Lehrstuhl für Computergestützte Modellierung und Simulation
TUM School of Engineering and Design
Technical University of Munich

Professional Software Engineering

Assignment 1
Submission Deadline: 13th November

1 Background

An implicitly-modelled geometry (or surface) is a geometry that can generally be modelled by eq. (1), *i.e.* the geometry (surfaces, volume etc.) is not explicitly defined; we are only representing the boundary surface (or line) by eq. (1).

$$F(x_i) = 0 \tag{1}$$

We can however determine whether a point in space (x_i) is inside the geometry ($x_i \in geometry$) by plugging x_i into eq. (1), which should only hold for points on the boundary of the geometry, with points outside and inside the geometry resulting in negative and positive $F(x_i)$ values, respectively.

$$r - ||x_i||_2 = r - \sqrt{x^2 + y^2} = 0 \tag{2}$$

An example implementation of an implicitly-modelled circle is shown below. Here, $F(x_i)$ represents the boundary of a circle (eq. (2), where r is the circle's radius). Keep in mind that this implementation does not take the origin of the circle into account and will only work for $x_{i/center} = [0, 0]$.

```
class Circle {
    public Radius {get; set;}
    // ... remaining properties and constructors
    public bool InsideOfCircle(double x , double y) {
        return Math.Sqrt (x * x + y * y ) <= Radius; // only works for
origin = 0,0
    }
  } // Circle

class MainClass {
    public static void Main(string[ ] argArray) {
        Circle circle = new Circle(5 , 0 , 0); //radius = 5m, origin is 0,0
        Console.Write(circle.InsideOfCircle(3 , 4)); // true
        Console.Write(circle.InsideOfCircle(2 , 3)); // true
        Console.Write(circle.InsideOfCircle(5 , 7)); // false
    }
  } // MainClass
```

Numerically, we can construct this circle by creating a two-dimensional grid, e.g. $x, y \in [0, 2]$ meters, and checking which points in the grid are inside the circle. The higher our grid resolution is, the closer our "numerical" circle will be to the actual circle.

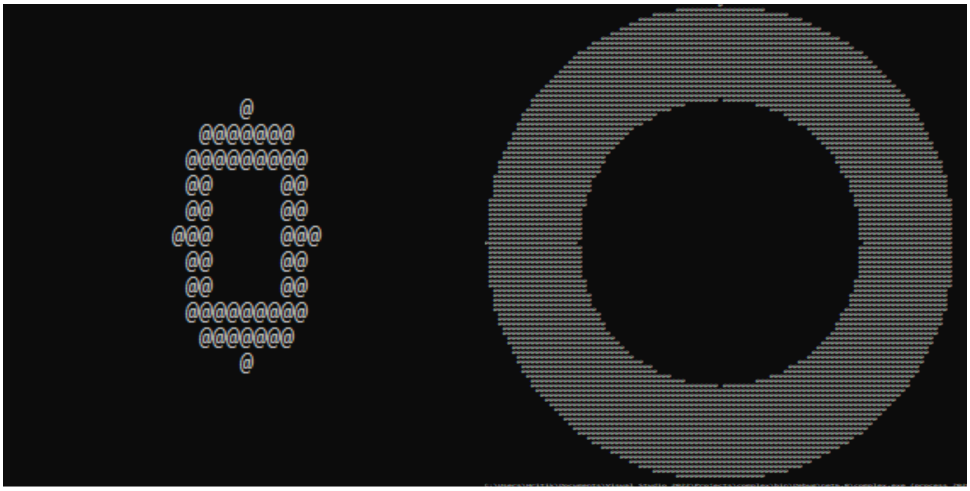
The same idea can be expanded to construct more complex geometries. Here, successive usage of union and intersection operators on primitive geometries (circles, rectangles etc.) can be used to model most geometries. A ring for instance, can be modelled as the difference[1] of two circle objects as shown below[2], and a union of five rings (with different origins) can represent the Olympic symbol.

[1] You can find the relation between the intersection and difference operators in eq. (3).
[2] This is the implementation for one point in space and not a complete grid.

```
class Ring {
    public OuterCircle {get; set;}
    public InnerCircle {get; set;}
    // . . . remaining properties and constructors
    public bool InsideOfRing(double x , double y) {
        bool insideOuter = OuterCircle.InsideOfCircle(x , y);
        bool outsideInner = !InnerCircle.InsideOfCircle(x , y);
        return insideOuter && outsideInner;
    }
 } // Ring

class MainClass {
    public static void Main(string[ ] argArray) {
        double x, y; // point in space/you need to initialize both variables
        Circle circle1 = new Circle(5 , 0 , 0);
        Circle circle2 = new Circle(4 , 0 , 0);
        Ring ring = new Ring(circle1 , circle2);
        bool insideRing = ring.InsideOfRing(x , y);
    }
 } // MainClass
```

Creating a grid and applying the ideas discussed in the preceding paragraphs results in fig. 1.



