

# Take Your C# Skills to the Next Level

Jeremy Clark

[jeremybytes.com](http://jeremybytes.com)

[github.com/jeremybytes/sdd-2024](https://github.com/jeremybytes/sdd-2024)



Ignore the number of slides  
in this deck. These are  
primarily for your reference.

Most of our time will be spent in code.

# Schedule

- Class Hours 9:30 a.m. – 5:30 p.m.
- Break ??:00 a.m. – ??:?? a.m.
- Lunch 1:00 p.m. – 2:00 p.m.
- Break ??:00 p.m. – ??:?? p.m.

Additional breaks and Q&A throughout the day

All Times are British Summer Time

# Overview

- What we want from software
- Principles that help
- Tools to get there
  - Interfaces
  - Delegates
  - Dependency Injection
  - Unit Tests

# Topics (in no particular order)

- What & Why?
  - Interfaces
  - Delegates
  - Dependency Injection (DI)
  - Unit Tests
- Explicit Interface Implementation
- Interface Inheritance
- Interface Granularity
- DI Patterns
- Stable vs. Volatile Dependencies
- Read-only/init-only Properties
- Using a DI Container
- Injecting Primitives
- Adding a Cache
- Injecting Behavior
- Testing `DateTime.Now`
- Test Fakes and Mocks

# What We Want from Software

As Developers and Users



# As a user, I want software...

- That works
- That is delivered quickly
- That does what I need it to do
- That can be fixed quickly
- That can change as my needs change





# As a developer, I want software...

- With no bugs
- Easy to write
- Fulfills the use cases
- Easy to fix when there are bugs
- Easy to change when the use cases change



# We want the same things

## What users want

- That works
- That is delivered quickly
- That does what I need it to do
- That can be fixed quickly
- That can change as my needs change

## What developers want

- With no bugs
- Easy to write
- Fulfills the use cases
- Easy to fix when there are bugs
- Easy to change when the use cases change

# Helpful Practice

- S** • Single Responsibility Principle
- O** • Open/Closed Principle
- L** • Liskov Substitution Principle
- I** • Interface Segregation Principle
- D** • Dependency Inversion Principle

# “Best” Practices





## No “Best” Practices

Helpful practices are a good place to start, but don't use them if they don't fit your situation.




# Tools

- Interfaces
  - Add “seams” to code
  - Modification points
  - Interception points
  - Testing points
- Delegates
  - Inject custom behavior
- Dependency Injection
  - Assemble the pieces
  - Containers automate assembly
- Unit Tests
  - Proof code works
  - Proof that I didn't break something

# Interfaces

Adding seams to code



An interface contains definitions for a group of related functionalities that a non-abstract class or struct must implement.

<https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/types/interfaces>





An interface describes  
a set of capabilities  
on an object.

“I have these functions.”

# Interface

Defines a contract

Implement any number  
of interfaces

Limited implementation code

No automatic properties

Properties  
Methods  
Events  
Indexers

# Abstract Class

Shared Implementation

Inherit from a single base class

Unconstrained implementation  
code

Can have automatic  
properties

Properties	Fields
Methods	Constructors
Events	Destructors
Indexers	



## Recommendation

Program to an abstraction  
rather than a concrete type.



## Recommendation

Program to an **interface**  
rather than a **concrete class**.

# Various Data Sources

Microsoft SQL Server

MongoDB

CSV

WebAPI

Oracle

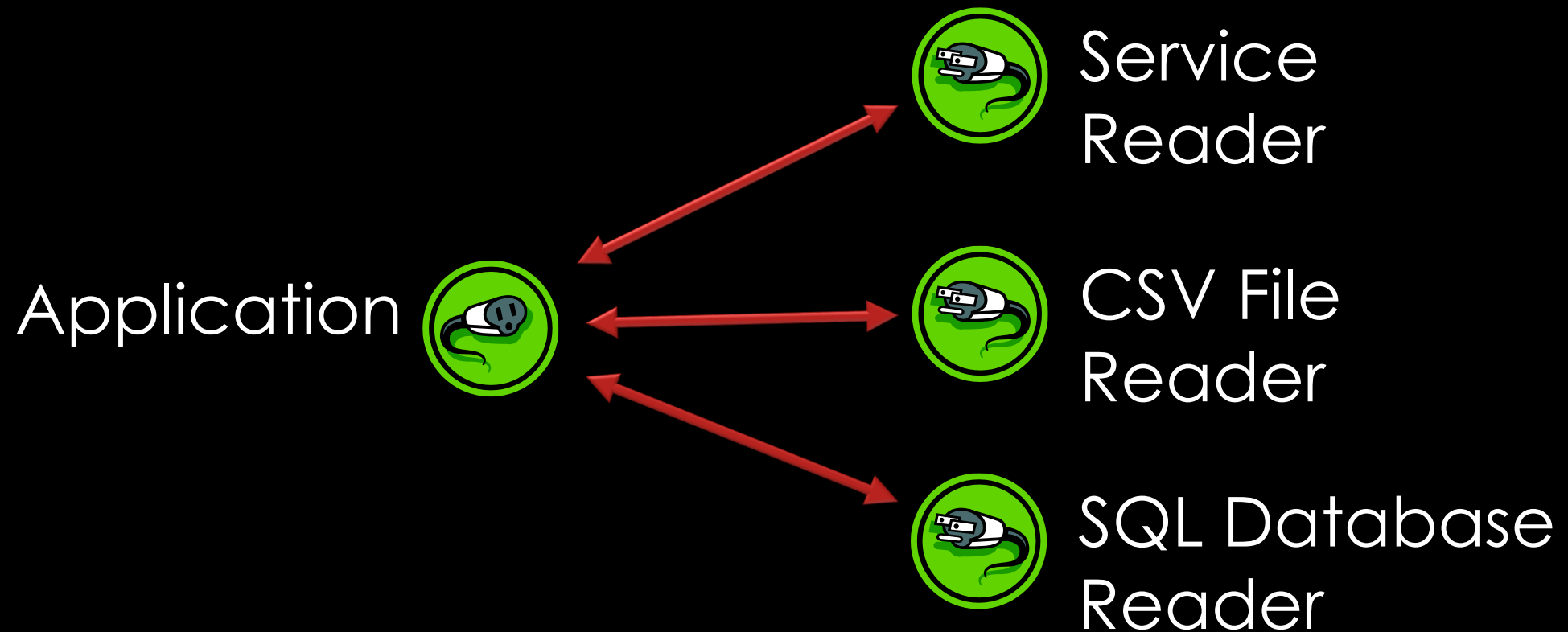
Amazon RDS

JSON

Azure Cosmos DB

Hadoop

# Pluggable Data Readers



# Data Reader Interface

```
public interface IPersonReader
{
    Task<IReadOnlyCollection<Person>> GetPeople();
    Task<Person?> GetPerson(int id);
}
```



# Interfaces and Flexible Code

**Resilience in the  
face of change**

**Insulation from  
implementation  
details**

# Dynamic Factory

- Check configuration
- Create an assembly load context
- Load the assembly
- Look for the type
- Create the data reader
- Return the data reader



## Other Benefits

Interfaces help us isolate code for easier **unit testing**.



## Other Benefits

Interfaces can make  
dependency injection easier.

# Interfaces: What & Why?

- An interface describes a set of capabilities of an object.
- Program to an abstraction (interface) rather than a concrete type (class).
- Resilience in the face of change.
- Insulation from implementation details.
- Easier unit testing.
- Easier dependency injection.

# Explicit Implementation

An explicitly implemented member **belongs to the interface** rather than the class.

