Labs

Set 11

DM562 Scientific Programming
DM857 Introduction to Programming
DS830 Introduction to Programming

1 Inheritance and Composition

A team developing a project realised that they need to represent circles and polygons in a 2D plane.

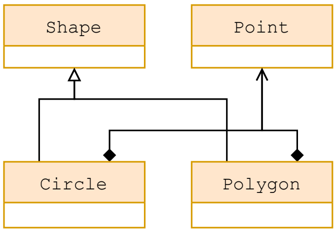

Figure 1: Class diagram for module `geometry_2D`.

1. Initially, the team decide that representing shapes with a fixed position is enough for the project and designed the class hierarchy in Figure 1.

   (a) Define a class `Point` whose instances represent points in the (Euclidean) plane. Decide which attributes this class should have and which getters and setters should be available for these attributes. This class should also provide the following methods.

   i. A constructor that creates a point given its coordinates.

   ii. A method `is_origin(self) -> bool` for testing whether this point corresponds to the origin of the coordinate system (i.e., whether its coordinates are `(0,0)`).

   iii. A method `distance_to(self,other:Point) -> float` that returns the distance[1] between this point and another.

   iv. A method `distance_to_origin(self) -> float` that returns the distance between this point and the origin of the coordinate system.

   v. A method `__repr__(self) -> str` that returns a textual representations of this point in Python-like syntax (e.g., `'Point(x,y)'` where `x` and `y` are the coordinates of the point).

   vi. A method `__str__(self) -> str` that returns a textual representations of this point in the format `'(x,y)'` where `x` and `y` are the coordinates of the point.

   vii. Methods `__eq__(self, other) -> bool`, `__lt__(self, other) -> bool`, and `__le__(self, other) -> bool` for comparing this point to another using the distance from the origin for ordering (e.g., $(-1,-1) < (2,2) < (-3,3)$).

[1] The Euclidean distance between a point at (x_1, y_1) and a point at (x_2, y_2) is given by the formula $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$.

(b) Define a class `Shape` whose instances represent shapes in the plane. Decide which attributes this class should have and which getters and setters should be available for these attributes. This class should also provide a method `perimeter(self) -> float` that returns the perimeter of this shape.

(c) Define a class `Circle` whose instances represent circles in the plane (hence shapes, see Figure 1). Decide which attributes this class should have and which getters and setters should be available for these attributes. Decide which attributes and methods should be inherited from its superclass. This class should also provide a constructor that takes two arguments, a point representing the centre of the circle and a floating point number representing its radius.

(d) Define a class `Polygon` whose instances represent polygons in the plane (hence shapes, see Figure 1). Decide which attributes this class should have and which getters and setters should be available for these attributes. Decide which attributes and methods should be inherited from its superclass. This class should also provide a constructor that takes as an argument a list of points representing the vertices of the polygon.

2. As the project grows, the team concluded that they need to be able to move shapes.

   (a) Modify the implementation of class `Point` to provide:

      i. A method `translate(self,dx:float,dy:float)` for moving this point by `dx` horizontally and `dy` vertically.

      ii. A method `copy(self) -> Point` that returns a copy of this point.

   (b) Modify the implementation of class `Shape` to provide:

      i. A method `translate(self,dx:float,dy:float)` for moving this point by `dx` horizontally and `dy` vertically.

      ii. A method `copy(self) -> Shape` that returns a copy of this shape.

   (c) Propagate these changes the remaining classes and enforcing encapsulation (e.g., client code should not be able to move a single point of a polygon and alter its shape).

3. A team member remembered that in a previous project they developed a module for displaying objects that can draw themselves using arcs and line segments. This module is called `visualiser` (see the course material for this lab) and provides the following classes.

   - A class `Visualiser` that displays a scene composed of drawable objects in a window.
   - A class `Canvas` that provides methods for drawing arcs and line segments on a scene.
   - A class `Drawable` that offers a base for every drawable object.

