Chapter

9

Object-Oriented Thinking

Smalltalk is one of the pure Object-Oriented (OO) languages. Unlike C++, which makes it very easy to write procedural code (ie, use C++ as a better C), Smalltalk makes it difficult to write procedural code. Thinking in objects is very different from thinking in procedural terms and it usually takes about 8-12 months for experienced procedural programmers to become fairly automatic in their OO thinking. To use a language that doesn't help that process makes it much easier to remain in a half-way schizophrenic state.

OO thinking is fun and it is different. One difference is that OO development tends to be more iterative than procedural development, more spiral. In OO development we tend to acknowledge that we don't fully understand our objects and their interactions. Some objects that we thought we needed will disappear, others will come into existence as we realize we need them, while others will find their behavior and responsibilities changed or split off to other objects.

Little steps

Because Smalltalk code is so easy to change, it's very natural to approach problems in an iterative way. When I'm faced with a difficult application problem, I don't usually know how I'm going to solve it. I do enough analysis and design to get a grasp on the problem, but the whole solution rarely appears to me. So, I just start somewhere and write Smalltalk code. The act of writing the code, creating the classes and defining their interactions gives me new insight into the problem and into solutions. Which then allows me to write code to explore those new ideas, which leads to fresh insights, and so on.

Smalltalk makes it very easy to approach problems in an iterative manner, taking very small steps, getting things working, then taking the next small step. In working like this you really are combining the analysis, design, and programming phases, doing a bit of each, then a bit more of each. The OO cycle is very much one of do some analysis — do some design — do some programming — gain some insights — do some analysis, etc.

Copyright © 1997 by Alec Sharp

Download more free Smalltalk-Books at:
- The University of Berne: http://www.iam.unibe.ch/~ducasse/WebPages/FreeBooks.html
- European Smalltalk Users Group: http://www.esug.org

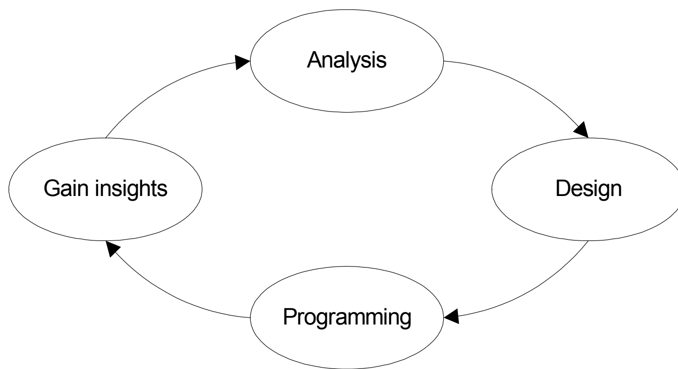

Figure 9-1.
The object-oriented development cycle.

Write down the concepts

It is extremely easy to write, test, and change code in Smalltalk. This ease of writing allows you to write your design in the code, to think in conceptual terms rather than worrying about the details (which is exactly the difference between design and implementation). In a method, just write down the things that need to be done, at a fairly high conceptual level. Someone should be able to easily read the method to see what it's doing (short method, very descriptive names), without actually seeing any of the work being done. So, for example, you might write a method as follows.

```
MyClass>>doSomething
   self myDoConceptualThingOne.
   self myDoConceptualThingTwo.
   self myDoConceptualThingThree
```

You've now written down the basic design. You can implement the design by writing `myDoConceptualThingOne`, etc. The beauty of being able to do this easily is that as you gain insights into the domain problem, you can easily incorporate changes. For example, it might become apparent that another object should be doing ConceptualThingTwo. However, you also need to do ConceptualThingFour. It is very easy to move `myDoConceptualThingTwo` to another object then to modify your code.

```
MyClass>>doSomethingWith: anObject
   self myDoConceptualThingOne.
   anObject doConceptualThingTwo.
   self myDoConceptualThingThree.
   self myDoConceptualThingFour
```

Admittedly, this is low level design; however, this is not a book on Design, so we won't explore how you work on the big-picture.

