Type Extension Skyline

A novel way to see how good or bad your extensions are in C#

One of the principles of object-oriented programming is to keep designs "open *for extension but closed for modification."* C# allows a functionality called extension function which allows users to add new functionality to existing types without modifying the code for the type. This is great because it allows users/programmers to extend a type beyond its initial design. However, there are some disadvantages. Primarily, if the extensions outgrow the number of functionalities (read functions/methods) in the original type, then it means that the type design is not well-thought-out, and a redesign by bringing these extensions inside the original implementations will be helpful.

Although extension functions are great to extend the functionalities of a type, sometimes people avoid them for sane and insane reasons. I have heard many. Here are some of the reasons I have come across thus far.

- Bob (a thought-of name!) wrote it, and Bob's nickname in the company is 'GOD.' So we don't touch it or dare to extend it. (*Insane reason*)
- It will break the backward compatibility. (Phew! New additional things can never break existing functionality. *(insane reason*)
- We are a prehistoric .NET shop, and we use a .NET version that doesn't support it yet. (*Come on!*)
- We can have ambiguous situations, although caught at compile time, but still a nuance we want to avoid. (Valid reason)

However, if you or your team chooses to use extension methods, then there is a possibility to get dragged and overdo it. It's fun! So you may overdo it without realizing it. But creative data visualization techniques can help you sniff out such detours early to help you stay on course.

The novel visualization

The Idea:

If we plot the number of member methods exposed by a type and the number of extension functions it has, then comparing these numbers side by side or on top of each other can create a nice visualization. If we map the number of exposed/public methods of a type to the height of a rectangle and if we map the width of the rectangle as the average width of the lines in the method bodies, then such a rectangle can represent the exposed surface area of a given type.

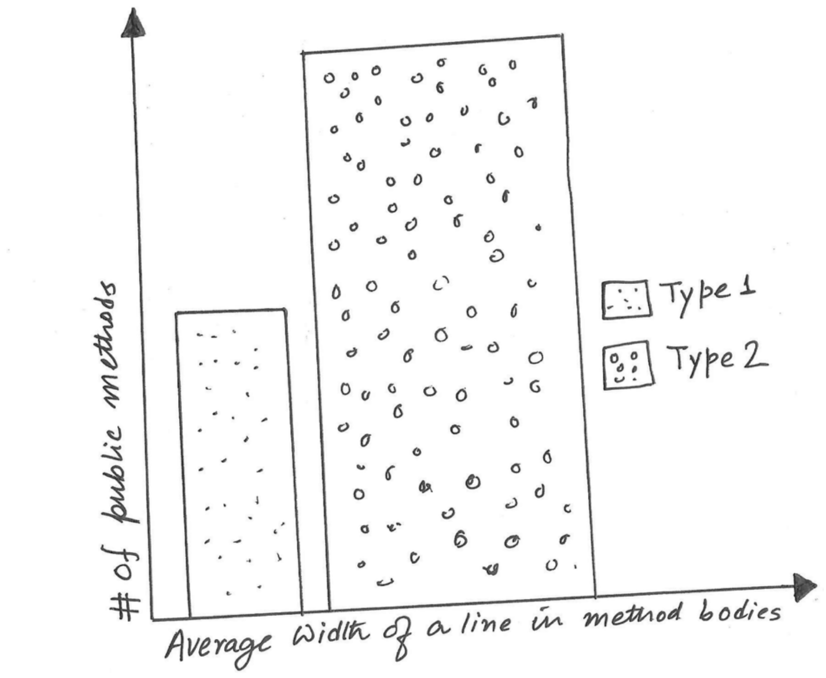

Figure 1. Representing public methods of a type as a rectangle

Extrapolating the idea

If we carry out the same experiment for all other types that extend a given type, we shall eventually be able to get another rectangle that depicts the size of the extensions of the given type. Then, if we plot these rectangles side-by-side, we shall get a picture like this.

