# Interface-Based Programming in C#

# Agenda

- Interfaces in C#
- Implementing Interfaces
- Using Interfaces
- Interface-Based Programming
- Interfaces Factoring and Design

# Interfaces in C#

- interface keyword defines a type that
  - Cannot have implementation (all methods are abstract)
  - Cannot be instantiated (like abstract class)

```
//This definition
public interface IMyInterface
{
    void Method1();
    void Method2();
    void Method3();
}
//is almost equivalent to this one
public abstract class MyInterface
{
    abstract public void Method1();
    abstract public void Method2();
    abstract public void Method3();
}
```

# Interfaces

- Interfaces are not the same as abstract classes
  - Abstract class can still have implementation
  - Class can derive from only one base class
  - Class can derive from multiple interfaces
  - Abstract class can derive from any other class or interface(s)
  - Interface can only derive from other interfaces
  - Abstract class can have non-public members
  - Abstract class can have constructors, static members and constants
- Differences are deliberate to provide for a formal public contract

# Interfaces

- COM-like semantics
  - Contract
  - Logical grouping of related methods
- Interfaces can only derive from other interfaces
- Interface can derive from multiple other interfaces
  - Unlike COM
- All interface methods are public
  - Contract semantics

# Interfaces

- Interface can define:
  - methods,
  - properties,
  - events,
  - indexes

  *All are public always!*

- Interface cannot define static and constants
- Interface has visibility
  - `public interface IMyInterface(){…}`
  - `internal interface IMyInterface(){…}`

# Implementing Interfaces

# Implementing Interfaces

- Subclass implementation must be public
- No need for `override` or `new`
- Must implement all methods in interface derivation chain!

```
public interface IMyInterface
{
    void Method1();
    void Method2();
    void Method3();
}
public class MyClass : IMyInterface
{
    public MyClass() {}
    public void Method1() {}
    public void Method2() {}
    public void Method3() {}
}
```

# Implementing Interfaces

- Can derive from multiple interfaces

```
public interface IMyInterface1
{
    void Method1();
}
public interface IMyInterface2
{
    void Method2();
}

public class MyClass : IMyInterface1, IMyInterface2
{
    public MyClass() {}
    public void Method1() {}
    public void Method2() {}
}
```

# Implementing Interfaces

- Can still derive from one concrete class, in addition to interfaces
  - But base class must be first in derivation chain!

```
public interface IMyInterface
{}
public interface IMyOtherInterface
{}
public class MyBaseClass
{}
public class MySubClass : MyBaseClass, IMyInterface, IMyOtherInterface
{}
```

# Implementing Interfaces

- **Interface Methods, Properties and Events**

```
public interface IMyInterface
{
    void Method1();                                  //A method
    int SomeProperty{ get; set; }                    //A property
    int this[int index]{ get; set;}                  //An indexer
    event NumberChangedEvent NumberChanged; //An event
}
public class MyClass : IMyInterface
{
    public void Method1(){…}
    public int SomeProperty
    { get{…} set{…} }
    public int this[intindex]
    { get{…} set{…} }
    public event NumberChangedEvent NumberChanged;
}
```

# Implementing Interfaces - Method Collision

- Interfaces may define the same method
- Implementing class can
  - Channel both to the same implementation

```
public interface IMyInterface1
{
    void MyMethod();
}
public interface IMyInterface2
{
    void MyMethod();
}

public class MyClass : IMyInterface1, IMyInterface2
{
public MyClass(){}
public void MyMethod(){}//Same implementation for both interfaces
}
```

# Implementing Interfaces - Method Collision

- Or implementing class can
  - Provide different implementation, by qualifying interface
    - Clients can only access via interface type!

```
public interface IMyInterface1
{
   void MyMethod();
}
public interface IMyInterface2
{
   void MyMethod();
}
public class MyClass : IMyInterface1, IMyInterface2
{
   public MyClass() {}
   void IMyInterface1.MyMethod(){} //First interface implementation
   void IMyInterface2.MyMethod(){} //second interface implementation
}
// client code
IMyInterface1 if1 = (IMyInterface1)new MyClass();
if1.MyMethod();
```

# An interface's implementation shall do what its methods says it does

- The name of a method should correspond to the operation that the implementation actually performs.
  - This is also known as **the Principle of Least Surprise**
- If the purpose and meaning of a method are not unambiguously obvious from the method's name and its place within an interface:
  - Then rename the method in the interface, or
  - Provide a clear documentation
    - As a written document,
    - As an example of implementation, or
    - As a set of interface test cases.

