Dependency Injection and Inversion of Control

Global Knowledge®

Objectives

- Dependency Injection
- Inversion of Control
- IoC containers

Dependency Injection (DI)

- Coding against abstractions is generally good idea
  - can use interfaces or abstract base class
  - easier for software to evolve and adapt
- Classes provided implementation of abstraction
  - class code uses abstraction, not concrete type
- Alter class behaviour without changing code
  - pass in different implementation of abstraction
  - dependency is passed in, or “injected”

Injecting dependencies

- Five ways to inject dependencies
  - constructor injection
  - method call injection
  - property injection
  - interface injection
  - ambient context

Dependency injection issues

- Concrete class still has to be coded somewhere
  - changing implementation means changing code
- Solve problem by specifying types in configuration
  - use factory class that reads configuration

```
public interface IToDoListRepositoryFactory
{
   IToDoListRepository CreateRepository();
}

public class ToDoListRepositoryFactory : IToDoListRepositoryFactory
{
   public IToDoListRepository CreateRepository()
   {
      string connectionString = ConfigurationManager.ConnectionStrings[
         "toDoListRepository" ].ConnectionString;

      return new ToDoListRepository( connectionString );
   }
}
```

Factory class issues

- Complex construction is relatively hard
  - constructor parameters
  - dependencies between types
  - results in lots of configuration

Inversion of Control (IoC)

- IoC containers
  - manage creation and disposal of dependencies
- IoC principle
  - container understands dependencies and constructs objects
  - normally class constructs dependencies (class is in control)
  - this is opposite, or “inverse” (container is in control)
  - hence being called Inversion of Control (IoC)
- Dependency injection is one consequence of IoC

IoC containers

- Many available
  - Unity
  - Castle Windsor
  - Structure Map
  - Spring.NET

Using IoC containers

- Container typically single instance
  - essentially dictionary of interface => concrete implementation
- Types are registered with container
  - in configuration file
  - or in code
- Container resolves types as needed

Registering types in code

- Call Register method of container

```
IUnityContainer unityContainer = new UnityContainer();

unityContainer.RegisterType<IToDoListRepository, ToDoListRepository>();
```

Registering types in configuration – 1

- Register type mappings
  - can use alias to shorten registration

```
<unity xmlns="http://schemas.microsoft.com/practices/2010/unity">

   <alias alias="IToDoListRepository"
      type="ToDoList.IToDoListRepository, ToDoList" />
   <alias alias="ToDoListRepository"
      type="ToDoList.ToDoListRepository, ToDoList" />

   <container>

      <register type="IToDoListRepository" mapTo="ToDoListRepository">
         <constructor>
            <param name="connectionString" value="metadata=..." />
         </constructor>
      </register>

   </container>

</unity>
```

Registering types in configuration – 2

- Alternatively, can register namespaces and assemblies

```
<unity xmlns="http://schemas.microsoft.com/practices/2010/unity">

   <namespace name="ToDoList" />
   <namespace name="System.Security.Principal" />

   <assembly name="ToDoList" />
   <assembly name="mscorlib" />

   <container>

      <register type="IToDoListRepository" mapTo="ToDoListRepository">
         <constructor>
            <param name="connectionString" value="metadata=..." />
         </constructor>
      </register>

   </container>

</unity>
```

Object lifetimes

- Objects lifetime can be controlled
  - six lifetime managers provided as standard
  - can create custom lifetime managers

```
<unity xmlns="http://schemas.microsoft.com/practices/2010/unity">

   <!-- ... -->

   <container>

      <register type="IToDoListRepository" mapTo="ToDoListRepository">
         <lifetime type="singleton" />
         <constructor>
            <param name="connectionString" value="metadata=..." />
         </constructor>
      </register>

   </container>

</unity>
```

Parameterised constructors

- Unity will automatically find types
  - will use constructor with most parameters by default
  - can specify constructor to use by specifying parameters

```xml
<register type="IToDoListRepository" mapTo="ToDoListRepository">
   <constructor>
      <param name="connectionString" value="metadata=..." />
   </constructor>
</register>

<register type="ToDoListController">
   <constructor>
      <param name="repository" />
      <param name="pageSize" value="10" />
   </constructor>
</register>
```

```csharp
public ToDoListRepository( string connectionString )
   : base( connectionString )
// ...

public ToDoListController(
   IToDoListRepository repository, int pageSize )
// ...
```

Registering parameters in code

- Create instance of InjectionConstructor
  - pass to Register method

```
InjectionConstructor injectionConstructor =
   new InjectionConstructor( connectionString );

unityContainer.RegisterType<IToDoListRepository, ToDoListRepository>(
   injectionConstructor );
```

Registering parameters in configuration

- Can also define parameter values in configuration

```
<register type="IToDoListRepository" mapTo="ToDoListRepository">
   <constructor>
      <param name="connectionString" value="metadata=..." />
   </constructor>
</register>

<register type="ToDoListController">
   <constructor>
      <param name="repository" />
      <param name="pageSize" value="10" />
   </constructor>
</register>
```

Resolving types

- Use instance of container
  - configure container
  - call its Resolve method

```
IUnityContainer unityContainer = new UnityContainer();

UnityConfigurationSection section =
ConfigurationManager.GetSection( "unity" ) as UnityConfigurationSection;
if ( section != null )
   section.Configure( unityContainer );

IToDoListRepository repository =
   unityContainer.Resolve<IToDoListRepository>();

// Or:

IUnityContainer unityContainer = new UnityContainer();

unityContainer.LoadConfiguration();

IToDoListRepository repository =
   unityContainer.Resolve<IToDoListRepository>();
```

Named type mappings

- Can name mappings
  - allows multiple implementations of interface
  - can choose implementation at run time

```
<register type="IToDoListSecurityService"
   mapTo="WinToDoListSecurityService" name="Windows" />

<register type="IToDoListSecurityService"
   mapTo="AspToDoListSecurityService" name="ASP.NET" />
```

```
IToDoListSecurityService toDoListSecurityService =
   unityContainer.Resolve<IToDoListSecurityService>( "Windows" );
```

Summary

- Inversion of Control lets framework control instantiation
- Allows for greater flexibility
- Types can be injected into code
- Containers make this easy through configuration