DEPENDENCY MANAGEMENT

MICHAEL JONES

MICROSOFT SENIOR SOFTWARE ENGINEER

mijones@Microsoft.com

OUTLINE

- My Path to Caring about Dependency Management
- Definitions
- Software Development Goals
- Industrial Strength Development
- Dependency Management Patterns & Antipatterns
- Dependency Injection Containers
- Components and Modules
- Putting it All Together

MY PATH TO DEPENDENCY MANAGEMENT

- Microsoft Patterns and Practices Unity (2008 - 2013)
  - https://msdn.microsoft.com/en-us/library/ff647202.aspx
- Mark Seemann
  - Book: Dependency Injection in .NET (2011, 2018)
  - Blog: http://blog.ploeh.dk/about/
- Castle Windsor (active)
  - http://www.castleproject.org/projects/windsor/
- Autofac (active)
  - https://autofac.org
- Martin Fowler
  - Book: Patterns of Enterprise Application Architecture (2002)

DEFINITIONS

- Component:

  An implementation of an single abstraction with the intent of being provided as a dependency to other components.

  Examples:

  - BasicSettingsProvider_AppConfig which implements IBasicSettingsProvider
  - TypedSettingsProvider which implements ITypedSettingsProvider

- Module:

  A collection of component choices used to satisfy most dependencies of a larger application function.*

  Examples:

  - AzureCloudServiceConfigurationDependencies which inherits from ConfigurationDependencies to provide most of the dependencies needed to read configuration settings from an Azure cloud service deployment.

    i.e. Specific IBasicSettingsProvider, ITypeConversionProvider, and various IConvert components

SOFTWARE DEVELOPMENT GOALS

I want to build:

- Succinct
- Configurable / Customizable / Adaptable
- Reproducible

Enterprise applications on top of industrial strength components.

INDUSTRIAL STRENGTH

Characteristics:

- Well Designed
- Well Tested
- Durable
- Performant
- Secure
- Transparent
- Ready to Work

Practical Principles:

- Dev Friendly
- Test Friendly
- Ops Friendly

Design Principles:

- SOLID
- DRY

SOLID

- Single Responsibility
  - Well defined responsibilities lead to more simple building blocks (components)
- Open for Extension, Closed for Modification
  - Prefer immutable interface abstractions
- Liskov Substitution Principle
  - Replaceable with continued correctness
  - * Desirable but difficult. Do not let this deter you.
- Interface Segregation Principle
  - Many specific interfaces are better than fewer general ones
  - * Type parameterized interfaces help keep you DRY
- Dependency Inversion Principle
  - Depend on abstractions, not concretions
  - * Proper interface separation forces you to comply

DON'T REPEAT YOURSELF

- Single Responsibility
- Configurable Abstractions
- Define Reusable *Baseline* Abstractions to Build From
  - Start with abstractions better than Tuples:
    - `interface IResult<out StatusT> { … }`
    - `interface IResult<out ValueT, out StatusT> { … }`
    - `interface IResult<out KeyT, out ValueT, out StatusT> { … }`
- Implement Common Implementations
  - `class Result : IResult<HttpStatusCode> { }`
  - `class Result<ValueT> : IResult<ValueT, HttpStatusCode> { }`
  - `class Result<KeyT, ValueT> : IResult<KeyT, ValueT, HttpStatusCode> { }`

ADD MORE TO THE BASELINE

```
public interface IBasicSettingsProvider<KeyT, BasicT>
{
    BasicT GetBasicSetting(KeyT key);
}
```

```
public class BasicSettingsProvider_AppConfig
    : IBasicSettingsProvider<string, string> { … }

public class BasicSettingsProvider_WebConfig
    : IBasicSettingsProvider<string, string> { … }

public class BasicSettingsProvider_Dictionary
    : IBasicSettingsProvider<string, string> { … }

public class BasicSettingsProvider_Json
    : IBasicSettingsProvider<string, string> { … }

public class BasicSettingsProvider_Xml
    : IBasicSettingsProvider<string, string> { … }

public class BasicSettingsProvider_Cli
    : IBasicSettingsProvider<string, string> { … }

public class BasicSettingsProvider_Database
    : IBasicSettingsProvider<string, string> { … }
```

