Nicole P. Satiembre **March 5, 2024**

Cebu Technological University-Main Campus

Clean Code

Chapter 2

1. Meaningful Names

Use Intention-Revealing Names. Choosing good names takes time but saves more than it takes. The name of a variable, function, or class, should answer all the big questions. It should tell you why it exists, what it does, and how it is used. If a name requires a comment, then the name does not reveal its intent. We should choose a name that specifies what is being measured and the unit of that measurement, for example, int elapsedTimeInDays;

Avoid Disinformation. Programmers should steer clear of introducing misleading signals that cloud the understanding of code. It is advisable to refrain from using words with established meanings that differ from our intended significance. It is also not a good practice to have variable names that look almost identical. It is very hard to spot the difference between *XYZControllerForEfficientHandlingOfStrings* and *XYZControllerForEfficientStorageOfStrings.* Another example would be using lowercase "l" or uppercase "O" as variable names, especially together because they closely resemble the constants "1" and "0," respectively.

Make Meaningful Distinctions. We create problems when we write code solely for the compiler. For example, because you can’t use the same name to refer to two different things in the same scope, you might be tempted to change one name arbitrarily. Sometimes this is done by misspelling one, leading to a surprising situation where correcting spelling errors leads to an inability to compile. Additionally, the text criticizes noise words, pointing out that distinctions like "Info" or "Data" without meaningful differences are counterproductive. The redundancy of noise words is stressed, discouraging the inclusion of words like "variable" in variable names or "table" in table names. Lastly, the text highlights a real-world example where ambiguous naming (e.g., getActiveAccount, getActiveAccounts, getActiveAccountInfo) creates confusion for programmers in determining which function to call.

Use Pronounceable Names. As humans, we are innately good with words, especially as a form of communication. We must then take good advantage of this by making our variable names pronounceable and not hard to pronounce. A bad example set in this chapter is *genymdhms* for generation date, year, month, day, hour, minute, and second. The new programmers have hard time understanding this. This is just one example why it is important touse pronounceable names when naming a variable.

Use Searchable Names. The text highlights a specific issue with single-letter names and numeric constants in programming. It points out that locating them across a body of code can be challenging compared to more descriptive names. While searching for a well-named constant like MAX_CLASSES_PER_STUDENT is straightforward, finding occurrences of the number 7 might be problematic due to its prevalence in various contexts. The text notes that using a single letter like 'e' as a variable name is not ideal for searchability. The popularity of 'e' in the English language makes it likely to appear frequently in any code, complicating the process of locating specific instances. Longer, more descriptive names are advocated, especially for variables or constants with broader usage in the code.

Avoid Encodings. Additional encodings to names in programming is not a good idea, according to the text. It makes things harder for developers, especially new ones, and encoded names are difficult to say and easy to type incorrectly.

Hungarian Notation. In the past, programmers used type encoding, like Hungarian Notation, because older languages had limitations in name length. However, with modern languages that have better type systems and compilers, this practice is seen as a hindrance. It makes code harder to read, change, and can confuse the reader, as shown in examples like `PhoneNumber phoneString;`.

Member Prefixes. You also don’t need to prefix member variables with m_ anymore. Your classes and functions should be small enough that you don’t need them. You should be using an editing environment that highlights or colorizes members to make them distinct.

Interfaces and Implementation. When creating something like a shape factory, the text suggests deciding on names for the interface and concrete class. The author prefers leaving interfaces simple, like ShapeFactory, instead of adding 'I' as in IShapeFactory, finding it distracting. If encoding is necessary, the author suggests encoding the implementation, like ShapeFactoryImp, rather than the interface.

Avoid Mental Mapping. Readers shouldn't have to mentally translate your names into other names they already know. Programmers are generally known to be smart and one way to show off their smarts is by demonstrating their mental juggling abilities. After all, if you can reliably remember that r is the lower-case version of the URL with the host and scheme removed, then you must be brilliant. However, the significant difference that makes a programmer Professional is knowing that clarity is king. Professionals use their powers for good and write code that others can understand.

Class Names. Classes and objects should have noun or noun phrase names like Customer, WikiPage, Account, and AddressParser. Avoid words like Manager, Processor, Data, or Info in the name of a class. A class name should not be a verb.

Method Names. Methods should have verb or verb phrase names like postPayment, deletePage, or save. Accessors, mutators, and predicates should be named for their value and prefixed with get, set, and according to the JavaBean standard.

Don't be Cute. If names are too clever, they will be memorable only to people who share the author's sense of humor, and only as long as these people remember the joke. Cuteness in code often appears in the form of colloquialisms or slang. For example, don't use the name whack() to mean kill(). Don't tell little culture-dependent jokes like eatMyShorts() to mean abort(). Say what you mean. Mean what you say.

Pick One Word Per Concept. We must be using a single word for one abstract concept to avoid confusion. Having similar methods like fetch, retrieve, and get in different classes can be perplexing, requiring users to remember the library or class origin to distinguish them. The importance of a consistent lexicon in a codebase is emphasized, as it helps programmers navigate and understand the functionality more easily, even in modern editing environments.

Don't Pun. It is not advisable to use the same word for two different purposes in programming, comparing it to a pun. Following the "one word per concept" rule can lead to multiple classes having methods with the same name, like "add," as long as their meanings align. However, the text cautions against using a term like "add" for consistency when the semantics differ, suggesting alternative names like "insert" or "append" to maintain clarity and avoid puns. The goal is to make code easily understandable, favoring a quick skim approach over an intense study, akin to the popular paperback model.

Use Solution Domain Names. It is known that programmers will read our code and these are technically inclined people. Thus, it is much more appropriate to use Computer Science terms such as algorithm names, pattern names, math names, and so on. Choosing technical names is much preferred so that our communication with our co-programmers will run smoothly.

Use Problem Domain Names. When there is no appropriate technical term to use, we can opt for problem domain names. Separating solution and problem domain concepts is part of the job of a good programmer and designer. The code that has more to do with problem domain concepts should have names drawn from the problem domain.

Add Meaningful Context. It is important to provide context to names in programming by encapsulating them within well-named classes, functions, or namespaces. Adding prefixes when necessary, such as using "addr" for address-related variables. However, it is highly preferred to have an even clearer approach by creating meaningful classes like "Address." The example of a method with variables lacking immediate context underscores the significance of transparent variable meanings for code comprehension, emphasizing the goal of making code easily understandable for readers.

Don’t Add Gratuitous Context. Shorter names are generally better than longer ones, so long as they are clear. Add no more context to a name than is necessary. The resulting names are more precise, which is the point of all naming.