# Dependency Injection: Why Is It Awesome and Why Should I Care?

Nolan Erck

# About Me

- Software Consultant (southofshasta.com)
  - Software Development, Training, Design
- ColdFusion, C++, Java, jQuery, PHP, .NET, HTML5, Android, SQL, etc...
- Manager of SacInteractive User Group
- Reformed Video Game Developer (Grim Fandango, SimPark, StarWars Rogue Squadron, etc).
- Music Junkie


I'M REAL PROUD OF MY RECORD COLLECTION

# Today's Agenda

- What is Dependency Injection?
- When/why do I want it in my projects?
- Intro to Bean Management
- Intro to Inversion of Control (IOC)
- Intro to Aspect Oriented Programming (AOP)
- Other info
- Questions

# Prerequisites

- Have some experience building classes and objects.
- Don't need to be an expert.
- Don't need to have used an MVC framework.
- D/I frameworks can be totally separate.
- And 1 more thing you need to know...

# Prerequisites

- ...Object Oriented Programming is hard.
- And some of it is confusing at first.
- And that's NORMAL.

# What Is Dependency Injection?

# What Is Dependency Injection?

- A “design pattern"
  - $6 phrase for “a way to organize your classes so they work together in a flexible way".
  - Like a for() loop, if(), array[]
  - Reusable technique, based around objects.
- There are lots of design patterns:
  - Dependency Injection is 1 design pattern
  - Inversion of Control (IOC), Aspect Oriented Programming (AOP) and Bean Management are all “variations on that theme".
- There are lots of others too.

# What Is Dependency Injection?

- It's not platform specific.
- Available for ColdFusion, Java, .NET, JavaScript, etc.
- For today's preso:
  - Using mostly pseudocode / concepts.
  - No specific code samples.
  - Some keywords will change.
  - All the concepts work the same across libraries and languages.

# When / Why Use D/I?

Short answer:
When you have lots of classes that talk to each other, want to keep code organized, and make it easy to swap different “types" of objects in/out of your application.

Longer answer:
That's what the rest of the slides are for...

# Does this mean...

I have to rebuild my entire app to use D/I?
*No. But you might get more benefits if you make a few changes to legacy code along the way.*

I have to learn a new way to code all my classes?
*No. You can leave everything as-is.*

*The nice thing about D/I is you can use it or not use it. If you want to write code the “old way" in part of your app, there's nothing stopping you from doing so. It will all still work as-is.*

Neat huh? Let's learn a bit more, shall we?


Bean Management

# What's a Bean?

A “thing" your D/I framework knows about.

```
class User{ ... }

class DateUtils { ... }
```

In your DI framework:

```
bean id="myUser" class="User"
bean id="myDateUtils" class="DateUtils"
```

(Exact syntax varies between DI libraries.)

# Bean Management - Example

MailService.cfc

```
<cfcomponent>
        <cffunction name="sendEmail">
                <cfargument name="from">
                <cfargument name="to">
                <cfagument name="body">
                [ code for sending the email ]
        </cffunction>
</cfcomponent>
```

Usage:

```
myMailService = CreateObject( “MailService" );
myMailService.sendEmail( to="...", from="...", body="..." );
```

# Bean Management - Example

```
myMailSvc = DILibrary.getBean( “MailService” );

myMailSvc.sendEmail( to=”...”,
       from=”...”,
       body=”...” );
```

Cute, but we haven't gained anything yet.
CreateObject() was 1 line of code.
DI.getBean() is 1 line if code.

No real improvements, but this is an easy way to start using DI:

Remove calls to CreateObject(), replace with calls to the DI library.

# Bean Management

```
class A {
  public class A() {...}
}

class B{
  public class B(A a) { ... }
}

class C {
  public class C(B b) { ... }
}

class D{
  public class D(C c) {...}
}
```

So if I *just* want a D object made...

# Bean Management

I have to do all this...

```
A a = new A();
B b = new B( a );
C c = new C( b );
D d = new D( c );
```

- 4 lines of code.
- The first 3 are “boilerplate” to create the variables I need to make a D.
- Multiply that out over all the objects in your app...
- = a LOT of boilerplate code!

Another way to say it:

*“D depends on A, B and C being made first”.*

# Bean Management

With a D/I library that does Bean Management, you'd write something like this instead...

```
D d = ColdSpring.getBean( “D” );
```

4 lines of code just became 1. And no more boilerplate stuff!

You tell your D/I framework “I need a D. Go find out what D depends on, make those as needed, and hand me back D when you're done.”

(“ColdSpring” is a placeholder here for any D/I framework. They all work about the same.)

# Bean Management

Quasi Real World Example.

Mailing department wide messages.

Mail Service CFC

- Needs a Department CFC to know which Department is being emailed.
- Department needs a Department DAO for it's SQL stuff.
- And a SecurityLogin CFC to make sure it's validated/logged in/whatever.

