Notes of Chapter 03 – Functions

Functions are the first line of organization in any program.

what is it that makes a function easy to read and understand? How can we make a function communicate its intent? What attributes can we give our functions that will allow a casual reader to intuit the kind of program they live inside?

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 it 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 easier to read and understand.

Do One Thing:

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

the reason we write functions is to
decompose a larger concept (in other words, the name of the function) into a set of steps at
the next level of abstraction.

Here is a way to know that a function is doing more than “one thing” is if you can
extract another function from it with a name that is not merely a restatement of its implementation.

One Level of Abstraction per Function:

Mixing levels of abstraction within a function is always confusing. Readers may not
be able to tell whether a particular expression is an essential concept or a detail. Worse, like broken windows, once details are mixed with essential concepts, more and more details tend to accrete within the function.

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.

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."

Switch Statements:

It's hard to make a small `switch` statement. 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:

It better describes what the function does.
Choose 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.
Don't be afraid to spend time choosing a name. try several different names and read the code with each in place.
Be consistent in your names. Use the same phrases, nouns, and verbs in the function
names you choose for your modules.

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.
Output arguments are harder to understand than input arguments.

Common Monadic Forms:

two very common reasons to pass a single argument into a function
asking a question, transforming it into something else and *returning it*.
An *event*. In this form there is an input argument but no output argument.
Try to avoid any monadic functions that don't follow these forms, Using an output argument instead of a
return value for a transformation is confusing.

Flag Arguments:

Flag arguments are ugly. Passing a boolean into a function is a truly terrible practice.

Dyadic Functions:

A function with two arguments is harder to understand than a monadic function.
we should never ignore any part of code. The parts we ignore are where the bugs will hide.

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:

When a function seems to need more than two or three arguments, it is likely that some of
those arguments ought to be wrapped into a class of their own. Consider, for example, the
difference between the two following declarations:

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

Reducing the number of arguments by creating objects out of them may seem like
cheating, but it’s not. When groups of variables are passed together, the way `x` and
`y` are in the example above, they are likely part of a concept that deserves a name of its
own.

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.

Have No Side Effects:

Side effects are lies. Your function promises to do one thing, but it also does other *hidden*
things.

Output Arguments:

Arguments are most naturally interpreted as *inputs* to a function.
Anything that forces you to check the function signature is equivalent to a double-take. It’s
a cognitive break and should be avoided.
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.

The real solution is to separate the
command from the query so that the ambiguity cannot occur.

```
if (attributeExists("username")) {
setAttribute("username", "unclebob");
...
}
```

Prefer Exceptions to Returning Error Codes:

```
if (deletePage(page) == E_OK)
```

This does not suffer from verb/adjective confusion but does lead to deeply nested structures.
When you return an error code, you create the problem that the caller must deal with
the error immediately.

Extract Try/Catch Blocks:

`Try/catch` blocks are ugly in their own right. They confuse the structure of the code and
mix error processing with normal processing. So it is better to extract the bodies of the `try`
and `catch` blocks out into functions of their own.

```
public void delete(Page page) {
try {
deletePageAndAllReferences(page);
}
catch (Exception e) {
logError(e);
}
}
private void deletePageAndAllReferences(Page page)
throws Exception {
deletePage(page);
registry.deleteReference(page.name);
configKeys.deleteKey(page.name.makeKey());
}
private void logError(Exception e) {
logger.log(e.getMessage());
}
```

In the above,
the `delete` function is all about error processing. It is easy to understand and then ignore.
The `deletePageAndAllReferences` function is all about the processes of fully deleting a `page`. Error handling can be ignored.
This provides a nice separation that makes the code easier to understand and modify.

Error Handling Is One Thing:

Functions should do one thing. Error handing is one thing. Thus, a function that handles errors should do nothing else. This implies (as in the example above) that if the keyword `try` exists in a function, it should be the very first word in the function and that there should be nothing after the `catch/finally` blocks.

Don't Repeat Yourself13:

Duplication is a problem;
because it bloats the code and will require four-fold modification should the algorithm ever have to change. It is also a four-fold opportunity for an error of omission.
Duplication may be the root of all evil in software.
Structured programming, Aspect Oriented Programming, Component Oriented Programming, are all, in part, strategies for eliminating duplication. It would appear that since the invention of the subroutine, innovations in software development have been an ongoing attempt to eliminate duplication from our source code.

Structured Programming:

Dijkstra said:
"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.

Those rules serve little benefit when functions are very small. It is only in larger functions that such rules provide significant benefit.

Conclusion:

Every system is built from a domain-specific language designed by the programmers to describe that system. Functions are the verbs of that language, and classes are the nouns. This is not some throwback to the hideous old notion that the nouns and verbs in a requirements document are the first guess of the classes and functions of a system. Rather, this is a much older truth. The art of programming is, and has always been, the art of language design.

This chapter has been about the mechanics of writing functions well. If you follow the rules herein, your functions will be short, well named, and nicely organized. the functions you write need to fit cleanly together into a clear and precise language to help you with that telling.