Best practices in writing code

Simona Andrieș

simona.andries@info.uaic.ro

http://profs.info.uaic.ro/~corinfor/teach/IntroP/

Table of contents

1. Clean code

What is clean code?

Clean code can be read, and enhanced by a developer other than its original author.

1. Clean code

Why writing clean code is important?

The broken windows metaphor

A building that has broken windows looks like nobody cares about it. So other people that pass by stop caring. They allow more windows to become broken. Eventually they actively break them.

This applies in code also.

2. Meaningful names

Names are everywhere in software.

You have to name your :

- variables
- functions
- arguments
- classes
- source files etc.

Because you do so much of it, **you'd better do it well**.

What follows are some simple rules for creating good names.

2. Meaningful names

1. Use Intention-Revealing Names
   - Choosing good names takes time but saves more than it takes.
   - Change them when you find better ones
   - The name **should answer all the big questions** :
     - why it exists?
     - what it does ?
     - how it is used ?

   If a name requires a comment, then the name does not reveal its intent.

   Example : instead of :

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

   use :

```
int elapsedTimeInDays;
```

2. Meaningful names

2. Avoid disinformation

- avoid words whose entrenched meanings vary from our intended meaning.

  `hp`, `aix`, and `sco` could be disinformative would because they are the names of Unix platforms.

- do not refer to a grouping of accounts as an accountList unless it's actually a List.

  The word list means something specific to programmers. If the container holding the accounts is not actually a List, it may lead to false conclusions.

2. Meaningful names

- Beware of using names which vary in small ways.

How long does it take to spot the difference between :

```
XYZControllerForEfficientHandlingOfStrings
XYZControllerForEfficientStorageOfStrings
```

2. Meaningful names

- Don’t change one name in an arbitrary way just to satisfy a compiler or interpreter.

- Number-series naming `(a1, a2, .. aN)` is the opposite of intentional naming.

Such names are not disinformative—they are noninformative.

Example:

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

2. Meaningful names

```
public static void copyChars(char source[], char destination[])
{
     for (int i = 0; i < source.length; i++) {
          destination[i] = source[i];
     }
}
```

2. Meaningful names

- **Use Pronounceable Names**

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

Use names composed of verbs and nouns rather than using abbreviations

2. Meaningful names

- **Use Searchable Names**

Avoid using single-letter names (you should use them **only** as local variables inside short methods)

The length of a name should correspond to the size of its scope.

Example :

```
int e = 0;
```

The use of the variable *e* is hard to find.

Avoid using numeric constants.

It’s easier to search for the usage of MAX_CLASSES_PER_STUDENT, than for the number 7, because the number 7 can have different meanings in different contexts.

2. Meaningful names

- original source code :

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

- refactored source code :

```
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 = (realTaskDays / WORK_DAYS_PER_WEEK);
      sum += realTaskWeeks;
}
```

2. Meaningful names

- **Avoid Mental Mapping**

Readers shouldn't have to mentally translate your names into other names they already know.

Avoid choosing variable names such as : `a, b, x1, y`.

A loop counter may be named `i` *or* `j` *or* `k` *(though never* `l`*!) if its scope is very small and no other names can conflict with it.* This is because those single-letter names for loop counters are traditional.

In most of the situations single-letter names are a poor choice.

*Don't use the name **c** because **a** and **b** were already taken.*

2. Meaningful names

- **Method Names**

Methods should have verb or verb phrase names, describing the action they execute.

Example : *postPayment, deletePage, save, create, updateCustomer*.

Accessors should be named for their value and prefixed with `get`, `set`.

```
string name = employee.getName();
customer.setName("mike");
if (paycheck.isPosted())...
```

2. Meaningful names

- **Pick One Word per Concept**

Pick one word for one abstract concept and stick with it, in order to ensure the consistency of your system.

Example :

- fetch
- retrieve
- get

2. Meaningful names

- **Add Meaningful Context**

Place the chosen names in context for your reader by enclosing them in well-named data structures or functions. When necessary, you can prefix the as a last resort.

Example :

```
firstName, lastName;
street, houseNumber;
city, state, zipcode;
```

Do they form an address?

You can add context by using prefixes: `addrFirstName`, `addrLastName`, `addrState`, etc.

Better : use a data structure named `Address`.

2. Meaningful names

Conclusion

What is difficult in choosing good names?

The hardest thing about choosing good names is that it requires good descriptive skills and a shared cultural background (is a teaching issue rather than a technical, business, or management issue).

