# **Chapter 5: Formatting**

* You should take care that your code is nicely formatted.
* You should choose a set of simple rules that govern the format of your code, and then you should consistently apply those rules.

**The Purpose of Formatting**

* Code formatting is ***important.***
* It is about communication, and communication is the professional developer’s first order of business.

**Vertical Formatting**

* Small files are usually easier to understand than large files are.
* **The Newspaper Metaphor**
  + We would like a source file to be like a newspaper article. The name should be simple but explanatory.
    - The name, by itself, should be sufficient to tell us whether we are in the right module or not.
* **Vertical Openness Between Concepts**
  + Nearly all code is 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.
  + It separates concepts.
* **Vertical Density**
  + It implies close association.
  + Lines of code that are tightly related should appear vertically dense.
* **Vertical Dense**
  + Concepts that are closely related should be kept vertically close to each other.
  + Closely related concepts should not be separated into different files unless you have a very good reason.
  + Those belong in the source file, their vertical separation should be a measure of how important each is to the understanding of the other.
  + **Variable Declarations** – local variables should appear at the top of each function. They should be declared as close to their usage as possible.
  + **Instance Declarations** – should be declared at the top of the class.
  + **Dependent Functions** – If one function calls another, they should be vertically close, and the caller should be above the callee.
  + **Conceptual Affinity** - Certain bits of code want to be near other bits. They have a certain conceptual affinity. The stronger that affinity, the less vertical distance there should be between them.
* **Vertical Ordering**
  + We want function call dependencies to point in the downward direction. This creates a nice flow down the source code module from high level to low level.

**Horizontal Formatting**

* You should never have to scroll to the right.
* **Horizontal Openness and Density**
  + We use horizontal white space to associate things that are strongly related and disassociate things that are more weakly related.
  + Another use for white space is to accentuate the precedence of operators.
* **Horizontal Alignment**
  + Unaligned declarations point out an important deficiency.
* **Indentation**
  + To make a hierarchy of scopes visible, we indent the lines of source code in proportion to their position in the hierarchy.
  + It visually lines up on the left to see what scope they appear in.
  + This allows programmers to quickly hop over scopes.
  + **Breaking Indentation** - Avoid collapsing scopes down to one line. Better expand and indent the scopes
  + **Dummy Scopes** - Dummy body must be properly indented and surrounded by braces.

**Team Rules**

* Every programmer has his ***own favorite formatting rules***, but if he works in a team, then the ***team rules.***
* Every member of the team should use agree what formatting style should use.
* Remember, a good software system is composed of a set of documents that read nicely. They need to have a consistent and smooth style.

**Uncle Bob’s Formatting Rules**

* You can use Uncle Bob’s Formatting Rules
* See the example on Page 91 – 92 in the book entitled “Clean Code” by Robert C. Martin

# **Chapter 6: Objects and Data Structures**

* There is a reason that we keep our variables private.
* Objects expose behavior and hide data.
* Data structures expose data and have no significant behavior.

**Data Abstraction**

* Hiding implementation is about abstractions.
* A class does not simply push its variables out through getters and setters. Rather it exposes abstract interfaces that allow its user to manipulate the essence of the data, without having to know its implementation.

**Data/Object Anti-Symmetry**

* Objects hide their data behind abstractions and expose functions that operate on that data.
* Data structure expose their data and have no meaningful functions.
* Objects hide their data behind abstractions and expose functions that operate on that data.
* Data structure expose their data and have no meaningful functions.
* The things that are hard for OO are easy for procedures, and the things that are hard for procedures are easy for OO.
* ***Myth***: Everything is an object.

**The Law of Demeter**

* A module should not know about the innards of the objects it manipulates.
* Object should not expose its internal structure through accessors because to do so is to expose, rather than to hide, its internal structure.
* A method *f* of a class *C* should only call the methods of these:
  + *C*
  + An object created by *f*
  + An object passed as an argument to *f*
  + An object held in an instance variable of *C*
* Talk to friends, not to strangers.
  + The method should not invoke methods on objects that are returned by any of the allowed functions.
* **Train Wrecks**
  + It looks like a bunch of coupled train cars.
  + Chains of calls are generally considered to be sloppy style and should be avoided.
* **Hybrids**
  + Hybrids make it hard to add new functions but also make it hard to add new data structure.
  + Avoid creating hybrids.
* **Hiding Structures**
  + Dots, slashes, file extensions, and objects should not be so carelessly mixed together, and mixed with the enclosing code. We can see that the intent of getting the absolute path of the scratch directory was to create a scratch file of a given name.
  + 
  + This allows ctx to hide its internals and prevents the current function from having to violate the Law of Demeter by navigating through objects it shouldn't know about.

**Data Transfer Objects**

* Data Transfer Objects (DTOs) - the quintessential form of a data structure is a class with public variables and no functions.
* It is useful when communicating with databases or parsing messages from sockets, and so on.
* **Active Records**
  + Special forms of Data Transfer Objects (DTOs)
  + Data structures with public variables; but they have typically navigational methods.
  + Direct translations from database tables or other data sources.
  + Treat the Active Record as a data structure and to create separate objects that contain the business rules and that hide their internal data (which are probably just instances of the Active Record).