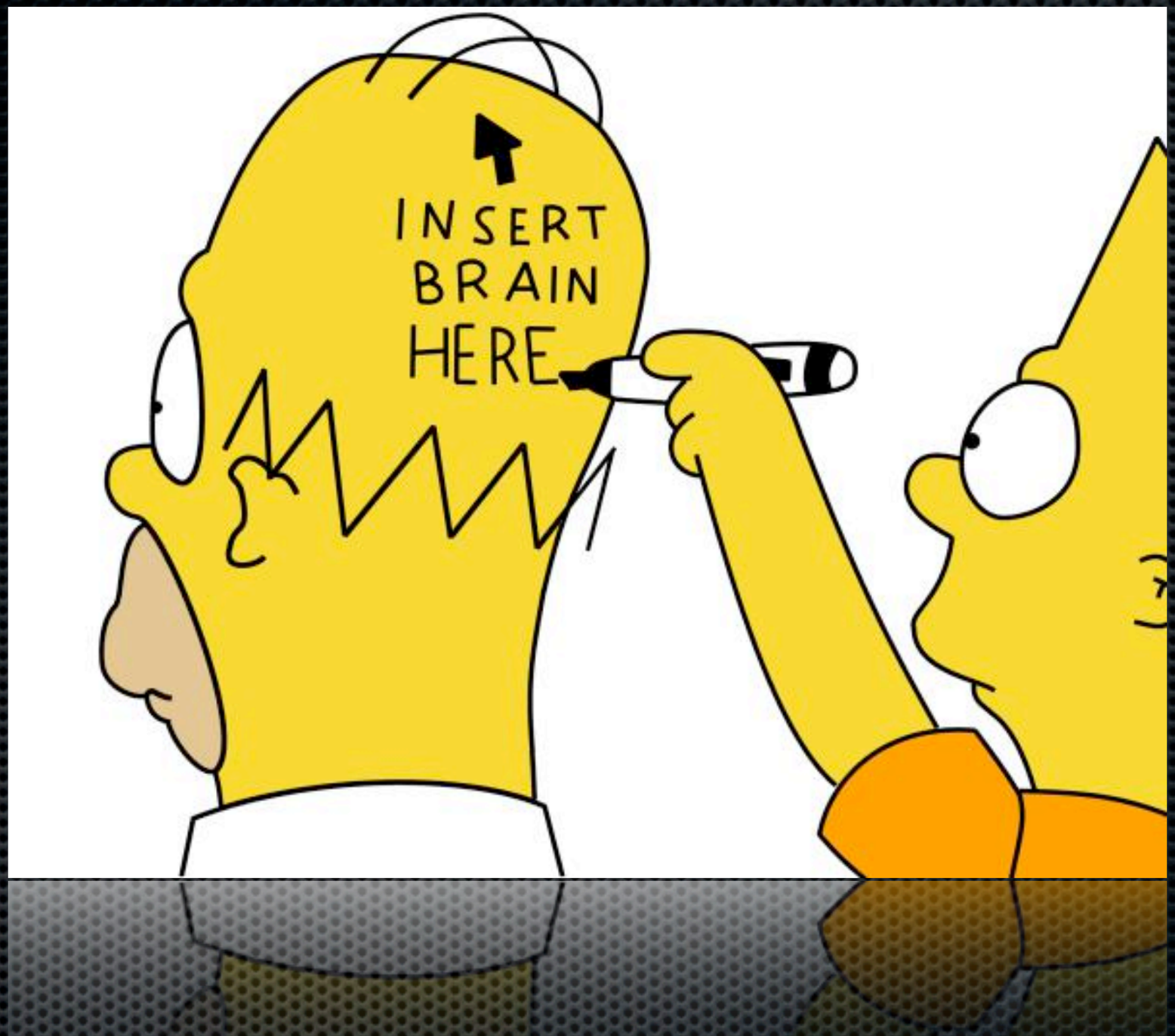


Dependency Injection

For dummies

Ortus Solutions
A Software Revolution

Who am I?

- Luis Majano - Computer Engineer
- Born in El Salvador ---------->

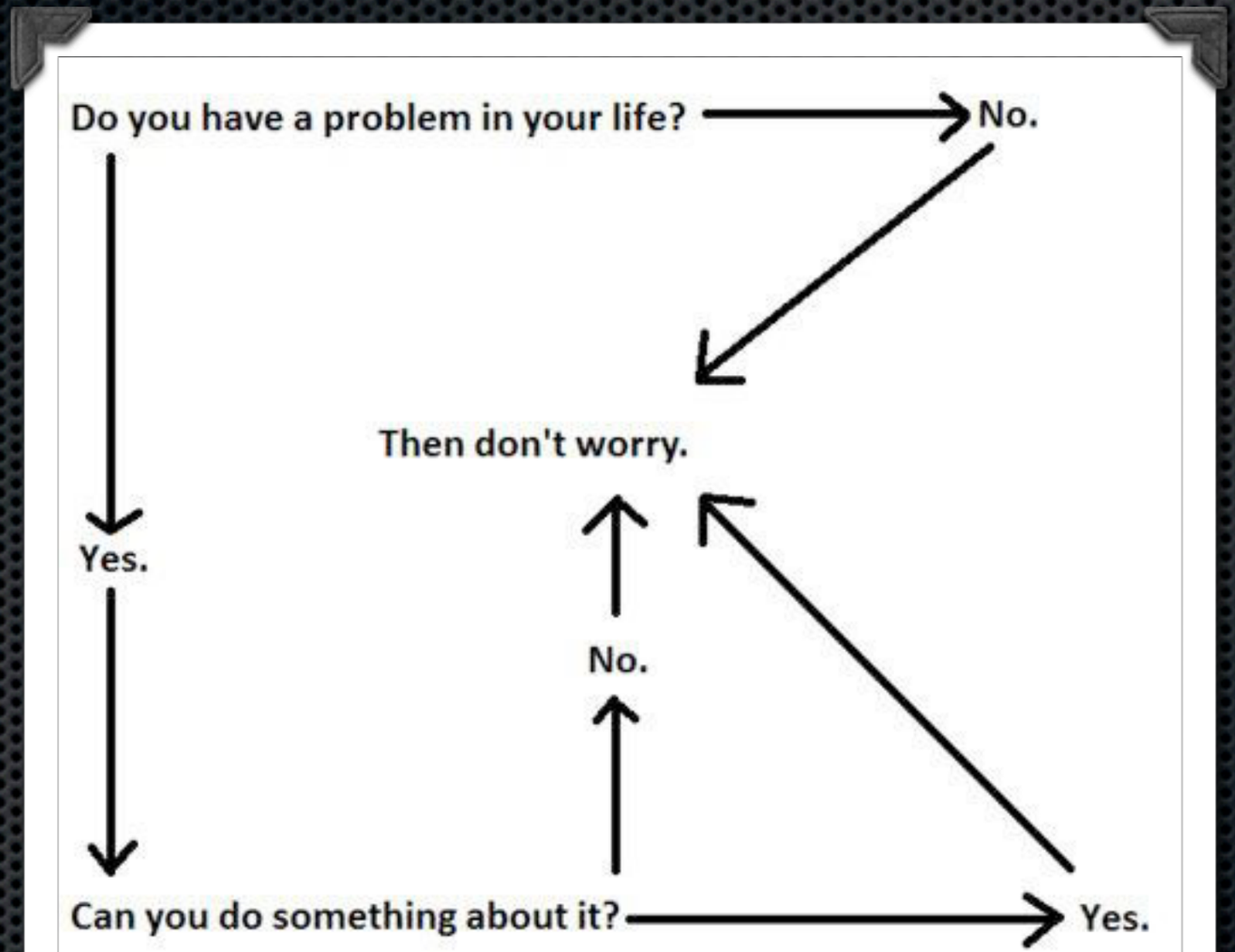
- President of Ortus Solutions
- Manager of the IECFUG (www.iecfug.com)
- Creator of ColdBox, MockBox, LogBox, CacheBox, WireBox, ContentBox, or anything Box!
- Documentation Lover!
  Don't be a hater!

