

Writing decoupled and testable code



Who am I?

- Lode Kennes
- 1 year @Axxes
- .NET Developer
- Projects at SWF, OTN Systems
- GreenR
- Self employed (secondary activity)
- Ask me anything!

Who are you?

Focus on?

Familiar with .NET landscape?

Experienced with IOC?

IT IS ABOUT PEOPLE



Practical

- Questions just ask them
- Part 1: 8.30-12.30
- Part 2: 13:30-17.00
- Break in between just ask for it
- Demo's and slides can be found on Gitlab
- Demo's are prepared on
 - Visual Studio 2019
 - Dotnet core 2.2
 - .NET 4.6

SOLID

Dependency inversion principle

High level modules should not depend on low level modules

Modules should depend on abstractions

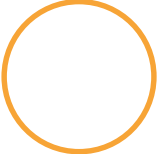
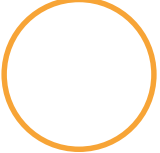
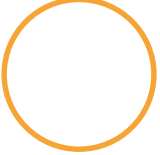
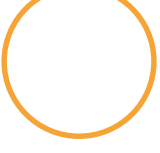
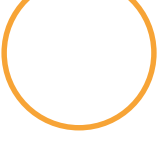
IT IS ABOUT PEOPLE



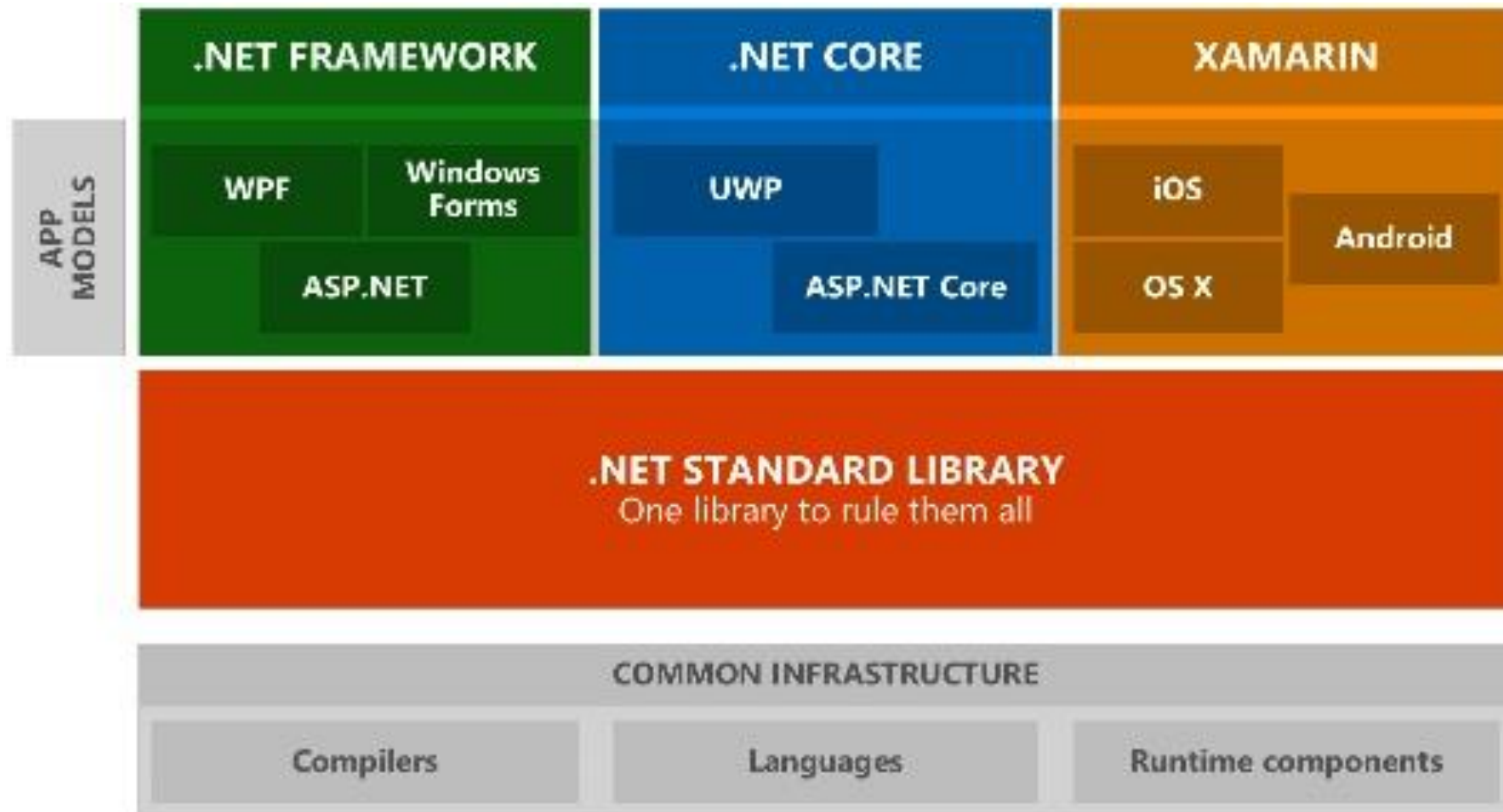
IT IS ABOUT PEOPLE

Agenda for today

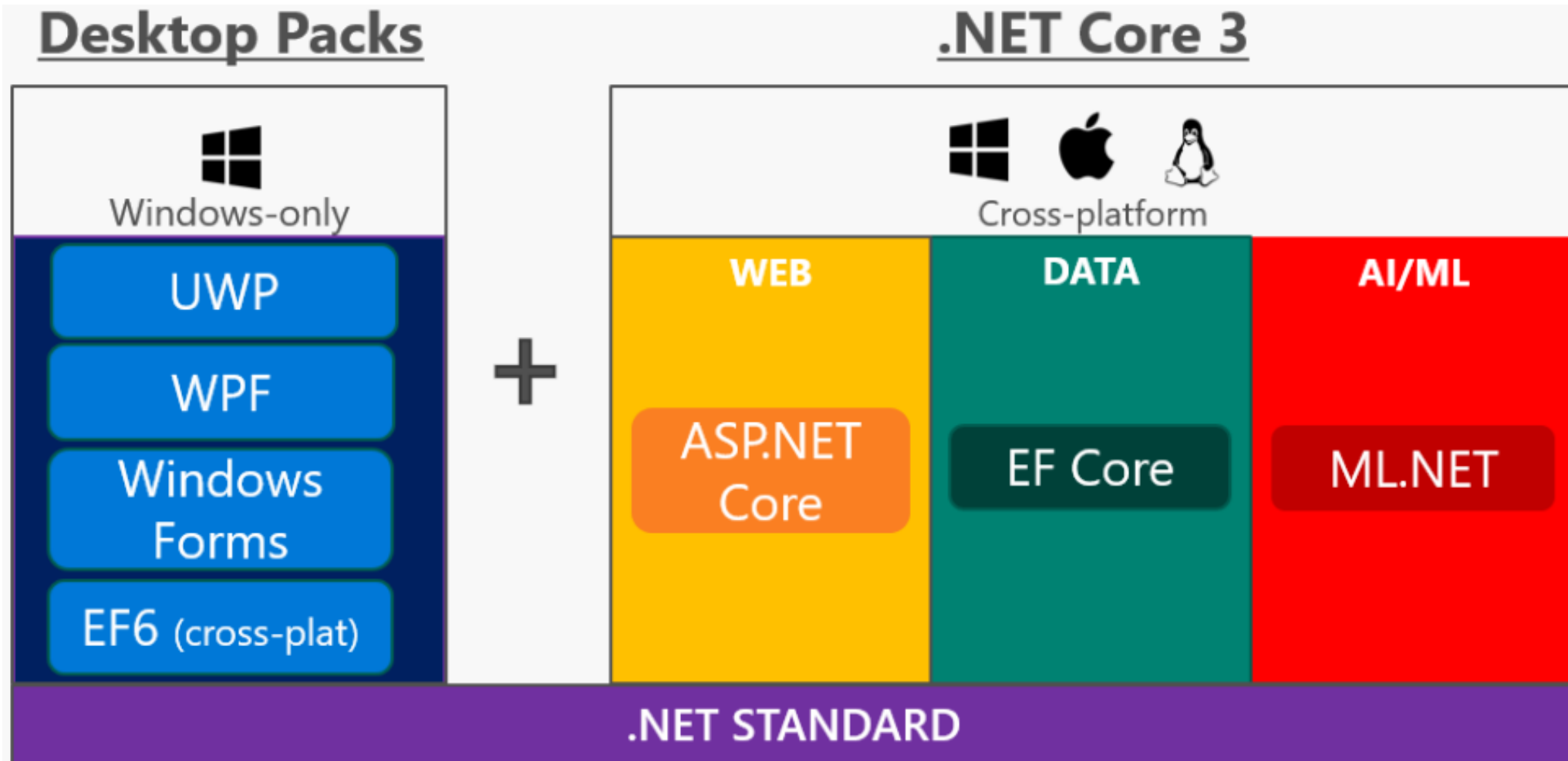


-  Dependencies and coupling
-  DI principles + ioc containers
-  Service locators
-  Other techniques for object creation
-  Architectural styles built around dependencies

.NET APP Models



.NET APP Models



.NET APP Models



Asp net core

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.

Martin Fowler's article on IOC

Inversion of Control Containers and the Dependency Injection pattern

In the Java community there's been a rush of lightweight containers that help to assemble components from different projects into a cohesive application. Underlying these containers is a common pattern to how they perform the wiring, a concept they refer under the very generic name of "Inversion of Control". In this article I dig into how this pattern works, under the more specific name of "Dependency Injection", and contrast it with the Service Locator alternative. The choice between them is less important than the principle of separating configuration from use.

