Name: Trac Duc Anh Luong

ID: 103488117

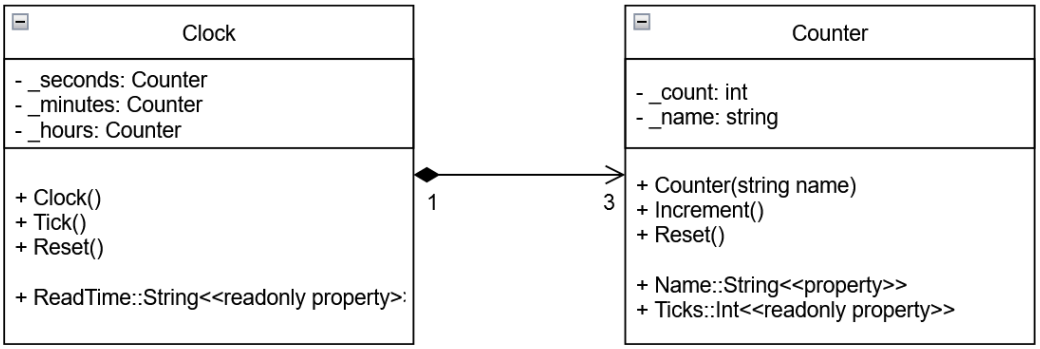
Object-Oriented Programming Principles

Encapsulation

Supposing we have a program. It has a few logically distinct objects that interact with one another following the program's established rules.

When each object maintains a private state inside of a class, encapsulation is achieved. This state is not directly accessible to other objects. Instead, users are limited to calling a set of methods, which are public functions.

Therefore, the object controls its own state using methods, and no other class is allowed to intervene unless specifically authorized. Use the techniques offered if you wish to interact with the object. However, you are unable to alter the status (by default).



For example, in the Clock program that we did in pass task 3.3, we have a Clock and a Counter class. The state of the Counter class is based on private variables _count and _name. It also has 3 methods of Counter(), Increment(), and Reset() which uses these variables. You cannot directly change the values which _count and _name hold, but you can call public methods like Increment() and Reset() in the main program to modify the internal state of the Counter class. This is an example of encapsulation.

Abstraction

You can consider abstraction as a logical progression from encapsulation.

Programs in object-oriented design are frequently enormous. Additionally, many items frequently converse with one another. Therefore, it is challenging to maintain a big codebase for years while making changes along the way.

A theory designed to solve this issue is an abstraction.

Each object should only offer a high-level usage method when abstraction is in place.

Internal implementation specifics should be concealed using this mechanism. Only those operations that are pertinent to the other objects should be revealed.

For example, a car can do many things and produce a lot of noise with its engine. However, you only need to learn how to use foot paddles, the gear stick, and some other simple components to drive a car.

This system should ideally be simple to use and change very infrequently throughout time. Imagine it as a tiny collection of public methods that any other class is free to use without having to "understand" how they operate.

