# Wasabi Coding Conventions

## CodeMaid

\*\*DO\*\* use [CodeMaid](http://www.codemaid.net/), a Visual Studio extension to automatically clean up your code on saving the file.

CodeMaid is a non-intrusive code cleanup tool.

Wasabi's CodeMaid settings [can be found in the root of the repository](https://github.com/zkSNACKs/WalletWasabi/blob/master/CodeMaid.config). They are automatically picked up by Visual Studio when you open the project, assuming the CodeMaid extension is installed. Unfortunately CodeMaid has no Visual Studio Code extension yet. You can check out the progress on this [under this GitHub issue](https://github.com/codecadwallader/codemaid/issues/273).

## .editorconfig

Not only CodeMaid, but Visual Studio also enforces consistent coding style through [`.editorconfig`](https://github.com/zkSNACKs/WalletWasabi/blob/master/.editorconfig) file.

If you are using Visual Studio Code, the built-in solution-level [`omnisharp.json`](https://github.com/zkSNACKs/WalletWasabi/blob/master/omnisharp.json) configuration file will be automatically used.

In order to enforce the code style, just do a `Format Document` command by pressing `Shift` + `Alt` + `F` on Windows, `Shift` + `Option` + `F` on Mac, or `Ctrl` + `Shift` + `I` on Linux.

## Refactoring

If you are a new contributor \*\*DO\*\* keep refactoring pull requests short, uncomplex and easy to verify. It requires a certain level of experience to know where the code belongs to and to understand the full ramification (including rebase effort of open pull requests) - [source](https://github.com/bitcoin/bitcoin/blob/master/CONTRIBUTING.md#refactoring).

## Comments

\*\*DO\*\* follow [Microsoft's C# commenting conventions](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/inside-a-program/coding-conventions#commenting-conventions).

- Place the comment on a separate line, not at the end of a line of code.

- Begin comment text with an uppercase letter.

- End comment text with a period.

- Insert one space between the comment delimiter (`//`) and the comment text, as shown in the following example.

- Do not create formatted blocks of asterisks around comments.

```cs

// The following declaration creates a query. It does not run

// the query.

```

## Asynchronous Locking

\*\*DO NOT\*\* mix awaitable and non-awaitable locks.

```cs

// GOOD

private AsyncLock AsyncLock { get; } = new AsyncLock();

using (await AsyncLock.LockAsync())

{

...

}

// GOOD

private object Lock { get; } = new object();

lock (Lock)

{

...

}

// BAD

using (AsyncLock.Lock())

{

...

}

```

## Null Check

\*\*DO\*\* use `is null` instead of `== null`. It was a performance consideration in the past but from C# 7.0 it does not matter anymore, today we use this convention to keep our code consistent.

```cs

if (foo is null) return;

```

## Empty quotes

\*\*DO\*\* use `""` instead of `string.Empty` for consistency.

```cs

if (foo is null)

{

return "";

}

```

## Blocking

\*\*DO NOT\*\* block with `.Result, .Wait(), .GetAwaiter().GetResult()`. Never.

```cs

// BAD

IoHelpers.DeleteRecursivelyWithMagicDustAsync(Folder).GetAwaiter().GetResult();

```

## `async void`

\*\*DO NOT\*\* `async void`, except for event subscriptions. `async Task` instead.

\*\*DO\*\* `try catch` in `async void`, otherwise the software can crash.

```cs

{

MyClass.SomethingHappened += MyClass\_OnSomethingHappenedAsync;

}

// GOOD

private async void Synchronizer\_ResponseArrivedAsync(object sender, EventArgs e)

{

try

{

await FooAsync();

}

catch (Exception ex)

{

Logger.LogError<MyClass2>(ex);

}

}

```

## `ConfigureAwait(false)`

Basically every async library method that does not touch the UI should use `ConfigureAwait(false)`

```cs

await MyMethodAsync().ConfigureAwait(false);

```

## Disposing Subscriptions in ReactiveObjects

\*\*DO\*\* follow [ReactiveUI's Subscription Disposing Conventions](https://reactiveui.net/docs/guidelines/framework/dispose-your-subscriptions).

\*\*DO\*\* dispose your subscription if you are referencing another object. \*\*DO\*\* use the `.DisposeWith()` method.

```cs

Observable.FromEventPattern(...)

.ObserveOn(RxApp.MainThreadScheduler)

.Subscribe(...)

.DisposeWith(Disposables);

```

\*\*DO NOT\*\* dispose your subscription if a component exposes an event and also subscribes to it itself. That's because the subscription is manifested as the component having a reference to itself. Same is true with Rx. If you're a VM and you e.g. WhenAnyValue against your own property, there's no need to clean that up because that is manifested as the VM having a reference to itself.

```cs

this.WhenAnyValue(...)

.ObserveOn(RxApp.MainThreadScheduler)

.Subscribe(...);

```

## ObservableAsPropertyHelpers Over Properties

\*\*DO\*\* follow [ReactiveUI's Oaph Over Properties Principle](https://reactiveui.net/docs/guidelines/framework/prefer-oaph-over-properties).

\*\*DO\*\* use `ObservableAsPropertyHelper` with `WhenAny` when a property's value depends on another property, a set of properties, or an observable stream, rather than set the value explicitly.

```cs

public class RepositoryViewModel : ReactiveObject

{

private ObservableAsPropertyHelper<bool> \_canDoIt;

public RepositoryViewModel()

{

\_canDoIt = this.WhenAnyValue(...)

.ToProperty(this, x => x.CanDoIt, scheduler: RxApp.MainThreadScheduler);

}

public bool CanDoIt => \_canDoIt?.Value ?? false;

}

```

\*\*DO\*\* always subscribe to these `ObservableAsPropertyHelper`s after their initialization is done.

## No code in Code-behind files (.xaml.cs)

All the logic should go into `ViewModels` or `Behaviors`.

## Main MUST be Synchronous

For Avalonia applications the Main method must be synchronous. No async-await here! If you await inside Main before Avalonia has initialised its renderloop / UIThread, then OSX will stop working. Why? Because OSX applications (and some of Unixes) assume that the very first thread created in a program is the UIThread. Cocoa apis check this and crash if they are not called from Thread 0. Awaiting means that Avalonia may not be able to capture Thread 0 for the UIThread.

## Avoid Binding expressions with SubProperties

If you have a `Binding` expression i.e. `{Binding MyProperty.ChildProperty}` then most likely the UI design is flawed and you have broken the MVVM pattern.

This kind of Binding demonstrates that your View is dependent not on just 1 ViewModel, but multiple Viewmodels and a very specific relationship between them.

If you find yourself having to do this, please re-think the UI design. To follow the MVVM pattern correctly to ensure the UI remains maintainable and testable then we should have a 1-1 view, viewmodel relationship. That means every view should only depend on a single viewmodel.

## Familiarise yourself with MVVM Pattern

It is very important for us to follow the MVVM pattern in UI code. Whenever difficulties arise in refactoring the UI or adding new UI features its usually where we have ventured from this path.

Some pointers on how to recognise if we are breaking MVVM:

\* Putting code in .xaml.cs (code-behind files)

\* Putting buisness logic inside control code

\* Views that depend on more than 1 viewmodel class.

If it seems not possible to implement something without breaking some of this advice please consult with @danwalmsley.