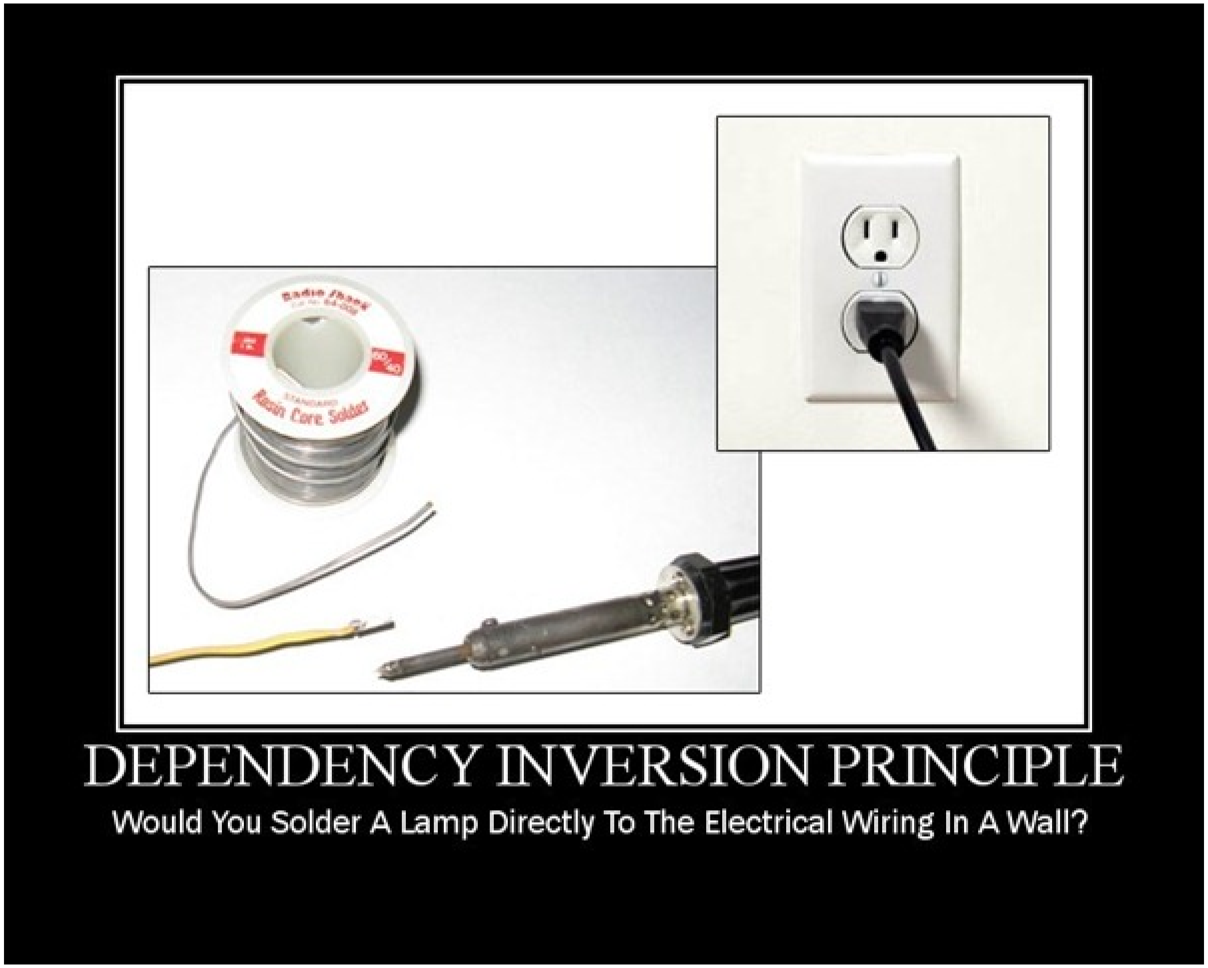
Dependency Injection

Paul Short



We will cover:

- Definition of IoC and DI
- Loose Coupling
- Iterations of “Hello World,” DI-style
- DI Containers
- Identifying Seams (When to use DI)
- DI in Relation to SOLID Principles
- Bastard Injection
- Service Locator
- Recommendations

“Inversion of Control is a common phenomenon that you come across when extending frameworks. Indeed it's often seen as a defining characteristic of a framework.... [This is] also known as the Hollywood Principle—‘Don't call us, we'll call you’.”

