11

Be Polite

I must always be polite. I must always be
I must always be polite. I must always
I must always be polite. I must always be
I must always be polite. I must always be
I must always be polite. I must always be
KOHNKE

We want our modules to be polite. Polite modules follow Ward Cunningham's definition of cleanliness. As you read the code, you find it to be "pretty much what you expected."

A module is a bounded set of functions and variables. This might be composed of one class, several classes, or no classes at all. What matters are the functions and variables. A module may also be contained in a single source file or several source files. What matters is the cohesion of the functions and variables, and not the artificial boundaries that separate them. Another common word for a module is a *component*.

If you follow something roughly akin to Extract Till You Drop, then your modules will be composed of a few public functions and the many private functions that were extracted from the public functions. Again, these may be organized into classes so that your module contains a few public classes and several private classes.

The public functions, possibly organized into public classes, represent the interface of your module. The private functions, also possibly organized into private classes, represent the implementation of your module.

To borrow a concept from John Ousterhout's wonderful book *A Philosophy of Software Design*,[1] that module has a narrow interface and a deep implementation—and that's a good thing.

1. [APOD].

Now, pick one of those public functions. It performs a task at the highest level of the module. The name of that function should identify that task in abstract terms. For example, consider the `makeStatement` function from the `RentalStatement` class in the previous chapter.

```
func (statement *RentalStatement) makeStatement() str
  statement.clearTotals()
  return
    statement.makeHeader() +
    statement.makeDetails() +
    statement.makeFooter()
}
```

The name of the function is abstract. It tells you what the function does, but not how the function does it. Looking inside the `makeStatement` function we see four calls to private functions that were extracted from it. Each of those names is also abstract, but at a lower level than `makeStatement`. Let's look at `clearTotals`.

```
func (statement *RentalStatement) clearTotals() {
  statement.totalAmount = 0.0
  statement.frequentRenterPoints = 0
}
```

We can go no further. We've reached the end of the chain of function calls. The implementation of `clearTotals` tells us exactly how the totals are cleared, in excruciating detail. There is no meaningful lower level to descend to.

Let's look instead at `makeDetails` .

```
func (statement *RentalStatement) makeDetails() strir
  rentalDetails := ""
  for _, rental := range statement.rentals {
    rentalDetails += statement.makeDetail(rental)
  }
  return rentalDetails
}
```

The name of this function tells us what the function does at this level of abstraction. The implementation creates a loop that calls yet another private method, `makeDetail` , which is one level of abstraction lower.

Thus, every public function creates a tree of calls to private functions at lower levels of abstraction.

Here, again, we see The Stepdown Rule. Each such descending call goes *one level of abstraction* down. Not two, not three. Just one. That's why the body of the loop within `makeDetails` calls the next function down. If I had placed the implementation of `makeDetail` within `makeDetails` , then `makeDetails` would have contained at least two levels of abstraction.

Do I always follow that rule? No, sometimes I allow two levels of abstraction within a function. Generally, however, I consider that to be rude.

The Newspaper Metaphor

Think of a well-written newspaper article. You read it vertically. At the top, you expect a headline that will tell you what the story is about and allow you to decide whether it is something you want to read. The first paragraph gives you a synopsis of the whole story, hiding all the details while giving you the broad-brush concepts. As you continue downward, the details increase until you have all the dates, names, quotes, claims, and other minutiae.

Most of us never read the entirety of an article. We start at the headline, and if it is interesting we read on. If that paragraph interests us we read on. And we stay in that very simple loop:

```
while (interested) readMore;
```

As soon as the article becomes boring, we stop. Articles that allow us to use that loop are *polite*—they let us out early. Articles that force us to waste time reading a lot of unnecessary detail are rude.

A newspaper is composed of many articles; most are very short. Some are a bit longer. Very few contain as much text as a page can hold. This makes the newspaper *usable*. If the newspaper were just one long story containing a disorganized agglomeration of facts, dates, and names, then we simply would not read it.

We would like the source code of a module to be like a newspaper article. The module's name should be simple but explanatory. The name, by itself, should be sufficient to tell us whether we are in the right module or not. The topmost parts of the module should provide the high-level concepts and algorithms. Detail should increase as we move downward,[2] until at the end we find the lowest-level functions and details in the module.

---

2. Some languages, such as Clojure, invert the vertical ordering because they require that declarations appear before their use. Thus, low-level details go at the top and the higher-level policies go toward the bottom.

A newspaper article that was a mixture of two different stories would be an abomination. So too is a module that contains two or more different topics.[3] So the classes and functions within a module should all be related to a single topic—a single responsibility—with a nice structure that proceeds from high-level policy to low-level detail.

---

3. The Single Responsibility Principle (SRP).

Be Polite

When you write functions that encapsulate a single level of abstraction, it allows your readers to escape early. They don't have to read a lot of code to understand what the function does at that level. Sometimes that level is all they care about, and they don't want to be bothered by all the details from the lower levels.

