

5/12/2009



MOBILECONTRIB

MOBILE CLIENT SOFTWARE FACTORY: CONTAINER MODEL APPLICATION BLOCK GETTING STARTED GUIDE

Contents

Introduction	3
Getting Started with the Data Access Application Block (DAAB).....	Error! Bookmark not defined.
Understanding the Data Access Application Block Quickstart Application	Error! Bookmark not defined.
Opening and Closing Databases.....	Error! Bookmark not defined.
Creating a Database Object	Error! Bookmark not defined.
Closing A Database.....	Error! Bookmark not defined.
Executing T-SQL Commands	Error! Bookmark not defined.
Creating and Dropping Tables.....	Error! Bookmark not defined.
Executing Parameterised T-SQL Commands.....	Error! Bookmark not defined.
Executing Commands that Return a Scalar.....	Error! Bookmark not defined.
Retrieving Data	Error! Bookmark not defined.
Fetching Records Using ExecuteReader.....	Error! Bookmark not defined.
Working with SqlCeResultSets.....	Error! Bookmark not defined.
Inserting Records	Error! Bookmark not defined.
Updating Records.....	Error! Bookmark not defined.
Retrieving Records	Error! Bookmark not defined.
Performing Updates In a Transaction	Error! Bookmark not defined.
Summary	Error! Bookmark not defined.

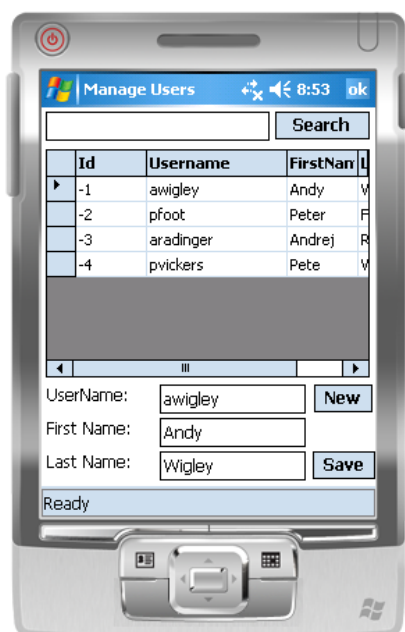
Introduction

This document and the accompanying code are not a Quickstart per-se, it is more of a tutorial on good architecture practices, and why the design patterns enabled by the Container Model application block are a good thing.

The sample code consists of three sample applications, all of which are visually identical, and which offer exactly the same functionality. However they are built in different ways. The applications show an initial launch screen which only has one icon on it, but in a larger application would typically offer many icons allowing access to the different parts of an application.



After you click on the 'Manage Users' icon, you are taken to a form that allows you to view a list of user records, search the list of users, to edit the details of a user and to create new users:



The three samples implement this functionality in different ways. The three samples are:

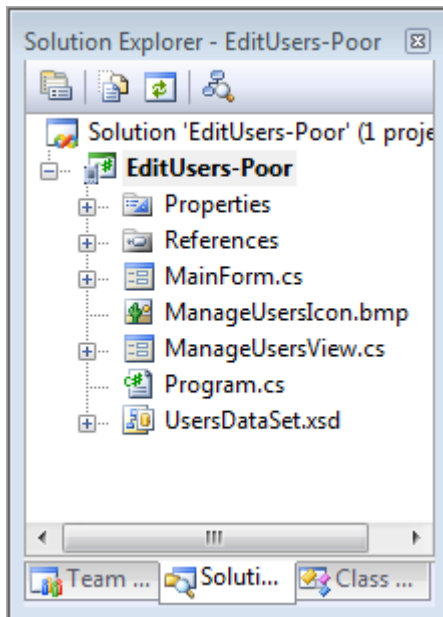
- **EditUsers-Bad** – this has only a few classes, but all the presentation logic, data access logic and business logic is mixed up inside the Form classes. This architecture is a bad thing, because it is difficult to change any part of it without the risk of introducing unwanted consequences – it is not easy to change, and change is one thing you can guarantee for any software project. You cannot reuse the ‘EditUsers’ module easily in another application, perhaps with a different UI, as it has not been created with the possibility of reuse in mind.
- **EditUsers-Better** – this sample is a great improvement. It uses the Model – View –Presenter design pattern so that presentation logic, business logic and data access logic are implemented separately and work through interfaces to interact with each other. This architecture encourages re-use; for example, it is easy to take the business logic (the ‘presenter’) and the data access code (the ‘model’) and reuse them with another presentation layer (the ‘view’) which is easy as long as the new view implements the same interface as the original. The presenter class only invokes functions in the view class through the view interface, so does not care how the view is implemented, only that it implements the view interface.
- **EditUsers-Best** – While the previous example is good, there is still room for improvement. In the previous version, the startup module still needs to hold a reference to all other modules in the application, and creates instances of them during application startup which in a large application could introduce unwanted memory pressure. This version modifies the previous example by using a Dependency Injection Container (implemented in the Container Model application block), which is an object that each module in an application uses to register its types, and the types on which it is dependant, and provides the factory function that the container can use to construct each object. The D. I. Container exposes the `Resolve<T>` method which client code uses whenever it needs an instance of a type, at which point the container looks up the factory function for that type in its internal dictionary and builds the requested instance; this happens only when the type is needed, not at application startup. By using the Container Model application block, you improve the EditUsers-Better example by enhancing loose coupling between types and testability.

