JJ's Reference Architecture

Author: Jan-Joost van Zon
Date: December 2014 – July 2017
[Under Construction]

Namespaces, Assemblies and Folder Structure

Contents

General Structure

Solution files are put in the code root.

Assembly names, namespaces and folder structure are similar to eachother. An assembly's name will be its root namespace. The folder structure will also correspond to the namespacing.

An assembly name is built up as follows:

Company.SoftwareLayer.BusinessDomain [.Technology] [.Test]

Internally in an assembly each pattern can get its own sub-folder:

Company.SoftwareLayer.BusinessDomain [.Technology] [.Test] [.DesignPattern]

If a project is very small, you might use a single sub-folder 'Helpers', instead of a folder for each design pattern.
When a project gets big, a design pattern folder might again be split up into partial domains or main entities:

Company.SoftwareLayer.BusinessDomain [.Technology] [.Test] [.DesignPattern] [.PartialDomain]

Root Namespace / Company Name

In this architecture the root namespace will be the 'company name', for instance:

JJ

Main Layers

The second level in the namespacing consists of the following parts:

JJ.**Data**	The data layer including the entity models and persistence.
JJ.**Business**	The business logic layer
JJ.**Presentation**	The presentation layer
JJ.**Framework**	Contains any reusable code, that is independent from any domain model. Any layer in the software architecture can have reusable framework code to support it.

And the less important:

JJ.**Demos**	Demo code for educational purposes
JJ.**Utilities**	Processes that are not run very often. Utilities contains small programs for IT. E.g. load translations, things to run for deployment.

Business Domains

The third level in the namespacing is the business domain. A business domain can be present in multiple layers, or missing in a specific layer, an app can use multiple business domains, a single business domain can have multiple front-ends. Examples:

JJ.Data.**Calendar**
JJ.Business.**Calendar**
JJ.Presentation.**Calendar**

The 'business domain' of the framework layer is usually a technical aspect. Examples:

JJ.Framework.**Validation**
JJ.Framework.**Security**
JJ.Framework.**Logging**

Technologies

The fourth level in the namespacing denotes the used technology. It is kind of analogus to a file extension. You can often find two assemblies: platform-independent and one platform-specific.

JJ.Data.Calendar
JJ.Data.Calendar.**NHibernate**

JJ.Presentation.Calendar
JJ.Presentation.Calendar.**Mvc**

JJ.Framework.Logging
JJ.Framework.Logging.**DebugOutput**

This means that the platform-indepent part of the code is separate from the platform-specific code. This also means, that much of the code is shared between platforms. It also means, that we can be very specific about which technologies we want to be dependent on.

Test Projects

Every assembly can get a Test assembly, which contains unit tests. For instance:

JJ.Business.Calendar.**Tests**
JJ.Presentation.Calendar.Mvc.**Tests**

Details

One Class, One File

The general rule is that all classes, interfaces, enums, etc. get their own file. The rule can be broken if the amount of classes really becomes big and also the rule does not count for nested classes. Also a single class can be spread among files, if they are partial classes.

Lone Classes (bad)

It is unhandy to have a whole bunch of your assembly's folders just containing one class or very few classes. Consider moving those classes into other folders. Another solution could be to put them all together, for instance in a folder called 'Helpers', if they indeed are just simple helper classes.

'Scramled' Technical and Functional Concerns

In our namespacing, the technical and functional pieces seem scrambled:

JJ.Business.Ordering.Validation.Products

These are the functional (or commercial) concerns:

JJ.Business.**Ordering**.Validation.**Products**

These are the technical concerns:

JJ.**Business**.Ordering.**Validation**.Products

The reason for 'scrambling' of technical and functional concerns, is rooted in that we are trying to project something 2-dimensional (funtional vs. technical) onto something sequential (written text). We could artificially keep functionality together and technical things together:

JJ.Ordering.Products.Business.NHibernate.Validation
JJ.Ordering.Products.***Business.NHibernate.Validation***

But this does not help us do our job.

What we instead try to do is organize things into bigger and smaller chunks. The split up into companies' intellectual property is the largest concern, while the second most important concern is the split up into main software layers (Data, Business, Presentation, etc.) A business

domain is a larger concern than the specific technology used (e.g. NHibernate, Mvc). And a design pattern is a level of detail even below that.

Also: in the less recommended namespacing it is not obvious that JJ.Ordering.Products is about validating the products, while if you put the Products sub-namespace at the end (…Validation.Products) this is obvious.

Another problem with starting with JJ.Ordering.Products instead of JJ.Business is that it suggests that Ordering has a Data, Business and Presentation layer, while really Ordering does not need to be present in all layers. It gives a false sense that you create a Presentation layer it must be put into an already existing business domain or that a business domain must always have al three layers present. It would also suggests that a presentation layer in one business domain can only use one business layer. The reality is, that a presentation layer can use multiple business layers.

It is less confusing from a software design perspective to have all layers present and whether a business domain is present in a layer is optional. That makes it more obvious that there is an n-to-n relationship between layers and business domains.

But the ordering in the namespace is arbitrary. It just helps us to 'scramble' the namespace parts wisely, so that it goes from one level of detail to the next.