Object-Oriented Design	Lecture 2
CS 3500 Fall 2009 (Pucella)	Tuesday, Sep 15, 2009

ADTs and Algebraic Specifications

One of the first questions we are faced with when trying to design a piece of software is "What is the data that the program must manipulate? What are the objects? For example, when designing a game or a simulation, likely data includes simulated artifacts such as spaceships, planets, and stars, enemy robots, maps representing the world to be simulated, and so one. These are all fairly concrete data. Other forms of data are more abstract—for instance, strategies for controlling computer-controlled robots.

For the purpose of our discussion today, let's focus on one kind of data that arises in a simulation-style computer game, namely maps representing the game world. To keep the discussion simple, I will consider very simple kind of map, namely two-dimensional square grid maps.

So how do we go about thinking about maps? When designing software, it's best to keep an open mind as to the exact form the software will take. We can simply assume that a map is an object without committing to anything else. In particular, we will not want to commit to a particular way of representing a map. Let's embody this into a principle: You'll like it, it's all about not doing work.

> **The Principle of Least Commitment**: Don't commit yourself any more or sooner than necessary.

As we will soon see, it does not really matter what objects are. What matters about objects is how they behave. So let's ask the question: how should maps behave?

That's a vague question. Let's refine it somewhat, and look for something specific. Behaviors are induced when objects are acted upon. So how do we want to act on maps? In other words, what operations do want to support on maps?

Here are some sensible operations on maps: creating new maps, adding an artifact to a map (e.g., adding a spaceship at a particular location in the map), removing an artifact from a map, moving an artifact in the map, loading and saving a map to disk, querying the content of a location on the map. You should be easily able to think of others.

Operations on maps naturally divide into three kinds of operations. The first kind of operations are those that create a map, which we will call *creators* or sometimes constructors. The second kind of operations are those that extract information from maps, which we will call *accessors* or *selectors*. When an accessor returns true or false we often call the accessor a *predicate*. The final kind of operations are those that do not extract information or create new maps, but rather effect a change in the real world, such as saving a map to disk and physically changing the content of the disk, which we will call *effectors*. Effectors are those operations that actually *do* something that you as a human being sitting at the computer notice. Printing out information on the screen for you to read is an effector operation. They cannot be undone.

For now, we will keep the design exceedingly small and simple. In particular, we'll leave out some behaviors and operations that we will need later. But that's okay. Part of the point of object-oriented design and programming is that it makes it easy to add behaviors to objects in the future.

I will also most likely make mistakes, something voluntarily, sometimes not. Between that and needing to add operations later on, we will certainly need to revise our design in the future. Revising a design is must less expensive that revising a program, because a program accumulates all sort of cruft that a design does not have, including implementation choices. Moreover, a good design is much easier to turn into correct code than a bad design. Worse, a buggy design makes it impossible to write correct code. Thus, it pays to get the design right.

The concept of data with an associated set of operations is important enough that we'll give it a name: an *abstract data type* (ADT for short):

> **Abstract Data Type**: An abstract data type is a set of data and a set of operations that can be performed on the data, along with a description of what those operations do.

Let's consider an ADT for maps. Let's consider the operations. What operations should we want to support on maps? First of all, we need to create maps—we want creators. There are several choices possible, and here are mine. I want an operation `empty` that creates an empty map, that is, a map without any artifacts in it. This is useful, in the same way that zero is useful in arithmetic. For the sake of simplicity, let's assume that maps have a fixed size. (It's easy to make size a parameter.) I want an operation `singleArtifact` that creates a map containing a single artifact in it, at some specified location. I also want to create more complex maps, recursively. To do so, I want an operation `merge` that creates a map by merging together two maps, so that the artifacts in both maps are included in the resulting map. (Other choices of creator operations are possible and equivalent to these three; as an exercise, try to come up with a few.)

Being able to create maps is a good first step. But we may also want to extract information from maps—we want accessors. Here are some operations that do so. First off, an operation `isEmpty` is useful to check if a map is empty. To extract the artifacts in a non-empty map, I want operations `firstArtifact` that returns the first artifact in map, and `restArtifacts` that returns a map containing all but the first artifact of a given map.

Signature of the Map ADT

A *signature* consists of the names of the operations together with their type.

For the Map ADT, a reasonable signature would be as follows:

```
empty :                            -> Map
singleArtifact :  Artifact nat nat -> Map
merge :                    Map Map -> Map

isEmpty :        Map -> boolean
firstArtifact :  Map -> Artifact
restArtifacts :  Map -> Map
```

This signature assumes that we have a type for artifacts which I will just assume is an ADT itself called `Artifact`, as well as types for Boolean values and for natural numbers.

Notice that the creators all return a `Map` object, as expected, while all the accessors take a `Map` object as an argument.

A signature describes one aspect of the interface, namely, what shape the operations have, that is, what they expect as arguments and what they return as a result. The signature gives no clue as to how those operations are meant to behave, however. We need to remedy that situation. This is the second part of the definition of ADT.

Specification of the Map ADT

A *specification* (or spec, for short) is a kind of guarantee (or contract) between clients and implementors.

Clients

- depend on the behavior guaranteed by the spec, and
- promise not to depend on any behavior not guaranteed by the spec.

