MAY 15, 2023 / #BEST PRACTICES

How to Write Clean Code – Tips and Best Practices (Full Handbook)

German Cocca

Hi everyone! In this handbook we're going to talk about writing "clean" code. It's a topic that used to confuse me a bit when I was starting out as a programmer, and I find that it has many nuances and possible interpretations.

So in this article we'll talk about what the term "clean code" means, why it's important, how can we assess whether a codebase is clean or not. You'll also learn some best practices and conventions you can follow to make your code cleaner.

Let's go!

Table of Contents

What Does it Mean to Write "Clean Code" and Why Should I Care?

Clean code is a term used to describe computer code that is easy to read, understand, and maintain. Clean code is written in a way that makes it simple, concise, and expressive. It follows a set of conventions, standards, and practices that make it easy to read and follow.

Clean code is free from complexity, redundancy, and other code smells and anti-patterns that can make it difficult to maintain, debug, and modify.

I can't overstate the importance of clean code. When code is easy to read and understand, it makes it easier for developers to work on the codebase. This can lead to increased productivity and reduced errors.

Also, when code is easy to maintain, it ensures that the codebase can be improved and updated over time. This is especially important for long-term projects where code must be maintained and updated for years to come.

How Can I Assess Whether a Codebase is

Clean or Not?

You can assess clean code in a variety of ways. Good documentation, consistent formatting, and a well-organized codebase are all indicators of clean code.

Code reviews can also help to identify potential issues and ensure that code follows best practices and conventions.

Testing is also an important aspect of clean code. It helps to ensure that code is functioning as expected and can catch errors early.

There are several tools, practices, and conventions you can implement to ensure a clean codebase. By implementing these tools and practices, developers can create a codebase that is easy to read, understand, and maintain.

It's also important to remember that there's an inevitable amount of subjectivity related to this topic, and there are a number of different opinions and tips out there. What might look clean and awesome for one person or one project might not be so for another person or another project.

But still there are a few general conventions we can follow to achieve cleaner code, so let's jump into that now.

Tips and Conventions to Write Cleaner Code

Effectiveness, Efficiency and Simplicity

Whenever I need to think about how to implement a new feature into an already existing codebase, or how to tackle the solution of a specific problem, I always prioritize this three simple things.

Effectiveness

First, our code should be **effective**, meaning it should solve the problem it's supposed to solve. Of course this is the most basic expectation we could have for our code, but if our implementation doesn't actually work, it's worthless to think about any other thing.

Efficiency

Second, once we know our code solves the problem, we should check if it does so **efficiently**. Does the program run using a reasonable amount of resources in terms of time and space? Can it run faster and with less space?

Algorithmic complexity is something you should be aware of in order to evaluate this. If you're not familiar with it, you can check this article I wrote.

To expand upon efficiency, here are two examples of a function that calculates the sum of all numbers in an array.

```
// Inefficient Example
function sumArrayInefficient(array) {
  let sum = 0;
  for (let i = 0; i < array.length; i++) {
    sum += array[i];
  }
  return sum;
}
```

This implementation of the `sumArrayInefficient` function iterates over the array using a `for` loop and adds each element to the `sum` variable. This is a valid solution, but it is not very efficient because it requires iterating over the entire array, regardless of its length.

```
// Efficient example
function sumArrayEfficient(array) {
  return array.reduce((a, b) => a + b, 0);
}
```

This implementation of the `sumArrayEfficient` function uses the `reduce` method to sum the elements of the array. The `reduce` method applies a function to each element of the array and accumulates the result. In this case, the function simply adds each element to the accumulator, which starts at 0.

This is a more efficient solution because it only requires a single iteration over the array and performs the summing operation on each element as it goes.

Simplicity

And last comes **simplicity**. This is the toughest one to evaluate because its subjective, it depends on the person who reads the code. But some guidelines we can follow are:

1. Can you easily understand what the program does at each line?
2. Do functions and variables have names that clearly represent their responsibilities?
3. Is the code indented correctly and spaced with the same format all along the codebase?

