C# Interfaces

INTRODUCING INTERFACES

Jeremy Clark

DEVELOPER BETTERER

@jeremybytes www.jeremybytes.com

Why Interfaces?

Maintain

Extend

Test

Trying to Learn Interfaces

Interfaces?

IFoo and bar class

NO!

Much reading

Conversations

This is awesome!

Our Roadmap

What?

Definition and technical bits

Why?

Maintain, extend, test

How?

Create and implement

Where?

Practical bits, dependency injection, and mocking

Prerequisites

Classes

Inheritance

Properties

Methods

What & Why

Definition

Differences

- Concrete classes
- Abstract classes
- Interfaces

Interfaces and flexible code

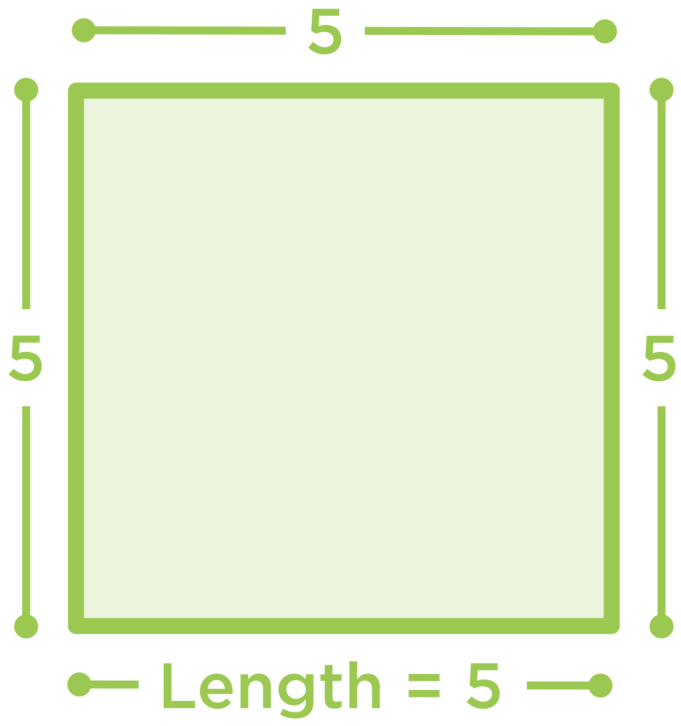
Interfaces describe a group of related functions that can belong to any class or struct.

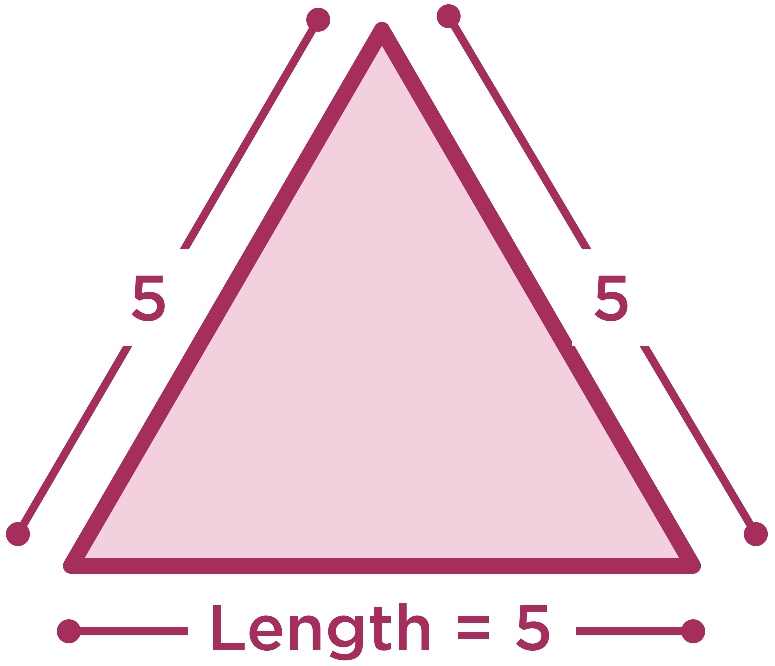
Contract

“I have these functions.”

