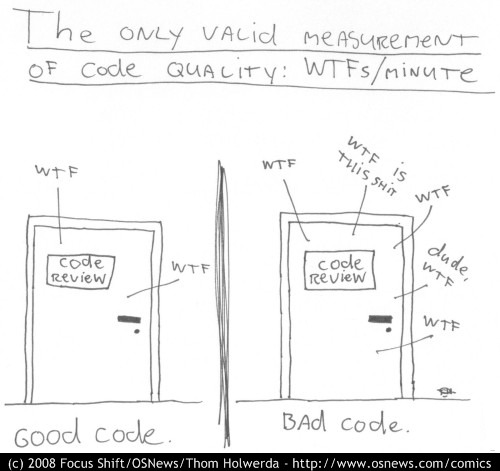
# Clean Code concepts for .NET/.NET Core

Research using Google, Medium Articles, ChatGPT, Claude, GitHub.

* [Clean Code concepts adapted for .NET/.NET Core](#X1a1dd953e68c6cd5e7f2209188f52fcde67f7c4)
* [Table of Contents](#table-of-contents)
* [Introduction](#introduction)
* [Clean Code .NET](#clean-code-net)
  + [Naming](#naming)
  + [Variables](#variables)
  + [Functions](#functions)
  + [Error Handling](#error-handling)
  + [Comments](#comments)

# 

# Introduction



# Clean Code Why?

## Just as Important: Clean code is as crucial as performance and following best practices.

## Keeps Things Manageable: It makes our code easier to maintain and update down the line.

## Team-Friendly: Clean code is a big help when collaborating with others—it makes your work understandable.

## Prepares for the Future: While speed and efficiency matter, clean code ensures your project can grow without becoming a mess.

## Naming

* **Avoid using bad names**

A good name allows the code to be used by many developers. The name should reflect what it does and give context.

**Bad:**

int d;

**Good:**

int daySinceModification;

* **Avoid Misleading Names**

Name the variable to reflect what it is used for.

**Bad:**

var dataFromDb = db.GetFromService().ToList();

**Good:**

var listOfEmployee = \_employeeService.GetEmployees().ToList();

* **Avoid Hungarian notation**

Hungarian Notation restates the type which is already present in the declaration. This is pointless since modern IDEs will identify the type. Here user add some **prefix** before variable name

**Bad:**

int iCounter;  
string strFullName;  
DateTime dModifiedDate;

**Good:**

int counter;  
string fullName;  
DateTime modifiedDate;

Hungarian Notation should also not be used in paramaters.

**Bad:**

public bool IsShopOpen(string pDay, int pAmount)  
{  
 // some logic  
}

**Good:**

public bool IsShopOpen(string day, int amount)  
{  
 // some logic  
}

* **Use consistent capitalization**

Capitalization tells you a lot about your variables, functions, etc. These rules are subjective, so your team can choose whatever they want. The point is, no matter what you all choose, just be consistent.

**Bad:**

const int DAYS\_IN\_WEEK = 7;  
const int daysInMonth = 30;  
  
var songs = new List<string> { 'Back In Black', 'Stairway to Heaven', 'Hey Jude' };  
var Artists = new List<string> { 'ACDC', 'Led Zeppelin', 'The Beatles' };  
  
bool EraseDatabase() {}  
bool Restore\_database() {}  
  
class animal {}  
class Alpaca {}

**Good:**

const int DaysInWeek = 7;  
const int DaysInMonth = 30;  
  
var songs = new List<string> { 'Back In Black', 'Stairway to Heaven', 'Hey Jude' };  
var artists = new List<string> { 'ACDC', 'Led Zeppelin', 'The Beatles' };  
  
bool EraseDatabase() {}  
bool RestoreDatabase() {}  
  
class Animal {}  
class Alpaca {}

* **Use pronounceable names**

It will take time to investigate the meaning of the variables and functions when they are not pronounceable.

**Bad:**

public class Employee  
{  
 public Datetime sWorkDate { get; set; } // what the heck is this  
 public Datetime modTime { get; set; } // same here  
}

**Good:**

public class Employee  
{  
 public Datetime StartWorkingDate { get; set; }  
 public Datetime ModificationTime { get; set; }  
}

* **Use Camelcase notation**

Use [Camelcase Notation](https://en.wikipedia.org/wiki/Camel_case) for variable and method paramaters.

