

Clean Code

There's no problem so simple that a bad developer can't make it complicated

Steve Bohlen

Programming is the art of telling another human what one wants the computer to do

Donald Knuth

Breadcrumbs

Tech Lead at Compufácil compufacil.com.br

Github: [jeanCarlomachado](#)

Contributor of:

- ▶ Doctrine
- ▶ Zend Framework
- ▶ PHPmd

Twitter: [JeanCarloMachad](#)

Researching optimization based on nature inspirations

What Clean Code stands for?

Reader-focused development style

Produces software that's easy to:

- ▶ write
- ▶ read
- ▶ maintain

Why it matters?

Broken windows theory

One *broken window* is all it takes to start the decline.

“All the rest of this code is crap, I'll just follow suit.”

Software entropy, software rot.

Why it matters?

Take the time to go quick.

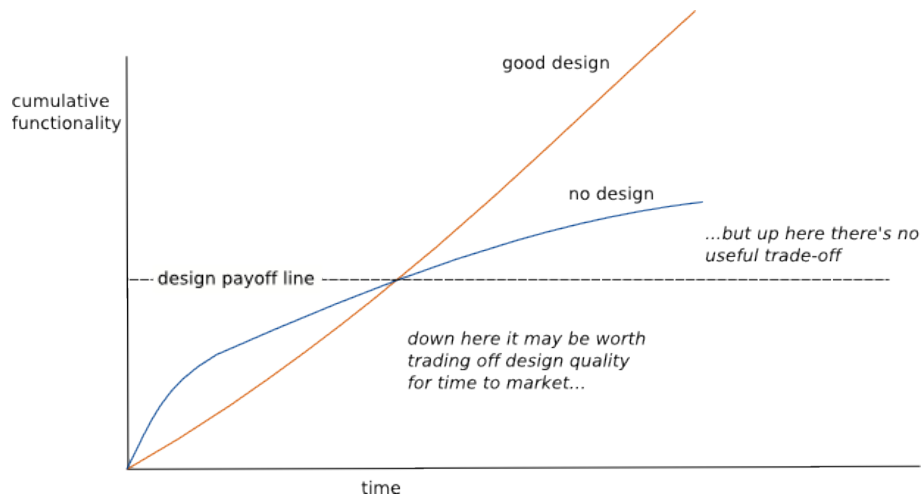


Figure 1: Features over time considering design quality

The four characteristics of Rotting software

- ▶ Rigidity
- ▶ Fragility
- ▶ Immobility
- ▶ Viscosity

Rotting software 1 - Rigidity

It's hard to solve simple problems.

Estimating is hard.

Rotting software 2 - Fragility

The software breaks too often.

A change in one unrelated part breaks others.

Changes must be echoed in many places.

Rotting software 3 - Immobility

It's the inability of reusing software from other places.

Rotting software 4 - Viscosity

It's easier to go to the *hacking mode* than to the *design preservation* mode.

Implications

- ▶ Software rot implies in frustrated developers
- ▶ Frustrated developers implies in more rotting
- ▶ Too much rooting implies in system rewrite
- ▶ Loss of money
- ▶ Loss of users

Solution?

- ▶ Good practices
- ▶ Software craftsmanship
- ▶ Clean code skills

Clean coder skills

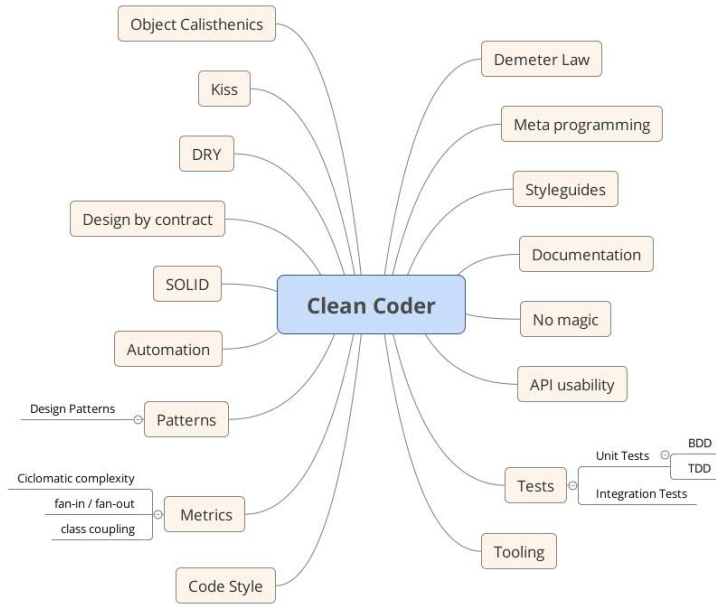


Figure 2:clean-coder-skills.jpg

Clean Coder Skills

Follow the ones what suites you most.

