SOLID: Open Closed Principle

Learn about the Open Closed Principle and its implementation in real-world problems.

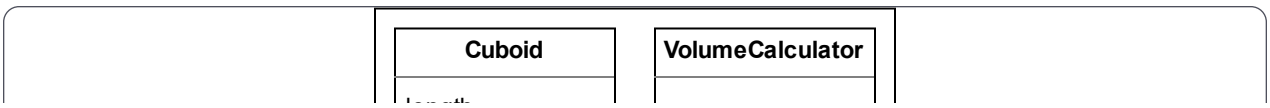
Introduction

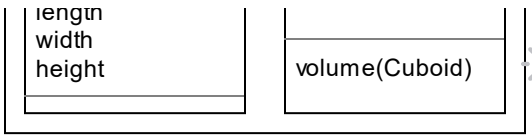
In 1988, Bertrand Meyer defined the **Open Closed Principle (OCP)** in the following way, “A software artifact should be open for extension but closed for modification.” This means that a system should improve easily by adding new code instead of changing the code core. This way, the core code always retains its unique identity, making it reusable.

One might think of OCP as inheritance, but remember that inheritance is only one of the OCP techniques. We use the interface because it is open for extension and closed for modification. Therefore, OCP is also defined as **polymorphic OCP.**

Example

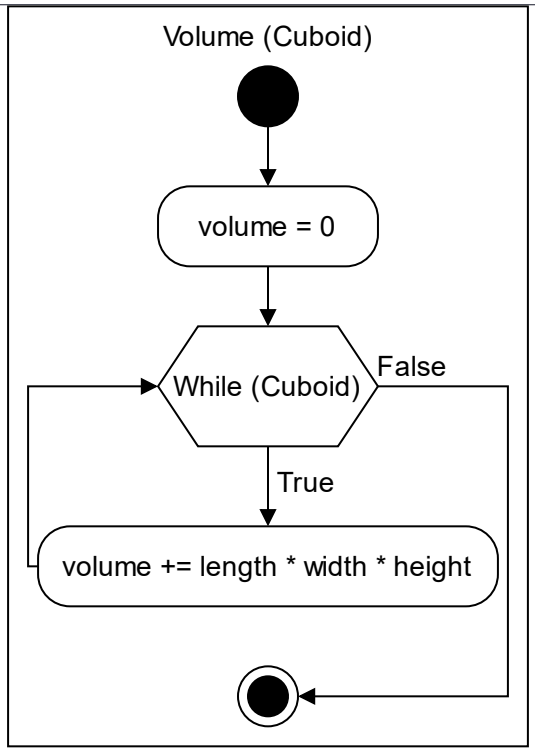
Suppose Alex had a cardboard business that sold boxes to its clients. We designed a class for calculating the volume of boxes. It takes the dimensions and calculates the volume of each box and adds it up to calculate the total volume of all boxes, as shown below.





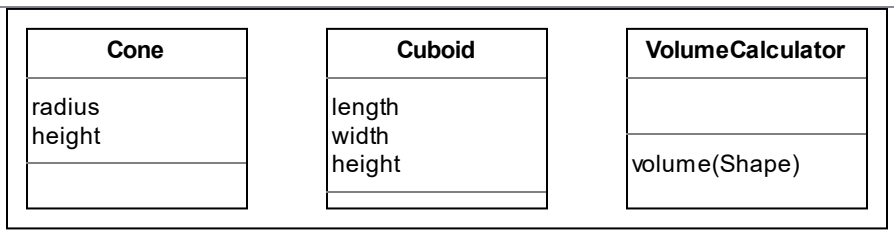
The volume calculator class

The algorithm for the `volume(Cuboid)` function is shown in the flowchart below.



