ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN
UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Proyecto Fin de Carrera

Simulador C# de una maqueta de trenes.

AUTOR: Aurelio Javier García García
DIRECTOR: Juan Ángel Pastor Franco
10 / 2007

Autor	Aurelio Javier García García
E-mail del Autor	aureliojg@yahoo.com
Director	Juan Ángel Pastor Franco
E-mail del Director	juanangel.pastor@upct.es
Título del PFC	Simulador C# de una maqueta de trenes
Descriptores	Ingeniería del software, control de maqueta ferroviaria, patrones de diseño

Resumen

Este Proyecto Fin de Carrera trata de diseñar e implementar una aplicación para la simulación y comunicación con una maqueta de trenes. Para alcanzar este fin, se realiza:

1. El modelado de los componentes de una maqueta ferroviaria.

2. Desarrollo de la aplicación para simular y controlar la maqueta de trenes desde una estación de trabajo en el laboratorio DSIE de la UPCT, conectada a la misma mediante un interfaz de puerto serie.

3. Establecer una separación coherente de las áreas de la aplicación, proporcionando bajo acoplamiento.

4. Desde un enfoque metodológico, identificar y utilizar patrones de diseño para resolver los puntos anteriores, en concreto, el patrón MVC (Modelo-Vista-Controlador)

Titulación	Ingeniero Técnico de Telecomunicación, Especialidad Telemática
Departamento	Tecnologías de la Información y las Comunicaciones
Fecha de Presentación	10- 2007

Índice general

Capítulo 1.
Introducción.

1.1. Contexto y motivación.

Este Proyecto Final de Carrera surge de la necesidad de diseñar e implementar una aplicación en C# para la simulación y comunicación con una maqueta de trenes. Posibilita a futuros proyectantes el desarrollo de nuevas aplicaciones y ampliaciones relacionadas con dicha maqueta de trenes, sin la necesidad de partir de cero. De esta manera, Se ha desarrollado un sistema que se encarga de la gestión de la maqueta de trenes (movimiento de los trenes, cambio del estado de las bifurcaciones, estado de los sensores…) y lo que es más importante una API a partir de la cual pueden "construirse" otras maquetas.

A la hora de realizar el diseño, se ha optado por la utilización del Patrón Modelo-Vista-Controlador permitiendo separar la interfaz de usuario de la aplicación de su lógica interna, consiguiendo que ambos evolucionen por separado. De esta manera se facilita la sustitución ciertas partes de la aplicación por otras sin afectar a las demás, ya que el simulador está dividido en tres áreas acopladas lo menos posible. Es de destacar, además, que la interfaz gráfica del usuario se ha hecho tan intuitiva y sencilla de manejar como ha sido posible.

La aplicación permite su uso en dos modos, llamados Simulador y Maqueta. El modo Simulador consiste en disponer de los modelos, las vistas y controladores en el mismo programa, y simular la maqueta del laboratorio, implementada a escala. El modo Maqueta, permite, partiendo de una interfaz desarrollada en anteriores proyectos, el funcionamiento de la maqueta real. Gracias al uso del patrón MVC, la única parte de la implementación que hay que cambiar es la correspondiente al modelo.

Como ampliación de este Proyecto durante el transcurso del mismo, se ha propuesto además realizar una distribución de la aplicación (modo Remoto) mediante la API de C#, .Net Remoting (Invocación Remota de Objetos en la plataforma .Net), estableciendo las vistas y los controladores en una máquina diferente a donde se encuentran la funcionalidad básica de la aplicación. Al hacer esto, han surgido problemas, que se explican en el capítulo 6, por lo que se ha dejado para el desarrollo de futuros proyectantes.

1.2. Objetivos.

Los objetivos que aborda este Proyecto Final de Carrera son:

1. Introducir la tecnología .Net y el lenguaje C# en el control de la maqueta de trenes.
2. Proporcionar ejemplos didácticos para la docencia.
3. Traducir a C# la aplicación Java de control de la maqueta de trenes.
4. Distribuir dichas aplicaciones mediante .Net Framework Remoting.

1.3. Entorno de desarrollo. Microsoft Visual Studio .NET 2003.

- ***Microsoft Visual Studio (1997).***

Microsoft Visual Studio es un entorno integrado de desarrollo (llamado en inglés por siglas: IDE) para sistemas Windows. Soporta varios lenguajes de programación tales como: Visual C++, Visual C#, Visual J#, ASP.NET y Visual Basic .NET, aunque actualmente se han desarrollado las extensiones necesarias para muchos otros.

Visual Studio permite a los desarrolladores crear aplicaciones, sitios y aplicaciones web, así como servicios web en cualquier entorno que soporte la plataforma .NET (a partir de la versión 6). Así se pueden crear aplicaciones que se intercomuniquen entre estaciones de trabajo, páginas web y dispositivos móviles.

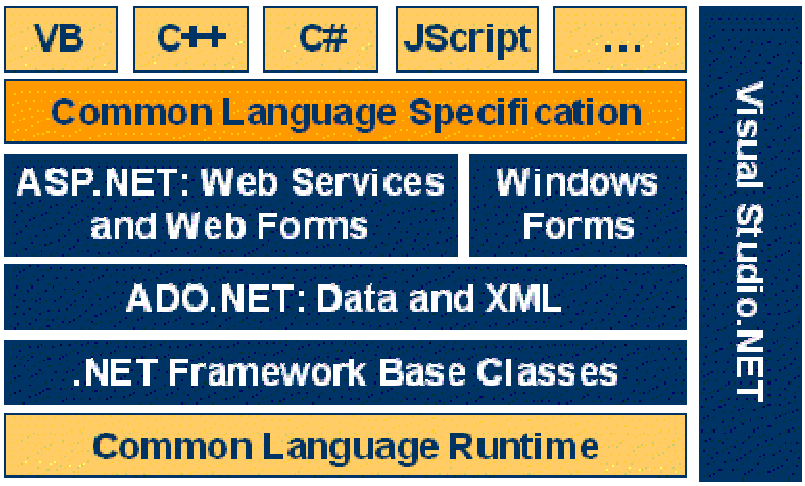
- ***Microsoft Visual Studio .NET (2002).***

