Functions

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?

1. 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. This is not an assertion that I can justify. I can't provide any references to research that shows that very small functions are better. What I can tell you is that for nearly four decades I have written functions of all different sizes. I've written several nasty 3,000-line abominations. I've written scads of functions in the 100 to 300 line range. And I've written functions that were 20 to 30 lines long. What this experience has taught me, through long trial and error, is that functions should be very small.

How short should a function be? In 1999 I went to visit Kent Beck at his home in Oregon. We sat down and did some programming together. At one point he showed me a cute little Java/Swing program that he called Sparkle. It produced a visual effect on the screen very similar to the magic wand of the fairy godmother in the movie Cinderella. As you moved the mouse, the sparkles would drip from the cursor with a satisfying scintillation, falling to the bottom of the window through a simulated gravitational field. When Kent showed me the code, I was struck by how small all the functions were. I was used to functions in Swing programs that took up miles of vertical space. Every function in this program was just two, or three, or four lines long. Each was transparently obvious. Each told a story. And each led you to the next in a compelling order. That's how short your functions should be!3

- **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.

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.

2. Do One Thing

The following advice has appeared in one form or another for 30 years or more.
So, another 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 imple- mentation [G34].

3. 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. 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. I call this The Step- down Rule.

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 paragraphs is an effective technique for keeping the abstraction level consistent.

4. Switch Statements

It’s hard to make a small switch statement. Even a switch statement with only two cases is larger than I’d like a single block or function to be.

5. Use Descriptive Names

It is hard to overestimate the value of good names. Remember Ward’s principle: “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.

6. 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.

- **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!

There are times, where two arguments are appropriate. For example, Point p = new Point(0,0); is perfectly reasonable.

- **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.

- **Argument Lists**

If the variable arguments are all treated identically, then they are equivalent to a single argument of type List.
So all the same rules apply. Functions that take variable arguments can be monads, dyads, or even triads. But it would be a mistake to give them more arguments than that.

- **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.

7. Have No Side Effects

Side effects are lies. Your function promises to do one thing, but it also does other hidden things. Sometimes it will make unexpected changes to the variables of its own class. Sometimes it will make them to the parameters passed into the function or to system globals. In either case they are devious and damaging mistruths that often result in strange temporal couplings and order dependencies.

- **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.

8. 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.

9. Prefer Exceptions to Returning Error Codes

Returning error codes from command functions is a subtle violation of command query separation. It promotes commands being used as expressions in the predicates of if statements.

```
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.

- **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 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.

10. Structured Programming

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.

How Do You Write Functions Like This?
Writing software is like any other kind of writing. When you write a paper or an article, you get your thoughts down first, then you massage it until it reads well. The first draft might be clumsy and disorganized, so you wordsmith it and restructure it and refine it until it reads the way you want it to read.