# If an implementation is unable to perform its responsibilities, it shall notify its caller

- An implementation should report problems that are encountered and that it cannot fix itself.
- The manner of report can be either a return code or be an exception thrown.
- The errors that are denoted on the interface are part of the interface contract
  - An interface implementation should produce only those errors.

# Using Interfaces

# Using Interfaces

- Explicitly cast object reference to required interface
  - **Compiler does not enforce type safety!**
  - Throws exception if not supported
  - Should use **try/catch, as or is**

```
public interface IMyInterface
{
void Method1();
}
public class MyClass : IMyInterface
{}
//Explicit cast
IMyInterface obj = (IMyInterface) new MyClass();
obj.Method1();
```

# Using Interfaces

- Explicitly cast when using class factories

```
public interface IMyInterface
{
   void Method1();
}
public interface IClassFactory
{
   object GetObject();
   IMyInterface Create();
}
public class MyClass : IMyInterface
{}

IClassFactory factory;
/*some code to initialize the class factory*/
IMyInterface obj;
obj = (IMyInterface)factory.GetObject();
obj.Method1();
/*or if using a specialized class factory*/
obj = factory.Create();
```

# Using Interfaces

- Explicitly cast when using multiple interfaces

```
public interface IMyInterface
{
   void Method1();
}
public interface IMyOtherInterface
{
   void Method2();
}
public class MyClass : IMyInterface, IMyOtherInterface
{…}

//Client side code:
IMyInterface obj1;
IMyOtherInterface obj2;
obj1 = new MyClass();
obj1.Method1();
obj2 = (IMyOtherInterface)obj1;
obj2.Method2();
```

# Using Interfaces

- Even better: use C#'s **as** operator
  - Like COM's **QueryInterface()**

```
Object obj1 = new SomeType();
IMyInterface obj2;

/* Some code to initialize obj1 */

obj2 = obj1 as IMyInterface;
if(obj2 != null)
{
   obj2.Method1();
}
else
{
   //Handle error in expected interface not supported
}
```

# Interface-Based Programming

# Interface-Based Programming

- Separation of interface from implementation is a core component-oriented principle
  - Changing service provider without affecting client
  - Client programs against an abstraction of the service, not a particular implementation (the object)
- .NET does not enforce the separation
  - Unlike COM
- But disciplined developers should **ALWAYS** enforce separation
  - Mandatory between components
  - Optional for internal classes

# Interface-Based Programming

- Developer has to provide for separation of interface from implementation
  - .NET lets you program against the object directly

```
using MyAssembly;

//Avoid doing this:
MyComponent obj;
obj = new MyComponent();
obj.ShowMessage();
```

# Interface-Based Programming

- Client-side programming:
  - Program against interface, not object
  - Never assume the object support an interface
    - Use try/catch, **as** or **is**

```
SomeType obj1;
IMyInterface obj2;
/* Some code to initialize obj1 */
obj2 = obj1 as IMyInterface;
if(obj2 != null)
{
   obj2.Method1();
}
else
{
   //Handle error in expected interface
}
```

# Interface-Based Programming

- Server-side programming:
  - Provide explicit interface member implementation
- Explicit implementation cannot be public
  - Or have any visibility modifier at all

```
public interface IMyInterface
{
   void Method1();
   void Method2();
}

public class MyClass : IMyInterface
{
public MyClass(){}
void IMyInterface.Method1(){} //explicit implementation
void IMyInterface.Method2(){} //explicit implementation
}
```

