Designing for Inversion of Control Architecture Camp 2007

Alex Henderson
DevDefined Limited

About Me

- Senior Developer / Director at DevDefined Limited.
- Used IoC in all major projects since 2005.
- Specialise in product development & consultancy.
- .Net Developer since First Beta of the 1.0 Framework.

- **Email:** alex@devdefined.com
- **Blog:** http://blog.bittercoder.com/

What drives us towards using IoC ?

- Difficulty responding to change.
- Tight coupling.
- Difficulty managing dependencies.
- Cant easily switch implementations at runtime.
- Need for Lifestyle management.

So what is IoC

It's a principle!

- Approach to designing components which are more reusable.
  - By moving control away from your components
- Better reusability by enforcing isolation.
- Key goal: **Loosen** the coupling between services.
- IoC Not really a design pattern.
- Dependency Injection (DI) helps achieve this isolation.
- Increases testability of components

Dependency Injection Example

In this example we look a service which can handle processing a purchase request for a lottery ticket.

- Checking we can still purchase the ticket for the selected draw number (has it already been drawn?)
- Collecting payment for the ticket.
- Send a notification message to the person buying the ticket with the outcome of their purchase.

First take on the Ticket Processor

```
public partial class LottoTicketProcessor : ITicketProcessor
{
    public void Process(Ticket ticket)
    {
        if (DrawManager.Instance
            .CanStillPurchaseFor(ticket.DrawNumber))
        {
            if (CollectionAgency.Instance
                .Collect(ticket.Customer, ticket.Cost))
            {
                ticket.TicketStatus = TicketStatus.Purchased;
                SendTicketDetailsNotification(ticket);
                return;
            }
        }

        SendPurchaseFailedNotification();
    }
}
```

Something smells..

There are problems with the implementation

- Heavy use of singletons
- Difficult, perhaps even impossible to test *
- Tightly coupled (where's the abstractions?)
- The ticket processor knows too much!
- Violates single responsibility principle
- Hard to reuse the ticket processor – not flexible.

** Well, maybe not impossible... There's always the big hammer known as TypeMock.Net (http://www.typemock.com/) – but being able to test it doesn't stop your code smelling!*

I heard factories were good…

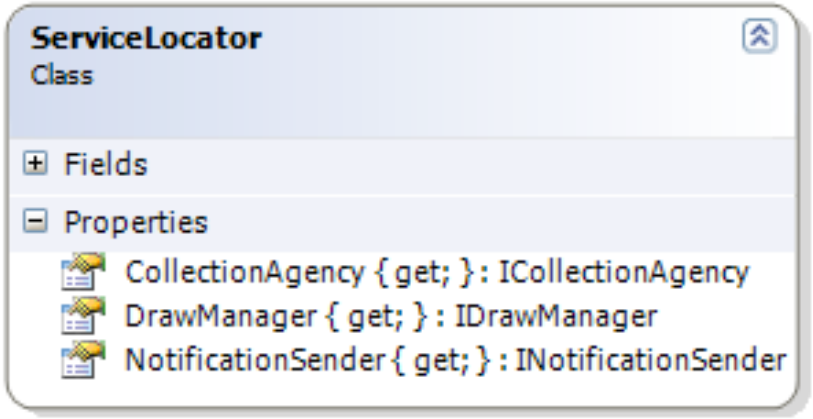
```
public partial class LottoTicketProcessor2 : ITicketProcessor
{
    public void Process(Ticket ticket)
    {
        if (DrawManagerFactory.Create().
            CanStillPurchaseFor(ticket.DrawNumber))
        {
            if (CollectionAgencyFactory.Create()
                .Collect(ticket.Customer, ticket.Cost))
            {
                ticket.TicketStatus = TicketStatus.Purchased;
                NotificationSenderFactory.Create()
                  .SendTicketDetailsNotification(ticket);
                return;
            }
        }
       NotificationSenderFactory.Create().SendPurchaseFailedNotification();
    }
}
```

We've just swapped knowledge of a singleton, for knowledge of a factory…

But at least we can change the concrete implementation used.

Service Locators

- Provides a way to manage dependencies.
- Concept already used in the .Net Framework.
- As the name suggest, a service locator is used... to locate services – could be seen as a "registry".