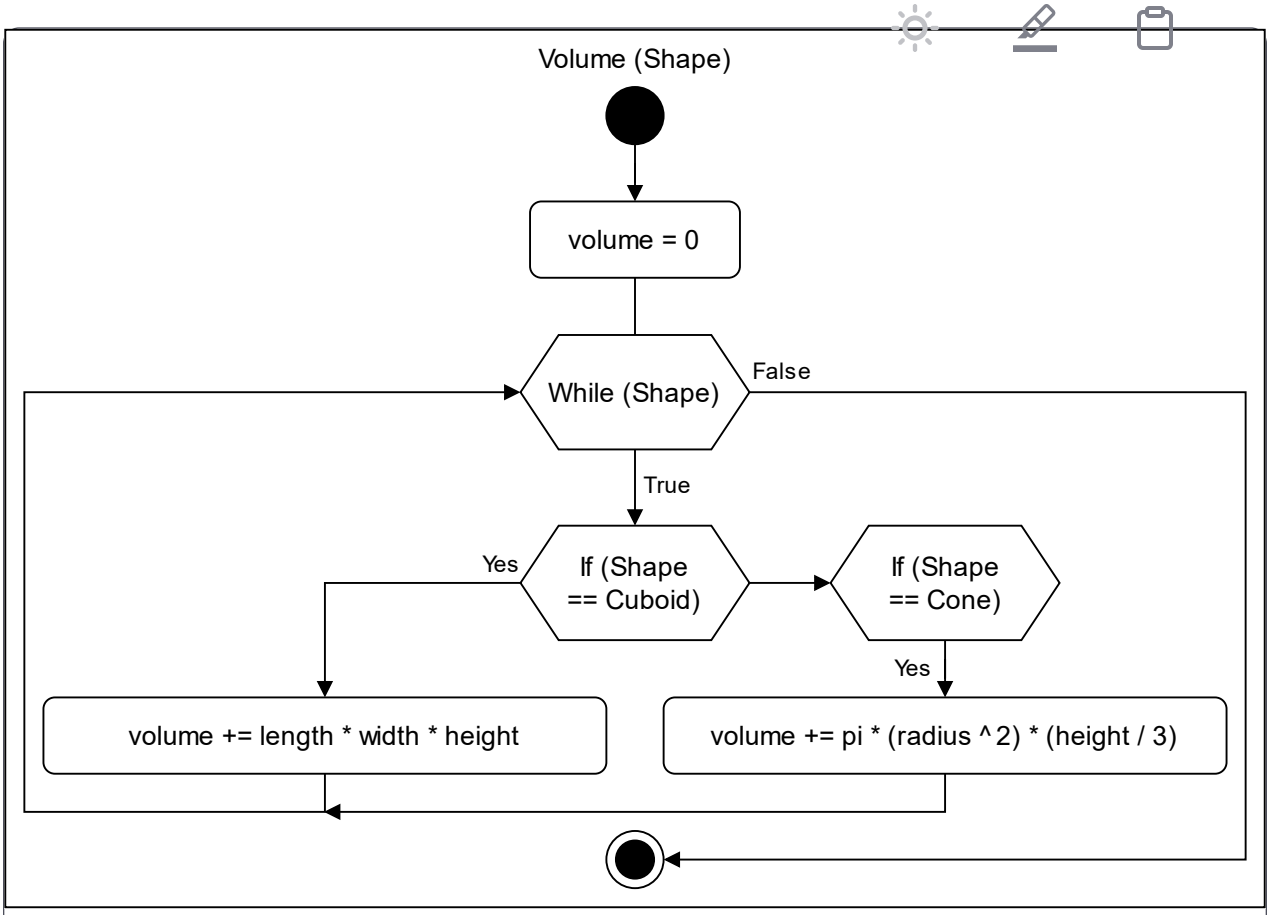
The volume(Cuboid) function

As the business grew, Alex also started selling cone-shaped boxes. To integrate the calculation of its volume, we need to make a `Cone` class and update the `volume()` function. See the updated classes below:



Add a Cone class and update the volume method

The algorithm for the `volume(Shape)` function is shown in the flowchart below.

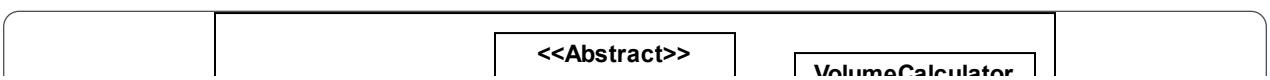


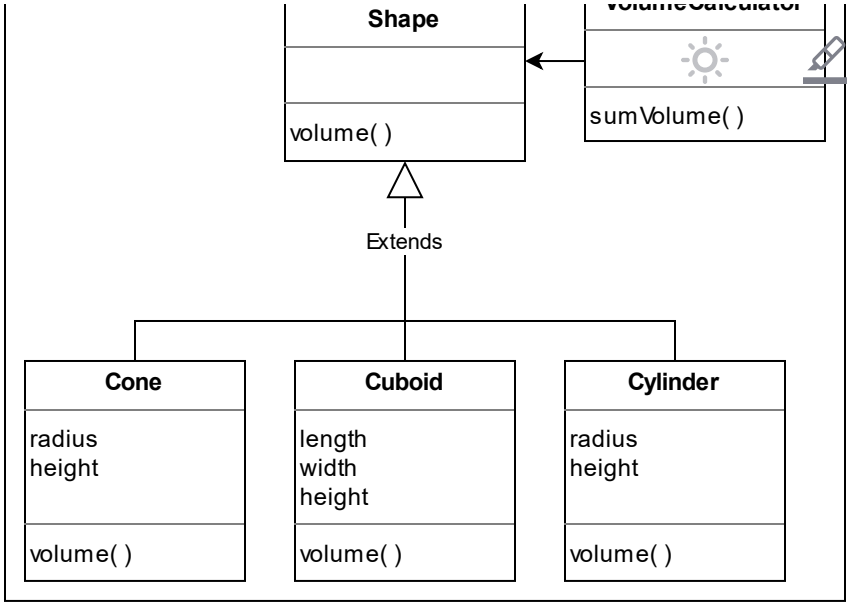
The volume (Shape) function

With only two types of boxes, the class structure looks fine, but what if Alex decides to deal with more types of boxes, e.g., a cylinder box? This will add complexity to the `volume(Shape)`. We will divide the code into segments using OCP to overcome this complexity.

Implementing the Open Closed Principle

We will make a parent class, `Shape`, which is an abstract class and has a `volume()` function, that is extended by its sub-classes, `Cuboid`, `Cylinder`, and `Cone`. These derived classes have their own `volume()` functions according to the shape. Then we have the `VolumeCalculator` class that only performs one task: adding the volume of all the boxes using the `sumVolume()` function.





The class diagram of VolumeCalculator according to OCP

Conclusion

We can conclude the OCP discussion as follows:

- A software system should be easy to extend without the need for modification in the existing system. For the software systems, this goal is achieved by OCP.
- The system must be divided into small components, which are arranged, so that core code is always protected from new code.

In the next lesson, we will talk about the Liskov Substitution Principle with its detailed explanation.

Back lesson
SOLID: Single Responsibility Principle

Completed
Next
SOLID: Liskov Substitution Principle