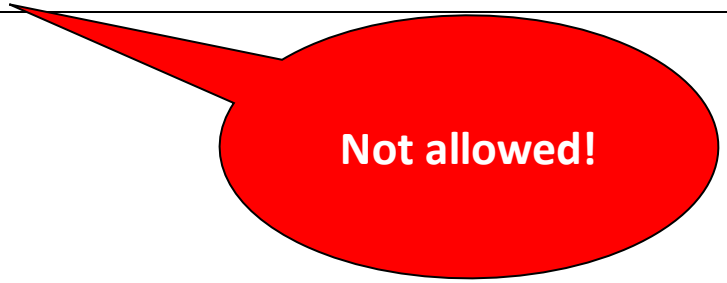
*Only accessibly through IMyInterface*

# Interface-Based Programming

- **Explicit implementation** forces client to program against interface, not object

```
IMyInterface obj1 = new MyClass();
obj1.Method1();

//This does not compile:
MyClass obj2 = new MyClass();
obj2.Method1();
```



# Interface-Based Programming

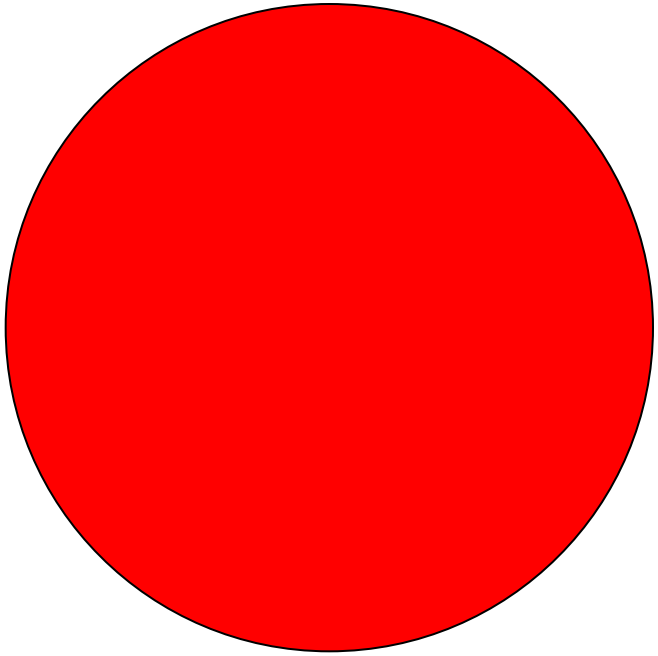
- Assemblies with interfaces only
  - **Because interfaces can be implemented by multiple components, it's good practice to put them in a separate assembly from that of the implementing components.**
  - **Allows concurrent development of the server and the client**
    - **once the two parties have agreed on the interface.**

- This separation of interfaces leads to many assemblies for each application
  - which may be inconvenient to distribute and install
  - So Microsoft Research has come up with an utility which can merge many assemblies to 1 assembly – **ILMerge** (does not work for WPF) http://research.microsoft.com/~mbarnett/ILMerge.aspx
  - An alternative to ILMerge is to embed the dll's as an embedded resource in the exe. Read here: http://blogs.msdn.com/b/microsoft_press/archive/2010/02/03/jeffrey-richter-excerpt-2-from-clr-via-c-third-edition.aspx

# Interfaces Factoring and Design

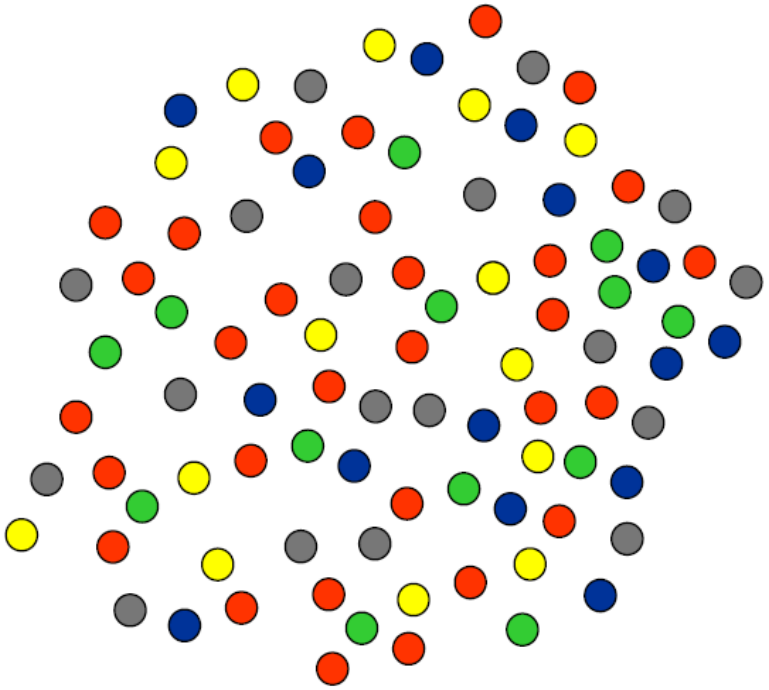
# Interface Granularity

- Is this a good design?