Example with Service Locator

```
public partial class LottoTicketProcessor3 : ITicketProcessor {
    private ServiceLocator _locator;
    public LottoTicketProcessor3(ServiceLocator locator) {
        _locator = locator;
    }
    public void Process(Ticket ticket) {
        if (_locator.DrawManager.CanStillPurchaseFor(ticket.DrawNumber))
        {
            if (_locator.CollectionAgency.Collect(ticket.Customer, ticket.Cost))
            {
                ticket.TicketStatus = TicketStatus.Purchased;
                _locator.NotificationSender
                      .SendTicketDetailsNotification(ticket);
                return;
            }
        }
        _locator.NotifcationSender.SendPurchaseFailedNotification(ticket);
    }
```

We can now test the ticket processor by mocking the service locator

Service Locators have issues

- All classes will depend on the service locator.
- Dependencies are generally resolved as-needed – requires additional coding to support "failing fast" when constructed.
- Often requires complicated code to setup as service count grows.
- Can impair reusability.

- But some people find it easier to "grok"

Dependency Injection

Range of mechanisms available...

- **The constructor**
  - Normally implies it's compulsory.
  - Should probably remove default constructor.
- **A property setter**
  - Often implying it's optional, but not always.
  - If optional should probably have a default value.
- **Other methods**
  - Interface (not that common)
  - Method (even less common)
  - Virtual property getter (very rare, needs a proxy to be generated)
  - And some even more obscure ideas...

Back to the example…

Constructor based injection

```
public partial class LottoTicketProcessor4 : ITicketProcessor
{
    private readonly ICollectionAgency _agency;
    private readonly IDrawManager _drawManager;
    private readonly IEmailSender _sender;

    public LottoTicketProcessor4(ICollectionAgency agency,
        IDrawManager drawManager, IEmailSender sender)
    {
        _agency = agency;
        _drawManager = drawManager;
        _sender = sender;
    }
```

Using read-only properties here - assert that these services should not change during lifetime..

And the implementation…

Simple eh?

```
public void Process(Ticket ticket)
{
    if (_drawManager.CanStillPurchaseFor(ticket.DrawNumber))
    {
        if (_agency.Collect(ticket.Customer, ticket.Cost))
        {
            ticket.TicketStatus = TicketStatus.Purchased;
            _sender.SendTicketDetailsNotification(ticket);
            return;
        }
    }
    _sender.SendPurchaseFailedNotification(ticket);
}
```

A taste of testing...

```
// start recording...
MockRepository repository = new MockRepository();
Ticket ticket = new Ticket(10); // Set Draw Number
IDrawManager drawManager = repository.CreateMock<IDrawManager>();
Expect.Call(drawManager.CanStillPurchaseFor(ticket.DrawNumber))
    .Return(false);
INotificationSender sender =
   repository.CreateMock<INotificationSender>();
sender.SendPurchaseFailedNotification(ticket);
LastCall.Constraints(Is.Same(ticket));
ICollectionAgency agency = repository.CreateMock<ICollectionAgency>();
repository.ReplayAll();

// start replaying...
LottoTicketProcessor4 processor =
    new LottoTicketProcessor4(agency, drawManager, sender);
processor.Process(ticket);
repository.VerifyAll();
```

Here we're setting expectations (that this method will be called)

Verify the expectations were met... throws if not

Designing for Dependency Injection

- Systems are comprised of small, focused services – Services are abstractions (Interfaces).
- Components implement the services – Concrete classes.
- A component declares exactly what dependencies (services) it relies on.
- A component does not dictate it's own lifestyle.
  - Don't implement singleton behaviour yourself.
- You can test all facets of your application.

IoC Containers

- Single point of access for all services in app.
- IoC containers can fill many needs, but simple containers generally handle:
  - Assembling services by resolving and injecting dependencies.
  - Managing the lifestyle of services.
- Some containers allow you to use configuration files.
- Automatically resolves all the dependencies - auto-wiring – constructors called for you via reflection.

Example container

```
public class SimpleContainer
{
    private IDrawManager _drawManager = new DrawManager();
    private INotificationSender _notificationSender =
         new SmsNotificationSender("cheapnsms.net.nz");
    private ICollectionAgency _collectionAgency = new MBillWebServiceCollectionAgency();

    public T Resolve<T>()
    {
        if (typeof(T) == typeof(IDrawManager)) return (T)_drawManager;
        if (typeof(T) == typeof(INotificationSender)) return (T)_notificationSender;
        if (typeof(T) == typeof(ICollectionAgency)) return (T)_collectionAgency;
        if (typeof(T) == typeof(ITicketProcessor))
        {
            ITicketProcessor processor =
                new LottoTicketProcessor4(_collectionAgency, _drawManager,
_notificationSender);
            return (T) processor;
        }
        throw new NoMatchingComponentException("Cant resolve service: {0}", typeof(T));
    }
```

Singleton lifestyle achieved by the container holding onto references...

