JJ's Reference Architecture

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

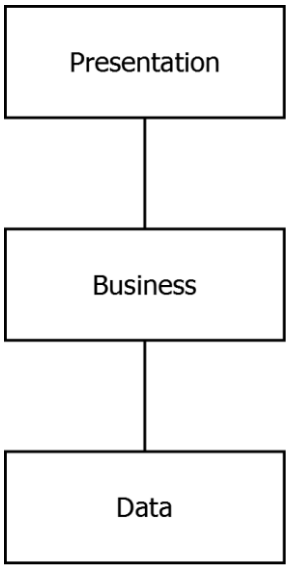
Layers

Contents

Introduction

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.

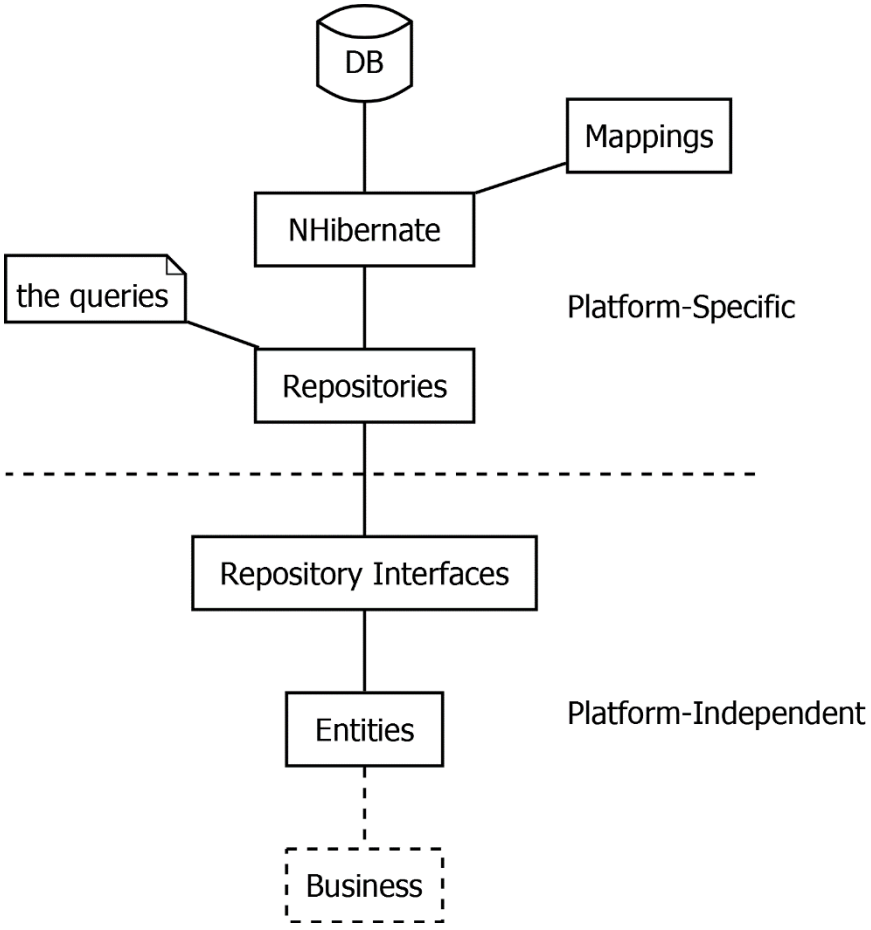
The business layer talks to the data layer, which models the business domain but does not process anything: it just stores and retrieves the data.

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.

