10

One Thing

Do you remember what Bjarne Stroustrup said about clean code? He said, "Clean code does one thing well." And, indeed, many other software authors have said things like that over the years. The problem with that statement is: What does *one thing* mean?

As I said in a previous chapter, I was once responsible for a C function that was 3,000 lines long. It was named `gi`, which stood for *graphic interpreter*. Had anyone challenged me about whether my 3,000-line function did one thing, I would have confidently replied: "Certainly! It interprets graphics."

Of course, a 3,000-line function must be doing a lot of *things* while it interprets graphics. I'm quite convinced that the problem of interpreting graphics can be broken down into quite a few *things*.

So how do we resolve this issue? If the advice is to be followed, we must have some reasonable definition of *one thing*. I think I know what that definition is, and I'm going to tell you now.

But, like Deep Thought said in *Hitchhiker's Guide to the Galaxy*: "You're not going to like it."

Extract Method Refactoring

Does your IDE have a *Refactor* menu? Mine does. One of the menu items is *Extract Method*. IDEs have had this menu item for two decades now. I'm sure you've all used it. But just in case, let me describe it for you.

If you see a function that's a little too big, and you decide you are going to split it into two functions, you can use your mouse to select the code you want to pull out (extract) from the parent function.[1] Then, you select the *Extract Method* refactoring and up pops a dialog box that asks you what name you want for the new function. That box also tells you what arguments it plans to pass into the new function, and also what it plans to return. The dialog box gives you the option to change a few of these things. When you are ready, you hit the *Refactor* button on that dialog box, and the IDE cuts the selected code out of the original function and replaces it with a call to the new function. Then, it pastes the selected code into that new function and creates the glue code that allows it to be called.

---

1. In 1974, no programmer could have made sense of that sentence. Forty years ago, some might have had a glimmer.

I mean … it's magic!

So, what does *one thing* mean?

A function does one thing if you cannot meaningfully extract another function from it.

OK, now disregard the word *meaningfully* for just a moment and think about that statement. Can you refute it? I don't think you can. If you can split one function into two, then the original function did two things. So, a function that does one thing is a function from which you cannot extract another.

OK, now, what does the word *meaningfully* mean in this context? Consider this function:

```
public void addRental(Rental rental) {
  rentals.addElement(rental);
}
```

It is not meaningful to extract the body of that function into a new function:

```
public void addRental(Rental rental) {
  doTheAdd(rental);
}

public void doTheAdd(Rental rental) {
  rentals.addElement(rental);
}
```

There is no point in extracting the entire body of a function, leaving behind a pure delegator. Such an extraction is not meaningful.[2]

---

[2]. Except in those rare circumstances where you are creating a delegator so that you can move the extracted function to a new module.

Or consider this function:

```
void clearTotals() {
  totalAmount=0;
  renterPoints=0;
}
```

It would almost certainly be meaningless to extract those two lines into two different functions:

```
void clearAmount() {totalAmount=0;}
void clearPoints() (renterPoints=0;}
```

This has taken extraction too far. The `clearAmount` and `clearPoints` functions have names that are indistinguishable from their implementations.

A meaningful abstraction is one that follows The Stepdown Rule, where the name of a function is more abstract than the implementation.

I'll leave the rest of the interpretation of the word *meaningful* up to you. Suffice it to say that you can extract too little, and you can extract too much.

So how do we ensure that all our functions do one thing? We extract, and extract, until we cannot meaningfully extract anymore. Or, in other words:

Extract till you drop!

OK, perhaps that's a little cheeky. I don't really want you to drop. On the other hand, I do want you to *consider* as many extractions as you can. You may not choose to do them all, but you should at least think about them.

As a consequence, the majority of your functions will be pretty small. How small? A few lines. That number might be three, or four, or six. It might be larger for certain `switch` statements, or certain formatting statements, or in certain languages. But, in general, following this recommendation will cause you to create *a lot* of *small* functions.

And what will you be extracting? Often you'll be extracting the bodies of `if`/`else` statements and `for`/`while` loops. You'll also likely be extracting the predicates out of those statements so that those statements will be composed of little more than the keywords and the function calls.

For example:

```
if (shouldDeleteRecord(r))
  deleteRecord(r);
```

If you read that carefully, you'll see that it reads like *well-written prose*.

This Shouldn't Be Controversial

Over the years, this particular recommendation, from the first edition of this book, has been the most controversial. But why? There are a lot of reasons.

1. People are afraid that if they follow this rule, they will drown beneath a sea of tiny little functions.
2. People are afraid that following this rule will obscure the intent of a function because they won't be able to see it all in one place.
3. People worry that the performance penalty of calling all those tiny little functions will cause their applications to run like molasses in the wintertime.
4. People think they're going to have to be bouncing around, from function to function, in order to understand anything.
5. People are concerned that the resulting functions will be too entangled to understand in isolation.

Let's address these one at a time.

1. Drowning

Long ago our languages were very primitive. The first version of FORTRAN, for example, allowed you to write one function. And it wasn't called a function. It was just called a program. There were no modules, no functions, and no subroutines. As I said, it was primitive.

