JJ's Reference Architecture

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

Patterns

Contents

Contents 1
Introduction 3
Data Access Patterns 3
  Entity 3
    Alternatives 3
    Considerations 3
  Mapping 3
  DTO 3
  Repository 4
  Repository Interfaces 4
Business Logic Patterns 4
  Business layer 4
  Repository Wrappers 4
  Validators 5
  Side Effects 5
  LinkTo 5
  Cascading Extensions 6
  Manager / Facade 6
    Get by ID not in the Manager 7
  Visitor 7
  String Resources 8
Presentation Patterns 8
  ViewModel 8
    Considerations 9
  Lookup Lists 9
  ToViewModel 10
  ToEntity 11
  Presenter 11
  ToEntity-Business-ToViewModel Round-Trip 12
  NullCoalesce (ViewModels) 12
  Views 13
  First Full Load – Then Partial Load – Then Native Code 13
  Temporary ID's 14
  Stateless and Stateful 14
  Considerations 15
    ToEntity / ToViewModel 15
Presentation Patterns (MVC) 15
  Controller 15

- Post-Redirect-Get ... 15
  - Considerations ... 16
- ValidationMessages in ModelState ... 16
- Polymorphic RedirectToAction / View() ... 17
- Html.BeginCollection ... 17
- Return URL's ... 18
- Back Buttons ... 20

Data Transformation Patterns ... 20

- Converter ... 20
- TryGet-Insert-Update ... 20
- TryGet-Insert-Update-Delete / Full-CRUD Conversion / Collection Conversion ... 20
  - Considerations ... 20
  - Alternative: Flagging ... 22
- DocumentModel ... 22
- Selector-Model-Generator-Result ... 22
  - Generating a Document ... 23
  - Data Source Independence ... 23
  - Multiple Import Formats ... 23
  - Limiting Complexity ... 23
  - MVC ... 23

Other Patterns ... 23

- Accessor ... 23
- Adapter ... 23
- Anti-encapsulation ... 23
- Initialization and Finalization ... 24
- Constructor Inheritance ... 24
- DebuggerDisplays ... 24
- Executor ... 24
- Inheritance-Helper ... 25
- Factory ... 25
- Factory-Base-Interface ... 26
- TryGet ... 26
- Get-TryGet-GetMany ... 26
- Helper ... 27
- Info ... 27
- IsSupported ... 27
- Mock ... 27
- Name Constants ... 27
- NullCoalesce ... 28
- Plug-In Model ... 28
- Progress and Cancel Callbacks ... 28
- Singular, Plural, Non-Recursive, Recursive and WithRelatedEntities ... 28
- Wrapper ... 30

Alternatives ... 30

- Rich Models ... 30

Introduction

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, error prone, and it can actually harm performance of queries.

Alternatives

Generally avoided, but not prohibited:

- Use inheritance anyway with the aforementioned downsides.
- Use interfaces for polymorphism instead.
- Instead of inheritance, consider a composition solution, rather than an inheritance solution.

Considerations

<TODO: Write some more about the difficulties of inheritance in entity models.>

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 save 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.

Repository Wrappers

Passing around lots of repositories creates long parameter lists, that are prone to change. To combat that problem, combine sets of repositories into repository wrappers and pass those around instead. This keeps the parameter lists short and less prone to change.

You can make a single RepositoryWrapper with all your domain model's repositories.

But that is not always enough. Some logic will use repositories out of multiple domains, so sometimes you are well off making a custom repository wrapper in that case. You could also choose to simply pass around multiple repository wrappers: one per domain model.

Also, you may want to create different, more limited repository wrappers. For instance ones for partial domain models. This keeps the width of dependency narrow, so logic that has nothing to do with certain repositories, do not become dependent on all of them.

An alternative to repository wrappers is dependency injection. See 'dependency injection'. There you will find some criticism about the techique, but those might be due to not using a very good dependency injection API. Repository wrappers and dependency injection could well be used in combination with eachother.

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.

Next to validators saying that user input is wrong, validators can be used to generate warnings, that are not blocking, but help the user do their work.

Validators can also be used for (complex) delete constraints, for instance when an entity is still in use, you might not be able to delete it.

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.

