JJ's Reference Architecture

Part 2

-

Author: Jan-Joost van Zon
Date: December 2014 – June 2017
Under Construction

Introduction

Definition of Software Architecture

Software architecture has thousands of definitions. If you believe them all, then software architecture is about everthing that has to do with software development.

Mostly it is about actually building the software.

It is about modular design of software components, which can be done by splitting up code into functionalities, like a shopping basket, or financial reports, or splitting up code into technical aspects, like validation, security and persistence. You can also split up by both technical aspects and functional aspects at the same time, giving you a more fine-grained (2-dimensional!) separation of concerns.

The programming side of software architecture is about making frameworks, coding the functionalities, combining different technologies and using best practices. It also involves technical design, which by the way you can do in your head as well as on paper.

Inter-Disciplinary Aspects

Software architecture also involves technical details outside of software programming, such as the basic outlining of hardware infrastructure, collaborating with infrastructure technicians, server administrators, hosting providers, software vendors.

Software architecture also includes soft-skills that do not have much to do with technology. Planning the development of software both in rough outlines as well as task details, guarding that work, prioritizing, organizing and replanning, making concessions, work preparation, managing software lifecycle, going from design to implementation to test to production and after care, having proper source control in place, managing the team that codes, the team that tests, discussing functionalities, goals and planning with management, stakeholders, staff and end-users. Basically talking to anyone even slightly involved in the development of the software. Coaching developers, expanding the teams knowledge, making the team work optimally together and efficiently, and give people room to focus, so a lot of work gets done well. It can involve managing budgets for hardware and software and also functional designing.

Fortunately this does not need to come down to one person. Even though a software architect can overview the whole process, lots of tasks can be delegated to other team members, so you can make software architecture work as a team.

From Definite Choices to Possible Choices

This is kind of a personal note on where this document stands right now.

Originally I described a fixed way of working here, that generally works well if you want to build large dependable systems with a lot of flexibility. I applied these methods of working in a team under my lead. It worked, but required a lot of discipline of team members to do things the way the boss wants.

I want to move away from this a little bit, and see the methods described here more like a suggestion box of different ways to do things. I will try to describe different alternatives next to the one I prefer and highlight the pros and cons, so you can perhaps see why I came to the conclusion that one method is better than the other.

Much of the document is still described in definites, rather than suggestions. As I find the time to work on this documentation, I will be changing the tone.

What you will also find is that I describe a lot of things you could do wrong. The suggestion is often that there is a better way to do it. I will try to reformulate things so they start with a positive approach rather than starting with the negative.

Currently (2017-06-28) it is full of TODO's that indicate texts I still want to write or rough texts to polish up. So please be forgiving of those.

But, now: back to business.

The Technical Approach

This document mostly goes into detail about technical aspects of software architecture. It talks about those aspect of software development, that go beyond the individual application: techniques and best practices that can be applied to the development of *any* application. You could call it '*functionality-independent software architecture*'.

This architecture could also be called a '*pattern stack*', because it takes a technical-first approach, rather than functional-first in that a layering is described where fixed design patterns are used from one layer to the next. The rationale behind this is that even though technology changes fast, functionality changes faster, so a technical-first order will be less likely to change, and this results in a more stable subdivision into parts upon which the functionality builds.

<TODO: Use some of these phrasings: Preservability of functionality, versus technique. Functionality changes even faster than the technology itself. That is one of the reasons why the code has a technical-first ordering, not functional-first. And to accommodate quickly changing technology, we use abstraction of these technologies to be able to replace them and not have to reprogram the whole application. >

<TODO: Possibly Use this phrase about extensibility: (Dutch)“Alles *past* erin, ook al zit het er nog niet in.” “Everything fits in it, but not everything is in it yet.”>

There are two parts of this software architecture:

- Application architecture
- Service architecture

The *application architecture* is the main part. It is about business domains and everything you could show on a screen, including a framework of reusable parts. The *service architecture* is explained separately and is mostly about linking different systems together.

