PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN
IIC 1102 INTRODUCCIÓN A LA PROGRAMACIÓN
Profesor: Rodrigo Sandoval U.

Capítulo IV – Programación Orientada a Objetos en C#

1 Programación Orientada a Objetos en C#

Siguiendo la idea de la programación orientada a objetos, y tomando como base la plataforma .NET y el lenguaje C#, a continuación se muestran algunas potencias de este lenguaje en implementar objetos con ciertas particularidades adicionales a lo que otros lenguajes ofrecen.

1.1 Namespaces

En los programas en diversos lenguajes, particularmente en los que se basan en un esquema orientado a objetos, se preferencia fuertemente la referencia a librerías externas, compuestas de elementos (en este caso de clases), que pueden resolver algunos elementos de tipo general en el programa en desarrollo.

1.1.1 Referencia de Namespaces

En C# particularmente, se hace referencia a una serie de colecciones de clases que resuelven funcionalidades requeridas por casi todos los programas. Estas colecciones de clases externas, que pueden haber sido programadas y compiladas en forma individual y ajena al programa que las referencia, se conocen en inglés como *Namespaces*.

Un ejemplo extremadamente omnipresente de namespace es la referencia System, el cual es una colección de clases relacionadas con el sistema (computador) de uso bastante frecuente. Entre sus clases más recurridas en los ejemplos de este curso, se encuentra la clase Console, que hace referencia a los métodos de escritura y lectura hacia la pantalla y desde el teclado (que componen la console).

Para hacer referencia a un namespace, se hace uso de la directiva *using* (que es un equivalente al #include del lenguaje C).

Es decir, un programa que hace acceso a la consola, y que referencia al namespace System, se ve así:

```
using System;

class MainProgram {

   // El método Main() es el algoritmo principal.
   public static void Main() {

      // Escribe texto en la consola.
      Console.WriteLine("Hola Mundo!");
      Console.ReadLine(); // Espera Enter para terminar
   }
}
```

Muchos de los namespaces incluidos en el framework de .NET, se referencian con nombres “fuertes”, es decir, que son nombrados de una forma que indirectamente estructuran una organización. Es así, como aparte del recurrido namespace System, existen otros que semánticamente se derivan de él, como es el que provee las clases para el acceso a archivos (con métodos similares a la clase Console), que se conoce como System.IO. Si bien es un namespace totalmente distinto, su nombre completo, compuesto de dos partes separadas por un punto, da una referencia semántica de su relación con el namespace System.

1.1.2 Creación de Namespaces

Por otro lado, cada programador puede crear sus propios namespaces, donde en forma lógica se agrupan clases que se complementan en su funcionalidad. Para ello, la declaración es de la siguiente forma.

```
namespace MiLibreria {
   class Clase1 {
   ...
   }
   class Clase2 {
   ...
   }
}
```

El uso de este namespace en un programa compuesto por otros archivos requiere la compilación conjunto, o bien la creación de una librería dinámica (DLL), que se referencia en el segundo archivo, que contiene el programa principal.

1.2 Constructores

Los constructores son un tipo particular de métodos pertenecientes a una clase. El constructor es siempre y únicamente invocado al instanciar una clase y proporcionan la capacidad de especificar las acciones a tomar para inicializar la instancia en particular. El constructor es un concepto establecido en prácticamente todos los lenguajes de programación OO, incluyendo C#.

En pocas palabras, el propósito del constructor es concentrar la lógica de inicialización de una instancia cuando ésta es creada. En muchos casos se define que el constructor recibe ciertos parámetros que condicionan esta inicialización, y que muchas veces se traducen en valores iniciales para los atributos de la instancia.

1.2.1 Sintaxis de Definición de Constructores

La declaración de los constructores es la siguiente:

```
public <NombreClase> (<ListaParámetros>) {
     <instrucciones>
}
```

- Por definición, los constructores son públicos, ya que son ejecutados al instanciar la clase, sin importar si sus otros elementos son públicos o privados. Por ello, siempre se indica “public”.
- Valor de retorno no se declara. No son void, ni ningún otro tipo de dato, ya que por definición conceptual del constructor, sólo devuelve una instancia creada del objeto o clase que representa.
- El nombre del constructor es siempre el nombre de la clase respectiva.
- Puede recibir los parámetros que se estimen convenientes en cada caso. Es incluso factible definir más de un constructor para una misma clase, cambiando sólo los parámetros recibidos. Esto cae en la categoría de sobrecarga de métodos, en este caso sobrecarga de constructores.
- Como cuerpo del constructor, se puede incluir cualquier tipo de instrucción válida en cualquier método miembro de la clase. Sólo es importante recordar que estas instrucciones, por definición conceptual, están destinadas a la inicialización de los elementos de la clase.