Put off the work

Because good Smalltalk methods are small, it's hard to do much work in such a method. Which leads to the next Smalltalk OOP principle: put off the work as long as possible — make another method do it. The ease of writing Smalltalk code makes it possible to put off real work until later, so a lot of methods will do nothing but divide up the work. Eventually someone has to do the work, but if you consciously put off the work as long as you can, you will end up with a more object-oriented system.

Smalltalk might be considered the procrastination language — procrastinate as long as possible and maybe you won't need to do anything. When in doubt, put it off. Here's an example of poor code that we will try to refine using the principles just described.

```
MyClass>>myDoLoop
   [object := self mySharedQueue next.
   object isHeartbeat
      ifTrue: [self myHeartbeat: object].
   object isSuccessResponse
      ifTrue: [self mySuccessResponse: object].
   object isFailureResponse
      ifTrue: [self myFailureResponse: object]
   ] repeat.
```

If we strip this down to the bare bones, we get something from a SharedQueue and process it, then repeat these actions. So we might write:

```
MyClass>>myDoLoop
   [object := self mySharedQueue next.
   self myProcessObject: object ] repeat.
```

If we put off the decision on where to get the object, we might end up with:

```
MyClass>>myDoLoop
   [self myProcessObject: self myGetObject] repeat
```

We now have a very generic loop, one that could be put in a superclass and inherited. It doesn't care where the input comes from or how we plan to process the object. In our case, `myGetInput` is very simple. It just does:

```
MyClass>>myGetInput
   ^self mySharedQueue next.
```

In another problem domain it might be a lot more complex, getting input from the user, a socket, a file, a serial line, a UNIX pipe, etc.

(A side benefit of doing the processing in other methods is that you can't modify a running loop and have the changes take effect. Since the code is already running, the changes will be ignored until next time you run the loop. By putting the processing code in other methods, you can put debugging changes in them and have the changes take effect the next time the method is executed.)

Tell, Don't Ask

What about `myProcessObject:` in the above section? We could certainly put our conditional code there — we would have moved it away from the main processing — but the code would still be very procedural. We need to rethink our objects so that we can simply ask the objects to process themselves. Procedural code gets information then makes decisions. Object-oriented code tells objects to do things.

```
MyClass>>myProcessObject: anObject
   anObject processYourself
```

Another example might be displaying graphical objects in a window. If we were thinking procedurally, our Window object might have code that looks like

```
MyWindow>>displayObject: aGraphicalObject
   aGraphicalObject isSquare ifTrue: [self myDisplaySquare:
aGraphicalObject].
   aGraphicalObject isCircle ifTrue: [self myDisplayCircle:
aGraphicalObject].
   aGraphicalObject isTriangle ifTrue: [self myDisplayTriangle:
aGraphicalObject].
```

We are asking questions, getting back information, and executing code based on the information. If we were thinking more in OO terms, we would be trying to avoid getting information and making decisions. Instead, we'd be *telling* the object to do something. Our example might be transformed into

```
MyWindow>>displayObject: aGraphicalObject
   aGraphicalObject displayYourselfOn: self
```

One benefit we get out of this transformation is that it becomes a lot easier to handle different types of graphical object. Rather than having to modify the window code every time we come across a new type of graphical object, we just need to make sure that each class of graphical object we write knows how to display itself. So if we add a Hexagon class, we just need to write `displayYourselfOn:` for the Hexagon (unless it can inherit from a superclass).

Don't check the results of doing things

In a procedural system we typically make a function call, get back some status information about the results of the call, check the status information, and possibly do some error handling. In a Smalltalk OO system it's much more appropriate to just tell an object to do something, then forget about it. It's as if we are sending messages out into space, never to return.

Obviously we need to handle error situations in production quality code, but Smalltalk provides a special mechanism for that. Chapter 20, Error Handling, provides more information, but briefly, we can wrap a whole hierarchy of message sends in a signal handler. If an exception is raised because of some error, the exception bubbles up the stack until it finds a signal handler that can handle it. The signal handler then takes the necessary actions. The beauty of this scheme is that once we wrap our code in an exception handler, we can write it as though every message send is successful.