Ortus Solutions
A Software Revolution

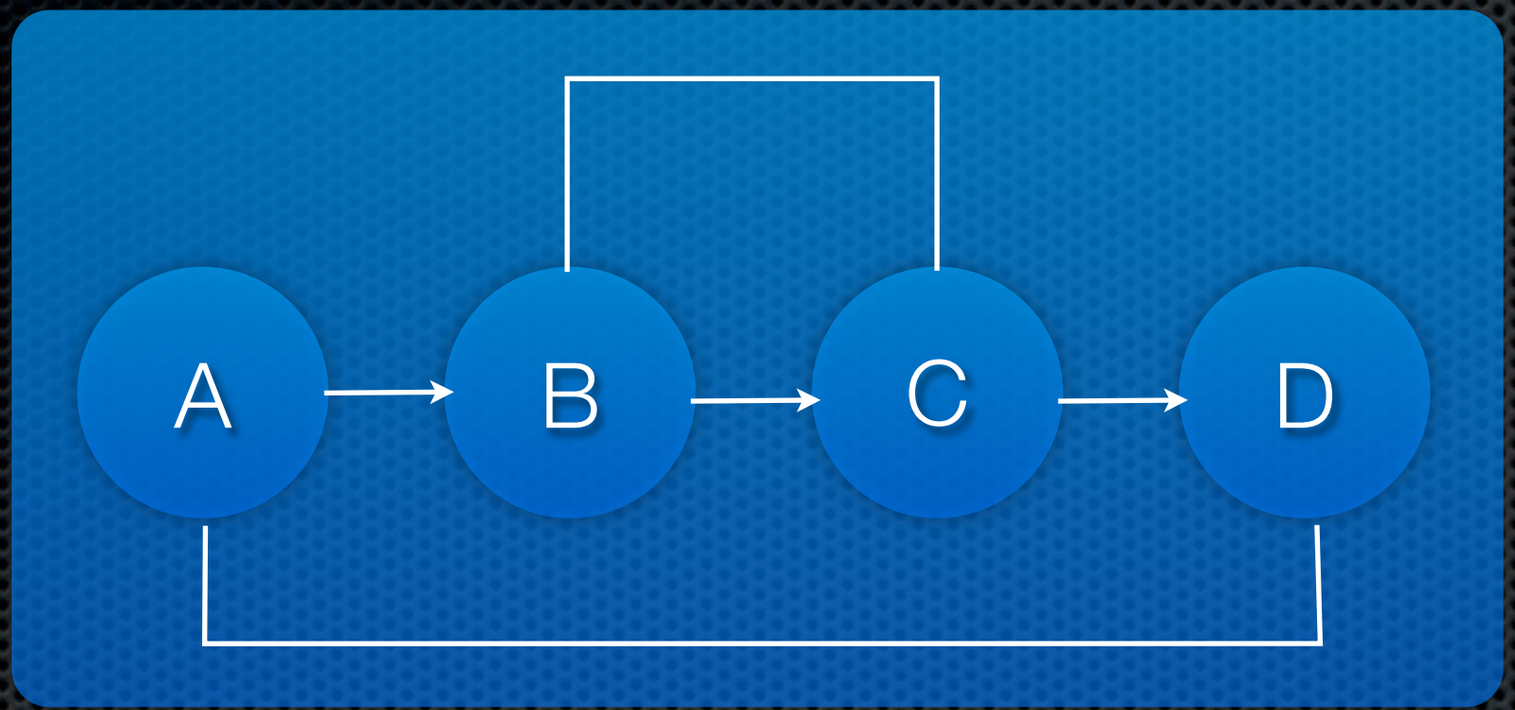
What we will cover?

- What is Dependency Injection (DI)
- Why DI?
- DI Evolution
- DI Basics
- Implementations

What is the problem?

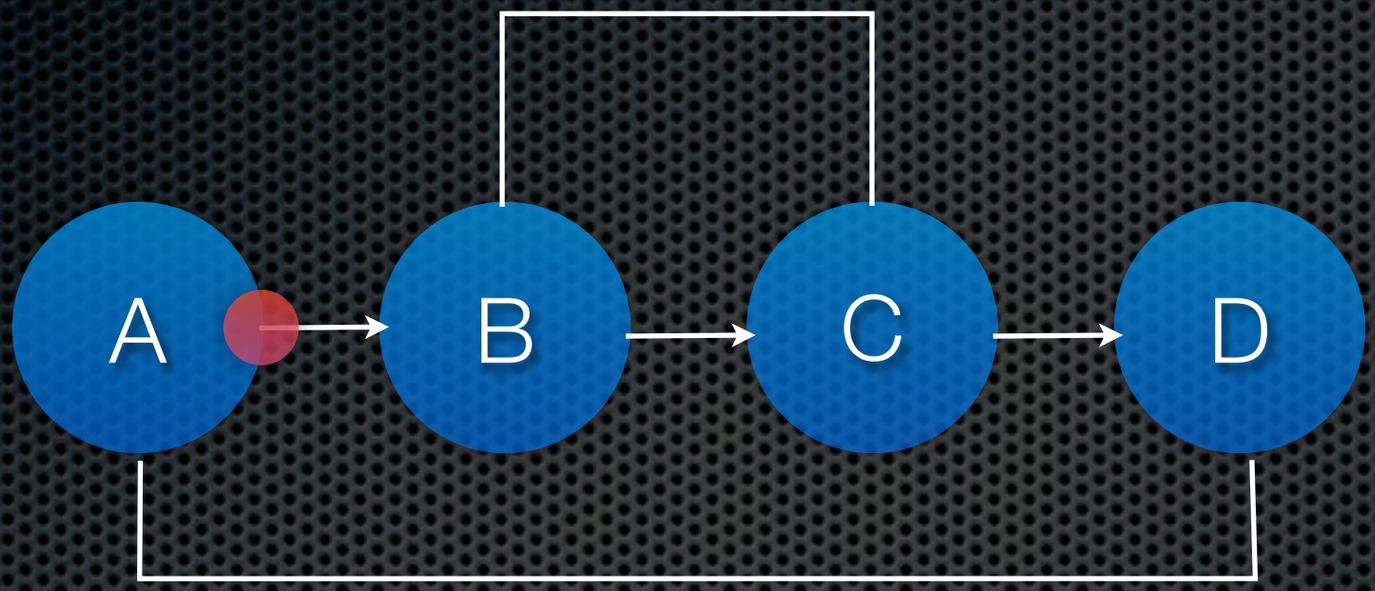


What is the problem?



- We are now working with lots of objects!
- An object may depend on other objects
- These objects might also depend on other objects
- An object without its dependencies cannot function
- This composite system -> **Object Graph**

