Dependency Injection

Agenda

- Motivation
- Dependency Injection Basics
- Spring.NET Fundamentals
- References

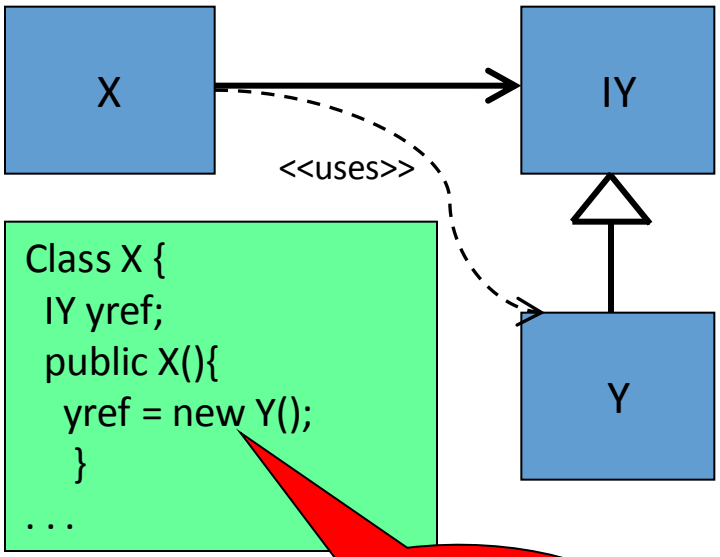
Dependency Injection Explained

What is a Dependency?

A Dependency between a source class X and a target class Y occurs when each instance X needs an instance of Y!

But what's the problem?

The problem is that X needs to know how to create the Y instance!




```
Class X {
  IY yref;
  public X(){
    yref = new Y();
    }
. . .
```

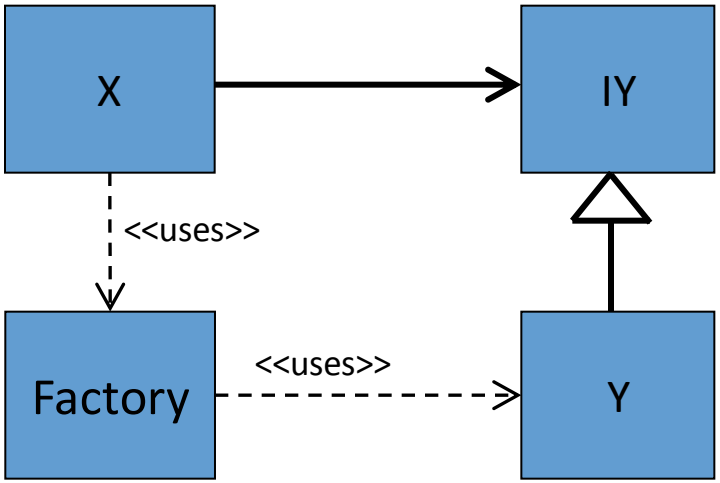
No injection here!

Factory Example

- The Factory Method design pattern is one method to avoid the direct coupling from class X to class Y.

```
Class X {
  IY yref;
  public X(){
    yref = Factory.CrateY();
     }
. . .
```

```
Class X {
  IY yref;
  public X(){
    yref = Factory.Crate("Y");
     }
. . .
```



Factories

- The use of a factory class or method is one alternative to DI.
- When a component creates a private instance of another class, it internalizes the initialization logic within the component.
- This initialization logic is rarely reusable outside of the creating component, and therefore must be duplicated for any other class that requires an instance of the created class.
- Factories are perfect for the simple scenario like the one on the previous slide
  - But for more complex scenarios they lack configuration capabilities.

Dependency Injection

- An object should not instantiate the objects it depends on.
  - Instead, these objects should be passed in “from outside”.