**Bad:**

var employeephone;  
  
public double CalculateSalary(int workingdays, int workinghours)  
{  
 // some logic  
}

**Good:**

var employeePhone;  
  
public double CalculateSalary(int workingDays, int workingHours)  
{  
 // some logic  
}

## Variables

* **Avoid nesting too deeply and return early**

Too many if else statements can make the code hard to follow. **Explicit is better than implicit**.

**Bad:**

public bool IsShopOpen(string day)  
{  
 if (!string.IsNullOrEmpty(day))  
 {  
 day = day.ToLower();  
 if (day == "friday")  
 {  
 return true;  
 }  
 else if (day == "saturday")  
 {  
 return true;  
 }  
 else if (day == "sunday")  
 {  
 return true;  
 }  
 else  
 {  
 return false;  
 }  
 }  
 else  
 {  
 return false;  
 }  
  
}

**Good:**

public bool IsShopOpen(string day)  
{  
 if (string.IsNullOrEmpty(day))  
 {  
 return false;  
 }  
  
 var openingDays = new[] { "friday", "saturday", "sunday" };  
 return openingDays.Any(d => d == day.ToLower());  
}

* **Avoid mental mapping**

Don’t force the reader of your code to translate what the variable means. **Explicit is better than implicit**.

**Bad:**

var l = new[] { "Austin", "New York", "San Francisco" };  
  
for (var i = 0; i < l.Count(); i++)  
{  
 var li = l[i];  
 DoStuff();  
 DoSomeOtherStuff();  
  
 // ...  
 // ...  
 // ...  
 // Wait, what is `li` for again?  
 Dispatch(li);  
}

**Good:**

var locations = new[] { "Austin", "New York", "San Francisco" };  
  
foreach (var location in locations)  
{  
 DoStuff();  
 DoSomeOtherStuff();  
  
 // ...  
 // ...  
 // ...  
 Dispatch(location);  
}

* **Avoid magic string**

Magic strings are string values that are specified directly within the application code(Hard Code) that have an impact on the application’s behaviour. Frequently, such strings will end up being duplicated within the system, and since they cannot automatically be updated using refactoring tools, they become a common source of bugs when changes are made to some strings but not others.

**Bad**

if (userRole == "Admin")  
{  
 // logic in here  
}

**Good**

const string ADMIN\_ROLE = "Admin"  
if (userRole == ADMIN\_ROLE)  
{  
 // logic in here  
}

Using this we only have to change in centralize place and others will adapt it.

* **Don’t add unneeded context**

If your class/object name tells you something, don’t repeat that in your variable name.

**Bad:**

public class Car  
{  
 public string CarMake { get; set; }  
 public string CarModel { get; set; }  
 public string CarColor { get; set; }  
  
 //...  
}

**Good:**

public class Car  
{  
 public string Make { get; set; }  
 public string Model { get; set; }  
 public string Color { get; set; }  
  
 //...  
}

* **Use meaningful and pronounceable variable names**

**Bad:**

var ymdstr = DateTime.UtcNow.ToString("MMMM dd, yyyy");

**Good:**

var currentDate = DateTime.UtcNow.ToString("MMMM dd, yyyy");

* **Use the same vocabulary for the same type of variable**

**Bad:**

GetUserInfo();  
GetUserData();  
GetUserRecord();  
GetUserProfile();

**Good:**

GetUser();

* **Use searchable names (part 1)**

We will read more code than we will ever write. It’s important that the code we do write is readable and searchable. By *not* naming variables that end up being meaningful for understanding our program, we hurt our readers. Make your names searchable.

