How do I DI?

Terms & Definitions

Code Smells

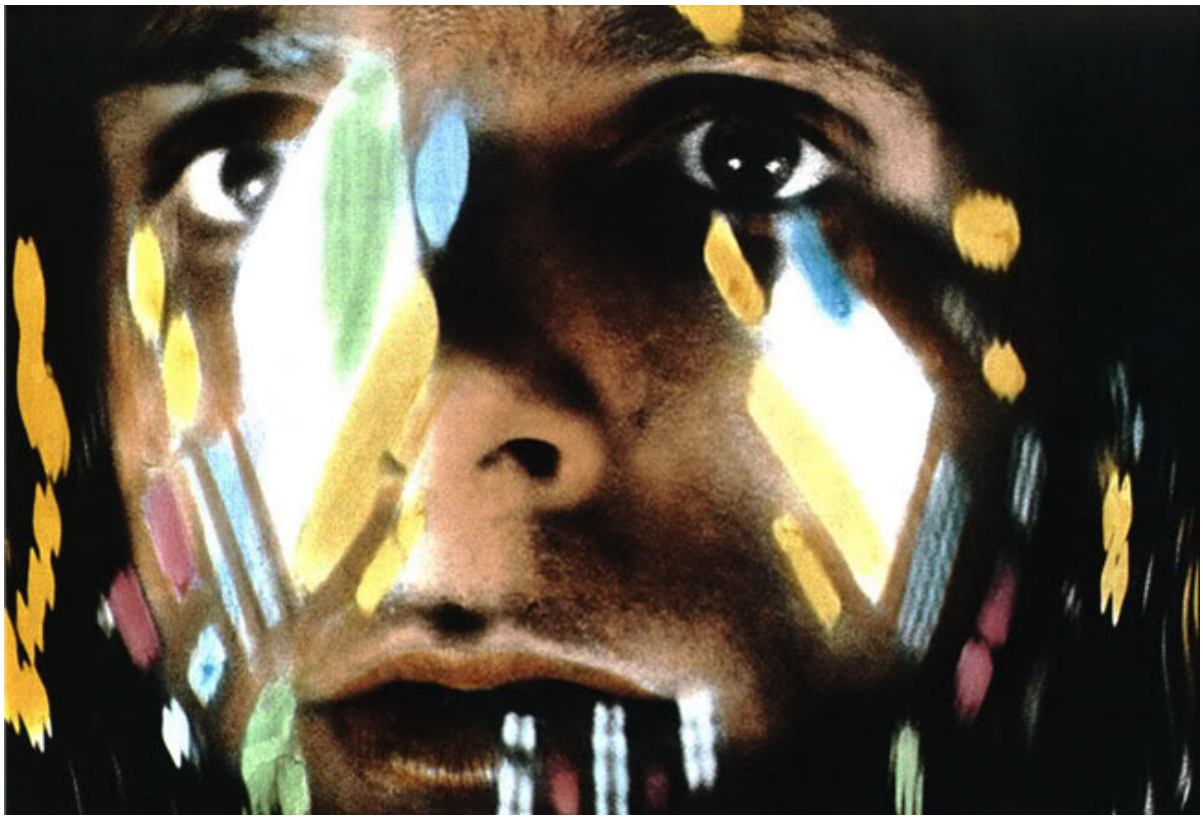
Patterns

Refactoring Examples

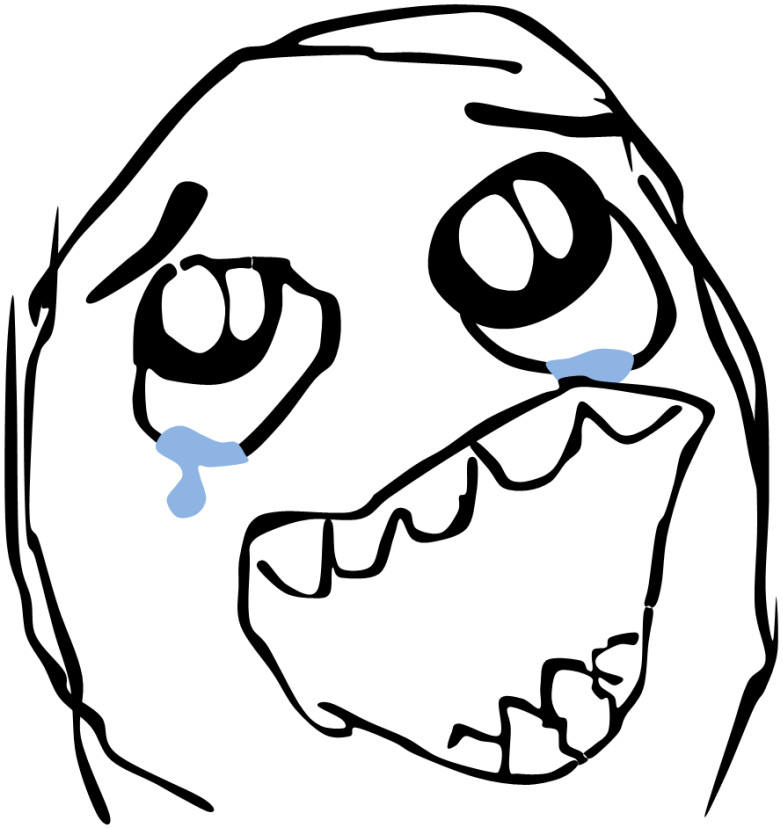
Why DI?

