



Dependency Injection

Mark Pollack

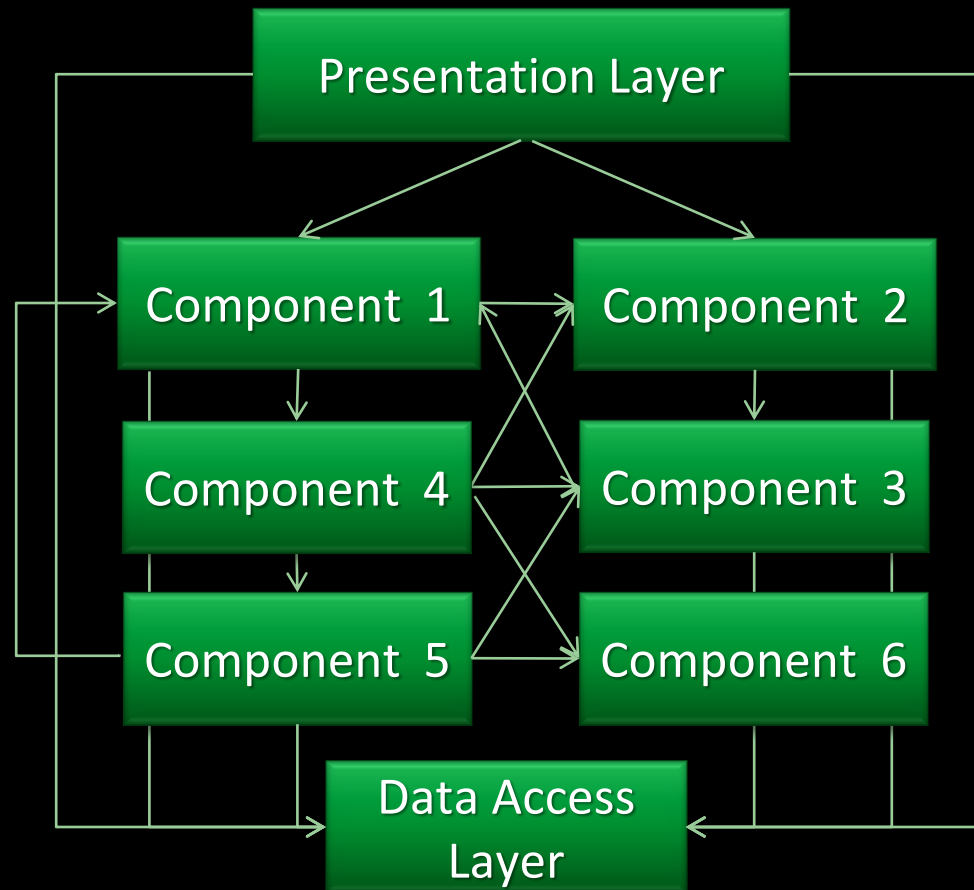
SpringSource/VMware

mark.pollack@springsource.com

What is Dependency Injection?

- An approach to application configuration
- Why should you care?
- Applications that use DI are more naturally 'loosely coupled'
- Loosely coupled applications
 - Are easier to test and maintain
 - Promote agile development and flexible design

Tight coupling in action





Example: Tight Coupling

```
public class OrderService : IOrderService {  
  
    private SqlOrderRepository orderRepository;  
    private SntpNotificationService notificationService;  
    private int alertQuantity = 10000;  
  
    public OrderService() {  
        this.orderRepository = new SqlOrderRepository();  
        this.notificationService = new SntpNotificationService();  
    }  
  
    public int AlertQuantity { set { alertQuantity = value; } }  
  
    public string CancelOrder(long orderId) {  
        // use orderRepository and notificationService  
        return guid;  
    }  
}
```

Example: Loose coupling

```
public class OrderService : IOrderService {  
  
    private IOrderRepository orderRepository;  
    private INotificationService notificationService;  
    private int alertQuantity = 10000;  
  
    public OrderService(IOrderRepository orderRepository,  
                       INotificationService notificationService) {  
        this.orderRepository = orderRepository;  
        this.notificationService = notificationService;  
    }  
  
    public int AlertQuantity { set { alertQuantity = value; } }  
  
    public string CancelOrder(long orderId) {  
        // use orderRepository and notificationService  
        return guid;  
    }  
}
```

