Nicole P. Satiembre **March 13-20, 2024**

Cebu Technological University-Main Campus

Clean Code

Chapter 5

5. Formatting

As professionals in our industry, we must ensure that our code is nicely formatted. We should choose a set of simple rules that govern the format of our code, and then we should consistently apply those rules. When working with a team, the team should agree to a single set of formatting rules and all members should comply.

The Purpose of Formatting. Communication is the professional developer's first order of business, which is why code formatting is significant. The functionality of our code continuously changes over time but the readability of our code has a profound effect on all the changes that will ever be made. This affects the maintainability of our code.

Vertical Formatting. Small files are usually easier to understand than large file sizes. The largest file in FitNesse is about 400 lines and the smallest is 6 lines. The smallest is 6 lines. The small difference in vertical position implies a very large difference in absolute size. Junit, FitNesse, and Time and Money are composed of relatively small files. None are over 500 lines and most of those files are less than 200 lines. This means that it is possible to build significant systems that are 200 lines long with an upper limit of 500, although this should not be considered an ultimate rule it is considered a desirable guideline.

The Newspaper Metaphor. This metaphor compares a well-written newspaper to our code. When you read a newspaper vertically, at the top, we expect a headline that will tell us what the story is about and allow us to decide whether it is something you want to read. We can compare our source file to be like a newspaper article. The name should be simple but explanatory. Detail should increase as we move downward, until at the end we find the lowest level functions and details in the source file. What makes a newspaper usable is that it is composed of a collection of articles, most of which are very small. This way, it is simple to read and the data are well organized.

Vertical Openness Between Concepts. Codes are typically read left to right and top to bottom. Each line represents an expression or a clause, and each group of lines represents a complete thought. Those thoughts should be separated from each other with blank lines. This gives a more pronounced effect when we unfocus our eyes, which makes each group of lines pop out.

Vertical Density. Vertical density signals association. Lines of code that are closely related should be shaped vertically dense because openness separates concepts.

Vertical Distance. To avoid getting lost in the long chunks of code, we must organize our code emphasizing vertical distance. Concepts that are closely related should be kept vertically close to each other. We want to avoid forcing our readers to hop around through our source files and classes. For those concepts that are so closely related that they belong in the same source file, their vertical separation should be a measure of how important each is to the understandability of the other.

Variable Declarations. Variables should be declared as close to their usage as possible. Local variables should appear at the top of each function because functions are mostly short. In rare cases, a variable might be declared at the top of a block or just before a loop in a long-ish function.

Instance Variables. It should be declared at the top of the class. Although there are other conventions like the scissors rule where you put all the instance variables at the bottom, it is still better to put them at the top. The important thing is for the instance variables to be declared in one well-known place.

Dependent Functions. If one function calls another, they should be vertically close, and the caller should be above the callee, if at all possible. Following this convention will make it easier for readers to trust that function definitions will follow shortly after their use.

Conceptual Affinity. The stronger the conceptual affinity between codes, the less vertical distance there should be between them.

Vertical Ordering. Function call dependencies should point in the downward direction where a function that is called should be below a function that does the calling. Just like in news articles, we want to read the most important concepts first and the low-level details to come last.

Horizontal Formatting. When considering how wide our codes must be, we should strive to keep our lines short. The old guideline ideals 80 but 100 or even 120 will do, beyond that is probably just careless. As a general rule, we should never have to scroll all the way to the right.

Horizontal Openness and Density. We use horizontal space to associate things that are strongly related and diassociate things that are more weakly related.

Horizontal Alignment. Horizontal alignment helps accentuate certain structures. It helps the code readability if you try to line up all the variable names in a set of declarations. However, the author discovers that this approach is not very useful as it leads your eyes away from the true intent. He now prefers unaligned declarations and assignments, because they point out an important deficiency. If there is a long list that needs to be aligned, the problem centers at the length of the list and not the lack of alignment.

Indentation. Our eyes can rapidly discern the structure of the indented file that is why it is important to use proper indentation in our codes. With proper indentation, we can instantly spot variables, constructors, accessors, and methods.

Breaking Indentation. It may sometimes be tempting to break the indentation rule for short if statements, short while loops, or short functions, however, as we move along, we find ourselves going back to put the indentation back in.

Dummy Scopes. As much as possible, we would want to avoid putting dummy codes at the body of a while or for statements or any other blocks of code. The semicolons can fool us while it's silently sitting at the end of a while loop on the same line. It’s very hard to detect.

Team Rules. Each one of us has a favorite programming rule but when working in a team, we should abide by the team’s rules. We want our software to have a consistent style. A good software system is composed of a set of documents that read nicely.