Dependency Injection On-Ramp

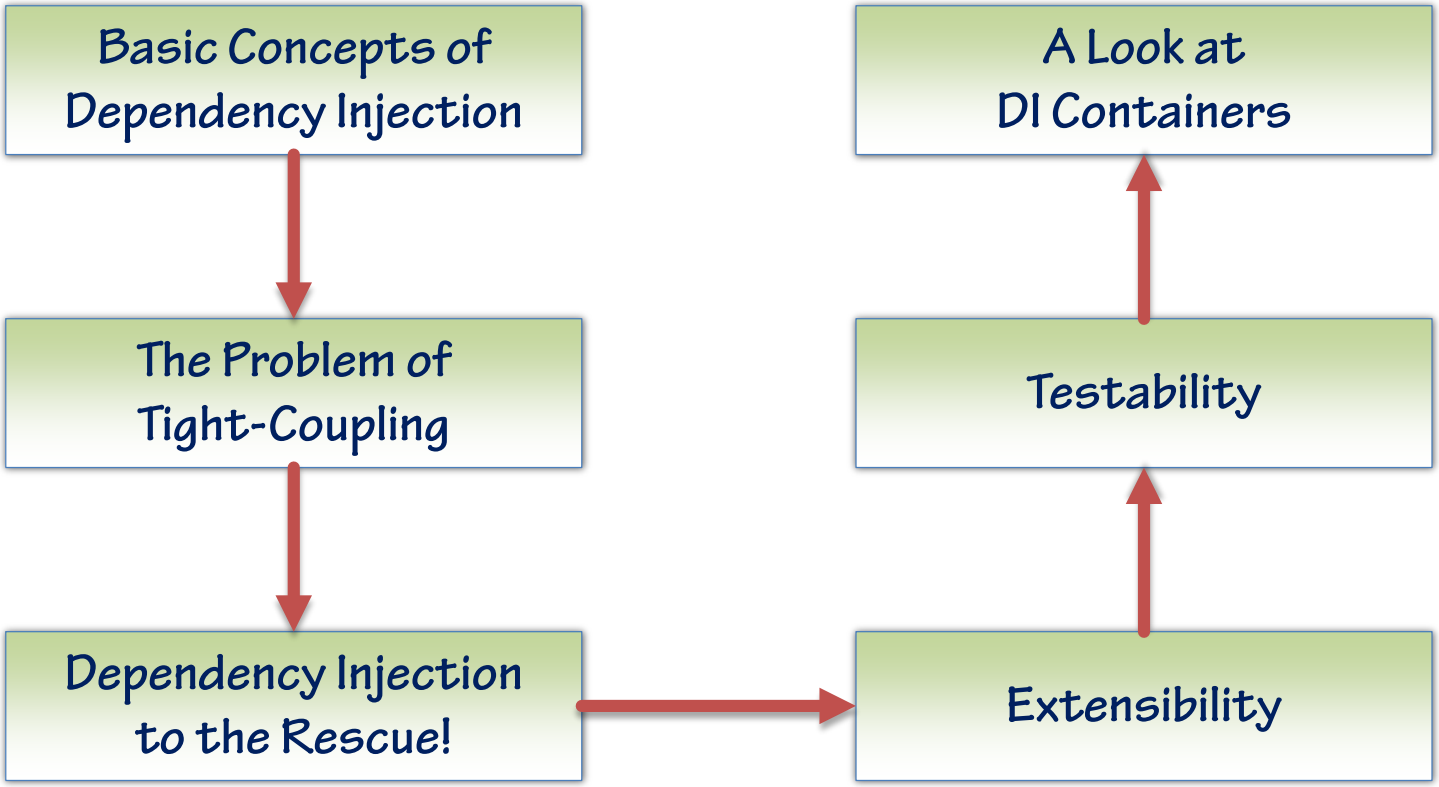
An Introduction to the Principles of Dependency Injection

