Visual Studio LIVE!
EXPERT SOLUTIONS FOR ENTERPRISE DEVELOPERS

MICROSOFT HQ

Dependency Injection in .NET

Hands-on Lab

Jeremy Clark

Developer Betterer

JeremyBytes.com

Intermediate

Today's Agenda

- Overview
  - What is it?
  - Why do we care?
  - How do we use it?
- Lab
- A Closer Look at DI
  - Dimensions of DI
  - SOLID Principles
  - Tips
- Lab

Today's Agenda

- Patterns and Abstractions
  - DI Patterns
  - Other useful Design Patterns
- Lab

- Common Stumbling Blocks
  - Constructor Over-Injection
  - Static Dependencies
  - IDisposable
- Lab

Today's Agenda

- DI Containers
  - Lifetime
  - Configuration
  - Stable vs. Volatile Dependencies
  - DI Container Overview
- Lab

More Information

https://github.com/jeremybytes/vslive-di-lab

Visual Studio LIVE!
EXPERT SOLUTIONS FOR ENTERPRISE DEVELOPERS

MICROSOFT HQ

Dependency Injection in .NET

Part 1: Overview

Jeremy Clark

Developer Betterer

JeremyBytes.com

Intermediate

What Is Dependency Injection?

- Dependency Injection is a software design pattern that allows a choice of component to be made at run-time rather than compile time.
  - Wikipedia 2012

What Is Dependency Injection?

- Dependency injection is a software design pattern that allows the removal of hard-coded dependencies and makes it possible to change them, whether at run-time or compile-time.
  - Wikipedia 2013

What Is Dependency Injection?

- Dependency injection is a software design pattern that implements inversion of control and allows a program design to follow the dependency inversion principle. The term was coined by Martin Fowler.
  - Wikipedia 2014

What Is Dependency Injection?

- In software engineering, dependency injection is a software design pattern that implements inversion of control for software libraries, where the caller delegates to an external framework the control flow of discovering and importing a service or software module. Dependency injection allows a program design to follow the dependency inversion principle where modules are loosely coupled. With dependency injection, the client part of a program which uses a module or service doesn't need to know all its details, and typically the module can be replaced by another one of similar characteristics without altering the client.

  - Wikipedia 2015

What Is Dependency Injection?

- In software engineering, dependency injection is a software design pattern that implements inversion of control for resolving dependencies. A dependency is an object that can be used (a service). An injection is the passing of a dependency to a dependent object (a client) that would use it. The service is made part of the client's state.[1] Passing the service to the client, rather than allowing a client to build or find the service, is the fundamental requirement of the pattern.

  - Wikipedia 2016

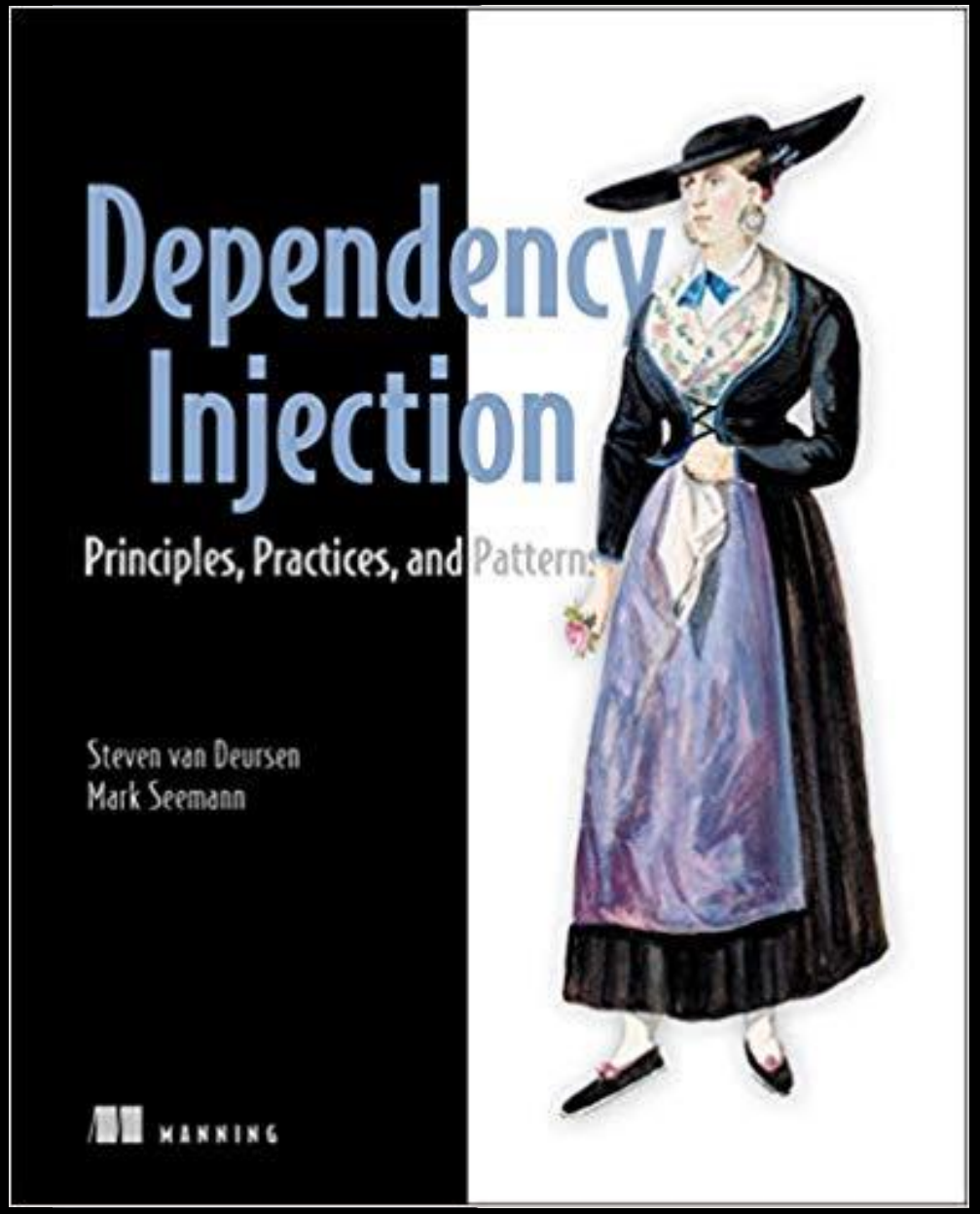
What Is Dependency Injection?