**Bad:**

// What the heck is data for?  
var data = new { Name = "John", Age = 42 };  
  
var stream1 = new MemoryStream();  
var ser1 = new DataContractJsonSerializer(typeof(object));  
ser1.WriteObject(stream1, data);  
  
stream1.Position = 0;  
var sr1 = new StreamReader(stream1);  
Console.Write("JSON form of Data object: ");  
Console.WriteLine(sr1.ReadToEnd());

**Good:**

var person = new Person  
{  
 Name = "John",  
 Age = 42  
};  
  
var stream2 = new MemoryStream();  
var ser2 = new DataContractJsonSerializer(typeof(Person));  
ser2.WriteObject(stream2, data);  
  
stream2.Position = 0;  
var sr2 = new StreamReader(stream2);  
Console.Write("JSON form of Data object: ");  
Console.WriteLine(sr2.ReadToEnd());

Use searchable names (part 2)

**Bad:**

var data = new { Name = "John", Age = 42, PersonAccess = 4};  
  
// What the heck is 4 for?  
if (data.PersonAccess == 4)  
{  
 // do edit ...  
}

**Good:**

public enum PersonAccess : int  
{  
 ACCESS\_READ = 1,  
 ACCESS\_CREATE = 2,  
 ACCESS\_UPDATE = 4,  
 ACCESS\_DELETE = 8  
}  
  
var person = new Person  
{  
 Name = "John",  
 Age = 42,  
 PersonAccess= PersonAccess.ACCESS\_CREATE  
};  
  
if (person.PersonAccess == PersonAccess.ACCESS\_UPDATE)  
{  
 // do edit ...  
}

* **Use explanatory variables (Regex)**

**Bad:**

const string Address = "One Infinite Loop, Cupertino 95014";  
var cityZipCodeRegex = @"/^[^,\]+[,\\s]+(.+?)\s\*(\d{5})?$/";  
var matches = Regex.Matches(Address, cityZipCodeRegex);  
if (matches[0].Success == true && matches[1].Success == true)  
{  
 SaveCityZipCode(matches[0].Value, matches[1].Value);  
}

**Good:**

Decrease dependence on regex by naming subpatterns.

const string Address = "One Infinite Loop, Cupertino 95014";  
var cityZipCodeWithGroupRegex = @"/^[^,\]+[,\\s]+(?<city>.+?)\s\*(?<zipCode>\d{5})?$/";  
var matchesWithGroup = Regex.Match(Address, cityZipCodeWithGroupRegex);  
var cityGroup = matchesWithGroup.Groups["city"];  
var zipCodeGroup = matchesWithGroup.Groups["zipCode"];  
if(cityGroup.Success == true && zipCodeGroup.Success == true)  
{  
 SaveCityZipCode(cityGroup.Value, zipCodeGroup.Value);  
}

* **Use default arguments instead of short circuiting or conditionals**

**Not good:**

This is not good because breweryName can be NULL.

public void CreateStudentId(string name = null)  
{  
 var studentId = !string.IsNullOrEmpty(name) ? name : "S364848";  
 // ...  
}

**Good:**

public void CreateMicrobrewery(string studentId = "S364848")  
{  
 // ...  
}

## Functions

* **Avoid negative conditionals**

More readability and easy-to-understand

**Bad:**

public bool IsDOMNodeNotPresent(string node)  
{  
 // ...  
}  
  
if (!IsDOMNodeNotPresent(node))  
{  
 // ...  
}

**Good:**

public bool IsDOMNodePresent(string node)  
{  
 // ...  
}  
  
if (IsDOMNodePresent(node))  
{  
 // ...  
}

* **Avoid type-checking**

**Bad:**

public Path TravelToTexas(object vehicle)  
{  
 if (vehicle.GetType() == typeof(Bicycle))  
 {  
 (vehicle as Bicycle).PeddleTo(new Location("texas"));  
 }  
 else if (vehicle.GetType() == typeof(Car))  
 {  
 (vehicle as Car).DriveTo(new Location("texas"));  
 }  
}

**Good:**

public Path TravelToTexas(Traveler vehicle)  
{  
 vehicle.TravelTo(new Location("texas"));  
}

or

* **Avoid flags in method parameters**

A flag indicates that the method has more than one responsibility. It is best if the method only has a single responsibility. Split the method into two if a boolean parameter adds multiple responsibilities to the method.