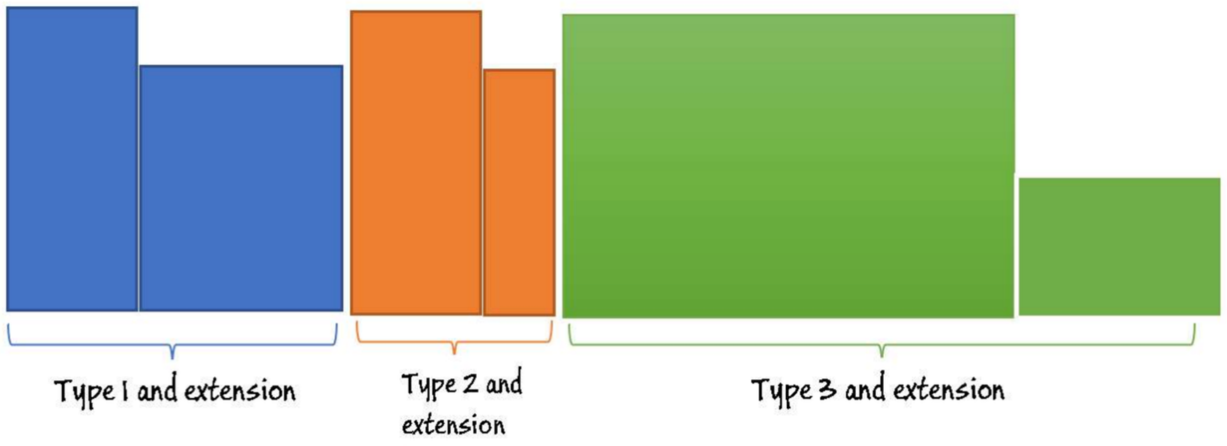

Figure 2. Types and their extensions represented as rectangular boxes.

Glimpse of the skyline

Just to make the plot a bit more like a real skyline of a city, we can add some caps to these rectangles. For example, the rectangles depicting the methods of the given type can have a triangular cap, and the boxes presenting the extensions can have a trapezium cap. This will look like the following figure.

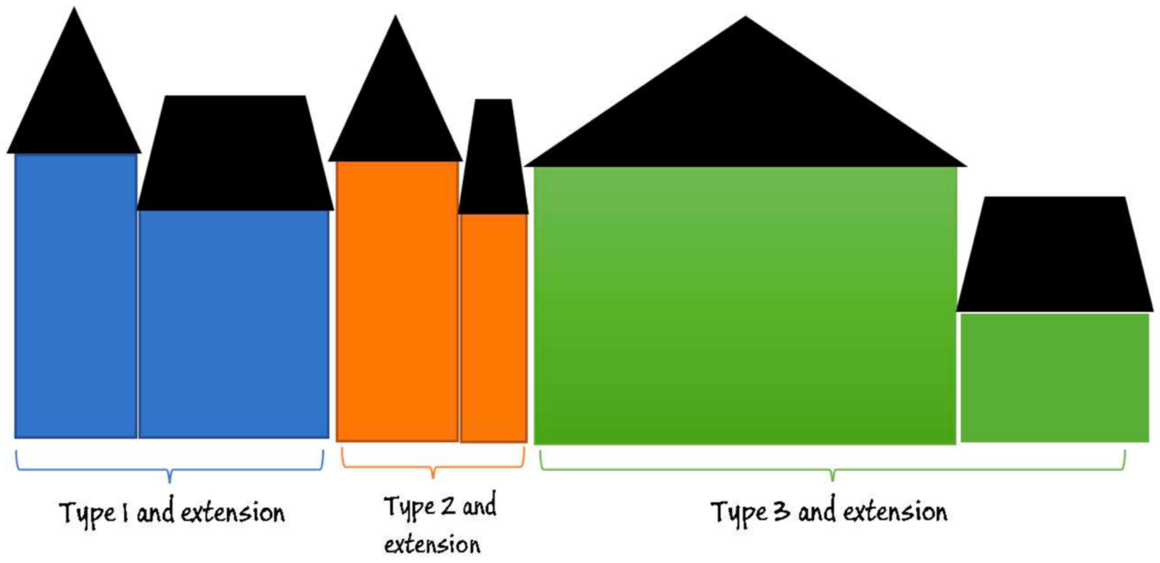

Figure 3. Skyline with types and their extensions representing buildings.