4. Is there any documentation available for the code? Are comments used to explain complex parts of the program?
5. How quick can you identify in which part of the codebase are certain features of the program? Can you delete/add new features without the need of modifying many other parts of the code?
6. Does the code follow a modular approach, with different features separated in components?
7. Is code reused when possible?
8. Are the same architecture, design, and implementation decisions followed equally all along the codebase?

By following and prioritizing these three concepts of effectiveness, efficiency, and simplicity, we can always have a guideline to follow when thinking about how to implement a solution. Now let's expand upon some of the guidelines that can help us simplify our code.

Format and Syntax

Using consistent formatting and syntax throughout a codebase is an important aspect of writing clean code. This is because consistent formatting and syntax make the code more readable and easier to understand.

When code is consistent, developers can easily identify patterns and understand how the code works, which makes it easier to debug, maintain, and update the codebase over time. Consistency also helps to reduce errors, as it ensures that all developers are following the same standards and conventions.

Some of the things we should think about regarding format and syntax are:

- **Indentation and spacing**

```
// bad indentation and spacing
const myFunc=(number1,number2)=>{
const result=number1+number2;
return result;
}

// good indentation and spacing
const myFunc = (number1, number2) => {
    const result = number1 + number2
    return result
}
```

Here we have an example of a same function, one done with no indentation and spacing, and the other properly spaced and indented. We can see that the second one is clearly easier to read.

- **Consistent syntax**

```
// Arrow function, no colons, no return
const multiplyByTwo = number => number * 2

// Function, colons, return
function multiplyByThree(number) {
    return number * 3;
}
```

Again, here we have very similar functions implemented with different syntax. The first one is an arrow function, with no colons and no return, while the other is a common function that uses colons and a return.

Both work and are just fine, but we should aim to always use the same syntax for similar operations, as it becomes more even and readable along the codebase.

Linterns and code formatters are great tools we can use in our projects to automatize the syntax and formatting conventions in our codebase. If you're not familiar with this tools, check out this other article of mine.

- **Consistent case conventions**

```
// camelCase
const myName = 'John'
// PascalCase
const MyName = 'John'
// snake_case
const my_name = 'John'
```

Same goes for the case convention we choose to follow. All of these work, but we should aim to consistently use the same one all through our project.

Naming

Naming variables and functions clearly and descriptively is an important aspect of writing clean code. It helps to improve the readability and maintainability of the codebase. When names are well-chosen, other developers can quickly understand what the variable or function is doing, and how it is related to the rest of the code.

Here are two examples in JavaScript that demonstrate the importance of clear and descriptive naming:

```
// Example 1: Poor Naming
function ab(a, b) {
  let x = 10;
  let y = a + b + x;
  console.log(y);
}

ab(5, 3);
```

In this example, we have a function that takes two parameters, adds them to a hardcoded value of 10, and logs the result to the console. The function name and variable names are poorly chosen and don't give any indication of what the function does or what the variables represent.

```
// Example 1: Good Naming
function calculateTotalWithTax(basePrice, taxRate) {
  const BASE_TAX = 10;
  const totalWithTax = basePrice + (basePrice * (taxRate / 100)) + BASE
  console.log(totalWithTax);
}

calculateTotalWithTax(50, 20);
```

In this example, we have a function that calculates the total price of a product including tax. The function name and variable names are well-chosen and give a clear indication of what the function does and what the variables represent.

This makes the code easier to read and understand, especially for other developers who may be working with the codebase in the future.

Conciseness vs Clarity

When it comes to writing clean code, it's important to strike a balance between conciseness and clarity. While it's important to keep code concise to improve its readability and maintainability, it's equally important to ensure that the code is clear and easy to understand. Writing overly concise code can lead to confusion and errors, and can make the code difficult to work with for other developers.

Here are two examples that demonstrate the importance of conciseness and clarity:

```javascript
// Example 1: Concise function
const countVowels = s => (s.match(/[aeiou]/gi) || []).length;
console.log(countVowels("hello world"));
```

This example uses a concise arrow function and regex to count the number of vowels in a given string. While the code is very short and easy to write, it may not be immediately clear to other developers how the regex pattern works, especially if they are not familiar with regex syntax.

```javascript
// Example 2: More verbose and clearer function
function countVowels(s) {
  const vowelRegex = /[aeiou]/gi;
  const matches = s.match(vowelRegex) || [];
  return matches.length;
}

console.log(countVowels("hello world"));
```

This example uses a traditional function and regex to count the number of vowels in a given string, but does so in a way that is clear and easy to understand. The function name and variable names are descriptive, and the regex pattern is stored in a variable with a clear name. This makes it easy to see what the function is doing and how it works.