*Relatively simple things can tolerate a certain level of disorganization.
However, as complexity increases, disorganization becomes suicidal.
Robert Martin*

Naming

Need to see the source for to know what a function does? Work on names!

Longer names are generally better than smaller one's.

No And or Or in names.

Comments

Usage scenarios

Put in the dock block at least the authors name.

Comments serves as well to discuss the purpose and trade-offs of implementations.

Avoid scenarios

The usual aim of comments is to express the code.

Inaccurate comments are way worse than no comments at all.

DRY

Don't Repeat Yourself

DRY on comments

```
/**  
 *  
 * @param $title The title of the CD  
 * @param $author The author of the CD  
 * @param $tracks The number of tracks of the CD  
 *  
 */  
public addCd($title, $author, int $tracks);
```

DRY on documentation

Code and documentation are different views of the same model

Two places to edit models? DRY violation

Generate code from diagrams or diagrams from code

zendframework/ZendDeveloperTools

Methods Arguments

The ideal number of arguments of a method is ZERO.

More than tree is unacceptable.

Flags

```
public function useResultCache($bool, $lifetime = null/**...*/)
{
    if ($bool) {
        $this->setResultCacheLifetime($lifetime);
        $this->setResultCacheId($resultCacheId);

        return $this;
    }
    //...
}
```

Classes - Journal Metaphor

Classes should be like journal articles.

In the header you get an general overview.

You are able to decide if you go further or not.

As you read down details increases.

Objects vs Data structures

Objects hide data and expose operations over it.

Data structures expose data and have no meaningful operation.

KISS - Keep It Simple Stupid

Related with few parts, few responsibilities, decoupling

Simplicity is different of easy

Easy is more related with your current understandings (relative)

There are systems which are easy but not simple

Over time simplificy pays off easiness

*UNIX is very simple, it just needs a genius to understand it's simplicity.
Dennis Ritchie*

No Magic

Before you commit to a framework, make sure you could write it

Do this by actually writing something simple that does the basics that you need

Write some assembly

Make sure all the magic goes away

Remove is better than adding

Don't let existing code dictate future code

Be ready to refactor

Build a *state of art* namespace

Orthogonality

Two or more things are *orthogonal* if changes in one do not affect any of the others

Benefits of Orthogonal Systems

- ▶ Changes are localized
- ▶ Disease sections of code are isolated
- ▶ The result system is less fragile

Tests

TDD is a design process

Prove the parts to prove the hole

Invert dependencies

Use gateways

What make testing harder is a bad in itself

Tests are the best documentation

```
test$method_$expectedResult_$conditions
```

Law of Demeter

Don't play with your toy's toys

If you need to change an object's state, get the object to do it for you

Any method of an object should call only methods belonging to:

- ▶ itself;
- ▶ any parameters received;
- ▶ any objects it creates and any directly held component objects

Composite Reuse

One should build upon interfaces

OO languages replace function pointers with convenient polymorphism
Robert C. Martin

Closure protocols and haskell type classe

Benefits

- ▶ Easier to maintain (no unexpected behaviors)
- ▶ Performance gain
- ▶ The inversion of source code and run time dependencies

Many little classes vs Few big ones

Any regular system will contain a vast quantity of logic

The same amount of business logic to care of

You prefer your tools being organized in boxes with little compartments and good names?

Or only a compartment and all inside?

SOLID

Or the “first five principles” by Michael Feathers.

To make systems that are: is easy to maintain and extend over time

Single Responsibility Principle (SRP)

If you can think of more than one reason for changing a class, then that class has more than one **responsibility**

```
<?php
interface UserInterface
{
    public function setId($id);
    public function getId();
    public function setName($name);
    public function getName();
    public function findById($id);
    public function insert();
    public function update();
    public function delete();
}
```

Open Close Principle (OC)

The interface is closed to modification and new implementation implement that interface

Rectangle, Square

Relates to composite reuse

Liskov substitution (LS)

It's possible to change subclasses without breaking the program

Usually `instanceof` is a sign of code smell

Interface segregation Principle (ISP)

It's better more interfaces than less

Clients should not be forced to depend upon interfaces that they don't use

```
abstract class AbstractPostgreSQLDriver implements  
Driver,  
ExceptionConverterDriver,  
VersionAwarePlatformDriver
```

Dependency inversion

One should depend only on abstractions.