Since the types can be in different namespaces, if we want to see how the namespaces are clouded with extension methods, we can do this experiment for types in different namespaces and plot them together like this.

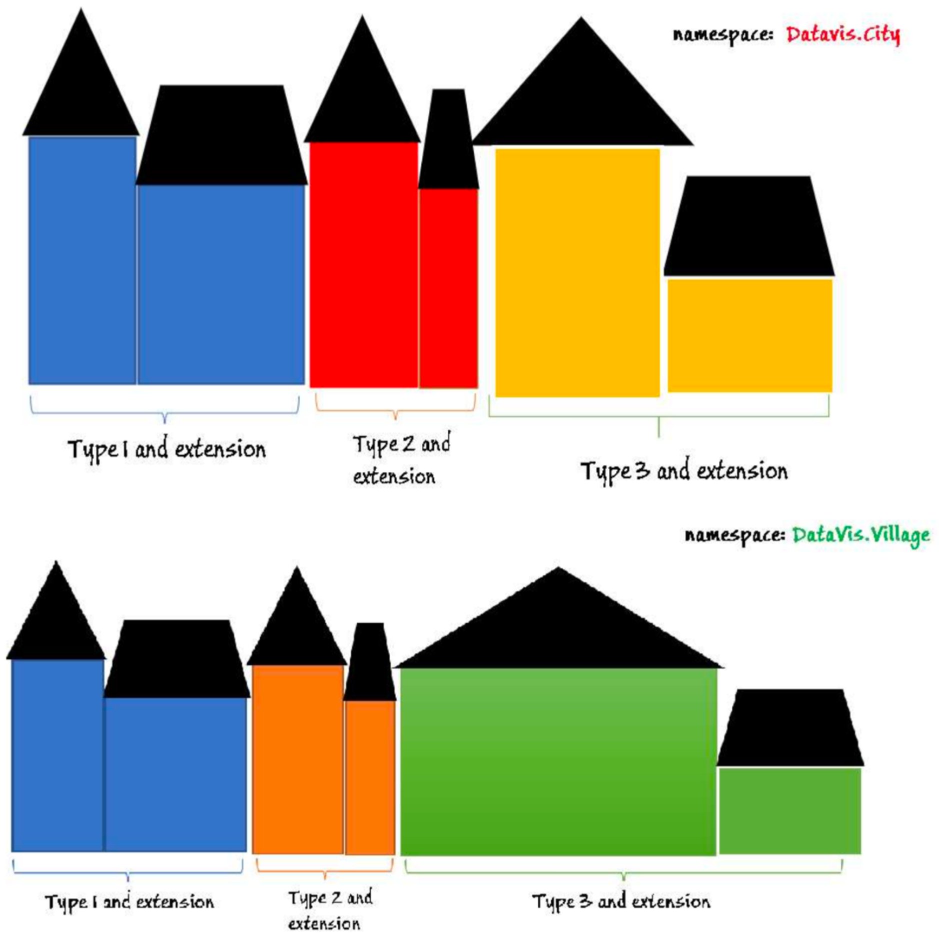

Figure 4. Skyline per namespaces

Interpreting the Visualization

As you can imagine, having a wider extension block to the side of the original rectangle representing the type is a bad signal. Because if the size of the extension grows bigger than the original type, then it is absolutely the best idea to bring the extension into the original type. Agreed that it may not always be possible because the type may be sealed or from the framework. But this is generally a good way to detect bad designs early.

So, Type 3 is good, but Type 1 can be improved by pulling some things from extension.