JJ Software Architecture

-

Author: Jan-Joost van Zon
Date: December 2014 – August 2016

Contents

Introduction

Software architecture involves those aspect of software development that go beyond the individual application: techniques and best practices that can be applied to any application.

This is a document that describes those software architectural rules.

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. The *service architecture* is explained separately and is mostly about linking different systems together.

Fundamental Principles

The main principles of this software architecture are:

- Maintainability
- Code scalability
- Platform 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.

Worst Practices

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 worst practices, that if solved, may greatly increase the quality of your software.

Worst practices:

1. Repeated code
2. Error hiding
3. Unclear names
4. Wrong indentation
5. No separation of concerns
6. No encapsulation

Almost worst practices:

7. Automatic correction of input data, instead of rejecting it.
8. Unclear interfaces
9. 'Handy extras'
10. Not solving the problem at the core, but working around it.
11. Trying to solve a bug without first reproducing it.
12. Bad use of encoding

Layers

The software should always be split up into 3 layers:
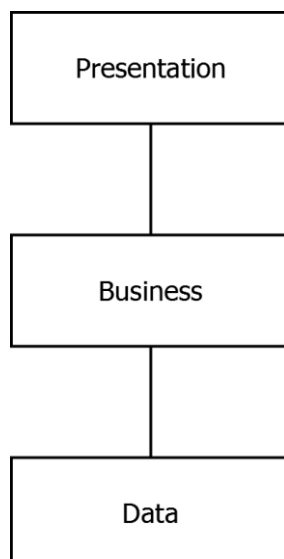
The presentation layer contains the screens of the system.

The presentation layer calls the business layer, which is non-visual code, which contains all the rules that surround 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 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.

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:
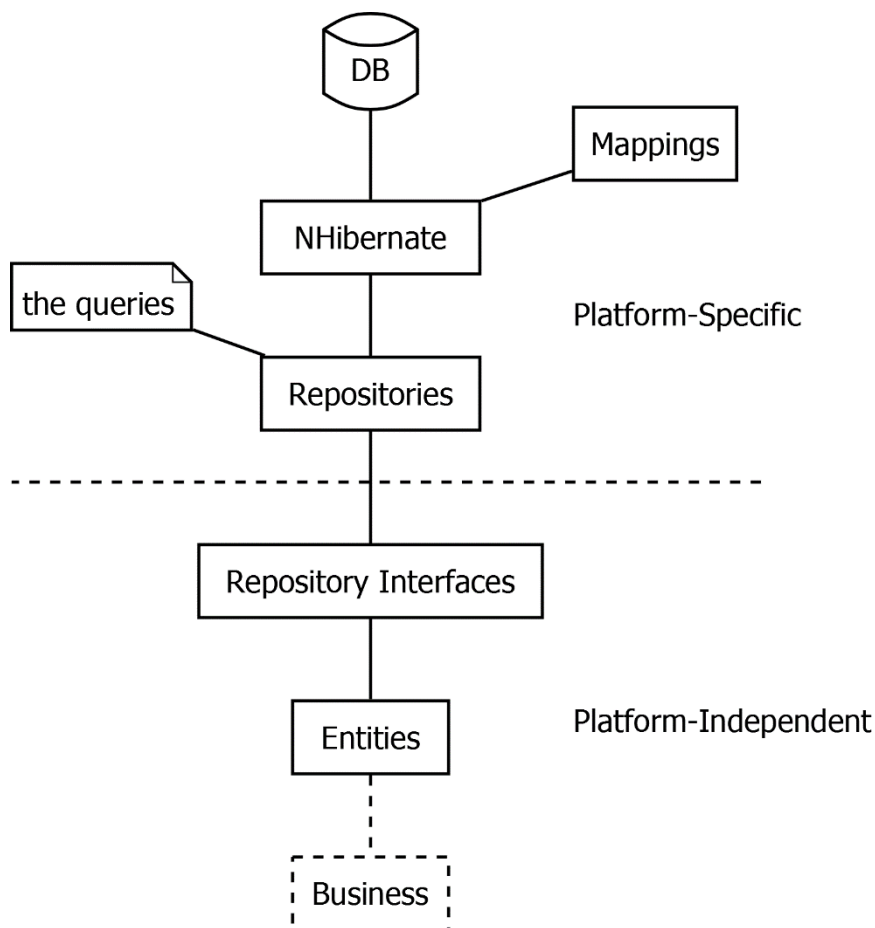
It all starts with the database. The database is not directly accessed by the rest of the code, but the database is talked to through NHibernate, an object-relational mapper. NHibernate will translate database records to instance of classes. Those classes have properties, that map to columns in the database, 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:
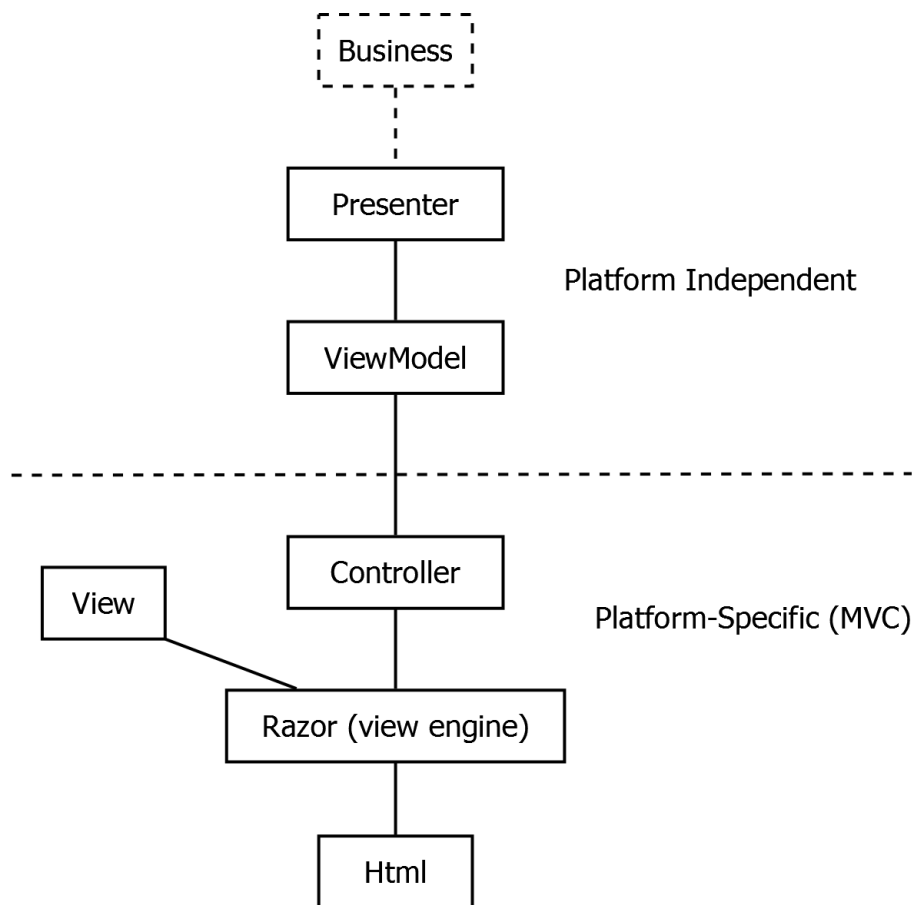
The presentation layer calls the business layer, which contains all the rules that surround the system.
Presenter classes talk to the business layer. The presenter is responsible for translating the data and the results of the business logic to a subset of data that is shown on screen.

The data that is exactly shown on screen is called 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.

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
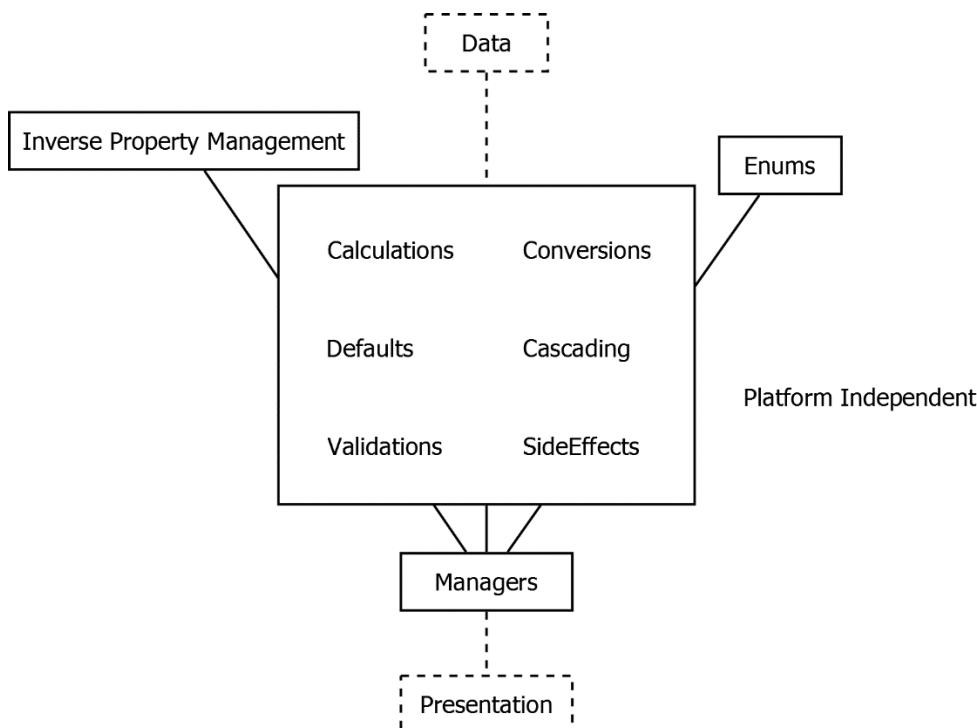
< TODO: Add 'Cloning' to big block above? It might stay too vague if you mention it there. >

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

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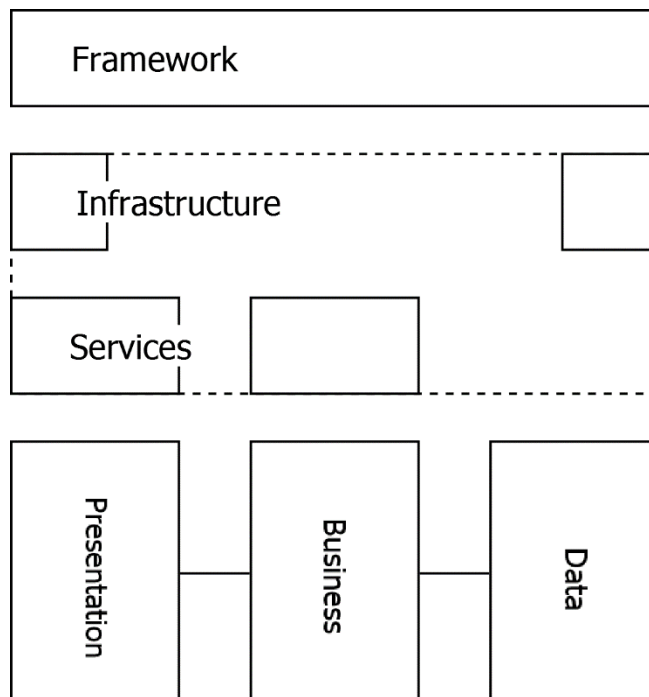
The Framework layer consists of API's that could support any aspect of software development, so could be used in any part of the layering. That is why it stretches right from Data to Presentation in the diagram.

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.

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.

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**

Design Patterns

Design patterns are coding techniques to solve common programming problems. They bring consistency to the code. They help us reuse best practices and prevent code from becoming messy. They also are an extension to the software layering.

Data Access Patterns

Entity	These are the classes that represent the domain model. The entity classes simply contain properties of simple types or references or lists to other entities. There will be no logic in the entity classes in our architecture. Collections should be created in the constructor, because NHibernate does not always create them, and you do not want to check whether collections are null all over your code. All public members should be virtual, otherwise persistence technologies can often not work with it. Do not use inheritance within your entity model, because it can make using persistence technologies harder, and it can actually harm performance of queries.
Mapping	Mappings are classes programmed for a particular persistence technology, e.g. NHibernate, that map the entity model to how the objects are persisted in the data store (e.g. an SQL Server database).
DTO	DTO = Data transfer object. DTO's only contain data, no logic. They are used solely to transfer data between different parts of the system, particularly in cases where passing an entity is not handy or efficient. For instance: A specialized, optimized SQL query may return a result with a particular record structure. You could program a DTO that is a strongly typed version of these records. In many cases you want to query for entity objects instead, but in some cases this is not fast / efficient enough and you should resort to a DTO. DTO's can also be used for other data transfers than for SQL queries.

Repository	A repository is like a set of queries. Repositories return or manipulate entities in the data store. Parameters of the methods must be simple types, not entities. The repository pattern is a way to put queries in a single place. The repository's job is also to provide an *optimal* set of queries. Typically, every entity type gets its own repository. DO NOT expose types from the underlying persistence technology.
Repository Interfaces	Any repository type will get an associated repository interface. This keeps our system loosely coupled to the underlying persistence technology.

Business Logic Patterns

Business layer	Presentation, entity model and persistence should be straightforward. If anything special needs to happen this belongs in the business layer. Any number of different patterns can be used. The business layer externally speaks a language of entities or sometimes data transfer objects (DTO's). Internally it can talk to repository interfaces for data access. It is preferred that business logic works with entities rather than repositories (even though there is a large gray area). This improves testability, limits queries and limits interdependence, dependency on a data source and passing around a lot of repository variables.
Validators	Use separate validator classes for validation. Make specialized classes derived from JJ.Framework.Validation.FluentValidator<T>. Try to keep validators independent from eachother. If multiple validators should go off, call them individually one by one. Try not to make them delegate to eachother. If you do decide to make a complex validator, add a prefix or suffix to the class name such as 'Recursive' or 'Versatile' to make extra clear that it is not just a simple validator.
Side Effects	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.

	We implement the interface ISideEffect for each side effect. It only has one method: Execute, but it allows us to have some sort of polymorphism over side effects so it is easier to execute multiple of them in one blow, or allows other more generic handlings of the side effects. Using a separate class for side effects, creates overview over those pieces of business logic, that are the most creative of all, and prevents those special things that need to happen from being entangled with other code.

LinkTo	This pattern is about inverse property management. Inverse property management means for instance that if a parent property is set: Product.Supplier = mySupplier, then automatically the product is added to the child collection too: Supplier.Products.Add(myProduct).
	
	To manage inverse properties even when the underlying persistent technology does not have inverse property management, you can link entities with LinkTo methods, instead of assigning properties or adding or removing from related collections directly. By calling the LinkTo methods, both ends of the relationship are kept in sync. Here is a template for a LinkTo method that works for 1-to-n relationships. Beware that all the checks can come with performance penalties.
	
	<pre>public static void LinkTo(this Child child, Parent parent)
	{
	    if (child == null) throw new NullException(() => child);
	
	    if (child.Parent != null)
	    {
	        if (child.Parent.Children.Contains(child))
	        {
	            child.Parent.Children.Remove(child);
	        }
	    }
	
	    child.Parent = parent;
	
	    if (child.Parent != null)
	    {
	        if (!child.Parent.Children.Contains(child))
	        {
	            child.Parent.Children.Add(child);
	        }
	    }
	}</pre>

	The class in which to put the LinkTo methods, should be called LinkToExtensions and it should be put in the LinkTo sub-namespace in your project.
	
	Only if the LinkTo method name is ambiguous, you can suffix it, e.g.:
	
	`LinkToParentDocument`
	
	Next to LinkTo method, you should have Unlink methods in an UnlinkExtensions class:
	
	``
	public static void UnlinkParent(this Child child)
	{
	    if (child == null) throw new NullException(() => child);
	    child.LinkTo((Parent)null);
	}
	``
	
	If you are linking objects together that you know are new, you may create better-performing variations for LinkTo, called NewLinkTo, that omit all the expensive checks:
	
	``
	public static void NewLinkTo(this Child child, Parent parent)
	{
	    if (child == null) throw new NullException(() => child);
	
	    child.Parent = parent;
	    parent.Children.Add(child);
	}
	``
	
	Be aware that executing NewLinkTo onto *existing* objects will result in a corrupted object graph.

Manager	A Manager class combines several related (usually CRUD) operations into one class that also performs additional business logic and validation, side effects, integrity constraints, conversions, etc. A Manager is a 'CRUD-oriented facade'. It delegates to other classes to do the work. If you do it using the manager you should be able to count on it that the integrity is maintained. It is a combinator class: a manager combines other (smaller) parts of the business layer into one offering a single entry point for a lot related operations. It is usually about a partial business domain, so manages a set of entity types together.

	Get by ID not in the Manager Even though Manager classes typically contain CRUD methods and is usually the entry point for all your business logic and data access operations, there is an exception: do not put a Get by ID method in your Manager class. Execute a simple Get by ID onto the repository. The reason is that you would get an explosion of dependency and high coupledness, since a simple operation executed all over the place, would now require a reference to a manager, which is a combinator class, meaning it is dependent on many repositories and other objects. So a simple Get goes through the repository.
Visitor	A Visitor class processes a recursive structure that might involve many objects and multiple types of objects. Usually a visitor translates a complex structure into something else. Examples are calculating a total price over a recursive structure, or filtering down a whole object graph by complex criteria. Visitors also give great performance when programmed well. Whenever a whole recursive structure needs to be processed, the visitor pattern is a good way to go. A visitor class will have a set of Visit methods, e.g. VisitOrder, VisitProduct, typically one for every type, possibly also one for each collection. A base visitor might simply follow the whole recursive structure, and has a Visit method for each node in the structure. All Visit methods are protected virtual and usually return void. Public methods will only expose the entry points in the recursion. Derived visitors can override any Visit method that they need. If you only want to process objects of a specific type, you only override the Visit method for that specific type. You can optimize performance by overriding Visit methods that would enter a part of the recursive structure that you do not use. Typically the result of a visitor is not put on the call stack, but stored in fields and used throughout the Visit methods. This is because the result usually does not have a 1-to-1 mapping with the source structure. By creating a base visitor and multiple specialized visitors, you can create short and powerful code for processing recursive structures. A coding error is easily made, and can break calculations easily. However, it is the best and fastest choice for complicated calculations that involve complex recursive structures. The classic visitor pattern has a design flaw in it, that we will not use. The classic visitor requires that classes used by the visitor have to be adapted to the visitor. This is adapting the wrong classes. We will not do that and we will not add Accept methods to classes used by a visitor.

	A good example of a Visitor class is .NET's own ExpressionVisitor, however we follow additional rules.