COBOL wasn't much better. You could jump around from one *paragraph* to the next using a `goto` statement; but the idea of functions that you could call with arguments was still a decade in the future.

When Algol, and later, C, came along, the idea of functions burst upon us like a wave. Suddenly we could create modules that could be individually compiled. Modular programming became all the rage.[3]

---

3. And as with all things in software, it was badly abused. Some managers insisted that no module should be longer than 500 bytes. So programmers were forced to put a goto at the end of their modules in order to call the next.

But as the years went by, and our programs got larger and larger, we began to realize that functions weren't enough. Our names started to collide. We were having trouble keeping them unique. And we really didn't like the idea of prefixing every name with the name of its module. I mean, nobody likes `io_files_open` as a function name.

So we added classes and namespaces to our languages. These gave us lovely little cubbyholes that had names into which we could place our functions and our variables. Moreover, these namespaces and classes quickly became hierarchical. We could have functions within namespaces that were within yet other namespaces.

And this is the solution for the "sea of tiny little functions" fear. You aren't going to drown beneath that sea, because there won't be a sea. Instead, you'll have a nicely arranged and nicely named *hierarchy* of namespaces and classes into which you can tuck your nicely named functions. Those nice names will be the guideposts that other programmers will use to locate your functions. Better still, those guideposts will provide the context that other programmers will use to understand your functions.

Let me say this a different way. Some of us really like working on largish functions. There's something pleasing about the shape of those functions. We like the undulating in and out of the indentations. Something in our hindbrain finds comfort in it.

Why? Perhaps because, if you rotate the listing to the right by 90°, it looks like the horizon. Those undulations become peaks and valleys. And we, hominids who evolved on the savannah, used those shapes on the horizon to determine our location. We knew that between the two peaks to the east there was a watering hole; and the valley on the right was home to a ravenous saber-toothed tiger.

Remember my `gi` function? I knew that function like the back of my hand. I knew it *geographically*! If someone asked me to change the scaling on the *x*-axis, I would have said to myself: "The scaling on the *x*-axis … that's in the third indent after the big comment block." And I would have scrolled down to the big comment block and counted 1..2..3 and there it would be—the code that set the scaling on the *x*-axis.

Now take a new programmer who'd never seen `gi`. Ask that poor sod to change the scaling on the *x*-axis. Where the hell should he go? He doesn't know where the watering hole is. He's got no idea that there's a saber-toothed tiger on the loose in that valley over there. He's just going to fumble around inside that 3,000-line morass of code until, by luck, he stumbles upon something that looks like it might set the scaling on the *x*-axis.

Wouldn't it be great for that poor programmer if there were a function named `setScalingOnX` ? Wouldn't it be just spectacular if that function were located in the `Axis` namespace? I hope I've made my point. So, on to point 2.

2. Small Functions Don't Obscure Intent

The answer to point 2 is the same as the answer to point 1. You cannot obscure the intent of the code by putting that code into a nicely named organizational structure. A well-designed and well-named structure of namespaces, classes, and functions *reveals* intent. It does not obscure it. More on this later.

3. Performance

OK, what about point 3? Performance: This is a real fear. But it is real only for those programmers who must preserve and protect every nanosecond under their control. If you are writing a first-person shooter game, or a high-speed trading app, or the guidance system for a hypersonic missile, then nanoseconds can be pretty important.

On the other hand, if you are writing a travel website, or a payroll system, or even the autopilot for a small airplane, nanoseconds are not your concern.

Long ago a function call took several microseconds. Machines were slow back then. So we were careful about how many functions we called. But nowadays, function calls take a few nanoseconds. There's almost no way that you can appreciably slow your application down by extracting methods. And, if your measurements show you that there is a slowdown, all you have to do is inline the most critical of those functions back again.

And remember, good compilers will often do that inlining for you, without your knowledge. If a good compiler sees that you've called a function from just one place, it’s very likely to simply replace the call with the code within the function itself.

4. Bouncing Around

OK, then. Point 4: bouncing around. I agree with this point—or at least I *would* agree if the functions in question were randomly scattered hither and yon throughout the code. But who would do a thing like that? It’d be very rude.

I like to organize my extracted functions in the order that they are called. So, when one function calls another, all you have to do is scroll down to see the called function—right there. If function A calls B and C, then their order in the source file should be A, B, C.[4] How hard is that?

---

4. In languages like Clojure, it would be C, B, A.

So, no, if the author is polite, you are not going to be bouncing all around inside the code.

Instead, you’ll be able to read down the length of the code, from high level to low level. Not only that, but the high-level code will be organized and separated from the low-level code—because when you extract functions, you virtually always extract lower-level code from higher-level code.

Indeed, I’ll show you an example of that later in this chapter. But first I want to talk about one last thing.

5. Entanglement

Of all the complaints, this is the most reasonable, and it is the point that John Ousterhout makes in the appendix. It is undeniably true that when you extract one function from another, you *may* be entangling the logic of those two functions. To say that differently, the lower-level function might not be independent of the operation of the higher-level function. To understand the lower-level function you must have a good understanding, of the higher-level function.