It is always a challenge to explain architectural ideas about loose coupling between modules using a sample application that only has one module! In your mind, you must scale up your conception of a mobile application to one containing 10, 20 or 30 modules – such applications are not unusual in most enterprise mobile applications. But it’s not only for enterprises, these ideas are of value to anyone building mobile applications, because the result is an architecture that encourages objects that are clean, have singularity of purpose, are loosely coupled to other modules and classes in the system and above all, are testable. A class or module that is only loosely coupled to the other parts of the system may easily be tested by creating mock objects that implement the interfaces with which the module under test is expecting to interact, but which actually only implements sufficient logic internally to satisfy the goals of the test. This is a key ingredient of successful unit testing.

Poor Architecture – the Visual Studio drag, drop and double click design approach

The first sample, **EditUsers-Poor**, is a typical example of the kind of application built by most developers who have not started to apply design patterns to their development. It was built using the RAD capabilities of the Visual Studio Windows Forms Designer, and consequently does not take very long to build.

It consists of two Windows Forms, and uses a DataSet object (UsersDataSet.xsd) to serve as an in-memory data store.



In this example, the ManageUsersView form has been created by dragging the UsersDataSet type from the Toolbox onto the form, which creates a DataGrid control on the form, and sets up data binding between a UsersDataSet instance and the DataGrid via a BindingSource object. The usersDataSet instance on this form has been made Public, which allows us to initialize it with some test data from within the startup logic on MainForm:

```
public partial class MainForm : Form
{
    private ManageUsersView manageUsersView;

    public MainForm()
    {
        InitializeComponent();
        manageUsersView = new ManageUsersView();

        // Add some test data
        manageUsersView.usersDataSet.User
            .AddUserRow("awigley", "Andy", "Wigley");
        manageUsersView.usersDataSet.User
            .AddUserRow("pfoot", "Peter", "Foot");
        manageUsersView.usersDataSet.User
            .AddUserRow("aradinger", "Andrej", "Radinger");
    }
}
```

```
        manageUsersView.usersDataSet.User
            .AddUserRow("pvickers", "Pete", "Vickers");
    }

    ...
}
```

As you can see data handling, application logic and data presentation is all mixed up in the MainForm and ManageUsersView Form object code.

This implementation is:

- **Poorly structured** – data persistence, business logic and data presentation is all mixed up, which will make the result harder to alter and harder to change
- **Brittle** – This kind of structure is an example of ‘spaghetti code’ and as it grows, it becomes harder to maintain and harder to extend
- **Inflexible** – what happens if you decide to store your data in a database rather than an in-memory DataSet instance, or you want to introduce a new UI for a different screensize? The structure does not make it easy to make architectural changes without rewriting the whole application.
- **Inextensible** – there is nothing in this structure that makes it easy to add in new modules. No thought has been given to developing new modules and plugging them into the system to extend it.

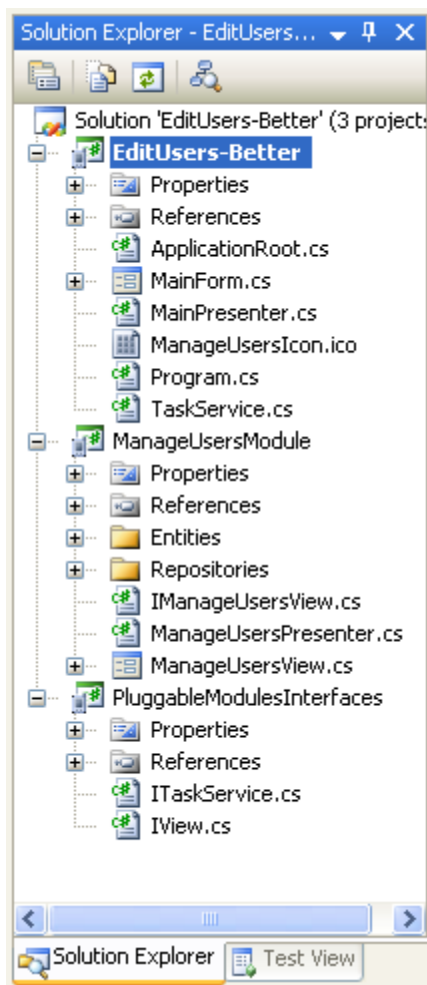
Better Architecture

The second sample, EditUsers-Better, introduces many design features that greatly improve the situation. Although this sample does not itself use a Dependency Injection Container, the architecture introduced here is a good step along the road to creating loosely coupled modules, which will be developed further in the final sample. Open the solution in Visual Studio 2008 and inspect the code as you read the following notes.

