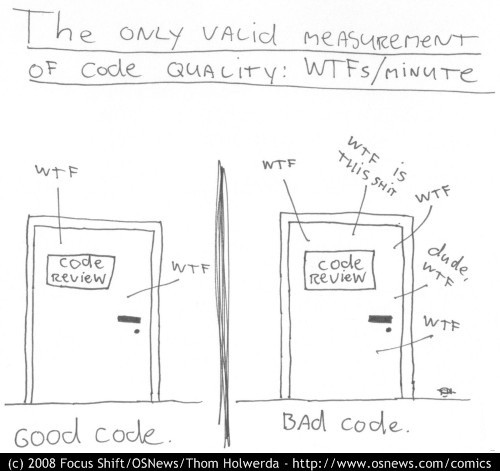
**Clean Code and Software Design Principles: Writing Clean Code**

**Introduction**

Code is clean when it is easily readable by developers other than the original author as well.

Writing clean code might seem like an overhead at the beginning. But it allows us to maintain and develop a piece of software fast in the long term. I've created this series of articles by building on the ideas discussed in the book 'Clean Code'. Clean Code is the bible of good coding practices written by Robert C. Martin (Uncle Bob).



**Why should you write clean code?**

We as developers write code for the computer to understand. At the same time, our code is not going to remain the same as we update the business logic or add new features. Our code might get updated by other developers as well. Hence, we must write code in such a way that other developers can also read and understand it.

**How to write clean code?**

We can write clean code by following a set of guidelines known as Software Design Principles. Software Design Principles is a set of guidelines proven to work over the years.

A few of the broad guidelines to write clean code are:

* **Give meaningful names to variables, functions, classes, and other entities in the code.**
* **Create functions that are small and do a single thing.**
* **Encapsulate related data and functions into small independent classes.**
* **Structure the code for better readability. Keep related code together and keep the lines smaller.**
* **Enhance readability with proper comments.**
* **Write readable, fast, independent, and repeatable tests.**

I've listed down some of the most popular and useful design principles. You should try to enforce them in your code with every iteration.

* **Avoid code repetitions (DRY).**
* **Keep the code simple (KISS).**
* **Hide implementation details (Abstraction).**
* **Keep the code extensible (Extensibility).**
* **Keep the code modular with minimal overlap (Separation of Concerns)**
* **Each module should do only one thing (Curly's law).**
* **Avoid unnecessary functionalities (YAGNI) and preoptimization (the root of all evil).**
* **Stay consistent and avoid surprises in the code (Principle of least astonishment).**
* **Make the code better than how you found it (Boy-Scout Rule).**

There are many more guidelines/principles. We will be discussing the most popular principles in the upcoming sections/articles.

A set of principles known as SOLID Design Principles have been derived from various design principles. **The SOLID Design Principles are:**

* **The Single-responsibility principle**: "There should never be more than one reason for a class to change." In other words, every class should have only one responsibility.
* **The Open–Closed principle**: "Software entities ... should be open for extension, but closed for modification."
* **The Liskov substitution principle**: "Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it".
* **The Interface segregation principle**: "Many client-specific interfaces are better than one general-purpose interface."
* **The Dependency inversion principle**: "Depend upon abstractions, [not] concretions."

We will learn more about the SOLID principles in further articles.

References: [Clean Code: A Handbook of Agile Software Craftsmanship by Robert C. Martin (Uncle Bob)](https://www.goodreads.com/book/show/3735293-clean-code)

# Clean Code and Software Design Principles: Writing Clean Code

## Meaningful Names (variables, functions, and classes)

One of the most difficult things while coding is naming things (variables, functions, and classes).

Most people go ahead with single or double letter variable names like A, v, d, mp, etc when they start coding. Most people use generic variable names like flag, value, map, arr, etc. These variable names might be easy to write but it makes the code difficult to read and makes debugging more time-consuming.

Follow these rules to create meaningful variables, functions, and classes:

* Use Intention-Revealing Names
* Name Functions as Verbs
* Name Classes as Nouns
* Use Meaningful Distinction
* Use Pronounceable Names
* Use Searchable Names
* Avoid Encodings



### Use Intention-Revealing Names

The name of the variable, function, class, etc should be sufficient enough to understand its purpose. One should not have to read the whole code to figure out what a function does or what a class represents or to understand why a variable exists.

The name should ideally not require a comment.