# Class with No Interface

## Declaration

```
public class Catalog
{
    public string Save()
    {
        return "Catalog Save";
    }
}
```

## Usage

```
Catalog catalog = new();
string result = catalog.Save();
// result = "Catalog Save"
```



# Standard Interface Implementation

## Declaration

```
public interface ISaveable
{
    public string Save();
}
```

```
public class Catalog : ISaveable
{
    public string Save()
    {
        return "Catalog Save";
    }
}
```

## Usage

```
Catalog catalog = new();
string result = catalog.Save();
// result = "Catalog Save"
```

```
ISaveable saveable = new Catalog();
result = saveable.Save();
// result = "Catalog Save"
```

# Explicit Implementation

## Declaration

```
public interface ISaveable
{
    public string Save()
}

public class Catalog : ISaveable
{
    public string Save()
    {
        return "Catalog Save";
    }

    string ISaveable.Save()
    {
        return "Interface Save";
    }
}
```

## Usage

```
Catalog catalog = new();
string result = catalog.Save();
// result = "Catalog Save"
```

```
ISaveable saveable = new Catalog();
result = saveable.Save();
// result = "Interface Save"
```

```
result = ((ISaveable)catalog).Save();
// result = "Interface Save"
```

# Explicit Implementation

## Declaration

```
public interface ISaveable
{
    public string Save()
}

public class Catalog : ISaveable
{
    // Save() method deleted
    string ISaveable.Save()
    {
        return "Interface Save";
    }
}
```

## Usage

```
Catalog catalog = new();
string result = catalog.Save();
// **COMPLIER ERROR**
```

```
ISaveable saveable = new Catalog();
result = saveable.Save();
// result = "Interface Save"
```

```
saveable = (ISaveable)catalog;
result = saveable.Save();
// result = "Interface Save"
```

# Interface Inheritance

```
public interface IEnumerable<T> : IEnumerable
```

- `IEnumerable<T>` inherits `IEnumerable`
- When a class implements `IEnumerable<T>`, it must also implement `IEnumerable`

# IEnumerable<T> / IEnumerable

```
public interface IEnumerable<T> : IEnumerable
{
    IEnumerator<T> GetEnumerator();
}

public interface IEnumerable
{
    IEnumerator GetEnumerator();
}
```

When a class implements IEnumerable<T>, it must also implement IEnumerable

# IEnumerable<T> / IEnumerable

```
public class FibonacciSequence : IEnumerable<double>
{
    public IEnumerator<double> GetEnumerator()
    { // implementation }

    public IEnumerator GetEnumerator()
    { // implementation }
}
```

**NOT ALLOWED**

Methods cannot be overloaded  
only on different return types.

# IEnumerable<T> / IEnumerable

```
public class FibonacciSequence : IEnumerable<double>
{
    public IEnumerator<double> GetEnumerator()
    { // implementation }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return this.GetEnumerator();
    }
}
```

SOLUTION: Explicit Implementation.



# Interface Segregation Principle

```
public class List<T> : IList<T>, IList,  
    ICollection<T>, ICollection,  
    IEnumerable<T>, IEnumerable,  
    IReadOnlyCollection<T>, IReadOnlyList<T>
```

Clients should not be forced to depend upon methods that they do not use.

**Interfaces belong to clients,**  
not hierarchies.

# Interface Segregation Principle

```
public class List<T> : IList<T>, IList,  
    ICollection<T>, ICollection,  
    IEnumerable<T>, IEnumerable,  
    IReadOnlyCollection<T>, IReadOnlyList<T>
```

We should have **granular interfaces** that only include the members that a particular function needs.

# List<T> Interfaces

```
public class List<T> : IList<T>, IList,  
    ICollection<T>, ICollection,  
    IEnumerable<T>, IEnumerable,  
    IReadOnlyCollection<T>, IReadOnlyList<T>
```

**IEnumerable<T>**

GetEnumerator()

**IEnumerable**

GetEnumerator()

# List<T> Interfaces

```
public class List<T> : IList<T>, IList,  
ICollection<T>, ICollection,  
IEnumerable<T>, IEnumerable,  
IReadOnlyCollection<T>, IReadOnlyList<T>
```

## ICollection<T>

Count  
IsReadOnly  
Add()  
Clear()  
Contains()  
CopyTo()  
Remove()

Plus  
Everything in  
IEnumerable<T>  
and  
IEnumerable

# List<T> Interfaces

```
public class List<T> : IList<T>, IList,  
    ICollection<T>, ICollection,  
    IEnumerable<T>, IEnumerable,  
    IReadOnlyCollection<T>, IReadOnlyList<T>
```

## **IList<T>**

Item / Indexer  
IndexOf()  
Insert()  
RemoveAt()

Plus  
Everything in  
ICollection<T>,  
IEnumerable<T>,  
and  
IEnumerable

# Granular Interfaces

- **If We Need to**

- Iterate over a Collection / Sequence
- Data Bind to a List Control
- Use LINQ functions



**IEnumerable<T>**

- **If We Need To**

- Add/Remove Items in a Collection
- Count Items in a Collection
- Clear a Collection



**ICollection<T>**

- **If We Need To**

- Control the Order Items in a Collection
- Get an Item by the Index



**IList<T>**

# Single Responsibility Principle

A class should have only **one reason to change**.

Gather together the things that **change for the same reasons**. Separate those things that change for different reasons.



# Open-Closed Principle

Software entities should be **open for extension**, but **closed for modification**.

Ex. Injecting behavior through a delegate or other entity.

# Liskov Substitution Principle

Substitutability: an object (such as a class) may be **replaced by a sub-object** without breaking the program.

Violation example: Rectangle vs. Square

# Interface Segregation Principle

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

# Dependency Inversion Principle

High-level modules should not import anything from low-level modules; both should **depend on abstractions**.

Abstractions should not depend on details. Details (concrete implementations) should **depend on abstractions**.

# Delegates

Inserting behavior



# What Is A Delegate?

## Definition

A type that defines a method signature

# Why Delegates?

- Decoupling Code
- Methods as Parameters
- Multicasting Support
- Callbacks and Event Handlers
- LINQ
- ASP.NET Core Minimal APIs



# Single Responsibility Principle

A class should have only **one reason to change**.

Gather together the things that **change for the same reasons**. Separate those things that change for different reasons.

# Open-Closed Principle

Software entities should be **open for extension**, but **closed for modification**.

Ex. Injecting behavior through a delegate or other entity.

# Dependency Injection

## Assembling the pieces

An abstract graphic at the top of the slide featuring a series of overlapping, wavy bands of color. From left to right, the colors transition from a bright yellow-orange to a deep red, then to a dark green, and finally to a light blue/cyan. The waves create a sense of movement and depth.

# Dependency Injection

The fine art of making things someone else's problem.

# Typical Introduction

```
private void BuildMainWindow()
{
    var builder = new ContainerBuilder();
    builder.RegisterType<SQLReader>().As<IPersonReader>()
        .SingleInstance();
    builder.RegisterSource(
        new AnyConcreteTypeNotAlreadyRegisteredSource());
    IContainer Container = builder.Build();
    Application.Current.MainWindow =
        Container.Resolve<PeopleViewerWindow>();
}
```

# 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?

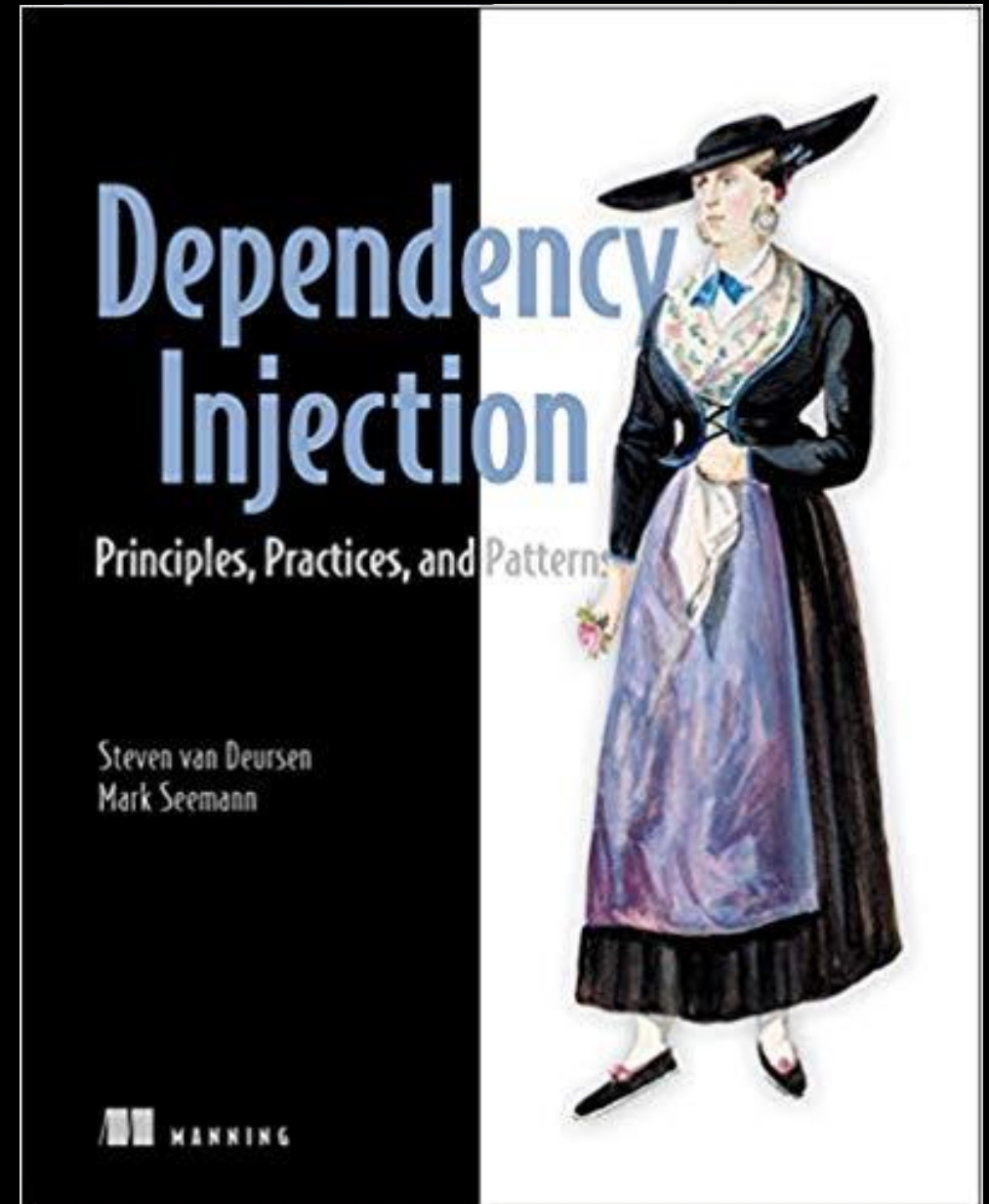
- Dependency Injection is a set of software design principles and patterns that enable us to develop **loosely coupled code**.