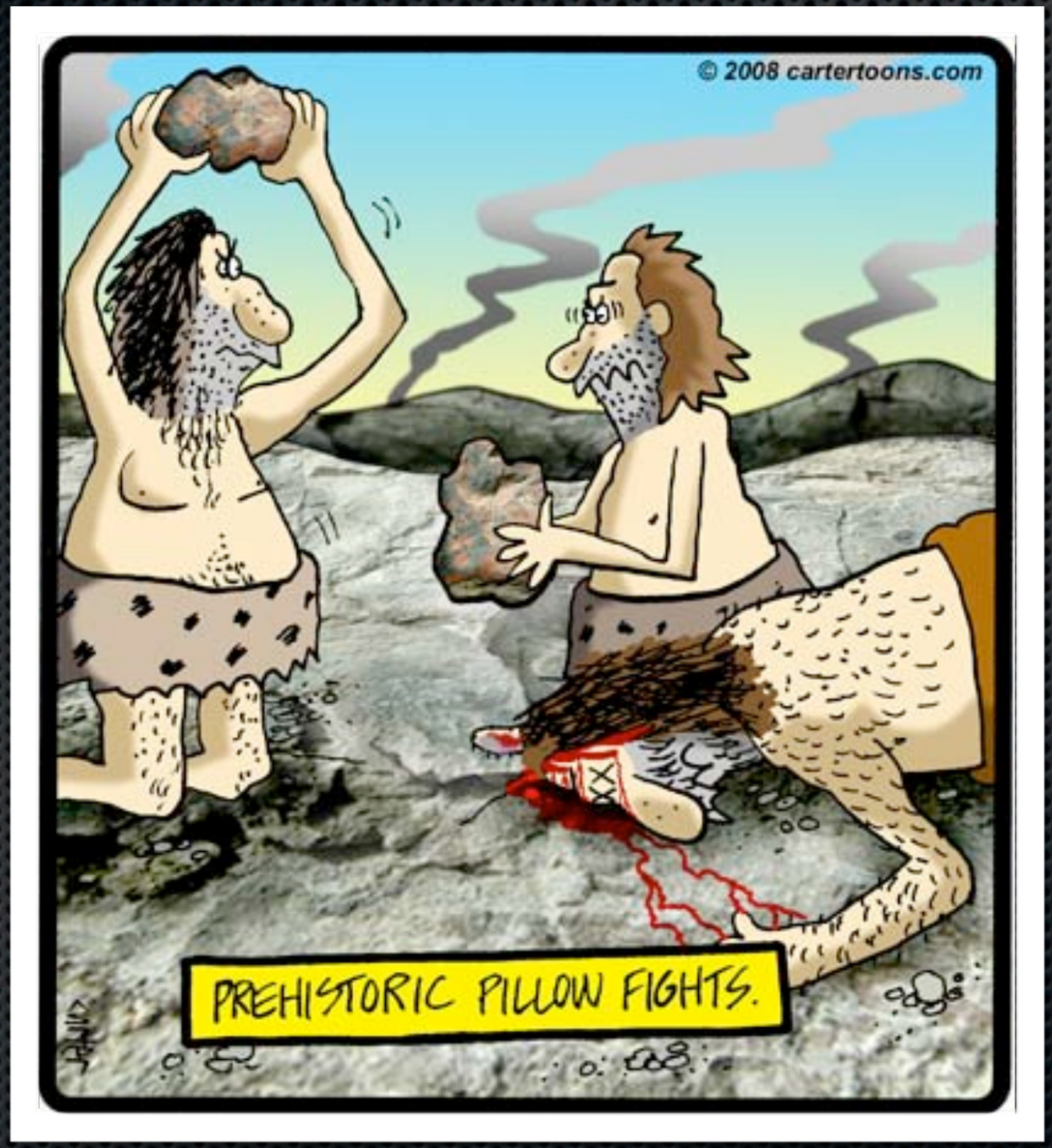
What is the problem?



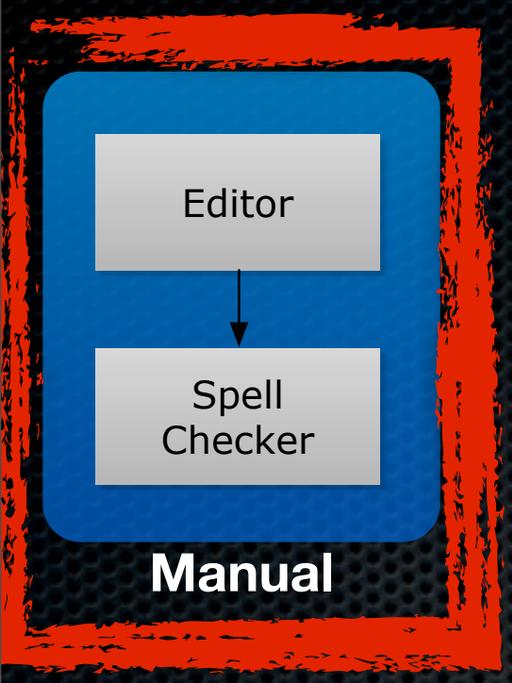
- Start by seeing objects as **services**
- An object is a **client** (dependency,dependent) of another object
- Objects have **identity** and **responsibilities** to accomplish specific tasks
- **Modeling** = Decompose services/functionality into object

DI = Reliably + Efficiently building object graphs, strategies and patterns.

PRE-DI Evolution



PRE-DI Evolution



PRE-DI Evolution

How do I test this?

What if those dependencies have more dependencies?

What if this has more dependencies?

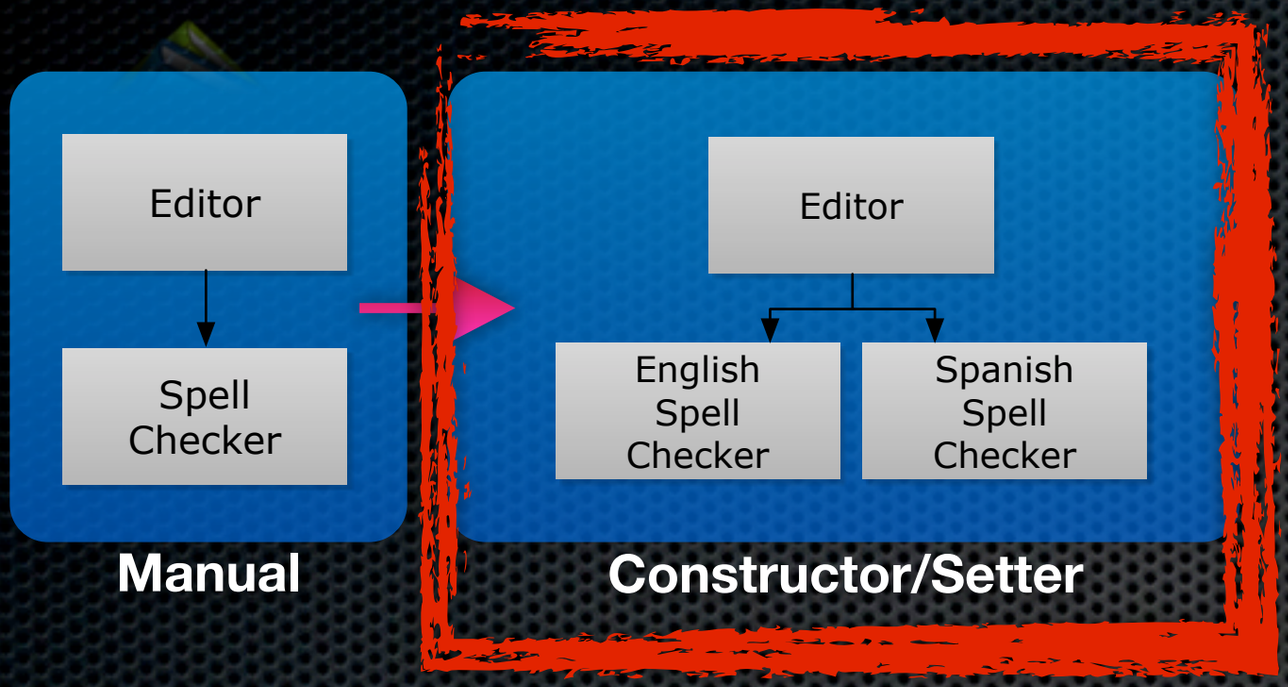
```
compone
    func

    }
}
```

How do I change Languages?

What if I need this in another object?

Getting Confused?

PRE-DI Evolution



PRE-DI Evolution

What about my instance data?

What if this has more dependencies?

```
component{
   function init(ISpellChecker checker){
      spellChecker = arguments.checker;
      cacheData = {};
      return this;
   }
}

component{
   function init(){
      cacheData = {};
      return this;
   }
   function setSpellChecker(ISpellChecker checker){
      variables.spellChecker = arguments.checker;
   }
}
```

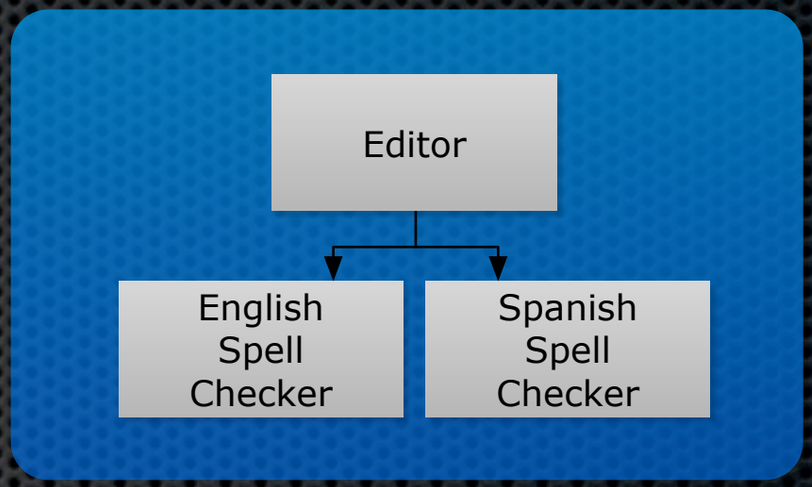
What if I need this in another object?