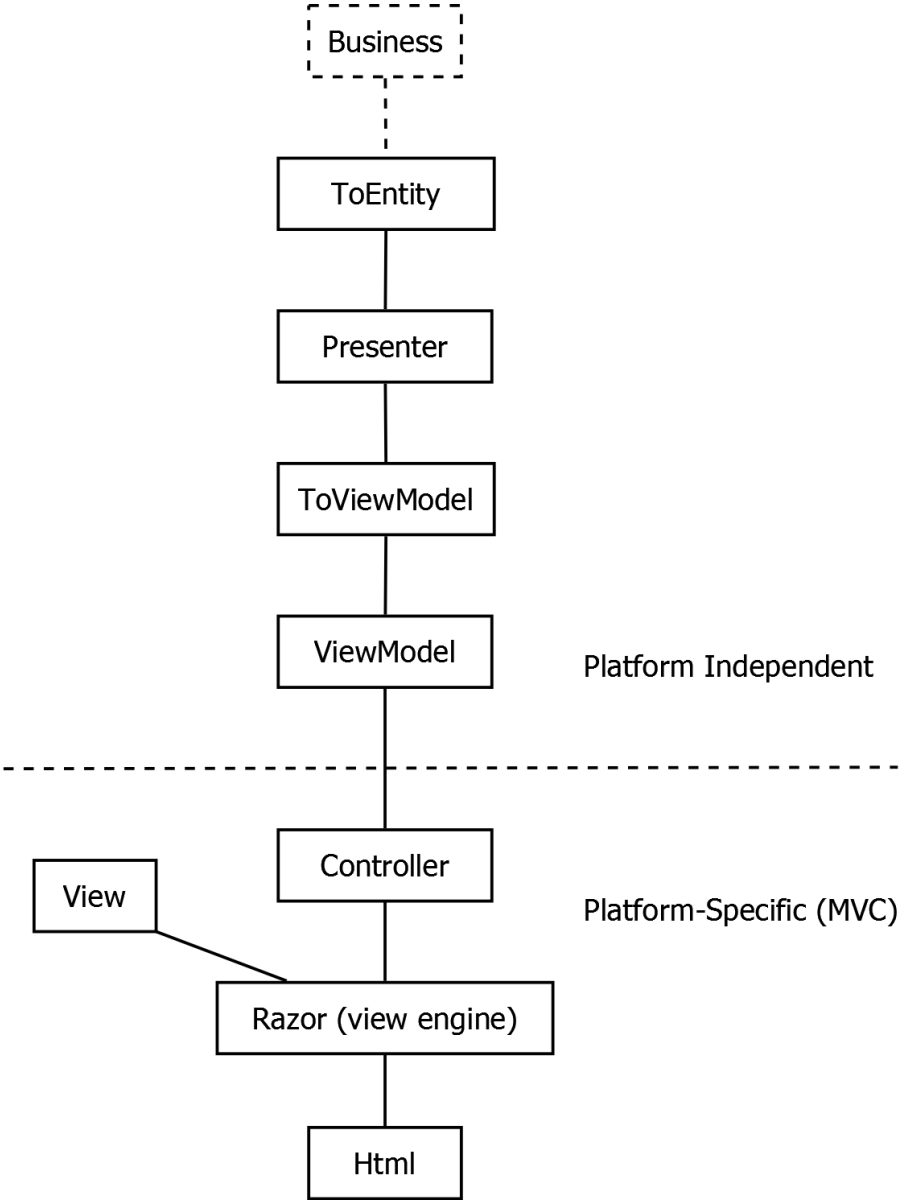
The dashed line going right through the diagram separates the platform-specific code from the platform independent code. The platform-specific code concerns itself with NHibernate and SQL Server, while the platform independent code is agnostic of what the underlying storage technology is. You may as 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: ToEntity is in a really odd spot if you read the diagram from top to bottom.>

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 combine several responsibilities around the presentation logic. I

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.

Presenter classes talk to the business layer.

A presenter delegates to the ToViewModel layer, to translating the data and the results of the business logic to a subset of data that is shown on screen: the view model.

A presenter delegates to the ToEntity layer, to translate user input back to entity data.

The presenter then calls upon the business layer again to save, validate, side-effects and execute other logic around the user action.

Because the presenters combine several responsibilities together they are the facades / combinators of the presentation layer.

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.

MVC will make sure that the request from the web browser will automatically make the right controller method going off. Each method in a controller represents a URL.

After the controller method is done, the view engine kicks in. It will render a piece of HTML. MVC will make sure that the view rendering automatically goes off after the controller method completes.