The way of working described here is just a suggestion. It describes *a* way of working, not *the* way of working. The described principles and practices can be used at will.

Fundamental Principles

The main principles of this software architecture are:

- Maintainability
- Code scalability
- Platform and protocol independence

Code scalability does not refer to hardware scalability, but rather that the code base can grow and grow, while keeping it maintainable. 50 apps should be as maintainable as 5. This means that quality demands are high. As a companies’ amount of software products grows, software architecture is necessary to create an economically viable amount of software maintenance, or a company might run into problems.

Another way of putting this is: The next software change should not be more difficult than the previous one, regardless of how large the system has become.

Platform and protocol independence is something given extra attention in this software architecture. A lot of split up into parts is, due to the fact that not every technology is supported on every platform. This allows us to take our pick from technologies more easily.

This software architecture also puts a lot of focus on fixed patterns of working. These patterns are proven to work well, and if we all work the same way and understand the system of organization, we can more easily navigate the code, regardless of who wrote it.

Top 12 Code Improvements

This document goes into detail about a lot of best practices. But to keep focus on what goes wrong most of the time, and would offer the greatest improvement of code, here is a list of practices, that if done right, may greatly increase the quality of your software.

1. Avoid code duplication
2. Avoid error hiding
3. Use clear names

4. Use correct indentation
5. Separation of concerns
6. Proper encapsulation

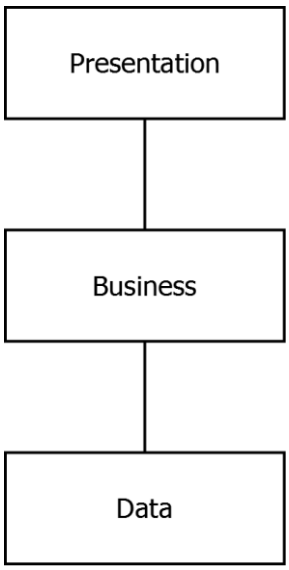
And second in line:

7. Reject, don't correct
8. Clear interfaces
9. 'No handy extras' / 'Ya ain't gonna need it'
10. Solve a problem at its core, instead of making work-arounds.
11. To solve a bug, first reproduce it
12. Proper use of encoding

Layers

This is a suggestion of how to split up your software into layers.

The software is split up into 3 layers:



The presentation layer contains the screens of the system.

The presentation layer calls the business layer, which is non-visual. It defines and enforces the rules of the system. Those are like the internal, mechanical parts of the system.

The business layer talks to the data layer, which models the business domain but does not process anything: it just stores and retrieves the data.

