Clean Code

Dimitri Merejkowsky

Mines Paris 2023-2024

Clean Code?

A first definition

Clean code always looks like it was written by someone who cares

Michael Feathers

Why do we want clean code

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."

Martin Fowler

We want clean code because other devs will maintain it in the future.

Code quality

One of the biggest contributors to team productivity . . .

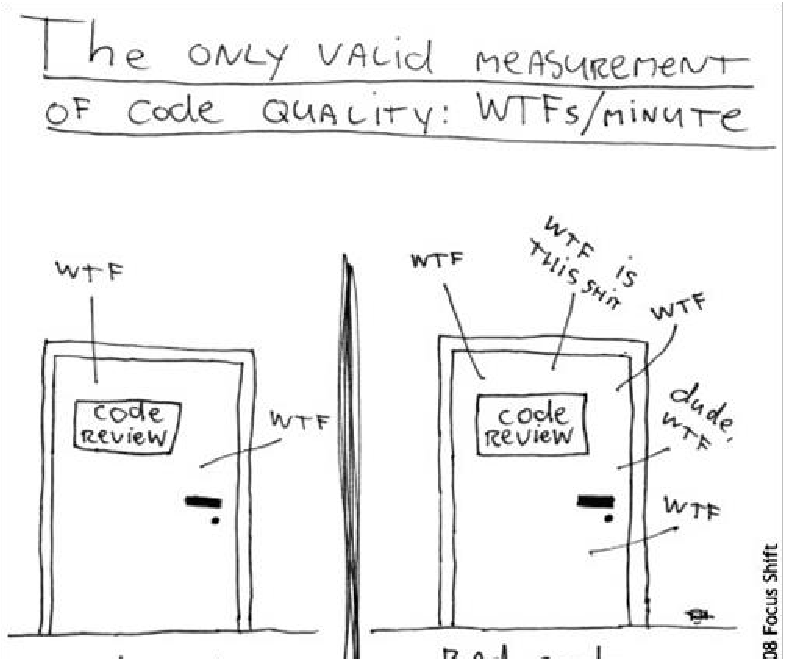
But one which is the most difficult to measure.

Measuring code quality

Code quality is a highly *subjective* metric.

It can vary depending on the context, background and experience of the team.

An illustration



Sad fact

The natural tendency of any code base is for the code quality to *degrade*.

Maintaining code quality is always a challenge.

That's why you should be *continuously refactoring*

Advice

Take your time!

It's usually better to have 80% well done than 100% with lower quality and more rush.

Note: some teams don't apply this rule

Boy scout rule

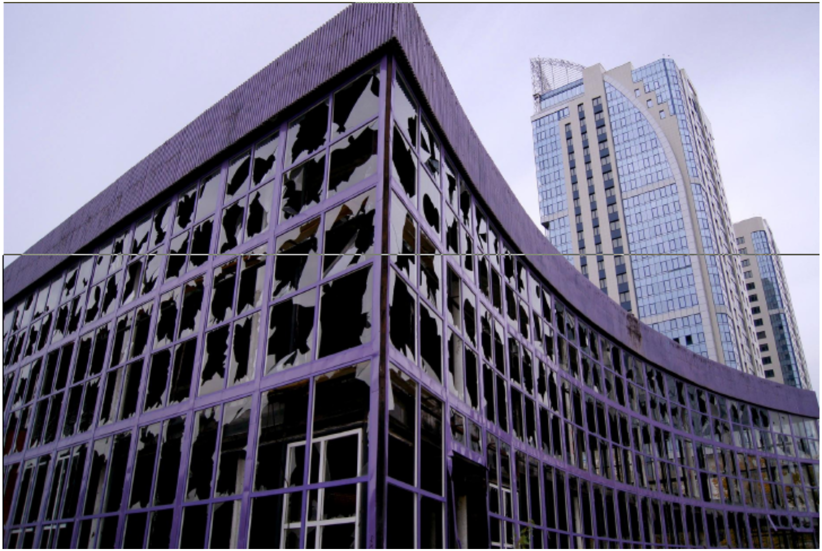
Try and leave the code a little better than you found it.

The psychopath

Always code as if the person who ends up maintaining your code will be a violent psychopath who knows where you live.

Martin Golding

Broken Window



Be careful with metrics

Metrics can be gamed.

Just because it's easy to measure, does not mean it's relevant.

On the other hand, they can be used to *discover facts* about the code and (hopefully) *create new, good habits*.

What the science has to say

Sleep is important

Don't work late ! And make sure to sleep well.

You'll write really bad code when you're sleep deprived, and you'll be *unable to notice how bad you are working*.

Code review works

Go ask other people to read and comment your code!

