# 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 an animation package, likely data includes points, lines, circles, drawings, canvases. Some choices are not so obvious. For animation, for example, some data is going to be more abstract, like trajectories for animation.

For the purpose of our discussion today, let's focus on one kind of data that arises in computer graphics, two-dimensional drawings. To keep the discussion simple, I will consider very simple kind of drawings, line drawings, made up of straight lines.

So how do we go about thinking about drawings? 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 drawing is an object without committing to anything else. In particular, we will not want to commit to a particular way of representing a drawing. 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 drawings 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 drawings? In other words, what operations do want to support on drawings?

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, 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 o data with an associated set of operations is important enough that we'll give it a name: an *abstract data type*:

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

So what operations should we want to support on drawings? First of all, we need to create drawings. There are several choices possible, and here are mine. We want an operation `empty` that creates just that, an empty drawing, that is, a drawing without any lines in it. This is useful, in the same way that zero is useful in arithmetic. We want an operation `oneLinee` that creates a drawing made up of only one line. We also want to create more complex drawings, recursively. So we want an operation `merge` that creates a drawing by merging together two drawings, so that the lines in both drawings are included in the resulting drawing. (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 drawings is a good first step. But we may also want to extract information from drawings. Here are some operations that do so. First off, an operation `isEmpty` is useful to check if a drawing is empty. To extract the lines in a non-empty drawing, we want operations `firstLine` that returns the first line of a drawing, and `restLines` that returns a new drawing containing all but the first line of a drawing.

Already, we see that the operations supported by a drawing (and in fact by an ADT in general) naturally split into two: operations to create drawings, which we will call creators or constructors, and operations to extract information from drawings, which we will call accessors or selectors. When a selector returns true or false, we often call the selector a predicate.

### Signature of the Drawing ADT

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

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

```
empty :                      -> Drawing
oneLine :      Point Point -> Drawing
merge :    Drawing Drawing -> Drawing

isEmpty :    Drawing -> boolean
firstLine :  Drawing -> Point Point
restLines :  Drawing -> Drawing
```

This signature assumes that we have a type for points, which I will just assume is an ADT itself called `Point`.

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

### Specification of the Drawing 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 (empty ()) = true
isEmpty (oneLine (p1,p2)) = false
isEmpty (merge (d1,d2)) = isEmpty (d1) & isEmpty (d2)
```

Specifying `firstLine` is just a bit more interesting:

```
firstNote (oneLine (p1,p2)) = (p1,p2)
firstNote (merge (d1,d2)) =
     firstLine (d2)      if isEmpty(d1)
     firstLine (d1)      otherwise
```

First off, there is no equation describing how `firstLine` behaves when applied to an empty drawing. 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 `firstLine` applied to a merged drawing is a conditional equation, because it depends on properties of the first drawing.

Specifying `restLines` is similar to `firstLine`, with many of the same subtleties:

```
restLines (oneLine (p1,p2)) = empty ()
restLines (merge (d1,d2)) =
    restLines (d2)               if isEmpty(d1)
    merge (restLines (d1),d2)    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 drawings 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 line out of a three-lines drawing, perhaps a triangle. If the triangle if formed of lines starting from point `p1` going to `p2` then to `p3` before coming back to `p1`, then extracting the second line from that drawing should give us back the line from `p2` to `p3`. That is, we want to check that the result of the following expression is the line from `p2` to `p3`:

```
firstLine (restLines (merge (oneLine (p1,p2),
                             merge (oneLine (p2,p3),
                                    oneLine (p3,p1)))))
```

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:

```
  firstLine (restLines (merge (oneLine (p1,p2),
                               merge (oneLine (p2,p3),
                                      oneLine (p3,p1)))))
= firstLine (merge (restLines (oneLine (p1,p2)),
                   merge (oneLine (p2,p3),
                          oneLine (p3,p1))))
= firstLine (merge (empty (),
                   merge (oneLine (p2,p3),
                          oneLine (p3,p1))))
= firstLine (merge (oneLine (p2,p3),
                   oneLine (p3,p1)))
= firstLine (oneLine (p2,p3))
= (p2,p3)
```

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 drawings 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 drawings include whether a drawing is empty, what the first line of a drawing is, and what all but the first line are.

The notion of observation is central to the next principle.

> **Principle of Observational Equivalence**: If two objects would behave 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 drawings is an operation to actually draw them on the screen. This is an operation that has an actual effect on the world, by creating visual output 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.