The key thing to observe is that all the functionality relating to management of users is now self-contained, situated in its own project, **ManageUsersModule**.

The solution is now divided into three separate projects:

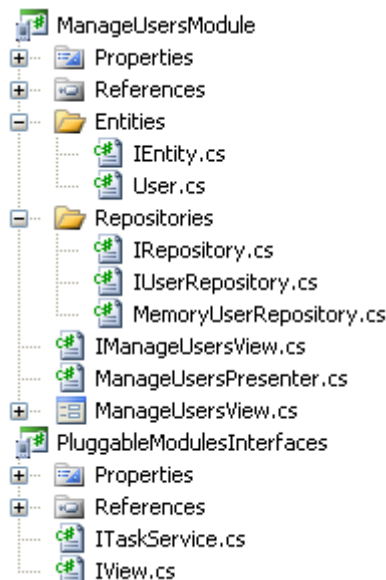
- **EditUsers-Better** – this is the application ‘shell’ and can be thought of as the ‘app launcher’, responsible for providing a framework into which other components of the application connect, and for providing navigation between components.
- **ManageUsersModule** – this component contains all the functionality for creating and editing users.
- **PluggableModulesInterfaces** – Contains only two interfaces, which provide the ‘contract’ between application modules and the shell.



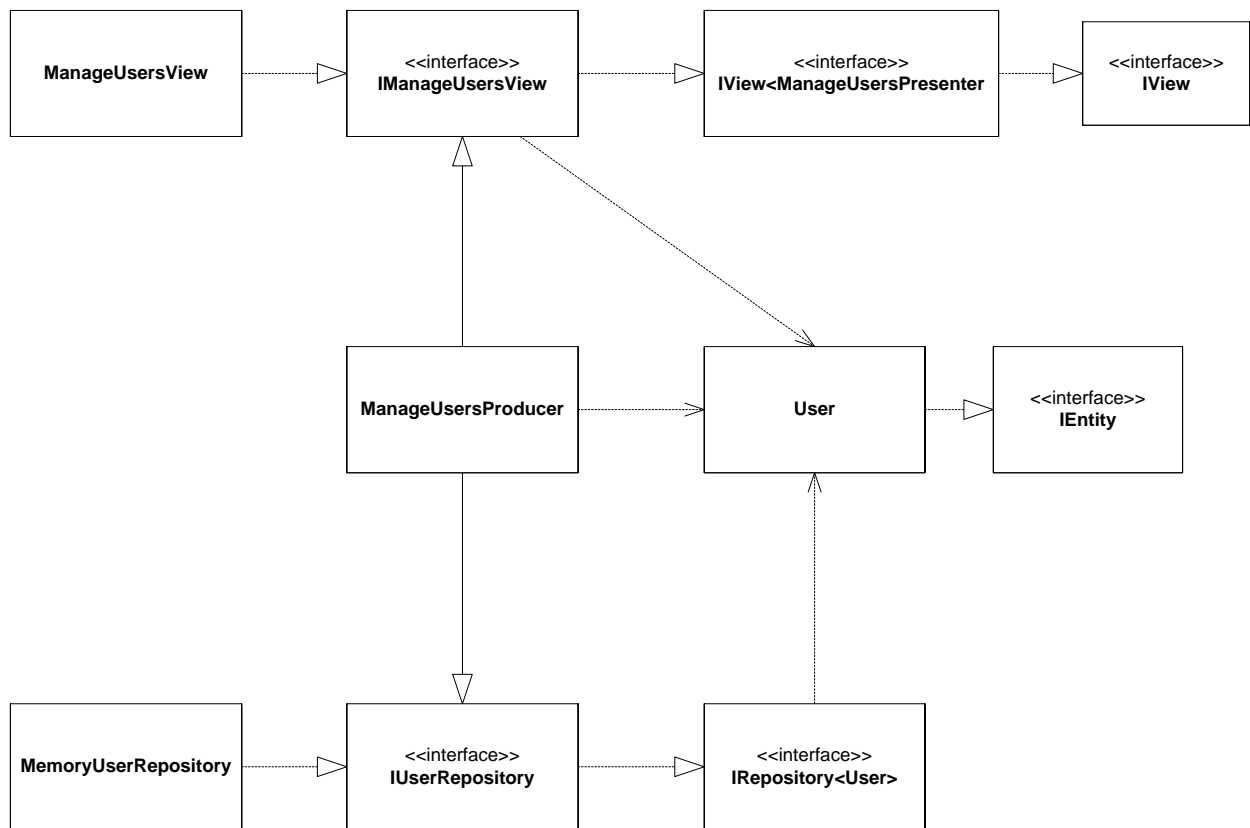
The different projects have as few dependencies on each other as possible. You could remove the `ManageUsersModule` from the solution, and it would only require commenting out of a few lines of code (those that are situated in the shell to configure and launch the `ManageUsersModule`) to get the remaining parts to build cleanly.