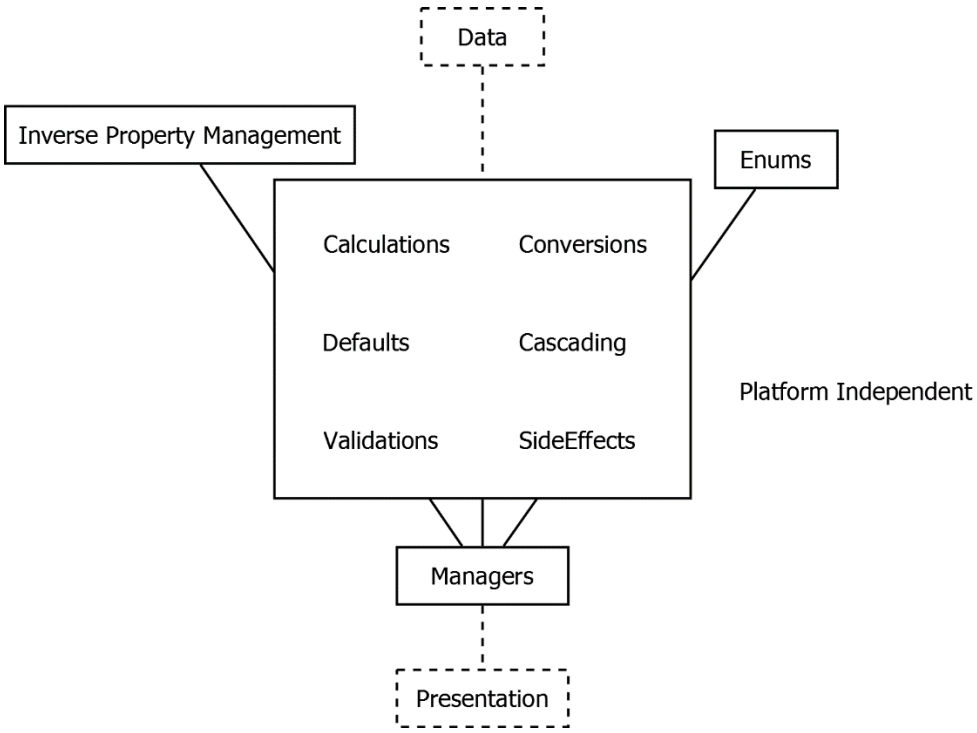
The view engine we use is Razor. It offers a concise syntax for programming views, in which you combine C# and HTML. Razor has tight integration with MVC. The view engine uses a view model as input, along with the view (template) and the output is a specific piece of HTML.

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: Keeping 'Inverse Property Management' and 'Enums' separate like that does not seem to add much to the clarity of the diagram. It is true that some aspects are used directly besides through the manager, but even those lines going from presentation directly to those aspects, are not present in the diagram, so that is kind of a fail. I think you could express that better.>

<TODO: Include 'Resources'. in the diagram >

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

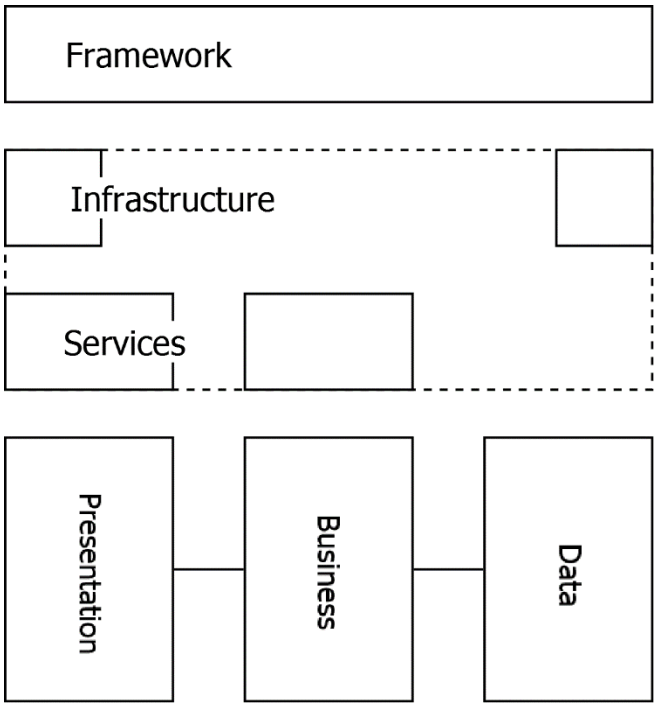
uses the business layer for anything special that needs to be done. Often when something special is programmed in the presentation layer, it actually belongs in the business layer instead.

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		