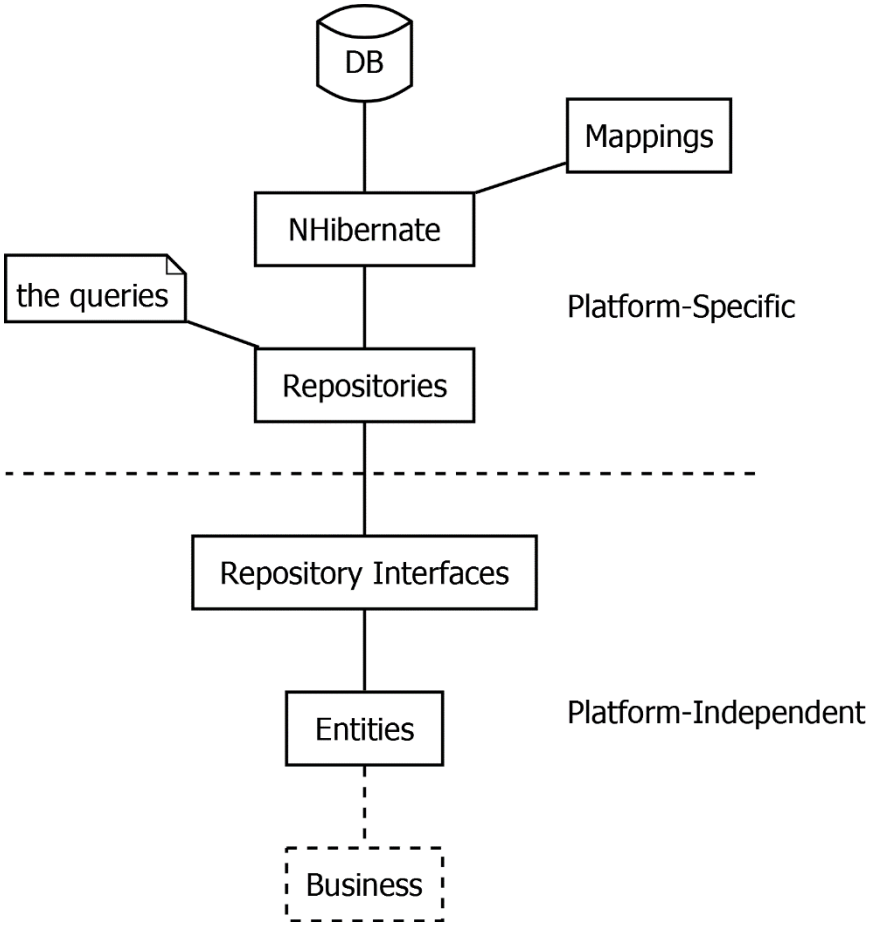
Data layer and presentation layer are programmed using fixed patterns. The business layer uses patterns too, but it gets a little more creative. If anything special needs to happen, this belongs in the business layer, since that is the machinery of the system.

The data layer is also called the 'data access layer' or 'persistence layer'.
The business layer is also referred to as 'business logic'.

The presentation layer is sometimes referred to as the ‘front-end’.

Data Layer

The data layer is built up of the following sub-layers:



It all starts with the database. The database is not directly accessed by the rest of the code, but the database is talked to through NHibernate, an object-relational mapper. NHibernate will translate database records to instance of classes. Those classes have properties, that map to columns in the database, and properties that point to related data. NHibernate needs to be given mappings, that define which class maps to which table and which columns map to which properties.

The data classes are called entities.

The entities are not directly read out of NHibernate by the rest of the code. The rest of the code talks to NHibernate through the repositories. You can see the repositories as a set of queries. Next to providing a central place to manage a set of optimal queries, the repositories also keep the rest of the code independent of NHibernate, in case you would ever want to switch to a different data storage technology.

The repository implementations are not used directly, but accessed through an interface, so that we can indeed use a different data access technology, just by instantiating a different repository implementation. The repository interfaces are also handy for testing, to create a fake in-memory data store, instead of connecting to a real database.