If this entanglement is severe—that is if there are several different facts established by the higher-level function that you must keep in mind while reading the lower-level function—then there is a reasonable chance that the extraction might not be worth doing. However, if the entanglement is minimal, then the fact that the extracted function has a nice descriptive name and that fact that it appears directly after its parent function will likely make the extraction worthwhile.

Or to say this even more differently: It’s a judgment call. Choose wisely.

What Are Large Functions Anyway?

Once again, imagine my `gi` function from long ago. Let’s imagine that it was written in Java and that it began like this:

```
public class GraphicInterpreter {
  public void gi() {
    int i;
    int j;

    …
  }
}
```

As we scroll down through the 3,000 lines of the `gi` function, we notice an `if` statement:

```
    if (…) {
      …
      i++;
      j++;
      …
    }
```

We decide we’d like to extract the body of that `if` statement into a new method. So we select the body with our mouse, and we invoke the Extract Method menu item. But the IDE complains that it cannot extract that method because the code modifies *two* local variables. Had the code modified only one local variable, the IDE could have returned the new value from the extracted method. But the IDE has no good way to return two variables. (This is Java and not Go, after all).

So, what should we do? How are we going to extract the body of that `if` statement into a new method? Well, what if we did this:

```
public class GraphicInterpreter {
  private int i;
  private int j;

  public void gi() {
    …
    if (…) {
      …
      i++;
      j++;
      …
    }
  }
}
```

I know what you are thinking: “Oh no! You promoted local variables into instance variables! You are violating encapsulation!” True. I am. But now I can extract the body of my `if` statement into a new function because the two variables remain in scope within the class:

```
public class GraphicInterpreter {
  private int i;
  private int j;
  public void gi() {

    …
    if (…) {
      f();
    }
  }

  private void f() {
    …
    i++;
    j++;
    …
    }
  }
```

Let's keep doing this. We find more and more `if` and `while` statements whose bodies manipulate `i` and `j`, and that we can extract. Indent after indent we extract and extract and extract. And what is the result?

The result is a whole bunch of methods that use the instance variables `i` and `j`. What do you call it when you've got a bunch of methods that manipulate a set of variables? You call that a class, don't you?

Now, think about this carefully.

Every large function is really hiding a class (or more) inside it.

Because every large function has a set of local variables and a set of indented regions that manipulate those variables. Those variables should really be fields of a class, and those indented regions should be methods of that class.

But to channel George Carlin, perhaps I've gone a bit too far, a bit too fast. So let's look at an example.

Allow me to introduce the famous Video Store example from the first edition of Martin Fowler's wonderful book, *Refactoring*. Back in 1999 Martin wrote this little example in Java. Here, we're going to see it in Go. But never fear, if you don't know Go, it's pretty easy to understand.

First come the tests, which Martin did *not* include in his book so long ago.

```
  ——customer_test.go——
  package GoVideoStore

  import (
    "testing"
  )

  var customer *Customer

  func setup() {
    customer = NewCustomer("Fred")
  }
```

```
func assertEquals(t *testing.T, expected string, stat
  if statement != expected {
    t.Errorf("\nExpected:\n%s\n\nGot:\n%s", expected,
  }
}

func TestSingleNewReleaseStatement(t *testing.T) {
  setup()
  customer.addRental(NewRental(NewMovie("The Cell", N
  assertEquals(t,
    "Rental Record for Fred\n"+
      "\tThe Cell\t9.0\n"+
      "Amount owed is 9.0\n"+
      "You earned 2 frequent renter points",
    customer.statement())
}

func TestDualNewReleaseStatement(t *testing.T) {
  setup()
  customer.addRental(NewRental(NewMovie("The Cell", N
  customer.addRental(NewRental(
                     NewMovie("The Tigger Movie", N
  assertEquals(t,
    "Rental Record for Fred\n"+
      "\tThe Cell\t9.0\n"+
      "\tThe Tigger Movie\t9.0\n"+
      "Amount owed is 18.0\n"+
      "You earned 4 frequent renter points",
    customer.statement())
}

func TestSingleChildrensStatement(t *testing.T) {
  setup()
  customer.addRental(NewRental(
                     NewMovie("The Tigger Movie", C
  assertEquals(t,
    "Rental Record for Fred\n"+
      "\tThe Tigger Movie\t1.5\n"+
      "Amount owed is 1.5\n"+
      "You earned 1 frequent renter points",
    customer.statement())
}

func TestMultipleRegularStatement(t *testing.T) {
  setup()
  customer.addRental(NewRental(
                     NewMovie("Plan 9 from Outer Sp
                     1))
  customer.addRental(NewRental(NewMovie("8 1/2", Regu
  customer.addRental(NewRental(NewMovie("Eraserhead",
  assertEquals(t,
    "Rental Record for Fred\n"+
      "\tPlan 9 from Outer Space\t2.0\n"+
      "\t8 1/2\t2.0\n"+
      "\tEraserhead\t3.5\n"+
      "Amount owed is 7.5\n"+
      "You earned 3 frequent renter points",
    customer.statement())
}
```

And then there's the code that passes these tests.

```
———Customer.go———
package GoVideoStore

import "fmt"

type Customer struct {
  name    string
  rentals []*Rental
}

func NewCustomer(name string) *Customer {
  return &Customer{name: name}
}

func (customer *Customer) addRental(rental *Rental) {
  customer.rentals = append(customer.rentals, rental)
}

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)
}
———Rental.go———
package GoVideoStore

type Rental struct {
  movie      *Movie
  daysRented int
}

func NewRental(movie *Movie, daysRented int) *Rental
  return &Rental{movie: movie, daysRented: daysRentec
}

———Movie.go———
package GoVideoStore

const (
  Regular int = iota
  NewRelease
  Childrens
)
```

```
type Movie struct {
  title     string
  movieType int
}

func NewMovie(title string, movieType int) *Movie {
  return &Movie{title: title, movieType: movieType}
}
```

The first thing we're going to do is attack the tests. The tests need a better design, because those tests are a *mess*! Here's why.