For PHP:

- ▶ <https://github.com/container-interop/container-interop>
- ▶ <https://github.com/auraphp/Aura.Di>
- ▶ <https://github.com/zendframework/zend-servicemanager>

Object Calisthenics

Seven code qualities premisses:

- ▶ Cohesion
- ▶ Loose coupling
- ▶ No redundancy
- ▶ Encapsulation
- ▶ Testability
- ▶ Readability
- ▶ Focus

1 - One level of indentation per method

Benefits

Finding bugs is much easier

If you have more than one indentation level you have more than one abstraction level

2 - Don't use ELSE keyword

Else's encourages the inclusion of more, intermediate, if's

When possible avoid even if's

Use polymorphism instead

3 - Wrap all primitives and Strings

Small objects make programs more maintainable

They serves as a container for logic that otherwise would be sparse

4 - First class collections

Any class with a collection shouldn't contain other member variables

5 - One dot per line

Bad:

```
this->myMemberObject  
    ->myMemberObjectMemberObject  
    ->doFoo();
```

Good:

```
this->myMemberObjectMemberObject  
    ->functionThatDoFooToo();
```

Relates to Law of Demeter

6 - Don't abbreviate

Abbreviation because of exhaustive use?

DRY violation

Too long names?

Maybe a SRP problem

7 - Keep all entities small

No classes over 50 lines and no packages over 10 files

8 - No classes with more than two instance variables

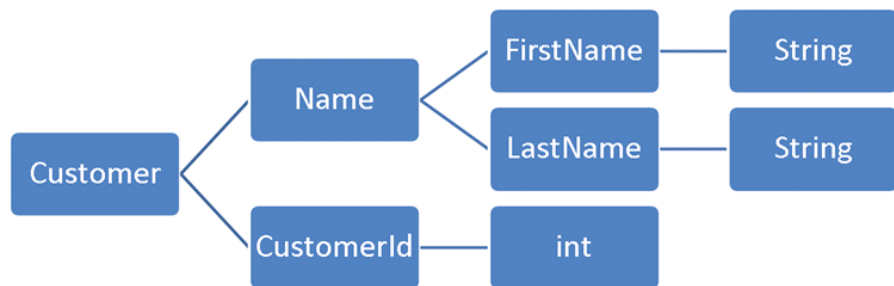


Figure 3:Instance variables

9 - No getters/setters/properties

Encapsulate everything

Make objects change dependencies by themselves

Invert Dependencies

Tools

- ▶ Codacy: <https://www.codacy.com/>
- ▶ <https://github.com/adoy/vim-php-refactoring-toolbox>
- ▶ <https://github.com/object-calisthenics/phpcs-calisthenics-rules>

Style Guides

Follow a coding standard, no matter which, but all the code must follow the chosen one.

- ▶ PSR2
- ▶ Zend
- ▶ Symfony

Literature

1. Clean code: A hand book of Agile Software craftsmanship; Robert C. Martin
2. The pragmatcal programmer; Andrew Hunt
3. Code Complete; Steve McConnell
4. Refactoring: Improving the Design of Existing Code;
5. Release It!: Design and Deploy Production-Ready Software; Michael T. Nygard

Those who do not remember the past are condemned to repeat it. Jorge Agustin Nicolas Ruiz de Santayana y Borrás

Conclusion

Quality is a team issue. Andy hunt.

Teams as a whole should not tolerate broken windows.

Clean code is not about perfection.. It's about honesty.

We made our best to leave the camp cleaner than we find it?

Conclusion

But if we aim for the 80% where code needs the most. We are good enough.
Parts not critical to performance must be clean - not optimized.

The best programmers are 28 times best than the worst ones. Robert Glass, Facts and Fallacies of Software Engineering

There's always room for improvement.

Questions?

Contact info

E-mail: contato@jeancarlomachado.com.br

IRC: JeanCarloMachado

Thanks