Martin Fowler, 2005

http://martinfowler.com/bliki/InversionOfControl.html

Inversion of Control (IoC)

- Normal control flow occurs when user code calls library code; Inversion of control occurs when library code calls user code.
- Many frameworks define the high level structure or flow of the program and rely on user code to perform lower-level tasks.
- A common direct application of the IoC principle is event-driven programming. By that definition, most software developed in .Net uses IoC.
- IoC is a broad term that includes, but isn't limited to, Dependency Injection (DI).
  - Early frameworks that managed dependencies were originally referred to as IoC Containers.
  - Martin Fowler introduced the term Dependency Injection to specifically refer to IoC in the context of dependency management.

Dependency Injection (DI)

- An approach to application configuration.
  - Dependency injection is a specific form of Inversion of Control (IoC) where the concern being inverted is the process of obtaining the needed dependency.
  - Composites objects are “injected” into a containing object
  - Cascading (dependency graph/tree)
- Forms of dependency injection
  - Constructor injection: preferred method; class invariant
  - Property/Setter injection: alternate; when dependency is optional (LOCAL DEFAULT: default implementation of abstraction defined within same assembly as consumer)
  - Method injection (when dependency differs for each operation, inject as parameter)
  - Interface injection: Poor man’s service location (novelty)

Loose Coupling from DI

- Applications that use DI are more naturally “loosely coupled.”
- Tight coupling restricts a system’s ability to change in an industry where the only constant is change.
- For example, in N-tiered applications, tight coupling exists between the different layers, albeit from upper to lower, not vice versa.
- Your business layer might know about and depend on your data access layer, but the reverse is not true. However:
  - Changes to the data access layer may require changes to the business layer (ripple effect).
  - Hard to unit test the different layers in isolation.

Benefits of Loose Coupling via DI

- Late binding
  - Services can be swapped, plugged-in
  - DI doesn't *require* late binding, but *enables* it
- Extensible
  - Extended and reused in unplanned ways
- Parallel Development
  - Developer surge (sometimes)
  - Scales upwards (large, complex applications)
- Maintainability
  - Classes with clearly defined responsibilities
- Testability: classes can be unit tested
  - DI techniques typically required for "clean" unit tests

“Hello, World!”: The tightly coupled way

```
class Program
{
    static void Main()
    {
        Console.WriteLine("Hello DI!");
    }
}
```

- . . . the greeting text is tightly coupled to the Console.

Looser “Hello, World!”: Scenario 1

```
public interface IMessageWriter
{
    void Write(string message);
}

public class Greeter
{
    private readonly IMessageWriter _writer;
    public Greeter(IMessageWriter writer)
    {
        if (writer == null) { throw new ArgumentNullException("writer"); }
        _writer = writer;
    }

    public void Greet()
   {
         _writer.Write("Hello DI!");
   }
}
```

“Hello, World!”: Scenario 1 (continued)

```
// Writer 1: Console
using System;
public class ConsoleMessageWriter : IMessageWriter
{
    public void Write(string message)
    {
        Console.WriteLine(message);
    }
}

class Program
{
    static void Main()
    {
        IMessageWriter writer = new ConsoleMessageWriter();
        var greeter = new Greeter(writer);
        greeter.Greet();
    }
}
```

“Hello, World!”: Scenario 2

```
// Writer 2: Debug stream
using System.Diagnostics;
public class DebugMessageWriter : IMessageWriter
{
    public void Write(string message)
    {
        Debug.WriteLine(message);
    }
}

class Program
{
    static void Main()
    {
        IMessageWriter writer = new DebugMessageWriter();
        var greeter = new Greeter(writer);
        greeter.Greet();
    }
}
```