#### Bad Examples

##### variable

//This is bad

int d;

String[] arr;

boolean flag;

##### function

//This is bad

int getAnswer(int a, int b) {

}

#### Good Examples

##### variable

//This is good

int courseDurationInDays;

String[] chapterNames;

boolean isCellVisited;

##### function

//This is good

int getSum(int firstNum, int secondNum) {

}

Writing descriptive variable names may look like it would take more time. Once you start writing descriptive names, it would become pretty intuitive and would result in saving more time in terms of collaboration, maintenance, and readability.

### Name Functions as Verbs

Function names should be verbs or verb phrases that explain what the function does. Getters (Accessors) and Setters (Mutators) should start with get/set.

Function names should also be descriptive. A long, descriptive name is better than using a comment to describe it. A function name should be descriptive enough to understand the intent of that function.

Ward’s principle: “You know you are working on clean code when each function turns out to be pretty much what you expected.”

Be consistent in naming functions and use the same convention.

### Name Classes as Nouns

Classes should have descriptive names such that it should be easy to understand their intent. Classes should have nouns or noun phrases as names. A class name should not be a verb.

### Use Meaningful Distinction

When two variables/functions/classes exist with similar names, make sure that there is a meaningful distinction between their names.

#### Number-series

Number-series naming is a pretty bad way to name variables as it is difficult to distinguish between variables.

##### Example

int[] arr1;

int[] arr2;

#### Noise words

Noise words like Data, Value, Info, Variable, Table, String, Object, etc which are used as a suffix do not offer any meaningful distinction. Noise words are redundant and should be avoided.

##### Examples

String status;

String statusValue;

class Product {

}

class ProductInfo {

}

getDistinctValue(int[] arr) {

}

getDistinctValues(int[] arr) {

}

### Use Pronounceable Names

Using pronounceable names makes the code easy to read and discuss about. Doing so allows discussing/explaining code in plain English.

##### Bad Examples

Date modDateYYMMDD;

##### Good Examples

Date modificationTimestamp;

### Use Searchable Names

In big codebases, you would have to search for variable/function/class names to find it. Small names or constant values might make it difficult to search. Proper names that make it easy to search make the code cleaner and easier to maintain.

#### Avoid magic numbers

Create named constants instead of using numbers or other constant values where it is supposed to denote something.

##### Bad Example

ParkingLot() {

int[] parkingSpots[100];

}

void printParkingSpots() {

for (int i = 0; i < 100; i++) {

System.out.println(parkingSpots[i]);

}

}

##### Good Example

final int NUMBER\_OF\_PARKING\_SPOTS = 100;

ParkingLot() {

int[] parkingSpots[NUMBER\_OF\_PARKING\_SPOTS];

}

void printParkingSpots() {

for (int i = 0; i < NUMBER\_OF\_PARKING\_SPOTS; i++) {

System.out.println(parkingSpots[i]);

}

}

#### Avoid short names

Smaller names should only be used in variables inside short functions (for temporary use) where it has no meaning/use outside the said function. However, it should be noted that the variable names can be small if the scope of the variable is very small given that it is sufficient to understand the intention.

### Avoid Encodings

Avoid using any unnecessary prefixes or suffixes. A variable/function/class name should not be unnecessarily prefixed/suffixed with type information or any other redundant information.

#### Bad Examples

//String suffix ties the variable to the data type that makes it difficult to be changed later.

String locationString;

//The prefix I should be avoided for Interfaces

interface IEmployee {

}

References: [Clean Code: A Handbook of Agile Software Craftsmanship by Robert C. Martin (Uncle Bob)](https://www.goodreads.com/book/show/3735293-clean-code)

# Clean Code and Software Design Principles: Writing Clean Code

## Designing Good Functions and Classes

### Designing Good Functions

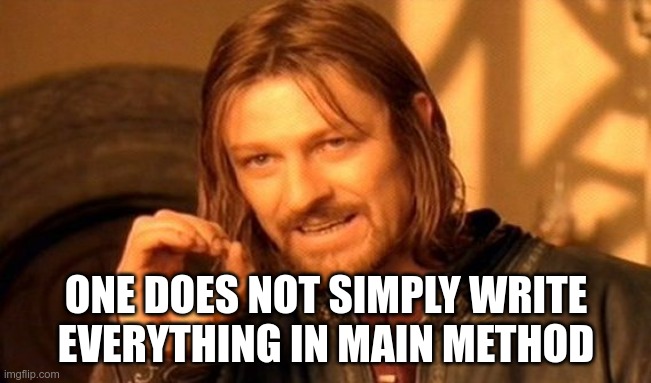
Functions are one of the most important parts of writing code. They make the code reusable and easy to read and maintain. Functions make the code organized.

