***Object Orientation***

We can solve any programming problem with a structured programming approach. However, in the last decades several other programming “paradigms” have been coined up, which help designing and implementing large-scale programs. **One of the most successful paradigms is “object orientation,”** and most modern programming languages support the object oriented paradigm.

While object orientation tends to provide a **natural way to look** at problems and solutions, designing an object oriented program can be quite hard. The reason that it is hard, is that you have to really think about your approach to a problem in all of its aspects, before you start coding. For bigger problems, this can be daunting, especially when you lack experience with programming. However, for bigger problems you have to spend a lot of time designing your solution anyway, and an object oriented approach may be quite helpful in creating it. Moreover, you will find that most modules provide object oriented implementations, and that object orientation can be helpful for many smaller problems too.

***The object oriented world***

While I am (**author**) typing this, I am sitting at my kitchen table. Next to me is a bowl of fruit. There are some apples in the bowl. While these apples share certain features, they have their differences too. They share their name, their price, and their age, but they all have (slightly) different weights. There are also some oranges in the bowl. Like the apples, they are fruits, but they have a lot of differences with apples: different names, different colors, different trees that they grow from. Still, they share some things with apples that all fruits share, and make them different from, for instance, the table I am sitting at. I can eat a fruit, i.e., I can eat apples and I can eat oranges. I am not going to try to eat a table.

When I try to model my world in a computer program, I have to model objects: objects such as apples, oranges, and tables. Some of these objects have a lot in common, for instance, *each apple shares a lot of features with every other apple. It behooves me to define a class “apple” which contains the features that all apples share, and only fill in the few features in which apples differ from each other for each individual apple object*. The same holds for oranges, they should get their own class “orange.” And while “apples” and “oranges” are quite different, they still share some features that entail that I would like to put them in the same class: the class “fruit.” Every object that belongs to the class “fruit” at least has the property that I can eat it. Which means that each individual apple object not only belongs to the class “apple,” but also belongs to the class “fruit” – just like the “oranges.”

Come to think of it: I can eat more things than only apples and oranges. I can eat cakes too. And mushrooms. And bread. And licorice. So maybe I need another class, which the class “fruit” also belongs to. The class “food,” perhaps? What this leads to, is that if I try to model the world, or part of the world, I need to model objects – and rather than modeling each separate object, I am better off defining classes of objects, as that means I can make statements about certain groups of objects in general. I can talk about the relationships between classes, and I can define functions that work on classes; for instance, I can define a functionality “eat” that works on every object that is part of the class “food,” which removes the object from the world and assigns its “nutrients” to the object that does the “eating.” Since I can “eat” objects that belong to the class “food,” I can eat “fruit.” And since I can eat “fruit,” I can eat any “apple” object.

A computer program is, in essence, a model of a part of the world. As such, there are many programs that benefit from the ability to deal with objects, classes, relationships, and functionalities (methods) that work on objects.

Ex: Any real world object (CellPhone)

*Students, teachers, and courses*

Many programs deal with persons. The student administration deals with students, who are persons. These students follow courses, which are taught by teachers, who are also persons. Undoubtedly, the student administration stores information on students and teachers, and probably the programmer who created the software for the student administration was smart enough to create a single interface that allows entering person data. What data do all persons share, as far as the student administration is concerned? Well, probably all persons have a first and a last name. They have an address. They also have an age and a gender. They all get assigned an administration number, so that for the administration they have at least one thing that makes them unique. These data elements are all “properties” or “attributes” of “persons.” I mentioned the properties first name, last name, address, age, gender, and administration number. One of these is actually more like a function than a property. Do you see which one? The answer is “age.” Age is calculated from date of birth and current date. While you can consider age a property, it is a property that should be calculated each time that it is needed. You cannot store it as a value, as tomorrow it might be different from today, without anything changing but the date. Therefore, if I design a class Person that models a person, I best make “date of birth” an attribute of the person, while “age” is a method of the person. Remember that methods are functions that belong to a certain data type: if a data type Person is defined, date\_of\_birth is an attribute of that data type, while age() is a method of that data type that returns the person’s age as an integer. students and teachers are both persons. They share the properties of the Person class. Yet there are differences. Teachers, for instance, get paid a salary, while students do not. Students, on the other hand, earn grades in courses, which teachers do not; they teach the courses. From this follow two obvious observations:

• While students and teachers are both persons, they have clear differences; besides a class Person I need a class Student and a class Teacher, both of which are derived from the class Person.

• “Courses” seem to be an inherent part of the student administration world, so a class Course might be needed too.

Once Course has become a class in the student administration world, relationships become visible. Students have relationships to multiple courses, and teachers do too, though in a different capacity. Students “enroll” in courses. It looks like an enroll() method is needed, that allows a student to get into a relationship with a course. The question is: is enroll()

a method of Student, that gets a course as argument, or is it a method of Course, that gets a student as argument? What do you think? The answer is: “it depends.” It depends on how you envision the student administration

world. To me, it feels more natural to make enroll() a method of a course, as

I view a course as a collection of students. However, in principle there is nothing against seeing a student as an entity who encompasses a collection of courses. You might also decide to make enroll() a method of each of them, or think of yet another class that contains the enroll() method that has both the student and the course as arguments.

This illustrates the difficulty of the object oriented view on program design: by designing the classes that form the world model that the program works with, choices need to be made that may have a big impact on how the program works. Weak choices may lead to difficulties in implementation. You need to spend considerable time on designing the object oriented model that underlies the program, and try to anticipate all the consequences of your choices. This is hard even for experienced designers. However, a solid object oriented model makes programs easy to read, understand and maintain. The object oriented paradigm is often worth the hassle.

***Classes, objects, and hierarchies***

In the object oriented world, every distinguishable entity belongs to a “class.” A class is a general model for a specific group of entities. It describes all the attributes that these entities have, and it describes the methods that the class offers the outside world to influence it. A class, by itself, is not an entity. An entity that belongs to the class is an “object.” The terminology is that an object is an “instance” of a particular class.

While the class describes its attributes, an object that is an instance of the class has values for these attributes. While the class describes the methods that it supports, to execute such a method one needs an object that is an instance of the class to call the method with.

*A class is a data type, an object is a value.*

Classes may exist in hierarchies. A general, high-level class may describe properties and methods that are shared by different subclasses. Each subclass may add properties, add methods, and even change properties and methods (though in general cannot – and should not – completely remove them). Each subclass may have further subclasses.

For instance, the class Apple may be a subclass of the class Fruit, which may be a subclass of the class Food. This means that where in a program an object of the class Food is needed, you can supply an object that is an instance of the class Food, but also an object that is an instance of the class Fruit, or an object that is an instance of the class Apple. This does not work the other way around, though. When, for instance, a function in a program

was designed for instances of Apple, you cannot use it with instances of Fruit, or other subclasses of Fruit. While an Apple is Fruit, Fruit is not an Apple, and Apples aren’t Oranges.

Such a hierarchy is implemented using “inheritance”.

***Reference:***

1. *Pieter Spronck, The Coder’s Apprentice, Learning Programming with Python 3.*