**The Naked Functions programming model**

An application written according to the Naked Functions programming model consists of:

* [Domain Types](#_Domain_Types)
* [Functions](#_Functions)

# Domain Types

Domain types are the entities of the system, such as Customer, Product and Order. Instances of these types are displayed on the user interface, and, *typicially* they are persisted to the database, though it is also possible to have ‘View Model’ domain types, that are displayed but not persisted.

Domain types in Naked Functions have some similarities to the concept of domain objects in the Naked Objects programming mode. They have properties that may hold basic value types (such as number, string, or date), references to other domain types, or collections of domain types. However, there are two important differences:

* Domain Types in the Naked Functions programming model do not have any methods. The only behaviour built into the types is in the type constructors.
* They are immutable.

## Implementation of types

As C# continues to expand its Functional Programming (FP) capabilities, it is possible that future version of Naked Functions will support, indeed encourage, the use of Records as the mechanism for defining types.

Similarly, at present, Entity Framework cannot persist immutable objects - although this might well be supported in a future version.

Therefore, to achieve the required programming pattern or working with immutable type the following principles apply:

* All domain types are defined as C# interfaces, where each interface defines only properties, and where each property is read-only (has only a getter).
* Each domain type interface is implemented by a class, wherein all persisted properties have both a getter and setter. In addition, each class has one or more constructors, defining parameters such than an instance may be created with all properties set up.
* Application functions deal only with the domain interfaces (as their parameters, or returned results), never with the classes directly. This principle should be enforced by build-time rules, for example by ensuring that all the domain type interfaces are defined in a xxx.api namespace, and domain classes in a xxx.impl namespace.

## Creating a new instance of a type

New instances are created through ‘factory’ functions, also defined within a xxx.api namespace, where the return type is defined as the interface e.g.:

IProduct CreateNewProduct([params list])

These factory functions delegate entirely to a constructor on the implementing class, but serve to keep the implementing class hidden from the calling code.

## Updating existing instances

Since domain types are *deliberately* (from the application programming perspective) immutable, existing instances cannot be updated directly. So instead, a new instance is created as a copy of an existing instance. Naked Functions provides an easy-to-use helper function - With - to do this, which may be called as an extension method on an existing instance. In the following example, p is an existing instance of an implementation of the domain type IProduct, and the following code will create a new instance *of exactly the same class as* p, with all properties copied from p except for the Description property, which has a new value :

p.With(x => x.Description, "New description");

How the newly created instance is related to the original one for the purposes of persisting the update in the database, is described under Actions below.

# Functions

In the Naked Functions programming model, all application behaviour is defined as static functions.

From a programming perspective, all functions in the domain code must be side effect free. The implementation of side effects - such as outputting information to the user-interface, or updating the database – is provided by the framework, not by the application programmer.

Similarly, functions must derive their returned result solely from the sources provided as function parameters.

Functions may call other functions that adhere to the same principles.

Where a function is intended to be invoked by a user, it is known as an ‘action’.

In addition to actions there are a number of other specific forms of function that are recognised by the framework: functions that provide behaviour associated with specific actions (for example to validate an input parameter at the user interface), and functions associated with specific instances of domain types (for example to define the title to display on the UI for that instance).

## Registration of actions

All functions intended as user actions must be registered via NakedObjectsRunSettings in the run project, in order to be introspected at start-up time.

Currently, C# requires that static functions must be defined on a class, and so the Functions property on NakedObjectsRunSettings returns a list of types, and all static functions on those types, which follow the programming model conventions for actions (as described below), are interpreted as actions. Thus:

private static Type[] Functions {

get { return new Type[] {typeof(ProductFunctions), typeof(CustomerFunctions), typeof(MenuFunctions)}; }

}

Registers all static functions on three classes as user actions. We may infer from the naming that the first two of these define actions contributed to Products and Customers respectively, and the third provides menu actions. However:

* What determines that an action is a menu action is the fact that the action (or, more precisely, the static class defining the action) is used to create a menu within the MainMenus method on NakedObjectsRunSettings.

And for all other actions registered via Functions:

* The domain type(s) that an action is contributed to is determined by the type of its first parameter.

[Stef: I am still uncomfortable with this requirement of ‘double registration’ of menu actions. I would prefer that menu actions were registered solely via the MainMenus property, and that the Functions property was then renamed to TypeFunctions. Longer term - as previously raised - I’d like to apply the same principle to NakedObjects, but I’d rather not change the latter for the time being.]

## Menu actions and type actions

Menu actions are invoked by the user from the main menus defined on the home screen. Such actions are most commonly used to create new instances of types, or to retrieve existing instances.

## Type actions

Type actions are invoked in the context of a specific instance of a type, or collection of instances, and are displayed to the user in a form that suggests that they form a part of that instance or collection. (They may the thought of as broadly equivalent to ‘object actions’ in the Naked Objects programming model).

Any public function that has been registered with the framework and is not a menu function will be interpreted as a type action. The first parameter must be a domain type - and the action will be contributed to all instances of that type. (That first parameter is sometimes called the ‘contributee’). The framework will provide the instance when the action is called. The first parameter in any function defining type action will not be rendered in the action dialog provided to the user.