A good function should have the following properties:

* Should be small
* Should do just one thing
* Should have fewer arguments
* Should not have side effects

One should be able to look at the function name and understand what that function does. If they want to know more, they should just be able to skim through the function to get detail which is at a lower level. For more lower-level details, they should look at the implementation of the functions called inside it.

A good function allows understanding it without going into lower-level details unless required.



#### Should be small

##### How small?

Functions should be very small. It should be hardly 20 lines long.

How to make functions smaller?

Anything inside that function that can be made into a separate function should be extracted. The extracted function should be called from the previous function.

##### Single line code blocks

Nested structures like if, else, while, for, try, etc should ideally call another function in their code block. This makes the code easier to read and understand.

#### Should do just one thing

A function should do one thing and do it well. In general, all the things in a function should be at the same abstraction level. ***A function should not have both lower-level and higher-level details.***

If another function can be extracted out of a function then it is doing more than one thing***. We should extract functions to achieve the same level of abstraction***,

Functions that can be divided into multiple sections do more than one thing.

#### Should have fewer arguments

*The ideal number of arguments for a function is zero (niladic). Next comes one (monadic), followed closely by two (dyadic). Three arguments (triadic) should be avoided where possible. More than three (polyadic) requires very special justification—and then shouldn’t be used anyway.*

*- Robert C. Martin (Uncle Bob)*

Arguments make the code harder to understand and test and so we should keep it as low as possible. In most cases, a set of related arguments used across functions deserves to be wrapped in a class of its own.

Output arguments are arguments that are required by the function only to be modified to contain the output. Output arguments make the function harder to understand and should be avoided.

**Flag arguments** are boolean arguments that make the code do two things based on the flag value. It **should be avoided** *as it breaks the rule of doing only one thing.*

#### Should not have side effects

A function should not promise one thing and do something else as a side effect.

Example: Changing the parameters/class properties in a get/query method.

A set/update (command) function should not ideally update the parameters. They can however update the properties of the class the function belongs to.

#### How to write good functions?

It is okay to start with long functions with multiple arguments which do more than one thing. Start refactoring larger functions into smaller functions till there is a single level of abstraction in the function and no more functions can be extracted. Make sure that the function names are descriptive and have fewer arguments. In the end, all the functions should follow the above rules.

### Designing Good Classes

Classes bind related data and expose functions that operate on that data. This helps make the code more organized.

There are many design principles and patterns that can help make our code better organized. Let's look at some rules which can help us design better classes:

* Organized and Encapsulated
* Should be small and should do just one thing
* Small number of instance variables

#### Organized and Encapsulated

A class should keep all data attributes and utility functions private. Only the functions that are supposed to be exposed should be kept public.

A class should be ordered like this:

* Start with the variables
  + **public static constants**
  + followed by **private static attributes**
  + followed by **private instance attributes**
* followed by **public functions**
* **private utility function** just after the public function that calls it.

#### Should be small and should do just one thing

A class should be very small. Different experts have different views on "how small?". Based on most of the views, a class should be small enough that it does just one thing. In general, it should have less than 20 functions.

If a class does more than one thing then it should be broken down into different classes each of which does a single thing.

#### Small number of instance variables

A class should have strong cohesion, i.e., the functions of a class should be strongly related in supporting a single central purpose.

A maximal cohesive class is one in which all functions work with all the instance variables. Achieving that is pretty difficult. We should try to make our classes as cohesive as possible. This can be done by having a smaller number of instance variables such that each function in the class work with as many of the instance variables as possible. This can be done by splitting the class into multiple classes based on the responsibility of each class.

References: [Clean Code: A Handbook of Agile Software Craftsmanship by Robert C. Martin (Uncle Bob)](https://www.goodreads.com/book/show/3735293-clean-code)

# Clean Code and Software Design Principles: Software Design Principles

## Basic Design Principles

### Don't Repeat Yourself (DRY)

Duplication of code can make code very difficult to maintain. Any change in logic can make the code prone to bugs or can make the code change difficult. This can be fixed by doing code reuse (DRY Principle).

The DRY principle is stated as "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system".

The way to achieve DRY is by creating functions and classes to make sure that any logic should be written in only one place.

### Curly's Law - Do One Thing

Curly's Law is about choosing a single, clearly defined goal for any particular bit of code: Do One Thing.

Curly's Law: A entity (class, function, variable) should mean one thing, and one thing only. It should not mean one thing in one circumstance and carry a different value from a different domain some other time. It should not mean two things at once. It should mean One Thing and should mean it all of the time.

### Keep It Simple Stupid (KISS)

The KISS principle states that most systems work best if they are kept simple rather than made complicated; therefore, simplicity should be a key goal in design, and unnecessary complexity should be avoided.

