COMP 210 - Lecture Notes - 01 - Object Orientation

January 10, 2016

Objects and Classes

Up to this point, we've designed programs in which data definitions and procedure definitions were, for the most part, separate. Sure, we'd define structs and basic classes as the problem design, but the core functionality of those types was largely carried out by procedures and functions that were defined separately.

The design and implementation of abstract data types[1] in COMP220 began to change this. Class-based definitions ENCAPSULATED operations on data[2] along with the data definition. Doing so opened up the ability to control access to parts of the design. We could hide implementation details within the class a private data and private methods while exposing the required functionality as public. Still, the remainder of the program, the application involving that ADT, was represented with procedures and functions.

[1] ADTs

[2] class METHODS

Object-Oriented programming changes this. Procedures and functions become the exception. They're used to support classes and class methods, not the other way around. Classes and class-methods become the rule. An OBJECT is, after all, an instance of a class. To orient our programming around objects, we should orient our program design around the classes that contain those objects. All the computation must be clearly attached to and defined within a Class. This turns the program into a series of interactions between different objects.

The object-oriented design process rightfully begins with identifying the classes, the data types, involved in our problem. We then identify the functionality needed for each class in order to carry out a series of interactions that solve the problem. Abstractly[3], we imagine that a program is carried out by *objects passing messages* to one another. When one object calls a method on another object, then we imagine that caller passing a message like, "do that thing, the method, you're really good at with this data, the method arguments." The value[4] returned from that method call is in turn a message from the callee to the caller, "Sure. This data, the method return value, is what I came up with." This style of programming can fit into an imperative regime or a functional one.

[3] or not so abstractly in some languages, i.e. Objective-C

[4] an object almost certainly

Class Hierarchies

Using class-based objects to encapsulate program logic is just where the fun begins. With object-oriented program design we don't just design classes, we design CLASS HIERARCHIES. Hierarchical data is nothing new. We've at least implicitly recognized hierarchies of data types in problems and used some non-object-oriented techniques to deal with them. At a primitive level, when we OVERLOAD a procedure to act on multiple types of a similar kind[5], we're creating a procedure that implicitly acts upon a hierarchy of data.

[5] i.e. lots of different types of numbers

In our work with C++ streaming I/O, we designed procedures that explicitly took advantage of class hierarchies. We designed and implemented I/O procedures to work on *istream* and *ostream* objects. When we tested these procedures we used an *istringstream* or an *ostringstream*. The compiler allowed it even though the procedure's stream parameter type wasn't the same as the argument type. This is because the string stream classes are defined as a SUBCLASS of their respective basic stream[6]. The subclass-superclass relationship is an important one to understand in OOP. Subclasses are extensions of the superclass. This means the add or modify existing functionality. For example, the *ifstream* class modifies the stream reading capabilities of its superclass *istream* to read from files and then adds a few file-specific methods.

[6] this makes the basic stream a SUPERCLASS

Perhaps the most important, misunderstood, and abused subclass-superclass relationship is INHERITANCE. The type *ofstream* inherits its superclass's type, *ofstream*. This means that all objects of type *ofstream* are also of type *ostream*. This is exactly what allowed us to assign the[7] of type *ofstream* or *ostringstream* to the *ostream* parameter of our C++ output procedures. The parameter itself exhibits POLYMORPHISM[8] as it can take on many different types, namely *ostream* and any of its decedents in its hierarchy, over the course of a program. The methods invoked from that *ostream* utilize POLYMORPHIC METHOD DISPATCH. The exact behavior of a method may depend on the exact type of the object stored in the parameter, so the code executed[9], can differ when the object's most specific type differs. Finally, the subclasses of *ostream* exhibit IMPLEMENTATION INHERITANCE from *ostream*. Some parts of their behavior are, in fact, implemented in *ostream* and do not show up at all in their own definitions. When inherited methods are invoked, the computer will recognize a lack of implementation in the subclass and default to that of the superclass. This can propagate all the way up the hierarchy.[10]

[7] object

[8] Greek for "many forms"

[9] or dispatched

[10] This paragraph is very important and merits several careful re-reads!

Polymorphism and inheritance can make code trickier to follow and harder to predict. The exact method implementation that gets called is now a dynamic, run-time property of the program. Previ-

ously, it was always clear. You, the programmer, created conditional branching logic that dispatched the appropriate variation of a procedure. For example, every time you checked to see if a list was empty or not, you were dispatching based on type. In an OOP setting with hierarchies of classes we can leave this mundane task to the compiler and computer. This is the trade off we make, more concise code without a lot of type-based branching, but less static predictability.

OOP in a Nutshell

Designing and implementing programs using object-oriented programming is first and foremost about analyzing the they problem from the perspective of hierarchical data types. We'll see that this often means turning computation, things that we'd typically develop as procedures, into tangible types, a thing that carries out a specific task. Just as often it simply means attributing ownership of a particular action to a specific type of data in a very clear and explicit manner by making that action a class method. Either way, all our code is now class-based. Our programs are now reasoned about as a series of interactions between objects the result of which will solve our problem. We design and implement things that can communicate in such a way that leads to the solution to our problem.