Front-End Patterns

ViewModel	A ViewModel class holds the data shown on screen.
	It is purely a data object. It will only have public properties. It should have no methods, no constructor, no member initialization and no list instantiation. (This is to make sure the code creating or handling the viewmodels is fully responsible for it.)
	A ViewModel should say *what* is shown, not *how* or *why*.
	Every screen gets a view model, e.g. ProductDetailsViewModel, ProductListViewModel, ProductEditViewModel, CategorySelectorViewModel.
	You can also reuse simple view models that represent a single entity, e.g. ProductViewModel, CategoryViewModel.
	ViewModels may only use simple types and references to other view models. A ViewModel should never reference data-store bound entities directly.
	Inheritance is *not* allowed, so it is a good plan to make the ViewModel classes sealed.
	Do not convert view models to other view models (except for yielding over non-persisted properties). Always convert from business domain to view model and from view model to business domain, never from view model to view model.
	A ViewModel should say *what* is shown on screen, not *how*: As such it is better to call a property CanDelete, than calling it DeleteButtonVisible. Whether it is a button or a hyperlink or Visible or Enabled property is up to the view.
	A ViewModel should say *what* is shown on screen, not *why*: For instance: if the business logic tells us that an entity is a very special entity, and it should be displayed read-only,

	the view model should contain a property IsReadOnly, not a property named ThisIsAVerySpecialEntity. Why it should be displayed read-only should not be part of the view model.
Lookup Lists	In a stateless environment, lookup lists in views can be expensive. For instance a drop down list in each row of a grid in which you choose from 1000 items may easily bloat your HTML. You might repeat the same list of 1000 items for each grid row. There are multiple ways to combat this problem. For small lookup lists you might include a copy of the list in each entity view model and repeat the same lookup list in HTML. Reusing the same list instance in multiple entity view models may seem to save you some memory, but a message formatter may actually repeat the list when sending a view model over the line. For lookup lists up until say 100 items you might want to have a single list in an edit view model. A central list may save some memory but, but when you still repeat the HTML multiple times, you did not gain much. You may use the HTML5 <datalist> tag to let a <select> / drop down list reuse the same data, but it is not supported by Safari, so it is of not much use. You might use a jQuery trick to populate a drop down just before you slide it open. For big lookup list the only viable option seems to AJAX the list and show a popup that provides some search functionality, and not retrieve the full list in a single request. Once AJAX'ed you might cache the popup to be reused each time you need to select something from it.
ToViewModel	An extension method that convert an entity to a view model. You can make simple ToViewModel methods per entity, converting it to a simple view model that represents the entity. You can also have methods returning more complex view models, such as ToDetailsViewModel() or ToCategoryTreeEditViewModel(). You may pass repositories to the ToViewModel methods if required. Sometimes you cannot appoint one entity type as the source of a view model. In that case you cannot logically make it an extension method, but you make it a helper method in the static ViewModelHelpers class. The ToViewModel classes should be put in the sub-folder / sub-namespace ToViewModel in your csproj. For an app with many views a split it up into the following files may be a good plan:

	ToIDAndNameExtensions.cs
	ToItemViewModelExtensions.cs
	ToListItemViewModelExtensions.cs
	ToPartialViewModelExtentions.cs
	ToScreenViewModelExtensions.cs
	ViewModelHelper.cs
	ViewModelHelper.EmptyViewModels.cs
	ViewModelHelper.Items.cs
	ViewModelHelper.ListItems.cs
	ViewModelHelper.Lookups.cs
	ViewModelHelper.Partials.cs
	ViewModelHelper.Screens.cs
	
	To be clear: the ViewModelHelper files are all ViewModelHelper partial classes. The other files have a class that has the same name as the file.
	
	Inside the classes, the methods should be sorted by source entity or application section alphabetically and each section should be headed by a comment line, e.g.:
	
	``csharp
	// Orders
	
	public static OrderListViewModel ToListViewModel(this IList<Order> orders) ...
	
	public static OrderEditPopupViewModel ToEditViewModel(this Order order) ...
	
	public static OrderDeletePopupViewModel ToDeleteViewModel(this IList<Order> orders) ...
	``

ToEntity	Extension methods that convert a view model to an entity. You typically pass repositories to the method. A simple ToEntity method might look up an existing entity, if it exists, it will be updated, if it does not, it will be created. A more complex ToEntity method might also update related entities. In that case related entities might be inserted, updated and deleted, depending on whether the entity still exists in the view model or in the data store. A ToEntity method takes on much of the resposibility of a Save action.

Presenters	Each view gets its own presenter. Each user action is a method. A presenter represents what a user can do in a screen. The methods of the presenter work by a ViewModel-in, ViewModel-out principle. An action method returns a ViewModel that contains the data to display on screen. Action methods can also receive a view model parameter containing the data the user has edited. Other parameters are also things the user chose. An action method can return a different view model than the view the presenter is about. Those are actions that navigate to a different view. That way the presenters are a model for what the user can do with the application. Sometimes you also pass infra and config parameters to an action method, but it is preferred that the main chunk of the infra and settings is passed to the Presenter's constructor. Internally a presenter can use business logic and repositories to access the domain model.
View	A template for rendering the view. It might be HTML. In WebForms this would be an aspx. In MVC it can be an aspx or cshtml. Any code used in the view should be dumb. That is: most tasks should be done by the presenter, which produces the view model, which is simply shown on screen. The view should not contain business logic.
ToEntity-Business-ToViewModel round-trip	A presenter is a combinator class, in that it combines multiple smaller aspects of the presentation logic, by delegating to other classes. It also combines it with calls to the business layer. A presenter action method should be organized into phases: - Security - ViewModel Validation

| | - ToEntity / GetEntities
- Business
- Commit
- ToViewModel
- Non-Persisted (yield over non-persisted data from old to new view model)
- Redirect

Not all of the phases must be present. ToEntity / Business / ToViewModel are the typical phases. Slight variations in order of the phases are possible. But separate these phases, so that they are not intermixed and entangled.

Comment the phases in the code in the presenter action method:

  // ToEntity
  Dinner dinner = userInput.ToEntity(_dinnerRepository);

  // Business
  _dinnerManager.Cancel(dinner);

  // ToViewModel
  DinnerDetailsViewModel viewModel = dinner.ToDetailsViewModel();

Even though the actual call to the business logic might be trivial, it is still necessary to convert from entity to view model and back. This is due to the stateless nature of the web. It requires restoring state from the view to the entity model in between requests. You might save the computer some work by doing partial loads instead of full loads or maybe even do JavaScript or other native code. |
| First full load – then partial load – then native code | You could also call it: first choice full load.
In web technology you could also call it:
Full postback - AJAX - JavaScript

When programming page navigation, the first choice for showing content is a full page load. Only if you have a very good reason, you might use AJAX to do a partial load. Only if you have a very good reason, you might start programming user interaction in JavaScript. |

	But it is always the first choice to do full postbacks.
	The reason is maintainability: programming the application navigation in C# using presenters is more maintainable than a whole lot of JavaScript. Also: when you do not use AJAX, the Presenter keeps full control over the application navigation, and you do not have to let the web layer be aware of page navigation details.
	Furthermore AJAX'ing comes with extra difficulties. For instance that MVC <input> tag ID's vary depending on the context and must be preserved after an AJAX call, big code blocks of JavaScript for doing AJAX posts, managing when you do a full redirect or just an update of a div. Keeping overview over the multitude of formats with which you can get and post partial content. The added complexity of sometimes returning a row, sometimes returning a partial, sometimes returning a full view. Things like managing the redirection to a full view from a partial action. Info from a parent view model e.g. a lookup list that is passed to the generation of a child view model is not available when you generate a partial view. Request.RawUrl cannot be used as a return URL in links anymore. Related info in other panels is not updated when info from one panel changes. A lot of times the data on screen is so intricately related to eachother, updating one panel just does not cut it. The server just does not get a chance to change the view depending on the outcome of the business logic. Sometimes an ajax call's result should be put in a different target element, depending on the type you get returned, which adds more complexity.
	Some of the difficulties with AJAX have been solved by employing a specific way of working, as described under *AJAX* in the *Aspects* section.
ViewModel.TemporaryID	When you edit a list, and between actions you do not commit you may need to generate ID's for the rows that are not committed, otherwise you cannot identify them individually to for instance delete a specific uncommitted row. For this you can add a TemporaryID to the view model, that are typically Guids. An alternative is to let a data store generate the ID's by flushing pendings statements to the data store, which might give you data-store-generated ID's. But this method fails when the data violates database constraints. Since the data does not have to be valid until we press save, this is usually not a viable option, not to speak of that switching to another persistence technology might not give you data-store-generated ID's upon flushing at all. The TemporaryID's can be really temporary and can be regenerated every time you create a new view model.

Front-End Patterns (MVC)

Controller	In an ASP.NET MVC application a controller has a lot of responsibilities, but in this architecture most of the responsibility is delegated to Presenters. The responsibilities that are left for the MVC controllers are the URL routing, the HTTP verbs, redirections, setting up infrastructural context and miscellaneous MVC quirks. The controller may use multiple presenters and view models, since it is about multiple screens.

Post-Redirect-Get	This is a quirk intrinsic to ASP.NET MVC. We must conform to the Post-Redirect-Get pattern to make sure the page navigation works as expected.
	
	At the end of a post action, you must call RedirectToAction() to redirect to a Get action.
	
	Before you do so, you must store the view model in the TempData dictionary. In the Get action that you redirect to, you have to check if the view model is in the TempData dictionary. If the view model exist in the TempData, you must use that view model, otherwise you must create a new view model.
	
	Here is simplified pseudo-code in which the pattern is applied.
	
	<pre>public ActionResult Edit(int id)
	{
	    object viewModel;
	    if (!TempData.TryGetValue(TempDataKeys.ViewModel, out viewModel))
	    {
	        // TODO: Call presenter
	    }
	
	    return View(viewModel);
	}
	
	[HttpPost]
	public ActionResult Edit(EditViewModel viewModel)
	{
	    // TODO: Call presenter
	
	    TempData[TempDataKeys.ViewModel] = viewModel2;
	    return RedirectToAction(ActionNames.Details);
	}</pre>

	There might be an exception to the rule to always RedirectToAction at the end of a Post. When you would redirect to a page that you can never go to directly, you might return View() instead, because there is no Get method. This may be the case for a NotFoundViewModel or a DeleteConfirmedViewModel.

ValidationMessages in ModelState	For the architecture to integrate well with MVC, you have to make MVC aware that there are validation messages, after you have gotten a ViewModel from a Presenter. If you do not do this, you will get strange application navigation in case of validation errors.
	
	You do this in an MVC HTTP GET action method.
	The way we do it here is as follows:
	
	if (viewModel.ValidationMessages.Any())
	{
	   ModelState.AddModelError(ControllerHelper.DEFAULT_ERROR_KEY,
	   ControllerHelper.GENERIC_ERROR_MESSAGE);
	}
	
	In theory we could communicate all validation messages to MVC instead of just communicating a single generic error message. In theory MVC could be used to color the right input fields red automatically, but in practice this breaks easily without an obvious explanation. So instead we manage it ourselves. If we want a validation summary, we simply render all the validation messages from the view model ourselves and not use the Html.ValidationSummary() method at all. If we want to change the appearance of input fields if they have validation errors, then the view model should give the information that the appearance of the field should be different. Our view's content is totally managed by the view model.

Polymorphic RedirectToAction / View()	A Presenter action method may return different types of view models.
	
	This means that in the MVC Controller action methods, the Presenter returns object and you should do polymorphic type checks to determine which view to go to.
	
	Here is simplified code for how you can do this in a post method:
	
	``
	var editViewModel = viewModel as EditViewModel;
	if (editViewModel != null)
	``

	``c#
{
    return RedirectToAction(ActionNames.Edit,
        new { id = editViewModel.Question.ID });
}

var detailsViewModel = viewModel as DetailsViewModel;
if (detailsViewModel != null)
{
    return RedirectToAction(ActionNames.Details,
        new { id = viewModel.Question.ID });
}
```

At the end throw the following exception (out of the Framework):

```c#
throw new UnexpectedTypeException(() => viewModel);
```

To prevent repeating this code for each controller action, you could program a generalized method that returns the right ActionResult depending on the ViewModel type. Do consider the performance penalty that it may impose and it is worth saying that such a method is not very easy code. |
| Html.BeginCollection | In MVC it is not straightforeward to post a collection of items or nested structures.

This architecture's framework has HtmlHelper extensions to make that easier: the Html.BeginCollection API. Using this API you can send a view model with arbitrary nestings and collections over the line and restore it to a view model at the server side. In the view code you must wrap each nesting in a using block as follows:

```c#
@using (Html.BeginItem(() => Model.MyItem))
{
    using (Html.BeginCollection(() => Model.MyItem.MyCollection))
    {
        foreach (var x in Model.MyItem.MyCollection)
        {
            using (Html.BeginCollectionItem())
            {

            }
        }
    }
}
``` |

So each time you enter a level, you need another call to the Html helper again and wrap the code in a using block. You can use as many collections as you like, and use as much nesting as you like. You can spread the nesting around multiple partials.

Input fields in a nested structure must look as follows:

    Html.TextBoxFor(x => x.MyProperty)

Or:

    Html.TextBoxFor(x => Model.MyProperty)

Not like this:

    Html.TextBoxFor(x => myLoopItem.MyItem.MyProperty)

Otherwise the input fields will not bind to the view model. This often forces you to program partial views for separate items. This is good practice anyway, so not that big a trade-off.

An alternative to Html.BeginCollection() is using for-loops.

```
@Html.TextBoxFor(x => x.MyItem.MyProperty)

@for (int i = 0; i < Model.MyItem.MyCollection.Count; i++)
{
    @Html.TextBoxFor(x => x.MyItem.MyCollection[i].MyProperty)
}
```

This solution only works if the expressions you pass to the Html helpers contain the full path to a view model property (or hack the HtmlHelper.ViewData.TemplateInfo.HtmlFieldPrefix) and therefore it does not work if you want to split up your view code into partials.

Another alternative to the BeginCollection() is the often-used BeginCollectionItem(string) API. Example:

```
@foreach (var child in Model.Children)
{
```

<table>
<tr>
<td></td>
<td>

```
using (Html.BeginCollectionItem("Children"))
{
    @* ... *@
}
}
```

The limitation of that API is that you can only send one collection over the line and no additional nesting is possible.

Beware that currently the different solutions do not mix well and you should only use one solution for each screen of you program.

</td>
</tr>
<tr>
<td>Return URL's</td>
<td>

- Return URL's indicate what page to go back to when you are done in another page.
- It is used when you are redirected to a login screen, so it knows what page to go back to after you login.
- Return URL's are encoded into a URL parameter, called 'ret' e.g.:
  http://www.mysite.com/Login?**ret=%2FMenu%2FIndex**
  The ret parameter is the following value encoded: /Menu/Index
  That is the URL you will go back to after you log in.
- The Login action can redirect to the ret URL like this:

```
[HttpPost]
public ActionResult Login(... string ret = null)
{
    ...
    return Redirect(ret);
    ...
}
```

ASSIGN DIFFERENT RET FOR FULL PAGE LOAD OR AJAX CALL.

- For full page loads, the ret parameter must be set to:

```
    Request.RawUrl
```

- For AJAX calls the ret parameter must be set to:

```
    Url.Action(ActionNames.Index)
```

</td>
</tr>
</table>

	- The ret parameter is set in a controller action method, when you return the ActionResult. Example: EXAMPLE WORKS FOR FULL PAGE LOAD ONLY!!! `return RedirectToAction(ActionNames.Login, ControllerNames.Account, new { ret = Request.RawUrl });` - That way you have an easily codeable, well maintainable solution. - Do not use RefferrerUrl, because that only works for HttpPost, not HttpGet.

Data Transformation Patterns

Converter	A class that converts one data structure to another. Typically more is involved than just converting a single object. A whole object graph might be converted to another, or a flat list or raw data to be parsed might be converted to an object structure or the other way around. By implementing it as a converter, it simplifies the code. You can then say that the only responsibility of the class is to simply transform one data structure to another: nothing more, nothing less and leave other responsibilities to other classes.
TryGet-Insert-Update	When converting one type to another one might use the TryGet-Insert-Update pattern. Especially when converting an entity with related entities from one structure to another this pattern will make the code easier to read. TryGet first gets a possible existing destination entity. Insert will create the entity if it did not exist yet, possibly setting some defaults. Update will update the rest of the properties of either the existing or newly created object. When you do these actions one by one for one destination entity after another, you will get readable code for complex conversions between data structures. Note that deletion of destination objects is not managed by the TryGet-Insert-Update pattern.

TryGet-Insert-Update-Delete / Collection Conversion	Used for managing complex conversions between data structures, that require insert, update and delete operations. There is no one way of implementing it, but generally it will involve the following steps:
	
	  - Loop through the source collection.
	      - TryGet: look up an item in the destination collection.
	      - Insert: create a new item in the destination collection if none exists.
	      - Update: update the newly created or existing destination item.
	  - Do delete operations after that:
	      - Generally you can use an Except operation on the collections of existing items and items to keep, to get the collection of items to delete.
	      - Then you loop through that collection and delete each item.

DocumentModel	An analog of a view model, but then for document generation, rather than view rendering. It is a class that contains all data that should be displayed in the document. It can end with the suffix 'Model' instead of 'DocumentModel' for brevity, but then it must be clear from the context that we are talking about a document model. Just as with view models, inheritance structures are not allowed. To prevent inheritance structures it may be wise to make the DocumentClasses classes sealed.

Selector-Model-Generator-Result	For data transformations you may want to split up the transformation in two parts:
	
	  - A Selector which returns the data as an object graph, or Model.
	  - A Generator (or Converter) that converts the object graph (or Model) into a specific format.
	
	This is especially useful if there are either multiple input formats or multiple output formats or both, or if in the future either the input format or output format could change.
	
	This basic pattern is present in many architectures and can be applied to many different parts of architecture.
	
	Examples:

Generating a document:

An example of where it is useful, is generating a document in multiple format e.g. XLSX, CSV and PDF. In that case the data selection and basic tranformations are programmed once (a Selector that produces a Model) and exporting three different file formats would require programming three different generators. Reusable generators for specific file formats such as CSV may be programmed. Those will make programming a specialized generators very easy. So then basically exporting a document is mostly reading out a data source and producing an object graph.

Data source independence:

The Selector-Model-Generator-Result pattern is also useful when the same document can have different data sources. Let's say you want to print an invoice out of the system, but print another invoice out of an ordering system in the same formatting e.g. a PDF. This requires 2 selectors, 1 model and 1 generator, instead of 2 generators with complex code and potentially different-looking PDF's.

Multiple import formats:

You might want to import similar data out of multiple different data sources or multiple file formats. By splitting the work up into a Selector and a generator you can share must of the code between the two imports, and reduce the complexity of the code.

Limiting complexity:

Even if you do not expect multiple input formats or multiple output formats or a change in input or output format, the split up in a Selector and a Generator can be used to make the code less complicated to write, and subsequently also prevent errors and save time programming and maintaining the code.

MVC:

MVC itself contains a specialized version of this very pattern. The following layering stacks are completely analogus to eachother:
- Selector - Model - Generator – Result
- Controller - ViewModel - view engine – View

Other Patterns

Anti-encapsulation	Encapsulation makes sure a class protects its own data integrity. Anti-encapsulation is the design choice to let a class check none of its data integrity. Then you know that something else is 100% responsible for the integrity of it, and the class itself will guard none of it. The reason not to use encapsulation is that it can go against the grain of many frameworks, such as ORM's and data serialization mechanisms. Anti-encapsulation can also be a solution to prevent spreading of the same responsibility over multiple places. If the class cannot check all the rules itself, it may be better the check all the rules elsewhere, instead of checking half the rules in the class and the other half in another place.
Constructor Inheritance	Sort of forces a derived class to have a constructor with specific arguments. Constructors are not inherited, but inheriting from a base class that has specific constructors forces your derived class to call that base constructor, often leading to exposing a similar constructor in the derived class.
Executor	Executor classes are classes that encapsulate a whole process to run. For processes that involve more than just a single function, for instance downloading a file, transforming it and then importing it, involving infrastructure end-points and possibly multiple back-end libraries. By giving each of those processes its own executor class, you make the code overviewable, and also make the process more easily runnable from different contexts, e.g. in a scheduler, behind a service method or by means of a button in a UI or in a utility.
Inheritance-Helper	A weakness of inheritance in .NET is that there is no multiple inheritance: you can only derive from one base class. This often leads to problems programming a base class, because one base will offer you one set of functionalities and the other base the other functionalities. (See the 'Cartesian Product of Features Problem'.) To still use inheritance to have behaviors turned on or off, but not have an awkward inheritance structure, and problems picking what feature to put at which layer of inheritance, you could simply program helper classes (static classes with static methods) that implement each feature, and then use inheritance, letting derived classes delegate to the helpers, to give each class a specific set of features and

	specific versions of the features, to polymorphically have the features either turned on or off. You will still have many derived classes, but no arbitrary spreading of features over the base classes, and no code repetition either. This allows you to solve what inheritance promises to solve, but does not do a good job at on its own. It basically solves the Cartesian Product of Features problem, the problem that there is no multiple inheritance and the problem with god base classes, all weakneses of inheritance.

Factory	A factory class is a class that constructs instances. But it usually means that it creates a concrete type, returning it as an abstract type. The concrete type that is instantiated depends on the input you pass to the factory's method:
	
	<pre>public static class ThingFactory
	{
	    public static IThing CreateThing(int parameter)
	    {
	        switch (parameter)
	        {
	            case 0:
	                return new NormalThing();
	
	            case 1:
	                return new SpecialThing();
	
	            default:
	                throw new Exception(String.Format("parameter value '{0}' is not supported.", parameter));
	        }
	    }
	}</pre>
	A factory class is used if you want to instantiate an implementation of a base class or interface, but it depends on conditions which implementation it has to be or if you wish to abstract away knowledge of the specific concrete types it produces.

	A class that returns instances with various states is also simply called a Factory, even though no polymorphism is involved. (The classic implementation is not used here, which is a static method in a base class.)
Factory-Base-Interface	The Factory-Base-Interface pattern is a common way the factory pattern is applied. Next to a factory, as described above in the 'Factory' pattern, you give each concrete implementation that the factory can return a mutual interface, which also becomes the return type of the factory method. To also give each concrete implementation a mutual base class, with common functionality in it, and also to sort of force an implementation to have a specific constructor (see 'Constructor Inheritance').
TryGet	A combination of a *TryGet* method and a *Get* method (e.g. *TryGetObject* and *GetObject*) means that *TryGet* will return null if the object does not exist and *Get* will throw an exception if the object does not exist. Call *Get* if it makes sense that the object should exist. Call *TryGet* if the non-existence of the object makes sense. If you call a *TryGet* you should handle the null value that could be returned. *TryGet* can throw other exceptions, even though it does not throw an exception if the object does not exist.

Get-TryGet-GetMany	Often you need a combination of the three methods that either get a list, a single item but allow null or get a single item and insist it is not null. You can implement the plural variation and base the Get and TryGet on it using the same kind of code every time:
	
	``csharp
	public Item GetItem(string searchText)
	{
	    Item item = TryGetItem(searchText);
	
	    if (item == null)
	    {
	        throw new Exception(String.Format("Item with searchText '{0}' not found.", searchText));
	    }
	
	    return item;
	}
	
	public Item TryGetItem(string searchText)
	``

<table>
<tr>
<td></td>
<td>

