Clear Intent

Last updated by | Vincent Boeijen | 6 feb 2024 at 21:27 CET

Clear Intent

General

In the world of software development, writing code is not just about making the computer understand instructions; it's also about making the code easily understandable for fellow developers. Clarity of intent is a crucial aspect of writing clean and maintainable code.

✋ Do use clear names for variables and methods

> Why? Names like int number, person or methods like Calculate or ProcessData give the reader next to no information about the intent of these objects. As a rule of thumb, one should not have to look at a method body or related code to understand the intent of a function or variable. If you find yourself having the need to write a lot of comments, your code likely isn't very clear in its intent.

```
// Poor naming
int d; // Days passed since last login

bool flag = true;

public double Calculate(int l, int w)
{
    return l * w;
}
```

```
// Improved naming
int daysSinceLastLogin;

bool isLoggedIn = true;

public double CalculateRectangleArea(int length, int width)
{
    return length * width;
}
```

⚠️ Don't use Magic Numbers or Strings

> Why? Magic numbers and strings can be a source of confusion for anyone reading the code. Instead, use constants or enumerations to give meaningful names to these values.

```
// Magic number
if (status == 1) { /* ... */ }
```

```csharp
// Improved readability using an enum for user status
public enum UserStatus
{
    Inactive = 0,
    Active = 1,
    Suspended = 2,
    // Add more statuses as needed
}

if (status == UserStatus.Active){ /* ... */ }
```

💡 Consider the var keyword

As a side note, consider not overusing the var keyword. Generally, the var keyword is fine if the type of the object can be inferred on the same line or by reading the value type. Try to avoid it when it's not clear unless one reads the related code.

```csharp
// Good use of var when the type is clear from the right-hand side
var firstName = "John";
var age = 25;
var prices = new List<decimal> { 10.5m, 20.2m, 15.8m };

foreach (var price in prices)
{
     ...
}

var average = prices.Average();
```

```csharp
// Bad use of var when the type is unclear
var result = CalculateSomething();

foreach (var item in GetItems())
{
     ...
}
```

✋ Do use SRP and short and focused methods

Why? Following the Single Responsibility Principle (SRP) ensures that each class or method has a single responsibility. This not only improves the modularity of your code but also makes the intent of each component clear. As a rule of thumb, try to keep methods shorter than a screen's length.

```csharp
// Violating SRP
class UserManager
{
    public void UpdateUser(User user) { /* ... */ }
    public void SendEmail(User user) { /* ... */ }
}
```

```
// Adhering to SRP
class UserManager
{
    public void UpdateUser(User user) { /* ... */ }
}

class EmailService
{
    public void SendEmail(User user) { /* ... */ }
}
```

⚠️ Don't use nested conditional logic

Why? Excessive nesting uses a lot of braces and indentantion which might degrade code readability. Use expressive conditional statements and guard clauses to enhance the readability of your code. Avoid nested conditions and aim for a flat structure that clearly articulates the decision-making process.

```
// Nested conditions
if (condition1)
{
    if (condition2)
    {
        // ...
    }
}
```

```
// Flat structure with guard clauses
if (!condition1) return;
if (!condition2) return;
// ...
```

✋ Do favor Readability Over Cleverness

Why? Code is read more often than it is written. Write code that is easy to understand rather than trying to be overly clever, so prioritize readability for the benefit of yourself and others.

```
// Clever but less readable
var result = x ^ y; // XOR to swap values

// Readable and clear
SwapValues(x, y);
```

✋ Do use Dependency Injection to decouple classes

Try to avoid newing up services in other classes as this tightly couples classes together. Use Dependency Injection to decouple classes. This also relieves the class of the burden (dependencies may have dependencies of their own) of having to create it's own dependencies. See the car analogy:

A car needs an engine and a collection of tires to drive, but it's not expected to install its own engine, or grow its own tires. These dependencies are provided to the car.