I can change languages!

Who creates checker?

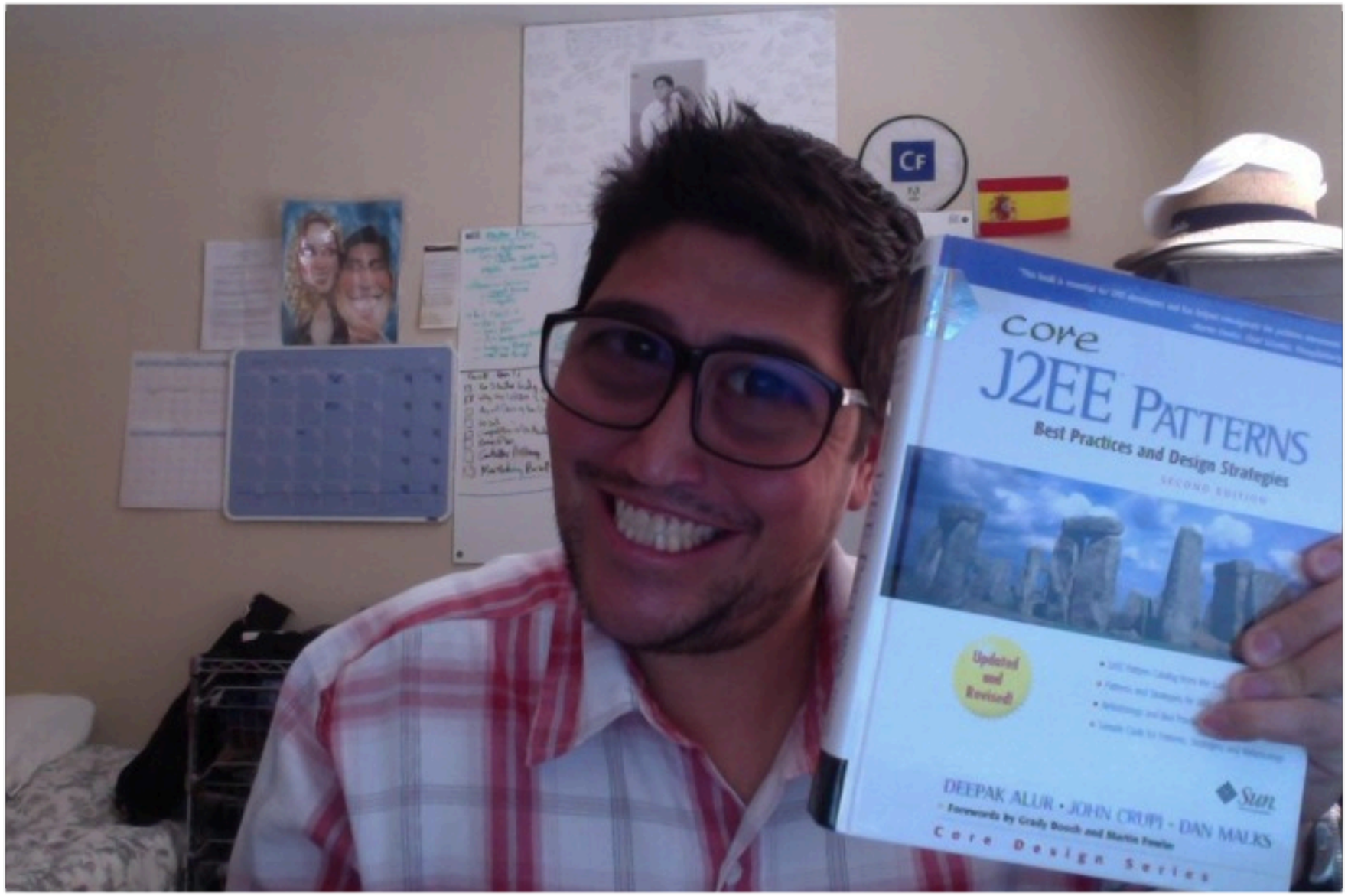
PRE-DI Evolution

- **Pros**:
  - We can test
  - We can use different spell checkers
- **Cons**:
  - Up to you to build dependencies
  - Need to know the entire object graph of every object
  - Use object in other locations (NOT DRY)
  - Violates encapsulation

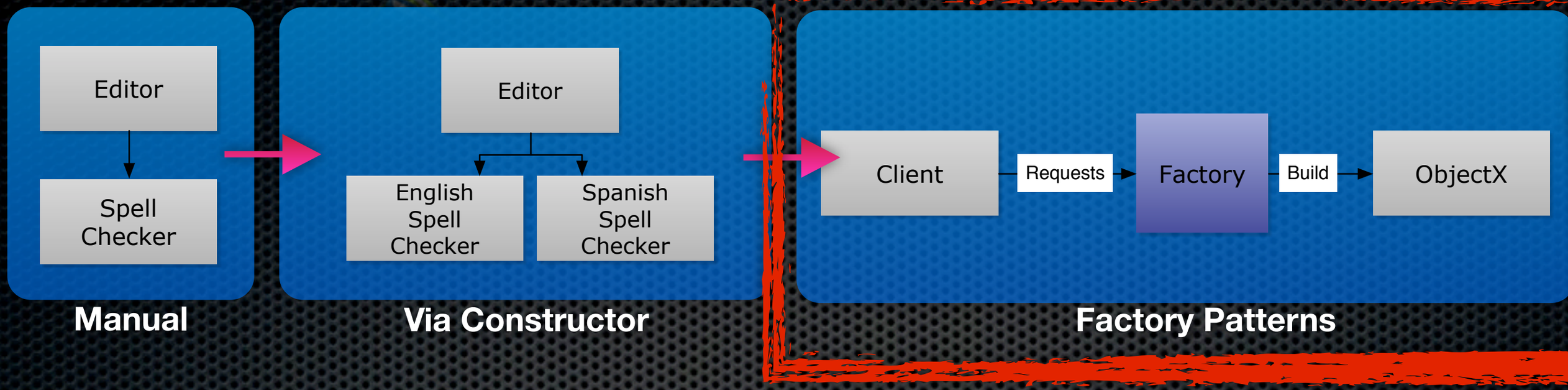


Constructor/Setter

Mr Smarty Pants



DI Evolution



PRE-DI E n

I can change languages!

Clean Separation of construction code

All clients need factories?

Who builds the factory?

What about persistence?

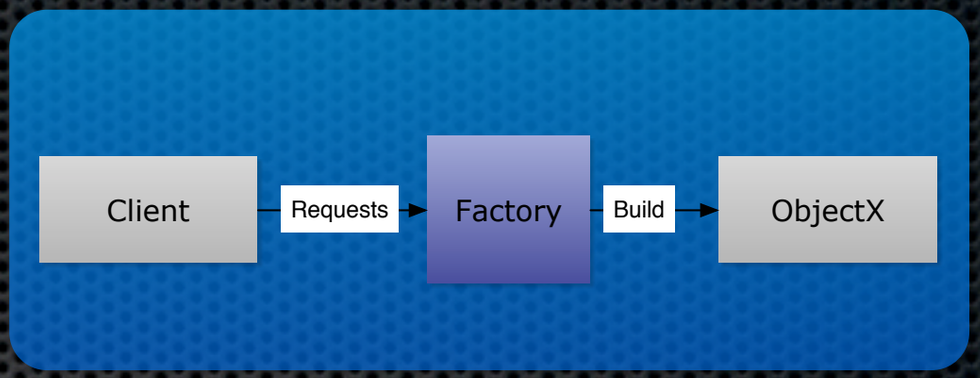
Creation gets verbose for new variations

How many factories do I need?

