The good code and the bad code

Software quality assurance plan template. Or how to speed up software development processes by improving the readability of application source code.

Joel Abenhaim, Lead developer

https://www.linkedin.com/in/joel-abenhaim

Summary

Introduction: The good code and the bad code

Everyone in France knows the sketch from “Les Inconnus” about the good hunter and the bad hunter. Well for the code, it's pretty much this story: There's the good code that runs and does something, and then there's the bad code that runs and does exactly the same thing, except It’s not the same! It's a fun introduction I think, but before talking about that, let me tell you where I come from.

I learned computers on my own, as a child, at home, without teaching. I copied programs in GW-Basic that I found in books and little by little I understood how they worked. Then I started to modify them, then to write my own, as the years went by, more and more complex, more and more large. I wrote the programs haphazardly, in all innocence: “ *If it worked, it had to be perfect the first time, I thought.* »

What I didn't know, and what it took me many failed projects to understand, is that if two codes work the same way at runtime, one of them is more readable than the other, more maintainable than the other, more valuable than the other! And this differential increases with the size of the code, I learned this the hard way. I missed many big projects. In some cases, I had gotten the technology wrong - today we would talk about stack -, and in other cases my programs had reached a critical size but did not have a quality assurance plan.

This notion of critical size is very important. Beyond a certain number of lines of code, around 50,000, or a certain number of source files, around 500, we can no longer afford to code haphazardly. No human memory, including collective memory, is large enough to store all the traps and all the side effects that a code of this size contains. A project created freely by senior developers unfortunately contains a certain number of quality defects which slow down maintenance and the addition of features.

So how do you do it? There are methods to get there. In this document, I will list some of these methods, then I will describe some advantages of them, and finally I will try to explain to what extent they can be applied to your new projects, but also to your existing projects.

Good code can be found

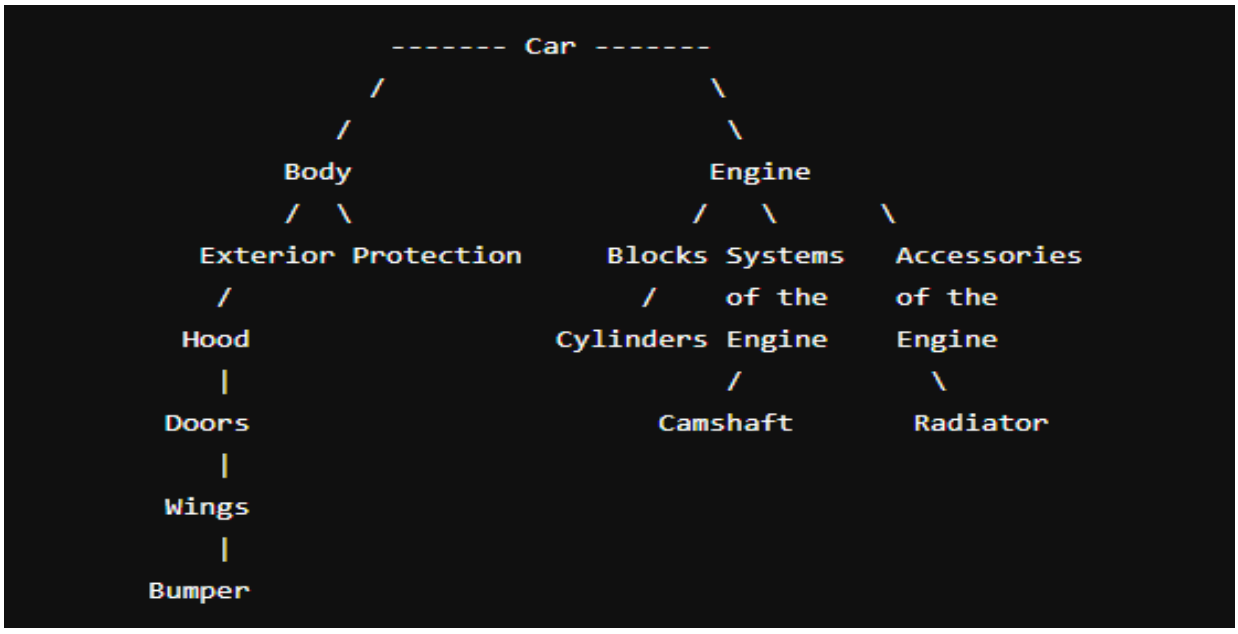
It may seem obvious to say, but the first thing before asking whether the code is well written is that it can be found. And it is the most frequent quality defect in a poorly executed project.

We assume that we want to correct or modify a functionality, a screen or an area of a screen. What we want is to quickly be able to determine which set of source files will need to be modified to accomplish the task. In general, we need to modify between 1 and 10 files, but in certain cases for refactoring it will be necessary to modify a very large number.

To be able to be found quickly, the code needs to be arranged in a certain order. Here are some non-exhaustive methods for classifying code:

- Microservices architecture is one way to categorize code. Each microservice does a particular thing, and that's a first level of classification. We can also use a monolithic architecture with plugins or services, which will classify the code just as well. Choosing whether or not to use microservices doesn't really impact ranking compared to ranking by plugin. It is rather a question of strategy at the project team management level.

- Directory structure is one of the keys to code classification. By using a large number of directories, and creating a deep tree structure, we make it easier to find a piece of the program.



- Likewise, one should separate the code into many files. You shouldn't have too much otherwise you will have "spaghetti" code which is difficult to read. But you have to separate the program source into files of around a few hundred lines if possible. Each file must have a functional scope large enough so that we do not have to open too many files each time, and small enough so that it is not too complicated to modify. And of course, each file name must be chosen carefully so that you know what it does directly by reading its name.

- Formatting lines of code is another way to categorize it. For example, by using an automatic formatter like Prettier, all the lines of code will have the same form, and it will be possible to do textual or regular expression searches, without missing any element.

- Typing is very important for classifying code. From the types of a program: Interfaces, classes, inheritance, composition, variable calls, constants, typed functions, a modern code editor such as Visual Studio Code is able to allow the developer to easily navigate the program.

- For the names of the front-end components, whether we use vanilla javascript, or tools like Angular, React, Vue, we must be able to easily find the elements, by naming them in the same way as the DOM elements, or in the same way as the framework elements. We often have an extension for the browser which allows the views of the interface to be debugged depending on the framework used, and the name of each component must be easily found in the program using its file name or class or function. As much as possible, use the same name everywhere for each element.

  This is an error that I saw in a project, where the developers had named interface components with names out of nowhere, for example *project-button*. To find the source code of the elements visible on the screen, it was extremely difficult and it was necessary to dig and almost do investigations. This problem would have been avoided by properly naming views and classes the same.

Good code is typed

Many web and node.js projects are written in javascript which is an untyped language. And that's a shame ! This is an industry standard today, and it is questionable. Some companies believe that transitioning from JavaScript to TypeScript is difficult or risky for their developers, but it's also important to consider the benefits of typing.