- The tests are all conducted through the UI. The statement strings *are* the UI, and they are volatile. Marketing people will change those strings on a whim. For example, if a marketing person says that the title of each statement should be `Rental Statement for…` instead of `Rental Record for…`, every test will break and will have to be changed. That makes the tests fragile. So, instead of testing the format of the statement many times, we should only test it once. The other tests should focus on the values being computed.
- Names like `Fred` and `The Cell` tell us nothing about what is being tested. Those names should be changed to help the reader of the tests understand what the tests are about. In general, tests should not contain production data. They should contain data that explains the intent of the test.
- The tests are wordy. We should be able to tighten them up a bit by using some variables for the movies, and a *composed assertion*.

```
———customer_test.go———
package GoVideoStore

import (
  "testing"
)

var customer *Customer
var newRelease1 *Movie
var newRelease2 *Movie
var childrensMovie *Movie
var regular1 *Movie
var regular2 *Movie
var regular3 *Movie
```

```
func setup() {
  customer = NewCustomer("Customer Name")
  newRelease1 = NewMovie("New Release 1", NewRelease)
  newRelease2 = NewMovie("New Release 2", NewRelease)
  childrensMovie = NewMovie("Childrens", Childrens)
  regular1 = NewMovie("Regular 1", Regular)
  regular2 = NewMovie("Regular 2", Regular)
  regular3 = NewMovie("Regular 3", Regular)
}

func assertStringsEqual(t *testing.T, expected string
  if actual != expected {
    t.Errorf("\nExpected:\n%s\n\nGot:\n%s", expected,
  }
}

func assertIntsEqual(t *testing.T, expected int, actu
  if actual != expected {
    t.Errorf("\nExpected:%d Got:%d", expected, actual
  }
}

func assertFloatsEqual(t *testing.T, expected float64
{
  if actual != expected {
    t.Errorf("\nExpected:%f Got:%f", expected, actual
  }
}

func assertOwedAndPoints(t *testing.T, owed float64,
  customer.statement()
  assertFloatsEqual(t, owed, customer.totalAmount)
  assertIntsEqual(t, points, customer.frequentRenterP
}

func TestTotalsForOneNewRelease(t *testing.T) {
  setup()
  customer.addRental(NewRental(newRelease1, 3))
  assertOwedAndPoints(t, 9.0, 2)
}

func TestTotalsForTwoNewReleases(t *testing.T) {
  setup()
  customer.addRental(NewRental(newRelease1, 3))
  customer.addRental(NewRental(newRelease2, 3))
  assertOwedAndPoints(t, 18.0, 4)
}

func TestTotalsForOneChildrensMovie(t *testing.T) {
  setup()
  customer.addRental(NewRental(childrensMovie, 3))
  assertOwedAndPoints(t, 1.5, 1)
}

func TestTotalsForManyRegularMovies(t *testing.T) {
  setup()
  customer.addRental(NewRental(regular1, 1))
  customer.addRental(NewRental(regular2, 2))
  customer.addRental(NewRental(regular3, 3))
  assertOwedAndPoints(t, 7.5, 3)
}

func TestStatementFormat(t *testing.T) {
  setup()
  customer.addRental(NewRental(regular1, 1))
  customer.addRental(NewRental(regular2, 2))
  customer.addRental(NewRental(regular3, 3))
  assertStringsEqual(t,
    "Rental Record for Customer Name\n"+
      "\tRegular 1\t2.0\n"+
      "\tRegular 2\t2.0\n"+
      "\tRegular 3\t3.5\n"+
      "Amount owed is 7.5\n"+
      "You earned 3 frequent renter points",
    customer.statement())
}
```

Remember, in the discussion about `gi`, how I broke encapsulation by moving some local variables into fields of the containing class? The changes I made to the tests caused me to do the same thing to the production code. I moved the `totalAmount` and `frequentRenterPoints` local variables into instance variables of the `Customer` class.

```
——Customer.go——
…
type Customer struct {
  name                 string
  rentals              []*Rental
  totalAmount          float64
  frequentRenterPoints int
}
…
```

