

Dependency Injection

Traineeship

AXXES.

Who am I?

Lode Kennes

4 years @ Axxes

.NET Developer

Projects at SWF, OTN Systems, Syneton

Integration layer between Axxes apps

Axxes App

Self employed (secondary activity)

Ask me anything!

Who are you?

—

Focus on?

Familiarity with .NET landscape?

Experienced with IOC?

Practical

Just ask questions!

Part 1: 13.00 – 17.00

Part 2: 13.00-17.00

Break in between? Just ask 😊

Demo's and slides on Bitbucket

Demo's prepared on

- VS 2022
- Rider
- .NET 6.0
- .NET 4.8

SOLID

—

Dependency inversion principle

High level modules should not depend on low level modules

Modules should depend on abstractions

Agenda for today

Dependencies and coupling

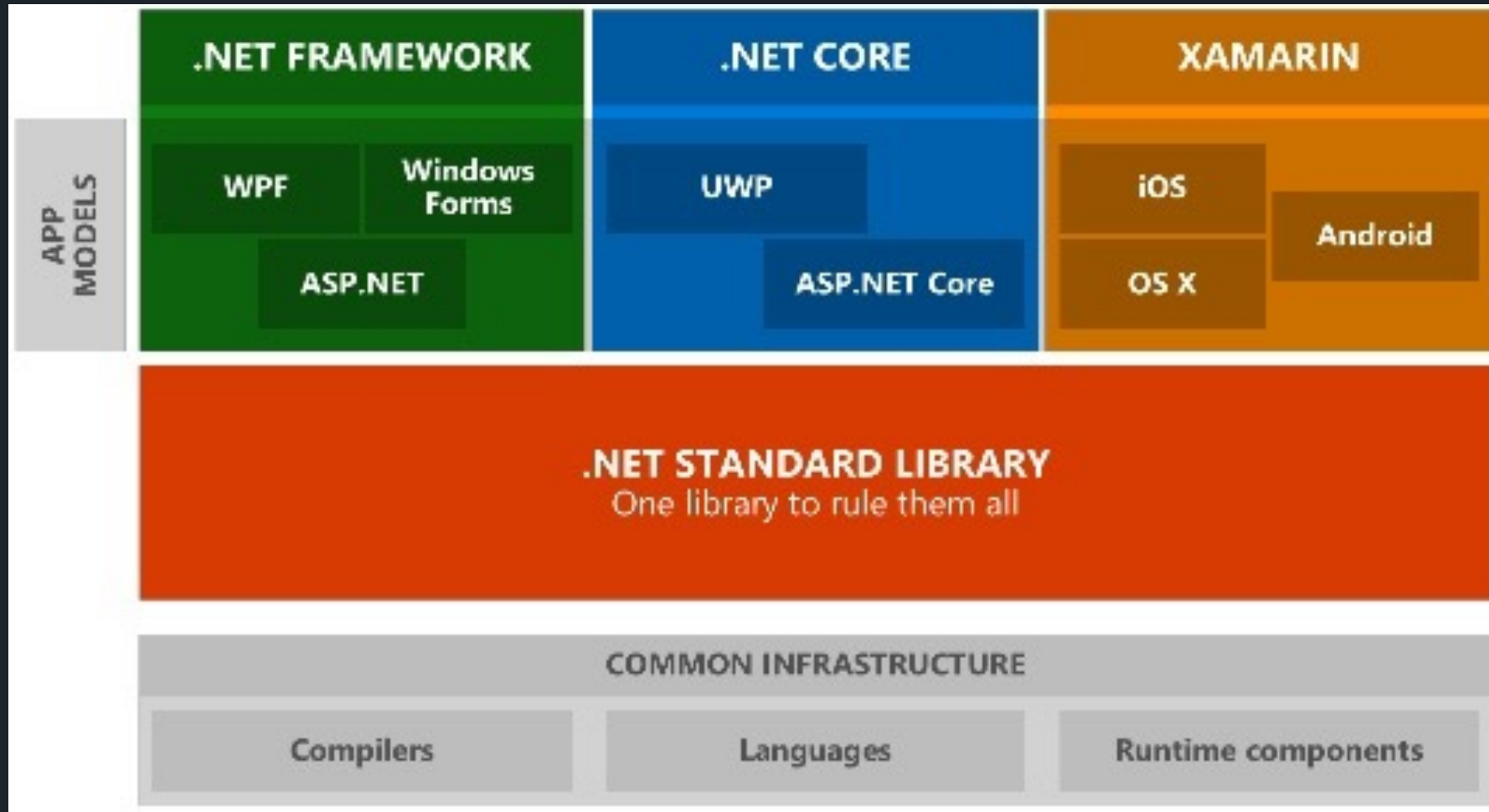
DI principles + IOC containers

Service locators

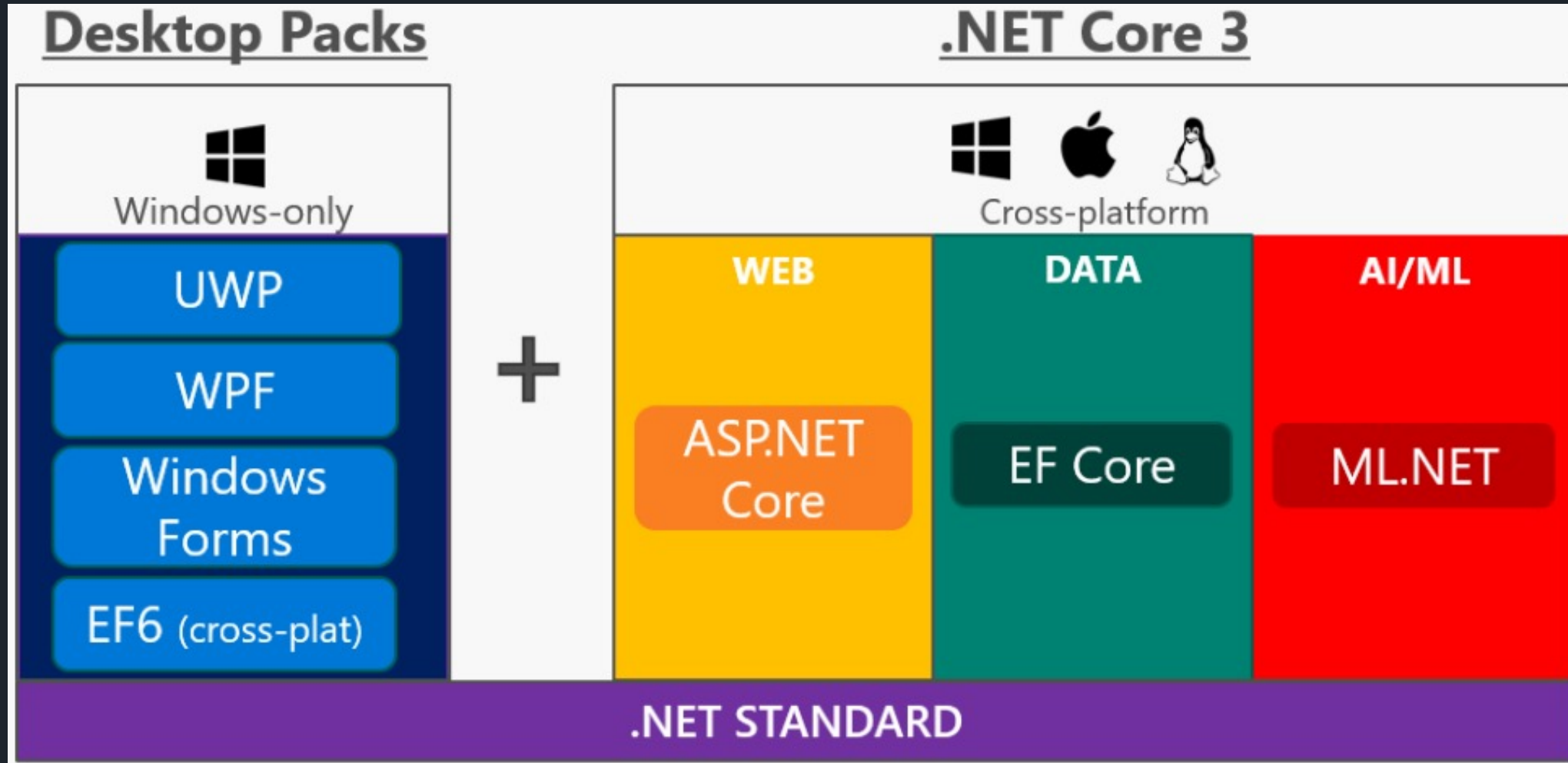
Other techniques for object creation

Architectural styles built around dependencies

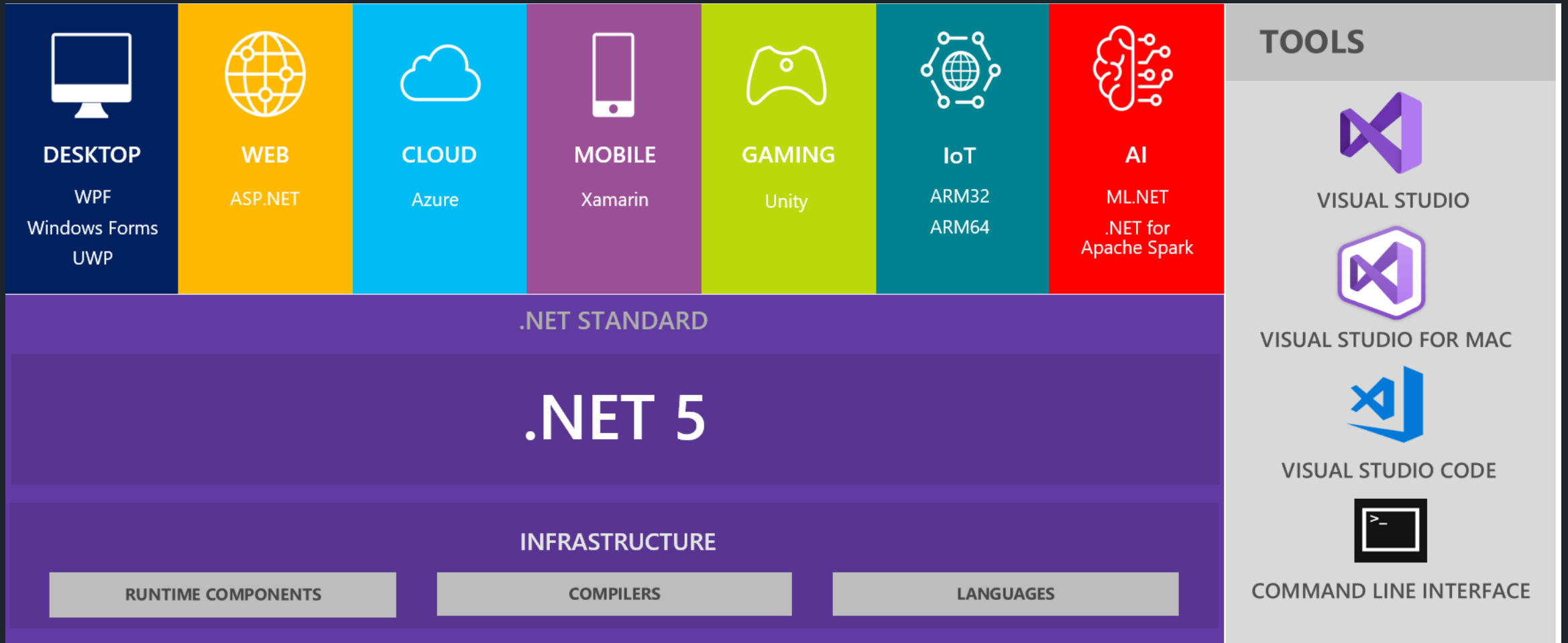
.NET App models



.NET App models



.NET App models



.NET App models



Dependency injection in ASP.NET Core

📅 07/02/2018 • ⌚ 17 minutes to read • Contributors 🦖 👤 👤 ⚙️ 👤 all

By [Steve Smith](#), [Scott Addie](#), and [Luke Latham](#)

ASP.NET Core supports the dependency injection (DI) software design pattern, which is a technique for achieving [Inversion of Control \(IoC\)](#) between classes and their dependencies.