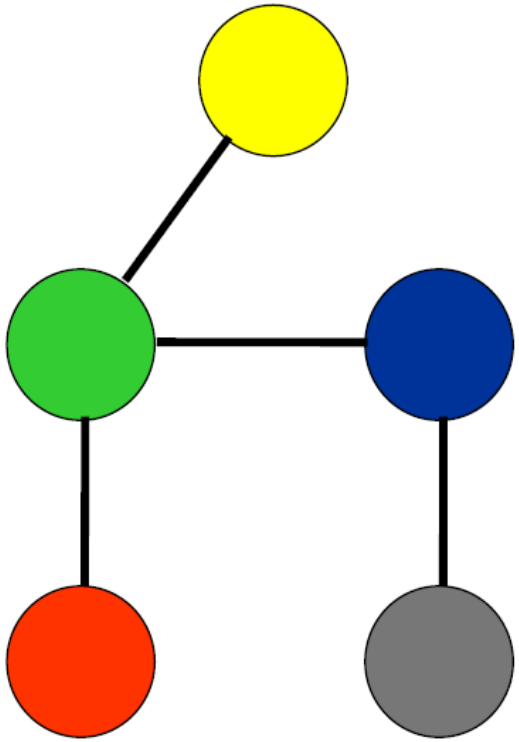
# Interface Granularity

- Is this a good design?



# Interface Granularity

- Is this a good design?



# A Bloated Interface

```
interface Iterator
{
boolean set_to_first_element();
boolean set_to_next_element();
boolean set_to_next_nth_element(in unsigned long n) raises(…);
boolean retrieve_element(out any element) raises(…);
boolean retrieve_element_set_to_next(out any element, out boolean more) raises(…);
boolean retrieve_next_n_elements (in unsigned long n, out AnySequence result, out boolean more) raises(…);
boolean not_equal_retrieve_element_set_to_next(in Iterator test, out any element) raises(…);
void remove_element() raises(…);
boolean remove_element_set_to_next() raises(…);
boolean remove_next_n_elements(in unsigned long n, out unsigned long actual_number) raises(…);
boolean not_equal_remove_element_set_to_next(in Iterator test) raises(…);
void replace_element(in any element) raises(…);
boolean replace_element_set_to_next(in any element) raises(…);
boolean replace_next_n_elements (in AnySequence elements, out unsigned long actual_number) raises(…);
boolean not_equal_replace_element_set_to_next(in Iterator test, in any element) raises(…);
boolean add_element_set_iterator(in any element) raises(…);
boolean add_n_elements_set_iterator (in AnySequence elements, out unsigned long actual_number) raises(…);
void invalidate();
boolean is_valid();
boolean is_in_between();
boolean is_for(in Collection collector);
boolean is_const();
boolean is_equal(in Iterator test) raises(…);
Iterator clone();
void assign(in Iterator from_where) raises(…);
void destroy();
};
```