Wire up a transient component every time it's requested...

Available containers (with .Net impls)

- Castle's Windsor Container & Micro Kernel - .Net
- ObjectBuilder – .Net (Microsofts Implementation)
  - Used in CAB & Enterprise Library
- Pico/Nano Container - Java, .Net, Ruby, PHP
- Seasar2 – Java, PHP, .Net (needs work!)
- Spring.Net – Java, .Net
- StructureMap - .Net

What a container can do for a project

- Save time and repetitive code to wire up dependencies.
- Provides an easy way to inject configuration information.
- Easily move between configurations
  - Test / Production
  - Different clients – different configurations.
- Transparently proxy components & intercept methods
  - Implement cross cutting concerns (logging, security) – AOP.
- Lifestyle management (Singleton, Transient, Per-Request, Per-Thread, Pooled etc.)
- Lifecycle management.

What they don't tell you

- Debugging container configuration problems can be frustrating
  - Often caused by no refactoring support.
- Components configured with the wrong lifestyle can introduce difficult to diagnose problems
  - may not become apparent during unit tests.
- Obfuscation and container configuration files don't mix.
- Cross-cutting concerns makes solutions less predictable, use AOP only if you understand the impacts.

Golden Rule

- **IoC should only be used** if you know how it helps you.
  - Otherwise you will think it's a pointless obstruction.

Resources

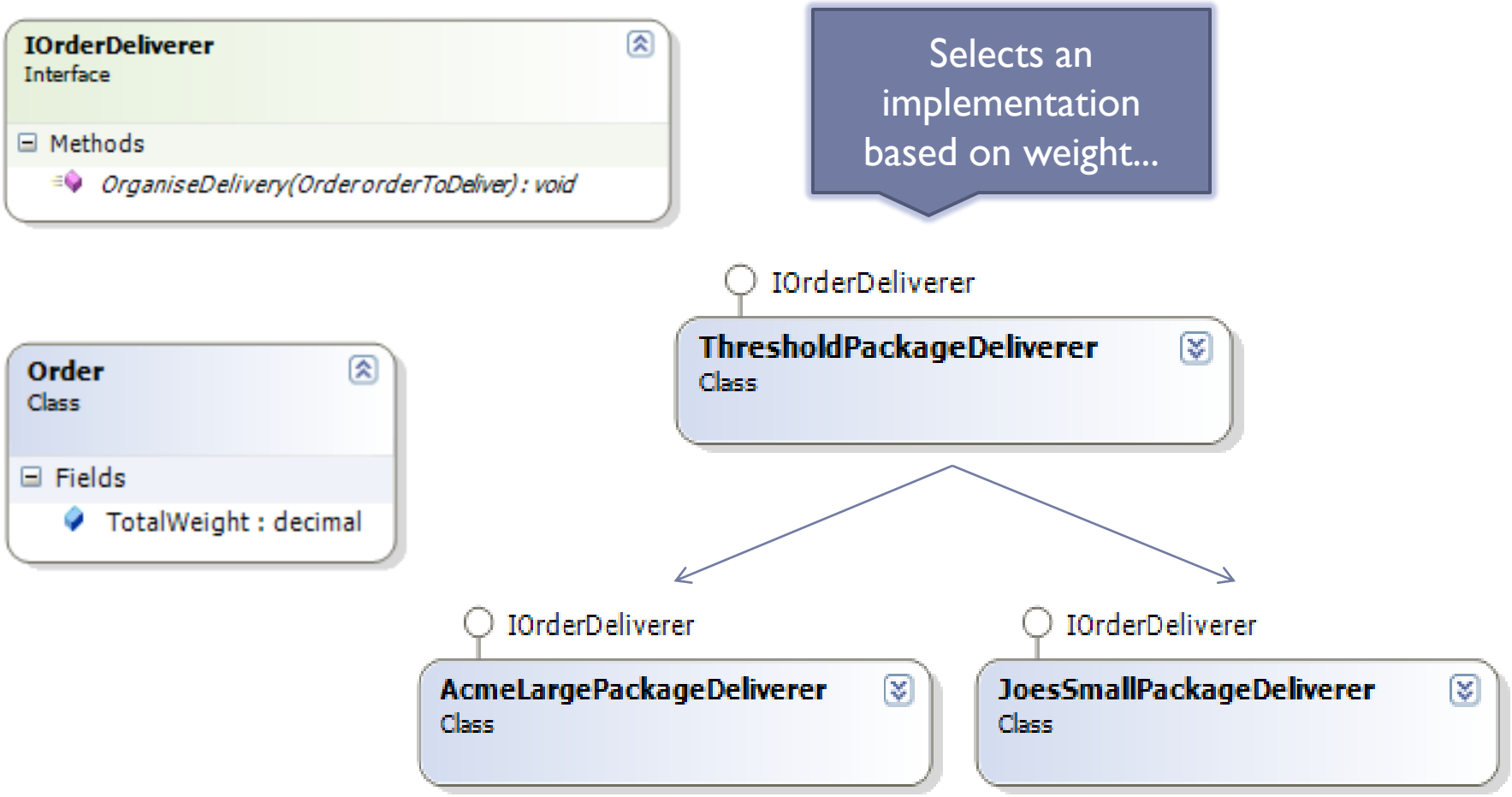
- Castle project
  - http://www.castleproject.org/
- Ayendes (Oren Eini) presentations & blog
  - http://www.ayende.com/
  - http://msdn2.microsoft.com/en-us/library/aa973811.aspx
- Object Builder
  - http://www.codeplex.com/ObjectBuilder
- Martin Fowler of Dependency Injection
  - http://www.martinfowler.com/articles/injection.html
- Stefano Mazzocchi – On Inversion of Control
  - http://www.betaversion.org/~stefano/linotype/news/38/