Inversion of control isn't hard

Why coupling is bad?

What does it mean?

Class / components depending on other classes and components

Limits functionality to a single known implementation

➔ You can't swap the implementation without breaking the code

Code is not testable

During a unit test you want to test a piece of the component not the whole flow -> Integration

Agile practices are hard to accomplish

Working with multiple team members on the same feature is hard

It's bad for your reputation as a developer 😊

Embrace abstraction

The secret to writing decoupled/testable software:

- Write against abstractions instead of concrete implementations

- Stop newing up classes

- Note: not talking about models or dto's

- New is glue

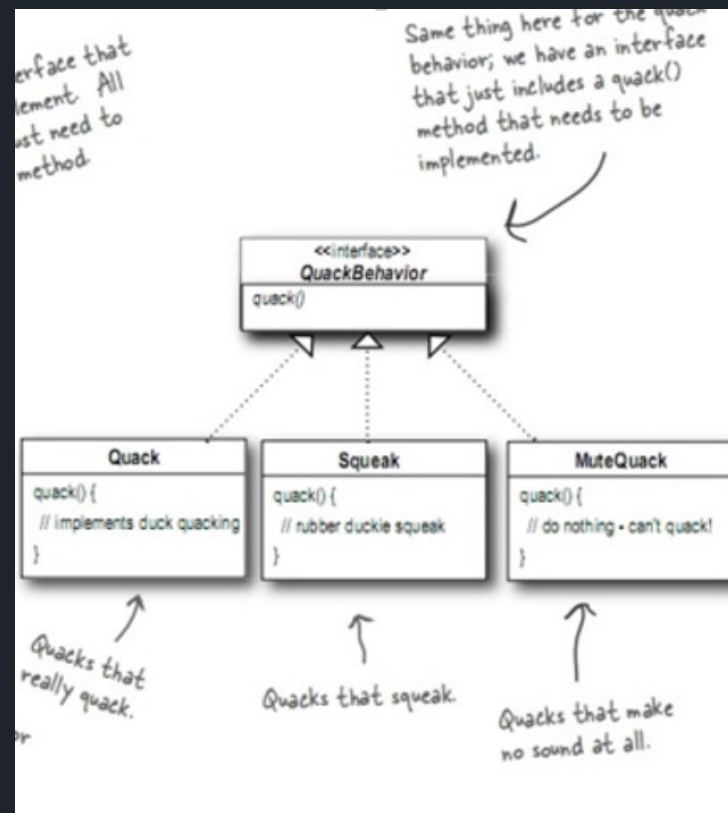
Embrace abstraction

Define dependencies as interfaces

Remember the definition of an interface:

A public contract where the consumer talks to the interface and does not know the actual implementation

Remember this



Embrace abstraction

This is how most educational programs used to learn students about interfaces

Only from a polymorphic point of view

Making classes implement interfaces is there a cost?

In the old days of OO not every class was an interface because working with an interface was slower

Embrace abstraction



Demo DI.Essentials.Coupled, DI.Essentials.Abstraction,
DI.Essentials.UnitTests

Object creation (IOC)

Invert the creation of objects

“Creation Inversion”

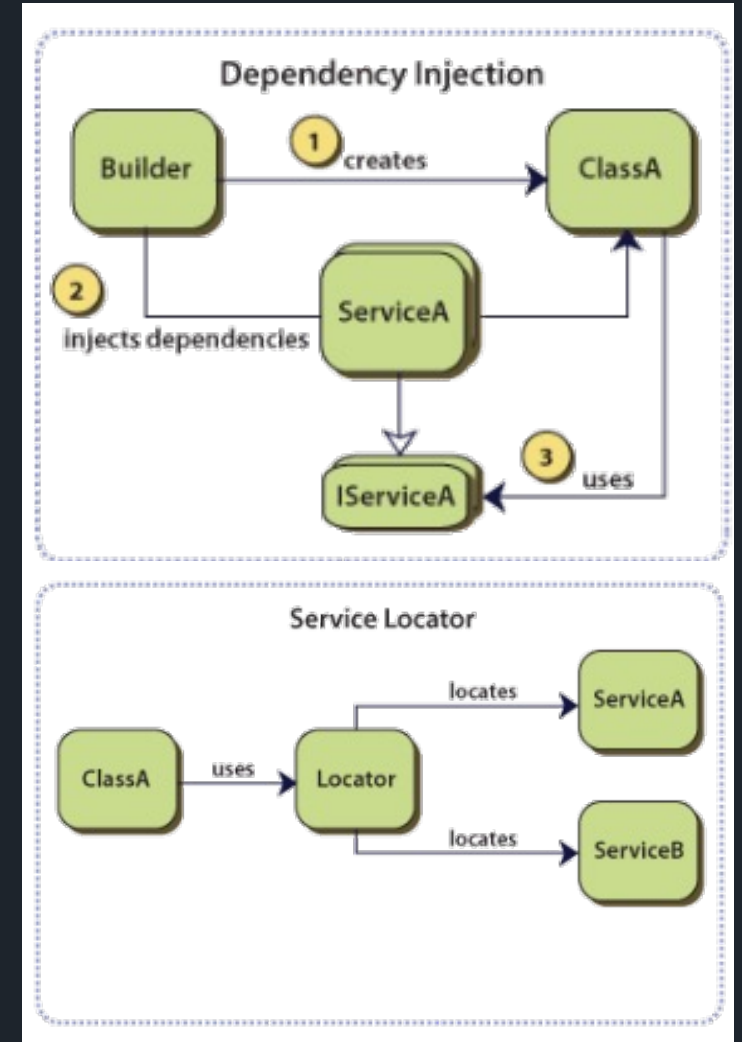
Create objects outside of the class they are being used