Properties, methods, events, indexers

Regular Polygons

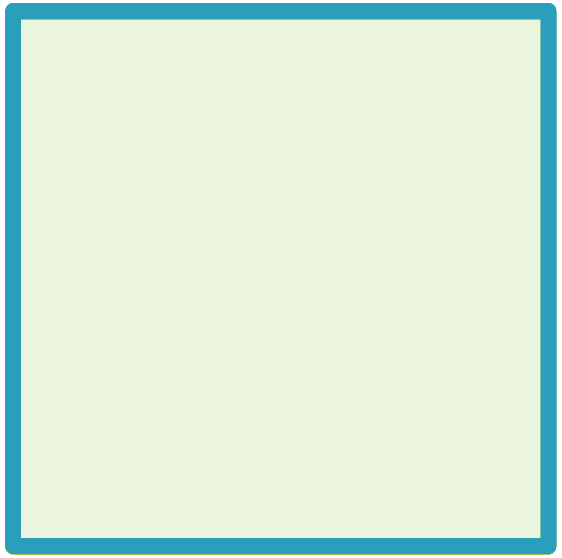




3 or more sides

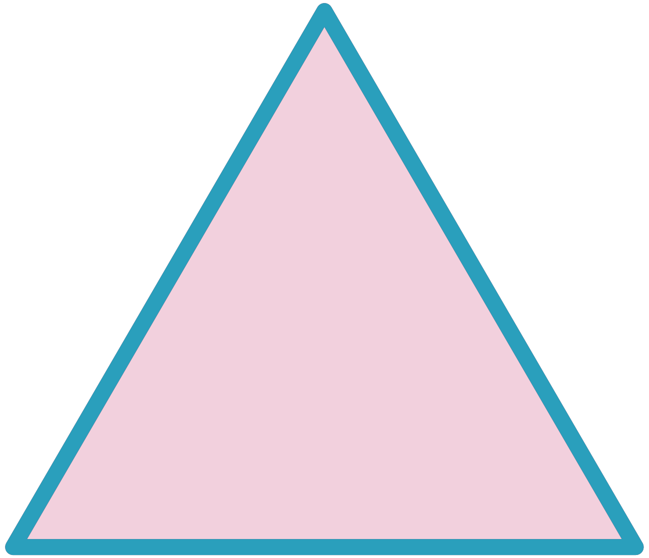
Sides are same length

Perimeter



Length = 5

perimeter = sides x length



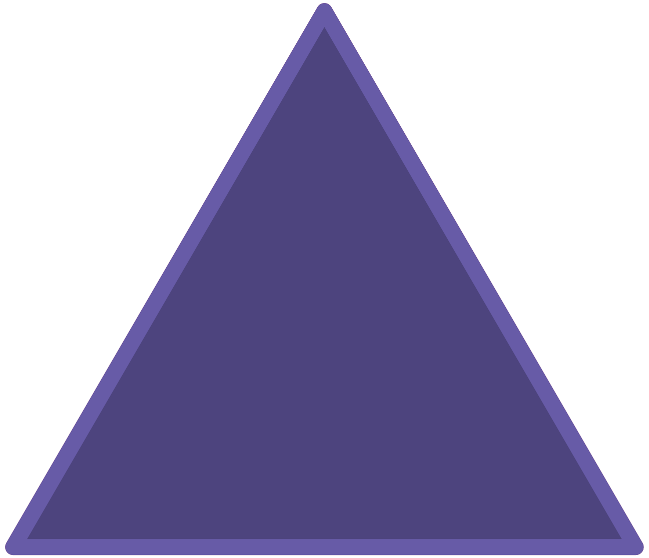
Length = 5

perimeter = sides x length

Area

Length = 5

area = length x length



Length = 5

area = length x length x sqrt(3) / 4

Demo

Regular polygon as

- Concrete class
- Abstract class
- Interface

Interfaces and Flexible Code

Resilience in the face of change

Insulation from implementation details

Program to an abstraction rather than a concrete type

Program to an interface rather than a concrete class

Concrete Collection Types

List<T>	◀ Collection with add/remove
Array	
ArrayList	
SortedList<TKey, TValue>	
HashTable	◀ Unordered bag of objects
Queue / Queue<T>	◀ First in / first out collection
Stack / Stack<T>	◀ Last in / first out collection
Dictionary<TKey,TValue>	
ObservableCollection<T>	

List<T> Interfaces

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

◀ **Allows iteration**

Used with

foreach
List boxes
LINQ

Demo

Code against a class and an interface

Change the type

See how the code responds

What & Why

Definition

Differences

- Concrete classes
- Abstract classes
- Interfaces

Interfaces and flexible code

Creating Interfaces to Add Extensibility

Jeremy Clark
DEVELOPER BETTERER

@jeremybytes www.jeremybytes.com

How

- Create a repository interface
- Implement the interface
  - Web service
  - Text file
  - SQL database
- Remove code duplication
- Focus on important functionality

Different Data Sources

Web service

Text file

SQL database

Document database

Cloud service

Azure functions

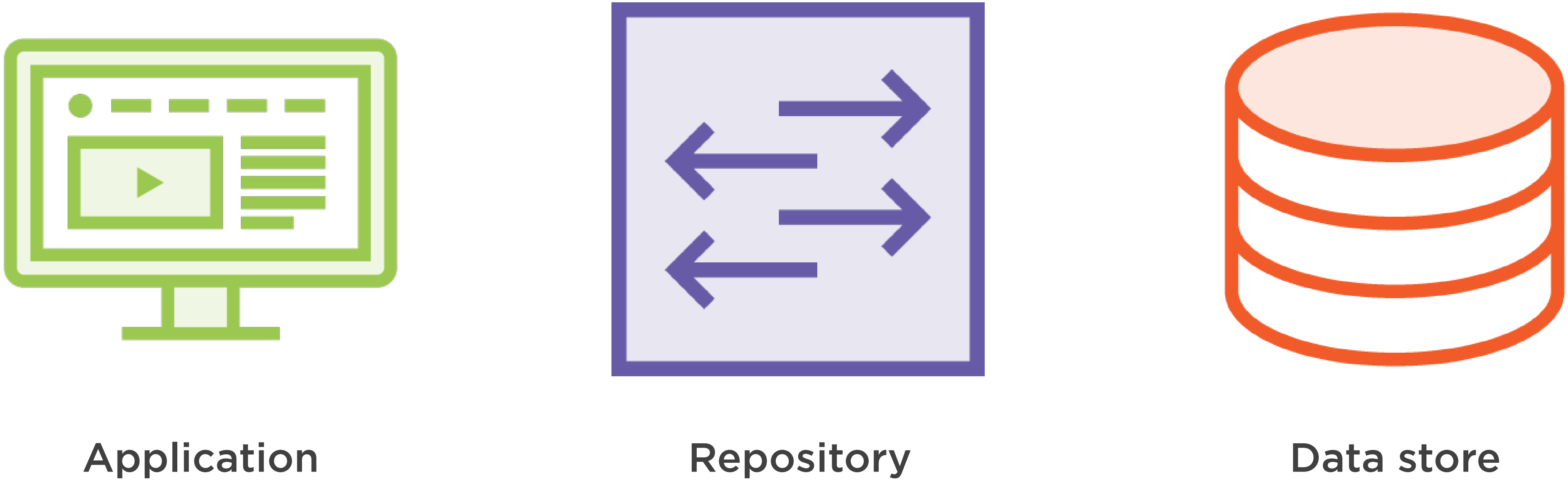
Repository Pattern

Mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects.

Fowler, et al. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.