Ejemplo:

```
class Persona {
      string nombre;
      int edad;

      public Persona(string n, int e) {
            nombre = n;
            if(e >= 0)
                  edad = e;
            else
                  edad = 0;
      }
}
```

Al instanciar un objeto de esta clase, dado que el único constructor declarado es el que recibe dos parámetros, obligatoriamente se deben pasar los dos parámetros requeridos. Esto se vería así:

```
class MainApp {
      static void Main() {
            Persona p1 = new Persona("Juan", 30);
            ...
      }
}
```

1.2.2 Llamado entre constructores de clases heredadas

En el caso de haberse implementado herencia entre clases, el llamado entre constructores aún puede ser invocado y controlado. Para efectos prácticos, el uso de la palabra clave "base()" es sinónimo de llamar la ejecución del constructor de la clase base.

Esto se ve en el siguiente ejemplo, cuyo código completo se encuentra en los ejemplos de este capítulo, unas páginas más adelante.

```
class Racional : NumeroCompuesto {

   public Racional(int n, int d) // Constructor de la clase derivada.
      : base(n,d)          // Se llama al constructor de la clase base.
   {
            ...  // Inicialización particular de la clase derivada
   }
```

Aquí se ve que la clase Racional hereda de NumeroCompuesto, que a su vez cuenta con su propio constructor. Al declarar el constructor de la clase derivada, que en este caso recibe los mismos dos parámetros de la clase base, se pasan directamente estos dos parámetros a la ejecución del constructor de la clase base, dejando la lógica de éste intacta.

El siguiente ejemplo muestra una herencia de tres niveles y el llamado a los constructores respectivos. Al probar este ejemplo se puede ver en pantalla el orden de ejecución de los tres constructores respectivos.

```
using System;

class Base {
   public Base(int n) {  // Constructor
     Console.WriteLine("Constructor Base: {0}",n);
   }
}
```

```
class Heredada1 : Base {
   public Heredada1(int n): base(n) {   // Constructor
     Console.WriteLine("Constructor Heredada1: {0}",n);
   }
}

class Heredada2 : Heredada1 {
   public Heredada2() : base(1) { // Constructor
     Console.WriteLine("Constructor Heredada2");
   }
}


class MainApp {
   static void Main() {
     Heredada2 h2 = new Heredada2();
     Console.ReadLine();
   }
}
```

1.3 Elementos estáticos

Una instancia (de objeto) se diferencia de otra por los valores de sus respectivos atributos que puedan tener en un cierto momento. Es decir, cada instancia mantiene una copia de cada uno de los atributos definidos para su clase. Sin embargo, es factible declarar tanto atributos como métodos que no son instanciados por cada objetivo individualmente, sino que son compartidos por todas las instancias de la clase correspondiente.

Para declarar cuáles de los elementos de una clase (atributos, métodos), son compartidos conjuntamente por todas las instancias de la clase, se le antepone la palabra clave “static” en su declaración.

Para ilustrar el concepto, el siguiente ejemplo declara una clase Producto, todas cuyas instancias tendrán diferentes valores para los atributo codigo y precio, sin embargo, el método CalcularValor() se basa en un porcentaje de margen, que es común para todas las instancias. De esa manera, modificar el valor de este atributo es factible de hacerse en cualquiera de las instancias de la clase, y ese cambio se refleja para todas las instancias.

```
using System;

class Producto {
      string codigo;
      int precio;
      static float margen;

      public Producto(string c, int p) { codigo = c; precio = p; }

      public void CambiarMargen(float m) { margen = m; }

      public float CalcularValor() {
            return( (float) precio*(1+margen) ) ;
      }
}

class MainApp {
      static void Main() {
            Producto p1 = new Producto("P001", 30);
```

```
            Producto p2 = new Producto("P002", 10);
            Producto p3 = new Producto("P003", 20);

            p1.CambiarMargen(0.15F); // Se aplica a p1, p2, p3

            Console.WriteLine("Valores: {0}; {1}; {2}",
                  p1.CalcularValor(), p2.CalcularValor(),
                  p3.CalcularValor());

            p1.CambiarMargen(0.10F); // Se aplica a p1, p2, p3

            Console.WriteLine("Valores: {0}; {1}; {2}",
                  p1.CalcularValor(), p2.CalcularValor(),
                  p3.CalcularValor());

            Console.ReadLine();
      }
}
```

Como resultado en pantalla se vería:

```
Valores: 34,5; 11,5; 23
Valores: 33; 11; 22
```

El mismo concepto “static” es aplicable a los métodos de las clases, estableciendo aquellos métodos que serían “independientes” de la instancia particular. Esto es comúnmente usado en la sobrecarga de operadores, por ejemplo, lo cual se ve más adelante en este capítulo.

Aquí cabe señalar con claridad que es ésta precisamente la razón por la cual el método Main() siempre se declara static.

1.4 Propiedades

Tomando en cuenta la característica privada de casi todos los atributos que se definen para una clase – las buenas prácticas de programación fomentan declarar como *private* todos los atributos de las clases – se hace necesario ofrecer cierta visibilidad controlada sobre los valores de estos atributos. Esta visibilidad se describe como la obtención del valor de cada uno de los atributos, como la eventual facultad de modificar dicho valor. Reiterando, si los atributos son privados, ninguna de éstas dos funcionalidades está disponible desde otras clases.

