# Seminario de C Sharp (Patrones de Diseño)

Luis Alejandro Lara
Jose Alejandro Labourdette
Nadia Gonzalez
Carlos Arrieta
Thalia Blanco
Grupo C-312

## 1 Mixins

Los mixins son clases destinadas solamente a contener funcionalidades a ser heredadas por otras subclases. Por esto, no están pensados para ser instanciados como objetos, sino para establecer una colección de funciones que queremos que estén incluidas en varias subclases. Por tanto, no es una forma de especialización, sino de coleccionar funcionalidades. En pocas palabras, se puede ver el mixin como una interfaz que contiene funcionalidades implementadas.
La programación con mixins está orientada a agrupar funcionalidades en clases(mixins), para luego poder heredar de uno o varios mixins si queremos usar sus funcionalidades. Esto es útil cuando se quiere reutilizar código, porque evita los conflictos que trae consigo la herencia múltiple.

### 1.1 Uso en C-sharp

Las versiones de C-sharp antes de la versión 8.0 no permite implementar métodos en interfaces. Para versiones anteriores, en particular para la versión 3.0, se puede hacer de otra forma: se implementa una interfaz con el nombre del *Mixin* y luego se le agregan métodos extensores.
Esto nos permite simular la implementación de métodos a las interfaces. Luego toda clase que implemente esta interfaz tendrá acceso a estos métodos.

#### 1.1.1 Código

Para ilustrar un ejemplo sencillo de cómo se implementa un *mixin*:

```
class Program
{
        static void Main(string[] args)
        {
                Numero n = new Numero();
                Console.WriteLine(n.ObtenerNumeroRandom());
                n.Imprimir_HelloWorld();
        }
}
```

```
public class Numero : MNumero{}

public interface MNumero{}

public static class MetodosExtensores
{
        //El this hace que el metodo estatico se enlace con el tipo MNumero
        public static int ObtenerNumeroRandom(this MNumero numero)
        {
                Random r = new Random();
                return r.Next();
        }

        public static void Imprimir_HelloWorld(this MNumero numero)
        {
                Console.WriteLine("Hello World");
        }
}
```

## 1.2 Creación de un árbol con mixin

A continuación se implementará una estructura de tipo árbol binario utilizando *mixins*.
Para ello se crearán dos *mixins*: uno llamado $MAltura$, que contendrá la funcionalidad de calcular la altura de un nodo del árbol, y otro llamado $MIterable$, que implementa un método para devolver un $IEnumerable$ de los valores de un nodo del árbol en $entre - orden$.
Nótese que los métodos extensores que se le asignan a las interfaces $MAltura$ y $MIterable$ que representan los *mixins*, se encuentran en la clase estática $Extensiones$:

```
public interface IArbolBinario
{
        IArbolBinario izquierdo { get; set; }
        IArbolBinario derecho { get; set; }
        int valor { get; set; }
}
public interface IArbolIterable : IArbolBinario { }
public interface IAltura : IArbolBinario { }
public interface IArbolIterableConAltura : IAltura, IArbolIterable { }

public static class Extensiones
{
        public static IEnumerable<int> EntreOrden(this IArbolIterable nodo)
        {
                if (nodo == null)
                yield break;

                foreach (var i in ((IArbolIterable)nodo.izquierdo).EntreOrden())
```

```
                yield return i;

                yield return nodo.valor;

                foreach (var i in ((IArbolIterable)nodo.derecho).EntreOrden())
                yield return i;
        }

        public static int Altura(this IAltura nodo)
        {
                if (nodo == null)
                        return -1;
                return Math.Max(((IAltura)nodo.derecho).Altura(),
                ((IAltura)nodo.izquierdo).Altura()) + 1;
        }
}
```

En este ejemplo vemos cómo los *mixins* nos ayudan a implementar dos funcionalidades distintas por separado. Esto se hace extremadamente útil si queremos establecer ciertas funcionalidades por separado, de forma tal que una clase pueda implementar cualquier subconjunto de ellas.