“Hand Coded DI”

```
public void Configure()  
{  
    IOrderRepository repo = new SqlOrderRepository();  
  
    INotificationService smtpService =  
        new SmtNotificationService();  
  
    IOrderService svc =  
        new OrderService(repo, smtpService);  
}
```

Now you can do unit tests...

```
public string CancelOrder(long orderId)
{
    Order order = orderRepository.FindOrder(orderId);

    string confirmationId = orderRepository.CancelOrder(order);
    if (order.Quantity > alertQuantity)
    {
        notificationService.SendCancelNotification(order);
    }
    return confirmationId;
}
```

Test code

```
[Test]
public void OrderService()
{
    IOrderRepository stubRepo = new StubOrderRepository();

    INotificationService stubService =
        new StubNotificationService();

    IOrderService svc = new OrderService(stubRepo, stubService);

    string confirmationid = svc.CancelOrder(123);
    Assert.IsNotNull(confirmationid);

    // Assertions on INotificationService behavior
    // Was it called?
}
```

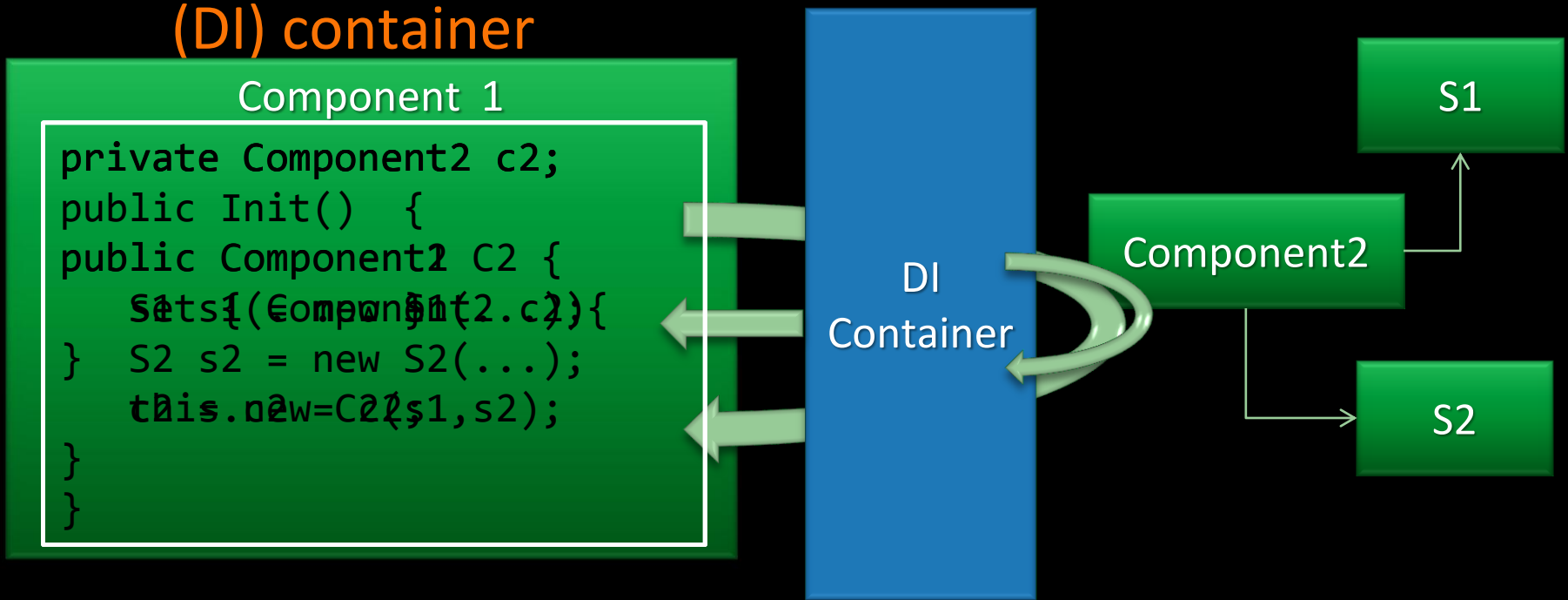
Loose Coupling

- Interface-based contracts promote loose coupling
- Assemble more of your application ‘lego style’
- But how?
 - Classes no longer manage collaborating objects
 - Abstract Factory design pattern
 - Close but no cigar...
 - Use a Dependency Injection container