- Dependency Injection is a set of software design principles and patterns that enable us to develop loosely coupled code.
  - Mark Seeman

Dependency Injection

Principles, Practices, and Patterns

- Mark Seeman
- Steven van Deursen



Primary Benefits

- Late Binding
- Extensibility
- Parallel Development
- Maintainability
- Testability

- Adherence to S.O.L.I.D. Design Principles.

Benefits – Late Binding

Services can be swapped with other services without recompiling code.

Benefits – Extensibility

Code can be extended in ways not explicitly planned for.

Benefits – Parallel Development

Code can be developed in parallel with less chance of merge conflicts.

Benefits – Maintainability

Classes with clearly defined responsibilities are easier to maintain.

Benefits – Testability

Classes can be unit tested,
i.e., easily isolated from other classes
and components for testing.

Benefits – SOLID Principles

- Single Responsibility Principle (SRP)
- Open/Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

Dependency Injection Concepts

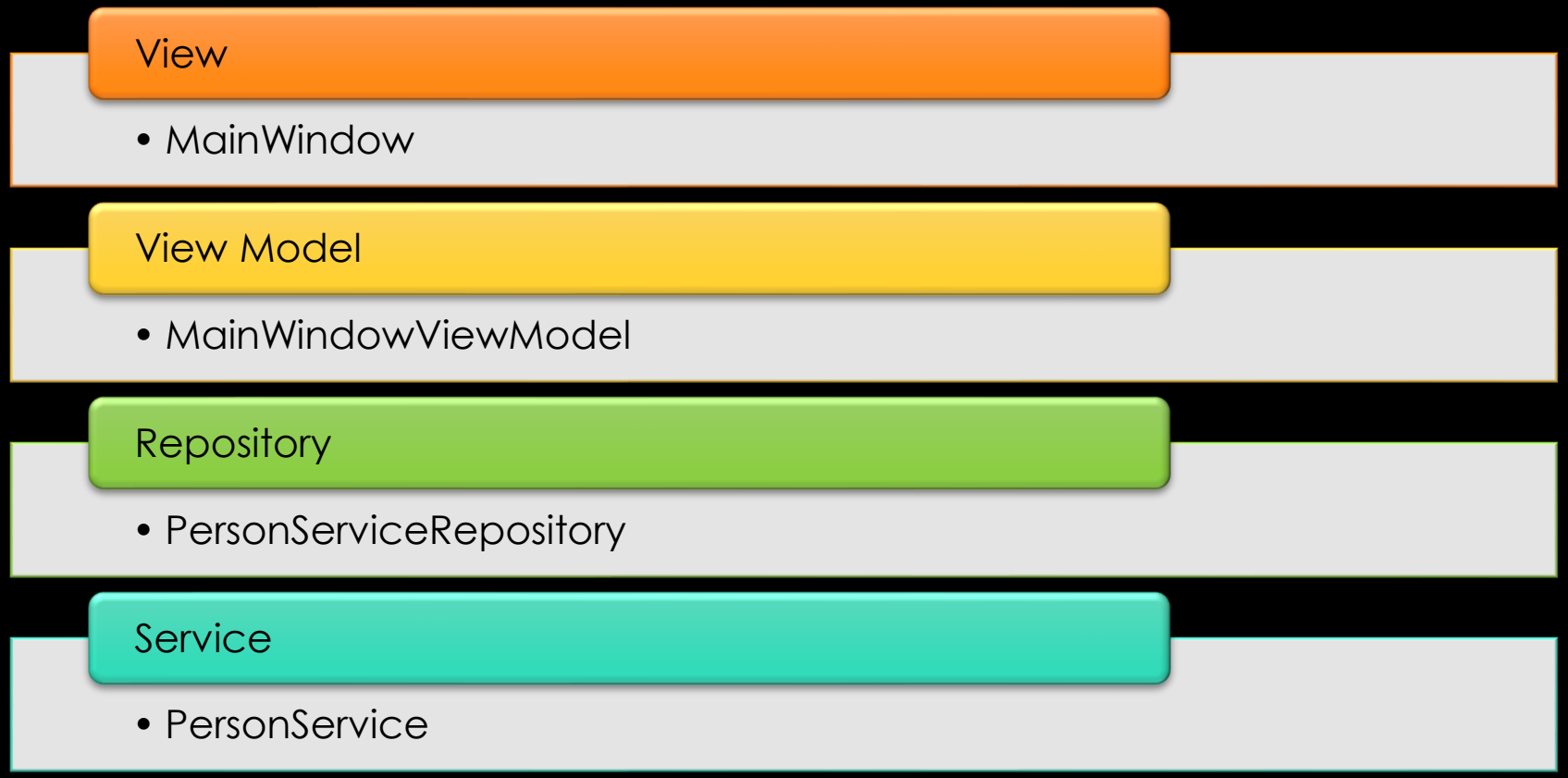
- DI Design Patterns
  - Constructor Injection
  - Property Injection
  - Method Injection
  - Ambient Context
  - Service Locator
- Dimensions of DI
  - Object Composition
  - Interception
  - Lifetime Management