```csharp
    {
        IList<Item> items = GetItems(searchText);

        switch (items.Count)
        {
            case 0:
                return null;

            case 1:
                return items[0];

            default:
                throw new Exception(String.Format(
                    "Multiple items found for searchText '{0}'.", searchText));
        }
    }

    public IList<Item> GetItems(string searchText)
    {
        return _items.Where(x => !String.IsNullOrEmpty(x.Name) &&
                                  x.Name.Contains(searchText))
                     .ToArray();
    }
```

The GetItem and TryGetItem methods are the same in any situation, except for names and exception messages. Only the plural method is different depending on the situation.
</td>
</tr>
</table>

Helper	Helper classes are static classes with static methods that help with a particular aspect of programming. They can make other code shorter or prevent repeating of code, for functions that do not require any more structure than a flat list of methods.
Info	Info objects are like DTO's in that they are usually used for yielding over information from one place to another. Info objects can be used in limited scopes, internal or private classes and serve as a temporary place of storing info. But info objects can also have a broader scope, such as in frameworks, and unlike DTO's they can have constructor parameters, auto-instantiation, encapsulation and other implementation code.
Mock	A mock object is used in testing as a replacement for a object used in production. This could be an entity model, an alternative repository implementation (that returns mock entities instead of data out of a

	database). A mock object could even be a database record. Unlike other patterns the convention is to put the word 'Mock' at the beginning of the class rather than at the end.

Names	To prevent typing in a lot of strings in code, make a static class with constants in it, that become placeholders for the name.
	e.g. ViewNames, with constants in it like this:
	
	``csharp
	public static class ViewNames
	{
	    public const string Edit = "Edit";
	}
	``
	
	the name of the constant should be exactly the same as the string text.
	Everywhere you need to use the name, refer to the constant instead of putting a literal string there.
	
	This prevents typing errors and makes 'find all references' possible.

Singular, Plural, Non-Recursive, Recursive and WithRelatedEntities	When processing object structures, it is best to split everything up into separate methods. Every entity type will get a method (the 'Singular' variation) that processes a single object. That method will not process any underlying related items, only the one object. In case of conversions from one object structure to another, every *destination* entity gets a Singular method, not the source entity, because that would easily create messy, unmanageable code. A 'Plural' method processes a whole list of items. Plural methods are less useful. Prefer singular methods over plural ones. Plural methods usually do not add anything other than a loop, which is too trivial to create a separate method for. Only when operations must be executed onto a whole list of objects (for instance determining a total price of a list of items or when there are specific conditions), it may be useful to create a separate Plural method. Singular or Plural methods do not process related entities unless they have the method suffix 'WithRelatedEntities' or 'Recursive' at the end of the method name. Keep the recursive methods separate from the non-recursive methods.

There is a subtle difference between 'WithRelatedEntities' and 'Recursive'. They are similar, but Recursive processing can pass the same object type again and again, while processing with related entities processes a tree of objects, in which the same object type does not recur at a deeper level.

Here is an example of some Singular, Plural, Non-Recursive and Recursive methods. Note that the words 'Singular' and 'Plural' are not used in the method names.

```csharp
private class MyProcess
{
    private StringBuilder _sb = new StringBuilder();

    public string ProcessRecipeRecursive(Recipe recipe)
    {
        if (recipe == null) throw new NullException(() => recipe);

        ProcessRecipe(recipe);

        ProcessIngredients(recipe.Ingredients);

        ProcessRecipesRecursive(recipe.SubRecipes);

        return _sb.ToString();
    }

    private void ProcessIngredients(IList<Ingredient> ingredients)
    {
        _sb.AppendLine("Ingredients:");

        foreach (Ingredient ingredient in ingredients)
        {
            ProcessIngredient(ingredient);
        }
    }

    private void ProcessIngredient(Ingredient ingredient)
    {
        _sb.AppendLine(String.Format("{0} {1} ({2})", ingredient.QuantityDescription, ingredient.Name, ingredient.ID));
    }

    private void ProcessRecipesRecursive(IList<Recipe> recipes)
    {
        _sb.AppendLine("Sub-Recipes:");

        foreach (Recipe recipe in recipes)
        {
            ProcessRecipeRecursive(recipe);
        }
    }
}
```

| | ```csharp
private void ProcessRecipe(Recipe recipe)
{
    _sb.AppendLine(String.Format("Recipe: {0} ({1})", recipe.Name, recipe.ID));
}
}
``` |
| String Resources | For button texts, translations of model properties in different languages, etc., use resx files in your .NET projects.
If you follow the following naming convention for resources files, .NET will automatically return the translations into the language of the current culture:
Titles.resx
Titles.nl-NL.resx
Titles.de-DE.resx
Most resources should be put in the front-end assemblies, since it is presentation logic, but display names of model properties should be put in the back-end, so they can be reused in multiple applications.

Framework.Resources contains reusable resource strings for common titles such as 'Delete', 'Edit', 'Save' etcetera.

The culture-inspecific resx has the en-US language.
The key should be representative of the text itself.

Extra information in Dutch about how to structure your resource files can be read in Appendix B. |
| Wrapper | A wrapper class is a class that wraps one or more other objects. This can be useful in various situations. You might give the wrapper additional helper methods that the wrapped object does not have. You might dispose the underlying wrapper object and create a new one, keeping the references to the wrapper object in tact even though the wrapped object does not exist anymore. You may hide a specific object in a wrapper and give it an alternative interface, you might wrap multiple objects in one wrapper to pass them around as a single object for convenience. |

Aspects

This section lists some choices concerning system aspects such as security and logging, etc, even though many system aspects are already described under 'Design Patterns'. Certain aspects already mentioned in the Design Patterns section may be repeated here, so it creates a clearer picture of what is regarded an aspect and what not.

Technology-Independent Aspects

Authoring	This aspect covers things such as marking objects with creation dates, modification dates, etcetera, adding an author's comment to objects and managing multiple versions of objects and logging which user made which change. <TODO: Add specific solutions. >
Cloning	<No description yet.>
Concurrency	In a web-based application time elapses between retrieving data to edit and saving changes. Between these actions the data may have been changed by another user. In this architecture the concurrency strategy is: the last user wins. This is accomplished in code using TryGet-Insert-Update-Delete pattern, that results in readable saving code and restoration of state, regardless of what another user did to it.

Configuration	For configuration we will use our own API: Framework.Configuration. It makes it easier to work with complex configuration files, while using .NET's System.Configuration directly can be quite a lot of work.
	
	We will use 3 ways of storing configuration settings:
	
	   - Custom configuration sections.
	   - Reading out the appSettings section.
	   - Reading out the connectionStrings section.
	
	There is another configuration method in .NET: the Settings designer in the project properties. We will not use that, because it is very error-prone. The synchronization between the entered data and the

XML does not work very well, and this creates the risk that you might put production settings in a test environment or test settings in a production environment.

Custom configuration sections:

If your configuration requires more than a flat list of key value pairs, you might make a custom configuration section. In a configuration section you can add as much hierarchy as you like. You can read out structures like the following:

```xml
<jj.demos.configuration>
    <items>
      <item name="Name1" value="1" />
      <item name="Name2" value="2" >
        <childItem name="Child" value="3" />
      </item>
    </items>
</jj.demos.configuration>
```

You can create any kind of nesting you want. With classic .NET, reading out nested configurations requires about 1 ½ hours of programming for a simple structure. With our own framework, it is easy.

You have to include the following line in the configSections element in your config file:

```xml
<section name="jj.demos.configuration" type="JJ.Framework.Configuration.ConfigurationSectionHandler,
JJ.Framework.Configuration"/>
```

You only need to replace the name 'jj.demos.configuration' with your assembly name converted to lower case. (You can also use a custom name, which you then have to use explicitly in your code, but that is not advised.)

To read out the config you call:

```csharp
ConfigurationSection config = CustomConfigurationManager.GetSection<ConfigurationSection>();
```

MyConfigurationSection is a class you have to program yourself. Its structure of properties corresponds 1-to-1 to the XML structure. The specific behavior of the mapping is documented in

the summary of CustomConfigurationManager. In short: properties map to XML elements unless you mark the property with [XmlAttribute]. Array items are expected to have the class name as the XML element name, but the element name of the array items can be specified explicty with [XmlArrayItem]. Here is an example:

```csharp
internal class ConfigurationSection
{
    [XmlArrayItem("item")]
    public ItemConfig[] Item { get; set; }
}

internal class ItemConfig
{
    [XmlAttribute]
    public string Name { get; set; }

    [XmlAttribute]
    public string Value { get; set; }

    public ItemConfig ChildItem { get; set; }
}
```

Note that C# will follow the convention that property names are pascal case, while this automatically maps to the convention in XML, in which element and attribute names are camel case.

appSettings:

An appSetting looks as follows in the App.config or Web.config:

```xml
<appSettings>
  <add key="TestInt32" value="10"/>
</appSettings>
```

Reading out the appSettings classically, is done as follows:

```csharp
int testInt32 = Convert.ToInt32(ConfigurationManager.AppSettings["TestInt32"]);
```

But you get errors if you mistype the string and an invalid cast exception if you convert to the wrong type. The alternative in Framework.Configuration is strongly typed. You first need to define an interface:

```csharp
internal interface IAppSettings
{
    int TestInt32 { get; }
}
```

This interface defines the names and types of the settings. To retrieve a setting you use:

```csharp
int testInt32 = AppSettings<IAppSettings>.GetValue(x => x.TestInt32);
```

It automatically converts to the right data type and allows you to use strongly-typed names.

connectionStrings:

Reading out connectionStrings is similar to reading out the appSettings. Connection strings in the App.config or Web.config look as follows:

```xml
<connectionStrings>
  <add name="OrderDB" connectionString="data source=10.40.XX.XX;Initial Catalog=OrderDB..." />
</connectionStrings>
```

This is the classic way of reading it out:

```csharp
string connectionString = ConfigurationManager.ConnectionStrings["OrderDB"].ConnectionString;
```

This is the alternative in Framework.Configuration:

```csharp
string connectionString = ConnectionStrings<IConnectionStrings>.Get(x => x.OrderDB);
```

You need to define an interface to be able to use the strongly-typed name:

```csharp
internal interface IConnectionStrings
{
    string OrderDB { get; }
}
```

Conversion	See: 'Converter', 'TryGet-Insert-Update', 'TryGet-Insert-Update-Delete / Collection Conversion', 'Singular, Plural, Non-Recursive, Recursive and WithRelatedEntities' under 'Design Patterns'.
Defaults	Implemented as side-effects that go off in a manager class's Create methods. See 'Side Effects', 'Manager' under 'Design Patterns'.
Entity Status Management	Entity status management (or 'object status management') is the recording of whether an entity is new, dirty, clean or deleted. Also it is recording if individual properties are dirty or clean. Currently entity status management is done explicitly by using an EntityStatusManager class, that is simply a wrapper for some dictionaries and HashSets that store this information. Then EntityStatusManager is then passed around the presentation and business layer for a particular functional domain.

Enums	*General rules:*
	
	- Use the 'Enum' suffix for enum types e.g. OrderStatus**Enum**.
	- Always give an enum the enum member Undefined with value 0:
	
	``
	enum MyEnum
	{
	    Undefined = 0
	}
	``
	
	This prevents you from accidently forgetting to assign the enum value.
	
	- Prefer not using specific underlying enum types. Enums 'derive' from int by default, but you can e.g. do the following, which is not recommended:
	
	``
	enum MyEnum : long
	{
	}
	``
	
	- When taking action depending on an enum value, you might use a switch statement. In that case always specify the default case and throw an exception in the default case:
	
	``
	MyEnum myEnum;
	switch (myEnum)
	{
	``

```
            case MyEnum.MyEnumMember1:
                // Do something
                break;

            case MyEnum.MyEnumMember2:
                // Do something
                break;

            default:
                throw new InvalidValueException(myEnum);
                // OR:
                throw new ValueNotSupportedException(myEnum);
        }
```

Not only is it informative for the programmer debugging a problem and does it prevent processing invalid or incomplete data, it is also a fail-safe for the fact that an enum is a very weak type. You can assign any int value to it, even ones that are not an enum member!

```
enum MyEnum
{
    Undefined = 0,
    MyEnumMember1 = 1,
    MyEnumMember2 = 2
}

var myEnum = (MyEnum)3; // WORKS!
```

The difference between throwing an InvalidValueException or a ValueNotSupportedException is that you would use InvalidValueException if all enum members except Undefined were part of the switch, because then it was not a sensible enum value. You would throw ValueNotSupportedException if the switch uses only some of the enum members, but other perfectly sensible members were not relevant in this particular case. But it is not a disaster to use these exception types interchangedly.

- Use enum member *Undefined* in place of *null*, so also *avoid nullable enum* types.

Enum-like entities:

- Entity models often contain enum-like entities:

```
public class SectionType
{
    public virtual int ID { get; set; }
    public virtual string Name { get; set; }
}
```

Often you do not need more than these two properties.
It is common to end the enum-like entity type with the suffix 'Type' (not a strict requirement).
The Name property will be filled with the string that is exactly the enum member name:

```
new SectionType
{
    ID = 5,
    Name ="SubChapter"
}
```

- Enum-like entities have an enum-equivalent in the *Business* Layer:

```
public enum SectionType
{
    Undefined = 0,
    Book = 1,
    Article = 2,
    Paragraph = 3,
    Chapter = 4,
    SubChapter = 5
}
```

Note that the enums themselves do not belong in the entity model, but in the Business layer.

- It is _not_ recommended to give enum-like entities an inverse property to the entities that use it.

```
public class SectionType
{
    // NOT RECOMMENDED!
    public virtual IList<Section> Sections { get; set; }
}
```

The problem with this is that the list is likely to become very large, and maintaining this list (for instance in the LinkTo methods) can result in queries very harmful for performance, while you are not even noticing you are doing anything significant.