Scatter

- **Class objects are created by the framework objects of the container. Dependency Injection (DI) container**



```

Component 1
private Component2 c2;
public Init() {
public Component2 C2 {
    Sets{(Component2.c2)}{
} S2 s2 = new S2(...);
    this.new=C2,s1,s2);
}
}

```

Component injection

Inversion of Control

- Release control of some logic to a framework
- Event-Driven Architecture
 - Framework polls or listens to an event source
 - Framework notifies or invokes a service
- Dependency Injection Container
 - Container is creating classes, setting properties
 - Container may 'wrap' objects with other services

Containers are not new

- MTS
- COM+ / Enterprise Services
 - `Server.CreateObject("Database.Connection")`
- But were 'heavyweight'
 - Inherit from 'magic' base class or interface
- Dependency Injection Containers
 - Are 'Lightweight' but provide same many benefits

Dependency Injection Containers

- Promote a component model that actually works

PLAIN OLD CLR OBJECTS (POCO)

- Non-invasive
 - POCOs not tied to the DI container
 - Beware of DI container attributes!

What is the downside to DI?

- One more thing to learn...
 - It is worth the effort
 - A few books, but mostly online resources.
- Another level of abstraction
- Need to select a DI container

Where can I get one of these...

- Autofac
- Castle Windsor
- Ninject
- Spring for .NET
- StructureMap
- Unity

The 1st order summary...

- All provide similar DI functionality
- Try not to think of DI ‘having an API’
 - Differences are how you configure the container to create objects and assemble your application
- Different extensibility models
- Some offer additional capabilities
 - AOP, Transaction Management,
 - Support for several runtime environments
 - ASP.NET WebForms/MVC, WCF, ...

The quick guide to using a DI container

- Configure a DI container's object creation and configuration rules via
 - XML
 - Configuration API
 - Attributes

Classes used in the example

```
public class ConsoleReport {  
  
    // private fields omitted  
  
    public ConsoleReport(IOutputFormatter outputFormatter,  
                        IPrimeGenerator primeGenerator) {  
        _outputFormatter = outputFormatter;  
        _primeGenerator = primeGenerator;  
    }  
  
    public int MaxNumber {  
        get { return _maxNumber; }  
        set { _maxNumber = value; }  
    }  
}
```

Classes used in the example

```
static void Main(string[] args)
{
    ConsoleReport report = new ConsoleReport(
        new OutputFormatter(),
        new PrimeGenerator(new PrimeEvaluationEngine()))
    );
    report.MaxNumber = 1000;
    report.Write();

    Console.WriteLine("--- hit enter to exit ---");
    Console.ReadLine();
}
```

Creating the Unity Container

```
class Program
{
    static void Main(string[] args)
    {
        using (var container = new UnityContainer())
        {
            //configure container

            //ask container for objects (configured) and use them
        }
    }
}
```

Configuring the Unity Container

```
class Program
{
    static void Main(string[] args)
    {
        using (var container = new UnityContainer())
        {
            //configure container
            container
                .RegisterType<IOutputFormatter, OutputFormatter>()
                .RegisterType<IPrimeGenerator, PrimeGenerator>();

            //ask container for objects (configured) and use them
        }
    }
}
```

