Algebraic Specifications

Last time I said a word about the abstraction barrier, separating the clients from the implementors. This abstraction barrier is often called the interface. (We will come across this word often in the context of this course.) This interface is what connects the clients and the implementors.

We illustrate this notion of interface by designing and implementing a simplified version of an abstract data type (ADT) that arises in computer music. I will not assume that you know much about computer music, so I will keep much of this discussion simple. We will design and implement an abstract data type to represent music that can be sung by a single voice. In keeping with standard musical terminology, we will call this ADT the Voice ADT.

A musical voice may have many properties, but in our simplified view of the worl, a voice is basically just a sequence of notes and rests.

What are the objects?

The main question in object-oriented design is ontological: what are the objects?

For this exampe, it seems clear that the objects of interest include voices, notes, and rests. In general, the choice need not be so obvious.

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 voice is an object without committing to anything else. In particular, we will not want to commit to a particular way of representing a voice. Let's embody this into a principle:

> **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 voices 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 voices? In other words, what operations do want to support on voices?

As I said, we will keep the design small and simple for now. 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—partly because a program accumulates all sort of cruft that a design does not have, including implementation choices. There are also several good reasons for wanting to revise a design rather than revising code. The rule is that a good design is much easier to turn into correct code than a bad design. Similarly, a buggy design makes it impossible to write correct code. Thus, it pays to get the design right.

So what operations should we want to support on voices? First of all, we need to create voices. There are several choices possible, and here are mine. We want an operation `emptyVoice` that creates just that, an empty voice, that is, a voice without any notes in it. This is useful, in the same way that zero is useful in arithmetic. We want an operation `oneNote` that creates a voice made up of only one note. We also want to create more complex voices, recursively. So we want an operation `concatenated` that creates a voice by concatenating together two voices, so that when played they get played in sequence. (Other choices of creator functions are possible and equivalent to these three; as an exercise, try to come up with a few.)

Being able to create voices is a good first step. But we may also want to extract information from voices. Here are some operations that do so. First off, an operation `isEmpty` is useful to check if a voice is empty. To extract the notes in a non-empty voice, we want operations `firstNote` that returns the first note of a voice, and `otherNotes` that returns a new voice containing all but the first note of a voice.

Already, we see that the operations supported by a voice (and in fact by an ADT in general) naturally split into two: operations to create voices, which we will call creators or constructors, and operations to extract information from voices, which we will call accessors, selectors, or predicates (when they return true or false).

Signature of the Voice ADT

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

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

```
emptyVoice :                  -> Voice
oneNote :              Note   -> Voice
concatenated : Voice x Voice -> Voice

isEmpty :    Voice -> boolean
firstNote :  Voice -> Note
otherNotes : Voice -> Voice
```

This signature assumes that we have a type for notes, which I will just assume is a class of objects called `Note`.

Notice that the creators all return a `Voice` object, as expected, while all the accessors take a `Voice` 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, and that clearly should be an important part of the interface.

Specification of the Voice 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 selector works on any object constructed using the creators.

Specifying `isEmpty` is straightforward:

```
isEmpty (emptyVoice ()) = true
isEmpty (oneNote (n)) = false
isEmpty (concatenated (v1,v2)) = isEmpty (v1) & isEmpty (v2)
```

Specifying `firstNote` is just a bit more interesting:

```
firstNote (oneNote (n)) = n
firstNote (concatenated (v1,v2)) =
     firstNote (v2)      if isEmpty(v1)
     firstNote (v1)      otherwise
```

First off, there is no equation describing how `firstNote` behaves when applied to an empty voice. 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.)

Second, the equation for `firstNote` applied to a concatenated voice is a conditional equation, because it depends on properties of the first voice.

Specifying `otherNotes` is similar to `firstNote`, with many of the same subtleties:

```
otherNotes (oneNote (n)) = emptyVoice ()
otherNotes (concatenated (v1,v2)) =
    otherNotes (v2)                       if isEmpty(v1)
    concatenated (otherNotes (v1),v2)     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.

Note that I have not said how voices 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 they behave. The focus on how to objects do what you want them to do, while often the focus of a programming course, is a decision that we will resist taking, per the Principle of Least Commitment.

Despite this lack of description of how objects work, 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 note out of a three-notes voice, that is, suppose I wanted to know the result of:

```
firstNote (otherNotes (concatenated (oneNote (n1),
                                     concatenated (oneNote (n2),
                                                   oneNote (n3)))))
```

Well, I can use the equations above to replace equals by equals and simplify the above expression, just like you would do in algebra (hence the name, algebraic specifications). Here is one possible derivation. See if you can spot the equations I used at each step:

```
firstNote (otherNotes (concatenated (oneNote (n1),
                                     concatenated (oneNote (n2),
                                                   oneNote (n3)))))
= firstNote (concatenated (otherNodes (oneNote (n1)),
                           concatenated (oneNote (n2),
                                         oneNote (n3))))
= firstNote (concatenated (emptyVoice (),
                           concatenated (oneNote (n2),
                                         oneNote (n3))))
= firstNote (concatenated (oneNote (n2),
                           oneNote (n3)))
= firstNote (oneNote (n2))
= n2
```

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 voices are implemented*!

Observational equivalence

At this point, all we care about an object is how it behaves. In particular, the only thing we can tell about an object's behavior is what observations we can make. In our case, the observations for voice include whether a voice is empty, what the first note of a voice is, and what all but the first note are.

The notion of observavtion is central to the next principle.

> **Principle of Observational Equivalence**: If two objects wold behav the same in all possible situations (i.e., yield exactly the same observations), then the two objects are indistinguishable and might as well be regarded as the same object.

This principle will be used as a justification for different implementations of the same specification being interchangeable.

Note that there other kind of observations we can make on objects, that correspond to other kind of operations that are not creators or accessors. For example, one operation we may want to have on voices is an operation to actually perform the voice, play a rendition of it through a computer speaker. This is an operation that actually has an effect on the world, by creating sound waves in this case. Operations that either affect the state of the world or the internal state of an object are said to perform a side effect, and such side-effecting operations are much harder to reason about. We will delay talking about them until later in the course.