Repository Pattern

Separates our application from the data storage technology

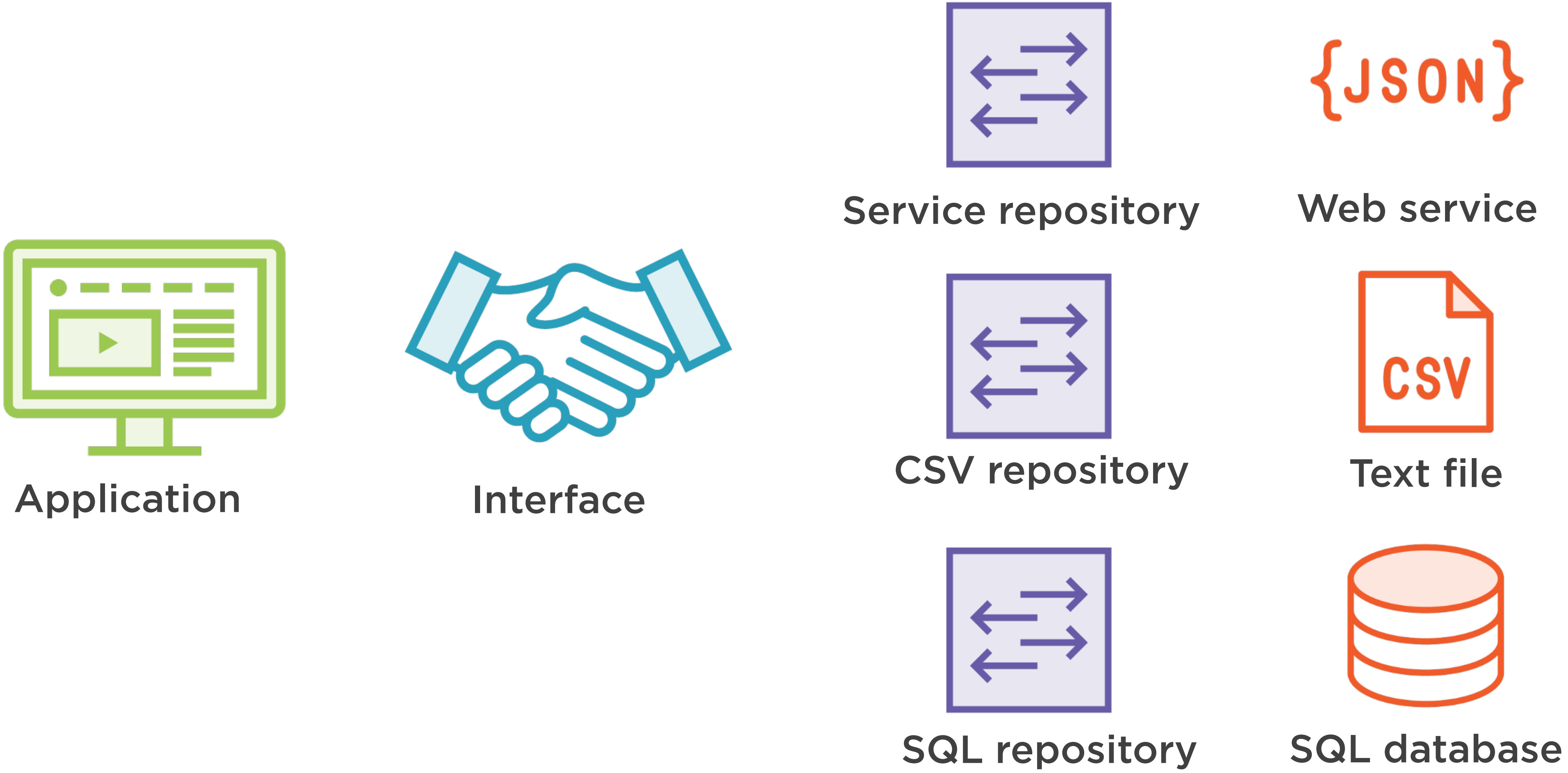


Application

Repository

Data store

Repository Pattern



Application

Interface

Service repository

Web service

CSV repository

Text file

SQL repository

SQL database

CRUD Repository

Create

Read

Update

Delete

```
public interface IPersonRepository
{

  void AddPerson(Person newPerson);

  IEnumerable<Person> GetPeople();

  Person GetPerson(int id);

  void UpdatePerson(int id,
    Person updatedPerson);

  void DeletePerson(int id);

}
```

◀ Create

◀ Read

◀ Read

◀ Update

◀ Delete

Demo

Implementing an interface

Multiple repositories

- Service
- Text file
- SQL database

Remove duplication

Repository Factory

```
IPersonRepository GetRepository(string repositoryType) {

  IPersonRepository repository = null;

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

Demo

Add factory method

Remove references to specific repositories

Application only knows the interface

No References to Specific Repositories

```
private void PopulateListBox(string repositoryType)
{
  ClearListBox();

  IPersonRepository repository =
    RepositoryFactory.GetRepository(repositoryType);

  var people = repository.GetPeople();

  foreach (var person in people)
    PersonListBox.Items.Add(person);

  ShowRepositoryType(repository);
}
```

How

Create a repository interface

Implement

- Web service
- Text file
- SQL database

Remove code duplication

Focus on important functionality

Interfaces and Dynamic Loading

Jeremy Clark

DEVELOPER BETTERER

@jeremybytes www.jeremybytes.com

How & Why

Focus on important functionality

Remove details

Run-time decisions

Change behavior without recompiling

Easier maintenance

Easier unit testing

Program to an abstraction rather than a concrete type

Program to an interface rather than a concrete class

```
private void FetchButton_Click(object sender, RoutedEventArgs e)
{
  ClearListBox();

  IPersonRepository repository = RepositoryFactory.GetRepository();

  var people = repository.GetPeople();

  foreach (var person in people)
    PersonListBox.Items.Add(person);
}
```

Program to an Interface

No references to concrete repository types

Compile-time Factory

```
IPersonRepository GetRepository(string repositoryType) {

  IPersonRepository repository = null;

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

Factory Comparison

Compile-time Factory	Dynamic Factory
Has a parameter	No parameter
Caller picks the repository	Repository based on configuration
Compile-time reference	No compile-time references
	Decisions made at run-time

```
public static IPersonRepository GetRepository()
{
    string repositoryTypeName =
        ConfigurationManager.AppSettings["RepositoryType"];
    Type repositoryType = Type.GetType(repositoryTypeName);
    object repository = Activator.CreateInstance(repositoryType);
    IPersonRepository personRepository =
        repository as IPersonRepository;
    return personRepository;
}
```

Dynamic Loading

Get Type and assembly from configuration

Load assembly through reflection

Create a repository instance with the Activator

Demo

Add dynamic loading code

No compile-time references

Change repository without recompiling

Unit Testing

Testing pieces of functionality in isolation

Interfaces help us isolate code for easier unit testing.