Use of Model-View-Presenter in the `ManageUsersModule`

Every component in a system should implement Model-View-Presenter to separate the persistence of data (the model) from the presentation (the view), with the Presenter sitting between these two to implement the logic of the component. This separation is enforced by using interface-driven design techniques, so that the points of contact between the View and the Presenter, and between the Presenter and the Model are defined by interfaces (a contract). The simple act of developing using interfaces helps developers focus on creating clean object models where each class has singularity of purpose. For example, it is easier to see when code relating to presentation of data is in the wrong class when it clearly ends up on the wrong side of an interface!



The most important relationships between the objects in this component are represented in the following UML class diagram:



The classes and interfaces on the top line implement the View, the **ManageUsersProducer** class implements the Presenter, and the bottom line implements the Model. The **User** class is a business entity that stores a Users' details, and which may be used to transfer data between all three layers.

As you can see from the diagram, the **ManageUsersProducer** uses the **IManageUsersView** to communicate with the View, and the **IUserRepository** to communicate with the Model. Interfaces are used here to enforce a 'contract' between the View and its user, the Producer, and between the Model and its user, again the Producer.

The use of Interfaces provides a number of benefits such as the clarity of design that results from dividing areas of functionality up and defining interfaces between them, and the ability to swap out one implementation of a view for example, and replace it with another for a different style of display. It also encourages reusability – it is far easier to take a module of code that has well defined borders (each point it interacts with other parts of the system being defined by an interface) and then modify it for use in a new system.

View Interface

The interface between the View and the Presenter is defined in **IManageUsersView**, which itself inherits from **IView<T>** (in the **PluggableModulesInterfaces** project). **IManageUsersView** looks like this:

```
public interface IManageUsersView : IView<ManageUsersPresenter>
{
```

```

        void SetResults(IEnumerable<User> searchResults);
        void SetStatus(string status);
    }

```

IManageUsersView inherits from IView<T>, which is defined as:

```

/// <summary>
/// Extended typed interface for views that
/// need a reference to its presenter.
/// </summary>
public interface IView<TPresenter> : IView
{
    /// <summary>
    /// The presenter instance.
    /// </summary>
    TPresenter Presenter { get; set; }
}

```

IView<T> itself inherits from IView - IView is the interface that the shell uses to activate the module:

```

/// <summary>
/// Basic interface used by the ITaskService to
/// show the views.
/// </summary>
public interface IView
{
    DialogResult ShowDialog();
}

```

See below for more information on how the shell activates a module through the IView interface.

Consequently, the ManageUsersView class that implements the IManageUsersView interface exposes:

- a public method used to display a list of User objects:
SetResults(IEnumerable<User> searchResults) [IManageUsersView interface]
- a method to display a status string: SetStatus(string status)
[IManageUsersView interface]
- a property getter and setter for some type that will act as the views' Presenter:
TPresenter Presenter { get; set; } [IView<T> interface]
- a method to activate the module: ShowDialog() [IView interface]

The use of interfaces is an effective way of expressing the usage of a module in a clear and unambiguous fashion. One important thing to note is that the actual method of displaying the data is not suggested by this interface but is left entirely up to the implementation of the View class. It would be easy to discard the **ManageUsersView** and replace it with a different concrete implementation of IManageUsersView that presented the data in an entirely different way, perhaps for a different sized display, or one that is not touch-sensitive.

Presenter class implementation

The Presenter is beautiful in its simplicity:

```
using System;
using System.Globalization;

namespace EditUsers_Better.ManageUsersModule
{
    public class ManageUsersPresenter
    {
        IManageUsersView view;
        IUserRepository repository;

        public ManageUsersPresenter(IManageUsersView view,
                                    IUserRepository repository)
        {
            this.view = view;
            this.repository = repository;
        }

        public void Search(string criteria)
        {
            if (!String.IsNullOrEmpty(criteria))
            {
                view.SetResults(repository.FindByName(criteria));
            }
            else
            {
                view.SetResults(repository.GetAll());
            }
        }

        public void Create(User user)
        {
            try
            {
                repository.Create(user);
                view.SetStatus(String.Format(
                    CultureInfo.CurrentCulture,
                    Properties.Resources.SavedUser,
                    user.FirstName, user.LastName));
            }
            catch (Exception)
            {
                view.SetStatus(String.Format(
                    CultureInfo.CurrentCulture,
                    Properties.Resources.SaveFailed,
                    user.FirstName, user.LastName));
            }
        }

        public void Save(User user)
        {

```

```

        if (repository.Update(user))
        {
            view.SetStatus(String.Format(
                CultureInfo.CurrentCulture,
                Properties.Resources.SavedUser,
                user.FirstName, user.LastName));
        }
        else
        {
            view.SetStatus(String.Format(
                CultureInfo.CurrentCulture,
                Properties.Resources.SaveFailed,
                user.FirstName, user.LastName));
        }
    }
}
}