You may recall that the reason I said we needed to break encapsulation was in order to support extraction. So let's get to that extraction. You're going to see me *extract till I drop*. Here's the first pass.

```
———Customer.go———
package GoVideoStore

import "fmt"

type Customer struct {
  name                 string
  rentals              []*Rental
  totalAmount          float64
  frequentRenterPoints int
}

func NewCustomer(name string) *Customer {
  return &Customer{name: name}
}

func (customer *Customer) addRental(rental *Rental) {
  customer.rentals = append(customer.rentals, rental)
}

func (customer *Customer) makeStatement() string {
  customer.clearTotals()
  return
    customer.makeHeader() +
    customer.makeDetails() +
    customer.makeFooter()
}

func (customer *Customer) clearTotals() {
  customer.totalAmount = 0.0
  customer.frequentRenterPoints = 0
}
```

```
func (customer *Customer) makeHeader() string {
  return "Rental Record for " + customer.name + "\n"
}

func (customer *Customer) makeDetails() string {
  rentalDetails := ""
  for _, rental := range customer.rentals {
    rentalDetails += customer.makeDetail(rental)
  }
  return rentalDetails
}

func (customer *Customer) makeDetail(rental *Rental)
  amount := customer.determineAmount(rental)
  customer.frequentRenterPoints += customer.determine
  customer.totalAmount += amount
  return customer.formatDetail(rental, amount)
}

func (customer *Customer) determineAmount(rental *Ren
  amount := 0.0
  switch rental.movie.movieType {
  case NewRelease:
    amount += float64(rental.daysRented * 3)
  case Regular:
    amount += 2
    if rental.daysRented > 2 {
      amount += float64(rental.daysRented-2) * 1.5
    }
  case Childrens:
    amount += 1.5
    if rental.daysRented > 3 {
      amount += float64(rental.daysRented-3) * 1.5
    }
  }
  return amount
}

func (customer *Customer) determinePoints(rental *Ren
  if rental.movie.movieType == NewRelease && rental.d
    return 2
  }
  return 1
}
```

```
func (customer *Customer)
formatDetail(rental *Rental, amount float64) string {
  return fmt.Sprintf("\t%s\t%.1f\n", rental.getTitle(
}

func (customer *Customer) makeFooter() string {
  return fmt.Sprintf("Amount owed is %.1f\n"+
    "You earned %d frequent renter points",
    customer.totalAmount, customer.frequentRenterPoir
}
```

I imagine that some of you who are fluent in Golang are now running away in horror.[5] If so, I apologize—I am not an accomplished Golang programmer. Others of you might be screaming as you run down the hall because of all those *little tiny functions*. Aaaarrrgggghhh!

---

5. See the Future Bob note near the end of this example.

But wait. Go back, right now, and read that `makeStatement` function. I'll wait.

How do you make a statement? You clear the totals and then compose the header, details, and footer together into a single string.

Now, if you think about it, *that's the high-level procedure*. Indeed, that's the high-level procedure for most reports—because most reports are composed of a header, a set of details, and a footer.

So, to those of you who feared that all this extracting would hide the intent of the code, I suggest quite the opposite has happened. The extraction has *exposed* the high-level intent of the code and simply moved the lower-level details into other functions, where they cannot distract you. If you have been told to fix a bug in the footer of this statement, you now know exactly where to go: the `makeFooter` function.

This code is polite. It is polite because, when the poor reader looks at it, there's little to decode. The reader does not have to pore and ponder over strings and variables and loops and `if` statements. Instead, the reader sees a simple statement of policy with directions pointing to the answers to any questions they might have.

Because this code is polite, it's fair to say that the original code was rude. And we don't want to be rude, do we?

Let's continue. It's pretty easy to see how to `clearTotals`. It's also pretty obvious how to make the header and the footer. Making the details is just a loop over all the rentals that simply calls `makeDetail`.

How do you make a detail? You determine the amount, then you determine the points, and then you format the detail. Yikes! How obvious can you get?

Formatting the detail is pretty obvious too. But those two `determine` functions are a bit problematic.

Look through all those methods. The first argument is always `customer *Customer`. This is just Go's way of saying that the function is part of the `Customer` class. Now look at the bodies of all the functions. Notice that they all use the `customer` argument—except for `formatDetail`, `determineAmount`, and `determinePoints`. We'll deal with `formatDetail` later. For now, there's an awful lot of code in those two `determine` functions that doesn't deal with the `Customer` class that they belong to. So … maybe they don't belong to that class.

What else gets passed into those `determine` functions? The `Rental`! So maybe those two `determine` functions belong in the `Rental` class. Let's move them there.

```
——Customer.go——
func (customer *Customer) makeDetail(rental *Rental)
  amount := rental.determineAmount()
  customer.frequentRenterPoints += rental.determinePc
  customer.totalAmount += amount
  return customer.formatDetail(rental, amount)
}

——Rental.go——
Package GoVideoStore

type Rental struct {
  movie      *Movie
  daysRented int
}
```

```
func (rental Rental) getTitle() string {
  return rental.movie.title
}

func (rental Rental) determineAmount() float64 {
  amount := 0.0
  switch rental.movie.movieType {
  case NewRelease:
    amount += float64(rental.daysRented * 3)
  case Regular:
    amount += 2
    if rental.daysRented > 2 {
      amount += float64(rental.daysRented-2) * 1.5
    }
  case Childrens:
    amount += 1.5
    if rental.daysRented > 3 {
      amount += float64(rental.daysRented-3) * 1.5
    }
  }
  return amount
}

func (rental Rental) determinePoints() int {
  if rental.movie.movieType == NewRelease && rental.c
    return 2
  }
  return 1
}

func NewRental(movie *Movie, daysRented int) *Rental
  return &Rental{movie: movie, daysRented: daysRentec
}
```

The tests still pass, unchanged. The tests have no idea that this fundamental change in design has happened. That means the tests have a design that tolerates change!