Una manera clásica en la programación orientada a objeto de resolver esta visibilidad, aún manteniendo el control sobre lo que se puede ver o modificar de un atributo, se utilizan métodos que definen una u otra de estas funcionalidades.

Tal sería el caso en el siguiente ejemplo.

```
class Persona {
      private int edad;
      public int getEdad() { return(edad); }
      public void setEdad(int e) { if(e>0) edad = e; }
}
```

En esta clase Persona se distingue el único atributo privado “edad”, que evidentemente no es visible fuera de la clase. Por otro lado, dos métodos ofrecen la visibilidad sobre dicho atributo, denominados getEdad() y setEdad(). El primero de ellos es un ejemplo muy simple de un método que al ser declarado público es visible fuera de la clase, y como esencial funcionamiento, retorna el valor del atributo cuando es invocado. A su vez, el segundo de ellos modifica el valor del atributo privado, siempre y cuando el nuevo valor cumpla cierta condición (en este caso sea número mayor que cero).

La combinación de las tres declaraciones permite establecer un atributo privado, pero accesible bajo control por medio de sus métodos.

Esta problemática fue enfrentada por los diseñadores del lenguaje C#, lo que ofrecieron una solución más eficiente, siempre manteniendo el objetivo presente: dar visibilidad controlada sobre atributos privados. Esta solución se basa en la denominadas Propiedades, que si bien no tienen la misma declaración de un método, semánticamente ofrecen lo mismo que los métodos: ejecución de instrucciones relacionadas con los atributos.

Su declaración se basa en tres aspectos: su declaración (identificador), que no lleva paréntesis, su declaración de funcionalidad de visibilidad (get) y la de modificación (set).

En este caso, el mismo ejemplo anterior quedaría.

```
class Persona {
      private int edad;
      public int Edad {
            get { return(edad); }
            set { if(value>0) edad = value; }
      }
}
```

En esta segunda versión del mismo ejemplo, utilizando la sintaxis de propiedades de C#, se identifica la declaración de dos elementos solamente: el atributo privado, y la propiedad pública. Esta declaración tiene las siguientes características:

- El identificador para la propiedad es distinto al del atributo, pero por convención se utiliza uno muy similar (por ejemplo diferenciando sólo por la primera letra mayúscula) y no lleva paréntesis como los métodos.
- Se puede declarar el bloque “get” o el “set”, sólo uno de ellos o ambos. Es decir, al declarar una propiedad sólo con bloque get, implícitamente se está indicando que la propiedad no acepta modificación de valor por asignación.
- En el bloque “set”, la manera de identificar el valor que se quiere asignar al respectivo atributo es por el uso de la palabra clave “value”, que en este contexto se toma como el valor (sea cual sea su tipo de dato correspondiente) que se asigna a la propiedad y por ende, al atributo.

1.5 Clases abstractas

Al usar el modificador “abstract” al declarar una clase, se indica que ésta está pensada exclusivamente para servir como clase base de otras que heredarán su definición básica. Por definición, estas clases abstractas no pueden ser instanciadas (no se pueden declarar instancias de dichas clases, sólo heredarlas). Aún así, una clase abstracta puede contener métodos abstractos (pensados en ser implementados en versiones definitivas en las clases derivadas), o bien métodos formalmente implementados, pensados en ser utilizados como tales en las clase derivadas (típicamente el caso de las propiedades centrales de esta clase base).

```
abstract class Persona {
      protected string nombre;
      protected int edad;

      ...

      public void MostrarDatos() {  }
}

class Alumno : Persona {
      int numero;

      public void MostrarDatos() {
            Console.WriteLine("{0}, {1}, {2}", nombre, edad, numero);
      }
}
```

1.6 Sobrecarga de métodos heredados

Aprovechando uno de los principales conceptos de la programación orientada a objeto, llamado Polimorfismo, en este caso representado por la sobrecarga de métodos, mediante la redeclaración de un método en una clase heredada, se deben considerar ciertos aspectos de diseño, que expliciten esta situación.

Por ejemplo, si se da un ejemplo de herencia como el siguiente, donde una clase base representa a una persona en particular, mientras que la heredada representa un alumno de una universidad, se puede hacer la siguiente definición del método MostrarDatos().

```
class Persona {
      protected string nombre;
      protected int edad;

      ...

      // En la clase base, el método MostrarDatos() no tiene
      // código, ya que no interesa implementar su lógica.
      // Sin embargo, se declara "virtual", explicitando que
      // la lógica de este método deberá ser implementada
      // por las clases derivadas.
      public virtual void MostrarDatos() { }
}

class Alumno : Persona {
      ...

      // En la clase derivada, el método MostrarDatos() debe
      // ser explícitamente una sobrecarga del mismo de la
      // clase base. Para ello, se declara "override".
      public override void MostrarDatos() {
            Console.WriteLine("Nombre: ", this.nombre);
            Console.WriteLine("Edad: ", this.edad);
      }
}
```