**Bad:**

public void CreateFile(string name, bool temp = false)  
{  
 if (temp)  
 {  
 Touch("./temp/" + name);  
 }  
 else  
 {  
 Touch(name);  
 }  
}

**Good:**

public void CreateFile(string name)  
{  
 Touch(name);  
}  
  
public void CreateTempFile(string name)  
{  
 Touch("./temp/" + name);  
}

* **Function arguments (2 or fewer ideally)**

Limiting the amount of function parameters is incredibly important because it makes testing your function easier. Having more than three leads to a combinatorial explosion where you have to test tons of different cases with each separate argument.

Zero arguments is the ideal case. One or two arguments is ok, and three should be avoided. Anything more than that should be consolidated. Usually, if you have more than two arguments then your function is trying to do too much. In cases where it’s not, most of the time a higher-level object will suffice as an argument.

**Bad:**

public void CreateMenu(string title, string body, string buttonText, bool cancellable)  
{  
 // ...  
}

**Good:**

public class MenuConfig  
{  
 public string Title { get; set; }  
 public string Body { get; set; }  
 public string ButtonText { get; set; }  
 public bool Cancellable { get; set; }  
}  
  
var config = new MenuConfig  
{  
 Title = "Foo",  
 Body = "Bar",  
 ButtonText = "Baz",  
 Cancellable = true  
};  
  
public void CreateMenu(MenuConfig config)  
{  
 // ...  
}

* **Functions should do one thing**

This is by far the most important rule in software engineering. When functions do more than one thing, they are harder to compose, test, and reason about. When you can isolate a function to just one action, they can be refactored easily and your code will read much cleaner. If you take nothing else away from this guide other than this, you’ll be ahead of many developers.

**Bad:**

public void SendEmailToListOfClients(string[] clients)  
{  
 foreach (var client in clients)  
 {  
 var clientRecord = db.Find(client);  
 if (clientRecord.IsActive())  
 {  
 Email(client);  
 }  
 }  
}

**Good:**

public void SendEmailToListOfClients(string[] clients)  
{  
 var activeClients = GetActiveClients(clients);  
 // Do some logic  
}  
  
public List<Client> GetActiveClients(string[] clients)  
{  
 return db.Find(clients).Where(s => s.Status == "Active");  
}

Function names should say what they do

* **Function callers and callees should be close**

If a function calls another, keep those functions vertically close in the source file. Ideally, keep the caller right above the callee. We tend to read code from top-to-bottom, like a newspaper. Because of this, make your code read that way.

**Bad:**

class PerformanceReview  
{  
 private readonly Employee \_employee;  
  
 public PerformanceReview(Employee employee)  
 {  
 \_employee = employee;  
 }  
  
 private IEnumerable<PeersData> LookupPeers()  
 {  
 return db.lookup(\_employee, 'peers');  
 }  
  
 private ManagerData LookupManager()  
 {  
 return db.lookup(\_employee, 'manager');  
 }  
  
 private IEnumerable<PeerReviews> GetPeerReviews()  
 {  
 var peers = LookupPeers();  
 // ...  
 }  
  
 public PerfReviewData PerfReview()  
 {  
 GetPeerReviews();  
 GetManagerReview();  
 GetSelfReview();  
 }  
  
 public ManagerData GetManagerReview()  
 {  
 var manager = LookupManager();  
 }  
  
 public EmployeeData GetSelfReview()  
 {  
 // ...  
 }  
}  
  
var review = new PerformanceReview(employee);  
review.PerfReview();

**Good:**

class PerformanceReview  
{  
 private readonly Employee \_employee;  
  
 public PerformanceReview(Employee employee)  
 {  
 \_employee = employee;  
 }  
  
 public PerfReviewData PerfReview()  
 {  
 GetPeerReviews();  
 GetManagerReview();  
 GetSelfReview();  
 }  
  
 private IEnumerable<PeerReviews> GetPeerReviews()  
 {  
 var peers = LookupPeers();  
 // ...  
 }  
  
 private IEnumerable<PeersData> LookupPeers()  
 {  
 return db.lookup(\_employee, 'peers');  
 }  
  
 private ManagerData GetManagerReview()  
 {  
 var manager = LookupManager();  
 return manager;  
 }  
  
 private ManagerData LookupManager()  
 {  
 return db.lookup(\_employee, 'manager');  
 }  
  
 private EmployeeData GetSelfReview()  
 {  
 // ...  
 }  
}  
  
var review = new PerformanceReview(employee);  
review.PerfReview();

* **Encapsulate conditionals**

**Bad:**

if (article.state == "published")  
{  
 // ...  
}

**Good:**

if (article.IsPublished())  
{  
 // ...  
}

* **Remove dead code**

Dead code is just as bad as duplicate code. There’s no reason to keep it in your codebase. If it’s not being called, get rid of it! It will still be safe in your version history if you still need it.