## 1.3 Mixins vs Herencia Múltiple

El ejemplo anterior se podría haber implementado en otro lenguaje como $c++$ utilizando herencia múltiple. Esto tiene ciertas ventajas y desventajas:

Ventajas de la Herencia Múltiple:

- Cada clase de la jerarquía define un concepto
- Se pueden instanciar las funcionalidades de las clases

Ventajas de los Mixins:

- Permite proveer funcionalidades a tipos ya definidos
- Agrupan funcionalidades que pueden ser utilizadas por tipos que no tengan relación de herencia entre ellos
- Al usar interfaces elimina la posibilidad de caer en situaciones como el problema del diamante
- Encapsula pequeñas funcionalidades, lo cual minimiza la repetición de código

## 1.4 ¿Tienen sentido/utilidad los mixins en un lenguaje con herencia múltiple?

En los lenguajes orientados a objetos, un mixin es una clase que contiene métodos para ser usados por otras clases sin necesidad de ser esta su padre. La utilización de mixins puede eliminar ambigüedades de la herencia como la del diamante.

### 1.5 Si se tiene una clase C que hereda de A y de B (herencia múltiple, por ejemplo en C++) Cómo refactorizar dicho código para hacerlo compilar en c sharp?

```
public interface IA {
        int a {get; set;}
        void A ();
}
public interface IB {
        int b {get; set;}
        void B ();
}

public static class ExtensionMethods{
        public static void A(this IA a) { ... }
        public static void B(this IB b) { ... }
}

public class C: IA, IB { ... }
```

### 1.6 Desventajas del uso de los métodos extensores

- La forma en la que los métodos extensores son importados. Es imposible obtener un método extensor de un namespace evitando cargar completamente el mismo.
- Si se añade un método extensor con el mismo nombre y signatura que un método de interfaz o clase, este nunca se llamará. Los métodos de extensión siempre tienen menos prioridad que los métodos de instancia definidos en el propio tipo.
- Los métodos de extensión no pueden tener acceso a las variables privadas en el tipo que extienden

## 2 Patrones de diseño

Existe un conjunto de patrones y principios que promueven el diseño de un código limpio y organizado utilizando la programación orientada a objetos.

### 2.1 Diseño desacoplado

Se establece que una de las formas de lograr un código limpio y organizado es por el diseño desacoplado, que consiste en programar con interfaces.
Entre las ventajas de este diseño desacoplado podemos citar:

- Detección de errores. Al tener bien desacopladas las funcionalidades de la aplicación, si apareciera algún error se podría buscar en la parte concerniente al error, sin modificar el resto de las partes, o sea, se reduciría el campo de búsqueda y facilitaría la tarea de encontrar el error.
- Mantenimiento. Se facilitaría el trabajo de sacar nuevas versiones y modificar las existentes.

- División del trabajo. Al estar bien definidas y desacopladas todas las partes de la aplicación se pudiera dividir el trabajo de desarrollo entre más personas.
- Mejora la legibilidad del código. Permite concentrarse solamente en una sección de la solución que se esté implementando, abstrayéndose del resto del código.
- Extensibilidad. Permite la reutilización de código para otros proyectos.

Para poner un ejemplo de diseño desacoplado:
//Mal Uso

```
class ConsoleLogger{
        public void Log(string message) {
                Console.WriteLine(message);
        }
}

class FileLogger{
        public void Log(string message) {
                //...
        }
}
```

//Código desacoplado
//(Loosely Coupled)

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

class ConsoleLogger: ILogger{
        public void Log(string message) {
                Console.WriteLine(message);
        }
}

class FileLogger: ILogger {
        public void Log(string message) {
                //...
        }
}
```

## 2.2 Inyección de dependencias

Cuando una clase requiere de una instancia de otra clase para su funcionamiento, dicha instancia se le debiera pasar como parámetro al constructor, y así adaptarlo a la interfaz que recibe, añadiéndole flexibilidad al código.
Un ejemplo de esta práctica sería el siguiente:

//Mal Uso

```
        public void Action() {
                ILogger logger = new ConsoleLogger();
                logger.Log("Bark");
        }
}