1.7 Sobrecarga de operadores (+, -, ...)

Conociendo la posibilidad de definir métodos con sobrecarga, existe en C# la particularidad de poder re-definir el funcionamiento de algunos operadores estándar, como son los operadores matemáticos (+, -, * , /), sobre objetos de tipo más complejo que los numéricos básicos. En el siguiente ejemplo se define un nuevo tipo de dato, en la forma de una clase, que representa los números complejos, compuestos por una parte real y otra imaginaria. En este caso se re-define la operación suma, identificada por el símbolo '+'.

```
class Complejo {
      private int real, imag;
      public Complejo(int r, int i) { real = r; imag = i; }

      public static Complejo operator+(Complejo c1, Complejo c2) {
         Complejo c = new Complejo(c1.real+c2.real, c1.imag+c2.imag);
         return c;
      }
}
```

Con esta declaración ya es factible armar una expresión con el operador de suma (+) entre dos instancias de la clase Complejo. Un ejemplo más completo de este concepto se ve más adelante, en la implantación de la clase Racional, como representante de los números racionales.

2 Soluciones Orientadas a Objeto

2.1 Ejemplo: Números Racionales – Clases Abstractas, Herencia, Operadores, Propiedades.

Para este ejemplo, se toma como referencia todos aquellos tipos de números que están compuestos por diferentes partes. Tal es el caso de los números complejos, compuestos de una parte real y otra imaginaria, o de los puntos en el espacio bidimensional, compuestos de dos coordenadas en la forma (x,y), y también los números racionales, compuestos por un numerador y un denominador.

En este caso, es precisamente el caso de los números racionales el que se quiere revisar. Como propósito final se pretende contar con una implementación del número racional lo suficientemente completa como para trabajar en forma transparente con ellos en un programa principal.

Como ejemplo se muestra a continuación un ejemplo de cómo podría utilizarse esta definición de Racional, conociendo sólo los métodos y características públicas. Conociendo esta interfaz es posible emplear este tipo de objetos para resolver problemas en los que es necesario manipular números racionales. Como puede verse, **no es necesario conocer la implementación**, de hecho, hasta el momento no sabemos cómo se representan los datos ni cómo están escritas las funciones que componen la interfaz.

Tipo de Dato (clase)	Racional. Cada Racional está compuesto por un numerador y un denominador
Declaración	Racional r = new Racional(n1, n2);
Métodos Básicos	Numerador(): entrega el valor entero del numerador del racional Denominador(): entrega el valor entero del denominador del racional Imprimir(): muestra en pantalla el racional en la forma (n/d)
Operadores	Suma (+), resta (-), multiplicación (*), división (/)

Teniendo clara esta información, se puede proceder a declarar un algoritmo principal en un programa, utilizando extensivamente los Racionales, sin necesariamente conocer su definición real ni su composición.

```
using System;

class Racional { .... }

class MainApp {

    static void Main() {
      Racional r1 = new Racional(2,3);
      Racional r2 = new Racional(4,5);

      Console.Write("r1 es "); r1.Imprimir(); Console.WriteLine();
      Console.Write("r2 es "); r2.Imprimir(); Console.WriteLine();

      Racional r3 = r1 + r2;
      Console.Write("r1 + r2 es "); r3.Imprimir(); Console.WriteLine();
      r3 = r2 - r1;
      Console.Write("r1 - r2 es "); r3.Imprimir(); Console.WriteLine();
      r3 = r2 * r1;
      Console.Write("r1 * r2 es "); r3.Imprimir(); Console.WriteLine();
      r3 = r2 / r1;
      Console.Write("r1 / r2 es "); r3.Imprimir(); Console.WriteLine();

      Console.Write("Presione ENTER.");  Console.ReadLine();
    }
}
```

La verdadera implementación de la clase Racional, evidentemente define una serie de elementos internos, que resuelven la problemática, pero la idea siempre es no requerir conocerla con detalle, sino que sólo saber usarla en un programa como el anterior.

Para este ejemplo, la implementación pasa por la definición de una clase abstracta base, que precisamente representa todos aquellos tipos de números compuestos de dos partes. La clase NumeroCompuesto define dos partes que lo componen. Junto con ello, se definen propiedades de acceso controlado a estas dos partes y el correspondiente constructor, que las inicializa con dos valores pasados como parámetros.

Teniendo esa clase base, se define la clase derivada: Racional, que hace la relación entre las dos partes de NumeroCompuesto, con el numerador y denominador. Es importante destacar en este ejemplo, que los componentes internos de la clase base, NumeroCompuesto, no son visibles en la clase derivada, aunque si lo son las propiedades que controlan el acceso. Haciendo buen uso de éstas, se logra definir la lógica de la implementación de esta clase derivada.

Adicionalmente, la clase Racional re-define los operadores básicos de números: suma, resta, multiplicación y división, de modo que sea aún más natural utilizar en un programa externo este tipo de dato definido.

