How to write clean code

What Is Clean Code

Reading code should make you smile the way a well-crafted music box or well-designed car would. No matter how elegant it is, no matter how readable and accessible, without tests, it be unclean. Clean code is code that has been taken care of. Someone has taken the time to keep it simple and orderly. They have paid appropriate attention to details. They have cared.

one thing about authors is that they have readers. Indeed, authors are responsible for communicating well
with their readers. The next time you write a line of code, remember you are an author, writing for readers who will judge your effort.

> if you want your code to be easy to write, make it easy to read.

Meaningful Names

It is easy to say that names should reveal intent. What we want to impress upon you is that we are serious about this. Choosing good names takes time but saves more than it takes. So take care with your names and change them when you find better ones. Everyone who reads your code (including you) will be happier if you do.

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.

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.

> Beware of using names which vary in small ways. How long does it take to spot the subtle difference between a `XYZControllerForEfficientHandlingOfStrings` in one module and, somewhere a little more distant, `XYZControllerForEfficientStorageOfStrings` ? The words have frightfully similar shapes.

Variables Names

Variables should be declared as close to their usage as possible. 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.

```
int d;  // elapsed time in days
```

The name d reveals nothing. It does not evoke a sense of elapsed time, nor of days. We should choose a name that specifies what is being measured and the unit of that measurement:

```
int elapsedTimeInDays;
int daysSinceCreation;
int daysSinceModification;
int fileAgeInDays;
```

Choosing names that reveal intent can make it much easier to understand and change code. What is the purpose of this code?

```
public List<int[]> getThem() {
List<int[]> list1 = new ArrayList<int[]>();
for (int[] x : theList)
if (x[0] == 4)
list1.add(x);
return list1;
}
```

Why is it hard to tell what this code is doing? There are no complex expressions. Spacing and indentation are reasonable. There are only three variables and two constants mentioned. There aren’t even any fancy classes or polymorphic methods, just a list of arrays (or so it seems).

The problem isn’t the simplicity of the code but the implicity of the code (to coin a phrase): the degree to which the context is not explicit in the code itself. The code implicitly requires that we know the answers to questions such as:

1. What kinds of things are in theList ?
2. What is the significance of the zeroth subscript of an item in theList ?
3. What is the significance of the value 4 ?
4. How would I use the list being returned?

The answers to these questions are not present in the code sample, but they could have been. Say that we’re working in a mine sweeper game. We find that the board is a list of cells called theList . Let’s rename that to gameBoard . Each cell on the board is represented by a simple array. We further find that the zeroth subscript is the location of a status value and that a status value of 4 means “flagged.” Just by giving these concepts names we can improve the code considerably:

```
public List<int[]> getFlaggedCells() {
List<int[]> flaggedCells = new ArrayList<int[]>();
for (int[] cell : gameBoard)
if (cell[STATUS_VALUE] == FLAGGED)
flaggedCells.add(cell);
return flaggedCells;
}
```

Notice that the simplicity of the code has not changed. It still has exactly the same number of operators and constants, with exactly the same number of nesting levels. But the code has become much more explicit. We can go further and write a simple class for cells instead of using an array of int s. It can include an intention-revealing function (call it isFlagged) to hide the magic numbers. It results in a new version of the function:

```
public List<Cell> getFlaggedCells() {
List<Cell> flaggedCells = new ArrayList<Cell>();
for (Cell cell : gameBoard)
if (cell.isFlagged())
```

```
flaggedCells.add(cell);
return flaggedCells;
}
```

With these simple name changes, it’s not difficult to understand what’s going on. This is the power of choosing good names.

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. Will they know what the function named `HolyHandGrenade` is supposed to do? Sure, it’s cute, but maybe in this case `DeleteItems` might be a better name. Choose clarity over entertainment value. 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.

Universal principles to follow

- **DRY (**Don’t Repeat Yourself)
- **SOLID** (Follow SOLID principles to write clean classes and well organised APIs)
  - **S :-** Single Responsibility Principle (The single responsibility principle (SRP) asserts that a class or module should do one thing only)
  - **O** :- Open/Closed Principle (The Open/Closed Principle states that code entities should be open for extension, but closed for modification)
  - **L** :- Liskov Substitution Principle (The LSP says, basically, that any child type of a parent type should be able to stand in for that parent without things blowing up)
  - **I** :- Interface Segregation (The Interface Segregation Principle (ISP) says that you should favor many, smaller, client-specific interfaces over one larger, more monolithic interface)
  - **D** :- Dependency Inversion (The Dependency Inversion Principle (DIP) encourages you to write code that depends upon abstractions rather than upon concrete details)
- **Design pattern :-** Follow a design pattern if it fits the problem space that you are trying to solve. This instantly makes your code much more accessible to fellow programmers
- **Law of Demeter :-** Law of Demeter says that 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

Follow the three laws of Test-Driven Development (TDD), and keep your test code as clean as production code.

