

Naked Objects 7.1-beta – Application Developer Manual

Contents

Recent changes	1
Version 7.1	1
New features.....	1
Version 7	1
Coding changes required to domain model(s)	2
Coding changes required to Run (Mvc) projects	3
Coding changes required to XAT projects	4
New features.....	5
Getting started.....	6
Writing your first Naked Objects application.....	6
Creating a domain model project.....	8
Define your DbContext(s).....	9
Overriding the default database schema generation	10
Using data fixtures with Code First	11
Running your domain model(s) as a Naked Objects MVC application	12
Programming Reference	13
Domain model - programming concepts	13
Domain object.....	13
Property	13
Action	14
Menus	15
Recognised method	19

Recognised attribute	19
View Model	20
Service	23
Factories and Repositories	23
External or System service	25
Contributed action	26
Injection of domain services into domain objects	28
The Domain Object Container.....	29
Application configuration	31
Specifying the Namespace(s) that cover your domain model	31
Registering domain services.....	32
Specifying any types that will not ordinarily be discovered by the reflector	32
Configuring the EntityObjectStore	33
Configuring Authorization	34
Configuring Auditing	34
Configuring Profiling	35
Configuring the RestRoot	36
System configuration using the Unity framework.....	36
The Naked Objects programming model	36
Recognised Value Types	37
Recognised Collection types	37
Recognised .NET attributes	38
Recognised Methods.....	41
NakedObjects.Attributes	45
NakedObjects.Types.....	56
NakedObjects.Helpers.....	57

<i>Adding behaviour to your domain objects - a how-to guide</i>	60
Using the Naked Objects IDE	60
Item Templates	60
Code Snippets	61
The object life-cycle	63
How to create an object	63
How to persist an object	64
How to update an object	64
How to delete an object	65
How to retrieve existing instances	65
How to insert behaviour into the object life cycle	65
How to specify that an object should never be persisted	66
How to specify that an object should not be modified by the user	66
How to specify that a class of objects has a limited number of instances	66
How to implement concurrency checking	66
Object presentation	66
How to specify a title for an object	66
How to specify the icon for an object	67
How to specify a name and/or description for an object	68
How to specify that an object should be always hidden from the user	68
Properties	69
How to add a property to a domain object	69
How to prevent the user from modifying a property	69
How to make a property optional (when saving an object)	70
How to specify the size of String properties	70
How to validate user input to a property	70

How to validate user input to more than one property	71
How to specify a default value for a property	72
How to specify a set of choices for a property	73
How to specify auto-complete for a property	74
How to set up the initial value of a property programmatically	75
How to trigger other behaviour when a property is changed	75
How to control the order in which properties are displayed	76
How to specify a name and/or description for a property	76
How to hide a property from the user	76
How to make a property non-persisted	77
How to handle File Attachments	77
How to display an image	79
How to handle enum properties	80
Collection properties	81
How to add a collection property to a domain object	81
Adding-to or removing objects from a collection	82
How to create a derived collection	82
How to control the order in which table rows are displayed	83
How to specify which columns are displayed in a table view	84
Actions	84
How to add an action to an object	84
How to specify the layout of the menu of actions on an object	84
How to define a contributed action	84
How to prevent a service action from being contributed to objects	84
How to specify parameter names and/or descriptions	84
How to make a parameter optional	85

How to specify a default value for a parameter	85
How to specify a set of choices for a parameter	86
How to allow selection of multiple choices	87
How to specify auto-complete for a parameter	87
How to specify the length or format for text-input parameters.....	88
How to obscure input text (e.g. for a Password)	88
How to validate parameter values	88
How to specify conditions for invoking an action	90
How to control the order in which actions appear on the menu.....	91
How to hide actions	91
How to pass a message back to the user	92
How to work with transactions	92
Advanced Entity Framework techniques	92
How to handle concurrency checking	93
How to specify 'eager loading' of an object's reference properties	93
How to implement complex types	94
How to work with multiple databases	94
How to work with multiple database contexts	94
How to write safe LINQ queries	96
How to handle associations that are defined by an interface rather than a class	98
Customising the MVC User Interface	104
Customising the CSS	104
Custom Views.....	106
Custom controllers	111
MVC - Additional How-Tos	118
How to inject Services into Controllers and Views	118

How to encrypt hidden fields in a transient object	119
How to disable server-side validation (Ajax)	119
Other.....	119
Getting hold of the current user programmatically	119
Creating an XML Snapshot of an object	120
<i>Creating and using a Restful Objects API.....</i>	122
Adding a Restful Objects API to a Naked Objects MVC application.....	122
Creating a standalone Restful Objects API.....	122
Conformance to the Restful Objects specification	123
Custom extensions	124
Restful Objects Server - additional how-to's	124
How to determine whether an action will require a GET, PUT or POST method	124
How to work with proto-persistent objects	124
How to specify the scheme for domain model type information	125
How to enforce concurrency-checking	125
How to make an application Read Only	126
How to implement automatic redirection for specific objects	126
How to change cache settings.....	127
How to handle Complex Types.....	127
How to change the format of the Object Identifier (Oid) in resource URLs	128
How to limit the scope of the domain model that is visible through the Restful API	129
How to switch off strict Accept header enforcement	130
How to limit the size of returned collections	131
How to enable cross-origin resource sharing (CORS).....	131
How to work with addressable View Models.....	132
Authorization in Restful Objects	132

<i>Running without a user interface</i>	<i>134</i>
Running Naked Objects as a .exe	134
Running multiple threads asynchronously	135
<i>Authorization</i>	<i>136</i>
Attribute-based Authorization.....	136
Custom Authorization	138
Controlling visibility of columns in a table view.....	139
<i>Auditing</i>	<i>140</i>
<i>Profiling</i>	<i>141</i>
<i>Testing</i>	<i>142</i>
Executable Application Tests (XATs)	142
Creating an XAT test class	142
Writing tests.....	143
Simulating users and roles	145
Running tests against a database.....	146
Using object fixtures within XATs.....	146
End to End Testing with Selenium.....	148
Adding a Selenium test project	148
Writing Selenium tests.....	148
<i>Internationalisation / Localisation</i>	<i>151</i>
Adding the Resources projects to your solution	151
Specifying that you wish to localise the application.....	152
Populating the Model resource file.....	152
Translating the resource files	153
Testing your localised application.....	154

Clusters	155
The Cluster Pattern	155
Hard Rules	155
Optional Rules	158
The Clusters project on GitHub	158
How to build the framework from source	159
Troubleshooting	160
Logging	160
Problems running the AdventureWorks example	160
Error locating server	160
The application runs very slowly	161
Problems working CodeFirst	161
Database is generated, but certain (or all) tables are not being generated	161
Errors thrown when starting an application	161
No known services	161
Unable to infer a key	162
Class not public	162
Errors thrown when running an application	162
A property is not virtual/overrideable	162
Invalid column name	163
Invalid object name	163
Collection not initialised	163
Could not find Entity Framework context for type	163
Unexpected behaviour in the user interface	163
A public method is not appearing as an object action	163
Default, Choices, Validate or other complementary methods are showing up as menu actions	164

Debugging	164
Life Cycle methods are not being called	164
The injected Container is null.....	165
An injected service is null.....	165
The application runs OK in execute mode, but throws an exception in debug mode	165

Recent changes

Version 7.1

The principal difference between versions 7.1 and 7.0 is an internal refactoring: NakedObjects.Mvc now depends only on NakedObjects.Facade, which is the new name for NakedObjects.Surface and represents a simplified interface into the Naked Objects server. The benefits of this re-structuring will become more obvious in future releases. This change does have some impact on existing views and controllers. The best way to identify the changes needed is to install a new MVC run project alongside your existing one and where your existing project has compile errors look at the new project to identify the differences.

New features

- [NakedObjectsType](#) and [NakedObjectsInclude](#) attributes.
- [Additional methods on the XAT types](#) to (optionally) make use of the nameof operator in C# 6.
- Editable reference properties or action parameters that do not already provide explicit choices, an auto-complete option, and are not marked up with the [\[FindMenu\]](#) attribute, will *not* now have a Find Menu added to them automatically. Rather they are provided with an ‘auto, auto-complete’ mechanism: the field takes an input string and if the user hits the [Space](#) bar, a drop-down list appears showing all recently-viewed objects of a type that matches the field. If the user continues to type then the list will be filtered to only those recently-viewed objects that match the type and whose title contains the specified sub-string (case-insensitive).

Version 7

Version 7.0 of the Naked Objects Framework (NOF) is the result of the largest refactoring of the code base in the seven year history of the Naked Objects framework (for .NET).

Most of the big changes are to the internal design and construction of the framework and have no impact on the application programmer. The motivations for these refactorings are:

- Improve performance, especially on large scale enterprise applications
- Make it easier for third parties to extend or customise the whole framework, by adding or replacing components
- Making the code base easier to read and maintain
- Improve flexibility for action menu design

Nonetheless, upgrading an existing application from NOF 6 will require some changes to your code. We will look first at coding changes required to domain model projects, and then to the projects that use the models.

Coding changes required to domain model(s)

Changes to your domain model project(s) in moving from NOF 6 to NOF 7 are quite small, as follows:

Action Menus

NOF 7 gives far greater control in the layout of action menus, including sub-menus, by means of the newly recognised [Menu method](#). You do *not* need to use this capability: if there is no [Menu](#) method an object's action menu will show all the actions for that object as before – honouring any ordering specified using [MemberOrder](#). You will only *have* to specify a [Menu](#) method if you were previously using the [Sequence](#) property of the [MemberOrder](#) attribute in order to create a sub-menu, as that [Sequence](#) property has been obsoleted.

TitleBuilder

The [TitleBuilder](#) class (in [NakedObjects.Helpers](#)) has been obsoleted – because its implementation could lead to performance issues in certain circumstances. The domain object container now has a method [NewTitleBuilder](#), which returns an implementation of [ITitleBuilder](#), which has the same method signature as the obsoleted [TitleBuilder](#). So in your code, change:

```
var t = new TitleBuilder();  
// to:  
var t = Container.NewTitleBuilder();
```

adding an [injected container](#) if your object does not already have one.

Contributed Actions

In NOF 6, any action defined on any registered service was potentially recognised as a contributed action – and contributed to domain object types that were parameters of that action. (In NOF 6 this could be suppressed by means of the [NotContributedAction](#) attribute).

In NOF 7, service actions are recognised as potential contributed actions only if the appropriate parameter(s) are marked up with the new [ContributedAction](#) attribute. ([NotContributedAction](#) has therefore been obsoleted). See [Contributed Actions](#) for full details. On existing applications you will therefore need to find all service actions that you require to be contributed to domain objects and add the attribute. This includes [query-result-contributed actions](#). Note that for object-contributed actions you can now specify the sub-menu name (or none) for the contributed actions.

The Find Menu

For a long time, our strategy has been to reduce the need for the ‘Find Menu’ on editable reference properties and action parameters - for example through the introduction of [conditional choices](#), and [auto-complete](#). With NOF 7, the Find Menu will be rendered automatically only where there is neither an auto-complete nor a choices option. If you wish to force the rendering of a Find Menu even where there is an auto-complete, you may add the [FindMenu](#) attribute on that action parameter or object property.

Where a Find Menu *is* rendered, it will now show only the actions (from registered services) that are explicitly marked up with the new [FinderAction](#) attribute. (The [ExcludeFromFindMenu](#) attribute has therefore been made obsolete). The actions on a Find Menu are now all rendered at the same level – not on sub-menus – as in most applications there would typically be just one or two finder actions in any one context.

Hidden attribute

The zero-parameter constructor of the [Hidden](#) attribute has been obsoleted. If a public property/method is never intended for use on the user interface then mark it with [\[NakedObjectsIgnore\]](#) (or make it a non-public method). If it is a property on a transient object then you can mark it with [\[Hidden\(WhenTo.Always\)\]](#).

Coding changes required to Run (Mvc) projects

Larger changes are needed to upgrade an existing Run (Mvc) project from NOF 6 to NOF 7 – and you might find that it is easier to create a brand new run project with NOF 7 installed and then copy across snippets of code from your existing run project where needed. Even if you don’t want to take that approach, we strongly recommend that you still create a clean new NOF 7 run project – at least for comparison. These are the main differences.

1. Your MVC ‘Run’ project(s) must be upgraded to .NET 4.5.2 before installing the [NakedObjects.Mvc-FileTemplates](#) package.
2. NOF 7 uses the Microsoft Unity framework to configure the components of the framework and inject them into each other as needed. (Note, this is separate from the mechanism that injects domain services into objects, the latter being an internal capability of the NOF and unaltered.) Installing the NOF will add a [UnityConfig](#) file that specifies all the components. *It is not necessary to edit this file for normal operation*, because the information needed for a typical application is delegated to a new [NakedObjectsRunSettings](#) file. (See [Application configuration](#) for full details). This has a somewhat similar structure to the old RunWeb class - and you will need to copy various things across, changing their form as needed:
 - a. You must now [specify the top level namespace\(s\)](#) that cover your domain model(s) in the `ModelNamespaces` property.
 - b. You must now [register all services](#) in the `Services` property, they are no longer separated into three groups (MenuServices, ContributedActions and SystemServices). Additionally, the services are now registered as types, not as instances.

- c. [Main menus](#) are now defined in the `MainMenus` property.
- d. The `EntityObjectStoreConfig` property plays a similar role to the `Persistor` property on `RunWeb`. The return type `EntityObjectStoreConfiguration` has a similar method signature to the former `IObjectPersistorInstaller`. See [Configuring the EntityObjectStore](#).
- e. If your domain model contains any types that are not ‘discoverable’ by the framework then you must [register these types](#) in the `Types` property. Note: if you are using `SimpleRepository<Foo>`, then the type `Foo` will not be discoverable by default, and hence no actions will show up on that repository! So in this case register the `Foo` type explicitly.
- f. The standard Controllers installed by the NOF (except for `RestfulObjectsController`) each now has a new constructor. If you have not modified these controllers in your own code, just take the new versions. Otherwise note that the new constructors all take this form:

```
public GenericController(INakedObjectsFramework nakedObjectsFramework) :
    base(nakedObjectsFramework) {
    // Uncomment following if you wish to have the Container and/or
    // services injected into the Controller
    // nakedObjectsContext.ContainerInjector.InitDomainObject(this);
}
```

(The injected implementation `INakedObjectsFramework` supersedes the old `NakedObjectsContext`. You may also inject it into your own custom controllers if needed, by providing a similar constructor. It may also be injected into domain objects for applications that need direct access to the framework - though this is not recommended unless you really need it as it would require your domain objects to have the `NakedObjects.Core` package installed, not just the `NakedObjects.ProgrammingModel`.)

- 3. There are some changes to `BundleConfig` and `RouteConfig`, `NakedObjects.css`, `NakedObjects-Ajax.js`, and `_NoLayout`. It is best just to take the new versions of any of these files if you can - if not you will need to identify the changes and merge with your own files.

Coding changes required to XAT projects

In existing XAT projects, the test methods themselves should not require any modification. However the set-up of the tests needs to be modified in line with the new approach to configuring and running applications. So upgrading the `NakedObjects.Xat` package will install Unity, for example. Even here, though, we have attempted to minimise the changes by providing methods/properties on `AcceptanceTestCase` which, though not identical to those in NOF6, at least mirror their broad structure so as to make the migration clearer:

- `AcceptanceTestCase` now has the following properties that work the same as for the `NakedObjectsRunSettings` class as described above for an Mvc Run project: `Namespaces`, `Services`, `Types`, `MainMenus`.
- However, we have also preserved the existing three separate properties for registering services as instances: `MenuServices`, `ContributedActions` and `SystemServices`. Unless

the new `Services` property is overridden to register services the new way, the default implementation will simply agglomerate those three previous lists of services and extract the types from the instances.

- `AcceptanceTestCase` requires the return of an `EntityObjectStoreConfiguration` instead of an `IObjectPersisterInstaller`, but, to facilitate migration, we have retained the property name `Persistor` on `AcceptanceTestCase` (where on `NakedObjectsRunSettings` it is called `EntityObjectStoreConfig`).
- Tests should now be initialised with this code:

```
[TestInitialize()]
public void TestInitialize() {
    InitializeNakedObjectsFrameworkOnce();
    StartTest();
}
```

- If you are using Fixtures within your XATs, see [Using object fixtures within XATs](#).

New features

- It is now possible to [inject multiple implementations of a service](#) interface into a domain object (or other service).

Getting started

If you're new to Naked Objects start by trying this ultra-simple example: [Writing your first Naked Objects application](#). Then you can progress in a slightly more formal manner.

A Naked Objects application will typically consist of multiple projects.

First, there will typically be one or more 'Model' projects, containing your domain classes. See: [Creating a domain model project](#)

Next you will need one or more 'Run' projects. If you want a ready-made user interface then you should add an MVC project. See: [Running your domain model\(s\) as a Naked Objects MVC application](#)

Later you might also decide to add a Restful API, as an alternative way to 'run' your domain model: [Creating and using a Restful Objects API](#)

Or the ability to run as a standalone executable, for example for Batch operations: [Running without a user interface](#)

You will also want to add one or more 'Test' projects. These may be conventional Unit Tests, but we strongly recommend that you also consider using the powerful Naked Objects XAT framework, to write functional or integration tests that are UI-independent: [Executable Application Tests \(XATs\)](#)

If you are sticking to the generic Naked Objects MVC User Interface then there is typically no need to then write UI tests (which are notoriously brittle to change). But if you customise the user interface (other than just by modifying the CSS) then you should write UI tests. We've written some helper classes to work with the respected Selenium framework, see: [End to End Testing with Selenium](#)

Writing your first Naked Objects application

Follow these steps to write your first ultra-simple Naked Objects application, featuring one simple domain class only:

1. Using Visual Studio 2013 or later (Express version is fine) create a new *ASP.NET Web Application* project called, say, *MyWebApp*, using the *MVC* template.
2. Set the project to use .NET 4.5.2 (this is important).
3. Install the NuGet Package *NakedObjects.Mvc-FileTemplates*, selecting *Yes To All* when asked if you wish to overwrite existing files.
4. In the Models folder add a new class *Customer* as follows. Note that:
 - All properties in a Naked Objects application must be virtual.
 - `[NakedObjectsIgnore]` specifies that this property is not for display on the user interface.
 - `[Title]` specifies that the value of the property should be displayed in the Tab.

```

using NakedObjects;

namespace MyWebApp {
    public class Customer {
        [NakedObjectsIgnore]
        public virtual int Id { get; set; }

        [Title]
        public virtual string Name { get; set; }
    }
}

```

5. Create a simple repository class as follows:

```

[DisplayName("Customers")]
public class CustomerRepository {

    //An implementation of this interface is injected automatically by the framework
    public IDomainObjectContainer Container { set; protected get; }

    public Customer CreateNew Customer() {
        return Container.NewTransientInstance<Customer>();
    }

    public IQueryable<Customer> AllCustomers() {
        return Container.Instances<Customer>();
    }
}

```

6. Create a `DbContext` object as follows. (This is standard Entity Framework Code First coding.)

```

using System.Data.Entity;

namespace MyWebApp {
    public class MyDbContext : DbContext {
        public DbSet<Customer> Customers { get; set; }
    }
}

```

7. In the `App_Start` folder find the `NakedObjectsRunSettings` class and edit three members as follows:


```

private static string[] ModelNamespaces {
    get {
        return new string[] { "MyWebApp" }; //Or the namespace you gave your project
    }
}

private static Type[] Services {
    get {
        return new Type[] { typeof(CustomerRepository) };
    }
}

public static EntityObjectStoreConfiguration EntityObjectStoreConfig() {
    var config = new EntityObjectStoreConfiguration();
    config.UsingCodeFirstContext(() => new MyDbContext());
    return config;
}

public static IMenu[] MainMenus(IMenuFactory factory) {
    return new IMenu[] {
        factory.NewMenu<CustomerRepository>(true)
    };
}
}

```

8. Run the project.
9. Using the actions on the *Customers* menu, try creating and retrieving Customer objects.

Creating a domain model project

Naked Objects uses Microsoft's Entity Framework to persist domain objects in a database. Most developers who work with Entity Framework now use it in 'Code First' mode - and this is what we now use throughout this manual. The name is slightly misleading: you can use 'Code First' mode even when creating an application to work against an existing database - 'Code First' just means that your persistence is entirely defined in program code (typically C#). Naked Objects can, however, work equally well with Entity Framework in the older mode, where the entity model is defined in XML with a .edmx file.

This manual does not attempt to provide an introduction to Entity Framework Code First development - rather it just emphasises what you need to do to make your project work with Naked Objects. We therefore recommend that you gain some general familiarity with Entity Framework Code First development: there are numerous on-line tutorials, and we also strongly recommend the book *Programming Entity Framework - Code First* by Julia Lerman and Rowan Millar.

When copying any domain code examples from the book or on-line Code First tutorials mentioned, please remember the following basic rules:

- Naked Objects requires that all properties are *virtual*.

- Naked Objects requires that all collections are `virtual`, and are initialised (but not in a constructor). See [Collection properties](#).
- Entity Framework Code First makes all properties optional in the database, unless specified as mandatory (using the `Required` attribute, or via the Code First Fluent API). However, at the user interface (in Naked Objects MVC) or Restful API (Restful Objects for .NET) the Naked Objects framework treats all properties as *mandatory*, unless marked up with the `Optionally` attribute - as we believe that this is the safer default behaviour.

To add a Model project, proceed as follows:

1. Install the [NakedObjects.Ide](#) package into your *solution*.
2. Create a new Class Library project.
3. Invoke Manage NuGet Packages, find and install the `NakedObjects.ProgrammingModel` package
4. Add POCO domain classes, following the simple programming conventions that are recognised by the Naked Objects framework. See [Domain object](#).
5. Create some services to act as Repositories/Factories.

Define your DbContext(s)

Next you will need to set up a `DbContext` class for your model project, which is defined in exactly the same way as for conventional Code First development. You can use the `Naked Objects > DbContext` item template to help create this class - as shown in the example below:

```
namespace DataAccess
{
    public class MyContext : DbContext {
        public MyContext(string name) : base(name) { }
        public MyContext() { }
        public DbSet<Foo> Foos { get; set; }
        public DbSet<Bar> Bars { get; set; }

        protected override void OnModelCreating(DbModelBuilder modelBuilder) {
            //Initialisation
            //Use the Naked Objects > DbInitialiser template to add a custom
            //initialiser, then reference thus:
            Database.SetInitializer(new
                DropCreateDatabaseIfModelChanges<MyContext>());

            //Mappings
            //Use the Naked Objects > Mapping template to add mapping classes &
            //reference them thus:
            modelBuilder.Configurations.Add(new Employee_Mapping());
        }
    }
}
```

The context inherits from the class `System.Data.Entity.DbContext` it defines methods to return one or more `DbSets`.

It is *not* necessary to define a `DbSet` for each of your domain classes - just for the 'root' classes in your model hierarchy.

Within the `OnModelCreating` method you may also:

- Add database mappings using the Code First 'Fluent API'. Use the Naked Objects > DbMapping item template to create a mapping class quickly.
- Add a database initialiser to determine when/whether to drop and re-create the schema. Use the Naked Objects > DbInitialiser item template to create an initialiser class quickly. You can use this initializer to seed the database.

Overriding the default database schema generation

By default, Entity Framework Code First creates the database schema by following a set of conventions, based on the class and property names. These convention-based schema may be over-ridden or enhanced, either by using Code First Data Annotations in the domain classes, or by means of the Code First Fluent API. The latter is invoked by creating one or more configuration classes (inheriting from `EntityTypeConfiguration<T>`) and referencing them from within the `DbContext`, as in the following example (quoted from Programming Entity Framework - Code First by Julia Lerman and Rowan Millar):

```

namespace DataAccessFluent {
    public class DestinationConfiguration : EntityTypeConfiguration<Destination>
    {
        public DestinationConfiguration() {
            Property(d => d.Name).IsRequired();
            Property(d => d.Description).HasMaxLength(500);
            Property(d => d.Photo).HasColumnType("image");
        }
    }

    public class LodgingConfiguration : EntityTypeConfiguration<Lodging> {
        public LodgingConfiguration() {
            Property(l => l.Name).IsRequired().HasMaxLength(200);
            Property(l => l.Owner).IsUnicode(false);
            Property(l => l.MilesFromNearestAirport).HasPrecision(8, 1);
        }
    }
    //...
    public class BreakAwayContextFluent : DbContext {
        public BreakAwayContextFluent(string name) : base(name) { }
        public BreakAwayContextFluent() { }
        public DbSet<Destination> Destinations { get; set; }
        public DbSet<Lodging> Lodgings { get; set; }
        //...
        protected override void OnModelCreating(DbModelBuilder modelBuilder) {
            modelBuilder.Configurations.Add(new DestinationConfiguration());
            modelBuilder.Configurations.Add(new LodgingConfiguration());
            // ...
        }
    }
}

```

Using data fixtures with Code First

When working Code First, you can create data fixtures (to pre-populate the database with) via the `Seed` method on the Database_INITIALIZER. The following example is quoted from *Programming Entity Framework - Code First* by Julia Lerman and Rowan Millar):

```

namespace DataAccess {
    public class DropCreateBreakAwayWithSeedData :
        DropCreateDatabaseAlways<BreakAwayContext> {
        protected override void Seed(BreakAwayContext context) {
            context.Destinations.Add(new Model.Destination {
                Name = "Great Barrier Reef" });
            context.Destinations.Add(new Model.Destination { Name = "Grand Canyon" });
            //...
        }
    }
}

```

1. This custom database initializer may be set within the `OnModelCreating` method on your `DbContext`.

Running your domain model(s) as a Naked Objects MVC application

Follow these steps to run your domain model(s) with the Naked Objects MVC user interface.

2. Create a new C# ASP.NET Web Application project, specifying MVC as the template, and set this as the Startup Project for your solution.
3. Invoke Manage Nuget Packages, find and install the [NakedObjects.Mvc-FileTemplates](#) package into this project
4. Add project reference(s) to your Model project(s).
5. If you are not using the Default database, then copy the connection string(s) from the [App.config](#) file(s) in your Model project(s) into the [Web.config](#) in your Run project. (Note that there are two [web.config](#) files in a standard MVC project - one within the Views folder - the purpose of which is to prevent direct access to the views without going through a controller - and the other at the project root level. The connection string needs to go into the latter.
6. [Register your services](#) and make any other changes you may require to the default run [configuration](#).
7. Run the project as a local app. This will launch a web-browser pointing at the start page of your application, running within [LocalHost](#).
8. Even if you selected the Internet project template, by default, the application is not initially set up to require a log-on. When you are ready to add authorization to your prototype, simply un-comment the commented-out the [Authorize](#) attribute on the [GenericController](#) and [SystemController](#) classes.
9. When you are ready, you can deploy the application to a server running IIS, in exactly the same way as any other ASP.NET MVC application. The simplest way is to right-click on the project and invoke the [Publish](#) action.

Programming Reference

Domain model - programming concepts

This section describes the key concepts in a Naked Objects application, and how they fit together. Together with the section [Adding behaviour to your domain objects - a how-to guide](#) this tells you all you need to know to create and deploy your own applications.

Domain object

Domain objects represent the persistent entities, or nouns, of the domain, such as: customers, products and orders.

In Naked Objects, any 'Plain Old CLR Object' (POCO) can function as a domain object - in other words a domain class does not have to inherit from any special class, nor implement any particular interface, nor have any specific attributes.

You can specify certain things about both the behaviour and presentation of domain objects by adding specific attributes or methods. See [Object presentation](#).

Property

For a property of a domain object to be recognised by Naked Objects, it must be `public` and `virtual`. There is no other special programming required. Use the `propv` snippet as a shortcut.

Value Property

A value property has a string, number, date, or other recognised value type. This will be rendered to the user as a textual field. Assuming that the user is allowed to modify that property, they may enter the value by typing in text, which will be validated and formatted according to the value type. (Certainly value types may provide alternative mechanisms for user input, such as a calendar-selector for a date field.)

Reference Property

A reference property is one where the type is another domain object. Reference properties are thus sometimes referred to as 'associations'. This will be rendered on screen as a field containing the referenced object (as an icon and title), to which the user may navigate. In edit mode, the user may specify the object to be associated by a number of mechanisms such as drag and drop, copy and paste, or selection from a drop-down list.

Collection Property

A collection property is a property that returns any of the recognised collection types. However, it is recommended that you specifically use `ICollection<T>`. Collections must be initialised. Collection properties are not paged: if the contents of the collection property are displayed (for example in the form of a list or a table) then the full contents will be shown.

Warning: It is therefore recommended that you never have large collections represented as collection properties. In other words do not represent associations that have large cardinality as being directly navigable. Where the number of associated objects may be large, then the navigation should be via action methods rather than via a collection property. This is recognised by many modellers as being good modelling practice - it is not just a constraint of Naked Objects.

You can specify certain things about both the behaviour and presentation of properties by adding specific attributes or methods. See [Properties](#) and [Collection properties](#).

Action

An action is a method that is intended to be invoked by a user - though it may also be invoked programmatically from within another method or another object.

By default, any public instance method that you add to a class (whether it is a domain class or a service) will be treated as a user action, provided that all its parameter types (if any) and its return type (if any) are types recognised by Naked Objects. A method will also *not* be treated as an action if it represents a property or another recognised method. Note also that `static` methods are ignored by Naked Objects.

Tip: Use the Action snippet (shortcut `act`) to create an action method.

If you have a method that you don't want to be made available as a user-action you should either:

- give it a non-public access modifier
- Mark it with the `NakedObjectsIgnore` attribute.

You can specify certain things about both the behaviour and presentation of actions by adding specific attributes or methods. See [Actions](#).

See also: [Contributed action](#).

When a method returns a reference the viewer will attempt to display that object. If the return value is `null` then nothing is displayed.

Actions that return a collection

An action (on a service or a domain entity object) may return any of the recognised collection types. A collection returned by an action will automatically be presented to the user in paged form. The default page size is 20 objects, but this may be overridden for an individual action using the `PageSize` attribute. If the number of objects in the returned collection is less than a single page full, or if the `PageSize` has been explicitly set to zero, then the page navigation controls will not be shown.

However, if you know that the collection being returned is likely to contain sufficient objects that paging will be shown, then it is strongly recommend that you return the collection as an `IQueryable<T>`. With the latter, the objects will automatically be retrieved from the database one page at a time. By contrast, if you return an `ICollection<T>` the entire collection of objects will be retrieved from the database, but only one page displayed; and this will be repeated for each page request.

Tip: The advantage of using `IQueryable<T>` is so great that it is recommended that you use this by default return for any action returning a collection, unless you have good reason not to.

Menus

Actions are typically rendered within Menus (unless, for example, you have written a custom view that renders one or more actions as free-standing buttons). There are two principle kinds of menu: Main Menu, which appear across the top of screen and provide the starting points for user interactions, and Object Menu which offer actions provided by an object instance. There is also a third type of menu, Find Menu, which are rather different in nature. The programming model is easier to understand by starting with object menus.

Object Menus

By default, the action menu for a domain object will be constructed from all the actions defined on that object, plus [contributed actions](#) (if any), recognising the order implied by the `MemberOrder` attributes on those actions (if specified). This may be overridden by providing a `static Menu` method as in the following example:

```
public static void Menu(IMenu menu) {...}
```

This method is called during the start-up process, and the Naked Objects framework will call it with a framework-generated implementation of `IMenu`, and which has its `Type` property set to the type of the object on which the `Menu` method is called. `IMenu` provides a number of methods for adding actions to the menu in an explicit order, and creating sub-menus. For example:


```

public static void Menu(IMenu menu) {
    menu.AddAction("Action2");
    menu.AddAction("Action1");
    var sub = menu.CreateSubMenu("Sub1");
    sub.AddAction("Action3");

    sub = menu.CreateSubMenu("Docs");
    sub.AddAction("Action4");
    menu.AddRemainingNativeActions();
    menu.AddContributedActions();
}

```

In this method, the ordering of actions is specified explicitly. Also sub-menus are defined. `IMenu` defines a number of other helper methods, not shown above, for constructing the custom menu.

Note that the action names are specified in the same format as the method names in code - even if any of the actions have been given a customised name using the [Named](#) or [DisplayName](#) attributes. (.NET 6 provides a `nameof` keyword and, we hope, this might allow the actions to be specified without the need for string literals).

Main Menus

Main menus offer actions that are defined on services. The menus are specified via the `MainMenus` method on `NakedObjectsRunSettings`. The following shows an example:

```

public static IMenu[] MainMenus(IMenuFactory factory) {
    var customerMenu = factory.NewMenu<CustomerRepository>();
    CustomerRepository.Menu(customerMenu);
    return new IMenu[] {
        customerMenu,
        factory.NewMenu<OrderRepository>(true),
        factory.NewMenu<ProductRepository>(true)
    };
}

```

The framework calls this method on start-up, passing in an implementation of `IMenuFactory`, which provides various methods for creating menus. In the above example, a default menu is created for each of two services (`OrderRepository` and `ProductRepository`), the `true` parameter indicating that *all* the actions from that type should be added into the menu, recognising the order implied by the `MemberOrder` attributes on those actions (if specified). But the `CustomerRepository` has its own static method (`Menu`) that defines a fully custom menu, which looks like this:

```

public static void Menu(IMenu menu) {
    menu.AddAction("FindCustomerByAccountNumber");
    menu.CreateSubMenu("Stores")
        .AddAction("FindStoreByName")
        .AddAction("CreateNewStoreCustomer")
        .AddAction("RandomStore");
    menu.CreateSubMenu("Individuals")
        .AddAction("FindIndividualCustomerByName")
        .AddAction("CreateNewIndividualCustomer")
        .AddAction("RandomIndividual");
    menu.AddAction("CustomerDashboard");
}

```

This method has the same structure and capabilities as the optional static `Menu` method on a domain object that defines an [object menu](#). However, a static `Menu` method on a *service* will *not* be called automatically – and must be called explicitly from the `NakedObjectsRunSettings.MainMenus` method, with a suitable menu object obtained from the factory, as shown above. This difference is deliberate, because Main Menus need not have a 1:1 relationship to underlying services.