Factory pattern

Service locator

Dependency injection

2 flavors: pull or push mechanisms



DI container

A repository for definitions typically relating an abstraction to a concrete class

Think of it as a bucket

Core functionality

Provide facility to register classes related to an interface

Provide facility for resolving a request from a given interface

So it's about registration and resolving

Push mechanism

DI container

Registrations or type associations can be achieved in different ways depending on the container you are using

Procedural (fluent interface)

Unity, Ninject, Castle Windsor, StructureMap, Autofac, Microsoft DI

Configuration

Spring.net, Autofac

Declarative (with attributes)

MEF

Registrations only needs to be done once. Mostly done in some kind of startup file.

DI container

Resolving process

- Recursively resolving dependencies

- Requesting one class from the container starts a chain reaction

Different mechanisms

- Constructor injection (most common)

- Properties injection

- Interface injection (never used this)

- Method injection (most frameworks don't support this)

DI container

Resolving process

- Recursively resolving dependencies

- Requesting one class from the container starts a chain reaction

Different mechanisms

- Constructor injection (most common)

- Properties injection

- Interface injection (never used this)

- Method injection (most frameworks don't support this)

Demo PoorMansContainer

Ninject

Open source ioc container

First version in 2007

Simple

Extensible

Good community -> a lot of nuget packages

Castle Windsor

Open source IoC container

Part of the Castle project

Set of tools for .NET and Xaml

One of the first IoC containers for .NET

Creator also worked on MEF as a program manager at Microsoft

Autofac

Open-source IoC container originally developed on Google Code

Was the first to also windows phone and windows store apps

Works with a Builder constructed with lambda's. The container is created by the builder.

Gaining a lot of popularity over the last years

Allows injection via xml configuration -> change wiring without recompiling

Microsoft Dependency Injection

Built-in

Fluent API

Really fast

Default in ASP .NET Core Applications

Demo DI.Containers.*

Which container?

Size and licensing?

Framework	License	Minimum required <u>dlls</u>	Size(kB)
Castle	Apache 2	2	192
Unity	MS PL	2	120
<u>Ninject</u>	Apache 2	1	147
<u>Autofac</u>	MIT	1	110
<u>Structuremap</u>	Apache 2	1	164

Which container?

Registration/configuration flavor

Framework	Fluent	Automatic registration	Xml usage
Castle	yes	yes	Supported but not required
Unity	yes	yes	Supported not required
<u>Ninject</u>	yes	yes	Possible via <u>Ninject.Extensions</u> package
<u>Autofac</u>	yes	yes	Supported not required
<u>Structuremap</u>	yes	yes	Supported not required

Benchmarks

Container	Singleton	Transient	Combined	Complex
No	41	49	69	99
	49	59	76	103
Autofac 4.9.4	593	754	1953	5877
	389	504	1191	3609
Microsoft Extensions DependencyInjection 2.2.0	81	124	148	197
	69	107	143	159
Ninject 3.3.4	3473	8686	23529	63579*
	2563	6969	17635	49285
StructureMap 4.7.1	1121	1281	3410	8312
	717	856	2166	6052
Unity 5.8.6	302	437	1282	3962
	281	405	1095	3553
Windsor 5.0.0	437	1821	6402	20536
	350	1108	3712	11821

Which container?

Benchmarks

<https://github.com/danielpalme/locPerformance>

Community

Active development

Documentation

Extensions

Reputation of contributors

Instance lifetime

Transient

- Resolved instance kept until parent goes away

- Default in most containers

Singleton

- Shared instance

- Resolve request returns the same instance until container goes away

- Be careful with threads

Scoped

- Every http request

- DbContext of EF you don't want to share this with other end users

Instance lifetime

How are objects coming from a container disposed?

In a regular program the instance is garbage collected when out of scope

In general the rule is only call Dispose() on object you are the owner

Objects you newed

You can not assume you are the only consumer of the object

IoC containers don't know when dispose was called on an instance

When the container is disposed the instances get disposed

Or you can tell the container when to dispose by defining a scope if it is supported by the container

Instance lifetime

Setting up registrations in big applications is a lot of work

Sometimes you want to work based on conventions

More dynamic

Accomplished via assembly scanning

Modules

When your application grows the bootstrap code for DI also grows

How to organize these registrations?

A module is a small class that can be used to bundle up a set of related components behind a 'facade' to simplify configuration and deployment.

Modules do not use dependency injection themselves

Most containers use modules for

- Packing similar services together

- Optional application futures (Plugin style)

- Environment specific registrations

<http://autofacn.readthedocs.io/en/latest/configuration/modules.html>

Multiple constructors

What with multiple constructors?

Different strategies depending on the container

The first constructor they find in code

The constructor with a metadata attribute

What I don't like here is scattering the DI containers code around in your code

The constructor with the most parameters -> greedy

Post-construction resolve

Kicks off resolve process AFTER the class has been instantiated

Useful when your class is created at a certain moment in time by something else

Remember! Only property injection is possible here, no construction injection

how did they build stack overflow before they had stack overflow?



