Item 19: Prefer Defining and Implementing Interfaces to Inheritance

Abstract base classes provide a common ancestor for a class hierarchy. An interface describes one atomic piece of functionality that can be implemented by a type. Each has its place, but it is a different place. Interfaces are a way to design by contract: A type that implements an interface must supply an implementation for expected methods. Abstract base classes provide a common abstraction for a set of related types. It's a cliché, but it's one that works: Inheritance means "is a," and interfaces means "behaves like." These clichés have lived so long because they provide a means to describe the differences in both constructs: Base classes describe what an object is; interfaces describe one way in which it behaves.

Interfaces describe a set of functionality, or a contract. You can create placeholders for anything in an interface: methods, properties,indexers, and events. Any type that implements the interface must supply concrete implementations of all elements defined in the interface. You must implement all methods, supply any and all property accessors and indexers, and define all events defined in the interface. You identify and factor reusable behavior into interfaces. You use interfaces as parameters and return values. You also have more chances to reuse code because unrelated types can implement interfaces. What's more, it's easier for other developers to implement an interface than it is to derive from a base class you've created.

What you can't do in an interface is provide implementation for any of these members. Interfaces contain no implementation whatsoever,and they cannot contain any concrete data members. You are declaring the contract that must be supported by all types that implement an interface.

Abstract base classes can supply some implementation for derived types, in addition to describing the common behavior. You can specify data members, concrete methods, implementation for virtual methods, properties, events, and indexers. A base class can provide implementation for some of the methods, thereby providing common implementation reuse. Any of the elements can be virtual, abstract, or nonvirtual. An abstract base class can provide an implementation for any concrete behavior; interfaces cannot.

This implementation reuse provides another benefit: If you add a method to the base class, all derived classes are automatically and implicitly enhanced. In that sense, base classes provide a way to extend the behavior of several types efficiently over time: By adding and implementing functionality in the base class, all derived classes immediately incorporate that behavior. Adding a member to an interface breaks all the classes that implement that interface. They will not contain the new method and will no longer compile. Each implementer must update that type to include the new member.

Choosing between an abstract base class and an interface is a question of how best to support your abstractions over time. Interfaces are fixed: You release an interface as a contract for a set of functionality that any type can implement. Base classes can be extended over time. Those extensions become part of every derived class.

The two models can be mixed to reuse implementation code while supporting multiple interfaces. One such example is `System.Collections.CollectionBase`.. This class provides a base class that you can use to shield clients from the lack of type safety in .NET collections. As such, it implements several interfaces on your behalf: `IList`, `ICollection`, and `IEnumerable`. In addition, it provides protected methods that you can override to customize the behavior for different uses. The `IList` interface contains the `Insert()` method to add a new object to a collection. Rather than provide your own implementation of `Insert`, you process those events by overriding the `OnInsert()` or `OnInsertCcomplete()` virtual methods of the `CollectionBase` class.

```
public class IntList : System.Collections.CollectionBase
{
  protected override void OnInsert( int index, object value )
  {
    try
    {
      int newValue = System.Convert.ToInt32( value );
      Console.WriteLine( "Inserting {0} at position {1}",
        index.ToString(), value.ToString());
        Console.WriteLine( "List Contains {0} items",
        this.List.Count.ToString());
    }
    catch( FormatException e )
    {
      throw new ArgumentException(
       "Argument Type not an integer",
       "value", e );
    }
  }

  protected override void OnInsertComplete( int index,
    object value )
  {
    Console.WriteLine( "Inserted {0} at position {1}",
      index.ToString( ), value.ToString( ));
    Console.WriteLine( "List Contains {0} items",
      this.List.Count.ToString( ) );
  }
}

public class MainProgram
{
  public static void Main()
  {
    IntList l = new IntList();
    IList il = l as IList;
    il.Insert( 0,3 );
    il.Insert( 0, "This is bad" );
  }
}
```

The previous code creates an integer array list and uses the `IList` interface pointer to add two different values to the collection. By overriding the `OnInsert()` method, the `IntList` class tests the type of the inserted value and throws an exception when the type is not an integer. The base class provides the default implementation and gives you hooks to specialize the behavior in your derived classes.

`CollectionBase`, the base class, gives you an implementation that you can use for your own classes. You need not write nearly as much code because you can make use of the common implementation provided. But the public API for `IntList` comes from the interfaces implemented by `CollectionBase`: the `IEnumerable`, `ICollection`, and `IList` interfaces. `CollectionBase` provides a common implementation for the interfaces that you can reuse.

That brings me to the topic of using interfaces as parameters and return values. An interface can be implemented by any number of unrelated types. Coding to interfaces provides greater flexibility to other developers than coding to base class types. That's important because of the single inheritance hierarchy that the .NET environment enforces.

These two methods perform the same task:

```
public void PrintCollection( IEnumerable collection )
{
  foreach( object o in collection )
  Console.WriteLine( "Collection contains {0}",
    o.ToString( ) );
}

public void PrintCollection( CollectionBase collection )
{
  foreach( object o in collection )
  Console.WriteLine( "Collection contains {0}",
    o.ToString( ) );
}
```

The second method is far less reusable. It cannot be used with `Arrays`, `ArrayLists`, `DataTables`, `Hashtables`, `ImageLists`, or many other collection classes. Coding the method using interfaces as its parameter types is far more generic and far easier to reuse.