BUILD COMPLEXITY FROM BASELINES

```
public interface IConvert<FromT, ToT>
{
    ToT Convert(FromT from);
}

public interface ITypeConversionProvider<BasicT>
{
    IConvert<BasicT, ToT> GetTypeConversion<ToT>();
}

public interface ITypedSettingsProvider<KeyT>
{
    T GetTypedSetting<T>(KeyT key);
}
```

```
public class TypedSettingsProvider<KeyT, BasicT>
    : ITypedSettingsProvider<KeyT>
{
    public TypedSettingsProvider(
        IBasicSettingsProvider<KeyT, BasicT> basicSettingsProvider,
        ITypeConversionProvider<BasicT> typeConversionProvider)
    {
        this.BasicSettingsProvider = basicSettingsProvider;
        this.TypeConversionProvider = typeConversionProvider;
    }

    private IBasicSettingsProvider<KeyT, BasicT> BasicSettingsProvider { get; }
    private ITypeConversionProvider<BasicT> TypeConversionProvider { get; }

    public T GetTypedSetting<T>(KeyT key)
    {
        var basicValue = this.BasicSettingsProvider.GetBasicSetting(key);
        var cvt = this.TypeConversionProvider.GetTypeConversion<T>();
        return cvt.Convert(basicValue);
    }
}
```

LAYER COMPLEXITY

```
public class KeyTransformedTypedSettingsProvider<KeyT>
    : ITypedSettingsProvider<KeyT>
{
    public KeyTransformedTypedSettingsProvider(
        Func<KeyT, KeyT> transformKey,
        ITypedSettingsProvider<KeyT> typedSettingsProvider)
    {
        this.TransformKey = transformKey;
        this.TypedSettingsProvider = typedSettingsProvider;
    }

    private Func<string, string> TransformKey { get; }
    private ITypedSettingsProvider<KeyT> TypedSettingsProvider { get; }

    public T GetTypedSetting<T>(KeyT key)
    {
        return this.TypedSettingsProvider.GetTypedSetting<T>(this.TransformName(key));
    }
}
```

```
public class SectionScopedTypedSettingsProvider<KeyT>
    : KeyTransformedTypedSettingsProvider<KeyT>
{
    public SectionScopedTypedSettingsProvider(SectionPath<KeyT> section,
        ITypedSettingsProvider<KeyT> typedSettingsProvider)
            : base(key => section.FullKey(key), typedSettingsProvider)
    { }
}
```

COMPLEXITY IN USE

```
public interface IMyComponentSettings
{
    string StringValue { get; }
    int IntValue { get; }
    TimeSpan TimeSpanValue { get; }
}
```

```
public class MyComponentSettings
    : IMyComponentSettings
{
    public string StringValue { get; set; }
    public int IntValue { get; set; }
    public TimeSpan TimeSpanValue { get; set; }
}

public class MyComponentSettingsFromSettingsProvider
    : IMyComponentSettings
{
    public MyComponentSettingsFromSettingsProvider(ITypedSettingsProvider<string> tsp)
    {
        this.Tsp = tsp;
    }

    private ITypedSettingsProvider<string> Tsp { get; }

    public string StringValue { get { return this.Tsp.GetTypedSetting<string>(nameof(this.StringValue)); } }
    public int IntValue { get { return this.Tsp.GetTypedSetting<int>(nameof(this.IntValue)); } }
    public TimeSpan TimeSpanValue { get { return this.Tsp.GetTypedSetting<TimeSpan>(nameof(this.StringValue)); } }
}
```