ASP .NET Core

Out of the box support for DI

Custom basic container implementation

It's possible to plug an existing DI container into the asp net core framework

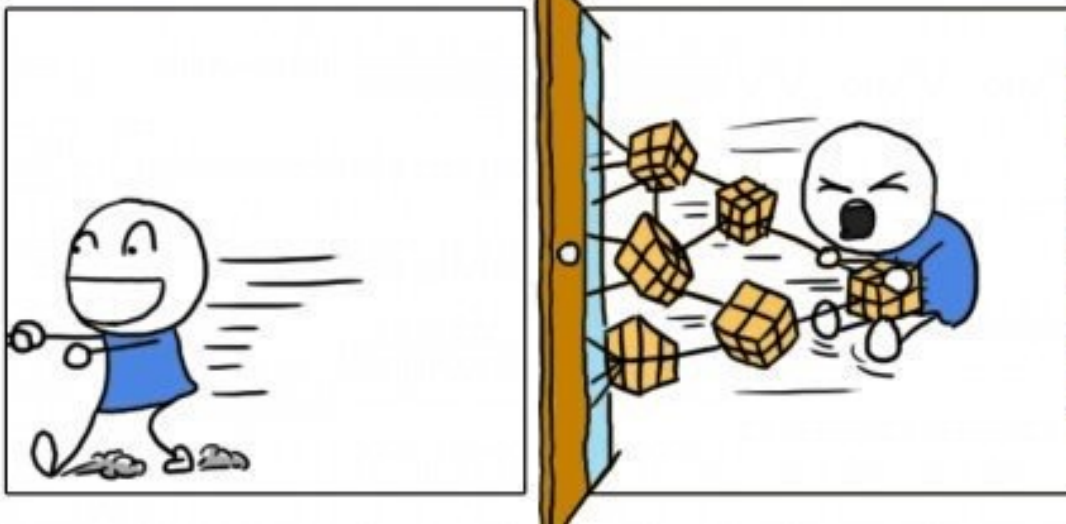
Asp net core is open source so all code for DI can be seen here

<https://github.com/dotnet/extensions>

<https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection>

Demo ASP Net Core DI

NPM DELIVERY



Service locator

Pull mechanism

Used when a class needs a lot of dependencies (stops constructor parameter explosion)

Can be used instead of post construction resolve

Returns the correct service based on a key

Wraps a registry which can be a dictionary, an IOC container or something else

In case DI container is wrapped -> on-demand instances

Biggest disadvantages

Only one registry -> pay attention with concurrency

More vulnerable for runtime errors

Service locator anti-pattern

According to some people service location is an anti-pattern

Originates from Mark Seemann's book and blogpost

<http://blog.ploeh.dk/2010/02/03/ServiceLocatorisanAnti-Pattern/>

In short the problem is you may not see the dependencies of a class with a service locator like you see them with a DI container

Causing runtime errors instead of compile time errors

Maintenance hell

In my opinion it depends how you use the locator

<https://stackoverflow.com/questions/22795459/is-servicelocator-an-anti-pattern/22795888#22795888>

Demo service locator

Factory pattern

Design pattern

Uses a pull mechanism

Mostly done with 2 abstractions

- on the factory itself

- on the instance the factory creates

Advantage client code must not change

Disadvantage when we need a new concrete implementation the factory code will need to change

Demo factory pattern

—

1. Create FastFoodFactory & OrganicFoodFactory
2. Make Test work

Conclusion

The core principle of testable code is usage of interfaces to build decoupled components