Getting Configured Objects

```
class Program {
    static void Main(string[] args) {
        using (var container = new UnityContainer()) {

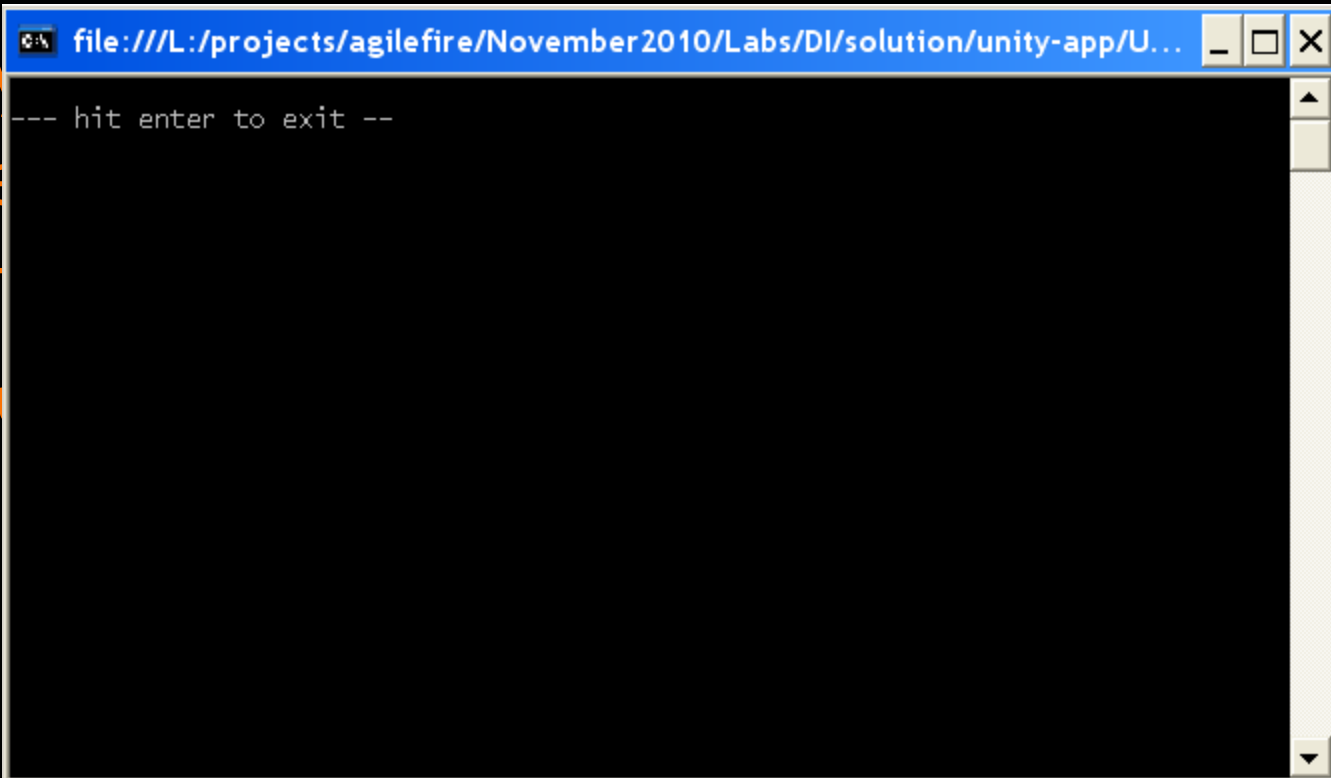
            //configure container
            container
                .RegisterType<IOutputFormatter, OutputFormatter>()
                .RegisterType<IPrimeGenerator, PrimeGenerator>();

            //ask container for objects (configured) and use them
            var report = container.Resolve<ConsoleReport>();
            report.Write();
        }
    }
}
```

However...

- Runs but no output ☹
 - Default max number is set to 0

- Only
type
– 'T
– B



```
file:///L:/projects/agilefire/November2010/Labs/DI/solution/unity-app/U...  
--- hit enter to exit ---
```