COMPLEXITY IN USE

```
public interface IMyComponent { }


public class MyComponent
    : IMyComponent
{
    public MyComponent(IMyComponentSettings myComponentSettings)
    {
        this.MyComponentSettings = myComponentSettings;
    }

    private IMyComponentSettings MyComponentSettings { get; }
}
```

```
new MyComponent(
    new MyComponentSettings
    {
        StringValue = "Foo",
        IntValue = 1,
        TimeSpanValue = TimeSpan.FromMinutes(5)
    }
);
```

```
new MyComponent(
    new MyComponentSettingsFromSettingsProvider(
        new SectionScopedTypedSettingsProvider(
            "Path.To.MyComponent",
            new TypedSettingsProvider<string, string>(
                new BasicSettingsProvider_AppConfig(),
                new SystemChangeTypeConversionProvider<string>()
            )
        )
    )
);
```

MORE BASELINE ABSTRACTIONS

Settings/Data Conversion:

- `BasicT IBasicSettingsProvider<KeyT, BasicT>.GetBasicSetting(KeyT)`
- `ToT IConvert<FromT, ToT>.Convert(FromT from)`
- `IConvert<FromT, ToT> ITypeConversionProvider<FromT>.GetTypeConversion<ToT>()`
- `T ITypedSettingsProvider<KeyT>.GetTypedSetting<T>(KeyT key)`

Factories/Data Access:

- `IResult<ProductT> IFactory<ArgT, ProductT>.Create(ArgT arg)`
- `IResult<KeyT, ResourceT> IResourceProvider<KeyT, ResourceT>.Provide(KeyT key)`

Formatting:

- `MessageT IItemFormatter<ItemT, MessageT>.Format(ItemT item)`
- `string IItemToString<ItemT>.ToString(ItemT item)`

Filtering:

- `void IApplicationOf<OfT>.Apply<ToT>(ToT to)`

SOME BASELINE IMPLEMENTATIONS

- Implement Standard Concretions
  - ```
    class OrderFactory
        : IFactory<OrderArgs, Order> { }
    ```
  - ```
    class DictionaryResourceProvider<KeyT, ResourceT>
        : IResourceProvider<KeyT, ResourceT> { }
    ```
  - ```
    class ConvertChangeType
        : IConvert { ToT Convert<FromT, ToT>(FromT from); }
    ```
  - ```
    class AttributesFormatXml<ItemT>
        : IItemFormatter<ItemT, XmlNode> { XmlNode Format(ItemT item); }
    ```
  - ```
    class ToString<ItemT>
        : IItemToString<ItemT> { string ToString(ItemT item); }
    ```
  - ```
    class ApplicationOfActions<ItemT>
        : IApplicationOf<OfT> { void Apply<ItemT>(ItemT item); }
    ```

INDUSTRIAL STRENGTH COMPONENTS

Keep extending this pattern

- `interface IDeploymentInfo`
  - `class DeploymentInfoFromSettingsProvider`
  - `class AzureDeploymentInfo`
- `interface IDiagnostics<>`
  - `class LocalDiagnostics`
  - `class AzureDiagnostics`
- `interface IResourceProvider<string, ILocalResource>`
  - `class LocalResourceMap`
  - `class AzureLocalResourceProvider`
- `interface IExternalService`
  - `class ExternalService`
  - `class ExternalServiceStub`

SOLID/DRY REPEAT!