Dependency Injection Containers

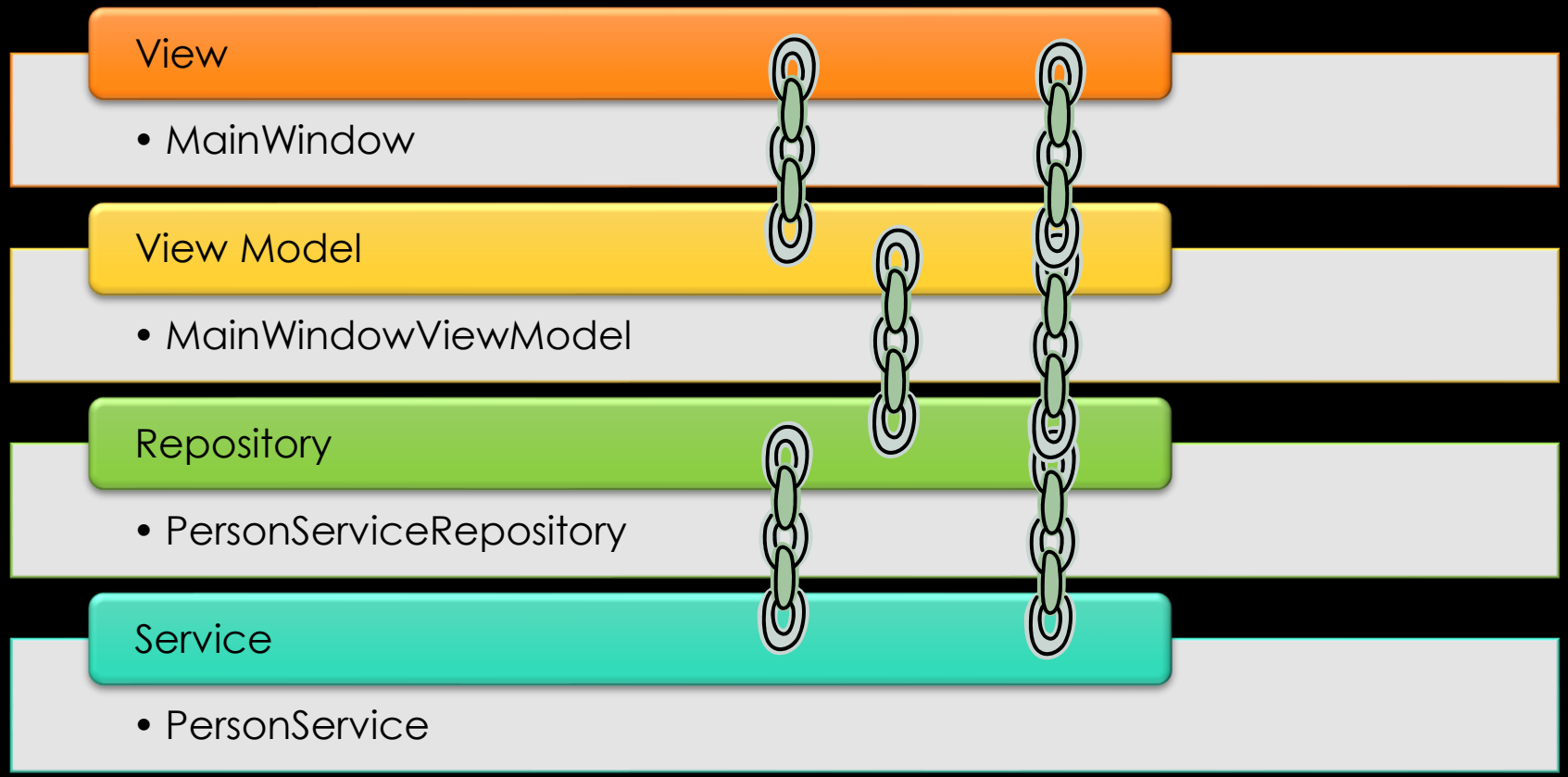
- C# Containers
  - Ninject
  - Autofac
  - Unity
  - Castle Windsor
  - Spring.NET
- Frameworks w/ Containers
  - ASP.NET Core
  - Angular
  - Prism