```

Note the following important features:

- The Presenter does not itself implement any interfaces. It could do to formalise the services it offers to the view, but it doesn't. The justification is that the MVP (or MVC) pattern is used to allow different views or model implementations to be used with the same Presenter, so the latter is the one unchanging thing in this pattern; you are not likely to swap out one Presenter and replace it with another that has the same public interface. Interfaces are a tool, not a religion!
- The constructor takes two arguments, an `IManageUsersView` instance (that is, a concrete class that implements `IManageUsersView`) and an `IUserRepository` instance). In this way, the Presenter is linked to its View and its Model.
- The Presenter never refers to its' view or model using concrete classes such as `ManageUsersView` or `MemoryUsersRepository`, instead always using the interface types. In this way, each class only concerns itself with the functionality offered by an object, not by its implementation – a key win of interface-based development.
- The Search, Create and Save public methods implement the business logic of this component. In fact there are called by the View class (which itself has a reference to its Presenter because it implements the `IView<TPresenter>` interface as described in the previous section.
- Inside the Search, Create and Save business logic methods, it calls methods of the `IManageUsersView` instance or the `IUserRepository` instance in order to get the View and the Model to carry out actions, as required to implement the business logic.

Model Implementation

The Model is all contained within the Repositories folder in this sample. The interfaces are:

```

public interface IUserRepository : IRepository<User>
{
    /// <summary>
    /// Finds a user by flexibly matching either first or last name.
    /// </summary>
    IEnumerable<User> FindByName(string criteria);
}

```

```

    }

    public interface IRepository<TEntity>
        where TEntity : IEntity
    {
        string Create(TEntity entity);
        bool Update(TEntity entity);
        bool Delete(TEntity entity);
        bool Delete(string entityId);
        TEntity Get(string entityId);
        IEnumerable<TEntity> GetAll();
    }

    public interface IEntity
    {
        string Id { get; set; }
    }

```

`IRepository` and `IEntity` together define standard CRUD (Create, Read, Update, Delete) operations on business objects (or 'Entities') and the ability to get/set an entity ID. The `IUserRepository` interface then extends `IRepository` by adding a method called `FindByName` which adds some functionality that is specific to User entities.

In this sample, `IUserRepository` is implemented by `MemoryUserRepository`, which stores User entities in memory, instead of a SQL Server Compact database as the 'Poor' sample did. You could easily create a `SqlUserRepository` class that persists to SQL Server Compact and use that instead of the `MemoryUserRepository`.

Implementing the Plumbing – Connecting Modules to the Shell

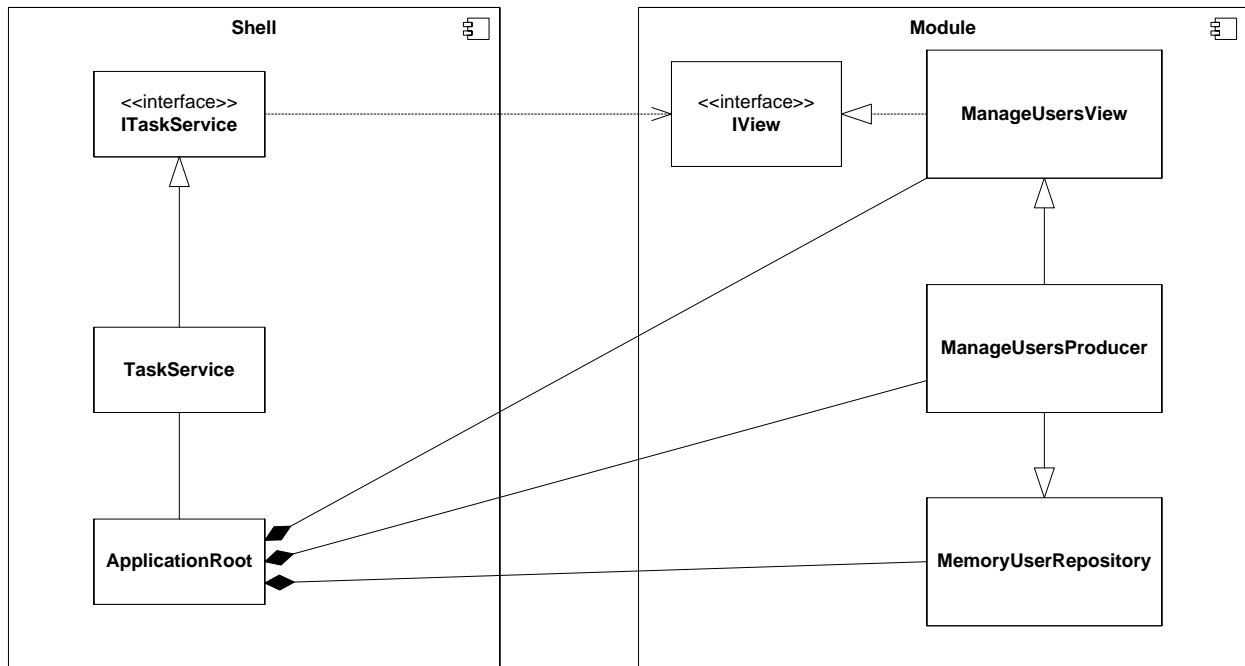
The `ManageUsersModule` is now a clearly defined separate component, but how do we launch it and how do we plug it into the shell?