Esta versión es la introducción de la plataforma .NET de Microsoft. .NET es una plataforma de ejecución intermedia multilenguaje. Los programas desarrollados en .NET no se compilan en lenguaje máquina (cómo C++), sino que se compilan en un lenguaje intermedio (CIL - Common Intermediate Language). El lenguaje usado es Microsoft Intermediate Language (MSIL). Cuando una aplicación MSIL se ejecuta es cuando se convierte al lenguaje máquina específico de la plataforma en la que se está ejecutando, haciendo que el código pueda ser independiente de plataforma (al menos de las soportadas actualmente por .NET). Las plataformas han de tener una implementación de Infraestructura de Lenguaje Común (CLI) para poder ejecutar programas MSIL.

También es la introducción del lenguaje C#, un lenguaje nuevo diseñado específicamente para la plataforma .NET basado en C++ y Java. Se presentó también el lenguaje J# -sucesor de J++- el cual en vez de ejecutarse en una máquina virtual de Java se ejecuta únicamente en el Framework .NET.

Todos los lenguajes están unificados en un único entorno. La interfaz se mejoró notablemente en esta versión, haciéndola más limpia y personalizable.

Visual Studio .NET puede usarse para crear programas basados en Windows (usando Windows Forms en vez de COM), aplicaciones y sitios web (ASP.NET y servicios web), y dispositivos móviles (usando el .NET Compact Framework).



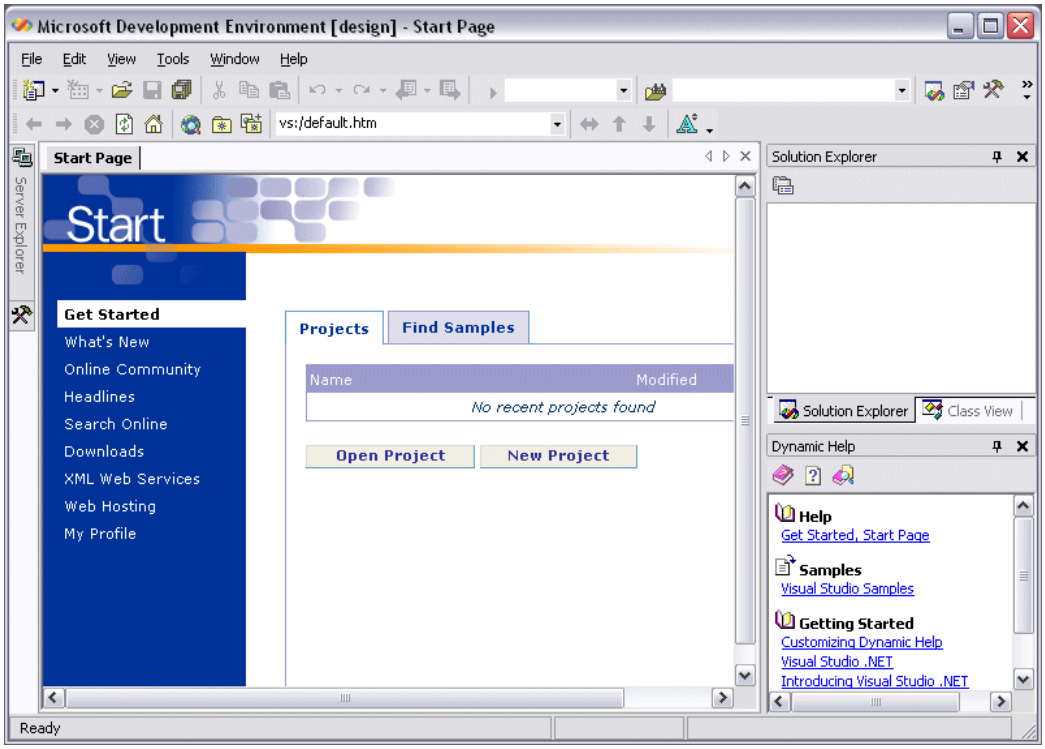
Estructura de Microsoft Visual Studio .NET.

- ***Microsoft Visual Studio .NET 2003.***

Visual Studio .NET 2003 es una actualización menor de Visual Studio .NET. Se actualiza el .NET Framework a la versión 1.1. También se añade soporte para escribir aplicaciones para determinados dispositivos móviles, ya sea con ASP.NET o con el .NET Compact Framework.

Visual Studio 2003 se lanzó en 4 ediciones: Academic, Professional, Enterprise Developer, y Enterprise Architect. La edición Enterprise Architect incluía una implementación de la tecnología de modelado Microsoft Visio, que se centraba en la creación de representaciones visuales de la arquitectura de la aplicación basadas en UML. También se introdujo "Enterprise Templates", para ayudar a grandes equipos de trabajo a estandarizar estilos de programación e impulsar políticas de uso de componentes y asignación de propiedades.

Este es el entorno de desarrollo utilizado para desarrollar este proyecto, en la edición **Professional**.



Microsoft Visual Studio .NET.

1.4. Lenguaje de programación. C#.

- ***Lenguaje C#.***

C# (pronunciado ***"si sharp"*** o **C sostenido**) es un lenguaje de programación orientado a objetos desarrollado y estandarizado por Microsoft como parte de su plataforma .NET, que después fue aprobado como un estándar por la ECMA e ISO.

Su sintaxis básica deriva de C/C++ y utiliza el modelo de objetos de la plataforma .NET el cual es similar al de Java aunque incluye mejoras derivadas de otros lenguajes (más notablemente de Delphi y Java). C# fue diseñado para combinar el control a bajo nivel de lenguajes como C y la velocidad de programación de lenguajes como Visual Basic.