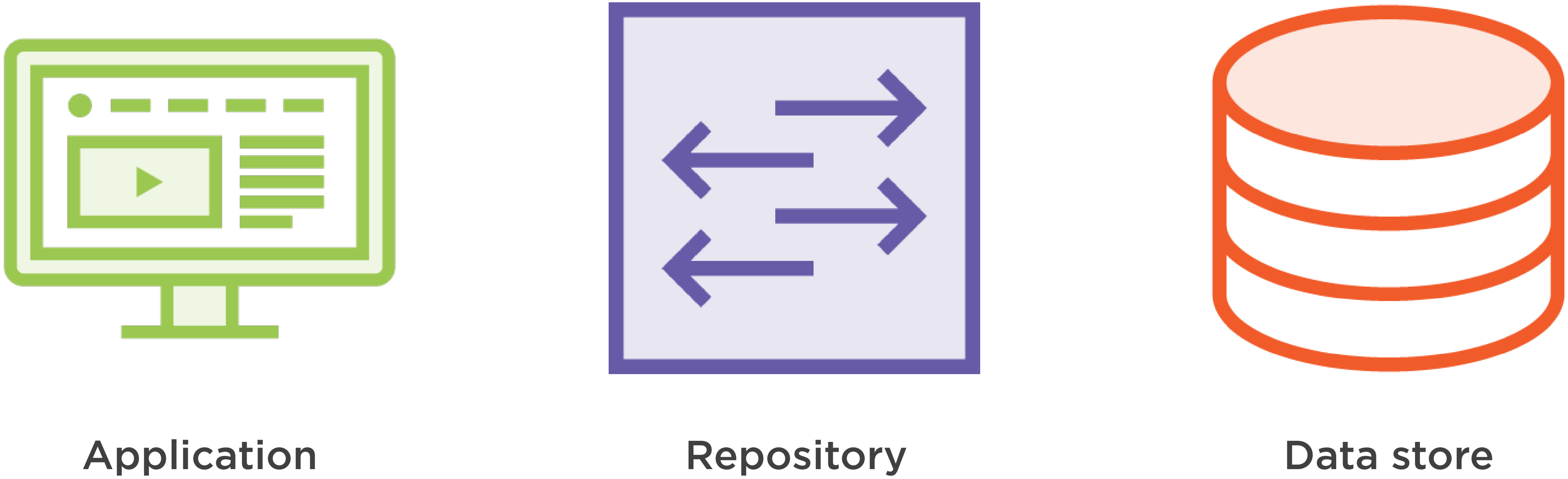
What We Want to Test

```
public partial class MainWindow : Window
{
    private void FetchButton_Click(object sender, RoutedEventArgs e)
    {
        ClearListBox();

        IPersonRepository repository =
            RepositoryFactory.GetRepository();

        var people = repository.GetPeople();
        foreach (var person in people)
            PersonListBox.Items.Add(person);

        ShowRepositoryType(repository);
    }
    ...
}
```

Dependent Objects

```
public partial class MainWindow : Window
{
    private void FetchButton_Click(object sender, RoutedEventArgs e)
    {
        ClearListBox();

        IPersonRepository repository =
            RepositoryFactory.GetRepository();

        var people = repository.GetPeople();
        foreach (var person in people)
            PersonListBox.Items.Add(person);

        ShowRepositoryType(repository);
    }
    ...
}
```

Current Application

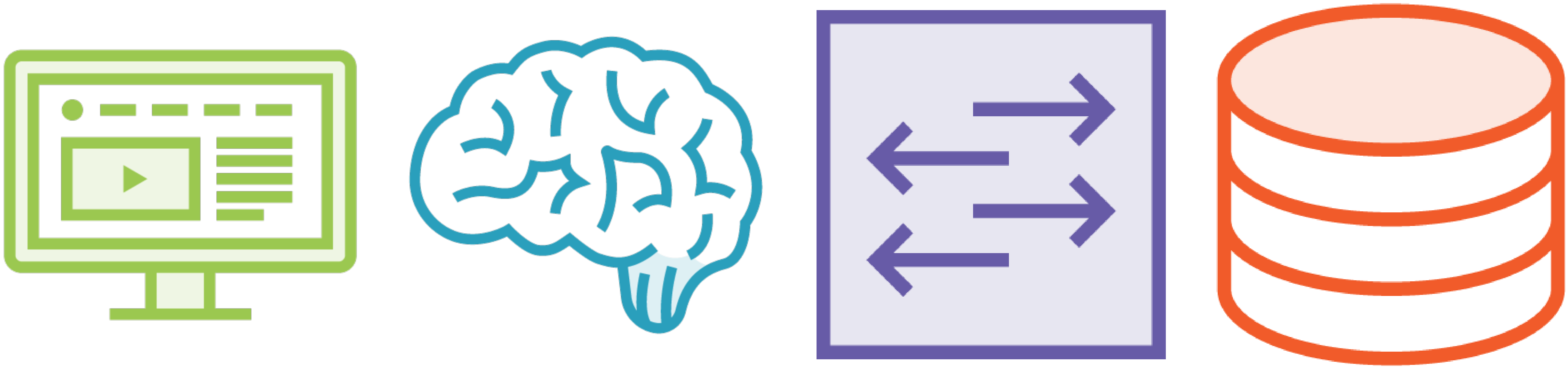


Application

Repository

Data store

Application with View Model



View
(UI elements)

View model
(UI logic)

Repository

Data store

Demo

Move functionality to a view model

Add a fake repository for tests

Unit test the view model functionality

How & Why

Focus on important functionality

Remove details

Run-time decisions

Change behavior without recompiling

Easier maintenance

Easier unit testing

Explicit Interface Implementation

Jeremy Clark
DEVELOPER BETTERER

@jeremybytes   www.jeremybytes.com

What & Why

Allow for more control

Resolve conflicting methods

IEnumerable<T> + IEnumerable

Standard Interface Implementation

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

```
public class Catalog : ISaveable
{
  public void Save()
  {
    Console.Write("Saved");
  }
}
```

```
Catalog catalog = new Catalog();

catalog.Save();
// "Saved"



ISaveable saveable = catalog;

saveable.Save();
// "Saved"
```

Explicit Interface Implementation

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

```
public class Catalog : ISaveable
{

  void ISaveable.Save()
  {
    Console.Write("Saved");
  }

}
```

```
Catalog catalog = new Catalog();

catalog.Save();
*** COMPILER ERROR ***



ISaveable saveable = catalog;

saveable.Save();
// "Saved"

(ISaveable(catalog)).Save();
// "Saved"
```

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

◀ Interface type

```
Catalog catalog = new Catalog();
catalog.Save();
*** COMPILER ERROR ***
```

◀ Interface not used

```
var varCatalog = new Catalog();
varCatalog.Save();
*** COMPILER ERROR ***
```

◀ Interface not used
(same as using "Catalog" type)

```
((ISaveable)catalog).Save();
// "Saved"
```

◀ Interface type

Mixed Methods

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

```
public class Catalog : ISaveable
{
  public void Save()
  {
    Console.Write("Saved (catalog)");
  }

  void ISaveable.Save()
  {
    Console.Write("Saved (interface)");
  }
}
```

```
Catalog catalog = new Catalog();

catalog.Save();
// "Saved (catalog)"



ISaveable saveable = catalog;

saveable.Save();
// "Saved (interface)"

(ISaveable(catalog)).Save();
// "Saved (interface)"
```