Containers offer different features

- For most part, all accomplish the same thing

- Choose the one that you like

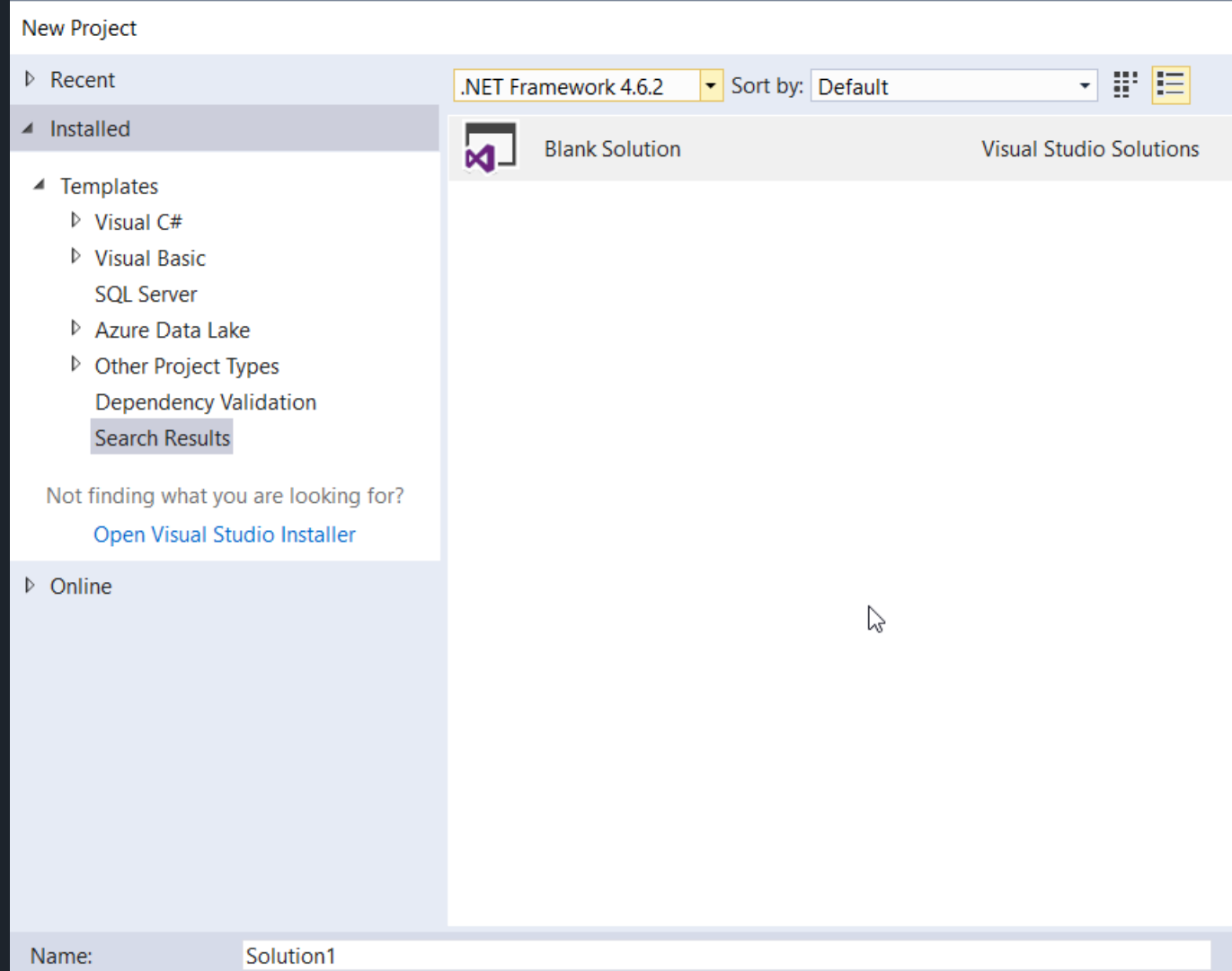
Inversion of control != DI container

A DI container does NOT facilitate testing

- The DI concepts do

Service locator is an anti-pattern -> it depends

The problem





Nice app



Because

Bad abstractions

Wrong cohesion

Which things belong together

2 much developer freedom

Coupling to frameworks and libraries

Utility projects

Frameworks scattered all over the application

Frameworks only help with encapsulation

No guidance

How do you start a new project?

A good solution

Classes that change together should be packaged together

Deployment deliverables

Separation of concerns

Applies to classes and components

Establish boundaries to separate behaviours and responsibilities in a system

Remember a class should only be responsible for only one thing, the same is true for components

Companies try to solve this in different ways

A written guide

- Publisher is a developer with lot's of experience

Several custom project templates

- Still too much responsibility for the developers

Code reviews

Guidance automation

- Best approach

- Problems to get it to work (installation, errors with recipe)

- Developers can still decide to put logic in the UI

Architectural patterns

Not design patterns

Help to structure your solution and software

Keep track of dependencies

Ways of laying out your software/solution

Help you to make decisions about coupling and cohesion

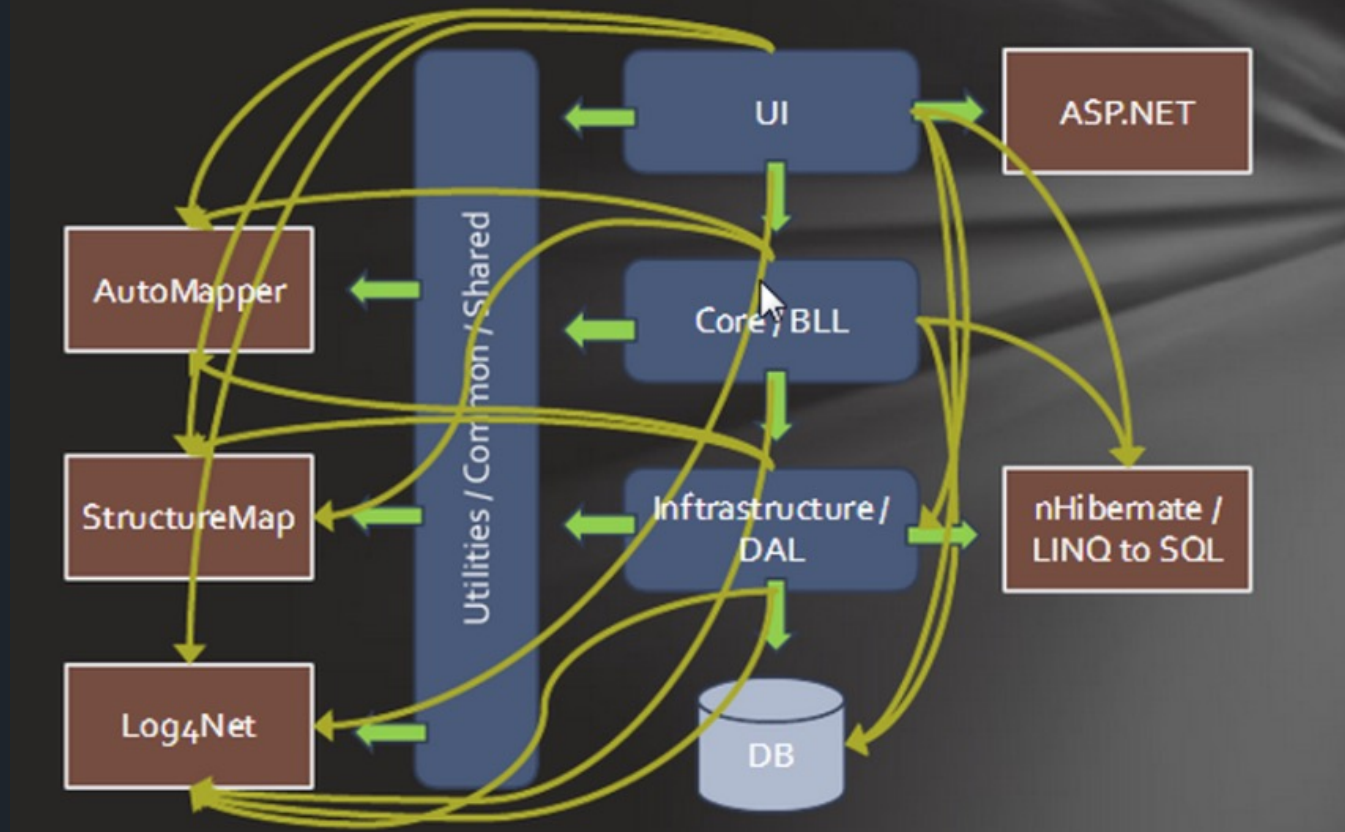
- How are things linked to each other?