//Inyección por construcción

class DogMartinFlower {
        private ILogger _logger;
        public Dog(ILloger logger){
                _logger = logger
        }

        public void Action() {
                _logger.Log("Bark")
        }
}
```

En el caso del primer código, si se quisiera hacer la acción *Action*() por otro medio(por *FileLogger* en vez de *ConsoleLogger*) habría que implementar de nuevo la clase Dog, con el nuevo comportamiento, al contrario, el código de la derecha está más general, ya que en dependencia de la implementación de la interfaz *Ilogger* que se le pase al constructor será el comportamiento de la instancia.

## 2.3 Contenedores con Inversión del Control

Los contenedores con "Inversión del control" (*IoCcontainers*) es un estilo de programación en el cual el programador le otorga la responsabilidad de la creación de los objetos a una *entidad* llamada *Contenedor*. Un ejemplo de esto es el siguiente:

```
interface IContainer {
        void Register<T>(Type implementation);
        T Resolve<T>();

}

class Container: IContainer { ... }

interface IAnimal { void Action(); }

class Dog: IAnimal {
        private ILogger _logger;
        public Dog(ILogger logger) { _logger = logger; }

        public void Action() {
                _logger.Log("Bark");
        }
}
```

```
class Cat: IAnimal {
        private ILogger _logger;
        public Dog(ILogger logger) { _logger = logger; }

        public void Action() {
                _logger.Log("Miau");
        }
}

class Cow: IAnimal { ... } // Muu

var container = new Container();
container.Register<ILogger>(typeof(ConsoleLogger));
container.Register<IAnimal>(typeof(Dog));
var animal = container.Resolve<IAnimal>();
animal.Action();
```

En la clase *Container* se implementan los métodos *Register* y *Resolve*:
El método *Register* genérico en T, registra el tipo que recibe de parámetro como el tipo que implementa $T$. Esto lo que hace es asignarle el tipo *implementador* a la interfaz $T$. En otras palabras, cada vez que se pida crear un objeto que implemente la interfaz $T$, se creará uno de tipo *implementador*.
El método *Resolve* no recibe nada y devuelve una instancia de un objeto que implementa $T$.
Por tanto, en el código anterior, después de declarado el *Container*, se ejecutan dos acciones *Register*. La primera le asigna el tipo *ConsoleLogger* a la interfaz *ILogger*, y la segunda le asigna *Dog* a *IAnimal*. En la próxima instrucción se manda a crear una instancia de un objeto que implemente la interfaz *IAnimal* y se le asigna a la variable *animal*. Como la interfaz *IAnimal* tiene asignado el tipo *Dog*, se va a crear un objeto de tipo *Dog* y se le va a asignar a la variable *animal*.
El constructor de la clase *Dog* recibe un objeto de tipo *ILogger*, por tanto el *Container* primero instancia un objeto de tipo *ILogger* y luego se lo pasa como parámetro al constructor de *Dog*. Para instanciar el objeto de tipo *ILogger*, el *Container* llama recursivamente a *Resolve* pasándole como parámetro *ILogger*. Esto va a crear un objeto de tipo *ConsoleLogger*, que es el tipo asignado a *ILogger*.
Luego de todo este procedimiento, estará creado un objeto de tipo *Dog* que contiene un objeto de tipo *ConsoleLogger*. Luego al realizar la acción *animal.Action*() se va a desplegar por consola el texto "Bark".

### 2.3.1 Implementación del Container

A continuación se implementan los métodos *Register* y *Resolve* de la clase *Container*:

```
class Container : IContainer
{
        private Dictionary<Type, Type> diccionario;
```

```
        private HashSet<Type> dependencias;

        public Container() {
                diccionario = new Dictionary<Type, Type>();
        }

        public void Register<T>(Type implementador) {
                Type tipo_interfaz = typeof(T);

                if (!tipo_interfaz.IsInterface)
                throw new Exception(String.Format("{0} debe ser una interfaz", tipo_interfaz.Name));

                if (implementador.GetInterface(tipo_interfaz.Name) == null)
                throw new Exception(String.Format("{0} debe implementar {1}",
                implementador.Name, tipo_interfaz.Name));

                diccionario[tipo_interfaz] = implementador;
        }

        public T Resolve<T>() {
                dependencias = new HashSet<Type>();
                return (T)Resolve(typeof(T));
        }

        private object Resolve(Type interfaz) {
                if (dependencias.Contains(interfaz))
                        throw new Exception(String.Format("Implementador de {0} tiene
                        dependencia circular", interfaz.Name));

                if (!diccionario.ContainsKey(interfaz))
                        return null;