ect

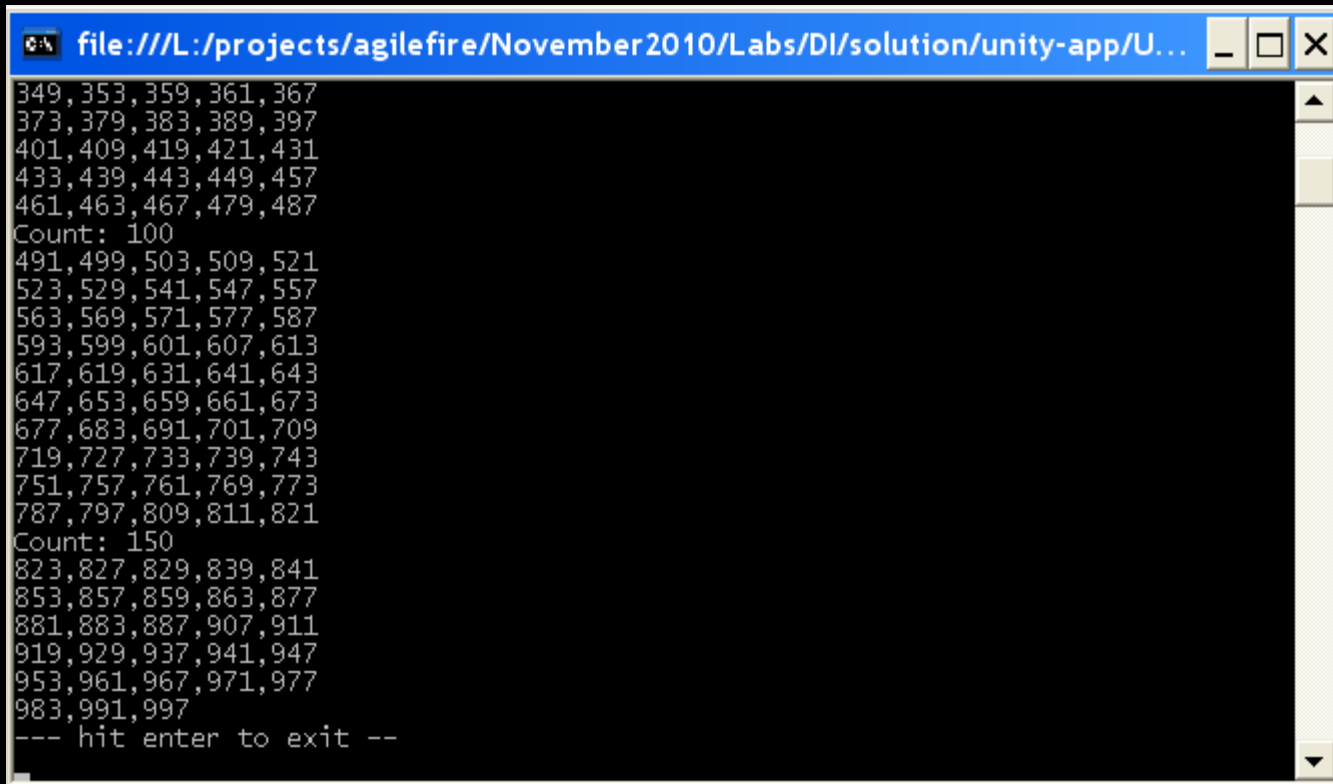
Configuring Simple Properties

```
class Program {
    static void Main(string[] args) {
        using (var container = new UnityContainer()) {

            //configure container
            container
                .RegisterType<IOutputFormatter, OutputFormatter>()
                .RegisterType<IPrimeGenerator, PrimeGenerator>();

            container.Configure<InjectedMembers>()
                .ConfigureInjectionFor<ConsoleReport>(
                    new InjectionProperty("MaxNumber", 1000)
                );
            //ask container for objects (configured) and use them
            var report = container.Resolve<ConsoleReport>();
            report.Write();
        }
    }
}
```

Now we get our prime numbers...



```
file:///L:/projects/agilefire/November2010/Labs/DI/solution/unity-app/U...
349, 353, 359, 361, 367
373, 379, 383, 389, 397
401, 409, 419, 421, 431
433, 439, 443, 449, 457
461, 463, 467, 479, 487
Count: 100
491, 499, 503, 509, 521
523, 529, 541, 547, 557
563, 569, 571, 577, 587
593, 599, 601, 607, 613
617, 619, 631, 641, 643
647, 653, 659, 661, 673
677, 683, 691, 701, 709
719, 727, 733, 739, 743
751, 757, 761, 769, 773
787, 797, 809, 811, 821
Count: 150
823, 827, 829, 839, 841
853, 857, 859, 863, 877
881, 883, 887, 907, 911
919, 929, 937, 941, 947
953, 961, 967, 971, 977
983, 991, 997
--- hit enter to exit ---
```

Object Lifecycle

- What happens if you call
 - `container.Resolve<ConsoleReport>()` ;
Twice...?
- Container has options to control the lifecycle
 - Transient/“Prototype” – new one each time
 - The default option
 - Singleton – one for the life of the container
 - `ContainerControlledLifetimeManager`
 - Externally Controlled – one for the life of the container
 - But container holds a weak reference to the object
 - `ExternallyControlledLifetimeManager`
 - `LifetimeManager` base class to customize