Typing was invented in Fortran over 50 years ago. This is very beneficial for modern programming work. This allows:

- to firmly define the structures of objects and data types, which are the basis of algorithms
- to detect certain errors before they happen
- to guide the following people who will have to intervene on a part, because the typing prevents misuse of a component
- to modify code without being afraid of breaking everything
- to easily refactor part of the software
- to do semi-automatic programming planning, by modifying a type and letting the compiler find what broke, and what will need to be modified to do the job
- it is also integrated documentation, which modern code editors use to facilitate the programmer's task

The saddest thing is that when you program in JavaScript, you do the typing mentally each time. But since we have an untyped language, we cannot write it in the program and the information is lost in the author's head. All subsequent programmers, including yourself, will have to do the same intellectual work over and over again, for nothing.

When you type a program with Typescript, you have the choice of typing each area of the program with different intensities.

- In some cases, you have to type normally. Mainly we will type the arguments of the functions and their return, the properties of the classes, and the structure of the types and interfaces.
- In the case where we type module entry points, we must type very strongly, sometimes even using parameterized generic types. Because we want to help the programmer who is going to invoke a module that he did not write. We want to prevent him from having errors, and avoid the need for him to read our internal code of the module.
- In other cases, we don't want to type at all. This can be done with any or unknown. This is useful when the code is unnecessarily cumbersome and unreadable. Or when we are typing legacy javascript code little by little.

Here is a typical example of the need for typing. We have a JavaScript program, and we wrote the name of a property of an object incorrectly. We put everything in the same file in the example, but we could easily assume that the object would be the return value of a remote rest api.

```
// Definition of an object with a correct property
let person = {
  name: "John",
  age: 30
};

// Attempt to access a misspelled property
console.log(person.ages); // undefined

// Attempt to use the misspelled property
// This will cause a runtime error because ages is undefined
console.log(person.ages.toString());
```

We're going to deliver this code and we'll get a runtime error. Of course, with unit tests covering this code perfectly, we could have detected the error. But using typescript, look how simple it is.

```
// Definition of an interface for the person object
interface Person {
  name: string;
  age: number;
}

// Definition of an object with the correct property
let person: Person = {
  name: "John",
  age: 30
};

// Attempt to access a misspelled property
console.log(person.ages);
//               ^^^^ <-- Compilation error
```

We directly have a compilation error in the code editor. In fact, by writing the line, we wouldn't even have had the chance to make this mistake, because the editor would have displayed a pop-up with the list of available properties and we would only have to choose it.

This is just a simple example, and there are many other reasons to type the program.

Good code is commented

For a number of years there has been a theory, I would say a fashion, which appeals to around a third of IT developers. According to them, comments should not be written in the source code of programs. The main reasons given are:

- We will update the code but not the comments, which will make the comments out of sync.
- If the code is clear enough, no comments are needed. If we feel the need to add a comment, it is because we need to redo the code to make it clearer, and then remove the comment.

We can refute this theory, as follows:

- Let's quickly go over the first reason and it says a lot about the lack of seriousness of people who think that we are not going to update the comments.
- The second reason is bad because what will actually happen is that the developer will not write a comment, and as a bonus his code will not be clear.

Then let's add the following arguments:

- When you write a program, you have logic in your head. If we don't write a comment, this logic is lost. We're going to have to find it every time we work on this program. We could have crystallized its logic in the program, by writing it in comments, once and for all, and saved this rehearsal time.
- Computer code is not designed to be readable in the first place. It is a language that the computer understands, which serves as a list of instructions to execute. We can indeed and must make it as readable as possible by choosing the identifiers and the flow of the program carefully. But we must not forget that at its core, it is computer code. And if computers are made to understand computer language, developers, humans, are made to understand human language.
- Even if the code is well written, and it is possible to understand it without commenting, it is tiring to do the compilation work in your head. It is much easier to read a comment frame, which explains the structure of the piece of code. By saving this fallacious mental compilation work, the developer will conserve his energy and time to carry out other, more difficult and creative tasks.
- Finally, be aware that code without comments constitutes a de facto union. Because the team that wrote this code is the only one who can modify it quickly. If their code is long enough and dirty enough, they will have the power to challenge their management.

Here is some example code that is very well written, but without comment. It's largely possible but not immediately easy to understand what it does.

```typescript
interface Employee {
  name: string;
  salary: number;
  performance: number;
}

const calculateBonus = (employee: Employee): number => {
  if (employee.performance > 8) {
    return employee.salary * 0.2;
  } else if (employee.performance > 5) {
    return employee.salary * 0.1;
  } else {
    return 0;
  }
}

const employee: Employee = { name: "Alice", salary: 50000, performance: 7 };
const bonus = calculateBonus(employee);
console.log(`${employee.name} will receive a bonus of ${bonus} euros.`);
```

The comments added to the corrected version save valuable proofreading time. And it costs nothing to add them when you write the code, because you know very well what you are doing at that moment.

```typescript
interface Employee {
  name: string;
  salary: number;
  performance: number;
}

// Function to calculate the bonus for an employee based on their performance
const calculateBonus = (employee: Employee): number => {
  // If performance is greater than 8, the bonus is 20% of the salary
  if (employee.performance > 8) {
    return employee.salary * 0.2;
  }
  // If performance is greater than 5 but not more than 8, the bonus is 10% of the salary
  else if (employee.performance > 5) {
    return employee.salary * 0.1;
  }
  // If performance is 5 or less, there is no bonus
  else {
    return 0;
  }
}

// Creating an employee with name, salary, and performance
const employee: Employee = { name: "Alice", salary: 50000, performance: 7 };

// Calculating the bonus for the employee
const bonus = calculateBonus(employee);
console.log(`${employee.name} will receive a bonus of ${bonus} euros.`);
```

You can understand and modify the code without comment, but it takes more time. And that time won't be used to do more interesting things than reread poorly documented code. Programming can be a game, and I've played it all my life, but first you have to make it fun by eliminating the tedious work, and reserving your intellectual power for design and algorithms.

And I will end this chapter with an anecdote:

My father, who worked at Framatome, told me the story of an engineer who had written an assembly program used to simulate nuclear reaction equations. He had written the program on the company's IT support. But he had written the comments separately in a small notebook, with the line numbers. This notebook belonged to him personally. He was able to continue like this for ten years, with his colleagues, without doing anything more, and his management was petrified.

Good code is symmetrical, as much as possible

To make code easy to maintain, it is important to keep it symmetrical. This means using structures, patterns, and conventions consistently throughout the project. Symmetric code is more readable, more understandable and easier to manage. When modifying part of the code, it is crucial to reflect this modification wherever necessary. This ensures consistency and maintainability of the code.

In the following example, we use certain syntax to define the program's functions. This syntax is called anonymous ES6 function.