After all, it’s how open source works :)

. . . and that's all. The *science* of software development has yet to reach consensus on *anything* but the importance of good sleep and code reviews.

And there's debate to know if there are more efficient things than code reviews, by the way.

Clean code basics

Convention - definition

There are several ways of doing the same thing, but the community agrees that one of the ways is better.

Note: often, arguments are used, but remember that every convention is *arbitrary*.

Following conventions

Follow established conventions.

Python has PEP8: https://www.python.org/dev/peps/pep-0008/

Write code in English - unless the core domain is in French.

Use the same tools (IDE...) as your team mates.

Careful: sometimes the conventions are *implicit*.

Comments

Doc Strings

- A special kind of comment

```
def do_stuff(x):
    """
    Documentation for do_stuff() goes here
    """
    ...
```

When should you use doc strings?

Lots of debate.

What I do:

- Put them in *all* public methods if you are writing a library (i.e, code that is going to be used by someone else)
- Don't bother too much otherwise

Comment position

Either:

- on an other line, before the described code
- on the same line, after the described code

```python
# example 1: - a long comment talking
# about foo() and bar() ...
foo()
bar()

x = spam  # short comment about x's assignation
```

Useless Comments (1)

Do not put useless comments:

```
"""
foo.py written by John Doe.
   Last modified on 2021-03-31
"""
```

- We already know the file name because we've just opened it!
- We often know (or don't care) about the author name
- Ditto for date of last change (which you'll forget to update)

This is sometimes called *rotten* comments.

Useless Comments (2)

```python
def add_two(x):
    """
    * Adds 2 to to x
    * @param x : the number to add 2 to.
    * @return the result of adding 2 to x.
    """
    return x + 2
```

We can know all this by looking at the *body of the function*!

Dead code

Do not leave dead code:

```python
def do_stuff():
    # x = do_x_v1()
    # y = do_y_v2()
    # z = combine(x, y)
    x = do_x_v2()
    y = do_y_v2()
    z = combine_v2(x, y)
```

Dead code also *rots* and *smells*.

Use (or learn!) version control instead

Useful comments

Do insert comments when the code is not enough.

Try to describe *why*, not *how*.

Further reading:

https://hackaday.com/2019/03/05/good-code-documents-itself-and-other-hilarious-jokes-you-shouldnt-tell-yourself/

Naming

Naming

One of the hardest problems in computer science

Some rules

- Be consistent
- Don't use abbreviations
- Don't put the type in the name
- Use a descriptive name

```
age = 16  # too short
minimal_age = 16 # nice
minimal_age_to_be_able_to_drive  16  # too long
```

Plural and singular

Use plural if it's a collection.

```
for foo in foos:
    ...
```

Or sometimes there's a nicer word:

```
# An example from Bouygues Telecom
for ligne in parc:
   ...
```

Here, using French is OK

Use good metaphors

Use names you could explain to the rest of the team.

Bad: “Unlock Device”

Good: “Verify Identity”

Rules for variable name size

Big scope, long name

Short scope, small name.

In general, you are allowed one-letter variables names in `for` loops and list comprehensions but almost nowhere else.

Functions

Grammar for function names

Use verbs at the imperative, present tense

```
# Good
def display_tree():
    ...

# Bad
def displays_the_tree():
    ...

# Bad
def tree_display():
    ...
```

Generic vs specific functions

The more generic the function, the shorter the name

```python
def make_coffee(sugar):
    if sugar:
        make_coffee_with_sugar()
    else:
        make_coffee_without_sugar()
```

Returning booleans

Use `is`, or `has`, etc so that the code reads better:

```
if is_allowed_to_drive(person):
    ...
```

How big should my function be?

- Functions should be really short.

Code smell : ‘and’

```
def do_bar_and_foo():
    ...
```

Or:

```
def  first_thing_then_other_things():
  ...
```

Code smell : “step” comments

A function that looks like this . . .

```python
def my_big_function():
    # step 1
    ...

    # step 2
    ...

    # step3
    ...
```

. . . can probably be split in 3 (the comments are the clue)

Code smell : Number of arguments

- 0 to 3 : probably fine
- more than 3 : danger -> consider introducing a struct or class for storing the parameters

Example - too many parameters

```python
class CoffeeShop:
    def order_coffee(self, tall, milk, no_sugar):
        coffee = self.make_coffe(tall, milk, no_sugar)
        self.serve(coffee)

    def make_coffee(self, tall, milk, no_sugar):
        if tall:
            self.add_water()
        if milk:
            self.add_milk()
        if not no_sugar:
            self.add_sugar()
```

Example - using a class for the order

```
class Order:
  def __init__(self, tall, milk, sugar):
    self.tall = tall
    self.milk = milk
    self.sugar = sugar
```

. . . and then:

```
class Coffee:
    def make_coffee(self, tall, milk, no_sugar):
        sugar = not no_sugar
        order = Order(tall, milk, sugar)
        coffe = self.make_coffe(order)
        self.serve_coffee()

    def make_coffe(self, order):
        if order.tall:
            self.add_water()
        if order.milk:
            self.add_milk()
        if order.sugar:
            self.add_sugar()
```

Avoid double negative

Notice how we went from

```
if not no_sugar:
  ...
```

to

```
if order.sugar:
  ...
```

which is more readable.

