Разширеното разбиране за интерфейси в .NET

* Когато не искаме да реализираме наследство на типове, но общ договор за поведение.
* CLR поддържа единично наследяване на имплементация и множествено на интерфейси
•Интерфейсите в C# могат да дефинират и събития, свойства без и с параметри, тъй като те са всъщност методи
• според CLR интерфейсите могат да притежават и статични методи, полета и конструктори (на типа), но някои езици (C#) не го позволяват.
* интерфейсите се маркират с public, protected, private, internal (както класовете или структурите)
* ако не укажете метод в имплементацията му с virtual, той се счита virtual, sealed
* типът трябва да имплементира всички методи на наследените интерфейси. В противен случай той е абстрактен (незавършен) тип.
* обекти от даден тип могат да се третират и като някой от неговите интерейси:

```
String s = “Student”;
//сега могат да се викат методите на  интерфейсите имплементирани
// в типа String и Object: IComparable; ICloneable; IConvertible, IEnumerable
IComparable comparable = s;
// с горната пром. могат да се викат само методите на IComparable
ICloneable cloneable = s;
// променлива от тип интерфейс може да се преобразува в друг
// интерфейс, ако и двата се поддържат в типа
IEnumerable enumerable = (IEnumerable) cloneable;
```

- Всички действия изпълнение с comparable, cloneable, enumerable или s, всъщност влияят на обекта “s”. Типът на променливата определя само допустимите действия, които се изпълняват с нея
- Реализацията на наследен интерф.метод отразява спецификата на конкретния обект, но при задължителна сигнатура на метода. Ако е необходимо, кодът прави нужните преобразувания в типа (напр. за IComparable())

пример на 3 дефинирани в FCL (Framework Class Library) интерфейса:

```
public interface System. Collection.IEnumerable
{           IEnumerator GetEnumerator();     }
public interface System.Collections.IEnumerator
{
            Boolean MoveNext();
            void Reset();
            Object Current   { get;}              //свойство за четене
}
public interface System.IComparable
{
            Int32 CompareTo (Object object);
}
```

Тип или интерфейс? Някои съображения:

- кое е важното: IS-A? или CAN DO (интерфейс)?
- Лекота на използване – наследявате готовата функционалност на типа
- Устойчива имплементация – интерфейсът всеки път се имплементира с възможност за грешки
- Версии – промени в базовия тип се наследяват без необходимост от промени в производния. Дори не е нужна прекомпилация.

Проблеми с промяна на полета в boxed value тип и решаване посредством интерфейси

ето проблемът:

```
using System;
struct Point
{           public Int32 x,y;
            public void Change(Int32 x, Int32 y) { this.x = x; this.y = y;          }
            public override String ToString()
                        { return String.Format( “{0}, {1}”, x, y);                          }
}
class App
            { static void Main()
                        {
                        Point p = new Point();
                        p.x = p.y = 1;
                        Console.WriteLine(p);          // (1,1)
                        p.Change(2, 2);
                        Console.WriteLine(p);          // (2, 2)
                        Object o = p;                                   // boxing
                        Console.WriteLine( o );        (2, 2)
                        (( Point) o).Change(3, 3);     // защото Change() е метод на Point
// “ o “ се разопакова. Полетата му се копират в нов value type Point в стека
                                                       //и се променят на 3
                        Console.WriteLine( o );        // (2, 2) – това е “пакетираният” обект
// C++ позволява промяна в полета на опакован тип, но C# - не. Решение:
```

ето решението с интерфейси:

```
using System;
interface IChangeBoxedPoint {
        void Change(Int32 x, Int32 y);
        }
struct Point  : IChangeBoxedPoint
{       public Int32 x,y;
        public void Change(Int32 x, Int32 y) { this.x = x; this.y = y;          }
        public override String ToString()
                { return Strinf.Format( “{0}, {1}”, x, y);                      }
}
class App
        { static void Main()
                {
                Point p = new Point();
                p.x = p.y = 1;
                Console.WriteLine(p);          // (1,1)
                p.Change(2, 2);
                Console.WriteLine(p);          // (2, 2)
                Object o = p;                                  // boxing
                Console.WriteLine( o );        // (2, 2)
                (( Point) o).Change(3, 3); // защото Change() е метод на Point
// “ o “ се разопакова. Полетата му се копират във value type Point в стека и се
// променят на 3
```

```
Console.WriteLine( o );
// (2, 2) – това е “пакетираният” обект   ‘o’

((IChangeBoxedPoint) p).Change(4,4);
                    // “p” се преобразува до интерфейс.
// Променят се стойности на 4,4, но обектът не се реферира
// повече и се маркира за излишен (garbage collector го унищ.)
Console.WriteLine(p);      // 2,2

((IChangeBoxedPoint) o).Change(5,5);
Console.WriteLine( o );    // (5,5)
 // имаме преобразуване, не пакетиране или препакетиране!!!
//  обръщението е към пакетирания вече  point
// извикването на Change() се отнася към него и всичко е ОК
```

Нека

Имплементираме повече от 1 интерфейси, имащи еднакъв метод

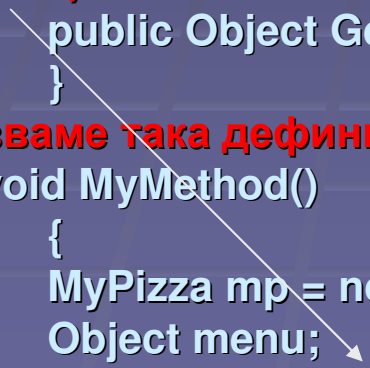
```
public interface IWindow
        { Object GetMenu();                    }
public interface IRestaurant
        { Object GetMenu();                    }
```