File `addition.ts`

```typescript
export const add = (a: number, b: number): number => {
  return a + b;
}
```

File `multiplication.ts`

```typescript
export const multiply = (a: number, b: number): number => {
  return a * b;
}
```

This anonymous syntax is completely valid, but at some point, and probably for good reason, a developer decides to change the syntax in one of the files, and use classic JavaScript function syntax.

File `addition.ts`

```typescript
export function add(a: number, b: number): number {
  return a + b;
}
```

File `multiplication.ts`

```typescript
export const multiply = (a: number, b: number): number => {
  return a * b;
}
```

What was needed at that time, and this is typically the work of the Tech lead during code review, was to normalize this modification and pass it on to the other file, to maintain symmetry in the program. As in the following corrected example.

File `addition.ts`

typescript Copier le code

```typescript
export function add(a: number, b: number): number {
  return a + b;
}
```

File `multiplication.ts`

typescript Copier le code

```typescript
export function multiply(a: number, b: number): number {
  return a * b;
}
```

But sometimes, we don't have enough time to do this exhaustively. In times like these, developers need to balance speed and quality. Even if there is no time, it is possible to document the modifications made and note those to be made later. This helps to keep traceability and plan for future improvements.

No rule in software quality assurance is absolute. Programming is more of an art than a science. Developers must use judgment and adapt to the specific circumstances of each project.

Good code is factorized, but not compressed

Good code must be factorized, that is, repetitive parts of the code must be put into reusable functions or modules. However, it should not be compressed to the point that it becomes unreadable.

Factoring code involves grouping similar or identical parts into a single function. This helps reduce repetition, making maintenance easier and reducing the risk of error. For example, if you have a mathematical formula used several times in your code, it is better to put it in a dedicated function rather than copying and pasting the formula everywhere with slight modifications.

Unfactorized	Factorized

`let result1 = (a + b) * (c - d);`
`let result2 = (a + b) * (e - f);`
`let result3 = (a + b) * (g - h);`
	`function calculate(a: number, b: number, x: number, y: number): number {`
	`  return (a + b) * (x - y);`
	`}`
	
	`let result1 = calculate(a, b, c, d);`
	`let result2 = calculate(a, b, e, f);`
	`let result3 = calculate(a, b, g, h);`

If later the formula needs to change, it will be enough to modify the function *calculate* and all occurrences will be updated automatically. This is one of the basics of programming.

But be careful, it is important not to be tempted by pseudo-optimizations which make the code more complex without providing significant benefits. For example, compressing 20 lines of code into a single line using functions like Array.reduceRight, Array.zip, or triple maps with bitwise operators can make the code unreadable and difficult to maintain.

Sometimes, programmers write code that is difficult to understand, which can lead to long-term maintenance issues.

Code too compressed

```
let result = data.map(x => x * 2)
    .filter(x => x % 3 === 0)
    .reduce((acc, x) => acc + x, 0)
    .toString(16)
    .split('')
    .map(c => c.charCodeAt(0))
    .reduce((acc, x) => acc ^ x, 0);
```

Normal code

```
// Double each element
let doubled = data.map(x => x * 2);

// Filter elements divisible by 3
let divisibleByThree = doubled.filter(x => x % 3 === 0);

// Reduce by summing all elements
let sum = divisibleByThree.reduce((acc, x) => acc + x, 0);

// Convert to hexadecimal
let hexString = sum.toString(16);

// Convert to an array of characters
let charArray = hexString.split('');

// Convert to character codes
let charCodes = charArray.map(c => c.charCodeAt(0));

// Reduce with XOR
let finalResult = charCodes.reduce((acc, x) => acc ^ x, 0);
```

Good code is easily readable

There are several ways to program the same functionality. You must choose the most readable way.

If it is difficult to define how exactly a code should be more readable, we must already assume that it must be, which is not obvious to everyone. In fact it is a problem of priority. This priority sequence should most often be followed:

- The first priority is that the software must do what it is supposed to do, and without errors.

- The second priority is that the code is easily readable and modifiable. If you have the choice between several ways of writing the same subroutine, which have the same result at execution, you must choose the one which is the simplest to read, to understand, and to modify.

- How to choose ? You can use one of these two methods:
  - If my memory is erased, and I then need to modify this code, what script will allow me to do so most efficiently.
  - If I abandon this project and resume it in ten years, what writing will allow me to resume the project in the best conditions.

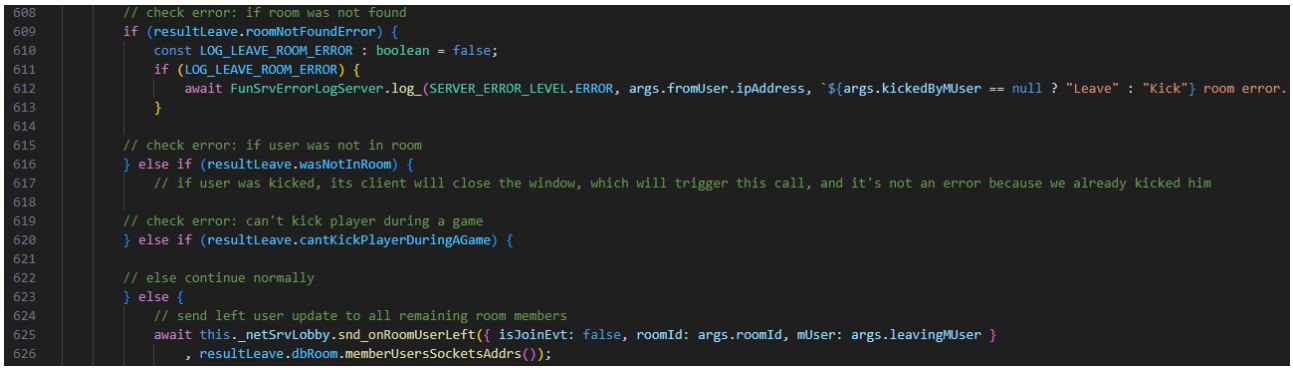
- Everything else: Execution speed, number of lines, code size, etc. is of no importance compared to readability. These criteria should be avoided, unless there is a particular constraint linked to specifications, or a blockage, or a real-time constraint, or the price of the equipment.
  - You should (not) optimize the speed of the code. We don't have to optimize 10 ms on a click, we don't care. We only optimize something that runs regularly on the server, something that slows down the response to the client, that slows down a batch process.
  - It is (not) necessary to optimize the size of the code. If the code is too big, you should use a compressor like UglifyJs. We can also better divide the bundles delivered to the client by http. Modern tools like NextJs take care of doing this work, but it can be configured manually for example in Webpack.

Why should maintainability be prioritized above most other criteria?

- If the code is maintainable, but it has bugs, we will easily fix the bugs.
- If the code is maintainable, but missing features, we will easily add the features.
- If the code is maintainable, but we want to go faster, we will easily add developers.
- On the other hand, if the code is not very maintainable, it has no bugs, it has a lot of features, and we already have lots of developers, it will still have to be modified, and it will still be necessary to replace developers. In this case, it will be extremely difficult to do so. It will be very expensive!