It's important to strike a balance between conciseness and clarity when writing code. While concise code can improve readability and maintainability, it's important to ensure that the code is still clear and easy to understand for other developers who may be working with the codebase in the future.

By using descriptive function and variable names, and making use of clear and readable code formatting and comments, it's possible to write clean and concise code that is easy to understand and work with.

Reusability

Code reusability is a fundamental concept in software engineering that refers to the ability of code to be used multiple times without modification.

The importance of code reusability lies in the fact that it can greatly improve the efficiency and productivity of software development by reducing the amount of code that needs to be written and tested.

By reusing existing code, developers can save time and effort, improve code quality and consistency, and minimize the risk of introducing bugs and errors. Reusable code also allows for more modular and scalable software architectures, making it easier to maintain and update codebases over time.

```
// Example 1: No re-usability
function calculateCircleArea(radius) {
  const PI = 3.14;
  return PI * radius * radius;
}

function calculateRectangleArea(length, width) {
  return length * width;
}

function calculateTriangleArea(base, height) {
  return (base * height) / 2;
}

const circleArea = calculateCircleArea(5);
const rectangleArea = calculateRectangleArea(4, 6);
const triangleArea = calculateTriangleArea(3, 7);

console.log(circleArea, rectangleArea, triangleArea);
```

This example defines three functions that calculate the area of a circle, rectangle, and triangle, respectively. Each function performs a specific task, but none of them are reusable for other similar tasks.

Additionally, the use of a hard-coded PI value can lead to errors if the value needs to be changed in the future. The code is inefficient since it repeats the same logic multiple times.

```
// Example 2: Implementing re-usability
function calculateArea(shape, ...args) {
  if (shape === 'circle') {
    const [radius] = args;
    const PI = 3.14;
    return PI * radius * radius;
  } else if (shape === 'rectangle') {
    const [length, width] = args;
```

```
      return length * width;
    } else if (shape === 'triangle') {
      const [base, height] = args;
      return (base * height) / 2;
    } else {
      throw new Error(`Shape "${shape}" not supported.`);
    }
  }

  const circleArea = calculateArea('circle', 5);
  const rectangleArea = calculateArea('rectangle', 4, 6);
  const triangleArea = calculateArea('triangle', 3, 7);

  console.log(circleArea, rectangleArea, triangleArea);
```

This example defines a single function `calculateArea` that takes a `shape` argument and a variable number of arguments. Based on the `shape` argument, the function performs the appropriate calculation and returns the result.

This approach is much more efficient since it eliminates the need to repeat code for similar tasks. It is also more flexible and extensible, as additional shapes can easily be added in the future.

Clear Flow of Execution

Having a clear flow of execution is essential for writing clean code because it makes the code easier to read, understand, and maintain. Code that follows a clear and logical structure is less prone to errors, easier to modify and extend, and more efficient in terms of time and resources.

On the other hand, spaghetti code is a term used to describe code that is convoluted and difficult to follow, often characterized by long, tangled, and unorganized code blocks. Spaghetti code can be a result of poor design decisions, excessive coupling, or lack of proper documentation and commenting.

Here are two examples of JavaScript code that perform the same task, one with a clear flow of execution, and the other with spaghetti code:

```
// Example 1: Clear flow of execution
function calculateDiscount(price, discountPercentage) {
  const discountAmount = price * (discountPercentage / 100);
  const discountedPrice = price - discountAmount;
  return discountedPrice;
}

const originalPrice = 100;
const discountPercentage = 20;
const finalPrice = calculateDiscount(originalPrice, discountPercentage

console.log(finalPrice);

// Example 2: Spaghetti code
const originalPrice = 100;
const discountPercentage = 20;

let discountedPrice;
let discountAmount;
if (originalPrice && discountPercentage) {
  discountAmount = originalPrice * (discountPercentage / 100);
  discountedPrice = originalPrice - discountAmount;
}

if (discountedPrice) {
  console.log(discountedPrice);
}
```

As we can see, example 1 follows a clear and logical structure, with a function that takes in the necessary parameters and returns the calculated result. On the other hand, example 2 is much more convoluted, with variables declared outside of any function and multiple if statements used to check if the code block has executed successfully.

