**Inversion of Control and MVC**

[**http://msdn.microsoft.com/rampup**](http://msdn.microsoft.com/rampup)

Contents

[Introduction 3](#_Toc264642469)

[Inversion of Control 3](#_Toc264642470)

[Dependency Injection 4](#_Toc264642471)

[Model View Controller 5](#_Toc264642472)

[Summary 7](#_Toc264642473)

# Introduction

This unit discusses two separate topics: Inversion of Control and Model-View-Controller.

# Inversion of Control

In the early days of programming, developers thought in terms of procedural logic. Code was organized in terms of dividing the problem into more manageable chunks.[[1]](#footnote-1) Thinking along these lines led people to layer programs so that the higher layers of the program controlled the program logic.

The problem with that approach is that it increased the coupling of the application because applications were written with specific implementations in mind. If an algorithm had to be changed, at the very minimum a massive recompile was required.[[2]](#footnote-2)

We have already illustrated in the preceding units how interfaces allow us to write an application that depends on behavior, without using a specific implementation. We can take this one step further using the idea of inversion of control.

Inversion of control pushes the details of the implementation to the lower layers of the application. By doing so, it minimizes the coupling of the application because the application is held together through abstract definitions of behavior in an interface. When we replace one type of concrete class with another, there is no structural effect on the rest of the system because the dependency is on the interface they both implement.

Remember, however, we are talking about minimizing inessential coupling. If you replace one concrete class with another there could be a huge affect on the system because the algorithms implementing the behavior could be widely different.

It is important to emphasize this point. People often claim that using inversion of control, or other layering techniques means that the system does not make assumptions about what other parts of the system can or should do. This is only a partially true. Parts of the system have to make assumptions about other parts of the system to which they are essentially coupled. A reporting system in an accounting application can run without crashing even though the journaling system does not work. The reports, however, make no sense. This is little comfort to the user. A properly structured application helps the developer fix the bug quickly with minimal repercussions, but the user may or may not care about that.

Where are the concrete instances created? This knowledge must exist somewhere. We have already seen examples of inversion of control. With the data mapping layer we have used a factory to create the concrete class. This concrete class was passed as an interface to the constructor of the CustomerManager class. For the CustomerRepository class, we created an instance in the CustomerManager class, and passed the IDataMapper interface to its constructor.

## Dependency Injection

Instead of using a factory to achieve inversion of control, you can use a dependency injection container. The container, rather than the factory decides which concrete classes get created. A factory, no matter how complicated, is created at compile time. A container, as will be explained, can allow for the decision as to which class is created occur at runtime. This would, for example, facilitate testing where you wanted to substitute a test class for the real class. The real class might work against a service that charges money, or takes too long for rapid unit tests. A container might also allow you inject code for a cross-cutting concern such as logging. The container is driven by a configuration mechanism that it uses to determine which classes to instantiate at runtime. Very often the application is not aware that a dependency injection mechanism is being used. This is the ultimate in decoupling because the decision on instantiation happens at runtime.

This unit uses very simple dependency injection mechanism to illustrate how this works. With the knowledge gained in looking at this simple example, you will be better informed to determine when it is appropriate to use this mechanism.

Our dependency injection container is implemented as a very simple plug in manager in a class called, appropriately enough, PluginManager. Its only method is LoadType.

 static public object LoadType(string originalType)

 {

 string substituteType = ConfigurationManager.AppSettings.Get(originalType);

 Assembly calling = Assembly.GetCallingAssembly();

 Type[] types = calling.GetExportedTypes();

 foreach (Type t in types)

 {

 if (t.Name == substituteType)

 {

 object type = calling.CreateInstance(t.Name);

 return type;

 }

 }

 return null;

}

An argument is passed into the method indicating the type that the application wants to instantiate. It looks up this type in a configuration file to get the name of the type that it should instantiate. This may or may not be the same type passed in. For simplicity we are using the application’s configuration file. A real dependency injection container might use its own configuration file or some other lookup mechanism.

It uses reflection to get this type. For simplicity, we only look in an assembly that calls the plug-in manager. A real dependency injection container could load any type it needed. Once this type is loaded it is instantiated and returned to the calling method.

In our example, the CustomerManager class uses the PluginManager:

dm = (ICustomerDataMapper)PluginManager.LoadType("CustomerDataMapper");

The CustomerManager class is relying on the PluginManager to instantiate the proper class. In the configuration file for the application, we have the entry:

<add key="CustomerDataMapper" value="CustomerDataMapper" />

Here, the application would load the CustomerDataMapper class we discussed in the last unit. In the configuration file for the CustomerManager tests, we have this entry:

<add key="CustomerDataMapper" value="TestCustomerDataMapper" />

The test loads a different version of the mapper. The code only relies on the behavior in the ICustomerDataMapper class, and not a concrete implementation. If you compare these two versions of the data mapper, you will see that they just implement the same interface differently. Hence, it is easy to introduce mocks and stubs into your test application.

