Leveling Up Dependency Injection in .NET

3: Patterns and Abstractions

Jeremy Clark
www.jeremybytes.com
@jeremybytes

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

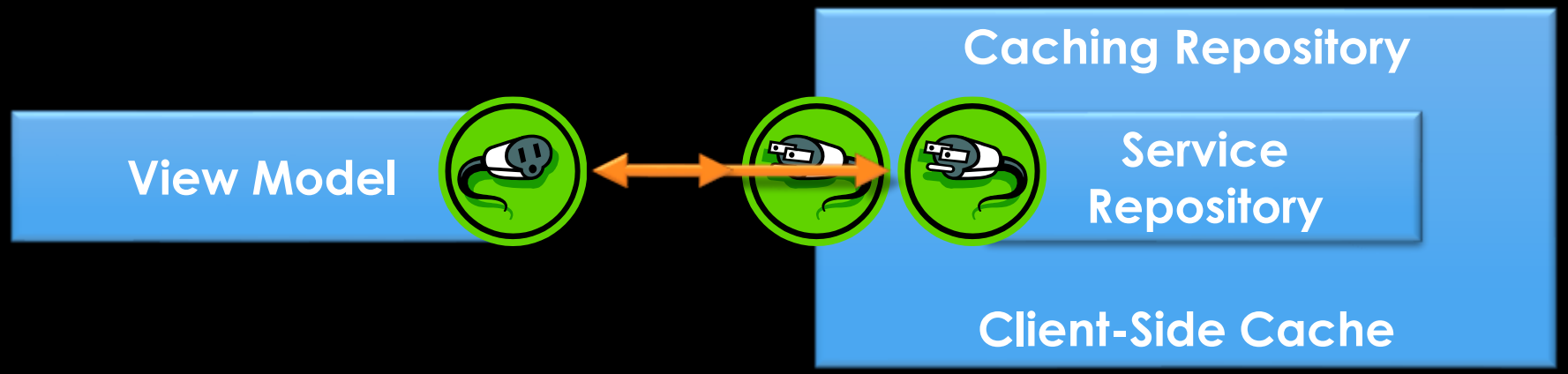
- Decorator
- Proxy
- Composite
- Null Object

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/di-dotnet-workshop