C# significa, "do sostenido" (C corresponde a do en la terminología musical anglo-sajona). El símbolo # viene de sobreponer "++" sobre "++" y eliminar las separaciones, indicando así su descendencia de C++.

C#, como parte de la plataforma .NET, está normalizado por ECMA desde diciembre de 2001 (ECMA-334 "Especificación del Lenguaje C#"). El 7 de noviembre de 2005 acabó la beta y salió la versión 2.0 del lenguaje que incluye mejoras tales como tipos genéricos, métodos anónimos, iteradores, tipos parciales y tipos anulables. Ya existe la versión 3.0 de C# en fase de beta destacando los tipos implícitos y el LINQ (Language Integrated Query).

Aunque C# forma parte de la plataforma .NET, ésta es una interfaz de programación de aplicaciones; mientras que C# es un lenguaje de programación independiente diseñado para generar programas sobre dicha plataforma. Aunque aún no existen, es posible implementar compiladores que no generen programas para dicha plataforma, sino para una plataforma diferente como Win32 o UNIX.

En la actualidad existen los siguientes compiladores para el lenguaje C#:

- **Microsoft .NET Framework SDK** incluye un compilador de C#, pero no un IDE.
- **Microsoft Visual C#**, IDE por excelencia de este lenguaje, versión 2002, 2003 y 2005.
- **#develop**, es un IDE libre para C# bajo licencia LGPL, muy similar a Microsoft Visual C#.
- **Mono**, es una implementación GPL de todo el entorno .NET desarrollado por Novell. Como parte de esta implementación se incluye un compilador de C#.
- **Delphi 2006**, de Borland Software Corporation.
- **dotGNU Portable.NET**, de la Free Software Foundation.

- ***Objetivos del lenguaje.***

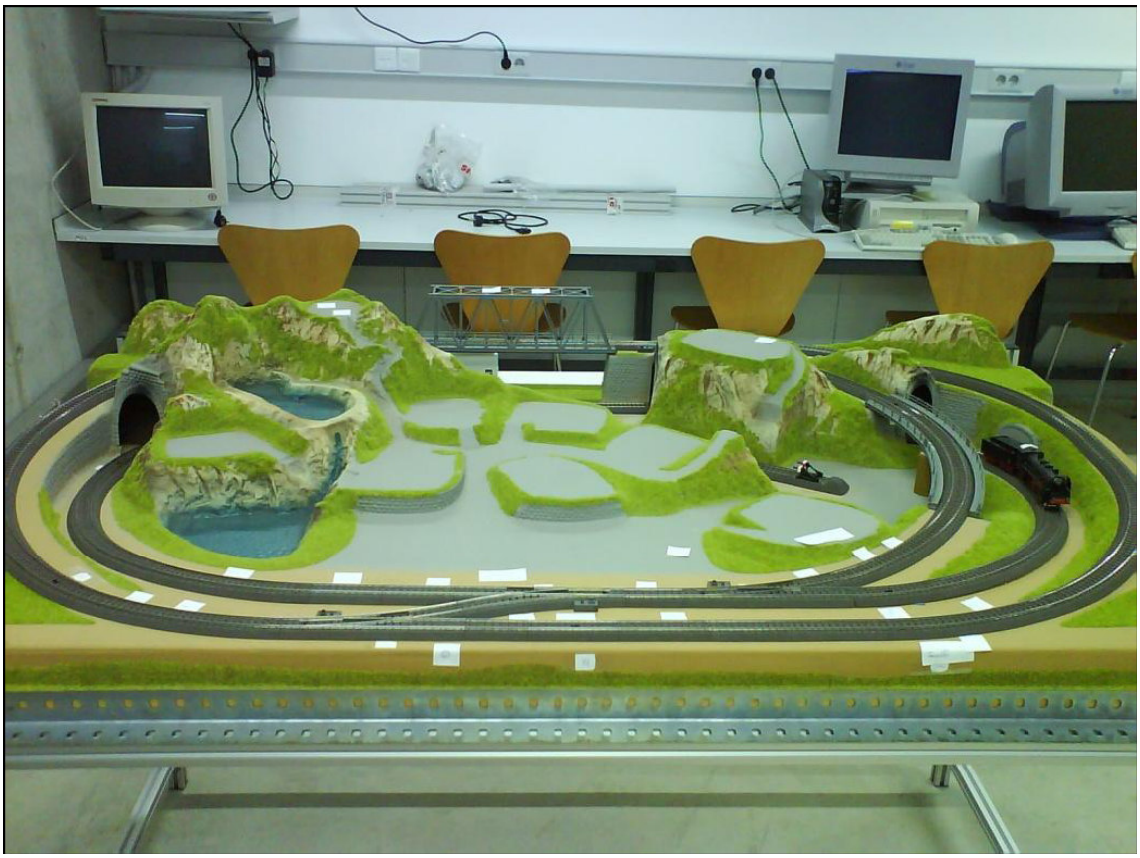
El estándar ECMA lista las siguientes metas en el diseño para C#:

- C# debe ser un lenguaje **simple, moderno, de propósito-general** de programación orientada a objetos.
- El lenguaje, las implementaciones de aquí en delante, deben proveer soporte para principios de ingeniería de software tales como revisión estricta de los tipos de datos, revisión de límites de arrays, detección de intentos de usar variables no inicializadas, y recolección de basura automática.
- Se espera que el lenguaje sea usado para desarrollar componentes de software que se puedan usar en ambientes distribuidos.
- Portabilidad de código fuente es muy importante, tal como es portabilidad del programador, especialmente para programadores familiarizados con C y C++.
- Soporte para internacionalización es muy importante.