Jeremy Clark
www.jeremybytes.com
jeremy@jeremybytes.com

pluralsight
hardcore developer training

Goal

- Get Comfortable with Dependency Injection



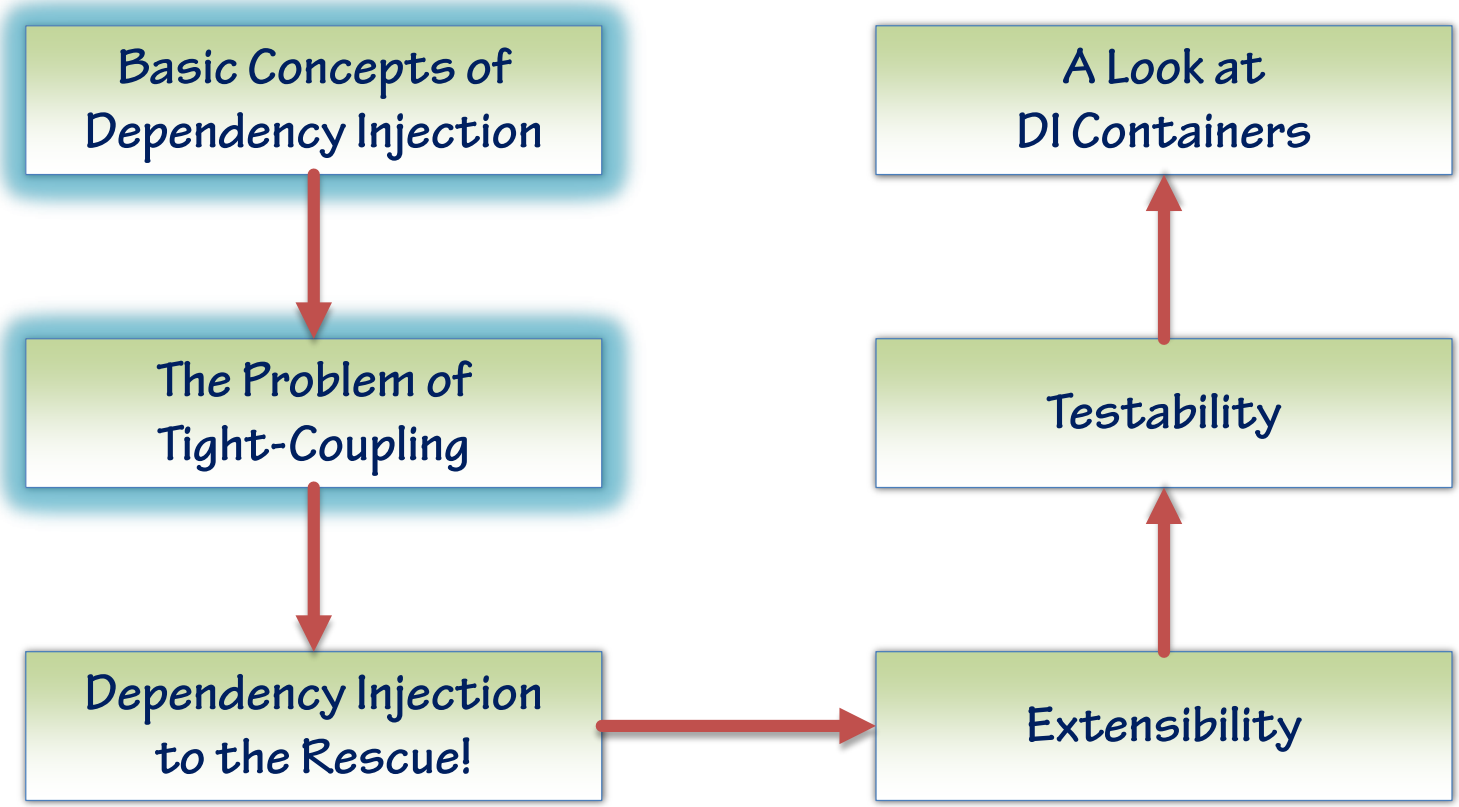
Pre-requisites

- **Good understanding of C# basics**
  - Constructors
  - Properties
  - Data Binding

- **Good understanding of Interfaces**

Goal

- Get Comfortable with Dependency Injection



What is Dependency Injection?