  and many others

Application Layers

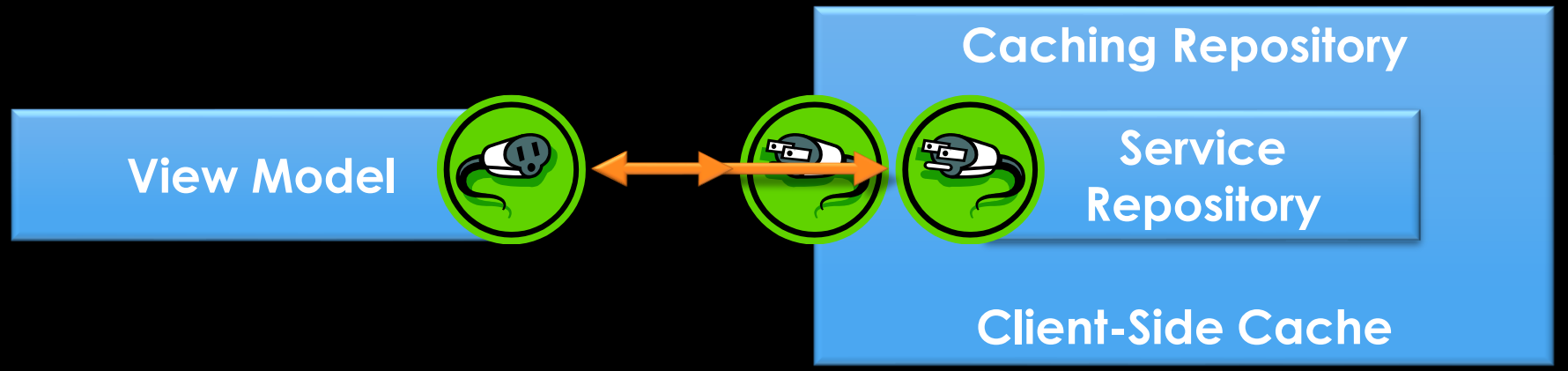


Tight Coupling

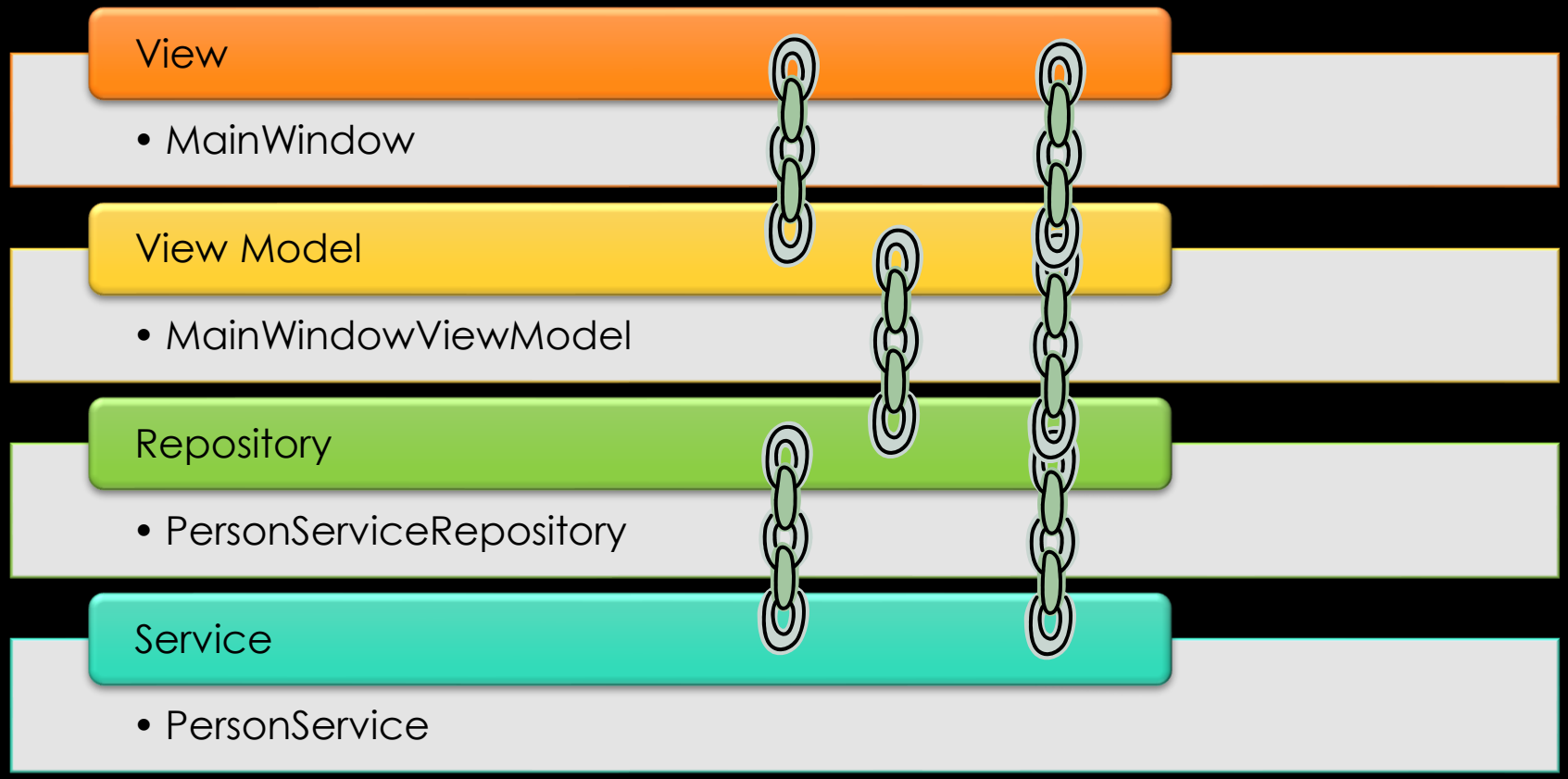


Creating a Caching Repository

The Decorator Pattern



Loose(r) Coupling



Primary Benefits

- Late Binding
- Extensibility
- Parallel Development
- Maintainability
- Testability

- Adherence to S.O.L.I.D. Design Principles.