The `MainForm` in `EditUsers-Better` contains only a `ListView` control, but the developer has not created any items inside the `ListView` during application development. It contains a `Go` menu item which is intended to launch a module, and an `Exit` menu item to quite the application.



The requirement is to provide a mechanism whereby any module can register itself with the shell so that the shell knows how to activate the services of the module. In this sample, we want to have an icon appear for each module in the ListView, and when the user selects the icon and clicks 'Go' the main view of the selected module displays.

We achieve this by defining two interfaces, one that the shell implements to define the methods that are used to register a module with the shell (called **ITaskService** in this sample), and one that each module implements to define the methods the shell can use to activate it (called **IView** in this sample). In UML, this looks like this:



This diagram expresses the relationships:

- The **ApplicationRoot** class is responsible for creating instances of the model, view and presenter of a module.
- The **ApplicationRoot** class registers the newly created module with the **TaskService** by using the methods of the **ITaskService** interface.
- The **ITaskService** interface itself has a dependency on the **IView** interface – in fact, it requires that any module you register with it implements the **IView** interface. Armed with the knowledge that any view class that is registered with it 'is a' **IView**, it now knows how to launch it (by using the method(s) defined by the **IView** interface).

Both the **ITaskService** and the **IView** interfaces are defined in a separate project (the **PluggableModulesInterfaces** project) so that both the shell project and each module project can reference it.

Task Service

The **ITaskService** interface defines the way in which modules are registered. Its **Register** method allows you to register a module, and its **Execute** method allows you to activate the module when required. By virtue of the fact that its **Register** method requires you to pass it an object that implements **IView**, it knows that it can activate the module by calling methods defined in **IView**.

```

/// <summary>
/// Service exposed by the shell that allows modules to publish tasks
/// they provide.
/// </summary>
public interface ITaskService
{
    /// <summary>
    /// Here we register an IView type, giving it a name and an icon.
  
```

```

        /// </summary>
        void Register(string taskName, IView view, System.Drawing.Icon
icon);

        /// <summary>
        /// Executes the task with the given name.
        /// </summary>
        void Execute(string taskName);

        /// <summary>
        /// Lists the registered Tasks
        /// </summary>
        Dictionary<string, TaskDetail> Tasks { get; }

    }

    public class TaskDetail
    {
        public Icon Icon { get; set; }
        public IView View { get; set; }
    }

```

As you can see, it contains methods to:

- Register an IView object along with a string name and an associated icon
- Execute a task (i.e. start a module running)
- Access the Dictionary of TaskDetail objects which contain the list of all registered tasks.

This interface is implemented by the **TaskService** class in EditUsers-Better, which is implemented as a singleton (see the code in Visual Studio for details).

How Modules Register with the Task Service and Module Instantiation

The very first code that runs in this application is in **Program.cs**, which calls `ApplicationRoot.Initialize()` and then starts up the message pump by calling `Application.Run` passing in the `MainForm` instance :

```

static class Program
{
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [MTAThread]
    static void Main()
    {
        ApplicationRoot.Initialize();
        Application.Run(new MainForm());
    }
}

```


`ApplicationRoot.Initialize()` is where we register all the modules in the system with the `TaskService`:

```
public static class ApplicationRoot
{
    public static MainForm MainForm;

    /// <summary>
    /// Sets up the modules in the application.
    /// </summary>
    public static void Initialize()
    {
        // Initialise the Manage Users module
        using (FileStream fs = new FileStream(
            new FileInfo(
                Assembly.GetExecutingAssembly().GetName().CodeBase
            ).DirectoryName + @"\ManageUsersIcon.ico", FileMode.Open))
        {
            ManageUsersView usersView = new ManageUsersView();
            TaskService.Instance.Register("Manage Users", usersView,
                new Icon(fs));

            // Initialise the Presenter for this view and wire the
            // presenter, model and view together
            usersView.Presenter =
                new ManageUsersPresenter(usersView,
                    new MemoryUserRepository());
        }

        //...any other modules
    }
}
```

This code creates a `FileStream` object to read in an icon for the `ManageUsersModule`, and then registers the `ManageUsersView` instance with the `TaskService`. It also creates an instance of `ManageUsersPresenter`, passing it the `ManageUsersView` instance and a new `MemoryUserRepository` instance.