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

Late Binding (run-time binding) is just one benefit of Dependency Injection

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

A little better: includes both compile-time and run-time bindings

What is Dependency Injection?

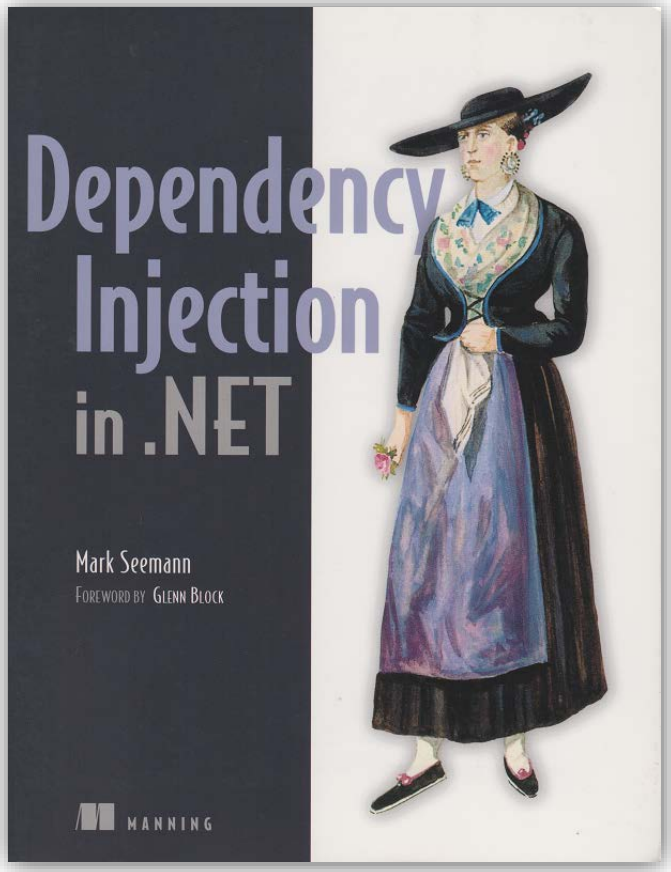
Dependency Injection is a set of software design principles and patterns that enable us to develop loosely coupled code.

-Mark Seemann

Seemann. *Dependency Injection in .NET*. Manning, 2012.

Who is Mark Seemann?

Dependency Injection in .NET



ISBN: 978-1-935182-50-4

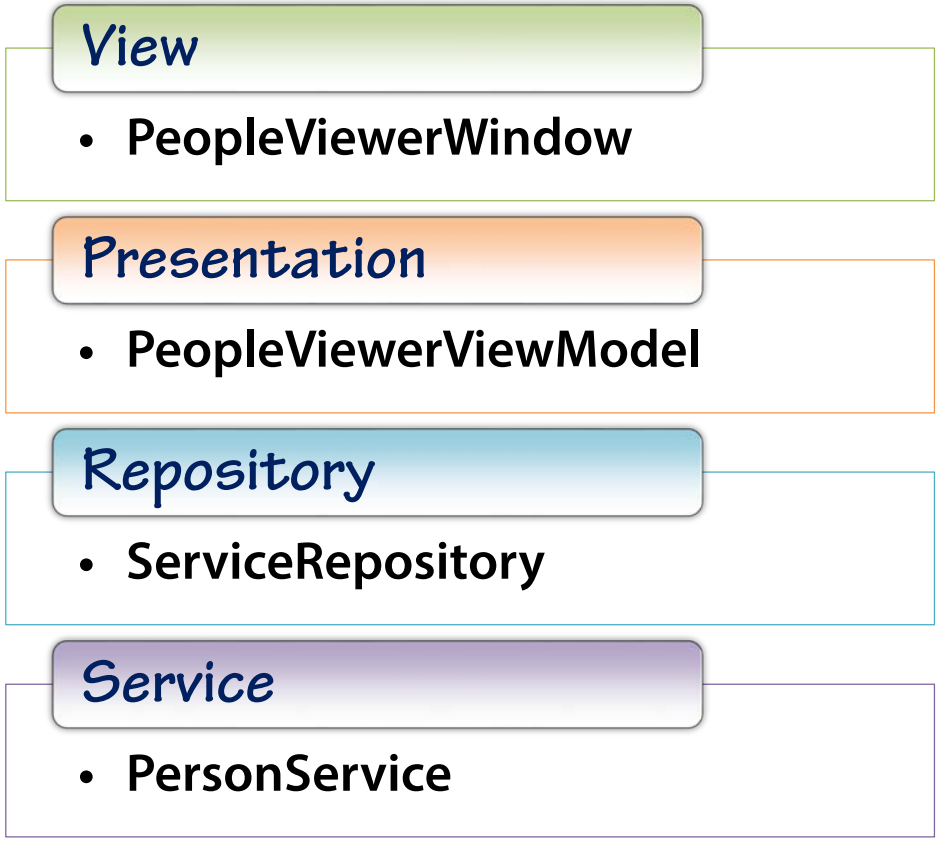
Why Loosely-Coupled Code?