Dependency Injection Concepts

- DI Design Patterns
  - Constructor Injection
  - Property Injection
  - Method Injection
  - Ambient Context
  - Service Locator
- Dimensions of DI
  - Object Composition
  - Interception
  - Lifetime Management

More Information

https://github.com/jeremybytes/vslive-di-lab

Visual Studio LIVE!
EXPERT SOLUTIONS FOR ENTERPRISE DEVELOPERS

MICROSOFT HQ

Dependency Injection in .NET

Part 2: A Closer Look at DI

Jeremy Clark

Developer Betterer

JeremyBytes.com

Intermediate

Primary Benefits

- Late Binding
- Extensibility
- Parallel Development
- Maintainability
- Testability

- Adherence to S.O.L.I.D. Design Principles.

Dependency Injection Concepts

- DI Design Patterns
  - Constructor Injection
  - Property Injection
  - Method Injection
  - Ambient Context
  - Service Locator
- Dimensions of DI
  - Object Composition
  - Interception
  - Lifetime Management

Deeper Dive

- Dimensions of Dependency Injection
- Programming to an Abstraction
- SOLID Principles

Tips / Techniques

- Read-Only Properties (for Constructor Injection)
- Guard Clauses (prevent unintended nulls)

Dimensions of Dependency Injection

- Object Composition
  - Snapping loosely coupled pieces together
- Lifetime Management
  - Managing creation and re-use of objects.
  - Transient, Singleton, Scoped, Thread
- Interception
  - Adding or replacing functionality in method calls

Programming to an Abstraction

- DI is made possible by programming against abstractions. When we program against an interface or base-class, we can swap out functionality by providing objects that implement the methods and properties of the base object.
- Abstractions provide the seams that make DI possible.

Benefits – SOLID Principles

- Single Responsibility Principle (SRP)
- Open/Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

Single Responsibility Principle

Each class should have only
one reason to change.

Single Responsibility Principle

Cohesion:

The functional relatedness of elements in a class or module. The lower the amount of cohesion, the higher the chance a class violates the Single Responsibility Principle.

Open/Closed Principle

A class should be open for extension,
but closed for modification.

Open/Closed Principle

Application design prevents us from having to make sweeping changes throughout the code base.

There is strong relationship between the Open/Closed Principle and the DRY principle (Don't Repeat Yourself).

Liskov Substitution Principle

Objects in a program should be replaceable with instances of their subtypes without altering the correctness of the program.

Liskov Substitution Principle

Every Dependency should behave as defined by its Abstraction.

Interface Segregation Principle

Clients should not be forced
to depend upon methods that
they do not use. Interfaces belong
to clients, not hierarchies.

Interface Segregation Principle

Many client-specific interfaces are better than one general-purpose interface.

Any time a consumer depends on an Abstraction where some of the members stay unused, this principle is violated.

Dependency Inversion Principle

Abstractions should not depend upon details; details should depend upon abstractions.

Dependency Inversion Principle

We should program against abstractions, and the consuming layer should be in control over the shape of a consumed abstraction.

Tips / Techniques

- Read-Only Properties (for Constructor Injection)
- Guard Clauses (prevent unintended nulls)

Read-Only Properties

- Properties marked as “readonly” are settable only in the constructor. This prevents the property from being inadvertently changed during the lifetime of the object.
- This is applicable to Constructor Injection; for obvious reasons, this would be a problem for Property Injection.

Guard Clauses

- Guard clauses (null checks) should be used in constructors, methods, and property setters to ensure that dependencies are not set to null.
- If a "null behavior" is required, consider using the Null Object pattern. This provides a valid implementation with no actual behavior.

Primary Benefits

- Late Binding
- Extensibility
- Parallel Development
- Maintainability
- Testability

- Adherence to S.O.L.I.D. Design Principles.

Dependency Injection Concepts

- DI Design Patterns
  - Constructor Injection
  - Property Injection
  - Method Injection
  - Ambient Context
  - Service Locator
- Dimensions of DI
  - Object Composition
  - Interception
  - Lifetime Management

More Information

https://github.com/jeremybytes/vslive-di-lab

Visual Studio LIVE!
EXPERT SOLUTIONS FOR ENTERPRISE DEVELOPERS

MICROSOFT HQ

Dependency Injection in .NET

Part 3: Patterns & Abstractions

Jeremy Clark

Developer Betterer

JeremyBytes.com

Intermediate

Primary Benefits

- Late Binding
- Extensibility
- Parallel Development
- Maintainability
- Testability

- Adherence to S.O.L.I.D. Design Principles.

Dependency Injection Concepts

- DI Design Patterns
  - Constructor Injection
  - Property Injection
  - Method Injection
  - Ambient Context
  - Service Locator
- Dimensions of DI
  - Object Composition
  - Interception
  - Lifetime Management

Dependency Injection Patterns

- Constructor Injection
- Property Injection
- Method Injection
- Ambient Context
- Service Locator

Constructor Injection

The dependency is injected into the class through a constructor parameter.

Where to use Constructor Injection

- A dependency will be used/re-used at the class level.
- A non-optional dependency must be provided.

- Advantage: it keeps dependencies obvious. Code will not compile if the dependency is not provided

Property Injection

The dependency is injected into the class by setting a property on that class.

