Small!
The passage stresses the importance of making functions in software development brief and clear. It talks about how the author's own experiences show that shorter functions make code easier to understand and work with, even though there isn't specific research mentioned to back this up. It gives an example of how a complex task can be broken down into simple functions, using a story involving Kent Beck. The text suggests that functions should be shorter than a certain length given, and it's best if the blocks inside functions are kept to just one line, preferably a clear function call. This helps not only in managing the size of functions but also in making the code more understandable with descriptive function names. By keeping functions small and avoiding deep nesting, code becomes simpler to grasp, update, and expand.

Do One Thing
The passage emphasizes the idea that functions should ideally do one specific job well, but it also recognizes the difficulty in defining what exactly counts as "one thing." Even though the advice to keep functions focused has been around for a long time, deciding how detailed tasks should be can be a matter of opinion. For instance, in the provided code snippet, there's a debate over whether a function is doing just one job or multiple jobs based on how abstract it is. The text suggests that if the steps inside a function closely match its name or purpose, it's probably only doing one thing. But if there are different levels of abstractness or if making another function out of it just repeats the code without changing its level of abstraction, then the function might be doing more than one thing. This example shows how important it is to make functions easy to understand and how tricky it can be to focus on just one task in real situations. It also points out that splitting a function into parts like declarations and initializations might go against the idea of doing just one thing, as functions meant to do one task shouldn't be split up like that.

One Level of Abstraction per Function
This passage talks about how important it is to keep code clear and easy to understand. It says that when writing functions, it's best to stick to one level of detail at a time. Mixing different levels of detail can confuse people reading the code and make it harder to understand.

The Stepdown Rule suggests that each function should lead to the next with less detail, like a story where each paragraph builds on the last. This makes the code easier to follow and helps each function focus on doing just one thing well.

Following this rule might be hard for programmers at first, but it's really important for making sure functions are short and the overall code is well-organized. Examples in the text show how following this rule makes code clearer and easier to work with.

Switch Statements

The passage talks about the problems and solutions related to using switch statements in writing computer programs. It points out that switch statements can become complicated over time and may not follow certain coding principles. To tackle this, the passage suggests hiding switch statements in specific parts of the code and using a technique called polymorphism. This helps make the code more flexible and easier to manage. The author also recommends using a design pattern called abstract factory to organize switch statements better. However, the author admits that there might be situations where breaking these rules is necessary. Overall, the passage stresses the importance of handling switch statements carefully to keep the code organized and scalable in software projects.

Use Descriptive Names

This paragraph talks about how important it is to give clear names to things in computer code. When developers name functions and methods in a way that accurately describes what they do, it makes the code easier to read and understand. Following Ward's principle means organizing code in a predictable way, which is done by giving functions names that show exactly what they do. It suggests using longer, descriptive names instead of short, confusing ones, saying that it's better to have clear names than to have to write long explanations. It tells developers to take time to choose good names and to use modern software tools to help them. It also says it's important to use the same naming style throughout the code to keep it organized and easy to follow. In short, it stresses that choosing good names not only makes the code easier to understand but also makes it better overall and easier to maintain.

Function Arguments

The passage gives helpful advice about how many arguments to use in designing functions. It suggests that it's best to have either no arguments or just one. Having two arguments is okay, but having more than that, like three or more, is not recommended. It explains that dealing with arguments can be mentally challenging for both programmers writing the code and people reading it, so it's important to think carefully about how many to use. The passage also talks about naming functions clearly and consistently. It says it's important to make it clear whether a function is for asking questions, changing things, or for special events. It criticizes using "flag" arguments because they can be confusing, and it recommends changing functions with two arguments into ones with just one if possible, to make things easier to understand and maintain. It also suggests grouping multiple arguments together when it makes sense and advises against using lists of variable arguments too often. Overall, it emphasizes the importance of planning carefully when designing functions and choosing good names to make software easier to understand and work with.

Have No Side Effects

The paragraph talks about something called "side effects" in programming functions. Side effects are extra things a function does besides its main job, and they can be surprising. It gives an example of a function called `checkPassword`, which not only checks passwords but also starts a session, which is not obvious from the name. This extra action can cause problems because the function might only work correctly in certain situations. The text suggests that if a function has to do something extra like this, it should be clear in its name.

It also talks about another problem with using output arguments in functions. This means a function not only gives a result but also changes something outside itself. While this was common in older programming styles, it's not recommended in modern languages because it can be confusing. Instead, functions should mainly change things within themselves rather than outside.

Command Query Separation

The excerpt emphasizes the importance of making functions clear and focused. It suggests that functions should either change something or provide information, but not both at the same time. It criticizes the confusion caused by functions that try to do both, giving an example of a function that both sets a value and tells if it was successful. This mixing up of purposes makes it hard for people reading the code to understand what the function is meant to do. The author suggests solving this problem by separating actions from inquiries, so functions have one clear purpose. This makes the code easier to read and maintain, improving the quality of the software design.

Prefer Exceptions to Returning Error Codes

The passage talks about how returning error codes from command functions can make code messy and hard to manage. It explains that using error codes often leads to complicated blocks of code with lots of if statements. Instead, it suggests using exceptions for handling errors, which makes code simpler and easier to read. The passage also recommends organizing error handling code into separate functions to keep things clear and easy to change. It highlights the importance of keeping functions focused on one task, suggesting that error-handling functions should only deal with errors. Additionally, it mentions that using exceptions is better than error codes because it's more flexible and doesn't require making big changes to the code when adding new error types. Overall, it stresses the advantages of using exceptions for cleaner, easier-to-maintain code.

Don’t Repeat Yourself

The passage talks about how copying code in software development can cause problems. It mentions a situation where a specific piece of code is repeated four times in a test suite. This repetition makes the code longer and harder to manage, increasing the chance of mistakes. To solve this, the passage suggests using a method called 'include' to remove the duplicates, which makes the code easier to understand and maintain.

It also suggests that copying code is a big issue in software, leading to the creation of rules and methods to deal with it. For example, in databases, there are rules called Codd's normalization forms, and in programming, there are different ways of organizing code, like object-oriented or structured programming, which all try to deal with copying code in different ways. Overall, the passage suggests that throughout the history of software development, people have been trying to find ways to avoid copying code, starting from when subroutines were first used.

Structure Programming

The passage talks about Edsger Dijkstra's ideas on structured programming, which say that functions and blocks within them should have one way in and one way out. While these guidelines are good for big functions, the text says that for really small functions, sticking to these rules might not be very helpful. It suggests that in these cases, allowing a few different ways to exit or break out of the function can actually make the code clearer and easier to understand. However, it strongly warns against using the "goto" statement, saying it's only useful in big functions and should be avoided entirely. This viewpoint respects the principles of structured programming but also points out that sometimes, flexibility is important too. It suggests that following strict rules should be balanced with making sure the code is easy to read and runs efficiently, especially in smaller functions.

How Do You Write Function Like This?

The passage compares writing software to other kinds of writing, saying they're similar because both processes involve making lots of changes to improve. Just like you might rewrite a story many times to make it clearer and better organized, writing software also requires lots of changes to make it work well. At first, the code might be messy and hard to understand, like a rough draft of a story. But by going through many rounds of testing and making changes, developers can make the code better. They might rename things, reorganize parts, and get rid of unnecessary stuff to make it easier to read and maintain. The main idea is to keep making small improvements until the software meets the standards we've set. The passage also points out that it's not realistic to write perfect code right away, but by continuously making it better, we can eventually create high-quality software.