- Which things belong together?

3-tier architecture

First answer on cohesion and coupling: 3 layered architecture

Traditional 3-layer Architecture



3 layered architecture

The concept of layers isn't something new

Think about the OSI model in networking

Every layer is depends on the layer beneath it

Every layer delegates subtasks to the layer underneath it

At least 3 layers but there is no guidance on the number of layers that should be used

In real life, you can usually count the number of layers within an application by knowing the number of tech leads previously involved on the project

Baklava architecture

3 layered architecture

The business layer depends on the data layer

Easy for developers to put logic in UI

Easy to pass business logic on purpose or accidentally and perform data access in UI

Devs struggle with finding correct place for code

Product is built on top of a technology which can change over time(asmx → WCF)

Logic is scattered, hard to locate code

Library explosions: easy to add references (log4net, automapper, IOC containers)

Onion architecture

Onion architecture

First defined in four blogposts by Jeffrey Palermo

<http://jeffreypalermo.com/blog/the-onion-architecture-part-1/>

<http://jeffreypalermo.com/blog/the-onion-architecture-part-2/>

<http://jeffreypalermo.com/blog/the-onion-architecture-part-3/>

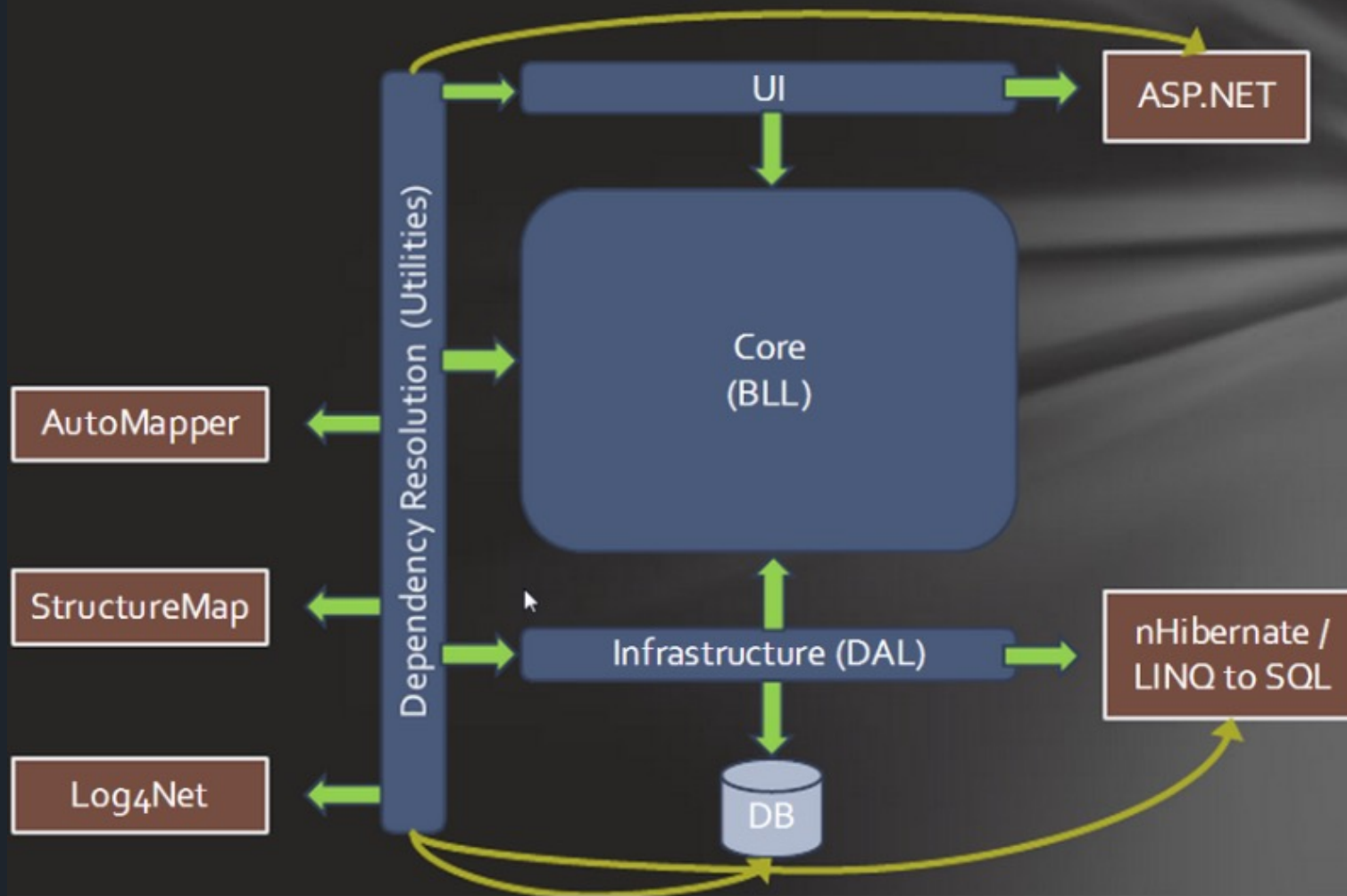
<http://jeffreypalermo.com/blog/onion-architecture-part-4-after-four-years/>

Very similar to hexagonal architecture

Architectural pattern where the core object model is represented in a way that does not accept dependencies on less stable code

Switch the direction of dependencies

Onion Architecture with Relative Sizes



Onion architecture

The application is built around an independent object model

Inner layers define interfaces. Outer layers implement interfaces

Direction of coupling is toward the center

All application core code can be compiled and run separately from infrastructure. (TDD)

Onion architecture

CORE

Everything unique to the business: domain models, validation rules, workflows

Defines all technical implementations as interfaces

No references to external libraries

No technology specific code: WCF, MVC, EF, gRPC, ...