Remember how the `ManageUsersPresenter` constructor took arguments of type `IManageUsersView` and `IUserRepository`, and never dealt with concrete types? This code in `ApplicationRoot.Initialize` is the only place in the whole application where actual concrete types are created, so this class is acting as a class factory to associate concrete types with their abstract interface types.

Note that in this sample the module registration is done here in code, but it could equally be done by reading an XML configuration file that defines the concrete classes to construct.

Activating a Module

In this sample, code in the `MainForm` class uses the `ITaskService` to discover what modules have been registered. It adds items to the `ListView` control displayed by the `MainForm` (see the `ShowTasks(ITaskService tasks)` method in `MainForm.cs`). At runtime, the `MainForm` displays the name and icon for each registered task like this:



When you click the 'Manage Users' item and then click the Go menu item, if you follow through the code that executes it ends up being actioned in the `Execute` method, part of the `ITaskService` interface and implemented by the `TaskService` class. In this sample, the `TaskService` invokes the `ShowDialog()` method on the main Form of the module:

```
public void Execute(string taskName)
{
    if (!Tasks.ContainsKey(taskName))
        return;

    // Execute the ShowDialog method for the IView
    Tasks[taskName].View.ShowDialog();
}
```

This mode of execution is specific to a Windows Forms-based solution, so may not be applicable to other solutions. Of course, other applications may choose different ways of exposing and navigating to tasks.

EditUsers-Better Summary

This sample shows how to create self-contained modules that are created to the Model – View – Controller pattern, and how to register them into a pluggable framework.

The use of interfaces to define contracts between components and between functional areas within components gives us clarity of design, the ability to change implementations. Also, although not demonstrated in these samples, it makes it much easier to unit test a component since you can easily create test scenarios where you test a class in isolation by linking it against minimal 'mock' test objects that implement any required interfaces with which the class under test expects to communicate. The mock objects implement only enough functionality to perform the test, giving repeatable and testable behaviour.

There are still a few areas in which we can improve the architecture by decoupling the components from each other even more than in this example, and this is covered by the last sample, EditUsers-Best.

Best Architecture

The EditUsers-Better sample can be improved by focusing on the TaskService, and the way in which objects are instantiated. You can raise the following criticisms of the EditUsers-Better sample:

- The shell is still fairly tightly coupled to each module. All the classes that make up each module – ManageUsersPresenter, ManageUsersView and MemoryUserRepository in the ManageUsers module in this sample – have to be instantiated during application startup, inside the ApplicationRoot.Initialize() method. The classes for a module may not be needed until sometime later, so this is wasteful of memory and will slow down application startup time. Wouldn't it be better if objects were only instantiated at the time they are needed at runtime?
- The shell project is the one that does the module instantiation, so it needs references to all application modules. While not a problem, this makes unit testing of any functionality contained within the shell function difficult, because you will have to supply mock objects for all interfaces in the whole system in order to get the shell object to run so that you can test it. It would be better if any module only had dependencies on interfaces it implemented or used, and that dependencies between modules were only resolved at runtime.

These problems are solved by use of a **Dependency injection Container**, which is used in EditUsers-Best. A new project has been added to the solution called **ContainerModel** which is a lightweight, fast dependency injection container for the .NET Compact Framework.

The solution is very little changed from EditUsers-Better. The EditUsers-Best version differs as follows:

- All interface types are registered with the D.I. container, along with factory functions that will be called to instantiate a particular concrete class that implements the interface. The instantiation is handled by the D.I. container the first time the application code needs to use it, not during application startup.
- You do not have to create an instance of any class to register them a module with the `ITaskService`. Instead you register the module name and icon as before, and you also register a function that will be called later on only when the application needs to call the module.
- Instead of the shell project containing code to instantiate and register a module, each module is self-registering so each module is responsible for registering its' own types and their factory functions with the D.I. container, and for registering with `ITaskService`. The `ApplicationRoot` class in the shell project now only registers the types in its own project, not the types of any other module. The shell literally needs just one line of code adding for each module to call the configuration code for that module.

These improvements are explained below.

Deferring Object Instantiation

During application startup, you register all your types and specify the function that will be required to create an instance of that type. In the case of an interface type, the factory function is one that creates a concrete implementation of that type.

For example, the code in `ApplicationRoot.Configure` that registers the `TaskService` as a singleton is as follows:

```
builder.Register<ITaskService>(c => new TaskService(c));
```

The defaults are used on this call to `Register<T>(Func<Container, TService> factory)` which means it is a singleton with global scope. Whenever any module needs to use an instance of the `ITaskService` type, they call `container.Resolve<ITaskService>()`. The first time any code makes this request to resolve the type, then the container will build one using the function defined in the

arguments to the `Register<TService>` call:

```
c => new TaskService(c)
```

What does that mean?

If you look at the `ContainerBuilder.Register<TService>` function definition in the source code

you find that it is a generic function defined as:

```
public IRegistration<TService> Register<TService>(Func<Container, TService> factory)
```

As you can see, the argument is of type `Func<Container, TService>`, which means a function that takes one argument of type `Container` and returns a `TService` (`TService` is the type you supply when calling this generic function).