23 January 2004



Martin Fowler

Translations: Chinese · Portuguese · French · Italian · Romanian · Spanish · Polish · Vietnamese

Find **similar articles** to this by looking at these tags: popular · design · object collaboration design · application architecture

Contents

- Components and Services
- A Naive Example
- Inversion of Control
- Forms of Dependency Injection
 - Constructor Injection with PicoContainer
 - Setter Injection with Spring
 - Interface Injection
- Using a Service Locator
 - Using a Segregated Interface for the Locator
 - A Dynamic Service Locator
 - Using both a locator and Injection with Avalon
- Deciding which option to use
 - Service Locator vs Dependency Injection
 - Constructor versus Setter Injection
 - Code or configuration files
 - Separating Configuration from Use
- Some further issues
- Concluding Thoughts

- <https://www.martinfowler.com/articles/injection.html>

Inversion of control can't be hard

IT IS ABOUT PEOPLE



Why is coupling bad

- What does it mean?
 - Class/components depending on other classes and components
- Limits functionality to a single known implementation
 - You cannot 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 member on the same feature is hard
- It's bad for your reputation as a developer 😊

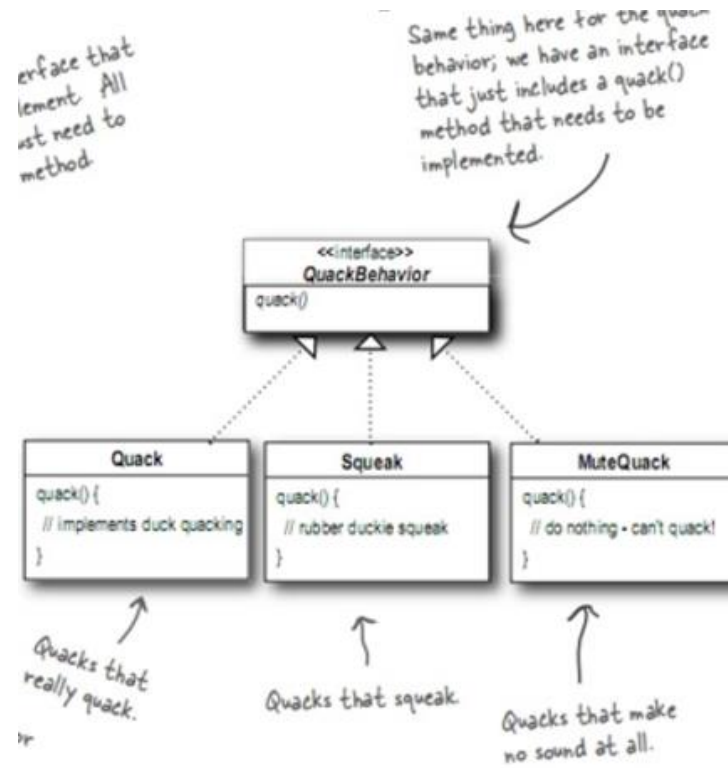
Embrace abstractions

- 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
- Previous years this was shocking.
 - How do you want to create instances?
 - Well there are other ways of doing this