- To make assigning an enum-like entity easier, you can put extension methods in your *Business* layer. You can put this in the *Extensions* folder and call the class *EnumExtensions*. They also ensure consistency in the way that enum-like types are handled. The enum extensions allow you to write code as follows to assign enum-like entities:

```
SectionTypeEnum sectionTypeEnum = section.GetSectionTypeEnum();

section.SetSectionTypeEnum(SectionTypeEnum.Paragraph, _sectionTypeRepository);
```

Here is an example implementation of the extension methods:

```
public static SectionTypeEnum GetSectionTypeEnum(this Section section)
{
    if (section == null) throw new NullException(() => section);

    if (section.SectionType == null) return SectionTypeEnum.Undefined;

    return (SectionTypeEnum)section.SectionType.ID;
}

public static void SetSectionTypeEnum(this Section entity, SectionTypeEnum enumValue,
ISectionTypeRepository repository)
{
    if (repository == null) throw new NullException(() => repository);

    if (enumValue == SectionTypeEnum.Undefined)
    {
        entity.UnlinkSectionType();
    }
    else
    {
        SectionType sectionType = repository.Get((int)enumValue);
        entity.LinkTo(sectionType);
    }
}
```

Localization:

- Localization of the enum member display names is done by means of resources, usually in the PropertyDisplayNames.resx in the Business layer. (See the 'Resources' pattern and Appendix B for explanations on how to manage resources). The key of the resource should exactly match the enum member name.
- The following code allows you to retrieve an enum member display name:

```
PropertyDisplayNames.ResourceManager.GetString(SectionTypeEnum.Paragraph.ToString()
)
```

But a helper extension methods can make the code much more readable. This allows you to for instance use:

```
string str1 = ResourceHelper.GetSectionTypeDisplayName(section);
string str2 = ResourceHelper.GetPropertyDisplayName(sectionType);
string str3 = ResourceHelper.GetPropertyDisplayName(sectionTypeEnum);
string str4 = ResourceHelper.GetPropertyDisplayName("Paragraph");
```

Put a class in your Business.Resouces namespace, can it for instance ResourceHelper. These are examples of such ResourceHelper methods:

```
public static class ResourceHelper
{
    public static string GetSectionTypeDisplayName(Section section)
    {
        if (section == null) throw new NullException(() => section);

        string str = GetPropertyDisplayName(section.SectionType);
        return str;
    }

    public static string GetPropertyDisplayName(SectionType sectionType)
    {
        if (sectionType == null) throw new NullException(() => sectionType);

        string str = PropertyDisplayNames.ResourceManager.GetString(sectionType.Name);
        return str;
    }

    public static string GetPropertyDisplayName(SectionTypeEnum sectionTypeEnum)
    {
```

JJ Software Architecture
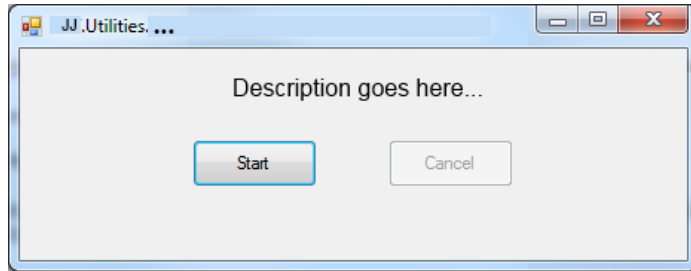
| | ```
            string str =
PropertyDisplayNames.ResourceManager.GetString(sectionTypeEnum.ToString());
            return str;
        }

        public static string GetPropertyDisplayName(string resourceName)
        {
            string str = PropertyDisplayNames.ResourceManager.GetString(resourceName);
            return str;
        }
    }
``` |
| Exceptions | - Do null-checks on arguments of public methods.
- Sometimes miscellaneous checks need to be performed on public methods.
- Arguments of private methods do not have to be checked if the class already enforces the rule elsewhere.
- For null-checks, use **NullException** (in Framework.Reflection).
- Use **NotImplementedException** for methods you have not finished yet.
- Use **NotSupportedException** for interface methods that are not supported.
- Use **InvalidValueException** or **ValueNotSupportedException** (in Framework.Common) in the default in a switch statement over an enum value.
- Use other exceptiont types out of Framework.Reflection and Framework.Common.
- Otherwise just throw the **Exception** base class.
- Mention the variable or property name in the exception message.
- Mention the ID of an object in the exception message.
- Possibly mention the invalid value in the exception message.
- For most exceptions, use dry formulations such as 'not found', 'cannot be null' and 'must be greater than 0'.
- Put exceptions at the beginning of a method if possible.
- Do not use exception filtering (catching specific exception types) unless you absolutely have to:
  try
  {
    // Do something
  }
  catch (IOException) |

	``
	{
	}
	``
	In fact, prefer not to retrieve information by catching an exception at all.

Inverse Relationship Management / Inverse Property Management	See 'LinkTo' under 'Design Patterns'.

Logging	Be careful how much you log. Logging unhandled exceptions is usually good enough. If you log a lot, it creates a performance penalty and can impose a serious strain on your infrastructure. Servers have crashed under the pressure of logging. A simple try-catch on a main level and a call to the logger will usually suffice.
	
	For logging we will use our own API: Framework.Logging. It has an easy interface and simple configuration. It allows you to log to a file or debug output and is extensible to support more such logging channels. You can use a log level with your log calls and configure which log levels are included in which logging channel. For instance: you might only log exceptions to a file, but log debug information to the debug output.
	
	Config example:
	
	``xml
	<configuration>
	  <configSections>
	    <section name="jj.framework.logging" type="JJ.Framework.Configuration.ConfigurationSectionHandler,
	JJ.Framework.Configuration"/>
	    <section name="jj.framework.logging.file" type="JJ.Framework.Configuration.ConfigurationSectionHandler,
	JJ.Framework.Configuration"/>
	  </configSections>
	
	  <jj.framework.logging>
	    <loggers>
	      <logger type="DebugOutput" level="Debug" />
	      <logger type="File" level="Exception" />
	    </loggers>
	  </jj.framework.logging>
	
	  <jj.framework.logging.file
	    filePathFormat="C:\Log\JJ.Utilities.MyUtility-DEV-{0}.log"
	    filePathDateFormat="yyyy_MM_dd_HH" />
	</configuration>
	``

	If you insist on using Log4Net, make a separate ILogger implementation behind which you hide Log4Net. The downside of Log4Net is that its configuration can be quite verbose and complicated. Framework.Logging is simple and can run on all platforms.
Multi-Language	For button texts and other labels in an application: see 'Resources' under 'Other Patterns'. That does not solve multi-lingual data, for which multiple solutions are possible, none of which may be described yet.
Naming	See 'Names' under 'Coding Style'.
Paging	All page numbering starts at 1. Even though we usually start counting at 0 as programmers, to the user the first page is still 1 and it is very confusing if you do not carry through the same numbering throughout the whole software layering. Only right before you retrieve something from a data store you may convert the numbers to fit your data store's needs. Throughout the software layering we pass through 1-based page numbers and page count. Our data store may need a first index instead, but we only convert to that number as deeply into the layering as possible.
Persistence	To access a data store (usually a database), Framework.Persistence will be used. Through that framework you can access data using different underlying persistence technologies, such as NHibernate and Entity Framework or even flat files or XML. The framework gives you a single interfacing regardless of the underlying persistence technology, loosely coupling the business logic and front-ends from the way you store your data. The main interface of the framework is IContext. Here is a ubiquitous quirk of ORM: Many methods of IContext work with uncommitted / non-flushed entities: so things that are newly created, and not yet committed to the data store. But IContext.Query usually does the opposite: it only returns committed / flushed entities. This asymmetry is common in ORM's and doing it any other way would harm performance.

Security	Authentication, authorization and user rights management in the application architecture will be interfaced with using pretty much the same pattern as the way we interface with persistence. Just like we create an IContext and repositories in the top-level project, often an MVC app, and pass it to the layers below that, the security context is also created in the top-level project, and passed to the layers below that. Both persistence and security are infrastructural things, and they will be handled in a symmetric way.
	There are the following interfaces:
	IAuthenticator IAuthorizer IRightsManager
	The interfaces might have different implementations, depending on the underlying security technology used.
	IAuthenticator will validate if a user's credentials are correct. IAuthorizer will verify if that user is permitted to access certain parts of the system. IRightsManager will allow you to manage and change the users' rights.

```
IAuthenticator
{
    bool IsAuthentic(string userName, …);
    void AssertAuthentication(string userName, …);
}

IAuthorizer
{
    bool IsAuthorized(string userName, params string[] securablePathElements);
    void AssertAuthorization(string userName, params string[] securablePathElements);
}

IRightsManager
```

<table>
<tr><td></td><td>

```
{
    bool UserExists(string userName);
    bool UserIsLocked(string userName);

    void CreateUser(string userName, string password);
    void DeleteUser(string userName);
    bool ChangePassword(string userName, string oldPassword, string newPassword);
    bool ChangeUserName(string oldUserName, string newUserName);
    string ResetPassword(string userName);
    bool UnlockUser(string userName);

    void Grant(string userName, params string[] securablePathElements);
    void Revoke(string userName, params string[] securablePathElements);

    void CreateSecurable(params string[] securablePathElements);
    void DeleteSecurable(params string[] securablePathElements);
    bool SecurableExists(params string[] securablePathElements);

    IList<string> GetPageOfUserNames(int pageNumber, int pageSize);
    int GetUserCount();
}
```

</td></tr>
<tr><td>Side Effects</td><td>See 'Side Effects' under 'Design Patterns'.</td></tr>
<tr><td>Unit Testing</td><td>Unit testing is not mandatory. It supports the goal of testing.
In certain cases unit testing can be an efficient way of testing.

Here are a few examples where unit testing could be useful.

Unit testing can be handy to debug a specific procedure in the system, without having to go through a user interface and several layers in between.</td></tr>
</table>

	Unit testing is also handy for very important functionality that must be guaranteed to work. Price calculations are a good example where unit testing becomes important. Not only must prices always be correct, but also when a price calculation is slightly off, it is easily missed in manual testing. Another case where unit testing comes in handy is when a calculation has many different variations. Sometimes manual testing might only cover 8%, while 50 unit test cover 99% of the situations and can be run each time you release the software. But in many cases simply debugging and testing functionally is still a better choice, for efficiency's sake.

Utilities	Utilities are processes that are not run very often. Utilities contains small programs for IT. E.g. load translations, things to run for deployment.
	
	Framework.WinForms contains a reusable window, SimpleProcessForm, in to start and cancel the process and show progress information.
	
	JJ.Utilities. ...
	
	Description goes here...
	
	Start Cancel
	
	Here is a code example:
	
	<pre>public partial class MainForm : SimpleProcessForm
	{
	    public MainForm()
	    {
	        InitializeComponent();
	    }
	
	    private void MainForm_OnRunProcess(object sender, OnRunProcessEventArgs e)</pre>

```
        {
            var executor = new MyExecutor (x => ShowProgress(x), () => !IsRunning);
            executor.Execute();
        }
    }
```

The MyExecutor class may look as follows and can call its callbacks at its own discretion:

```
    internal class ExecutorDemo
    {
        private Action<string> _progressCallback;
        private Func<bool> _isCancelledCallback;

        public ExecutorDemo(Action<string> progressCallback = null, Func<bool> isCancelledCallback =
    null)
        {
            _progressCallback = progressCallback;
            _isCancelledCallback = isCancelledCallback;
        }

        public void Excecute(IList<MyClass> list)
        {
            if (list == null) throw new NullException(() => list);

            DoProgressCallback("Starting.");

            foreach (MyClass item in list)
            {
                DoProgressCallback("Busy..."); // TODO: include percentage or '3/100' in the text.

                if (DoIsCancelledCallback())
                {
                    DoProgressCallback("Cancelled.");
                    return;
                }
            }
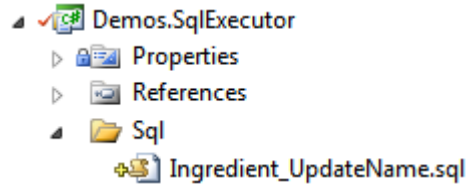
            DoProgressCallback("Finished.");
        }

        private void DoProgressCallback(string message)
        {
            if (_progressCallback != null)
            {
                _progressCallback(message);
            }
        }
    }
```

<table>
<tr>
<td></td>
<td>

```csharp
private bool DoIsCancelledCallback()
{
    if (_isCancelledCallback != null)
    {
        return _isCancelledCallback();
    }

    return false;
}
}
```

</td>
</tr>
<tr>
<td>Validation</td>
<td>See 'Validators' under 'Design Patterns'.</td>
</tr>
</table>

Technology-Specific Aspects

AJAX	We make our own wrapper ajax methods around ones from e.g. jQuery, so we can AJAX with a single code line and handle both partial loads and full reloads. We choose full loads first, before resorting to AJAX. See 'First full load – then partial load – then native code'.
Entity Framework 5	*<TODO: Add story about enabling MSDTC and transactionality.>*

SQL	Executing queries onto a database is normally done through ORM, but if performance is an issue, it can be combined with SQL.
	
	We will not use stored procedures or views. Instead we store SQL files directly in our .NET projects. We put the SQL files in a sub-folder named 'Sql'.
	
	▲ ✓⬜ Demos.SqlExecutor
	  ▷ 🔒 Properties
	  ▷ 🖼 References
	  ▲ 📂 Sql
	    ➕ Ingredient_UpdateName.sql

The classic way of executing SQL in .NET is to use System.Data.SqlClient. Instead, we will use our own SqlExecutor API. With that we can execute SQL in a strongly-typed way, often with only a single code line.

The best method of doing it is to make the SQL file embedded resources:
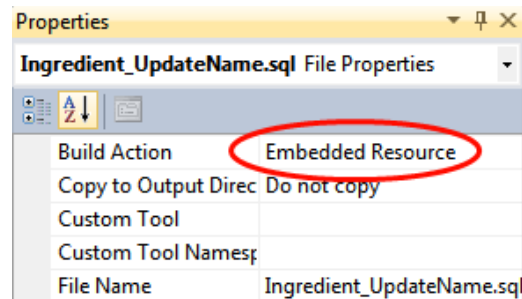
The SQL may look as follows:

```
update Ingredient set Name = @name where ID = @id;
```

Then put an enum in the SQL folder in your .NET project:

Add enum members that exactly correspond to the file names of the sql files:

```
namespace JJ.Demos.SqlExecutor.Sql
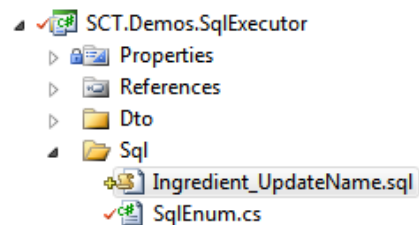{
    internal enum SqlEnum
    {
        Ingredient_UpdateName
```

```
        }
    }
```

You need to create an SqlExecutor as follows:

```
ISqlExecutor sqlExecutor = SqlExecutorFactory.CreateSqlExecutor(SqlSourceTypeEnum.EmbeddedResource, connection,
transaction);
```

We passed the SqlConnection and SqlTransaction to it.

Then you can call a method that executes the SQL:

```
sqlExecutor.ExecuteNonQuery(SqlEnum.Ingredient_UpdateName, new { id, name });
```

The method names are similar to what you might be used to using SqlCommand. You pass SQL parameters along with the SqlExecutor as an anonymous type:

```
new { id, name }
```

The name and type of the variables `id` and `name` correspond to the parameters of the SQL. You do not need to use an anonymous type. You can use any object. As long as its properties correspond to the SQL parameters, they will be correctly used:

```
var ingredient = new IngredientDto
{
    ID = 10,
    Name = "My ingredient"
};

sqlExecutor.ExecuteNonQuery(SqlEnum.Ingredient_Update, ingredient);
```

You can also retrieve records as a collection of strongly typed objects:

```
IList<IngredientDto> records =
sqlExecutor.ExecuteReader<IngredientDto>(SqlEnum.Ingredient_GetAll).ToArray();

foreach (IngredientDto record in records)
{
```

```
        // ...
    }
```

The column names in the SQL are *case sensitive!*

It is smart to let the SQL file names begin with the entity type name, so they stay neatly grouped together:

```
⊿  📂 Sql
      ⊕📄 Ingredient_GetAll.sql
      ✓📄 Ingredient_GetName.sql
      ⊕📄 Ingredient_Update.sql
      ⊕📄 Ingredient_UpdateName.sql
```
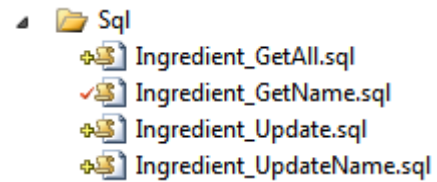
With NHibernate:

If you use SqlExecutor in combination with NHibernate you have to use the NHibernateSqlExecutorFactory instead of the default SqlExecutorFactory:

```
ISession session = ...;
ISqlExecutor sqlExecutor = NHibernateSqlExecutorFactory.CreateSqlExecutor(SqlSourceTypeEnum.EmbeddedResource,
session);
```

This version uses an NHibernate ISession. In order for the SQL to run in the same transaction as the SQL that NHibernate executes, it needs to be aware of the ISession.

It is usually the best choice to include the SQL as an embedded resource, but you can also use files or literal strings.

Files instead of Embedded Resources:
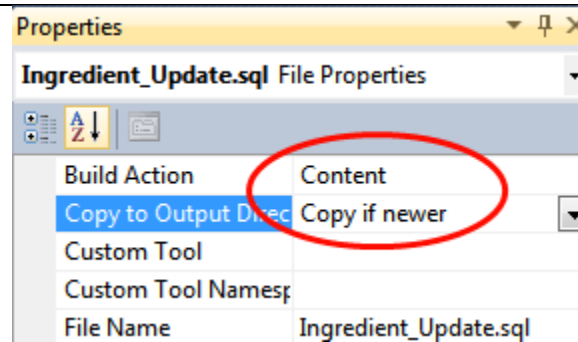
The is the code to create the SqlExecutor and execute an SQL file:

```
ISqlExecutor sqlExecutor = NHibernateSqlExecutorFactory.CreateSqlExecutor(SqlSourceTypeEnum.FileName,
session);
sqlExecutor.ExecuteNonQuery(@"Sql\Ingredient_Update.sql", new { id, name });
```

So the SqlEnum cannot be used anymore. You have to use the (relative) file path.

Strings instead of Embedded Resources:

It is not recommended to use SQL strings in your code! But it is possible all the same using the following code:

```
ISqlExecutor sqlExecutor = NHibernateSqlExecutorFactory.CreateSqlExecutor(SqlSourceTypeEnum.String,
session);
sqlExecutor.ExecuteNonQuery("update Ingredient set Name = @name where ID = @id", new { id, name });
```

In that case no SQL files have to be included in your project.

XML	Always choose XElement (LINQ to XML) over XmlDocument except when you have to use XPath.
	Prefer the XmlHelper methods over using the API's directly, because the helper will handle nullability and unicity better.

Coding Style

This section lists trivial coding rules, that should be followed throughout the code.
Coding standards mostly conform to the Microsoft standard described in the following documents:

http://msdn.microsoft.com/en-us/library/vstudio/ff926074.aspx
http://msdn.microsoft.com/en-us/library/aa260844%28v=vs.60%29.aspx

Casing, Punctuation and Spacing

Rule	Example
Properties, methods, class names and events are in pascal case.	`MyProperty` `MyMethod`
Local variables and parameters are in camel case.	`myLocalVariable` `myParameter`
Fields are in camel case and start with underscore.	`_myField`
Constants in capitals with underscores in between words.	`MY_CONSTANT`
No prefixes, such as "strName".	
Avoid abbreviations.	
For long identifiers, use underscores to separate 'the pieces'.	
Abbreviations of 2 letters with capitals.	`ID`
Abbreviations of 3 letters or more in pascal case.	`Mvc`
Start interface names with 'I'.	`IMyInterface`
Partial view names in MVC should begin with an underscore	`_MyPartialView`

Rule	Wrong	Right
Keep Visual Studio's autoformatting enabled and set to its defaults.		

No extra enters between braces.	`        }`
	
	`    }`
		`        }`
		`    }`

Put enters between switch cases.	`switch (x)`
	`{`
	`    case 1:`
	`        break;`
	`    case 2:`
		`switch (x)`
		`{`
		`    case 1:`
		`        break;`

| | ```
        break;
}
``` | ```
    case 2:
        break;
}
``` |
| No braces for single-line if-statements. | ```
if (condition) { Bla(); }
``` | ```
if (condition) Bla();
``` |
| Loops always on multiple lines. | ```
foreach (var x in list) { Bla(); }
``` | ```
foreach (var x in list)
{
    Bla();
}
``` |
| Use braces for multi-line if's and loops. | ```
foreach (var x in list)
    Bla();

if (condition)
    Bla();
``` | ```
foreach (var x in list)
{
    Bla();
}

if (condition)
{
    Bla();
}
``` |
| Put enters between methods. | ```
void Bla()
{
}
void Bla2()
{
}
``` | ```
void Bla()
{
}

void Bla2()
{
}
``` |
| Each property at least its own line. | ```
int A { get; set; } int B { get; set; }
``` | ```
int A { get; set; }
int B { get; set; }