To make code readable, we can use a few principles:

- Identifiers must be explicit. This concerns directories, files, classes, interfaces, types, properties, methods, functions, constants, variables, etc.
- A module, a class, a method, a paragraph, must do one thing, do it clearly, with a simple methodology. The steps of the algorithms are well separated. We are not trying to do two things at the same time.
- Paragraphs must be separated with empty lines. To air out the text. As for a document.
- Useful and numerous comments are needed.
- The pitfalls must be explained with comments. Or avoid pitfalls with strong typing. Or with defensive programming on arguments or intermediate results.
- You should reread yourself immediately after writing each instruction, and ask yourself whether it is easy to understand or not.



Readable production code example

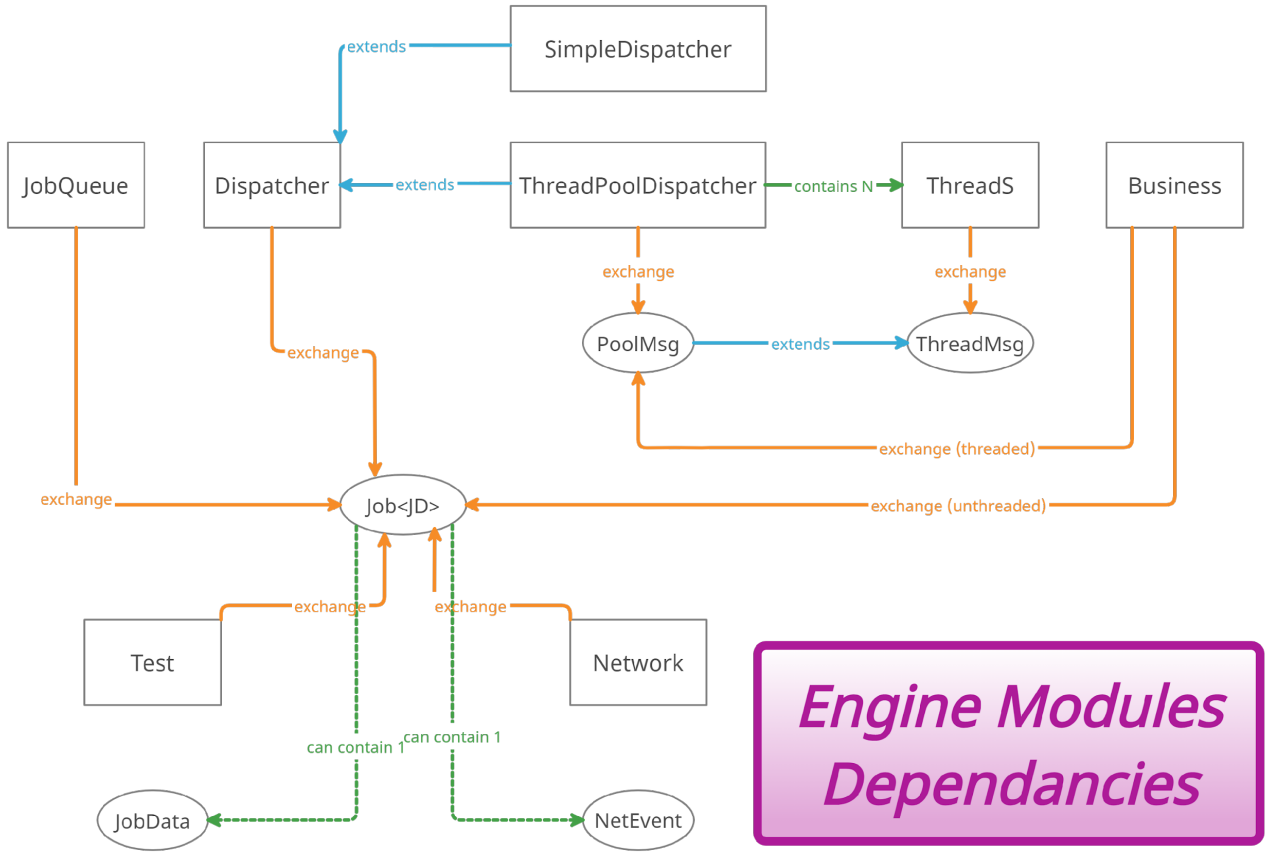
The basic principle: A program is readable if I can easily maintain it after my memory has been erased.

Good code is architected

Good software architecture is based on thoughtful structuring of types and components. The type structure is the basis of any algorithm and architecture. It is from this base that we build the rest of the program, taking care not to adapt the types to the code in an arbitrary manner.

Program components should be isolated from each other and have clear, minimal interfaces for inputs and outputs. This makes it easier to maintain and evolve the code.

For example, here is a diagram representing the central engine of the Player22 software. Note that we don't code as we go, as it comes. On the contrary, everything is planned, and nothing is left to chance. This diagram was created before the engine code, which made the engine code solidly architected.



It is also necessary to limit side effects. To do this, it is obviously necessary to limit the use of global variables.

But more subtly, a side effect occurs when a function or method modifies or uses state external to its own context, which can cause unexpected behaviors. For example, if we have a variable a which is always equal to 1, and in the program we replace the condition *if a=1 and b=2* about *you b=2*, assuming that a will always be equal to 1, we create an edge effect. As soon as a is changed, the code will break. It is therefore crucial to avoid these simplifications and maintain the complete conditions to ensure the stability of the code.

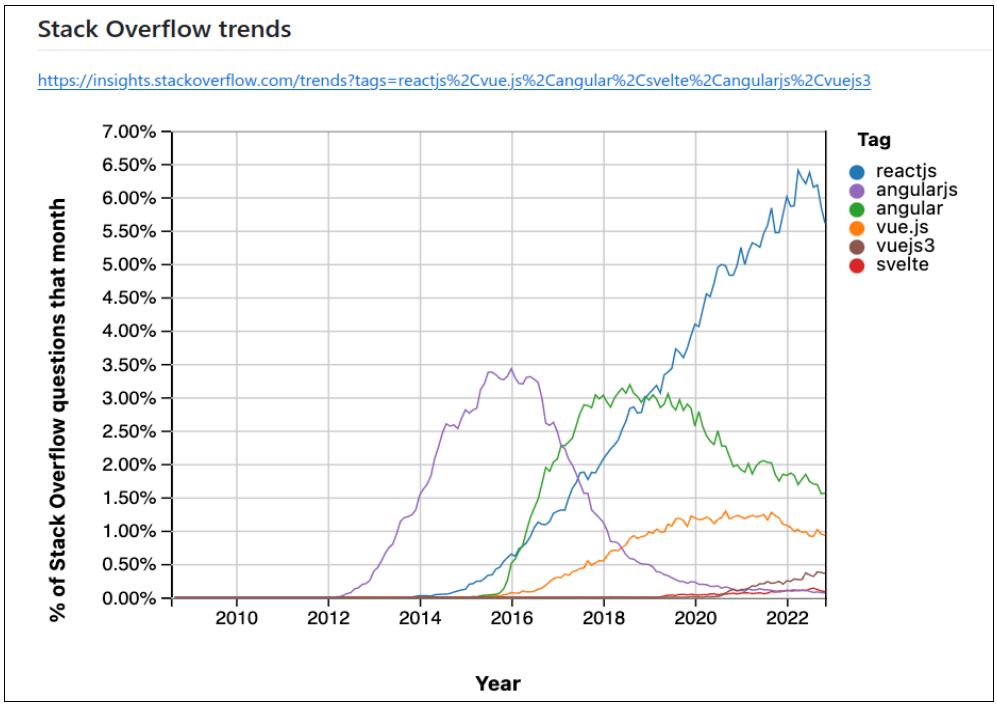
You shouldn't exaggerate either. It is essential to find a balance between adding enough code to avoid side effects and not adding too much to the point of neglecting functional evidence. The developer's experience is the key to guessing what is likely to change and what will remain constant. He will be able to add flexible areas with over-engineering where it is necessary, and write constants or fixed code where he guesses that it will not move.