In the example above, the three main menus still *do* correspond to three different services, but this is not a requirement. A main menu may combine actions from different services and, conversely, the actions on a single service might be split across multiple main menus. There need be no correspondence at all. When adding actions from multiple services, you will need to set the `Type` property of the `IMenu` to the service on which the action exists *before* calling `AddAction`, otherwise you will get an exception indicating that the requested action does not exist. For example:

```

public static void Menu(IMenu menu) {
    menu.WithName("My Hybrid Menu");
    menu.Type = typeof(ServiceA);
    .AddAction("ActionA1")
    .AddAction("ActionA2")
    menu.Type = typeof(ServiceB); // changing the type for the next additions
    .AddAction("ActionB1")
    .AddAction("ActionB2")
    .CreateSubMenu("Cs").
        .Type = typeof(ServiceC) // changing the type for the sub-menu now
        .AddAction("ActionC1")
    menu.AddAction("ActionB3") // menu itself still has type ServiceB
}

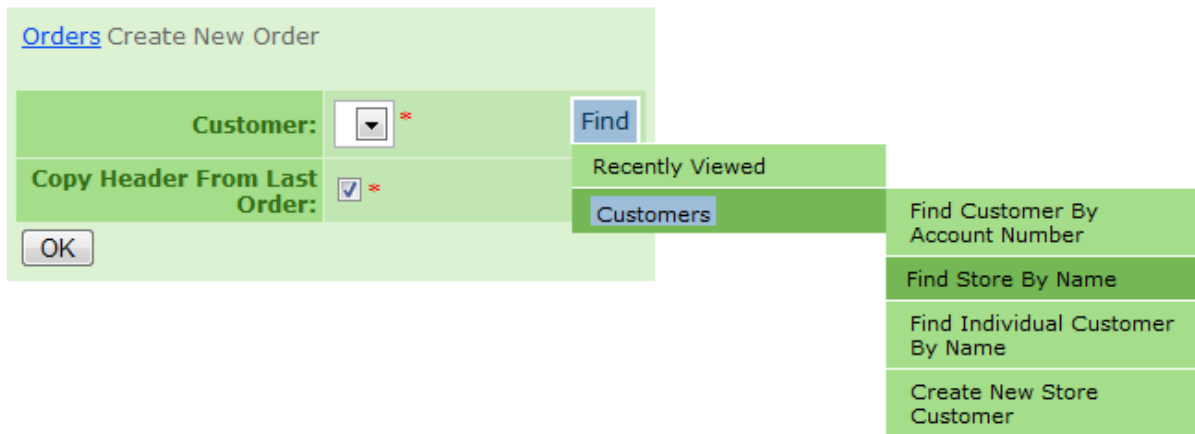
```

Find menus

Wherever the user is required to specify a reference to another domain object - either within an action dialog, or when editing a domain object - then it is preferable to provide one of the following two options:

- an explicit set of [choices](#)
- an [auto-complete](#) option

Where neither of these patterns has been provided by the domain model, then the UI will render a Find menu. An example is shown below:

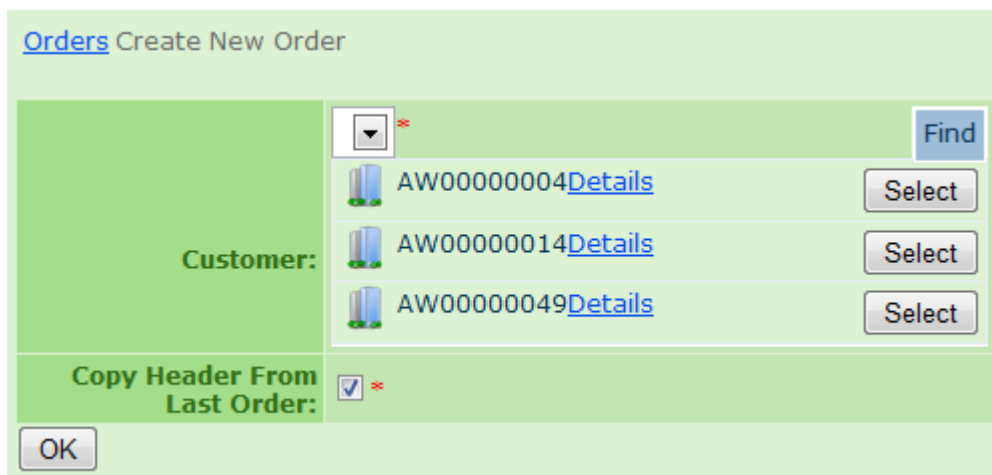


In the above example the Find menu has been rendered on the Customer parameter within an action dialog. The top-most option - Recently Viewed - appears on all instances of the Find menu. It will return a small set of objects that are of the required type and that have recently been displayed to the user. This set is continuously updated.

If the type of the parameter or property is defined by an Interface, then the Recently Viewed option will display recently-viewed objects of any type that implements that interface.

The other options on the Find menu will be service actions that return an object, or collection of objects, of the right type - and that have been marked up with a [FinderAction](#) attribute.

Invoking one of the Find menu actions will result directly in one of three things, as illustrated in the following screen shots. The first case is that the action will directly return one or more objects with a Select button next to each. Hitting Select will place a reference to the selected object within the required field.



The second case is where the Find menu action requires parameters. In this case a dialog will be rendered in situ.

Orders Create New Order

Customer:

Find Store By Name

Name:

Copy Header From Last Order:

Find

OK

OK

The Find menu will not permit the inclusion of any action that would ordinarily include a Find menu on one of its parameters. This is to avoid ending up with a Find dialog inside a Find dialog - which would be potentially very confusing to a user, and very difficult to implement as an HTML form.

Selecting OK will render the returned object(s) as shown previously.

The third case is where the Find menu action is to create a new object. Here the user will be presented with an unsaved object to be completed and saved, and then to be selected as before.

If the unsaved object requires one or more reference object properties to be specified, and these do not have an explicit set of choices, then it will not be possible to proceed - for the same reason given above that we cannot end up with Find menus inside Find menus.

Recognised method

Naked Objects uses 'programming by convention': methods with specific names are recognised by the framework as intending to specify a certain behaviour. Some of these recognised methods are standalone (e.g. `Title()`, `Created()`), but the majority take the form of method prefixes (e.g. `Validate[Something]`, `Choices[Something]`) which provide behaviour in relation to either a specific property or action. See the complete list of [Recognised Methods](#).

Recognised attribute

Naked Objects recognises a set of attributes that may optionally be applied to domain objects, properties, or actions, to alter presentation or behaviour. For the full list of recognised attributes see [Recognised .NET attributes](#) and [NakedObjects.Attributes](#).

View Model

Naked Objects is designed to expose persistent domain objects (or 'entities') directly to the user. This pattern encourages the development of a domain model that corresponds directly to the user's own representation of the problem domain.

On occasions, it may be desirable to present objects to the user that do not correspond directly to a persistent domain object - perhaps amalgamating information and/or functional behaviours from more than one underlying domain object. In these circumstances, you can use a View Model.

IViewModel and IViewModelEdit

In Naked Objects, a View Model is coded the same way as a domain object but must implement the interface `NakedObjects.IViewModel`, (or one of its two sub-types `IViewModelEdit` and `IViewModelSwitchable` – more on these shortly) which requires these two methods to be implemented:

- `DeriveKeys` must return a string array that defines the key(s) to this object. Typically these will be derived from the key(s) of the persistent object(s) that this view model represents, but could also directly represent value properties.
- `PopulateUsing` takes the same array of keys as a parameter and is used to re-populate the view model whenever it is referenced in a request (either to refresh the view or to invoke an action on the view model, for example).

These two methods are needed in order that the Naked Objects framework running on the server - which adopts a stateless pattern - can re-create the instance of the view model each time the client makes a request of it, making it appear (to the client) as if it were a persistent object.

Implementation of those two methods is best illustrated by example. In the code below, `CustomerComparison` holds a reference two separate `Customer` objects, for the purposes of comparison. Other properties (not shown in the code below) derive information from these two customers (or from associated objects) for presentation in the view model.

```

[NotMapped] //Needed if working with EF Code First, so that no table is created
public class CustomerComparison : IViewModel
{
    public IDomainObjectContainer Container { set; protected get; } //Injected
    service

    public virtual Customer Customer1 { get; set; }

    public virtual Customer Customer2 { get; set; }

    public string[] DeriveKeys()
    {
        return new string[] { Customer1.Id.ToString(), Customer2.Id.ToString() };
    }

    public void PopulateUsing(string[] keys)
    {
        int id1 = int.Parse(keys.ElementAt(0));
        Customer1 = Container.Instances<Customer>().Single(x => x.Id == id1);
        int id2 = int.Parse(keys.ElementAt(1));
        Customer2 = Container.Instances<Customer>().Single(x => x.Id == id2);
    }

    //Properties (not shown), derive their information from the two persistent
    Customers.
}

```

Note: It is *not* necessary to have a reference to the persistent Customer, if you don't want it. The view model could simply hold the Customer's Id.

In Naked Objects MVC view models are rendered using an `ObjectView.aspx` (or `.cshtml`) views, like any other object - either the generic one in the `Views > Shared` folder, or a custom one located in a folder named by the type of the view model.

`IViewModelEdit` and `IViewModelSwitchable`

`IViewModelEdit` is a specialised sub-type of `IViewModel`, with the only difference being that properties will be rendered as input fields (unless they are deliberately disabled) - and that the values will be taken on by the object when any action is invoked. In Naked Objects MVC, an `IViewModelEdit` will be rendered using the `ViewModel.aspx` (or `.cshtml`) view, not (as might be imagined) `ObjectEdit`. The reason for this is that the `ObjectEdit` view automatically renders a Save button, which is not applicable here. You might choose to write an action called `Save` (or `Next` or `Cancel`) on your editable view model, but it is up to you to decide what that (for example it might use the entered values to create one or more persistent objects).

`IViewModelSwitchable` allows a view model to be rendered either in view mode or in edit mode, depending on the value returned by the `IsEditView()` method that must then be provided. In Naked Objects MVC, an `IViewModelSwitchable` will be rendered using the `ViewModel.aspx` (or `.cshtml`) view when it is in edit mode, or using an `ObjectView.aspx` (or `.cshtml`) view when it is not in edit mode.

Actions on Editable View Models - Limitations

Editable view models were designed to support actions, with the principal intent of implementing multi-step processes or wizards - so you could add actions for e.g. Next, Previous, Start, and Finish. This is why - with the default `.css` - actions on editable view models are rendered as buttons rather than a drop-down menu. When you click on such an action, any values you have entered into the editable fields are submitted to the server; the view model re-created on the server; and its properties then updated with the values entered by the user before the action method is invoked (whether or not the action needs them).

The limitation is that this does not apply to actions that take parameters - and which thus return a modal dialog. In this case, when you hit OK on the dialog, the form sent back to the server is the dialog form, containing values you entered into the dialog NOT the form representing the fields on the underlying editable view model. As a secondary effect of this, if the action-with-parameters is void then when the underlying `ViewModel` is re-displayed, any values entered into that form will have been lost.

Instantiating a view model

To use any View Model, it should be created by means of a special method on the domain object container: `NewViewModel`, as shown in the example below:

```
var cc = Container.NewViewModel<CustomerComparison>();
cc.Customer1 = customer1;
cc.Customer2 = customer2;
return cc;
```

If your View Model is designed to present details from a single persistent domain object (optionally including information from other objects that are associated with that single object) - which is the most common scenario - then you may simply extend the `NakedObjects.ViewModel<T>` helper class. For example:

```
[NotMapped] //Needed if working with EF in Code First mode, so no table is created
public class MyCustomerViewModel : ViewModel<Customer>
{
    //Properties (not shown), derive their information from the 'Root' property
    (defined on ViewModel), which will be of type Customer, for example:

    public string Name {
        get {
            return Root.Name;
        }
    }
}
```

Concurrency Checking and View Models

For all uneditable view models, and even for some editable ones, concurrency checking is irrelevant. Where concurrency is relevant - for example where an editable view model is being used to update an underlying persistent entity - then the Naked Objects concurrency checking mechanism can be applied, in much the same way as for a persistent entity. You will need to have a single property on the view model, marked up with `[ConcurrencyCheck]`

that acts as the version (it can be hidden from the user). In a view model it would be common to make this version property derive from an equivalent version property on the underlying persistent entity - so that if that entity has been updated by another user then the submission of new values will fail. If your view model represents more than one underlying persistent entity, and you want to guard against updates to any of them, then you can make your derived version property be a hash of all the underlying ones.

Service

Services perform three roles in a Naked Objects application:

- To provide methods for creating and retrieving domain objects where the user does not have an existing object to navigate from. Services that perform this role are often referred to as [Factories & Repositories](#).
- To provide a bridge to external functionality. See [External or System service](#).
- To provide functionality that is to be shared by multiple classes of domain objects which do not necessarily have any common superclass. This is achieved through the concept of contributed actions whereby methods are written on a service but appear to the user as actions on a domain object. See [Contributed Action](#).

In whichever of these roles, a service is just an ordinary POCO class but without any state - just methods. Just like a domain object it does not have to inherit from any special class, nor implement any interface, nor include any specific attribute. What makes it a service is simply that it is [registered as a service](#). Registering the service instructs the NOF to inject that service into any domain object that needs access to it. See [Injection of domain services into domain objects](#).

Factories and Repositories

A repository is a service that provides method(s) for finding one or more domain objects. A factory is a service that provides method(s) for creating a new instance of a domain object class. Some domain modellers feel strongly that these two roles should be kept distinct; others see merit in merging them (in which case the term 'repository' is sometimes used to mean repository/factory). Naked Objects permits either style of coding.

Incidentally, there is no formal reason to stick to the idea of having a separate repository for each class: you might have a single repository to deal with a number of closely-related domain classes. And there is no need for any repository methods to deal with classes that are always retrieved by navigating from an associated object - as that functionality is provided automatically by the Naked Objects framework. You only need a repository when it is necessary to create and/or retrieve objects directly, rather than by navigating from an associated object.

Inheriting from AbstractFactoryAndRepository

A convenient way to create a Repository is to inherit from `NakedObjects.Services.AbstractFactoryAndRepository`. You can use the Factory and Repository template to create a sub-class of `AbstractFactoryAndRepository` with some helpful code formatting and comments using

`AbstractFactoryAndRepository` provides a number of methods that you can call from within your own methods, of which the two most commonly used are:

- `NewTransientInstance` returns a new object of a specified type in a transient (not yet persistent) state. If this transient object is returned to the user then it will appear in edit mode ready to be completed and then saved with the Save button.
- `Instances` returns all the instances of a specified type as an `IQueryable`, upon which LINQ queries can then be performed. There are two overloaded versions of this method: one is templated and should be used wherever the type of object being retrieved is known statically. The other takes a `System.Type` as a parameter and may be used where the type is not known statically.

Since the result is queryable, you may query it using LINQ to obtain more specific matches.

The helper method `SingleObjectWarnIfNoMatch` returns the first item from the `IQueryable` and converts it to an object of the corresponding type. If the `IQueryable` contained no items, a warning message will be given to the user as illustrated in this example:

```
public Employee FindEmployeeByName(string name) {  
    var query = from employee in Instances<Employee>()  
                where employee.Name == name  
                select employee;  
    return SingleObjectWarnIfNoMatch(query);  
}
```

Tip: Use the Find Query snippet (shortcut: `find`) to create a method that finds a single matching instance. Use the List Query snippet (shortcut: `list`) to create a method that returns a list of matching instances.

`AbstractFactoryAndRepository` also provides the `Random`, which returns a random instance of a specified type of object.

The use of these methods is illustrated in the following example code (note the use of the `DisplayName` to give the repository a friendlier name when presented to the user):

```
[DisplayName("Customers")]
public class CustomerRepository : AbstractFactoryAndRepository
{
    public CustomerRepository RandomCustomer()
    {
        return Random<Customer>();
    }
}
```

Writing a Repository from scratch

If you are unable or unwilling to inherit from `AbstractFactoryAndRepository` then you can write your own repository from scratch. You simply need to provide a write-only property into which the framework can inject a Domain Object Container, and then make use of the `NewTransientInstance` and `Instances` methods on that container, as illustrated in the following example:

```
public class CustomerRepository {

    public IDomainObjectContainer Container { set; protected get; }

    public Customer CreateNewCustomer() {
        return Container.NewTransientInstance<Customer>();
    }

    public Customer FindCustomerByNumber(string num) {
        var query = from obj in Container.Instances<Customer>()
                    where obj.Number == num select obj;
        return query.FirstOrDefault();
    }
}
```

Accessing one Repository from inside another

Sometimes, it is desirable to be able to access one Repository from within another: for example accessing the `ProductRepository` from within the `OrderRepository`. This is handled by [dependency injection](#).

External or System service

The second role that services perform within a Naked Objects application is as a bridge to external functionality. The following are examples of what we mean by bridging domains:

- Linking to functionality that already exists, or has to exist, outside of the Naked Objects application, such as pre-existing services, or functionality within legacy systems.
- Bridging between technical domains, such as between the object domain and the relational database domain, or the email domain.
- (Less commonly) Bridging between isolated modelling domains. The Naked Objects philosophy is to aim, where possible, for a single coherent enterprise object model

running within the same application space. Where this is not possible then services may be used to communicate between the domains without requiring common object definitions and/or identities.

The following example shows a service to send an email message. The service is defined by an interface `ISender`, which in turn defines a single method `SendTextEmail`. The following code defines a particular implementation of that service interface that uses the `SmtpClient` to send the message.

```
public class SmtpMailSender : AbstractService, ISender
{
    private static string SMTP_HOST_NAME = "localhost";
    private static string SMTP_USER = "admin@example.com";

    public void SendTextEmail(string toEmailAddress, string text)
    {
        SmtpClient client = new SmtpClient();
        client.Host = SMTP_HOST_NAME;

        MailMessage message = new MailMessage();
        message.Sender = new MailAddress(SMTP_USER);
        message.From = new MailAddress(SMTP_USER);
        message.To.Add(new MailAddress(toEmailAddress));
        message.Subject = "Expenses notification";
        message.Body = text;

        client.Send(message);
    }
}
```

Contributed action

A contributed action is an action that is defined on a service, but which is presented to the user as an action on an individual domain object or a collection of objects.

This is a very powerful feature of Naked Objects, but it is one that takes a bit of getting used to. A contributed action has some conceptual similarity to the .NET programming concept of an 'extension method', but they are not the same. Extension methods may be used within a Naked Objects application to provide internal functionality, but extension methods are not recognised as actions by Naked Objects. (Extension methods are also static, whereas contributed actions are instance methods on a service).

Actions contributed to individual objects

If an action defined on any service takes a domain object as one of its parameters, then adding a [ContributedAction](#) attribute to that parameter, means that action will automatically be contributed to any domain object of that type (or, if the parameter is an interface type, any domain object that implements that interface). The attribute may optionally specify a sub-menu that the action is to be contributed to.

In the following example, the method `CreateNewTask` is defined on a service called `TaskContributedActions`. The method will appear as an action on any domain object that implements `ITaskContext` - under a sub-menu called `Tasks`.

```
public class TaskContributedActions : AbstractService
{
    public Task CreateNewTask([ContributedAction("Tasks")] ITaskContext context)
    {
        var task = Container.NewTransientInstance<Task>;
        task.Context = context;
        return task;
    }
    //...
}
```

Because the method `CreateNewTask` takes just one parameter, the action will appear on the domain object as though it was a zero-parameter action - under the covers the method on the service will be called, but with the domain object from which it was initiated as the parameter. If the method had multiple parameters then it would appear on the domain object as a multi-parameter method which, when invoked, will return a dialog box showing all parameters, but with the domain object representing one of those parameters.

Actions contributed to query results

As well as being contributed to individual domain objects, actions may also be contributed to collections of domain objects that are returned as the result of queries.

Any such action needs to be declared on a service, just like other contributed actions, but will take an `IQueryable<T>` (where `T` is a domain entity type) as one of the parameters. If the parameter is marked up with `ContributedAction` this method will show up on the action menu on any `IQueryable<T>` returned as the result of some kind of finder or query action. You may invoke the action on all the elements in the query result, or you may select a sub-set of objects - in which case, behind the scenes, a new collection will be formed from just those objects selected and this new collection will be passed into the action method as a parameter.

This is best illustrated by example, taken from an accounting application, where the class `Transaction` represents an accounting transaction and has an instance method `MarkAsReconciled`. In addition there is a method also called `MarkAsReconciled`, declared on a service called `TransactionContributedActions`, but which takes an `IQueryable` of `Transaction` as its parameter:

```
public void MarkAsReconciled(
    [ContributedAction()] IQueryable<Transaction> transactions) {

    foreach (Transaction t in transactions.ToList())
    {
        t.MarkAsReconciled();
    }
}
```

Note: the sub-menu parameter of the ContributedAction attribute, is ignored for query-result-contributed actions. (It would be unusual to have more than just a handful of actions contributed to any one type of collection.

The following limitations apply to query-result-contributed actions:

- Query-result-contributed actions may not themselves return a collection. This is because the stateless architecture of Naked Objects MVC would not allow the resulting collection to be re-created automatically. However, if you have a requirement to return a collection of objects from the action, then you can do this by returning a View Model that is capable of re-generating the same collection on demand.
- Actions may be contributed only to `IQueryable<T>` and not to, say, `ICollection<T>` or `IEnumerable<T>`.
- Where the results of a query are large enough to be presented as more than one page of results, then selection of objects (for the purpose of invoking a contributed action to that selection) operates only within a single page. It is not possible to make a selection that spans multiple pages of a query result. However, it is possible to expand the [page size](#).

Injection of domain services into domain objects

Naked Objects has in-built mechanism to inject registered domain services into domain objects (or, indeed, into other domain services). This mechanism is separate from the Dependency Injection mechanism (Unity, by default) that manages the [configuration of the system](#).

To use capability, simply provide a property that specifies the type of service required. The property should have a `protected get`. This type may be a concrete class (such as `CustomerRepository`) or it may be an interface (such as `ICustomerRepository`). The advantage of the latter is that you can then have multiple implementations of that interface. It also means that you can have a single class that implements several separate services each defined by a separate interface (this is sometimes useful during prototyping in particular).

This is illustrated in the following example, where a `Product` object has a method to find products of similar colour, which it delegates to the `ProductRepository`.

```
public class Product
{
    public string Colour { }

    public ProductRepository ProductRepository { set; protected get; }

    public IQueryable<Product> FindOtherProductsOfSameColour()
    {
        return ProductRepository.ListProductsForColour(this.Colour);
    }
}
```

Tip: Use the `injs` snippet to create this code.

It is also possible to register multiple services that implement a common interface, and inject them as an array, as illustrated below:

```
public class Product
{
    public IPricingMechanism[] PricingMechanisms { set; protected get; }

    public void CalculateBestPrice()
    {
        this.Price = PricingMechanisms.Min(pm => pm.Price(this));
    }
}
```

Note however that if you do have multiple implementations then you *must* use an array. A property allowing only a single implementation, i.e:

```
public IPricingMechanism PricingMechanism { set; protected get; }
```

Will result in a run-time exception being thrown. However, the reverse is fine: you can inject a single implementation into a matching array property.

The Domain Object Container

The design ethos of Naked Objects is that it is the responsibility of the framework to determine how the application should run, by calling upon the domain objects - not vice versa. However, there are a few situations where it is necessary for the domain objects to be able to communicate with the framework.

This is done via the domain object container, which is defined by the interface [NakedObjects.IDomainObjectContainer](#).

The container is accessed by means of dependency injection. Any domain object (whether it represents an entity or a service) that needs to call any of the container methods, simply needs to provide a property of type `IDomainObjectContainer`. The property should have a protected `get` method.

```
public IDomainObjectContainer Container { set; protected get; }
```

The framework recognises that this property is not intended for display to the user, so there is no need to mark it as `Hidden`. The property is not persisted.

Tip: Use the Injected Container snippet (shortcut `injc`) for creating this property.

The object may then invoke any of the methods on the container e.g.:

```
var cust = Container.NewTransientInstance<Customer>();
```

IDomainObjectContainer methods

`NakedObjects.IDomainObjectContainer` (which may be found in `NakedObjects.Attributes.dll`) defines the members available on the container object. An implementation of this interface will be automatically injected by the container into any object that provides a property of type `IDomainObjectContainer`.

- `NewTransientInstance` returns a new object of a specified type in a transient (not yet persistent) state. If this transient object is returned to the user then it will appear in edit mode ready to be completed and then saved with the Save button. Alternatively, the programmer may persist the transient object with ...
- `Persist(object)`. Note that an object is only ever persisted once - thereafter it remains in a persistent state (and is managed by the object Persistor) and its properties are updated as needed. Calling `Persist` on an object that is already persistent will throw an error.
- `IsPersistent(object)` allows you to check if a particular object is already persistent.
- `DisposeInstance(object)` allows you to delete a persistent object. The programmer must take responsibility to ensure that any references to this object from other associated objects are cleared as part of the same transaction.
- `Instances` returns all the instances of a specified type as an `IQueryable`, upon which LINQ queries can then be performed. There are two overloaded versions of this method: one is templated and should be used wherever the type of object being retrieved is known statically. The other takes a `System.Type` as a parameter and may be used where the type is not known statically.
- `Principal` returns a `System.Security.Principal.IPrincipal` that represents the logged on user. From this `IPrincipal` it is possible to object the name and/or roles of the user.
- `InformUser(string)` sends an informational message to the user.
- `WarnUser(string)` sends a warning message to the user.
- `RaiseError(string)` sends an error message to the user.
- `Resolve(object)` ensures that an object has been fully resolved from the Persistor. Note that there is no need to call this method explicitly in a typical Naked Objects application since it is called automatically by the proxy object that is created behind-the-scenes by the Entity Framework.
- `ObjectChanged(object)` is used to notify Naked Objects that an object has changed and that those changes may need to be notified to the object Persistor and/or to the user interface. Note that there is no need to call this method explicitly in a typical Naked Objects application since it is called automatically by the proxy object that is created behind-the-scenes by the Entity Framework.

- `Principal` (property) returns a `System.Security.Principal.IPrincipal`, from which you may obtain the user's identity and thence the name.
- `AbortCurrentTransaction`. Any uncaught exception thrown within an action method will automatically abort the transaction. If you wish to abort a transaction without throwing an exception you may use this method.
- `NewTitleBuilder()` (plus two overloads). Returns an implementation of `ITitleBuilder`, which provides convenience methods for building object titles from properties and references.
- `TitleOf(object)` As an alternative to using `NewTitleBuilder`, this method provides a convenient way to obtain the title of another domain object, without having to know how that title is defined.

Application configuration

As an application developer, there are things that you need to specify in order to run an application under the Naked Objects framework, such as:

- Specify the Namespace(s) that cover your domain model (so that the Naked Objects introspector knows which types to ignore).
- Register domain services
- Specify main menus
- Configure the `EntityObjectStore`
- Register any domain entity types that cannot be navigated to directly, or indirectly, from any of the methods on the registered service

All such application configuration is done via specific methods on the `NakedObjectsRunSettings` class – which is found your run project(s). These methods are described in the following sections

Specifying the Namespace(s) that cover your domain model

At start-up time, the Naked Objects ‘Reflector’ introspects the domain model to build an internal metamodel. It does this by starting from the domain services that you have registered (see below) and, from the method signatures on those services, traversing the whole model. It is necessary to specify the namespace(s) that encompass your model, so that the reflector does not disappear off into, for example, library code that provides only technical capabilities, not domain objects. You specify the namespaces using the `ModelNamespaces` property on `NakedObjectsRunSettings`:


```
private static string[] ModelNamespaces {  
    get {  
        return new string[] { "Model1", "Model2" };  
    }  
}
```

If all your domain classes (including services) fall within a single root namespace, you need register only that root. Note that the namespace(s) need only cover the domain classes and services, not the run project classes, which do not need to be introspected.

Registering domain services

All domain services, whether they provide methods intended to be menu actions, or just technical capabilities, and that may need to be [injected](#) into domain objects (or into other domain services) should be registered – as types – in the services property on `NakedObjectsRunSettings`, for example:

```
private static Type[] Services {  
    get {  
        return new Type[] {  
            typeof(CustomerRepository),  
            typeof(OrderRepository),  
            typeof(ProductRepository),  
            typeof(EmployeeRepository),  
            typeof(SalesRepository),  
            typeof(SpecialOfferRepository),  
            typeof(ContactRepository),  
            typeof(VendorRepository),  
            typeof(PurchaseOrderRepository),  
            typeof(WorkOrderRepository));  
        typeof(OrderContributedActions),  
        typeof(CustomerContributedActions),  
        typeof(SimpleEncryptDecrypt);  
    };  
}
```

Specifying any types that will not ordinarily be discovered by the reflector

For most simple domain models, the framework's introspector will be able to find all the domain types just by traversing the code from the methods on the registered services. However, in more complex domain models, you might have domain types that cannot be reached this way. The most common example of this is where a domain type has sub-types that are created/retrieved inside a method, but where the method signatures deal only in the super-types.

If an unknown domain type is encountered at run-time an error will be thrown because the framework will not be able to find any metadata about it. The exception message will typically be a `NakedObjectsSystemException` with the message *'failed to find spec for [type]'*.

To avoid this, you need to register any of these undiscoverable domain types in the `Types` property on `NakedObjectsRunSettings`, for example:

```
private static Type[] Types {
    get {
        return new[] {
            typeof (CustomerCollectionViewModel),
            typeof (OrderLine),
            typeof (QuickOrderForm)
            //These types must be registered because they are defined in
            NakedObjects.Mvc, not in Core.
            typeof (ActionResultModelQ<>),
            typeof (ActionResultModel<>)
        };
    }
}
```

In the example above, the first three registered types are from the application domain model. The final two are types used by Naked Objects Mvc to display query results. They need to be registered here because they do not form part of the core framework (they are used in Naked Objects Mvc only). You will see that these two types are already registered in the template version of `NakedObjectsRunSettings` that is added on a new install.

It is not necessary to register all domain types - only the undiscoverable ones. However, there is no harm in explicitly registering a type that may also be discovered automatically. Indeed, if your model contains only domain types, you might be able register them all in one go using something like this code:

```
private static Type[] Types {
    get {
        var allTypes = AppDomain.CurrentDomain.GetAssemblies()
            .Single(a => a.GetName().Name == "AdventureWorksModel").GetTypes();
        return allTypes.Where(t => (t.BaseType == typeof(AWDomainObject)) &&
            !t.IsAbstract).ToArray();
    }
}
```

Configuring the EntityObjectStore

The `EntityObjectStore` can be configured to operate either in 'code-first' mode ...

```
public static EntityObjectStoreConfiguration EntityObjectStoreConfig() {
    var config = new EntityObjectStoreConfiguration();
    config.UsingCodeFirstContext(() => new MyDbContext());
    return config;
}
```

or with a `.edmx` file (though `.edmx` is now considered an out-dated way of working as is expected to be dropped from Entity Framework version 7)...

```
public static EntityObjectStoreConfiguration EntityObjectStoreConfig() {
    var config = new EntityObjectStoreConfiguration();
    config.UsingEdmxContext("Model");
    return config;
}
```

Configuring Authorization

(Read the section on [Authorization](#) first).

You will need create an instance of an implementation of `IAuthorizationConfiguration`. A convenient place to create this instance is in `NakedObjectsRunSettings`, where there is already a default (null) implementation. For example:

```
public static IAuthorizationConfiguration AuthorizationConfig() {
    var config = new AuthorizationConfiguration<MyDefaultAuthorizer>();
    config.AddNamespaceAuthorizer<MyAppAuthorizer>("MyApp");
    config.AddNamespaceAuthorizer<MyCluster1Authorizer>("MyApp.MyCluster1");
    config.AddTypeAuthorizer<Bar, MyBarAuthorizer>();
    return config;
}
```

Each authorizer must implement `NakedObjects.Security.INamespaceAuthorizer` or `ITypeAuthorizer<T>` where T is a specific domain type. The 'default authorizer' (and only that one) must implement `ITypeAuthorizer<object>`

This authorization config must then be registered in `Unity.config`, and you must also register the `AuthorizationManager` as a facet decorator. If you created your Authorization config in `NakedObjectsRunSettings` then the registrations will happen automatically, because of this default code:

```
if (NakedObjectsRunSettings.AuthorizationConfig() != null) {
    container.RegisterType(typeof(IFacetDecorator), typeof(AuthorizationManager),
        "AuthorizationManager", new ContainerControlledLifetimeManager());
    container.RegisterInstance(typeof(IAuthorizationConfiguration),
        NakedObjectsRunSettings.AuthorizationConfig(), new
        ContainerControlledLifetimeManager());
}
```

Configuring Auditing

(Read the section on [Auditing](#) first).

You will need to create an instance of an implementation of `IAuditConfig`. A convenient place to create this instance is in `NakedObjectsRunSettings`, where there is already a default (null) implementation. For example:

```
private static IAuditConfiguration MyAuditConfig() {
    var config = new AuditConfiguration<MyDefaultAuditor>();
    config.AddNamespaceAuditor<MyAuditor1>("MySpace.Foo");
    config.AddNamespaceAuditor<MyAuditor2>("MySpace.Bar");
    return config;
}
```

You will need to have a default auditor, and then you may optionally register an unlimited number of other auditors for specific namespaces each of which must implement `NakedObjects.Audit.IAuditor`.

This audit config must then be registered in `Unity.config`, and you must also register the `AuditManager` as a facet decorator. If you created your Authorization config in `NakedObjectsRunSettings` then the registrations will happen automatically, because of this default code:

```
if (NakedObjectsRunSettings.AuditConfig() != null) {
    container.RegisterType(typeof(IFacetDecorator), typeof(AuditManager),
        "AuditManager", new ContainerControlledLifetimeManager());
    container.RegisterInstance(typeof(IAuditConfiguration),
        NakedObjectsRunSettings.AuditConfig(),
        new ContainerControlledLifetimeManager());
}
```

Configuring Profiling

(Read the section on [Profiling](#) first).

You need to configure your implementation of `IProfiler`, together with the set of events to be profiled, using an `IProfileConfiguration`. A convenient place to do this is in `NakedObjectsRunSettings` using the concrete `ProfileConfiguration<T>` class:

```
public static IProfileConfiguration ProfileConfig() {
    var events = new HashSet<ProfileEvent>() { ProfileEvent.ActionInvocation }; //etc
    return new ProfileConfiguration<MyProfiler>() { EventsToProfile = events };
}
```

This profile configuration must then be registered in the `UnityConfig` file. `ProfileManager` must also set up as a facet decorator, for example:

```
if (NakedObjectsRunSettings.ProfileConfig() != null) {
    container.RegisterType(typeof(IFacetDecorator), typeof(ProfileManager),
        "ProfileManager", new ContainerControlledLifetimeManager());
    container.RegisterInstance(typeof(IProfileConfiguration),
        NakedObjectsRunSettings.ProfileConfig(),
        new ContainerControlledLifetimeManager());
}
```

Configuring the RestRoot

The root URL for the Restful Objects API is specified on the RestRoot property of the `RestfulObjectsConfig` class, for example:

```
public static string RestRoot {  
    get { return "api"; }  
}
```

If a Restful Objects API is not required, return `null` (the default).

System configuration using the Unity framework

Naked Objects uses ‘dependency injection’ (DI) to wire-together the various components of the framework. By default Naked Objects uses the Microsoft Unity framework to manage this DI. It is possible to use another DI framework (sometimes called a ‘DI container’), such as Ninject, Spring, or any other framework that supports ‘constructor injection’.

Having installed `NakedObjects.Mvc-FileTemplates` into your MVC project (the pattern is broadly similar for other types of ‘run’ project), you will notice that the `App_Start` folder contains a file `UnityConfig.cs`. This is where the application is configured. The `RegisterTypes` method defines all the different components that are used within the framework.

It is not necessary for an application developer modify any of the code in the `RegisterTypes` method in order to run an application. Indeed you would typically only modify the list of registered types if you wanted to replace any of the default components of the framework in order to significantly change its behaviour. For that you would anyway need to have developed a good understanding of the internal operation of the framework.

One case where you might need to edit the `UnityConfig.cs` file is to ...

Configure Localization and Internationalization

See [Internationalisation / Localisation](#). To use internationalisation, you need to register the `I18NManager` as a ‘Facet Decorator’ with the following line of code:

```
container.RegisterType(typeof(IFacetDecorator), typeof(I18NManager), "II18NManager",  
    new ContainerControlledLifetimeManager());
```

The Naked Objects programming model

The Naked Objects programming model defines the ways in which domain model code is recognised by the Naked Objects framework. Broadly there are two main aspects to this. The first is a set of recognised conventions - which include a set of .NET types, .NET attributes,

and method names that will be interpreted by the Naked Objects framework when you run your domain model against it.

The second is a set of specific artifacts that are installed via the `NakedObjects.ProgrammingModel` NuGet package, in three assemblies: `NakedObjects.Attributes`, `NakedObjects.Types`, and `NakedObjects.Helpers`, and which you may choose to reference within domain model code.

Recognised Value Types

Naked Objects recognises the following .NET types are recognised as 'value objects':

- `System.Boolean`
- `System.Byte`
- `System.Byte[]` (represents a 'blob' - typically an attached file)
- `System.Character`
- `System.Decimal`
- `System.Double`
- `System.Enum` (strictly speaking, sub-classes of `Enum` that are defined and used within the domain model)
- `System.Float`
- `System.Single`
- `System.Int16`
- `System.Int32`
- `System.Int64`
- `System.SByte`
- `System.UInt16`
- `System.UInt32`
- `System.UInt64`
- `System.String`
- `System.DateTime`
- `System.TimeSpan`
- `System.Guid`

Recognised Collection types

Collections are recognised by Naked Objects in two different contexts:

1. As returned by an action (on an object or service), for example an action that finds, or even creates, multiple objects
2. As a property on an object that represents a multiple association

These are defined below.