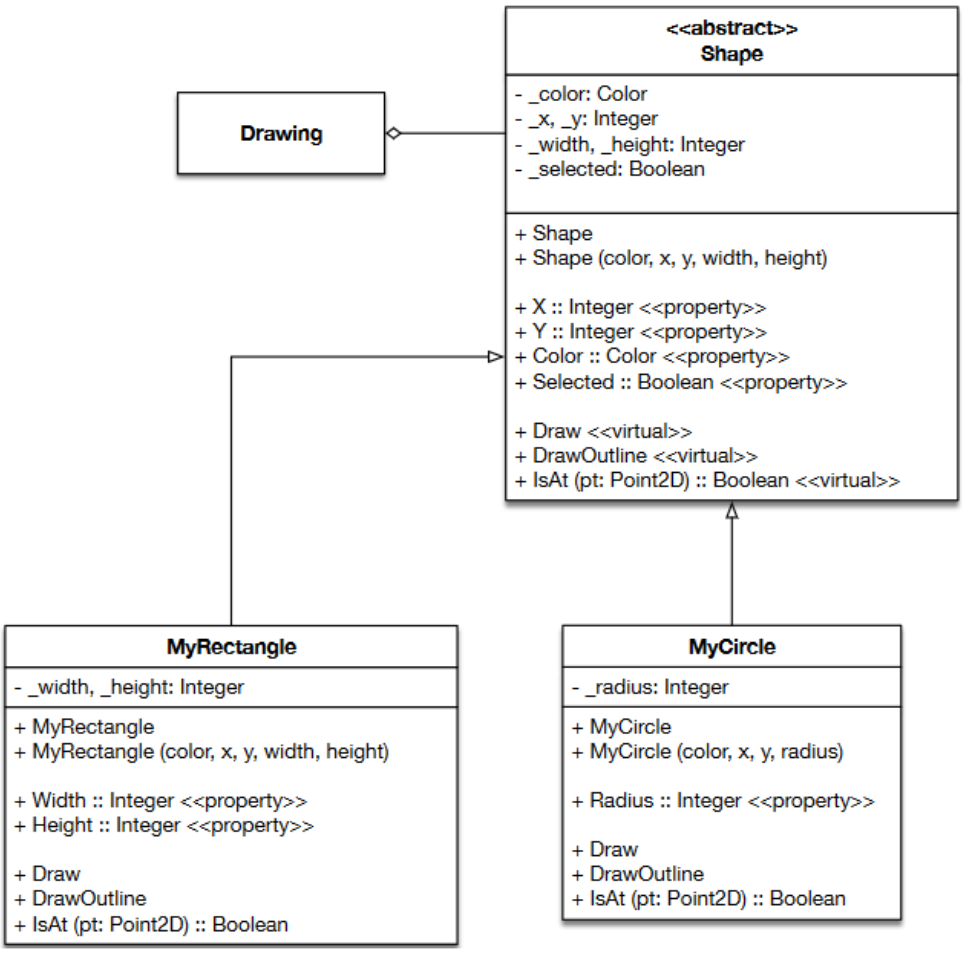
Inheritance

While encapsulation and abstraction can help us develop and maintain a big program with numerous code files, there are other common problems in OOP design. One of which is the similarity of objects, which share the same logic, but are not entirely identical.

A question occurs: “How do we reuse the similar logic and provide the unique logic into a separate class in the program?” This is when inheritance comes to use.

This principle means that you create a child class by deriving from a parent class. In this way, we can create a tree hierarchy. The child reuses all fields and methods of the parent class and can implement its unique features.

<The UML diagram is presented on the next page>



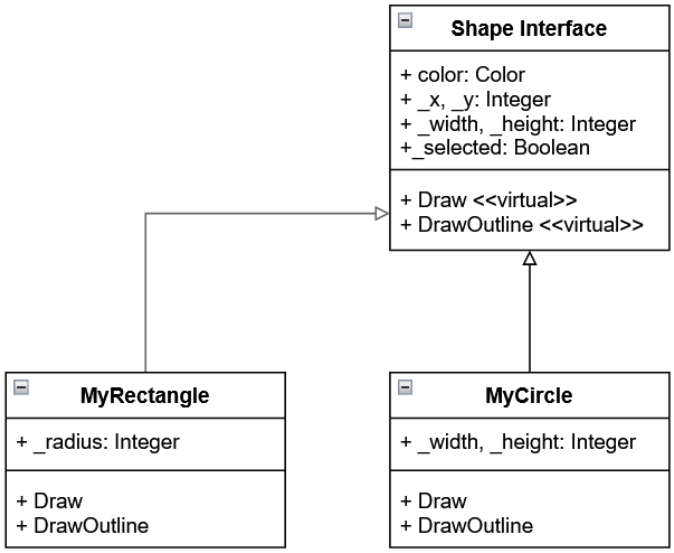
For example, in the Shape Drawer program that we did in pass task 4.1, MyRectangle and MyCircle are the children classes of Shape, which is a parent (abstract) class that provides basic properties like color, x, y, width, and height. The 2 children classes then override their parent to implement their unique features without having to change the common logic.

Polymorphism

This principle is related to inheritance in OOP design. If we have a parent class and some other children classes that inherit the properties from their parent, sometimes we want to use a collection (a list) that contain all of the parent and children classes. Or to be easier to understand, we have a method that is implemented in the parent class, and we would like to use it in the children classes as well. The answer to this is polymorphism.