Dang it! More Problems!

```
component{
   function init(ISpellChecker checke ){
      spellChecker = arguments.checke r;
      cacheData = {};
      return this;
   }
}

component name="EditorFactory"{
   function init(MailerFactory mailerFactory){
      variables.mailerFactory = arguments.mailerFactory;
      return this;
   }
   function Editor newEnglishEditor(){
      var e = new Editor( new EnglishSpellChecker() );
      e.setAddressBook( new AddressBook() );
      e.setMailer( mailerFactory.newEnglishMailer() );
      return e;
   }
   function Editor newSpanishEditor(){
     var e = new Editor( new SpanishSpellChecker() );
     e.setAddressBook( new AddressBook() );
     e.setMailer( mailerFactory.newSpanishMailer() );
     return e;

      ditor newGermanEditor(){}
```

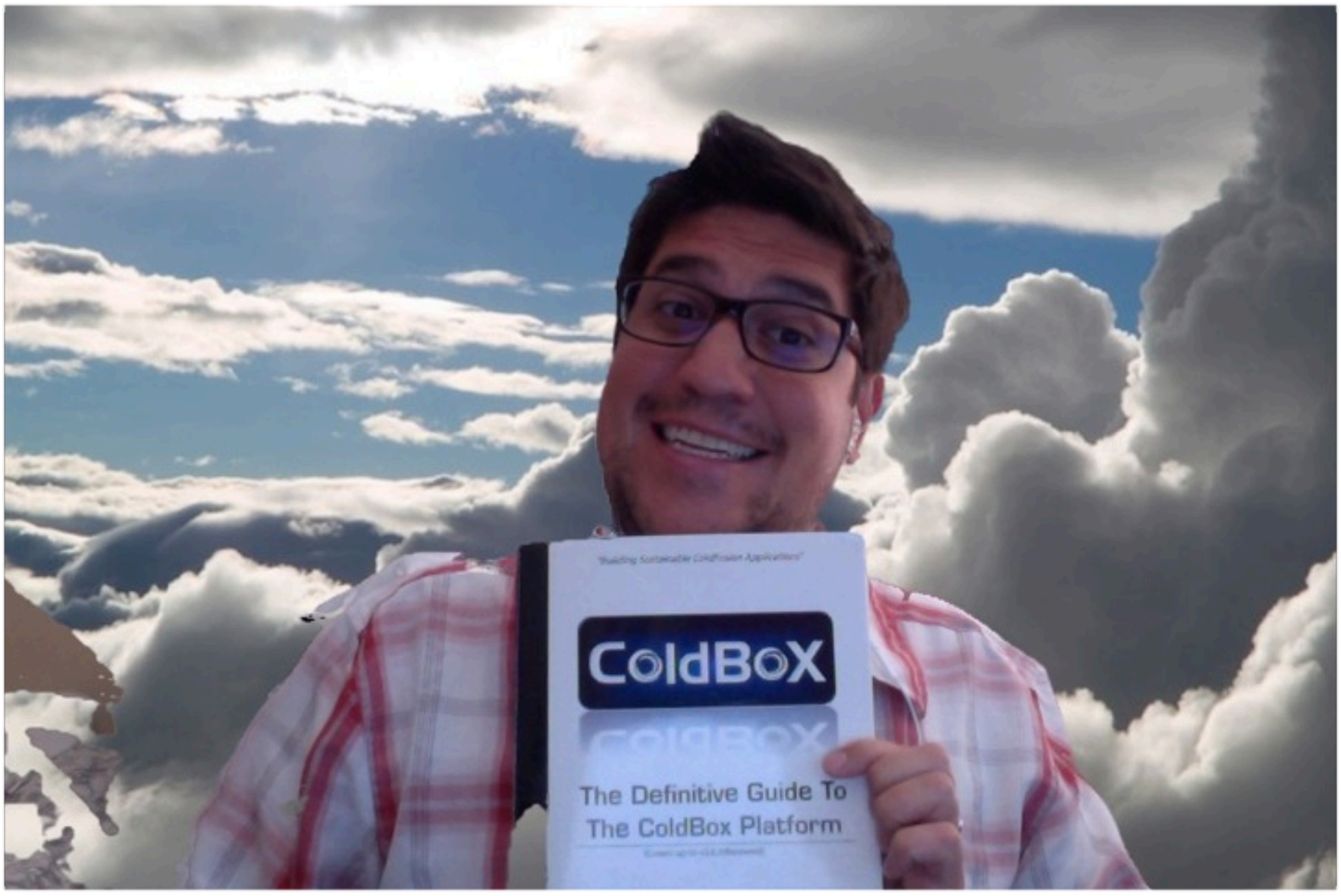
PRE-DI Evolution



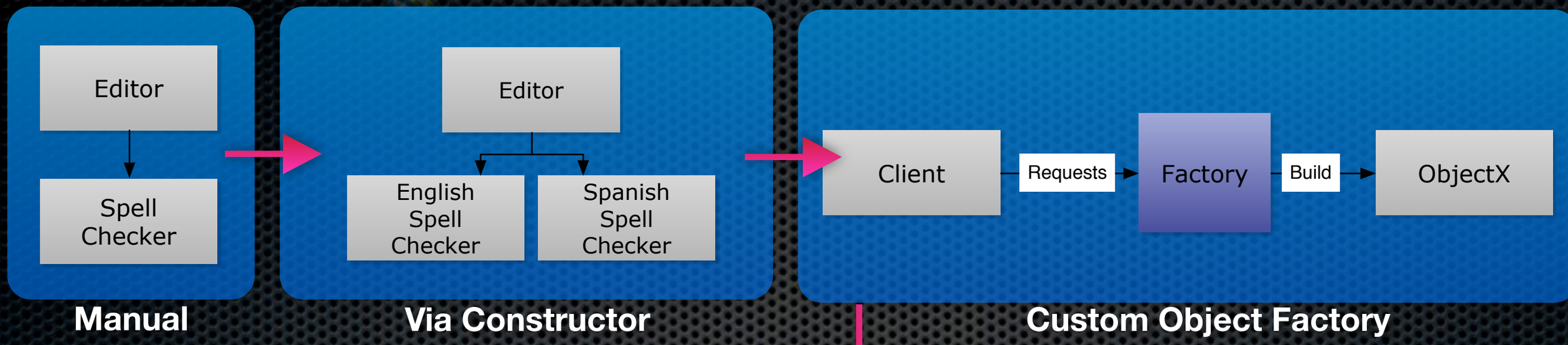
Factory

- **Pros**:
  - Factory creates and assembles objects
  - The client code remains clean and cohesive
  - No more repetitive code, we have achieved reusability
  - Encapsulation is maintained as client no longer build objects but request them
- **Cons**:
  - Our Factory becomes difficult to test as Editors are created and recreated
  - Persistence is not achieved
  - The factory must exist in every client that needs it
  - The factory must implement every variation of what it produces
  - The factory code will get messy and repetitive over time

Embrace DI



DI Evolution



Inversion Of Control

"An **abstract** principle describing an **aspect** of some software architecture designs in which the **flow of control** of a system is **inverted** in comparison to procedural programming." – Wikipedia

- A principle
- Common characteristic of frameworks
- What is Dependency Injection inverting?
- Any other patterns that invert?

Dependency Injection

"Dependency injection (DI) in **object-oriented** computer programming is a **technique** that indicates to a part of a program which other parts it can use, i.e. to supply an external **dependency**, or **reference**, to a software component." - Wikipedia

- A design pattern that applies IoC principles
- Inverts the responsibility of creating, assembling and wiring objects where needed
- DRY

DI Benefits

- Best parts of the evolution
- Testability and Mockability (Yes, that's a word!)
- Can enable object state or persistence
- Loose Coupling
  - Objects don't know about their dependencies
  - Easily switch dependencies
- DRY principles
  - Removes code clutter and write less boilerplate code
  - Ability to dynamically influence objects: AOP, Mixins, More