Collections returned by an action

In the this context Naked Objects recognises as a collection any implementation of `IEnumerable<T>` where `T` is a domain entity type (class or interface). For example, it will recognise any of the following:

```
public IEnumerable<Customer> Xxx() {}  
public IQueryable<IPerson> Xxx() {}  
public ICollection<IDocument> Xxx() {}  
public IList<Employee> Xxx() {}  
public SalesOrder[] Xxx() {}
```

It will also recognise an untyped collection (`System.Collections.ICollection`), though it is recommended that you always type the collection whenever possible as this gives the option to render the results in table form.

The framework deliberately does not recognise as collections implementations of `IEnumerable<T>` where `T` is a value type, or any other type not recognised as a domain entity type. For example, it will *not* recognise any of the following:

```
public IEnumerable<decimal> Xxx() {}  
public IQueryable<DateTime> Xxx() {}  
public ICollection<int> Xxx() {}  
public IList<IComparable> Xxx() {}  
public String[] Xxx() {}
```

Collections as properties on an object

For a property that represents a multiple association, Naked Objects will recognise any implementation of `ICollection<T>`, for example:

```
public ICollection<IDocument> Xxx {get{} set{}}  
public IList<Employee> Xxx {get{} set{}}  
public SalesOrder[] Xxx {get{} set{}}
```

Properties returning `IQueryable<T>` are not recognised by Naked Objects.

Recognised .NET attributes

Naked Objects recognises the following .NET attributes.

ConcurrencyCheck

`System.ComponentModel.DataAnnotations.ConcurrencyCheck`

Allows you to specify the property on an object that participates in concurrency checking. This could be any type of property, but is most commonly a `DateTime`. You must ensure that the value of this property changes with each update of the object. The typical way to do this

is to implement the behaviour in the database; however you may choose to implement the behaviour in your domain model, for example within the [Updating](#) life-cycle method.

See Microsoft documentation.

ComplexType

The `System.ComponentModel.DataAnnotations.Schema.ComplexType` attribute is used to indicate that a domain class is treated as a complex type by Entity Framework (you will need to add the Entity Framework NuGet package to your model project if you want to use it). The annotation is also detected, independently, by the Naked Objects framework: this is why (from NOF 7.0 onwards) you must annotate all complex types with this attribute, even if you have specified that a type is a complex type using the Entity Framework code-first fluent mappings.

This means that the annotated type will always stored in-line with its parent object. In the following example, the `Person` object has a property of type `Name`, which in turn has properties `FirstName` and `LastName` and has been annotated with the `Inline` attribute. This indicates that the `FirstName` and `LastName` properties will in fact be stored as though they were direct properties of the `Person` object - in the same table if you are working with a relational database. It also means that the identity of the `Name` object is tied to the identity of the `Person` object - so no other persistent object should attempt to hold a reference directly to that `Name`. In the user interface the user will only be able to edit the `Name` when the `Person` is being edited also. (See also the [Root](#) attribute).

```
public class Person {  
    public virtual Name FullName { get; set; }  
}  
  
[ComplexType]  
public class Name {  
    public virtual string FirstName { get; set; }  
    public virtual string LastName { get; set; }  
}
```

DataType

Adding `[DataType(DataType.Password)]` to a string parameter on an action, will cause the content to be rendered as an obscured password.

This should not be used on string properties as - though it will obscure the content in edit mode - it will not obscure the content in view mode. (Normally, if a Password needs to be stored on an object then it would be marked as Hidden anyway, and should only be update-able through an action method.

DefaultValue

`System.ComponentModel.DataAnnotations.DefaultValue`

Allows you to specify, declaratively, the default value for any value input field.

See Microsoft documentation.

Description

`System.ComponentModel.Description`

May be used as an alternative to the [DescribedAs](#) attribute in order to specify additional information that may be provided to the user (for example as a tool-tip).

See Microsoft documentation.

DisplayName

`System.ComponentModel.Description`

May be used as an alternative to the `NakedObjects.Named` attribute to specify the name of a property or object that overrides the default name.

See Microsoft documentation.

MaxLength

`System.ComponentModel.DataAnnotations.MaxLength`

`MaxLength` is included with .NET 4.5, or, if running under .NET 4.0, with the Entity Framework package v 5.0.

Allows you to specify the maximum length that the user may input to a string property or action parameter. This is an alternative to the [StringLength](#) attribute.

The `MaxLength` attribute indicates the maximum number of characters that the user may enter into a `String` property, or a `String` parameter in an action. (It is ignored if applied to a property or parameter of any other type.) For example:

```
public class Customer
{
    [MaxLength(30)]
    public virtual string FirstName() { get; set; }
}
```

MinLength

`System.ComponentModel.DataAnnotations.MinLength`

This may be used in conjunction with an `AutoComplete` method. See [How to specify auto-complete for a property](#) and [How to specify auto-complete for a parameter](#).

Range

`System.ComponentModel.DataAnnotations.Range`

See Microsoft documentation.

Range may be used to specify the minimum and/or maximum value for a user input to a numeric or date, property or parameter.

When applied to a date property or parameter, the range values represent the number of days relative to today. Thus `Range(1, 30)` means any day from tomorrow to 30 days from now (inclusive) and `Range(-30, 0)` means any of the last 30 days including today.

RegularExpression

`System.ComponentModel.DataAnnotations.RegularExpression`

This may be used as an alternative to the [Regex](#) attribute - to specify a regular expression that input text must satisfy on a string property. Note, however, that unlike the [Regex](#) attribute you cannot specify a formatting expression, nor explicitly control the case-sensitivity.

Required

Note that [Required](#) may also be used when operating Code First to ensure that the column in the generated database is Not Nullable. (At present this applies only to primitive properties, not to reference properties).

ScaffoldColumn

This may be used to hide properties in the viewer. It is an alternative to the [Hidden](#) attribute. `ScaffoldColumn(False)` is equivalent to `Hidden` or `Hidden(WhenTo.Always)`, and `ScaffoldColumn(True)` is equivalent to `Hidden(WhenTo.Never)`.

StringLength

`System.ComponentModel.DataAnnotations.StringLength`

Allows you to specify the maximum length that the user may input to a string property or action parameter. This is an alternative to the [MaxLength](#) attribute.

At present, Naked Objects does not support the [MinimumLength](#) property of the [StringLength](#) attribute.

Recognised Methods

This section defines the explicit method conventions that are defined by the 'Naked Objects (.NET) Programming Model'. All such methods must be public and should start with an upper-case letter.

Public methods are, by default, are deemed to be action methods that we expect the user to invoke via the user interface:

```
public void <actionName>([<parameter type> param]...)
public <return type> <actionName>([<parameter type> param]...)
```

When such a method returns a reference the framework will attempt to display that object. If the method returns a null value then nothing will be displayed.

The exception to this rule are methods that follow specific conventions. These may be divided into three broad categories:

- Complementary methods.
- LifeCycle methods
- Other recognised methods.

The specific recognised methods names are listed below under those three headings.

Complementary methods

These methods complement an action or a property. They take the form of a prefix followed by the corresponding action or property name. Examples of these prefixes include: [Validate](#), [Modify](#).

A complementary method must be declared on the same class as the action or property that it complements. A complementary method may be overridden in a sub-class without having to override the action or property that it complements.

- [AutoComplete](#): Used to generate a dynamic drop-down list of matching objects or values.
- [Choices](#): A complementary method used in conjunction with a property or an action. A [Choices](#) method specifies a set of explicit choices from which the user must select for a particular property (when editing an object) or for a parameter within an action dialog.
- [Clear](#): A complementary method used in conjunction with a property, the [Clear](#) method is called when the user (rather than the framework) clears a reference field, or blanks (so there is no entry) a value field. Changing a property from one value to another value, is deemed by the framework to be a 'clear field', immediately followed by a 'modify field'.
- [Default](#): A complementary method used in conjunction with a property or an action. See [How to specify a default value for a property](#) and [How to specify a default value for a parameter](#).
- [Disable](#): A complementary method used in conjunction with a property or an action. The [Disable](#) dynamically controls whether a field is editable, or an action can be initiated. If a [String](#) is returned the field or action is disabled and the [String](#) is made visible to user to inform them why it is disabled. If the method returns a null value then field or action remains enabled.

- **Hide:** A complementary method used in conjunction with a property or an action. The `Hide` method allows a property or action to be dynamically hidden from the user. This is typically used for security reason, such as hiding a field once it is set up. Returning a true value makes the property or action invisible.
- **Modify:** A complementary method used in conjunction with a property. The `Modify` method is called when the user (rather than the framework) sets a reference or value field. This is typically used to initialise an association (where an association is combination of references, such as a back link), or to trigger other behaviours such as updating a total.
- **Validate:** A complementary method used in conjunction with a property or an action

LifeCycle methods

The following is a list of methods that correspond to various events in the life-cycle of a domain object. If a domain object implements any of these methods (they are all optional) then the framework will call that method whenever the corresponding event occurs. For example, `Persisted()` is called after an object has been persisted.

- **Created:** Life cycle method called by framework when an object is first created. This is the instance's logical creation. This method will *not* be called when the object is retrieved from persistent storage into memory.
- **Deleted:** Life cycle method called by framework when an object has just been removed from the persistent store. At this point the object will exist in memory, but no longer exist in the persistent store.
- **Deleting:** Life cycle method called by framework when an object is just about to be removed from the persistent store. At this point the object still exists in the persistent store.
- **Loaded:** Life cycle method called by framework when an object has just been loaded in from the persistent store. At this point the object has had its state fully restored. `Loaded` will be called after the object has been loaded and before the transaction has completed. When retrieving an object via the user interface this means that `Loaded` will have been called by the time the object appears on the screen. However, if you are processing objects programmatically - whether from within a user action or from an external call - then be aware that the `Loaded` might not be called on any (or all) of the objects being processed until the very end of the transaction. So if your method involves loading objects and processing them, you cannot assume that `Loaded` will have been called before you get hold of each object. In general, it is recommended that you use `Loaded` only for very simple, non-invasive purposes, such as calculating a total for display purposes before an object is returned to the user.
- **Loading:** Life cycle method called by framework when an object is just about to be loaded from the persistent store. At this point the object exists in memory but has not had its state restored. You should never attempt to reference any non-scalar property within your `Loading` method. Unlike the other pairs of methods, there may be a considerable time gap between the calling of `Loading()` and `Loaded()` on an object - with the latter being called only when the object is first displayed or used programmatically. For most application purposes, `Loaded()` is a more useful event than `Loading()`.
- **OnPersistingError:** Life cycle method called by the framework if the object persistor throws an exception when an object is persisted. Typically this will be a

`DataUpdateException` or an `OptimisticConcurrencyException`. By adding the `OnPersistingError` method to your code you can intercept this exception and parse its message to establish details. When the `OnPersistingError` method exits, the framework will still throw an exception to be caught by the user interface - however, you may specify the message for that exception by returning the desired message as a string from your method. There is no point in adding this method to your code unless you want to change the message that is passed to the user - otherwise just leave the exception handling to the framework. Note that if your returned string message contains one or more line-breaks (`\n`) then any text after the first line-break will be moved into the Details section of the resulting dialog.

- `OnUpdatingError`: Life cycle method called by the framework if the object persistor throws an exception when an object is being updated. Works in a similar manner to `OnPersistingError`.
- `Persisted`: Life cycle method called by framework *after* a transient object has been persisted. **Important:** unlike `Persisting`, the `Persisted` method will be a separate transaction to the persisting of the object. This is useful because it means, for example, that if the key is database-generated then that generated value will be visible from within the `Persisted` method, but not in the `Persisting` method. Note, however, that the two transactions will still occur within an over-arching super-transaction, such that any exception occurring within the scope of the `Persisted` method will cause the whole action to fail i.e. the object will not itself be persisted.
- `Persisting`: Life cycle method called by framework when a transient object is just about to be persisted via the object store, as part of the same transaction. At this point the object exists only in memory and not in the persistent store.
- `Updated`: Life cycle method called by framework when a modified persistent object has just been saved to the persistent store. At this point the object in the persistent store will be in its new state.
- `Updating`: Life cycle method called by framework when a persistent object has just been modified and is about to be saved to the persistent store. At this point the object's data held in the persistent store will not yet have been modified.

Other recognised methods

- `IconName`: provides an alternative mechanism (to the `IconName` attribute) for specifying the icon to be used by an object. It should be used wherever you wish to specify a different icon for different instances of a class, or according to the status of the instance. See [How to specify the icon for an object](#).
- `ToString`: If no title attribute, or title method has been specified, then the framework will call the object's `ToString` method to get a title for the object.
- `Title`: If a single value property may be used to define a title for the object (and this approach is recommended where possible) then this is specified by means of a title attribute. If you wish to construct a title from several properties (value and or reference properties) and you do not wish to cache this combination as a property in its own right then you may specify the title in the form of a `Title` method that returns a string.

NakedObjects.Attributes

`NakedObjects.Attributes.dll` includes a set of attributes that are recognised by the framework.

AuthorizeAction

Specifies the users and/or roles to whom an action is to be made available. May be applied to an individual action, or at class level to apply to all actions in that class (including sub-classes).

AuthorizeProperty

Specifies the users and/or roles that may view, and, separately, Edit a property. May be applied to an individual property, or at class level to apply to all properties in that class (including sub-classes).

Bounded

For immutable objects where there is a bounded set of instances, the `Bounded` attribute can be used. For example:

```
[Bounded]
public class County {...}
```

The number of instances is expected to be small enough that all instances can be held in memory. The viewer will use this information to render all the instances of this class available to the user in a convenient form, such as a drop-down list.

ContributedAction

Applied to a specific reference parameter within an action on a service, indicating that that action should be [contributed](#) to the action menu on that type of domain object, or, if the parameter is an `IQueryable` of a domain type - as a query-result-contributed action.

DescribedAs

Subsequent to Naked Objects' introduction of the `DescribedAs` attribute, Microsoft has introduced its own `Description` attribute. As the latter is now recognised by Naked Objects, we recommend that you use it in place of `DescribedAs`.

The `DescribedAs` attribute is used to provide a short description of something that features on the user interface. How this description is used will depend upon the viewing mechanism - but it may be thought of as being like a tool tip. Descriptions may be provided for objects, members (properties, collections and actions), and for individual parameters within an action method. `DescribedAs` therefore works in a very similar manner to [Named](#).

To provide a description for an object, use the `DescribedAs` attribute immediately before the declaration of that object class. For example:

```
[DescribedAs("A Customer who may have originally become known to us via the marketing system or who may have contacted us directly.")]
public class ProspectiveSale {...}
```

Any member (property, collection or action) may also provide a description. To specify this description, use the `DescribedAs` attribute immediately before the declaration of that member. For example:

```
public class Customer
{
    [DescribedAs("The name that the customer has indicated that they wish to be addressed as (eg Johnny rather than Jonathan)")]
    public virtual string FirstName() {get; set;}
}
```

To provide a description for an individual action parameter, use the `DescribedAs` attribute in-line. For example:

```
public class Customer
{
    public Order PlaceOrder(
        Product product,
        [DescribedAs("The quantity of the product being ordered")]
        int quantity)
    {...}
}
```

Disabled

The `Disabled` attribute means that the member cannot be used in any instance of the class. When applied to the property it means that the user may not modify the value of that property (though it may still be modified programmatically). When applied to an action method, it means that the user cannot invoke that method. For example:

```
public class Customer
{
    [Disabled]
    public void AssessCreditWorthiness() {...}

    [Disabled]
    public virtual Money InitialCreditRating() {...}
}
```

Note that if an action is marked as `Disabled`, it will be shown on the user interface but cannot ever be invoked. One possible reason to do this is during prototyping, to indicate an action that is still to be developed. If a method is intended for programmatic use, but not intended ever to be invoked directly by a user, then it should be marked as `Hidden` instead. This attribute can also take a single parameter indicating when it is to be hidden, for example the following code would disable the action until the object has been saved.

```
public class Customer
{
    [Disabled(WhenTo.UntilPersisted)]
    public void AssessCreditWorthiness() {...}
}
```

The acceptable values for the parameter are: `WhenTo.Always`, `WhenTo.Never`, `WhenTo.OncePersisted` and `WhenTo.UntilPersisted`. By default the annotated property or action is always disabled.

Eagerly

Eagerly may be applied to any action or property, including a collection, within a class. The attribute may also be applied at class level - indicating that *all* properties and collections within that class should be eagerly rendered.

If the model is being run with the Restful Objects Server this specifies that the details and contents ('value') of the property are to be rendered in-line - containing the same JSON representation as if the client had followed the link(s) directly.

If the model is being run with Naked Objects MVC, this attribute causes the associated object or collection to be rendered in-line within the HTML. In the case of a reference property, the resulting HTML is the same as if the user had viewed the object and then expanded the referenced object in-line; in the case of a collection it is as if the user had viewed the object and then selected the table view on that collection. If applied to an action that returns a collection, then the result will be rendered as a table rather than a list. (This may be useful combined with the use of the `TableView` attribute to specify the columns to be included.)

The syntax for the **Eagerly** attribute is as follows.

```
[Eagerly(EaglerlyAttribute.Do.Render)]
public Customer Customer {get; set;}
```

In future releases we expect that the `Do` Enum will offer more options.

At present, **Eagerly** is a mechanism to specify the 'eager rendering' of information passed from the server to the client - it does not have any bearing on the loading of information from the database to the server ('eager loading'). Again, this is likely to change in a future release.

Executed

When running a Naked Objects MVC application using the Ajax functionality, `Validate` methods will normally be called on the server as the user is typing in a value. If the `Validate` method contains very complex logic, then this could result in a significant performance overhead. Server-side validation using Ajax may therefore be suppressed on a field-by-field basis, by adding the attribute `[Executed(Ajax.Disabled)]`.

FinderAction

Applied to an action on a service that returns a single domain object, or collection of domain objects, indicating that that action should be included within the [Find Menu](#) for objects of that type, wherever a Find Menu is rendered.

FindMenu

May be applied to an individual reference parameter on an action on a domain object, or to a reference property on an object, to indicate that you want a [Find Menu](#) to be rendered on the UI, even if there is also an auto-complete option.

Hidden

The `Hidden` attribute indicates that the member (property, collection or action) to which it is applied should not be visible to the user. It takes a single parameter indicating when it is to be hidden, for example the following code would show the `InternalId` property until the object has been saved, and then would hide it.

```
public class Customer
{
    [Hidden(WhenTo.OncePersisted)]
    public virtual int InternalId() { get; set; }
}
```

The acceptable values for the parameter are: `WhenTo.Always`, `WhenTo.Never`, `WhenTo.OncePersisted` and `WhenTo.UntilPersisted`.

IconName

The `IconName` attribute applies at class level, and provides the simplest mechanism for specifying the name of the icon to be used for that type. See [How to specify the icon for an object](#).

```
[IconName("person.gif")]
public class Customer {...}
```

Idempotent

Applies to an action. Used to indicate to a client that the action is 'idempotent' - invoking it more than once in succession (with the same arguments) has the same effect as invoking it once. Note that this is purely an indicator: adding the attribute does not force a method to behave in this fashion. Application developers should therefore take considerable care to ensure that this attribute is applied correctly. (See also the [QueryOnly](#) attribute).

Immutable

The `Immutable` attribute may be applied to a class, to indicate that the user may not ever edit any properties on an object. For example:

```
[Immutable]
public class Country {...}
```

This attribute can also take a single parameter indicating when it is to become immutable, for example the following code would allow the user to create an email object, specifying its properties before saving, and then prevent any changes once it has been saved.

```
[Immutable(WhenTo.OncePersisted)]
public class Country {...}
```

The acceptable values for the parameter are: `WhenTo.Always`, `WhenTo.Never`, `WhenTo.OncePersisted` and `WhenTo.UntilPersisted`. By default the annotated property or action is always immutable.

Mask

The `Mask` attribute allows you to apply standard .NET masks to format the presentation of `DateTime` or numeric value properties. For example, `[Mask("d")]` will present only the date part of a `DateTime` value, and `[Mask("c")]` applied to a `Decimal` property will render it as a currency value.

A very handy cheat sheet on using .NET masks may be downloaded from: <http://www.cheat-sheets.org/saved-copy/msnet-formatting-strings.pdf>.

Note that the mask applies only to the presentation of the value - it does not have any impact on the parsing of the input. For Naked Objects MVC, the input parsing is determined by the language settings on the user's browser. (Though it is always better to avoid ambiguity by using the JQuery date picker for date entry if possible.)

If you are writing an application that may be used by users with different language/locale settings, then you should avoid using very specific masks that could lead to an existing value being mis-interpreted when an edited object is saved. Thus, while `[Mask("d")]` is safe, `[Mask("dd-MM-yy")]` is not safe - because a user with US settings could interpret certain existing dates as "MM-dd-yy".

MemberOrder

`MemberOrder` is the recommended mechanism for specifying the order in which fields and/or actions are presented to the user. `MemberOrder` is specified at the individual member level, on a relative basis. The syntax is:

```
[MemberOrder(<relative position>)]
```

where `relative position` may be a number or a `String`. The actual sequence is determined by comparing all the values of the sequence specifier `String`, using the standard `String` comparator.

The simplest convention is to use numbers - 1, 2, 3 - though it is a better idea to leave gaps in the numbers - 10, 20, 30 perhaps - such that a new member may be added without having

to edit existing numbers. A useful alternative is to adopt the dot-decimal notation - 1, 1.1, 1.2, 2, 3, 5.1.1, 5.2.2, 5.2, 5.3 - which allows for an indefinite amount of future insertion. For example:

```
public class Customer {  
    [MemberOrder(2.1)]  
    public virtual string Address {get; set;}  
  
    [MemberOrder(1.1)]  
    public virtual string FirstName {get; set;}  
  
    [MemberOrder(1.2)]  
    public virtual string LastName {get; set;}  
  
    [MemberOrder(3)]  
    public virtual Date DateOfBirth {get; set;}  
}
```

If a member does not have a specified order then it will be placed after those that are specified. (Two members may have the same relative position specified, but in such a case the relative ordering of those members will be indeterminate.)

This approach is especially useful when dealing with inheritance hierarchies, as it allows sub-classes to specify where their additional members should be placed in relation to those inherited from the super-class.

Note that certain styles of user interface will lay out an object's properties and its collections separately, in which case the relative member order of properties and collections will be evaluated separately. However, since other styles of user interface may interleave properties and collections, it is safer to assume the latter.

MultiLine

The `MultiLine` attribute indicates to the user interface that a string property should be rendered over multiple lines.

```
public class BugReport {  
    [MultiLine(NumberOfLines = 10, Width = 40)]  
    public virtual string StepsToReproduce() { get; set; }  
}
```

The `NumberOfLines` specifies the number of lines to be displayed - if the text contains more lines (subject to any restriction of maximum length imposed by a separate `StringLength` attribute) then the field will be rendered with vertical scrolling.

In NakedObjects MVC, the `Width` parameter only has an effect if the width of the `textarea` is not overridden by the `.CSS`.

NakedObjectsIgnore

This attribute may be applied to a property or method that is used for programmatic purposes only – it is never intended to be visible to a user. The difference between marking a public

member with `NakedObjectsIgnore`, rather than `Hidden`, is that in the former case the Reflector will not build up any metadata ('Specifications') about the member, and nor will it introspect any types used within that member's signature.

NakedObjectsInclude

May be considered as having the opposite effect from `NakedObjectsIgnore`.

`NakedObjectsInclude` is used in individual members when the type has been marked up with `NakedObjectType(ReflectOver.ExplicitlyIncludedMembersOnly)`.

NakedObjectType

This attribute is applied at class level only. It specifies whether a class, and/or its members, should be reflected over. The attribute is specified with a value from `ReflectOver` as follows:

- `ReflectOver.All` – reflect over the Type and all members (except where marked `[NakedObjectsIgnore]`)
- `ReflectOver.TypeOnlyNoMembers` - Typically used on a system service, where the type must be in the meta-model, but there are no actions/properties for user display.
- `ReflectOver.ExplicitlyIncludedMembersOnly` - Allows an 'additive' style of coding, where only those members marked `[NakedObjectsInclude]` are reflected over
- `ReflectOver.None` - The type and all members to be ignored by Naked Objects (and hence excluded from the meta-model)

Named

Subsequent to Naked Objects' introduction of the `Named` attribute, Microsoft has introduced its own `DisplayName` attribute. As the latter is now recognised by Naked Objects, we recommend that you use it in place of `Named`.

The `Named` attribute is used when you do not want to use the name generated automatically by the system. It can be applied to objects, members (properties, collections, and actions) and to parameters within an action method.

NotPersisted

This attribute indicates that a class is not to be persisted. The user may still edit such objects and the Save button will confirm those edits - but the object will never be persisted.

This attribute will work with Entity Framework in Model First or Database First mode, but not Code First. The Microsoft Code First team has said that they intend to introduce such an attribute called, perhaps, `StoreIgnore` in the near future. When that appears, we will obsolete our own `NotPersisted` attribute in favour of the Microsoft one.

This is typically used on classes that are defined solely for the purpose of inputting data or presenting data. For example:

```
[NotPersisted]
public class CustomerSummary {...}
```

The `NotPersisted` attribute may also be applied at the level of an individual property, as shown below. (If the property is read-only (has only a `get` method in C#) then it will not be persisted anyway. The `NotPersisted` attribute is necessary only if the property needs to have a `set` method for other purposes.)

```
public class Order {
    [NotPersisted]
    Public virtual Order PreviousOrder() { get; set; }
}
```

Note that if a persisted class has a property of a type that is marked `NotPersisted` then the property itself must also be marked `NotPersisted` - otherwise an error will arise.

Optionally

May be applied to a property member on an object, or to a parameter of an action method, to indicate that the value is optional rather than mandatory.

PageSize

This attribute overrides the default page size (of 20 objects) for a specific method that returns a collection of objects. The page size may be set to any integer value. If the value is set to zero, no paging will occur and the full set of objects returned.

Plural

Where the framework displays a collection of objects it may use the plural form of the object type in view. By default the plural name will be created by adding an 's' to the end of the singular name (whether that is the class name or another name specified using the `DisplayName` attribute). The framework will also handle words ending in 'y', changing `Country` to `Countries`, for example. Where these conventions do not work, the programmer may specify the plural form of the name using the `Plural` attribute. For example:

```
[Plural("Children")]
public class Child {...}
```

PresentationHint

`PresentationHint` is a simple mechanism for providing hints from the domain code to the presentation layer for customising the user interface. The attribute may be applied to a class, a property (or collection), an action or a parameter. The attribute takes a single string (though this may consist of multiple hints separated by spaces).

```
[PresentationHint("red-background collections-on-right no-tab")]
public class Foo {

    [PresentationHint("rich-text extra-wide")]
    public string Text {get; set;}

    [PresentationHint("button")]
    public void DoSomething() {}
}
```

In Naked Objects MVC these hints are rendered as class(es) in the `<div>` (or other Html element) that most closely corresponds to the location of the hint. For example, the Html for the Text field above will be rendered thus:

```
<div class="nof-property rich-text extra-wide" id="Foo-Text">...</div>
```

This information may be used by adding a custom style sheet and/or custom JavaScript. (The generic Naked Objects MVC user interface does not recognise any specific hints.)

In Restful Objects, the hints are added as a custom extension to the relevant representation - as specifically allowed for in the Restful Objects specification. For example:

```
x-ro-nof-presentationHint: "rich-text extra-wide"
```

ProgramPersistableOnly

This attribute, when applied at class level, indicates that transient instances of a class may only be persisted programmatically - not directly by the user. This is typically used where you wish to create one transient object within another transient object (sometimes referred to as the 'aggregation' pattern) and you do not wish the user to be able to persist the child object separately from the parent object. By marking the child object with the `ProgramPersistableOnly` attribute the user cannot persist it directly (hitting the Save button while that object is transient confirms the edits but does not persist the object). However assuming that the Parent object holds a reference to that Child object, then the child will be persisted automatically when the parent is persisted.

QueryOnly

Applies to an action. Used to indicate to a client that the action is 'side-effect free' (does not change the persistent state of the system). Note that this is purely an indicator: adding the attribute does not prevent a method from changing the state. Application developers should therefore take considerable care to ensure that this attribute is applied correctly. (See also the Idempotent attribute).

Regex

Subsequent to Naked Objects' introduction of the `Regex` attribute, Microsoft has introduced its own `RegularExpression` attribute. As the latter is now recognised by Naked Objects, we recommend that you use it in place of `Regex`.

The `Regex` attribute may be applied to any property, or to any parameter within an action method, that allows the user to type in text as input. The syntax is:

```
[Regex(Validation = <regex string>, Message = <string>, CaseSensitive = <True|False>)]
```

The first parameter is required; the message property is optional and defines a message to return to the user if the text entered does not match the `Regex` string; `caseSensitive` is optional, defaulting to false. The following example shows the `Regex` attribute applied to the `Email` property, to ensure that any entry adheres to the correct form for an email address:

```
public class Contact {  
    [Regex(Validation = @"^\[-\w\.]+\@[-\w\.]+\.[A-Za-z]+$", Message = "Not a valid  
    email address")]  
    public virtual string Email { get; set; }  
}
```

When applying the `Regex` attribute to a value parameter within an action method, the attribute should precede that parameter:

```
public class Organisation{  
    public void NewContact(  
        string contactName,  
        [Regex(Validation = @"^\[-\w\.]+\@[-\w\.]+\.[A-Za-z]+$",  
                Message = "Not a valid email address")]  
        string email)    {...}  
    }
```

In the above examples, note the use of the `^` and `$` symbols to anchor the text to the start and end of line respectively. This is to ensure that the whole of the input string matches the `Regex`. If these symbols are omitted, then the framework will look for *any* match to the `Regex` string within the input text.

Root

The `Root` attribute is intended for use in conjunction with the `ComplexType` attribute, where an in line object needs to access its parent object programmatically. The in line object should be provided with a property of the same type as the parent object (or any type implemented by the parent) and this property should be marked up with `Root`. The framework will then inject the parent into that property, in the same way that it might inject a domain object container or other service. The following example builds on an [earlier example](#).

```
[ComplexType]
public class Name {
    [Root]
    public virtual Person Person { get; set; }

    public virtual string FirstName { get; set; }

    public virtual string LastName { get; set; }
}
```

TableView

The `TableView` attribute allows you to specify the columns that will appear in a table view of a collection; it may be applied to a collection property or to an action that returns a collection, as shown in the example below:

```
[TableView(true, "Product", "OrderQty", "UnitPrice")]
public virtual ICollection<SalesOrderDetail> Details {...}

[TableView(false, "Description", "DueDate", "onfidential")]
public IQueryable<Task> MyTasks() {...}
```

The first, boolean, parameter of the attribute determined whether or not the title of each object will be rendered as the first column of the table - since this title also acts as a link to a standalone view of that object then the parameter will typically be set to `true` if you want the user to be able to navigate to an object in the collection; it is typically set to `false` where the table view shows all the information the user needs and there is nothing to be gained from navigating the individual object. This parameter is followed by the list of columns that you want rendered, each as a separate string parameter.

Note the following:

- The columns should take the same format as the property name in the code, not as displayed to the user: "OrderQty" not "Order Qty" - the column headers will automatically pick up the correct display format (or the overridden display name if one has been specified).
- `[TableView(true)]` will result in a single-column table with just the title of the object, equivalent to a 'list view'.
- If an object or collection is displayed using a custom view that specifies the columns to be displayed then the latter will take precedence over the `TableView` annotation (if one exists).

Title

When applied to a property, the `Title` attribute signifies that property should be used as the title for the object. (This will override a title method if one existed). Normally, this should be applied to a property containing a value type such as `String`, or `Date`. However, it may be applied to a reference property, indicating that the title of the object in that reference property should be used as the title of the object that owns the property also. However, this is not a recommended practice as it would force the loading of the reference property earlier than would otherwise be the case - with possible performance implications.

TypicalLength

The `TypicalLength` attribute indicates the typical length of a `String` property or parameter in an action. This may be used by the viewing mechanism to determine the space that should be given to that property or parameter in the appropriate view. For example, in a table view, this will determine the default width for that column.

```
public class Customer {  
    [StringLength(30)] [TypicalLength(15)]  
    Public virtual string FirstName() { get; set; }  
}
```

If the typical length is the same as the `StringLength` then there is no need to specify `TypicalLength` as well. If the value specified is zero or negative then it will be ignored.

In NakedObjects MVC, `TypicalLength` only has an effect if the width of the `input` is not overridden by the `.CSS`.

ValidateProgrammaticUpdates

If the `ValidateProgrammaticUpdates` attribute is applied to a class, then all forms of property validation (including `Validate...` methods, `Range`, `RegularExpression` and `StringLength` attributes, and the enforcement of any mandatory/required properties) will be enforced by the framework even when an object is updated (or persisted for the first time) programmatically. Without this attribute, these forms of validation are enforced only when changes are made via the user interface.

If any of the validation rules are violated then a `DomainException` will be thrown.

NakedObjects.Types

The `NakedObjects.Types` assembly provides a number of interfaces and classes that are explicitly recognised by the Naked Objects framework, and that intended to be used within a domain model.

NakedObjects

- `DomainException`. All exceptions that are generated within the application domain code should inherit from this class or throw it directly as this allows the Naked Objects Framework to discriminate between potential framework errors and exceptions raised in application code.
- `IDomainObjectContainer`. See [IDomainObjectContainer methods](#).
- `IViewModel` and `IViewModelEdit`. See [View Model](#).

NakedObjects.Async

- `IAsyncService`. Interface representation of the `AsyncService` class (in `NakedObjects.Core.dll`) - for injection into domain code.

NakedObjects.Audit

- **IAuditor**. Allows domain programmers to define an auditing service. See [Configuring Auditing](#).
- **INamespaceAuditor**. Specific sub-type of **IAuditor**, where the methods will only be called in relation to types that fall within the specified namespace.

NakedObjects.Redirect

- **IRedirectedObject**. (Intended for use with Restful Objects) Implemented by a 'stub' class that acts as a proxy for an object managed on another server.
- **IRedirectedService**. (Intended for use with Restful Objects). Implemented by a 'stub' class that acts as proxy to a service implemented on another server. Note that, unlike **IRedirectedObject**, this defines functions, not properties.

NakedObjects.Security

- **INamespaceAuthorizer**. An implementation of this interface provides authorization for a single fully-qualified type, or for any types within a namespace. See [Custom Authorization](#).
- **ITypeAuthorizer**. Implement this interface to manage authorization for a specific class of domain objects rather than a whole namespace.

NakedObjects.Snapshot

- **IXmlSnapshot**. Interface definition of the class **IXMLSnapshotService** that will be generated by the **XMLSnapshotService** (see below).
- **IXmlSnapshotService**. Interface definition for the class **XmlSnapshotService** (defined in **NakedObjects.Snapshot.Xml.dll**) allowing this service to be injected into domain code without requiring a reference to the framework.

NakedObjects.Value

- **FileAttachment**. See [How to handle File Attachments](#).
- **Image** provides similar functionality to **FileAttachment**, but displays the contents as an in-line image rather than as a link. See [How to display an image](#).
- **IStreamResource**. Interface implemented by both **FileAttachment** and **Image**. (Not intended to be used directly within domain code.)

NakedObjects.Helpers

The **NakedObjects.Helpers.dll** contains a number of helper classes, extension methods, and other artifacts that you may find useful in enriching your domain model. However, the use of any of these capabilities is optional - it is not necessary for a domain model project to have a reference to this assembly.

NakedObjects

- [DateTimeExtensions](#). Useful methods that will automatically be added to instances of `DateTime` for comparing dates, deliberately ignoring the time element, for example: `IsBeforeToday()`
- `ViewModel`. A ready-made implementation of `IViewModel`, for view models based on a single Root type. See [View Model](#).
- `IHasGuid`. This interface is intended for use with 'Polymorphic Associations' and specifically to work in conjunction with `NakedObjects.Services.ObjectFinder`. If the class being associated implements `IHasGuid`, then the compound key will use this Guid (together with the fully qualified type name) to form the compound key. This has the advantage that the Guid may be set up when the object is created rather than waiting until the object is persisted (if the keys are database generated, that is). This is important when defining interface associations between transient objects that are all persisted in one transaction.
- `IHasIntegerId`. Merely defines that object has a single integer key called `Id`. Used by `PolymorphicNavigator`, for example.
- `IPolymorphicLink`. See [How to handle associations that are defined by an interface rather than a class](#).
- `PolymorphicLink`. A ready-made implementation of `IPolymorphicLink`.
- `ReasonBuilder`. Helper object for constructing a string reason to be rendered to the user within, say, a `Validate` or `Disable` method. Separates multiple appended reasons with semi-colons.
- `TitleBuilder` and `NewTitleBuilder`. Helper objects for constructing a title string on an object. (The latter provides a slightly different syntax that offers a finer level of control).