# Bean Management

## Old Way

```
mySecLogin = CreateObject( “component”, “SecurityLogin”);
myDeptDAO = CreateObject(“component”,“DepartmentDAO”);
dept = CreateObject( “component”, “Department” );

dept.init( mySecLogin, myDeptDAO );
mailSvc = CreateObject( “component”, “MailService” );
mailSvc.init( dept );
mailSvc.sendEmail();
```

# Bean Management

New Way

```
WireBox.getInstance( “mailService” ).sendMail();
```

Yes, 1 line of code.

Yes, it is THAT easy.

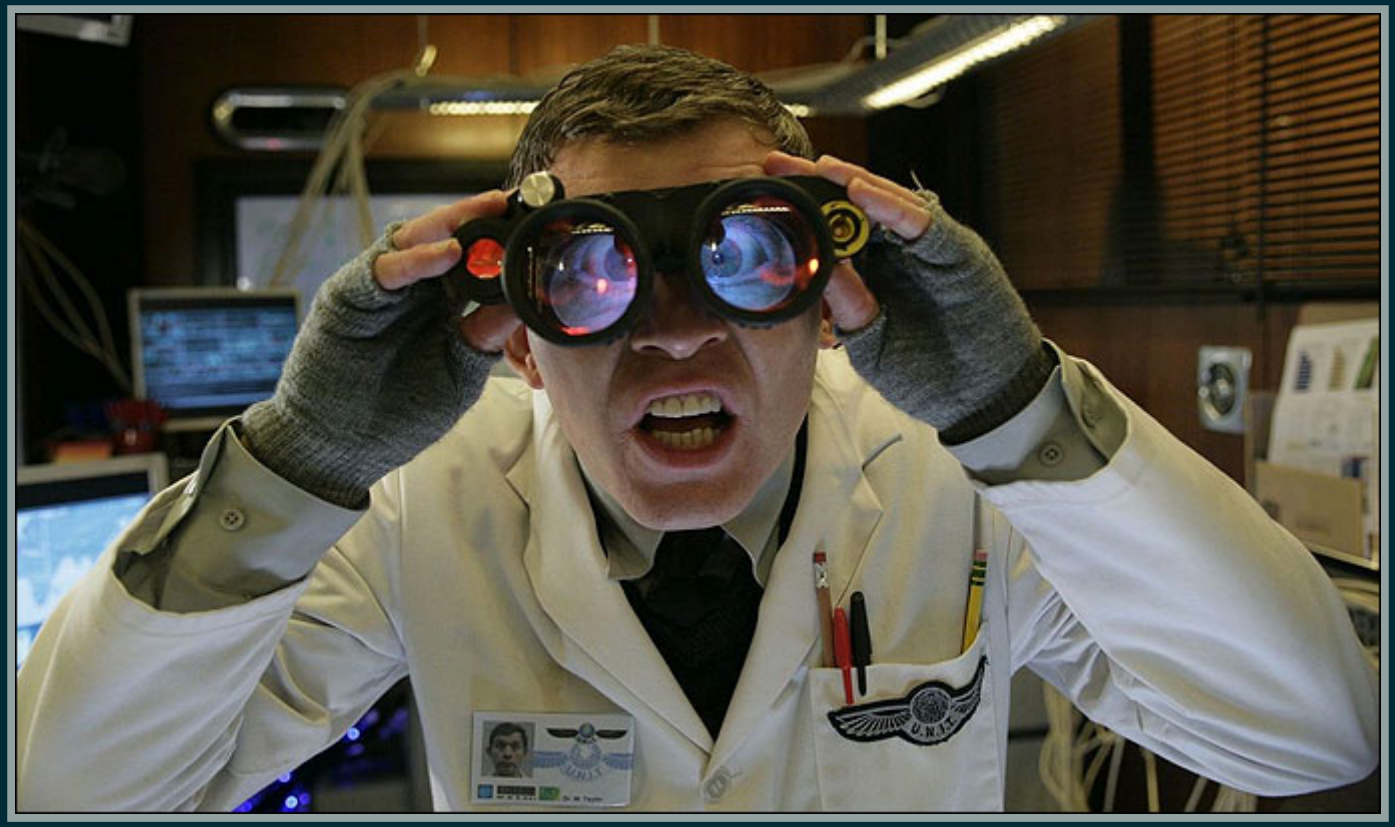
(I'm over simplifying a little.)

# Bean Management

So how does the D/I framework "know" that I need an A, B and C to make a D object?

- Varies a little from framework to framework.
- Configuration based:
  - XML file that "describes" the relationship between the objects. (ColdSpring, Spring)
  - Nice for, say, white-box testing w/ hands-on QA team.
- Convention based:
  - Works automatically by which folders you store classes in, and how the classes are named. (DI/1, WireBox)
- Just pick one and try it out, they all do the same core "things".