- **First Law**: You may not write production code until you have written a failing unit test.
  Second Law: You may not write more of a unit test than is sufficient to fail, and not compiling is failing.
  Third Law: You may not write more production code than is sufficient to pass the currently failing test.

Functions

- A Function should do one thing only and do it really well.
  - A function should not return null
  - Do not pass boolean into a function, as there is supposed to be a if statement. Which causes it to do more than one thing.
  - A function should either do something or return something. if a function is returning boolean, then it should not do any operation with in a function.
  - Prefer Exceptions to returning Error Codes and extract error handling `try catch` into their own function.
  - Function should be read like a newspaper, It will be a good practice to avoid long lines of horizontal code. Because Our eyes are more comfortable when reading tall and narrow columns of text.

Object and data structures

- Variables should be private so that we can change their type or implementation when required. There is no need to add getters/setter to each variable to expose them as public.
- Hiding implementation is not just a matter of putting a layer of functions between the variables. Hiding implementation is about abstractions! We do not want to expose details of data but rather express data as abstract terms

Tests

Clean tests should follow F.I.R.S.T principles:

- **Fast**: Tests should be fast. They should run quickly. When tests run slow, you won't want to run them frequently
- **Independent**: Tests should not depend on each other. One test should not set up the

conditions for the next test. You should be able to run each test independently and run the tests in any order you like.

- **Repeatable**: Tests should be repeatable in any environment. They should run in the production environment, the QA environment, and even on your laptop while riding home on the train without a network.
- **Self-Validating**: The tests should have a boolean output. Either they pass or fail. You should not have to read through a log file to tell whether the tests pass.
- **Timely**: The tests need to be written in a timely fashion.

Code organisation and design

According to Kent Beck, a design is “simple” if it follows these rules in order of importance:-

• Runs all the tests
• Contains no duplication
• Expresses the intent of the programmer
• Minimises the number of classes and methods

Standards

The good programmers first analyse and keep their focus on the business and understand the feature behind it. They understand the architecture and work on their part accordingly.

On the code side each language have their own coding styles. Coding styles can be like defining variables, commenting codes and the structure of the code.

Why Coding styles

Following the coding style is important as the code built by one developer can be easily understand by another developer if the code is properly commented and have documented methods/classes.

Optimize code for the reader, not for the writer :-

Other developers can come in and pick up your code with less cognitive load to get into the spirit of it, and it's easier to edit other people's stuff. There is style guide available for each language

style guide .

- **Commenting & Documentation :-** Commenting code is good for the readability. but do not comment code which has obvious meaning.
- **DRY Principle :-** DRY stands for `Don't Repeat Yourself`. Also knows as DIE which means `Duplication is Evil`.
  The principle states:

  > "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system."

- **Avoid Deep Nesting :-** Nesting can make code difficult to read. so avoid nesting for the sake of readability.
- **Limit Line Length :-** It will be a good practice to avoid long lines of horizontal code. Because Our eyes are more comfortable when reading tall and narrow columns of text. This is precisely the reason why newspaper have short length horizontally.

In the early stage of your coding read open source code and it will be helpful to you to understand how commenting should be done and the Directory structure of the projects.

Auto-formatters :-

Add the appropriate save hooks to the editors. Following are the formatters that can be used for a particular language.

- Go: gofmt
- Python: pyformat
- C/C++: clang-format
- Java: google-java-format

Compiler warnings, clang-tidy, pylint :-

These tools can help to identify bugs within the editor and it helps to write bug free code fast.

- Go: go vet
- Python: pylint
- C/C++: clang-tidy
- Java: errorprone.info

Automate tests and experiments early on :- Developer should start writing the unit test as early as possible to rapidly develop the code. Unit tests helps developer to identify if they have done something stupid in the code.

Revision #27

Created 6 months ago by Arrow

Updated 5 seconds ago by Arrow