NakedObjectsServices

- [AbstractFactoryAndRepository](#). Convenience super class for factories and repositories that wish to interact with the container.
- [IKeyCodeMapper](#). Defines a service that can convert between a key and a string code. Possible uses include: key encryption, custom key separators.
- [IObjectFinder](#). Defines a mechanism for retrieving a domain object given a 'compound key' (a single string that defines both the type and the identity of that object). Also provides a method for determining the compound key for a given object. The intent of making both methods templated, is that it allows for the possibility of different types (or perhaps different namespaces of types) having different ways of putting together the compound key.
- [ITypeCodeMapper](#). Defines a service that can convert between a Type and a string code where you don't wish to use the fully-qualified type name as the string representation. Possible uses include: to create compound keys for defining polymorphic associations; To create Oids for use in URLs.
- [ObjectFinder](#). An implementation of `IObjectFinder`. Works with multiple keys, of type Integer, String, Short, or Char
- [ObjectFinderWithTypeCodeMapper](#). An implementation of `IObjectFinder` that will delegate the string representation of a type to an injected `ITypeCodeMapper` service, if one exists. (Otherwise it will default to using the fully-qualified class name).

- `PolymorphicNavigator`. See [How to handle associations that are defined by an interface rather than a class](#).
- `SimpleRepository`. This is a simple, typed, implementation of a repository that can be useful when writing tests.

NakedObjects.Utills

- `AttributeUtills`. Utility methods for getting custom attributes declared on domain types, members or method parameters.
- `KeyUtills`. Utility methods for obtaining and making use of domain object keys, whether explicitly defined, or inferred by convention.
- `NameUtills`. Utility methods for manipulating names of domain model elements - for use in presentation, for example.
- `TypeUtills`. Utility methods for safely obtaining and using types defined within a domain model.

Adding behaviour to your domain objects - a how-to guide

This section is concerned with adding to, or modifying, the behaviour of those domain objects, in order to build a richer application. It is presented in the form of a 'How-to Guide': we have tried to cover all the common questions that programmers raise about adding behaviour.

Using the Naked Objects IDE

The Naked Objects 'IDE' consists of a set of ItemTemplates and Code Snippets designed to boost productivity and coding standards when creating domain models for running with Naked Objects. They are installed via the NakedObjects.Ide NuGet package, which is automatically installed with the NakedObjects.ProgrammingModel package.

Item Templates

Item templates are installed here: `..\Visual Studio [VS version]\Templates\ItemTemplates\C#\Naked Objects`

If the templates are not appearing for you - check that your Visual Studio directory does already have the `\Templates\ItemTemplates\` sub-directories. Sometimes problems arise when people have been using a Beta version of Visual Studio and then upgraded to the released version - with the result that they do not have the correct directory structure.

The templates will be offered, within a Naked Objects folder, whenever you select `Add > New Item > Naked Objects`. Here is a list of the current templates:

- **DomainObject** Creates a new domain model class with a ready-made `Id` property, and standardised code regions
- **DbContext** Creates a new class inheriting from `DbContext`
- **DbMapping**. Creates a new class for specifying database mappings in Code First.
- **DbInitialiser**. Creates a new class for initialising the database upon model changes in Code First.
- **Fixture** Creates an [object fixture](#) class for use within XATs.
- **Repository** Creates a new repository (and/or factory) class inheriting from `AbstractFactoryAndRepository`.
- **XAT** creates a new class to contain XATs.
- **Default_T4Template** Adds a T4 template that can translate a `.edmx` file into domain classes suitable for use with Naked Objects. Intended for use when creating a model from an existing database.
- **CodeFirstReverseEngineerTemplate**. See [Using the Entity Framework Power Tools to reverse engineer a database into code](#).

Code Snippets

The Code Snippets will have been installed here: `..\Visual Studio [VS version]\Code Snippets\[Language]\My Code Snippets\NakedObjects`

If the snippets are not appearing for you - check that the your Visual Studio directory does already have the `\Templates\ItemTemplates\` sub-directories. Sometimes problems arise when people have been using a Beta version of Visual Studio and then upgraded to the released version - with the result that they do not have the correct directory structure.

Snippets for creating Actions

The following code snippets are relevant to defining an action, either within a Domain Object or a Service:

Table 1. Summary of code Snippets relevant to actions

Name	Shortcut	Description
Action	<code>act</code>	Adds a method intended to be a user action.
Action - default parameter	<code>actdef</code>	Adds a method to specify the default value for a parameter on a corresponding action method.
Action - choices	<code>actcho</code>	Adds a method to specify the choices (drop-down lists) for a parameter of a corresponding action method.
Disable	<code>dis</code>	Adds a method to disable a corresponding property or action dynamically.
Hide	<code>hide</code>	Adds a method to hide a corresponding property or action dynamically.
Validate	<code>val</code>	Adds a method to validate the value(s) being entered into a corresponding property or as parameters for a corresponding action.

Snippets for creating Properties or Collections

The following code snippets are relevant to defining a property within a Domain Object:

Table 2. Summary of code Snippets relevant to properties

Name	Shortcut	Description
Property - Virtual	<code>propv</code>	Adds a <code>virtual</code> property (C#).
Property - choices	<code>propcho</code>	Adds a method to specify the choices (drop-down list) for a corresponding property.
Property - default	<code>propdef</code>	Adds a method to specify the default value for a corresponding property.

Name	Shortcut	Description
Disable	dis	Adds a method to disable a corresponding property or action dynamically.
Hide	hide	Adds a method to hide a corresponding property or action dynamically.
Modify	mod	Adds a method to intercept a user modification to a corresponding property.
Validate	val	Adds a method to validate the value(s) being entered into a corresponding property or as parameters for a corresponding action.
Collection	coll	Adds a collection, with actions for adding and removing objects.
Polymorphic Property	polyprop	Adds a property that is defined by an interface with many potential implementations (implementation delegates to an injected PolymorphicNavigator)
Polymorphic Collection	polycoll	Adds a collection that is defined by an interface with many potential implementations (implementation delegates to an injected PolymorphicNavigator)

Snippets for creating or retrieving objects

The following code snippets for retrieving objects are intended for use either within a domain object or a service.

Table 3. Summary of code Snippets relevant to retrieving objects

Name	Shortcut	Description
Factory Method	fact	Adds a factory method that returns a new instance of a specified type.
New Transient Instance	newt	Code to create a new transient instance of a specified class.
Find Query	find	Adds a query method to find a single matching object.
List Query	list	Adds a query method to return a list of matching objects.

Other Snippets

Table 4. Summary of other code Snippets

Name	Shortcut	Description
Injected Service	injs	Adds a property containing a reference to a Service, to be injected automatically when the object is instantiated.
Injected	injc	Adds a property containing a reference to an

Name	Shortcut	Description
Container		IDomainObjectContainer, to be injected automatically when the object is instantiated.
Title	title	Adds a Title() method to function as the domain object's title.
Icon Name	icon	Adds an IconName() method to specify the icon manually

The object life-cycle

Domain objects exist in one of two states: transient or persistent. A transient object exists only in memory: it is not known-to or managed-by the object Persistor. A persistent object is known-to and managed by the object store. A common scenario is that a new object is created in a transient state and returned to the user interface - where it appears in Edit mode, ready for the user to complete any mandatory fields before hitting the Save button, which changes the state of the object to persistent.

Important: Once an object has been made persistent, it remains in a persistent state - it never goes back to being transient. If the user edits an already persistent object then hitting the Save button will then cause the persistent object to be *updated* with the changes made. It is not correct to say that the object is being persisted again.

A second scenario is where an object is created in a transient state, has its mandatory fields set up and is then persisted within the application code before being made available to the user.

A third, less common, scenario is a domain object that only ever exists in a transient state and is never persisted.

How to create an object

When you create any domain object within your application code, the Naked Objects framework must be made aware of the existence of this new object - in order that it may subsequently be persisted, and/or in order that any services that the new object needs are injected into it. Just specifying `New Customer()`, for example, will create a `Customer` object, but that object will *not* be known to the framework.

The correct way to create an object within your application code is to invoke the `NewTransientInstance` method on the Domain Object Container. For example:

```
Customer newCust = Container.NewTransientInstance<Customer>;
```

The new object will have been created in a transient state. It may be returned to the user to be completed and persisted, or persisted explicitly within your code.

Warning: It is possible to create a transient object within another transient object, but you need to be careful. When the framework persists any transient object, it will automatically persist any other transient object referenced by that object - so you will need to ensure that they all exist in a valid state before persisting any of them. Moreover, if any of these transient objects are to be exposed to the user (while in their transient state), then you need to anticipate the fact that the user could elect to save any of the transient objects at any point - which could cause the graph of related objects to be persisted in an invalid state.

The recommended approach is, if possible, not to permit the user to create a new transient object that is a child of an existing transient object, but, rather, to require the user to save the parent object first. This can be done by marking-up actions that create child objects with `Disabled(WhenTo.UntilPersisted)`.

If wish to trigger some additional behaviour when the object is created by code outside of the object see [How to insert behaviour into the object life cycle](#).

How to persist an object

There is no need to write any special code for persisting an object. When a new transient is returned to the user then the user is provided automatically with a Save button which will change the object to a persistent state (assuming all mandatory fields have been completed).

If you wish to persist an object within your code invoke the `NewTransientInstance` method on the Domain Object Container. For example:

```
Customer newCust = Container.NewTransientInstance<Customer>;
newCust.Name = "Charlie";
Container.Persist<Customer>(ref newCust);
```

If you wish to trigger some additional behaviour when the object is persisted (by the user or by code outside of the object) see [How to insert behaviour into the object life cycle](#).

How to update an object

There is no need to write any special code for updating an object. If the user edits a persisted object then the changes will be made automatically when they save their changes. And if the programmer changes a property on an object then the changes will be notified to the object Persistor automatically.

If you wish to trigger some additional behaviour when the object is updated (by the user or by code outside of the object) see [How to insert behaviour into the object life cycle](#).

How to delete an object

As with creating an object, if you wish to delete an object that is already persistent then you must notify the framework via the `DisposeInstance` method on the Domain Object Container:

`Container.DisposeInstance(persistentObject)`

Note: When working with the Entity Object Store, this method will delegate much of the work to the Entity Framework's delete functionality. As well as deleting the appropriate row(s) from the database table(s), Entity Framework will attempt to delete any persisted references to the object held in other objects - in other words will attempt to delete Foreign Keys to the deleted rows. If any of those Foreign Key columns are non-nullable in the database, then an error will be thrown and the whole transaction rolled back. In such circumstances, the programmer must take responsibility to delete the associated objects first - though this can be as part of the same transaction. For example, if you want to delete a persisted Customer object, but the Order object has a non-nullable reference to a Customer, then the method for deleting the Customer should first delete any Orders associated with that Customer.

How to retrieve existing instances

If you need to retrieve instances from within a method on a domain object then you have two options:

- If a suitable method exists on a Repository then just inject that Repository into the domain object and call the method.
- Call the `Instances` method on an injected Domain Object Container. The method returns an `IQueryable` of the required object type, on which you may then invoke LINQ queries.

How to insert behaviour into the object life cycle

See [LifeCycle methods](#).

The following are some examples of using these those life-cycle methods:

- Using the `Created` method to set the object into an initial state.
- Using the `Loaded` method to calculate the total, if you don't wish to have that total persisted explicitly.
- Using the `Updating` method to change a version property (e.g. `LastUpdated`) on an object just before it is persisted. Note, however, that you should update the private variable (e.g. `myLastUpdated`) not the public property - as the latter would initiate a new call to `Updating` and result eventually in a Stack Overflow Error!

How to specify that an object should never be persisted

Use the `NotPersisted` attribute. (See also the [ProgramPersistableOnly](#) attribute).

How to specify that an object should not be modified by the user

Use the `Immutable` attribute.

How to specify that a class of objects has a limited number of instances

Use the `Bounded` attribute. A common way of describing this is that the whole (limited) set of instances may be rendered to the user as a drop down list - but the actual interpretation will depend upon the form of the user interface.

How to implement concurrency checking

See [How to handle concurrency checking](#).

Object presentation

How to specify a title for an object

A title is used to identify an object to the user in the user interface. For example, a Customer's title might be the organization's customer reference, or their name. The simplest way to specify a title for a domain object is to add a `title` attribute to one of the value properties (usually a `string` - but it may be any type of property). For example:

```
public class Employee {  
    [Title()]  
    public virtual string Name() { get; set;}  
    ...  
}
```

If you wish to construct the title from more than one property then you may provide a `Title` method, that returns a `String`.

A recommended practice is to use a `NakedObjects.ITitleBuilder` object to build the title. You may obtain this by calling `Container.NewTitleBuilder()`. `ITitleBuilder` works much like `StringBuilder`, but provides a number of useful methods to help in the construction of titles.

The `Append` method will add a property to a title with a space in between; `Concat` adds the property without a space. There are overloaded versions of each method, that provide various formatting options, including the use of joiners (such as punctuation) and formatting strings for use when adding dates and other values to which standard formatting strings may be applied. If the property being added has a null value, then that `Append` or `Concat` statement will be ignored. For example, the following code would produce a title of the form:

Joe Bloggs, 07-Jan-63

```
public string Title()
{
    var t = Container.NewTitleBuilder();
    t.Append(FirstName).Append(LastName).Append(", " DateOfBirth, "dd-MMM-yy");
    return t.ToString();
}
```

Use the Title snippet (shortcut `title`) for creating this method.

If there is no title attribute on an object, and no title method, then the framework will use the object's `ToString()` method as the title. This means that in the above example, you could choose to rename the above method from `Title` to `ToString` (overriding the inheriting `ToString()`) and it would work the same way.

The recommended best practice is to construct titles from value fields within the objects, such as strings and dates. It is OK to include reference properties within a title, indeed the `ITitleBuilder.Append` method is designed to cope with this. However, you should be aware that if you include one or more reference properties within your title, then this will force those properties to be resolved when the object is loaded - instead of being resolved lazily as they are needed. This all happens transparently and is not usually a problem. However, it could slow down the performance of your application.

How to specify the icon for an object

Naked Objects allows you to associate an icon with an object, either based on the object's type (using the the `IconName` attribute) or specific to the instance (using an `IconName()` method) - explained below. Naked Objects can accept most common image types as an icon, including `.gif`, `.png`, and `.ico`.

For Naked Objects MVC, the icons should be included in the `Images` folder of the Run project - as with any ASP.NET MVC project. If no icon name is specified for an object, then Naked Objects MVC will specify the icon in the generated HTML as `default.png`.

The the `IconName` attribute is applied at class level as shown below:

```
[IconName("person.png")]  
public class Customer {...}
```

The `IconName` method allows you to specify an individual icon for each instance of a class, if you need to. For example, an instance of `Product` could use a photograph of the product as an icon, using:

```
public string IconName() {  
    return ProductName() + "-photograph.png";  
}
```

Use the `IconName` snippet (shortcut `icon`) for creating this method.

It is also possible to vary the icon according to the status of the object:

```
public string IconName() {  
    return "Order-" + Status() + ".png";  
}
```

As shown in these examples, we recommend that you include the file extension with the icon name. If no extension is specified, the system will look for a `.png` file with that name.

How to specify a name and/or description for an object

By default, the name (or type) of an object, as displayed to the user will be the class name. However, if a `DisplayName` attribute is included, then this will override the default name. This might be used to include punctuation or other characters that may not be used within a class name.

By default the framework will create a plural version of the object name by adding an 's' to singular name. For irregular nouns or other special case, the `Plural` attribute may also be used to specify the plural form of the name explicitly.

(Note that there is an entirely separate mechanism for dealing with Internationalisation, which is described elsewhere).

The programmer may optionally also provide a `Description` attribute, containing a brief description of the object's purpose, from a user perspective. The framework will make this available to the user in a form appropriate to the user interface style - for example as 'balloon' help.

How to specify that an object should be always hidden from the user

Use the `Hidden` attribute.

Properties

The following conventions are concerned with specifying the properties of an object, and the way in which users can interact with those properties.

How to add a property to a domain object

Properties can be 'auto properties' but they must be marked `virtual`. The simplest way to add a property is using the `propv` code snippet (standing for 'property - virtual').

How to prevent the user from modifying a property

Preventing the user from modifying a property value is known as 'disabling' the property.

To disable a property always, use the `Disabled` attribute.

To disable a property under certain conditions, use a `Disable` method. The syntax is:

```
public string Disable<PropertyName>()
```

A non-null return value indicates the reason why the property cannot be modified. The framework is responsible for providing this feedback to the user. For example:

```
public class OrderLine {
    public virtual int Quantity { get; set;}

    public string DisableQuantity() {
        if (HasBeenSubmitted()) {
            return "Cannot alter any quantity after Order has been submitted";
        }
        return null;
    }
}
```

The `NakedObjects.ReasonBuilder` class may be used to construct the message. The `Reason` property on `ReasonBuilder` will return the message as a string; if no message has been added, it will return a null value, as shown below:

```

public class OrderLine {
    public virtual int Quantity { get; set;}

    public string DisableQuantity() {
        var rb = new ReasonBuilder();
        rb.AppendOnCondition(HasBeenSubmitted(), "Cannot alter any quantity after Order
has been submitted");
        return rb.Reason;
    }
}

```

Use the `Disable` snippet (shortcut `dis`) for creating this method.

To apply the same rules to all the properties on an object - for example to disable all the properties once the object has been persisted - create a method `DisablePropertyDefault` that returns a string, just as for an individual disable-property method. For example:

```

public string DisablePropertyDefault() {
    if (Container.IsPersistent(this)) {
        return "Cannot edit property once the object is saved";
    }
    return null;
}

```

If you wish to apply a rule to most, but not all of the properties then you may add the `DisablePropertyDefault` method, and then provide individual `Disable` methods for properties where you wish to override this default behaviour.

Having a `DisablePropertyDefault` method on a class and then using the `Disable` attribute on individual properties is not recommended - as the behaviour is not guaranteed.

How to make a property optional (when saving an object)

Use the `[Optionally]` attribute.

How to specify the size of String properties

Use the `StringLength`, `TypicalLength` and `MultiLine` attributes.

How to validate user input to a property

To validate that an input falls within a specific range, using the `RangeAttribute`.

To validate that an input value conforms to a particular format, use the `Mask` or `RegularExpression` attributes.

For more complex forms of validation, use a `Validate` method, for which the syntax is:

```
public string Validate<PropertyName>(object value)
```

If the proffered value is deemed to be invalid then the property will not be changed. A non-null return `String` indicates the reason why the member cannot be modified/action be invoked. The framework is responsible for providing this feedback to the user. For example:

```
public class Exam {  
    public virtual int Mark { get; set; }  
  
    public string ValidateMark(int mark) {  
        if (! (mark >= 0 & mark <= 30)) {  
            return "Mark must be in range 0 to 30";  
        }  
        return null;  
    }  
}
```

```
}
```

This example is intended to illustrate the syntax of a `Validate` method. If your validation logic is actually as simple as defining a range for a numeric value, then you can just use the `Range` attribute instead of a `Validate` method.

The `NakedObjects.ReasonBuilder` class may be used to construct the message. The `Reason` property on `ReasonBuilder` will return the message as a string; if no message has been added, it will return a null value, as shown below:

```
public string ValidateMark(int mark){  
    ReasonBuilder rb;  
    rb.AppendOnCondition(! (mark >= 0 & mark <= 30), "Mark must be in range 0 to 30");  
    return rb.Reason;  
}
```

Use the `Validate` snippet (shortcut `val`) for creating methods like this.

How to validate user input to more than one property

Sometimes you need to be able to validate more than one property together. For this purpose you can use a `Validate` method that takes multiple parameters. This may be used in conjunction with individual validate methods on any or all of the properties. The classic example is having two date properties, where the `FromDate` cannot be after the `ToDate`, as shown in the following example:


```

public virtual DateTime FromDate { get; set; }

public string ValidateFromDate(DateTime d) {
    if (!d.IsAfterToday()) {
        return "Must be after Today";
    }
    return null;
}

public virtual DateTime ToDate { get; set; }

public string Validate(DateTime fromDate, DateTime toDate) {
    if (fromDate.Date > toDate.Date) {
        return "From Date cannot be after To Date";
    }
    return null;
}

```

In this example the `FromDate` property has a corresponding `ValidateFromDate` method to ensure that the date is after today. The `ToDate` property has no corresponding method - though it could have if, for example, you wanted to limit it to the next 12 months. The other `Validate` method is concerned with the relationship between the two properties. Note that this method is 'connected' to the two properties by dint of the specific names used for the parameters - writing the method as `Validate(DateTime d1, DateTime d2)` would result in the method being ignored as there are no `Date` properties called `d1` or `d2`. You can thus have several of these multi-property validation methods on an object - each addressing a different set of properties. In theory a single property can participate in multiple such validation methods (as well as its own individual validation method); in practice such an approach would likely lead to much confusion and be difficult to debug.

A multi-property validation method will only be called once each of the properties is itself in a valid state. In the above example, both the `FromDate` and `ToDate` properties are mandatory (because they have not been marked up as `Optional`). So the `Validate` method will only be called when both dates have been entered correctly. This means that, for example, there is no need to check for null values. The (string) message returned by the validate method (if validation has failed) will be displayed next to the property that has just been entered.

How to specify a default value for a property

If your property is a value type (the user types in text) and the required default value may be statically defined, then you can just use the `DefaultValue` attribute on the property.

If your property is a reference type (another domain object) *or* you wish to specify a default value dynamically, then you can create a `Default` method, for which the syntax is:

```

public <Property type> Default<PropertyName>()

```

For example:

```
public class Order
{
    public virtual Address ShippingAddress() Address { get; set;}

    public Address DefaultShippingAddress()
    {
        return Customer().NormalAddress();
    }
}
```

Use the Property - default snippet (shortcut `propdef`) for creating this method.

Value properties (such as dates and numbers) will, by default, show up on a transient object as blank fields. If you want a non blank field then you need to create a `Default` method as shown above. This is deliberate - so that the default behaviour is that the user is forced to enter a value.

If you create a transient object programmatically and set any value on that object within the same method then this will override any default value - as you would expect.

How to specify a set of choices for a property

See also [How to handle enum properties](#) and [How to specify auto-complete for a property](#) .

The simplest way to provide the user with a set of choices for a property (possibly rendered as a drop-down list, for example) is to ensure that the type used by the property is marked with the `Bounded` attribute - which will result in all instances of that type being offered to the user as a set of choices (typically as a drop-down list). If you wish to present the list with a sub-set of these, or with another customised set of choices - for example the set of all the Addresses known for a particular Customer - then you can write a `Choices` method:

And for specifying a list of choices is:

```
public <array or collection of property type> Choices<PropertyName>()
```

The full code for our example above is:

```
public class Order {
    public virtual Address ShippingAddress() Address { get; set;}

    public List<Address> ChoicesShippingAddress() {
        return Customer().AllActiveAddresses();
    }
}
```

Use the Property - default snippet (shortcut `propdef`) and the Property - choices snippet (shortcut `propcho`) for creating these methods.

Conditional Choices

It is possible to specify a set of choices for a property based on the selection(s) already made for another property or properties - for example to vary the available choices for a Province property based on the Country selected. We refer to this pattern as 'Conditional Choices'. An example is shown below:

```
public class Address {  
  
    public virtual Country CountryOfResidence {get; set;} //Where Country is a  
    [Bounded] class  
  
    public virtual Province Province {get; set;}  
  
    public IList<Province> ChoicesProvince(Country countryOfResidence) {  
        if (country == null) { return new List<Province>; }  
        var q = from p in Container.Instances<Province>()  
                where p.Country.Id == countryOfResidence.Id  
                select p  
        return q.ToList();  
    }  
}
```

In the above example code the selected value for the `CountryOfResidence` property is passed in as a parameter to the `ChoicesProvince` method. Note that the parameter name `countryOfResidence` must exactly match the property name `CountryOfResidence`, except for the character case, and the types (`Country`) must also match. Note also that the code guards against being called with a null value, returning an empty set of choices in this case (it could also return a default set of choices).

If used on a transient object, any values on which the set of choices depends must be passed in as parameters to the Choices method: you cannot refer to property values directly.

How to specify auto-complete for a property

A common pattern in a web application is to allow the user to start typing a string and to provide the user with a dynamically-generated drop-down list of matches. This can be achieved using an `AutoComplete[PropertyName]` recognised method. This may be applied to a string property, but is typically most useful in the context of reference objects, as in the example below:

```
public virtual Customer ForCustomer { get; set; }  
  
[PageSize(10)]  
public IQueryable<Customer> AutoCompleteForCustomer( [MinLength(3)] string name) {  
    return CustomerRepository.FindCustomerByName(name);  
}
```

When Naked Objects detects a matching `AutoCompleteXxx` method, as above, when the object is in Edit mode the user will be given a text field in which a string may be typed. The user

will be presented with a dynamically-generated drop-down list of object titles, based on the `IQueryable<T>` returned by the method. Note that it is up to the implementation of the method how the match is performed - for example whether the match is a 'starts with' or 'contains' and whether or not it is case-sensitive.

The `PageSize` attribute specifies the number of matches to be presented to the user (there is no ability to page through more matches, however). If no `PageSize` attribute is added then the default page size for the system will be used - to avoid the risk of returning a very large number of matches.

The `MinLength` attribute is also optional - this specifies the minimum number of characters that the user must provide before an attempted match is made. If no `MinLength` attribute is specified, the search will be initiated on entering a single character.

Note: For a reference property, the return type of the `AutoComplete` method must be either `IQueryable<T>` where `T` is the type of the property, or just `T` (i.e. a single matching object). For a `string` property only, the return type of the `AutoComplete` method may be `IEnumerable<string>`.

How to set up the initial value of a property programmatically

Initial values for properties may be set up programmatically within the `created()` method on the object. (See [The object life-cycle](#)).

How to trigger other behaviour when a property is changed

If you want to invoke functionality whenever a property is changed by the user, then you should create a `modify<propertyName>` and include the functionality within that. For example:

```
public virtual int Amount { get; set;}

public void ModifyAmount(int newAmount)
{
    Amount = newAmount;
    AddToTotal(newAmount);
}
```

Use the `Modify` snippet (shortcut `mod`) to create this method.

The reason for the `ModifyAmount` method is that it would not be a good idea to include the `AddToTotal` call within the property's `set` method, because that method may be called by the persistence mechanism when an object is retrieved from storage.

You may optionally also specify a `Clear<PropertyName>` which works the same way as `modify <propertyName>` but is called when the property is cleared by the user. `Clear<PropertyName>` does not take any parameters.

If the value of a property is changed by the user, from one non-null value to another non-null value, then the framework will first call the `Clear<PropertyName>` method (if it exists) and will then call the `Modify<PropertyName>` (if it exists) with the new value.

How to control the order in which properties are displayed

Use the `MemberOrder` attribute.

How to specify a name and/or description for a property

Specifying the name for a property

By default the framework will use the property name itself to label the property on the user interface. If you wish to over-ride this, use the `DisplayName` attribute on the property.

Specifying a description for a property

Use the `Description` attribute on the property itself.

The framework will take responsibility to make this description available to the user, for example in the form of a 'balloon help'.

How to hide a property from the user

To hide a property always: use the `Hidden` attribute.

To hide a property under certain conditions use a `Hide` method. The syntax is:

```
public bool Hide<PropertyName>()
```

A true return value indicates that the property is hidden. For example:

```
public class Order {
    public virtual string ShippingInstructions { get; set;}

    public bool HideShippingInstructions()
    {
        return hasShipped();
    }
    ...
}
```

Use the `Hide` snippet (shortcut `hide`) to create this method.

To apply the same rules to all the properties on an object - for example to hide all the properties once the object has been persisted. To do this, just create a method `HidePropertyDefault` that returns a `boolean`, just as for an individual hide-property method. For example:

```
public bool HidePropertyDefault() {  
    return Container.IsPersistent(this);  
}
```

If you wish to apply a rule to most, but not all of the properties then you may add the `HidePropertyDefault` method, and then provide individual `Hide` methods for properties where you wish to override this default behaviour.

Having a `HidePropertyDefault` method on a class and then using the *Hidden attribute* on individual properties is not recommended - as the behaviour is not guaranteed.

How to make a property non-persisted

If the property has a `get` but no `set` method then the field is not only unmodifiable but will also *not* be persisted. This may be used to derive a property from other information available to the object, for example:

```
public class Employee  
{  
    public virtual Department Department { get; set;}  
  
    //this is the derived property  
    public Employee Manager  
    {  
        get  
        {  
            if (Department == null)  
            {  
                return null;  
            }  
            else  
            {  
                return Department.Manager();  
            }  
        }  
    }  
    ...  
}
```

If you need to have a `get` and `set` for the property but do not wish to have that property persisted, use the `NotPersisted` attribute.

How to handle File Attachments

A common requirement is to be able to attach a file to a domain object.

Viewing a file attachment will typically result in a temporary file being written to your client machine in order to launch the viewing application. This is common practice but may pose security issues for certain types of business application.

The simplest way is to use a byte array, which can store any binary file. Entity Framework will recognise a byte array property and persist it as a single Binary column. At the user interface, Naked Objects will present the property as a link, and clicking on the link will attempt to launch a separate viewer to display the contents of the byte array.

The byte array property will be disabled automatically - the user will not be able to edit it directly. However, if an action method contains a byte array parameter, then this will be rendered on the user interface with a Browse button, allowing the user to browse for a file to upload. Thus, for an attached file that the user may modify the code will look something like this:

```
public virtual byte[] Attachment { get; set; }

public void AddOrChangeAttachment(byte[] newAttachment) {
    Attachment = newAttachment;
}
```

The use of a simple byte array to handle a file attachment has two limitations though:

- There is no ability to associate a file name with the content, so on the user interface the link will always be rendered as a generic **Show File** link.
- There is no associated MIME type, to give a hint of which type of viewer to launch.

To overcome these limitations, you have the option to use the `FileAttachment` type. The `NakedObjects.Value.FileAttachment` type (in `NakedObjects.Helpers.dll`) has properties for the attachment name, the MIME type, and the contents (as a byte array). The name is used as the text for the link on the user interface, and the MIME allows the system to determine how the content should be viewed. (If you are using Naked Objects MVC then a broad range of MIME types can be viewed directly inside the browser: other types will result in the launch of a separate viewer.)

The `FileAttachment` type is used for display purposes only, and will typically be derived from three separate persisted properties, which will typically be hidden from the user. As with byte array, you will need a separate action to allow the file attachment to be uploaded, as shown in the following example code:

```

public virtual FileAttachment Attachment {
    get {
        if (AttContent == null ) return null;
        return new FileAttachment(AttContent, AttName, AttMime){ DispositionType =
"inline" };
    }
}

[NakedObjectsIgnore]
public virtual byte[] AttContent { get; set; }

[NakedObjectsIgnore]
public virtual string AttName { get; set; }

[NakedObjectsIgnore]
public virtual string AttMime { get; set; }

public void AddOrChangeAttachment(FileAttachment newAttachment) {
    AttContent = newAttachment.GetResourceAsByteArray();
    AttName = newAttachment.Name;
    AttMime = newAttachment.MimeType;
}

//Alternatively:
//public void AddOrChangeAttachment(FileAttachment newAttachment, string withNewName)
//{
//    AttContent = newAttachment.GetResourceAsByteArray();
//    AttName = withNewName;
//    AttMime = newAttachment.MimeType;
//}

```

By setting the 'disposition type' to "inline" ({ `DispositionType` = "inline" } in the above code), the file attachment will be opened within the browser window (assuming that your browser has a viewer component for the mime type of the attachment. This is an optional behaviour.

How to associate multiple file attachments

To associate multiple file attachments, you will need to define a domain entity type (e.g. `Attachment`) that wraps the functionality shown above, and hold a collection of these new entity types, with suitable action methods for creating new ones (from an uploaded byte array or `FileAttachment`) or deleting existing ones.

How to display an image

Any byte array or `FileAttachment` property may contain an image, but it will be displayed as a link. To display an image within an object view, you need to use the `NakedObjects.Value.Image` type, from the `NakedObjects.Helpers.dll`. Image works in a very similar manner to `FileAttachment` (in fact it is a sub-type) but is specifically recognised by Naked Objects as indicating that the contents should be rendered as an in-line image. The following shows a typical example of how to add an image property - to be displayed in-line - within an object.


```

public virtual Image Photo {
    get {
        return new Image(PhotoContent, PhotoName, PhotoMime);
    }
}

[NakedObjectsIgnore]
public virtual byte[] PhotoContent {get; set;}

[NakedObjectsIgnore]
public virtual string PhotoName {get; set;}

[NakedObjectsIgnore]
public virtual string PhotoMime {get; set;}

public void AddOrChangePhoto(Image newImage) {
    PhotoContent = newImage.GetResourceAsByteArray();
    PhotoName = newImage.Name;
    PhotoMime = newImage.MimeType;
}

//Alternatively:
//public void AddOrChangePhoto(Image newImage, string withNewName)
//{
//    PhotoContent = newImage.GetResourceAsByteArray();
//    PhotoName = withNewName;
//    PhotoMime = newImage.MimeType;
//}

```

How to handle enum properties

Naked Objects can use [Enums](#) - either for properties or for action parameters. There are two patterns for doing this (shown here for properties).

If your project is built to .NET 4.5 or above then the simplest and best option is to use the [Enum](#) as the property type, as shown below:

```

public Sexes Sex {get; set;}
...
public enum Sexes {Male=1, Female=2, Unknown=3, NotSpecified=4}

```

At the user interface, the property will be displayed with the corresponding Name from and in Edit mode will be presented as a drop-down list. Note that the Names will be formatted using the same logic as class- and method-names in Named Objects, so that [NotSpecified](#) in the above example will be presented as Not Specified on screen. The options will be presented in alphabetical order: if you need to specify a different order, you may do this in a corresponding Choices method. (You may also specify a default value for the property).

The pattern above will not work with .NET 4.0, but you may use the alternate pattern shown below.

The second is to define an integer (or other 'integral type' such as a `short`, `long`, or `byte`) and then use the `System.ComponentModel.DataAnnotations.EnumDataType` to declare the `Enum` type that it corresponds to, as shown in this example:

```
[EnumDataType(typeof(Sexes))]  
public int Sex {get; set;}  
...  
public enum Sexes {Male=1, Female=2, Unknown=3, NotSpecified=4}
```

When using this pattern, a corresponding `Choices` or `Default` methods should return the same type as the property (an integer in the example above).

Collection properties

This section defines patterns and practices that are specific to properties that represent collections of domain objects. For the definition of what Naked Objects recognises as a collection, see [Recognised Collection types](#).

How to add a collection property to a domain object

Collections should be defined by the generic `System.Collections.Generic ICollection` interface but will need to be initialised with a concrete type such as `List`. The property must be marked `virtual`. The simplest way to add a collection is using the `coll` code snippet.

The following example shows an `Order` object containing a collection of `OrderLine` objects:

```
public class Order {  
    ...  
    private ICollection<OrderLine> myLines = new List<OrderLine>();  
  
    public virtual ICollection<OrderLine> Lines {  
        get {  
            return myLines;  
        }  
        set {  
            myLines = value;  
        }  
    }  
}
```

Naked Objects does not support multiple associations of value types (such as strings, numbers). This is not considered to be a significant constraint as a collection of values is not a common modelling pattern, and considered bad practice by some modellers - who suggest that any such collection should be implemented as domain (entity) objects. If you need to use a collection of value types for programmatic purposes, it is recommended that you mark the collection as `private` or `protected`, to ensure that it is ignored by the framework. Note, however, that the framework can make use of lists of value types to provide a set of choices for a single value property.