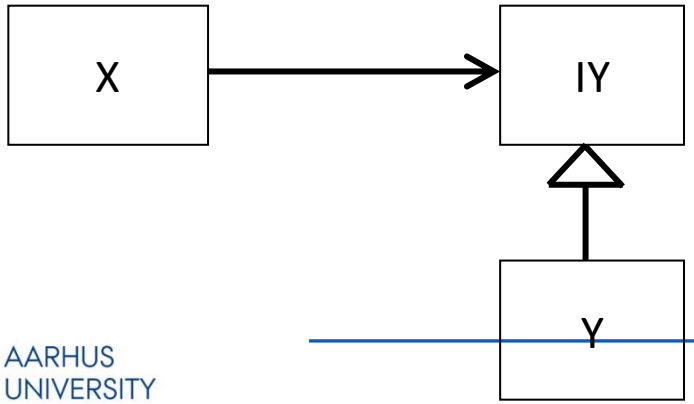
- The two major different types of injection are:

Constructor injection

```
Class X {
  IY yref;
  public X(IY ay)
  {
    yref = ya;
  }
. . .
```

Setter injection

```
Class X {
  IY yref;
  public X(){
    …
  }
  public void SetY(IY ay)
  {
    yref = ay;
  }
. . .
```



Dependency Injection By-hand

Constructor injection

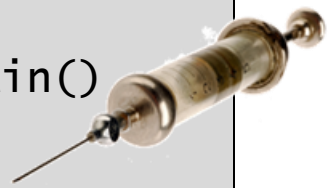
```
Class App {
  X x;

  public static Main()
  {
    x = new x(new Y());
  }
. . .
```

```
Class X {
  IY yref;
  public X(IY ay)
  {
    yref = ya;
  }
. . .
```

An application initializer is injecting an IY implementation into a class X instance.

Setter injection

```
Class App {
  X x;

  public static Main()
  {
    x = new x();
    x.SetY(new Y());
  }
. . .
```



```
Class X {
  IY yref;
  public X(){
    …
  }
  public void SetY(IY ay)
  {
    yref = ay;
  }
. . .
```

DI Using IoC Containers

- Instead of hard-coding the dependencies by hand, a component just lists the necessary services and a DI framework (called an IoC container) supplies these
- An IoC container is a component that is responsible for object management and configures the object graph.
- Containers allow for objects to be configured by the container, as opposed to being configured by the client application.

Inversion of Control

- Inversion of Control is a principle used by frameworks as a way to allow developers to extend the framework or create applications using it.
- The basic idea is that the framework is aware of the programmer's objects and makes invocations on them.
- This is the opposite of using an API, where the developer's code makes the invocations to the API code.
- Hence, frameworks invert the control:
  - it is not the developer code that is in charge,
  - instead the framework makes the calls based on some stimulus.

IoC is also known as the Hollywood Principle, which states:
"don't call us, we'll call you"