- Mark Seemann

# Dependency Injection

Principles, Practices, and Patterns

- Mark Seemann
- Steven van Deursen



# Primary Benefits

- Extensibility
  - Parallel Development
  - Maintainability
  - Testability
  - Late Binding
- 
- Adherence to S.O.L.I.D. Design Principles.



## 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 – Late Binding

Services can be swapped with other services **without recompiling** code.

# 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
    - Autofac
    - Ninject
  - Frameworks w/ Containers
    - ASP.NET Core
    - Angular
    - Prism
- and many others

# Application Layers

## View

- PeopleViewerWindow

## View Model

- PeopleViewModel

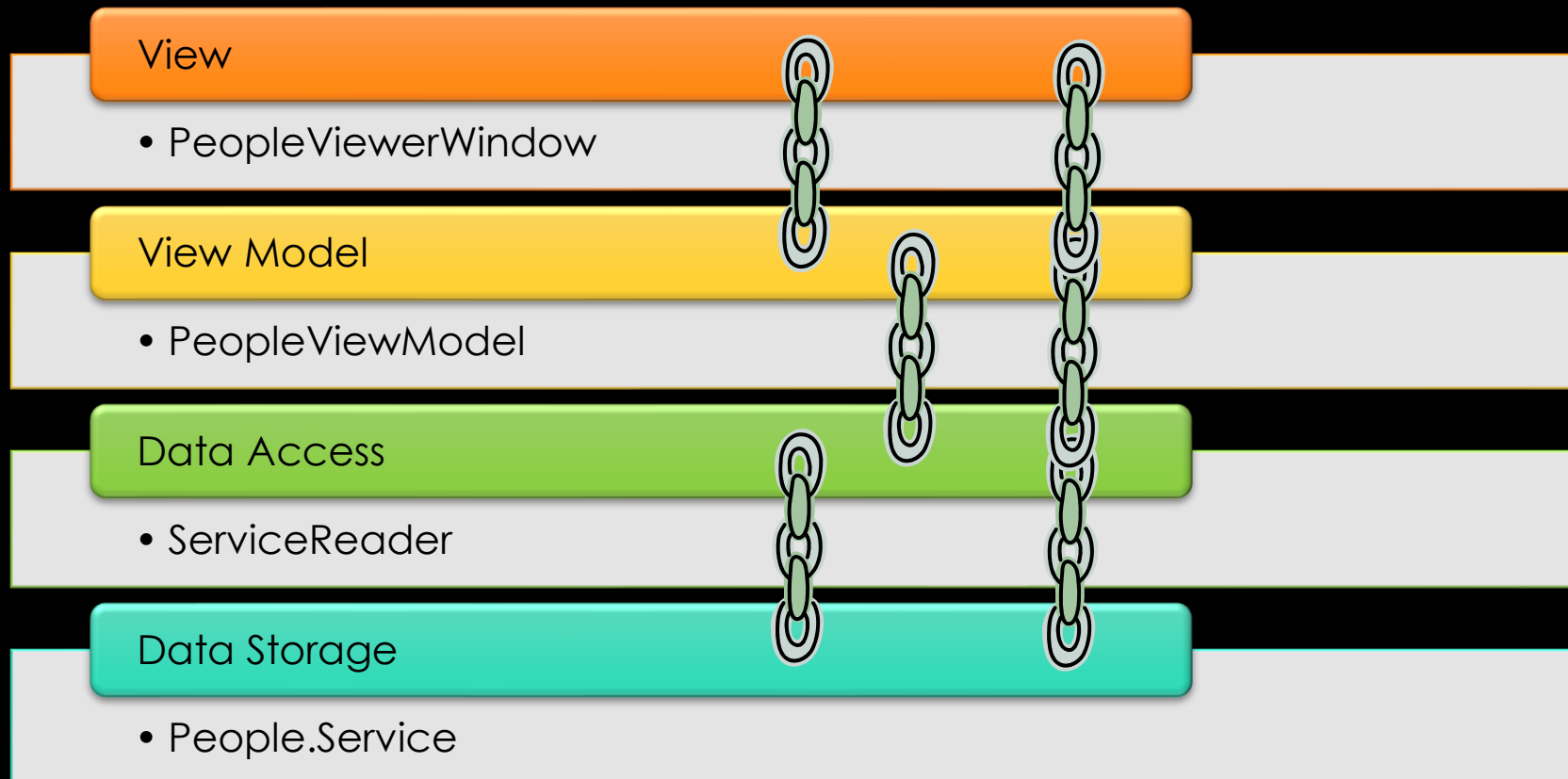
## Data Access

- ServiceReader

## Data Storage

- People.Service

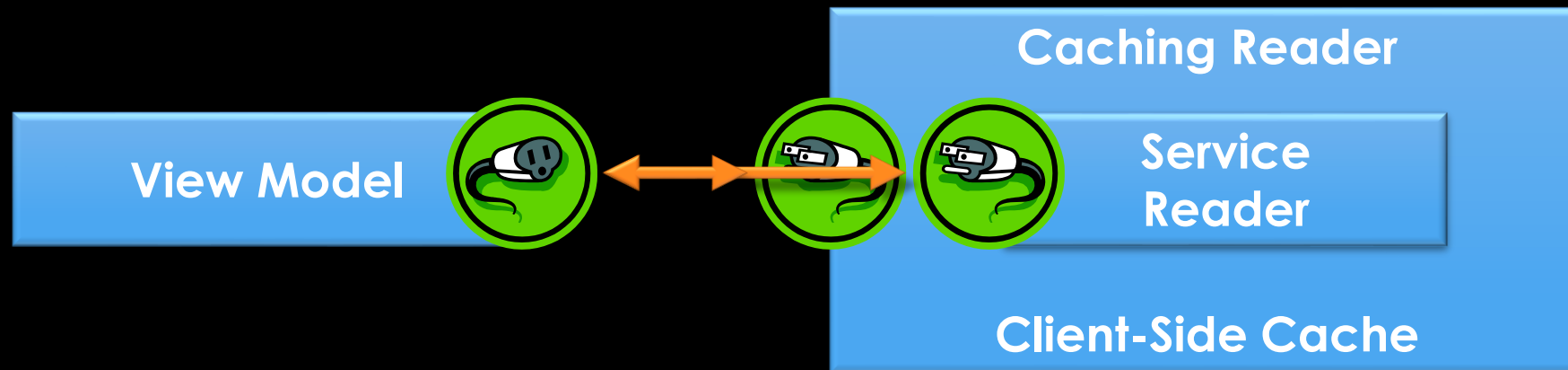
# Tight Coupling



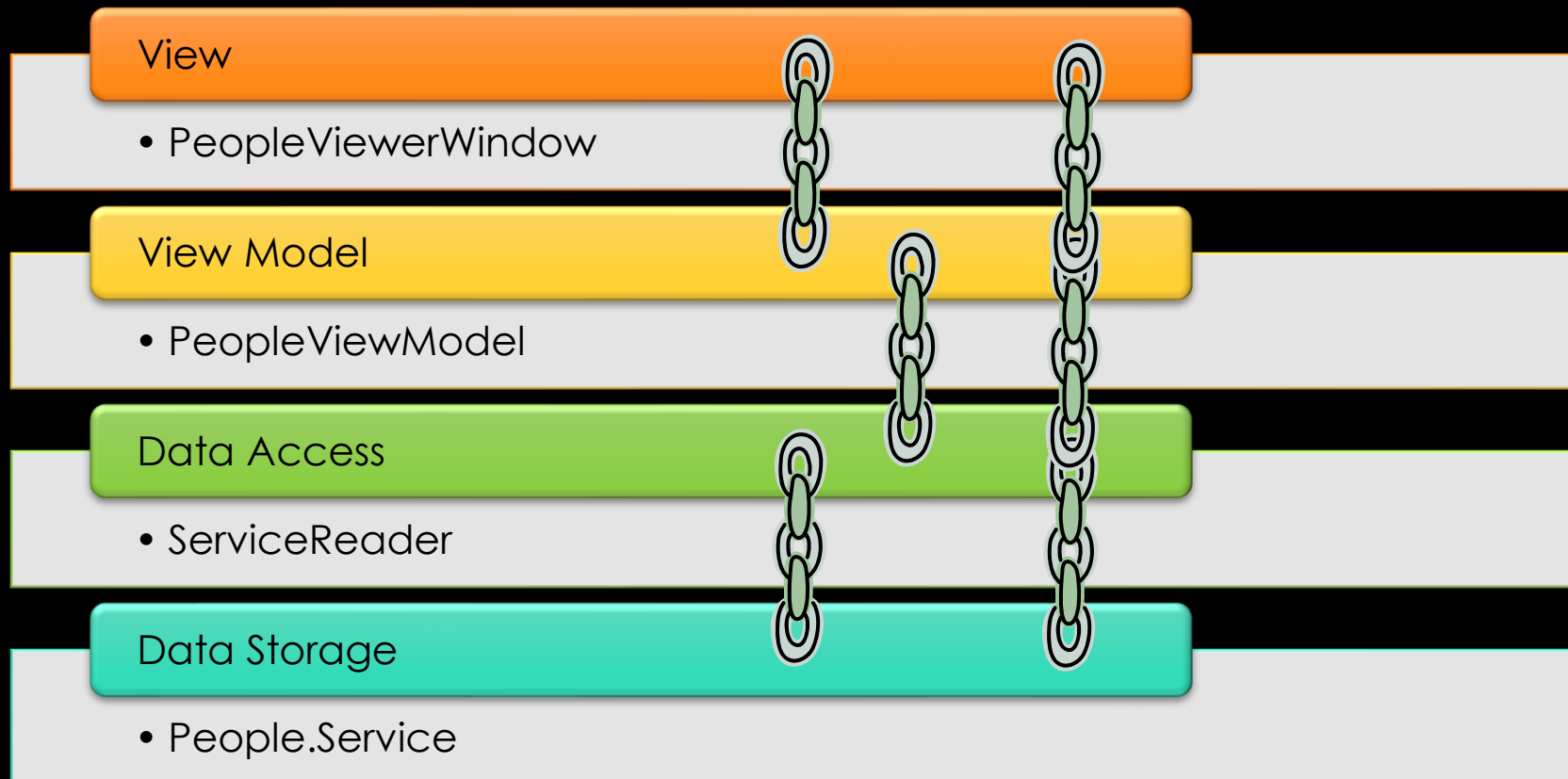


# Creating a Caching Reader

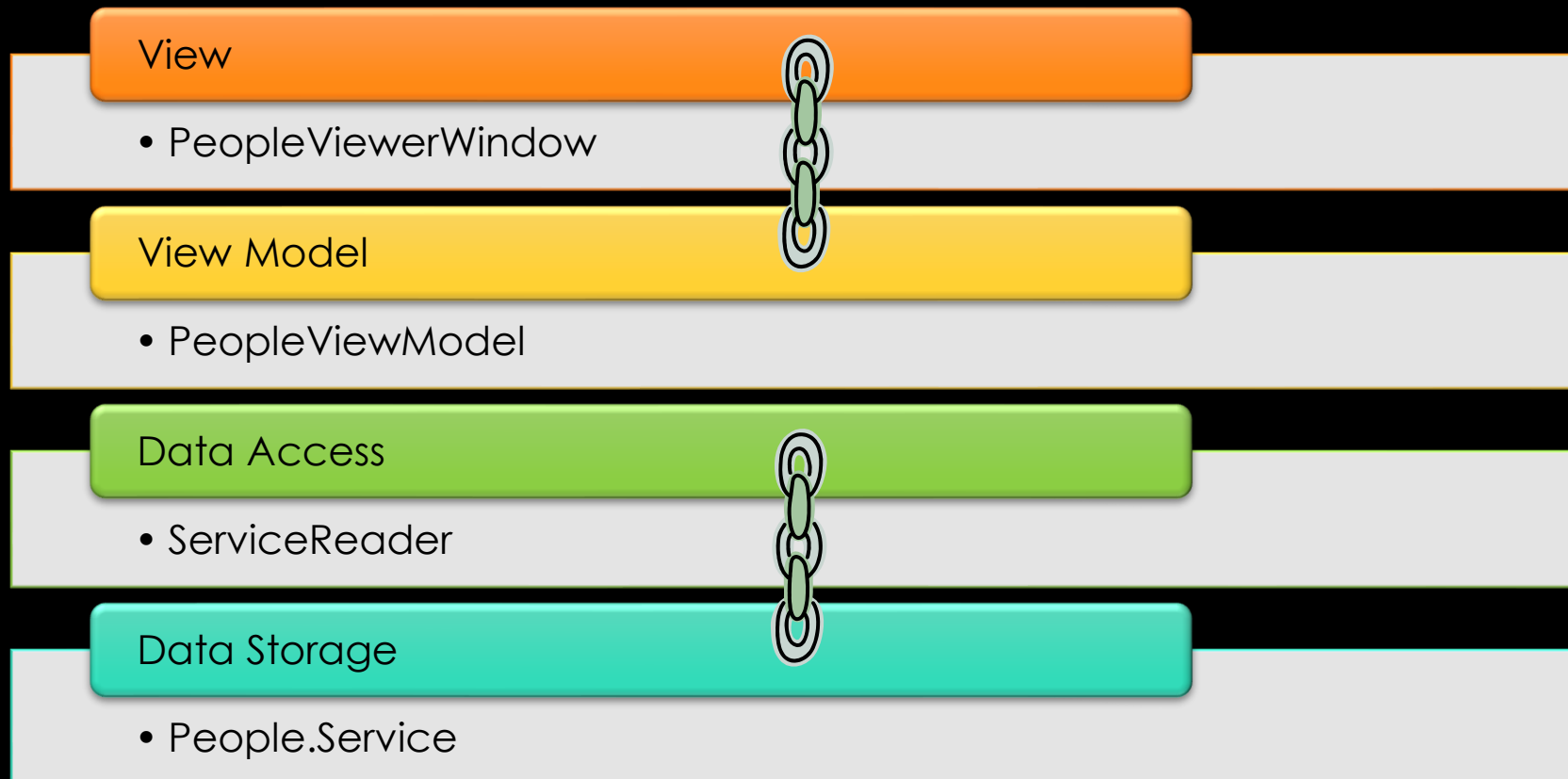
## The Decorator Pattern



# Loose(r) Coupling



# Loose(r) Coupling



# Primary Benefits

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



# 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

# Primary Constructors

- When using a primary constructor, the constructor parameter can either initialize a field (or property) or be used directly.
- When initializing a property, the parameter is given an “unspeakable” name (like a field for an automatic property).
- When used directly, the parameter is class-level and modifiable.





# 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, 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 3<sup>rd</sup> party library
- The dependency contains non-deterministic behavior
  - Random number generator
  - `DateTime.Now`



## Tips / Techniques

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

# Read-Only / init-Only Properties

- Properties marked as “readonly” or with “init” for a setter are settable only during object construction. 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.

# Unit Testing

Code faster

# Different Kinds of Tests

- Unit Testing
- Integration Testing
- Performance Testing
- Exploratory Test
- Penetration Testing
- User Acceptance Testing (UAT)

# What are Unit Tests?

A unit test is an automated piece of code that invokes a unit of work in the system and then checks a single assumption about the behavior of that unit of work.

*The Art of Unit Testing* by Roy Oshero



Non-Threatening  
Text Here





Threatening  
Text Here





# What are Unit Tests?

A unit test is an automated piece of code that invokes a unit of work in the system and then checks a single assumption about the behavior of that unit of work.