Adding-to or removing objects from a collection

In Naked Objects MVC the user may not directly add to or remove from a collection within an object in Edit mode: *all collection properties are automatically disabled*. If you want the user to be able to add to, or remove from a collection, then you should provide explicit actions to do this. The following shows an example:

```
public class Customer {

    #region Vehicles (collection)
    private ICollection<Vehicle> _Vehicles = new List<Vehicle>();

    public virtual ICollection<Vehicle> Vehicles {
        get {
            return _Vehicles;
        }
        set {
            _Vehicles = value;
        }
    }

    public virtual void AddToVehicles(Vehicle value) {
        if (!(_Vehicles.Contains(value))) {
            _Vehicles.Add(value);
        }
    }

    public virtual void RemoveFromVehicles(Vehicle value) {
        if (_Vehicles.Contains(value)) {
            _Vehicles.Remove(value);
        }
    }

    public IList<Vehicle> ChoicesRemoveFromVehicles(Vehicle value) {
        return Vehicles.ToList();
    }
    #endregion
}
```

Note the following:

The `AddToVehicles` and `RemoveFromVehicles` methods will show up in the user interface as actions. If the user invokes the `RemoveFromVehicles` action, the `ChoicesRemoveFromVehicles` method will provide the user with a drop-down list of all the existing vehicles in the collection.

If you use the `coll` snippet to add a collection into your object, then `Add` and `Remove` methods are automatically generated for you.

How to create a derived collection

Collections can be derived, in the same way as properties. These are not persisted, but are represented as `ReadOnly` collections. For example:

```

public class Department
{
    // Derived collection
    [NotPersisted, NotMapped]
    public ICollection<Employee> TerminatedEmployees
    {
        get
        {
            List<Employee> results = new List<Employee>();
            foreach (Employee e in Employees)
            {
                if (e.IsTerminated())
                {
                    results.Add(e);
                }
            }
            return results;
        }
    }
    ...
}

```

If you are working Code First, you should mark up the derived collection with a `NotMapped` attribute. Otherwise, Entity Framework will attempt to map the collection to database relationship. Adding the `NakedObjects.NotPersisted` attribute is optional, but is marginally more efficient when executing.

How to control the order in which table rows are displayed

To control, programmatically, the order in which rows are displayed, you can just incorporate the Linq `OrderBy` method into the `get` method for the collection. In the following example, the collection of `SalesOrderDetails` will be displayed (by default) ordered by the `UnitPrice`:

```

private ICollection<SalesOrderDetail> _details = new List<SalesOrderDetail>();

public virtual ICollection<SalesOrderDetail> Details {
    get {
        return _details.OrderBy((x) => x.UnitPrice).ToList();
    }
    set {
        _details = value;
    }
}

```

If you use this pattern, however, be aware that the collection returned by the property's `get` is not the same as the underlying private collection. Therefore, when you are adding to or removing from the collection, it is important that you work with the underlying collection. The best way to do this is to ensure that you always work through the `AddTo` / `RemoveFrom` associated methods - which are automatically added when you create a collection using the `coll` code snippet.

How to specify which columns are displayed in a table view

Use the [TableView](#) attribute.

Actions

An 'action' is a method that we expect the user to be able to invoke via the user interface, though it may also be invoked programmatically within the object model. The following conventions are used to determine when and how methods are made available to the user as actions.

How to add an action to an object

See [Action](#).

How to specify the layout of the menu of actions on an object

See [Object Menus](#).

How to define a contributed action

See [Contributed action](#).

How to prevent a service action from being a contributed to objects

If you want an action on a service to be available to the user on the service menu, but not contributed to object menus, use the [NotContributedAction](#) attribute.

How to specify parameter names and/or descriptions

As with properties, the framework will pick up parameter names reflectively and reformat these for presentation to the user. If you wish to override this mechanism and specify a different name then use the [DisplayName](#) attribute. This is especially useful if you wish to include punctuation or other characters that would not be permissible in a parameter name.

Similarly, any parameter may be given a short user-description using the [Description](#) attribute. The framework takes responsibility to make this available to the user.

How to make a parameter optional

Use the `Optionally` attribute.

How to specify a default value for a parameter

When an action is about to be invoked, then default values may be specified for any or all of its parameters.

If the parameter is a value type (the user types in text) and the required default value may be statically defined, then you can just use the `DefaultValue` attribute on the parameter.

If your parameter is a reference type (another domain object) *or* you wish to specify a default value dynamically, then you can use a `Default` method.

The default value for each parameter is specified via a separate method, with parameters numbered from zero. You need only write such methods for those parameters where you require a default value. There are two alternative versions of the syntax as follows:

```
public <parameter type> Default<ActionName>([<parameter type>
parameterNameSameAsOnAction])
//OR
public <parameter type> Default<parameter number><ActionName>()
```

The second syntax may be used where the action has more than one parameter of the same type. Note that parameters are numbered from zero.

Each method returns a single value of the appropriate type for its corresponding parameter . The following code shows both forms of the syntax:

```
public class Customer
{
    public Order PlaceOrder(Product product, int quantity, int promotionCode) {...}

    public Product DefaultPlaceOrder()
    {
        return ProductMostRecentlyOrderedBy(this.getCustomer());
    }

    public int Default1PlaceOrder()
    {
        return GetQuantityFromPreviousOrder();
    }
    ...
}
```

Use the Action - default snippet (shortcut `actdef`) to create this method.

How to specify a set of choices for a parameter

See also [Enums](#).

See also [How to specify auto-complete for a parameter](#).

Where the type of a parameter is annotated with `Bounded`, then the user will automatically be provided with each of the instances of that type in the form of a drop-down list or equivalent selection device. Sometimes, however, it is desirable to specify a set of choices that does not constitute a bounded set. This is achieved by adding one or more `Choices` methods. You need only write such methods for those parameters where you wish to specify a set of choices other than by using a bounded set. The recommended is:

```
public List<parameter type> Choices<parameter number><ActionName>()
```

Note that parameters are numbered from zero.

Each such method returns a collection (or an array) of the same type as the corresponding parameter. (Note that parameters are numbered from zero). For example:

```
public class Customer {
    public Order PlaceOrder(Product product, int quantity) {...}

    public List<Product> Choices0PlaceOrder()
    {
        return LastFiveProductsOrderedBy(this.Customer());
    }

    public Product Default0PlaceOrder() {...}

    ...
}
```

As shown in the examples, above, you may specify `Choices` and a `Default` value for the same parameter.

Use the Action - choices snippet (shortcut `actcho`) to create this method.

It is possible to specify a set of choices for a parameter based on the selection(s) already made for another parameter or parameters - for example to vary the available choices for a Province property based on the Country selected, as shown below:

```

public Address CreateAddress(string line1 As String, Country country, Province
province)

public IList<Province> Choices2CreateAddress(Country country) {
    if (country == null) { return new List<Province>; }
    var q = from p in Container.Instances<Province>()
            where p.Country.Id == countryOfResidence.Id
            select p
    return q.ToList();
}

```

In the above example code the numeral 2 in `Choices2CreateAddress` indicates that this method provides the choices for parameter 2 in the `CreateAddress` method - which is the province parameter (parameters are numbered from zero). However, the selected value for the `country` parameter (in this example the Country class is assumed to be a `Bounded` set) is passed in to this `Choices` method. Note also that the code guards against being called with a null value, returning an empty set of choices in this case (it could also return a default set of choices).

How to allow selection of multiple choices

If you want the user to be able to select multiple options from a list, then there are the following two options:

1. The parameter should be an `IEnumerable` of a domain object type and either:
 - that domain type is a bounded set
 - or a set of choices is provided via an action `Choices` method.
2. The parameter type is an `IEnumerable` of string or integer and a set of choices is provided via an action `Choices` method.
3. The parameter type is an `IEnumerable` of an enum

The first case is illustrated in the following example:

```

public void AddStandardComments(IEnumerable<string> comments) {
    foreach (string comment in comments) {
        Comment += comment + "\n";
    }
}

public string[] Choices0AddStandardComments() {
    return new[] {
        "Payment on delivery",
        "Leave parcel with neighbour",
        "Send SMS on delivery"
    };
}

```

How to specify auto-complete for a parameter

A common pattern in a web application is to allow the user to start typing a string and to provide the user with a dynamically-generated drop-down list of matches. This can be

achieved using an `AutoComplete[parameterNumber][ActionName]` recognised method. This may be applied to a `string` parameter, but is typically most useful in the context of reference parameters, as in the example below:

```
public virtual Order CreateNewOrder(Customer forCustomer) {...}

[PageSize(10)]
public IQueryable<Customer> AutoComplete0CreateNewOrder( [MinLength(3)] string name) {
    return CustomerRepository.FindCustomerByName(name);
}
```

When Naked Objects detects a matching `AutoCompleteXxx` method, as above, the user will be presented with a dynamically-generated drop-down list of object titles, based on the `IQueryable<T>` returned by the method. Note that it is up to the implementation of the method how the match is performed - for example whether the match is a 'starts with' or 'contains' and whether or not it is case-sensitive.

The `PageSize` attribute specifies the number of matches to be presented to the user (there is no ability to page through more matches, however). If no `PageSize` attribute is added then the default page size for the system will be used - to avoid the risk of returning a very large number of matches.

The `MinLength` attribute is also optional - this specifies the minimum number of characters that the user must provide before an attempted match is made. If no `MinLength` attribute is specified, the search will be initiated on entering a single character.

Note: For a reference parameter, the return type of the `AutoComplete` method must be either `IQueryable<T>` where `T` is the type of the parameter, or just `T` (i.e. a single matching object). For a `string` parameter only, the return type of the `AutoComplete` method may be `IEnumerable<string>`.

How to specify the length or format for text-input parameters

Use the `StringLength`, `TypicalLength`, `MultiLine`, `Mask` or `RegularExpression` attributes.

How to obscure input text (e.g. for a Password)

Use the `DataType` attribute: `[DataType(DataType.Password)]`.

How to validate parameter values

To validate that an input falls within a specific range, using the `RangeAttribute`.

To validate that an input value conforms to a particular format, use the `Mask` or `RegularExpression` attributes.

For more complex forms of validation, use a `Validate` method, which may be applied to any of the parameters individually, or to the set of parameters as a whole.

To validate a single parameter, there are two alternative forms of syntax:

```
public string Validate<ActionName>(<parameter type> parameterNameSameAsOnAction)
//OR
public string Validate<ParameterNumber><ActionName>(<parameter type> anyOldName)
```

The first syntax is easier to read. The second syntax may be used where the action has more than one parameter of the same type (to avoid the possibility of ending up with two validate methods that have identical signatures). Note that parameters are numbered from zero.

In both cases, a non-null return `String` indicates the reason why the member cannot be modified/action be invoked, and the viewing mechanism will display this feedback to the user. The following example code shows both forms of the syntax in use:

```
public class Customer {

    public Order PlaceOrder(Product p, int quantity
                           string purchaseOrderNumber, string comment)
    {...}

    public string ValidatePlaceOrder(Product p) {
        if (p.IsOutOfStock()) {
            return "Product is out of stock";
        }
        return null;
    }

    //Parameter number 2 is used to match up to 'purchaseOrderNumber'
    public string Validate2PlaceOrder(string pon)
    {
        if (! pon.StartsWith("PO"))
        {
            return "Purchase Order Number must start with 'PO'";
        }
        return null;
    }
}
```

You may also validate multiple parameters together in a single validate method. The parameter names should match those used in the action method.

```

public class Price {
    public void SpecifyApplicability(DateTime fromDate, DateTime toDate) {...}

    public string ValidateSpecifyApplicability(DateTime fromDate, DateTime toDate) {
        if (toDate.Date < fromDate.Date) {
            return "From date cannot be before To date";
        }
        return null;
    }
}

```

Use the Validate snippet (shortcut `val`) to create this method.

How to specify conditions for invoking an action

Disabling an action based on the state of the object

There may be circumstances in which we do not want the user to be able to initiate the action at all - for example because that action is not appropriate to the current state of the object on which the action resides. Such rules are enforced by means of a `Disable` method.

The syntax is:

```

public string Disable<ActionName>(<parameter type> param)

```

A non-null return `String` indicates the reason why the action may not be invoked. The framework takes responsibility to provide this feedback to the user. For example:

```

public class Customer
{
    public Order PlaceOrder(Product p, int quantity) {...}

    public string DisablePlaceOrder(Product p, int quantity)
    {
        if (isBlackListed())
        {
            return "Blacklisted customers cannot place orders";
        }
        return null;
    }
}

```

It is also possible to permanently disable an action using the `Disabled` attribute. One possible reason for doing this might be during prototyping - to indicate to the user that an action is planned, but has not yet been implemented.

Use the Disable snippet (shortcut `dis`) to create this method.

Disabling multiple actions on an object

You may wish to apply the same rules to all the actions on an object - for example to disable all the actions until the object has been persisted. To do this, just create a method `DisableActionDefault` that returns a string, just as for an individual disable-action method. For example:

```
public string DisableActionDefault() {
    if (!Container.IsPersistent(this)) {
        return "Cannot invoke this action until the object has been saved";
    }
    return null;
}
```

If you wish to apply a rule to most, but not all of the actions then you may add the `DisableActionDefault` method, and then provide individual `Disable` methods for actions where you wish to override this default behaviour.

Having a `DisableActionDefault` method on a class and then using the `Disable` attribute on individual actions is not recommended - as the behaviour is not guaranteed.

How to control the order in which actions appear on the menu

Use the `MemberOrder` attribute.

How to hide actions

To hide an action always use the `Hidden` attribute. (This is generally used where a `public` method on an object is not intended to be a user action).

To hide an action under certain conditions use a `Hide` method. The syntax is:

```
public bool Hide<ActionName>(<parameter type> param)
```

A `true` return value indicates that the action should not be shown. For example:

```
public class Order
{
    public void ApplyDiscount(int percentage) {...}

    public bool HideApplyDiscount()
    {
        return isWholesaleOrder();
    }
    ...
}
```

Use the `Hide` snippet (shortcut `hide`) to create this method.

To apply the same rules to all the actions on an object - for example to hide all the actions until an object has been persisted. To do this, just create a method `HideActionDefault` that returns a `boolean`, just as for an individual hide-action method. For example:

```
public bool HideActionDefault() {  
    return !Container.IsPersistent(this);  
}
```

If you wish to apply a rule to most, but not all of the properties then you may add the `HideActionDefault` method, and then provide individual `Hide` methods for actions where you wish to override this default behaviour.

Having a `HideActionDefault` method on a class and then using the `Hidden attribute` on individual actions is not recommended - as the behaviour is not guaranteed.

To hide an action for certain users or roles, see 'Properties: Hiding a property for specific users or roles'. The same technique can be applied to actions. However, the caveats apply.

How to pass a message back to the user

Sometimes, within an action it is necessary or desirable to pass a message to the user, for example to inform them of the results of their action ('5 payments have been issued') or that the action was not successful ('No Customer found with name John Smith').

`DomainObjectContainer` defines two methods for this purpose:

```
public void InformUser(string message)  
  
public void WarnUser(message As string)
```

How to work with transactions

All action methods are automatically wrapped in a transaction by the Naked Objects framework - so there is usually no need to write any specific code to start or end a transaction. If an exception is thrown within an action method, and not caught within application code, this will automatically cause the transaction to be aborted.

If you wish to abort a transaction, without throwing an exception, then you can call the `AbortCurrentTransaction` method on the Container.

Advanced Entity Framework techniques

This section provides a few notes on using more advanced features of the Entity Framework with Naked Objects for .NET.

How to handle concurrency checking

Naked Objects provides full support for concurrency checking - such that before a user saves any edits, or invokes any action upon an object, the framework will check to see that no other user has changed the state of that object.

Any domain object that needs to make use of this capability, must have a 'version' property that is guaranteed to change each time the persisted state of that object is changed. This property may be of type `DateTime` (acting as a time stamp), a `string`, a numeric value or a byte array. It may be visible to, or hidden from, the user. The responsibility for updating the property when changes are persisted may be performed by the domain code, but will more commonly be performed by the database, by means of a trigger or a calculated column.

The property must be marked up with the `ConcurrencyCheck` attribute, as shown in the example below:

```
public class Employee {  
    ...  
  
    [ConcurrencyCheck]  
    public virtual DateTime LastUpdated { get; set; }  
}
```

If you have any inheritance within your domain model, then the `ConcurrencyCheck` attribute should be applied to a property on the top-most class of each hierarchy, and should not be duplicated within any sub-classes. (Sub-classes may have their own `LastUpdated` or similar properties for other purposes, but these do not play a role in concurrency checking.)

How to specify 'eager loading' of an object's reference properties

By default, Entity Framework uses lazy loading - if an object holds a reference to another object then that second object is retrieved from the database only when the property is accessed. However, if a user retrieves and displays an object then many of the object's reference properties will be accessed, and this will result in multiple round-trips to the database. If this results in unacceptable performance (often, it won't) then the answer is to force the object to load some or all of its associated objects in one trip - this is known as 'eager loading'.

This is done using the `.Include` method in your LINQ queries. With the Entity Framework CTP Microsoft has provided an extension method that makes the the `.Include` method much easier to work with. To use this, ensure that you have a reference to the CTP within your project. This may be found at:

```
C:\Program Files\Microsoft ADO.NET Entity Framework Feature  
CTP5\Binaries\Microsoft.Data.Entity.CTP.dll
```

(on a 64-bit machine that's C:\Program Files (x86)...))

The following code shows an example of eager loading within a query method - ensuring that the object referenced within the `Customer's Pet` property is loaded at the same time as the `Customer`:

```
using system.data.entity;

var petOwners = Container.Instances<Person>().Where(x => x.HasPet).Include(x =>
x.Pet);
```

How to implement complex types

The core Naked Objects framework works well with the full capabilities of Complex Types, as defined by Entity Framework. However, the ability to use Complex Types within your application does depend upon the 'viewer' that you are using with the core framework:

- The Naked Objects MVC user interface supports ordinary Complex Types, but does not support *nested* Complex Types (where a Complex Type has a property that is itself a Complex Type).
- The Restful Objects API does not currently support Complex Types at all (though this is might be added in a future release).

(As of NOF 7.0) When working Code First it is necessary to annotate the complex type class with the `ComplexType` attribute, *even if you have specified a complex type by means of the Code First Fluent API*. This is because Naked Objects needs to be able to detect that it is a complex type independently of Entity Framework.

How to work with multiple databases

Naked Objects can cope with multiple databases. You will need to add the `DbContext` for each database in the `EntityObjectStoreConfiguration`. See [Creating a Model project from scratch](#).

How to work with multiple database contexts

Naked Objects provides very good support for working with multiple contexts, including explicitly-coded contexts (as used in Code First mode) or implicit contexts (as used when working with entity models defined in `.edmx` files), or a combination of the two, as shown below:

```

public static EntityObjectStoreConfiguration EntityObjectStoreConfig() {
    var config = new EntityObjectStoreConfiguration();
    config.UsingEdmxContext("ModelA"); //Has corresponding .edmx file
    config.UsingEdmxContext("ModelB"); //Has corresponding .edmx file
    config.UsingCodeFirstContext(() => new ModelCContext());
    config.UsingCodeFirstContext(() => new ModelDContext());
    return config;
}

```

These various contexts may correspond to separate databases, or they may point to the same shared database(s).

At run-time, when an object is retrieved (whether explicitly via `Container.Instances<T>`, or just by navigation) Naked Objects transparently identifies which of the contexts the type resides in and instructs Entity Framework to retrieve the object from that context. This is a very powerful capability, and enables things like polymorphic association.

However, this pattern does carry an overhead, that grows with the number of contexts you have. This is because - due to a severe limitation in the design of Entity Framework - the only way to find out if a context knows about a given domain type is to ask for that type, and catch the exception thrown if that type is not known. Exceptions are relatively expensive in processing terms, so raising lots of them - from polling multiple contexts - can be slow. Naked Objects does cache the mapping of types to contexts, but this cache is generated progressively, as each type is encountered.

This overhead can be eliminated by explicitly associating types with their context, using the syntax below:

```

public static EntityObjectStoreConfiguration EntityObjectStoreConfig() {
    var config = new EntityObjectStoreConfiguration();
    config.UsingEdmxContext("ModelA").AssociateTypes(ModelATypes);
    ...
    config.UsingCodeFirstContext(() => new
        ModelCContext()).AssociateTypes(ModelCContext.ModelCTypes);
    ...
    return config;
}

private static Type[] ModelATypes() {
    return new Type[] {typeof(Payment), typeof(Invoice), ... };
}
...

```

The `AssociateTypes` method (on `ContextInstaller`, which is returned by `Using...Context`) takes as a parameter any method that returns an array of types. Each of these specified associations will be cached on the session. The methods defining the array of types could be local - as shown for `ModelATypes()` above - or as an external static method, for example, on the appropriate code first `DbContext` - as shown for `ModelCTypes` above.

Although it is sensible to prime the cache with all the types that you will be using, there is no requirement to do so. If the framework comes across a type for which it does not know the context, it will poll the contexts to find it, and then cache the association - albeit with the overhead mentioned previously.

As a further refinement to this performance optimisation, there is a method `SpecifyTypesNotAssociatedWithAnyContext` on the `EntityObjectStoreConfiguration`. This method only adds value where you have a domain class (typically an abstract class, though not necessarily) that is not itself an entity - and therefore unknown to Entity Framework - but from which one or more entity types inherit.

A simple example of this exists within our AdventureWorks sample project, where many domain classes inherit from `AWDomainObject` - which provides a small amount of functionality common to many classes. In ordinary operation, Naked Objects will work up the inheritance hierarchy, until it finds a class that is unknown to any `DbContext`, and then work with the next level down. This incurs the overhead of a thrown and caught exception mentioned earlier. To improve efficiency, the AdventureWorks sample project now contains this code:

```
public static EntityObjectStoreConfiguration EntityObjectStoreConfig() {
    var config = new EntityObjectStoreConfiguration();
    config.UsingEdmxContext("Model").AssociateTypes(AllPersistedTypesInMainModel);
    config.SpecifyTypesNotAssociatedWithAnyContext(() => new[] {typeof
(AWDomainObject)});
    return config;
}
```

If there was more than just the class `AWDomainObject` to which this applied, then the method may be called with a delegate, in a manner similar to `AssociateTypes` e.g.:

```
installer.SpecifyTypesNotAssociatedWithAnyContext(TypesToIgnore);
```

The `EntityObjectStoreConfiguration` has a `RequireExplicitAssociationOfTypes`, which is set to false by default. If set to true when initialised, then the framework will throw an uncaught exception if the persistor is asked for any domain type that has not either been associated with a specific `DbContext` (using `AssociateTypes`) or with no context (using `SpecifyTypesNotAssociatedWithAnyContext`).

How to write safe LINQ queries

Don't use the equality operator on objects; test for equality on the value properties

Don't write:

```
public IQueryable<Product> FindProducts(ProductCategory category) {
    return Container.Instances<Product>().Where(x => x.Category == category);
}
```

Write:

```
public IQueryable<Product> FindProducts(ProductCategory category) {  
    return Container.Instances<Product>().Where(x => x.Category.Id == category.Id);  
}
```

Don't call any method on a domain object within a query; refer only to properties

Don't write:

```
public IQueryable<Product> ListDiscontinuedProducts() {  
    return Container.Instances<Product>().Where(x => x.IsDiscontinued());  
}
```

Write:

```
public IQueryable<Product> ListDiscontinuedProducts() {  
    return Container.Instances<Product>().Where(x => x.Status == "Discontinued");  
}
```

Note, though that you can call methods on System classes e.g. Trim().ToUpper() on string.

Don't navigate references on any objects passed into the query; pass in any such required indirect references as variables in their own right

Don't write:

```
public IQueryable<Order> FindOrdersNotSentToBillingAddress(Customer cust) {  
    return Instances<Order>().Where(x => x.SentTo.Id != cust.BillingAddress.Id);  
}
```

Write:

```
public IQueryable<Order> FindOrdersNotSentToBillingAddress(Customer cust) {  
    Address billing = cust.BillingAddress;  
    return Instances<Order>().Where(x => x.SentTo.Id != billing.Id);  
}
```

When doing a join, don't try to use Container.Instances<T>() more than once inside a query; define a separate IQueryable<T> outside the query

Don't write:

```
var q = from p in Container.Instances<Product>()
        from c in Container.Instances<Customer>()
        where ...
```

Write:

```
var customers = Container.Instances<Customer>();
var q = from p in Container.Instances<Product>()
        from c in customers
        where ...
```

Or, for greater clarity:

```
var customers = Container.Instances<Customer>();
var products = Container.Instances<Product>();

var q = from p in products
        from c in customers
        where ...
```

How to handle associations that are defined by an interface rather than a class

Entity Framework requires that any association is defined by a type that a Class (whether concrete or abstract). It does not natively support the concept of an association that is defined by an Interface.

To overcome this limitation of Entity Framework, *Naked Objects* provides two patterns for achieving the goal: the 'result interface association' and the 'polymorphic association'.

Result interface association

This is the simpler case where a property is defined by an interface but there is expected to be just a single implementation of that interface. In such cases the Interface is not defining a role for multiple objects to play - it is simply defining a reduced view of an object that is returned as the result of a query, say, perhaps in order to hide other aspects of that object's implementation. The 'role interface association' pattern as described above would still work here, but it is overkill. The specific 'result interface association' pattern is simpler to code, and potentially faster in execution. (It also means that the relationship may be implemented at database level as a foreign key, which the role interface association does not permit.)

The example below shows how to code the pattern.

Use the Result Interface Association snippet (shortcut `resulttia`) to help create this code.

```

public class Order {

    [Hidden()]
    public virtual int CustomerId {get; set;}

    private ICustomer _Customer;

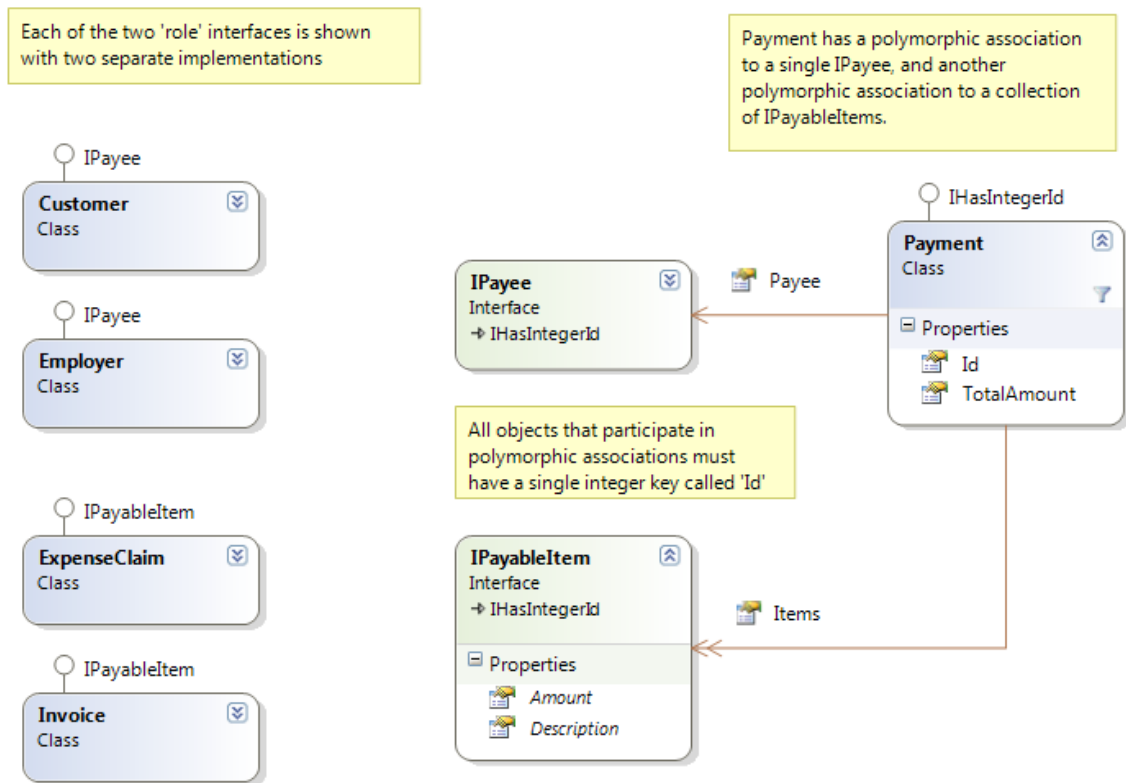
    [NotPersisted()]
    public ICustomer Customer {
        get {
            if (_Customer == null && CustomerId > 0) {
                _Customer = CustomerRepository.FindById(CustomerId);
            }
            return _Customer;
        }
        set {
            _Customer = value;
            if (value == null) {
                CustomerId = 0;
            }
            else {
                CustomerId = value.Id;
            }
        }
    }
}

```

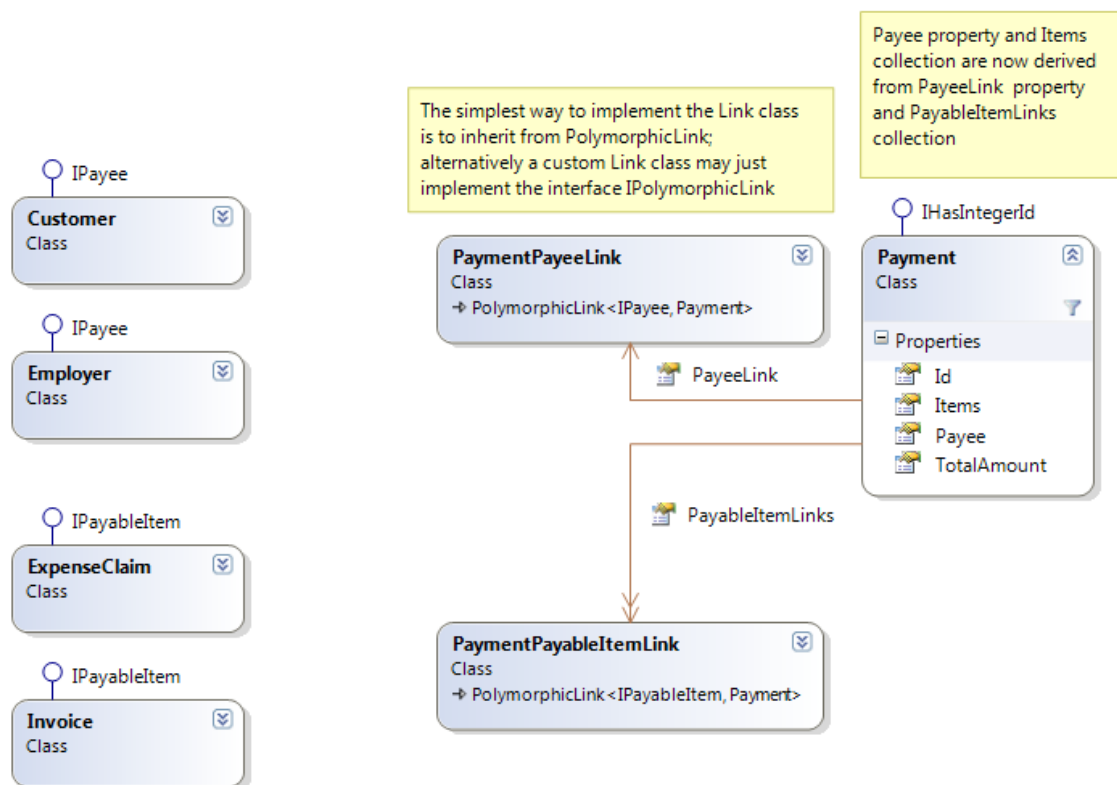
Polymorphic Association

Polymorphic Associations (PAs) - associations between objects where the type is defined by an 'role' interface rather than by a concrete or abstract class.

The following example is based on a Payment object that has two PAs. The first is a single association to a Payee, defined by the role interface IPayee. In the example there are two separate implementations of IPayee: Customer and Employer. In practice there may be many more. The second PA is a multiple association, to a collection of type IPayableItem. In this example there are two implementations of the role IPayableItem: ExpenseClaim and Invoice. The intent of the domain model is represented in this diagram:



Next we turn to the implementation of the PA - initially from a domain modelling perspective, as shown below:



Note the addition of two new domain classes: `PaymentPayeeLink` and `PaymentPayableItemLink`. Both of these inherit from a generic helper class - `PolymorphicLink<TRole, TOwner>` - which is defined in the `NakedObjects.Helpers` assembly, installed by the `NakedObjects.ProgrammingModel` package.

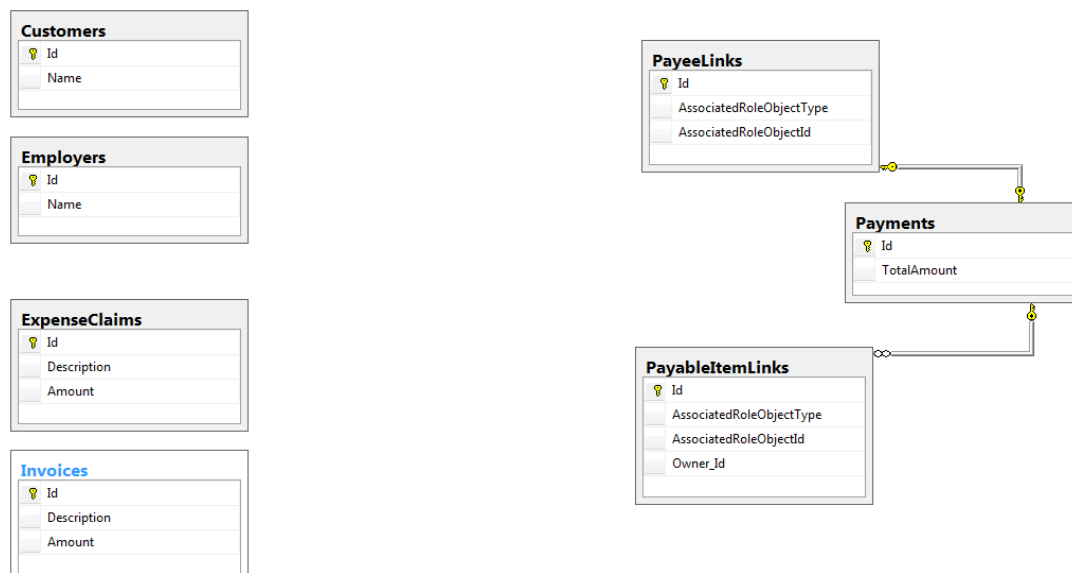
For this particular example, the two new classes do not add any properties or methods to the generic helper class, they are simply defined thus:

```
public class PaymentPayeeLink : PolymorphicLink<IPayee, Payment> { }
```

```
public class PaymentPayableItemLink : PolymorphicLink<IPayableItem, Payment>
```

However, these Link classes could, in principle, add custom properties and/or methods. A particular use of this might be to de-normalise some properties (such as a Title property or other summary information) from the associated object.

Working CodeFirst, each of these Link domain objects will result in an equivalent table in the database. This is shown in the database diagram below (table names have been shortened for clarity):



Note that each of the Link tables has a foreign key reference back to the 'owner' (in this case a Payment). In the case of `PayeeLinks`, the foreign key is also the Id - because it is a single association. Each Link table has two columns that, in combination, specify the associated

object: `AssociatedRoleObjectType` and `AssociatedRoleId`. The latter is always an integer (one constraint of this pattern is that all objects that participate in the PA must have a single integer `Id` property called 'Id' in the code (the naming convention in the database is not mandated). The `AssociatedRoleObjectType` is a string, and may either take the form of the fully-qualified type name (e.g. 'MyApp.Customer') or a standard abbreviation (e.g. 'CUS') - this is configurable and may even be mixed. These two items permit Naked Objects to retrieve the - corresponding object - using methods provided by the generic `PolymorphicLink` class (which, in turn, delegates to an injected service called `PolymorphicNavigator`, which is also included in the `NakedObjectsProgramming` model).