- Single Responsibility
  - Well defined responsibilities lead to more simple building blocks (components)
- Open for Extension, Closed for Modification
  - Prefer immutable interfaces
- Liskov Substitution Principle
  - Replaceable with continued correctness
  - * Desirable but difficult. Do not let this deter you.
- Interface Segregation Principle
  - Many specific interfaces are better than fewer general ones
  - * Type parameterized interfaces help keep you DRY
- Dependency Inversion Principle
  - Depend on abstractions, not concretions
  - * Proper interface separation forces you to comply

DEPENDENCY MANAGEMENT PATTERNS

With SOLID/DRY principles in place, how do we manage component dependencies

- Dependency Declarations
- Depend on Abstractions
- ‘Readonly’ Dependency References { get; ~~set;~~ }
- Dependency Selection
- Dependency Lifetime

DEPENDENCY DECLARATIONS

- As a Package Reference
- As an Assembly Reference
- As an Import Statement
- **As a Constructor Parameter**
- As a Field/Property
- As a method variable

DEPEND ON ABSTRACTIONS

- Dependency Inversion Principle
  - Allow something external to choose the implementation
  - Hollywood Principle: Don’t call us, we’ll call you.
  - Our Component doesn’t care who provides the dependency, just that something does

```
public MyComponent(IMyComponentSettings myComponentSettings)
{
    this.MyComponentSettings = myComponentSettings;
}
```

- ‘Interface Separation’ pattern helps keep abstractions abstract
  - Define all interfaces in interface only assemblies
  - Only allow interface only assemblies to depend on other interface only assemblies and system assemblies
  - Minimize the number of system assembly dependencies

READ ONLY DEPENDENCY REFERENCES

- Get, without set properties in .NET allow for setting that dependency only on construction
- Prefering immutable properties helps with concurrency and lifetime compatibility
- If you need mutable properties, you will need to make that implementation thread safe for singleton lifetimes

```
private IMyComponentSettings MyComponentSettings { get; }
```

- Even though the MyComponentSettings implementation uses { get; set; }
- The IMyComponentSettings is { get; } only.
- MyComponent is thus using the ‘get’ only view.

DEPENDENCY SELECTION

- Something needs to ‘Provide’ the dependency.

- You can always ‘new’ it up yourself
  - In a single code location (the Composition Root*), new your object graph and use what you need.
    - Control object lifetimes with scopes (statics, members, cached, locals)
  - Mark Seemann calls this Pure Dependency Injection
    - He used to call this Poor Man’s Dependency Injection but is now using ‘Pure’
    - Mark argues that this is preferable in many cases: http://blog.ploeh.dk/2012/11/06/WhentouseaDIContainer/

- Use a Dependency Injection Container

DEPENDENCY MANAGEMENT ANTI-PATTERNS

- Service Locator
- Depending on a Dependency Provider
- Conforming Provider

SERVICE LOCATOR

- Described by Martin Fowler: http://martinfowler.com/articles/injection.html
- But is considered an anti-pattern and should be avoided

```
public MyComponent()
{
    this.MyComponentSettings = ServiceLocator.MyComponentSettings;
}
```

- We took the constructor parameter out to use ServiceLocator
- This hides the dependency between MyComponent and IMyComponentSettings
- It is even more hidden if used like:

```
public void DoWork()
{
    var myComponentSettings = ServiceLocator.MyComponentSettings;
}
```

DEPENDING ON THE DEPENDENCY PROVIDER

- This similarly hides the true dependency link

```
public MyComponent(DiContainer diContainer)
{
    this.MyComponentSettings = diContainer.Resolve<IMyComponentSettings>();
}
```

- But it's tempting (especially if you abstract the resolver*):

```
public TypeConversionProvider(IDependencyResolver dependencyResolver)
{
    this.DependencyResolver = dependencyResolver;
}

private IDependencyResolver DependencyResolver { get; }

public IConvert<FromT, ToT> GetTypeConversion<FromT, ToT>()
{
    return this.DependencyResolver.Resolve<IConvert<FromT, ToT>>();
}
```

- The proper* way to do this is with an abstract IFactory<IConvert<FromT, ToT>>

CONFORMING PROVIDER

- “It’s tempting, especially if you abstract the resolver”!
- Abstracting the resolver is pretty simple
  - Until you want to support parameter overrides …
- Abstracting the registrar is not simple since every container implementation a number of different registration, override, interception features and lifetime managers
- Trying to do this is hard and unnecessary
  - But many still try
  - Including Microsoft (ASP, .NET Core), me ☹