List your dependencies at the top of the class as if they were ingredients in a recipes. A user appreciates being told up front what is needed instead of having to provide new ingredients halfway into the cooking process.

```
// Poor example using new
public class PoorExample
{
    private IMyDependency _dependency;

    public PoorExample ()
    {
        _dependency= new MyDependency();
    }
}

// Even worse
public class AtrociousExample
{
    private IMyDependency _dependency;

    public PoorMyDependency()
    {
        _dependency= new MyDependency(new ConcreteDependencyA(), new ConcreteDependencyB());
    }
}


// Good example using Constructor Injection, the creation responsibility is delegated elsewhere.
public class MyClass
{
    private IMyDependency _dependency;

    public MyClass(IMyDependency dependency)
    {
        _dependency = dependency;
    }
}
```

✋ Do use Allman style braces

Why? The Allman style [1] is a popular coding style in C# that involves placing the opening brace of a code block on a new line, and the closing brace on a new line as well. This style is also known as the "curly brace" style.
The main advantage of using Allman style braces is that it makes the code more readable and easier to understand. It also helps to avoid errors that can occur when braces are not used consistently.

It is generally recommended to use braces even for single-line code blocks. This is because it can help to avoid errors [2] and make the code more consistent.

[1] wikipedia for more information.
[2] Apple's goto fail bug

⚠️ Don't use regions

> Why? The #region directive in C# is used to group related code together and make it collapsible in the Visual Studio IDE. Because Visual Studio collapses regions by default, they hide code from other developers so they can't fully understand what the code is doing.

Only in (unit)test code it is sometimes ok to use regions, especially when the setup code is to long.
But even then, your first reaction should be to create a 'setup' method.

✋ Do use the dotNET Lab .editorconfig in your solution

> Why? An .editorconfig file helps maintain consistent coding styles for multiple developers working on the same project across various editors and IDEs.

see for more information about the .editorconfig format.

✋ Do enable the build-in dotNET analyzers

> Why? To have a consistence coding experience enable the build in .NET analyzers.

Add the following block to each *.csproj file.

```
<PropertyGroup>
    <AnalysisLevel>latest-all</AnalysisLevel>
    <EnableNETAnalyzers>True</EnableNETAnalyzers>
    <EnforceCodeStyleInBuild>True</EnforceCodeStyleInBuild>
    <TreatWarningsAsErrors>true</TreatWarningsAsErrors>
</PropertyGroup>
```

✋ Do use StyleCop.Analyzers in your C# projects

> Why? Adding the StyleCop.Analyzers will mark errors in the IDE **while** you are coding. This will help you to spot issue early one and avoid that errors only popup when you build the solution on the build

server.

Add the following block to each *.csproj file.

```
<PropertyGroup>
    <AnalysisLevel>latest-all</AnalysisLevel>
    <EnableNETAnalyzers>True</EnableNETAnalyzers>
    <EnforceCodeStyleInBuild>True</EnforceCodeStyleInBuild>
    <TreatWarningsAsErrors>true</TreatWarningsAsErrors>
</PropertyGroup>

<PackageReference Include="StyleCop.Analyzers" Version="1.1.118">
    <PrivateAssets>all</PrivateAssets>
    <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
</PackageReference>
```

✋ Do add a Version tag in all your C# projects

Why? The dotNETlab build pipelines will add a unique version number to each build. The format will be 1.0.0.BuildId where BuildId is an incremental build number.

Add the following block to each *.csproj file.

```
<PropertyGroup>
    <Version>1.0.0</Version>
    <AssemblyFileVersion>$(Version).0</AssemblyFileVersion>
    <AssemblyVersion>$(Version).0</AssemblyVersion>
</PropertyGroup>
```

💡 Consider using DotNetAnalyzers.DocumentationAnalyzers in your C# projects

Why? The DotNetAnalyzers.DocumentationAnalyzers will help you to enforce a certain structure of all public classes, methods and properties in your project. Consider this packages when you write a reusable library, so that other developers know how to use your code.

Add the following block to each *.csproj file.

```
<PropertyGroup>
    <TreatWarningsAsErrors>true</TreatWarningsAsErrors>
    <GenerateDocumentationFile>true</GenerateDocumentationFile>
</PropertyGroup>

<PackageReference Include="DotNetAnalyzers.DocumentationAnalyzers" Version="1.0.0-beta.59">
      <PrivateAssets>all</PrivateAssets>
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
</PackageReference>
```