Now as singletons...

```
class Program {
    static void Main(string[] args) {
        using (var container = new UnityContainer())
        {
            //configure container
            container
                .RegisterType<IOutputFormatter, OutputFormatter>(new
                    ContainerControlledLifetimeManager())
                .RegisterType<IPrimeGenerator, PrimeGenerator>(new
                    ContainerControlledLifetimeManager())
                .RegisterType<ConsoleReport>(new ContainerControlledLifetimeManager());

            container.Configure<InjectedMembers>()
                .ConfigureInjectionFor<ConsoleReport>(
                    new InjectionProperty("MaxNumber", 1000)
                );
            //ask container for objects (configured) and use them
            var report = container.Resolve<ConsoleReport>();
            report.Write();
        }
    }
}
```

Setter Dependency Injection

```
class Program {
    static void Main(string[] args) {
        using (var container = new UnityContainer()) {
            //configure container
            container
                .RegisterType<IOutputFormatter, OutputFormatter>()
                .RegisterType<IEmailService, EmailService>()
                .RegisterType<IPrimeGenerator, PrimeGenerator>();

            container.Configure<InjectedMembers>()
                .ConfigureInjectionFor<ConsoleReport>(
                    new InjectionProperty("EmailService",
                        new ResolvedParameter<IEmailService>())
                );

            //ask container for objects (configured) and use them
            var report = container.Resolve<ConsoleReport>();
            report.Write();
        }
    }
}
```

Setter Dependency Injection (II)

```
public class ConsoleReport
{
    private IEmailService _emailService;
    private IOutputFormatter _outputFormatter;
    private IPrimeGenerator _primeGenerator;

    public ConsoleReport(IOutputFormatter outputFormatter,
                        IPrimeGenerator primeGenerator)
    {
        _outputFormatter = outputFormatter;
        _primeGenerator = primeGenerator;
    }

    [Dependency]
    public IEmailService EmailService
    {
        set { _emailService = value; }
    }
}
```

No longer a POCO

Unity XML Configuration Outline

```
<configuration>
  <configSections>
    <section name="unity" type="..." />
  </configSections>
  <unity>
    <typeAliases>
      <!-- register alias to reduce verbosity -->
    </typeAliases>

    <containers>
      <container>
        <types>
          <type type="Primes.ConsoleReport, Primes">
            <lifetime type="singleton" />
            <typeConfig extensionType="...">
              <!-- configure constructor and properties -->
            </typeConfig>
          </type>
          <!-- repeat for more more type configurations -->
        </types>
      </container>
    </containers>
  </unity>
```

Unity XML Configuration - I

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="unity"
type="Microsoft.Practices.Unity.Configuration.UnityConfigurationSection,
Microsoft.Practices.Unity.Configuration"/>
  </configSections>
  <unity>
    <typeAliases>
      <!-- register aliases to reduce verbosity -->
      <typeAlias alias="singleton"

type="Microsoft.Practices.Unity.ContainerControlledLifetimeManager,
Microsoft.Practices.Unity" />
      <typeAlias alias="IOutputFormatter"
        type="Primes.IOutputFormatter, Primes" />
      <typeAlias alias="IPrimeGenerator"
        type="Primes.IPrimeGenerator, Primes" />
    </typeAliases>
  </unity>
</configuration>
```

Unity XML Configuration - II

```
<containers>
  <container>
    <types>
      <type type="Primes.ConsoleReport, Primes">
        <typeConfig
extensionType="Microsoft.Practices.Unity.Configuration.TypeInjectionElement,
Microsoft.Practices.Unity.Configuration">
          <constructor>
            <param name="outputFormatter" parameterType="IOutputFormatter"/>
            <param name="primeGenerator" parameterType="IPrimeGenerator"/>
          </constructor>
          <property name="MaxNumber" propertyType="System.Int32, mscorlib">
            <value value="1000" type="System.Int32"/>
          </property>
        </typeConfig>
      </type>

      <type type="IOutputFormatter" mapTo="Primes.OutputFormatter, Primes">
        <lifetime type="singleton" />
      </type>
    </types>
  </container>
</containers>
```

Unity XML Configuration - III