Embrace abstractions

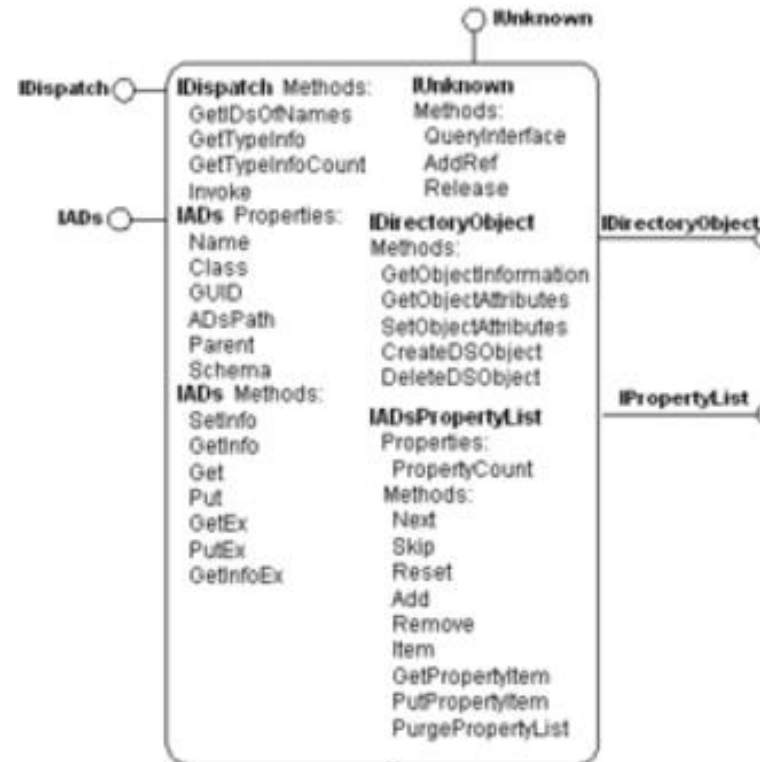
- Define dependencies as interfaces
- Remember the definition of an interface:
 - A public contract where the consumer talks to the interface and does not know the actual implementation

Remember this



Embrace abstractions

- 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 abstractions

- Component Object Model
- 1993
- Microsoft
- Wikipedia explanation:
 - De werking van COM-objecten is vergelijkbaar met die van objecten in een object-georiënteerde programmeertaal. Een COM-object bevat een interface, waarin gedefinieerd is welke hoofdfuncties het COM-object heeft. Het gebruiken van het COM-object door andere objecten verloopt dan ook via de functies in de COM-interface. De interne werking van het COM-object blijft verborgen voor gebruikers. Omdat de werking en het gebruik van COM-objecten zoveel lijkt op de manier van programmeren in een OO-taal, zijn COM-objecten ook goed te gebruiken in een dergelijke omgeving.

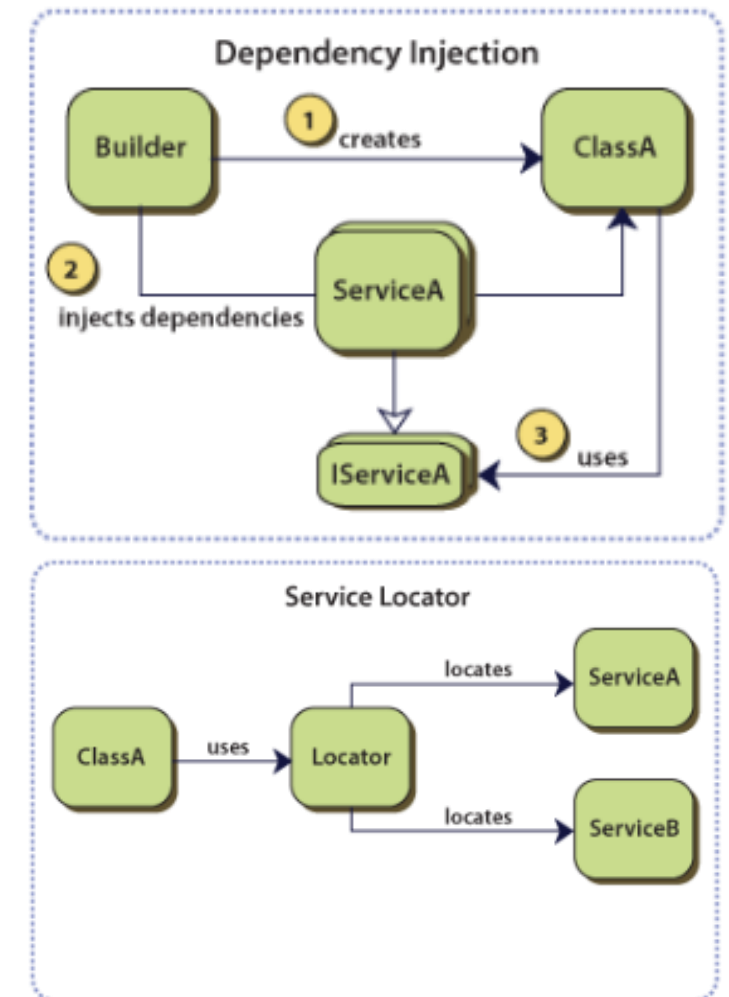
Demo DI.Coupled, DI.Abstraction, DI.UnitTests

IT IS ABOUT PEOPLE



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
- Configuration
 - Spring.net, Autofac
- Declarative (with attributes)
 - MEF
- Registrations only needs to be done once. Mostly done in some kind of bootstrapper.