**automated piece of code**

**a unit of work**

**checks a single  
assumption**

*The Art of Unit Testing* by Roy Osherove

# Assertions

- The Assert class throws exceptions when the assertion fails
  - <https://learn.microsoft.com/en-us/dotnet/api/microsoft.visualstudio.testtools.unittesting.assert?view=visualstudiosdk-2022>
- NUnit provides a custom Assert class that contains additional functionality
  - <https://docs.nunit.org/articles/nunit/writing-tests/assertions/assertions.html>



# Benefits

- Confirming Functionality
- Checking Regression
- Pinpointing Bugs
- Documenting Functionality

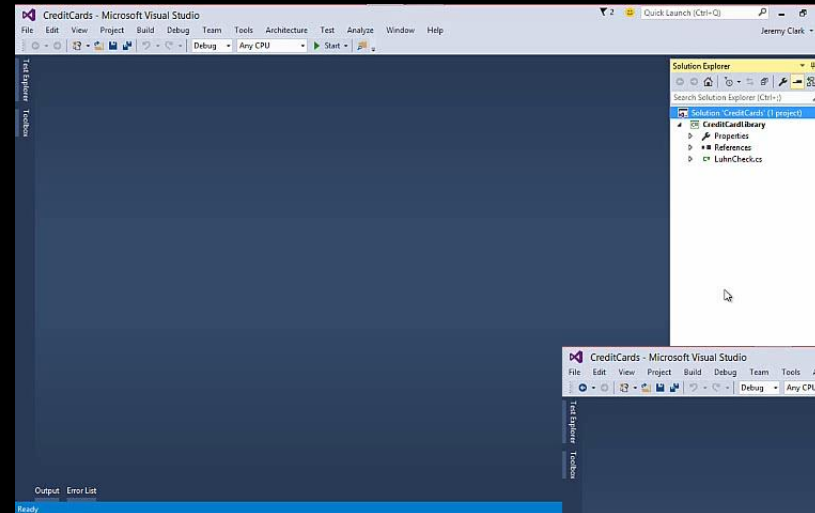
An abstract graphic at the top of the slide featuring a series of overlapping, wavy bands in shades of orange, red, yellow, and green, set against a black background.

# Confirming Functionality

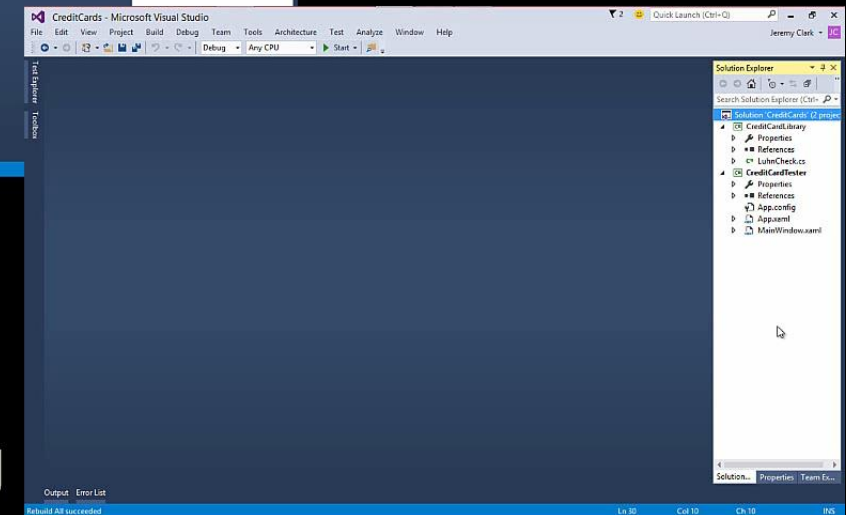
Unit Tests are ***proof*** that my code  
does what I ***think*** it does

# Build Time Comparison

# Test Application



# Unit Testing



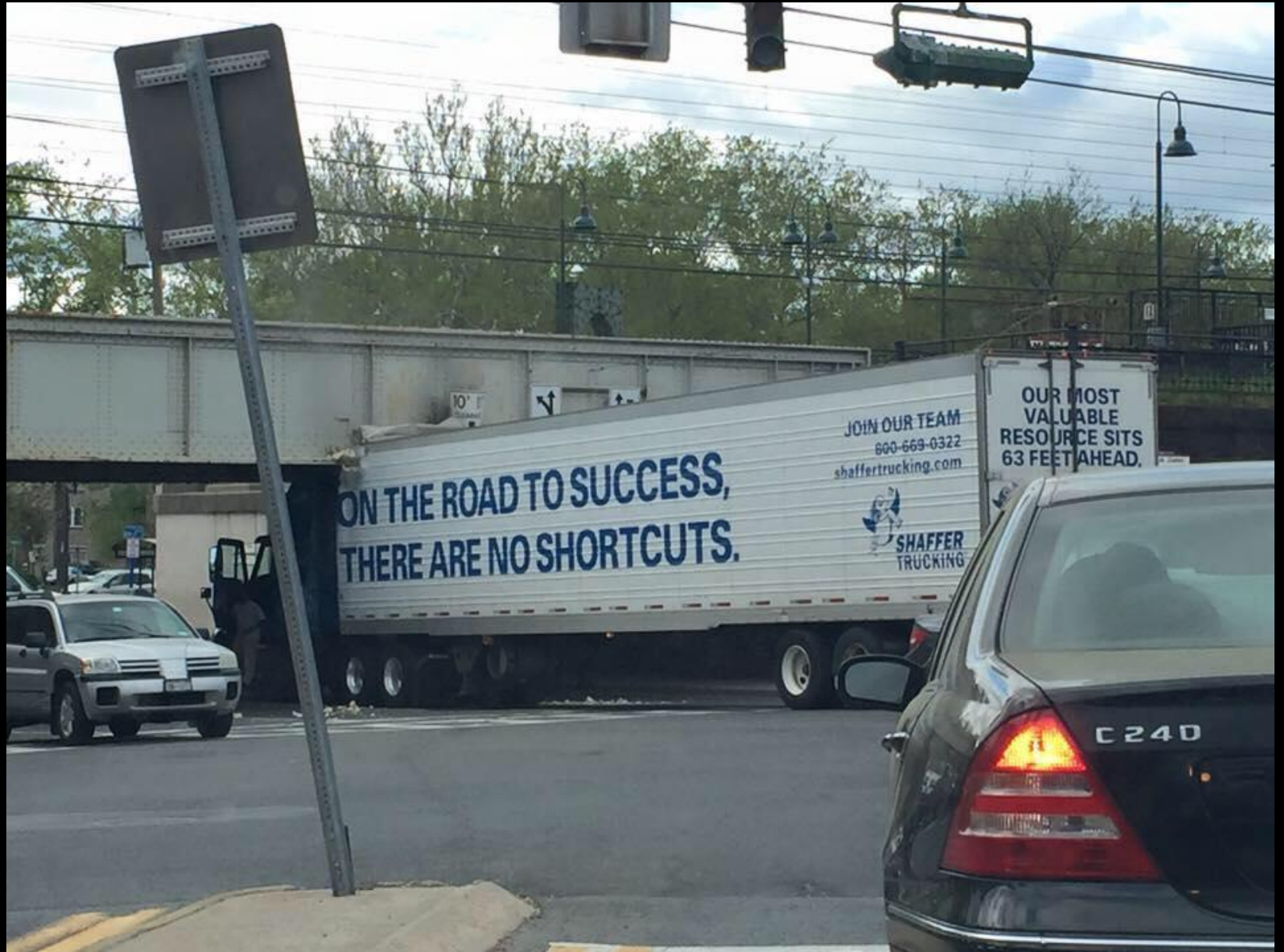
An abstract graphic at the top of the slide featuring a series of overlapping, wavy bands in shades of orange, red, yellow, and green, set against a black background.

# Disclaimer

We get these advantages when we are ***comfortable*** writing ***good*** tests.



# Realistic Expectations



# Checking Regression

The screenshot displays the Visual Studio IDE with the Test Explorer on the left and the code editor on the right. The Test Explorer shows 15 passed tests for the LuhnCheck library, including invalid and valid number checks. The code editor shows the implementation of the LuhnCheck class in LuhnCheck.cs, which includes a static method PassesLuhnCheck that calculates the checksum for a given card number.

**Test Explorer**

Streaming Video: Improving quality with unit tests and fakes

Run All | Run... | Playlist: All Tests

**Passed Tests (15)**

Test Name	Duration
PassesLuhnCheck_OnInvalidNumber_ReturnsFalse("-01233454567")	< 1 ms
PassesLuhnCheck_OnInvalidNumber_ReturnsFalse("123")	< 1 ms
PassesLuhnCheck_OnInvalidNumber_ReturnsFalse("7147894289")	8 ms
PassesLuhnCheck_OnInvalidNumber_ReturnsFalse("9876543210987654")	< 1 ms
PassesLuhnCheck_OnInvalidNumber_ReturnsFalse("abc")	< 1 ms
PassesLuhnCheck_OnValidNumber_ReturnsTrue("3530111333300000")	< 1 ms
PassesLuhnCheck_OnValidNumber_ReturnsTrue("3566002020360505")	< 1 ms
PassesLuhnCheck_OnValidNumber_ReturnsTrue("371449635398431")	< 1 ms
PassesLuhnCheck_OnValidNumber_ReturnsTrue("378282246310005")	< 1 ms
PassesLuhnCheck_OnValidNumber_ReturnsTrue("4012888888881881")	< 1 ms
PassesLuhnCheck_OnValidNumber_ReturnsTrue("4111111111111111")	< 1 ms
PassesLuhnCheck_OnValidNumber_ReturnsTrue("5105105105105100")	< 1 ms
PassesLuhnCheck_OnValidNumber_ReturnsTrue("5555555555554444")	< 1 ms
PassesLuhnCheck_OnValidNumber_ReturnsTrue("6011000990139424")	< 1 ms
PassesLuhnCheck_OnValidNumber_ReturnsTrue("6011111111111117")	< 1 ms