# Inversion of Control (IOC)



# Inversion of Control (IOC)

Fancy term for “flipping” some code around from the “traditional” or “normal” way you'd build objects.

Instead of building an Employee object, then a Department object, then putting the Employee IN the Department.

You say “get me a fully populated/constructed Department”.

Magic happens, automatically building the Employee object for you.

# Inversion of Control (IOC)

```
class RockGroup
{
   private String groupName;
   public String getGroupName(){
          return groupName;
   }

   public void setGroupName(String name) {
          this.groupName = name;
     }
}
```

- No Constructor.
- “groupName” is private.
- So we have to call the setter to initialize the variable with something.

# Inversion of Control (IOC)

With regular code, we'd do this:

```
String myBandName = “Depeche Mode”;
RockGroup rg = new RockGroup();
rg.setGroupName( myBandName );
```

That works but...

Every time I want to provide a different default value, I have to change code, and recompile.

# Inversion of Control (IOC)

What if we could provide that default value in a “config file”, outside of code?

And whenever we create a RockGroup object, it automatically gets a default value for groupName passed in at that instant?

You can!

# Inversion of Control (IOC)

Using a D/I framework, you can have a “config” that notes default values for your classes:

```
<class name="RockGroup">
        <property name="groupName" value="Depeche Mode"/>
</class>

<rg = DI.getBean( “RockGroup” );>
```

...any time I make a RockGroup, it will automatically initialize the “groupName” variable to the default value I provided. No extra lines of code. And changing this default is now done in a config file. No recompiling code to change it!

# Inversion of Control (IOC)

What about classes inside of other classes?

AKA “Composition”, the “has a” relationship.

Kind of the same thing...

# Inversion of Control (IOC)

```
class RockGroup{
        private String groupName;
        public String getGroupName(){ ... }
        public void setGroupName( String name ) {...}

        private Musician singer;
        public Musician getSinger(){ ... }
        public void setSinger( Musician m ) { ... }
}
class Musician{
        private String name;
        private int yearsSober;
        private int age;
        public String getName() { return name; }
        public void setName(String n){ this.name = n; }
        public int getYearsSober() { ... }
        public void setYearsSober(int y) { ... }
        public void setAge( int a ){ ... }
        public int getAge(){ return age; }
}
```

# Inversion of Control (IOC)

With straight code...

```
Musician m = new Musician();
m.name = “George Harrison”;
m.age = 64;
m.yearsSober = 10;

RockGroup rg = new RockGroup();
rg.setGroupName( “The Beatles” );
rg.setMusician( m );
```

# Inversion of Control (IOC)

With D/I...

```
RockGroup rg = Spring.getBean("RockGroup");
```

# Inversion of Control (IOC)

The Musician class has a String property, that gets initialized just like the String property in RockGroup. Ditto for the “int” properties, etc:

```
Spring.getBean( “Musician” );
```

“name” will always be whatever is in my config for the Musician class.

```
<class name="Musician">
        <property name="name" value="George Harrison" />
        <property name="age" value="64" />
        <property name="yearsSober" value="10" />
</class>
```

# Inversion of Control (IOC)

You can also refer to other classes in the config, and “inject” them into each other, just like we “injected” the String and int into Musician:

```
<class name="Musician" id="drummerBean1">
        <property name="name" value="Pete Best" />
        <property name="yearsSober" value="10" />
        <property name="age" value="64" />
</class>

<class name="RockGroup">
        <property name="groupName" value="The Beatles" />
        <property name="drummer" ref="drummerBean1" />
</class>
```

# Inversion of Control (IOC)

So now when I do this...

```
rg = DIFramework.getBean( “RockGroup” );
```

- rg will be fully constructed...
- It will have a “groupName” property set to “The Beatles”.
- It will have a “drummer” property set to “Pete Best”, he'll be 64 years old and 10 years sober.
- 1 line of code, no boilerplate stuff.
- If I want to change those defaults, I don't have to recompile, I just change a “config” setting.

# Inversion of Control (IOC)

Swapping out one type of Musician for another doesn't require recompiling code, just change a config:

```
<class name="Drummer" id="drummerBean1">
        <property name="name" value="Pete Best" />
        <property name="yearsSober" value="10" />
        <property name="age" value="55" />
</class>
<class name="RockGroup">
        <property name="groupName" value="The Beatles" />
        <property name="drummer" ref="drummerBean1" />
</class>
```

# Inversion of Control (IOC)

Swapping out one type of Musician for another doesn't require recompiling code, just change a config:

```
<class name="Drummer" id="drummerBean2">
        <property name="name" value="Ringo Starr" />
        <property name="yearsSober" value="22" />
        <property name="age" value="57" />
</class>
<class name="RockGroup">
        <property name="groupName" value="The Beatles" />
        <property name="drummer" ref="drummerBean2" />
</class>
```

Off topic: discussing if Pete is better/worse than Ringo.