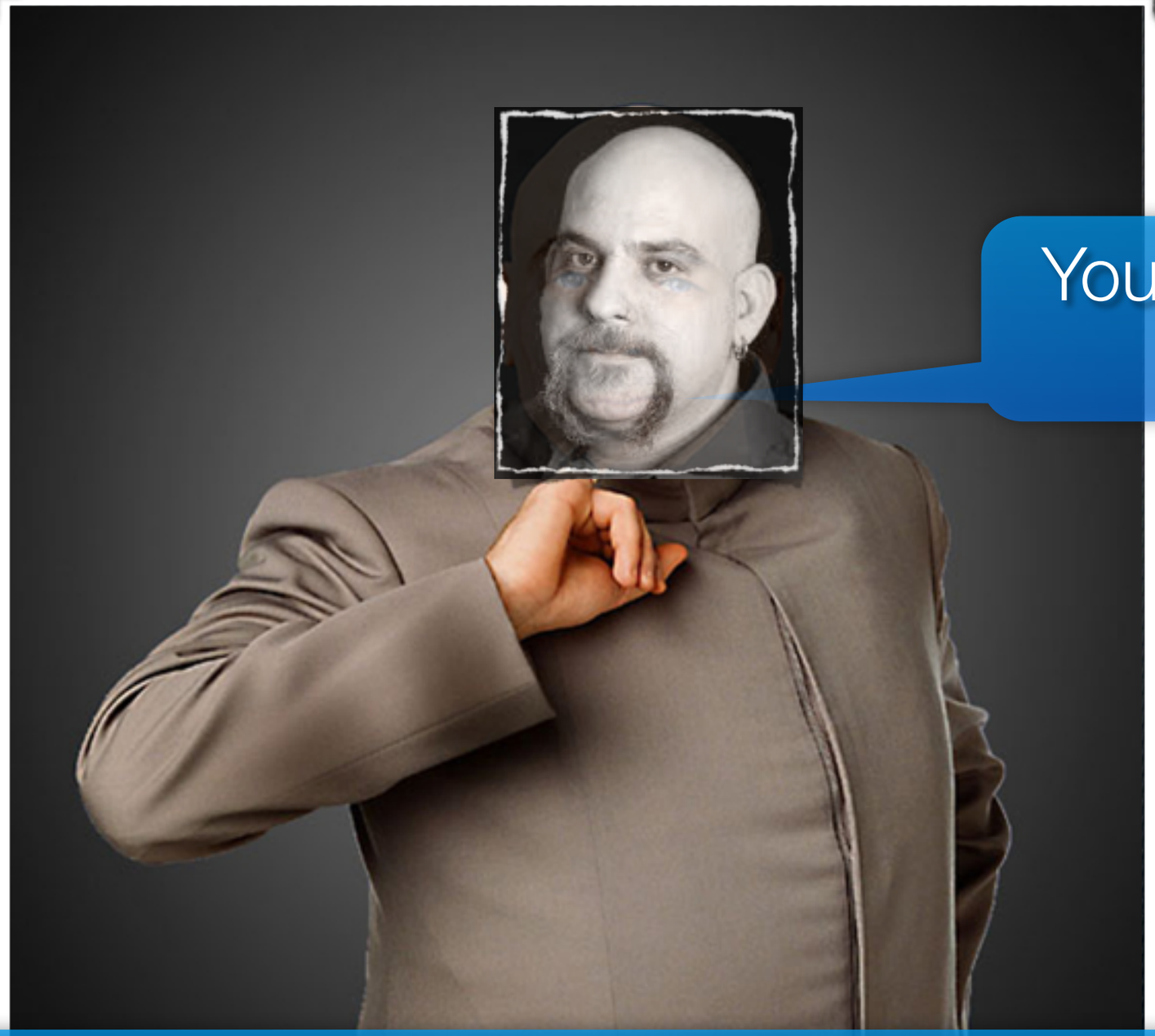
Applications become tolerant to rapid structural and behavioral change

DI Golden Rule #1

- Instead of pulling your dependencies in, you opt to receive them from someplace and you don't care where they come from.
- Instead of **pull** you **push**
- **Hollywood principle -> IoC = Inversion of Control**

Don't call us; we'll call you!

DI Golden Rule #2



createObject/new() is evil

DI Universe

Annotated Code

Configuration XML or Programmatic

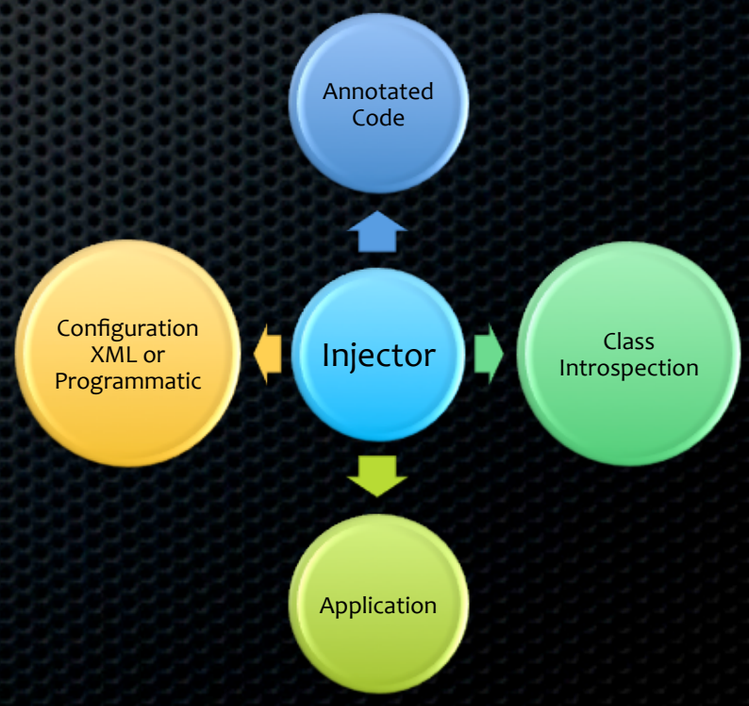
Injector

Class Introspection

Application

DI Basics

- Ask Injector for objects by **named** keys
- Injector manages object persistence for you -> **Scoping**
- Dependency Discovery
  - Introspection (Annotations+Methods+Arguments)
  - Configuration (XML or Programmatic)
- Injector **Autowires** = Automatic resolving and injecting of dependencies
- Dependency Injection Idioms:
  - Constructor arguments
  - Setters/Methods
  - Mixins (CFProperty)



DI IDIOMS

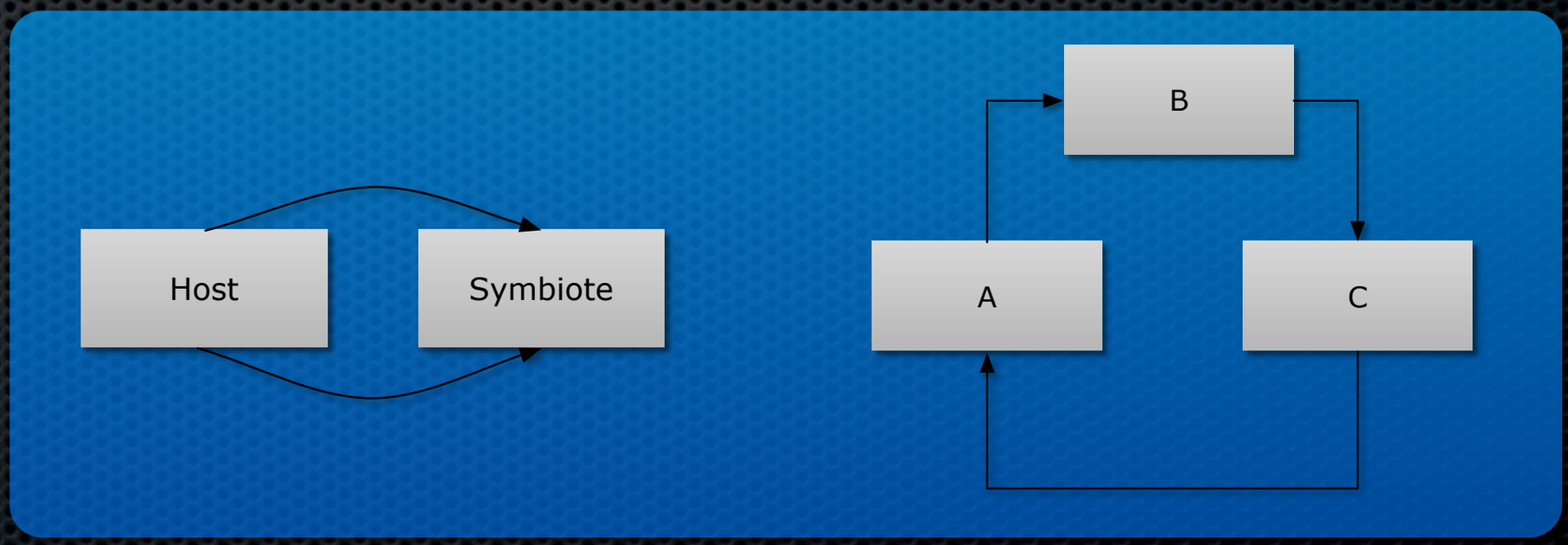
```
// Constructor
function init(service){
   variables.service = arguments.service;
   return init;
}
```

```
// Setters
function setService(service) inject="service"{
  variables.service = arguments.service;
}
```

```
// CF Property Injection
property name="service" inject;
property name="service" inject="id:CoolService";
property name="log" inject="logbox:logger:{this}";
```