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)

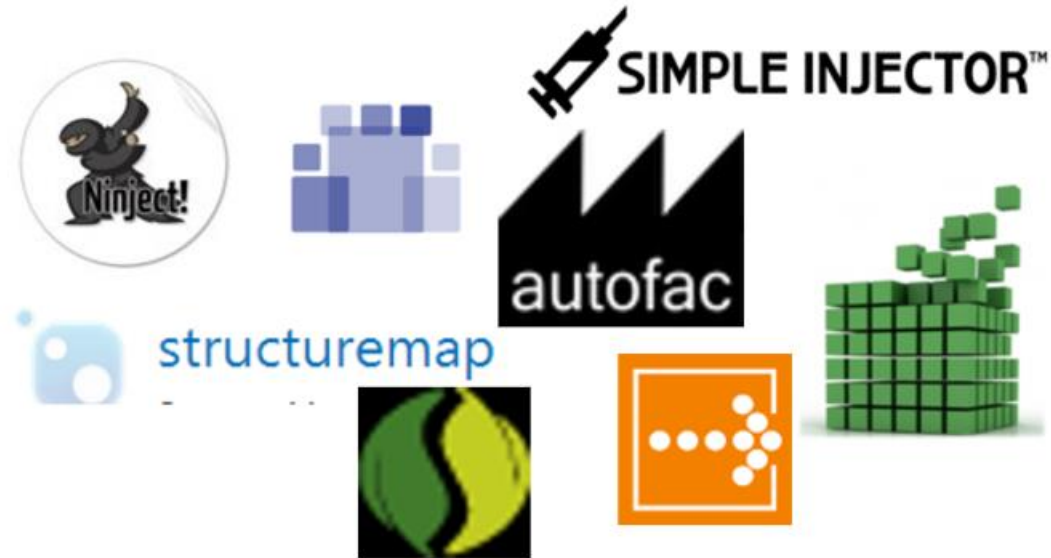
Demo DI.PoorMansContainer

IT IS ABOUT PEOPLE



DI container

- Different flavor's



Ninject

- Open source ioc container
- First version in 2007
- Simple
- Extensible
- Good community -> a lot of nuget packages

Unity

- IoC container from Microsoft
- Delivered by the patterns and practices team
- Part of the Enterprise Library
- Can be downloaded together with the Enterprise Library or as a Nuget package
- Is extensible

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 support 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.*

IT IS ABOUT PEOPLE



Which container

Size and licensing

Framework	License	Minimum required dlls	Size(kB)
Castle	Apache 2	2	192
Unity	MS PL	2	120
Ninject	Apache 2	1	147
Autofac	MIT	1	110
Structuremap	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
Ninject	yes	yes	Possible via Ninject.Extensions package
Autofac	yes	yes	Supported not required
Structuremap	yes	yes	Supported not required

Benchmarks

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

Demo instance lifetime

IT IS ABOUT PEOPLE



Autowiring dependencies

- 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

Demo Autowiring

IT IS ABOUT PEOPLE



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>

Demo modules

IT IS ABOUT PEOPLE



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

Demo constructor finder

IT IS ABOUT PEOPLE



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
 - Wcf servicehost or Service Fabric
- Remember this will have to use property injection since the resolve process kicks off after the class is instantiated, constructor injection is not possible

Demo post-construction resolve

IT IS ABOUT PEOPLE



Demo with multiple implementations

IT IS ABOUT PEOPLE



Platform specific

IT IS ABOUT PEOPLE



W P F

- Used to resolve ViewModel classes
 - Used to resolve dependencies injected into ViewModels
 - Used to resolve nested ViewModels
- ViewModels can be tested and test dependency implementations used
- Registrations in the App class
- Resolved in the MainWindow class

Demo WPF

IT IS ABOUT PEOPLE



ASP NET MVC

- Used to resolve controller classes
 - Used to resolve injected dependencies into controllers
- Can use a custom controller factory or dependency resolver
- Controllers can be tested and test dependency implementations used
- Most DI containers have specific NuGet integration package for each MVC version

Demo MVC

IT IS ABOUT PEOPLE



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



ASP NET Core

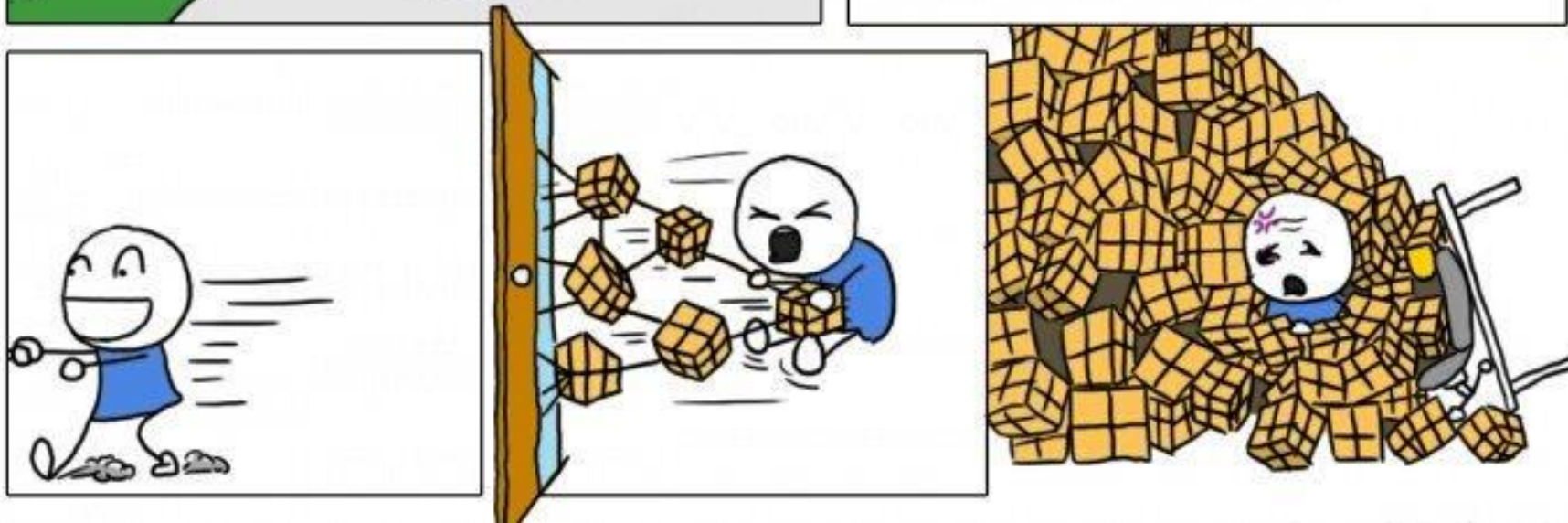
- Out of the box support for DI
- Custom basic container implementation
- It's possible to plug an existing DI container into the asp net core framework
- Asp net core is open source so all code for DI can be seen here
 - <https://github.com/aspnet/DependencyInjection>
- <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection>

Demo Asp net core DI

IT IS ABOUT PEOPLE



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

IT IS ABOUT PEOPLE



Factory pattern

- Gang of four 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

IT IS ABOUT PEOPLE



When they ask me 'How did you become Developer'



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

IT IS ABOUT PEOPLE



New Project

▸ Recent

▾ Installed

▾ Templates

▸ Visual C#

▸ Visual Basic

SQL Server

▸ Azure Data Lake

▸ Other Project Types

Dependency Validation

Search Results

Not finding what you are looking for?