You can also now see the problems with dependency injection containers. It introduces a large level of indirection at the cost of complexity. With such a container it could often be difficult to figure out which class is actually being instantiated. In a debugging session you cannot step through the configuration file. Stepping through the code in a real container to figure out what is actually happening can be quite difficult. On the other hand it can be easier than writing factories, and might simplify your application if used appropriately.

Since using a dependency injection container allows you great flexibility in building tests for individual parts of an application, many developers feel dependency injection should be used throughout an application. To me this is a judgment call. How much testing and code coverage you need is a tradeoff compared with other project constraints (code complexity, time, money, need to deliver functionality in a timely manner). Too much dependency injection complicates an application to the point where it can be hard to decipher what is going on. Dependency injection works best at component boundaries because that is where flexibility is needed, and fragility in the application exists. In the second unit of this track we discussed where component boundaries lie in an application.

# Model View Controller

One of the most important isolation layers in an application is between the domain model and the user interface. Typically, user interfaces are hard to test, so separating the two makes it easier to test the business logic independent of the user interface. Further, an application might have multiple user interfaces for the same domain model. It also allows the user interface to evolve without impacting the domain model.

The Model-View-Controller (MVC) pattern is one of the most fundamental patterns for separating the user interface from the business logic in an application. Our initial application for this pattern will be in our console application.

The *controller* is responsible for handling the input events from the user interface. The *view* represents the user interface. The *model* represents the business logic associated with the application. By separating the model from the view you can test the model without the user interface. It allows you to evolve the user interface and the business logic separately.

Our model is represented in the class *ModelCustomer*. It has 5 static member functions:

AddCustomer

AddOrderForCustomer

GetCustomer

GetCustomer

GetCustomers.

The member variables represent the data associated with a particular customer. If you look inside the implementation of the ModelCustomer class you will see no calls to any user interface input or output.

The ICustomerView interface has functions to allow it to interact with the view:

public interface ICustomerView

{

 void SetController(CustomerController controller);

 void WriteCustomer(IList<ModelCustomer> customers);

}

SetController associates a particular controller with the view so that the view can receive user input. WriteCustomer allows the view to produce output. Note that while the model knows nothing about the view, any view has to know about the model. The view had to know what information it is displaying or what actions have to be undertaken. The view also has member functions corresponding to the appropriate user interface actions. Because of the simplicity of our application, this corresponds pretty closely to the model functionality, but this does not have to be the case in a more sophisticated application:

AddNewCustomer

RetrieveById

RetrieveByName

RetrieveAllCustomers

EnterOrderForCustomer

The main routine then can be rewritten to use the controller.[[3]](#footnote-3)

 if (result == 1)

 break;

 else if (result == 2)

 controller.AddNewCustomer();

 else if (result == 3)

 controller.RetrieveById();

 else if (result == 4)

 controller.RetrieveByName();

 else if (result == 5)

 controller.RetrieveAllCustomers();

 else if ( result == 6)

 controller.EnterOrderForCustomer();

The controller’s constructor is associated with the view it is responsible for:

public CustomerController(ICustomerView view)

{

this.view = view;

 view.SetController(this);

}

The application initialization associates the controller and view:

 CustomerView view = new CustomerView();

 CustomerController controller = new CustomerController(view);

In the next unit we will demonstrate the flexibility of this approach when we convert the UI, keeping the model exactly the same.

# Summary

In this unit we have covered two very important concepts: inversion of control and model-view-controller (MVC).

The first allows for an application to be written so that the application is not coupled together at the highest layers of the application. The lower layers of the application can then be written in a flexible way using interfaces. We noted in the previous unit that we used inversion of control. Here we expanded this notion to include dependency injection. Dependency injection is a powerful way to introduce flexiblity into an application, but has to be used judiciously.

Model-View-Controller decouples the model and the view. Hence the business logic (model) can be test and evolved independently of the user interface.

In the next unit we will use the foundation we have built. To convert our application into a three tier web app that uses a relational database and web services.

1. Niklaus Wirth’s Algorithms + Data Structure s = Programs (1976)’ is a classic. He views programming as a process of stepwise refinement. [↑](#footnote-ref-1)
2. C/C++ programmers of a certain generation still remember include file hell. [↑](#footnote-ref-2)
3. We cannot use callbacks here because we are in a console application. [↑](#footnote-ref-3)