Nicole P. Satiembre **March 7, 2024**

Cebu Technological University-Main Campus

Clean Code

Chapter 3

3. Functions

During the early days, systems were composed of routines and subroutines, then Fortran came and it progressed to programs, subprograms, and functions. Today, only the function remains surviving. That is why it is important to know how to write better functions.

Small. An important rule for functions is that they should be small. As a general guideline, lines should not be 150 characters long. Functions should not be 100 lines long. Functions should hardly ever be 20 lines long. The author shares about his friend Kent Beck who made a cute Java/Swing program called *Sparkle*. He was amazed that every function in this program was just two, three, or four lines long. Each was transparently obvious.

Blocks and Indenting. In the case of blocks within if statements, else statements, while statements, etc., they should be one line long, most preferably 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. Lastly, functions should be easy to read and understand.

Do One Thing. Functions are designed to do one thing. They should do it well, they should do it only. 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 implementation.

Sections within Functions. An obvious symptom of a function doing more than one thing is you can divide it into declarations, initializations, and sieve. Functions that do one thing cannot be reasonably divided into sections.

One Level of Abstraction per Function. We need to make sure that the statements within our function are all at the same level of abstraction. This way, we can make sure that our functions are doing “one thing”. 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.* The passage talks about a helpful coding rule called The Stepdown Rule. It suggests arranging code like a story, where each part is followed by the next at a similar level of detail. The aim is to make the code clear by organizing functions like paragraphs, each dealing with a specific level of detail. The text uses examples related to tests to show how following this rule can be challenging for programmers but is crucial for keeping functions short and focused on doing one thing.

Switch Statements. Making switch statements small is another challenge. Switch statements are designed to do multiple things, making them hard to manage. While it's difficult to completely avoid them, a better approach is to confine switch statements to low-level classes, ensuring they are not repeated. This is achieved through polymorphism, where the switch statement is hidden behind an abstract factory. The general rule is to tolerate switch statements if they are used once, create polymorphic objects, and are concealed behind an inheritance relationship.

Use Descriptive Names. 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. We shouldn’t be afraid to spend time choosing a name, this way we can help clarify the design of the module. Lastly, we must be consistent in our naming conventions.

Function Arguments. The best case scenario for function arguments is to have none, zero (niladic). Next comes one (monadic), followed closely by two (dyadic). Three arguments (triadic) should be avoided as much as possible. More than three (polyadic) requires very special justification – and then shouldn’t be used anyway. Arguments are hard and they are at a different level of abstraction than the function name and it forces us to know a detail that isn’t particularly important at that point.

Common Monadic Forms. The two very common reasons to pass a single argument into a function are when you are asking a question about that argument or you may be operating on that argument, transforming it into something else and returning it. A somewhat less common, but still very useful form for a single argument function is an event. We must use this form with care and choose names and contexts carefully.

Flag Arguments. Flag arguments are ugly because booleans complicate the signature of the method signaling 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. Dyadic functions are harder to understand than monadic functions but there will always be a time when we will need to write them. However, we should be aware that they come at a cost and should take advantage of what mechanisms may be available to us to convert them into monads.

Triads. It is significantly harder to understand triads compared to dyadic functions. It doubles the issues of ordering, pausing, and ignoring. That is why we must be very careful and think well before considering using triads.

Argument Objects. We can consider creating objects out of arguments so we can reduce them. When groups of variables are passed together, they are likely part of a concept that deserves a name of its own.

Argument Lists. When all variable arguments are treated uniformly, they can be viewed as equivalent to a single argument of type List. The passage emphasizes that functions with variable arguments can be monads, dyads, or triads but should not have more arguments than that, illustrated with examples like void monad(Integer... args), void dyad(String name, Integer... args), and void triad(String name, int count, Integer... args).

Verbs and Keywords. Choosing meaningful names for functions can significantly enhance clarity by conveying the function's intent and the purpose of its arguments. In the case of a monad, it's beneficial to create a verb/noun pair, such as the illustrative example of write(name), where something referred to as "name" is being "written." An even clearer name, like writeField(name), specifies that the "name" is a "field." Additionally, using the keyword form of a function name, like assertExpectedEqualsActual(expected, actual) instead of assertEquals, helps alleviate the challenge of remembering the argument order by encoding them into the function name.

Have No Side Effects. It is risky to have side effects functions as it sometimes makes unexpected changes to the variables of its class. The function promises to do one thing, but it also does other hidden things. Sometimes it will make them to the parameters passed into the function or to system globals. Temporal couplings can also confuse, especially when hidden as a side effect. If it is needed, we should make it clear in the name of the function.

Output Arguments. Though when we think of arguments, what usually comes into our mind is “input” we can also have it as an output rather than an input. However, this might not be obvious on the first look. For example, appendFooter(s). It doesn’t take long to look at the function signature and see: public void appendFooter (StringBuffer report). This way, the issue is clarified but it costs us additional checking of the declaration of the function. We encounter double-take when we are forced to check the function signature. It’s a cognitive break and should be avoided. As a general rule, output arguments should be avoided.

Command Query Separation. Our functions should only do something like changing an object’s state or answering something like returning some information about that object, but it cannot do both. Doing both only leads to confusion.

Prefer Exceptions to Returning Error Codes. It is advisable to use exceptions instead of returning error codes. Returning error codes from command functions is a subtle violation of command query separation. When you return an error code, the caller needs to deal with the error immediately. Whereas when using exceptions, the processing code can be separated from the happy path code and can be simplified.

Extract Try/Catch Blocks. Try/Catch blocks can confuse the structure of the code and mix error processing with normal processing that is why it is best to extract the bodies of the try and catch blocks into functions of their own.

Error Handling is One Thing. Functions should only do one thing, and error handling is a separate task, so if a function handles an error, it shouldn’t do anything else.

The Error.java Dependency Magnet. Returning error codes would mean that there is a class or enums in which all the codes are defined. This leads to a dependency magnet. Using exceptions in programming allows for more flexible error handling compared to error codes. With exceptions, new error scenarios can be added without requiring recompilation or redeployment of existing code, making the system more maintainable and adaptable.

Don’t Repeat Yourself. Sometimes, it may not be easy to spot duplications because instances intermixed with other codes aren’t uniformly duplicated. This duplication bloats the code and requires multiple modifications should the algorithm change, it’s also a multiple opportunity for errors. Because of this, there have been many technological attempts to eliminate duplications such as the invention of subroutines.

Structured Programming. The passage discusses Edsger Dijkstra's rules of structured programming, emphasizing the idea that every function and block within a function should have one entry and one exit. While these rules align with the Open Closed Principle, the text suggests that in smaller functions, the benefits are limited, and occasional multiple return, break, or continue statements can be more expressive. However, the use of goto is discouraged even in larger functions.

How do you write Functions Like This? Just like writing an essay, writing functions follow a similar pattern. First, you have a thought in your head, and then you write a draft. After that, you revamp the draft, restructuring it until we arrive at how we wanted our writings would look. Writing code is the same, we refine that code, split out functions, change names, and eliminate duplication. We can also shrink the methods and reorder them. Or we can even break out whole classes, all the while keeping the tests passing. This way, we can write better functions.