[Open Visual Studio Installer](#)

▸ Online

.NET Framework 4.6.2 ▾

Sort by: Default ▾



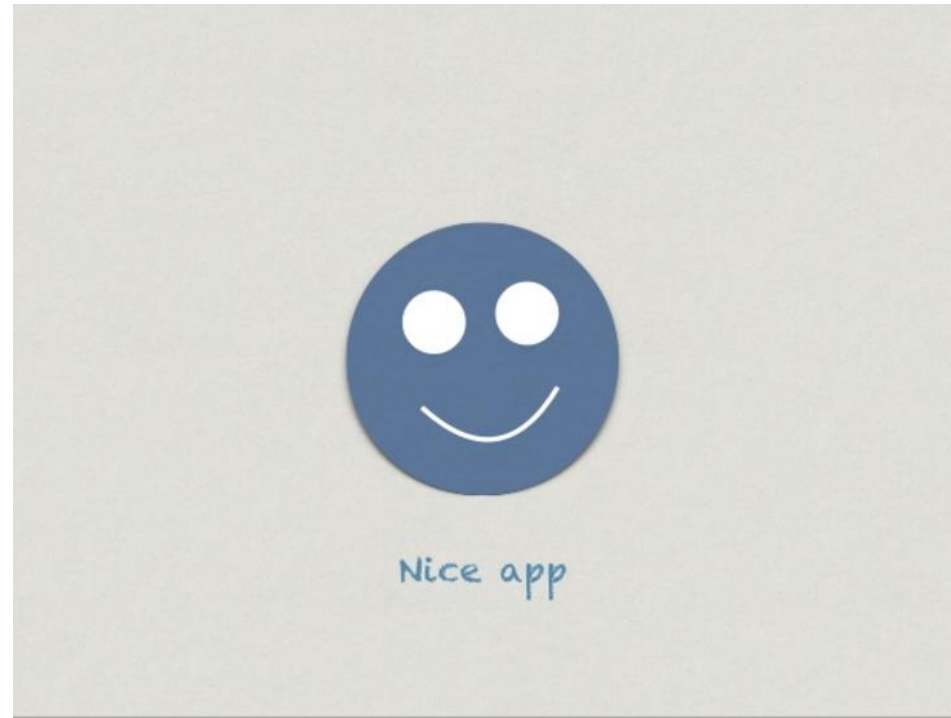
Blank Solution

Visual Studio Solutions

Name:

Solution1

What we want



Quickly becomes



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

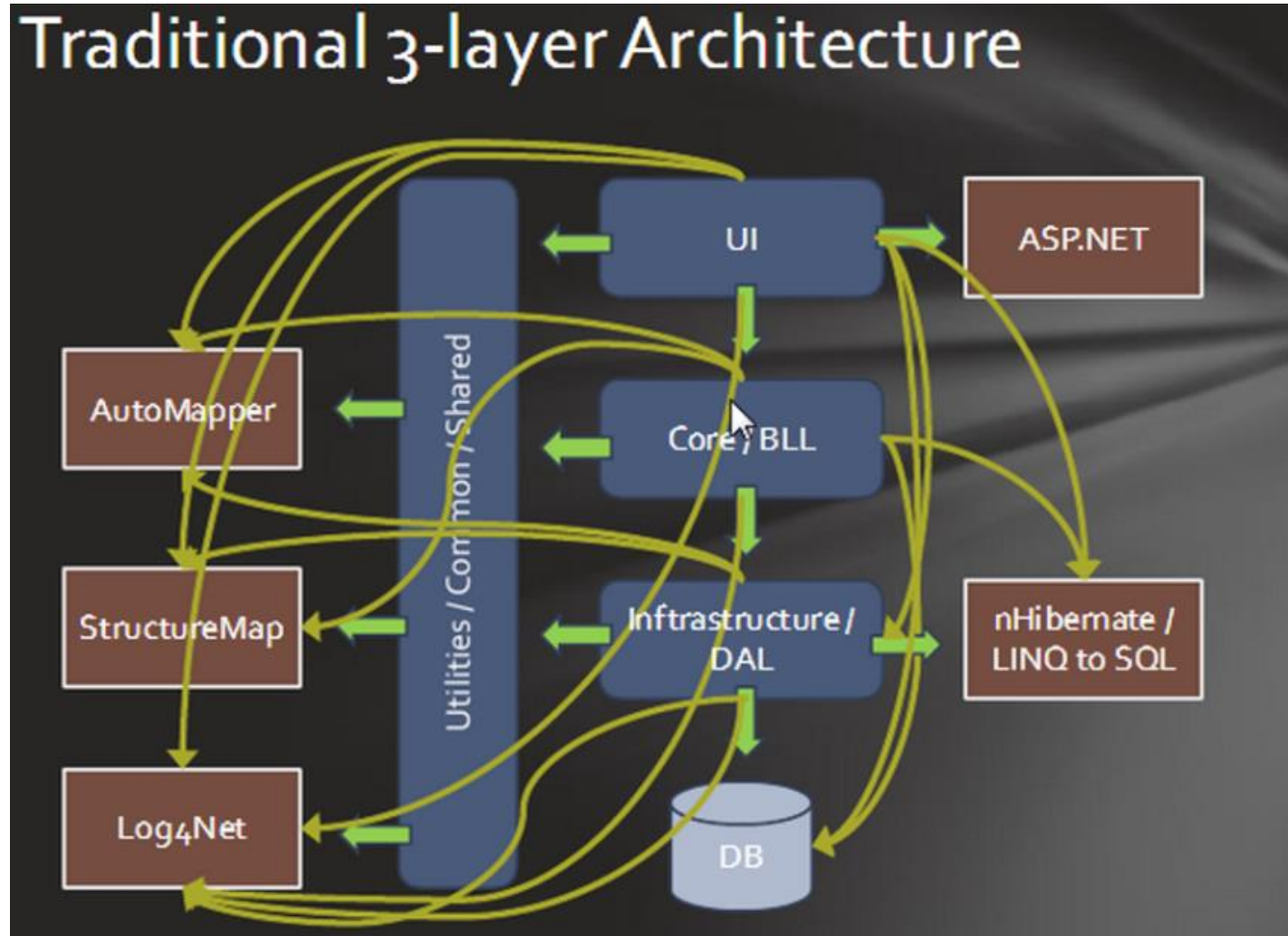
Companies try to solve this in different ways

- A written guide
 - Publisher is a developer with lot's of experience
- Several custom project templates
 - Still to much responsability 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?