Where to use Property Injection

- A dependency will be used/re-used at the class level.
- A dependency is optional.
- A dependency has a good default value that can be used if a separate implementation is not provided.
- Advantage: we do not need to supply a dependency if we want to use the default behavior
- Disadvantage: the dependency is hidden. It may not be obvious to developers that a separate behavior can be provided.

Method Injection

The dependency is injected into a method through a method parameter.

Where to use Method Injection

- A dependency will only be used by a specific method – i.e., it will not be stored by the class and used in other methods.
- A dependency varies for each call of a method.

Ambient Context

ANTI-PATTERN

The dependency is available as a global object.

Where to use Ambient Context

- This is an anti-pattern and should be avoided.
- This short-circuits the DI principles of Object Composition, Interception, and Lifetime Management

Service Locator

ANTI-PATTERN

The class resolves its own dependencies by requesting them from a service locator.

Where to use Service Locator

- This is an anti-pattern and should be avoided.
- This violates the Dependency Inversion Principle. The class takes responsibility for resolving its own dependencies.
- Dependencies are also hidden. If a new dependency is added to the class, the need is not obvious.
- Errors are moved to runtime if the class is unable to resolve its own dependency.

Useful Design Patterns

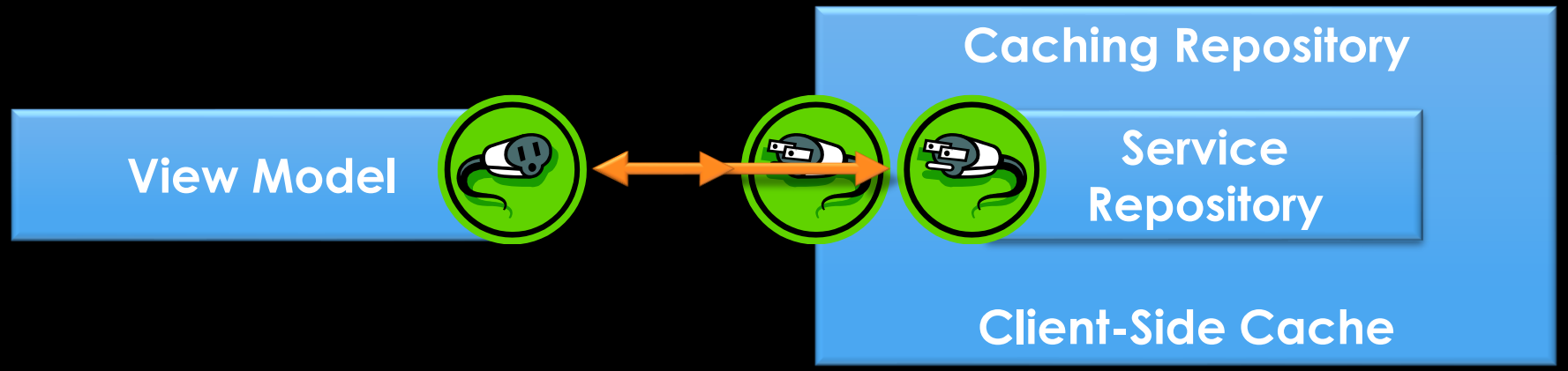
- Decorator
- Proxy
- Composite
- Null Object

Decorator

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Decorator

Caching Decorator



Where to use the Decorator Pattern

- Cross-cutting concerns
- Interception

Proxy

Provide a surrogate or placeholder
for another object to control access to it.

Where to use the Proxy Pattern

- Can be used to encapsulate IDisposable classes.

```
public Task<IEnumerable<Person>> GetPeopleAsync()
{
    using (var repository = new SQLRepository())
    {
        return repository.GetPeopleAsync();
    }
}
```

Composite

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Where to use the Composite Pattern

- A dependency can be a single object or a collection of objects. The client does not need to care.

Business Rules - Composite Example

Interface

```
public interface IOrderRule
{
    bool ValidateRule(Order order);
}
```

Business Rules - Composite Example

Client

```
public Order (IOrderRule rule)…

private bool CheckRules()
{
    rule.ValidateRule(this);
}
```

Business Rules - Composite Example

Basic Rule

```
public TotalItemsRule : IOrderRule
public bool Validate(Order order)
{
  return order.TotalItems < 100;
}
```

Business Rules - Composite Example

```
Composite Rule
public AllOrderRules : IOrderRule
{
    public AllOrderRules(IEnumerable<IOrderRule> rules) …
    public bool Validate(Order order)
    {
        foreach(var rule in rules)
            if (!rule.Validate) …
        return isValid;
    }
}
```

Business Rules - Composite Example

Client is the same regardless of whether “rule” is a single rule or a composite rule.

```
public Order (IOrderRule rule)…

private bool CheckRules()
{
    rule.ValidateRule(this);
}
```

Null Object

Instead of using a null reference
to convey absence of an object,
one uses an object which implements
the expected interface, but whose
method body is empty.

Null Object

The advantage of this approach over a working default implementation is that a null object is very predictable and has no side effects: it does nothing.

Where to use the Null Object Pattern

- Can be used for optional dependencies (which are truly optional).
- Rather than having null checks. A null object can provide empty functionality without the risk of null reference exceptions.

Null Object Example

```
public class NullLogger : ILogger
{
    public Log(string message)
    {
        // Does nothing (also no NullReferenceException)
    }
}
```

Primary Benefits

- Late Binding
- Extensibility
- Parallel Development
- Maintainability
- Testability

- Adherence to S.O.L.I.D. Design Principles.