However, something even more magical has taken place. All the business rules that were in the `Customer` class have fled to the `Rental`. The `Customer` class is now all about the formatting of the statement—the business calculations, other than a few sums, are now done in the `Rental`.

This calls into question why the `Customer` class was given that name. It's not about the customer anymore—if it ever was to begin with. Now it's all about the rental statement. So that's what its name should be: `RentalStatement`.

```
——RentalStatement.go——
…
type RentalStatement struct {
  name                 string
  rentals              []*Rental
  totalAmount          float64
  frequentRenterPoints int
}
…
```

But now let's consider that `Rental` class. The `determine` functions are strongly coupled to the `Rental` class in that they use both the `movie` and the `daysRented` fields. That suggests a strong cohesion. However, the `determine` functions are very strongly coupled to the `Movie` class as well, and that's a bit troubling.

The coupling to the `Movie` is all about the `movieType`. That type field is used to select the various business rules for calculating the amount and the points. So why are the type fields and the type constants in the `Movie` class?

The original tests contain a hint. If you look back, you'll see that `"The Tigger Movie"` was considered both a `NewRelease` and a `Childrens` movie. That suggests that a movie can be rented as a new release and later rented as a children's movie. And that suggests that the type field actually belongs to the `Rental`. Let's try that.

```
——Rental.go——
…
type Rental struct {
  movie      *Movie
  daysRented int
  rentalType int
}
```

```
const (
  Regular int = iota
  NewRelease
  Childrens
)
…
func NewRental(movie *Movie, rentalType int, daysRent
  return &Rental{
    movie:      movie,
    rentalType: rentalType,
    daysRented: daysRented}
}
…
```

This change affected the tests because the `NewRental` constructor was changed to include the type. But overall this was a pretty quick change. However, it begs another question. Whenever you have a `switch` statement that acts on that type code, you might want to consider a polymorphic dispatch.

Why? Because `switch` statements are like gerbils—given enough time they'll reproduce all over your code and fill it to the brim. Indeed, there's already a baby `switch` statement growing in the `determinePoints` method.

In Chapter 12, "Objects and Data Structures," we'll talk about the pros and cons of `switch` statements. For now, let's see if we can get rid of these by creating a type hierarchy.

```
——RentalType.go——
package GoVideoStore

type RentalType interface {
  determineAmount(daysRented int) float64
  determinePoints(daysRented int) int
}

——Rental.go——
package GoVideoStore

type Rental struct {
  movie      *Movie
  daysRented int
  rentalType RentalType
}

func (rental Rental) getTitle() string {
  return rental.movie.title
}

func (rental Rental) determineAmount() float64 {
  return rental.rentalType.determineAmount(rental.day
}

func (rental Rental) determinePoints() int {
  return rental.rentalType.determinePoints(rental.day
}

func NewRental(movie *Movie,
               rentalType RentalType,
               daysRented int) *Rental {
  return &Rental{
    movie:      movie,
    rentalType: rentalType,
    daysRented: daysRented}
}

———NewReleaseRental.go———
package GoVideoStore

type NewReleaseRental struct{}

func (rentalType NewReleaseRental)
determineAmount(daysRented int) float64 {
  return float64(daysRented * 3)
}

func (rentalType NewReleaseRental) determinePoints(da
  if daysRented > 1 {
    return 2
  }
  return 1
}

———ChildrensRental.go———
package GoVideoStore
```

```
type ChildrensRental struct{}

func (rentalType ChildrensRental)
determineAmount(daysRented int) float64 {
  amount := 1.5
  if daysRented > 3 {
    amount += float64(daysRented-3) * 1.5
  }
  return amount
}

func (rentalType ChildrensRental) determinePoints(day
  return 1
}

——RegularRental.go——
package GoVideoStore

type RegularRental struct{}

func (rentalType RegularRental) determineAmount(daysR
{
  amount := 2.0
  if daysRented > 2 {
    amount += float64(daysRented-2) * 1.5
  }
  return amount
}

func (rentalType RegularRental) determinePoints(daysR
  return 1
}
```

That’s a nice polymorphic dispatch. Now all the business rules for each type are safely encapsulated within their own type, and the `Rental` couldn’t care less which one it is using. Of course, the tests had to change, but only a little bit. Here’s just one to give you the idea.

```
——RentalStatementTest.go——
func TestTotalsForOneNewRelease(t *testing.T) {
  setup()
  statement.addRental(NewRental(newRelease1, NewRelea
  assertOwedAndPoints(t, 9.0, 2)
}
```

The `Movie` class is virtually empty at this point.