DEPENDENCY INJECTION

- You don’t need a Dependency Injection Container
- But they are magic!
  - This is a double edge sword
  - Wield it carefully

PURE DEPENDENCY INJECTION

- **Develop:** Components using the SOLID/DRY principles without assuming a container exists

  <CompositionRoot>

  - **New**: For some interface use this implementation. Repeat.
    - Manually provide dependencies as you ‘new’ more complex implementations.
    - Manually control object lifetime
  - **Output**: The ‘main’ interface.

  </CompositionRoot>

- **Use**: The main interface as the entry point to your program

DEPENDENCY INJECTION CONTAINER

- **Develop:** Components using the SOLID/DRY principles without assuming a container exists

  <Container>

  - **Register**: For some interface use this implementation. Repeat.
    - Container can figure out how to ‘new’ your implementation choice by providing constructor parameters (dependencies) from other registrations
    - You can tweak the container’s construction method
    - Specify object lifetime at registration
    - Should not Resolve() at this stage
  - **Finalize**: The container to a resolution scope.
    - After finalization, cannot Register() without creating a nested resolution scope
  - **Resolve**: from the scope, an instance of the desired interface
    - Aim for a single Resolve()
  - **Forget**: you’re using a DI Container

  </Container>

- **Use**: the resolved interface as the entry point to your program

DEPENDENCY INJECTION CONTAINERS

- Feature Rich
  - Code Config
  - File Config
  - By Convention Config
- Feature imparity
  - Default Lifetime
  - IDisposable handling
  - .NET Requirements

- .NET
  - Unity
  - Autofac
  - Castle Windsor
  - Tiny IOC
  - More…
- C++
  - boost::di
    https://github.com/boost-experimental/di
- D
  - poodinis:
    https://github.com/mbierlee/poodinis
- Java & others

COMPONENTS

- An application is an object graph
- The whole process of providing dependencies between components is called dependency injection
  - Without a container: Pure DI
  - With a container: Unity, Autofac, etc…

PURE VS CONTAINER

```
new MyComponent(
    new MyComponentSettingsFromSettingsProvider(
        new SectionScopedTypedSettingsProvider(
            "Path.To.MyComponent",
            new TypedSettingsProvider<string>(
                new BasicSettingsProvider_AppConfig(),
                new SystemChangeTypeConversionProvider<string>()
            )
        )
    )
);
```

```
var diBuilder = new DiBuilder();

diBuilder.RegisterType<ITypeConversionProvider, SystemChangeTypeConversionProvider<string>>();
diBuilder.RegisterType<IBasicSettingsProvider<string>, BasicSettingsProvider_AppConfig>();
diBuilder.RegisterType<ITypedSettingsProvider, TypedSettingsProvider<string>>();

diBuilder.RegisterType<IMyComponentSettings, MyComponentSettingsFromSettingsProvider>(
    c => new MyComponentSettingsFromSettingsProvider(
        new SectionScopedTypedSettingsProvider(
            "Path.To.MyComponent",
            c.Resolve<ITypedSettingsProvider>()
        )
    )
);

diBuilder.RegisterType<IMyComponent, MyComponent>();
using (var diScope = diBuilder.Finalize())
{
    var myComponent = diScope.Resolve<IMyComponent>();

    myComponent.DoWork();
}
```

MODULES

- Modules are a bundle of component registrations
- Make providing a standard set of components easier
- Can define the bundle close to the component code (a library)
- Reuse bundles across applications