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

Dependency Injection Concepts

- **Patterns**
  - Constructor Injection
  - Property Injection
  - Method Injection
  - Ambient Context
  - Service Locator
- **Object Composition**
  - Composition Root
- **DI Containers**
  - Unity, Ninject, Castle Windsor, Autofac, StructureMap, Spring.NET, and many others

Application Layering

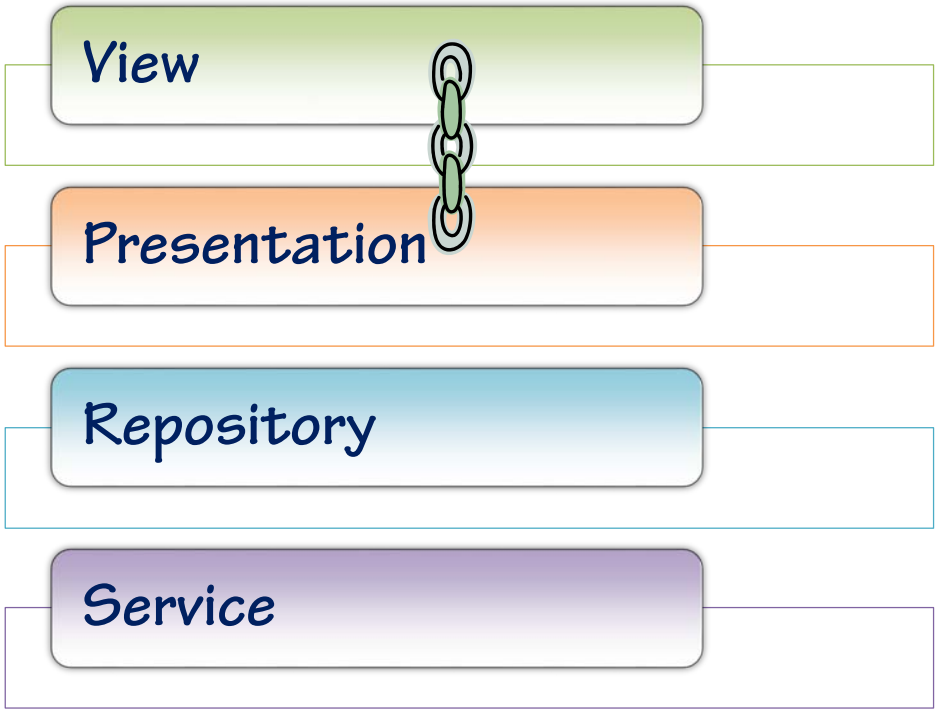


View – View Model Relationship

```
public partial class PeopleViewerWindow : Window
{
    public PeopleViewerWindow()
    {
        InitializeComponent();
        DataContext = new PeopleViewerViewModel();
    }
    ...
}
```