Using the code snippets

Almost all the domain code needed to implement this pattern is generated by the use of two code snippets that are installed with the `NakedObjects.Ide` package (which you should install at solution level):

- `polyprop` for a polymorphic property such as the payee in the example above.
- `polycoll` for a polymorphic collection such as the payable items in the example above.

You will need to [register](#) the `NakedObjects.PolymorphicNavigator` as a service.

The code below shows code from the example that was generated entirely by the `polyprop` snippet:

```
#region Payee Property of type IPayee ('role' interface)

[Disabled]
public virtual Payment_Payee_Link PayeeLink { get; set; }

private IPayee _Payee;

[NotPersisted]
public IPayee Payee {
    get {
        return PolymorphicNavigator.RoleObjectFromLink(ref _Payee, PayeeLink, this);
    }
    Set {
        _Payee = value;
        PayeeLink = PolymorphicNavigator.UpdateAddOrDeleteLink(_Payee, PayeeLink, this);
    }
}

public void Persisting() {
    PayeeLink = PolymorphicNavigator.NewTransientLink
        <Payment_Payee_Link, IPayee, PolymorphicPayment>(_Payee, this);
}
#endregion
```

The example code below was generated using the `polycoll` snippet:

```

#region PayableItems Collection of type IPayableItem

private ICollection<Payment_PayableItem_Link> _PayableItem =
    new List<Payment_PayableItem_Link>();

[NakedObjectsIgnore]
public virtual ICollection<Payment_PayableItem_Link> PayableItemLinks {
    get {
        return _PayableItem;
    }
    set {
        _PayableItem = value;
    }
}

public void AddToPayableItems(IPayableItem value) {
    PolymorphicNavigator.AddLink
        <Payment_PayableItem_Link, IPayableItem, PolymorphicPayment>(value, this);
}

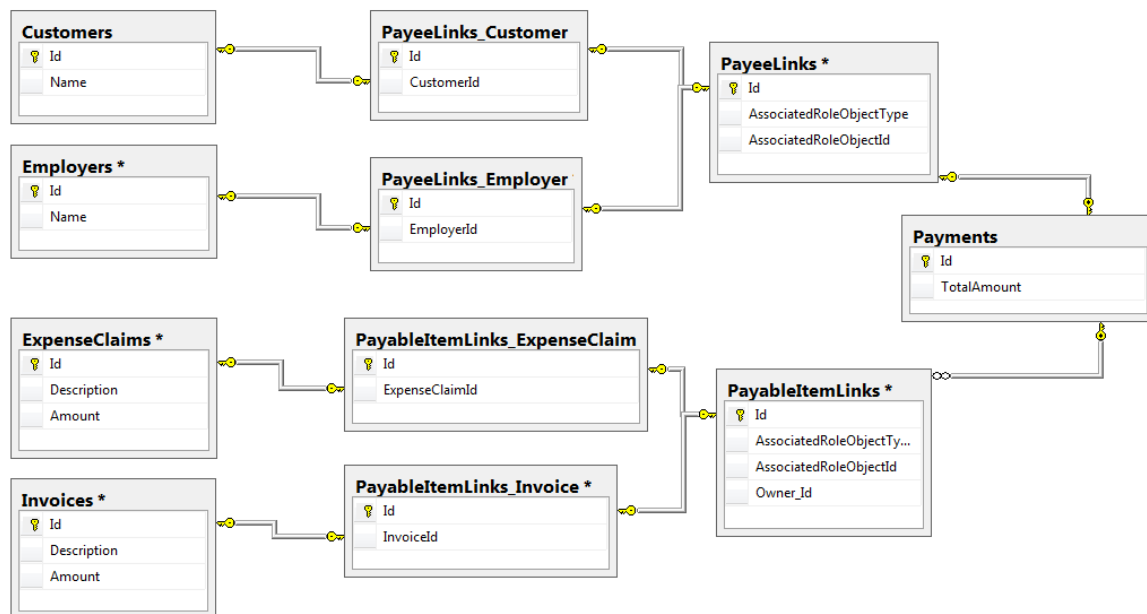
public void RemoveFromPayableItems(IPayableItem value) {
    PolymorphicNavigator.RemoveLink
        <Payment_PayableItem_Link, IPayableItem, PolymorphicPayment>(value, this);
}

[NotPersisted]
public ICollection<IPayableItem> PayableItems {
    get {
        return PayableItemLinks.Select(x => x.AssociatedRoleObject).ToList();
    }
}
#endregion

```

Adding optional (database) referential integrity to a polymorphic association

As may readily be observed from the diagram above, at this stage the database schema does not have referential integrity across the PA; and navigating for reporting purposes, while possible, will not be easy. If this is an issue for your application, then it may optionally be addressed added by manually adding further tables as shown in the example below:



Notice that for each of the polymorphic link tables (on the right hand side) there is now an added table *for each of the implementations of that role*. Thus PayeeLinks has associated PayeeLinks_Customer and PayeeLinks_Employer tables. Each of these ‘second half’ tables (on the left-hand side of the diagram) has an FK both to a row in the corresponding Link table, and to its ‘role’ object. These ‘second-half’ tables will need to be created manually, and maintained by custom database triggers: they are invisible to Entity Framework and the domain code.

Customising the MVC User Interface

There are three ways that you can customise the generic HTML user interface that is created by Naked Objects MVC automatically:

Customising the CSS

The generic user interface created by Naked Objects MVC relies on CSS (Cascading Style Sheets) for all of its styling. From within your browser you can temporarily switch the styling off - the result is not pretty but it is fully functional. (You will see, for example, that the drop-down menus are created entirely through styling rules. The only use of JavaScript is for the pop-up calendar helper, which will simply not show if you have JavaScript turned off).

You can customise the generic user interface to a surprisingly high degree just by modifying the .css file(s). You can, for example, apply styling changes to individual classes of domain object, individual properties within an object, individual actions, or individual parameters within an action dialog, and so on. This is because the HTML generated by Naked Objects

MVC makes extensive use of the `class` and `id` attributes. Consider the following view of a `Store` object from the AdventureWorks demo:

Naked Objects MVC



[About](#)[Home](#)

Customers	Orders	Products	Employees	Sales	Special Offers	Contacts	Vendors	Purchase Orders	Work Orders
-----------	--------	----------	-----------	-------	----------------	----------	---------	-----------------	-------------



Eastside Cycle Shop, AW00000664

Store Actions

Account Details:	 AW00000664
Store Name:	Eastside Cycle Shop
Demographics:	AnnualSales: 800000 AnnualRevenue: 80000 BankName: Guardian Bank BusinessType: BM YearOpened: 1988 Specialty: Touring SquareFeet: 21000 Brands: 2 Internet: T1 NumberEmployees: 11
Sales Person:	 Lynn Tsoflias
Last Modified:	13/10/2004 11:15:07
Contacts:	1 Store Contact List Table
Customer Addresses:	1 Customer Address List Table

[Edit](#)

The HTML for the `Sales Person` property is shown below (formatted for clarity):

```
<div class="Property" id="Store-SalesPerson">
  <label>Sales Person:</label>
  <div class="Object" title="">
    
    <a href="/Store/Details/AdventureWorksModel.SalesPerson%3bAdventureWorksModel.
      SalesPerson%3b1%3bSystem.Int32%3b290%3bFalse%3b%3b">Lynn Tsoflias</a>
  </div>
</div>
```

We can use the `class` and `id` to specify styling just for this property, for example, by adding the following to the `Site.css` (or to another `.css` file that is imported into `site.css`):

```
.Property#Store-SalesPerson { color: Red; }
```

to get the following effect:

Account Details:	 AW00000221
Store Name:	Bike Dealers Association
Demographics:	AnnualSales: 1500000 AnnualRevenue: 150000 BankName: United Security BusinessType: OS YearOpened: 1974 Specialty: Road SquareFeet: 40000 Brands: 3 Internet: T1 NumberEmployees: 40
Sales Person:	 Shu Ito
Last Modified:	13/10/2004 11:15:07
Contacts:	1 Store Contact <input type="button" value="List"/> <input type="button" value="Table"/>
Customer Addresses:	1 Customer Address <input type="button" value="List"/> <input type="button" value="Table"/>

This is a trivial example, but you can use similar techniques to control the whole layout and visual presentation of individual objects and actions.

Writing `.css` can be a challenging exercise and you are advised not to try it until you have gained a thorough grounding in the technology and techniques of `.css`, which is outside the scope of this manual. However, customisation entirely through `.css` offers these huge advantages:

- You can modify the `.css` files while the solution is running to see the effect (you just need to hit Refresh on your browser each time).
- Because changing the CSS does not impact the functionality of the application, you can manage the styling of the application as a completely independent process - for example delegating it to a specialised designer. The application development is not then dependent upon the styling process.

Custom Views

If you are unable to achieve the customisation that you need entirely through `css`, then the next option is to create custom views. A typical example of this is where you wish to display more on a page than exists for the standard generic view. For example:

- If an `Order` object contains a collection of `OrderLine` objects, then the default view will show this in summary form, requiring the user to click on the List or Table button to see more (this 'lazy loading' approach allows the Order to be retrieved faster). However you might wish to have the collection of order lines rendered as a table even when the order is first retrieved.
- The Order will probably also have an action `AddNewLine` available on its action menu, and which returns a dialog asking for the `Product` and `Quantity`, say. With a customised view you could arrange to have this action dialog exist on the same page as the Order - but only if the order status showed that it was not already shipped.

Writing a new view is quite straightforward. The easiest approach is to start from one of the default views that are used by a Naked Objects MVC project: you will find them under:

`Views > Shared`. These views follow standard ASP.NET MVC conventions. For example, by default a single object will be displayed using `ObjectView.aspx`.

To create a customised view for the `Order` object, just create a new folder within `Views` called `Order`, copy the `ObjectView.aspx` file into this folder, keeping the same name for the file - and then customise the new view. Naked Objects MVC will automatically detect, and use, this new view whenever it is displaying an `Order`. If you subsequently remove or rename that file or folder then Naked Objects MVC will revert to using the default `ObjectView.aspx` file in `Views > Shared`.

Similarly, you can create a customised view for a collection of `Order` objects by copying `StandaloneTable.aspx` from the `Shared` folder to the `Order` folder, and then modifying it.

When writing custom pages and using AJAX you should be aware that only the content on the page within the `div#main` tag will be updated from the AJAX request. This means that any custom content *outside* of `div#main` (such as scripts and style tags added into the custom page) will not be copied into the page. For this reason we recommend that scripts and style tags are not directly added to a custom page but are coded into separate `.css` or `.js` files. These may then be referenced from the master pages. Scripts should adopt the 'unobtrusive' style of Javascript programming and use JQuery events and binding. If you do add scripts and style tags into a custom page be aware that you may need to do some further custom coding to have those scripts execute.

A set of Naked Objects HTML helper methods make it really easy to create custom views - and keep the resulting code really small. As with the standard HTML helper methods (that form part of ASP.NET MVC) these take the form extension methods on the type `HtmlHelper`. They are defined on two classes: `NofHtmlHelper` and `CustomHtmlHelper` (both in the namespace `NakedObjects.Web.Mvc.Html`). Those defined on `NofHtmlHelper` are used within the standard shared views in Naked Objects MVC but these may also be used in your own custom views; those defined on `CustomHtmlHelper` provide additional capabilities intended specifically for custom views.

The following example shows a custom page to view a `Workflow` object using several of the helper methods:

```

<%@ Page Title="" Language="C#"
MasterPageFile="~/Views/Shared/Site.WithServices.Master"
Inherits="System.Web.Mvc.ViewPage<MyApp.Model.Workflow>" %>
<%@ Import Namespace="NakedObjects.Web.Mvc.Html"%>
<%@ Import Namespace="MyApp.Model.Workflow"%>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
    <%: Html.ObjectTitle(Model)%>
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <div class="ObjectView" id="<%: Html.ObjectTypeAsCssId(Model) %>">
        <%: Html.Object(Model)%>
        <%: Html.Menu(Model)%>
        <%: Html.UserMessages() %>
        <%if (Html.ObjectHasVisibleFields(Model)) {%>
            <%: Html.PropertyListWithout(Model, x=> x.SubWorkflows)%>

            <div id="Process-PropertyList1" class="PropertyList">
                <div class="Property" id="Workflow-SubWorkflows"><label>Sub
Workflows:</label>
                <%: Html.CollectionTableWith(Model.SubWorkflows, "Order", "WorkflowName",
"Status", "Priority")%>
                </div>
            </div>

            <%: Html.GenericAction("Edit", "EditObject", Model)%>
        <%}%>
    </div>
</asp:Content>

```

Note the following:

- `Inherits="..."` This is what is known in ASP.NET MVC terminology as a 'strongly-typed' view - all references to `Model` will be to an object of type `MyApp.Model.Workflow`.
- `<%: Html.ObjectTitle(Model)%>` specifies the title of the page to be the title of the (Workflow) object that we are viewing.
- `<div class="ObjectView" id="<%: Html.ObjectTypeAsCssId(Model) %>">` Specifying the `id` of the `div` using this helper method allows the `.css` to style this object view specifically, should we want that.
- `<%: Html.Object(Model)%>` adds the icon and the title for the object
- `<%: Html.Menu(Model)%>` adds the menu of actions for this (Workflow) object.
- `<%: Html.UserMessages() %>` specifies that this is where any user messages, such as validation errors, are to be rendered.
- `<%: Html.PropertyListWithout(Model, x=> x.SubWorkflows)%>` generates a list of all the properties of the object, *except for the* `SubWorkflows` *property* (which is rendered separately, below). Note that we could have written this more simply as `<% Html.PropertyList(Model, "SubWorkflows") %>`. The advantage of the slightly more complex syntax, which uses a Lambda expression, is that it is type-safe: if the `SubWorkflows` is subsequently renamed in the domain object to, say, `SubsidiaryWorkflows`, then this will be automatically renamed here; this would not be so if the name of the property is defined by a literal string.
- The next three lines render a view of the objects `SubWorkflows` collection, showing only the columns `Order`, `WorkflowName`, `Status` and `Priority`. The reason for

manually adding the two `div`s around this is simply to pick up the styling for collections from the standard `.css` - it could have been omitted and styled separately.

- `<%: Html.GenericAction("Edit", "EditObject", Model)%>` provide the generic Edit action (rendered as a button by the standard `.css`).

The following is an incomplete list, showing just some of the helper methods that you may like to use in creating your own custom views:

ServiceExtensions

`Services` - renders the complete set of service menus available to the user.

`SingleServiceMenu` - renders a single service menu.

`SystemMenu` - adds a menu that provides certain system level actions. This is normally used during prototype development only, as the actions are not typically relevant (or safe) for end-users.

ObjectExtensions

`Object` - renders the icon and title for a specified object

`IconName` - displays the name of the icon for a specified object

PropertyExtensions

`ObjectPropertyView` - renders a complete view of a single specified object, with an Edit button.

`ObjectPropertyEdit` - renders a complete view of a single specified object in Edit mode, with a Save button.

`PropertyList` - displays all of an object's properties

`PropertyListWithout` - displays all of an object's properties, except those specified.

`PropertyListWith` - displays a specified list of properties of an object.

`PropertyListEdit` - As for `PropertyList` but rendered in Edit mode.

`PropertyListEditWithout` - As for `PropertyListWithout` but rendered in Edit mode.

`PropertyListEditWith` - As for `PropertyListWith` but rendered in Edit mode.

`ObjectHasVisibleFields` - Returns true if an object has any properties that could be displayed to this user.

`Contents` - displays the contents of a specified object property

Name - displays the name of a specified object property

Description - displays the description (if one has been provided in the Model) of a specified object property

TypeName - displays the object type of the property

ActionExtensions

Menu - Renders all of the actions on an object (that are available to the user) as a menu

ObjectAction - Renders a specified action from an object in the form of a button.

ObjectActionAsDialog - Renders a specified action from an object in the form of a dialog.

ParameterList - renders the list of parameters for a specified action.

ActionDialogId - renders the ID for an action (to allow it to be invoked by the Naked Objects framework).

ActionName - renders the (formatted) name of the action.

ControllerAction - renders one of the generic framework actions buttons such as Edit, Select, or Remove.

UserMessages - renders the message resulting from validation errors (on an action dialog or attempting to save an edited object)

CollectionExtensions

CollectionTable - renders a specified collection as a table view showing all properties as columns

CollectionTableWith - renders a specified collection as a table view, including only the specified properties as columns

CollectionTableWithout - renders a specified collection as a table view, including all properties, except those specified, as columns

Helpers that apply to objects, properties and/or parameters

Contents - renders the contents of a specified object property or action dialog parameter.

TypeName - renders the name of the object type (class) for a specified object, object property or action dialog parameter.

Name - renders the name (label) for a specified object property or action dialog parameter.

Description - renders the description (tool-tip text) for a specified object property or action dialog parameter.

Custom controllers

The third level of customisation is to write your own custom controllers. Examples of where this would be needed include:

- Where you have more than one view for an object (or collection of objects) and you wish to use different views in different contexts. (However, you should first ask yourself if it would be possible to achieve the same effect by having a single view that had conditional statements built into it - as this would require no new controller).
- Where you wish to provide a view that contains than one domain object (other than a collection of domain objects, which is handled already), or a domain object plus some additional data. This can be done either by feeding additional objects into the `ViewData` associated with a view, or by creating a 'View Model'. These are both considered to be fully legitimate within ASP.NET MVC. However, we caution against over-use of these patterns. If you are having to create a lot of View Model objects, for example, this might suggest that your underlying domain model is not a good match to the business domain.
- Where you wish to control, explicitly, the flow of the application. By default, a Naked Objects MVC application allows the user to perform any legitimate action in any context (while still enforcing necessary rules). This is a very 'empowering' style of user interface and gives the user a great deal of control. But this style of user interface has a learning curve - the user is not told what to do next. It is appropriate for many internal-facing applications for a business, but very seldom appropriate for external-facing applications, where a more 'narrow', step by step, style of interface is called for.

Custom controllers are written using standard ASP.NET MVC patterns. The more custom controllers you write, the more your application starts to resemble a conventional ASP.NET MVC application. So add them only as you are sure that you need them. A recommended strategy is:

1. Start by creating an internal 'power-user' application using default Naked Objects patterns and with the minimum possible use of custom controllers (you might well find that none are needed).
2. Then add one or more narrowly-scripted applications for external use, written using conventional ASP.NET MVC patterns, but leveraging the Naked Objects helper methods in the views and/or controllers.

To start to write custom controllers in Naked Objects MVC, you must be familiar with how to write conventional ASP.NET MVC applications. Books, videos and other training materials are readily available from Microsoft and other sources. If you do not have familiarity with writing MVC applications, you will have difficulty following the example below, let alone write your own.

Example

This example demonstrates a Naked Objects MVC application with a highly-constrained user interface that makes use of (amongst other things) a custom controller. It is taken from the simple e-commerce application was used on the Naked Objects website *prior to the framework being open-sourced!* Let's first of all walk through the finished application from a user's viewpoint. The starting point is a regular HTML page on the website:

Licensing

Naked Objects MVC - Licenced Version Prices

Naked Objects MVC is licensed on a per developer basis. The price includes 12 months of upgrades.

Currency	Price per developer	Purchase
UK Pounds	£ 249*	Click to purchase
Euro	€ 299*	Click to purchase
US Dollars	\$ 399*	Click to purchase


* VAT will be added to the price shown for all sales to UK customers.

Clicking on the top-most Click to purchase link takes us to a page where we specify the number of licenses to purchase and customer details:

[Return to Naked Objects website](#)



Enter Order Details:


Product:	 Naked Objects MVC
Quantity:	<input type="text" value="1"/> *
Currency:	GBP
Unit Price:	249.00
Name:	<input type="text"/> *
Organisation:	<input type="text"/>
Address Line1:	<input type="text"/> *
Address Line2:	<input type="text"/>
Address Line3:	<input type="text"/>
Address Line4:	<input type="text"/>
Country:	<input type="text"/> ▼ *
Email:	<input type="text"/> *

Having entered at least all the mandatory fields and hitting save, we are presented with a confirmation screen:

[Return to Naked Objects website](#)



Confirm Order Details:

Product:	 Naked Objects MVC
Quantity:	1 UNITED KINGDOM
Email:	joe.smith@hmail.com

Pressing Confirm & Proceed to Payment will take you to on the PayPal website; Edit takes you back to the previous screen to amend details.

Let's now look at the implementation.

If we look at the HTML source for the first screen, we can see that the three Click to purchase links take the form:

```
<a href="http://salesnakdobjects.cloudapp.net/OrderEntry/
  CreateNewOrderEntry?productCode=00020&currency=GBP">Click to purchase<a>
```

The URL of the link gives away that the application is actually running in the cloud, under Microsoft's Azure. If we look at the `RegisterRoutes` method in the application's `Global.asax.cs` file ...

```

public static void RegisterRoutes(RouteCollection routes) {
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

    routes.MapRoute(
        "Product Details",
        "Product/Details",
        new {controller = "Generic", action = "Details"}
    );

    routes.MapRoute(
        "Country Details",
        "Country/Details",
        new {controller = "Generic", action = "Details"}
    );

    routes.MapRoute(
        "EditObject",
        "OrderEntry/EditObject",
        new {controller = "Generic", action = "EditObject"}
    );

    routes.MapRoute(
        "Edit",
        "OrderEntry/Edit",
        new {controller = "Generic", action = "Edit"}
    );

    routes.MapRoute(
        "Default",
        "{controller}/{action}",
        new {controller = "OrderEntry", action = "Index"}
    );
}

```

... we can see that the URL ...`OrderEntry/CreateNewOrderEntry?`... will only be matched by the last entry ("Default"), and given that both the controller and the action name are specified in the URL, this means that it will route directly to a `CreateNewOrderEntry` method on the `OrderEntryController`, which is reproduced below:

```

public class OrderEntryController : CustomController {

    #region Injected Services
    public IRepository OrderRepository { set; protected get; }
    #endregion

    [HttpGet]
    public ActionResult CreateNewOrderEntry(string productCode, string currency) {
        OrderEntry oe = OrderRepository.CreateNewOrderEntry(productCode, currency);
        return View("ObjectEdit", oe);
    }
    ...
}

```

Note the following:

- The `OrderEntryController` inherits from the Naked Objects `CustomController`. This is a convenience, not a requirement.

- Services can be injected into a controller, just as they are into domain objects.
- `CreateNewOrderEntry(string productCode, string currency)` picks up the two parameters from the URL.
- The method returns the `ObjectEdit` view of the newly created `OrderEntry` object, which we will now look at ...

```
[NotPersisted]
public class OrderEntry {

    [Disabled, MemberOrder(10)]
    public Product Product { get; set; }

    ...
}
```

We can see that the `OrderEntry` object is a regular domain class, but has been marked up with `[NotPersisted]`, indicating that this object will only ever be transient. (See the warning about Code First). Note that the user will still be presented with a Save button - but in this case it just takes the entered values into the object in memory, not the persisted store (database).

This is what is known in MVC terminology as a 'view model' - a domain object that exists solely for user interface purposes - it is not persisted.

This will be first presented to the user with the `Views > OrderEntry > ObjectEdit` view, shown here:

```
<%@ Page Title="" Language="C#"
MasterPageFile="~/Views/Shared/Site.WithoutServices.Master"
Inherits="System.Web.Mvc.ViewPage<LicenseManagement.OrderEntry>" %>
<%@ Import Namespace="NakedObjects.Web.Mvc.Html"%>
<%@ Import Namespace="LicenseManagement"%>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
    <%= Html.ObjectTitle(Model)%>
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <div class="ObjectEdit" id="<%= Html.ObjectTypeAsCssId(Model) %>">
        <%=Html.ValidationSummary(true, "Edit was unsuccessful. Please correct the
errors and try again.")%>
        <%=Html.UserMessages() %>
        <h2>Enter Order Details:</h2>
        <%
            using (Html.BeginForm("Edit",
                                Html.TypeName(Model).ToString(),
                                new {id = Html.GetObjectId(Model).ToString()},
                                FormMethod.Post,
                                new {@class = "Edit"}))%>
                <%=Html.PropertyListEditWithout(Model,"SubTotal", "VAT", "Total")%>
            </div>
        </asp:Content>
```

This is effectively just a standard object edit view, but with the `Html.PropertyListEditWithout` specifying that a number of properties are not to be shown at this stage (as they are calculated later).

Until you are familiar with the standard URL patterns generated by Naked Objects MVC, the best thing is to develop and run the application step by step, and view the HTML source in the browser. If we view the HTML source generated by the above view we can see that all the fields are contained within a form, which begins:

```
<form action="/OrderEntry/Edit?id=LicenseManagement.OrderEntry ... >
```

This tells us that the URL generated when the user hits Save, will be ...OrderEntry/Edit?... This is picked up by this entry in RegisterRoutes:

```
routes.MapRoute("Saving details",  
    "OrderEntry/Edit",  
    new {controller = "Generic", action = "Edit"} );
```

This directs the flow to the Edit action on the Naked Objects GenericController which we know from general operation will bring up the standard ObjectView - with the values taken into the object. If we provide our own view in (Views > OrderEntry > ObjectView) this will be picked up automatically:

```
<%@ Page Title="" Language="C#"
MasterPageFile="~/Views/Shared/Site.WithoutServices.Master"
Inherits="System.Web.Mvc.ViewPage<LicenseManagement.OrderEntry>" %>
<%@ Import Namespace="NakedObjects.Web.Mvc.Html"%>
<%@ Import Namespace="LicenseManagement"%>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
    <%= Html.ObjectTitle(Model)%>
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <div class="ObjectView" id="<%= Html.ObjectTypeAsCssId(Model) %>">
        <%= Html.UserMessages() %>
        <h2>Confirm Order Details:</h2>
        <% using (Html.BeginForm("EditObject",
                                Html.TypeName(Model).ToString(),
                                new {id = Html.GetObjectId(Model).ToString()},
                                FormMethod.Post,
                                new {@class = "Edit"})) {%>
            <%= Html.PropertyList(Model)%>
            <%= Html.PropertyListEditHidden(Model, true)%>
        <%}%>
        <!-- A separate form for the other action, containing all props but hidden-->
        <% using (Html.BeginForm("ConfirmOrder", "OrderEntry",
                                new {id = Html.GetObjectId(Model).ToString()},
                                FormMethod.Post,
                                new {@class = "Edit"})) {%>
            <%= Html.PropertyListEditHidden(Model, false)%>
            <button type="submit">Confirm & Proceed to Payment</button>
        <%}%>
    </div>
</asp:Content>
```

This view contains two parts. The first is a form that provides the user with the Edit button (to go back and make changes). All the visible fields are displayed by <%= Html.PropertyList (Model) %>. However, because this is a not persisted object, we need to

ensure that *all* the fields, including any hidden fields, are contained in the view (albeit some of them invisibly) - otherwise they will be lost. This is done with `<%:`

`Html.PropertyListEditHidden(Model, true)%>` and it is the `true` parameter here that causes the Edit button to be rendered.

The second half of the view provides the user with the Confirm & Proceed to Payment button. This must be implemented as a separate form, and again, inside that form we must include all the properties of the not persisted model object using `PropertyListEditHidden`, but this time with the `withEditButton` parameter set to `false`.

If the user selects this button, then (from the "Default" routing above) be directed to the `ConfirmOrder` action on the `OrderEntry` controller, which is shown below:

```
[HttpPost]
public ActionResult ConfirmOrder(FormCollection fc) {
    var oe = RecreateTransient<OrderEntry>(fc);
    var order = oe.Confirm();
    return Redirect(PaypalConstants.PaypalAddress + "?" +
        order.GeneratePayPalMessagePayload());
}
```

Because the `OrderEntry` object that we had created in the previous controller method was `[NotPersisted]` the session will have no record of it. We must therefore create a brand new instance of `OrderEntry` and copy all the data from the posted form into this new instance. `RecreateTransient<OrderEntry>(fc)` is just a shortcut, that saves us from having to write all that code. The method then goes on to call a `Confirm` method on that `OrderEntry`, the code for which is shown below:

```
public Order Confirm() {
    //Set up Customer
    var cust = Container.NewTransientInstance<Customer>();
    cust.Name = Name;
    // etc.

    //Set up Order
    var order = Container.NewTransientInstance<Order>();
    order.Customer = cust;
    // etc.
    Container.Persist(ref order);

    //Set up PayPal request
    OrderService.OrderViaPayPal(order);

    return order;
}
```

This method is copying data out of the not-persisted `OrderEntry` and into separate new `Customer` and `Order` objects. `Container.Persist(ref order)` will persist both the `Order` and the associated `Customer` object at the same time. Going back to the controller method above, we can see that it goes on to call a method `GeneratePayPalMessagePayload` on the (persisted) `Order`, and then uses this to re-direct the browser to the PayPal site to take payment.

MVC - Additional How-Tos

How to inject Services into Controllers and Views

Any [registered service](#) may be injected into a Controller and/or into a View.

Injecting into Controllers

Services are injected into Controllers in exactly the same way as services are injected into domain objects - use the injs snippet. However, the controller's constructor needs to take an `INakedObjectsFramework` as a parameter (the standard controllers provided with the framework - e.g. `GenericController` already have this) and you will need to add the following new line of code into that constructor.

```
public GenericController(INakedObjectsFramework nakedObjectContext) :  
base(nakedObjectContext) {  
    nakedObjectContext.ContainerInjector.InitDomainObject(this);  
}
```

The above line of code is not included in the standard code because it would add unnecessary overhead if you are not in fact requiring any domain services to be injected.

Injecting into Views

To make services available to your views via injection, first create a new master view. For Razor this means creating a sub-class of `WebViewPage<T>` (for `.aspx` it would be `ViewPage`). Add the following code into it:

```
public abstract class CustomWebPage<T> : WebViewPage<T> {  
    private INakedObjectsFramework framework;  
  
    [Dependency] // Indicates to Unity to inject  
    public INakedObjectsFramework Framework {  
        get { return framework; }  
        set {  
            // guard as otherwise InitDomainObject will also inject Framework  
            causing loop and stack overflow  
            if (framework == null) {  
                framework = value;  
                framework.ContainerInjector.InitDomainObject(this);  
            }  
        }  
    }  
}
```

You can then add in properties for each of the services (or container) that you need access to, just as for domain object code (use the 'injs' or 'injc' [code snippets](#)).

Then ensure that any views that need these service inherit from that new master. For Razor you can register it as the base class within Views > Web.config (*not* the main Web.config) thus:

```
<pages pageBaseType="NakedObjects.Mvc.App.Views.CustomWebPage">
```

How to encrypt hidden fields in a transient object

If you display a transient (not yet persisted) object that has some hidden properties (this is not a common scenario!) then those hidden properties will exist within the HTML, so that they can be posted back with the properties that the user has entered. (This is not the case for Hidden properties on persisted objects).

If this is a security concern (in other words if those hidden properties contain sensitive data) then it is possible to specify that any hidden properties within the HTML should be encrypted, using a pluggable encryption service. Naked Objects provides a default implementation: `NakedObjects.Web.Mvc.Helpers.SimpleEncryptDecrypt`. Alternatively, if you wish to use a specific form of encryption, you may create your own implementation of `NakedObjects.Web.Mvc.IEncryptDecrypt`. The service simply needs to be [registered](#).

How to disable server-side validation (Ajax)

See [Executed](#)

Other

Getting hold of the current user programmatically

If your domain model needs to get direct access to the user name of the current user, this may be obtained via the `Principal` method on the injected Container. The `Principal` method returns a `System.Security.Principal.IPrincipal`, from which you may obtain the user's identity and thence the name. This may then be used for example to retrieve a domain object such as an `Employee` that represents that user. For example, the following method returns the `Employee` object representing the logged on user:


```

public class EmployeeRepository {
    //(Add Injected Container)

    public Employee CurrentUser() {
        var userName = Container.Principal.Identity.Name;
        var query = from employee in Container.Instances<Employee>()
            where employee.UserName == userName
            select employee;

        return query.FirstOrDefault();
    }
}

```

Recommended pattern for accessing the current user as a domain object. The recommended pattern is to create an interface that defines a single templated method as follows:

```

public interface IUserService {
    T CurrentUser<T>();
}

```

And then to implement this interface in the repository for whichever type of domain object also corresponds to users - for example:

```

public class EmployeeRepository : IUserService {
    public T CurrentUser<T>() {
        if (! (typeof(T).IsAssignableFrom(typeof(Employee)))) {
            throw new DomainException("Cannot convert an Employee to type:" +
            typeof(T).ToString());
        }

        var userName = Container.Principal.Identity.Name;
        var query = from employee in Container.Instances<Employee>()
            where employee.UserName == userName
            select employee;

        return query.Cast<T>().FirstOrDefault();
    }
}

```

This allows other objects to have the `IUserService` injected, and to call `CurrentUser()` with a type `T` (for example: `ICommunicablePerson`) that they are interested in, without being coupled to `Employee` - provided that `Employee` implements or extends that type.

Creating an XML Snapshot of an object

Sometimes it can be very useful to create an XML Snapshot of a domain object, for example:

- For auditing purposes (to capture and store the complete state of an object at a given moment).
- To facilitate merging data from the model into a letter-template
- To exchange information with an external service that uses XML

To use this capability, you will need to [register](#) the `NakedObjects.Snapshot.XMLSnapshotService`.

This service can be injected into any domain object with a suitable property of type `NakedObjects.Snapshot.IXmlSnapshotService`, an interface which is defined in `NakedObjects.Helpers.dll` (installed into your model project as part of the `NakedObjects.ProgrammingModel` package). You can then call the `GenerateSnapshot` method on this service, passing in the domain object of interest. Having generated the snapshot, *but before reading the XML itself* (via its `Xml` property), you can also specify associated objects that should be in-lined within the snapshot - navigating as far down the graph of associated objects as you wish. All these ideas are illustrated in the code below, but we recommend that the best way to learn how this works is simply to experiment:

```
public class Product {  
  
    ...  
  
    //Injected service  
    public ProductRepository Snapshotter {set; protected get}  
  
    public void Archive() {  
        IXmlSnapshot = Snapshotter.GenerateSnapshot(this);  
        snap.Include("Manufacturer"); //In-lines an XML representation of the object in  
the Product's Manufacturer property  
        snap.Include("Manufacturer/Address"); //In-lines an XML representation of the  
object in the Manufacturer's Address property  
        string xml = snap.Xml;  
        ...  
    }  
}
```

`IXmlSnapshot` also provides methods to return the XML schema definition (`Xsd`), and to transform the generated XML using a standard XSL Transform passed in as a string (`TransformedXml`).

Creating and using a Restful Objects API

This section of the manual is designed to be read in conjunction with the Restful Objects specification, which may be downloaded from restfulobjects.org.

Restful Objects for .NET is an implementation of the Restful Objects specification for the Microsoft .NET platform.

It allows you to write a domain object model in any .NET language and then expose the complete functionality of the domain model via a set of 'restful' resources, conforming to the Restful Objects standard.

Restful Objects for .NET is built on top of the Naked Objects for .NET framework, and therefore follows the same, very simple, programming conventions for domain object models.

Adding a Restful Objects API to a Naked Objects MVC application

Any MVC project that has the NakedObjects.Mvc-FileTemplatesNuget package installed into it, will have the ability to provide a Restful Objects API alongside the MVC user interface. For security reasons the Restful Objects API is not enabled by default. To enable it you just need to [configure the RestRoot](#).

The Restful Objects API will then be available on the specified URL segment. For example (if you are prototyping), the home page of the MVC application might be:

<http://localhost:50326/System>

and the home resource for the Restful API on

<http://localhost:50326/myrestfulapi>

If you wish to have the Restful API on a different port altogether, then you can do this by adding a standalone Restful Objects API as described in the next section.