Good code uses recent and adapted technologies

To explain here, let's take the counter-example of a fictitious company that we will call PortailImmobilier.com. This is a textbook case of technical debt due to stack choice.

PortailImmobilier.com is written with JavaScript, untyped and archaic. The front end uses Jade, which is an eyesore. Jade is being translated to Vue2, which is almost as bad as Jade. In addition, the translation into Vue2 is botched and injects Jade inside.

Here's what Vue2 technology is worth on the market. It is not worth much as you can see in the following graph. And the saddest thing is that it was already not worth much when it was chosen in 2020 by the PortailImmobilier.com teams to replace the Jade technology. On the same graph, we clearly see that React stands out from the crowd.



Source : https://gist.github.com/tkrotoff/b1caa4c3a185629299ec234d2314e190

On the other hand, the back-end uses Express, which is not bad at all as it stands, even if NestJs and especially NextJs are much better today. As for the database in MongoDb, it remains an excellent choice so far, as well as Elastic Search.

While certain technological choices were sensible at the time they were made, they can now represent technical debt. This does not mean, however, that we should reconsider these choices at all costs. These are difficult decisions to make, and which must then be accepted.

Good code uses standard, market-leading technologies

When you use standard, market-leading technology, you get lots of free benefits, and lots of problems solved for free.

- Help and code examples are easy to find.
- Errors are easily fixed by typing the error text into Google and visiting forums such as Stackoverflow, Reddit or Serverfault.
- We are often entitled to quality maintenance. And even version migration solutions that are well put together and easy to apply. It's because so many important businesses use it that everything is done seriously.
- We give value to our employees. A developer who has NodeJs Typescript React experience has a greater value in the market, and their obsolescence date is later.
- We recruit the best profiles more easily, because they want to work on standard and market-leading technologies.
- We onboard more easily technologically. Because there are tutorials that explain exactly how to do it: We do standard! There are even computer science schools that directly teach these technologies to students.

What I have noticed from experience: Often when we choose an outsider technology, and we do so in good faith after analyzing the strengths and weaknesses of the alternatives, we will pay for it a few years later. The leader will eventually catch up with everyone else, and even buy out their engineers, and the underdog technology will move in the wrong direction.

To choose an element of the stack, a dependency, a library, you have to find out about specialized forums, read the opinions and debates between defenders of this or that option. You have to look at the star rating comparisons on GitHub for example, or the popularity of keywords on StackOverflow. You have to look at the last modification dates of the project, the number of maintainers and their credibility, the number of uncorrected bugs. It is necessary to check the health of the company or structure which oversees the project, its history of project abandonment, its backward compatibility policy, its economic model.

In short, you should not choose outbuildings on impulse. You have to do your homework.

Good code is tested

Automated testing is essential to ensure code quality. They ensure that the program works as expected and make it easier to maintain. They detect regression bugs as early as possible.

Types of Tests

- Unit Tests: Check small units of code (functions, methods). Fast, easy to write, accurately locate errors.
- Integration Tests: Verifies that the modules work well together.
- End-to-End (E2E) Testing: Verifies the overall operation of the system from the user's perspective.

Tools

- Jest: For JavaScript/TypeScript, ideal for React.
- JUnit: For Java, widely used.
- Postman: To test APIs.
- Cypress: For integration testing and E2E front-end.

Good practices

- Write Relevant Tests: Reflect real-world scenarios.
- Keep Tests Up to Date: Adapt them with code.
- Prioritize Testing: Cover essential features.
- Automate CI/CD Tests: Integrate into the continuous integration pipeline.
- Reduce Execution Times: Rapid testing to avoid slowing down development.

Benefits

- Early Bug Detection: Reduces fixing costs.
- Facilitating Refactorings: Refactor with confidence.
- Quality Improvement: Encourages good practices.
- Reduction of Maintenance Costs: Facilitates maintenance and development.

By applying these practices, we guarantee high quality code that is easy to maintain and evolve. We won't be afraid to break everything with each modification or refactor.

Other considerations

The quality of software does not only depend on the stack, code and tests. There are other equally important aspects to consider to ensure reliable software.

Another aspect that should not be overlooked is the use of a linter. Properly calibrated, it will perform an automatic and free code review.

We can also cite project management processes: for example agility, code reviews, documentation of commits and pull-requests, peer programming, etc.