Single Responsibility Principle

The Single Responsibility Principle (SRP) is a principle in software development that states that each class or module should have only one reason to change, or in other words, each entity in our codebase should have a single responsibility.

This principle helps to create code that is easy to understand, maintain, and extend.

By applying SRP, we can create code that is easier to test, reuse, and refactor, since each module only handles a single responsibility. This makes it less likely to have side effects or dependencies that can make the code harder to work with.

```
// Example 1: Withouth SRP
function processOrder(order) {
  // validate order
  if (order.items.length === 0) {
    console.log("Error: Order has no items");
    return;
  }

  // calculate total
  let total = 0;
  order.items.forEach(item => {
    total += item.price * item.quantity;
  });

  // apply discounts
  if (order.customer === "vip") {
    total *= 0.9;
  }

  // save order
  const db = new Database();
  db.connect();
```

```
    db.saveOrder(order, total);
  }
```

In this example, the `processOrder` function handles several responsibilities: it validates the order, calculates the total, applies discounts, and saves the order to a database. This makes the function long and hard to understand, and any changes to one responsibility may affect the others, making it harder to maintain.

```
// Example 2: With SRP
class OrderProcessor {
  constructor(order) {
    this.order = order;
  }

  validate() {
    if (this.order.items.length === 0) {
      console.log("Error: Order has no items");
      return false;
    }
    return true;
  }

  calculateTotal() {
    let total = 0;
    this.order.items.forEach(item => {
      total += item.price * item.quantity;
    });
    return total;
  }

  applyDiscounts(total) {
    if (this.order.customer === "vip") {
      total *= 0.9;
    }
    return total;
  }
}

class OrderSaver {
  constructor(order, total) {
    this.order = order;
    this.total = total;
  }

  save() {
    const db = new Database();
    db.connect();
    db.saveOrder(this.order, this.total);
  }
}

const order = new Order();
const processor = new OrderProcessor(order);

if (processor.validate()) {
  const total = processor.calculateTotal();
  const totalWithDiscounts = processor.applyDiscounts(total);
  const saver = new OrderSaver(order, totalWithDiscounts);
  saver.save();
}
```

In this example, the `processOrder` function has been split into two classes that follow the SRP: `OrderProcessor` and `OrderSaver`.

The `OrderProcessor` class handles the responsibilities of validating the order, calculating the total, and applying discounts, while the `OrderSaver` class handles the responsibility of saving the order to the database.

This makes the code easier to understand, test, and maintain, since each class has a clear responsibility and can be modified or replaced without affecting the others.

Having a "Single Source of Truth"

Having a "single source of truth" means that there is only one place where a particular piece of data or configuration is stored in the codebase, and any other references to it in the code refer back to that one source. This is important because it ensures that the data is consistent and avoids duplication and inconsistency.

Here's an example to illustrate the concept. Let's say we have an application that needs to display the current weather conditions in a city. We could implement this feature in two different ways:

```
// Option 1: No "single source of truth"

// file 1: weatherAPI.js
const apiKey = '12345abcde';

function getCurrentWeather(city) {
  return fetch(`https://api.weather.com/conditions/v1/${city}?apiKey=$
    .then(response => response.json());
}

// file 2: weatherComponent.js
const apiKey = '12345abcde';

function displayCurrentWeather(city) {
  getCurrentWeather(city)
    .then(weatherData => {
      // display weatherData on the UI
    });
}
```

In this option, the API key is duplicated in two different files, making it harder to maintain and update. If we ever need to change the API key, we have to remember to update it in both places.

```
// Option 2: "Single source of truth"

// file 1: weatherAPI.js
const apiKey = '12345abcde';

function getCurrentWeather(city) {
  return fetch(`https://api.weather.com/conditions/v1/${city}?apiKey=$
    .then(response => response.json());
}

export { getCurrentWeather, apiKey };


// file 2: weatherComponent.js
import { getCurrentWeather } from './weatherAPI';

function displayCurrentWeather(city) {
  getCurrentWeather(city)
    .then(weatherData => {
      // display weatherData on the UI
    });
}
```

In this option, the API key is stored in one place (in the `weatherAPI.js` file) and exported for other modules to use. This ensures that there is only one source of truth for the API key and avoids duplication and inconsistency.