El siguiente Diagrama de Clases en UML refleja esta relación entre Racional y NumeroCompuesto.

En este diagrama, los elementos que se enumeran con un símbolo '-', son privados, los que tienen '+' son públicos, y los que anteponen un '#' son protegidos.

El código completo de estas dos clases se encuentra a continuación, que es totalmente consecuente con el algoritmo principal anteriormente indicado:

```
using System;

////////////////////////////////////////////////////////////////////////////////
// NumeroCompuesto: un tipo de número genérico compuesto por dos partes (en este contexto)
// En este caso se trata de una clase abstracta: definida exclusivamente para ser heredada
// y no para ser instanciada.
////////////////////////////////////////////////////////////////////////////////
abstract class NumeroCompuesto {

    private int parte1;   // parte1 y parte2 son invisibles al exterior de la clase
    private int parte2;   // incluso en las clases derivadas que la heredan.

    protected NumeroCompuesto(int p1, int p2) {  // Constructor
      parte1 = p1; parte2 = p2;
    }

    // Definición de las propiedades de la clase NumeroCompuesto
    protected int Parte1 {
           get { return (parte1); }
           set { parte1 = value; }
    }
    protected int Parte2 {
           get { return (parte2); }
           set { parte2 = value; }
    }
}
```

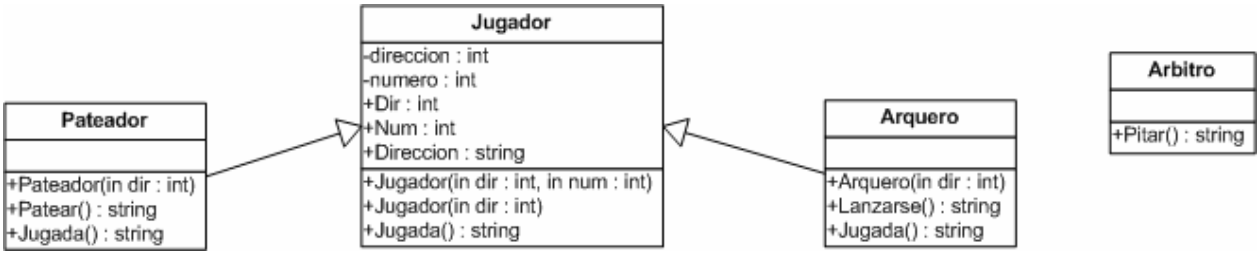
```
////////////////////////////////////////////////////////////////////////////
// Racional: heredando las características de NumeroCompuesto (dos partes)
//     : en este caso relacionando parte1->numerador ... parte2->denominador
////////////////////////////////////////////////////////////////////////////

class Racional : NumeroCompuesto {

   public Racional(int n, int d)    // Constructor de la clase: inicialización
    : base(n,d)               // se llama al constructor de la clase base.
   {
   }

   // Métodos básicos: acceso a las partes del número Racional.
   // Esto también se puede hacer con el formato de Propiedades
   public int Numerador()   { return(this.Parte1); }
   public int Denominador() { return(this.Parte2); }

   // Método simple: imprime en pantalla el número con sus partes.
   public void Imprimir() { Console.Write("{0}/{1}", this.Parte1, this.Parte2); }



   // Sobrecarga de operadores: +, -, *, /

   // Suma de Racionales: la suma de las multip. cruzadas / multiplicación denom.
   public static Racional operator+(Racional r1, Racional r2)
   {
    Racional r = new Racional(
           r1.Numerador()*r2.Denominador() + r1.Denominador()*r2.Numerador(),
           r1.Denominador()*r2.Denominador());
    return r;
   }

   // Resta de Racionales: la resta de las multip. cruzadas / multiplicación denom.
   public static Racional operator-(Racional r1, Racional r2)
   {
    Racional r = new Racional(
           r1.Numerador()*r2.Denominador() - r1.Denominador()*r2.Numerador(),
           r1.Denominador()*r2.Denominador());
    return r;
   }

   // Multiplicación de Racionales: multiplicación numeradores / multiplicación denom.
   public static Racional operator*(Racional r1, Racional r2)
   {
    Racional r = new Racional(
           r1.Numerador()*r2.Numerador(),
           r1.Denominador()*r2.Denominador());
    return r;
   }

   // División de Racionales: multiplicaciones cruzadas.
   public static Racional operator/(Racional r1, Racional r2)
   {
    Racional r = new Racional(
           r1.Numerador()*r2.Denominador(),
           r1.Denominador()*r2.Numerador());
    return r;
   }
}
```

2.2 Ejemplo: Ejecución de un Penal – Herencia, Propiedades, Sobrecarga Métodos Heredados.

Profundizando un ejemplo del capítulo 1, la ejecución del lanzamiento penal en fútbol se puede modelar desde un enfoque totalmente orientado a objeto, una vez que se han entendido los elementos involucrados. Simplificando en este caso, se requiere de dos jugadores, los cuales si bien comparten parte de los elementos que los definen, tienen ciertos comportamientos y elementos que los diferencian.