(It is also possible to use the C# keyword this, in front of the first parameter, such that the action may also be invoked programmatically as an *extension method.*)

TODO: query-contributed actions

## Other injected parameters

In addition to the first parameter on a type-action, the framework may also provide (or ‘inject’) certain, other parameter values automatically. Such parameters must be annotated with the [Injected] attribute, and must be of a type that the framework knows how to provide.

[Stef: although in the current prototype, the IQueryable<T> parameter is injects automatically, certain other examples must be explicit, so I am coming around to the view that we should require all injected parameters, including IQueryables, to be explicitly annotated with [Injected]. This does not include the first parameter in a type action, which we deem to be the ‘contributee’ rather than injected as such.]

A common example of this is a parameter of type IQueryable<T>, which then allows the body of the function to read selected instances of type T from the database.

Other recognised types that can be injected are:

* **Guid**. By default, the framework will call new Guid() and inject that value, but an alternative implementation may be registered via Unity.config for testing or other reasons.
* **DateTime**. By default, the framework will call DateTime.Now and inject that value, but an alternative implementation may be registered via Unity.config for testing or other reasons.
* **Random** **numbers**. Any Integer parameter marked up with [Injected] will be provided as a random number between 0 and MaxValue.
* **Principal**. Any IPrincipal parameter marked up with [Injected] will be provided with the Principal representing the current user.

In the following example:

public static List<Order> FindOrders(Product p, [Injected] IQueryable<Order> orders, DateTime fromDate)

the first parameter is the instance from which the action has been invoked; the second is provided by the framework; the third requires a user-entered value. So the action FindOrders will be contributed to all instances of type Product, and when invoked will render a dialog in the UI, requesting a single value of type DateTime to be entered.

## Returned values

All actions (indeed all functions) must return a value, which may be simple or compound in its structure. Thus, no function may be defined as void, though it may return a null value.

## Side-effect free actions

Where the intent of an action (menu action or type action) is just to retrieve one or more existing domain type instances from the database, and/or to create non-persistent ‘View Model’ instances, and render these on the user interface, then the action function should simply return these objects.

In the previous example:

public static List<Order> FindOrders(Product p, IQueryable<Order> orders, DateTime fromDate)

The action returns the resulting list of Orders to the user.

Actions following this form are deemed to be side-effect free, and will be invoked from the UI using the Http GET method.

## Actions that create or update persistent objects

Where the intent of an action is to create or update persistent objects then the action should return a Tuple. The first value in the Tuple will be the instance(s) to be rendered to the user (as per the previous point); the second value in the tuple will be the instances to be persisted or updated in the database.

These two sets of instances may be the same, or different. For example, an action might create/update one or more instances, but return null to the user.

For example, the following action:

public static (Order,OrderLine) AddOrderLine(Order o, Product p, int quantity)

Will create and persist a new OrderLine associated with the Order. However, the OrderLine is not explicitly returned to the user, just the Order, because the latter will have its collection of OrdeLines automatically updated with the new line. (In fact, the action implementation could just return null because that would then anyway leave the Order on which the action was invoked on the UI).

[Stef? Assuming, here, that the Order has a collection of OrderLines, and that OrderLine has a reference to its Order, is it sufficient that only the new OrderLine is specified as the to-be-updated-or-persisted value in the tuple, or would the Order need to be in there also?]

Any action returning a Tuple is deemed not to be side-effect free and will therefore be invoked from the UI via the Http POST method.

## Actions that render messages on the UI

Actions that render messages to the user, for example providing feedback information about an invoked action, or a warning, must return a three-value Tuple, where the third value is the message. The first two values are as described above, but either or both of those first two values may be null.

## Complimentary functions

TBD.

(Equivalents to all complementary functions in the NOF programming model. In all cases the first parameter must be the ‘contributee’ - even if it is not used within the function. In addition to parameters defined in the NO model, complementary functions may take additional parameters injected by the framework.

## Other recognised functions

### Title

For a functional domain type to have a title there needs to be a registered function, named Title, with a single parameter of the domain type, and returning a string. The Title function may explicitly delegate to a more general Title function for a supertype, but there is no automated ‘walking the hierarchy’.

### LifeCycle method

The Naked Functions programming model supports lifecycle functions broadly equivalent to the Persisting, Persisted, Updating, Updated instance methods in the Naked Objects programming model. (It does not support Created, since there are no transient objects, and certain other lifecycle methods are still to be decided.)

For the Persisting, Persisted, Updating, Updated the functions take the domain type as the first parameter and may optionally take additional parameters injected by the framework. Since Persisting, and Updating may ‘modify’ the instance, they always return the a new instance of the same type and this is swapped for the existing one by the framework. Updated and Persisted may not ‘modify’ the instance, but will typically generate other instances. They therefore return a single domain type instance, or collection of instances, and this is interpreted by the framework as a list of instances to be persisted or updated (according to whether they have Id values or not).

# Open questions

How to handle ViewModels? Currently they must implement IViewModel, which requires the implementation of instance methods – DeriveKeys and PopulateUsingKeys. This does not with the FP paradigm. Could we make it so that a view model had to implement an empty interface (IFunctionalViewModel say) but that there would need to be registered functions called DeriveKeys and PopulateUsingKeys taking in the specific type as first param?