If we ever need to update the API key, we can do it in one place and all other modules that use it will automatically get the updated value.

Only Expose and Consume Data You Need

One important principle of writing clean code is to only expose and consume the information that is necessary for a particular task. This helps to reduce complexity, increase efficiency, and avoid errors that can arise from using unnecessary data.

When data that is not needed is exposed or consumed, it can lead to performance issues and make the code more difficult to understand

and maintain.

Suppose you have an object with multiple properties, but you only need to use a few of them. One way to do this would be to reference the object and the specific properties every time you need them. But this can become verbose and error-prone, especially if the object is deeply nested. A cleaner and more efficient solution would be to use object destructuring to only expose and consume the information you need.

```
// Original object
const user = {
  id: 1,
  name: 'Alice',
  email: 'alice@example.com',
  age: 25,
  address: {
    street: '123 Main St',
    city: 'Anytown',
    state: 'CA',
    zip: '12345'
  }
};

// Only expose and consume the name and email properties
const { name, email } = user;

console.log(name); // 'Alice'
console.log(email); // 'alice@example.com'
```

Modularization

Modularization is an essential concept in writing clean code. It refers to the practice of breaking down large, complex code into smaller, more manageable modules or functions. This makes the code easier to understand, test, and maintain.

Using modularization provides several benefits such as:

1. Re-usability: Modules can be reused in different parts of the application or in other applications, saving time and effort in development.
2. Encapsulation: Modules allow you to hide the internal details of a function or object, exposing only the essential interface to the outside world. This helps to reduce coupling between different parts of the code and improve overall code quality.
3. Scalability: By breaking down large code into smaller, modular pieces, you can easily add or remove functionality without affecting the entire codebase.

Here is an example in JavaScript of a piece of code that performs a simple task, one not using modularization and the other implementing modularization.

```javascript
// Without modularization
function calculatePrice(quantity, price, tax) {
  let subtotal = quantity * price;
  let total = subtotal + (subtotal * tax);
  return total;
}

// Without modularization
let quantity = parseInt(prompt("Enter quantity: "));
let price = parseFloat(prompt("Enter price: "));
let tax = parseFloat(prompt("Enter tax rate: "));

let total = calculatePrice(quantity, price, tax);
console.log("Total: $" + total.toFixed(2));
```

In the above example, the `calculatePrice` function is used to calculate the total price of an item given its quantity, price, and tax rate. However, this function is not modularized and is tightly coupled with the user input and output logic. This can make it difficult to test and maintain.

Now, let's see an example of the same code using modularization:

```
// With modularization
function calculateSubtotal(quantity, price) {
  return quantity * price;
}

function calculateTotal(subtotal, tax) {
  return subtotal + (subtotal * tax);
}

// With modularization
let quantity = parseInt(prompt("Enter quantity: "));
let price = parseFloat(prompt("Enter price: "));
let tax = parseFloat(prompt("Enter tax rate: "));

let subtotal = calculateSubtotal(quantity, price);
let total = calculateTotal(subtotal, tax);
console.log("Total: $" + total.toFixed(2));
```

In the above example, the `calculatePrice` function has been broken down into two smaller functions: `calculateSubtotal` and `calculateTotal`. These functions are now modularized and are responsible for calculating the subtotal and total, respectively. This makes the code easier to understand, test, and maintain, and also makes it more reusable in other parts of the application.

Modularization can also refer to the practice of dividing single files of code into many smaller files that are later on compiled back on to a single (or fewer files). This practice has the same benefits we just talked about.

If you'd like to know how to implement this in JavaScript using modules, check out this other article of mine.

Folder Structures

Choosing a good folder structure is an essential part of writing clean code. A well-organized project structure helps developers find and modify code easily, reduces code complexity, and improves project scalability and maintainability.

