Object oriented programming principles

Jordan Boekel ID: 101625077

Abstraction

Programming today is based primarily on the idea that the programmer keep track of as few things as possible – Human working memory is restricted to 3-5 items (Cowan), so it is critical that life is made as *easy as possible* for the programmer when writing code. This is the basis for the idea of abstraction and encapsulation in programming. By combining information into chunks (eg. Classes in OO) we can more easily remember relevant information. In essence, remembering "Vegetable" is much easier than remembering "Corn, Carrot, Tomato"…etc).

Object oriented programming's primary methods for dealing with this abstraction are classes and objects. Classes allow information (fields), roles and responsibilities to be encapsulated into a single "object". By doing so, we can delegate/encapsulate the responsibility for implementing programmatic behaviour (methods) to something else, and just think about the intended *effect* of an action (eg. .ToString() to convert information into a string). The programmer does not care how the function works, just that it works on some type of data and has a particular effect.

Combining concepts into objects is a significant step, but it can be taken further. Polymorphism is the concept that something can fulfill multiple roles. For example, the areas of a circle, square and triangle are all defined differently, but we can construct multiple definitions of a function "Area()" that change depending on the object that calls them, despite having the same name (ToString() is similar). Again, the programmer does not have to think about this difference, they just want to know something, and the function handles it. The function changes depending on the implementation, but still fulfills the same underlying conceptual job.

Maintainability

Another aspect 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 needs to be done the *exact same way*, there should only be one implementation of it. This is the core of the existence of inheritance and collaboration. By ensuring that something is implemented in one place, we know that whenever we use it, it is

- Implemented the same way
- Changing the implementation affects everything the same way

If we need to use something, we should then either inherit it, collaborate with it etc, and so ensure that responsibility is kept with a single module. This concept clashes somewhat with polymorphism, but generally polymorphic implementations are such because there is something that must be done differently to still achieve the same conceptual task.

Taken to the extreme, one would argue, why don't we just put everything into one class? This underlies *cohesion*. Cohesion signifies how "focused" a class is – if it has low cohesion, it attempts to fulfill many responsibilities at once. Consider a Shape class that draws circles, rectangles and lines. It needs to know how to do each of these things, and so would need information on how to do all of these – even if you only care about rectangles! This increases the complexity burden of the class. At a low level, this is not very significant, but as complexity grows it becomes a critical consideration.

Coupling in a program indicates the degree to which tasks and concerns have been kept separate (note that this is a tradeoff with cohesiveness) Ideally, a class should depend on as few other classes as possible; otherwise, a change in a dependency can result in breaking functionality on the other side of the program. This is somewhat antithetical to the DRY principle above, and so these concerns need to be balanced against eachother when designing a program. This is part of why inheritance and interfaces are useful; They allows classes that are like each other to be coupled, yet loosely coupled elsewhere. Interfaces in particular are used to minimise coupling between unrelated parts of the program by guaranteeing the parts of another class that we care about.

Summary

In general, modern programming is all about maximising abstraction and maintainability. This makes it easy to write and easy to change. OO design encourages this by encapsulating responsibilities together, coupling cohesive classes together with inheritance, polymorphism and collaboration, and decoupling dissimilar classes together with interfaces.

Cowan N. (2010). The Magical Mystery Four: How is Working Memory Capacity Limited, and Why?. *Current directions in psychological science*, *19*(1), 51–57. https://doi.org/10.1177/0963721409359277