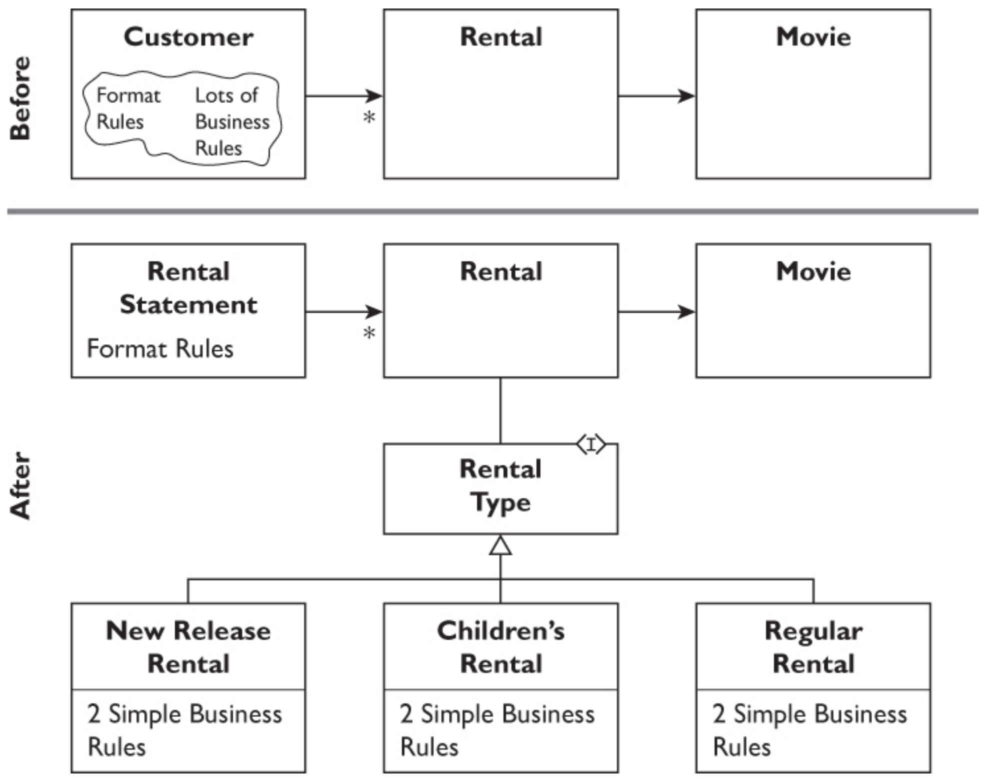
```
——Movie.go——
package GoVideoStore

type Movie struct {
  title string
}

func NewMovie(title string) *Movie {
  return &Movie{title: title}
}
```

And the `RentalStatement` class is concerned only with the formatting of the statement—which is why the `FormatDetail` function belongs there even though it doesn't use any of the `RentalStatement` 's fields.

Now, let's consider what we've just done. The diagram below shows a before-and-after picture.



Our cleanup procedure extracted the business rules and separated them from the formatting rules. Then, it moved those business rules, first into the

`Rental` and then decomposed them down into the polymorphic hierarchy. Nice! And in the midst of all that, we realized that some names and concepts were wrong—and that's pretty typical. When you clean things up, you see things that were hidden before.

Finally, notice that there is an architectural boundary forming below the `RentalType`. I didn't draw it, but you ought to be able to sense it. All those little business rule details are isolated from the higher-level policies in `RentalStatement` and `Rental`. Now when the requirements change, we can add lots of new `RentalType`s; and none of this code will need to change to accommodate those changes. That's the Open–Closed Principle (OCP) at work.

So, remember when I said that large functions are always hiding classes? That's what we found here. That one large `statement` function in the original code got broken up and distributed into three new classes and one existing class.

Future Bob:

Robert Laszczak happened to be perusing GitHub and saw the above meager attempt at Golang. He has kindly offered a blog post with improvements to this example that are more consistent with common Golang style. You can see this at https://threedots.tech/clean-code/. For my part, as a Golang novice, I agree with most of his changes, but I prefer to keep my names a bit longer, and prefix my getters with `get`.

Extraction and Classes

Consider the following Java code for solving the quadratic formula. The class is just a convenient namespace for the `solve` function.

```
import java.util.Arrays;
import java.util.List;

import static java.util.Arrays.asList;

public class QuadraticFormula {
  public static List<Double> solve(double a, double b
    if (a == 0) {
      if (b == 0) {
        return asList();
      }
      return asList(-c / b);
    } else {
      double d = b * b - 4 * a * c;
      if (d > 0) {
        double sqrtd = Math.sqrt(d);
        double x1 = (-b + sqrtd) / (2 * a);
        double x2 = (-b - sqrtd) / (2 * a);
        return asList(x1, x2);
      } else if (d == 0) {
        return asList(-b / (2 * a));
      } else {
        return asList();
      }
    }
  }
}
```