Implementors

- guarantee that a provided abstraction behaves as specified by the spec, and
- do not guarantee any behavior not covered by the spec.

It is hard to specify how objects behave. Usually, this is done in English, in an informal way. But the resulting specification is often incomplete, incorrect, ambiguous, or confusing. You'll see examples of those very often.

Let me introduce a *formal* way of specifying behavior, as a set of algebraic equations that the operations of the interface must obey. Because of that, we will call it an *algebraic specification*. (There are other ways of specifying behavior, which we may get to before the end of the course.)

The basic rule is to describe how each accessor works on any object constructed using the creators.

Specifying `isEmpty` is straightforward:

```
isEmpty (empty ()) = true
isEmpty (singleArtifact (a,i,j)) = false
isEmpty (merge (m1,m2)) = isEmpty (m1) & isEmpty (m2)
```

where & is the *and* operation that is defined to be true if both arguments are true, and false otherwise.

Specifying `firstArtifact` is just a bit more interesting:

```
firstArtifact (singleArtifact (a,i,j)) = a
firstArtifact (merge (m1,m2)) =
     firstArtifact (m2)      if isEmpty (m1)
     firstArtifact (m1)      otherwise
```

First off, there is no equation describing how `firstArtifact` behaves when applied to an empty map. That's on purpose: no behavior is specified, because it should be an error. The implementor is free to do as she wishes. (In general, she will report an error through an exception or a similar mechanism, but it may make sense in some situations to silently ignore the application and proceed with the rest of the computation.)

Second, the equation for `firstArtifact` applied to a merged map is a conditional equation, because it depends on properties of the first map, namely whether it is empty or not.

Specifying `restArtifacts` is similar to `firstArtifact`, with many of the same subtleties:

```
restArtifacts (singleArtifact (a,i,j)) = empty ()
restArtifacts (merge (m1,m2)) =
    restArtifacts (m2)              if isEmpty (m1)
    merge (restArtifacts (m1),m2)   otherwise
```

These equations seem only to specify the behavior of the accessors, but in fact, they describe the interaction between the accessors and the creators, and thereby implicitly also specify the behavior of the creators.

Let me emphasize again: the above specification *describes* how the operations behave, what they do. And in particular, they say a lot more than they seem to say. Look at how `firstArtifact` and `restArtifacts` are defined. They assume that there is an order to the artifacts in a map (what that order is is irrelevant), and that when you merge two maps, the order of the artifacts is preserved, in the sense that the artifacts in the first map all come before the artifacts in the second map in the resulting merged map. Putting it another way, the description of the operations assumes that the artifacts in a map are put in a list, and merging two maps corresponds to appending their list of artifacts. That may or may not be what you had in mind when I informally described how maps behaved, but that's irrelevant. The above is the specification, and the specification tells you the contract I expect the implementors of the Map ADT to abide to.

Exercise: *Define a different Map ADT such that the operation* ***`firstArtifact`*** *returns the "upper leftmost" artifact in the map, and* ***`restArtifacts`*** *returns the map obtained by removing the "upper leftmost" artifact. Take whatever sensible definition of "upper leftmost" you can come up with. This is a very challenging exercise, and more intended to get you to think about what goes into a specification than anything else. And yes, you probably have to assume some things about the Artifact ADT as well.*

Note that I have not said how maps are implemented. And I don't care at this point. I do not care how the operations do what they claim to do, I am just describing how I want them to behave. Figuring out how to implement objects, how to go from a description of their behavior to actual code that implements that behavior, is a decision that we will resist making for as long as possible, per the Principle of Least Commitment.

Despite this lack of description of how objects are implemented, the specification is still powerful enough to tell us the result of complex operations. For instance, suppose that I want to check what is the result of extracting the second artifact out of a map with three artifacts "starship", "star", and "planet", in that order, at respective positions (0,0), (1,1), and (2,2). (For simplicity, assume that an artifact is represented by a string.) Clearly, the result of that extraction should be the artifact "star". That is, we want to check that the result of the following expression is the artifact "star":

```
firstArtifact (restArtifacts (merge (singleArtifact ("starship",0,0),
                                     merge (singleArtifact ("star",1,1),
                                            singleArtifact ("planet",2,2)))))
```

Well, I can use the equations above to replace equals by equals and simplify the above expression, just like you would do in algebra. This is were the name algebraic specification comes from, by the way. Here is one possible derivation. See if you can spot the equations I used at each step:

```
  firstArtifact (restArtifacts (merge (singleArtifact ("starship",0,0),
                                       merge (singleArtifact ("star",1,1),
                                              singleArtifact ("planet",2,2)))))
= firstArtifact (merge (restArtifacts (singleArtifact ("starship",0,0)),
                        merge (singleArtifact ("star",1,1),
                               singleArtifact ("planet",2,2))))
= firstArtifact (merge (empty (),
                        merge (singleArtifact ("star",1,1),
                               singleArtifact ("planet",2,2))))
= firstArtifact (merge (singleArtifact ("star",1,1),
                        singleArtifact ("planet",2,2)))
= firstArtifact (singleArtifact ("star",1,1))
= "star"
```

This is, of course, the result that we expected. But the point is, the whole point is, we can compute the result *without having ever said a word about how maps are implemented*!