Conflicting Method Signatures

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

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

```
public class Catalog : ISaveable, IDbSaver
{
  public void Save()      // Catalog & ISaveable
  {
    Console.Write("Saved from ISaveable interface");
  }
  string IDbSaver.Save()  // IDbSaver (explicit)
  {
    return "Saved from IDbSaver interface";
  }
}
```

Another Explicit Implementation

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

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

```
public class Catalog : ISaveable, IDbSaver
{
  void ISaveable.Save()   // ISaveable (explicit)
  {
    Console.Write("Saved from ISaveable interface");
  }
  public string Save()    // Catalog & IDbSaver
  {
    return "Saved from IDbSaver interface";
  }
}
```

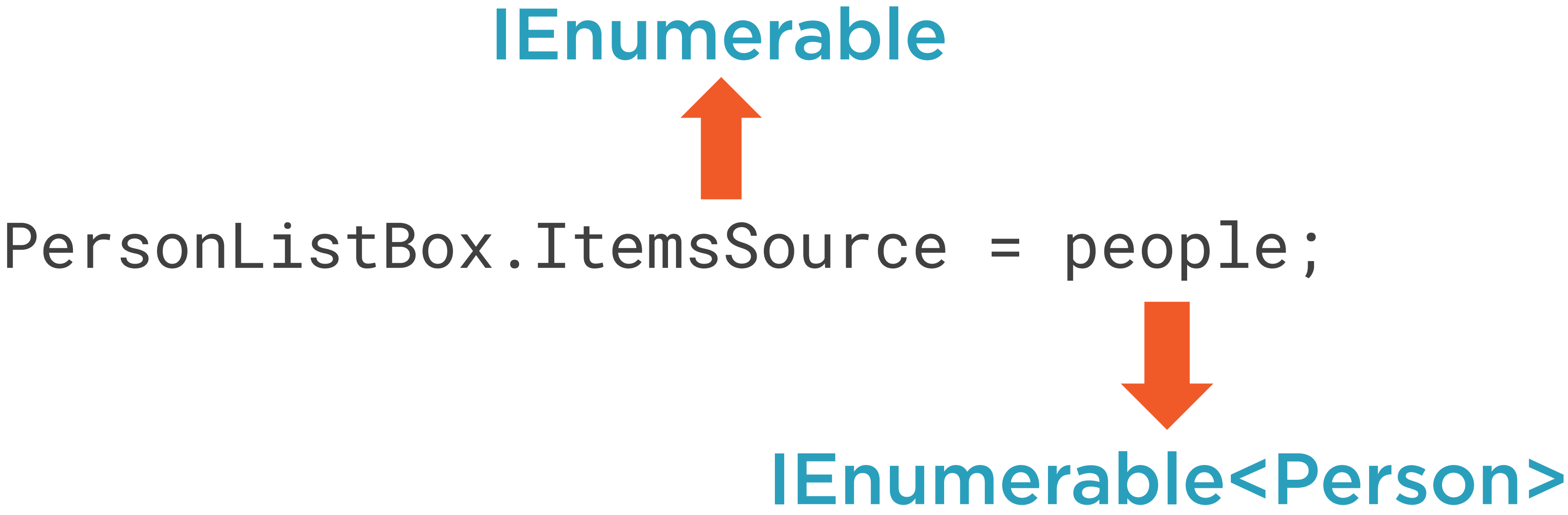
Both Explicitly Implemented

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

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

```
public class Catalog : ISaveable, IDbSaver
{
  void ISaveable.Save()   // ISaveable (explicit)
  {
    Console.Write("Saved from ISaveable interface");
  }
  string IDbSaver.Save()  // IDbSaver (explicit)
  {
    return "Saved from IDbSaver interface";
  }
}
```

Type Mismatch?