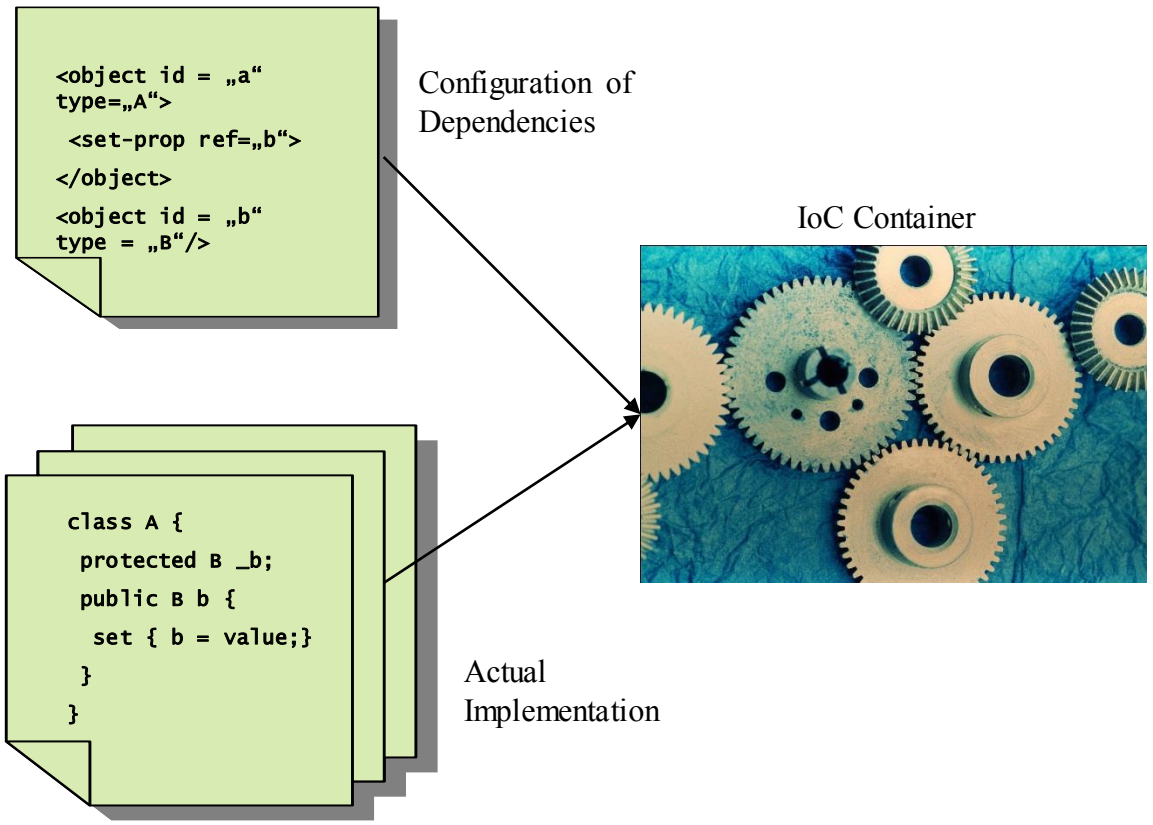
Inversion of Control Container

- An Inversion of Control Container uses the IoC principle to manage classes:
  - creation,
  - destruction,
  - lifetime,
  - configuration, and
  - dependencies.

- This way classes do not need to obtain and configure the classes they depend on.
- This dramatically reduces coupling in a system
  - And simplifies reuse and testability.

DI Using IoC Containers

- But how does the Container know which dependencies to inject?



DI Example Using Castle Windsor IoC Container

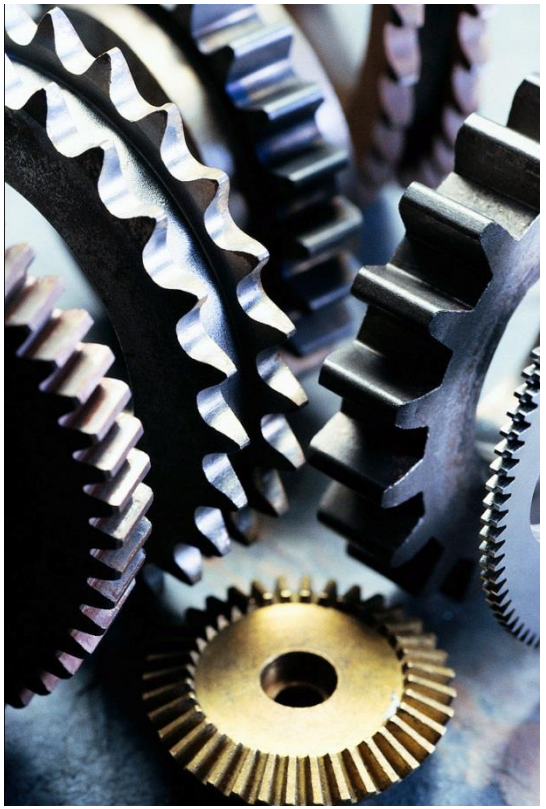
```
IWindsorContainer container = new WindsorContainer();

container.AddComponent("HttpFileDownloader",
                       typeof(IFileDownloader),
                       typeof(HttpFileDownloader));
container.AddComponent("StringParsingTitleScraper",
                       typeof (ITitleScraper),
                       typeof(StringParsingTitleScraper));
container.AddComponent("HtmlTitleRetriever",
                       typeof (HtmlTitleRetriever));

HtmlTitleRetriever retriever =container.Resolve<HtmlTitleRetriever>();

string title = retriever.GetTitle(new Uri("some uri..."));

container.Release(retriever);
```

IoC Containers vs. Factories

- There are several reasons to use containers in your application development:
- Containers provide the ability to wrap vanilla objects with a wealth of other services.
- This allows the objects to remain ignorant about certain infrastructure and plumbing details.
- The component code does not need to be aware of the container, so there is no real dependency on the container itself.
- These services can be configured declaratively, meaning they can be configured via some external means, including GUIs, XML files, property files, or .NET-based attributes.