“Hello, World!”: Scenario 3

```
// Writer 3: Check permissions before delegating work to another writer
// (INTERCEPTION via the DECORATOR design pattern)

using System.Threading;
public class SecureMessageWriter : IMessageWriter
{
    private readonly IMessageWriter _writer;
    public SecureMessageWriter(IMessageWriter writer)
    {
        if (writer == null)  { throw new ArgumentNullException("writer");  }
        _writer = writer;
    }

    public void Write(string message)
    {
        if (Thread.CurrentPrincipal.Identity.IsAuthenticated)
       {
           _writer.Write(message);
       }
    }
}
```

“Hello, World!”: Scenario 3 (continued)

```
using System.Threading;
class Program
{
    static void Main()
    {
        Thread.CurrentPrincipal = new WindowsPrincipal(WindowsIdentity.GetCurrent());

        IMessageWriter writer = new SecureMessageWriter(new ConsoleMessageWriter());
        var greeter = new Greeter(writer);
        greeter.Greet();
    }
}
```

… note that in all three examples, the dependencies were “injected into” the Greeter’s constructor. This illustrates Constructor Injection, the most basic form of Dependency Injection. It is also one of the first things to learn when distinguishing the difference between an automated unit test and an automated integration test.

“Hello, World!”: Scenario 4

To achieve an even higher level of decoupling, let’s introduce “Constrained Construction” late binding so that we can dynamically determine the dependency and concrete type at runtime:

```
// No need for “using System.Diagnostics;”
class Program
{
    static void Main()
    {
       var typeName =  ConfigurationManager.AppSettings["messageWriter"];
       var type = Type.GetType(typeName, true);
       IMessageWriter writer =  (IMessageWriter)Activator.CreateInstance(type);
       var greeter = new Greeter(writer);
       greeter.Greet();
    }
}
```

…in app.config:

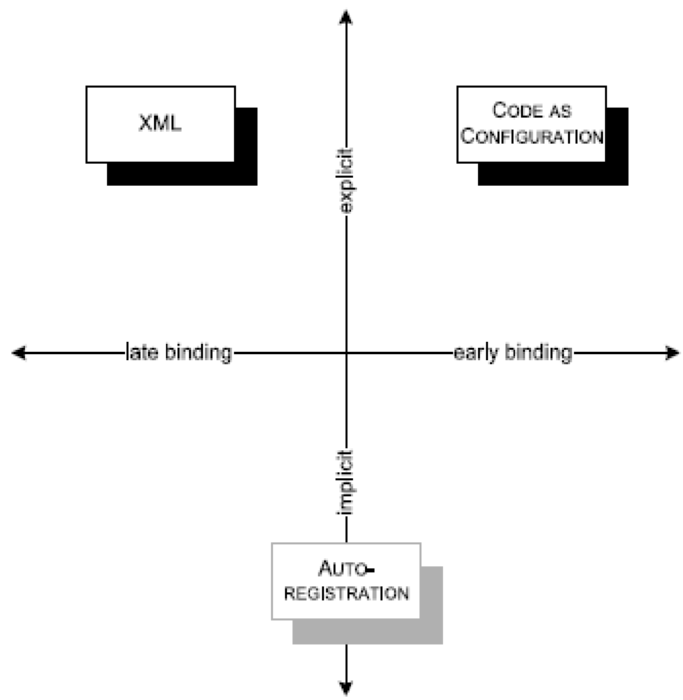
```
<appSettings>
    <add key=“messageWriter” value=“Example.DebugMessageWriter, HelloDI”>
</appSettings>
```

Issues Raised by Scenario 4 (Dynamic)

- What if we want to specify SecureMessageWriter late-bound? It is a DECORATOR that requires another message writer…
  - someone needs to specify *ConsoleMessageWriter*, *DebugMessageWriter*, or some other concrete message writer.
  - this brings up the issue of dependency chains.
  - you can *write more code*, or use a **Dependency Injection (DI) Container**.
- COMPOSITION ROOT: A central place in an application where the entire application is composed from its constituent modules.
- *SecureMessageWriter* illustrates the concept of INTERCEPTION, a technique used to address CROSS-CUTTING CONCERNS.
- Who manages lifetime?