De tal manera, se hace una definición de la clase Jugador genérico, que representa lógicamente a cualquier jugador. Junto a esta definición, se declara el árbitro. Y como derivado del Jugador genérico aparecen el Pateador y el Arquero.

Esta estructura de objetos se refleja en el siguiente Diagrama de Clases en UML. Todas estas clases están contenidas dentro del Namespace Deportes.Futbol.



El código que representa esta definición, contenido en el namespace Deportes.Futbol, se encuentra a continuación. Cabe destacar que ninguna de las clases es abstracta, por lo que cualquiera de ellas podría ser instanciada, y por ende, conformar un equipo completo, o dos para un partido oficial.

```
namespace Deportes.Futbol {

      // ---------------------------------------------
      // clase Arbitro
      // ---------------------------------------------
      public class Arbitro {
            public string Pitar() {
                  return ("¡PRIIIP!");
            }
      }


      // ---------------------------------------------------------------
      // clase Jugador: representa a cualquier jugador de la cancha
      // ---------------------------------------------------------------
      public class Jugador {
            private int direccion;  // 1: izquierda;  2: derecha
            private int numero;

            public Jugador(int dir) {
                  if( dir<1 || dir>2 ) dir = 1;
                  direccion = dir;
                  numero = 2;
            }
            public Jugador(int dir, int num) {
                  if( dir<1 || dir>2 ) dir = 1;
                  direccion = dir;
```

```
                numero = num;
            }

            public int Dir {
                get { return(direccion); }
                set { if( value<1 || value>2 ) direccion = 1;
                      else direccion = value; }
            }
            public int Numero { get { return(numero); } }

            public string Direccion {
                get {
                      if (direccion == 1) return("izquierda");
                      if (direccion == 2) return("derecha");
                      return("-");
                }
            }

            public virtual string Jugada() { return(""); }
      }


      // ------------------------------------------------------------
      // clase Pateador, derivada de Jugador:
      // Representa a un jugador que pateará un tiro libre o penal
      // ------------------------------------------------------------
      public class Pateador : Jugador {

            public Pateador(int dir) : base(dir, 9) {}

            public string Patear() {
                return("Jugador " + Numero.ToString() +
                                        " patea hacia la " + Direccion);
            }
            public override string Jugada() { return(Patear()); }
      }


      // ------------------------------------------------------------
      // clase Arquero, derivada de Jugador:
      // Representa a un arquero que se lanzará para atajar un tiro
      // ------------------------------------------------------------
      public class Arquero : Jugador {

            public Arquero(int dir) : base(dir, 1) {}

            public string Lanzarse() {
                return("Arquero se lanza hacia la " + Direccion);
            }

            public override string Jugada() { return(Lanzarse()); }
      }
}
```

Teniendo implementada la lógica de los participantes de esta ejecución de penales, en este namespace Deportes.Futbol, se puede aprovechar e incluir en cualquier programa que implemente un algoritmo principal, que haga referencia y uso de estas clases.

De tal manera, a continuación se entregan dos posibles implementaciones de la ejecución del penal, una versión simple de Consola, donde se le preguntan las opciones al usuario, y una segunda donde la lógica de operación es exactamente la misma, pero se interactúa con el usuario por medio de una ventana de Windows.

2.2.1 Ejecución de un penal – Versión Consola

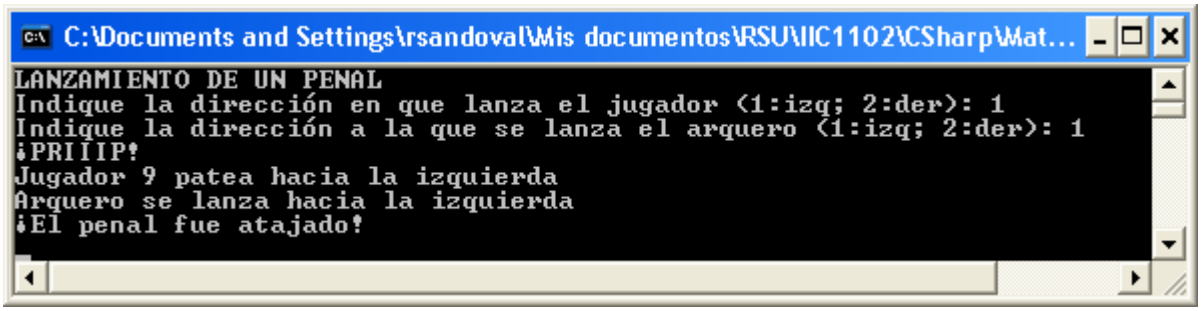
El siguiente código corresponde a la clase principal del programa, donde se hace referencia al namespace que define los jugadores que participan. Aquí se reconoce un Main() que tiene el algoritmo principal, interactuando con el usuario por medio de la consola.

```
using System;
using Deportes.Futbol;


class CMain {
       public static void Main()  {
              int dir1, dir2;
              Console.WriteLine("LANZAMIENTO DE UN PENAL");
              Console.Write("Indique dirección en que lanza el jugador (1:izq; 2:der): ");
              dir1 = int.Parse(Console.ReadLine());
              Console.Write("Indique dirección a la que se lanza el arquero (1:izq; 2:der): ");
              dir2 = int.Parse(Console.ReadLine());

              Arbitro b = new Arbitro();
              Pateador p = new Pateador(dir1);
              Arquero a = new Arquero(dir2);

              Console.WriteLine(b.Pitar());
              Console.WriteLine(p.Jugada());
              Console.WriteLine(a.Jugada());

              if (p.Direccion == a.Direccion) Console.WriteLine("¡El penal fue atajado!");
              else Console.WriteLine("¡El gol fue convertido!");

              Console.ReadLine();
       }
}
```