- Essential for building SOLID software
- Testable
- Maintainable
- Extendible
- Promotes discrete units (components)

My God. It's full of stars.



It's perfect!



What is DI?

- DI stands for [D]ependency [I]njection
- *Pass* dependencies rather than *create* them
- Implements IOC

Working without DI

```
// Constructor
internal StarWarsLegos()
{
  Pilot = new LukeSkywalker
  {
    Weapon = new LaserBlaster(),
    Helmet = new BrainBucket()
  },
  Droid = new ArtooDeetoo(),
  Engine = new QuantumInverter()
}
```

Dep

How to inject?

- Prefer constructor injection
  - Dependencies are obvious
  - Parameters should be non-null
  - Allows stricter class invariants
- Property injection is problematic*

*Discussion to follow.

Define constructors

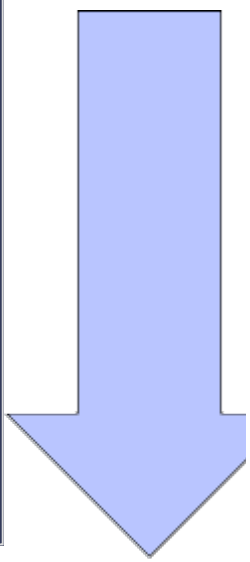
```
class StarWarsLegos
{
  StarWarsLegos(Pilot pilot, Droid droi
}

class Pilot
{
  Pilot(Weapon weapon, Helmet helmet) {
}
```

Construct with injection

```
var toy = new StarWarLegos(
  new LukeSkywalker(
    new LaserBlaster(),
    new BrainBucket()
  ),
  new ArtooDeetoo(),
  new QuantumInverter()
);
```

Dependen



What are components?

- Objects
- Callbacks
- Values
- Anything that encapsulates a concept

```
IToy CreateToy(
  IPartsFactory partsFa
  ToySettings settings,
  Func<Color> getColor,
  int targetAge
) { }
```

What is IOC?

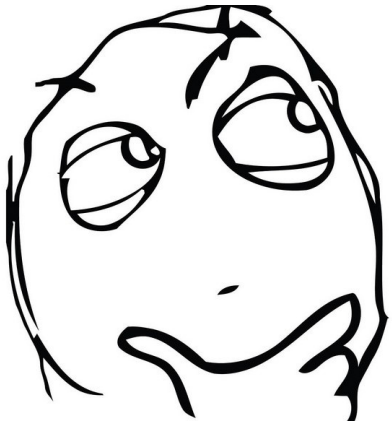
- IOC stands for [I]nversion [O]f [C]ontrol
  - The general concept implemented by DI
  - Equivalent to [D]ependency [I]nversion [P]rinciple
  - Transfers control from the *callee* to the *caller*
- Build a graph of components from *top down*
- A container *supports* IOC*

*It's optional, but you'll almost certainly want one...

```
// Bottom-up (callee makes decisions)
var pilot = new LukeSkywalker();
// weapon and helmet initialized in con

// Top-down (caller makes decisions)
var pilot = new LukeSkywalker(
  new LaserBlaster(),
  new BrainBucket()
);
```

This doesn't seem like rocket science



Sooo...we cool?

Good. Let's move on to containers...

Containers

What is a container?

- Implements registration and retrieval for IOC
- Applications create registrations
- The container is sealed
- Application retrieves objects

What does sealed mean?