int C
{
    get { ... }
    set { ... }
}

int D
{
    get
    {
        ...
    }
    set
    {
        ...
    }
``` |

		`}`

Put enters inside methods between 'pieces that do something' (that is vague, but that is the rule).	<pre>void Bla()
	{
	    var x = new X();
	    x.A = 10;
	    var y = new Y();
	    y.B = 20;
	    y.X = x;
	    Bla2(x, y);
	}</pre>
		<pre>void Bla()
		{
		    var x = new X();
		    x.A = 10;
		
		    var y = new Y();
		    y.B = 20;
		    y.X = x;
		
		    Bla2(x, y);
		}</pre>

Each variable declaration on its own line.	<pre>int i, j;</pre>	<pre>int i;
		int j;</pre>

Avoid 'tabular form'. It should only rarely be used. This tabular form will often be undone by auto-formatting. It is non-standard, so it is better to get your eyes used to non-tabular form.	<pre>public int ID { get; set; }
	public bool IsActive { get; set; }
	public string Text { get; set; }
	public string Answer { get; set; }
	public bool IsManual { get; set; }</pre>
		<pre>public int ID { get; set; }
		public bool IsActive { get; set; }
		public string Text { get; set; }
		public string Answer { get; set; }
		public bool IsManual { get; set; }</pre>

Align the elements of linq queries as follows:	<pre>var arr = coll.Where(x => x...).
	    OrderBy(x => x...).ToArray()</pre>
		<pre>var arr = coll.Where(x => x...)
		            .OrderBy(x => x...)
		            .ToArray()</pre>

Use proper indentation	<pre><TODO: Example.></pre>	<pre><TODO: Example.></pre>

Trivial Rules

Rule	Wrong	Right
Give each class (or enum) its own file (except nested classes).	-	-

Keep members private as much as possible.		<pre>private void Bla()
		{
		}</pre>

Keep types internal as much as possible.		<pre>internal class MyClass
		{
		}</pre>

Use explicit access modifiers (except for interface members).	<pre>int Bla() { ... }</pre>	<pre>public int Bla() { ... }</pre>
No public fields. Use properties instead.	<pre>public int X;</pre>	<pre>public int X { get; set; }</pre>

Put nested classes at the top of the parent class's code.	```internal class A
{
    public int X { get; set; }

    private class B
    {
    }
}``` | ```internal class A
{
    private class B
    {
    }

    public int X { get; set; }
}``` |
| Avoid getting information by catching an exception. Prefer getting your information without using exception handling. | ```bool FileExists(string path)
{
    try
    {
        File.Open(path, ...);
        return true;
    }
    catch (IOException)
    {
        return false;
    }
}``` | ```bool FileExists(string path)
{
    return File.Exists(path);
}``` |
Do not use type arguments that can be inferred.	```References<Child>(x => x.Child)```	```References(x => x.Child)```
Use interface types as variable types when they are present.	```List<int> list = new List<int>;```	```IList<int> list = new List<int>;```
Prefer ToArray over ToList.	```IList<int> collection = x.ToList()```	```IList<int> collection = x.ToArray()```
Use object initializers for readability.	```var x = new X();
x.A = 10;
x.B = 20;```	```var x = new X
{
A = 10,
B = 20
}```
Put comment for members in <summary> tags.	```// This is the x-coordinate.
int X { get; set; }```	```/// <summary>
/// This is the x coordinate.
/// </summary>
int X { get; set; }```
Comment in English.	```// Dit is een ding.```	```// This is a thing.```
Do not write comment that does not add information	```// This is x
int x;```	```int x;```
Avoid compiler directives	```#if FEATURE_X_ENABLED
// ...
#endif```	```if (config.FeatureXEnabled)
{
// ...
}```

Do not use them unless you absolutely cannot run the code on a platform unless you exclude a piece of code. Otherwise use a boolean variable, a configuration setting, different concrete implementations of classes or, anything.		

An internal class should not have internal members.
 	
The members are automatically internal if the class is internal. If you have to make the class public, you do not want to have to correct the access modifiers of the methods.
	`internal class A`
	`{`
	`internal void B`
	`{`
	`}`
	`}`
		`internal class A`
		`{`
		`public void B`
		`{`
		`}`
		`}`

Default switch case at the bottom.	`switch (x)`
	`{`
	`    default:`
	`        break;`
	
	`    case 0:`
	`        break;`
	
	`    case 1:`
	`        break;`
	`}`
		`switch (x)`
		`{`
		`    case 0:`
		`        break;`
		
		`    case 1:`
		`        break;`
		
		`    default:`
		`        break;`
		`}`

Prefer .Value and .HasValue for nullable types.	`int? number;`
	`if (number != null)`
	`{`
	`string message = String.Format(`
	`"Number = {0}", number);`
	`}`
		`int? number;`
		`if (number.HasValue)`
		`{`
		`string message = String.Format(`
		`"Number = {0}", number.Value);`
		`}`

Member Order

Try giving the members in your code file a logical order, instead of mixing them all up. Suggested possibilities for organizing your members:

Chronological	When one method delegates to another in a particular order, you might order the methods chronologically.
By functional aspect	When your code file contains multiple functionalities, you might keep the members with the same function together, and put a comment line above it.

By technical aspect	You may choose to keep your fields together, your properties together, your members together or group them by access modifier (e.g. public or private).
By layer	When you can identify layers of delegation in your class you might first list the members of layer 1, then the members of layer 2, etc.

The preferred ordering of members might be chronological if applicable and otherwise by functional aspect, but there are no rights and wrongs here. Pick the one most appropriate for your code.

Miscellaneous Rules

Description	Not Recommended	Recommended

Test class names end with 'Tests'.	`[TestClass]`
	`public class Tests_Validator()`
	`{`
	`}`
		`[TestClass]`
		`public class ValidatorTests()`
		`{`
		`}`

Test method names start with Test_ and use a lot of underscores in the name because they will be long, because they will be very specific.	`[TestMethod]`
	`public void Test ()`
	`{`
	`    ...`
	`}`
		`[TestMethod]`
		`public void Test_Validator_NotNullOrEmpty_NotValid()`
		`{`
		`    ...`
		`}`

var should be avoided. The variable type should be visible in the code line instead of 'var'. Exceptions are:	`var x = y.X;`	
- An anonymous type is used.	`X q = from x in list select new { A = x.A };`	`var q = from x in list select new { A = x.A };`
- The code line is a 'new' statement.	`X x = new X()`	`var x = new X()`
- The code line is a direct cast.	`X x = (X)y;`	`var x = (X)y;`
- The code line is WAAAY too long and unreadable without 'var'.	`foreach (KeyValuePair<Canonical.ValidationMessage, Tuple<NonPhysicalOrderProductList, Guid>> entry in dictionary)`	`foreach (var entry in dictionary)`
- Use var in your **view** code.	`<% foreach (OrderViewModel order in Model.Orders %>`	`<% foreach (var order in Model.Orders %>`
To check if a string is filled use IsNullOrEmpty.	`str == null`	`String.IsNullOrEmpty(str)`
To equate string use String.Equals.	`str == "bla"`	`String.Equals(str, "bla")`

Avoid using Activator.CreateInstance. Prefer using the 'new' keyword. Using generics you can avoid some of the Activator.CreateInstance calls. A call to Activator.CreateInstance should be rare and the last choice for instantiating an object.	`Activator.CreateInstance(typeof(T))`	`T = new T()`

Entity equality checks are better done by ID than by reference comparison, because persistence frameworks do not always provide instance integrity, so code that compares identities is less likely to break.	`if (entity1 == entity2)`	`if (entity1.ID == entity2.ID)`
		
		`// (Also do null checks if applicable.)`

The following data types are not CLR-complient and sould be avoided	`Unsigned types such as:`
	`uint`
	`ulong`
	
	`And also:`
	`sbyte`
		`int`
		`long`
		`byte`

Parameter order:
When passing infrastructure-related parameters to constructors or methods, first list the entities (or loose values), then the persistence related parameters, then the security related ones, then possibly the culture, then other settings.
		`class MyPresenter`
		`{`
		`    public MyPresenter(`
		`        MyEntity entity,`
		`        IMyRepository repository,`
		`        IAuthenticator authenticator,`
		`        string cultureName,`
		`        int pageSize)`
		`    {`
		`        ...`
		`    }`
		`}`

No long code lines <TODO: Describe better.>		
When evaluating a range in an 'if', mention the limits of the range and mention the start of the range first and the end of the range second.	`if (x <= 100 && x >= 10)` `if (x >= 11 && x <= 99)`	`if (x >= 10 && x <= 100)` `if (x > 10 && x < 100)`

Namespace Tips

Avoid using full namespaces in code, because that makes the code line very hard to read:

NOT RECOMMENDED:
```
JJ.Business.Cms.RepositoryInterfaces.IUserRepository userRepository = PersistenceHelper.CreatCmsRepository<
JJ.Business.Cms.RepositoryInterfaces.IUserRepository>(cmsContext);
```

Using half a namespace is also not great, because when you need to rename a namespace, you will have a lot of manual work:

NOT RECOMMENDED:
```
Business.Cms.RepositoryInterfaces.IUserRepository userRepository = PersistenceHelper.CreateCmsRepository<Business.Cms.RepositoryInterfaces.IUserRepository>(cmsContext);
```

Instead, try giving a class a unique name or use aliases:

```
using IUserRepository_Cms = JJ.Business.Cms.RepositoryInterfaces.IUserRepository;

...

IUserRepository_Cms cmsUserRepository = PersistenceHelper.CreateCmsRepository<IUserRepository_Cms>(cmsContext);
```

Names

Boolean Names

Use common boolean variable name prefixes:

Prefix	Example	Comment
Is	IsDeleted	This is the most common prefix.
Must	MustDelete	
Are	AreEqual	For plural things.
Can	CanDelete	Usually indicates what *user* can do.
Has	HasRecords	

If it is ugly to put the prefix at the beginning, you can put it in the middle, e.g.: LinesAreCopied instead of AreLinesCopied.

Method Names

Method names start with verbs, e.g. CreateOrder.
Names for other constructs should not start with a verb.

Common verbs:

Verb	Description

Add	E.g.
	
	    List.Add(item)
	    ListManager.Add(list, item)
	
	In cases such as the last example, it is best to make the list the first parameter.

Calculate	
Clear	
Convert	
ConvertTo	
Create	When a method returns a new object.
Delete	
Execute	
Generate	
Get	
Invoke	
Parse	
Process	
Remove	
Save	
Set	
Try	
TryGet	
Validate	A method that generates **validation messages** for user-input errors
Assert	A method that throws **exceptions** if input is invalid.

Class Names

Class names usually end with the pattern name or a verb converted to a noun, e.g.:

      Converter
      Validator

And they start with a term out of the domain:

      OrderConverter
      ProductValidator

Keep variable names similar to the class names, and end them with the pattern name.

Common 'last names' for classes apart form the pattern names are:

Resolver	A class that does lookups that require complex keys or different ways of looking up depending on the situations, fuzzy lookups, etc.
Dispatcher	A class that takes a canonical input, and dispatches it by calling different method depending on the input, or sending a message in a different format to a different infrastructural endpoint depending on the input.
Invoker	Something that invokes another method, probably based on input or specific conditions.
Provider	A class that provides something. It can be useful to have a separate class that provides something if there are many conditions or contextual dependencies involved in retrieving something. A provider can also be used when something has to be retrieved conditionally or if retrieval has to be postponed until later.

Enum Names

Use the 'Enum' suffix for enum types e.g. OrderStatus**Enum**.
Another acceptable alternative is the suffix 'Mode', e.g. Connection**Mode**, but the first choice should be the suffix 'Enum'.

Collection Names

Collection names are plural words, e.g.:

Products
Orders

Variable names for amounts of elements in the collection are named:

Count

So avoid using plural words to denote a count and avoid plural words for things other than collections.

Event Names / Delegate Names

Event names and delegate names, that indicate what just happened have the following form:

Deleted
TransactionCompleted

Event names and delegate names, that indicate what is about to happen have the following form:

Deleting
TransactionCompleting

UI-related event names do not have to follow that rule:

Click
DoubleClick
KeyPress

Delegate names can also have the suffix 'Callback' or 'Delegate':

ProgressInfoCallback
AddItemDelegate

Sometimes the word 'On' is used:

OnSelectedIndexChanged
OnClick

Or the prefix 'Handle':

HandleMouseDown

DateTime Names

A DateTime property should be suffixed with 'Utc' or 'Local':

StartDateLocal
OrderDateTimeUtc

Prefixes and Suffixes

Suffix	Description
source.. dest…	In code that converts one structure to the other, it is often clear to use the prefixes 'source' and 'dest' in the variable names to keep track of where data comes from and goes to.
existing…	Denotes that something already existed (in the data store) before starting this transaction.
new…	Denotes that the object was just newly created.
original…	Denotes that this is an original value that was (temporarily) replaced.
…WithRelatedEntities …WithRelatedObjects	Indicates that not only a single object is handled, but the object including the underlying related objects.
Versatile…	A class that handles a multitude of types or situations.
…With…	When tou make a specialized class that works well for a specific situation, you could use the word 'With' in the class name like this: - CostCalculator - CostWithTaxCalculator

File-Related Variable Names

Variable names that indicate parts of file paths can easily become ambiguous. Here is a list of names that can be used to disambiguate it all:

Name	Value
FileName	"MyFile.txt"
FilePath	"C:\MyFolder\MyFile.txt"
FolderPath	"C:\MyFolder"
SubFolder	"MyFolder"
RelativeFolderPath (sometimes also called 'SubFolder' or 'SubFolderPath')	"MyFolder\MyFolder2"
RelativeFilePath	"MyFolder\MyFile.txt"
FileNameWithoutExtension	"MyFile"
FileExtension	".txt"
AbsoluteFilePath	"C:\MyFolder\MyFile.txt"
AbsoluteFolderPath	"C:\MyFolder"
AbsoluteFileName	DOES NOT EXIST
FileName**Pattern**, FilePath**Pattern**, etc.	*****.xml C:\temp\BLA_**????**.csv
FileName**Format**, FilePath**Format**, etc.	order-**{0}**.txt orders-**{0:dd-MM-yyyy}***.*

Best Practices

Null-Checks

The most common programming error is a missing null-check.

Code should be strict when it comes to nullability. The general message is: check for null if you expect it. But sometimes you can omit the null-checks to keep your code clean. Here are the rules:

- Entity models are anti-encapsulated: none of the data is protected.
- A data store often guards nullability.

- Validators should also guard nullability.
- Those two things determine whether an entity property is not nullable.
- You might mark the entity property with the word 'not nullable' in its summary.
- For not nullable properties you never have to do null-checks in your business logic, even though in theory null could be assigned.
- List-properties in the entity models require no null checks at all. They should be created in the constructor of the entity class and we simply assume nobody will assign null to it.
- This means that serious business logic should not be executed on entities that have not been validated yet. When you just retrieved entities from the data store, you may assume the data is valid.
- This saves us a lot of null-checks, which makes code more readable.
- The other entity properties are considered nullable.
- For nullable properties, business logic must have an alternative flow. Some business logic could throw an exception if a null is encountered. Other business logic might have to be null-tollerant and skip certain things. (Reflect this in the code by using words like 'Try' and 'IfNeeded'.)
- Null-checks can be omitted if you know that a variable was verified before. For instance: if you throw an exception in the constructor in case an argument is null, you can leave out null-checks in the rest of your class.

See also: 'Error hiding'.

Allow nulls as little as possible. Similar rules also apply to other integrity constraints (e.g. "> 0"), but null-checks are the most common.

Here are rules for null-checks for other constructs:

DTO's	Usually the same rules apply to DTO's as do for entities. Especially if they just transfer data from SQL statements to application logic.
Strings	To check if a string is filled in, use IsNullOrEmpty: if (String.IsNullOrEmpty(str)) throw new NullOrEmptyException(() => str); So this is wrong: if (str == null) throw new NullException(() => str); Another common mistake is this: obj.ToString() This will crash if the object can be null. This is the solution: Convert.ToString(obj)

Value types	Value types, for instance int and decimal, cannot be null unless they are nullable, for instance 'int?' and 'decimal?', so do not execute null-checks on non-nullable value types.

Parameters	Execute null-checks on arguments of public methods. Use NullException (out of Framework.Reflection). You can omit null-checks on arguments of private methods, if the rest of the class already guarantees it is not null.
	
	Avoid allowing null parameters. But if a parameter *is* nullable, you can denote this in several ways.
	
	- Assign null as the default value of the parameter, so it is clear that it can accept null:
	`private void MyMethod(object myParameter = null)`
	
	- Document with XML tags, that it is nullable:
	`/// <param name="myParameter ">nullable</param>`
	`private void MyMethod(object myParameter)`
	
	- Add an overload that does not have the parameter:
	`private void MyMethod() { }`
	`private void MyMethod(object myParameter) { }`
	Ideally in the overload with the parameter, do a null-check.

ViewModels	ViewModels that are passed to Presenters may contain nulls. You can use the NullCoalesce pattern to resolve the nulls before processing the view model object, so that null-checks can be omitted from the rest of the code.
Our own framework API's	For API's in our own framework you can count on an object when you call a Get method. You have to take null into consideration when you call TryGet.

| Your own application code | Conform to that same pattern in your own application code, so you know when you can expect null. |
| Third-party API's | Some .NET API's and third party API's may return null when you call a Get method. Some do not. You have to learn which methods can return null and do null-checks appropriately. |

Bad Practices

Anti-patterns are coding techniques that are not recommended. They are not forbidden, but they are an indication that a better technique may be available. Code smells are vaguer indications that there might be something wrong with your code.

Code smells and anti-patterns can help you review (your own) code.

2 API's for the same thing	Choose one API and stick with it. It is not recommended to use e.g. two different XML API's in your application.
Anti-programming	It seems more efficient to reuse many third party software components instead of programming them yourself. However, your own code might actually be better than that of the third party component. Third party components often come with overhead and bugs and problems with integration so that a custom solution might actually be more efficient. The choice for a third party component might be related to your own inability to program it. That is why it is anti-programming. You always start out as an anti-programmer. However, as you progress, you might start getting better. Do not fall into the trap of thinking that using a third party component is always more efficient than programming it yourself.
Asymmetry	When several pieces of code that do similar things are differently structured, this is an indication that you should make these pieces of code consistent. If these pieces of code do not have a similar structure, you should have a good reason for it and understand this reason and be able to explain it.

	Even though this may seem a vague point, symmetry in code is very important for good software design.
Auto-instatiation	Sometimes auto-instatiation on first use can be replaced by initializing a field in a constructor or type initializer. This performs better becasue it prevents the auto-instatiation 'if' and might make the field only initialize once in the lifetime of the app domain.
'But it works' ('Maar het werkt toch?')	This is the false conclusion that when the output is OK, it must mean that the program is coded well. This is often paired with the argument that additional coding work is a waste of time. A senior programmer might tell a junior programmer to change his code, and a junior programmer might think it is a whole lot of unnecessary work, because his program already worked. Things that could still be wrong with the program are for instance that it might have poor performance, crashes in exceptional cases, produces corrupt data in exceptional cases, is programmed in a way that another programmer does not understand, poor maintainability so that a change takes ages or possibly means it has to be rewritten completely or a change is error-prone, that it is not adapted to contextual changes such as culture or when variables change, they may be hard-codedly interwoven into the code or the assumption that databases will be available when they are not, security leaks, crashes when services are offline, does not give validation or exception messages when something goes wrong, etcetera. These are all examples of what could still be wrong with the code if a program 'seems to work'. Another false argument against doing all this work is the claim that these are just unimportant details.
Consistent stupidity	Too much effort into making code consistent can result in nonsensical code when looking at the individual cases. In a worst case scenario it even results in code that works incorrectly. It is better for each piece of code to make as much sense as possible individually. "Consistent stupidity is still stupidity."
Copy-paste programming	Copy-past programming is the practice of implementing new functionality by copying a old code and then slightly changing it. This is very bad practice and creates a lot of slightly different copies of code, that make it difficult to

	change their mutual functionality. The alternative is to create one generic piece of code that can be used in multiple different ways and not repeat it (see also: 'Do not repeat yourself (DRY)').
	Do note that you are allowed to repeat *trivial* code and two copies of code that must not affect eachother if one of them gets changed.
Correcting input data	Do not correct input data, but require that input data is correctly entered. Code that creates tollerance towards user entry errors can quickly get out of hand, while simply rejecting the use input with a validation message would suffice. It also gives the user more control over what happens, instead of the system's wrongly interpreting the user input.
Delegitis	This is over-use of delegates. Here is an example of a method with too many delegates passed to it. The problem is that it is very unreadable:

```
public void MyDelegator(
    Func<Child, Parent> getParent, Action<Child, Parent> setParent,
    string parentPropertyName, string listPropertyName,
    Func<Parent> createParent, Func<Child> createChild)
{
    // ...
}
```

Solution: create a base class and a derived class that overrides the methods with specific implementations. This may create more classes, but it might be better overviewable.