First answer on cohesion and coupling: 3 layered 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)

Hexagonal architecture

- Ports and adapters
 - There is nothing hexagonal about the hexagonal architecture
 - Just to represent it visually and it foresees room for drawing ports and adapters
- Introduced by Alistair Cockburn in 2000
- He was working as a consultant and needed to stop development because a different team first had to write the data access code (integration between smalltalk and rdms)
- Prevents the infiltration of the business logic into the UI code
- Limits the number of layers
 - Only inside and outside the hexagon

Hexagonal architecture

- Divide the software in 2 distinct regions
 - Inside the hexagon (the business application logic)
 - Outside the hexagon (Infrastructure code like database code, web api's)
- Protect the inside with ports and adapters
 - The port is a description of the dependency, the intention
 - The adapter is what you code to bridge the domain and the dependency
 - The adapter belongs in infrastructure
 - The port belongs in the domain
 - The port is to go in the hexagon/domain
 - The adapter is to go out the hexagon/domain
- Wiring is done with IOC principles
- It's about configurable dependencies on the left and the right side of the hexagon

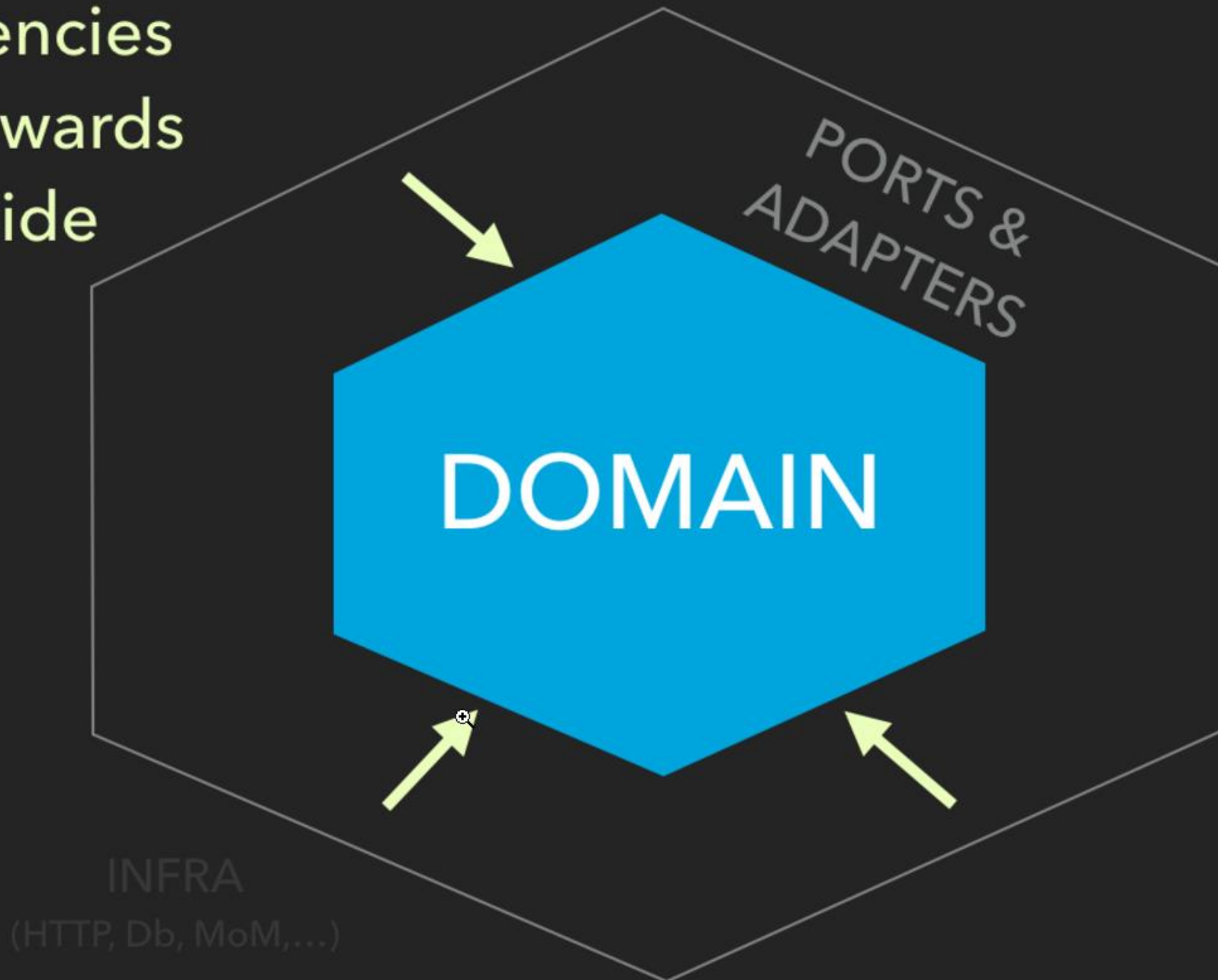
Hexagonal architecture

- Drivers
 - A driver could be a script, web api, other applications, user or automated test
 - The test is the first user of your system
- Recipients and repositories
 - Recipients can be anything that receives notifications from the app
 - Repositories is the persistence store of the application
- Drivers are primary actors
- Recipients and repositories are secondary actors
- Who starts the communication (trigger) is the primary actor
 - Primary actor knows the dependency

Hexagonal architecture

- Pizza shop example (primary actor)
 - Imagine you are the hexagon -> responsible for preparing pizza
 - People can order at the table
 - People can use the drive in
 - People can order online
 - It can be the boss who just want's to time how long it takes to get a pizza and throw it away
- Lot's of drivers but you are inside the hexagon and the outcome is all the same to you

Dependencies
always towards
the inside



Infra code
(adapters) can
reference the
domain code.

