ANSWERS

A1.10 Project: the behaviour controller

Programming techniques and object-oriented programming · A level · OCR H446 1.2.4, AQA 7517 4.1.1.2, Eduqas A500QS 1.4 · about 35 min

What this lesson is about

Get past a wall to the dock with prioritised behaviour classes, a controller that aggregates them, and an exception to finish.

- The job
- The design
- Where each idea comes in
- Build it in this order

Questions

5 marks in all

1. The controller is given the list [Dock(), Sidestep(), Cruise()]. Sidestep and Cruise both return True from wants(). Which acts? [1 mark]

- ☐ **A** Sidestep, because it comes first in the priority order
- ☐ **B** Cruise, because it is last
- ☐ **C** Both, one after the other
- ☐ **D** Neither, because two behaviours want to act

> **Answer:** A. step() stops at the first behaviour whose wants() is True.

2. Behaviour.wants() raises NotImplementedError. What is the point of that? [1 mark]

- ☐ **A** It makes Behaviour abstract in practice: a subclass that forgets to override wants() fails loudly
- ☐ **B** It stops Cruise from working
- ☐ **C** It is how Python calls the constructor
- ☐ **D** It catches errors in subclasses

> **Answer:** A. The base class defines the interface; each subclass must supply the body.

3. Why does the controller stop the motors in a finally block? [1 mark]

- ☐ **A** So they are stopped however the run ends: docked, given up or crashed
- ☐ **B** Because finally runs faster than except
- ☐ **C** Because stop() raises an exception
- ☐ **D** So the robot docks sooner

> **Answer:** A. finally runs after success, a caught exception or an uncaught one.

4. What does this program print? [1 mark]

```
class Stop(Exception):
    pass

class Step:
    def __init__(self, n):
        self.n = n
    def act(self):
        if self.n == 3:
            raise Stop()
        print("step", self.n)

try:
    for s in [Step(1), Step(2), Step(3), Step(4)]:
        s.act()
except Stop:
    print("stopped")
finally:
    print("done")
```

Answer:

```
step 1
step 2
stopped
done
```

Step 3 raises Stop, which leaves the loop, so step 4 never acts; except then finally run.

5. Which object-oriented ideas does the controller's step() method rely on? [1 mark]

Tick every answer that is true.

- ☐ **A** Polymorphism: it calls wants() and act() without knowing each object's class
- ☐ **B** Aggregation: the behaviours were created outside and handed to the controller
- ☐ **C** Encapsulation: the step count is a private attribute
- ☐ **D** Multiple inheritance: each behaviour has two superclasses

Answer: A, B, C. Each behaviour has only one superclass, Behaviour.

The task: the behaviour controller

The robot starts at the bottom left, facing north (heading 0). The starter declares the constants `SAFE_CM = 25`, `STEP_CM = 10` and `DOCK_Y = 60`, the exception class `Docked` and the abstract class `Behaviour`. Write: - `Cruise(Behaviour)`: name `cruise`. `wants()` returns `True` when `distance()` is more than `SAFE_CM`. `act()` drives forward `STEP_CM` cm. - `Sidestep(Behaviour)`: name `sidestep`. `wants()` returns `True` when `distance()` is `SAFE_CM` or less. `act()` moves sideways to the right, with `right`, by `STEP_CM + 5` cm. - `Dock(Behaviour)`: name `dock`. `wants()` returns `True` when the y part of `position()` (cm north of the start) is `DOCK_Y` or more. `act()` turns the LED green, plays one note, and raises `Docked`. - `Controller`: its constructor takes a list of behaviours in priority order and keeps them, and a step count starting at 0, in private attributes. `step()` goes through the behaviours in order, and for the **first** one whose `wants()` is `True` it adds 1 to the count, prints `step <n>: <name>` and calls its `act()`, then returns. `run(max_steps)` calls `step()` up to `max_steps` times inside a `try` block. It catches `Docked` and prints `docked after <n> steps`, and in a `finally` block calls `stop()` and prints `controller stopped`. The main program makes `Controller([Dock(), Sidestep(), Cruise()])` and calls `run(30)`. Do not use `global` or `isinstance`.

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

SAFE_CM = 25
STEP_CM = 10
DOCK_Y = 60

class Docked(Exception):
    pass

class Behaviour:
    def __init__(self, name):
        self.name = name

    def wants(self):
        raise NotImplementedError

    def act(self):
        raise NotImplementedError

# write Cruise, Sidestep, Dock and Controller

forward(50, distance=STEP_CM)
```

The hint students can ask for: Get the base class and one behaviour working before the controller. The controller's job each step is the same: go through the behaviours in priority order and let the first one that wants to act do so. Docking ends the run by raising an exception, so the controller needs to catch it, and something must stop the motors however the loop ends.

A solution

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

SAFE_CM = 25
STEP_CM = 10
```

```
DOCK_Y = 60

class Docked(Exception):
    pass

class Behaviour:
    def __init__(self, name):
        self.name = name

    def wants(self):
        raise NotImplementedError

    def act(self):
        raise NotImplementedError

class Cruise(Behaviour):
    def __init__(self):
        super().__init__("cruise")

    def wants(self):
        return distance() > SAFE_CM

    def act(self):
        forward(50, distance=STEP_CM)

class Sidestep(Behaviour):
    def __init__(self):
        super().__init__("sidestep")

    def wants(self):
        return distance() <= SAFE_CM

    def act(self):
        right(50, distance=STEP_CM + 5)

class Dock(Behaviour):
    def __init__(self):
        super().__init__("dock")

    def wants(self):
        x, y = position()
        return y >= DOCK_Y

    def act(self):
        led("green")
        tone(880, 0.3)
        raise Docked()

class Controller:
    def __init__(self, behaviours):
        self.__behaviours = behaviours
        self.__steps = 0

    def step(self):
        for behaviour in self.__behaviours:
            if behaviour.wants():
                self.__steps = self.__steps + 1
                print(f"step {self.__steps}: {behaviour.name}")
                behaviour.act()
```

```
                return

    def run(self, max_steps):
        try:
            for i in range(max_steps):
                self.step()
            print("gave up")
        except Docked:
            print(f"docked after {self.__steps} steps")
        finally:
            stop()
            print("controller stopped")

controller = Controller([Dock(), Sidestep(), Cruise()])
controller.run(30)
```

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