Follow some of these rules and see whether you don't improve the readability of your code. It will pay off in the short term and continue to pay in the long run.

3. Functions

First rule :

Functions should be small !

Second rule :

Functions should be smaller than that !

3. Functions

- **Blocks and Indenting**

Blocks within if statements, else statements, while statements, and so on should be one line long.

If they are longer, **introduce a new function** !

This keeps the enclosing function small and adds documentary value because the function called within the block can have a nicely descriptive name.

Functions should not be large enough to hold nested structures. Therefore, the indent level of a function should not be greater than one or two.

This makes the functions easier to read and understand.

3. Functions

- **Functions should do one thing !** *(single responsibility principle)*

How to test if a functions does only one thing :

1. Should have only one answer for the question : “What does the function do?”

2. If you can extract another function from it with a name that is not merely a restatement of its implementation

- **Reading Code from Top to Bottom: *The Stepdown Rule***

The code should be read like a top-down narrative

3. Functions

- **Use Descriptive Names**

It’s nothing wrong in introducing a long name.

A long descriptive name is better than a short enigmatic name.

A long descriptive name is better than a long descriptive comment.

Use a naming convention that allows multiple words to be easily read in the function names.

Example : `calculateEmployeeSalaryBonus`

Be consistent in your names ! Use the same phrases, nouns, and verbs in the function names you choose for your modules.

3. Functions

- **Function Arguments**

Based on the number of arguments, the functions can be :

  - Niladic – zero number of arguments (ideal case)
  - Monadic
  - Dyadic
  - Triadic – should be avoided
  - Polyadic – shouldn’t be used

3. Functions

- **Common Monadic Forms**

Common reasons to pass a single argument into a function :

I. You may be asking a question about that argument.

```
boolean fileExists("MyFile")
```

II. You may be operating on that argument, transforming it into something else and *returning it*.

```
InputStream fileOpen("MyFile")
```

Other case for monadic forms : *events*.

In this form there is an input argument but no output argument.

The overall program is meant to interpret the function call as an event and use the argument to alter the state of the system.

```
void passwordAttemptFailedNtimes(int attempts)
```

3. Functions

- **Flag Arguments –** avoid using them

Passing a boolean into a function is a truly terrible practice.

It violates the single responsibility principle for the function (the function does one thing if the flag is true and another if the flag is false).

It immediately complicates the signature of the method.

3. Functions

- **Dyadic Functions**

Dyadic functions are harder to understand than monadic functions.

```
writeField(name)
writeField(output-Stream, name)
```

There are cases, of course, where two arguments are appropriate.

```
Point p = new Point(0,0);
```

- **Triads**

Functions that take three arguments are significantly harder to understand than dyads.

You got to have a very good reason for introducing a function that takes 3 arguments.

3. Functions

- **Argument Objects**

When a function seems to need more than two or three arguments, it is likely that some of those arguments ought to be wrapped into a data structure of their own.

```
Circle makeCircle(double x, double y, double radius);
Circle makeCircle(Point center, double radius);
```

X and Y are part of a concept that deserve a name of its own, so wrapping them in a separate data structure is a good idea.

3. Functions

- Conclusion

If you follow the previously presented rules, your functions will be :

short

well named

nicely organized

Never forget that your real goal is to tell the story of the system, and that the functions you write need to fit cleanly together into a clear and precise language to help you with that telling.

4. Comments

Comments are not “pure good”. Comments are, at best, a **necessary evil**.

The proper use of comments is to compensate for our failure to express ourself in code. Comments are always failures.

We must have them because we cannot always figure out how to express ourselves without them, but their use is not a cause for celebration.

So when you find yourself in a position where you need to write a comment, think it through and see whether there isn’t some way to turn the tables and express yourself in code.

4. Comments

• Comments Do Not Make Up for Bad Code

One of the more common motivations for writing comments is bad code.

Rather than spend your time writing the comments that explain the mess you've made, spend it cleaning that mess.

• Explain Yourself in Code

There are certainly times when code makes a poor vehicle for explanation, but most of the time it's easy to express yourself through code

Which would you rather see? This:

```
// Check to see if the employee is eligible for full benefits
if ((employee.flags & HOURLY_FLAG) &&
                (employee.age > 65))
```

Or this?

```
if (employee.isEligibleForFullBenefits())
```

It takes only a few seconds of thought to explain most of your intent in code. In many cases it's simply a matter of creating a function that says the same thing as the comment you want to write.