Not the other way
round!

To exit from the
domain code
=> Dependency
Inversion Principle
(DIP)

REST
consumers

PORTS &
ADAPTERS

DOMAIN

Monitoring
back-end

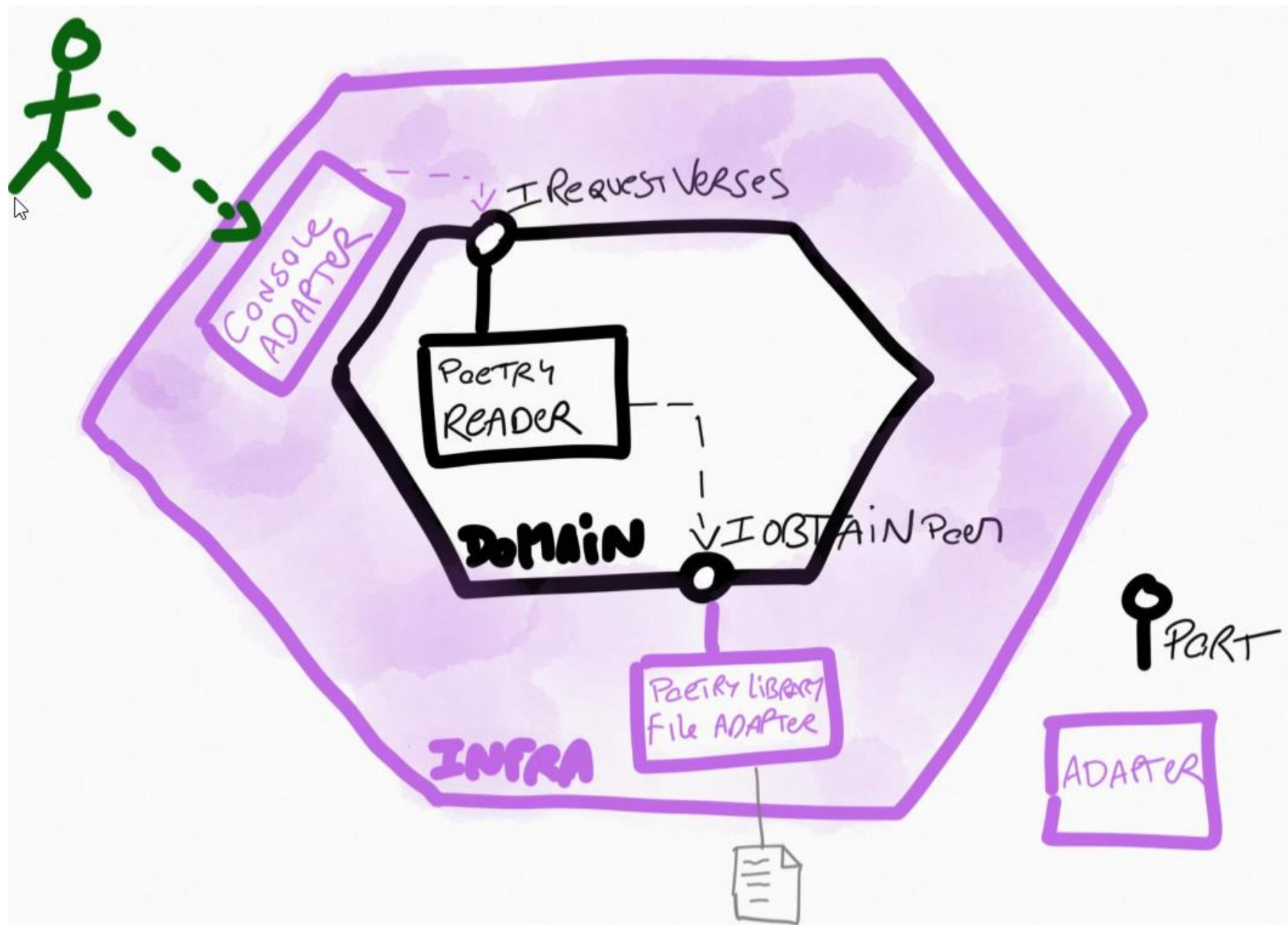
* MQ
consumers

Db









Demo Hexagonal

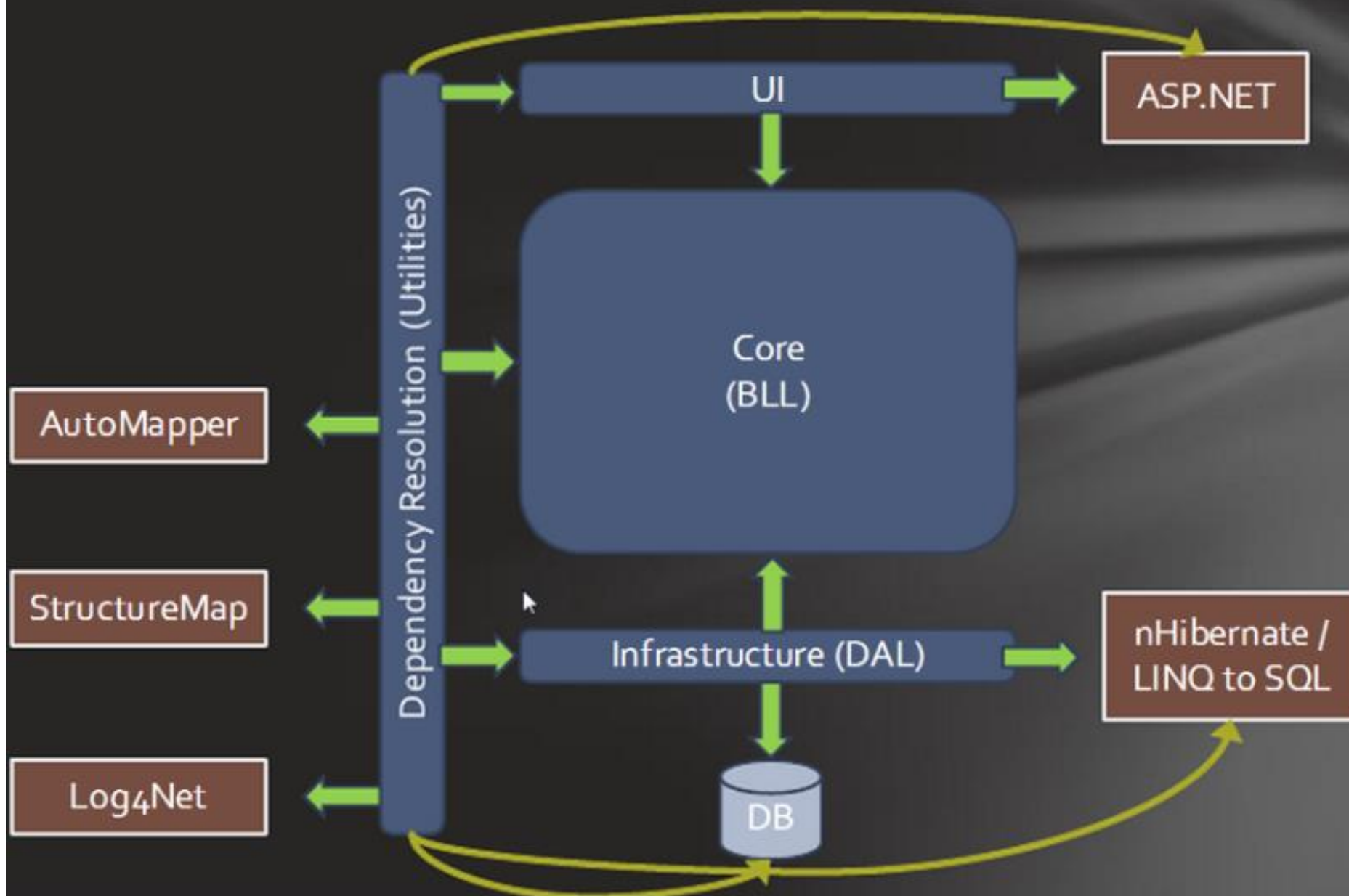
IT IS ABOUT PEOPLE



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

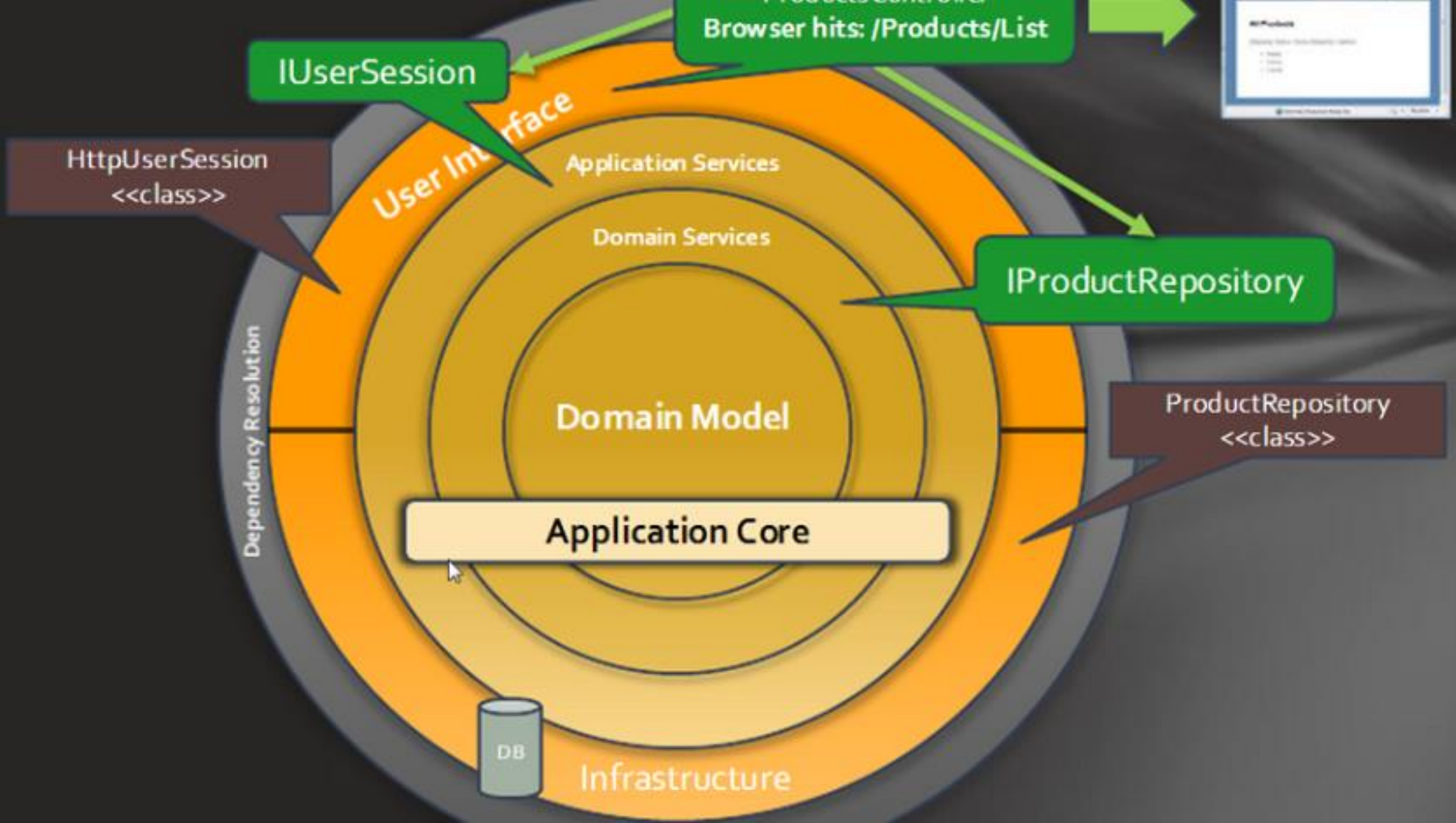
- CORE
- Everything unique to the business: domain models, validation rules, workflows
- Defines all technical implementations as interfaces
- No references to external libraries
- No technology specific code: WCF, MVC, EF

Onion architecture

- INFRASTRUCTURE
- Provide implementations to CORE interfaces
- Call web services, acces 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

IT IS ABOUT PEOPLE



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

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)