Polymorphism provides an approach to using a child class exactly like its parent without mixing types, and the child will be able to keep its own methods. To do this we will need to define an interface for it to be reused. This parent will have many methods, and its children will implement their own versions of these methods.

Let’s simplify and make some adjustments to the UML diagram of the Shape Drawer program that we have used to make an example in the **Inheritance** section.



The two children classes MyRectangle and MyCircle inheriting the parent Shape Interface allow us to create a list of mixed rectangles and circles and treat them like the same type of object. If the list tries to draw the shape or draw the outline, the correspondent method will be found and executed. If it is a rectangle, the rectangle’s Draw() and DrawOutline() will be called. If it is a circle, the circle’s Draw() and DrawOutline() will be called. Using a function that interacts with a figure using its parameter, the redundancy of defining the shape twice - a rectangle, or a circle, will be removed. We can use Shape as an argument to pass a rectangle or a circle, as long as it uses the two given methods, its type does not matter in the context.

Roles

A role is a reference to the system object, it represents the object’s functions, and what it is delegated, expected, or required to do. A role is a set of related responsibilities that can be used interchangeably.

Responsibilities

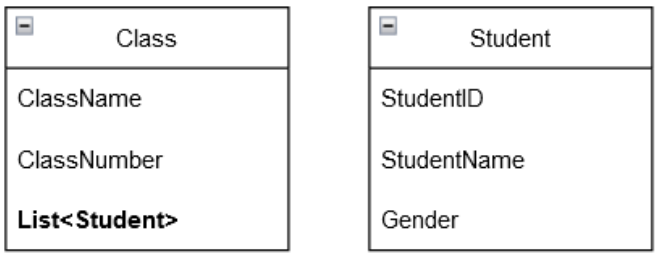
Responsibility is a contractor or obligator of the classifier. Responsibilities are related to an object’s behaviour obligations. These include performing a task or knowing some pieces of information.

Collaborations

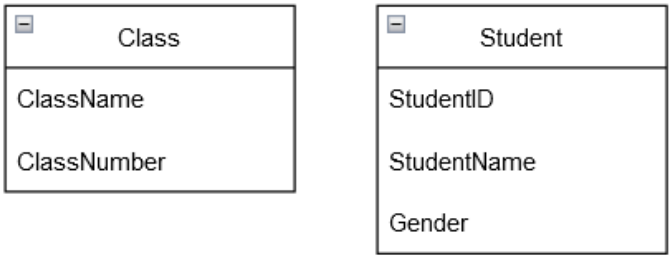
Collaborations are the sharing processes between objects. They are interactions of objects and roles (sometimes both).

Coupling

Coupling refers to the dependency and relationship between the 2 classes to each other. If the 2 classes have low coupling, it means that changing the code in one thing would not affect the other. On the other hand, software development and OOP design, in general, should avoid high coupling, as the 2 classes are closely related, changing something could require adjustments for the entire system, making it difficult for code maintenance and modification.



High Coupling

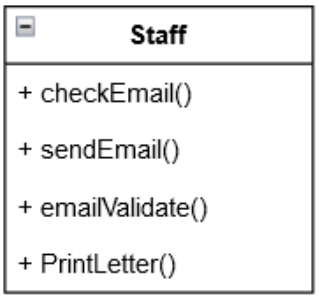


Low Coupling

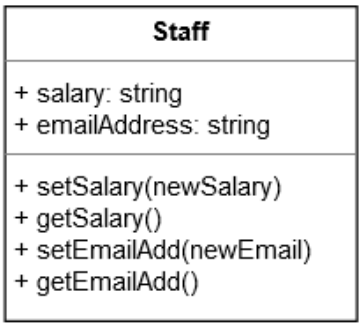
Cohesion

Cohesion refers to the functionality of a class. If a class has low cohesion, it means that this class has a wide variety of methods and functions, without any concentration on what it should do. On the other hand, high cohesion means that the class is focused on what it should be doing, as the initial intention of the class.

<The UML diagram is presented on the next page>



Low Cohesion



High Cohesion

⇨ Good OOP design should have low coupling and high cohesion.