两边相等，但不满足三角形不等式时应为“非三角形”
        assert classify_triangle3(a, b, c) == "非三角形"

    # 额外增强：一般三角形的排列不变性
    @pytest.mark.parametrize(
        "a,b,c",
        [
            (3, 4, 5),
            (4, 5, 3),
            (5, 3, 4),
        ],
        ids=["3-4-5", "4-5-3", "5-3-4"],
    )
    def test_permutation_invariance_general_triangle(self, a, b, c):
        assert classify_triangle3(a, b, c) == "一般三角形"

    # 仅依据题目文档的“正整数”与 src 行为进行测试，不引入额外规范假设
```

测试文件 `ttest_triangle_program4.py` 完整代码如下：

```python
import pytest
from src.triangle_program4 import classify_triangle4


@pytest.mark.cause_effect
class TestCauseEffectTriangleProgram4:
    """
    因果图法测试集（被测程序：triangle_program4）

    原因：
    - C1: a,b,c 均为整数类型
    - C2: a,b,c 均为正数
    - C3: 满足三角形不等式（a+b>c 且 a+c>b 且 b+c>a）
    - C4: 三边全相等（a==b==c）
    - C5: 任意两边相等（a==b 或 a==c 或 b==c）

    结果：
    - E1: 无效输入
    - E2: 非三角形
    - E3: 等边三角形
    - E4: 等腰三角形
    - E5: 一般三角形

    约束：
    - 优先级：E1 > E2 > E3 > E4 > E5
    - C4 => C5（全相等蕴含两边相等）
    - not C3 -> 不能出现 E3/E4/E5

    根据因果路径覆盖生成至少 10 个测试。
    """

    @pytest.mark.parametrize(
        "a,b,c,expected",
        [
            # 路径：not C1 -> E1
            (1.2, 2, 3, "无效输入"),
            ("x", 2, 3, "无效输入"),

            # 路径：C1 & not C2 -> E1
            (0, 2, 3, "无效输入"),
            (1, -2, 3, "无效输入"),

            # 路径：C1 & C2 & not C3 -> E2
            (1, 2, 3, "非三角形"),
            (3, 1, 5, "非三角形"),

            # 路径：C1 & C2 & C3 & C4 -> E3
            (7, 7, 7, "等边三角形"),
```

```
            # 路径：C1 & C2 & C3 & not C4 & C5 -> E4
            (6, 6, 7, "等腰三角形"),
            (6, 7, 6, "等腰三角形"),
            (7, 6, 6, "等腰三角形"),

            # 路径：C1 & C2 & C3 & not C4 & not C5 -> E5
            (5, 6, 7, "一般三角形"),
        ],
    )
    def test_cause_effect_core(self, a, b, c, expected):
        assert classify_triangle4(a, b, c) == expected

    @pytest.mark.parametrize(
        "a,b,c",
        [
            (2, 2, 4),  # not C3 & C5 -> 应为 E2
            (2, 5, 2),  # not C3 & C5 -> 应为 E2
        ],
    )
    def test_cause_effect_isosceles_not_triangle(self, a, b, c):
        assert classify_triangle4(a, b, c) == "非三角形"

    # 额外增强：一般三角形的排列不变性
    @pytest.mark.parametrize(
        "a,b,c",
        [
            (3, 4, 5),
            (4, 5, 3),
            (5, 3, 4),
        ],
        ids=["3-4-5", "4-5-3", "5-3-4"],
    )
    def test_permutation_invariance_general_triangle(self, a, b, c):
        assert classify_triangle4(a, b, c) == "一般三角形"

    # 仅依据题目文档的“正整数”与 src 行为进行测试，不引入额外规范假设
```

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 -o addopts=
"tests/test_triangle_program3.py" "tests/test_triangle_program4.py"
```

终端输出：

```
ST/Labs/Lab01-BlackBox [🐍 v3.13.11(st-labs-venv)]
❯ "/tmp/st-labs-venv/bin/python" -m pytest -o addopts= "tests/test_triangle_program3.py" "tests/test_triangle_program4.py"
============================================================================= test session starts ==============================================================================
platform linux -- Python 3.13.11, pytest-9.0.2, pluggy-1.6.0
rootdir: /home     :/Cloud/Nutstore/Sync/WorkSpace/ST/Labs/Lab01-BlackBox
configfile: pytest.ini
plugins: html-4.1.1, cov-7.0.0, metadata-3.1.1, mypy-1.0.1, timeout-2.4.0, xdist-3.8.0
collected 35 items

tests/test_triangle_program3.py ...................                                                                                                                    [ 54%]
tests/test_triangle_program4.py ................                                                                                                                       [100%]

============================================================================== 35 passed in 0.08s ==============================================================================
```

七、讨论与结论

1. 方法效果分析

- 判定表法适合"条件较多但取值有限"的逻辑，能系统枚举条件组合，避免凭经验漏掉某种组合。
- 因果图法在此基础上进一步突出原因–结果关系和约束条件，尤其适合检查"优先级"和"互斥关系"是否被正确实现。
- 与黑盒测试1中的经验性用例相比，本部分先建模、再设计用例的过程更有结构化，便于后续维护与扩展。

2. 实验反思与心得

- 建模阶段（画判定表 / 因果图）虽然比直接写用例稍费时间，但能帮助理清业务规则，减少"想到什么测什么"的随意性。
- 对于本实验这样的逻辑程序，判定表法就已经足够；在规则较多、约束复杂时，再引入因果图法会更直观。
- 结合 pytest 的参数化功能，可以直接把判定表或因果图中的规则翻译成测试数据，避免手写大量重复测试函数。

3. 结论

本实验通过对 Program 3 与 Program 4 的测试实践，表明判定表法和因果图法能够有效地指导测试用例设计，帮助验证实现是否完全覆盖规格要求，并在存在缺陷时提供清晰的定位线索。与单纯靠经验选取测试数据相比，基于模型的黑盒测试方法更有利于系统性地发现和分析逻辑错误。