```
abstract class MyBase
{
    private string _propertyName;
    private string _listPropertyName;

    public MyBase(string propertyName, string listPropertyName)
    {
        // ...
    }

    public void Execute()
    {
```

<table>
<tr>
<td></td>
<td>

```
        // ...
    }

    protected abstract Parent GetParent(Child child);
    protected abstract void SetParent(Child child, Parent parent);
    protected abstract Parent CreateParent();
    protected abstract Child CreateChild();
}
```

</td>
</tr>
<tr>
<td>Dependency injection</td>
<td>

For dependency injection we will not use frameworks like Ninject anymore. Ninject uses a 'magic hat' principle: an object came from somewhere and you have no idea where it came from or if the object is even there. NInject allows you to define a set of implementations of several interfaces centrally and retrieve that implementation from arbitrary places in the code:

```
// Bind it
Bind<IMyDependency>().To<MyDependency>();

// Use it
MyClass obj = new MyClass();

class MyClass
{
    private IMyDependency _myDependency;

    public MyClass()
    {
        myDependency = ServiceLocator.Resolve<IMyDependency>()
    }
}
```

We will not use this technique anymore, because it has serious disadvantages and the object oriented paradigm completely falls apart:

- You get null-reference exceptions, since there are no guarantees that the dependency is actually there.

</td>
</tr>
</table>

- The setup of dependencies is tricky, because you would need to analyse all the code to actually know which dependencies you need to set up, or use an example, which is probably different for your new application.
- You get problems with using multiple instances of the dependency, and it is really hard to figure out where your object came from.
- It takes considerable hours of trouble shooting to set it up or to solve problems and you have a sense of having no control over what is going on.
- Some variations on the technique even require making members public that really need to be private.
- Using constructor arguments in the dependencies is not type safe.
- It is unclear from an class's members and constructors that it is dependent on something, what it is dependent on.
- It creates spaghetti code where everything is potentially dependent on everything else.
- It gets worse because injected dependencies can be dependent on yet again more injected dependencies.
- Instantiation is slower, because it has to go through a framework.
- Also, dependency injection needs a framework that might not be supported on all platforms.

Here is the proper alternative:

```
IMyDependency myDependency = new MyDependency(); // Or another means of instantiation.
MyClass obj = new MyClass(myDependency);

class MyClass
{
    private IMyDependency _myDependency;

    public MyClass(IMyDependency myDependency)
    {
        if (myDependency == null) throw new NullException(() => myDependency);
        _myDependency = myDependency;
    }
}
```

It is simply a combination of interfaces and constructor arguments.

	Now we have accomplished the same thing, only instantiation is explicit and not magic. You know you need to create the dependency (with dependency injection you did not). The dependency is null-checked (with dependency injection it was not). You can use multiple instances in the right places (with dependency injection multiple instantiation is tricky and has limited capabilities). It is faster, because it does not go through a framework and you do not need to include a framework, that might not work on all platforms.
Distortion	When you diverge from a pattern, you are probably not using it right. Find a way to keep the use of a pattern clean. It is an indication that your separation of concerns is not right or another design mistake. Perhaps you are using the wrong pattern, perhaps you are putting the responsibility for something in the wrong spot.
Double negation	If you give a variable name the word 'not' in it, then your code is likely to be less readable, since you might get a lot of double negations like "!not" and such. It is usually a better idea to use the 'positive' name as the variable name.

Empty if-block	Do not do this:
	
	``
	if (condition)
	{
	    <<No code>>
	}
	else
	{
	    <<Some code>>
	}
	``
	
	It looks like you forgot to write code. It looks weird. Either use negation:
	
	``
	if (!condition)
	{
	    // (Some code)
	}
	``
	
	Or do an early return:

| | ```
if (condition)
{
    return;
}

// (Some code)
``` |
Execution order dependence	If methods only work if you execute them in a particular order, why not have one method that executes them in that specific order?
Fluff	A lot of code that does not add much functionality is an indication that a more elegant solution might be possible.
God base class	A base that that is a union of the functionality needed in the derived classes, instead of containing just the basic functionality. Consider moving code to the derived classes that need it, delegating to different helper classes instead of putting everything in the base class, or as a last resort add intermediate inheritance levels, gradually extending functionality.
God-object	An object that is used everywhere and can do anything. Consider splitting it up into multiple classes and only using the classes you need where you need them. It is an indication of bad separation of concerns.
Handy extras	Do not add things to you code (and in particular to interfaces) 'that might be handy for the future'. The opposite is true. Extra code requires maintenance in case of changes. Also: If programmers use these 'handy things', they will be hard to get rid of and then you are stuck with it. It is better to keep the code minimalistic and add the extras at the time that you actually need them. Specialized case: Overloads that are never used, should be removed from the code.

Hatch / 'Doorgeefluik'	A method, that does not do anything but delegate to another method. For example: let's say there is a method GetImage in both an ImageRepository and an ImageManager. All ImageManager.GetImage does is call ImageRepository.GetImage. The thinking error might be that you want to consistently call the ImageManager for everything and that it is a good preparation for the future, because the Manager might add extra rules later. But it usually a better plan to directly call ImageRepository.GetImage and leave out the method ImageManager.GetImage. If you leave in the method that does nothing, then when a deeper layer changes, you'd have to change a lot of pointless layers above it. Also by adding a method to the Manager class, you create the false illusion, that more is done than just retrieving an image, giving you a lessened sense control what is going on. If you see a method that does nothing but delegate to another method you have to consider removing this method.
High-throughput	It is not recommended to let a parameter be both input and output. Usually it is a better plan to let the parameter be input, and not write to it, and to return new output. A parameter's being 'througput' should be an exception, rather than the rule. The reason is that it is often confusing to a programmer calling your method. You usually do not expect the data you pass to the method to get deformed.
Helperitis	Helpers are static methods with static functions that support a specific aspect of programming for which no more than a flat list of methods is required. Helpers are ofcourse helpful, but sometimes you can end up with code in which everything is delegates to helpers, obscuring what is actually going on.
Inappropriate conversions	The architecture contains multiple layers that require converting one type to another, for instance converting a view model to an entity. However, additional conversions such as converting one type of view model to another type of view model are not recommended.
Interface contamination	Interfaces are supposed to be lean. You should keep as much as possible out of an interface. An indication of an interface that is too rich, is for instance that a method has many parameters. You might see a method's name and expect it does not need all those parameters. That indicates a degree of interdependency that is too high. This method's responsibilities might have to be redistributed among different parts of the system. An interface might

	use a type from an API, while you might expect that interface to be API neutral. You might see a data class passed to an interface that you expected to be independent of that data model. These are all indications of interface contamination and the solution is usually to distribute responsibilities differently over different parts of the system. Interface contamination becomes a big problem, when an interface is used in many different places. Then all those parts have a high degree of dependence on things they really have nothing to do with. It also makes interfaces difficult to implement, and poorly reusable.
Keep the names consistent	Use the same name for something everywhere. For instance if the business domain contains one name, do not reinvent a new name for it. Use the same name everywhere in class names, property names and variable names. Do not come up with a new name for things half way the code.

Leaky abstractions	An interface tries to abstract / generalize multiple similar problems into one solution. If an interface exposes the underlying solution, we speak of a leaky abstraction.
	
	For example: this repository interface exposes the underlying NHibernate technology:
	
	    interface IMyRepository
	    {
	        Entity TryGetByCriterion(NHibernate.Criterion.AbstractCriterion);
	    }
	
	The repository interface should not have shown NHibernate-related types, because it is supposed to hide the underlying technology. You can also do too much with the interface now. AbstractCriterion allows you to build any query you want. That is also leaky about this interface. It is the repository's job is to offer a set of optimal queries. With the leaky interface above, a repository cannot do its job anymore.
	
	The following example is better.
	
	    interface IMyRepository
	    {
	        Entity TryGetByCriteria(string name, DateTime dateCreated);

	``` } } ``` You might add extra methods or parameters if more filtering options are needed.
Lying names	Names in code should tell the truth. If a method does more than what the name says, then it is a bad name. If a method name only gives an indication what it does, but in fact it does other stuff too, it is a bad name. If a programmer could not come up with a good name, so just typed something vague, then this is bad practice.
Magic	When a call to a member does more than you would expect and has unintuitive side effects that you do not see. Solution: execute the side effects explicitly, so you see what is going on, instead of letting the side effects go off automatically.
Many properties	It is not a good plan to substitute parameter passing with the use of properties or fields. It is better to let methods use their own parameters and return values, than to let many methods simply control the same set of properties. The reason for this is that preferring properties over parameters, you have less control over where the data comes from and goes to. You have no idea what the output of a method actually is, because it may change any of the properties.
Method not self-sufficient	It can be better to repeat the same work in multiple methods, rather than do the shared work once and pass the result to multiple methods. This might harm performance, but this makes the methods self-sufficient. It is worse for the method to only work if you do very specific work beforehand. It is usually better to then let each of the methods repeat the same work. Performance is the trade-off here, but it makes the methods more reliable. An alternative is to create an instantiatable class that does the shared work in the constructor to not repeat it in the instance methods.
Method too long / class too long	If a method is long, for instance 25+ code lines, consider if it should be split up into multiple methods. If a class is long, for instance 800+ code lines, consider if it should be split up into multiple classes.

	In both cases this usually means that the method or class has too many responsibilities. This code smell can also apply to other code than C# classes and methods.

Nested loops	This is a nested loop:
	
	```\nforeach (var x in list1)\n{\n foreach (var y in list2)\n {\n // …\n }\n}\n```
	
	Nested loops usually come with a performance penalty, because compared to a single loop with n iterations it might have n^2 iterations. It is not always wrong to have a loop in a loop, but you are only comparing two lists, using a hashset or dictionary might be a better solution, changing the n^2 problem back to a *2n* problem:
	
	```\nDictionary<int, X> dictionary = list1.ToDictionary(x => x.ID);\n\nforeach (var y in list2)\n{\n var x = dictionary[y.ID];\n // …\n}\n```

Nesting too deep	Too much nesting in code can be confusing. It can be prevented by splitting the code up into multiple methods. It can also be prevented by using early returns. So instead of: ```\nif (condition)\n{\n // A lot of code…\n // …\n```

<table>
<tr><td></td><td>

```
    // …
    // …
  }
  else
  {
    validationMessages = …;
  }
```

You could do the following:

```
  if (!condition)
  {
    validationMessages = …;
    return;
  }

  // A lot of code
  // …
  // …
  // …
```

This keeps cause and effect closer together, making alternative flows in code less confusing.

</td></tr>
<tr><td>Null-tollerance / error hiding</td><td>

A lot of code contains too much null-tollerance, possibly because of being paranoid about getting exceptions. This is the wrong way to go. You MUST throw an exception. If you build in a lot of null-tollerance you will run into the problem that 'nothing happened, and we have no error message'. Or 'the data is corrupted and we have no error message'. What otherwise would have been a clear error message just turned into a horrible problem to solve, in the worst case we will not even be able to solve it at all.

It can also result in a lot of complex code that tries to recover from a faulty situation, that should never occur in the first place and really should result in an error.

</td></tr>
</table>

	If something is null, that should not be null, an exception MUST be thrown. You have to make your code strict when it comes to faulty data and throw an exception when it is encountered. Exceptions are there to tell us what's wrong.
Spread responsibility	Keep the responsibility for one thing in one piece of code and do not spread it across multiple pieces of code. E.g. when setting defaults for a new object, try to keep that in one spot in the code. (Not to be confused with the 'single responsibility principle'.)

Syntactic sugar	Not always bad practice, but is can be a cause of confusion.
	
	Syntactic sugar is creating a notation that does not add anything, other than a simpler notation.
	
	Object initializers are an example of syntactic sugar. In C# 2.0 you had to initialize an object's properties as follows:
	
	`Cat cat = new Cat();`
	`cat.Age = 7;`
	`cat.Name = "Nala";`
	
	In later versions of C# you are able to do the same thing with the following code:
	
	`var cat = new Cat`
	`{`
	`    Age = 7,`
	`    Name = "Nala"`
	`};`
	
	These two pieces of code do exactly the same thing. The shorter notation introduced is 'syntactic sugar'.
	
	You can also program syntactic sugar into your own code. You can do this by introducing shorter names or a shorter notation, helper classes or implicit conversion operators (which can be very confusing). This may create a concise notation, but breaks the rule 'clarity over brevity'. Often such notations lie a little about what is really going on. So use it sparsely and in most cases go for a more explicit notation.

	(The object initializer notation above actually is the recommended notation, not undesirable syntactic sugar.)

Returning a parameter	Do not return a parameter again. If you write directly to a passed object, there is not need to return it again. So this is wrong:
	
	    A x = Bla(x);
	
	    private A Bla(A x)
	    {
	        x.Y = 10;
	        return x;
	    }
	
	Do this instead:
	
	    Bla(x);
	
	    private void Bla(A x)
	    {
	        x.Y = 10;
	    }
	
	Returning a parameter suggests that new output is created and the parameter is not written to, while neither is true: the input and output are the same object and the input object is changed. This is very unintuitive.

Too many responsibilities	When a class does too many different things, you might want to split it up into separate individual classes. When a method does different things depending on its input, you might want to split it up into different individual methods. Most of the time in the code calling your method, it is already clear which of those different things are needed there, and it does not need to be generic.
Unclear interfaces	Interfaces of classes, methods and other members must clearly show what the member will do. A method's name should say what it does. Preferrably, if the method needs input, it should be a parameter, if a method has output it

<table>
<tr><td></td><td>should be a return value. If an input parameter has an invalid value assigned to it, you should get an exception. The interface should guide the programmer, so there is really only one way to do it, and not several incorrect ways to do it.

Examples of unclear interfaces are:
- A method's input parameters make the method do nothing, instead of throwing an exception.
- You can assign null to one of the parameters and the method will change its behavior completely. A better solution is to have two separate methods.
- The input of a method is a property, while the method is the only one using the property. A better solution is to remove the property, and pass it as a parameter.
- The output of a method is a property. It may be better to actually return the output, rather than assign a property. A programmer may not be able to guess that the effect of the method is that a property is assigned.
- A method does nothing unless a property is assigned.
- A property could have been a constructor argument. The class does nothing or throws exceptions unless the property is assigned, while really it could have been made a mandatory constructor parameter, so the object's state is valid right after construction.</td></tr>
<tr><td>Unused parameter</td><td>Remove parameters from methods if they are no longer used.</td></tr>
<tr><td>Variable not declared where it is used</td><td>For instance: when a variable is used in one place and declare in a totally different place, you might want to move the variable closer to where it is used. This may apply to local variables. This may also apply to using local variables instead of fields.</td></tr>
<tr><td>Variables that change meaning</td><td>When you use a variable for one thing and later overwrite it with semantically something else, it can be confusing to someone reading your code. Consider using a second variable instead.</td></tr>
<tr><td>Constructor calls an overridable</td><td>Calling an overridable member in a base constructor breaks inheritance principles. It creates a chicken and egg problem. Fields in the derived class need to be initialized before running a method, but the field can only be initialized after the base constructor went off, which runs the method! See the following code:

```
abstract class MyBase
{
    public MyBase()
```
</td></tr>
</table>

```
        {
            Execute();
        }

        protected abstract Execute();
    }

    class MyDerivedClass()
    {
        private int _value;