- Se espera que C# sea adecuado para escribir aplicaciones desde las más grandes y sofisticadas como sistemas operativos hasta las más pequeñas funciones.
- Aunque las aplicaciones en C# estén orientadas a ser económicas respecto a los requisitos de memoria y proceso, el lenguaje no fue hecho para competir directamente en velocidad o tamaño con C o lenguaje ensamblador.

1.5. Infraestructura del laboratorio.

A mediados del año 2001, el Área de Lenguajes y Sistemas Informáticos del departamento TIC (Tecnologías de la Información y las Comunicaciones) estudió la compra y montaje de varias maquetas ferroviarias.



Maqueta sobre la que se ha basado el Proyecto.

En la figura se observa la disposición actual de la maqueta sobre la que se ha basado este Proyecto. Se trata de la maqueta que se encuentra en el laboratorio DSIE (División de Sistemas e Ingeniería Electrónica) en el edificio Cuartel de Antiguones de la UPCT.

La posibilidad de utilizar esta instalación como marco de un Proyecto Final de Carrera despierta un gran interés ya que se puede considerar como una primera práctica seria orientada al mundo laboral.

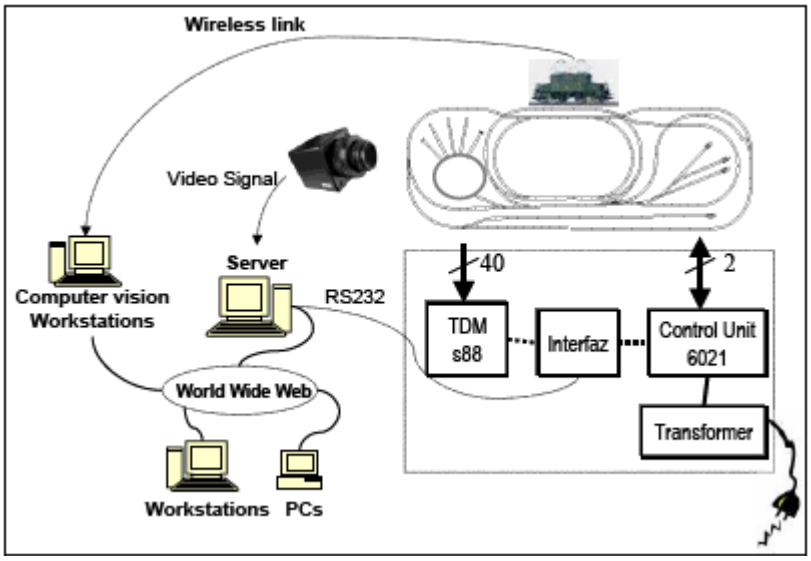
El elemento fundamental del laboratorio es la maqueta de trenes. Su instalación actual incluye cinco desvíos o agujas y una docena de patines o sensores de paso. Además existen otros elementos, no presentes actualmente, como semáforos y un hangar rotatorio para el almacenaje de trenes. Todos estos elementos pueden controlarse desde una unidad de control básica proporcionada por el fabricante, Märklin.

Todos los dispositivos de monitorización y control son digitales y pueden ser leídos o accionados desde un ordenador. Los sensores de paso están cableados a la unidad de control, en tanto que los comandos para los trenes, los semáforos y los cruces se transmiten a través de las vías, para lo cual se usa un protocolo de Motorola especialmente adaptado para hacer frente al ruido generado por los motores de los trenes.

La información que suministran los sensores de paso sirve para realizar las aplicaciones de control. Estos sensores detectan tato el paso del tren como el sentido de su movimiento. La distancia entre sensores limita la exactitud con la que puede determinarse la posición del tren en las vías. Por esta razón, se instalaron sensores adicionales y en algunos proyectos el seguimiento de los trenes se ha realizado mediante cámaras web, aplicando técnicas de Visión Artificial, que en ocasiones complementan a los sensores y en otras los sustituyen.

En cuanto a puestos y recursos, el laboratorio responde al esquema de la figura 1.4. El elemento principal es la unidad de control 6021 de Märklin, que recibe comandos bien desde una botonera o bien desde un módulo interfaz y los reenvía a través de la vía a trenes y desvíos. Este módulo dispone de un puerto RS232 para su conexión a un computador y además recoge información de estado de los sensores de paso desde el módulo de detección cableado a los mismos (el TDM s88 de Märklin).

El módulo interfaz está conectado a través de su puerto serie a una estación de trabajo Sun, en la cual se centralizan las rutinas de control de trenes y maqueta. A esta estación se conectan los puestos (ordenadores tipo Intel, con Sistema Operativo Windows y Linux) a través de una red de área local.



Esquema del laboratorio.

Resumiendo, la maqueta está formada por un conjunto de elementos simples, tomados de entre los disponibles en el catálogo de la casa Märklin, y conectados para formar circuitos cerrados. Todos ellos son digitales y se pueden controlar mediante una consola central de mando o mediante la interfaz del puerto serie de una computadora.

Se realiza una clasificación de estos elementos presentes en la maqueta del laboratorio:

1. **Tramos de vía rectos.** De estos, cabe diferenciar, sin medir su longitud, entre los que tienen un sensor o patín de vía y los que no. Adicionalmente, hay algunos otros disponibles en catálogo, como vías de desenganche, etc., aunque no estén presentes en el laboratorio.

2. **Tramos de vía curvos.** De nuevo, se debe distinguir entre los que tienen sensor o los que no.

3. **Desvíos.** Pueden darse tanto entre un tramo recto y otro curvo, como entre dos tramos curvos.

4. **Vagones del tren.** Se diferencia entre vagones con motor (locomotoras) y vagones destinados a transporte de pasajeros o mercancías. En el Simulador, se ha diferenciado a los trenes que hay sobre la maqueta por el número de vagones que representa cada uno.

No se han considerado, y por lo tanto tampoco descrito aquellas piezas que están presentes en nuestra maqueta, aunque existan y la casa Märklin las distribuya.

Se considera la maqueta como un elemento complejo formado por la agregación de otros más simples, en cantidad acorde con el tamaño final.