IoC Containers for .NET

- Ninject.org:lightweight and .Net specific IoC container.
- PicoContainer.NET: lightweight and highly embeddable IoC container (port of Java version).
- StructureMap: lightweight Inversion of Control (IoC) Container written in C#.
- Castle: Tools for application development including IoC container.
- Spring.NET: full featured IoC container (port of Java version).
- Unity: from Microsoft
- LinFu: Open Source



Comparing Different DI-IOC frameworks

Framework	Version	.NET	License	Minimum required dlls	Size (KB)
Castle	2.0	2.0	Apache 2	2	240
Unity	1.2	2.0	MS-PL	2	136
Ninject	1.0	2.0	Apache 2	1 (2 for smart autowiring)	147 (154)
Autofac	1.4.4	3.5	MIT	1	126
StructureMap	2.5.3.0	3.5	Apache 2	1	217
Spring.Net	1.3RC1	1.1	Apache 2	2	840
LinFu	2.2.0.0	3.5	LGPL	1	690
OpenNETCF.IoC	1.0.9280	CF	Public domain + attribution	1	37

Framework	Fluent registration	Automatic registration	Unregistered resolution	Attribute usage	XML usage
Castle	yes	yes	not supported	supported, not required	supported, not required
Unity	yes	no	yes	supported, not required	supported, not required
Ninject	yes	yes	yes	supported, not required	not supported, implementable on top
Autofac	yes	included as example on offical site	requires special opt-in	not supported, not needed	supported, not required
StructureMap	yes	yes	yes	supported, not required	supported, not required
Spring.Net	method chaining only	not supported or not easy	either not supported or not easy	supported, not required	supported, not required, encouraged approach
LinFu	yes	yes	yes	supported, not required	not supported, implementable on top
OpenNETCF.IoC	no	yes	not supported	supported, not required	supported, not required

Reference:
http://code.google.com/p/net-ioc-frameworks/wiki/Charts

Summary

- The two cornerstones of DI are *programming to interfaces* and expecting outside forces to supply instances of required dependencies.

In other words:

- You need to define **Abstractions** and code against these,
- And to enable external callers to supply concrete instances that implement these **Abstractions.**

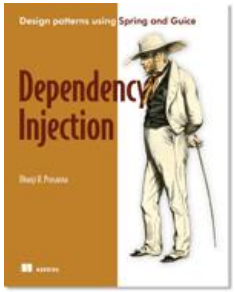
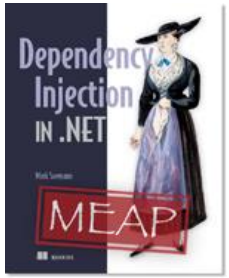
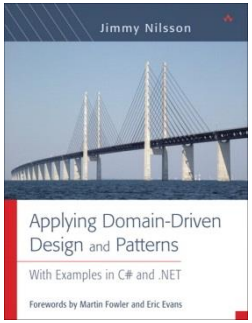
- The choice between use of object Factories, Service Locator and Dependency Injection is less important than the principle of separating service configuration from the use of services within an application.

Resources

- Articles
  - **http://www.martinfowler.com/articles/injection.html**
  - http://msdn.microsoft.com/msdnmag/issues/05/09/DesignPatterns/
  - http://en.wikipedia.org/wiki/Dependency_injection

- Open source DI frameworks
  - **Windsor**: http://www.castleproject.org
  - **Unity**: http://msdn.microsoft.com/en-us/library/dd203101.aspx
  - **Ninject**: http://ninject.org/
  - Structuremap: http://sourceforge.net/projects/structuremap
  - Guice: http://code.google.com/p/google-guice/
  - PicoContainer: http://www.picocontainer.org/
    http://docs.codehaus.org/display/PICO/Ports
  - NanoContainer: http://docs.codehaus.org/display/NANO/Home
  - Spring: http://www.springframework.net/
  - (Wikipedia has the full list)

Resources - Books

	Dependency Injection *Design patterns using Spring and Guice* by Dhanji R. Prasanna http://www.manning.com/prasanna
	Dependency Injection in .NET By Mark Seemann http://www.manning.com/seemann/
	Applying Domain-Driven Design and Patterns: With Examples in C# and .NET http://www.awprofessional.com/title/0321268202 by Jimmy Nilsson *Contains sections on Inversion of Control, Depenency Injection*