Onion architecture

INFRASTRUCTURE

Provide implementations to CORE interfaces

Call web services, access database, logging

Can reference external libraries

Only technology specific code belongs here

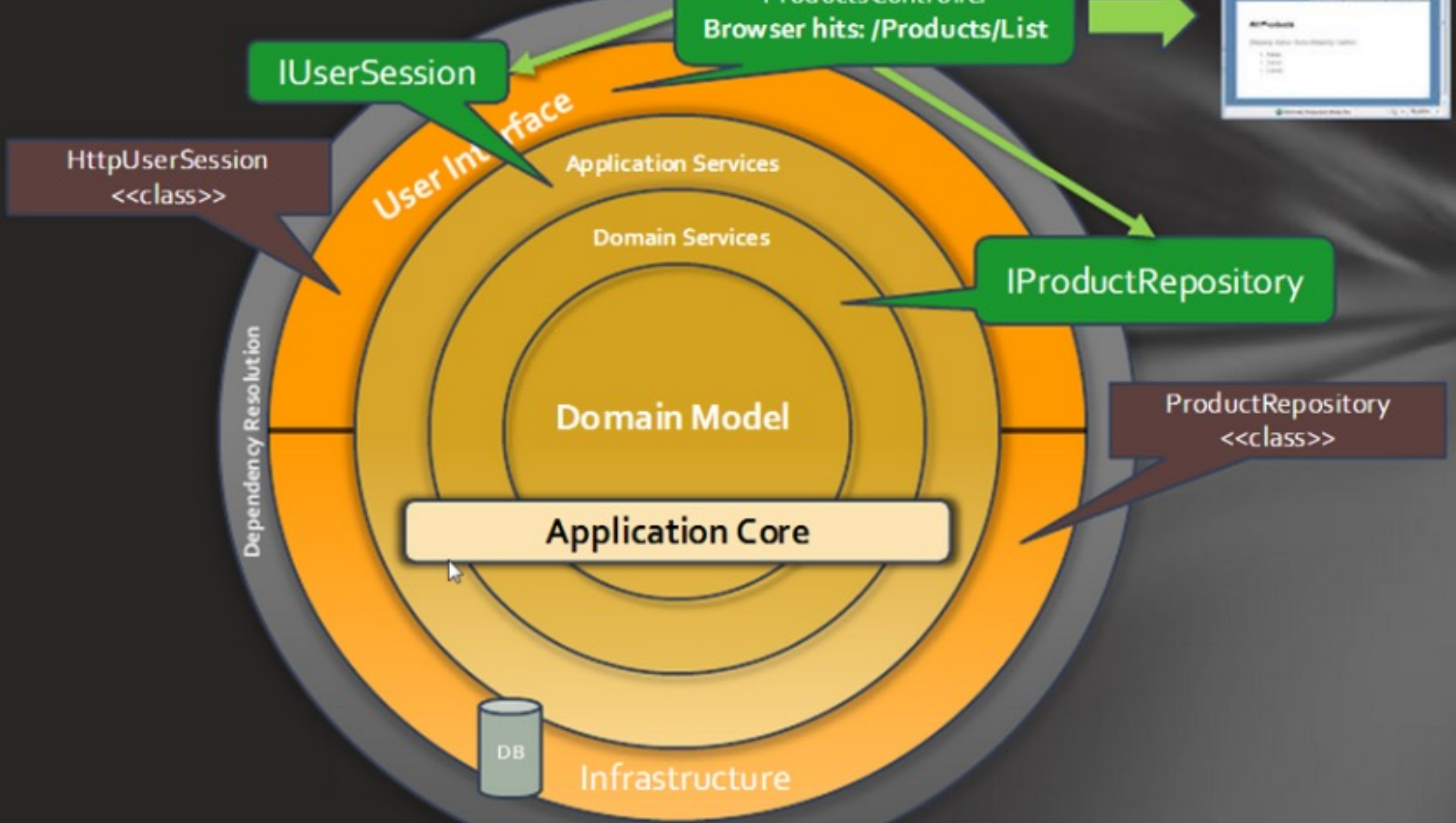
Onion architecture

DEPENDENCY RESOLUTION

Thin layer, contains no logic

Wires Core interfaces to Infrastructure implementations

Runs startup/configuration logic (bootstrapper)



Demo poetry reader

Onion architecture

True loose coupling

No need for shared utilities projects

Compiler enforces dependencies boundaries, impossible for core to add reference to infrastructure

UI and data access become smaller, only technology related stuff in there

Devs know where to put code (easy model)

Software is portable not dependent on technology

Excercises

Conclusion

Think about the structure of your solution, this impacts your deployments and maintenance

Don't use too many layers

Don't couple to a delivery mechanism like the web or an implementation detail like a database

Protect your domain, this is the important stuff

Make the UI, database and other infrastructure a plugin to your application

Learning journey

<https://www.youtube.com/user/IAmTimCorey/videos> (IAmTimCore)

<https://www.pluralsight.com/courses/inversion-of-control> (PluralSight)

<https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection>

<https://channel9.msdn.com/Shows/Visual-Studio-Toolbox/Dependency-Injection> (Channel 9)

https://sourcemaking.com/design_patterns/creational_patterns

<https://www.youtube.com/watch?v=th4AgBcrEHA> (Alistair Cockburn about hexagonal architecture)

<https://dzone.com/articles/hexagonal-architecture-is-powerful> (Dzone)

<https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/> (Jeffrey Palermo)

<https://youtu.be/IAcxetnsiCQ> (Ian Cooper about Onion and Ports and adapters in .NET core)