Para poder generar un archivo penales.exe que contenga la lógica de ambos archivos separados (el namespace Deportes.Futbol en el archivo futbol.cs y el algoritmo principal en penales.cs), se debe ejecutar una compilación de dos entradas de la siguiente forma (archivo build.bat). Los tres archivos deben estar en el mismo directorio.

```
@%WINDIR%\Microsoft.NET\Framework\v1.1.4322\csc.exe /out:.\penales.exe futbol.cs penales.cs
@pause
```

Compilando, aparece el archivo penales.exe, y al ejecutarlo se ve la siguiente consola:



2.2.2 Ejecución de un penal – Versión Windows

Utilizando como base el mismo *namespace* Deportes.Futbol, sin hacer ni una sola modificación, se reaprovecha su lógica en la implementación de una versión equivalente a la de consola, esta vez logrando que la interacción con el usuario se realice por medio de controles Windows, es decir, elementos gráficos con los que el usuario manifiesta sus preferencias.

La ventaja de la arquitectura en base a componentes en la Programación Orientada a Objeto, es precisamente la capacidad de reutilización de elementos (lógica de funcionamiento), ya implementada, en otras soluciones.

A continuación se incluye el código fuente de la clase principal de este programa en versión Windows, la cual mantiene la misma lógica operativa de la versión de consola. Este ejemplo en particular, a diferencia del anterior, presenta muchas más líneas de código y métodos adicionales, cuyo propósito es específicamente trabajar con los objetos gráficos de la ventana de Windows con la que interactúa el usuario, dejando la lógica de decisión sobre el problema de ejecución del penal, en un método final, invocado al activar el botón del usuario.

```
using System;
using System.Drawing;
using System.IO;
using System.Windows.Forms;
using System.Diagnostics;


using Deportes.Futbol;


namespace Deportes.Futbol.Penal {

	/// <summary>
	/// Ventana de simulación de la ejecución de un penal,
	/// donde se elige la dirección en que lanza el pateador
	/// y la que se lanza el arquero.
	/// Según eso, calcula si el penal fue convertido o atajado.
	/// </summary>
	public class VentanaPenales : Form {

		Pateador p;
		Arquero a;

		private Label lbTitulo;
		private Label lbPateador;
		private Label lbArquero;
		private Label lbResultado;
		private ComboBox cbPateador;
		private ComboBox cbArquero;
		private Button btEjecutar;

		public VentanaPenales() {
			p = new Pateador(1);
			a = new Arquero(1);

			cbPateador = new ComboBox();
			cbArquero = new ComboBox();
			btEjecutar = new Button();
			lbResultado = new Label();
			lbPateador = new Label();
			lbArquero = new Label();
			lbTitulo = new Label();
			this.SuspendLayout();

			lbTitulo.FlatStyle = System.Windows.Forms.FlatStyle.Popup;
			lbTitulo.Font = new System.Drawing.Font("Microsoft Sans Serif", 12F,
System.Drawing.FontStyle.Bold, System.Drawing.GraphicsUnit.Point, ((System.Byte)(0)));
			lbTitulo.Location = new System.Drawing.Point(16, 16);
```

```
                    lbTitulo.Name = "lbTitulo";
                    lbTitulo.Size = new System.Drawing.Size(250, 32);
                    lbTitulo.TabIndex = 5;

                    lbPateador.FlatStyle = System.Windows.Forms.FlatStyle.Popup;
                    lbPateador.Font = new System.Drawing.Font("Microsoft Sans Serif", 8F,
System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, ((System.Byte)(0)));
                    lbPateador.Location = new System.Drawing.Point(16, 55);
                    lbPateador.Name = "lbPateador";
                    lbPateador.Size = new System.Drawing.Size(120, 20);
                    lbPateador.TabIndex = 6;
                    lbArquero.FlatStyle = System.Windows.Forms.FlatStyle.Popup;
                    lbArquero.Font = new System.Drawing.Font("Microsoft Sans Serif", 8F,
System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, ((System.Byte)(0)));
                    lbArquero.Location = new System.Drawing.Point(16, 80);
                    lbArquero.Name = "lbPateador";
                    lbArquero.Size = new System.Drawing.Size(120, 20);
                    lbArquero.TabIndex = 7;

                    cbPateador.DropDownWidth = 112;
                    cbPateador.Location = new System.Drawing.Point(140, 53);
                    cbPateador.Name = "cbPateador";
                    cbPateador.Size = new System.Drawing.Size(112, 21);
                    cbPateador.TabIndex = 1;
                    cbPateador.SelectedIndexChanged += new
