CLEAN CODE

What Is Clean Code? There are probably as many definitions as there are programmers. So I asked some very well-known and deeply experienced programmers what they thought. Bjarne Stroustrup, inventor of C++ and author of The C++ Programming Language I like my code to be elegant and efficient.

The logic should be straightforward to make it hard for bugs to hide, the dependencies minimal to ease maintenance, error handling complete according to an articulated strategy, and performance close to optimal so as not to tempt people to make the code messy with unprincipled optimizations. Clean code does one thing well. Bjarne uses the word “elegant.” That’s quite a word! The dictionary in my MacBook® provides the following definitions: pleasingly graceful and stylish in appearance or manner; pleasingly ingenious and simple. Notice the emphasis on the word “pleasing

Clean Code as per Grady Booch

Grady Booch, author of Object Oriented Analysis and Design with Applications Clean code is simple and direct. Clean code reads like well-written prose. Clean code never obscures the designer’s intent but rather is full of crisp abstractions and straightforward lines of control

“Big” Dave Thomas, founder of OTI, godfather of the Eclipse strategy

Clean code can be read, and enhanced by a developer other than its original author. It has unit and acceptance tests. It has meaningful names. It provides one way rather than many ways for doing one thing. It has minimal dependencies, which are explicitly defined, and provides a clear and minimal API. Code should be literate since depending on the language, not all necessary information can be expressed clearly in code alone

Big Dave shares Grady’s desire for readability, but with an important twist. Dave asserts that clean code makes it easy for other people to enhance it. This may seem obvious, but it cannot be overemphasized. There is, after all, a difference between code that is easy to read and code that is easy to change.

Michael Feathers, author of Working Effectively with Legacy Code

I could list all of the qualities that I notice in clean code, but there is one overarching quality that leads to all of them. Clean code always looks like it was written by someone who cares. There is nothing obvious that you can do to make it better. All of those things were thought about by the code's author, and if you try to imagine improvements, you're led back to where you are, sitting in appreciation of the code someone left for you—code left by someone who cares deeply about the craft.

Ward Cunningham, inventor of Wiki, inventor of Fit, coinventor of eXtreme Programming. Motive force behind Design Patterns. Smalltalk and OO thought leader. The godfather of all those who care about code

You know you are working on clean code when each routine you read turns out to be pretty much what you expected. You can call it beautiful code when the code also makes it look like the language was made for the problem

CLEAN CODE REQUIREMENTS

Meaningful Names
Use Intention Revealing Names
Avoid Disinformation
Make Meaningful Distinction
Use Pronounceable Names

Make meaning full names

Programmers create problems for themselves when they write code solely to satisfy a compiler or interpreter. 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 in an arbitrary way. Sometimes this is done by misspelling one, leading to the surprising situation where correcting spelling errors leads to an inability to compile.2 It is not sufficient to add number series or noise words, even though the compiler is satisfied. If names must be different, then they should also mean something different

Number-series naming (a1, a2, .. aN) is the opposite of intentional naming. Such names are not disinformative—they are noninformative; they provide no clue to the author’s intention.

Consider: public static void copyChars(char a1[], char a2[]) { for (int i = 0; i < a1.length; i++) { a2[i] = a1[i]; } }

This function reads much better when source and destination are used for the argument names.

Use Pronounceable words

Humans are good at words. A significant part of our brains is dedicated to the concept of words. And words are, by definition, pronounceable

If you can’t pronounce it, you can’t discuss it without sounding like an idiot

Compare

```
class DtaRcrd102 {
private Date genymdhms;
private Date modymdhms;
private final String pszqint = "102";
/* ... */
};
```

to

```
class Customer {
private Date generationTimestamp;
private Date modificationTimestamp;;
private final String recordId = "102";
/* ... */
};
```

Use Searchable Names

Single-letter names and numeric constants have a particular problem in that they are not easy to locate across a body of text. One might easily grep for MAX_CLASSES_PER_STUDENT, but the number 7 could be more troublesome. Searches may turn up the digit as part of file names, other constant definitions, and in various expressions where the value is used with different intent

. It is even worse when a constant is a long number and someone might have transposed digits, thereby creating a bug while simultaneously evading the programmer’s search. Likewise, the name e is a poor choice for any variable for which a programmer might need to search. It is the most common letter in the English language and likely to show up in every passage of text in every program. In this regard, longer names trump shorter names, and any searchable name trumps a constant in code. My personal preference is that single-letter names can ONLY be used as local variables inside short methods. The length of a name should correspond to the size of it’s scope

If a variable or constant might be seen or used in multiple places in a body of code,

it is imperative to give it a search-friendly name. Once again compare

```
for (int j=0; j<34; j++) {
s += (t[j]*4)/5;
}
```

to

```
int realDaysPerIdealDay = 4;
const int WORK_DAYS_PER_WEEK = 5;
int sum = 0;
for (int j=0; j < NUMBER_OF_TASKS; j++) {
int realTaskDays = taskEstimate[j] * realDaysPerIdealDay;
int realTaskWeeks = (realdays / WORK_DAYS_PER_WEEK);
sum += realTaskWeeks;
}
```

Note that sum, above, is not a particularly useful name but at least is searchable. The

intentionally named code makes for a longer function, but consider how much easier it

will be to find WORK_DAYS_PER_WEEK than to find all the places where 5 was used and filter

the list down to just the instances with the intended meaning