Creating a standalone Restful Objects API

If you do not want to have a Naked Objects MVC user interface for your application - just a Restful Objects API - then you may proceed as follows:

1. Create a new ASP.NET Web Application, specifying MVC as the template, and set this as the Startup Project for your solution. (We'll refer to this as the 'Run' project)
2. Find and install the NuGet package `NakedObjects.RestfulObjects.Server` into this project
3. Add project reference(s) from the Run project to your Model project(s).
4. [Register your services](#), and make any other changes you may require to the default run configuration.
5. [Configure the Entity Object Store](#).
6. Run the server project as a local app. This will launch a web-browser pointing at the 'home' resource for the application, running within `LocalHost`.

The home resource, as with all Restful Objects resources, will return a JSON representation. Microsoft Internet Explorer does not recognise JSON and so will offer to Save the returned representation as a file. We therefore recommend the use of the Chrome or Firefox browsers, with a suitable plug-in such as JSONView and/or REST Client.

Conformance to the Restful Objects specification

The current release of Restful Objects for .NET, implements the full mandatory requirements of the Restful Objects specification version 1.1.0. The optional capabilities defined in that version of the spec, are implemented as follows (this information is available via the `/version` resource in the API):

- **blobsClobs** - supported..
- **domainModel** - Both the simple and the formal scheme are implemented, including the 'selectable' option. See [How to specify the scheme for domain model type information](#).
- **protoPersistentObjects** - supported. See [How to work with proto-persistent objects](#).
- **validateOnly** – supported
- **inlinedMemberRepresentations** - supported.
- **deleteObjects** - Not supported. (This is a deliberate policy. If you want to delete objects, simply provide an action method for deleting, either on an object or a service).

Lists and Sets

Although the Naked Objects programming model will recognise both Lists and Sets as multiple associations (i.e anything that implements `IEnumerable<T>`), Restful Objects for .NET treats all collections as being Lists. In other words, adding an object to a collection via the Restful Objects API will always require a POST method, even if the underlying type is in fact a Set. Note that this is the safe option. If you attempt to add the same object twice to a Set, using a POST, the second attempt will effectively be ignored. In a future release we expect to be able to allow objects to be added to a Set using PUT, as per the Restful Objects specification.

Custom extensions

The Restful Objects specification permits the addition of custom extensions in defined places. Restful Objects for .NET adds the following custom extensions, where appropriate (all have the `x-ro-nof` prefix):

- `x-ro-nof-serviceType` takes one of the values of `menu`, `contributed`, or `system` depending on how the service was [registered](#).
- `x-ro-nof-renderInEditMode` - true or false
- `x-ro-nof-presentationHint` see [PresentationHint](#) attribute
- `x-ro-nof-mask` provides the string value specified using the [Mask](#) attribute.
- `x-ro-nof-choices` provides a map of name values for a enum property or action parameter

Restful Objects Server - additional how-to's

How to determine whether an action will require a GET, PUT or POST method

Restful Objects for .NET adopts the convention that any action that returns an `IQueryable` of a valid domain object type will be treated as a 'query only' action (side-effect free) method, and may therefore be invoked using the http GET method. (The programmer must therefore ensure that any actions that change the state of the system do not return an `IQueryable` result).

All other actions will, by default, require a POST method.

However, this default behaviour may be over-ridden by annotating the method (in the domain model) with a `QueryOnly` attribute (which will allow invocation via a GET) or `Idempotent` attribute (invocation via PUT). The application developer must take care to ensure that these attributes are applied correctly: they will impact the way that the action is rendered via the restful interface, *but do not enforce that the method's behaviour is consistent with the attribute*.

For further information about these distinctions, please refer to the Restful Objects specification.

How to work with proto-persistent objects

Restful Objects for .NET supports proto-persistent objects as defined in the Restful Objects spec - but these are referred to as 'transient objects' within the Naked Objects programming model. When persisting a transient object (via a POST to the `objects` resource), you will need to provide all of the properties of the object - including any that might be disabled or hidden.

For example, a method on the `Customer` object might return a new transient instance of an `Order`, having set the `Customer` property on the `Order` in order to associate it with the `Customer` when it is persisted. But this `Order.Customer` property will almost certainly be disabled and might even be hidden, since the user typically does not need to be reminded of it.

This is usually straightforward to handle, since the values of any properties that were set up programmatically before the transient object was returned from the server will be included in the arguments map associated with the `persist` link.

Where it becomes slightly tricky is if the transient object has `Disabled` or `Hidden` properties that are not always populated. (This is, admittedly, a fairly obscure use-case). In this circumstance the Restful Objects server may return an error, indicating that values have not been supplied for mandatory properties. Normally, a property that is `Disabled` or `Hidden` would be ignored when checking for mandatory properties, but in the case of transient objects it is not. If this situation arises, then the solution is to also mark those properties explicitly with the `Optionally` attribute.

How to specify the scheme for domain model type information

The Restful Objects specification defines two approaches to the supply of domain type information - the 'simple scheme' and the 'formal scheme'. Implementations may support none, either or both of these, with the option for the client to select between them. Restful Objects for .NET supports both schemes and unless the client request specifies which is required, both forms of domain type metadata will be included in every response. This provides maximum flexibility, but also increases the size of the returned representations; it also makes them harder for a human to read when debugging for example. A further issue is that the formal scheme may reveal information about the domain model that is considered sensitive. For example while the visibility of the `Customer.EstimatedNetWorth` property may be restricted using authorization, the fact that such a property exists within the system is visible via the domain type representation under the formal scheme. By contrast, under the simple scheme, if the property was hidden from a user, then the existence of such a property would also be fully hidden.

For all these reasons, Restful Objects for .NET allows the developer to restrict the options available on a particular deployment. This is done by adding the following line of code into the `RegisterRoutes` method on the `NakedObjectsStart` class in your run project:

```
RestfulObjectsController.DomainModel = RestControlFlags.DomainModelType.None;
```

where `None` may be replaced by any of the values defined in the `DomainModelType` enumeration: `None`, `Simple`, `Formal`, `Selectable`.

How to enforce concurrency-checking

To switch on concurrency checking, add the following line of code into the `RegisterRoutes` method on the `NakedObjectsStart` class in your run project.

```
RestfulObjectsController.ConcurrencyChecking = true;
```

With this capability switched on, the client will need to provide the `if-match` header information, as defined in the Restful Objects specification. And if your application is using the Entity Framework for persistence, you will need to use the [ConcurrencyCheck](#) attribute.

In order for the RESTful API generated by the server to be compliant with the Restful Objects specification, concurrency checking must be switched on. The only reason that it has been made switchable in the implementation, is because it can be burdensome in the early stages of application development, because clients must provide the `If-Match` header.

How to make an application Read Only

If your application is intended as read-only, add the following line of code into the `RegisterRoutes` method on the `RestfulObjectsStart` class in your run project.

```
RestfulObjectsController.IsReadOnly = true;
```

The effect of this will be to disallow all but GET methods. Links to any PUT, POST or DELETE methods may still be shown, but any attempt to invoke them will result in an error.

How to implement automatic redirection for specific objects

If a domain object implements the interface `NakedObjects.Redirect.IRedirectedObject` (defined in the `NakedObjects.Helpers.dll`) then when an attempt is made to retrieve the object (either directly via a link to the object resource, or as an action result), Restful Objects for .NET will return an http '301 Re-direct' to an alternative Restful Objects object resource that may even be on another server. That url for that alternative object resource is constructed from the values of the `ServerName` and `Oid` properties that the implementation of `IRedirectedObject` requires, as follows:

```
http://{ServerName}/object/{Oid}
```

This pattern is particularly effective as a mechanism for integrating multiple existing systems. In the following example, the `Order` object may be thought of as existing in a 'new' system and the `Customer` in on another ('old') system. The `Customer` class on the new system is in fact merely a 'stub' - the class is persisted on the new system (which is why it has its own `Id` property), but the persisted values for `ServerName` and `Oid` refer to a specific instance of a `Customer` object on the old system. The (optional) `Title` property contains information that is duplicated from the data in the old system - to reduce the need for the user to navigate the link in order to see the identity of the customer. (Ideally, this title will be static, identity, information: if it were subject to change then it would be necessary to add a mechanism to enforce consistency).

```
//Stub object
public class Customer : IRedirectedObject {

    public virtual int Id { get; set; }

    [Title]
    public virtual string Title { get; set; }

    #region Implementation of IRedirect

    public string ServerName { get; set; }

    public string Oid { get; set; }

    #endregion
}
```

How to change cache settings

Based on the three 'shorthand' types of response defined in the Restful Objects specification, the default cache settings for the Restful Objects for .NET implementation are as follows,

- Transactional representations: 0 seconds (i.e. no caching)
- User representations: 3600 seconds (1 hour)
- Non-expiring representations: 86400 seconds (1 day)

These may be over-riden by adding the following line into the

`RestfulObjectsStart#RegisterRoutes` method, and changing the three integer values as required:

```
RestfulObjectsControllerBase.CacheSettings = new Tuple<int, int, int>(0, 3600, 86400);
```

How to handle Complex Types

Restful Objects for .NET does not currently support Complex Types. If a domain object does have a property that is a Complex Type, then this will be rendered the same way as a regular reference property (i.e. an association), complete with the title of the complex type object. However, the url in the link just points to the parent object (i.e. the object you are currently viewing). There is currently no way to view or edit the properties of the complex type object directly.

If you need to use a Complex Type (perhaps because the same domain model is used by another application running Naked Objects MVC, for example) and you need to view and/or edit the properties of that complex type (other than just viewing the title) then you will need to add one or more dedicated actions, as illustrated with the following example.


```

public class Customer {
    //This property is a complex type, see below
    [Disabled]
    public Address HomeAddress { get; set; }

    public void EditAddress(string line1, string line2) {
        HomeAddress.Line1 = line1;
        HomeAddress.Line2 = line2;
    }

    public string Default0EditAddress() {
        return HomeAddress.Line1;
    }

    public string Default1EditAddress() {
        return HomeAddress.Line2;
    }
}

[Inline] //Complex Type
public class Address {
    //Title (would probably truncate each line)
    public string ToString() {
        return Line1 + Line2;
    }
    public string Line1 { get; set; }
    public string Line2 { get; set; }
    //etc
}

```

How to change the format of the Object Identifier (Oid) in resource URLs

Many of the resources (URLs) defined in the Restful Objects specification include an 'object identifier' in the form: {DType}/{IID}, where {DType} is the *domain type identifier* and {IID} is the *instance identifier*; taken together these are referred to as the *object identifier* or *Oid*. By default, the Restful Objects for .NET server renders the domain type identifier as the fully-qualified class-name, and the instance identifier as the key of the object or, if it has a compound key, as the keys separated by dashes e.g.:

MyApp.Customers.WholesaleCustomer/10876

MyApp.Sales.OrderLine/8566-1055

You can override the format of either or both parts of the Oid. This might be for readability, for example:

WholesaleCustomer/10876

ORDLIN/8566---1055

or it might be to encrypt one or both parts, to prevent a rogue user from guessing the URL of an object they could not otherwise retrieve (though note that this could also be prevented through the Naked Objects custom authorization):

MyApp.Customers.WholesaleCustomer/f56bk-xx803h-jk4788ggweq2

Yet another reason would be if a domain object has one or more string keys that contain the default key separator. For example, if you have a natural (single) key of `sku-10452`, then the Restful Objects Server would, by default, interpret that as a two-part key and be unable to retrieve the object.

Control over the format is managed by creating an implementation of one or both of the following service definitions, each of which defines just two simple methods for converting each way:

- `NakedObjects.ITypeCodeMapper` - to take control over the format of the *domain type identifier*.
- `NakedObjects.IKeyCodeMapper` - to take control over the format of the *instance identifier*.

Your implementation(s) of these service interfaces should then be [registered](#).

How to limit the scope of the domain model that is visible through the Restful API

By default, Restful Objects for .NET will create a Restful API to the whole of your domain model. (Strictly speaking, it is to all the parts of your domain model that can be reached directly, or indirectly, from the run project). If you wish to restrict the Restful API to a subsection of the domain model, then there are several different techniques available to you:

Mark up certain classes, properties, or methods with a `Hidden` attribute. This is best used when you only wish to exclude small parts of the model from the Restful API. However, this approach will also hide the same parts of the model from any Naked Objects MVC user interface - which might not be your intent. In which case consider the other patterns:

- Use custom authorization to control visibility of classes, properties and actions to certain users or groups of users. Note that this approach could also be used to hide sections of the model from all users - without necessarily impacting any Naked Objects MVC user interface - because the two 'interfaces' can be generated by two different run projects, each of which can register a different custom authorizer.
- Make use of the [ITypeCodeMapper](#). The primary intent of these interfaces was to allow control over the format of the object `_identifier` in resource URLs. Note, however, that they could also be used simply to ensure that large sections of the domain model, or possibly certain ranges of object Ids, can never be mapped as resource URLs in the Restful API. For example, the following code would ensure that nothing in the `MyApp.Payments` or `MyApp.Employees` namespaces of the domain model is mapped:

```

public class CustomRestrictiveTypeCodeMapper : ITypeCodeMapper {

    public Type TypeFromCode(string code) {
        return IsExcluded(code) ? null : TypeUtils.GetType(code);
    }

    public string CodeFromType(Type type) {
        string typeName = TypeUtils.GetProxiedTypeFullName(type);
        return IsExcluded(typeName) ? null : typeName;
    }

    private bool IsExcluded(string typeName) {
        return typeName.ToUpper().StartsWith("MYAPP.PAYMENTS") ||
        typeName.ToUpper().StartsWith("MYAPP.EMPLOYEES");
    }
}

```

The use of the two methods `TypeUtils.GetType` and `TypeUtils.GetProxiedTypeFullName`. These are the safest ways to convert between Types and strings in Naked Objects. Calling `foo.GetType()` on a persistent object will return the Entity Framework proxied type, not the raw domain type - using `TypeUtils.GetProxiedTypeFullName` will get you the raw domain type.

Returning null from each of the methods when the intended type is to be excluded is sufficient to ensure that that type can never be accessed via the Restful API. A similar approach could be used with an implementation of `IKeyCodeMapper` to exclude certain ranges of *instance Ids* if desired.

How to switch off strict Accept header enforcement

By default, the Restful Objects for .NET server strictly enforces Accept headers. Unless the request includes, at minimum, `application/json` (or a broader match such as `*/*`) then a 406 error will be returned.

Sometimes, during the early stages of development this can be a little inconvenient, especially if you are using a client tool that does not allow Accept headers to be set up by default. In these circumstances it is possible to switch off strict Accept header enforcement, by adding the following code into the `RegisterRoutes` method on the `RestfulObjectsStart` class:

```
RestfulObjectsControllerBase.AcceptHeaderStrict = false;
```

With strict Accept header enforcement switched off, the generated API is no longer strictly conforming to the Restful Objects spec. You should therefore ensure that strict enforcement is switched back on again before you deploy the live application.

How to limit the size of returned collections

The Restful Objects 1.0 specification does not provide explicit support for paging through returned collections. A sketch of a possible approach to paging is included in Section 34.3 of the 1.0 specification under 'Ideas for Extensions to the Specification' - and something like this is likely to be added to a future release.

Currently the Restful Objects for .NET server will only return the first 'page' (meaning the first 20 elements) from any collection. This is to avoid the risk of accidentally returning thousands of elements. If the returned collection has more elements, only the first 20 will be shown, but the returned header will contain a warning message (code 299) that this has occurred.

You can override the default page size of 20 in the `RegisterRoutes` method on the `RestfulObjectsStart` class in your run project, for example:

```
RestfulObjectsControllerBase.DefaultPageSize = 50;
```

A value of 0 means 'unlimited page size' - query methods will return all elements in the collection. This should be used with caution.

Pending the introduction of framework-level support for paging, you can implement paging within your own query methods, for example:

```
public IQueryable<Order> Orders(int pageNumber){
    int pageSize = 50;
    int skip = pageSize*pageNumber;
    return Container.Instances<Order>().Skip(skip).Take(pageSize);
}
```

Note that for the above code to work, you should ensure that the `DefaultPageSize` is at least as large as the page size used within your query methods.

How to enable cross-origin resource sharing (CORS)

If your Restful Objects client is browser-based and is not loaded from the same location as the Restful Objects Server, then you are probably going to run into the 'same origin policy' issue. The easiest way to proceed is to enable cross-origin resource sharing (CORS). CORS can also be useful if you want to create a client that will integrate functionality from multiple Restful Objects servers.

You can implement CORS for yourself, but a much easier solution is to use the `Thinkecture.IdentityModel` NuGet package.

1. Add `Thinkecture.IdentityModel` to the run project through the NuGet Package Manager.
2. Add a `CorsConfig.cs` file under `App_Start`:

```

using System.Web.Http;
using Thinkteckture.IdentityModel.Http.Cors.WebApi;

public class CorsConfig {
    public static void RegisterCors(HttpConfiguration httpConfig) {
        var corsConfig = new WebApiCorsConfiguration();

        // this adds the CorsMessageHandler to the HttpConfiguration's
        // MessageHandlers collection

        corsConfig.RegisterGlobal(httpConfig);

        // this allows all CORS requests to the RestfulObjects controller
        // from the http://foo.com origin.

        corsConfig.ForResources("RestfulObjects").ForOrigins("http://foo.com").AllowAll();
    }
}

```

3. Hook in the config in RestfulObjectsStart.PostStart

```

public static void PostStart() {
    // existing code
    .....

    // add CORS
    CorsConfig.RegisterCors(GlobalConfiguration.Configuration);
}

```

How to work with addressable View Models

Section 2.2 of the Restful Objects specification (v 1.0) introduces the idea of addressable View Models, and section 32 provides a code sketch as to how this might be supported. The Restful Objects Server (as of v 1.1) does support this pattern using an approach broadly similar to that code sketch.

See [View Model](#).

Authorization in Restful Objects

The Restful Objects Server honours the same approaches to authorization as Naked Objects MVC - including attribute authorization and, custom authorization. However, you will need to provide your own mechanism for determining the user and roles based on the approach to authentication that you wish to use on your application.

By way of illustration only, the Restful Objects Server package installs a `BasicAuthenticationHandler` (in the `App_Start` folder). In order to use this, you must register it

within the `PostStart()` method in `RestfulObjectsStart` (also to be found in the `App_Start` folder), as shown here:

```
public static void PostStart() {  
    ...  
    GlobalConfiguration.Configuration.MessageHandlers.Add(new  
    BasicAuthenticationHandler());  
}
```

This `BasicAuthenticationHandler` checks the incoming request to see if it has any credentials. If not, it raises a simple `LogIn` request back to the client.

The `BasicAuthenticationHandler` also contains the following method:

```
private UserCredentials Validate(string user, string password) {  
    //THIS CODE IS A MOCK IMPLEMENTATION FOR ILLUSTRATION ONLY  
    //Fail if either the user name or password is null or empty  
    if (string.IsNullOrEmpty(user)) return null;  
    if (string.IsNullOrEmpty(password)) return null;  
    //Otherwise, authenticate any user/password combination, with a standard set of  
    roles  
    return new UserCredentials(user, password, new List<string> { "Role1", "Role2" });  
}
```

As the comments make clear, this basic code does not validate the user name or password; a real implementation would validate these against a database or, more likely, delegate to some authorization service. The last line of the code associates two default roles with the user - again this could be delegated to an external service.

Meantime, for exploration and testing purposes only, you can modify the last line (or any of the code in this method) to ensure that you are working with a user that has the correct roles assigned.

Running without a user interface

It is possible to run Naked Objects without any user interface, for example to implement batch operations, or a custom publish & subscribe mechanism.

Running Naked Objects as a .exe

Follow these steps

1. Add into your solution a new 'Windows Application' project. The simplest way to do this is to add a project of type 'Console Application' and then in the project's Application Properties, set the Output Type to Windows Application (as you don't typically want the console to appear).
2. Delete any existing `Program.cs` file, so that it can be replaced by the next step ...
3. Using the NuGet Package Manager, add the `NakedObjects.Batch` package . This will add a new `Program.cs` file:

```
public class Program {
    public static void Main() {
        UnityActivator.Start();

        UnityConfig.GetConfiguredContainer().Resolve<IBatchRunner>().Run(new
BatchStartPoint());

        UnityActivator.Shutdown();
    }
}
```

4. [Configure the application](#) in the `NakedObjectsRunSettings` class.
5. The package install will also have added a class `BatchStartPoint`; add your code into the (empty) `Execute` method. Alternatively, you may provide any class that implements `NakedObjects.Boot.IBatchStartPoint`, and instantiate that class within the `Run` method, above. Note that either `BatchStartPoint`, or your own implementation of `IBatchStartPoint` can have the `IDomainObjectContainer` and/or any registered domain services injected into it - to give you access to the domain model from this start point.

You may choose to leave this `.exe` running permanently on your server, or to launch it at regular intervals via the Windows Task Manager.

If your batch implementation can be broken down in a series of separate tasks, then, rather than performing them sequentially, you should consider using the Naked Objects Async capability...

Running multiple threads asynchronously

.NET 4 introduced a new async capability. Naked Objects now leverages this to provide a convenient mechanism for running multiple Naked Objects threads in parallel. To use this capability, you will need to inject a `NakedObjects.Async.IAsyncService` (defined within the `NakedObjects.ProgrammingModel` package) into your code:

```
public IAsyncService AsyncService { set; protected get; }
```

(The Naked Objects framework provides an implementation of `IAsyncService` interface (as it does for `IDomainObjectContainer`) `NakedObjects.Async.AsyncService`. You will need to [register this service](#) (or another implementation if you wish to write your own).

Within your code, you can call the `RunAsync` method on this service, passing in the action to be run as a lambda. This method returns a `System.Threading.Tasks.Task`; by collecting these tasks into an array you can then instruct the system to wait until all such tasks are completed, using the standard .NET code of `Task.WaitAll(tasks)`, as illustrated in the following example code:

```
public void RunAllProcessesDueBy(DateTime dateTime) {  
    IList<BatchProcessDefinition> due =  
    BatchRepository.ListProcessesWithNextRunsDueBy(dateTime);  
    var tasks = due.Select(bpd => AsyncService.RunAsync(() =>  
    BatchRepository.FindAndRunProcessDefinition(bpd.Id))).ToArray();  
    Task.WaitAll(tasks);  
}
```

In the example above, the `BatchRepository` and `BatchProcessDefinition` are straightforward domain classes. The first line just returns a set of objects that contain details of tasks due to be run by the specified `DateTime`. The second line invokes a method on `BatchRepository` to find and run each of those due processes, asynchronously. The third line waits until all the asynchronous processes are completed.

Each of the asynchronously-run processes is fully independent. You should not attempt to pass *domain objects* into or between these processes, but you may pass .NET primitives, including object Ids - to be retrieved independently inside the new thread. This is why, in the above example, `FindAndRunProcessDefinition` is passed in the Id of the `BusinessProcessDefinition`, not the domain object itself.

Note: The Naked Objects `AsyncService` is designed only to be used within a standalone executable run project (as illustrated above). It will not work within an `HttpContext` (and therefore within an MVC run project).

Authorization

Authorization, means the ability to control what an individual user can see and do within an application, based upon their identity, the role(s) assigned to them, and/or other credentials or 'claims'. (It follows that authentication is a necessary precursor to authorization.) Naked Objects supports 'fine-grained authorization', meaning that it is possible to specify whether or not a user may View - and, separately, whether they may Edit - individual properties on object types, and whether or not they may invoke individual actions on objects or services.

Currently, there are three choices for authorization:

- **Attribute-based Authorization.** This is the default mechanism for authorization. Simply mark up any properties or action methods with attributes that specify the Roles (or users) to whom they should be available. This approach is best suited to applications where you have a relatively simple domain model and/or a small number of distinct user roles. Attribute-based authorization may be used with any of the three approaches to authentication listed above.
- **Custom authorization.** In this approach you plug-in your own mechanisms for managing authorization, which may be self-contained or may delegate responsibility to an external service. Using this approach it is possible to control authorization at the level of individual object instances.

Attribute-based Authorization

In this approach, authorization rules are implemented by applying attributes to properties and/or actions inside the domain model classes. If any property or action does not have an authorization attribute specified then that property/action is assumed to be available to all users.

Consider the following example:

```
Imports NakedObjects.Security

Public Class Customer

    <AuthorizeAction(Users:="fred", Roles:="CustomerService, Manager")>
    Public Function PlaceNewOrder() As Order
        ...
    End Function
    ...
End Class
using NakedObjects.Security;

public class Customer {

    [AuthorizeAction(Users="fred", Roles="CustomerService, Manager")]
    public Order PlaceNewOrder() {...}
    ...
}
```

Here the user action Place New Order will only appear on the menu of users who have either the role `CustomerService`, or `Manager`, plus the user named `Fred` who will see the action irrespective of his role(s).

Both the syntax and the semantics of `NakedObjects.Security.AuthorizeAction` attribute are similar to those of the `System.Web.Mvc.Authorize` attribute - so why not just use the latter? In part this is because we do not want domain models to become dependent upon ASP.NET MVC - or any other specific user interface (the `NakedObjects.Security.AuthorizeAction` is contained within the tiny `NakedObjects.Attributes` assembly with the other Naked Objects attributes). The other reason is that the `System.Web.Mvc.Authorize` attribute would not allow the separate specification of View and Edit rights for properties, as described below.

For properties, we use the `NakedObjects.Security.AuthorizeProperty` attribute. This allows us to specify roles and/or users that may View the property, and, separately, roles and/or users that may Edit the property. Consider the following example:

```
<AuthorizeProperty(ViewRoles:="CustomerService, Finance", EditRoles:="Finance")>
Public Overridable Property CreditRating() As Decimal
    [AuthorizeProperty(ViewRoles="CustomerService, Finance", EditRoles="Finance")]
    public virtual decimal CreditRating {get; set;}
```

Here, the Credit Rating property may be viewed by any user with the role `CustomerService` or `Finance`, but may only be edited by a user with the role `Finance`. (Individual users may also be authorized, by adding `ViewUsers` and `EditUsers` parameters to this attribute).

A user or role will not be able to Edit a property unless they are also authorized to View that property - hence the appearance of `Finance` in both the `ViewRoles` and `EditRoles` above.

The `AuthorizeProperty` and/or the `AuthorizeAction` attribute, may also be applied at class level:

```
<AuthorizeAction(Roles="Manager")>
<AuthorizeProperty(ViewRoles="CustomerService, Manager", EditRoles="CustomerService,
Manager")>
Public Class Customer
    ...
End Class
[AuthorizeAction(Roles="Manager")]
[AuthorizeProperty(ViewRoles="CustomerService, Manager", EditRoles="CustomerService,
Manager")]
public class Customer {...}
```

This then applies the authorization rules to all properties and/or actions within that class. The class-level attributes will over-ride any such attributes applied to individual properties or actions within that class, including properties/actions specified on sub-classes. Attempting to mix the two approaches is not recommended. On initialisation of the system, if such an over-ride is detected, a system warning will be raised.

Custom Authorization

Custom authorization is specified using one or more authorizer classes, each which implements *either* `ITypeAuthorizer<T>`, where `T` is a concrete type from your domain model, or `INamespaceAuthorizer` (to provide authorization logic for any types within a specified namespace).

Each authorizer class must be [registered](#).

When the framework needs to determine authorization for a given object, it will search for the authorizer that provides the most specific match to that type object; if no such authorizer is found, it will use the default authorizer, which by implementing `ITypeAuthorizer<object>` will be able to work with domain objects of all types. Thus, you will typically only add a type authorizer for types (or groups of types within a namespace) that require their own specific approach to authorization.

The code below shows a skeletal implementation of the `FooAuthorizer`:

```
public class FooAuthorizer : ITypeAuthorizer<Foo> {
    public bool IsEditable(IPrincipal principal, Foo target, string memberName) {...}
    public bool IsVisible(IPrincipal principal, Foo target, string memberName) {...}
}
```

- `IsVisible` is called to determine whether a given object-member (a property or an action method) should be visible to the current user. The user-name may be derived from the `principal` parameter; the specific object instance is passed in as the `target` parameter (this is needed only if you are writing 'instance-based authorization' logic); the third parameter gives the `memberName`.
- `IsEditable` is called to determine whether a given object property may be edited by the user. `IsEditable` has no determined meaning for action members.

Within the `IsVisible` and `IsEditable` methods, you can either write custom logic, or delegate out to some external service.

There is a useful method in `TypeUtils` (within the `NakedObjects.Helpers` assembly) that facilitates type-safe testing of a property name, for example:

```
TypeUtils.IsPropertyMatch<Customer, DateTime>(target, memberName, x => x.DateOfBirth).
```

This is especially useful within an implementation of `INamespaceAuthorizer` as the method checks both the type of the target and the match for the `memberName`.

An implementation of `ITypeAuthorizer<T>` or `INamespaceAuthorizer` can have domain services and/or the `IDomainObjectContainer` injected into it.

Controlling visibility of columns in a table view

In a table view the authorizer will be called for each object instance being rendered in the table. If a given property is not visible for any of the instances (rows) in the table then that property will not be rendered as a column in the table. If the property is visible for any instance(s) being rendered then the column will be shown, but the value within it rendered only for those instances that authorize its visibility.

Auditing

Naked Objects provides a general purpose mechanism for recording user actions, either to support formal auditing, or just to provide a user-accessible mechanism for identify who did what. In order to use the audit capability, you need to create one or more ‘audit services’ and then [register](#) them.

Each time the user persists or updates an object, or invokes an action (on a service, or on a persisted object), the framework will look for the auditor that most precisely fits type of the object or service on which the action/update/persist has been invoked, and call the appropriate method, as defined on `IAuditor`:

```
void ActionInvoked(IPrincipal byPrincipal, string actionName, object onObject, bool queryOnly, object[] withParameters);

void ActionInvoked(IPrincipal byPrincipal, string actionName, string serviceName, bool queryOnly, object[] withParameters);

void ObjectUpdated(IPrincipal byPrincipal, object updatedObject);

void ObjectPersisted(IPrincipal byPrincipal, object persistedObject);
```

Note that the `queryOnly` parameter will be set to true if the action can be determined by the framework to be 'query only' i.e. any action that returns an `IQueryable<T>` and/or that is marked up with the `QueryOnly` attribute.

The implementations of `IAuditor` may manipulate the provided details of the user action and then optionally persist them as special ‘audit record’ domain objects. The persisted audit records may then be made available to suitably-authorized users via, say, an `AuditRepository` service.

Note that any implementation of `IAuditor` can have domain services and/or the `IDomainObjectContainer` injected into it.

Profiling

If you want to profile your application, you can hook into various events generated by the framework, and then write out to a console, file, or logging framework. The pattern is somewhat similar to the patterns for [Authorization](#) and [Auditing](#). Start by writing your own implementation of `NakedObjects.Profile.IProfiler`, for example:

```
public class MyProfiler : IProfiler {  
  
    public void Begin(IPrincipal principal, ProfileEvent profileEvent,  
                     Type onType, string memberName) {  
        Debug.WriteLine(profileEvent.ToString() + " Start:" + DateTime.Now);  
    }  
  
    public void End(IPrincipal principal, ProfileEvent profileEvent,  
                   Type onType, string memberName) {  
        Debug.WriteLine(profileEvent.ToString() + " End:" + DateTime.Now);  
    }  
}
```

`ProfileEvent` is an enum that defines the set of Naked Objects events that can be profiled:

```
public enum ProfileEvent {  
    ActionInvocation,  
    PropertySet,  
    Created,  
    Deleted,  
    Deleting,  
    Loaded,  
    Loading,  
    Persisted,  
    Persisting,  
    Updated,  
    Updating,  
}
```

Next you need to specify which of these events you wish to profile as a `set`, for example:

```
new HashSet<ProfileEvent>() { ProfileEvent.ActionInvocation, ProfileEvent.Updating,  
                             ProfileEvent.Updated };
```

Note that in addition to `ActionInvocation`, there are events that correspond to the Life Cycle methods for a domain object. Note: *you do not have to have any of those methods explicitly on your domain object - the events are generated whether or not you need to add behaviour into them via explicit methods.*

When profiling a Naked Objects MVC application you would not normally use the `PropertySet` event, as this profiles only the time taken on domain code behaviour associated with an individual property set - not the total time to update the object. If you wish to profile the time taken from when a use hits Save on an object-edit view, then you can monitor from the start of `Updating` to the end of `Updated`.

Finally, you must [configure the profiling](#) in `UnityConfig`.

Testing

Executable Application Tests (XATs)

Executable Application Tests (XATs for short) allow you to test your application's functionality from the perspective of a user. XATs test complete user scenarios. In effect they execute a sequence of user actions, and they provide specific methods for testing what a user is allowed to see or do at any point within a scenario.

The XAT framework is fully compatible with the Microsoft test framework.

Create an XAT project as follows:

1. Create a new MS Test project.
2. Install the `NakedObjects.Xat` NuGet package into this project

Creating an XAT test class

The easiest way to create an XAT class like this is to use this template:

Add New Item > Visual C# Items > Naked Objects > XAT
--

which is installed when you install the [NakedObjects.Ide](#) NuGet package into your solution.

```

[TestClass()]
public class ExampleXAT : AcceptanceTestCase {

    [TestInitialize()]
    public void Initalize() {
        InitializeNakedObjectsFramework(this);
        StartTest();
    }

    protected override Type[] Services {
        get {
            return new Type[] {
                typeof(EmployeeRepository),
                typeof(ClaimRepository),
                typeof(RecordedActionRepository),
                typeof(RecordActionService),
                typeof(RecordedActionContributedActions),
                typeof(DummyMailSender); }
        }
    }
}

```

Writing tests

The following code shows an example test. This method is testing functionality to create a new Expenses Claim in (an imaginary) Expenses Processing application:


```

[TestMethod()]
public virtual void ExampleTestMethod() {
    ...
    SetUser("sven");
    ITestObject claim = GetTestService("Claims").GetAction("Create New
Claim").InvokeReturnObject("test");
    claim.GetPropertyByName("Description").AssertIsVisible().AssertIsModifiable();
    claim.GetPropertyByName("Date Created").AssertIsVisible().AssertIsUnmodifiable();
    ...

    claim.GetAction("Create New Expense
Item").AssertIsVisible().AssertIsInvalidWithParms(null);
    claim.GetAction("Copy An Existing Expense
Item").AssertIsVisible().AssertIsInvalidWithParms(null);
    claim.GetAction("Copy All Expense Items From Another Claim").AssertIsVisible();
    ...

    claim.GetAction("Approve Items").AssertIsVisible();
    ...
    claim.GetAction("Return To
Claimant").AssertIsVisible().AssertIsInvalidWithParms("");

    ITestParameter[] tps = claim.GetAction("Copy All Expense Items From Another
Claim").Parameters;

    tps[0].AssertIsMandatory().AssertIsNamed("Claim or Template");
    tps[1].AssertIsOptional().AssertIsNamed("New date to apply to all items");

    tps = claim.GetAction("Create New Claim From This").Parameters;

    tps[0].AssertIsMandatory().AssertIsNamed("Description").AssertIsDescribedAs("");
    tps[1].AssertIsOptional().AssertIsNamed("New date to apply to all
items").AssertIsDescribedAs("");

    ITestNaked[] choices = claim.GetAction("Create New Expense
Item").Parameters[0].GetChoices();

    Assert.IsTrue(choices.Length == 8, "Wrong number of choices in choice array");

    ((ITestObject)(choices[0])).AssertIsImmutable().AssertIsDescribedAs("");

    claim.GetAction("Submit").AssertIsInvalidWithParms(null, true);
}

```

Having specified that the test is to be run as though by the user named "sven" (because this particular application makes use of the user name to retrieve an Employee object corresponding to that name) the next line of the test may be read as follows:

On the service named Claims, invoke the action named Create New Claim with test as the single (string) parameter.

This action is expected to produce a (new) `Claim` object, which we store in a variable `claim`. But why is that variable specified as being of type `ITestObject` rather than of type `Claim`? This `ITestObject` can be thought of as a wrapper that holds the actual `Claim` object inside it. (Actually something very similar happens within the Naked Objects user interface, but this is transparent to both the user and the application programmer).