**Bad:**

public void OldRequestModule(string url)  
{  
 // ...  
}  
  
public void NewRequestModule(string url)  
{  
 // ...  
}  
  
var request = NewRequestModule(requestUrl);  
InventoryTracker("apples", request, "www.inventory-awesome.io");

**Good:**

public void RequestModule(string url)  
{  
 // ...  
}  
  
var request = RequestModule(requestUrl);  
InventoryTracker("apples", request, "www.inventory-awesome.io");

## Error Handling

Basic concept of error handling

Thrown errors are a good thing! They mean the runtime has successfully identified when something in your program has gone wrong and it’s letting you know by stopping function execution on the current stack, killing the process (in .NET/.NET Core), and notifying you in the console with a stack trace.

Don’t use ‘throw ex’ in catch block

If you need to re-throw an exception after catching it, use just ‘throw’ By using this, you will save the stack trace. But in the bad option below, you will lost the stack trace.

**Bad:**

try  
{  
 // Do something..  
}  
catch (Exception ex)  
{  
 // Any action something like roll-back or logging etc.  
 throw ex;  
}

**Good:**

try  
{  
 // Do something..  
}  
catch (Exception ex)  
{  
 // Any action something like roll-back or logging etc.  
 throw;  
}

* **Don’t ignore caught errors**

Doing nothing with a caught error doesn’t give you the ability to ever fix or react to said error. Throwing the error isn’t much better as often times it can get lost in a sea of things printed to the console. If you wrap any bit of code in a try/catch it means you think an error may occur there and therefore you should have a plan, or create a code path, for when it occurs.

**Bad:**

try  
{  
 FunctionThatMightThrow();  
}  
catch (Exception ex)  
{  
 // silent exception  
}

**Good:**

try  
{  
 FunctionThatMightThrow();  
}  
catch (Exception error)  
{  
 NotifyUserOfError(error);  
  
 // Another option  
 ReportErrorToService(error);  
}

* **Use multiple catch block instead of if conditions.**

If you need to take action according to type of the exception, you better use multiple catch block for exception handling.

**Bad:**

try  
{  
 // Do something..  
}  
catch (Exception ex)  
{  
  
 if (ex is TaskCanceledException)  
 {  
 // Take action for TaskCanceledException  
 }  
 else if (ex is TaskSchedulerException)  
 {  
 // Take action for TaskSchedulerException  
 }  
}

**Good:**

try  
{  
 // Do something..  
}  
catch (TaskCanceledException ex)  
{  
 // Take action for TaskCanceledException  
}  
catch (TaskSchedulerException ex)  
{  
 // Take action for TaskSchedulerException  
}

* **Keep exception stack trace when rethrowing exceptions**

C# allows the exception to be rethrown in a catch block using the throw keyword. It is a bad practice to throw a caught exception using throw e;. This statement resets the stack trace. Instead use throw;. This will keep the stack trace and provide a deeper insight about the exception. Another option is to use a custom exception. Simply instantiate a new exception and set its inner exception property to the caught exception with throw new CustomException("some info", e);. Adding information to an exception is a good practice as it will help with debugging. However, if the objective is to log an exception then use throw; to pass the buck to the caller.

**Bad:**

try  
{  
 FunctionThatMightThrow();  
}  
catch (Exception ex)  
{  
 logger.LogInfo(ex);  
 throw ex;  
}

**Good:**

try  
{  
 FunctionThatMightThrow();  
}  
catch (Exception error)  
{  
 logger.LogInfo(error);  
 throw;  
}

**Good:**

try  
{  
 FunctionThatMightThrow();  
}  
catch (Exception error)  
{  
 logger.LogInfo(error);  
 throw new CustomException(error);  
}