Side effects should evaluate the conditions internally as much as possible. So the called of the side effect class does not know what conditions are tied to it doing anything at all. This makes the side effect fully responsible for what happens. The side effect's doing anything can also be dependent on entity status. See 'Entity Status Management'.

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.

```
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);
        }
    }
}
```

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.

Cascading Extensions

<TODO: Describe how to organize your DeleteRelatedEntitiesExtensions and UnlinkRelatedEntitiesExtensions. >

Manager / Facade

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. You could also call it a combinator class.

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.

<TODO: Make a good text out of this, covering handling polymorphism in visitors. Merge this with the main text:

- Document that a Visitor that handles polymorphism, should have a Polymorphic visitation that delegates to a concrete visitation, that delegates to a base visitation, and you need all those methods delegating in the right order, for the visitation to happen in the correct order.

- Visitor pattern: mention that you always need to call polymorphic, otherwise you will not get all the objects when you override the polymorphic. >

<TODO: Code example.>

<TODO: Describe this: Patterns, Visitor: Figure out a good way to prevent calling those Polymorphic visit methods if not required.>

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:

```
Resources.resx
Resources.nl-NL.resx
Resources.de-DE.resx
```

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

<TODO: Mention the resource formatter pattern, e.g. MessageFormatter.>

Resources seem part of the presentation, but they are extensively used in the business layer, so are put in the business assemblies. Especially the 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.

Extra information in Dutch about how to structure your resource files can be read in Appendix B.

Presentation 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.

<TODO: Describe the ViewModel pattern more strictly: entity view models, partial view models and screen view models and the words Details, Edit, List, NotFound, Delete, Deleted and Overview. And that those words are there to indicate that it is a screen view model, not an entity or partial view model. LoginViewModel may be an exception. >

Considerations

The reason there should be no inheritance is because that would create an unwanted n^2 dependency between views and the base view model: *n* views could be dependent on 1 view model and *m* view models could be dependent on 1 base view model, making *n* * *m* views dependent on the same base view model. This means that if the base view model changes *n* * *m* views could break, insteaf of just *n*. *m* is even likely to become greater than *n*. If multiple layers of inheritance are used, it gets even worse. That can get out of hand quickly and create a badly maintainable application. By using no inheritance, a view model could only break *n* views (the number of views that use that 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
ToViewModelHelper.cs
ToViewModelHelper.EmptyViewModels.cs
ToViewModelHelper.Values.cs
ToViewModelHelper.Items.cs
ToViewModelHelper.ListItems.cs
ToViewModelHelper.Lookups.cs
ToViewModelHelper.Partials.cs
ToViewModelHelper.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.:

```
// Orders

public static OrderListViewModel ToListViewModel(this IList<Order> orders) ...

public static OrderEditPopupViewModel ToEditViewModel(this Order order) ...

public static OrderDeletePopupViewModel ToDeleteViewModel(this IList<Order> orders)
...
```

Some view models do not take one primary entity as input. So it does not make sense to turn it into an extension method, because you cannot decide which entity is the this argument. In that case we put it in a ViewModelHelper class with static classes without this arguments. ViewModelHelper is also part of the ToViewModel layer.

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.

<TODO: Describe the organization of the ToEntity extensions.>

Presenter

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 action method 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.

All view model creation should be delegated to the ToViewModel layer (rather than inlining it in the Presenter layer), because then when the ViewModel creation aspect should be adapted, there is but one place in the code to look. It does not make the presenter a needless hatch ('doorgeefluik'), because the presenter is responsible for more than just view model creation, it is also resposible for retrieving data, calling business logic and converting view models to entities.

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.

<TODO: Consider this: Patterns, Presentation: There is something wrong with the pattern 'ToEntity-Business-ToViewModel-NonPersisted' sometimes it is way more efficient to execute the essence of the user action onto the user input view model. Sometimes it is even the only way to execute the essense of the user action onto the user input view model. Examples are removing a row an uncommitted row or collapsing a node in a tree view.>

NullCoalesce (ViewModels)

When you user input back as a ViewModel from your presentation framework of choice, for instance MVC, you might encounter null-lists in it, for lists that do not have any items. To prevent other code from doing null-coalescing or instead tripping over the nulls, you can centralize the null-coalescing of pieces of view model and call it in the presenter.