DI Container

- A.k.a IoC Containers or Lightweight Containers
  - Heavyweight: MTS, Com+
  - Lightweight: POCO (like POJO but C=CLR)
- Features:
  - Object *registration*
  - Manages composition root (object graph). Container should only be referenced from composition root.
  - Auto-wiring, auto-registration (varies)
  - Object *resolution* (mapping)
  - Object lifetime (*release)*
  - Interception, AOP (advanced usage)
- Register, Resolve, and Release
- Most DI containers support late binding (usu in an XML config file), but late binding is not required.



“Hello, World!”: Revisited With DI Container

Recall: Interface **IMessageWriter** implemented by classes **ConsoleMessageWriter**, **DebugMessageWriter**, **SecureMessageWriter** (decorator); injected into **Greeter**, called from **Main**.

```
// Unity “Code as Configuration”
class Program
{
    static void Main()
    {
        var container = new UnityContainer();
        container.RegisterType<IMessageWriter, SecureMessageWriter>(
            new InjectionConstructor(
                new ResolvedParameter<ConsoleMessageWriter>())); // also other ways...

        var writer = container.Resolve<IMessageWriter>();
        var greeter = new Greeter(writer);
        greeter.Greet();
    }
}
```

“Hello, World!”: DI Container with XML

True late-binding can be achieved by defining dependencies in XML. Unity has extensive support for configuration in XML, but it is complicated and difficult to debug.

```
// Unity “XML as Configuration”
class Program
{
    static void Main()
    {
         var container = new UnityContainer();
         container.LoadConfiguration();
         var writer = container.Resolve<IMessageWriter>();
         var greeter = new Greeter(writer);
         greeter.Greet();
    }
}
<configSections>
    <section name="unity" type="Microsoft.Practices.Unity.Configuration.UnityConfigurationSection,
                                Microsoft.Practices.Unity.Configuration"/>
</configSections>
<unity>
    <assembly name="Example.HelloWorld" />
    <container>
        <register type="IMessageWriter" mapTo=“SecureMessageWriter" />
... TBD: long-winded syntax for adding ConsoleMessageWriter as constructor parameter ...
    </container>
</unity>
```

Identifying SEAMS (When to use DI)

- The concept of a SEAM can help determine when to introduce loose coupling and when not. A seam is a place where an application is assembled from its constituent parts.
  - When we decide to program against an interface instead of a concrete type, we introduce a seam into the application.
  - Introducing seams into an application is extra work, so you should only do it when necessary. Not all dependencies should be treated equally; you can live with certain stable dependencies.
- **STABLE DEPENDENCY**: class or module already exists, new versions won't contain breaking changes, deterministic algorithms, never expect to have to replace the class or module with another.
  - The .Net Base Class Library (BCL) assemblies.
- **VOLATILE DEPENDENCIES**:
  - Requirement to set up and configure a runtime environment: Databases, File I/O, Message Queues, Web Services, some 3rd-party libraries.
  - Non-deterministic behavior: randomness, date/time.
  - Dependency doesn't yet exist, still in development or changing (parallel development)

The Dependency Inversion Principle (DIP)

- The Dependency Inversion Principle (DIP) is the principle that guides Dependency Injection (DI).
  - *Abstractions should not depend on details. Details must depend on abstractions.*
  - *High level modules should not depend upon low level modules and vice-versa. Both should depend on abstractions.*
  - *Program to an interface instead of a concrete implementation.*

- DIP is the "D" in SOLID.
- Has a relationship with each of the other four SOLID principles.