Using interfaces to define the APIs for a class also provides greater flexibility. For example, many applications use a `DataSet` to transfer data between the components of your application. It's too easy to code that assumption into place permanently:

```
public DataSet TheCollection
{
  get { return _dataSetCollection; }
}
```

That leaves you vulnerable to future problems. At some point, you might change from using a `DataSet` to exposing one `DataTable`, using a

DataView, or even creating your own custom object. Any of those changes will break the code. Sure, you can change the parameter type, but that's changing the public interface to your class. Changing the public interface to a class causes you to make many more changes to a large system; you would need to change all the locations where the public property was accessed.

The second problem is more immediate and more troubling: The DataSet class provides numerous methods to change the data it contains. Users of your class could delete tables, modify columns, or even replace every object in the DataSet. That's almost certainly not your intent. Luckily, you can limit the capabilities of the users of your class. Instead of returning a reference to the DataSet type, you should return the interface you intend clients to use. The DataSet supports the IListSource interface, which it uses for data binding:

```
using System.ComponentModel;

public IListSource TheCollection
{
  get { return _dataSetCollection as IListSource; }
}
```

IListSource lets clients view items through the GetList() method. It also has a ContainsListCollection property so that users can modify the overall structure of the collection. Using the IListSource interface, the individual items in the DataSet can be accessed, but the overall structure of the DataSet cannot be modified. Also, the caller cannot use the DataSet's methods to change the available actions on the data by removing constraints or adding capabilities.

When your type exposes properties as class types, it exposes the entire interface to that class. Using interfaces, you can choose toexpose only the methods and properties you want clients to use. The class used to implement the interface is an implementation detail that can change over time (see Item 23).

Furthermore, unrelated types can implement the same interface. Suppose you're building an application that manages employees, customers, and vendors. Those are unrelated, at least in terms of the class hierarchy. But they share some common functionality. They all have names, and you will likely display those names in Windows controls in your applications.

```
public class Employee
{
  public string Name
  {
    get
    {
      return string.Format( "{0}, {1}", _last, _first );
    }
  }

  // other details elided.
}

public class Customer
{
  public string Name
  {
    get
    {
      return _customerName;
    }
  }

  // other details elided
}

public class Vendor
{
  public string Name
  {
    get
    {
      return _vendorName;
    }
  }
}
```

The `Employee`, `Customer`, and `Vendor` classes should not share a common base class. But they do share some properties: names (as shown earlier), addresses, and contact phone numbers. You could factor out those properties into an interface:

```
public interface IContactInfo
{
  string Name { get; }
  PhoneNumber PrimaryContact { get; }
  PhoneNumber Fax { get; }
  Address PrimaryAddress { get; }
}

public class Employee : IContactInfo
{
  // implementation deleted.
}
```

This new interface can simplify your programming tasks by letting you build common routines for unrelated types:

```
public void PrintMailingLabel( IContactInfo ic )
{
  // implementation deleted.
}
```

This one routine works for all entities that implement the `IContactInfo` interface. `Customer`, `Employee`, and `Vendor` all use the same routine—but only because you factored them into interfaces.

Using interfaces also means that you can occasionally save an unboxing penalty for structs. When you place a struct in a box, the box supports all interfaces that the struct supports. When you access the struct through the interface pointer, you don't have to unbox the struct to access that object. To illustrate, imagine this struct that defines a link and a description:

```
public struct URLInfo : IComparable
{
  private string URL;
  private string description;

  public int CompareTo( object o )
  {
    if (o is URLInfo)
    {
      URLInfo other = ( URLInfo ) o;
      return CompareTo( other );
    }
    else
      throw new ArgumentException(
        "Compared object is not URLInfo" );
  }

  public int CompareTo( URLInfo other )
  {
    return URL.CompareTo( other.URL );
  }
}
```

You can create a sorted list of `URLInfo` objects because `URLInfo` implements `IComparable`. The `URLInfo` structs get boxed when added to the list. But the `Sort()` method does not need to unbox both objects to call `CompareTo()`. You still need to unbox the argument `(other)`, but you don't need to unbox the left side of the compare to call the `IComparable.CompareTo()` method.

Base classes describe and implementcommon behaviors across related concrete types. Interfaces describe atomic pieces of functionality that unrelated concrete types can implement. Both have their place. Classes define the types you create. Interfaces describe the behavior of those types as pieces of functionality. If you understand the differences, you will create more expressive designs that are more resilient in the face of change. Use class hierarchies to define related types. Expose functionality using interfaces implemented across those types.

Item 20: Distinguish Between Implementing Interfaces and Overriding Virtual Functions

At first glance, implementing an interface seems to be the same as overriding avirtual function. You provide a definition for a member that has been declared in another type. That first glance is very deceiving. Implementing an interface is very different from overriding a virtual function. Members declared in interfaces are not virtual—at least, not by default.