Dependency Injection Concepts

- DI Design Patterns
  - Constructor Injection
  - Property Injection
  - Method Injection
  - Ambient Context
  - Service Locator
- Dimensions of DI
  - Object Composition
  - Interception
  - Lifetime Management

More Information

https://github.com/jeremybytes/vslive-di-lab

Visual Studio LIVE!
EXPERT SOLUTIONS FOR ENTERPRISE DEVELOPERS

MICROSOFT HQ

Dependency Injection in .NET

Part 4: Common Stumbling Blocks

Jeremy Clark

Developer Betterer

JeremyBytes.com

Intermediate

Common Stumbling Blocks

- Constructor Over-Injection
- Static Dependencies
- Dealing with IDisposable (and other lifetime concerns)
- Using Factory Methods
- Configuration Strings

Constructor Over-Injection

- Symptom: a constructor contains a large number of parameters.
- Code Smell: this often points to a violation of the Single Responsibility Principle

Constructor Over-Injection

- Possible Solution:
  Break up the class along the functionality lines. This generally results in object groupings and dependencies that are more manageable in size.

Constructor Over-Injection

- Possible Solution:
  Create Parameter Objects.
- A parameter object can combine multiple dependencies into a single parameter. This allows grouping of parameters along functional lines.

Static Dependencies

- Symptom: A class relies on a static object as a dependency.
- Problem: This makes it difficult to swap out functionality for testing.
- Example: DateTime.Now()

Static Dependencies

- Possible Solution:
  Instead of relying on the static object directly, a class can wrap that dependency in a property. By default, the static dependency will be used, but it's possible to provide a different implementation for testing or other purposes.

Dealing with IDisposable

- Symptom: a dependency implements IDisposable.
- Code Smell: This is a leaky abstraction. The requirement to dispose of the object "leaks" out; the consuming class needs to know this about the dependency.

Dealing with IDisposable

- Possible Solution:
  Create a proxy class to wrap the functionality.
- Each method call creates the underlying object inside a “using”, the makes the call.
- The object is disposed and resources released.

- Example: SQL Repository

Factory Methods

- Symptom: A class uses a factory method and has a private constructor.
- Problem: This breaks auto-wiring in DI containers.

- Solution: We'll take a closer look after exploring DI containers further.

Factory Methods

- Symptom: A class uses a factory method and has a private constructor.
- Problem: This breaks auto-wiring in DI containers.

- Solution: We'll take a closer look after exploring DI containers further.

Configuration Strings

- Symptom: A class constructor needs a string as a parameter, such as a connection string.
- Problem: This breaks auto-wiring in DI containers.

- Solution: We'll take a closer look after exploring DI containers further.

Primary Benefits

- Late Binding
- Extensibility
- Parallel Development
- Maintainability
- Testability

- Adherence to S.O.L.I.D. Design Principles.

Dependency Injection Concepts

- DI Design Patterns
  - Constructor Injection
  - Property Injection
  - Method Injection
  - Ambient Context
  - Service Locator
- Dimensions of DI
  - Object Composition
  - Interception
  - Lifetime Management

More Information

https://github.com/jeremybytes/vslive-di-lab

Visual Studio LIVE!
EXPERT SOLUTIONS FOR ENTERPRISE DEVELOPERS

MICROSOFT HQ

Dependency Injection in .NET

Part 5: Containers

Jeremy Clark

Developer Betterer

JeremyBytes.com

Intermediate

Primary Benefits

- Late Binding
- Extensibility
- Parallel Development
- Maintainability
- Testability

- Adherence to S.O.L.I.D. Design Principles.

Dependency Injection Concepts

- DI Design Patterns
  - Constructor Injection
  - Property Injection
  - Method Injection
  - Ambient Context
  - Service Locator
- Dimensions of DI
  - Object Composition
  - Interception
  - Lifetime Management

Dependency Injection Containers

- C# Containers
  - Ninject
  - Autofac
  - Unity
  - Castle Windsor
  - Spring.NET
- Frameworks w/ Containers
  - ASP.NET Core
  - Angular
  - Prism

  and many others

Object Composition

- Composing objects should happen as close to the application entry point as possible.
- In a desktop application, this means application startup.
- In an ASP.NET MVC application, this means the start of the request (generally creation of the controller).
- For other web applications, the entry point may be framework specific.

Object Composition

- The composition root should be the ONLY place a DI container is used. If the container is used in other areas, this is a code smell that the code violates DI principles.
- This often happens when the Service Locator anti-pattern is used.

Lifetime Management

- Transient
- Singleton
- Scoped
- Thread (not as relevant as it used to be)

Transient Lifetime

- A new instance of a dependency is used whenever there is a request for that dependency.
- Each instance is independent and will get cleaned up / garbage collected as it goes out of scope.

Singleton Lifetime

- A single instance of a dependency is used whenever there is a request for that dependency.
- The lifetime is managed by the DI container. It may or may not be released when all references have been released.

Scoped Lifetime

- A new instance is used for each “scope” of an application.
- If a dependency is needed multiple times within the same scope, a single instance of that dependency is used.
- Scope example: In a web application, the scope generally refers to the current request.
- Container scopes can be explicitly defined.

Thread Lifetime

- A new instance is used for each thread of an application.
- This lifetime is less common due to an increase in asynchronous programming.
- Scoped lifetime is preferred over thread lifetime.