Una vez conocidos los elementos de la maqueta, se debe tomar en cuenta el orden en que serán dispuestos para formar el trazado, puesto que no es posible considerar toda conexión de forma idéntica. Según las piezas que se conecten entre sí, se obtendrán diferentes rutas.

Por lo tanto en la aplicación se ha construido un modelo capaz de ofrecer el control sobre la maqueta del laboratorio. La vista de dicha maqueta sobre la interfaz de usuario, es exactamente igual, pero a escala.

1.6. Metodología de trabajo. Fases del proyecto.

Para llevar a cabo la realización de este proyecto, programa "Simulador C# de una maqueta de trenes", se han seguido las siguientes fases con sus respectivas metodologías.

1. **Familiarización con lenguaje C# y tecnología .Net.**
   Para la introducción al lenguaje C# y la tecnología .Net se ha traducido el código Java a C# de una práctica docente de la asignatura Complementos de Informática, cursada por el director de este proyecto. Esto se verá explicado detalladamente en el capítulo 2.

2. **Traducción automática de Java a C# y adaptación manual del código.**
   Para llevar a cabo el desarrollo del programa "Simulador C# de una maqueta de trenes" nos hemos basado en un proyecto [1], del cual hemos heredado su estructura y metodología, siempre que ha sido posible. Para obtener una primera visión global de la estructura del programa en el lenguaje C# hemos utilizado el recurso disponible por Microsoft Visual Studio .NET 2003, "Ayudante para la conversión del lenguaje Java", el resultado del uso de este recurso nos ofrece un boceto del proyecto en C# desde el cual partir hasta conseguir un programa compilable y ejecutable, siendo este el proceso más laborioso. Esto se explica detalladamente en el capítulo 4.

3. **Distribución de la aplicación.**
   Para llevar a cabo la distribución del proyecto hemos utilizado la tecnología .Net Remoting, que es el equivalente para C# de Java RMI. Esto se explica detalladamente en el capítulo 6.

1.7. Contenido y estructura de la memoria.

El resto de esta memoria se estructura en varios capítulos, cuyo contenido se presenta brevemente a continuación.

En el capítulo 2 se describe la traducción de una práctica así como su ejecución. La finalidad de este capítulo es comparar Java y C#.

A continuación, en el capítulo 3 se presentan los diferentes paquetes que conforman el software desarrollado, y su porqué. De esta manera, permite a cualquier persona que lea esta documentación, tener información de relevancia y global acerca del modelado, pero no de la implementación.

En el capítulo 4 se explican los pasos que se han seguido a la hora de hacer la traducción del programa del lenguaje Java a C#.

Posteriormente, el capítulo 5 expondrá el modelado y la implementación de los componentes de cada uno de los modos de utilización del Proyecto, realizando una distinción del código desde un punto de vista estático (definiendo las estructuras de clases, herencia y composición), y dinámico (definiendo el comportamiento mediante diagramas de secuencia).

El capítulo 6 trata sobre la decisión que se ha tomado a la hora de poder realizar la distribución del Proyecto, indicando la tecnología usada, .NET Remoting.

En el capítulo 7 se incluye una guía de uso del software generado en este Proyecto. Se explica paso a paso cómo se deben de ejecutar las aplicaciones y cómo se utiliza, mediante imágenes descriptivas.

Se han expuesto en el capítulo 8 las conclusiones obtenidas tras el desarrollo de este Proyecto, las mejoras que puede aportar al entorno de trabajo del laboratorio DSIE para futuros proyectos finales de carrera, así como puntos concretos que se pudieran mejorar o realizar en siguientes proyectos.

Finalmente, en el capítulo 9 se presenta la bibliografía y referencias que se han utilizado como consulta para poder desarrollar este Proyecto.

Como parte adicional, se han adjuntado tres capítulos, denominados Anexos.

En el Anexo A podemos encontrar el enunciado de la práctica utilizada en el capítulo 2.

En el Anexo B se explica la funcionalidad de cada uno de los patrones de diseño utilizados.

Por último, en el anexo C se explica la arquitectura .NET Remoting, sus elementos de distribución y pasos a seguir para su implementación en una aplicación a distribuir.

Además, con esta memoria va un CD en el que se incluye el código de las aplicaciones y sus archivos ejecutables.

Capítulo 2.
Java y C#.

2.1. Introducción.

El lenguaje utilizado para desarrollar la aplicación de este proyecto, es el lenguaje de programación C#.

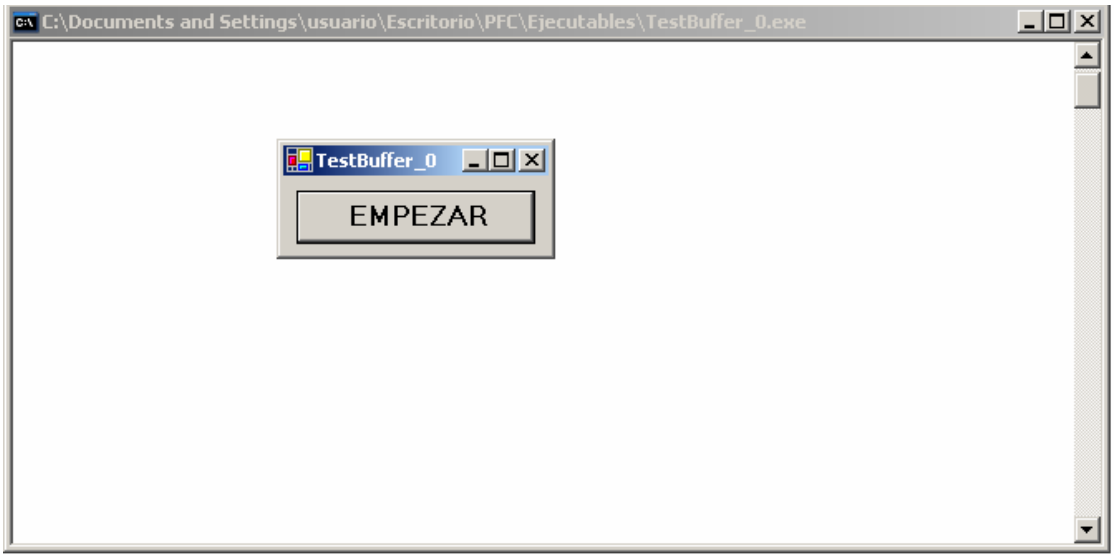
Una de las finalidades de este proyecto es desarrollar el programa "Simulador C# de una maqueta de trenes" basándonos en el proyecto [1]. O sea, pasar de lenguaje Java a lenguaje C#.