IEnumerable

```
PersonListBox.ItemsSource = people;
```

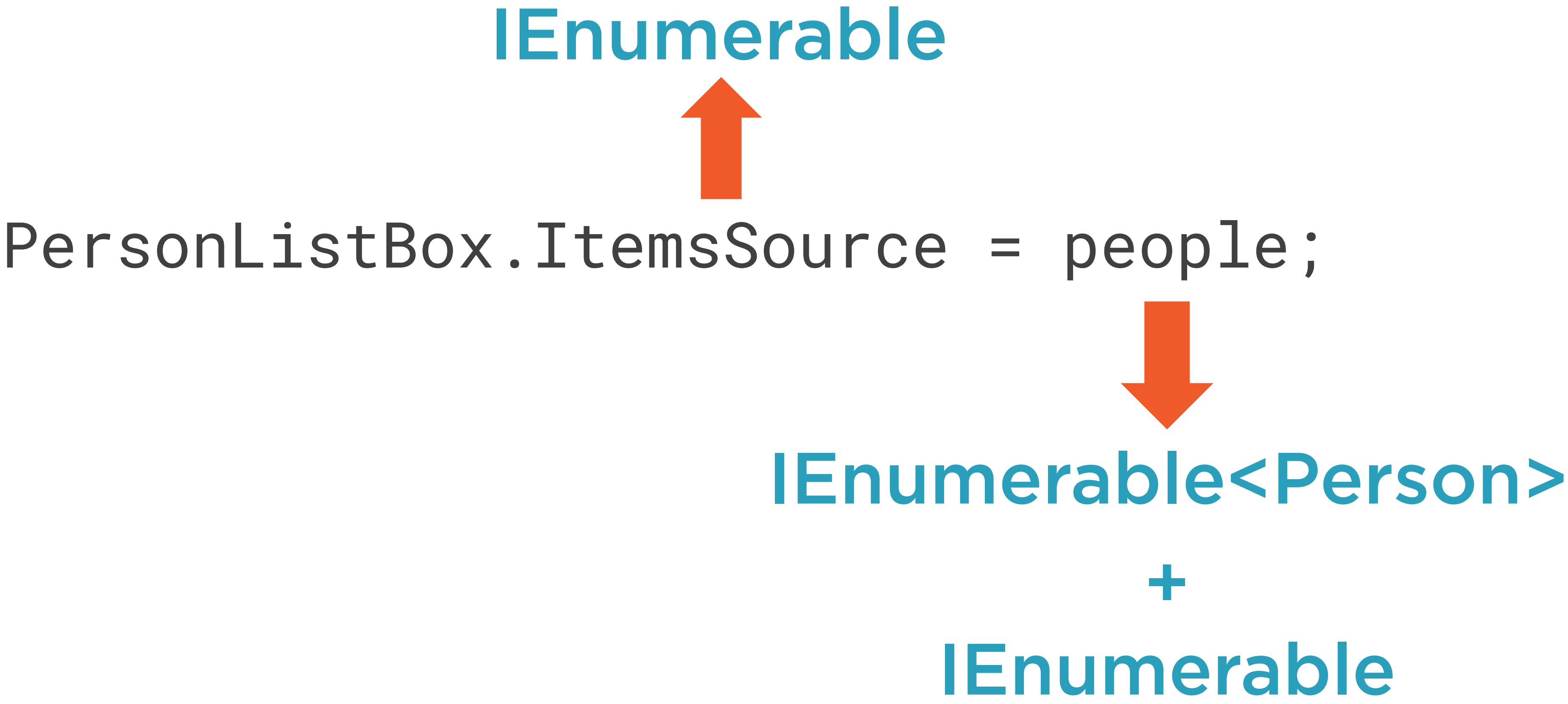
IEnumerable<Person>

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

Interface Inheritance

IEnumerable<T> includes all members from IEnumerable

No Type Mismatch



IEnumerable

```
PersonListBox.ItemsSource = people;
```

IEnumerable<Person>

+

IEnumerable

IEnumerable Members

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

Conflicting Signatures

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

Demo

Explicit interface implementation

- IEnumerable
- IEnumerable<T>

What & Why

Allow for more control

Resolve conflicting methods

IEnumerable<T> + IEnumerable

Designing Effective Interfaces

Jeremy Clark
DEVELOPER BETTERER

@jeremybytes www.jeremybytes.com

How

Danger of too many interfaces

Interface Segregation Principle

Updating interfaces

Default implementation

Interface inheritance

Interfaces vs. abstract classes

Program to an abstraction rather than a concrete type

Program to an interface rather than a concrete class

Be careful of too many interfaces

Add interfaces as you need them (not before).

Demo

Abstraction and code navigation

Abstraction and debugging

Interface Segregation Principle

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

Martin and Martin. *Agile Principles, Patterns, and Practices in C#*. Prentice Hall, 2007.

Translation

Interfaces should only include what the calling code needs

```
public interface IPersonRepository
{
  void AddPerson(Person newPerson);

  IEnumerable<Person> GetPeople();

  Person GetPerson(int id);

  void UpdatePerson(int id,
    Person updatedPerson);

  void DeletePerson(int id);
}
```

◀ Create

◀ Read

◀ Read

◀ Update

◀ Delete

Read-Only Client

```
private void PopulateListBox(string repositoryType)
{
  ClearListBox();

  IPersonRepository repository =
    RepositoryFactory.GetRepository(repositoryType);

  var people = repository.GetPeople();

  foreach (var person in people)
    PersonListBox.Items.Add(person);

  ShowRepositoryType(repository);
}
```

Read-only

```
public interface IPersonRepository
{
  ~~void AddPerson(Person newPerson);~~                  ◀ UNUSED

  IEnumerable<Person> GetPeople();

  Person GetPerson(int id);

  ~~void UpdatePerson(int id,~~                          ◀ UNUSED
    ~~Person updatedPerson);~~

  ~~void DeletePerson(int id);~~                         ◀ UNUSED
}
```

A Better Interface

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

Demo

Break up repository interface

- Read
- Update

An interface is a contract

Adding Members Breaks Implementers

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

```
public class Catalog : ISaveable
{
  public void Save()
  {
    Console.Write("Saved (catalog)");
  }
}
```

Adding Members Breaks Implementers

```
public interface ISaveable {
  void Save();
  void Save(string message); // Added Member
}
```

```
public class Catalog : ISaveable
{
  public void Save()
  {
    Console.Write("Saved (catalog)");
  }
}
*** ERROR Save(string) is missing ***
```

Removing Members Breaks Callers

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

```
public class InventoryItem
{

  ISaveable saver = new SQLSaver();
  saver.Save("Added inventory");

}
```

Removing Members Breaks Callers

```
public interface ISaveable {
  void Save();
  // void Save(string message) REMOVED
}
```

```
public class InventoryItem
{

  ISaveable saver = new SQLSaver();
  saver.Save("Added inventory"); *** ERROR ***

}
```

An interface is a contract

Existing Interface

```
interface ILogger
{
    void Log(LogLevel level, string message);
}

class ConsoleLogger : ILogger
{
    public void Log(LogLevel level, string message) { ... }
}
```

Default Implementation

```
interface ILogger
{
    void Log(LogLevel level, string message);

    void Log(Exception ex) =>
        Log(LogLevel.Error, ex.ToString()); // New overload
}

class ConsoleLogger : ILogger
{
    public void Log(LogLevel level, string message) { ... }

    // Log(Exception) gets default implementation
}
```