<TODO: Better description. Also incorporate:
- Also add a code example.
- Consider making a separate pattern description for NullCoalesce methods in general and move it to the Other Patterns section to which you then refer from this section NullCoalesce (ViewModels).
- Null-coalesce. Applied to viewmodels that are passed to presenters. The choice is made here to only null-coalesce things that a view / client technology might leave out. Theoretically it might be better to null-coalesce everything in the view model, but this does take full traversal of the tree, which comes with a (small) performance penalty. Also: the null-coalesce procedures take some typing time for the programmer, and requires maintenance when the structure changes. That is why the choice is made to only null-coalesce a select set of things, that is adapted to our specific needs, rather than something that will always work. >

Views

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.

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.

Temporary ID's

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.

The TemporaryID's can be really temporary and can be regenerated every time you create a new view model.

The TemporaryID concept breaks down, as soon as you need to use it to refer to something from multiple places in the view model.

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.

Another alternative is a different ID generation scheme. You may use an SQL Sequence, or use GUID's, which you assign from your code. Switching from int ID's to GUID's is a high impact change though, and does come with performance and storage penalties.

Stateless and Stateful

The presentation patterns may differ slightly if used in a stateful environment, but most of it stays in tact. For instance that Presenters have action methods that take a ViewModel and output a new ViewModel is still useful in that setting. In a stateless environment such as web, it is needed, because the input view model only contains the user input, not the data that is only displayed and also not the lookup lists for drop down list boxes, etc. So in a stateless environment a new ViewModel has to be created. You cannot just return the user input ViewModel. You would think that in a stateful environment, such as a Windows application, this would not be necessary anymore, because the read-only view data does not get lost between user actions. However, creating a new view model is still useful, because it creates a kind of transaction, so that when something fails in the action, the original view model remains untouched.

You will be making assumptions in your Presenter code when you program a stateful or stateful application. Some things in a stateful environment environment will not work in a stateless environment and you might make some objects long-lived in a stateful environment, such as Context, Repositories and Presenters. But even if you build code around those assumptions, then when switching to a stateless environment – if that will ever happen – the code is still so close to what's needed for stateless, that it will not come with any insurmountable problems. I would not beforehand worry about 'will this work in stateless', because then you would write a lot of logic and waste a lot of energy programming something that will probably never be used. And programming something for no reason at all, handling edge cases that would never occur, is a really counter-intuitive, unproductive way of working.

Considerations

ToEntity / ToViewModel

<TODO: Explain the argument that ViewModel, ToEntity and ToViewModel does require programming a lot of conversion code, but gives you complete freedom over your program navigation, but the alternative, a framework prevents writing this conversion code for each application, but has the downside that you are stuck with what the framework offers and loose the complete freedom over your how your program navigation works.>

Presentation Patterns (MVC)

<TODO: Mention ModelState.ClearErrors.>
<TODO: Mention: Using Request.UrlReferrer in Http Get actions crashes. Use Request.RawUrl.
>

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.

Entity names put in controller should be plural. So Customer**s**Controller not CustomerController.

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.

```
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);
}
```

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.

<TODO:
- Mention that return View in case of validation messages is the way to go, because otherwise MVC will not remember un-mappable wrong input values, like Guids and dates entered as strings. (In one case this lead to the browser asking for resending postdata upon clicking the back button, so check whether this is actually a good idea.)
- Not using return View() in a post action makes old values not be remembered.>

Considerations

If you do not conform to the Post-Redirect-Get pattern in MVC, you may get to see ugly URL's. When you hit the back button, you might go to an unexpected page, or get an error. You may see original values that you changed re-appear in the user interface. You may also see that MVC keeps complaining about validation errors, that you already resolved. So conform to the Post-Redirect-Get pattern to stay out of trouble.

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)
{
    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):

```
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:

```
@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)
{
    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.

Return URL's

- 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)
```

- 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 });
```

- A return URL should always be optional, otherwise you could never serparately debug a view.
- That way you have an easily codeable, well maintainable solution.
- Do not use RefferrerUrl, because that only works for HttpPost, not HttpGet. Use Request.RawUrl instead.

- There is a built-in error proneness in return URLs'. If you pass the same return URL along multiple HTTP requests, only one action has to forget to pass along the return URL and a back or close button is broken and you will find out very late that it is, because it is not an obvious thing to test. The same error-proneness is there for return actions with return actions with return actions, or with bread-crumb like structures with multiple return actions built in.

<TODO: Incorporate this: Ret parameters can be done with new { ret = Request.RawUrl } for full load, and for AJAX this works: { ret = Url.Action(ActionNames.Index) } if you always make sure you have an Index action in your controller, which is advisable.>

Back Buttons

There is a pitfall in builing back buttons. If you mix back buttons being handled at the server side, compared to window.history.back() at the client-side, you run the risk that the back button at one point keeps flipping back and foreward between pages.

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 / Full-CRUD Conversion / 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.

Considerations

Converting one collection to another may involve more than creating a destination object for each source object. What complicates things, is that there may already be a destination collection. That means that insert, update and delete operations are required. There are different ways to handle this depending on the situation. But a general pattern that avoids a lot of complexity, is to do the inserts and updates in one loop, and do the deletes in a second loop. The inserts and updates are done first by looping through the source collection and applying the TryGet-Insert-Update pattern on each item, while the delete operations are done separately after that by comparing collections of entities to figure out which items are obsolete.

In a little more detail:

- 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.

Here follows some pseudo code for how to do it:

```
void ConvertCollection(IList sourceCollection, IList destCollection)
{
    foreach (var sourceItem in sourceCollection)
    {
        var destItem = TryGet(...);
        if (destItem == null)
        {
            destItem = Insert();
        }
        destItem.Name = sourceItem.Name; // Update
    }

    var itemsToDelete = destCollection.Except(sourceCollection);
    foreach (var itemToDelete in itemsToDelete)
    {
          Delete(itemToDelete);
    }
}
```

The specific way to implement it, is different in every situation. Reasons that there are many ways to do it are:

- You cannot always count on instance integrity.
- You cannot always count on identity integrity.
- The key to a destination item might be complex, instead of just an ID.
- You do not always have a repository.
- It does not always need to be full-CRUD.
- You might need to report exactly what operation is executed on each entity.
- You might need a separate normalized *singular* form of the conversion, that may conflict with the way of working in the plural form.
- An alternative isNew detection might be needed.

- Some persistence technologies will behave unexpectedly when first retrieving and then writing and then retrieving again. Intermediate redundant retrievals should be avoided. Or not, depending on the situation.

Each variation has either overhead or elegance depending on the situation. If you always pick the same way of doing it, you may end up with unneccesary and unsensical overhead, or with an overly complicated expression of what you are trying to do.

The general forms above is a good starting point. Then it needs to work correctly. The next quality demand is a tie between readability and performance.

Alternative: Flagging

An alternative to TryGet-Insert-Update-Delete pattern, which kind of does a full diff of a source and destination structure, is maintaining a kind of flagging in the source structure: Added, Modified, Deleted and Unmodified.
A downside is that when two people try to save a piece of data at the same time, you may end up with a corrupted structure. It depends on the situation whether this would happen at all, since not all data is edited by every user.

Another downside to flagging is that the source structure must be adapted to it, which is not always an option / a good option.

The TryGet-Insert-Update-Delete pattern, though, creates a last-user-wins situation, because not flagging determines whether it is an update or insert, but actual existence of dest object determines it.

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.

Her follow some 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

Accessor

An accessor class allows access to non-public members of a class. This can be used for testing or for special access to a class from special places. JJ.Framework.Reflection has an implementation of a reusable Accessor class.

Adapter

<TODO: Describe what an adapter is in general and what kind of variations you can think of.>

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.

Initialization and Finalization

Cleanup code should be symmetric to the set-up code. Build something up in the constuctor then dispose things in the finalizer, start a service at startup then stop a service at shutdown, etc. If in the constructor you bind an event, then in the destructor you unbind it.

You can also choose to implement IDisposable. This is useful if you want to be able to explicitly trigger finalization. Finalizers/destructors only go off when the garbage collector feels like it, and you might want to imperatively tell an object to clean up its stuff.

- If you implement IDisposable, call Dispose from the finalizer/destructor.:
  ```
  ~MyClass
  {
     Dispose();
  }
  ```
- Make sure the dispose can successfully run regardless of state, so check any variable you might use for null first and be tollerant towards null.

  ```
  public void Dispose()
  {
     _myConnection?.Close();
  }
  ```

- Also call GC.SuppressFinalize() in the Dispose() method, because then the garbage collector will skip a few unneeded steps in getting rid of the object.

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.

DebuggerDisplays

<TODO: Describe how I handle DebuggerDisplays. Snippet of text: DebuggerDisplays with private property DebuggerDisplay.>

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:

```
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));
        }
    }
}
```

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:

```
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)
{
    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.

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.

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:

```
OrderStatusEnum IOrderManager.GetOrderStatus();
bool IOrderManager.GetOrderStatusIsSupported { 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.

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.

Name Constants

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:

```
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.

<TODO: Consider not assinging the string value at all, but using nameof(ViewNames.Edit). Consider using nameof() over an existing member to begin with.>

NullCoalesce

<TODO: See NullCoalesce (ViewModels) and write some good text here.>

Plug-In Model

<TODO: Describe my implementation of a nice plug-in model including the ReflectionHelper.GetImplementation methods.>

Progress and Cancel Callbacks

To make a process cancellable and report process without being dependent on the presentation framework, you can simply pass a few callback delegates to a method or class.

```
public Excute(Action<string> progressCallback, Func<bool> isCanceledCallback
{
    progressCallback("Starting.");

     if (isCanceledCallback())
     {
         progressCallback("Cancelled.");
         return;
     }


    // ...

    progressCallback("Finished.");
}
```

It depends on your problem whether those callbacks are nullable and you should do the appropriate null-checks depending on the situation.

<TODO: Add explanations and code examples about the client code.>

Sometimes it is useful to separate Cancel into two: Canceling and Canceled. This is because a process might not cancel immediately. A UI should not immediately enable a Start button again after the user pressed Cancel. A isCancelingCallback then allows the client to signal to the process that cancellation is requested. And an isCanceledCallback will let the process signal the client that cancelation has complete, so it can enable the start button again.

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 and RelatedEntities methods separate from the not-with-RelatedEntities methods. 'Related entities' means entities intrinsically part of the entity, not links to reused entities. Also, not the parent.

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.

Finer details about the Singular form:

- They do not process child entities, they can however link to reusable entities, such as enum-like types, or categories.
- They usualy do not assign a parent. The caller of the Singular form does that. That way methods are more independent of context and better reusable and code better rewritable. There are exceptions to that rule.

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.

```
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);
            }
        }

        private void ProcessRecipe(Recipe recipe)
        {
            _sb.AppendLine(String.Format("Recipe: {0} ({1})", recipe.Name, recipe.ID));
        }
    }
```

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.

Alternatives

Rich Models

<TODO: Write story with pros and cons. Include:
>> Arch: another downside of rich models is the magic of it. You are not sure what happens and all sorts of non-obvious side-effects may go off.

- Arch: anemic models and separation of concerns and no rich models can have the consequence that you loose identity and instance integrity, because derived structures are more common. For controls for instance, with gesture events, the original object needs to be found and raised an event on, and you cannot get away with doing it on a derived object unless you clone everything. Not sure how to descrive this clearly. It is about rich models vs. anemic models and when and how to apply which and what are the pros and cons and I do not have a clear image of that yet.
- Explain the problems with rich models in the business layer comprised of derived classes out of the persistence layer, that extend the model with specific relations, constraints and rules, that you do not enforce in the persistence layer.
  Problems with putting it in the business layer include that the persistence layer does not know how to instantiate the derived class, so it must instantiate the base class. Also: if rules are strictly enforced in an extended entity model, it is hard to separate creating entities from validating it, so instead of user-generated validation messages, you might get exception messages instead.
  Problems with putting the specialized classes in the persistence layer include that the entity model must stay as clean as possible: anything you put in the data layer is hard to get rid of.
  When you do need an business logic interfacing that is comprised of an 'extended' entity model, then you cannot really use inheritance. You might be able to create a manager class that creates a wrapper class around the 'base' class out of the entity model.
  Currently the choice is to not make an extended entity model in the business layer.

- Arch: against rich models: You want your entity model to be a direct depiction of what is actually stored, so that you have control over that. If it is obscured, this means less control over what is going on. >

<TODO: Compare rich models with the 2D separation of concerns.>