Para llevar a cabo este proyecto, se partió de una fase de familiarización con el nuevo lenguaje. Esto consistió en traducir una práctica de la asignatura Complementos de Informática. Se recomienda la lectura del enunciado de esta práctica que se encuentra en el anexo A. Esta práctica está orientada al lenguaje Java, el impartido en la asignatura. Con este proyecto se desea contribuir a que el alumno que en adelante curse esta asignatura, pueda tener la opción de elegir un lenguaje u otro, o incluso los dos, para así abarcar un mayor conocimiento de las tecnologías que se encontrará en su vida profesional.

2.2. Ejecución del programa de la práctica.

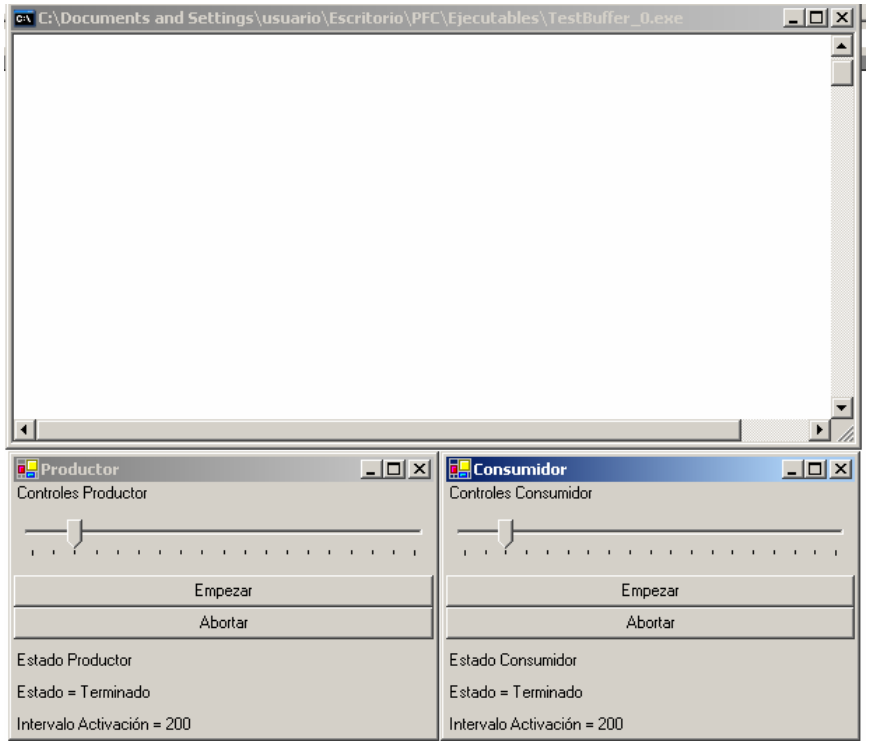
Para ejecutar el programa de la práctica, haga doble clic sobre el archivo ejecutable **`TestBuffer_0.exe`**, que se encuentra en la carpeta **`Capitulo 2/Ejecutables`** del CD que acompaña a este proyecto.

En la siguiente figura se observa una consola donde aparecerá la información generada por el programa y un botón que al ser pulsado dará inicio a la aplicación.



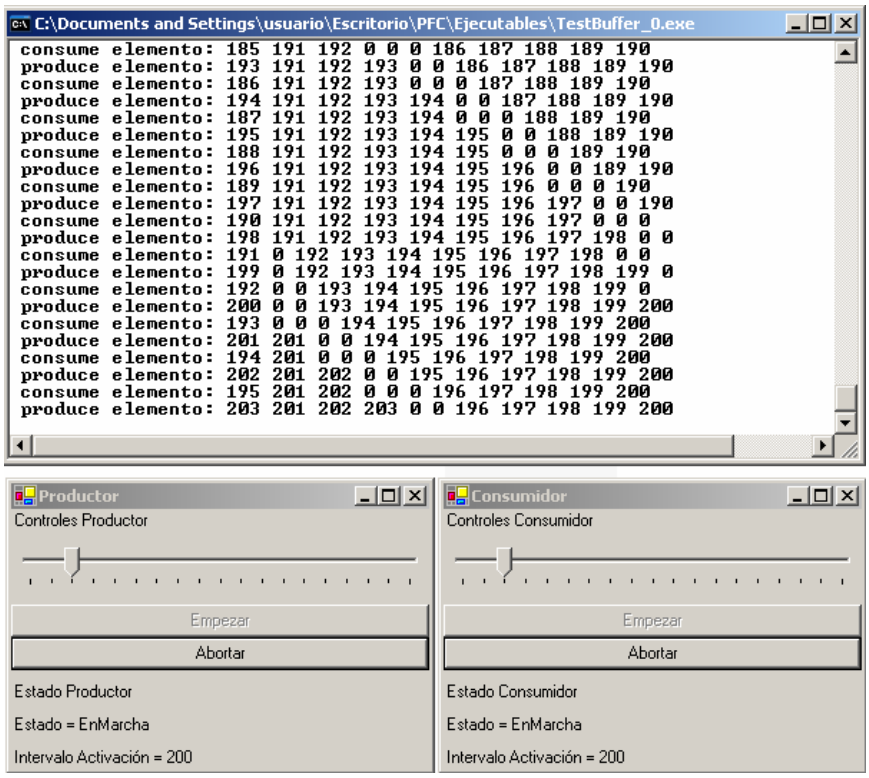
***Resultado de hacer doble clic sobre el archivo* `TestBuffer_0.exe`.**

En la siguiente figura se observa el resultado de pulsar el botón **`Empezar`**, aparecen dos ventanas, una que será la vista y el controlador del Productor y la otra la del Consumidor.