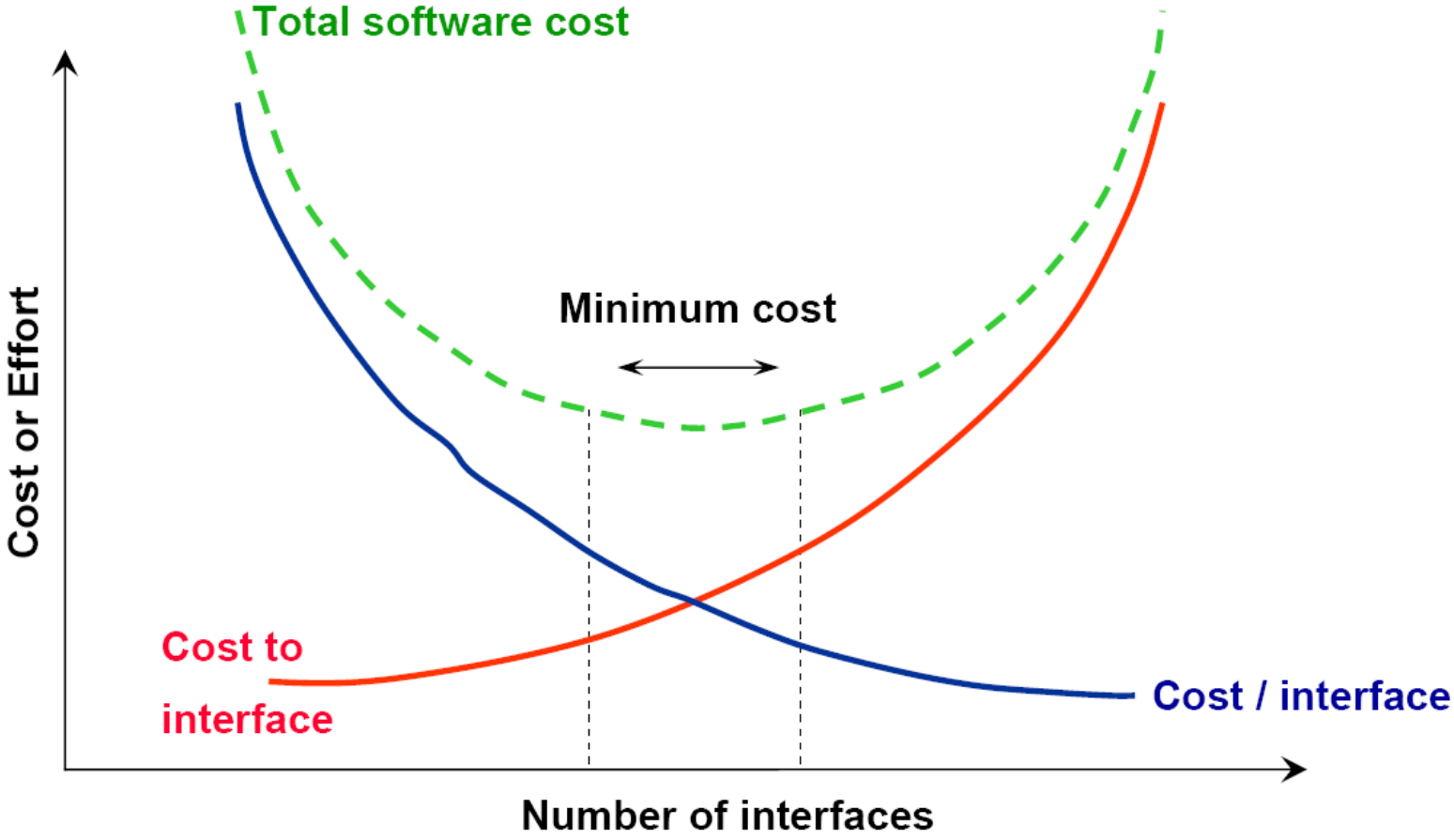
- This interface is from the Corba specification.
- Kevlin Henney calls this “Design by Committee”.

# A Well factored Interface

```
public interface IEnumerator
{
    bool MoveNext ();
    object Current {get;}
    void Reset ();
}
```

- This is the .Net interface similar to Corbas iterator interface.

# Balance number of Interfaces with development effort



# Designing Interfaces

- When factoring interface, think always in terms of reusable elements
- Example: a dog interface
- Requirements
  - Bark
  - Fetch
  - Veterinarian clinic registration number
  - A property for having received shots

# Remember The Common Reuse Principle

- Could define **IDog**

```
public interface IDog
{
    void Fetch();
    void Bark();
    long VetClinicNumber{ get; set; }
    bool HasShots{ get; set; }
}
public class Poodle : IDog
{…}
public class GermanShepherd : IDog
{…}
```

- BUT this interface is not well factored
  - **Bark()** and **Fetch()** are more logically related to each other than to **VetClinicNumber** and **HasShots**

# Better factoring

```
public interface IPet
{
    long VetClinicNumber{ get; set; }
    bool HasShots{ get; set; }
}
public interface IDog
{
    void Fetch();
    void Bark();
}
public interface ICat
{
    void Purr();
    void CatchMouse();
}
public class Poodle : IDog, IPet
{…}
public class Siamese : ICat, IPet
{…}
```