This code is pretty straightforward, but let's pretend that it's more complicated and involved than shown here. Let's also assume that it is going to *grow* with time. We might improve the readability by doing a few simple extractions and some renaming.

```
  import java.util.List;
  import static java.util.Arrays.asList;

  public class QuadraticFormula {
    public static List<Double> solve(double a, double b
      if (a == 0)
        return linearSolution(b, c);
      return quadraticSolution(a, b, c);
    }

    private static List<Double> linearSolution(double b
      if (b == 0)
        return asList();
      return asList(-c / b);
    }

    private static
    List<Double> quadraticSolution(double a, double b,
      double discriminant = b * b - 4 * a * c;
      if (discriminant < 0)
        return complexSolution();
      if (discriminant == 0)
        return singleQuadraticSolution(a, b);
      return twoQuadraticSolutions(a, b, discriminant);
    }

    private static List<Double> complexSolution() {
      return asList();
    }

    private static
    List<Double> singleQuadraticSolution(double a, douk
      return asList(-b / (2 * a));
    }

    private static List<Double>
    twoQuadraticSolutions(double a, double b, double di
      double sqrtd = Math.sqrt(discriminant);
      double x1 = (-b + sqrtd) / (2 * a);
      double x2 = (-b - sqrtd) / (2 * a);
      return asList(x1, x2);
    }
  }
```

Now again, for the sake of argument, we are pretending that this is a much more involved system that is likely to grow. I hope you agree that the extractions have lent some organization to the code that might make something more complicated a bit easier to read. But having done this, we are missing the opportunity of taking advantage of the class. Rather than passing arguments between all those extracted functions, we can load those values into variables within the class.

```
import java.util.List;

import static java.util.Arrays.asList;

public class QuadraticFormula {
  private static double a, b, c, discriminant;

  public static List<Double> solve(double a, double k
    QuadraticFormula.a = a;
    QuadraticFormula.b = b;
    QuadraticFormula.c = c;
    if (a == 0)
      return linearSolution();
    return quadraticSolution();
  }

  private static List<Double> linearSolution() {
    if (b == 0)
      return asList();
    return asList(-c / b);
  }

  private static List<Double> quadraticSolution() {
    discriminant = b * b - 4 * a * c;
    if (discriminant < 0)
      return complexSolution();
    if (discriminant == 0)
      return singleQuadraticSolution();
    return twoQuadraticSolutions();
  }

  private static List<Double> complexSolution() {
    return asList();
  }

  private static List<Double> singleQuadraticSolutior
    return asList(-b / (2 * a));
  }

  private static List<Double> twoQuadraticSolutions()
    double sqrtd = Math.sqrt(discriminant);
    double x1 = (-b + sqrtd) / (2 * a);
    double x2 = (-b - sqrtd) / (2 * a);
    return asList(x1, x2);
  }
}
```

This makes things just a little bit cleaner. In a more complex module, this could make things *a lot* cleaner. However, there is a risk. Those static variables could get corrupted in a multithreaded environment. This is easy enough to correct by changing all the functions and variables from `static` to instance methods and variables. But I'm not going to worry about that for this example.

The bottom line is that sometimes when we extract functions, it is convenient and clean to allow those functions to communicate through variables in the scope (the class) that contains them.

So now let's look at this problem in a *functional* language like Clojure.

```
(defn quad [a b c]
  (let [discriminant (- (* b b) (* 4 a c))]
    (cond
      (zero? a)
      (if (zero? b)
        []
        [(/ (- c) b)])

      (zero? discriminant)
      [(/ (- b) (* 2 a))]

      (neg? discriminant)
      []

      :else
      (let [sqrt-desc (Math/sqrt discriminant)
            x1 (/ (+ (- b) sqrt-desc) (* 2 a))
            x2 (/ (- (- b) sqrt-desc) (* 2 a))]
        [x1 x2]))))
```

This is very pretty. I would not be tempted to improve this by extracting. But again, let's pretend that this is a much larger and more involved function. How might we extract functions to improve it?

```
(defn quad [a b c]
  (let [discriminant (- (* b b) (* 4 a c))
        linear-solution
        (fn []
          (if (zero? b)
            []
            [(/ (- c) b)]))

        single-solution
```

```
          (fn []
            [(/ (- b) (* 2 a))])

          two-solutions
          (fn []
            (let [sqrt-desc (Math/sqrt discriminant)
                  x1 (/ (+ (- b) sqrt-desc) (* 2 a))
                  x2 (/ (- (- b) sqrt-desc) (* 2 a))]
              [x1 x2]))]

      (cond
        (zero? a) (linear-solution)
        (zero? discriminant) (single-solution)
        (neg? discriminant) []
        :else (two-solutions))))
```

OK, now for the last time, keep in mind that we are pretending that this module is larger and more complicated. That last `cond` statement is pretty nice. Notice that we're not passing arguments to the extracted functions. That's because a function in Clojure creates a kind of object that has variables and functions scoped within it. Notice also that those variables are not static, and so our concurrency worries disappear.

Conclusion

So, should you practice Extract Till You Drop? And if so, how much?

To some extent this is a matter of style and preference. You may like your functions a bit bigger than I like mine. Perhaps you are more comfortable with 20-line functions or 25-line functions. That's up to you.

However, remember what you just saw. It was the extraction of all those methods in the Video Store example that allowed us to see that some of that code did not belong in the class it was in. That was not evident before we did the extractions. So, if you are going to leave your functions larger, make sure you don't miss those kinds of opportunities. In fact, you may find it worthwhile to extract first and then do a little strategic inlining once you have determined where everything actually belongs.

But remember, when you inline a function, you destroy its name.