- **The View takes responsibility for creating and managing the View Model**

Tight Coupling



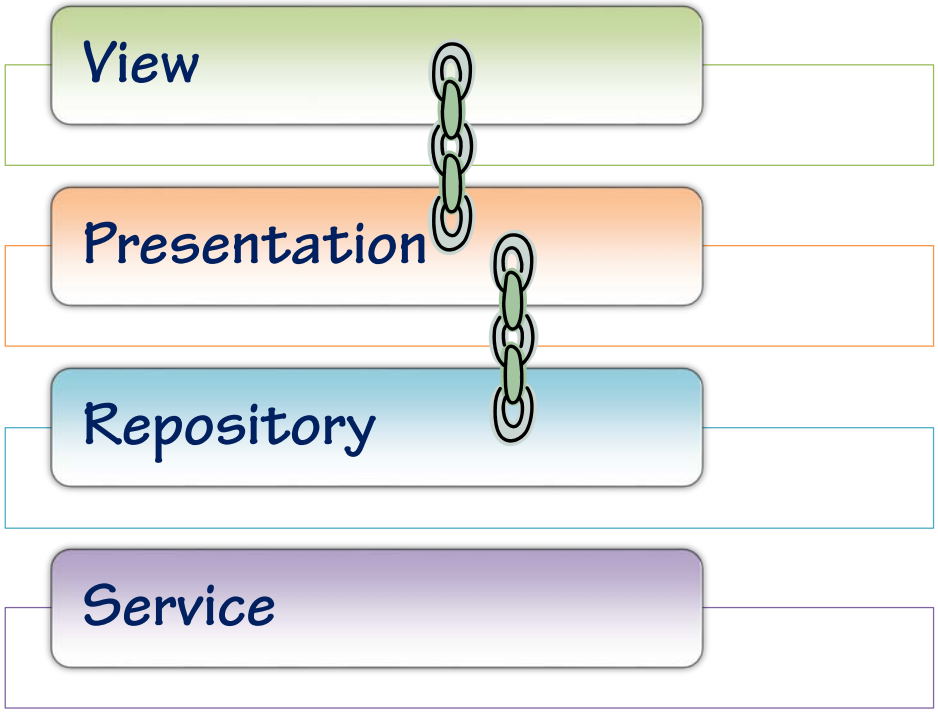
View Model – Repository Relationship

```
public class PeopleViewerViewModel : INotifyPropertyChanged
{
    protected ServiceRepository Repository;

    public PeopleViewerViewModel()
    {
        Repository = new ServiceRepository();
    }
    ...
}
```

- **The View Model references a concrete type of Repository**
- **The View Model takes responsibility for creating and managing the Repository**

Tight Coupling



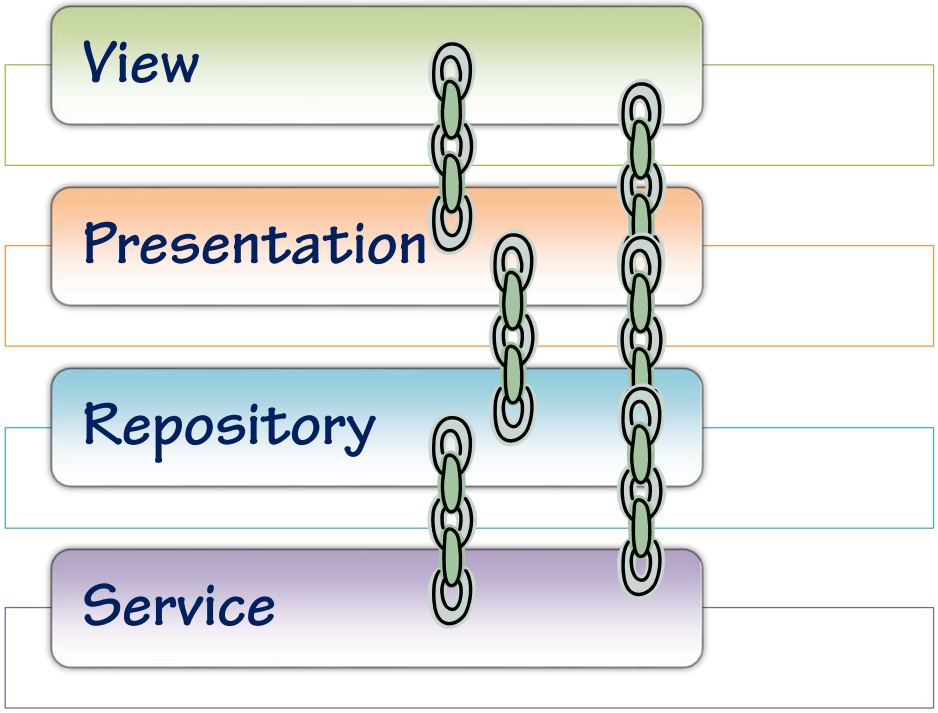
Repository – Service Relationship

```
public class ServiceRepository
{
    PersonServiceClient _serviceProxy = new PersonServiceClient();
    ...
}
```