**Code Editor**

**LuhnCheck.cs**

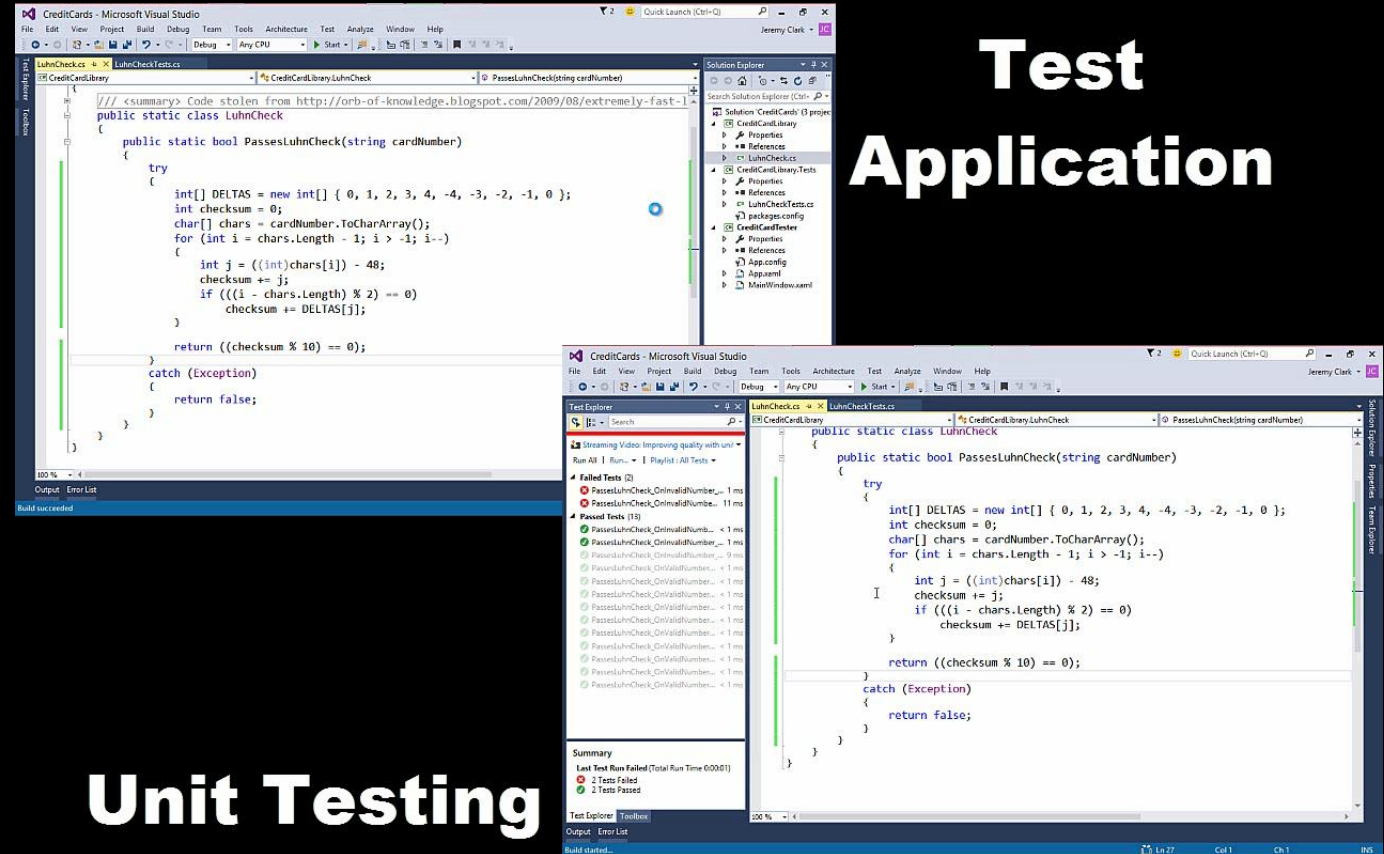
```
public static class LuhnCheck
{
    public static bool PassesLuhnCheck(string cardNumber)
    {
        try
        {
            int[] DELTAS = new int[] { 0, 1, 2, 3, 4, -4, -3, -2, -1 };
            int checksum = 0;
            char[] chars = cardNumber.ToCharArray();
            for (int i = chars.Length - 1; i > -1; i--)
            {
                int j = ((int)chars[i]) - 48;
                checksum += j;
                if (((i - chars.Length) % 2) == 0)
                    checksum += DELTAS[j];
            }

            return ((checksum % 10) == 0);
        }
        catch (Exception)
        {
            return false;
        }
    }
}
```



# Regression Comparison

# Test Application



# Unit Testing

# Pinpointing Bugs

The screenshot displays the Visual Studio IDE with the Test Explorer on the left and the source code for `CatalogViewModel.cs` on the right.

**Test Explorer:**

- Failed Tests (6):**
  - `CatalogViewModel_OnInitialization_ModelIsPopulated` (161 ms)
  - `ModelSelectedItems_AddToSelectionWithExistingPerson_SelectionIsUnchanged` (2 ms)
  - `ModelSelectedItems_AddToSelectionWithNewPerson_PersonAdded` (3 ms)
  - `ModelSelectedItems_RemoveFromSelectionWithExistingPerson_PersonRemoved` (1 ms)
  - `ModelSelectedItems_RemoveFromSelectionWithNewPerson_SelectionIsUnchan...` (1 ms)
  - `ModelSelectedPeople_OnClearSelection_IsEmpty` (1 ms)
- Passed Tests (13):**
  - `Catalog_FilterDoesNotInclude00s_00sRecordsIsNotIncluded` (< 1 ms)
  - `Catalog_FilterDoesNotInclude70s_70sRecordsIsNotIncluded` (1 ms)
  - `Catalog_FilterIncludes00s_00sRecordsIsIncluded` (< 1 ms)
  - `Catalog_FilterIncludes70s_70sRecordsIsIncluded` (1 ms)
  - `CatalogService_OnRefreshAndCacheExpired_ServicesCalledTwice` (31 ms)
  - `CatalogService_OnRefreshAndCacheNotExpired_ServicesCalledOnce` (1 sec)
  - `CatalogViewModel_OnInitialization_CatalogIsPopulated` (1 ms)
  - `CatalogViewModel_OnInitializationAndCurrentOrderMissing_ThrowsException` (1 ms)
  - `CatalogViewModel_OnInitializationAndPersonServiceMissing_ThrowsException` (< 1 ms)
  - `CatalogViewModel_OnInitializationWithNoServiceException_DoesNotThrowE...` (198 ms)
  - `CatalogViewModel_OnInitializationWithServiceException_ThrowsExceptionOnC...` (2 ms)
  - `Filters_OnRefreshAndCacheExpired_AreResetToDefaults` (1 ms)

**Summary:**

- Last Test Run Failed** (Total Run Time 0:00:03)
- 6 Tests Failed
- 13 Tests Passed

**CatalogViewModel.cs:**

```
#region Methods

public void Initialize()
{
    _service = GetServiceFromContainer();
    GetModelFromContainer();

    RefreshCatalog();
}

private CatalogOrder GetModelFromContainer()
{
    if (!_container.IsRegistered<CatalogOrder>("CurrentOrder"))
        throw new MissingFieldException(
            "CurrentOrder is not available from the DI Containe");
    return _container.Resolve<CatalogOrder>("CurrentOrder");
}

private IPersonService GetServiceFromContainer()
{
    if (!_container.IsRegistered<IPersonService>())
        throw new MissingFieldException(
            "IPersonService is not available from the DI Contai");
    return _container.Resolve<IPersonService>();
}

public void RefreshCatalog()...

private void RefreshCatalogFromService()...
```

# Documenting Functionality

- ✓ Catalog\_FilterDoesNotInclude00s\_00sRecordsIsNotIncluded
- ✓ Catalog\_FilterDoesNotInclude70s\_70sRecordsIsNotIncluded
- ✓ Catalog\_FilterIncludes00s\_00sRecordsIsIncluded
- ✓ Catalog\_FilterIncludes70s\_70sRecordsIsIncluded
- ✓ CatalogService\_OnRefreshAndCacheExpired\_ServicesCalledTwice
- ✓ CatalogService\_OnRefreshAndCacheNotExpired\_ServicesCalledOnce
- ✓ CatalogViewModel\_OnInitialization\_CatalogsPopulated
- ✓ CatalogViewModel\_OnInitialization\_ModelsPopulated
- ✓ CatalogViewModel\_OnInitializationAndCurrentOrderMissing\_ThrowsException
- ✓ CatalogViewModel\_OnInitializationAndPersonServiceMissing\_ThrowsException

- ✓ Filters\_OnRefreshAndCacheExpired\_AreResetToDefaults
- ✓ Filters\_OnRefreshAndCacheNotExpired\_AreResetToDefaults

- ✓ ModelSelectedItem\_AddToSelectionWithExistingPerson\_SelectionIsUnchanged
- ✓ ModelSelectedItem\_AddToSelectionWithNewPerson\_PersonAdded

An abstract graphic at the top of the slide featuring a series of overlapping, wavy bands in shades of orange, red, yellow, and green, set against a black background.