Aspect Oriented Programming (AOP)
scientist's stuff

# Aspect Oriented Programming (AOP)

- AKA “Cross Cutting”.
- Change the “aspect” of how a function is called.
- Say I have a foo() method.
- Any time foo() is called...
  - Automatically...
    - Call code before or after foo()
    - Wrap some sort of code “around” foo()
    - e.g. try/catch blocks
  - It no longer just does “foo()”.
- Also does whatever you define as an aspect of calling foo().

# Aspect Oriented Programming (AOP)

Example: drawSalesReport()

In your AOP library:

```
<func name="drawSalesReport">
        <aspect before run="checkIfLoggedIn" />
        <aspect after run="saveToDebugLogFile" />
</func>
```

# Aspect Oriented Programming (AOP)

In your code, it USED to look like this:

```
checkIfLoggedIn();
drawSalesReport( UserID=555,
                 dtCreated='1/1/2015' );
SaveToDebugLogFile();
```

What happens now is...

```
DIFramework.runWithAdvice( “drawSalesReport”,
                           { UserID=555,
                           dtCreated='1/1/2015' } );
```

and that does all 3 things, in correct order.

# Aspect Oriented Programming (AOP)

But wait, there's more!

Can also wrap “blocks” of code around each other, not just function calls before/after each other.

e.g. try/catch blocks.

# Aspect Oriented Programming (AOP)

Say I have a query like so:

```
<cfquery name="qryGetUsers">
        SELECT * FROM Users
</cfquery>
```

In Production, I want that wrapped in a try/catch, but in QA/Development, I don't. (So I can see the debug output, etc.)

# Aspect Oriented Programming (AOP)

I'd have to set some kind of “flag”:

```
<cfif isProduction>
        <cftry>
                <cfquery name="qryGetUsers">
                        <SELECT * FROM Users>
                </cfquery>
        <cfcatch>
                <cflog ... />
        </cfcatch>
</cftry>
<cfelse>
        <cfquery name="qryGetUsers">
                <SELECT * FROM Users>
        </cfquery>
</cfif>
</cfelse>
```

Duplicate Code!

Remember the DRY rule!

# Aspect Oriented Programming (AOP)

Instead of calling a getUsers() method directly:

```
GetUsers();
```

In your AOP library:

```
<advice around functionName="getUsersWithAdvice">
        <method name="getUsers" />
</advice>
```

# Aspect Oriented Programming (AOP)

```
<advice around functionName="getUsersWithAdvice">
        <method name="getUsers" />
</advice>

function getUsersWithAdvice( funcToRun ) {
     if( isProduction ) {
         try{
              #funcToRun#();
         }
         catch{ ... }
     }else{
         #funcToRun#();
     }
}
```

# Aspect Oriented Programming (AOP)

Can also use it for...

SaaS feature flags - turn on functionality only when a customer pays for it. No need to edit code/database, just toggle a config.

Swap out 3rd party/vendor integrations easily.

Cranky vendor: “it will take you months to remove our tech from your product.”

App with AOP: "No, actually it will take me 5 minutes"

# So what's the catch?

A new way of thinking about how you build your classes

Takes some getting used to

A little extra "set up" work.
Naming conventions, or typing some XML.

Debugging errors can be slower at first due to learning curve

...but...

# So what's the catch?

...but...

It makes your code more reusable…

More flexible...

Easier to test...

Easier to turn features (aka "aspects") on/off easily...

# So what's the catch?

"But now I have tons of files in my app!"

So what?

Breathe.

Use version control to keep them safe.

Each file does "1 thing". Easy to know which file you'll need to edit.

# Remember...

- You don't have to do this all at once.
  - Start with (for example) Bean Management.
  - Add other bits as you get comfortable.
- It's not like an MVC framework where the whole app has to be considered.
  - Find 1 tiny spot in your app, add a little DI there.
  - Add more in increments, go as you learn.

# Also Remember...

Object-Oriented Programming is hard.

It's different than Procedural code.

Takes getting used to.

That's NORMAL.

Nobody learns all this stuff instantly.
It takes some time.
(But it is the way many languages are going these days.)

# Other Resources

Book: Spring In Action (Java)

Book: Head First Design Patterns

WireBox documentation

SpringByExample.org

Java code, but explanation of the general concepts is pretty good.

Framework/1, DI/1 and AOP/1

The ColdSpring docs are good too.

# Questions? Comments?

Ways to reach me...

Email: nolan@southofshasta.com
Twitter: @southofshasta
Blog: southofshasta.com

Slides and code samples on GitHub:
nolanerck/Dependency-Injection-Preso

Thanks!