Dependency Injection & Mocking

Equuleus Technologies

What is Dependency Injection?

This is a pattern used for decoupling components and layers in the system. The pattern is implemented through injecting dependencies into a component when it is constructed. These dependences are usually provided as interfaces for further decoupling and to support testability.

The sources used for this exercise can be found on the final slide.

Pure Dependency Injection (Poor man's DI)

Pure Dependency Injection uses constructor or property Injection where lower level components are passed using constructors or properties.

```
var svc = new ShippingService(new ProductLocator(),
   new PricingService(), new InventoryService(),
   new TrackingRepository(new ConfigProvider()),
   new Logger(new EmailLogger(new ConfigProvider())));
```

Equuleus Technologies

DI container

DI Container simplifies the creation of Compostion Roots especially for dependencies having complex setup scenarios.

DI Container also provides benefits like Interception (for Aspect oriented programming) for cross-cutting concerns like logging, caching security etc.

```
var svc = IoC.Resolve<IShippingService>();
```

Equuleus Technologies

Comparing DI Containers

	Autofac	Dryloc	Simple Injector	Unity
Configuration	XML/Auto		Auto	XML/Auto
Custom Lifetimes		Yes	Yes	Yes
Interception	Yes		Yes	Yes
Auto Diagnostics			Yes	
Speed	Average	Fast	Fast	Average

Equuleus Technologies

Pros & Cons

	Advantage	Disadvantages
Pure DI	Easy to learn Strongly typed Rapid feedback	High maintenance
DI Container	Low maintenance Convention registration Xml registration	Weakly typed Harder to learn Slower feedback

Equuleus Technologies

DI Container Capabilities to Consider

List/array dependencies - When several registrations of IService exist in a container, many frameworks automatically provide IService[] / List<IService>.

- Autofac/DryIoc/SimpleInjector have a better support than Unity

Open Generics - Service<> can be registered as IService<>, and then any request for IService<X> should be resolved with Service<X>.

Optional parameters - Optional parameters can be used to specify an optional dependency

- Autofac/DryIoc have better support than SimpleInjector/Unity

Func & Lazy support - When TService is registered in a container, many frameworks automatically provide Func<..., TService> & Lazy<TService>

- DryIoc > Autofac> Unity > SimpleInjector

Covariance & Contravariance

- Varying capabilities with different containers.

Equuleus Technologies

Unity

Unity was originally developed by Microsoft, though it is no longer supported by Microsoft and it is currently maintained by the community.

Unity allows for configuration using both code and xml, configuration of the containers is similar and swapping them out is a fairly straightforward task.

We are using unity for our examples

Unity with XML configuration/ServiceLocator

Code

```
UnityContainer container = new UnityContainer();

// load xml configuration
container.LoadConfiguration();

// register with the service location
ServiceLocator.SetLocatorProvider(() => new UnityServiceLocator(container));

// resolving using the service locator
IClient client = ServiceLocator.Current.GetInstance<IClient>();
```

Configuration using Code

```
container.RegisterType<IService, SomeService>();
Container.RegisterType<IClient, SomeClient>();
```

Resolving using the container instead of Service Locator

```
IClient client = container.Resolve<IClient>();
```

Equuleus Technologies

Unity Configuration - XML

```
<configSections>
  <section name="unity" type="Microsoft.Practices.Unity.Configuration.UnityConfigurationSection,
           Microsoft.Practices.Unity.Configuration" />
</configSections>
<unity xmlns="http://schemas.microsoft.com/practices/2010/unity">
  <sectionExtension
   type="Microsoft.Practices.Unity.InterceptionExtension.Configuration.InterceptionConfigurationExtension,
       Microsoft.Practices.Unity.Interception.Configuration" />
  <container>
    <extension type="Interception"/>
    <register type="ClassLibrary.IClient, ClassLibrary" mapTo="ClassLibrary.SomeClient, ClassLibrary">
      <interceptor type="InterfaceInterceptor"/>
      <interceptionBehavior type="Unity.LoggingInterceptionBehavior, Unity"/>
    </register>
    <register type="ClassLibrary.IService, ClassLibrary"  mapTo="ClassLibrary.SomeService, ClassLibrary">
    </register>
  </container>
</unity>
```

Equuleus Technologies

Unity Interception

```
public class LoggingInterceptionBehavior : IInterceptionBehavior
{

    void WriteLog(string message)...

    //returns the interfaces required by the behavior for the intercepted objects
    public IEnumerable<Type> GetRequiredInterfaces()...

    // The Invoke method is required to actually execute the behavior logic
    public IMethodReturn Invoke(IMethodInvocation input, GetNextInterceptionBehaviorDelegate getNext)...

    //It simply returns a flag indicating if the behavior will actually do anything when invoked,
    //and if not, enables the interception mechanism to skip the behavior
    public bool WillExecute...
}
```

Equuleus Technologies

Mocking

The correct way of unit testing is to isolate the behavior of the object we want to test from its external dependencies. This way we can get easily and comprehensively test all code paths to verify system behavior.

Mock is an object that simulates the behavior of a real method/object in controlled ways. We replace the other objects by mocks that simulate the behavior of the real objects.

Moq

Moq is a free mocking library that helps in writing tests, it has a clean fluent interface that makes it easy to use.

```
var tester = new Mock<ITester>();
// Setup a callback for a void method. -------
tester.Setup(t => t.Void(It.IsAny<string>())).Callback<string>(s => voidArg = s);
// Setup the result of a method. -------------
tester.Setup(t => t.Bool()).Returns(true);
// Setup that throws an exception ------------
tester.Setup(t => t.Exception(It.IsAny<string>)) .Throws(new Exception());
// Ensure that a function was called. --------
tester.Verify(m => m.Void("A"), Times.Once);
// Ensure that a function was NOT called. ----
tester.Verify(m => m.Void("B"), Times.Never);
```

Equuleus Technologies

AutoFixtures

AutoFixtures helps in the setup of data for unit tests, this way developers can focus on data that is tested rather than how to setup the test scenario, by making it easier to create objects graphs containing test data

```
// Fixture setup with Moq integration

var fixture = new Fixture().Customize(new AutoMoqCustomization());

// Freeze returns the same Mock<IDAL> implementation, the one with expectations

Mock<IDAL> mapMock = fixture.Freeze<Mock<IDAL>>();

// Creates an anonymous variable of the type ClassA

// If the constructor below is parameterized that takes in an implementation of IDAL, it will use
the frozen mock object along with other parameters

ClassA objectA = fixture.Create<ClassA>();
```

Equuleus Technologies

Shouldly

Shouldly provides better assertion than what the testing frameworks provide.

Improved test code readability.

```
Shouldly - contestant.Points.ShouldBe(1337);

Test Framework - Assert.That(contestant.Points, Is.EqualTo(1337));
```

Better test failure messages.

```
Shouldly - contestant.Points should be 1337 but was 0

Test Framework - Expected 1337 but was 0
```

Equuleus Technologies

Resources

Dependency Injection & Mocking – code samples

DI_Mock.zip

Equuleus Technologies