***Resultado de pulsar el botón* `Empezar`.**

En la siguiente figura se observa el resultado de, primero, pulsar el botón **`Empezar`** del Productor y segundo, pulsar el botón **`Empezar`** del Consumidor.



Resultado de pulsar el botón `Empezar` del Productor y del Consumidor, respectivamente.

Cada ventana está compuesta de su controlador y de su vista. El controlador está formado por una barra de desplazamiento, encargada de modificar el intervalo de activación, el botón **`Empezar`**, el cual inicia la producción o consumo de enteros, según corresponda, y el botón **`Abortar`**, el cual finaliza la actividad iniciada por el botón **`Empezar`**. Cualquier actuación sobre cualquiera de los controles del controlador mencionados anteriormente se refleja en la vista, en su leyenda correspondiente.

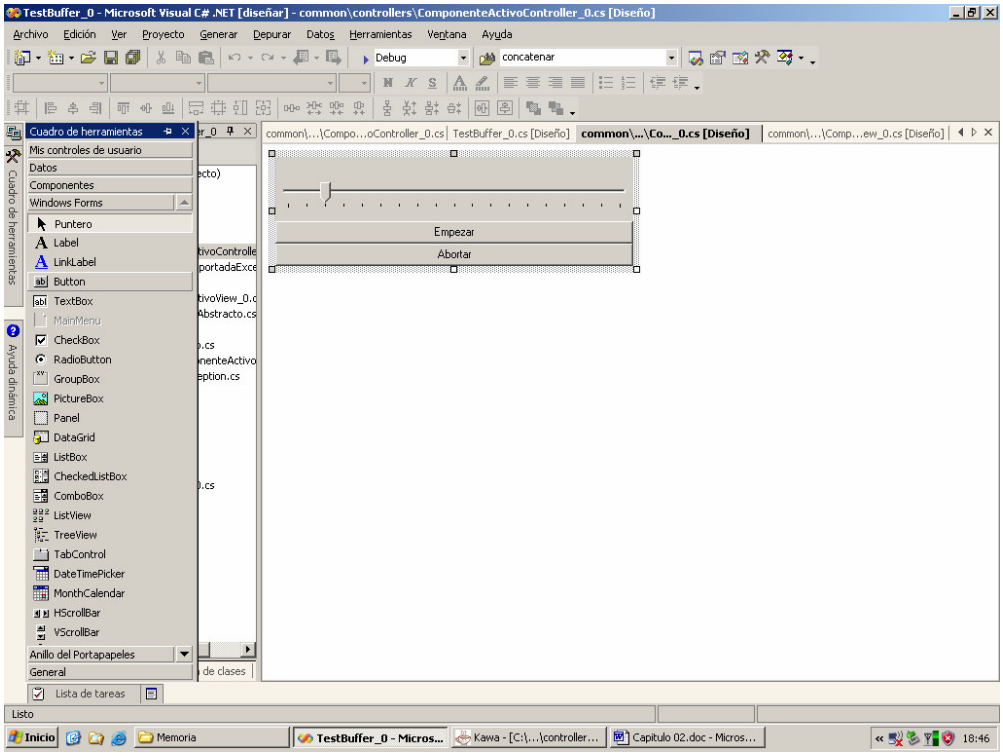
Como se observa, la aplicación cumple con la especificación solicitada en el enunciado de la práctica.

2.3. Traducción del código Java a C#.

A partir del código Java de la solución de la práctica proporcionada por el profesor, se trabajó para obtener su equivalente en C#. La finalidad de este apartado es comparar ambos códigos, centrándonos en código que realiza la misma función pero con distinta sintaxis.

Para la confección del código en C# se ha seguido la misma estructura en cuanto a clases, variables… del programa en java.

La primera gran diferencia con la que nos enfrentamos es en la creación de la interfaz gráfica, para ello hemos utilizado el Diseñador de Windows Forms, mediante el cual seleccionamos de forma gráfica los componentes que integran la aplicación. Automáticamente disponemos del código generado por la acción anterior. A partir de este código añadiremos funcionalidad a los componentes.



Ejemplo de uso del Diseñador de Windows Forms.

C# y Java son lenguajes con una sintaxis muy parecida. Además, ambos interpretan de una forma bastante similar los conceptos de orientación a objetos, siendo posible en muchos casos una traducción casi directa.

- ***Palabras clave.***

Hay una gran cantidad de semejanzas sintácticas entre Java y C#, prácticamente casi cada palabra clave de Java su equivalente en C# excepto unas pocas como **`transient`**, **`throws`** y **`strictfp`**. En la tabla se muestran las respectivas palabras equivalentes.