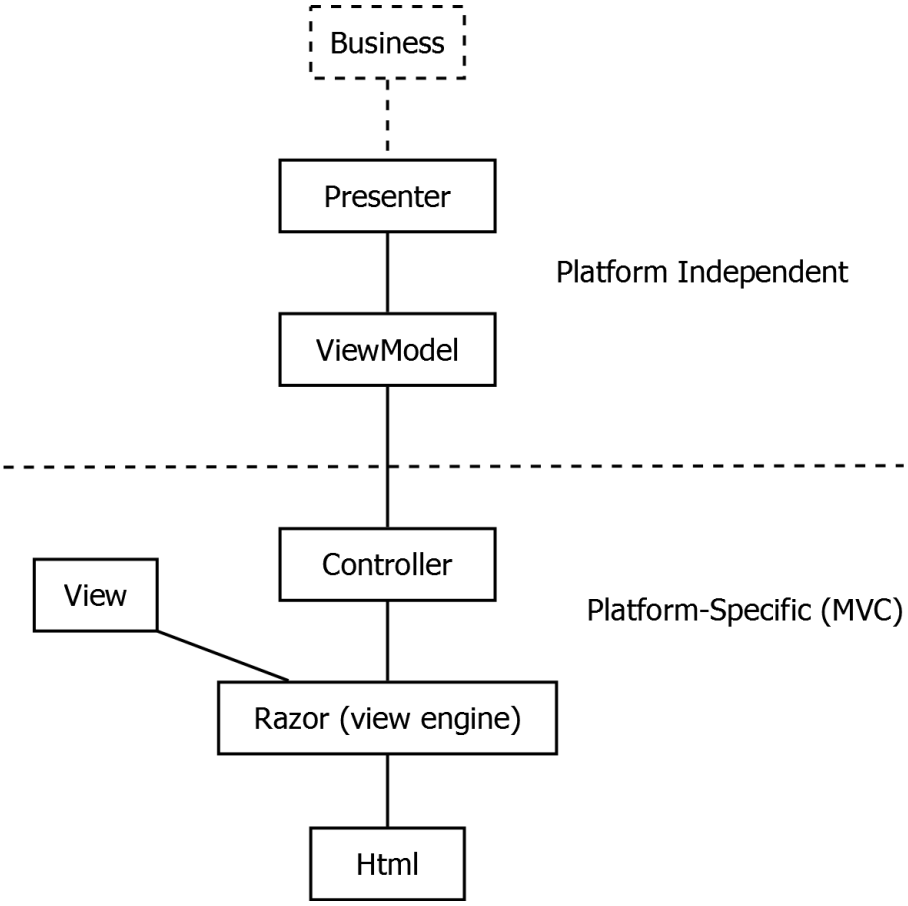
The dashed line going right through the diagram separates the platform-specific code from the platform independent code. The platform-specific code concerns itself with NHibernate and SQL Server, while the platform independent code is agnostic of what the underlying storage technology is. You may as well stick an XML file under it and not use SQL Server and NHibernate at all. This allows you to program against the same model, regardless of how you store it. This also allows you to deploy this code in any environment that can run .NET code, such as a mobile phone.

Because the architecture is multi-platform, the labels in the diagram are actually too specific:

- 'DB' can actually be any **data store** – that is the proper term for it: 'data store': an XML file, flat file or even just in-memory data.
- 'NHibernate' can be an any **persistence technology**: another 'ORM' ('object relational mapper'), like Entity Framework, a technology similar to NHibernate. The persistence technology can also be simply writing to the file system, or an XML API, or SqlClient with which you can execute raw SQL.

Presentation Layer

The presentation layer is built up of the following sub-layers:



<TODO: Put ToEntity en ToViewModel in the diagram.>

The presentation layer calls the business layer, which contains all the rules that surround the system.

The data that is exactly shown on screen is called the *view model*.

Presenter classes talk to the business layer. The presenter is responsible for translating the data and the results of the business logic to a subset of data that is shown on screen: the view model.

The presenter is also responsible for translating user input back to data, passing it through the business logic. The business logic then executes validations and side effects around the data access.

<TODO: Write about ToEntity / ToViewModel, etc., to indicate that the presenter is a combinator of things.>

The presenter layer forms a model of your program navigation. Each screen has its own presenter and each method in that presenter is a specific user action.

MVC is the web technology of choice we use for programming user interfaces. In our architecture the MVC layer builds on top of the presenter layer.

In MVC we use controllers, which are similar to presenters in that they group together related user actions and each user action has a specific method.

Each method in a controller represents a URL. The request from the web browser will lead to the controller method going off, which eventually results in a view engine rendering a piece of HTML. The view rendering goes off automatically. The view engine we use is Razor, which offers a concise syntax for programming views, in which you combine C# and HTML. Razor has tight integration with MVC. The view engine combines a view model and a view to form a specific piece of HTML.

