Let me start this by describing, how I'd like to program.

To me this is an important principle of good programming. If your code is composed of small blocks, that contain little code, they are likely also only coupled to the interfaces (or classes), they actually need.

When writing a class in C#, we generally write the same style of code multiple times in multiple methods and properties:

* Range check for input variables
* Set/get fields values in properties
* The 'INotifyPropertyChanged'-pattern
* Returning an invalid value (some cases a 'magic number')
* Doing different calculations or returning values based on conditions
* External framework code, for example a database framework. This often requires same code style for all methods/properties

To avoid doing all this code repeatable, programmers often use tricks to avoid writing the same code repeatedly:

* Auto-generated code (like Resharper or code snippets)
* Post compilation injected code
* A common base class containing helper functions/fields
* Reflection-based code (like serialization)

I'm not saying it's always a bad idea to use these tools, but they all have major drawback and pitfalls, if not used correctly. All the methods work well, but you end up spending a lot of time of the cases, that don't fit the problem, they were trying to solve.

I think auto-generated code in most cases have fewer bugs that human written code. For the tools that generate the code lines for you, the logic is quite strange. For example, you are telling the tool: I want a property of this type, with a field connected to it, with a notification on change, and you are doing it again for a different property, and again, and again …

The post-compilation and reflection-based hide what’s actually going on, and only solve specific problem is a very restricted way.

So, to reverse things a bit, here is how I'd like my class to be written:

class Person : IPerson

{

string FirstName

get

.Getter()

.XmlEntry(”FirstName”);

set

.ValidityCheck(name => !IsNullOrEmpty(name))

.SkipIfEqual()

.Setter()

.NotifyChanged()

.XmlEntry(”FirstName”);

}

In addition, I would like all new properties that are added to the interface to be auto-implemented as I specify, and without any changes to my code file.

I can always dream …

Anyway, there are some possibilities if the class is generated runtime using the Castle package.

So we lose control over the actual code, everything will have to go through the composer.

Introducing an IInterceptor and helper classes for generating properties and methods, I can get closer to my dream.

So the code would look like this:

IPerson PersonFactory()

{

return factory.Generate<IPerson>(

Property<string>(”FirstName”)

. ValidityCheck(name => !IsNullOrEmpty(name))

.SkipIfEqual()

.SetterGetter()

.NotifyChanged()

.XmlEntry(”FirstName”));

}

Or defining a frequently used composition:

private static Property CreateMvvmStringProperty(string propertyName)

{

return new Property<string>(propertyName)

.ValidityCheck(name => !string.IsNullOrEmpty(name))

.SkipIfEqual()

.SetterGetter()

.NotifyChanged()

.XmlEntry(propertyName);

}

And use it:

IPerson PersonFactory()

{

return factory.Generate<IPerson>(

CreateMvvmStringProperty(”FirstName”));

}