System.EventHandler(this.cbPateador_SelectedIndexChanged);

                    cbArquero.DropDownWidth = 112;
                    cbArquero.Location = new System.Drawing.Point(140, 78);
                    cbArquero.Name = "cbArquero";
                    cbArquero.Size = new System.Drawing.Size(112, 21);
                    cbArquero.TabIndex = 2;
                    cbArquero.SelectedIndexChanged += new
System.EventHandler(this.cbArquero_SelectedIndexChanged);

                    btEjecutar.Location = new System.Drawing.Point(160, 110);
                    btEjecutar.Name = "btEjecutar";
                    btEjecutar.Size = new System.Drawing.Size(90, 24);
                    btEjecutar.TabIndex = 3;
                    btEjecutar.Text = "Ejecutar";
                    btEjecutar.Click += new System.EventHandler(this.btEjecutar_Click);

                    lbResultado.FlatStyle = System.Windows.Forms.FlatStyle.Popup;
                    lbResultado.Font = new System.Drawing.Font("Microsoft Sans Serif", 12F,
System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, ((System.Byte)(0)));
                    lbResultado.Location = new System.Drawing.Point(8, 140);
                    lbResultado.Name = "lbResultado";
                    lbResultado.Size = new System.Drawing.Size(264, 64);
                    lbResultado.TabIndex = 4;

                    // Dimensionamiento y estilo de la ventana.
                    this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
                    this.ClientSize = new System.Drawing.Size(280, 215);
                    this.Controls.AddRange(new System.Windows.Forms.Control[] {
                                lbTitulo, lbPateador, lbArquero, btEjecutar, cbPateador,
cbArquero, lbResultado });
                    this.Name = "VentanaPenales";
                    this.Text = "Ejecución de Penal";
                    this.ResumeLayout(false);


                    cbPateador.Items.Add ("Izquierda");
                    cbPateador.Items.Add ("Derecha");
                    cbPateador.SelectedIndex = 0;
                    cbArquero.Items.Add ("Izquierda");
                    cbArquero.Items.Add ("Derecha");
                    cbArquero.SelectedIndex = 0;
                    lbTitulo.Text ="Ejecución de Penal";
                    lbPateador.Text ="Pateador patea hacia la";
                    lbArquero.Text ="Arquero se lanza a la";
                    lbResultado.Text ="(Penal aún no ejecutado)";
```

```
		}


		/// <summary>
		/// Algoritmo principal de la aplicación.
		/// Se abre una ventana para pedir los datos al usuario y ejecutar.
		/// </summary>
		[STAThread]
		static void Main()
		{
			Application.Run(new VentanaPenales());
		}

		private void btEjecutar_Click(object sender, System.EventArgs e) {
			lbResultado.Text = p.Patear() + "\n" + a.Lanzarse();
			if(p.Direccion == a.Direccion)
				lbResultado.Text += "\nEl penal fue atajado ";
			else
				lbResultado.Text += "\n¡El penal fue convertido!";
		}



		private void cbPateador_SelectedIndexChanged(object sender, System.EventArgs e) {
			p.Dir = cbPateador.SelectedIndex + 1;
		}
		private void cbArquero_SelectedIndexChanged(object sender, System.EventArgs e) {
			a.Dir = cbArquero.SelectedIndex + 1;
		}
	}
}
```

Para poder crear el ejecutable de este programa, es necesario realizar una compilación que incluya ambos códigos fuentes: el namespace Deportes.Futbol (futbol.cs), y el Main() para la ventana Windows (penales.cs). En este caso particular se definión una arquitectura de compilación diferentes: primero se compila futbol.cs, produciendo como resultado un archivo independiente de tipo ".NET Module". Luego se compila el archivo principal, tomando en la compilación el módulo compilado anterior, produciendo en la suma el ejecutable final. Esta idea se refleja en el siguiente archivo build.bat, en este caso de tres líneas: una para cada compilación y un "pause" final.

```
@%WINDIR%\Microsoft.NET\Framework\v1.1.4322\csc.exe /target:module /debug+ /d:TRACE futbol.cs

@%WINDIR%\Microsoft.NET\Framework\v1.1.4322\csc.exe /target:winexe /debug+ /d:TRACE /addmodule:futbol.netmodule /r:System.Windows.Forms.dll /r:System.Drawing.dll /r:System.dll penales.cs

@pause
```

Finalmente, al producirse el archivo penales.exe como resultado de esta compilación en etapas, al ejecutarlo aparece la ventana de interacción con el usuario que finalmente podría usarse para simular la ejecución del penal según las opciones seleccionadas por el usuario.