On the other hand, a poor folder structure can make it challenging to understand the project's architecture, navigate the codebase, and lead to confusion and errors.

Here are examples of a good and a bad folder structure using a React project as an example:

```
// Bad folder structure
my-app/
├── App.js
├── index.js
├── components/
│   ├── Button.js
│   ├── Card.js
│   └── Navbar.js
├── containers/
│   ├── Home.js
│   ├── Login.js
│   └── Profile.js
├── pages/
│   ├── Home.js
│   ├── Login.js
│   └── Profile.js
└── utilities/
    ├── api.js
    └── helpers.js
```

In this example, the project structure is organized around file types, such as components, containers, and pages.

But this approach can lead to confusion and duplication, as it's not clear which files belong where. For example, the `Home` component is present in both the `containers` and `pages` folders. It can also make it challenging to find and modify code, as developers may need to navigate multiple folders to find the code they need.

```
// Good folder structure
my-app/
├── src/
│   ├── components/
│   │   ├── Button/
│   │   │   ├── Button.js
│   │   │   ├── Button.module.css
│   │   │   └── index.js
│   │   ├── Card/
│   │   │   ├── Card.js
│   │   │   ├── Card.module.css
│   │   │   └── index.js
│   │   └── Navbar/
│   │       ├── Navbar.js
│   │       ├── Navbar.module.css
│   │       └── index.js
│   ├── pages/
│   │   ├── Home/
│   │   │   ├── Home.js
│   │   │   ├── Home.module.css
│   │   │   └── index.js
│   │   ├── Login/
│   │   │   ├── Login.js
│   │   │   ├── Login.module.css
│   │   │   └── index.js
│   │   └── Profile/
│   │       ├── Profile.js
│   │       ├── Profile.module.css
│   │       └── index.js
│   ├── utils/
│   │   ├── api.js
│   │   └── helpers.js
│   ├── App.js
│   └── index.js
└── public/
    ├── index.html
    └── favicon.ico
```

In this example, the project structure is organized around features, such as components, pages, and utils. Each feature has its own folder, which contains all the files related to that feature.

This approach makes it easy to find and modify code, as all the files related to a feature are located in the same folder. It also reduces code duplication and complexity, as features are separated, and their related files are organized together.

Overall, a good folder structure should be organized around features, not file types, and should make it easy to find and modify code. A clear and logical structure can make a project easier to maintain, understand and scale, while a confusing and inconsistent structure can lead to errors and confusion.

If you're interested in learning more about this, in this article I wrote about software architecture I expanded upon the topic of folder structures and well-known patterns you can follow.

Documentation

Documentation is an essential part of writing clean code. Proper documentation not only helps the developer who wrote the code understand it better in the future but also makes it easier for other developers to read and understand the codebase. When code is well-documented, it can save time and effort in debugging and maintaining the code.

Documenting is specially important in cases in which simple and easy to understand solutions can't be implemented, cases when the business logic is quite complex, and cases in which people who are not familiar with the codebase have to interact with it.

One way to document code is by using comments. Comments can provide context and explain what the code is doing. But it's important to use comments wisely, only commenting where necessary and avoiding redundant or unnecessary ones.

Another way to document code is by using inline documentation. Inline documentation is embedded in the code itself and can be used to explain what a specific function or piece of code does. Inline documentation is often used in combination with tools like JSDoc, which provides a standard for documenting code in JavaScript.

Tools like Typescript can also provide automatic documentation for our codebase, which is hugely helpful.

If you'd like to know more about Typescript, I wrote a beginner friendly guide a while ago.

And Lastly, tools like Swagger and Postman can be used to document APIs, providing a way to easily understand how to interact with them

If you're interested in knowing how to fully implement, test, consume and document APIs, I recently wrote two guides for REST APIs and GraphQL APIs.

Wrapping Up

Well everyone, as always, I hope you enjoyed the article and learned something new.

If you want, you can also follow me on LinkedIn or Twitter. See you in the next one!

German Cocca
I'm a full stack developer (typescript | react | react native | node | express) and computer science student. In this blog I write about the things I learn along my path to becoming the best developer I can be.