The actual function supplied as the factory function for the `ITaskService` is `c => new TaskService(c)` which is a lambda function. This is actually shorthand for a factory function that looks like this:

```
ITaskService myAnonymousFactoryFn(Container c)
{
    return new TaskService(c);
}
```

The most important thing to understand about this is that no objects are created when you register a type with the D.I. container, you are simply defining a factory function that will be used – at some point – to create the type.

You can actually express dependency relationships between types. For example, the registration statement for the `MainForm` type is as follows:

```
// Register the MainForm type and its construction function which takes
// no arguments.
// Also define an Initialisation function that the container must call
// after construction.
// In this case, the init function sets the Presenter property of the
// new MainForm instance - a good example of using 'Setter injection'
builder
```

```
.Register(c => new MainForm())
.InitializedBy((c, v) => v.Presenter = c.Resolve<MainPresenter>());
```

This shows that in order to construct a `MainForm` instance, the container just needs to execute `new MainForm()`. But it also specifies an `InitializedBy` function that executes after the object is constructed, and that initialiser function means 'set the `Presenter` property of the type you just built to an instance of `MainPresenter`'. When the D.I.container comes to execute this code, it will resolve the `MainPresenter` type, creating it for the first time if necessary.

The `MainPresenter` type is of course also registered with the container using:

```
// Register the MainPresenter, the constructor of which takes arguments
// of a MainForm instance and the ITaskService instance.
// This is an example of using 'Constructor injection'
builder.Register(
    c => new MainPresenter(
        c.Resolve<MainForm>(),
        c.Resolve<ITaskService>()) );
```

so construction of that will in turn result in construction of a `MainForm` and an `ITaskService`.

Examine the code in `ApplicationRoot.cs` to see further examples.

Module Self-Registration

In this new sample, the `ManageUsersModule` is unchanged, apart from the addition of a single file called **`ManageUsersModuleConfigurator`** that handles registration of the types in the module with the applications D.I. container, and registration of the module entry point with the `ITaskService`. In a real application consisting of many modules, all modules will have a similar configurator class which implements `IContainerModule`, which contains a single method `Configure(Container container)`.

The shell project calls a module configurator with simple code that constructs an instance of the configurator class, then calls its `Configure` method, for example:

```
// Call the configuration method for the ManageUsers module
new ManageUsersModuleConfigurator(container.Resolve<ITaskService>())
    .Configure(container);
```

Registering with the Task Service

The `TaskService` is changed a little from `EditUsers-Better`. It now uses similar techniques to those described in the previous section to register an `Action` function to be executed when the module is to be invoked, instead of registering an actual instance of the type to be executed. In this way, the objects in the module are created only when they are needed and not at application startup. It is left as an exercise to the reader to examine `TaskService.cs` to see how this is implemented.

The code to register with the Task service is no longer in the shell program, but now is entirely inside the `ManageUsersModuleConfigurator` class:

```
private void RegisterTasks(Container container)
{
    // This is the entry point to our module.
    using (FileStream fs = new FileStream(new
        FileInfo(Assembly.GetExecutingAssembly().GetName().CodeBase)
        .DirectoryName + @"ManageUsersIcon.ico", FileMode.Open))
    {
        tasks.Register<IManageUsersView>("Edit Customers",
            new Icon(fs));
    }
}
```

Registering Types with the D.I. Container

The other thing the `ManageUsersModuleConfigurator` class does is register all its interface types and their factory functions with the D.I. container. In this way, all the code that associates a concrete type with a particular interface is contained within this one class.

For example, the registration code for the `IUserRepository` type looks like this:

```
// Register the IUserRepository type, with its factory function and
// also (just for fun) some test data entered through an InitializedBy
// function.
builder
    .Register<IUserRepository>((
        c => new MemoryUserRepository()))
    .InitializedBy((c, r) =>
    {
        // Initialize with some fake data.
        r.Create(new User { UserName = "wigley_1",
            FirstName = "Andy",
            LastName = "Wigley"
        });
        r.Create(new User { UserName = "blackb_1",
            FirstName = "Andrew",
            LastName = "Blackburn"
        });
    });
```

EditUsers-Best Summary

This sample improves upon `EditUsers-Better` through use of a Dependency Injection container. On the downside, usage of a D.I. container is yet another framework to learn, and does introduce an element of code obscurity since the exact time of an objects' instantiation can no longer be

understood simply by reading the code looking for a `new` keyword. However, once you understand how to register types and use the `Resolve<T>()` method to get a reference to a type, it is really quite easy to use. Proper use of a D. I. Container can in fact decrease overall code complexity.

By adopting loose coupling techniques such as these, you will find it easier to:

- Develop self-contained modules
- Plug them into an application framework
- Produce applications that are efficient in their memory usage and (most importantly) run faster
- Create and execute unit tests