ANSWERS

A1.9 Aggregation, composition and class diagrams

Programming techniques and object-oriented programming · A level · OCR H446 1.2.4, AQA 7517 4.1.2.3, Eduqas A500QS 1.4 · about 25 min

What this lesson is about

Has-a relationships, drawing them on class diagrams, and the design principles that favour composition over inheritance.

- Association, aggregation and composition
- Class diagrams for "has a"
- Design principles

Questions

5 marks in all

1. A Robot creates its own Wheel objects in its constructor, and they are destroyed with it. What is this relationship? [1 mark]

- ☐ **A** Composition
- ☐ **B** Aggregation
- ☐ **C** Inheritance
- ☐ **D** Polymorphism

> **Answer:** A. The parts are owned by the whole and cannot exist without it.

2. In a class diagram, which symbol shows aggregation? [1 mark]

- ☐ **A** A hollow diamond at the whole's end of the line
- ☐ **B** A filled diamond at the whole's end of the line
- ☐ **C** A hollow triangle at the superclass end
- ☐ **D** A dashed arrow

> **Answer:** A. Hollow diamond for aggregation, filled diamond for composition, hollow triangle for inheritance.

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

```
class Route:
    def __init__(self, legs):
        self.legs = legs

class Rover:
    def __init__(self, route):
        self.route = route

shared = Route([20, 30])
a = Rover(shared)
b = Rover(shared)
a.route.legs.append(10)
del a
print(len(b.route.legs), sum(shared.legs))
```

Answer:

```
3 60
```

Both rovers aggregate the same Route object, so the leg added through a is seen through b and shared, and it survives del a.

4. Why is 'favour composition over inheritance' a design principle? [1 mark]

- ☐ **A** Parts can be combined and swapped while the program runs, without a subclass for every combination
- ☐ **B** Composition makes programs run faster
- ☐ **C** Inheritance is not allowed in most languages
- ☐ **D** Composition removes the need for methods

Answer: A. Inheritance is fixed when the code is written; composition chooses parts at run time.

5. What does 'program to interfaces, not implementation' mean? [1 mark]

- ☐ **A** Write code that relies only on the methods an object provides, not on which class it is or how it works
- ☐ **B** Always write the user interface first
- ☐ **C** Put all code inside one class
- ☐ **D** Never use private attributes

Answer: A. Code that only needs move() works with any class that provides move().

The task: a rover made of parts

Write three classes from the class diagram above. - `Buzzer` has one method, `beep(self, note)`, which plays `note` Hz for 0.2 seconds. - `Route` has a constructor `__init__(self, legs)` that stores `legs`, a list of `(cm, angle)` tuples, in a private attribute; `get_legs(self)` returns the list; `length(self)` returns the total of the cm parts, worked out from the legs. - `Rover` has a constructor `__init__(self, name, route)` that stores the name, **creates its own** `Buzzer()` in an attribute (composition), and stores the `route` it was given (aggregation). Its method `run(self)` drives each leg in turn, forward `cm` then turn right `angle` degrees, then beeps once at 880 Hz and prints `<name> drove <cm> cm`, using the route's `length()`. In the main program make `route = Route([(20, 90), (20, 90), (20, 180)])` and `rover = Rover("scout", route)`, and call `rover.run()`. Then `del rover`, and print `route still has <cm> cm` using `route.length()`. Do not type the length.

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

class Buzzer:
    def beep(self, note):
        tone(note, 0.2)

route = [(20, 90), (20, 90), (20, 180)]
```

The hint students can ask for: Decide which part belongs to the rover alone and which one it only uses. The part it owns is created inside its constructor; the part it uses is made outside and handed in. The route's length is the sum of its legs, so the Route class can work it out.

A solution

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

class Buzzer:
    def beep(self, note):
        tone(note, 0.2)

class Route:
    def __init__(self, legs):
        self.__legs = legs

    def get_legs(self):
        return self.__legs

    def length(self):
        total = 0
        for cm, angle in self.__legs:
            total = total + cm
        return total

class Rover:
    def __init__(self, name, route):
        self.__name = name
        self.__buzzer = Buzzer()
        self.__route = route
```

```
    def run(self):
        for cm, angle in self.__route.get_legs():
            forward(50, distance=cm)
            turn_right(30, angle=angle)
        self.__buzzer.beep(880)
        print(self.__name, "drove", self.__route.length(), "cm")

route = Route([(20, 90), (20, 90), (20, 180)])
rover = Rover("scout", route)
rover.run()
del rover
print("route still has", route.length(), "cm")
```

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