(a) Coarse Grid *(b) Fine Grid*

Figure 1: Example of an implicitly modelled ring. Subfigures (a) and (b) present coarse and fine grids, respectively. The grid was looped over and the symbol "@" was printed for *InsideOfRing(x, y) == true*. A white space (" ") was printed otherwise.

2 Task

In this assignment, your task is to implement the class hierarchy presented in fig. 2. You can use the examples provided in section 1 as a starting point, where the class *Ring* can be thought of as a special type of the class *Difference* in fig. 2.

Once you finish implementing your class hierarchy, create a grid in your main method and use it to print out the TUM logo, *i.e.* the letters T, U and M (you can use your own dimensions). Refer to fig. 1 for a printing example.

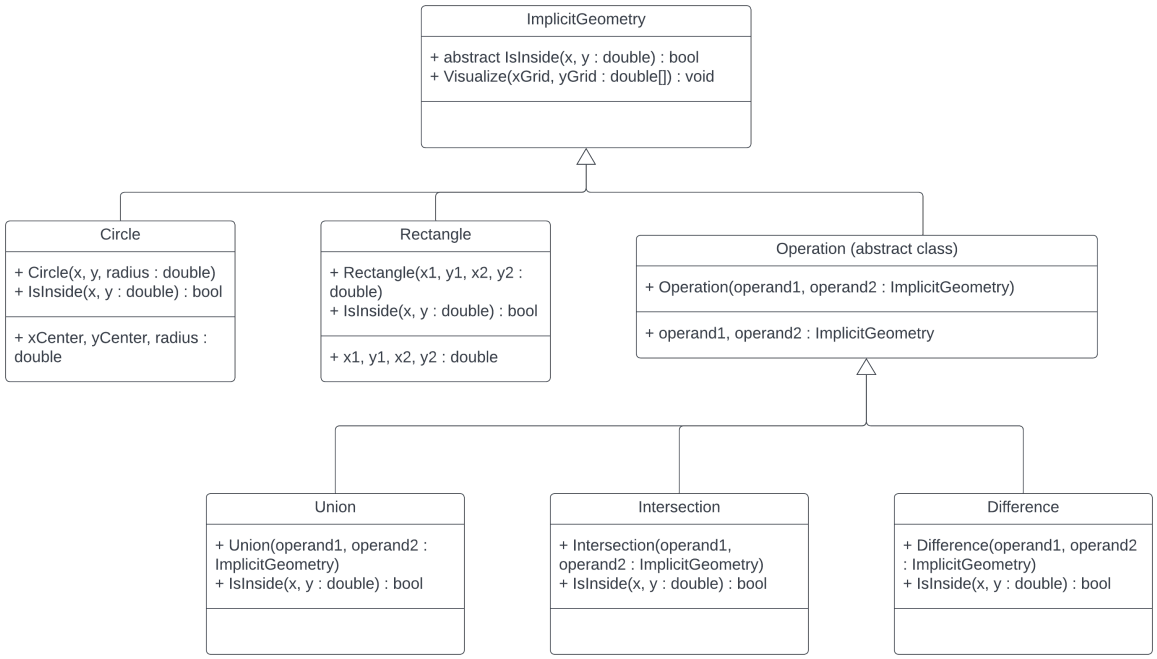


Figure 2: Class hierarchy structure, where each class block is divided into three sections: the name of the class (should not be changed), class methods, and class properties. Each arrow points towards the base class from the derived ones. **Note:** *ImplicitGeometry* and *Operation* are both abstract classes.

Some useful hints

- All the classes derived from *Operation* do not explicitly contain member properties as they inherit them from their base class. These properties still need to be initialized inside the derived class constructors using the following syntax:
public DerivedClassConstructor(DataType1 arg1, DataType2 arg2 . . .) : base(arg1, arg2 . . .) {}.

- The difference of two operands $(A - B)$ is defined using eq. (3), where B' is the inverse of B and $A \cap B$ is the intersection of the operands A and B.

$$A - B = A \cap B' \tag{3}$$

- Equation (4) summarizes the operations required to draw the letter M.

$$Rectangle1 \cup Rectangle2 \cap (Circle1 - Circle2) \cup Rectangle3 \tag{4}$$

- When you print out your implicit geometry, use *Console.Write()* and only start a new line for every row of grid (*Console.Write("\n")*). You should also keep in mind that you need to start from the last row and not the first one, otherwise, your implicit geometry will be flipped about the horizontal axis.