- **The Repository references a concrete type of Service**
- **The Repository takes responsibility for creating and managing the Service proxy**

Note: This coupling isn't all that bad, but it can still be improved.

Tight Coupling



Scenario 1: Different Repositories

```
public class PeopleViewerViewModel : INotifyPropertyChanged
{
    protected ServiceRepository Repository;

    public PeopleViewerViewModel()
    {
        Repository = new ServiceRepository();
    }
    ...
}
```

- Add the option for a SQL Server Data Store
- Add the option for a CSV File Data Store
- Include other data stores in the future

Scenario 1: Different Repositories

```
public class PeopleViewerViewModel : INotifyPropertyChanged
{
    protected IPersonRepository Repository;

    public PeopleViewerViewModel()
    {
        var repositoryType = ConfigurationManager.AppSettings["RepositoryType"];

        switch (repositoryType)
        {
            case "Service": Repository = new ServiceRepository();
                break;
            case "SQL": Repository = new SQLRepository();
                break;
            case "CSV": Repository = new CSVRepository();
                break;
        }
    }
    ...
}
```

Should our Presentation Layer be responsible for this?

Scenario 2: Client-Side Caching

```
public class PeopleViewerViewModel : INotifyPropertyChanged
{
    protected IPersonRepository Repository;

    public PeopleViewerViewModel()
    {
        var repositoryType = ConfigurationManager.AppSettings["RepositoryType"];

        switch (repositoryType)
        {
            case "Service": Repository = new ServiceRepository();
                break;
            case "SQL": Repository = new SQLRepository();
                break;
            case "CSV": Repository = new CSVRepository();
                break;
        }
    }
    ...
}
```

Code gets much more complicated

Scenario 3: Unit Testing

```
public class PeopleViewerViewModel : INotifyPropertyChanged
{
    protected ServiceRepository Repository;

    public PeopleViewerViewModel()
    {
        Repository = new ServiceRepository();
    }
    ...
}
```

```
public class ServiceRepository
{
    PersonServiceClient _serviceProxy = new PersonServiceClient();
    ...
}
```

- **To test the View Model**
  - Must create a ServiceRepository
  - Must create a PersonServiceClient
  - Service must be running

The Bigger Question

- **Who should be responsible for the Repository?**

> *Single Responsibility Principle*
> *A class should have only one reason to change.*

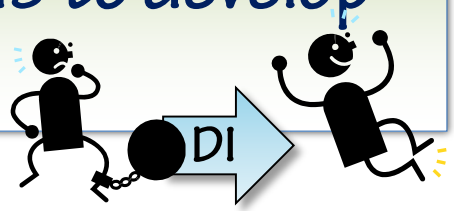
- S.O.L.I.D. Principles

- **The responsibility of the View Model is to control the Presentation.**
- **It should not also select which data store (Repository) to use.**
- **We should not need to change our View Model if we want to add a different Repository.**

The Solution

- **Loose Coupling will help us resolve these issues.**

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



Summary

- **What is Dependency Injection?**
- **Basic Patterns and Concepts**
- **The Problems of Tight-Coupling**
  - Difficult to Extend
  - Difficult to Test

- **Next Up: Adding Dependency Injection**
  Loose-Coupling with Constructor Injection