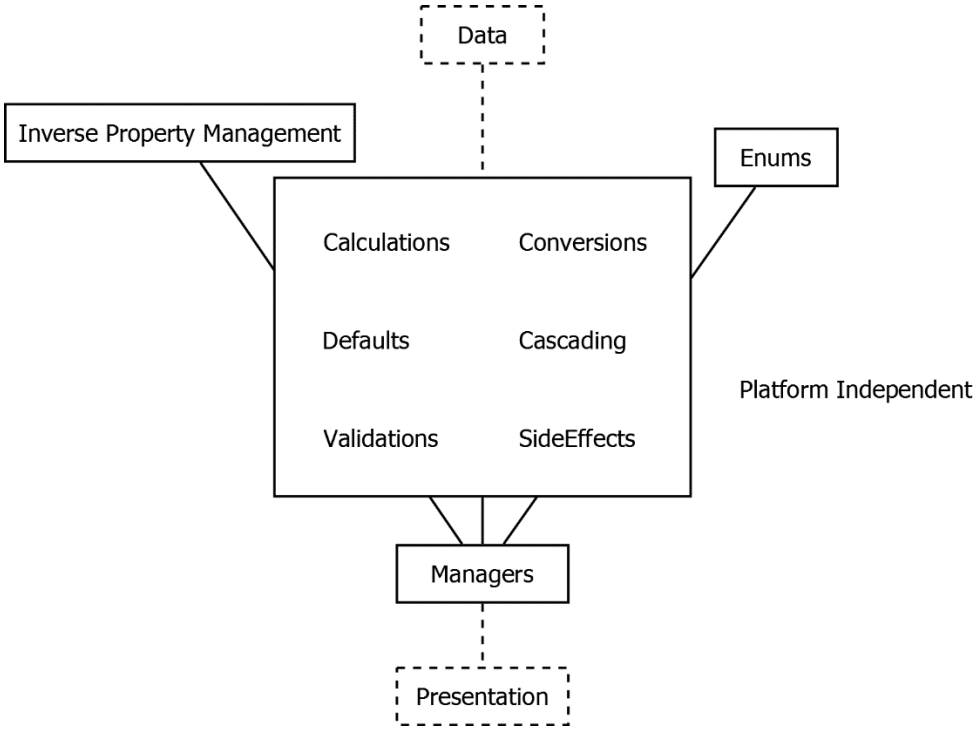
The dashed line going right through the diagram separates the platform-specific code from the platform independent code. The platform-specific code concerns itself with MVC, HTML and Razor, while the platform independent code is agnostic of what presentation technology we use. That means that we can use multiple presentation techniques for the same application navigation model, such as offering an application both web based as well as based on WinForms. This also provides us the flexibility we need to be able to deploy apps on mobile platforms using the same techniques as we would use for Windows or web.

Because the architecture is multi-platform, the labels in the diagram above are actually too specific:

- The *controller* is very specific to MVC and an equivalent might not even be present on other presentation platforms, even though it is advisable to have a central place to manage calls to the presenter and showing the right views depending on its result.
- The views in WinForms would be the *Forms and UserControls*. It is advised that even if a view can have ‘code-behind’ to only put dumb code in it and delegate the real work elsewhere.

- ‘Html’ can be replaced by the type of presentation output. In WinForms it is the controls you put on a form and their data. But it can also be a generated PDF, or anything that comes out of any presentation technology.

Business Layer



<TODO: Include ‘Resources’. >

What is business logic? Basically anything that is not presentation or data access, is business logic.

<TODO: Layers: Say something about infrastructure, next to persistence, business and presentation. Because then you can say: everything that is not persistence, presentation or infrastructure, is business logic.>

The business layer resides in between the data access and the presentation layer. The presentation layer calls the business layer for the most part throught the Manager classes. The manager classes are combinators that combine multiple aspects of the business logic, by calling validators, side effects, cascadings and other things. They are ‘CRUD-oriented facades’.

The business layer executes validations that verify, that the data corresponds to all the rules. Also, the business layer executes side effects when altering data, for instance storing the date time modified or setting default values when you create an entity, or for instance automatically generating a name. The business layer is also responsible for calculations and many other things as represented in the diagram above.