```
<type type="IPrimeGenerator" mapTo="Primes.PrimeGenerator, Primes">
  <lifetime type="singleton"/>
  <typeConfig
extensionType="Microsoft.Practices.Unity.Configuration.TypeInjectionElement,
Microsoft.Practices.Unity.Configuration">
    <constructor>
      <param name="primeEvaluationEngine"
parameterType="Primes.PrimeEvaluationEngine, Primes"/>
    </constructor>
  </typeConfig>
</type>

  <type type="Primes.PrimeEvaluationEngine, Primes"/>

</types>
</container>
</containers>
</unity>

</configuration>
```

Spring.NET Code Configuration

(see <http://www.springframework.net/codeconfig>)

```
[Configuration]
public class PrimesConfiguration {

    [Definition]
    public virtual ConsoleReport ConsoleReport() {
        return new ConsoleReport(OutputFormatter(), PrimeGenerator())
            {MaxNumber = 1000};
    }

    [Definition]
    public virtual IOutputFormatter OutputFormatter() {
        return new OutputFormatter();
    }

    [Definition]
    public virtual IPrimeGenerator PrimeGenerator() {
        return new PrimeGenerator(new PrimeEvaluationEngine());
    }
}
```

Creating and Configuring Spring.NET Container using Code Config

```
class Program
{
    static void Main(string[] args)
    {
        using (var container = new ScanningApplicationContext())
        {
            //scan over all .dll/.exe to find object definitions
            container.ScanAllAssemblies();
            //initialize object definitions
            container.Refresh();
            //ask container for objects (configured) and use them
            var report = container.GetObject<ConsoleReport>();
            report.Write();
        }
    }
}
```

Creating and Configuring Spring.NET Container using XML object definitions

```
class Program
{
    static void Main(string[] args)
    {
        using (var container = new XmlApplicationContext("context.xml"))
        {
            //ask container for objects (configured) and use them
            var report = container["ConsoleReport"] as ConsoleReport;
            report.Write();
        }
    }
}
```

Spring.NET XML Configuration

```
<objects xmlns="http://www.springframework.net"
  default-autowire="autodetect">

  <object name="ConsoleReport" type="Primes.ConsoleReport, Primes">
    <property name="MaxNumber" value="1000"/>
  </object>

  <object type="Primes.PrimeGenerator, Primes"/>

  <object type="Primes.OutputFormatter, Primes"/>

  <object type="Primes.PrimeEvaluationEngine, Primes"/>

</objects>
```

Spring.NET XML Configuration Explicit Wiring

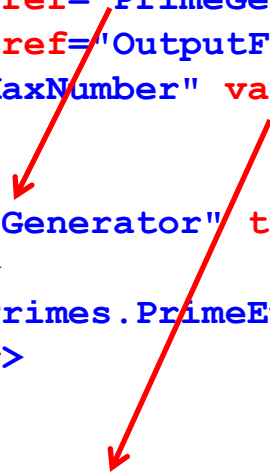
```
<objects xmlns="http://www.springframework.net">

  <object name="ConsoleReport" type="Primes.ConsoleReport, Primes">
    <constructor-arg ref="PrimeGenerator"/>
    <constructor-arg ref="OutputFormatter"/>
    <property name="MaxNumber" value="1000"/>
  </object>

  <object name="PrimeGenerator" type="Primes.PrimeGenerator, Primes">
    <constructor-arg>
      <object type="Primes.PrimeEvaluationEngine, Primes"/>
    </constructor-arg>
  </object>

  <object name="OutputFormatter" type="Primes.OutputFormatter, Primes"/>

</objects>
```



Dependency Injection Summary

- Benefits
 - Code easier to test and maintain
 - Promotes loose coupling between
 - Classes
 - “Modules”/Subsystems
 - Will see signs of
 - Reuse
 - Better code – ‘harder to do bad things’
- Drawbacks
 - Another level of abstraction
 - Need to learn a DI container technology

Insert Tab A into Slot B...

DEPENDENCY INJECTION LAB

Lab Exercise

- Configure the Prime Number application using
 - Unity
 - Fluent API
 - XML
 - Spring.NET
 - XML
 - C#
- Following instructions in docs...