4. Comments

- **Good Comments**

Some comments are necessary or beneficial.

Keep in mind, that the only truly good comment is the comment you found a way not to write.

- **Bad Comments**

Most comments fall into this category.

Usually they are excuses for poor code or justifications for insufficient decisions.

4. Comments

Rule : Don’t Use a Comment When You Can Use a Function or a Variable

- **Position Markers**

Sometimes programmers like to mark a particular position in a source file.

```
// Actions ////////////////////////////////
```

There are rare times when it makes sense to gather certain functions together beneath a banner like this.

- **Closing Brace Comments**

Sometimes programmers will put special comments on closing braces.

If you find yourself wanting to mark your closing braces, try to shorten your functions instead.

4. Comments

- **Commented-Out Code**

Few practices are as odious as commenting-out code.

- **Nonlocal Information**

If you must write a comment, then make sure it describes the code it appears near.

- **Function Headers**

Short functions don’t need much description.

A well-chosen name for a small function that does one thing is usually better than a comment header.

4. Comments

- Conclusion

When you feel the need to introduce a new comment, it's better to extract a function with a descriptive name or introduce variable

5. Formatting

- **Vertical Formatting**

How big should a source file be?

- **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.

5. Formatting

- **Vertical Distance**

Concepts that are closely related should be kept vertically close to each other .

- **Variable Declarations.**

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

Because our functions are very short, local variables should appear a the top of each function.

- **Dependent Functions.**

If one function calls another, they should be vertically close, and the caller should be above the callee, if at all possible.

This gives the program a natural flow.

5. Formatting

- **Indentation**

To make the hierarchy of scopes visible, we indent the lines of source code in proportion to their position in the hierarchy.

Statements at the level of the file are not indented at all.

Methods within a class are indented one level to the right of the class.

Implementations of those methods are implemented one level to the right of the method declaration.

Block implementations are implemented one level to the right of their containing block, and so on.

6. Testing your code

- **The Three Laws of TDD –** *test driven development*

First Law You may not write 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.

7. Summary

- Comments :

C1: Inappropriate Information(ex: author, last modified-date)

C2: Obsolete Comment

C3: Redundant Comment

A comment is redundant if it describes something that adequately describes itself.

```
i++; // increment I
```

C4: Poorly Written Comment

If you are going to write a comment, take the time to make sure it is the best comment you can write

C5: Commented-Out Code

7. Summary

- **Functions**

F1: *Too Many Arguments*

Functions should have a small number of arguments.

F2: *Output Arguments*

Readers expect arguments to be inputs, not outputs. If your function must change the state of something, have it change the state of the object it is called on.

F3: *Flag Arguments*

Boolean arguments states that the function does more than one thing.

F4: *Dead Function*

Methods that are never called should be discarded.

7. Summary

- **General**

G1: *Inconsistency*

If you do something a certain way, do all similar things in the same way.

G2: *Function Names Should Say What They Do*

```
Date newDate = date.add(5);
```

Would you expect this to add five days to the date? Or is it weeks, or hours?

Better use : `addDaysTo` or `increaseByDays`.

G3: *Replace Magic Numbers with Named Constants*

The number 86,400 should be hidden behind the constant SECONDS_PER_DAY.

7. Summary

- **G4: *Encapsulate Conditionals***

Boolean logic is hard enough to understand without having to see it in the context of an `if`

or `while` statement.

Extract functions that explain the intent of the conditional.

Example:

```
if (shouldBeDeleted(timer))
```

is preferable to

```
if (timer.hasExpired() && !timer.isRecurrent())
```

7. Summary

- **G5: *Functions Should Do One Thing***

It is often tempting to create functions that have multiple sections that perform a series of operations.

Functions of this kind do more than one thing, and should be converted into many smaller functions, each of which does one thing.

7. Summary

- **Names**

N1: Choose Descriptive Names

This is not just a “feel-good” recommendation. Names in software are 90 percent of

what make software readable. You need to take the time to choose them wisely and keep

them relevant.

N2: Use Long Names for Long Scopes

The length of a name should be related to the length of the scope. You can use very short

variable names for tiny scopes, but for big scopes you should use longer names.

N3: Avoid Encodings

Names should not be encoded with type or scope information

Bibliography

- **The Clean Coder:** A Code of Conduct for Professional Programmers **(**Robert C. Martin Series)

- **Clean Code**: A Handbook of Agile Software Craftsmanship (Robert C. Martin Series)