Chapter 3 - Functions

Functions are the first line of organization in any topic.

Small!!

The first rule of functions is that they should be small. The second rule of functions is that they should be smaller than that.

Blocks and Indenting

This implies that the blocks within `if` statements, `else` statements, `while` statements, and so on should be one line long. Probably that line should be a function call. Not only does this keep the enclosing function small, but also adds documentary value because the function called within the block can have a nicely descriptive name.
This also implies that functions should not be large enough to hold nested structures. Therefore, the indent level of a function should not be greater than one or two. This, of course, makes the functions easy to read and understand.

Do One Thing

FUNCTIONS SHOULD DO ONE THING. THEY SHOULD DO IT WELL. THEY SHOULD DO IT ONLY.

Sections within Functions

If you have a function divided in sections like *declarations*, *initialization* etc, it's a obvious symptom of the function is doing more than one thing. Functions that do one thing cannot be reasonably divided into sections.

One Level of Abstraction per Function

In order to make sure our functions are doing "one thing", we need to make sure that the statements within our function are all at the same level of abstraction.

Reading Code from Top to Bottom: The Stepdown Rule

We want the code to read like a top-down narrative. 5 We want every function to be followed by those at the next level of abstraction so that we can read the program, descending one level of abstraction at a time as we read down the list of functions.

To say this differently, we want to be able to read the program as though it were a set of TO paragraphs, each of which is describing the current level of abstraction and referencing subsequent TO paragraphs at the next level down.

```
- To include the setups and teardowns, we include setups, then we include the test page
content, and then we include the teardowns.
- To include the setups, we include the suite setup if this is a suite, then we include the
regular setup.
- To include the suite setup, we search the parent hierarchy for the “SuiteSetUp” page and add
an include statement with the path of that page.
- To search the parent...
```

It turns out to be very difficult for programmers to learn to follow this rule and write functions that stay at a single level of abstraction. But learning this trick is also very important. It is the key to keeping functions short and making sure they do “one thing.” Making the code read like a top-down set of TO paragraphs is an effective technique for keeping the abstraction level consistent.

Switch Statements

It’s hard to make a small switch statement. 6 Even a switch statement with only two cases is larger than I’d like a single block or function to be. It’s also hard to make a switch statement that does one thing. By their nature, switch statements always do N things. Unfortunately we can’t always avoid switch statements, but we can make sure that each switch statement is buried in a low-level class and is never repeated. We do this, of course, with polymorphism.

Use Descriptive Names

You know you are working on clean code when each routine turns out to be pretty much what you expected

Half the battle to achieving that principle is choosing good names for small functions that do one thing. The smaller and more focused a function is, the easier it is to choose a descriptive name.

Don’t be afraid to make a name long. A long descriptive name is better than a short enigmatic name. A long descriptive name is better than a long descriptive comment. Use a naming convention that allows multiple words to be easily read in the function names, and then make use of those multiple words to give the function a name that says what it does.

Choosing descriptive names will clarify the design of the module in your mind and help you to improve it. It is not at all uncommon that hunting for a good name results in a favorable restructuring of the code.

Function arguments

The ideal number of arguments for a function is zero (niladic). Next comes one (monadic), followed closely by two (dyadic). Three arguments (triadic) should be avoided where possible. More than three (polyadic) requires very special justification—and then shouldn’t be used anyway.

Arguments are even harder from a testing point of view. Imagine the difficulty of writing all the test cases to ensure that all the various combinations of arguments work properly. If there are no arguments, this is trivial. If there's one argument, it's not too hard. With two arguments the problem gets a bit more challenging. With more than two arguments, testing every combination of appropriate values can be daunting.

Output arguments are harder to understand than input arguments. When we read a function, we are used to the idea of information going in to the function through arguments and out through the return value. We don't usually expect information to be going out through the arguments. So output arguments often cause us to do a double-take.

Common Monadic Forms

There are two very common reasons to pass a single argument into a function. You may be asking a question about that argument, as in `boolean fileExists("MyFile")` . Or you may be operating on that argument, transforming it into something else and returning it. For example, `InputStream fileOpen("MyFile")` transforms a file name `String` into an `InputStream` return value. These two uses are what readers expect when they see a function. You should choose names that make the distinction clear, and always use the two forms in a consistent context.

Flag Arguments

Flag arguments are ugly. Passing a boolean into a function is a truly terrible practice. It immediately complicates the signature of the method, loudly proclaiming that this function does more than one thing. It does one thing if the flag is `true` and another if the flag is `false`!

Dyadic Functions

A function with two arguments is harder to understand than a monadic function. For example, `writeField(name)` is easier to understand than `writeField(output-Stream, name)`
There are times, of course, where two arguments are appropriate. For example, `Point p = new Point(0,0);` is perfectly reasonable. Cartesian points naturally take two arguments.
Even obvious dyadic functions like assertEquals(expected, actual) are problematic. How many times have you put the actual where the expected should be? The two arguments have no natural ordering. The expected, actual ordering is a convention that requires practice to learn.

Dyads aren't evil, and you will certainly have to write them. However, you should be aware that they come at a cost and should take advantage of what mechanims may be available to you to convert them into monads. For example, you might make the writeField method a member of outputStream so that you can say outputStream. writeField(name) . Or you might make the outputStream a member variable of the current class so that you don't have to pass it. Or you might extract a new class like FieldWriter that takes the outputStream in its constructor and has a write method.

Triads

Functions that take three arguments are significantly harder to understand than dyads. The issues of ordering, pausing, and ignoring are more than doubled. I suggest you think very carefully before creating a triad.

Argument Objects

Compare:

```
Circle makeCircle(double x, double y, double radius);
```
vs

```
Circle makeCircle(Point center, double radius);
```

Verbs and Keywords

Choosing good names for a function can go a long way toward explaining the intent of the function and the order and intent of the arguments. In the case of a monad, the function and argument should form a very nice verb/noun pair. For example, `write(name)` is very evocative. Whatever this “name” thing is, it is being “written.” An even better name might be `writeField(name)` , which tells us that the "name" thing is a "field".
This last is an example of the keyword form of a function name. Using this form we encode the names of the arguments into the function name. For example, `assertEquals` might be better written as `assertExpectedEqualsActual(expected, actual)`. This strongly mitigates the problem of having to remember the ordering of the arguments.

Output Arguments

In general output arguments should be avoided. If your function must change the state of something, have it change the state of its owning object.

Command Query Separation

Functions should either do something or answer something, but not both. Either your function should change the state of an object, or it should return some information about that object. Doing both often leads to confusion.

Prefer Exceptions to Returning Error Codes

Returning error codes from command functions is a subtle violation of command query separation.

Don't Repeat Yourself

Duplication may be the root of all evil in software. Many principles and practices have been created for the purpose of controlling or eliminating it.

Structured Programming

Some programmers follow Edsger Dijkstra's rules of structured programming. Dijkstra said that every function, and every block within a function, should have one entry and one exit. Following these rules means that there should only be one return statement in a function, no `break` or `continue` statements in a loop, and never, *ever*, any `goto` statements.
While we are sympathetic to the goals and disciplines of structured programming, those rules serve little benefit when functions are very small. It is only in larger functions that such rules provide significant benefit.

So if you keep your functions small, then the occasional multiple `return` , `break` , or `continue` statement does no harm and can sometimes even be more expressive than the single-entry, single-exit rule. On the other hand, `goto` only makes sense in large functions, so it should be avoided