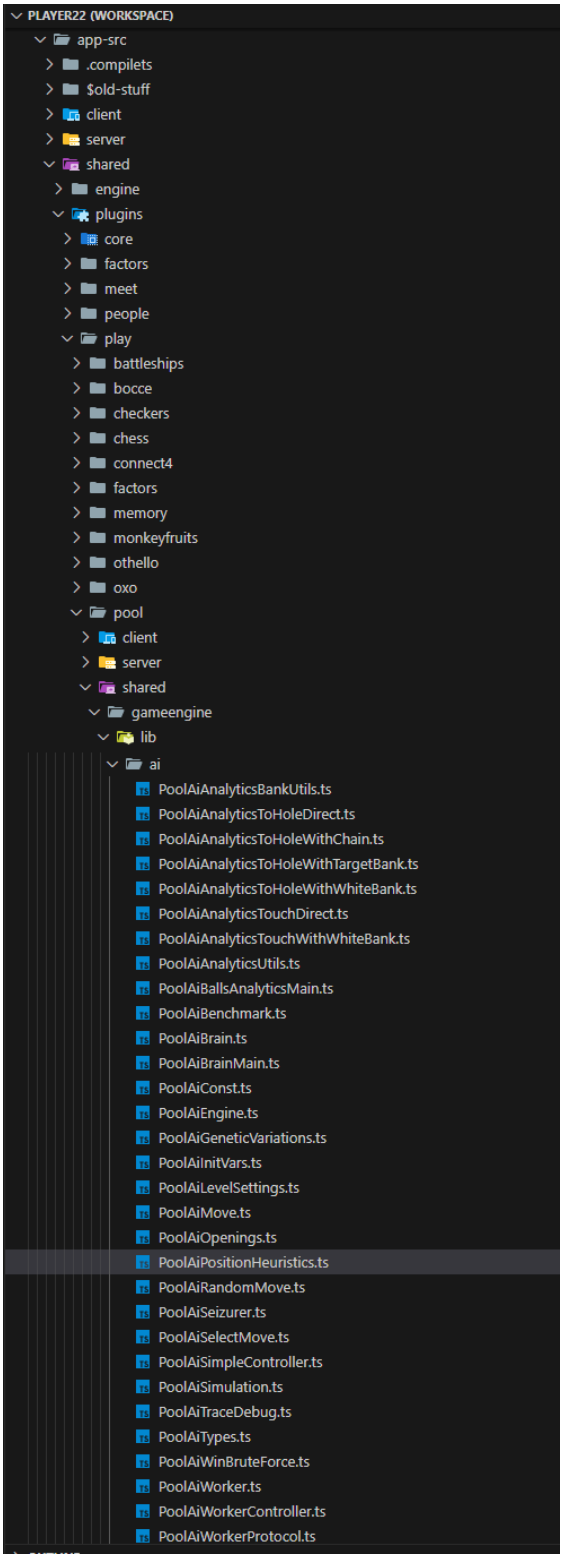
We can mention DevOps practices, such as setting up a CI/CD pipeline and delivering often. These elements are essential for the continuity and stability of development, but they are not included in this document.

It is also necessary to produce documentation concerning the replication of a development environment, and to update it regularly. And do the same with production environments, including if some automated scripts probably already exist.

Although these aspects are a bit off-topic and not covered here, since this document specifically deals with the quality of the code itself, they are important elements in ensuring the overall quality of the software.

Comparison: « Quality » code

Here is how I arranged the directories and files, for example to store the artificial intelligence files of the snooker game, on the Player22 program:



We can notice several things:

- Firstly, the depth of the files, a sign of fine classification, which will allow navigation in the code. Since first of all, you have to be able to find the code. In this example, the AI file on which we are going to zoom is in the path

  shared\plugins\play\pool\shared\gameengine\lib\ai\PoolAiPositionHeuristics.ts

  The smaller you make directories, the better you classify your files, and the more easily you will find them. Here we have 9 levels in the subdirectory tree, which allow the file to be perfectly classified *PoolAiPositionHeuristics.ts*.

- Then you can note that for a simple function, we have a multitude of files. I counted 31 files, only for the AI of a game. The fact of cutting a file into several files of 100 to 500 lines is positive because once again we will easily find the code we are looking for. It is an architecture that functions like a library. In a library, you find what you are looking for. The books are classified by theme, by period, by author...

- Look at the file names. Each file is prefixed with the module name “PoolAi”. It's not obligatory, but it helps you work and know what you're doing when you have lots of files open. The rest of the name of each file is self-explanatory. We didn't choose them randomly. The name of a file should allow you to know what has been put in it. And the file must not be too big, nor too small, average a few hundred lines, and it must deal with a single subject and nothing else, and do it in isolation.

Of course, there are other ways to do this ranking. But what is important is to find a logical way, and to take the time to regularly reclassify a part as soon as it grows. You have to do refactoring, very regularly, and not be lazy.

CLASSIFYING a large code base is THE MOST IMPORTANT THING in a quality assurance plan. The most important thing is to be able to find what you are looking for.

Now let's look at the file *PoolAiPositionHeuristics.ts,* and in particular to the function *evalPositionPocketFactor()*.

```
    /**
     * Eval a position's pocket factor (ppf).
     *     + The pocket factor is a heuristic representing the advantageous placement of the target balls on the pool table.
     *     + It is obtained by moving the white ball virtually to a number of pre-dertimined significant positions,
     *       and couting in how many holes how many target balls can be pocketed directly.
     *     + A formula is applied and this number is brought to 0..1
     *     + It is possible to compute this factor for any of the 2 players.
     *     + It is possible to include or not the black ball in this factor's evaluation.
     *       It should be included if it's the ai's turn to play, so it could win without losing turn to play when it's time to pocket black.
     *       It should not be included if the ai is going to lose the turn to play, so it won't help the opponent to have an easy black ball.
     *     + Cost: 1ms for an average position.
     */

    private static readonly WANTED_WHITE_POSITIONS : Array<[ float, float ]> = [ // % of width, % of height
                                [ 1/2, 1/10 ],
        [ 1/5, 1/5 ],                           [ 4/5, 1/5 ],
                                [ 1/2, 1/2  ],
        [ 1/5, 4/5 ],                           [ 4/5, 4/5 ],
                                [ 1/2, 9/10 ],
    ];
    private static readonly DIRECTIONS : Array<[ int, int ]> = [ // delta sign hor, delta sign ver
        [ -1, -1 ], [ 0, -1 ],  [ +1, -1 ],
        [ -1,  0 ],             [ +1,  0 ],
        [ -1, +1 ], [ 0, +1 ],  [ +1, +1 ],
    ];

    public evalPositionPocketFactor(v : PoolAiVars, args : {
        arrTargetBall : Array<PoolBall>,
        includeBlack : boolean,
    }) : float {
        // init vars: save white position
        const whitePosOrigX : float = v.ballsAnalysis.whiteBallPos.x_;
        const whitePosOrigY : float = v.ballsAnalysis.whiteBallPos.y_;
        // init vars: add black to target balls if required
        const arrTargetBall : Array<PoolBall> = ArrayUtils.arrayCloneNotDeep(args.arrTargetBall);
        if (args.includeBlack && ! arrTargetBall.some((curPoolBall : PoolBall) : boolean => curPoolBall.ballColor == POOL_BALL_COLOR.BLACK)) {
            arrTargetBall.push(v.board.blackBall);
        }
        // init vars: counts
        let totalWhitePositions : int = 0;
        let totalCountPocketableInHoles : int = 0;

        // for each wanted white position
        for (const wantedWhitePosition of PoolAiPositionHeuristics.WANTED_WHITE_POSITIONS) {
            const wantedXWhite : float = v.board.tableBounds.left_ + wantedWhitePosition[0] * v.board.tableBounds.width_;
            const wantedYWhite : float = v.board.tableBounds.top_ + wantedWhitePosition[1] * v.board.tableBounds.height_;

            // find a position for white, at wanted position if possible, otherwise close to it
            // first try the wanted position
            const vPosWhite = new Vect(wantedXWhite, wantedYWhite);
            let isPosWhiteOk : boolean = v.board.isBallPosOk({ vPos: vPosWhite, boundsTolerance: 0, doAutofixBoundsLimits: false });
            // if wanted position is occupied: try to place the white close to it, in each direction
            if (! isPosWhiteOk) {
                for (let distFactor : int = 1; distFactor <= 2; distFactor++) {
                    for (const direction of PoolAiPositionHeuristics.DIRECTIONS) {
                        vPosWhite.x_ = wantedXWhite + direction[0] * distFactor * PoolBoard.BALL_SIZE;
                        vPosWhite.y_ = wantedYWhite + direction[1] * distFactor * PoolBoard.BALL_SIZE;
                        isPosWhiteOk = v.board.isBallPosOk({ vPos: vPosWhite, boundsTolerance: 0, doAutofixBoundsLimits: false });
                        if (isPosWhiteOk) break;
                    }
                    if (isPosWhiteOk) break;
                }
            }

            // if we could find a position for white near the wanted position (if we couldn't, we simply skip this position)
            if (isPosWhiteOk) {
                totalWhitePositions++;
                // place the white on the computed position
                v.ballsAnalysis.whiteBallPos.setInPlace(vPosWhite);
                // for each target ball
                for (const targetPoolBall of arrTargetBall) {
                    // count the number of holes the ball can be directly pocketed into
                    const countPocketableInHoles : int = this._ai.anaHoleDirect.computeBallAnalysisToHoleDirect(v, targetPoolBall, true);
                    totalCountPocketableInHoles += countPocketableInHoles;
                }
            }
        }

        // put back the white ball where it was
        v.ballsAnalysis.whiteBallPos.x_ = whitePosOrigX;
        v.ballsAnalysis.whiteBallPos.y_ = whitePosOrigY;

        // compute factor
        const positionPocketFactor : float =
              totalCountPocketableInHoles
            / totalWhitePositions
            / arrTargetBall.length
            / v.board.holes.length
        ;
        assert2(positionPocketFactor <= 1);
        return positionPocketFactor;
    }

