COS20007 OBJECT ORIENTED PROGRAMMING (HANOI)

TASK 7.1P

GIA HUY NGUYEN
103441107
SWH00039

Semester September 2021

Introduction

Object Oriented Programming is a technique of generating programs using objects and their data, as opposed to a procedural program that runs from top to bottom, as in a shopping list. This encourages modularity and, in some ways, allows for a more natural and, in some ways, efficient programming style.

Abstraction

Today's programming is mainly built on the concept that the programmer should keep track of as few items as feasible — Because human working memory is limited to 3-5 items (Cowan), it is important that the programmer's life is kept as simple as possible while developing code. The concept of abstraction and encapsulation in programming is based on this. We can retain more important information by grouping information into chunks (for example, classes in OO). In essence, memorizing "Vegetable" is considerably simpler than remembering "Corn, Carrot, Tomato"and so on).

Putting ideas into objects is a big step, but it can go much farther. Polymorphism is the idea that anything can have many functions. For example, the areas of a circle, square, and triangle are all specified differently, but while having the same name, we may create several versions of the function "Area ()" that change based on the object that uses them (To String() is similar). Again, the programmer does not need to consider this distinction; all they want to know is anything, and the function takes care of it. The function varies depending on the implementation, but it still accomplishes the same basic goal.

Class and Object

Classes and objects are the major techniques for dealing with this concept in object-oriented programming. Information (fields), roles, and responsibilities can all be contained in a single "object" called a class. We may then delegate/encapsulate the responsibility for implementing programmatic behavior (methods) to something else, allowing us to focus on the desired consequence of an action (e.g., To String() to transform data into a string). The programmer doesn't care how the function works; all that matters is that it works with a specific type of data and has a specific impact.

Polymorphism

Polymorphism, which means "many forms," occurs when numerous classes are linked via inheritance. OOP programming makes considerable use of polymorphism. For instance, in Shape Drawing Task 4.1P, I have the abstract class "Shape," and the circle, line, and rectangle classes override the "Shape class's" draw function.

Inheritance

It's crucial for code reuse: you may reuse fields and methods from another class when constructing a new one. It can also help the developer avoid errors and problems. For example, in the Shape Drawing Task, I have the abstract class "Shape," and the circle, line, and rectangle inherit the "Shape" class's properties and methods (color, position).

Encapsulation

The purpose of encapsulation is to keep "sensitive" data hidden from the user. Control of class members has been improved (reduces the possibility of coder messing up the code). The protection of personal information has been enhanced. Using a "private" variable, we may alter one piece of the code while leaving the rest alone. When we establish the private class in the Shape Drawing Task and make changes to it, it will have no effect on the other classes since we have made it private and the other classes will not need to know about it.

Interface

In C#, another technique to accomplish abstraction is to use an interface. Interfaces can contain properties and methods, but not fields or variables; they can't have a constructor (since they can't be used to build objects), and they don't have a body (the body is provided by the "implement" class).

Method

They are encapsulated methods specified by an object, such as the draw function in the shape drawing job; the user simply needs to call the function and does not need to understand how it works.

Roles

The term "role" refers to a set of duties. A role is a type that holds behavior and state in the same way that a class does. It can be combined into a type (a class, a struct, or another role) that includes all of its members (but not nested types).

Cohesion

The term "cohesion" refers to how modules are connected. It shows how well the components work together. The better the program design, the more cohesiveness there is.

Coupling

Coupling depicts the interaction between modules, as well as how classes and objects are related and interdependent.

Collaboration

Collaboration is a term used to describe the interaction that occurs between different classes.

Responsibility

Another element of "responsibility" is that there should ideally only be one implementation of any code (Don't Repeat Yourself/DRY). That is, if you have a task that must be completed exactly the same manner every time, there should only be one implementation of it. This is the foundation of ancestry and partnership. By ensuring that something is implemented in a single location, we can be confident that it will be available whenever we need it: Implemented the same way and Changing the implementation affects everything the same way

If we need to utilize anything, we should either inherit it, cooperate with it, or work with another module to guarantee that responsibility is retained with a single module. This idea is somewhat at odds with polymorphism, but in general, polymorphic implementations are so because something must be done differently in order to fulfill the same conceptual objective.

Taken to its logical conclusion, one may ask, why don't we just lump everything together? Cohesion is based on this. Cohesion indicates how "focused" a class is; if it has low cohesion, it is attempting to perform several duties at the same time. Consider the Shape class, which creates circles, rectangles, and lines. It has to know how to perform each of these things, therefore it would require information on how to accomplish all of them — even if you just care about rectangles! This adds to the class's complexity. This is not particularly relevant at first glance, but when complexity increases, it becomes a key concern.

Coupling in a software reflects how well tasks and concerns have been kept distinct (note that this is a trade off with cohesiveness) A class should ideally be dependent on as few other classes as feasible; otherwise, a change in a dependence might result in broken functionality on the opposite side of the program. This runs counter to the DRY concept, therefore these considerations must be balanced while building a software. This is one of the reasons why inheritance and interfaces are helpful; they allow classes that are similar to each other to be connected while being loosely coupled elsewhere. Interfaces are used to reduce coupling between unrelated portions of the program by ensuring that the bits of another class that we care about are guaranteed.

Summary

Modern programming is all on increasing abstraction and maintainability. As a result, it is simple to write and modify. Encapsulating responsibilities, linking cohesive classes with inheritance, polymorphism, and cooperation, and disconnecting different classes with interfaces are all examples of how OO architecture facilitates this.