   Class `Visualiser` offers the following methods.

   - A constructor that takes as an optional argument a list of drawable objects composing the scene (the list is copied by the constructor).
   - Methods `add(self,drawable:Drawable,refresh:bool=True)` and `remove(self, drawable:Drawable,refresh:bool=True)` for adding and removing `drawable` from the scene, optionally refreshing the scene (see next method).
   - A method `refresh(self,pause:float=0.01)` that refreshes the scene displayed in the window and then pauses for `pause` seconds (see method `pause`).
   - A method `pause(self,interval:float)` that pauses for `interval` seconds to allow the window to process pending interactions with the user e.g., mouse clicks.

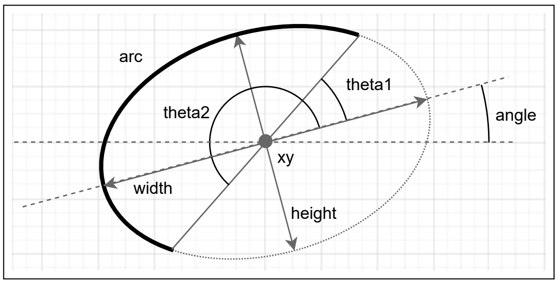

Figure 2: Example of elliptical arc.

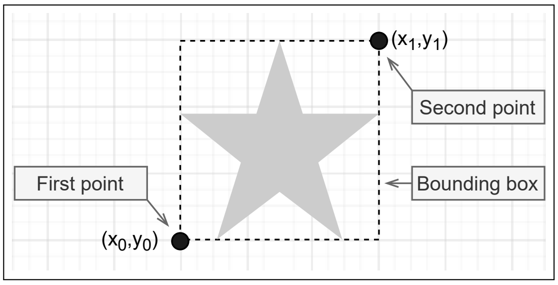

Figure 3: Example of bounding box for a shape.

- A method `wait_close(self)` that pauses until the window of this visualiser is closed.

Class `Canvas` offers the following methods.

- A method `draw_line(self,p1:Tuple[float,float],p2:Tuple[float,float])` that draws a line segment connecting `p1` and `p2`.
- A method `draw_arc(self,xy,width,height,angle=0.0,theta1=0.0,theta2=360.0)` that draws a segment of an ellipse. The ellipse is centred at `xy:Tuple[float,float]`, has axes with length `width:float` and `height:float` respectively, is rotated by `angle:float`. The arc starts at the angle `theta1` and ends at angle `theta2` (both relative to `angle`) as shown in Figure 2. For instance, `draw_arc(0,0,1,1)` draws a circle centred in the origin and with radius 1.

Class `Drawable` offers the following methods (whose implementation is delegated to the inheritors).

- A method `bounding_box(self)->Tuple[Tuple[float,float],Tuple[float,float]]` that returns an pair of points (as pairs of `float` values) corresponding to the opposing corners of the smallest rectangle enclosing the drawing (every side of the rectangle touches the shape) as exemplified in Figure 3. This method is called by `Visualiser` to retrieve the area of the scene where this object intends to draw itself.
- A method `draw(self,canvas:Canvas)` that draws this object using `canvas`.

Realising that this could help in testing and debugging, the rest of the team decided to integrate this module. To this end, modify your implementation of the shapes to extend `Drawable` (classes may have multiple superclasses). Below is a short program to test your implementation.

```python
import math
from geometry_2D import Point, Circle, Polygon
from visualiser import Visualiser
vis = Visualiser()
# add a circle at 0,0 with radius 1
vis.add( Circle(Point(0, 0), 1) )
# add some regular polygons inscribed in this circle
for sides in range(3,8):
  a = math.radians(360 / sides)
  vis.add( Polygon([ Point( math.cos(a * i), math.sin(a * i) )
                     for i in range(sides) ]) )
vis.wait_close()
```