                Type implementador = diccionario[interfaz];
                ConstructorInfo constructor = implementador.GetConstructors()[0];
                ParameterInfo[] parametros = constructor.GetParameters();

                if (parametros.Length == 0)
                        return Activator.CreateInstance(implementador);

                object[] parametros_resueltos = new object[parametros.Length];
                for (int i = 0; i < parametros.Length; i++)
                        parametros_resueltos[i] = Resolve(parametros[i].ParameterType);

                return constructor.Invoke(parametros_resueltos);
        }
}

public interface IContainer
{
        void Register<T>(Type implementador);
```

```
    T Resolve<T>();
}
```

Este código está implementado para que las siguientes líneas lancen excepción, ya que se verifica en *Register* que el tipo *implementation* implemente la interfaz $T$, y que $T$ sea una interfaz válida:

```
container.Register<ILogger>(typeof(Dog))
container.Register<Dog>(typeof(Wolf)); // Wolf hereda de Dog
```

### 2.3.2 Características de C-sharp

A continuación se explican algunas de las características de $C-sharp$ utilizadas en el ejercicio anterior:

- $Reflection$: La habilidad de descripción completa de tipos, usando metadatos, es un elemento clave de la plataforma $.NET$. Es muy útil este recurso en la serialización de objetos que requiere conocer el formato de tipos en tiempo de ejecución, y para la capacidad $IntelliSense$ de un $IDE$ que descansa en una descripción de tipos concreta. Puede ser usada para crear, en tiempo de ejecución, una instancia de un tipo. Puede obtener el tipo de un objeto existente e invocar sus métodos o acceder a sus campos y propiedades, etc.
  Creando una instancia de un objeto en tiempo de ejecución y trabajando con sus funcionalidades:

```
class Program
{
        static void Main(string[] args)
        {
                Type testType = typeof(TestClass);
                ConstructorInfo ctor = testType.GetConstructor(Type.EmptyTypes);
                if(ctor != null)
                {
                        object instance = ctor.Invoke(null);
                        MethodInfo methodInfo = testType.GetMethod("TestMethod");
                        Console.WriteLine(methodInfo.Invoke(instance,new object[] { 10 }));
                }
        }
}

public class TestClass
{
        private int testValue = 42;
        public int TestMethod(int numberToAdd)
        {
                return this.testValue + numberToAdd;
        }
}
```

- $typeof$: Es una de las formas de obtener información de Tipos en $C-sharp$. A diferencia de $Object.GetType()$ no existe la necesidad de crear

una nueva instancia para extraer la información del tipo.
La sintaxis para su utilización es la siguiente:
$Type\ t = typeof(< Object >)$;

- *Genericidad*: Permite definir funcionalidades que no se limiten a un tipo especifico. En $C-sharp$ pueden definirse métodos y clases genéricas para permitirlo.
  Digamos por ejemplo que se necesite intercambiar el contenido de dos variables de igual Tipo. Definamos entonces un método *swap* genérico.

  ```
  static void Swap<T>(ref T first, ref T second)
  {
          T temp = first;
          first = second;
          second = temp;
  }
  ```

  De esta forma independientemente del tipo en el que esten definidas las variables, para cualquier par, se podrán *swapear*.
  La sintaxis es la siguiente:

  ```
  Method_or_Class_name<T>(args)
  ```

- *Herencia*: Es uno de los atributos fundamentales de la programación orientada a objetos. Permite definir clases hijas que reusan, extienden o modifican el comportamiento de la clase Padre. La clase cuyos miembros se heredan se denomina Base, la otra se llama clase derivada.
  $C-sharp$ y $.Net$ solo admiten la herencia única, sin embargo la herencia es transitiva por lo que se puede definir una jerarquía de herencia para un conjunto de tipos. Las clases pueden definirse como abstractas lo que no permite una instanciación de un objeto de dicho tipo, para acceder a sus funcionalidades se necesitaría una instancia de un tipo derivado.
  Se introducen dos notaciones para las propiedades, *internal* que permite que solo sea accesible por las clases que estén en el mismo *namespace* y *protected* para que sea visible solo por las clases derivadas, independientemente de su *scope*.