The `ITestObject` wrapper provides methods that allow you to get hold of the underlying objects properties and actions, for example:

```
claim.GetPropertyByName("Description")...  
claim.GetAction("Create New Expense Item")...
```

which return an `ITestProperty` and an `ITestAction` respectively. These in turn provide methods for making assertions about the attributes or behaviour of those properties or actions from the perspective of the user, such as `AssertIsVisible` and `AssertIsMandatory`. For actions, you can drill down further and get to the individual parameters (as `ITestParameter`) for that action and make assertions about their attributes and behaviour.

Using the C# 6 nameof operator to create ‘refactor-safe’ XATs

If you are using C# 6, then it is possible to make use of the new `nameof()` operator, by means of new methods added (as of NOF 7.1). This means that your tests can be ‘refactor-safe’. For example:

```
claim.GetPropertyById(nameof(Claim.Description))...  
claim.GetActionById(nameof(Claim.CreateNewExpenseItem))...
```

also:

```
GetTestService<ClaimRepository>()...  
laim.GetActionById(nameof(Claim.CreateNewExpenseItem))  
    .AssertHasFriendlyName("Create New Expense Item");
```

Simulating users and roles

The `AcceptanceTestCase` has a `SetUser` method that allows you to specify the name of the user and optionally the roles for the simulated user for the test. This allows you to test the visibility of actions and properties, and the modifiability of properties for a given user and/or roles, for example:

```

[TestMethod]
public void AuthorizedForEditAndView() {
    SetUser("Bob");
    prop1.AssertIsVisible();
    prop1.AssertIsModifiable();
}

[TestMethod]
public void AccessByAnonUserWithViewRole() {
    SetUser("Anon", "sysAdmin");
    prop1.AssertIsVisible();
    prop1.AssertIsUnmodifiable();
}

```

The form of authorization tested will depend upon how you have configured the test class. By default the test will assume use of attribute authorization. Or you may override the `Authorizer` property on the test class to specify, for example, custom authorization.

Running tests against a database

Resetting the database

The Naked Objects XAT framework provides a class `NakedObjects.Xat.Database.DatabaseUtils`, which provides methods for restoring the database to a known condition - typically a snapshot or a backup. (These utility methods make use of Microsoft's SQL Server Management Objects (SMO) framework.) You can make the snapshot/backup manually before running the tests, or create it automatically within the tests.

Backing-up or restoring a database can be a very time-expensive operation, so you should use this functionality with care. For example, it would be theoretically possible to restore the database for each individual test (within the `Initialise` method), which would guarantee that tests never trample on each other in the database. But in practice this is likely to be too time-consuming. A better approach would be to back-up/restore at class level (in a method marked up with the `ClassInitialize` attribute). In this case the programmer must make sure that the various test methods in that class do not trample on each other. This is only an issue when writing data, and can be simply managed by techniques such as:

- Ensuring that each test writes data to a different context - such as a different Customer.
- (Where this is not possible) Ensuring that any created objects are unique - for example by using a GUID as part or all of the content.

Using object fixtures within XATs

Object fixtures are used to set up objects for use in XATs, as an alternative to data fixtures (which are installed and managed by the database). The principal advantage of using object

fixtures is that you can make use of methods and behaviour in the object model to set them up.

Naked Objects provides a dedicated pattern for creating and installing object fixtures before tests are run. In this pattern, an object fixture may be defined by any class that has a `public void Install()` method. Fixtures may have services (including the Container) injected into them, just like any domain object or service. The following code illustrates a simple object fixture class:

```
public class CustomerFixture {

    public IDomainObjectContainer Container {protected get; set;}

    public void Install() {
        CreateNewCustomer("Fred Smith", "0001");
        CreateNewCustomer("Joe Bloggs", "0002");
    }

    public Customer CreateNewCustomer(string name, string number) {
        Customer cust = Container.NewTransientInstance<Customer>();
        cust.Name = name;
        cust.Number = number;
        return Container.Persist<Customer>(ref cust);
    }
}
```

Fixtures are registered by overriding the `Fixtures` property on the `AcceptanceTestCase`, for example:

```
protected override object[] Fixtures {
    get {
        return new object[] {new FixtureEntities(), new FixtureLinksUsingTypeName()};
    }
}
```

The fixtures are installed by calling the `RunFixtures()` method, which will typically be done within the test initialization method, for example:

```
private static bool fixturesRun;

[TestInitialize()]
public void TestInitialize() {
    InitializeNakedObjectsFramework(this);
    if (!fixturesRun) {
        RunFixtures();
        fixturesRun = true;
    }
    StartTest();
}
```

(In the example code above, a static flag is used to ensure that the fixtures are run only once for the whole test class.)

End to End Testing with Selenium

When using Naked Object MVC, user-interface testing is not always necessary, or even valuable:

- If you are using only generic views you can rely on the fact that Naked Objects MVC is extensively auto-tested, and therefore concentrate all your testing effort on the domain model itself, for example using XATs.
- If your customisation is limited to CSS then you can't impact the functionality of the application; the worst you could do is render the application unusable by hiding elements.

However, if you are using any custom views or controllers then it will be necessary to test any parts of the user interface impacted by those custom views or controllers - either relying on manual testing or through an automated UI test framework.

Selenium is browser-based automated UI test framework. Naked Objects provides a set of helper methods that can facilitate the writing of Selenium tests specifically to work with a Naked Objects MVC application.

Adding a Selenium test project

1. Add a new Test Project to your solution using the standard Microsoft Test Project template. There is no need to add any references to any other projects in the solution, because you will be testing against an application running within a local (or remote) web server.
2. Find and install the NuGet package `NakedObjects.Mvc.Selenium` into your test project; this will automatically install the Selenium framework and other dependencies.
3. In order to run your Selenium tests, you will first need to run the application on local host, or deploy it to a remote IIS. If running in local host, take a note of the server url including the port number e.g. `http://localhost:53686/`
4. On the `Site.WithServices.Master` view, comment out the references to any `.css` files. The Selenium tests ignore `.css`, but you will find that the drop-down menu implementation on the default `.css` will render the tests inoperable because it depends on mouse-location to un-hide menu items.
5. Then simply run the tests as you would run regular unit tests.

Writing Selenium tests

The following shows an example of a Selenium test that is written to run against Naked Objects MVC:

```

using NakedObjects.Web.UnitTests.Selenium;
using OpenQA.Selenium;
using OpenQA.Selenium.IE;

[TestClass]
public class MyTests {

    protected const string url = "http://localhost:53686/"; //Replace the url with your
    application server url
    protected IWebDriver br;

    [TestInitialize]
    public virtual void InitializeTest() {
        br = new InternetExplorerDriver();
        br.Navigate().GoToUrl(url);
    }

    [TestMethod]
    public void FindEmployeeByName() {
        br.ClickAction("EmployeeRepository-FindEmployeeByName");
        br.GetField("EmployeeRepository-FindEmployeeByName-Name").TypeText("paul", br);
        br.ClickOk();
        br.AssertContainsObjectView();
        br.AssertPageTitleEquals("Paul, 190");
    }
}

```

Some points to note:

- This test is written to use Internet Explorer. Selenium drivers for the Firefox and Chrome browsers are also available.
- The class `NakedObjects.Web.UnitTests.Selenium.SeHelpers` defines a large number of extension methods for `IWebDriver` and `IWebElement`, a few of which are shown in the example above (e.g. `ClickAction`, `ClickOK`). These mimic the gestures typically applied to a Naked Objects MVC user interface.
- The identifiers provided as arguments to several of these helper methods are the Ids found in the generated HTML

Writing tests to run on multiple browsers

You can write your tests to run against multiple browsers, using the following pattern:

```

public abstract class MyAbstractTests {

    protected const string url = "http://localhost:53686/"; //Replace the url with your
application server url
    protected IWebDriver br;

    protected void CommonInitialize() {
        br.Navigate().GoToUrl(url);
    }

    public abstract FindEmployeeByName();

    protected void CommonFindEmployeeByName() {
        br.ClickAction("EmployeeRepository-FindEmployeeByName");
        br.GetField("EmployeeRepository-FindEmployeeByName-Name").TypeText("paul", br);
        br.ClickOk();
        br.AssertContainsObjectView();
        br.AssertPageTitleEquals("Paul, 190");
    }
}

[TestClass]
public class MyIETests {

    [TestInitialize]
    public virtual void InitializeTest() {
        br = new InternetExplorerDriver();
        CommonInitialize();
    }

    [TestMethod]
    public override void FindEmployeeByName() {
        CommonFindEmployeeByName();
    }
}

[TestClass]
public class MyFirefoxTests {

    [TestInitialize]
    public overrides void InitializeTest() {
        br = new FirefoxDriver();
        CommonInitialize();
    }

    [TestMethod]
    public override void FindEmployeeByName() {
        CommonFindEmployeeByName();
    }
}

```

Internationalisation / Localisation

Naked Objects has full support for internationalisation (sometimes referred to as *'i18n'*) and localisation (*'l10n'*) - allowing an application to be presented in different languages and/or cultures for different users without having to alter the domain code.

Following standard .NET practice, localisation is specified in a set of `.resx` files, which are compiled into their own assemblies. If you look in the `bin` directory of your Run projects you will see three Resources `.dlls` as follows:

- `NakedObjects.PMResources.dll` (the resources associated with the `NakedObjects.ProgrammingModel` NuGet package)
- `NakedObjects.MVCResources.dll` (the resources associated with the `NakedObjects.Mvc` NuGet package)
- `NakedObjects.FWResources.dll` (the resources associated with the `NakedObjects.Framework` NuGet package, on which the `Mvc` package depends)

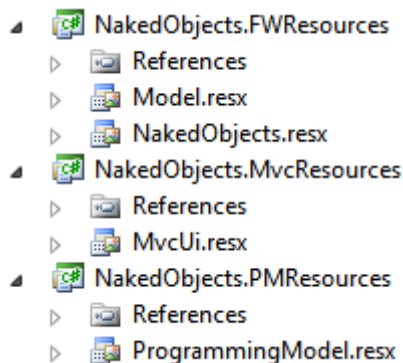
Adding the Resources projects to your solution

The first step toward localisation is to locate the source code projects that create these three assemblies (download them from CodePlex):

- `NakedObjects.PMResources` (the resources associated with the `NakedObjects.ProgrammingModel` NuGet package)
- `NakedObjects.FWResources` (the resources associated with the `NakedObjects.Framework` NuGet package)
- `NakedObjects.MVCResources` (the resources associated with the `NakedObjects.Mvc` NuGet package)

and to add these three projects to your solution.

Opening each project will show the `.resx` files as shown here:



The role of these is as follows:

- `Model.resx` contains the various labels that are generated reflectively from your own model project(s) - object names, property names, action names, parameter names and so on. The version of this file included in the `NakedObjects.Resources` project is empty by default. You will need to generate the file from your application - which is a very simple process, described below.
- `NakedObjects.resx` specifies all the strings used in the core Naked Objects framework. Typically these are generic validation, warning or error messages such as 'Field not editable'.
- `MvcUI.resx` specifies all the strings used within the generic Naked Objects MVC user interface, such as the labels for the `Edit`, `Save` and `Select` buttons, and the generic actions in the `Find` menu.
- `ProgrammingModel.resx` specifies all the strings used in the (optional) Helper classes.

In addition to these standard resource files, you may wish to add further resource files, for example to cover strings used within your model code (such as application-specific error, warning or validation messages), or for text used in any custom views. Any such additional resource files will need to be referenced using the standard .NET patterns for localisation.

Specifying that you wish to localise the application

By default a Naked Objects application is not localised. However, date and currency formats will always be localised to the formats specified on the user's machine.

To localise the application see [Configure Localization and Internationalization](#).

Then add the following method into the `Global.asax.cs` file, to tell the system to pick up the language/locale from the browser's settings, and if none is specified by the browser to default to English for the US locale (or whatever your preference):

```
protected void Application_BeginRequest(object sender, EventArgs e) {
    using (var fakePage = new Page()) {
        var ignored = fakePage.Server;
        fakePage.Culture = "auto:en-US";
        fakePage.UICulture = "auto:en-US";
    }
}
```

Populating the Model resource file

First, [register](#) the fully-qualified path name for the `Model.resx` file in the `NakedObjects.FWResources` project.

Now run the application. The file will be created during the initial start-up phase of the application, but the file will only be terminated when the application is properly terminated.

When running Naked Objects MVC you should stop the Visual Studio Test Server and wait a few seconds to allow the server to finalise. The created file will over-write any existing `Model.resx` file.

Warning: You should ensure that this overridden property is removed, or commented-out, for the live distribution.

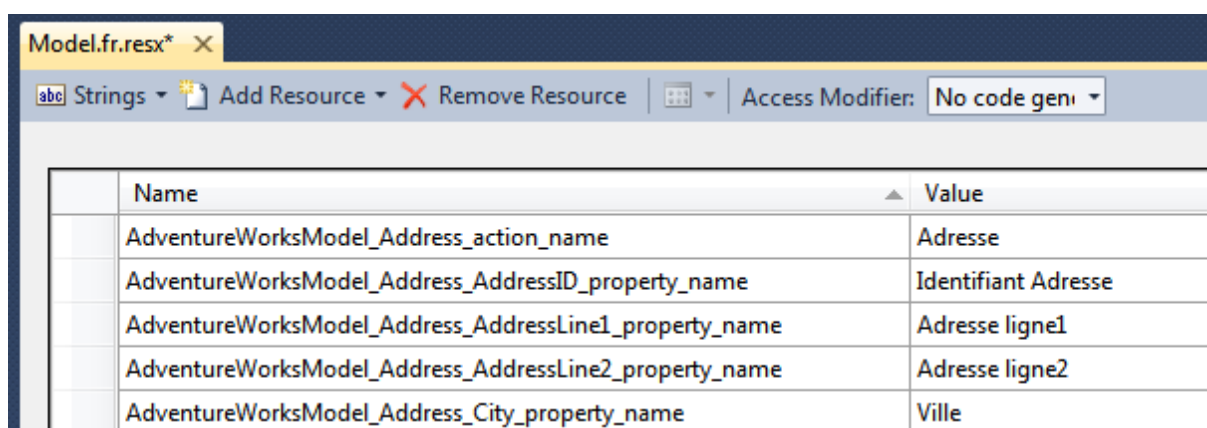
Translating the resource files

The next step is to create a translated version of each of the `.resx` files for each of the locale that you wish to support. For example, to add support for the French (`fr`) language, you will need to end up with these files:

- `Model.fr.resx`
- `NakedObjects.fr.resx`
- `MvcUi.fr.resx`
- `ProgrammingModel.fr.resx`
- (Optionally) Your own resource file(s) such as `MyAppStrings.fr.resx` if you wish to localise strings within your own application code or custom views.

`fr` is the generic designation for the French language, whereas `fr-FR` is French specifically for the France locale, and `fr-BE` French language for Belgium locale. If you make your resource file specific to, say, `fr-FR` then it will not be picked up by browsers set to `fr` (though the reverse does work). This is all standard localization practice, not specific to Naked Objects.

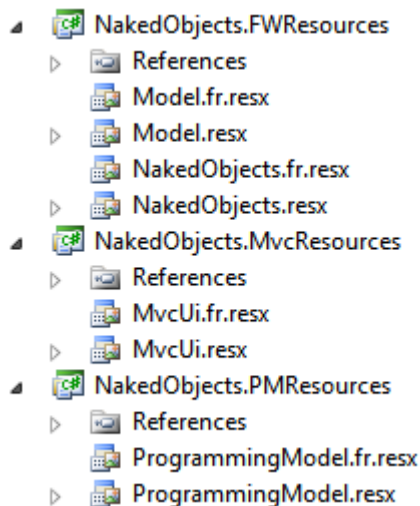
You can copy and edit the `.resx` files by hand, or use a third-party translation tool or online service. The screenshot below shows a small part of a translated `.resx` file for the AdventureWorks sample model:



The screenshot shows a Visual Studio window titled 'Model.fr.resx'. The interface includes a toolbar with 'Strings', 'Add Resource', and 'Remove Resource' buttons, along with an 'Access Modifier' dropdown set to 'No code gen'. Below the toolbar is a table with two columns: 'Name' and 'Value'.

Name	Value
AdventureWorksModel_Address_action_name	Adresse
AdventureWorksModel_Address_AddressID_property_name	Identifiant Adresse
AdventureWorksModel_Address_AddressLine1_property_name	Adresse ligne1
AdventureWorksModel_Address_AddressLine2_property_name	Adresse ligne2
AdventureWorksModel_Address_City_property_name	Ville

The translated resources files must be included in the projects, for example as shown below:

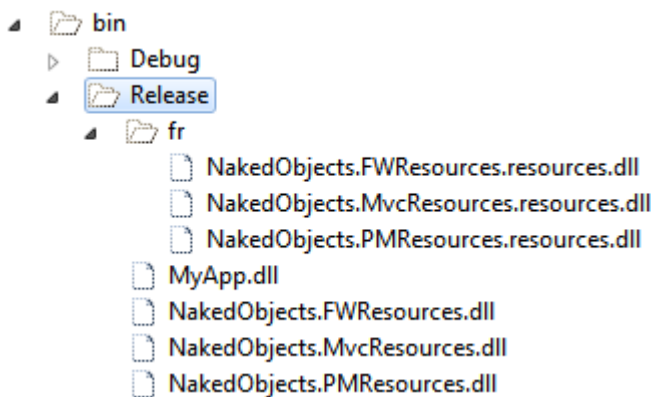


Finally, in your Run project, remove the direct references to the three resource .dlls, and replace these with project references to the three resources projects shown above.

When done, re-build the entire solution. If you show the `bin` directory of your Run project (hit Refresh if necessary) then you should find that it now contains:

- The original `resources` .dlls.
- A folder for each language for which you added translated `.resx` files

This is shown in the example below (other .dlls have been removed, for clarity):



Testing your localised application

You are ready to test your localised application. (Remember to remove or comment out the `ModelResourceFile` property if you haven't already done so, but to leave the overridden `Localise` property set to `true` - see above). Set the language preference on your browser (how this is done varies between browsers). Then run the application as usual and check for the translated labels throughout.

Clusters

The Cluster Pattern

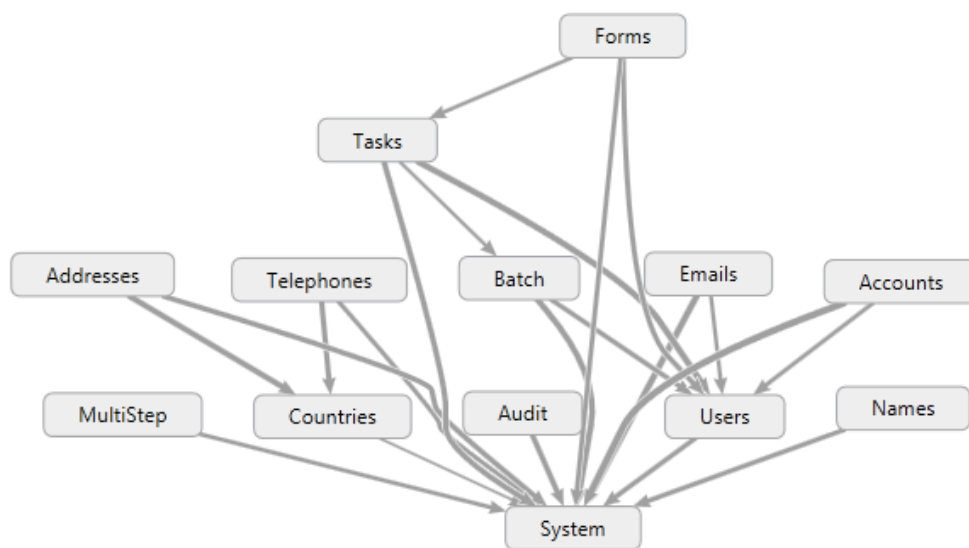
The Cluster Pattern is a specific high-level pattern for breaking up a large domain object model – following a very strict set of rules. Each cluster provides a distinct, re-usable, piece of business functionality. The term 'cluster' has been used only to distinguish this pattern from the more general idea of re-usable 'modules' or 'components' which, although they might have some features in common with the cluster pattern, typically do not enforce such strict rules.

The Cluster Pattern does not depend upon the Naked Objects Framework (NOF). However, the NOF makes it easier to implement the pattern, and makes its value more explicit.

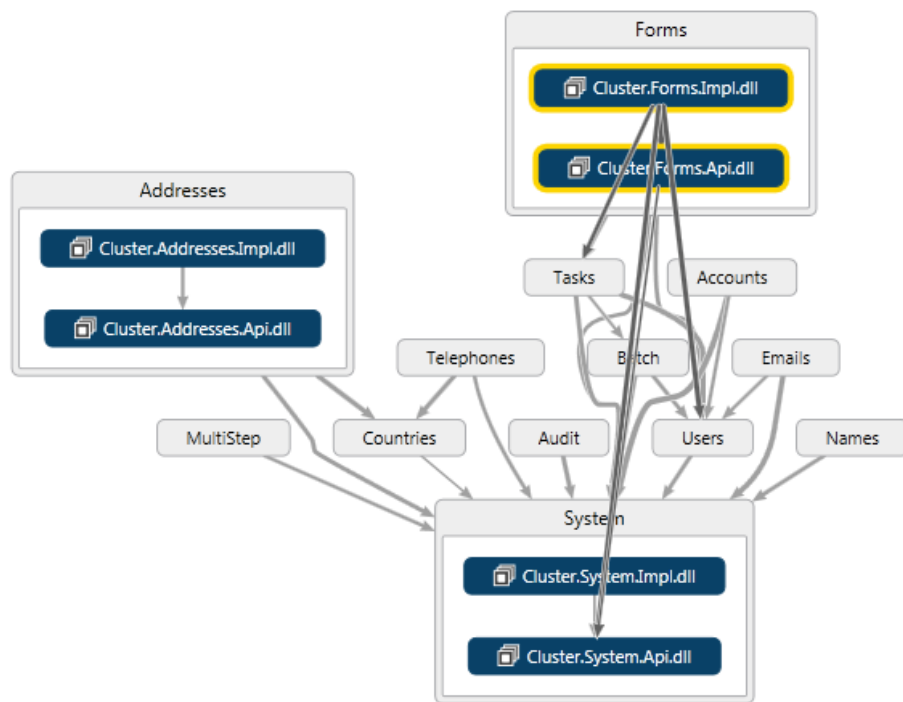
Hard Rules

1. Each Cluster is defined by separate Api and Impl projects - the latter referencing the former. The Api defines only the programmatic interface to the Cluster (as may used by other Clusters) - it does not define the user view of the cluster. In general the Api exposes the minimum possible of the implementation.
2. A cluster depends on other clusters only where this clearly makes sense from a business perspective. For example, the Emails cluster depends upon the Documents cluster as the created emails may be stored as documents.
3. A Cluster may only ever reference the Api of another Cluster; it may never reference another Impl project. (It is recommended that this be enforced by build script rules on your build server).
4. A Cluster Api will typically consist of interfaces. An Api may also contain Enums, Constants, and (less commonly) static classes and methods. Members on interfaces should use only other interface types, Enums, or .NET value types. An Api may NOT contain any persistable classes, whether abstract or concrete. They may contain ViewModels or other non-persistent classes, though this is not encouraged as it can lead to confusion. It follows from the above that:
 - Classes in a Cluster Impl may not inherit from classes in other clusters: this is a deliberate constraint.
 - Any associations between objects in different clusters must be defined by interfaces, and must therefore follow a [Polymorphic Association](#) pattern.
5. The interfaces in a Cluster Api are explicitly labelled as one of three types:

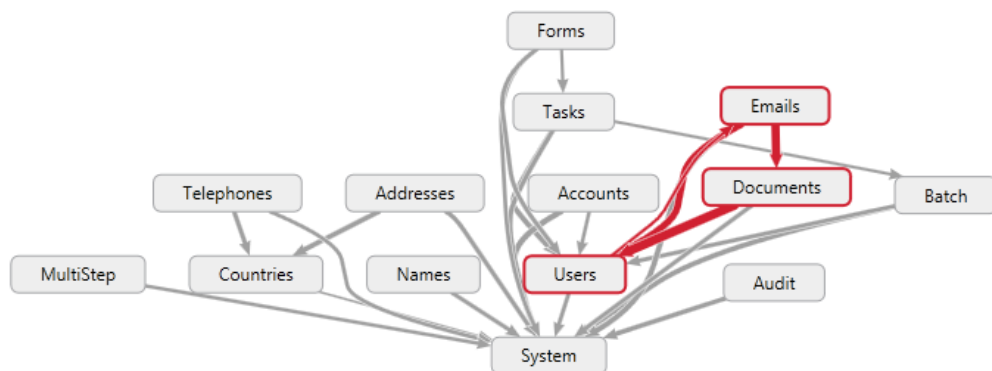
- A **service** interface defines a service for which there is an implementation in the Impl. Other clusters may require an implementation of this service interface injected into them.
 - A **result** interface is a restricted view of a domain type that is defined within the cluster's Impl and may be created and/or retrieved by means of service interface methods. Good practice says that result interfaces should hide data behind higher-value behaviours where possible i.e. provide methods in preference to properties; any properties that *are* exposed should be read-only except in rare cases – in which case the rationale should be documented with comments.
 - A **role** interface is intended to be implemented by objects in *other* clusters in order that those objects can take advantage of behaviour implemented in the cluster. Thus, those role interfaces may form input-parameters for methods on the service interfaces defined in the API. Additionally, the cluster Impl may define ContributedActions for that role interface.
6. Clusters should form a natural hierarchy of dependency: if the Api and Impl were treated as a single entity there should be no circular dependencies between them. This is a subtle point, not the same thing as saying that there should be no circular dependencies between projects (which would not compile anyway). This is illustrated in the following diagrams. First, an example of a cluster hierarchy:



Notice that each cluster depends only on cluster below it. Expanding the view of three of the clusters, we can see that each one has an Api and Impl project; that the Impl depends on the Api, and that the Api *and/or* the Impl projects may depend on the Api's in other clusters, but never on other impls:



Finally, here is a diagram of an earlier iteration of the same cluster project:



The diagramming tool (Visual Studio Ultimate in this case) has detected a circularity between clusters, shown in red. This is not strictly a circular reference between projects (as that would not compile). It occurs, in this example, because the `Users.Impl` project referenced `Emails.Api`, and `Emails.Impl` referenced `Users.Api`, in addition to the (intended) dependencies of `Emails` on `Documents`, and `Documents` on `Users`. The problem with this sort of circularity is that it means that `Emails`, `Documents` and `Users` cluster would always have to be used together and effectively form a ‘mega-cluster’. This error was easy to correct, however.

Optional Rules

The following represent good practices which are followed in this code base, but are not strictly definitional to the cluster pattern:

6. Each cluster defines its own DbContext and mappings.
7. Each cluster manages its own authorization via a standard pattern. The roles used by this authorizer are defined as constant strings on the cluster API.
8. Each cluster has its own test project, with emphasis on XATs that include full persistence, rather than on unit testing. (We follow Jim Coplien's advice that unit testing is best applied only to functions that have a clear, objective, and public, definition such as an algorithm.)
9. All projects in this library are set to All Warnings As Errors - so that they are warning free.
10. Clusters may be deployed as NuGet Packages, one each for the Api and Impl respectively, using SemVer rules for versioning.

The Clusters project on GitHub

The open-source Clusters project on GitHub (see <https://github.com/NakedObjectsGroup/Clusters>) provides an example cluster hierarchy that follows the rules specified above. Those clusters may be used and/or modified within a domain model project

Important: While the Clusters project does contain a significant number of tests, the test coverage is not comprehensive. If you use the code from the Clusters in your own project we recommend that you write additional tests as needed. Unlike the Naked Objects Framework, the Clusters project does not currently have a formal release process.

How to build the framework from source

The source code for the Naked Objects Framework is hosted on GitHub here:

<https://github.com/NakedObjectsGroup/NakedObjectsFramework>

To build the framework locally from source, open one of the top-level solutions such as Core.Sln. Ensure that you have the latest version of the NakedObjects.Ide NuGet package installed at solution level, as this installs a set of PowerShell scripts. To see these scripts, open the Package Manager Console window and type:

```
PM> get-help nakedobjects
```

Which will return:

Name	Category	Module	Synopsis
----	-----	-----	-----
New-NakedObjectsCleanBuildTest of the Naked Objects Framework ...	Function		Do a clean build
New-NakedObjectsCleanBuildNoTest of the Naked Objects Framework ...	Function		Do a clean build
Get-NakedObjectsAllPackageVers... versions of the NakedObjects package...	Function		Display the
Update-NakedObjectsAllPackageV... versions from nof-package-versions...	Function		Reads the new
Update-NakedObjectsPackageVersion version of a single NakedObjects pack...	Function		Update the
Update-NakedObjectsPackageConf... packages.config files and all .csproj...	Function		Update all

You can obtain further help, including examples, for each of these functions by e.g.:

```
PM> get-help New-NakedObjectsCleanBuildNoTest
```


Troubleshooting

This section contains hints and tips on troubleshooting.

Logging

Logging is managed through the Common.Logging framework. To add logging to your application you need to:

1. Add your choice of a common logging implementation into your Run project. Search the NuGet Public Gallery for 'common.logging', which will list a number of implementations including: Log4Net (the package in this case is Common.Logging.Log4Net1211), NLog, and Elmah.
2. Also in your Run project, find the class `NakedObjectsStart`, and the method `InitialiseLogging()`.
3. Add code into this method appropriate to the implementation you have added. For example, if you have added one of the Log4Net implementations, you might add the following code:

```
var properties = new NameValueCollection();
properties["configType"] = "INLINE";
properties["configFile"] = @"C:\Naked Objects\nologfile.txt";
LogManager.Adapter = new
Common.Logging.Log4Net.Log4NetLoggerFactoryAdapter(properties);
```

4. Again, depending on the implementation you added, you may wish/need to add some logging configuration into the `.config` file.

Having set up for logging, the Naked Objects Framework will log a significant number of generic internal events. You may also choose to add you own custom logging events from within your domain code.

Problems running the AdventureWorks example

Error locating server

If you get this error message, it is likely to be because the wrong data source is specified in the connection string, which is located in the `App.config` file.

The application runs very slowly

With the exception of the very first access to the database (which is always slow) the Naked Objects AdventureWorks application should run at a very good speed. The application will run more slowly if you run it in Debug mode from Visual Studio - especially so if you are connecting to a remote database server. It is strongly recommended that you run the application as an executable from Visual Studio (**Ctrl-F5**) unless you specifically need to debug the application.

Problems working CodeFirst

Database is generated, but certain (or all) tables are not being generated

This suggests that the framework is not identifying any domain classes. Two possible reasons for this are:

- Domain class namespaces beginning with `NakedObjects`. The Naked Objects framework assumes that any classes within this namespace are part of the framework, and ignored by the domain model reflector. This is to avoid accidentally treating framework classes as domain entities, and thus building them into the database schema. Choose another namespace for your domain models.
- Domain classes not reachable. Naked Objects builds its metamodel (which in turn is passed to Entity Framework to create the entity model and hence the database schema) by walking the graph from known start points - which means the set of [registered services](#). So if there is no way to get to a particular class, either directly or indirectly, via any method on any registered service - then the class won't have been reflected on during the start-up phase.

Errors thrown when starting an application

This section contains hints and tips if your Naked Objects application throws an error during the start up process.

No known services

This error indicates that you have not [registered](#) any services..

Unable to infer a key

This error should only occur if you are running Code First with Entity Framework. It indicates that the class `FooBar` does not have a key property. You need to ensure that the class has an integer property that can serve as the database key. If the key property follows the naming convention of `[classname]Id` or just `Id` then it will be picked up by the Code First mechanism automatically; if you do not wish to follow this convention, then you may instead mark up the property with the `System.ComponentModel.Key` attribute.

The best way to avoid this error occurring is to create new domain classes using the Domain Object item template, which automatically creates a key property.

Another possibility that can give rise to this error is if you have registered the class `FooBar` as a service, but it contains one or more properties. Services may not have properties. See [Service](#).

Class not public

This error was caused by the domain class `FooBar` not being `public`. Visual Studio creates Visual Basic classes `public` by default, but creates C# classes as `internal` by default.

The best way to avoid this error occurring is to create new domain classes using the Domain Object item template, which automatically creates a `public` class.

Errors thrown when running an application

This section contains guidance on error messages that might appear while you are using a Naked Objects application.

A property is not virtual/overrideable

This error has arisen because the class `FooBar` contains:

- One or more properties that have not been made `virtual`. This is a requirement in order to allow the Entity Framework to create a proxy for the object. The best way to avoid this error occurring is to create all new properties using the `propv` code snippet.
- A collection that is not of type `ICollection`. See [Collection properties](#). The best way to avoid this error occurring is to create new collections using the `coll` code snippet.

Invalid column name

This error is quite common when prototyping in Code First mode with Entity Framework - where a new property (in this case called `Surname`) has been added to a class since the database was created. The solution is either to delete the database and run again (which will re-create the database), or to manually add a new column to the appropriate database table.

Invalid object name

As indicated by the fact that this is a `SqlException`, this error arises where a domain object class does not have a corresponding table in the database. The remedy is the same as for Invalid Column Name (above).

Collection not initialised

This error has arisen because a collection was not initialised (which is standard programming practice). See [Collection properties](#). The best way to avoid this error occurring is to create new collections using the `coll` code snippet.

Could not find Entity Framework context for type

This is most likely to occur if you are working with multiple entity models (see [How to work with multiple databases](#)). You need to ensure that:

- Each `.edmx` file has a different name (for example, based on the assembly name for the project within which it lives)
- Each generates a different connection string name.
- The connection strings are all copied into the `App.config` file within the run project.

Unexpected behaviour in the user interface

This section contains suggestions on what to look for if your application is not behaving as you expect at the user interface.

A public method is not appearing as an object action

Possible causes:

- Naked Objects does not recognise any overloaded methods as actions. (This is because the various overloaded versions would show up on the menu with the same label.)
- Naked Objects does not recognise templated methods as actions.
- Naked Objects only recognises as actions those methods with parameter types that are either recognised value types or domain objects.

Default, Choices, Validate or other complementary methods are showing up as menu actions

If you have written a Default, Choices, Validate or other 'complementary method' that is intended to work with a Property or Action, and it shows up as a menu action on the user interface then the method isn't being recognised as a complementary method by the Naked Objects framework. This is probably due to one or more of the following:

- The name of the helper method (following the prefix) does not exactly match the name of the Property or Action it is intended to assist. The difference might be in spelling or case.
- The parameters of the helper method differ in type from those of the Property or Action it is intended to assist.
- A complementary method in a sub-class will only be applied to a property or action defined in a super-class, if that complementary method is also defined on the super-class and overridden in the sub-class. To put that another way: complementary methods must be defined in the same class as the property or action they apply to, but may then be overridden in sub-classes.

The name of the corresponding action happens to begin with one of the prefixes recognised by Naked Objects (for example: `ClearComment`). The best policy is to avoid using action names that begin with this prefixes - give the action a different name and then use the `DisplayName` attribute to change it back to what you want on the user interface. Alternatively, just mark up the complementary method with the `Hidden` attribute.

Debugging

This section deals with debugging your code.

Life Cycle methods are not being called

If you suspect that an life-cycle method is not being called, then you should first verify this by inserting a break point on the method. If it is not being called as expected, then the most likely explanations are:

- The method is not `public`.

- The method is not `void`.
- The method name is mis-spelled
- The method name does not begin with an initial upper-case letter.

The injected Container is null

The most likely explanation for this is that you have accidentally shadowed the property for the injected container in your domain class hierarchy. You can set Visual Studio to warn against this.

An injected service is null

The most likely explanation is that you haven't registered the service in your run-class. See also the point above ('injected Container is null'), which may apply to services also.

The application runs OK in execute mode, but throws an exception in debug mode

Because Naked Objects makes heavy use of reflection, Visual Studio cannot always detect that an exception being thrown by one part of the framework is going to be picked up by another part of the framework, and will consequently break (when in debug mode) when the exception is thrown but before it is caught. Hitting Continue will typically work. If this behaviour proves annoying then you can change the settings on Visual Studio. Navigate to Debug > Exceptions > Common Language Runtime Exceptions and un-check the User-unhandled checkbox: