JJ's Reference Architecture

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

Code Style

Contents

Introduction

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

Use Resharper. Seriously. Finetune it to automatically check your coding style. Use it to keep code clean as write code or change existing code.

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'.	`Sine_OperatorCalculator_VarFrequency_WithPhaseTracking`
Type arguments start with the letter T or are just the letter T	`T TEntity TViewModel`
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	Not Recommended	Recommended
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:`
	`  break;`
	`}`
		`switch (x)`
		`{`
		`  case 1:`
		`  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).	void Bla()
	{
	var x = new X();
	x.A = 10;
	var y = new Y();
	y.B = 20;
	y.X = x;
	Bla2(x, y);
	}
		void Bla()
		{
		var x = new X();
		x.A = 10;
		
		var y = new Y();
		y.B = 20;
		y.X = x;
		
		Bla2(x, y);
		}

Each variable declaration on its own line.	int i, j;	int i; int j;
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.	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; }	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; }
Align the elements of linq queries as follows:	var arr = coll.Where(x => x...). OrderBy(x => x...).ToArray()	var arr = coll.Where(x => x...) .OrderBy(x => x...) .ToArray()
Use proper indentation	<TODO: Example.>	<TODO: Example.>

Generic constraints on next line.
(So they stand out)
	class MyGenericClass<T> where T: MyInterface
	{
	}
		class MyGenericClass<T>
		where T: MyInterface
		{
		}

For one-liners, but generic constraints on same line instead.	interface IMyInterface
	{
	void MyMethod(T param)
	where T : ISomething
	}
		interface IMyInterface
		{
		void MyMethod(T param) where T : ISomething
		}

Trivial Rules

Rule	Wrong	Right

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

Keep members private as much as possible.		`private void Bla()`
		`{`
		`}`

Keep types internal as much as possible.		`internal class MyClass`
		`{`
		`}`

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

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.`	**`/// <summary>`**

	`int X { get; set; }`	`/// 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
 	
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.
	`#if FEATURE_X_ENABLED`
	`// ...`
	`#endif`
		`if (config.FeatureXEnabled)`
		`{`
		`  // ...`
		`}`

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);`
		`}`

Do not leave unused (outcommented) around. If needed, move it to an Archive folder, or Outtakes.txt, but do not bug your coworkers with out-of-use junk lying around.		

it is appreciated when a file stream is opened specifying all three aspects FileMode, FileAccess and FileShare explicitly with the most logical and most limiting values appropriate for the particular situation.		

Miscellaneous Rules

Description	Not Recommended	Recommended

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

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) %>`
Handle null and empty string the same way everywhere.		
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(typeof(T))`	`T = new T()`

Activator.CreateInstance should be rare and the last choice for instantiating an object.		

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);
```

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.

Naming

See also: Casing, Punctuation and Spacing.

Boolean Names

Use common boolean variable name prefixes and suffixes:

Prefix / Suffix	Example	Comment
Is…	IsDeleted	This is the most common prefix.

Must…	MustDelete	
Can…	CanDelete	Usually indicates what *user* can do.
Has…	HasRecords	
Are…	AreEqual	For plural things.
Not…	NotNull	A valid prefix, but be careful with negative names for readability's sake. See 'Double Negatives'.
Include…	IncludeHidden	Even though it is verb, it makes sense for booleans.
Exclude…		Even though it is verb, it makes sense for booleans.
… Exists	FileExists	

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

Some boolean names are so common that they do not get any prefixes:

Visible
Enabled

Class Names

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

Converter
Validator
Calculator

And they start with a term out of the domain:

OrderConverter
ProductValidator
PriceCalculator

A more specialized class can get prefixes or suffixes as follows:

OptimizedPriceCalculator
OrderWithPriorityShippingValidator

Or alternatively:

OrderValidatorWithPriorityShipping

Abstract classes get the preferred suffix 'Base':

ProductValidatorBase

This is because it is very important to see in code whether something is a base class.
Exceptions to the suffix rule can be made if it would otherwise result in less readable code. For instance, base classes in entity models might not look good with the 'Base' suffix.

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.
Asserter	<TODO: Describe>
	Any method verb could become a class name, by turning it into a verby noun, e.g. Convert → Converter.

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.

DateTime Names

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

```
StartDateLocal
OrderDateTimeUtc
```

An alternative possible suffix for DateTimes would be 'When':

```
ModifiedWhen
OrderedWhen
```

But that looks less nice when you add the Local and Utc suffices again:

```
ModifiedWhenUtc
OrderedWhenLocal
```

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'.

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
```

Or the suffix `Requested`, if your event looks like a method name.

```
RemoveRequested
```

Pardon the ambiguity, but the naming above can be used for the names of events, but some of them also serve well as names for methods that fire/emulate or otherwise handle the event. The prefix '`On`' for instance and the prefix '`Handle`' may very well be used for the methods that actually raise the event. '`Fire`' and '`Do`' are also alternatives.

Avoid event names that indicate that it is an event in two different ways. For instance '`OnDragging`' can be shortened to just '`Dragging`', because the suffix -ing is already an indication that it is an event. '`OnMouseUp`' can be shortened to just '`MouseUp`', because that is an established event name.

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.
Assert	A method that throws **exceptions** if input is invalid.
Calculate	
Clear	
Convert	
ConvertTo	
Create	When a method returns a new object.
Delete	
Ensure	Sets up a state if it is not set up yet. If Ensure means throw an exception if a state is not there, then consider using the verb 'Assert' instead.
Execute	
Generate	
Get	
Invoke	
Parse	
Process	
Remove	
Save	
Set	
Try	
TryGet	
Validate	A method that generates **validation messages** for user-input errors

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}***.*

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
...Polymorphic	Handles a multitude of differrent derived types, possibly each in a different way.
…IfNeeded	If something is executed conditionally. This is a nice alternative for the less pretty suffixes 'Conditionnally' or a prefix 'Conditional', which obscures the name that comes after.
…Unsafe	When it lacks e.g. thread-safety or executes unmanaged code, or lacks a lot of checks.
…Recursive	(Some people tend to use 'Recursively' instead, probably insisting it is better grammer, but Recursive is shorter and not grammatically incorrect either. It is a characteristic, as in 'Is it *recursive*?'.)
To…	For conversion from one to another thing. Usually 'this' is source of the conversion, for example: `array.ToHashSet()`

Less commonly the 'To' prefix is used when the 'this' is not the source, for instance:

```
MyConverter.ToHashSet(object[] array)
```

The Convert or ConvertTo verbs might be more appropriate there:

```
MyConverter.ConvertToHashSet(object[] array)
```

From… For conversion from one to another thing. A lot like 'To…' executed on the dest object instead:

```
dest.FromSource(source)
```

The 'To…' prefix is more common, and usually more readable.

Miscellaneous Names

- For number sequences you can use names like: `ListIndex`, `IndexNumber`, `SortOrder`. (Avoid `Index` because it is an SQL keyword.)