# Interfaces Factoring and Design

- If operations are logically related, but repeated in different interfaces → factor to hierarchy of interfaces

```
public interface IMammal
{
    void ShedFur();
    void Lactate();
}

public interface IDog :  IMammal
{
    void Fetch();
    void Bark();
}

public interface ICat : IMammal
{
    void Purr();
    void CatchMouse();
}
```

# Interfaces Factoring and Design

- Interface factoring results in interfaces with fewer members
- Balance out two counter forces
  - Too many granular interfaces Vs few complex, poorly factored interfaces
- Just one member is possible, but avoid it
  - Dull facet
  - Too many parameters
  - Too coarse: should be factored into several methods
  - Refactor into an existing interface
- Optimal number: 3 to 5
- Acceptable number: 2 and 6 to 12
- Never to be exceeded number: 20

# Interfaces Factoring and Design

- Ratio of methods, properties and events
  - Interfaces should have more methods than properties
  - Just-enough-encapsulation
  - Ratio of at least 2:1
  - Exception is interfaces with properties only
    - Should have no methods
  - Avoid defining events

# Interfaces Factoring and Design

- Is .Net Well-Factored?
- .NET Factoring Metrics
  - 300+ interfaces examined
  - On average, 3.75 members per interface
  - Methods to properties ratio of 3.5:1
  - Less than 3 percent of the members are events
  - On average, .NET interfaces are well factored

# Data Interfaces or Service Interfaces

- Data Interfaces
  - The methods in a data interface typically set or retrieve the values of some attributes in the implementing class.
- Service Interfaces
  - The methods in a service interface typically operate mostly on the parameters that are passed to it.
- You may design interfaces that is a mixture of the 2 extreme cases

# Stateless or Stateful Interfaces

- In a stateful interface, the methods operate differently based on the current state
  - which is changed by the sequence of method invocations
- In a stateless interface, the behavior is not dependent on the history of method invocations
  - Its behavior is always the same
- Typically we find that:
  - Data Interfaces are stateful
  - Service Interfaces are stateless

# Stateless or Stateful IF Example

- **MFC uses a stateful interface when outputting graphics and text to the graphic engine CDC:**

```
void CWindow::OnPaint()
{
   CPaintDC dc(this);
   CPen *oldPen;
   CPen pen(PS_SOLID, 2, RGB(0,0,255));
   oldPen = dc.SelectObject(&pen); // change pen in GDI-engine
   for (x=0; x<rect.Width(); x+=10) {
      dc.MoveTo(0, 0);
      dc.LineTo(x, rect.Height());
      }
   dc.SelectObject(oldpen);
}
```


- **.Net uses a stateless (nearly) interface when outputting graphics and text:**

```
protected override void OnPaint(PaintEventArgs e)
{
   Graphics g = e.Graphics;
   Pen pen = new Pen(Color.FromArgb(0,0,255), 2);
   for (int x = 0; x < ClientSize.Width; x += 10) {
      g.DrawLine(pen, 0, 0, x, ClientSize.Height);
   }
}
```

# Stateless versus Stateful Interfaces

- Stateless
  - Advantages
    - Order of method calls does not matter.
    - No risk of server and client states getting out of sync.
    - Easy for a server to handle multiple concurrent clients.
  - Disadvantage
    - Parameter lists are longer
- Stateful
  - Advantage
    - Parameter lists are shorter – less data to transfer on each call
  - Disadvantage
    - Order of method calls are important
    - Handling multiple concurrent clients need special considerations.
- It is possible to transform a stateful IF to a stateless IF, and you can also transform a stateless IF to a stateful IF.

# Push or Pull Interfaces

- Interfaces move data in one of two ways: push or pull.
  - **Pull style**:
    You poll through the interface for data – ex. `Int GetX()`

    ex.: whenever you type in an URL, you ask a webserver for information, which is then returned to you.

  - **Push style**:
    The client send a delegate (function pointer) through the interface, which is used to send the data back to the client.

    ex.: your email program receives mail, when someone sends you a an email.

# References

- Programming .NET Components
  by Juval Löwy
- InterfaceOriented Design
  by Ken Pugh