Use wisely

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

Interface Inheritance

IEnumerable<T> includes all members from IEnumerable

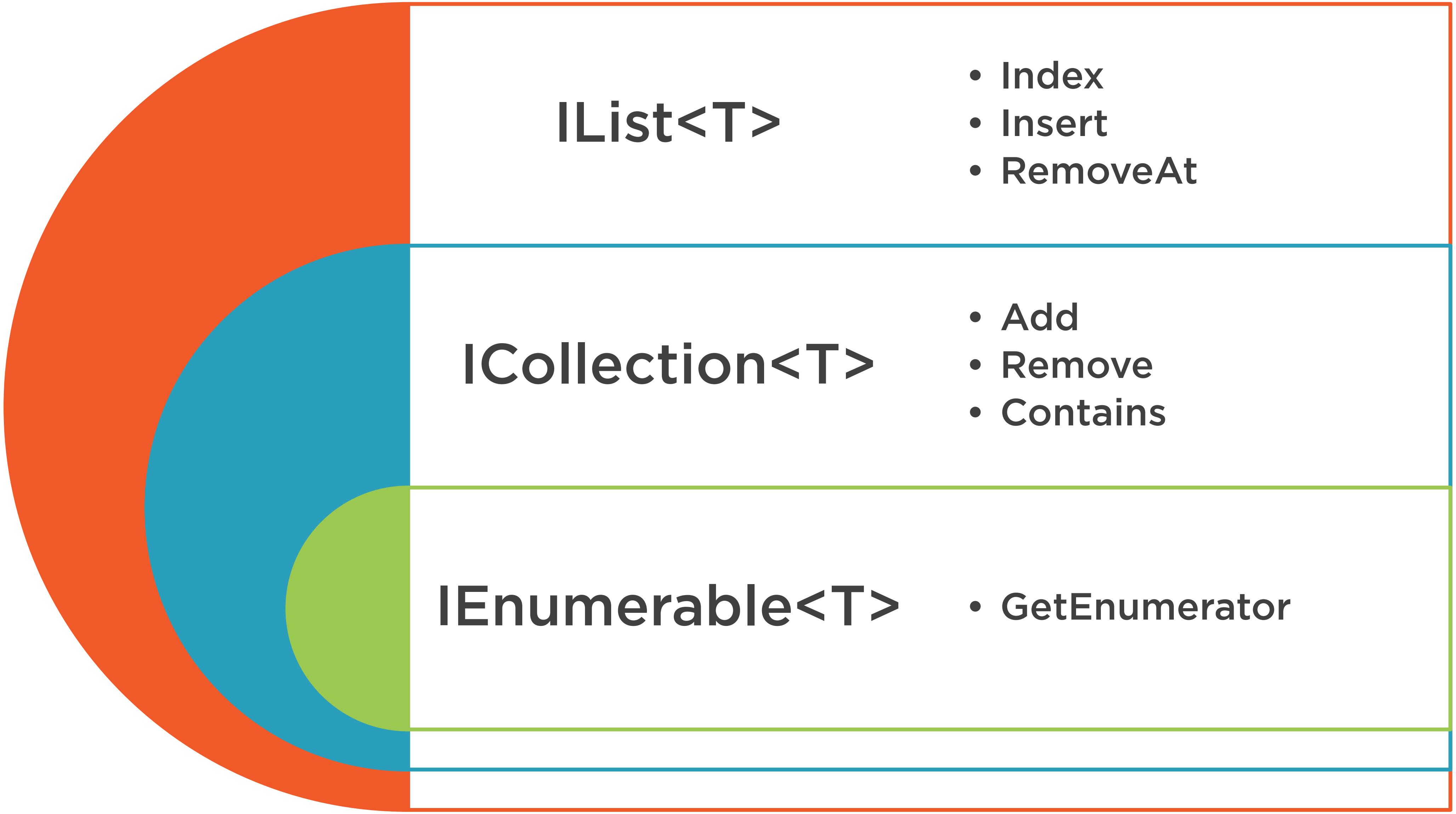
```
public class List<T> : IList<T>, ICollection<T>,
    IEnumerable<T>, IReadOnlyCollection<T>,
    IReadOnlyList<T>, IList, IEnumerable
```

Interface Inheritance

IList<T>

ICollection<T>

IEnumerable<T>



IList<T>

- Index
- Insert
- RemoveAt

ICollection<T>

- Add
- Remove
- Contains

IEnumerable<T>

- GetEnumerator

Implementations

IEnumerable<T>

List<T>
Array
SortedList<T, V>
Queue<T>
Stack<T>
Dictionary<T, V>

Custom Types

ICollection<T>

List<T>
SortedList<T>
Dictionary<T, V>

CustomTypes

IList<T>

List<T>

CustomTypes

Read-only Repository

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

Read-write Repository

```
public interface IPersonRepository : IPersonReader
{
    void AddPerson(Person newPerson);
    void UpdatePerson(int id, Person updatedPerson);
    void DeletePerson(int id);
}
```

Comparing Interfaces and Abstract Classes

Interface	Abstract Class
No implementation code*	May have implementation code
Implement any number of interfaces	Single inheritance
Members automatically public	Access modifiers on members
Properties methods events indexers	Properties methods events indexers fields constructors destructors

* Exception: default implementation

```
// Polygon

public int NumberOfSides {…}
public int SideLength {…}
public double GetPerimeter()
public double GetArea()
```

◀ Shared

◀ Shared

◀ Shared

◀ Not shared

Abstract Class

Repositories

```
public IEnumerable<Person> GetPeople() {
  string result = client.DownloadString(baseUri);
  var people = JsonConvert.DeserializeObject<...>(result);
  return people;
}

public IEnumerable<Person> GetPeople() {
  var people = new List<Person>();
  if (File.Exists(path))
    using (var reader = new StreamReader(path)) {...}
  return people;
}

public IEnumerable<Person> GetPeople() {
  using (var context = new PersonContext(options)) {
    return context.People.ToArray();
  }
}
```

Interface

How

Danger of too many interfaces

Interface Segregation Principle

Updating interfaces

Default implementation

Interface inheritance

Interfaces vs. abstract classes

Interfaces in Frameworks and Patterns

Jeremy Clark

DEVELOPER BETTERER

@jeremybytes www.jeremybytes.com

Where

Dependency injection

Design patterns

- Repository
- Factory method
- Decorator

Mocking

Dependency Injection (DI)

A set of software design principles and patterns that enable us to develop loosely coupled code.

van Deursen and Seeman. *Dependency Injection in .NET*. Manning, 2018.

Interfaces help us create loose coupling.

```
private void FetchButton_Click(object sender, RoutedEventArgs e)
{
  ClearListBox();

  IPersonRepository repository = RepositoryFactory.GetRepository();

  var people = repository.GetPeople();

  foreach (var person in people)
    PersonListBox.Items.Add(person);
}
```

Delegating Details

No references to concrete repository types

"Seam" allows easy swapping of repositories

Getting a Dependency

```
public class PeopleViewModel : INotifyPropertyChanged
{
    private IPersonRepository repository;

    public PeopleViewModel()
    {
        repository = RepositoryFactory.GetRepository();
    }

    public void FetchData()
    {
        People = repository.GetPeople();
    }
    ...
}
```

Injecting a Dependency

```
public class PeopleViewModel : INotifyPropertyChanged
{
    private IPersonRepository repository;

    public PeopleViewModel(IPersonRepository injectedRepo)
    {
        repository = injectedRepo;
    }

    public void FetchData()
    {
        People = repository.GetPeople();
    }
    ...
}
```