        public MyDerivedClass(int value)
        {
            _value = value;
        }

        protected override Execute()
        {
            // PROBLEM: _value is not initialized yet!
        }
    }
```

The solution is to make Execute() public and insist that that it is called explicitly in the code that creates the instance. Ofcourse if the method is not overridable it would be no problem, and if the method was not called in the base constructor it would be no problem, and but calling an overridable member from the base class's constructor could mean trouble.

(Validation framework uses this anti-pattern however, because there is too much danger that someone forgets to call Execute. It uses a trick to be able to initialize the members anyway, but it is quite dirty.)

Magic numbers / magic strings	<TODO: Describe. >
Magic default	<TODO: Describe. >
Hard-coding and soft-coding	<TODO: Describe.>
Kama-sutra pattern	So many overloads you cannot see which to pick.

Service Architecture

What has been described so far is the *application architecture*. A second part of the software architecture is the *service architecture*, which is mainly about linking systems together. This section is an addition to the documentation with regards to the service architecture. Currently the services are programmed using WCF.
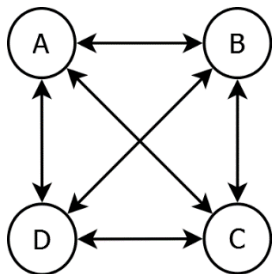
The ESB Concept

ESB stands for Enterprise Service Bus, which is a system for exchanging data between different systems of different organizations in different formats with different protocols. Central components are used to make integration between these systems more manageable. One important concept is the canonical model.
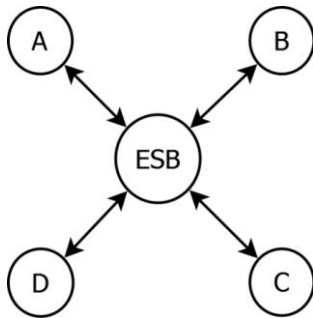
Canonical Model

The canonical model helps us exchange data between systems. Data can be retrieved from multiple systems and is converted to a canonical form, so that the same code may be reused for data that comes from various systems. The canonical model should be as pure and general as possible, so indeed information of any system can fit into it with very little modification.

Less Integration Code

Say you have 4 systems: A, B, C and D and you want to connect all 4 of them together. Theoretically you would have to write 12 different message conversion as you can see from the arrows in the diagram below:
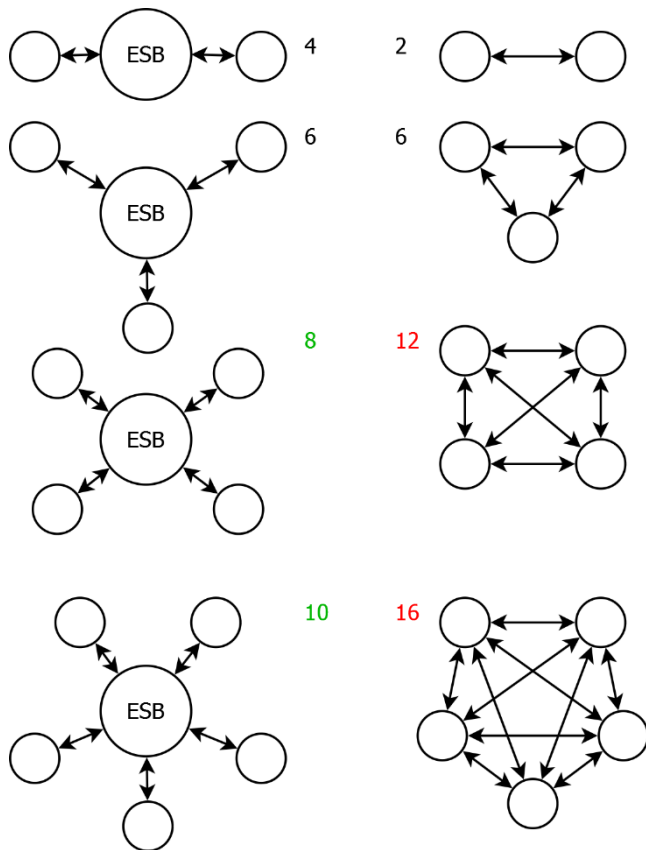

By connecting a system to the ESB, instead of connecting individual systems together, you have to implement only 8 different message conversions as you can see from the arrows below:

You just saved yourself 33% of the work!
With every system you add to your ESB it gets better as you can see from the numbers below that indicate the amount of message conversions.

The first integration between 2 systems you program using your ESB you actually program more message conversions, but with the next system it is already a tie between ESB and no ESB. The 4th integration you introduce you will have saved 33% of the overall work.

It gets better with each system you introduce in your ESB. When messages from a system are converted to and from the canonical model, you can automatically connect it to all the other systems.

Clearer Integration Code

But it gets better. You save yourself even more work. The conversion code from message to canonical model is often easier than converting from one system's format to the other system's format, because instead of converting from one quirky format to another quirky format, which is really difficult to do, you convert from one quirky format to a very clear format, which is much easier to program.

In Practice

In practice not every system sends every type of message back and forth to every other system. And sometimes the messaging is not bidirectional but one way only. But the benefits of the ESB still hold and you will link systems together with less code and less effort than custom programming every integration between two systems.

Standard ESB vs Custom ESB

There are standard Enterprise Service Bus software packages available. Yet, we choose to build a custom one ourselves. The concepts are not that hard to implement. And generic ESB's are really complex and have a steep learning curve, require training, specialists. This all while you are going to have to custom program much of the message conversion code yourself anyway, and design your own canonical model, which is basically all of the work. Therefore we build it ourselves.

ESB Model

On top of a canonical model, we need more facilities. The ESB model will offer a model for administrating connection settings and register enterprises that can log in to our system to get access to our services.

Next will be listed the main entities of this model.

Enterprises

Every enterprise involved in our service architecture is registered in out ESB database. Some of these enterprises will actually log into our system. Those will get an associated User entity with (encrypted) credentials stored in it.

ConnectionTypes

Every type of connection between systems is registered in a table of ConnectionTypes. Each ConnectionType is a very specific way of integrating with a system, with a specific messaging protocol, message format and implementation.

Connections

Every individual connection between two parties is registered in the Connection table with the connection settings stored with it. Each connection has an associated ConnectionType that indicates what type of integration it is. Note that some connections are not between parties but involve only one party. Connections do not have to be complete messaging implementations. Sometimes they are simply database connection settings or even the path of a network folder.

Keys

Often systems have different identifiers for e.g. orders or other objects. There is often a need to map a reference number from one system to the reference number of another system. The ESB model has entities and logic to manage these key mappings.

Transmissions

Optionally you can log the transferred messages that went over a connection. Do note that logging all messages can significantly impact performance and storage requirements so use it sparsely.

Service Implementations

The implementation of a service involves mostly message transformation and transmission. Data is received through some communication protocol, the message format is parsed and then converted to a canonical model. Conversely, canonical models are converted back to a specific message format and the sent over a communication protocol.

Multi-Dispatch

The content of a canonical model can determine what service to send it to. For instance, one canonical Order might have to be sent to one supplier using their own specific integration protocol, another order might simply be e-mailed to the supplier. The service architecture enables you to retrieve a message from one system, for instance an order, and then send that message to an arbitrary other system. That is part of the power of the canonical model, where multiple systems' messages being converted to canonical model, enables all those systems to communicate with eachother.

Namespaces

These namespaces use a hypothetical Ordering system as an example.

JJ.Services	Root namespace for web services / WCF services

JJ.LocalServices	Root namespace for windows services. (Not part of the service architecture, but this is where that other type of service goes.)
JJ.Data.Canonical	Where are canonical entity models are defined.
JJ.Data.Esb	Entity model that stores Enterprises, Users, ConnectionTypes, Connections, etc. Basically the configuration settings of the architecture.
JJ.Data.Esb.NHibernate	Stores the Esb entity model using NHibernate.
JJ.Data.Esb.SqlClient	SQL queries for working with the stored Esb entity model.
JJ.Business.Canonical	Some shared logic that operates on canonical models.
JJ.Business.Esb	Business logic for managing the Esb model.
JJ.Services.Ordering.Interface	Defines interfaces (the C# kind) that abstract the way messages are sent between different ordering system. These interfaces use the canonical models.
JJ.Services.Ordering.Dispatcher	Makes sure messages (orders, price updates) are received from and sent to the right system depending on message content.
JJ.Services.Ordering.Email	A specific implementation of an ordering interface, behind which we send the order by e-mail.
JJ.Services.Ordering.SuperAwesomeProtocol	A specific implementation of an ordering interface, behind which we implement the hypothetical 'super awesome protocol' for sending orders.
JJ.Services.Ordering.Wcf	A WCF service that allows you to communicate with the multi-dispatch ordering system.
JJ.Services.Ordering.Wcf.Interface	Defines the interface of the WCF service that allows you to communicate with the multi-dispatch ordering system. This service interface can is used by both service and client.
JJ.Services.Ordering.Wcf.Client	Allows code to connect to the WCF service using the strongly typed service interface.
JJ.Services.Ordering.JsonRest	Exposes the multi-dispatch ordering service using the Json and Rest protocols.
JJ.Services.Ordering.WebApi	There is no reason Web API should not be involved in this service architecture, in fact, the idea of WCF being the default for service, might not be a very long-lived.
JJ.Presentation.Shop.AppService.Wcf	A special kind of service is an app service, that exposes presentation logic instead of business logic and returns ViewModels.

Service-Related Patterns

Facade	An interface behind which a lot of other interfaces and classes are used, with the goal of simplifying working with these systems. This concept is used in this architecture to give a service interface an even simpler interface than the underlying business logic has. It may hide interactions with multiple systems, and hide infrastructural setup.

IsSupported	A service environment may contain the same interface for accessing multiple systems. But not every system is able to support the same features. You could solve it by creating a lot of different interfaces, but that would make the service layer more difficult to use, because you would not know which interface to use. Instead, you could also add 'IsSupported' properties to the interface to make an implementation communicate back if it supports a feature at all, for instance: void IConnectionToSupplier.UpdatePrices(); bool IConnectionToSupplier.UpdatePricesIsSupported { get; } Then when running price updates for multiple systems, you can simply skip the ones that do not support it. Possible a different mechanism is used for keeping prices up-to-date, possibly there is another reason why price updates are irrelevant. It does not matter. The IsSupported booleans keeps complexity at bay, more than introducing a large number of interfaces that would all need to be handled separately.

Design Principles

The design concepts mentioned here are abstract principles that you can use in software design.

Separation of Concerns

This is the concept that you split your code into pieces and create separate classes and methods. It is perhaps the single most important design principle of this software architecture.

Separation of concerns can be a split up into functionalities, such as code that handles a whole order and code that handles a separate product. The split up into functional concerns is usually similar to the split up into entities, for instance entities like Order, Product, Customer, but this is not necessarily leading for the split up into functionality.

Separation of concerns can also be applied to technical aspects, such as validation, calculation and security. For instance: you can split up the code to check the validity of an order's data from the code that calculates the total price of the order. The split up into technical concerns is usually similar to the split up into *design patterns*.

Classes

In this architecture we apply both a split up into functional and technical aspects, creating a 2-dimensional separation of concerns. This produces a matrix of classes:

	Dto	**Mapping**	**Validator**	**ViewModel**	**Presenter**	**…**
Order	OrderDto	OrderMapping	OrderValidator	OrderViewModel	OrderPresenter	…
Product	ProductDto	ProductMapping	ProductValidator	ProductViewModel	ProductPresenter	…
Customer	CustomerDto	CustomerMapping	CustomerValidator	CustomerViewModel	CustomerPresenter	…
…	…	…	…	…	…	…

Plus: you can have specialized variations of these classes, for intance: OrderEditPresenter, SubscriptionProductValidator.

This is a very maintainable structure, because everybody that understands the system of organization, can find the code exactly where he would expect it. A change has low impact, because it can at most impact code that references that specific class. Code is better reusable and recombinable. E.g. you can reuse the same validations in multiple places or use specific validations in specific situations.

Using this split up into classes could impact data integrity, since every class can be independently used, and there is not one thing that guards all the rules. The solution is to use Manager classes that guard (most of) the integrity rules. The separate concern itself is actually better guarded, since it cannot get entangled with other code, because it is a separate class.

Assemblies

The separation into technical and functional concerns extends further than the class split-up. It can also be noticed in the assembly subdivision. Those also have a 2-dimensional separation of concerns:

These are the functional concerns:

JJ.Data.**Ordering**.NHibernate
JJ.Business.**Ordering**.Validation
JJ.Presentation.**Cms**
JJ.Presentation.**Cms**.Mvc

And these are the technical concerns:

JJ.**Data**.Ordering.**NHibernate**
JJ.**Business**.Ordering.**Validation**
JJ.**Presentation**.Cms
JJ.**Presentation**.Cms.**Mvc**
JJ.**Presentation**.Cms.**Mobile**

The assemblies are split up by functional domains.
There are separate assemblies for presentation, business and other main layers.
There are separate assemblies for specific protocols and technologies.

It is clear from the assembly name which technique is used, and what the functional domain is and the position within the software layers.

The result of this split up is that we are not stuck with a 1-to-1 relation between an application and its platform.

For instance: if an Ordering back-end was programmed to use a very specific persistence technology, you might only use it with NHibernate and an SQL Server database. You could not use the entity model on a platform that does not support this (e.g. mobile platforms). If a Cms front-end is programmed to specifically use MVC, it can only be deployed as a web site and not as a Windows application or mobile app.

By further splitting up our assemblies we can reuse the Ordering back-end in multiple front-ends. Furthermore: a single front-end could be deployed to either web or mobile platform and we can store entity models differently depending on the infrastructural context. On a mobile platform we might store an entity model in XML, while in a web environment we might store things in SQL Service using NHibernate.

Other

Many of the principes mentioned have pros and cons that need to be weighed off in every decision about the software design.

Abstract / concrete	Abstract means that instead of referring to specific items in a set, you refer to a set of items by stating what they have in common. Concrete means talking about a very specific item in the a set.
	In software programming abstraction means you can generalize multiple problems and offer a single solution for it.
	It is an art to pick when to abstract problems or when to handle a concrete problem. Both have their benefits. In one case abstraction may prevent complexity, while in another case being specific prevents complexity.

	A concrete problem is easier to work out, better to understand, and can tap into specific requirements. On the other hand, abstracting a problem makes you able to write code once and apply it to many different situations. It is harder to write, but might be more maintainable, because it is much less code.
Blackboxing and whiteboxing	Blackboxing means that you put something in a little machine, something comes out, but you cannot see how it was processed exactly.
	Blackboxing is the use of encapsulation to hide complexity. It is also a concept of creating an interface in front of an implementation: hiding what is exactly done.
	Blackboxing can create a lower degree of dependence between different parts of a system, making things easier to change. It is also present in the concept of interfaces, allowing you to hide multiple implementations behind a similar interface.
	Sometimes blackboxing creates problems, because not seeing exactly what is going on is a big downside. Being open about what happens could give a programmer the overview and control he needs. This is the concept of whiteboxing.
	Both concepts are important and have their place. Whether blackboxing or whiteboxing is the right way to go, should be evaluated on a case-by-case basis.
	Simply giving something a name that reveals its inner workings is a common form of whiteboxing.
Bottom-up and top-down	Bottom-up design means you first design the lower layers of a system, for instance the data model and gradually work your way up to the front-end. You can also say bottom-up design is starting with the smaller parts and working your way up to creating bigger and bigger parts out of it.
	Top-down design means you first desing the higher layers of the system, for instance the front-end and gradually work your way down to the data model the little details. You can also say top-down design is starting to think about the bigger parts first and then gradually working out smaller and smaller details.
	No method is best. They are simply two different strategies to attack a problem.

Cartesian product of features problem	Say you have some behaviors that you want a class to either have or not have. What if you want some derived classes that either have or do not have that feature in it. Then you would get as many derived classes as 2 to the power of the number of features. If you have 4 features, you would need 2^4 = 16 derived classes, with each of the features either turned on or turned off. In cases like this it is hard to come up with a good inheritance structure, because neither feature builds on top of eachother. You could make class variations WithFeature1, WithoutFeature1, WithFeature2WithFeature1, WithFeature2WithoutFeature1. All very awkward. Arbitrarily Feature1 was picked to be more basic than Feature2. Also: you would have to repeat the code of Feature2 in two derived classes! Another alternative is also not so good: building a base class that simply has all features in it and derived classes having the feature either turned on or off. This would be called the 'god base class' anti-pattern. It would break the way you work with base classes, since base classes should be more basic with less features in it than derived classes; base classes should not have more features than derived classes. It does not just apply to turning features on and off. If you have 4 variation on a feature and you want to combine it with one out of 4 variations of another feature, then you have 4^2 base classes and which feature will be in the deeper base class? It is the same situation as the problem as described above. This is a weaknesses of inheritance, that makes it so that inheritance should not always be your first choice in constructs to solve your problem. The Inheritance-Helper pattern may solve some of the issues.
Chicken and egg	<TODO: Describe.>
Clarity over brevitity	A longer name in code is better than a short, inspecific one. Even through you may think brevity supports readability, if it creates ambiguity, a longer, unambiguous name usually works out better.
Do not repeat yourself (DRY)	This is the principe that you should not repeat code. It is a general rule that you must then put the shared code in a separate class or method, and reuse the same code in multiple places. But keep in mind that there are exceptions. Very trivial things can be repeated, and the same code might be repeated in placed where a change to one copy of the code, should not affect other parts of the system.
'Too difficult' principle	'Don't be too hard on yourself' principle / 'It can't be that hard' principle / 'It is not allowed to be hard':

	Allow yourself to admit, that implementing it a certain way is just going to be too difficult (for you). Look at it another way: if it is that difficult, perhaps there is a simpler solution, that you have overlooked. This may help you keep an open mind for other solutions. Do not let this be an excuse for laziness. Just keep an open mind. If things become difficult to implement, think back to the core of the problem and that if the problem sounds simple, the solution might be too. (Large gray area.)
First try specific, then try generic	It is often a good solution to design something generic, rather than something that only works for one specific situation. But sometimes it is hard to do this. A strategy can be that when you have the feeling a generic solution is appropriate, but you cannot figure it out, to first develop a specific solution, and then refactor it to become more general.
Granularity	Granularity can be compared to sand. Large pebbles are large granules, while fine sand is made up of a small granules. In code it means that a piece of code might tap into a big object, while it really only needs to depend on a smaller object. A piece of code may use a specialized object, while it can tap into a more generalized form. It can also be expressed as 'defined at the wrong level'.
Loose coupling	Loose coupling or a low coupling is the concept of keeping a low degree of dependence between things. Tight coupling or high coupling means many things are tightly dependent on eachother, making it hard to change one of those things without breaking or changing the other things. The benefit of loose coupling is that a change to one thing affects a minimum of other things. Low coupling does not mean that the total amount of links between things is lower. The amount of links between things may actually go up. The low degree of coupling is about an individual type's links to other types. That is what gives us the benefit of one change only affecting a limited set of other things. One technique of limiting the degree of coupling is the use of interfaces. By letting code talk to an interface rather than directly to a specific implementation, this makes the code dependent on that interface without being directly dependent on multiple concrete classes. This makes you able to write one piece of code that handles multiple concrete things, which makes that code dependent on one type rather than the union of concrete implementations.

	A symptom of high coupling is what happens if you hit Shift-F12 in code ('Find all references'). If you get a whole lot of results, you have a case of high coupling and you might be in trouble. If you get very little results it is a sign or low coupling and you can make a sigh of relief. There are cases where high coupling is normal. For instance in case of base classes, combinator classes, framework classes, simple types, canonical models.
Power of abstraction / power of generalization	When you are able to generalize multiple problems into a single solution, you can code something once and solve multiple problems at the same time. The other side of it is, that it is difficult to abstract multiple problems into a more general problem. Sometimes it is also difficult to understand the solution, because it requires the same abstract thinking. However, by doing it you can save a lot of work and complexity.
Quick and dirty / dirty	<TODO: Describe.>
Ripple-effect	This is an analogy to throwing a rock in a pond. If you throw a rock in a pond, a wave is created that goes through the whole pond. In software programming, it means that if you change one piece of code it may require you to adapt many other pieces of code. It can be an indication of a bad design choice where too many things directly dependent on eachother. It is not always a bad design choice, but simply a hard to prevent high-impact change.
Substractive and additive	Also called 'inclusive' or 'exclusive'. Substractive or additive can be a strategy in programming. Substractive starts with everything and then you start excluding things. Additive starts with nothing and then you start adding things. An example is security. It is often better to use an additive approach and start with no user rights at all and gradually add rights. Another example might be storing object structures. You might create a storage mechanism that stores everything unless you exclude something, or you might create a storage mechanism that stores nothing unless you explicitly specify it is included.

Toilet-role principle	Someone reading you code will look at it through the hole of a toilet-role, seeing only a small piece of code at a time. This means smaller pieces of code must make sense on their own. Someone maintaining or correcting your code should not first need to understand 1000'nds of lines of code before being able to correct a minor problem. There are many coding style tricks and design techniques to support this goal, that are talked about in this documentation.
Trade-offs	Every technique in software development has pros and cons. It is the job of the software designer to weigh off all the pros and cons of every possible design choice and come up with a balance best suited to the situations, that will make us run into the least problems in the future. A striking example is the principe of generalization which is good, and the principe of low coupling which is good. There are cases where high coupling is normal. For instance in case of base classes, combinator classes, framework classes, simple types and canonical models, a high degree of coupling with these types can be expected. Basically when you generalize and make something very reusable, you can automatically expect a high degree of coupling with it, because it is reused so often. That is another reason why generalized solutions should be of such high quality. There *are* techniques that in general are bad, and techniques that in general are good, but there also good principles that contradict eachother, that need to be weighed off in every design decision. The danger of such a large gray area, is that people thing they can just do whatever. But you should not do just whatever. You should learn the pros and cons of things and do a careful weigh-off every time. However, some things are generally bad and some things are generally good.

Database Conventions

Developing a Database

Developing a database generally involves the following steps:

- Create tables and columns
- Add primary keys

- Limit the nullable columns as much as possible
  (but keep them nullable where it functionally makes sense)
- Make foreign keys on columns that link to other tables
- Add indexes on foreign keys columns
- Add unique indexes
  (However, sometimes ORM's will trip over unique keys at which we promptly remove the unique constraint.)
- Add indexes to search columns and alternative keys
- Add indexes for problem queries

Keep in mind to limit the use of 'exotic' data types.

For instance: if a number would fit in a tinyint, use an 'int' anyway, because this saves the system a lot of casting, and a 32-bit number is better for memory / disk alignment, which is better for performance. Chances are, a system will reserve 32 bit for a 8-bit number anyway to accomplish this memory alignment. But this is just an example.

Here are the recommended data types:

- bit
- int
- decimal
- float
- real
- datetime
- nvarchar
- uniqueidentifier
- varbinary

Only if you need a bigger range:

- bigint
- datetime2

Naming Conventions

Object Type	Example Name
Database name	ShopDB
Tables	MyTable
Columns	MyColumn
ID column	ThingID
Indexes	IX_MyTable_MyColumn
Primary keys	PK_MyTable
Foreign keys	FK_MyTable_OtherTable Or when there are multiple relations between the same tables: FK_MyTable_*<OtherTable_ColumnName_MinusID>* FK_MyTable_ThingA FK_MyTable_ThingB
Unique keys	When not many columns: IX_MyTable_MyColumn_Unique When many columns and only one constraint in the table: IX_MyTable_Unique
Stored procedures	SP_DoSomething / spDoSomething
Functions	FN_DoSomething / fnDoSomething
Triggers	TR_MyTable_OnInsert / trgMyTable_Insert / …

Rules

Do not use the following object types, because these things are managed in .NET:

- Views
- Stored Procedures
- Functions
- Triggers

- Defaults
- Check constraints
- Computed columns
- Cascade rules
- Synonyms
- Assemblies
- Types
- Rules

For new databases, prefer int's as primary keys over guids, because guids create performance penalties throughout the software stack.
Only use additional guid columns as an alternative key for entities that need to be unique across multiple systems or databases. Do not forget to put an index on the guid column. Prefer surrogate keys rather than complicated composite keys. Prefer auto-incremented ID's, except for enum-like tables.

For development databases use the "DEV_" prefix, e.g. DEV_ShopDB.
For test use the prefix "TEST_" and for acceptance use the prefix "ACC_". For production use no prefix at all.
On development databases add the user dev with password dev. For test add the user test with password test. For acceptance you might use specific user names depending on security demands, otherwise add user name acc with password acc. In production databases use the administrator user's password with the administrator password for databases or create a separate user name for production with a strong password.

Upgrade Scripts

Database upgrade scripts are managed as follows.

Excel Sheet

Each database structure gets an Excel in which all the upgrade SQL scripts are registered.
The Excel sheet and SQL scripts are put in a Visual Studio project to manage them easily.
Always edit the Excel in the dev branch, because Excels cannot be merged.

The name of an SQL file has a specific format:

    2014-08-28 040 ShopDB Supplier.Name not null.sql

So it has the format:

{Date} {Number} {DatabaseStructureName} {DatabaseObject}{SubDatabaseObject} {Change}.sql

	Description	Examples
Date	Use the format yyyy-mm-dd	2014-08-28
Number	Use 3 digits and count in 10's so you might insert one in betweeen	040
DatabaseStructureName		ShopDB
DatabaseObject	A table name or index name or other database object name	Supplier IX_Supplier_Name FK_Supplier_Branch
SubDatabaseObject	Optional. Usually a column name	.Name
Change	Optional. Usually left out. You can sometimes mention a specific change, but be brief.	not null

In the Excel, add a column for each database instance for that database structure. There can be different databases with the same structure for different staging areas (dev, test, acc, prod) or a database for different customers or databases running on different servers. Put 'TRUE' (or 'WAAR' in Dutch) where the upgrade script has been executed. For instance:

OrderDB Structure Changes	SCRIPTS	DEV		TEST		ACC		PROD		
		10.40.10.12	10.40.10.121	88.22.55.12	88.22.55.12	88.22.55.12	88.22.55.12	213.51.23.142	213.51.23.146	
Script		OrderDB	ShopDB	OrderDB	ShopDB	OrderDB	ShopDB	OrderDB	ShopDB	Release Date
2015-01-23 010 OrderDB Order.DeliveryDateTimeUtc.sql	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	2015-02-16
2015-01-23 010 OrderDB Order.DeliveryDateTimeUtc not null.sql	TRUE	TRUE	TRUE							

Also include a column saying whether you have scripted it at all (for if you are in a hurry and have no time to script it). A release date column is also handy, to get some sense of when things went live.

Exceptional Cases

For upgrades that should only be executed on a specific database, put 'N/A' (or 'N.V.T.' in Dutch) in the appropriate spread sheet cell.
You can also add something to the SQL file name to indicate this:

2015-01-23 010 OrderDB SHOPDB ONLY Order.DeliveryDateTimeUtc.sql

Some things should be done manually and not with SQL. Those actions should also be mentioned in the Excel:

2015-01-23 020 OrderDB OrderID Identity Yes DO MANUALLY

If a script requires that you be extra careful, you can mention this as follows:

2015-01-23 010 OrderDB Order.DeliveryDateTimeUtc.sql CHECK MANUALLY
2015-01-23 010 OrderDB Order.DeliveryDateTimeUtc.sql EXECUTE SEPARATELY

But be sparse with that, because the person running the script might not actually know what it is he is supposed to check and will feel uneasy executing this script since it is obviously so dangerous, while he has no idea why.

Summary

This section covered:

- Visual Studio project
- Excel sheet
- SQL script name format
- Upgrades for specific databases and manual upgrades

Scripts

The individual upgrade SQL scripts should not contain GO statements. GO is not an SQL keyword, it is a Management Studio command telling it to execute the script up until that point. What must be separated by GO statements in Management Studio must be split up into multiple SQL files in the database upgrade scripts.

Also get rid of any automatically generated SET ANSI_NULLS ON and SET QUOTED_IDENTIFIER ON statements. Those are the default behavior anyway, and it just add unnecessary fluff to your scripts. Also: SET ANSI_NULLS OFF will generate an error in future versions of SQL Server anyway.

The upgrade scripts should be incremental: DO make assumptions about the previous state of the database structure and script a specific change. Do not write scripts like 'if not exists' then add, or 'drop and create table' scripts, because you may be throwing away data, or execute things on the wrong database. It is better to make a specific change and *not* be tolerant to differences.

DO NOT script changes from Identity Yes to Identity No or the other way around. Changes in the Identity property of a column require recreating the whole database table. If you script it now, executing it onto a database does not only add the Identity Yes property, it will also restore the whole table structure to the state it had at the time you scripted the Identity.

Summary

This section covered:

- No GO statements
- Split up into separate files
- Incremental scripts (no 'if exists' checks or drop and recreate).
- DO NOT script Identity Yes and Identity No

Deployment

To deploy multiple database structure changes you can use the Excel.
Always edit the Excel in the dev branch, because Excels cannot be merged.

OrderDB Structure Changes	SCRIPTS	DEV		TEST		ACC		PROD		
		10.40.10.12	10.40.10.121	88.22.55.12	88.22.55.12	88.22.55.12	88.22.55.12	213.51.23.142	213.51.23.146	
Script		OrderDB	ShopDB	OrderDB	ShopDB	OrderDB	ShopDB	OrderDB	ShopDB	Release Date
2015-01-23 010 OrderDB Order.DeliveryDateTimeUtc.sql	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	2015-02-16
2015-01-23 010 OrderDB Order.DeliveryDateTimeUtc not null.sql	TRUE	TRUE	TRUE							

You can easily see which scripts are still to be executed onto the database.
After you have executed them, put TRUE in the appropriate spread sheet cells.

You could execute the scripts one by one, but there is a handier, safer way to do it.

With some creative copying and pasting the SQL file names, you can create a composite upgrade script like this:

