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);         ◀ Create

  IEnumerable<Person> GetPeople();          ◀ Read

  Person GetPerson(int id);                 ◀ Read

  void UpdatePerson(int id,                 ◀ Update
    Person updatedPerson);

  void DeletePerson(int id);                ◀ Delete
}
```

Read-Only Client

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

  IPersonRepository repository =
    RepositoryFactory.GetRepository(repositoryType);

  var people = repository.GetPeople();   Read-only

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

  ShowRepositoryType(repository);
}
```

```
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

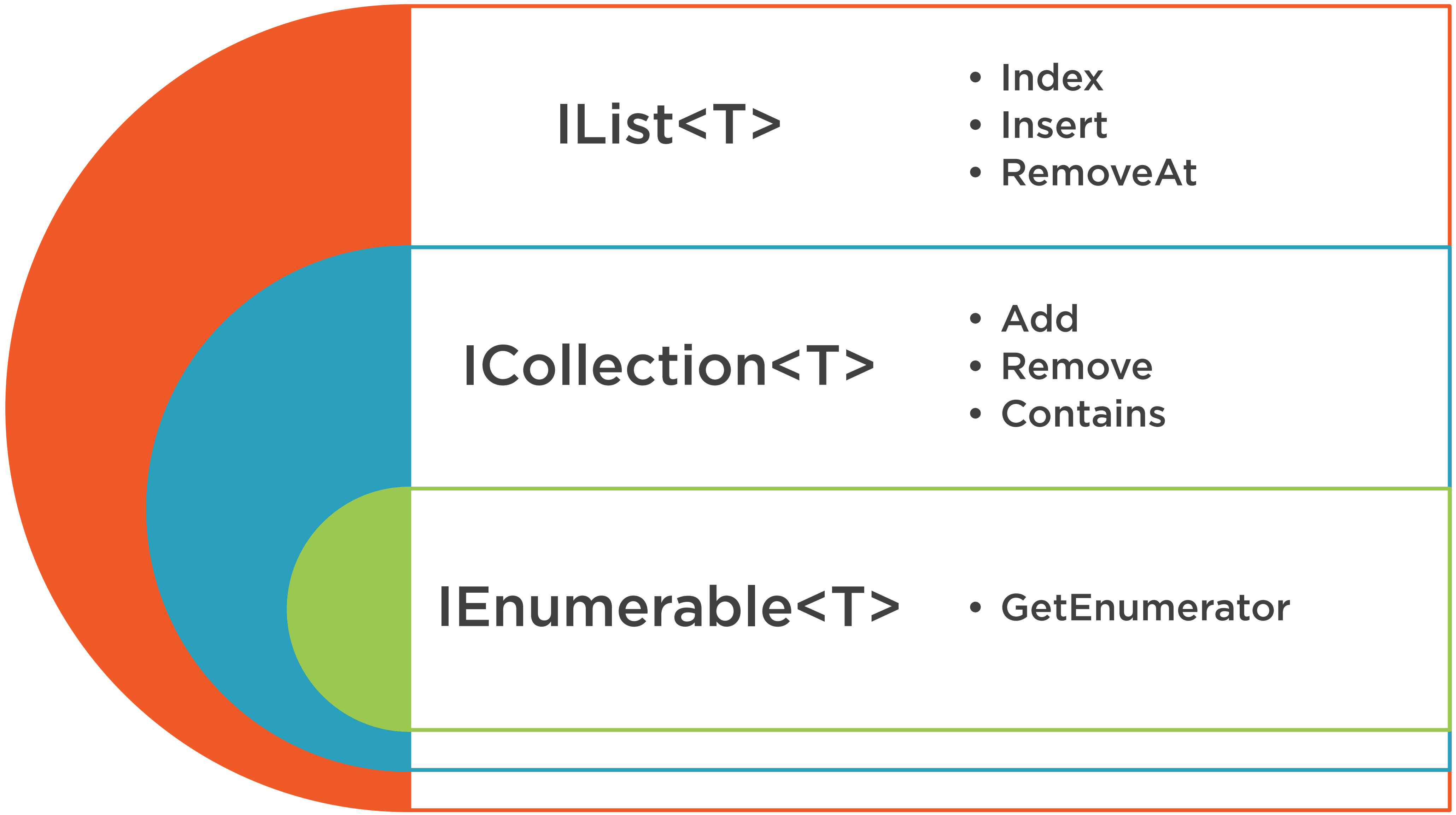
```
public class List<T> : IList<T>, ICollection<T>,
    IEnumerable<T>, IReadOnlyCollection<T>,
    IReadOnlyList<T>, IList, IEnumerable
```

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