For example, I consider the `makeStatement` function above to be polite. The reader can glance at it and see what it does. If the reader is not worried about anything else, then they are done and go back to whatever else they were doing.

Now consider the original code that `makeStatement` evolved from.

```
func (customer *Customer) statement() string {
  totalAmount := 0.0
  frequentRenterPoints := 0
  result := "Rental Record for " + customer.name + "\
  for _, rental := range customer.rentals {
    thisAmount := 0.0
    switch rental.movie.movieType {
    case NewRelease:
      thisAmount += float64(rental.daysRented * 3)
    case Regular:
      thisAmount += 2
      if rental.daysRented > 2 {
        thisAmount += float64(rental.daysRented-2) *
      }
    case Childrens:
      thisAmount += 1.5
      if rental.daysRented > 3 {
        thisAmount += float64(rental.daysRented-3) *
      }
    }
    frequentRenterPoints++
    if rental.movie.movieType == NewRelease && rental
      frequentRenterPoints++
    }
    result += fmt.Sprintf("\t%s\t%.1f\n", rental.movi
    totalAmount += thisAmount
  }
  return result +
    fmt.Sprintf(
      "Amount owed is %.1f\nYou earned %d frequent re
      totalAmount, frequentRenterPoints)
}
```

I consider this code to be rude. I consider the author to have been inconsiderate. To understand the high-level intent of this code I must deal with all the low-level details—and that takes time.

Think of it this way. Let's say that you were reading code somewhere and you came across a call to the polite `makeStatement()`. The name of the function tells you what it does, but perhaps you'd like to check that the statement includes the footer. So you click over to the implementation of `makeStatement` and you see four simple lines of code that tell you exactly what you need to know. Two seconds later you click back to where you were and continue reading, suffering no feeling of interruption—just a simple question asked and answered in two seconds.

Now let's say that you were reading code somewhere and you came across a call to the rude `statement()` function. The name is a noun, which describes no action. You have to infer that the function builds the statement. Again, you're not sure if the statement includes the footer. So you click over to the implementation and are faced with 30 lines of deeply detailed code. Is the footer in there? Does the code that builds the footer get called at the right time? A minute or two later you've satisfied yourself that the footer is intact, but now you've forgotten why you were worried about that. You click back to where you were, and you've lost your train of thought and have to backtrack several lines, and several more minutes, to remind yourself of what you were doing and re-create that train of thought in your mind.

That's rude. It's as rude as the guy who bursts into your cubicle and throws a question at you without waiting for you to acknowledge him.

The Stepdown Rule: Once Again

So, the rule I like to follow, in order to be polite, is to descend one level of abstraction at a time. Each function has a name that defines what the function does, and each function consists of a set of lines that describe the implementation and that are all at the same level of abstraction. Each of those lines, if it is not already at the lowest level, calls functions at the next level down.

How do you determine how big one level of abstraction is? All I can say to that is to use your best judgment. For me, it's pretty simple. If I *can* meaningfully extract, then I do the extraction and check that it makes sense.

This is a polite practice, and it helps avoid the abstraction roller coaster.

The Abstraction Roller Coaster

Look again at that rude 30-line `statement` function. Right at the top, why do `totalAmount` and `frequentRenterPoints` need to be set to zero? What are we doing? This is the first line of a high-level function, and we are confronted with the lowest-level kind of detail. Then, we pop up a level to deal with the `result`, which we must infer to be the header, and then there's a loop. The first line of the loop descends immediately to another low-level detail and then enters a `switch` statement.

Can you feel the roller coaster? We are going up and down through levels of abstraction. Sometimes we are high, as we set up the loop through the rentals, and sometimes we are low, as we zero out totals and check individual cases of the switch.

Riding that roller coaster is not fun for the reader. The reason is simple. Every time you go from a line at a high level to a line at a low level, you have to push your mental model up on your mental stack to deal with that low-level concern. Once you understand it, then you pop your mental stack back to pick up where you left off.

The problem with that is that we do not have a mental stack. When we push our train of thought aside to deal with a detail, more often than not we lose that train of thought. Again, it's like that guy who bursts into your cubicle and shouts a question at you.

This Is How We Write, but Not How We Want to Read

The problem is that we write code on the roller coaster. We all do. We all must. Imagine the person writing that original rude `statement` function.

Perhaps the first line they wrote was the loop, but then they remembered to zero out the `totalAmount`. Later, after they wrote the switch, they realized they needed to deal with the frequent renter points, and so they went back to the top and added that line.

In short, we construct our code by rolling through the undulating track of abstractions and details. Our code comes out looking like the roller coaster. And this is where Kent Beck's advice becomes critical:

First, make it work. Then, make it right.

You are not done when the code works. You are done when the code reads well. You are done when you have extracted and organized and laid out the tree of functions descending from the public functions in a manner convenient to the reader.

Now, again, I don't follow these rules religiously or dogmatically. There are situations in which following them is inappropriate. But these rules are my default. I will follow them unless I have a good reason not to.