# Disclaimer

We get these advantages when we are ***comfortable*** writing ***good*** tests.

# Good Unit Tests

- Maintainable
- Dependable
- Runnable

# Qualities of a Good Test

## Maintainable

- Not Tricky
- Easy to Read
- Easy to Write
- Well-Named

## Dependable

- Consistent Results
- Isolated
- Continued Relevance
- Tests the Right Things

## Runnable

- FAST



# Michael C. Feathers on Speed

“A unit test that takes  $1/10^{\text{th}}$  of a second to run is a slow unit test.”

“Unit tests run fast. If they don't run fast, they aren't unit tests.”

*Working Effectively with Legacy Code* by Michael C. Feathers

# Qualities of a Good Test

## Maintainable

- Not Tricky
- Easy to Read
- Easy to Write
- Well-Named

## Dependable

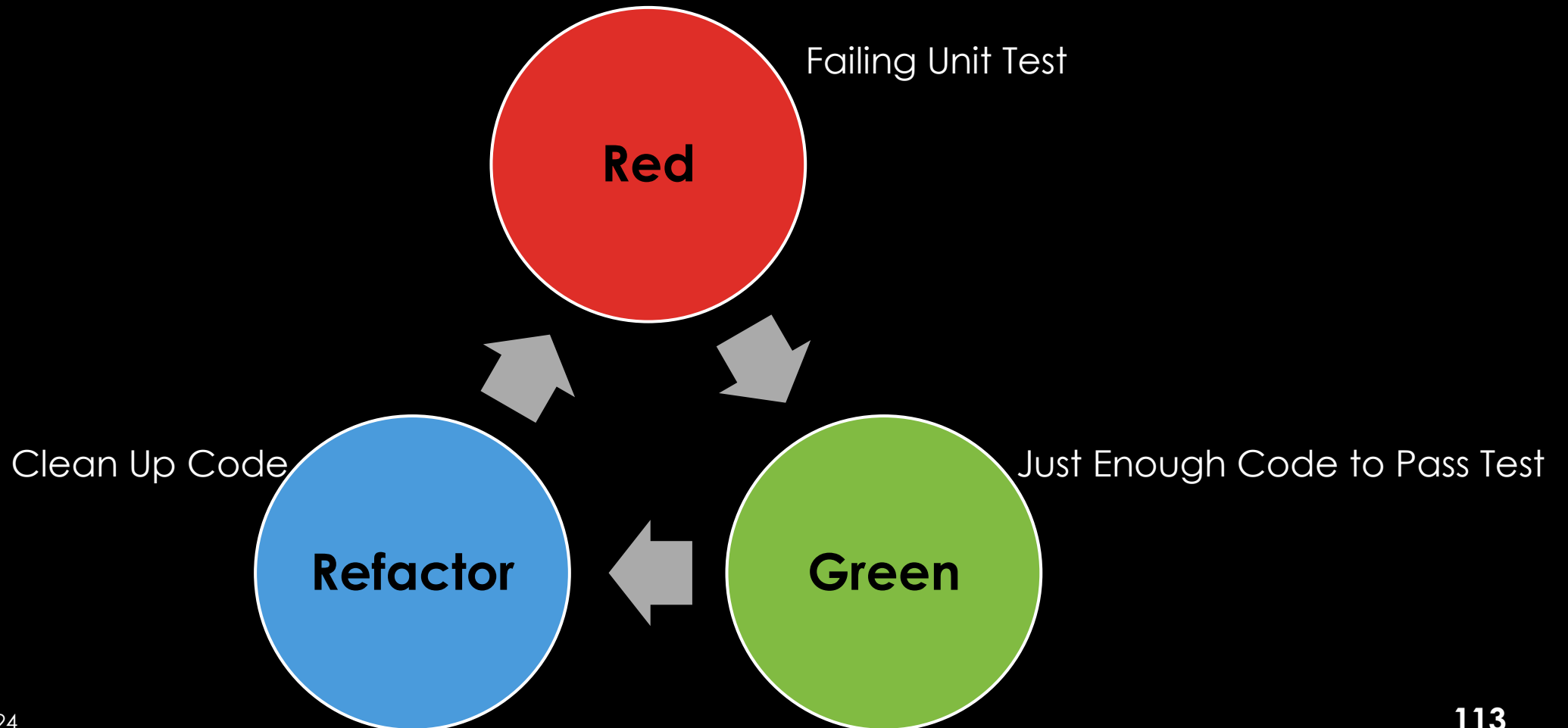
- Consistent Results
- Isolated
- Continued Relevance
- Tests the Right Things

## Runnable

- FAST
- Single Click
- Repeatable
- Failure Points to the Problem



# Test Driven Development



# Fizz Buzz

- Print Numbers 1 to 100
- If divisible by 3  
replace with “Fizz”
- If divisible by 5  
replace with “Buzz”
- If divisible by 3 and 5  
replace with “FizzBuzz”

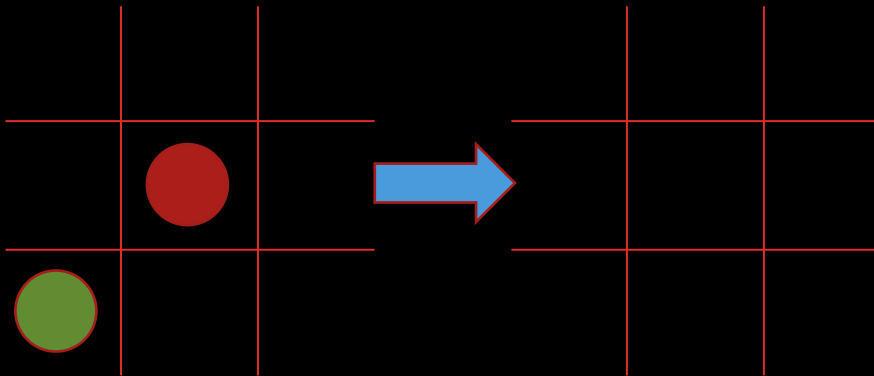
- |        |            |
|--------|------------|
| • 1    | • BUZZ     |
| • 2    | • 11       |
| • Fizz | • Fizz     |
| • 4    | • 13       |
| • BUZZ | • 14       |
| • Fizz | • FizzBUZZ |
| • 7    | • 16       |
| • 8    | • 17       |
| • Fizz | • Fizz     |

# Conway's Game of Life

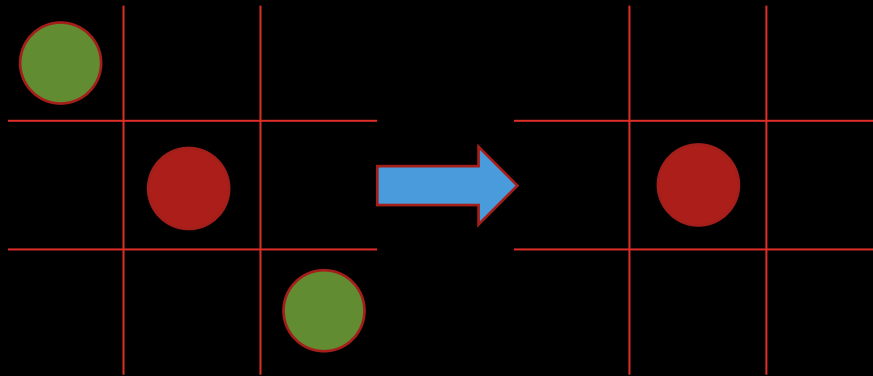
- Any live cell with fewer than two live neighbours dies.
- Any live cell with two or three live neighbours lives.
- Any live cell with more than three live neighbours dies.
- Any dead cell with exactly three live neighbours becomes a live cell.