```
begin try
    print 'Begin transaction.';
    begin transaction;

    declare @verbose bit = 0;
    declare @folder varchar(255) = 'C:\JJ\Install\SqlScripts';
```

```
        exec spExecuteSqlFile @folder, '2015-01-23 010 OrderDB Order.DeliveryDateTimeUtc.sql', @verbose;
        exec spExecuteSqlFile @folder, '2015-01-23 010 OrderDB Order.DeliveryDateTimeUtc not null.sql', @verbose;

        --print 'Rolling back transaction.';
        --rollback transaction;

        print 'Committing transaction.';
        commit transaction;
    end try
    begin catch
        print Error_Message();
        print 'Rolling back transaction.';
        rollback transaction;
    end catch
```

This safely executes all changes in a single transaction and shows error information if something goes wrong.

This does require you to add the following stored procedure to the database:

```
create procedure spExecuteSqlFile(@folderPath varchar(255), @fileName varchar(255), @verbose bit = 0) as
begin
    set nocount on;

    print 'Executing ''' + @fileName + '''';

    declare @filePath varchar(255) = @folderPath + '\' + @fileName;

    declare @readFileSql varchar(1024) = 'select BulkColumn from openrowset(bulk ''' + @filePath + ''', single_blob) x;'

    declare @temp table (contents varchar(max));
    insert into @temp exec (@readFileSql);

    declare @sql varchar(max);
    set @sql = (select top 1 contents FROM @temp);

    -- Remove BOM from UTF-8.
    if (LEFT(@sql, 3) = 'ï»¿') set @sql = RIGHT(@sql, LEN(@sql) - 3);

    exec (@sql);

    if (@verbose = 1) print @sql;
end
```

Summary

This section covered:
- Composite upgrade scripts
- spExecuteSqlScript

Server Architecture

The server subdivision is subject to the needs of the organization, so this overview functions as a good example, that you may or may not implement exactly like this. The main concerns are safeguarding and keeping things optimal. Economics might force you to look for alternatives, but from a technical point of view the full set of servers with recommended configuration is advised. Cutting corners might make your IT run less efficiently, which would translate to cost overhead too.

<TODO: Mention the split-up into a C: and D: drive.>
<TODO: Reconsider the sizes of the development workstation drives after some cleaning up and counting disk space and considering extra dev tool requirements.>
<TODO: Consider the machine configuration needs in more detail.>

Stage	Name	Remarks	Configuration Focus Points
Development	Database server	Stores a development copy of all the databases we use.	SQL Server, decent performance, particular focus on having enough RAM.
Development	App server	Where development can use a shared FTP server if needed, run long processes to aleviate the development workstations. Can also host shared web services, be it third party, be it internally developed ones, even though for that last thing it is usually better to run it on the development workstations.	IIS, preferrably many-core. SQL Server installation is advised, for delegating number crunching from the main development database server to another server. RAM is also important, since heavy number crunching processes may use a lot of memory.
Development	Source control server	For storing the source control database, running the source control services, running builds,	TFS. Must be decent configuration for each checking requires a heavy process to run, and

		unit tests and code analysis upon each check-in.	the development team has to be able to work efficiently.
Development	Workstations	Each software developer's own machine.	Two 21" monitors. No laptops, those run 2x slower. Core i5 for junior and medior developmers. Core i7 for senior developers and sofware architects, since they will more commonly work with larger solutions. At least 8 GB RAM, so you can run large-cache applications and services. SSD for of 256 GB. Split up into a C: drive for windows <specify size> and a D: drive for data, main source code, but also aother frequenty accessed things. An extra 'spinning disk' drive with at least 256 GB storage for amont other things the ability to hold large database backup files.
Development	Laptop	One laptop for a whole team, just to connect to your workstation when you are in a meeting or being on the road to a customer	Relatively low specs. Core i3, a moderate amount of RAM.
Production	Database server, number cruching	For doing the heavy processing, like datawarehouse imports and processes, heavy reporting, heavy pre-calculation processes, to aleviate high-traffic production servers.	Many cores
Production	App server, number crunching	Same as above.	Many cores
Production	Database server, high traffic	For all the databases involved in high-traffic, storing data of user applications and services.	
Production	App server, high traffic	For all the production web sites and services.	Note that RAM is very relevant, to meet in-memory caching needs.
Test	Database server		

Test	App server		

DTAP

<TODO: Write something about this.>

Folders

If your servers and all your development workstations have a D: drive for data, then put the folders on the D: drive, otherwise put the the folders on the C: drive. Make a folder in the root of the drive with your company name:

D:**JJ**

In this folder, each environment gets a sub-folder written in all capitals.

D:\JJ**TEST**
D:\JJ**PROD**
D:\JJ**DEV**

Even machines with only one environment on it, should get a sub folder with the environment. That makes it better visible what environment you are working on, you can easily emulate the situation on a development workstation without additional configuration and it allows you to move environments from one server to the other.

Also add the following folders:

D:\JJ**Install**
D:\JJ**Backup**

< TODO: Add descriptions to each folder above.>

Those are not put in the environment folder.

The environment folder can contain the following sub folders:

D:\JJ\PROD**Images**
D:\JJ\PROD**IO Files**
D:\JJ\PROD**Log**
D:\JJ\PROD**Utilities**
D:\JJ\PROD**Web**

< TODO: Add descriptions to each folder above.>

The Images folder contains images e.g. uploaded from an application, not images that are content of the application, so not icons or basic content of the web site, but user-uploaded images or images uploaded by content management systems. The Images folder can contain sub-folders to keep images apart from eachother, that belong to a different application or set of applications.

D:\JJ\PROD\Images**QuestionAndAnswer**
D:\JJ\PROD\Images**Synthesizer**

An application image sub-folder can contain again sub-folders, for resized images with particular dimensions:

D:\JJ\PROD\Images\QuestionAndAnswer	Contains original images.
D:\JJ\PROD\Images\QuestionAndAnswer**100x100**	Images resized to 100x100 pixels
D:\JJ\PROD\Images\QuestionAndAnswer**320x280**	Images resized to 320x280 pixels

The resolutions above are simply examples. You can have different sizes per application.

<TODO: Describe the Utilities folder, that you use fully qualified application names and version sub folders. Same for the Web folder, and add to that that it contains both web services as well as web applications.>

Development Workstation

Development workstations should have the same kind of folder subdivision, also put on the same drives as on the servers. You might not put web sites in these folders, but Imags, IO Files and Logs should be present in the same folder structure as the servers.

Put the source code folders in the same spot as all your coworkers. Then things keep cooperating with eachother. Most of the times relative paths work, but sometimes they don't so it is a good plan to all have out copies of the source code in the same location. In case of TFS it should be D:\TFS

that is mapped to the outermost root of the source control system. Not to a branch, not to a Collection, but to the server name or IP address of the source control server.

Backups

<...>

Appendices

Appendix A: Layering Checklist

This checklist might be used if you want to bulk-program the architecture for an application by going through all the layers one by one:

- Data: Database structure
- Data: Data migration
- Data: Entity classes
- Data: Repository interfaces
- Data: Default repositories
- Data: NHibernate mappings
- Data: SQL queries
- Data: NHibernate repositories (optional)
- Data: Other repositories (optional)
- Data: Other mappings (optional)
- Business: LinkTo
- Business: Unlink
- Business: Enums
- Business: Resources
- Business: EnumExtensions
- Business: DeleteRelatedEntitiesExtensions
- Business: UnlinkRelatedEntitiesExtensions
- Business: EntityWrappers (optional)
- Business: Validators
    - o   Delete Validators too

- o Warning Validators (optional)
- Business: SideEffects
    - o SetDefaults SideEffects too
- Business: Managers
- Business: Extensions
- Business: RepositoryWrappers
- Business: Calculations
- Presentation: ViewModels
    - o Item ViewModels
    - o List item ViewModels (some may only need IDNameDto, no ListItem view model)
    - o List ViewModels
    - o Detail ViewModels
- Presentation: ToViewModel
    - o Singular forms
    - o WithRelatedEntities forms
    - o ToListItemViewModel
    - o ToScreenViewModel
    - o CreateEmptyViewModel (not every view model needs one)
- Presentation: ToEntity
    - o Singular forms
    - o WithRelatedEntities forms
    - o From screen view model
- Presentation: Presenters
    - o List Presenters
    - o Detail Presenters
    - o (Edit Presenters)
    - o Save methods in Detail (or Edit) Presenters.
- Presentation: Views (Mvc)
    - o List Views
    - o Detail Views

Appendix B: Knopteksten en berichtteksten in applicaties (resource strings) (Dutch)

Er is een bepaalde structuur waar binnen we werken voor knopteksten en meldingen in onze applicaties. De hele bedoeling is maximale herbruikbaarheid, minimaal vertaal werk en correcte teksten. Dat doen we door hele algemene teksten op plaats X te zetten, zo veel mogelijk domein termen op plaats Y, en alleen wat er dan over is, komt in specifieke projecten te staan. Dit kan het verschil betekenen tussen 100'en of 10000 teksten.

Resources worden op dit moment overal neergezet waar ze niet thuis horen, met verkeerd hoofdlettergebruik en verkeerde interpunctie. En op andere plekken worden resources gewoonweg niet gebruikt en staat alles hard op 1 taal.

Hier moet secuurder mee om worden gegaan. Van ontwikkelaars wordt verwacht zowel de Nederlandse taal als de Engelse taal in de resource files te zetten. Bij twijfel over Engels, vraag het een collega.

Varianten van resource files

1) 'Messages' bevatten **volzinnen** met correct hooflettergebruik en interpunctie.
2) 'PropertyDisplayNames' bevatten alleen vertalingen voor **property namen** en **class namen** (en andere members) die in de code voor komen. Meervoudsvormen worden hier ook in gezet.
3) 'Titles' bevatten overige kreten die als titel of bijv. als **knoptekst** dienen.

Hoofdletters, interpunctie, spelling

Hoofdlettergebruik etc. is conform de taalregels van de betreffende taal.

1) Volzinnen, moeten correct geschreven zijn: In het Nederlands begint dat met een hoofdletter en eindigt het met een punt, tenzij het een vraag is, dan met een vraagteken. Blijkbaar is het nodig om dit aan te geven, want het gebeurt vaak niet goed.
2) Voor Engels gebruiken we de Amerikaanse spelling.

Titels:

3) Titels en PropertyDisplayNames in het Nederlandse zijn als volgt: "Links in artikel", dus alleen beginnen met een hoofdletter. En dus geen punt erachter.
4) Hoofdlettergebruik in Engelse titels doen we als volgt: "Table of Contents", dus alle woorden beginnen met een hoofdletter, alleen onbelangrijke woorden zoals 'in', 'and', etc. in kleine letters.

Assemblies

Resources worden met de assemblies meegecompileerd*.

Termen worden zo veel mogelijk hergebruikt. Daarom zijn er plekken bedacht waar de termen thuis horen. Je moet in deze volgorde op zoek naar een resource die misschien al bestaat:

1) 'Save', 'Close', 'Edit', etc. staan in Framework.Resources.
2) Validatiemeldingen uit Framework.Validation.
3) CanonicalModel: een tussenmodel voor uitwisseling van gegevens tussen verschillende systemen.
4) Business layers bevatten alleen vertalingen voor de overige teksten die niet in het canonical model staan.
5) Pas als zelfs de business layer de term niet bevat, mag je het in je front-end project zetten.

Tips

1) Gebruik van placeholders zoals {0} is toegestaan, maar dan moet je wel een class erbij maken, die de placeholders vervangt. Zie Framework.Resources voor een voorbeeld. Het is dan verstandig om de resources zelf internal te maken en alleen de class die de placeholders vervangt public te maken. Kijk echter uit dat je het daarbij geschikt houdt voor meerdere talen, want een creatief met placeholder opgebouwde resource string werkt al gauw niet voor een andere taal.
2) Negeer dat de beschreven werkwijze kan resulteren in berichtteksten met hoofdlettergebruik zoals: 'Het **Ordernummer** is niet ingevuld bij de **Bestelling**.' Als we dit aanpakken, doen we dat met een algoritme, niet met nog meer resources.
3) Het is verstandig om teksten in de applicatie algemeen te houden. Dus bijv. 'Artikelen', i.p.v. 'Artikelen in dit boek'. Dit scheelt vertaalwerk. Ook dit is verstandig: 'Artikel 1: Naam is verplicht.' Daarbij zijn de teksten 'Artikel', 'Naam' en '{0} is verplicht.' waarschijnlijk allang vertaald. Door de 'dubbele punt' notatie aan te houden ('Artikel 1:'), voorkom je het verwerken van de term in een zin, wat voor iedere taal een compleet andere grammatica kan zijn, wat voorkomt dat er complete volzinnen vertaald moeten worden.

* Resources staan niet in een database, omdat het applicaties trager maakt en kun je de code niet draaien op omgevingen waarbij je geen toegang hebt tot de database. In sommige situaties kun je niet eens compileren zonder toegang te hebben tot een specifieke database. Bovendien geeft mee compileren van resources in specifieke projecten ons de mogelijkheid dubbelzinnige termen anders te vertalen per business domein.