Circular Dependencies

- Two objects depend on each other
- Refer to the same instances of each other
- Constructor injection is not possible (maybe)



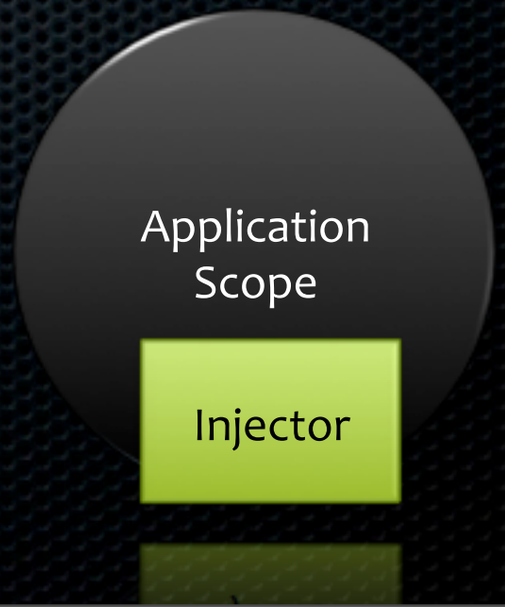
What is WireBox?

"A next generation conventions based dependency injection and AOP framework for ColdFusion"

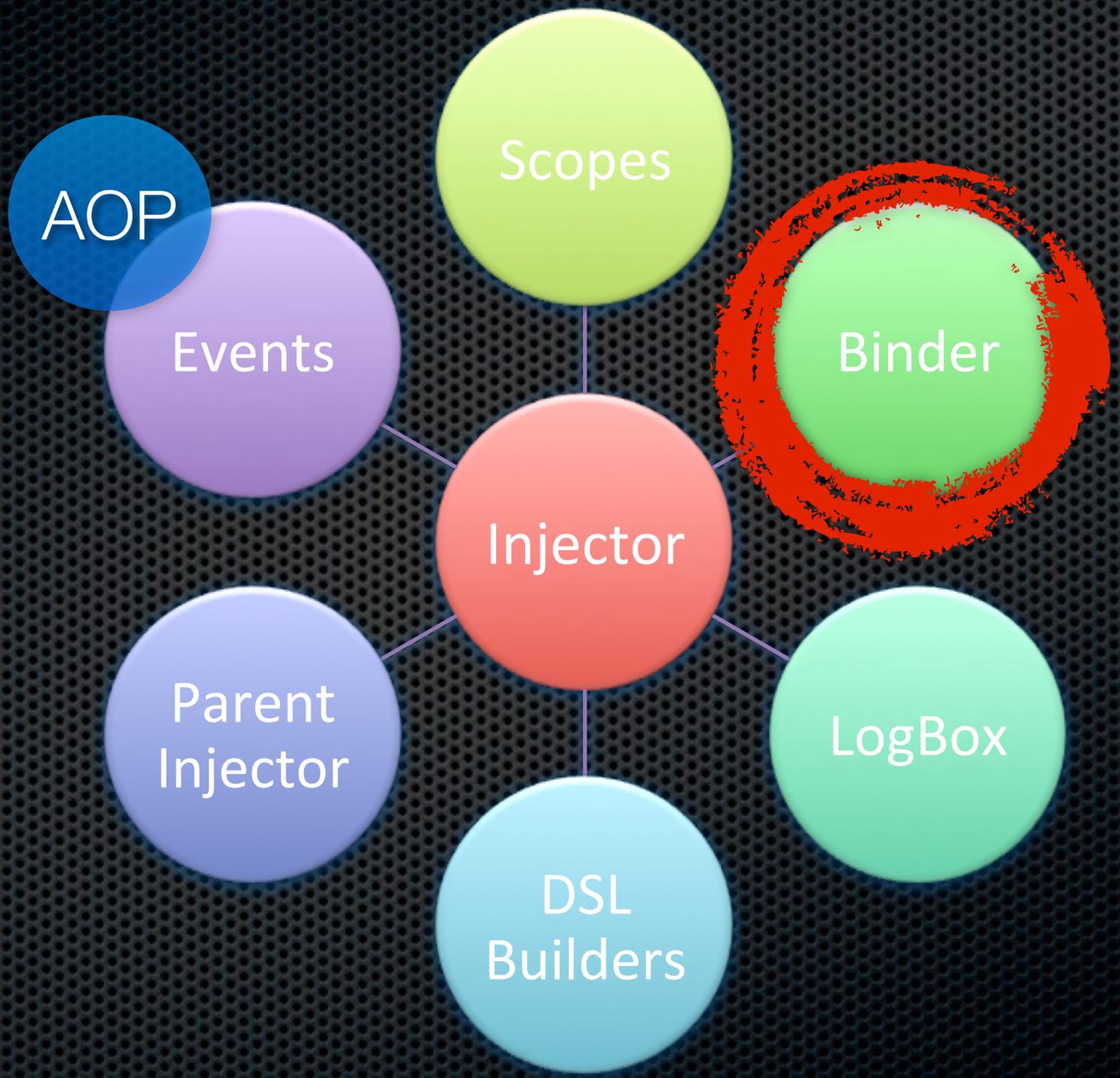
"Build objects the MACHO way!"

Features

- **Documentation** & Professional Services
- Annotation driven DI
- 0 configuration or programmatic configuration mode (NO XML)
- More than CFCs
- Persistence scopes: singleton, session, request, cache, etc.
- Integrated logging and debugging
- Object Life Cycle Events
- Automatic CF Scope registration



