# Leveling Up Dependency Injection in C#

## 2: Deeper Dive

Jeremy Clark
www.jeremybytes.com
@jeremybytes

# What Is Dependency Injection?

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

# Primary Benefits

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

# Dependency Injection Concepts

- DI Design Patterns
  - Constructor Injection
  - Property Injection
  - Method Injection
- Dimensions of DI
  - Object Composition
  - Interception
  - Lifetime Management

# 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.

# 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, cloud services
- 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

# 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.

# Useful Design Patterns

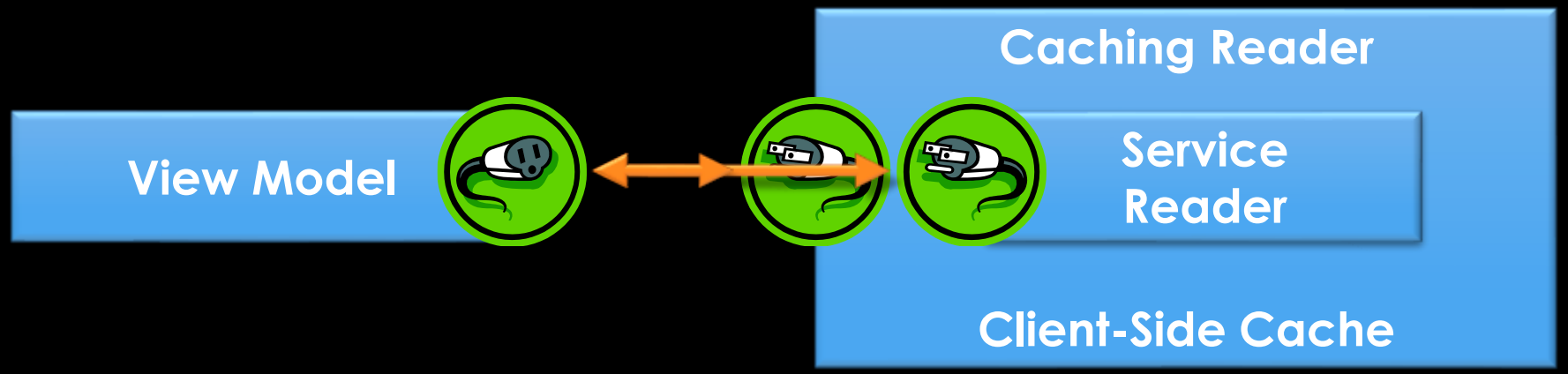
- Decorator
- Proxy
- Null Object

# 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 async Task<IReadOnlyCollection<Person>> GetPeople()
{
    using var reader = new SQLReader(sqlFileName);
    return await reader.GetPeople();
}
```

*Note: we must “await” the proxied reader here. This will ensure that the “GetPeople” method completes before the reader is disposed.

# 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)
    }
}
```

# 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 indicates that a class is trying to do too much.

# 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.

# 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.

# 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

# Dependency Injection Containers

- C# Containers
  - Ninject
  - Autofac
- 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.

# 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

# 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.

# 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());
```

# Factory Methods

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

- ASP.NET Core Example:

```
builder.Services
    .AddSingleton<ConcreteType>(s => FactoryForType());
```

# 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

- Sample (strong-typed parameter):

```
builder.Services
    .AddSingleton<ServiceReaderUri>(s =>
        new ServiceReaderUri("http://localhost:9874"));
builder.Services
    .AddSingleton<IPersonReader, ServiceReader>();
```

# Configuration Strings

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

- Ninject Example:

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

# Resources

## Code Samples & Resources

https://github.com/jeremybytes/di-dotnet-workshop-2022

# Thank You!


## Jeremy Clark

- http://www.jeremybytes.com
- jeremy@jeremybytes.com
- @jeremybytes