Simple code has the following benefits:

* less time to write
* less chances of bugs
* easier to understand, debug and modify

Do the simplest thing that could possibly work.

#### Don't make me think

Code should be easy to read and understand without much thinking. If it isn't then there is a prospect of simplification.

### You Aren't Gonna Need It (YAGNI)

You Aren't Gonna Need It (YAGNI) is an Extreme Programming (XP) practice which states: "Always implement things when you actually need them, never when you just foresee that you need them."

Even if you're totally, totally, totally sure that you'll need a feature, later on, don't implement it now. Usually, it'll turn out either:

* you don't need it after all, or
* what you actually need is quite different from what you foresaw needing earlier.

This doesn't mean you should avoid building flexibility into your code. It means you shouldn't overengineer something based on what you think you might need later on.

There are two main reasons to practice YAGNI:

* You save time because you avoid writing code that you turn out not to need.
* Your code is better because you avoid polluting it with 'guesses' that turn out to be more or less wrong but stick around anyway.

### Premature Optimization is the Root of All Evil

Programmers waste enormous amounts of time thinking about or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered.

*We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.*

*- Donald Knuth*

### Boy-Scout principle

Any time someone sees some code that isn't as clear as it should be, they should take the opportunity to fix it right there and then - or at least within a few minutes.

This opportunistic refactoring is referred to by Uncle Bob as following the boy-scout rule - always leave the code behind in a better state than you found it.

The code quality tends to degrade with each change. This results in technical debt. The Boy-Scout Principle saves us from that.

### Code for the Maintainer

Code maintenance is an expensive and difficult process. Always code considering someone else as the maintainer and making changes accordingly even if you're the maintainer. After a while, you'll remember the code as much as a stranger.

Always code as if the person who ends up maintaining your code is a violent psychopath who knows where you live.

### Principle of Least Astonishment

Principle of Least Astonishment states that a component of a system should behave in a way that most users will expect it to behave. The behavior should not astonish or surprise users.

Code should do what the name and comments suggest. Conventions should be followed. Surprising side effects should be avoided as much as possible.

References:

* [Extreme Programming](http://c2.com/xp/ExtremeProgramming.html)
* [martinfowler.com](https://martinfowler.com/)

**Clean Code and Software Design Principles: Software Design Principles**

**Design Principles (Functions & Classes)**

**Hide Implementation Details**

Hiding implementation details helps to make changes in a component without making changes in the other modules/clients using that component. This can be achieved by creating interfaces and using them instead of the concrete classes.

Encapsulation with proper access management should also be done to expose only the required public functions.

**Separation of Concerns**

*Separation of concerns (SoC) is a design principle for separating a computer program into distinct sections such that each section addresses a separate concern.*

A program that embodies SoC well is called a **modular program**. Modularity, and hence separation of concerns, is *achieved by creating well-encapsulated classes that have well-defined interfaces*.

We should ***strive to separate the program into separate sections such that the overlap is as minimal as possible***. SoC helps maintenance and also code reuse.

**Maximize Cohesion**

***Cohesion is the degree to how strongly related and focused are the various responsibilities of a module***. ***It is a measure of the strength of the relationship between the class’s methods and data themselves.*** We should strive to maximize cohesion. *High cohesion results in better understanding, maintaining, and reusing components.*

Cohesion is increased if:

* *The functionalities embedded in a class, accessed through its methods, have much in common.*
* *Methods carry out a small number of related activities, by avoiding coarsely grained or unrelated sets of data.*
* *Related methods are in the same source file or otherwise grouped together; for example, in separate files but in the same sub-directory/folder.*

**Minimize Coupling**

***Coupling is the degree to which each module depends on other modules; a measure of how closely connected two modules are.*** We should strive to ***minimize coupling***.

**Coupling is usually contrasted with cohesion. Low coupling often correlates with high cohesion and vice versa.**

Tightly coupled modules have the following ***disadvantages***:

* ***Change in one module might break another module.***
* ***Change in one module usually forces a ripple effect of changes in other modules.***
* ***Reusability decreases as dependency over other modules increases.***
* ***Assembly of modules might require more effort and/or time.***

Coupling can be reduced by:

* By hiding inner details and interacting through interfaces.
* Avoid interacting with classes that it can avoid directly dealing with.

Components in a loosely coupled system can be replaced with alternative implementations that provide the same services.