WireBox Universe

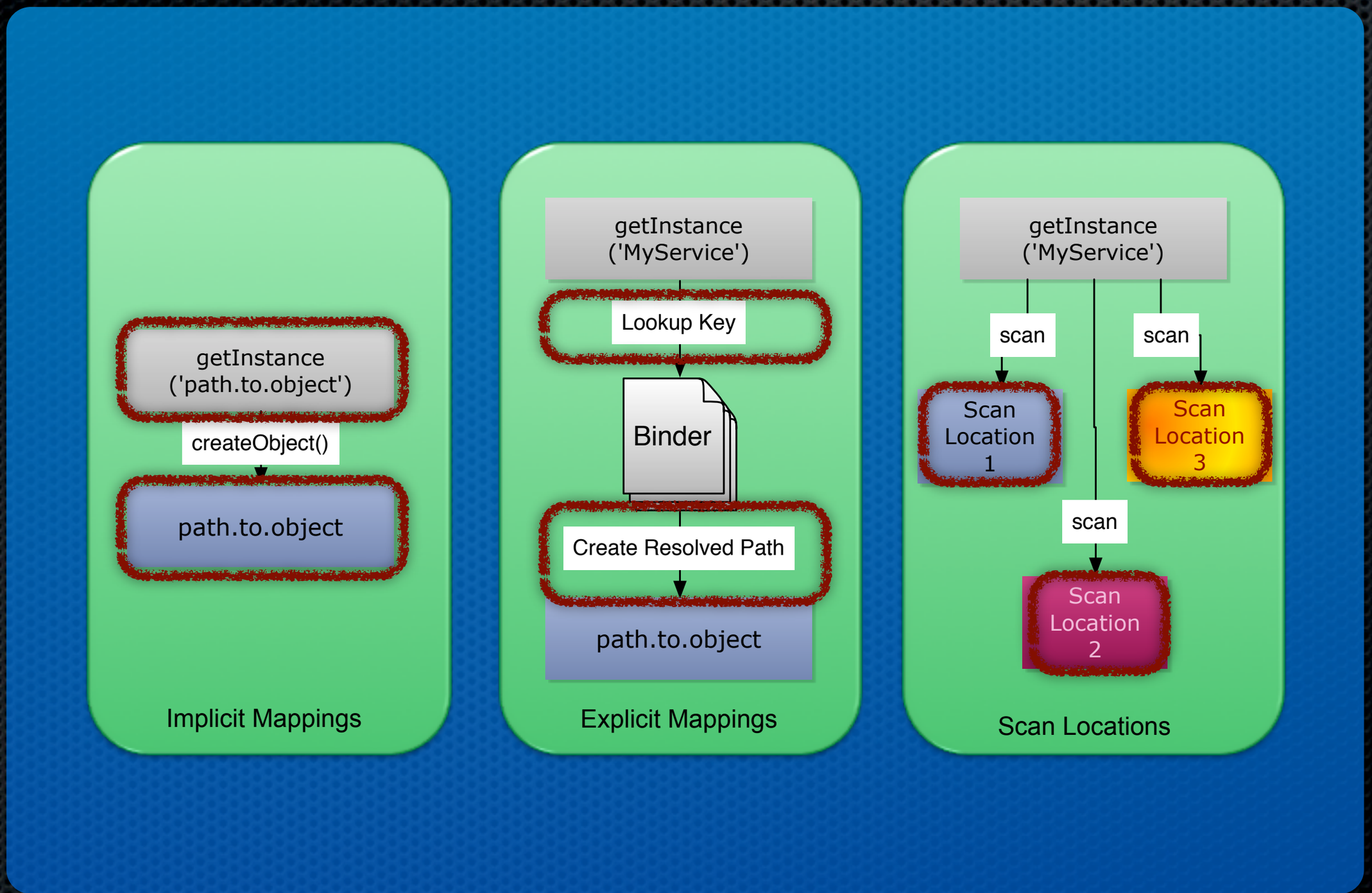


WireBox Injector

- Creates and wires all objects for you
- **getInstance(‘named key or path’)**

```
injector = new wirebox.system.ioc.Injector();
injector = new wirebox.system.ioc.Injector(binder=”path.to.Binder”,
            properties={prop1=val1,prop2=val2});

obj = injector.getInstance(“model.path.Service”);

obj2 = injector.getInstance(“NamedKey”);
```

Creation Styles



Configuration Binder

- Simple CFC
  - **Configure()**
- Define WireBox Settings
- Define Object Mappings
- Mapping DSL

```
component extends="wirebox.system.ioc.config.Binder"{

    configure(){

    }
}
```

Mapping DSL

- Used by concatenating calls to itself, returns the binder always
- Very readable bursts of logic:
  - ***map(“Luis”).toJava(“cool.Java.App”).into(this.SCOPES.SESSION);***
- Extend it!

```
component extends=”wirebox.system.ioc.config.Binder”{

    configure(){

      map(“Luis”).toJava(“cool.java.Service”)
         .asSingleton()
         .asEagerInit();

    }

}
```

More than CFCs

Type	Description
CFC	ColdFusion Component
JAVA	Any Java object
WEBSERVICE	Any WSDL
RSS	Any RSS or Atom feed
DSL	Any registered or core injection DSL string
CONSTANT	Any value
FACTORY	Any other mapped object
PROVIDER	A registered provider object

```
// RSS Integration With Caching.
map("googleNews")
    .toRSS("http://news.google.com/news?pz=1&ned=us&hl=en&topic=h&num=3&output=rss")
    .asEagerInit()
    .inCacheBox(timeout=20,lastAccessTimeout=30,provider="default",key="google-news");

// Java Integration with init arguments
map("Buffer").
    toJava("java.lang.StringBuffer").
    initArg(value="500",javaCast="long");

// Java integration with initWith() custom arguments and your own casting.
map("Buffer").
    toJava("java.lang.StringBuffer").
    initWith( javaCast("long",500) );

// Constant
map("MyEmail").
    toConsant("info@coldbox.org");

// Factory Methods
map("FunkyEpresso").
    toFactorymethod(factory="SecurityServic",method=”getEspresso”).
    methodArg(name=”funkyLevel”,value=”45”);

// Property injections
map("SecurityService")
    .to("model.security.SecurityService")
    .in(this.SCOPES.SERVER)
    .property(name="userService", ref="UserService", scope="instance")
    .property(name="logger", dsl="LogBox:root", scope="instance")
    .property(name="cache", dsl="CacheBox:Default", scope="instance")
    .property(name="maxHits", value=20, scope="instance"
```

What about persistence?

Scope	Comment
NOSCOPE	Transient objects
PROTOTYPE	Transient objects
SINGLETON	Only one instance of the object exists
SESSION	The CF Scope
REQUEST	The CF Scope
APPLICATION	The CF Scope
SERVER	The CF Scope
CACHEBOX	Time persisted objects in any CacheBox provider

Scoping by Mapping

```
// map google news
map(“GoogleNews”)
      .toRSS(“http://news.google.com/news?output=rss”)
      .asEagerInit()
      .inCacheBox(timeout=”30”,lastAccessTimeout=10);

// Wire up java objects
map(“SecurityService”)
   .toJava(“org.company.SecurityService)
   .asSingleton()
   .setter(name=”userService”,ref=”userService”);
map(“UserService”)
   .asSingleton()
   .toJava(“org.company.UserService”);

// request based objects
map(“SearchCriteria”)
   .to(“model.search.Criteria”)
   .into(this.SCOPES.REQUEST);

// session based objects
map(“UserPreferences”)
   .to(“model.user.Preferences”)
   .into(this.SCOPES.SESSION);
```

Scoping by Annotation

```
component{}

component singleton{}

component scope="session"{}

component scope="request"{}

component cache cacheTimeout="30"{}

component cachebox="ehCache" cacheTimeout="30"{}
```

Injection Styles

Style	Order	Motivation	Comments
Constructor	1	Mandatory dependencies for object creation	Each argument receives an inject annotation with its required injection DSL. Circular dependencies will fail via constructor injection unless WireBox Providers are used.
CFProperty	2	Great documentable approach to variable mixins to reduce getter/setter verbosity. Great for visualizing object dependencies. Safe for circular dependencies.	Mixin variables at runtime by using the cfproperty annotations. Cons is that you can not use the dependencies in an object's constructor method.
Setter/ Methods	3	Legacy or classic style	The inject annotation MUST exist on the setter method if the object is not mapped. Mapping must be done if you do not have access to the source or you do not want to touch the source.

Dependencies, Scope, Names?

Style	Pros	Cons
Annotations	• Documentable • Better visibility • Rapid Workflows • Just Metadata!	• Can pollute code • Some call intrusive • Unusable on compiled code
Configuration	• Compiled/Legacy Code • Multiple configurations per object • Visible Object Map	• Tedious • Slower workflow • Lower visibility

Injection Annotation

- Use one annotation: **inject**
  - Tells the Injector what to inject
  - Concatenated strings separated by ":"
  - First section is called DSL namespace
  - **inject="id:MyService", inject="coldbox:plugin:Logger"**

Namespace	Description
ID-Model-Empty	Mapped references by key
WireBox	WireBox related objects: parent injectors, binder, properties, scopes
CacheBox	CacheBox related objects: caches, cache keys, etc
Provider	Object Providers
LogBox	LogBox related objects: loggers, root loggers
ColdBox	ColdBox related objects: interceptors, entity services, etc
Custom	Your own live annotations

Annotations

```
// CF Property Injection
property name="service" inject;
property name="service" inject="id:CoolService";
property name="log" inject="logbox:logger:{this}";

// Constructor
function init(service inject){
    variables.service = arguments.service;
    return init;
}

// Setter
function setService(service) inject{
  variables.service = arguments.service;
}
```

CFC Annotations

- @**autowire** = boolean [true]
- @**alias** = list of know names for this CFC
- @**eagerInit** [false]
- @**threadSafe** [false]
- @**scope** = valid scope
- @**singleton**
- @**cachebox** = cache provider [default]
- @**cache** [default]
- @**cacheTimeout** = minutes
- @**cacheLastAccessTimeout** = minutes
- @**parent** = parent ID
- @**mixins** = A list of UDF templates to mixin

Event Model

- Announce events throughout injector and object life cycles
- Create simple CFC listeners or enhanced ColdBox interceptors
- Modify CFCs, metadata, etc
- Extend WireBox-CacheBox YOUR WAY!

```
component{

    configure(injector,properties){}

    afterInstanceCreation(interceptData){
      var target = arguments.interceptData.target;

      target.$formatdate = variables.formatDate;
    }

    function formatDate(){}
}
```

Source Code

- If you are a source junky and want to help out:
- https://github.com/ColdBox/coldbox-platform

Issues & Mailing List

- Bugs, enhancements, ideas:
- https://ortussolutions.atlassian.net/browse/WIREBOX
- http://groups.google.com/group/coldbox

Q & A

Thanks!