# Conway's Game of Life

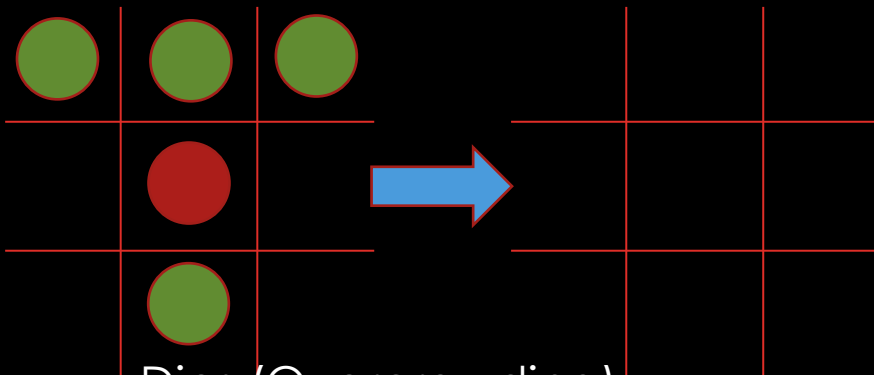
Dies (Loneliness)



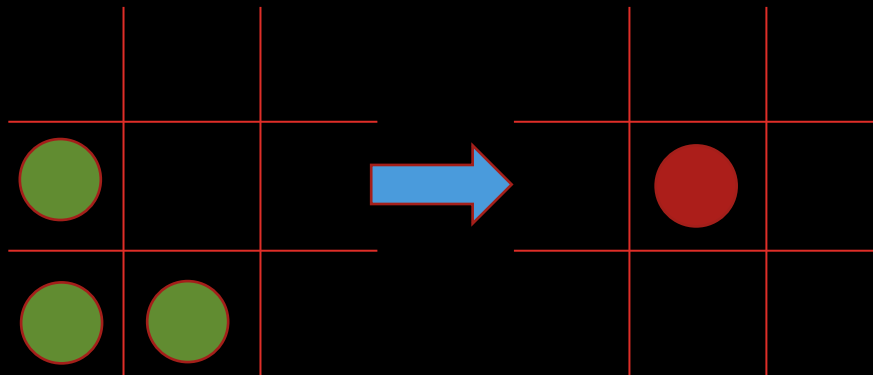
Lives (Happy)



Dies (Overcrowding)



Lives (Reproduction)



# Code Coverage

100% Code Coverage  
is not a guarantee





# Conversations about Code Coverage

“What parts of your application are  
okay ***not*** to test?”



# The Stahl Standard

“What parts of your application do  
your users ***not*** care about?”

-Barry Stahl

Twitter: @bsstahl    <http://www.cognitiveinheritance.com/>



# Know the Goals

- Don't do the right thing for the wrong reason.
- Unit testing will not fix bad development practices.



<http://www.jenders.com/2012/01/08/thief-almost-caught-on-camera-stealing-thin-lg-television/>





## Martin Fowler on Fear

“Don’t let the fear that testing  
can’t catch ***all*** bugs  
stop you from writing the tests  
that will catch ***most*** bugs.”

*Refactoring* by Martin Fowler et al.

# Handling Dependencies

- Create Interfaces to add “seams” to our code
- Use Dependency Injection for loose-coupling
- Use Mocking to inject dependencies for testing

# Mocking

- Create “Placeholder” Objects
  - In-Memory
  - Only Implement Behavior We Care About
- Mocking Frameworks
  - NSubstitute
  - Moq

# Moq

- Available in NuGet
- Documentation
  - <https://github.com/Moq/moq4/wiki/Quickstart>

# Simple Setup

```
public IPersonRepository GetTestRepository()
{
    var people = new List<Person>()
    {
        new Person() {FirstName = "John", LastName = "Smith",
            Rating = 7, StartDate = new DateTime(2000, 10, 1)},
        new Person() {FirstName = "Mary", LastName = "Thomas",
            Rating = 9, StartDate = new DateTime(1971, 7, 23)},
    };

    var repoMock = new Mock<IPersonRepository>();
    repoMock.Setup(r => r.GetPeople()).Returns(people);
    return repoMock.Object;
}
```

# Setup with Parameters

```
public IPersonService GetTestService()
{
    var people = new List<Person>()
    {
        new Person() {FirstName = "John", LastName = "Smith",
            Rating = 7, StartDate = new DateTime(2000, 10, 1)},
        new Person() {FirstName = "Mary", LastName = "Thomas",
            Rating = 9, StartDate = new DateTime(1971, 7, 23)},
    };

    var svcMock = new Mock<IPersonService>();
    svcMock.Setup(s => s.GetPeople()).Returns(people.ToArray());
    svcMock.Setup(s => s.GetPerson(It.IsAny<string>()))
        .Returns((string n) => people.FirstOrDefault(p => p.LastName == n));
    return svcMock.Object;
}
```

# Setup vs. Factory Methods

- Setup Methods in unit tests are run automatically before any test is invoked.
- Setup Methods can be used for object initialization.

## HOWEVER

- Factory Methods are not automatically run.
- Factory Methods are explicitly called within the test.
- Readability of tests can be increased by using Factory Methods instead of Setup Methods.



# Sample with Setup Method

```
IPersonRepository _repository;

[TestInitialize]
public void Setup()
{
    var people = new List<Person>()
    {
        new Person() {FirstName = "John", LastName = "Smith",
            Rating = 7, StartDate = DateTime.Parse("10/01/2000")},
        new Person() {FirstName = "Mary", LastName = "Thomas",
            Rating = 9, StartDate = DateTime.Parse("07/23/1971")},
    };

    var repoMock = new Mock<IPersonRepository>();
    repoMock.Setup(r => r.GetPeople()).Returns(people);
    _repository = repoMock.Object;
}
```

```
[TestMethod]
public void People_OnRefreshCommand_IsPopulated()
{
    // Arrange
    var vm = new MainWindowViewModel(_repository);

    // Act
    vm.RefreshPeopleCommand.Execute(null);

    // Assert
    Assert.IsNotNull(vm.People);
    Assert.AreEqual(2, vm.People.Count());
}
```

# Sample with Factory Method

```
public IPersonRepository GetRepositoryStub()
{
    var people = new List<Person>()
    {
        new Person() {FirstName = "John", LastName = "Smith",
            Rating = 7, StartDate = DateTime.Parse("10/01/2000")},
        new Person() {FirstName = "Mary", LastName = "Thomas",
            Rating = 9, StartDate = DateTime.Parse("07/23/1971")},
    };

    var repoMock = new Mock<IPersonRepository>();
    repoMock.Setup(r => r.GetPeople()).Returns(people);
    return repoMock.Object;
}
```

```
[TestMethod]
public void People_OnRefreshCommand_IsPopulated()
{
    // Arrange
    IPersonRepository repository = GetRepositoryStub();
    var vm = new MainWindowViewModel(repository);

    // Act
    vm.RefreshPeopleCommand.Execute(null);

    // Assert
    Assert.IsNotNull(vm.People);
    Assert.AreEqual(2, vm.People.Count());
}
```

# NUnit Parameterization

[Values()]

- The Values Attribute can be applied to parameters directly for individual values.
  - <https://docs.nunit.org/articles/nunit/writing-tests/attributes/values.html>

# [Values()] Attribute

```
[Test]  
public void FizzBuzzWhenDivisibleBy3_ReturnsFizz(  
    [Values(3, 6, 9, 12, 18)] int input)...
```

# NUnit Parameterization

## [Range()]

- The Range Attribute can be applied to parameters directly to test for a range of values.
  - <https://docs.nunit.org/articles/nunit/writing-tests/attributes/range.html>

# [Range()] Attribute

```
[Test]  
public void LiveCell_MoreThan3LiveNeighbors_Dies(  
    [Range(4, 8)] int liveNeighbors)...
```

# NUnit Parameterization

## [TestCase()]

- The [TestCase()] attribute can be applied to a test method to submit values for parameters.
  - <https://docs.nunit.org/articles/nunit/writing-tests/attributes/testcase.html>



# [TestCase()] Attribute

```
[TestCase("6011000990139424")]  
[TestCase("3530111333300000")]  
[TestCase("3566002020360505")]  
public void PassesLuhnCheck_OnValidNumber_ReturnsTrue(string cardNumber)...
```

# Testing for Exceptions

- Manual Testing:
  - Create method with try/catch block.
  - “Assert.Fail()” if no exception is thrown.
  - Catch block can check exception for specific type/properties.

# Testing for Exceptions

- NUnit Testing
  - Use “Assert.Throws()” method to check a block of code.
  - Return value is the exception and can be checked for specific properties.
  - Alternately, use the “IResolveConstraint” specific to NUnit.
- <https://docs.nunit.org/articles/nunit/writing-tests/assertions/classic-assertions/Assert.Throws.html>



Thank You!

Jeremy Clark

- [jeremybytes.com](https://jeremybytes.com)
- [youtube.com/jeremybytes](https://youtube.com/jeremybytes)
- [github.com/jeremybytes/sdd-2024](https://github.com/jeremybytes/sdd-2024)