A short, pocket-hand notes on how to write good code

By Franco Montenegro

Who are you coding for

Let's begin really simple and by establishing this fact that more than often gets forgotten by developers. Code is written for and by developers, this means:

- it has to be readable, **easily**
- do not try to be smart
- do not hide behind clever abstractions
- do not over engineer solutions
- leave comments and notes

But, this is harder than what it looks like, specially with how the "business" and the market is, where everyone is trying to push features like crazy and get some money so we can rush to the next thing. Now, do not get my wrong, I understand the need of doing so and I'm not complaining, this is why this book is written taken this into consideration and any other constrains such as deadlines as well. If you think the current state of the market is sad because of this, think of it as a challenge, are you able to write good code under this constrains? if you had all the time in the world nothing would get done and it would just be plain boring, so, let's crack into it!

Names

Simple and strait forward, names need to be short, sweet and to the point. If you can't think of a good name, leave a comment and make sure that, when you have a deeper understanding of the problem you come back and refactor. Sometimes when we can't name our variables, functions and any other components properly it's because we don't quite understand the problem we are solving yet, and that perfectly normal and valid, as long as, when you gain deeper knowledge you come back and clean after yourself by choosing a more descriptive name.

```
void n()
{
    int p = 42;
    return p;
}
```

Extreme example, I know, but let's refactor:

```
int get_maximum_age()
{
    int maximum_age = 42;
    return maximum_age;
}
```

Short, sweet and to the point, no ambiguity whatsoever.

Code style

The key point here is to be consistent, it doesn't matter if you wanna use tabs or spaces for your indentation; or if you wanna use camel case, snake case, etc., just focus on being consistent. Another important key point is that, every project is like a house. When you are in your friend's house you need to play along their rules and be respectful, you don't act like crazy and nor should you do so with a project. Learn what are the rules of your project, learn the code styling they are using and apply it. You can provide suggestions, but you do not want to enforce anything.

If you can start your project with an automatic code formatter that's pretty good and will save you some time. If on the other hand the project already exists, try formatting it all at once, this will avoid dealing with huge commits because of the formatter doing it's job on everyone's changes. Make sure that, if you do so, you let your entire team know to avoid dealing with merge conflicts.

Over engineer

Do not start trying to hit the perfect implementation, API or abstraction, you wont. The reason is that, you are trying to perfect something that doesn't even exist yet, which means, you are establishing facts based on things that, again, you don't still know. What you will end up with is, a clearly forced over engineer abstraction and this is one of the worst things to end up with.

Missing an abstraction is fine, we can always come back, refactor and built one based on the errors/imperfections we can **clearly** see, but, having the wrong abstraction could lead you to much worse results. This is because you are enforcing a way of doing things.

Expanding on "clearly" see the errors/imperfections

Every single project you come in, you probably are disgusted by how things are built and, if you are not, I'm very happy for you, you found a pretty good project to work on and learn from!

Now, going back to reality, probably you are on a project where you can see tons of errors or things that wanna make you go crazy and hunt down the previous developer. Just don't, and let me tell you why. When that developer was working on the crap you are sitting on, he/she didn't knew the things you now can clearly see, such as, patterns, structures. You are literally seeing his/her experience on your monitor, so let's learn from it and **slowly** and **safely** refactor, avoiding the mess from getting bigger.

Slow and safe refactoring

Going back to my previous suggestion on slowly and safely refactoring, I'm making special emphasis on **slowly** and **safely** refactoring because you don't wanna be the new developer who came in and broke the entire system just because a refactor. That won't gonna cut it.

I understand your urge to go and clean everything up, after all, you see all this mess piling up for quite some time now, but that's not how the world works and it's simply not a very good strategy. This is a team effort, a task that will take time and that needs to be done by everyone little by little. This allows things to not break from one day to another, and it also allows your teammates to become familiar with the new changes.

Coding things you don't quite understand yet

Sometimes we get a rough idea on how the system, library, component will work but we are not clear on the inner details, sometimes is quite the opposite, we can clearly see how the internal components will work but we haven't yet decided how this will integrate in the bigger picture. My advice to this is to simple begin with the things you currently know and don't assume anything you are not sure about. Begin simple and dirty, you will be able to refactor it later when you clearly see the errors/imperfections.

Let's go with an example, we need to build a function that allows to move an entity from A to B.

We could start with

```
void move_entity(...)
{

}
```

Now what?, do we ask for the whole entity in the parameters? do we create a new structure to hold coordinates with X and Y? what if we later decide to add 3D, we would also need Z in the structure, should we add it? what if...

You get the idea, just stop and work with the details you got and know.

```
void move_entity(struct Entity entity, int to_x, int to_y)
{
    entity.x = to_x;
    entity.y = to_y;
}
```

Simple right?, now it's the time to refactor:

```
struct Location
{
    int x;
    int y;
};

void move_entity(struct Entity entity, struct Location to)
{
    entity.x = to.x;
    entity.y = to.y;
}
```

Can we do better? well it would make sense to refactor `struct Entity` to have a `struct Location` instead of plain `x` and `y`. But, before doing so make sure you do a slow and safe refactor.

Comments

Some people will tell you that you don't really need comments, that your code should be descriptive enough to tell the developers reading it everything they need to know. Others will tell you, comments are essential and that you need to write comments for every single function you got. Being honest, I went through both of these statements, and my suggestion would be to apply both. There is code that definitely needs comments and some that simply don't, here is an example:

```
int get_pi()
{
    return PI;
}
```

Would you add comments to that function? of course no. Would you comment every single line of comment?

```
int i;
// Set i to 42
i = 42;
```

Again no, there is no need for that, the code is already telling us that. But what about intention?, why are we setting `i = 42`? have you thought about that?

And this is the kind of comments projects usually need, intention, and notes that we can't clearly and simple see by just reading the code.

Understanding what the code does

If something you wrote works but you don't know why, that's a ticking bomb ready to explode in the near feature. Not only that, it's very unprofessional, imagining yourself not being able to answer to your client why and how does this works.

Understand the instructions you are giving to the computer, and if you don't, it may suggest that either you don't fully understand the problem you are solving or, your code is not simply easy to follow/read. In any case, go back until you get a simple solution.

Parameters

TODO

Clear responsibility on functions

TODO

Design patterns

TODO

Handling errors

TODO