**Law of Demeter/Principle of Least Knowledge**

***Code components should only talk to its direct relations and not to strangers.***

The Law of Demeter for functions requires that a method `m` of an object `a` may only invoke the methods of the following kinds of objects:

* a itself;
* m's parameters;
* any objects instantiated within m;
* a's attributes;
* global variables accessible by a in the scope of m.

In particular, an object should avoid invoking methods of an object returned by another method. ***For many modern object-oriented languages that use a dot as a field identifier, the law can be stated simply as "use only one dot". That is, the code a.m().m() breaks the law where a.m() does not***. As an analogy, when one wants a dog to walk, one does not command the dog's legs to walk directly; instead one commands the dog which then commands its own legs.

**Command-Query Separation**

***Command-Query Separation (CQS) states that every method should either be a command that performs an action, or a query that returns data to the caller, but not both***. In other words, asking a question should not change the answer.

***Query: Returns a result without changing the state. Command: Changes the state but does not return any value.***

***This way the query method could be used anywhere without changing the data/state.*** We should apply naming conventions (get, set, add, etc.) to imply whether it is a command or a query.

References:

* [Extreme Programming](http://c2.com/xp/ExtremeProgramming.html)
* [martinfowler.com](https://martinfowler.com/)
* [Wikipedia](https://en.wikipedia.org/)

**Clean Code and Software Design Principles: Software Design Principles**

**SOLID Design Principles**

SOLID Design Principles are a set of five software design principles promoted by Robert C. Martin (Uncle Bob) in his paper "[Design Principles and Design Patterns](https://fi.ort.edu.uy/innovaportal/file/2032/1/design_principles.pdf)".

The SOLID principles aim to make software design easy to understand, maintain, and, flexible.

The SOLID Design Principles are:

* The ***Single-responsibility principle***: "There should never be more than one reason for a class to change." In other words, every class should have only one responsibility.
* The ***Open-Closed principle***: "Software entities ... should be open for extension, but closed for modification."
* The ***Liskov substitution principle***: "Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it".
* The ***Interface segregation principle***: "Many client-specific interfaces are better than one general-purpose interface."
* The ***Dependency inversion principle***: "Depend upon abstractions, [not] concretions."

Before we get into understanding each of the five principles in-depth, let's note these three points:

* ***Make sure that classes interact with each other through interfaces.***
* ***Create small classes that do a single thing.***
* ***Create small interfaces such that a class only has to implement functionalities that it should.***

***The above three points if followed well will also result in following the SOLID principles.***

**Single Responsibility Principle (SRP)**

The ***Single Responsibility Principle*** states that *there should never be more than one reason for a class to change*. In other words, *every class should have only one responsibility.*

If there are two reasons to change a class then it has more than one responsibility and should be split. Having multiple reasons to change means that the module has high coupling.

The more responsibilities your class has, the more often you'll need to change it. Since they won't be independent of each other, change to one responsibility may affect the other.

**Open-Closed Principle (OCP)**

The **Open-Closed Principle** states that a software entity (class, module, function) should be *open for extension but closed for modification*. We should write our modules so that they can be extended, without requiring them to be modified.

*Open for extension:* ***A class should be extensible, i.e., we should be able to make a class behave in different ways as new requirements come.***

*Closed for modification: The extension should happen without any modification to the class.*

***This can be achieved through abstraction, i.e., by creating interfaces. Interfaces should not change based on new requirements. Interfaces are extensible and new requirements can be taken care of by implementing classes.***

**Liskov Substitution Principle (LSP)**

The **Liskov Substitution Principle** states that *subclasses should be substitutable for their base classes*. *Derived classes should be substitutable for their base classes wherever the derived class is used*.

*A client using a base class should continue to function properly even if a derived class is passed to it*. This means that we should be able to pass an object of class B to any function that expects an object of class A given that class B is a subclass of class A.

**Dependency Inversion Principle (DIP)**

The **Dependency Inversion Principle** states that "**Depend upon abstractions, [not] concretions**". This means that ***classes should depend on abstractions (interfaces) of other classes instead of the concrete implementations.***

The dependencies are inverted as the concrete classes depend upon abstractions instead of being dependent upon by other classes.

**Interface Segregation Principle (ISP)**

The Interface Segregation Principle states that many client-specific interfaces are better than one general-purpose interface.

Clients should not be forced to depend upon interfaces that they do not use. It is better to have smaller interfaces than fewer, fatter interfaces. An interface should be dependent more on the code that calls it rather than the code that implements it.

Therefore, a class should be implementing different interfaces created based on the different types of clients. Each of these interfaces might have common methods as well.

References: [Design Principles and Design Patterns](https://fi.ort.edu.uy/innovaportal/file/2032/1/design_principles.pdf)