дефинираме тип:

```
public class MyPizza : IWindow, IRestaurant
        {
        Object IWindow.GetMenu()          {.....................}
        Oblect IRestaurant.GetMenu()      { ....................}
 //обърнете внимание – интерфейсните методи не са public дефинирани
// те са понякога public, понякога – private (както ще видим в обръщенията
// към тях )
        public Object GetMenu()           {.....................}
        }
```

използваме така дефинирания тип в някакъв наш метод:

```
static void MyMethod()
        {
        MyPizza mp = new MyPizza();       // инстанция на типа
        Object menu;
```



```
menu = mp.GetMenu(); //това е единствения явно дефиниран public
menu = ((IWindow) mp).GetMenu();
menu = ((IRestaurant) mp).GetMenu();
}
```

Явна имплементация на интерфейсни членове

когато базов интерфейс работи с параметри от даден тип, наследяващият го следва да работи с същите типове. Например:

```
public interface IComparable
        { Int32 CompareTo( Object other);      }          // сравнява обекти
```

създаваме наш тип:

```
struct SomeValuetype : IComparable
{       private Int32 x; public SomeValueType(Int32 x) {this.x = x;}
        public Int32 CompareTo( Object other) {return(x-((SomeValueType)other).x);}
```

/ * можем да сравняваме обекти, а не на наши типове. Лошото е че компилаторът не открива грешката ако напишем:*/

.Нека напишем следния код:

```
SomeValueType v = new SomeValueType(0);
Object o = new Object();
Int32 n = v.CompareTo(o); // грешка!
```

къде в 'o' има член 'x'?

- решение на проблема е "явна имплементация на член на интерфейс":
- В нашия тип следва да имаме 2 имплементации на CompareTo(). Едната сравнява обекти (както е в сигнатурата на метода на наследения интерфейс) и препраща към собствена на типа реализация на CompareTo(). В нея се сравняват исканите (конкретни) стойности.
- Така компилаторът "хваща " грешката при подаден неправилен тип:

```
struct SomeValueType : IComparable {
private Int32 x;
public SomeValueType(Int32 x) {this.x = x;}

public Int32 CompareTo(SomeValueType other) {
return(x – other.x);}

// явна имплементация на интерфейсния метод
Int32 IComparable.CompareTo(Object other) //името е пълно!!
{ return CompareTo((SomeValueType) other);                }
```

Използването сега е типово-обезопасено:

```
SomeValueType x1 = new SomeValueType(1);
SomeValueType x2 = new SomeValueType(2);
Int32 n;

n = x1.CompareTo(new Object()); //грешка при компилация
n = x1.CompareTo(x2);         //OK

IComparable comp = (IComparable) x1;
n = comp.CompareTo(x2);    // x2 се пакетира и всичко е ОК
```

Бележка за обмисляне:

Имплементацията IComparable.CompareTo() не може да има модификатор за достъп public или private. Тя е понякога public (когато работим с обекти IComparable), понякога private (когато работим с SomeValuetype)!!

Целта е осигуряване на типова обезопасеност!

Друга възможна цел на явните имплементации е избягване на нежелани операции по пакетиране, като се вика подходящата имплементация

Съвместимост на COM IDL дефиниции с CLR (Common Language Runtime) Damien 99

- (COM Interop) за съвместимост на COM и CLR
- работи се с метаданните
- създава прокси обекти в клиентския и в сървърен процес. Той подменя работата с QueryInterface(), AddRef() или Release() от COM технологията.
- tools подпомагащи прехода COM ← → components in CLR:

  - Type Library Importer (tlbimp.exe): от type library информация → създава assembly с метаданново описание за компонента
  - Type Library Exporter (tlbexp.exe): type library ← CLR assembly
  - Register Assembly (regasm.exe): запис в Registry на информация за асемблито, така че COM компонентите могат да виждат и ползват обекти от него.

```
пример:
имаме COM компонент HelloATL  с IDL файлова дефиниция:
import “ oaidl.idl”;
[ object, uuid (FFO3 ……),
]
interface IHelloATL : IDispatch
{
        HRESULT Hello([in] BSTR bstr);
}
library HELLOATLLib
{
        importlib(“stdole32.tlb”);
        coclass HelloATL
        {
        [default] interface IHelloATL;
        };
};
```

- компилираме idl файла. Получаваме типова библиотека.
- type library importer с тази библиотека създава CLR файл
- създадени са нужните метаданни: 2 CLR интерфейса (HelloATL и IHelloATL) и CLR клас – HelloATLClass, съобразно COM описанието. Асемблито е в DLL – HELLOATLLib.dll
- всъщност са създадени proxy за COM компонента – цялата работа продължава да се изпълнява от него

ако разгледаме типовото описание от метаданните:

```
HelloATL h = new HelloATL();
Type t = h.GetType();
MethodInfo[ ] ma = t.GetMethods();
```

за новосъздадения тип – HelloATL ще видим следните методи:

```
Void Hello(System.String)            // нашият метод от idl интерфейса

System.Runtime.Remoting.ObjRef CreateObjRef(System.Type)
                                     // за създаване на proxy от дадения тип
Boolean Equals(System.Object)        // станд. – за сравняване на типове
System.String ToString()             // станд. – за връщане името на типа
System.Type GetType()                // стандартен – връща обект Type с
                                     // описания на типовите метаданни
```