```

We have choosen this function because it is the heart of the snooker game AI, the one that can *to feel* whether one position is more strategically advantageous than another, and its calculation is immensely complex. However, you will notice that even without being a computer scientist, we can guess what she does, and how she does it. This is easily readable code. By writing an entire program like this, you make maintenance work so much easier that you do everything ten times faster.

The code is breezy because who wants to read paragraphs stuck together without spaces to understand how it's broken up? The code is typed because we want to know what we are working on. The code is commented because when we write it, we know what we are doing, and we must write down the information IN THE CODE and not keep it in our head. The code is structured and each paragraph does one thing, only one thing, and does not overflow. Cartesian thinking is always better than clutter and shortcuts.

Quality code is an investment made by a diligent developer...

- He is not someone who thinks ten times faster, because in fact he asks himself additional questions and thus he thinks less quickly...
- He's not someone who writes ten times faster, because in fact he writes more things and thus he writes slower...
- He's not someone who delivers ten times faster, because in fact he refactors and thus delivers slower...

But he is someone who takes more time than others to produce readable code. This code will be easier to find, understand, and modify. This code will be typed and will avoid stupid mistakes. This code will cause fewer bugs and regressions. This code will be malleable, movable, maintainable. This code will make it possible, in the long term, to work ten times faster. If not a hundred times!

Whereas the lazy programmer, who delivers quickly and goes to drink his coffee, will spend eternity correcting the same errors, and re-learning what he could have noted in the typing documentation and comments. The quality developer better invests his time, that of the team, and wins in the long term. And anyone can learn to do that if they are trained correctly and want to progress.

Comparison: Senior developer code « that works »

After reviewing an example of good code, let's compare it with a senior developer's "working" code. From the following example, can you guess what is in the directory and what it is for?

No, I can't do it. So what do you think we should do? Well you have to struggle and debug and spend weeks, months on it, and you will barely understand what's inside because everything is mixed up and poorly named. However, this code works as well as the previous one, when we look at the execution result in the browser. There is good code, and there is bad code, but it's impossible to see if you just compare the results. It's the engine, hidden under the shiny hood, which is the problem and which will sooner or later require repairs.

Now let's zoom in on the file *dataMixer.js* and let's look at what's in it:

```
const _ = require('lodash');
const ultimateMixerFunction = () => {
  const mixData = (a, b) => _.flatten(_.zip(a, b).map(x =>
    x.map(y => _.isArray(y) ? y : _.times(_.random(1, 3), () => y)))
    .map(z => _.shuffle(z.flat()))).reduce((m, n) => {
    let p = _.filter(m, q => q !== n);
    return _.concat(p, n);
  }, []);
  const processArray = x => _.chain(x).groupBy(y => typeof y)
    .mapValues(z => _.flatten(z)).values().flatten().value();
  const dataMixer = (arr1, arr2) => {
    let mixed1 = mixData(arr1, processArray(arr2));
    let mixed2 = mixData(processArray(arr1), arr2);
    return _.union(mixed1, mixed2);
  };
  const randomizeArray = arr => _.shuffle(_.flatten(arr.map(x =>
    _.isArray(x) ? x : [x])));
  const finalizeMix = (d1, d2) => dataMixer(randomizeArray(d1), randomizeArray(d2));
  const mixAndMatch = (x, y) => finalizeMix(processArray(x), processArray(y));
  const ultimateMixer = (input1, input2) => mixAndMatch(_.shuffle(input1), _.shuffle(input
  return { ultimateMixer };
};
module.exports = ultimateMixerFunction();
```

Can you understand anything? I don't, and yet I have 35 years of programming experience. What if I was asked to change this code, would I be able to do it? And how long do you think it would take me?

Consequences of lack of maintainability

It's simple, when a program is not maintainable enough:

- We waste time, therefore money.
- We have a lot of side effects, and therefore regression bugs. As soon as we touch a line on the left, it explodes on the right.
- We are slowly adding features.
- Developers are discouraged. They do too much decoding, and too little design.
- We are having trouble adding developers to the team. As a corollary, it is difficult to replace developers who leave.
- Management is very dependent on developers. The scale of the technical debt forces it to manage a crisis situation.

We will end up throwing out the program and re-writing it, or renovating it and it will cost a fortune. It's inevitable.

Conclusion: What is concretely feasible in your company

In this document, based on my professional experience, I advocated for the quality of the code, with particular emphasis on its findability, typing, comments, symmetry, readability, architecture, dependencies and his tests. And should you, and how could you introduce this level of programming quality into your business?

1. If you want to comfortably manage your development department, and add a certain level of software quality assurance in your company, the determining criterion is: Are you starting from scratch or do you need to improve an existing project? For a project *from scratch*, just follow good practices. To renovate an existing project, you have to cut it up, reread it, comment on it, type it, and do that little by little, module by module, starting with the most urgent.

2. The second criterion is the need for maintenance: If you are a software publisher or end customer, it is very important to produce quality. Since if we have to maintain the code ourselves, it is better to ensure quality from the start. Conversely, if you are a service provider with a contract, and once you deliver the project to a client, you will no longer have to take care of it, you can do things with less quality.