Demo

Injecting a repository

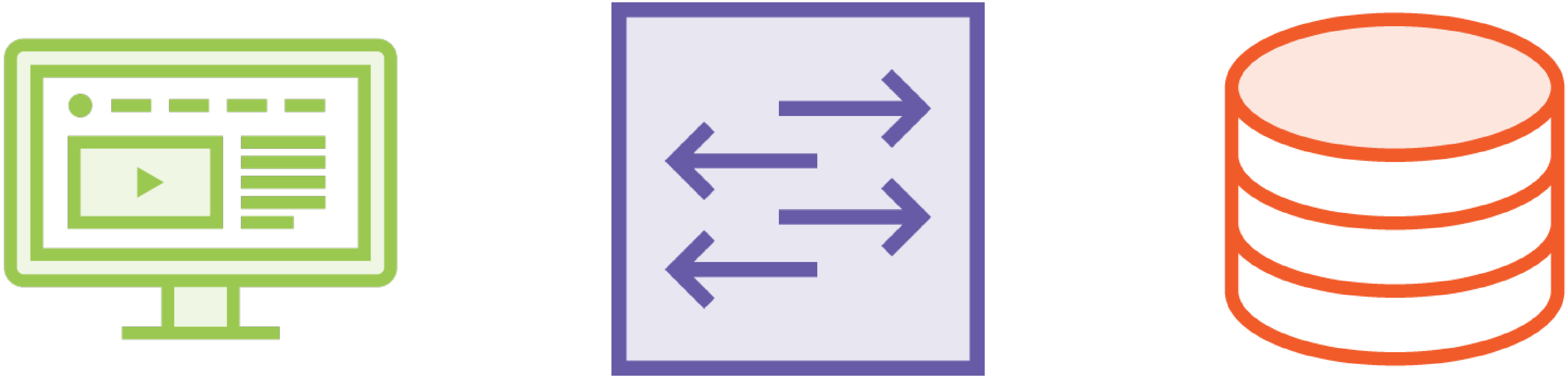
- Manual construction
- Dependency injection container

Unit tests with DI

Interfaces help with implementing design patterns.

Repository Pattern

Separates our application from the data storage technology



Application

Repository

Data store

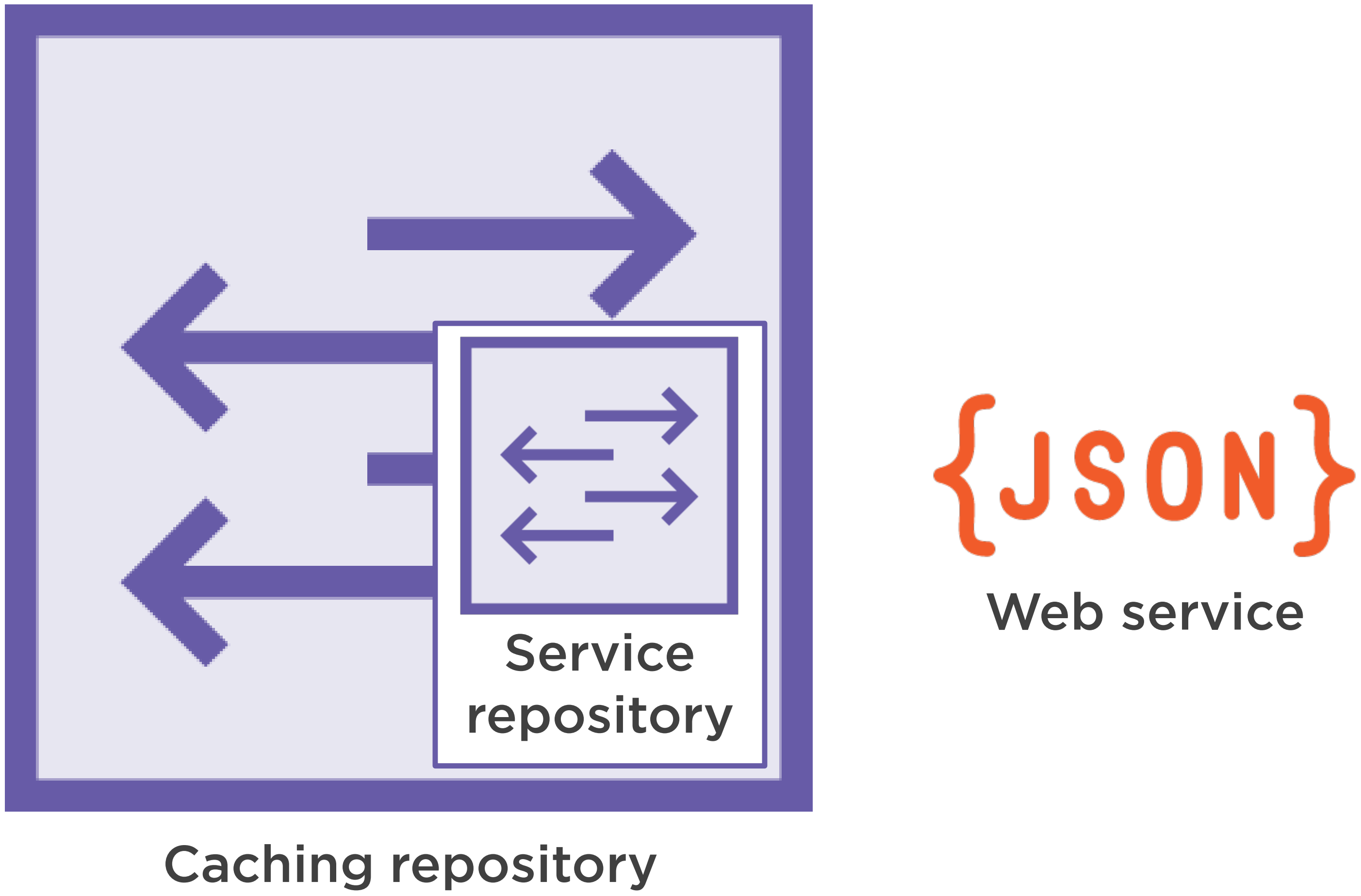
Factory Method Pattern

```
IPersonRepository GetRepository(string repositoryType) {

  IPersonRepository repository = null;

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

Decorator

Wrap an existing interface to add functionality

Repository Decorator



{JSON}

Service repository

Web service

Caching repository

Demo

Caching decorator

Mocking

Creating an in-memory object for testing purposes

Fake Repository

```
public class FakeRepository : IPersonRepository
{
    public IEnumerable<Person> GetPeople()
    {
        var people = new List<Person>() {...};
        return people;
    }

    public Person GetPerson(int id)
    {
        var people = GetPeople();
        return people.FirstOrDefault(p => p.Id == id);
    }
}
```

In-Memory Repository with MOQ

```
private IPersonRepository GetMockRepository()
{
    var testPeople = new List<Person>() {...};

    var mockRepo = new Mock<IPersonRepository>();

    mockRepo.Setup(m => m.GetPeople()).Returns(testPeople);

    return mockRepo.Object;

}
```

Demo

Test with mock repository

Where

Dependency injection

Design patterns

- Repository
- Factory method
- Decorator

Mocking