## Why a Businessrule common code library?

A program is used inside of a business, in order to facilitate a process by enforcing some rules.  
For example, a fish market can sell tuna at the wholesale price plus 20 percent.

It is possible to write a program in a file that can sell any fish according to any rule(s).  
However, that job will be repeated for all processes with the same, or possibly some other rules.

This creates a group of programs that are difficult to to understand, even if they use the same rules.  
It is difficult because each rule is getting written again with new words and perspectives continuously.

A great solution to that confusion is to create a single separate program which is the only place where such a rule is actually written. Anybody could then use that single version by simply giving all the unique parts to their process via the entrance to that program, and then continue with the returns.

Like that, we stop writing things that are the same for everybody who wants to follow a rule for their program. We also make a promise to do things one way, this makes anything easier to understand.

Our change in writing programs has now helped us and any future programs to be shorter and easier to understand. Moving things in one place also opened up new possibilities to do expansions easier.

## Type-erased version of Interfaces:

Inspect code:  
public interface IBusinessRule : IBusinessRuleWithIdentifierString

{

IBusinessRuleEvaluationStrategy EvaluationStrategy { get; }

}

public interface IBusinessRuleEvaluationStrategy

{

}

public interface IBusinessRuleEvaluationStrategy<TInput, TOutput> : IBusinessRuleEvaluationStrategy

{

TOutput ExecuteStrategy(TInput input);

}

Interfaces are important members of the buildings blocks that make up an application.  
They promise other programs that something they have has certain properties or features.

This means that a small program is able to use hundreds of differing things without having to know their names and specifics, it just needs to know if something can promise to have what it needs.

However, sometimes there are two programs that rely upon the same interface to work with hundreds of differing things as long as they promise to have certain properties and features.  
But, both programs do not require the same level of information about those properties and features.

In order to solve that problem, interfaces can be split into two versions wherein one provides details about everything that someone has to promise to be there, and another only provides few details.

A program can now choose to treat hundreds of things equally, as long as they promise to share common things, and another can do the same without focusing on details about what’s shared.

This is basically a set of two answer to two different situations:

Imagine that there’s a business rule in which a business rule execution strategy lies.  
Whenever someone creates such a business rule, they replace it by the accompanying interface.  
Like this, any program can keep using any varying business rules under the same interface name.

However, business rules can also be created by a factory as opposed to creation where needed.  
Whereas we do know of the type of an execution strategy at runtime, we do not known this at compiletime inside of the factory.

So, in order to still be able to ensure that any program can keep using any varying business rules under the same interface name we duplicate the interface but without the type of the strategy.

We also make the type-erased interface inherit from the actual interface.  
Like this, our factory can talk about business rules that have a strategy without having to specifiy the type arguments for that strategy which only the user of the factory knows of.

Another program could also retrieve the toys for each animal