Better to have a variable with default value of `True`, rather than a negative variable with a default value of `False`.

More or less lines?

- It depends!

Less lines is better

```
if can_vote:
    return True
else:
    return False
```

vs

```
return can_vote
```

More lines is better

```
if (
  age >= 18
  and nationality == "French"
  and name in electors_list(sector)
):
    return "can vote"
```

vs

```
adult = age >= 18;
french = nationality == "French";
registered = name in electors_list(sector)

if adult and french and registered:
  return "can vote"
```

Early return - Before

```python
def return_stuff(arg1, arg2):
    if arg1.is_valid():
        if arg2.is_valid():
            thing = do_stuff(arg1, arg2)
            if thing is not None:
                return "Stuff"
            else:
                # thing was None
                return None
        else:
            # arg2 was not valid
            return None
    else:
        # arg1 was not valid
        return None
```

- “Happy path” in the middle and to the right

Early return - Before

```python
def return_stuff(arg1, arg2):
    if arg1.is_valid():
        if arg2.is_valid():
            thing = do_stuff(arg1, arg2)
            if thing is not None:
                return "Stuff"
            else:
                # thing was None
                return None
        else:
            # arg2 was not valid
            return None
    else:
        # arg1 was not valid
        return None
```

- “Happy path” in the middle and to the right
- Cause of the problem far from the result

Early return - After

```python
def return_stuff(arg1, arg2):
    if not arg1.is_valid():
        return None

    if not arg2.is_valid():
        return None

    thing = do_stuff(arg1, arg2)

    if thing is None:
        return None

    return "Stuff"
```

- Comments are gone

Early return - After

```python
def return_stuff(arg1, arg2):
    if not arg1.is_valid():
        return None

    if not arg2.is_valid():
        return None

    thing = do_stuff(arg1, arg2)

    if thing is None:
        return None

    return "Stuff"
```

- Comments are gone
- Less horizontal space taken

Early return - After

```python
def return_stuff(arg1, arg2):
    if not arg1.is_valid():
        return None

    if not arg2.is_valid():
        return None

    thing = do_stuff(arg1, arg2)

    if thing is None:
        return None

    return "Stuff"
```

- Comments are gone
- Less horizontal space taken
- “Happy path” *after* the “sad path”

Early return - After

```python
def return_stuff(arg1, arg2):
    if not arg1.is_valid():
        return None

    if not arg2.is_valid():
        return None

    thing = do_stuff(arg1, arg2)

    if thing is None:
        return None

    return "Stuff"
```

- Comments are gone
- Less horizontal space taken
- “Happy path” *after* the “sad path”
- Cause of problem right before the return value

Clean code and Classes

Foreword

Classes, composition and inheritance are powerful but *dangerous* tools.

- They hide the control flow.
- It's very easy to make a mess!

Naming

Classes names are *nouns* and start with an uppercase letter.

Avoid meaningless words like `Handler`, `Manager`, `Data`, `Info` . . .

Use inheritance with caution

Lots of powerful and dangerous features:

- Multiple inheritance (for languages that have them)
- Some attribute may be shared (ditto)

Advice

- Only every inherit from *one* parent
- Use inheritance for exceptions
- Use inheritance for abstract base classes (more on this later)

And that's all!

APIs

Definition

Application Programming Interface

A set of functions (or methods) or class you can use as an *external* user of a piece of code

APIs and libraries

Often you use an API from a library.

Example with `datetime`:

https://docs.python.org/3/library/datetime.html

What makes good APIs

- Good naming
- Good metaphors
- Consistency
- Easy to use
- Hard to misuse
- Simple things should be easy, complex things should be possible

Making good APIs

It's *hard*

What can help:

- Brainstorms
- Tests
- Documentation
- Examples
- Review

And you should do all of this *before* writing the production code, because architecture is much harder to change afterwards!

The big problem with APIs

They are hard to change.

Sometimes they *break*, which means *all the code that use them* must change, and that can have really bad consequences.

The bad news

As soon as you *write a function* in a piece of code, you *are* defining an API from the point of view of the rest of the code.

All of the above applies (minus the fact breaking them is not as bad)

Advice

Treat any piece of code as it was part of a public library usable by anyone.