The business layer uses entities, but sometimes will call repositories out of the data access layer, even though your first choice should be to just use the entities. The presentation layer uses the business layer for anything special that needs to be done. Often when something

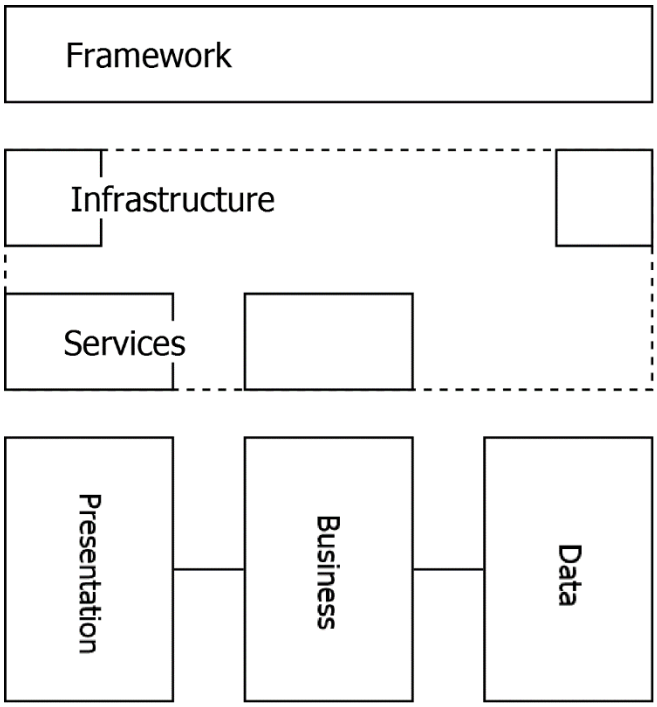
special is programmed in the presentation layer, it actually belongs in the business layer instead.

The business layer is platform independent and the code can be deployed anywhere. This does sometimes require specific API choices or using our own framework API's. These choices are inherently part of this architecture. But because most things are built on entities and repository interfaces, the business logic is very independent of everything else, which means that the magic of our software can be deployed anywhere.

<TODO: Add 'Cloning' to big block in the diagram? It might stay too vague if you mention it there. >
<TODO: Consider this:
- Mention in the layering diagrams that Inverse Property Management is also called LinkTo and Unlink in our architecture and that Cascading is also called UnlinkRelatedEntitiesExtensions and DeleteRelatedEntitiesExtensions. Whether you should pollute the diagrams with that is an open question, because it is a really specific choice that may be broken in the future. On the other hand, the diagrams serve to clarify and are specific to this architecture already.>

Perpendicular Layers

The subdivision into data, business and presentation is just about the most important subdivision in software design. But there are other additional layers, called perpendicular layers:



The Framework layer consists of API's that could support any aspect of software development, so could be used in any part of the layering. That is why it stretches right from Data to Presentation in the diagram.

Infrastructure is things like security, network connections and storage. The infrastructure can be seen as part at the outer end of the data layer and part at the outer end of the presentation layer, because the outer end of the data layer is actually performing the reading and writing from specific data source. However it is the presentation layer in which the final decision is made what the infrastructural context will be. The rest of the code operates independent of the infrastructure and only the top-level project determines what the context will be.

<TODO: Encorporate this phrase: It is hard to explain what the position of infrastructure is in the architecture. One thing you can say is that the infrastructure should be loose coupled. >

Services expose business logic through a network interface, often through the SOAP protocol. A service might also expose a presentation model to the outside world. Because it is about a specific network / communication protocol, the service layer is considered part of the infrastructure too.

Another funny thing about infrastructure, for example user right management, is that a program navigation model in the presenter layer can actually adapt itself to what rights the user has. In that respect the platform-independent presentation layer is dependent on the infrastructure, which is a paradox. The reason the presenter layer is platform-independent is that it communicates with the infrastructure using an interface, that may have a different implementation depending on the infrastructural context in which it runs.

Alternatives

	Benefits	Downsides
Data and Business in one layer	- Might be easier to understand	More likely for data access and business to get entangled
No repositories		

Namespaces, Assemblies and Folder Structure

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.