Evaluations and Contact

- Thanks for listening!
- Don't forget to fill in your evaluations
- Presentation downloads available from
  - My Blog
  - Also available from http://www.dot.net.nz within a week of this event.

- Email: alex@devdefined.com
- Work: http://www.devdefined.com/
- Blog: http://blog.bittercoder.com/

Decorator Patterns in IoC



Decorator Patterns in IoC

```
public class ThresholdPackageDeliverer : IOrderDeliverer {
    private readonly decimal thresholdInKgs;
    private readonly IOrderDeliverer smallDeliverer;
    private readonly IOrderDeliverer largeDeliverer;

    public ThresholdPackageDeliverer(IOrderDeliverer smallDeliverer,
     IOrderDeliverer  largeDeliverer, decimal thresholdInKgs) {
        this.smallDeliverer = smallDeliverer;
        this.largeDeliverer = largeDeliverer;
        this.thresholdInKgs = thresholdInKgs;
    }

    public void OrganiseDelivery(Order orderToDeliver) {
        if (orderToDeliver.TotalWeight < thresholdInKgs) {
            smallDeliverer.OrganiseDelivery(orderToDeliver);
        }
        else {
            largeDeliverer.OrganiseDelivery(orderToDeliver);
        }
    }
```

Notice both config and dependencies in constructor...

Decorator Patterns in IoC

```
<component
  id="joesDeliverer"
  service="MyApp.JoesSmallPackageDeliverer, MyAssembly"
  type="MyApp.JoesSmallPackageDeliverer, MyAssembly"/>

<component
  id ="acmeDeliverer"
  service="MyApp.AcmeLargePackageDeliverer, MyAssembly"
  type="MyApp.AcmeLargePackageDeliverer, MyAssembly"/>

<component
  id="defaultDeliverer"
  service="MyApp.IOrderDeliverer, MyAssembly"
  type="MyApp.ThresholdPackageDeliverer, MyAssembly">
  <parameters>
    <smallDeliverer>${joesDeliverer}</smallDeliverer>
    <largeDeliverer>${acmeDeliverer}</largeDeliverer>
    <thresholdInKgs>2000</thresholdInKgs>
  </parameters>
</component>
```

Threshold package deliver is wired up the joes and acme deliverer + threshold in the config.

Decorator Patterns in IoC

```
<!-- on strike, so just use joes deliverer -->
<component
  id="joesDeliverer"
  service="MyApp.IOrderDeliverer, MyAssembly"
  type="MyApp.JoesSmallPackageDeliverer, MyAssembly"/>

<!-- delivery company on strike ... uncomment after friday...
<component
  id ="acmeDeliverer"
  service="MyApp.AcmeLargePackageDeliverer, MyAssembly"
  type="MyApp.AcmeLargePackageDeliverer, MyAssembly"/>

<component
  id="OrderDeliverer.Default"
  service="MyApp.IOrderDeliverer, MyAssembly"
  type="MyApp.ThresholdPackageDeliverer, MyAssembly">
  <parameters>
    <smallDeliverer>${joesDeliverer}</smallDeliverer>
    <largeDeliverer>${acmeDeliverer}</largeDeliverer>
    <thresholdInKgs>20</thresholdInKgs>
  </parameters>
</component>-->
```

No recompile to handle the strike... We just swap implementations at runtime, then swap them back again after a week...

In this case the only real change is the service attribute on the joesDeliverer component.