There will be some situations where it may be appropriate to get back a status object from a method, such as if we are requesting information from an external source. However, the majority of error handling can be done with exceptions rather than by checking status returns.

Signs of Object-Oriented Thinking

Short methods

The average Smalltalk method length is seven lines. I've always made it a rule that methods should fit into my browser window and I very seldom break that rule. If a method is too long, I'll try to find two or three

conceptual things that the method is doing, and create separate methods for each of them. If you consistently have methods that don't fit in a browser window, you are probably thinking procedurally. (The other value in having all methods fit within the window is that if you can see the whole thing, you can better grasp what is going on.)

No dense black methods

Another indication that your methods are doing too much is when they look too black. This is an aesthetic judgment, but can be a useful heuristic to determine whether you are trying to do too much in a method. Try to make methods "stupid". A stupid method is one that is so simple that it's obvious what it's doing; it doesn't need to be documented. Black methods are usually the opposite of stupid methods, too much black being an indication that a lot is going on in the method.

No super-intelligent objects

There is no rule for how many methods an object has, but there are some aesthetic judgments you can make. If you find that one object has many more methods than other objects, it may be that you are not distributing the workload well. In a good Object-Oriented system, you don't have super-intelligent objects. Instead, you have many peer-type objects cooperating. Super-intelligent objects often indicate that there is too much procedural thinking (in a sense a procedural system is one with a single super-intelligent object). It may be time to rethink your objects if you have classes with an uneven distribution of methods between them.

No manager objects

It's very easy to write systems with manager objects telling other objects what to do. A good object-oriented system tends to have a lot of peers cooperating, with a reasonably equal distribution of responsibilities and workload. Managers tend to make a lot of decisions that other objects could be making. If you have a classes whose name ends in Manager, you might give some thought to how you could distribute the manager's workload to the objects it is managing.

Objects with clear responsibilities

Each class should have a clear responsibility. If you can't state the purpose of a class in a single, clear sentence, then perhaps your class structure needs some thought.

Not too many instance variables

If a class has a lot of instance variables, this can mean that it's trying to do too much, that you are not distributing the workload well. It may be possible to create other objects that can act in a supporting role or can collaborate with this objects of this class.

Getting started with Object-Oriented Thinking

It can be difficult to start thinking in terms of objects if you've been programming in a procedural language for some time. One approach I've found very useful in learning Smalltalk and Object-Oriented techniques is to

work in pairs at a workstation. One keyboard, one screen, two people. One person types, while the other person makes suggestions, asks questions, and points out errors. After a while they change roles and the other person types. In this approach, you get the best of two people rather than the best of one person. Designs are better thought through because there's another person asking why and suggesting alternatives. Code has fewer bugs because there's someone else looking at the code. The code usually ends up being more efficient, maintainable, and consistent. Because there is a second person looking at all the code, there's much less need to review low-level designs and code.

Since two people work on all features, you now have two people who understand the code rather than just one. And both people learn from each other. Programmers have their own styles, techniques, and toolboxes of solutions. By working in pairs, they learn these things from each other and grow as programmers. By periodically change partners, these techniques and solutions are passed from person to person, with each person in effect learning from everyone their partner has previously worked with.

When I started programming in Smalltalk, I worked with another developer as a pair. It was great fun and we both learned a lot from each other. Since then I've always favored the pair approach, and so I read with interest two pieces that I later came across, both praising the concept of programming in pairs. In his article in the July/August 1995 issue of *The Smalltalk Report*, Kent Beck says "The most productive form of programming I know (functionality/person/hour) is to have two people working with one keyboard, mouse, and monitor." The second piece was a section called *Dynamic Duos* on page 118 of Larry Constantine's book *Constantine on Peopleware*. Constantine praises the two person concept, remarking that "The operating principle here is very broad: Increasing work visibility leads to increased quality!"