Interception

- Interception is used for cross-cutting concerns.
- By using a Decorator, an object can intercept calls to the underlying object and add its own behavior.
- Examples:
  - Auditing
  - Logging
  - Authorization
  - Caching

Interception – Authorization

```
public void UpdateOrder(Order order)
{
    if (!authorized)
      throw new NotAuthorized Exception;
    realUpdater.UpdateOrder(order);
}
```

Interception – Logging

```
public void UpdateOrder(Order order)
{
    LogMethodEntry();
    realUpdater.UpdateOrder(order);
    LogMethodExit();
}
```

Interception – Combining Decorators

- UpdateOrder on authorization decorator checks authorization and then calls…
- UpdateOrder on logging decorator logs the method entry and then calls…
- UpdateOrder on the real repository which returns to…
- UpdateOrder on logging decorator logs the method exit and then returns to…
- UpdateOrder On authorization decorator.

Configuring DI Containers

- Auto-Wiring (container can figure out concrete types)
- Auto-Registration
- Configuration as Code
- Configuration Files

Auto-Wiring

- The container can determine the object composition based on parameters and concrete types.
- This works as long as there are no abstract dependencies (such as interfaces or abstract classes).
- With abstract dependencies, additional configuration is required.

Auto-Registration

- Reflection is used to load an assembly and pull out the associated types. These types are registered with the container.
- Since this registration is at runtime, it allows for late binding / dynamic loading of types.
- Since there is no compile-time checking, this can lead to missing dependencies and other runtime errors.

Auto-Registration

- Auto-Registration works well when there is a naming convention, such as "…Repository" or "…Command".
- This also works well when each dependency implements only one interface.

Configuration as Code

- Abstractions are matched up to concrete types in code.
- This also allows for factory methods to be associated with particular types.
- Benefits include compile-time type checking.

Configuration Files

- Abstractions are matched to concrete types in configuration files (such as JSON or XML).
- This allows for runtime binding.
- The files can get complex quite quickly.
- There is no compile-time checking.
- There is no debugging.

Preferred Configuration

- Use Auto-Registration where possible
  - This leads to less configuration to maintain.
  - This works well when using name conventions.
- Prefer Configuration as Code
  - This allows for compile-time checking and easier debugging.
  - There is also flexibility when it comes to factory methods and other unusual bindings.
- Use Configuration Files sparingly.
  - These are the most brittle, but they are often the best solution for late binding.

Dependency Injection Containers

- ASP.NET MVC Core
- Ninject
- Autofac

ASP.NET MVC Core

Applies to ASP.NET MVC Core

- Built-in dependency injection
- Can be used as a stand-alone product, but a third-party container would be better.
- Supports Constructor Injection and Method Injection.
- Supports lifetime management.

ASP.NET MVC Core

Documentation:

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-2.2

Ninject

Applies to .NET Applications (Framework & Core)

- Supports major DI features.
- Auto-wiring, configuration as code.
- Auto-registration requires manual reflection.
- Lifetime Management

Ninject

- Constructor Arguments
  - .WithConstructorArguments()
- Property Injection
  - .WithPropertyValue()
- Factory Method
  - .ToMethod<T>(c => myFactoryMethod())

Ninject

Documentation:

https://github.com/ninject/Ninject/wiki

Autofac

Applies to .NET Applications (Framework & Core)

- Supports major DI features.
- Auto-wiring, configuration as code.
- Auto-registration.
- Lifetime Management

Autofac

- Decorators
  - .RegisterDecorator<TDecorator>(…)
- Property Injection
  - .WithProperty<TPropertyType>(…)
- Late Binding (with configuration file)
  - .RegisterModule(new ConfigurationModule(config)

Autofac

Documentation:

https://autofaccn.readthedocs.io/en/latest/

Stable and Volatile Dependencies

- A stable dependency is one that is not likely to change over the life of the application. For example, classes in the .NET Base Class Library (BCL)
- A volatile dependency is one that is likely to change or needs to be swapped out for fake behavior in unit tests.

Criteria for Stable Dependencies

- The class or module already exists
- You expect that new versions won't contain breaking changes
- The types in question contain deterministic algorithms
- You never expect to have to replace, wrap, decorate, or intercept the class or module with another

Criteria for Volatile Dependencies

- The dependency introduces a requirement to set up or configure a runtime environment for the application
  - Web services, databases, network calls
- The dependency doesn't yet exist or is still in development

Criteria for Volatile Dependencies

- The dependency isn't installed on all machines in the development organization
  - Expensive 3rd party library
- The dependency contains non-deterministic behavior
  - Random number generator
  - DateTime.Now

Factory Methods

- Symptom: A class uses a factory method and has a private constructor.
- Problem: This breaks auto-wiring in DI containers.

Factory Methods

- Possible Solution:
  Most DI containers have a way to bind to a factory method.

- Ninject Example:

```
Container
    .Bind<ConcreteType>()
    .ToMethod(c => FactoryForConcreteType());
```

Configuration Strings

- Symptom: A class constructor needs a string as a parameter, such as a connection string.
- Problem: This breaks auto-wiring in DI containers.

Configuration Strings

- Possible Solution:
  Create a parameter object to hold the string. This gives a strongly-typed object that can be configured and resolved by the container.

- This is a preferred method since it gives additional type safety.

Configuration Strings

- Alternate Solution:
  Use the factory method syntax to inject the string manually.

- Ninject Example:

```
Container
    .Bind<ConcreteType>()
    .ToMethod(c => new ConcreteType(paramString))
```

More Information

https://github.com/jeremybytes/vslive-di-lab