Black Box Testing - PE & AVL

Objective

Through a simple example, this exercise illustrates how to apply **Equivalence Partitioning (EP)** and **Boundary Value Analysis (BVA)** techniques, and how to propose the corresponding test cases.

Statement

Using the black box testing technique, design a test plan for the following function:

```
def isPassed(grade: Decimal) -> bool:
    ...
```

The function evaluates a grade within the range **0 to 10** and determines whether the result is **PASS** or **FAIL**.

The following tasks must be completed:

1. Identify and define the **equivalence partitions**
2. Determine the **boundary values** for input data
3. Propose and implement the corresponding **test cases**

Equivalence Partitions (EP)

For the input parameter `grade`, the following equivalence classes can be identified based on the function's expected behavior:

Partition	Input Range / Type	Expected Result
P1	`grade < 0` (e.g. `-5`, `-1`)	Invalid input → raises `ValueError`
P2	`0 ≤ grade < 5` (e.g. `0`, `2`)	Valid input → returns `False` (Fail)
P3	`5 ≤ grade ≤ 10` (e.g. `5`, `7.3`, `10`)	Valid input → returns `True` (Pass)
P4	`grade > 10` (e.g. `11`, `15`)	Invalid input → raises `ValueError`
P5	Non-numeric values (e.g. `None`, `"seven"`)	Invalid input → raises `ValueError`

These partitions ensure that all types of valid and invalid inputs are tested: - both sides of the numeric limits (`0` and `10`), - different numeric categories (fail, pass), - and data type validation.

Boundary Value Analysis (BVA)

Based on the valid domain `[0, 10]`, the **critical boundary values** tested are:

- **Lower boundary:** `-1` (just below 0) and `0` (lowest valid grade)
- **Passing threshold:** `5` (limit between fail and pass)
- **Upper boundary:** `10` (highest valid grade) and `11` (just above 10)

These values correspond directly to the edge cases implemented in the test suite:

- `-1` and `11` → expected `ValueError` (invalid range)
- `0` → valid but not passed (`False`)
- `5` → valid and passed (`True`)
- `10` → valid and passed (`True`)

This selection guarantees that the function is evaluated at all decision boundaries and transitions between equivalence partitions.

Implementation

Function Under Test

The following function implements the logic to evaluate a numeric grade and determine if it represents a passing score.

```
from decimal import Decimal

def isPassed(grade: Decimal) -> bool:
    if not isinstance(grade, (int, float, Decimal)):
        raise ValueError("The grade must be a number")

    grade = Decimal(grade)
    if grade < 0 or grade > 10:
        raise ValueError("The grade must be between 0 and 10")

    return grade >= 5
```

Explanation:

- The function first ensures that the input is numeric.
- Then, it validates that the grade lies within the range `[0, 10]`.
- Finally, it returns `True` if the grade is greater than or equal to 5, otherwise `False`.

Test Cases

The tests cover all equivalence partitions and boundary values identified above. They are implemented using Python's built-in `unittest` module.

```
import unittest
from decimal import Decimal
from main import isPassed

class TestIsPassedValid(unittest.TestCase):
    def test_failed_grades(self):
        self.assertEqual(isPassed(Decimal("2")), False)

    def test_passed_grades(self):
        self.assertEqual(isPassed(Decimal("7.3")), True)

class TestIsPassedInvalid(unittest.TestCase):
    def test_below_range(self):
        with self.assertRaises(ValueError):
            isPassed(Decimal("-5"))

    def test_above_range(self):
        with self.assertRaises(ValueError):
            isPassed(Decimal("15"))

    def test_non_numeric_value(self):
        with self.assertRaises(ValueError):
            isPassed(None)
        with self.assertRaises(ValueError):
            isPassed("seven")

class TestIsPassedBoundary(unittest.TestCase):
    def test_minimum_valid(self):
        self.assertEqual(isPassed(Decimal("0")), False)

    def test_maximum_valid(self):
        self.assertEqual(isPassed(Decimal("10")), True)

    def test_passing_valid(self):
        self.assertEqual(isPassed(Decimal("5")), True)

    def test_below_minimum_invalid(self):
        with self.assertRaises(ValueError):
            isPassed(Decimal("-1"))

    def test_above_maximum_invalid(self):
        with self.assertRaises(ValueError):
            isPassed(Decimal("11"))

if __name__ == '__main__':
```

```
    unittest.main(verbosity=2)
```

Conclusions

The combination of **Equivalence Partitioning** and **Boundary Value Analysis** ensures that:

- The function is tested with both valid and invalid inputs.
- The behavior at critical boundaries (0, 5, 10) is verified.
- Errors are correctly handled when non-numeric or out-of-range values are provided.