DEFINING MODULES

```
class DependencyBundle { protected abstract void Register(DiBuilder diBuilder); }
class ComponentDependencies : DependencyBundle { }

class MyComponentDependencies : ComponentDependencies
{
    public MyComponentDependencies(string sectionMyComponent) { this.SectionMyComponent = sectionMyComponent; }

    private string SectionMyComponent { get; }

    protected override void Register(DiBuilder diBuilder)
    {
        base.Register(diBuilder);

        diBuilder.RegisterType<ITypeConversionProvider, SystemChangeTypeConversionProvider<string>>();
        diBuilder.RegisterType<IBasicSettingsProvider<string>, BasicSettingsProvider_AppConfig>();
        diBuilder.RegisterType<ITypedSettingsProvider, TypedSettingsProvider<string>>();

        diBuilder.RegisterType<IMyComponentSettings, MyComponentSettingsFromSettingsProvider>(
            c => new MyComponentSettingsFromSettingsProvider(
                new SectionScopedTypedSettingsProvider(
                    this.SectionMyComponent,
                    c.Resolve<ITypedSettingsProvider>()
                )
            )
        );

        diBuilder.RegisterType<IMyComponent, MyComponent>();
    }
}
```

USING MODULES

```
var diBuilder = new DiBuilder();

var myComponentDependencies = new MyComponentDependencies();
myComponentDependencies.Register(diBuilder);

using (var diScope = diBuilder.Finalize())
{
    var myComponent = diScope.Resolve<IMyComponent>();

    myComponent.DoWork();
}
```

DEV, TEST, OPS FRIENDLY MODULES

Build dependency bundles (modules) around large application requirements

- `class DeploymentDependencies`
  - `class LocalDeploymentDependencies`
  - `class AzureDeploymentDependencies`
- `class DiagnosticDependencies`
  - `class LocalDiagnosticsDependencies`
  - `class AzureDiagnosticsDependencies`
- `class AuthenticationDependencies`
  - `class AnnonymousAuthenticationDependencies`
  - `class ClientCertificateAuthenticationDependencies`
  - `class ClaimsTokenAuthenticationDependencies`
- `class ExternalServiceDependencies`
  - `class LegacyExternalServiceDependencies`
  - `class CloudExternalServiceDependencies`
  - `class UnitTestExternalServiceDependencies`

IN PRACTICE

- By defining an abstract module ‘family’ and providing concrete implementations of that family, you can create a graph of modules that define your application
- Reuse the composition root & DI container concepts we just learned
  - Module container + component container
  - Register module choices in a module container, creating the module graph
  - Resolve a root module from the module container, which in turn resolves all module dependencies through the graph
  - Use that module to configure a new component container
  - Resolve a root component from the component container
  - Use the root component as the entry point into your application

IN PRACTICE

```
abstract class DependencyBundle {
    protected DependencyBundle(params DependencyBundle[] bundles)
    { this.Bundles = bundles; }

    protected virtual void Register(DiBuilder diBuilder) {
        foreach (var bundle in this.Bundles) {
            bundle.Register(diBuilder);
        }
    }
}

abstract class ProgramDependencies : DependencyBundle {
    protected ProgramDependencies(params DependencyBundle[] bundles)
        : base(bundles) {}
}

class MyProgramDependencies : ProgramDependencies {
    public MyProgramDependencies(ComponentDependencies componentBundle)
        : base(componentBundle) { }
}
```

```
public static int Main(string[] args) {
    var dbBuilder = new UnityDependencyBuilder();

    dbBuilder.RegisterType<ComponentDependencies, MyComponentDependencies>(
        c => new MyComponentDependencies("Path.To.MyComponent")
    );
    dbBuilder.RegisterType<ProgramDependencies, MyProgramDependencies>();

    using (var dbScope = dbBuilder.Finalize()) {
        var programDependencies = dbScope.Resolve<ProgramDependencies>();

        var diBuilder = new UnityDependencyBuilder();
        programDependencies.Register(diBuilder);

        using (var diScope = diBuilder.Finalize())
        {
            var myComponent = diScope.Resolve<IMyComponent>();

            myComponent.DoWork();
        }
    }
}
```

THE END

- Questions
- Comments
- Contact me: mijones@microsoft.com