- A container is sealed as soon as it provides an instance
- How else to guarantee singletons?
- **Register** via the service registration handler
- **Retrieve** via the service request handler

```
// Register
var container = CreateContainer()
  .RegisterSingle<IToy, StarWarsLegos>(

// Seal
var services = container.GetServices();

// Request
var toy = container.GetInstance<IToy>()

// Error!
container.RegisterSingle<IToy, BoardGam
```

Registration required?

- Yes, for interfaces
- Unregistered objects are transient
- Let the container construct objects for you
- Avoids dependency on constructor parameters

```
class A { }
class B { B(A a) { } }
class C : A { }

var s1 = CreateContainer().GetServices(

var b = s1.GetInstance<B>(); // uses A

var s2 = CreateContainer()
  .Register<A, C>()
  .GetServices();

var b = s2.GetInstance<B>(); // uses C
```

What is a Service Locator?

- Provides any instance
- Overuse is an anti-pattern*
- Provides the composition root

*Discussed later

What is a composition root?

- The root of your application's object graph
- It's (ideally) the *only* object that you should create*

*If you're using a container; otherwise, you'll call `new` a lot.

Overusing the service locator

```
// Get all objects...
var toy = services.GetInstance<IToy>();
var player = services.GetInstance<IPers
var game = services.GetInstance<IGame>(

// Use them
game.Start(player, toy);
```

What to do instead?

Think of a higher-level concept.

```
var services = CreateContainer()
  .RegisterSingle<IGame, Game>()
  .RegisterSingle<IToy, StarWarsLegos>(
  .RegisterSingle<IPerson, Child>()
  .GetServices();

class PlayDate
{
  PlayDate(IGame game, IPerson person,
}

// Get new play date (transient)
services.GetInstance<PlayDate>().Start(
```

What are object lifestyles?

- Determines when the container uses an object
- Singleton = forever
- Transient = once
- Scoped:
  - Per graph
  - Per web request
  - Per thread
  - Custom*

*See **SimpleInjector Object Lifetime Management** for more information.

Transients and singletons

- **Singleton**: one instance per container
- **Transient**: new instance per use
- lifestyle mismatch => transient injected to singleton*

*More precisely, an object can only inject objects with a scope less than or equal to its own.

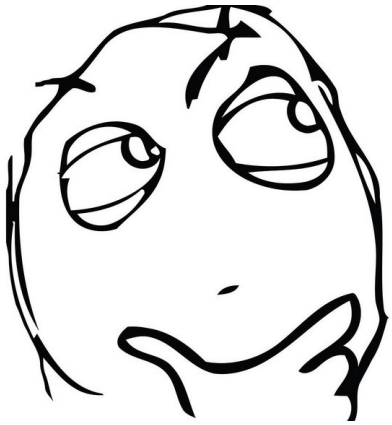
Lifestyle mismatch

```
class StarWarsLegos
{
  StarWarsLegos(IPerson owner) { }
}

var services = CreateContainer()
  .RegisterSingle<IToy, StarWarsLegos>(
  .Register<IPerson, Child>()
  .GetServices();

var toy = services.GetInstance<IToy>();
```

Containers make things easier...



Sooo...we *still* cool?

Good. Let's move on to anti-patterns...

Anti-patterns

Anti-pattern: Property Injection

- Risks opening your object to modification
- More complexity in the container (reflection?)
- Complicates **nullability** (can't be marked non-nullable)
- Complicates **immutability** (can't be read-only)

Nullable and mutable

Objects require discipline or help from the container to use safely.

```
class StarWarsLegos
{
  StarWarsLegos() { } // Pilot is null

  [NotNull]                    // Not en
  IPerson Pilot { get; set; } // Not im
}

var n = new StarWarsLegos().Pilot.Name;
var toy = services.GetInstance<IToy>();
var n = toy.Pilot.Name; // OK if proper
```

Non-null and immutable

Your objects cannot be mis-used.

```
class StarWarsLegos
{
  StarWarsLegos([NotNull] IPilot pilot)

  [NotNull]                  // Enforced
  IPerson Pilot { get; } // Immutable
}

var n = new StarWarsLegos().Pilot.Name;
var toy = services.GetInstance<IToy>();
var n = toy.Pilot.Name; // OK
```

Anti-pattern: Service Locator

- Provides access to *all* other objects
- Difficult to reason about dependencies
- Use only for plugins or factories (e.g. `IProvider` in Quino)
- Inject only the request handler (not the whole container)