DI in Relation to SOLID Principles

Principle	Summary	Relation to DI
SINGLE RESPONSIBILITY PRINCIPLE (SRP)	A class should only have a single responsibility. Opposite of this principle is the God Class anti-pattern.	Not always easy to achieve, but code written using the SRP makes it easier to spot potential SEAMS (places where DI should be used).
OPEN/CLOSED PRINCIPLE (OCP)	A class should be open for extensibility, but closed for modification; behavior can be added to existing class without modifying its code.	Many ways to make a class extensible (decorator, *interception,* etc.), but whatever the details, DI makes this possible by enabling us to compose objects.
LISKOV SUBSTITUTION PRINCIPLE (LSP)	A client should treat all implementations of an ABSTRACTION equally. Replace any implementation with a different implementation without breaking the consumer.	The LSP is a foundation of DI. When consumers don't observe it, you can't replace DEPENDENCIES at will, and we lose the benefits of DI.
INTERFACE SEGREGATION PRINCIPLE (ISP)	The ISP states that interfaces should model only a single concept, whereas the SRP states that implementations should have only one responsibility.	Violations of the ISP (LEAKY ABSTRACTION) makes it harder to replace DEPENDENCIES because some of the interface members may not make sense in a context different from what originally drove the design.
DEPENDENCY INVERSION PRINCIPLE (DIP)	Program to an interface instead of a concrete implementation.	DIP is the principle that guides DI.

Bastard Injection

- "Poor man's" dependency injection: Greedy Constructor – An overloaded constructor that takes all of your class's dependencies.
- ***Bastard Injection*** *(arguably an anti-pattern):* Default Constructor calling Greedy Constructor – *Creates instances of known dependency implementations and delegates to the greedy constructor.* **Concerns:**
  - Although often used because it enables testability, it drags along dependencies we do not want (re-introduces tight coupling outside of unit test path).
    - Uses a foreign default that makes parallel development more difficult.
    - Circumvents auto-wiring feature of certain DI containers.
  - ASP.Net MVC AccountController class is an example of bastard injection.
  - I've been guilty of heavily using this "anti-pattern".
  - I've also used **Service Locator**, which might also be considered to be a DI Anti-pattern.

Bastard Injection: Opinions

- Careful construction of a COMPOSITION ROOT, limited use of PROPERTY INJECTION, and use of a DI container are alternatives to instantiating items in a default constructor.

- Many projects may already be using Bastard Injection, especially if they haven't been using a DI container from the beginning. You'll need to see if it's worth refactoring. Depends on what you are trying to achieve.

- Mark Seeman treats Bastard injection as "anti-pattern" in his book, *Dependency Injection in .Net* (Manning, 2012). Chapter 5, "DI anti-patterns" does a much better job of explaining this than I could. Here's an interview where the author explains his opinions: http://www.infoq.com/articles/DI-Mark-Seemann

ServiceLocator

- A Static Factory that clients call to locate services. Fowler, 2004: http://martinfowler.com/articles/injection.html
- Functionally equivalent to statically wrapping a DI container then having all clients call resolve on it (*container.resolve<T>*)
- Easy to understand, common, …but
- Mark Seeman takes an contrarian view by considering it an "anti-pattern" in his book, *Dependency Injection in .Net* (Manning, 2012):
  - **All classes that consume it are now dependent on it.**
  - Isn't apparent DI is being used.
  - His observation: you should ask a container or locator to resolve a dependency graph from the COMPOSITION ROOT; you should not ask for granular services from anywhere else (**sprinkled throughout the code**).

Recommendations

- Begin using DI techniques for unit tests, but avoid using Bastard Injection early on if possible.
- Although ServiceLocation is tempting, let's try to maintain a composition root instead.
  - Full disclosure: I haven't accomplished this myself yet.
  - Might be painful to do this on large, established projects.
- Although Unity is more common, Ninject might be a better fit for smaller projects.
  - Slick, easier to use
  - Code as configuration (but no XML support)
- Need to think about object lifetime a bit more.
- Need to unlearn a few more things. That includes ME ☺.

References

- *Dependency Injection in .Net*, Mark Seemann. Manning Publications Co. 2012.
  - http://www.manning.com/seemann/
  - http://www.amazon.com/Dependency-Injection-NET-Mark-Seemann/dp/1935182501
  - http://www.infoq.com/articles/DI-Mark-Seemann
- http://en.wikipedia.org/wiki/Dependency_injection
- http://martinfowler.com/bliki/InversionOfControl.html
- http://martinfowler.com/articles/injection.html
- http://www.blackwasp.co.uk/DependencyInjection.aspx
- http://www.agilefirestarter.com/Content/Firestarter4/Agile%20Firestarter%20Winter%202011%20Dependency%20Injection.pdf
- http://elegantcode.com/2009/01/07/ioc-libraries-compared/

END