ANSWERS

A14.5 Testing strategies

Software development, law and ethics · A level · OCR H446 1.2.3, AQA 7517 4.13.1.4, Eduqas A500QS 1.7 · about 45 min

BugBotLab

What this lesson is about

Black box and white box testing, alpha, beta and acceptance testing, destructive testing, and a black box test plan that finds two bugs.

- What testing is for
- Test data at A level
- Black box testing
- White box testing
- Who tests, and when
- Destructive testing
- A test plan

Questions

6 marks in all

1. What is black box testing? [1 mark]

- ☐ **A** Designing tests from the specification, without looking at the code
- ☐ **B** Testing every path through the code
- ☐ **C** Testing by the client before they accept the system
- ☐ **D** Testing that deliberately tries to break the hardware

> **Answer:** A. Black box tests treat the system as inputs and outputs; white box tests are designed from the code.

2. A game is released to a limited group of players outside the company to find problems before the full launch. What is this? [1 mark]

- ☐ **A** Beta testing
- ☐ **B** Alpha testing
- ☐ **C** Acceptance testing
- ☐ **D** White box testing

> **Answer:** A. Beta testing uses real users outside the company; alpha testing is in-house, before release.

3. A speed is allowed from 10 up to but not including 40. Which set is boundary data? [1 mark]

- ☐ **A** 9, 10, 39 and 40
- ☐ **B** 25 and 30
- ☐ **C** "far" and -5
- ☐ **D** 0 and 100

> **Answer:** A. Boundary data tests each edge and the value just either side of it.

4. What is the purpose of acceptance testing? [1 mark]

- ☐ **A** For the client to check the system meets the agreed requirements before accepting it
- ☐ **B** For programmers to test each module as it is written
- ☐ **C** For testers to try to crash the system
- ☐ **D** To check the code compiles

> **Answer:** A. Acceptance testing is done by the client or end users against the requirements, often as a contract sign-off.

5. What does this white box coverage check print? [1 mark]

```
ran = set()
def speed_for(gap):
    if gap < 10:
        ran.add("stop")
        return 0
    elif gap < 40:
        ran.add("slow")
        return 30
    ran.add("fast")
    return 60
for gap in [5, 20, 30]:
    speed_for(gap)
print(len(ran), sorted(ran))
```

> **Answer:**
>
> ```
> 2 ['slow', 'stop']
> ```
>
> 5 runs the stop branch and 20 and 30 both run the slow branch, so the fast branch is never tested.

6. Why is testing described as showing the presence of errors but not their absence? [1 mark]

- ☐ **A** No set of tests can try every possible input, so an untested input could still fail
- ☐ **B** Tests always contain errors themselves
- ☐ **C** Errors cannot be found by testing
- ☐ **D** Only syntax errors can be found by testing

> **Answer:** A. Test data is chosen to be the inputs most likely to reveal errors, but passing tests cannot prove the program correct.

The task: test the black box

A teammate has written `speed_for(gap)`. Its specification is: - `gap` is a distance in cm, and should be an `int` or a `float`. - If `gap` is not a number, or is less than 0, it must raise `ValueError`. - Otherwise it returns `0` if `gap` is less than 10, `30` if `gap` is from 10 up to but not including 40, and `60` if `gap` is 40 or more. Do not change `speed_for`: your job is to test it from the specification. Make a list `tests` of tuples `(value, expected, kind)`, where `expected` is the number the specification says, or the string `"error"` if it should raise `ValueError`, and `kind` is `"normal"`, `"boundary"` or `"erroneous"`. Include at least two normal tests, the four boundary values `9`, `10`, `39` and `40`, and the erroneous values `"far"` and `-5`. Run each test, treating a `ValueError` as the result `"error"`. For each, print `<value!r> <kind>: pass` if the result matches, or `<value!r> <kind>: FAIL (got <result>)` if it does not, where `<value!r>` is the value as `repr` shows it (so the string appears as `'far'`). At the end print `failed: <n>`. The robot stays still.

```
# the two lines every program starts with: the commands, then the robot
from bugbot import *
connect()

def speed_for(gap):
    if not isinstance(gap, (int, float)):
        raise ValueError("gap must be a number")
    if gap < 10:
        return 0
    elif gap <= 40:
        return 30
    else:
        return 60

tests = []
```

The hint students can ask for: Write the expected result for each test from the specification alone, before you look at the code. The boundaries are where the result changes, so test the value on each side of each change. For an erroneous test the expected result is an error, so a test passes if calling the function raises one.

A solution

```
# the two lines every program starts with: the commands, then the robot
from bugbot import *
connect()

def speed_for(gap):
    if not isinstance(gap, (int, float)):
        raise ValueError("gap must be a number")
    if gap < 10:
        return 0
    elif gap <= 40:
        return 30
    else:
        return 60

tests = [(25, 30, "normal"), (70, 60, "normal"), (9, 0, "boundary"), (10, 30, "boundary"),
         (39, 30, "boundary"), (40, 60, "boundary"), ("far", "error", "erroneous"), (-5,
"error", "erroneous")]

failed = 0
for value, expected, kind in tests:
    try:
```

```
        actual = speed_for(value)
    except ValueError:
        actual = "error"
    if actual == expected:
        print(f"{value!r} {kind}: pass")
    else:
        print(f"{value!r} {kind}: FAIL (got {actual})")
        failed += 1
print("failed:", failed)
```

Any program that meets the task's checks is marked correct in the simulator; this is one way, not the only way.