3. The third criterion is quantity. You have to define to what extent you want quality: are you willing to invest time and money in it? Or do we prefer to deliver quickly even if it means having problems later, which can be a winning strategy for example for a start-up? The targeted level of quality will depend on the company culture, and must be adapted tactfully so as not to upset the various stakeholders.

   If you think you will have to maintain, translate or rewrite the code later, it is better to invest in quality from the start. Unless we're in a hurry and just want to release a product to beat the competition: We could make something quick and not very clean. But be careful, the resulting technical debt is often difficult to manage.

Before taking a position on this subject, it will be important to inform decision-makers that an additional investment in work will be necessary, but that this will bring long-term benefits. It will also allow them to take full control of their business, ensuring better collaboration with developers.

And it will be necessary to explain to developers, who might be disturbed to see their code modified, that it is also an opportunity to learn, and an opportunity to be even more proud of their own work. And that by exchanging knowledge, everyone improves.

In conclusion, although software quality assurance requires an initial effort, it will provide lasting benefits for the business and for developers.

Licence Creative commons

Creative Commons Legal Code

CC0 1.0 Universal

CREATIVE COMMONS CORPORATION IS NOT A LAW FIRM AND DOES NOT PROVIDE LEGAL SERVICES. DISTRIBUTION OF THIS DOCUMENT DOES NOT CREATE AN ATTORNEY-CLIENT RELATIONSHIP. CREATIVE COMMONS PROVIDES THIS INFORMATION ON AN "AS-IS" BASIS. CREATIVE COMMONS MAKES NO WARRANTIES REGARDING THE USE OF THIS DOCUMENT OR THE INFORMATION OR WORKS PROVIDED HEREUNDER, AND DISCLAIMS LIABILITY FOR DAMAGES RESULTING FROM THE USE OF THIS DOCUMENT OR THE INFORMATION OR WORKS PROVIDED HEREUNDER.

Statement of Purpose

The laws of most jurisdictions throughout the world automatically confer exclusive Copyright and Related Rights (defined below) upon the creator and subsequent owner(s) (each and all, an "owner") of an original work of authorship and/or a database (each, a "Work").

Certain owners wish to permanently relinquish those rights to a Work for the purpose of contributing to a commons of creative, cultural and scientific works ("Commons") that the public can reliably and without fear of later claims of infringement build upon, modify, incorporate in other works, reuse and redistribute as freely as possible in any form whatsoever and for any purposes, including without limitation commercial purposes. These owners may contribute to the Commons to promote the ideal of a free culture and the further production of creative, cultural and scientific works, or to gain reputation or greater distribution for their Work in part through the use and efforts of others.

For these and/or other purposes and motivations, and without any expectation of additional consideration or compensation, the person associating CC0 with a Work (the "Affirmer"), to the extent that he or she is an owner of Copyright and Related Rights in the Work, voluntarily elects to apply CC0 to the Work and publicly distribute the Work under its terms, with knowledge of his or her Copyright and Related Rights in the Work and the meaning and intended legal effect of CC0 on those rights.

1. Copyright and Related Rights. A Work made available under CC0 may be protected by copyright and related or neighboring rights ("Copyright and Related Rights"). Copyright and Related Rights include, but are not limited to, the following:

i. the right to reproduce, adapt, distribute, perform, display, communicate, and translate a Work; ii. moral rights retained by the original author(s) and/or performer(s); iii. publicity and privacy rights pertaining to a person's image or likeness depicted in a Work; iv. rights protecting against unfair competition in regards to a Work, subject to the limitations in paragraph 4(a), below; v. rights protecting the extraction, dissemination, use and reuse of data in a Work; vi. database rights (such as those arising under Directive 96/9/EC of the European Parliament and of the Council of 11 March 1996 on the legal protection of databases, and under any national implementation thereof, including any amended or successor version of such directive); and vii. other similar, equivalent or corresponding rights throughout the world based on applicable law or treaty, and any national implementations thereof.

2. Waiver. To the greatest extent permitted by, but not in contravention of, applicable law, Affirmer hereby overtly, fully, permanently, irrevocably and unconditionally waives, abandons, and surrenders all of Affirmer's Copyright and Related Rights and associated claims and causes of action, whether now known or unknown (including existing as well as future claims and causes of action), in the Work (i) in all territories worldwide, (ii) for the maximum duration provided by applicable law or treaty (including future time extensions), (iii) in any current or future medium and for any number of copies, and (iv) for any purpose whatsoever, including without limitation commercial, advertising or promotional purposes (the "Waiver"). Affirmer makes the Waiver for the benefit of each member of the public at large and to the detriment of Affirmer's heirs and successors, fully intending that such Waiver shall not be subject to revocation, rescission, cancellation, termination, or any other legal or equitable action to disrupt the quiet enjoyment of the Work by the public as contemplated by Affirmer's express Statement of Purpose.

3. Public License Fallback. Should any part of the Waiver for any reason be judged legally invalid or ineffective under applicable law, then the Waiver shall be preserved to the maximum extent permitted taking into account Affirmer's express Statement of Purpose. In addition, to the extent the Waiver is so judged Affirmer hereby grants to each affected person a royalty-free, non transferable, non sublicensable, non exclusive, irrevocable and unconditional license to exercise Affirmer's Copyright and Related Rights in the Work (i) in all territories worldwide, (ii) for the maximum duration provided by applicable law or treaty (including future time extensions), (iii) in any current or future medium and for any number of copies, and (iv) for any purpose whatsoever, including without limitation commercial, advertising or promotional purposes (the "License"). The License shall be deemed effective as of the date CC0 was applied by Affirmer to the Work. Should any part of the License for any reason be judged legally invalid or ineffective under applicable law, such partial invalidity or ineffectiveness shall not invalidate the remainder of the License, and in such case Affirmer hereby affirms that he or she will not (i) exercise any of his or her remaining Copyright and Related Rights in the Work or (ii) assert any associated claims and causes of action with respect to the Work, in either case contrary to Affirmer's express Statement of Purpose.

4. Limitations and Disclaimers.

a. No trademark or patent rights held by Affirmer are waived, abandoned, surrendered, licensed or otherwise affected by this document. b. Affirmer offers the Work as-is and makes no representations or warranties of any kind concerning the Work, express, implied, statutory or otherwise, including without limitation warranties of title, merchantability, fitness for a particular purpose, non infringement, or the absence of latent or other defects, accuracy, or the present or absence of errors, whether or not discoverable, all to the greatest extent permissible under applicable law. c. Affirmer disclaims responsibility for clearing rights of other persons that may apply to the Work or any use thereof, including without limitation any person's Copyright and Related Rights in the Work. Further, Affirmer disclaims responsibility for obtaining any necessary consents, permissions or other rights required for any use of the Work. d. Affirmer understands and acknowledges that Creative Commons is not a party to this document and has no duty or obligation with respect to this CC0 or use of the Work.Creative Commons Legal Code