Injected container

Dependencies are hidden from the caller

```
bool CanFly(Container container)
{
  var droid = container.GetInstance<IDr
  var pilot = container.GetInstance<IPi

  return droid.Online && pilot.Awake;
}
```

Global container

Dependencies are hidden from the caller

```
bool CanFly()
{
  var droid = ServiceLocator.GetInstanc
  var pilot = ServiceLocator.GetInstanc

  return droid.Online && pilot.Awake;
}
```

Code smell: new

- Not customizable
- Not mockable
- Never use `new` for a singleton
- Prefer factories to maintain flexibility
- Use `new` for small, mutable objects

Code smell: static methods

- Not customizable
- Not mockable
- Nested static calls compounds the problem
- Use extension/static methods for quick helpers
- Useful for configuration/fluent APIs

Code smell: virtual

- How to override?
- Do you have to replace the parent object, too?
- Use a component or factory instead
- Use composition over inheritance

Examples from Quino

- Split up `ConfigurationDataLoader`
  - `IConfigurationDataNodeProvider`
  - `IConfigurationDataFileProvider`
- Split up `FileLoggerBuilder`
  - `LogFileNameResolver`
  - `ApplicationFileLogNameResolver` (override)
  - Fixed code smell in `SerilogLoggerBuilder`

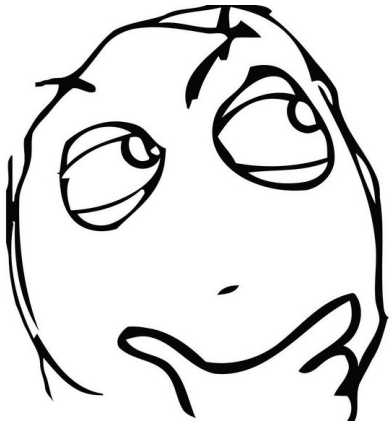
Code smell: Protected

- Why? Who's going to use it?
- What if the base class no longer needs it?
- Use private instead
- Let descendants take a reference in the constructor
- Protected fields are not CLS-compliant

Code smell: Public everything

- Use `internal` in end-products
- Use `sealed` in end-products
- Frameworks should be more careful with `sealed`/`internal`
  - Help avoid re-inventing the wheel
  - Copy/pasted code doesn't get tested
  - It also doesn't get bug fixes

Is my code really that smelly?



Sooo...we *still* cool?

Good. Let's move on to patterns...

Patterns & recommendations

Framework vs. Product

- Is there a difference in how you code?
- No. Not really.
- Does it apply to tests?
- Yes, those are code, too.

Prefer immutable, stateless singletons

- Does your component need state?
- All of it?
- Can you extract the parts that do?
- Can you use a factory?

Examples from Quino

- `IDataSession` is stateful
- Inject `IDataSessionFactory` instead

Configuration & Settings

Collect mutable properties in a separate object

```
interface IStarWarsLegosSettings
{
  int MaximumSpeed { get; set; }
  int ShieldStrength { get; set; }
}

class StarWarsLegos
{
  StarWarsLegos(IStarWarsLegosSettings
}
```

When should you use interfaces?

- Use interfaces for parameters (injection)
- Return values can usually be classes
- Unless the method is `virtual`
- Without an interface, you're *forcing* a base class
- Have a look at the **Design chapter** in the C# Handbook

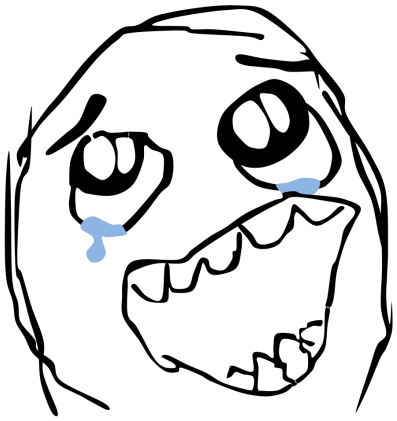
How to design an interface

- A single method is easy to customize
- Larger components are easier to inject
- Build larger components out of smaller ones
- See `ITextConsoleTools` in Quino for an example

Customizing Retrieval

- Use factories to create transient instances
- Use GetAllInstances() for a list of objects
- Use `IProvider<T>` in Quino
- Where necessary, *wrap* a transient in a singleton
- Use ad-hoc classes to inject (see `PlayDate`)

I can haz clean coding?



I hope this helped