C#	Java	C#	Java	C#	Java	C#	Java
abstract	abstract	extern	native	operator	N/A	throw	throw
As	N/A	false	false	out	N/A	true	true

base	Super	finally	finally	override	N/A	try	try
bool	boolean	fixed	N/A	params	...	typeof	N/A
break	Break	float	float	partial	N/A	uint	N/A
byte	N/A	For	for	private	private	ulong	N/A
case	Case	foreach	for	protected	N/A	unchecked	N/A
catch	Match	Get	N/A	public	public	unsafe	N/A
char	Char	goto	goto	readonly	N/A	ushort	N/A
checked	N/A	If	if	ref	N/A	using	import
class	class	implicit	N/A	return	return	value	N/A
const	const	In	N/A	sbyte	byte	virtual	N/A
continue	continue	Int	int	sealed	final	void	void
decimal	N/A	interface	interface	set	N/A	volatile	volatile
default	default	internal	protected	short	short	where	extends
delegate	N/A	Is	instanceof	sizeof	N/A	while	while
Do	do	lock	synchronized	stackalloc	N/A	yield	N/A
double	double	long	long	static	static	:	extends
else	else	namespace	package	string	N/A	:	implements
enum	N/A	New	new	struct	N/A	N/A	strictfp
event	N/A	null	null	switch	switch	N/A	throws
explicit	N/A	object	N/A	this	this	N/A	transient

- ***Clases no heredables.***

Java y C# proporcionan mecanismos para especificar que una clase debe ser la última en una jerarquía de herencia y no se puede utilizar como clase base. En Java esto se hace precediendo a la declaración de la clase con la palabra clave **`final`** mientras que en C# esto se hace precediendo a la declaración de la clase con la palabra clave **`sealed`**.

C# Code

```
sealed class Student {
          string fname;
          string lname;
          int uid;
          void attendClass() {}
}
```

Java Code

```
final class Student {
          String fname;
          String lname;
          int uid;
```

```
            void attendClass() {}
}
```

- ***Método Main().***

El punto de entrada de los programas de C# y de Java es un método principal. Hay una diferencia superficial en que el método principal en C# comienza con un "M mayúscula" (al igual que todos los nombres del método del marco de .NET, por la convención) mientras que el método principal en Java comienza con un "m minúscula" (al igual que todos los nombres del método de Java, por la convención). La declaración del método principal es igual en ambos casos a excepción de que la declaración del método principal en C# puede tener un parámetro **`void`**.

C# Code

```
using System;

class A {
          public static void Main(String[] args) {
                    Console.WriteLine("Hello World");
          }
}
```

Java Code

```
class A {
          public static void main(String[] args) {
                    System.out.println("Hello World");
          }
}
```

- ***Sintaxis de la herencia.***

C# utiliza la sintaxis de C++ para la herencia, para la herencia de clases y la implementación de interfaces en contraposición a las palabras clave de Java **`extends`** e **`implements`**.

C# Code

```
using System;

class B : A , IComparable {
          int CompareTo(){}

          public static void Main(String[] args) {
                    Console.WriteLine("Hello World");
          }
}
```

Java Code

```
class A extends A implements Comparable {

          int compareTo(){}

          public static void main(String[] args) {
                    System.out.println("Hello World");
          }
```

```
}
```

- ***Namespaces.***

Un **namespace** en C# es una manera de agrupar una colección de clases y es de una forma similar usado a como los paquetes de Java. Los usuarios de C++ notarán las semejanzas entre el sintaxis del namespace de C# y ése en C++. En Java, los nombres del paquete dictan la estructura del directorio de los archivos de fuente en un uso mientras que en C# los namespaces no dictan la disposición física de los archivos de fuente en directorios solamente su estructura lógica.

C# Code

```
namespace com.carnage4life {
          public class MyClass {
                    int x;
                    void doStuff(){}
          }
}
```

Java Code

```
package com.carnage4life;

public class MyClass {
          int x;
          void doStuff(){}
}
```

La sintaxis del namespace de C# permite namespaces anidados.

C# Code

```
using System;

namespace Company {
          public class MyClass { /* Company.MyClass */
                    int x;
                    void doStuff() {}
          }

          namespace Carnage4life {
                    public class MyOtherClass {  /* Company.Carnage4life.MyOtherClass */
                              int y;
                              void doOtherStuff(){}

                              public static void Main(string[] args) {
                                        Console.WriteLine("Hey, I can nest namespaces");
                              }
                    }// class MyOtherClass
          }// namespace Carnage4life
}// namespace Company
```

- ***Constructores, Destructores y Finalizadores.***

La sintaxis y la semántica para los constructores en C# es idéntica en Java. C# también tiene el concepto de los destructores que utilizan una sintaxis similar a la sintaxis del destructor de C++ pero tiene sobre todo la misma semántica que los finalizadores de Java.

C# Code

```
using System;

public class MyClass {
          static int num_created = 0;
          int i = 0;

          MyClass(){
                    i = ++num_created;
                    Console.WriteLine("Created object #" + i);
          }

          ~MyClass(){
                    Console.WriteLine("Object #" + i + " is being finalized");
          }

          public static void Main(string[] args){
                    for(int i=0; i < 10000; i++)          new MyClass();
          }
}
```

Java Code

```
public class MyClass {
          static int num_created = 0;
          int i = 0;

          MyClass(){
                    i = ++num_created;
                    System.out.println("Created object #" + i);
          }

          public void finalize(){
                    System.out.println("Object #" + i + " is being finalized");
          }

          public static void main(String[] args){
                    for(int i=0; i < 10000; i++)          new MyClass();
          }
}
```

- ***Sincronización de los métodos y bloques de código.***

En Java es posible especificar bloques de código sincronizados que nos asegure que sólo un hilo puede acceder a un determinado objeto a la vez y, a continuación, crear una sección crítica de código. C # proporciona el bloqueo, que es semánticamente idéntico a la declaración sincronizada en Java.

C# Code

```
public void WithdrawAmount(int num {
          lock(this){
                    if(num < this.amount)
                              this.amount -= num;
          }
}
```

Java Code

```
public void withdrawAmount(int num) {
          synchronized(this) {
                    if(num < this.amount)
                              this.amount -= num;
          }
}
```

- ***Declaración de Constantes.***

C# Code

```
using System;

public class ConstantTest{

    /* Constantes en tiempo de compilación */
    const int i1 = 10;   // Implícitamente una variable estática.

    // El código no compila por la palabra clave static.
    // public static const int i2 = 20;

    /* Constantes en tiempo de ejecución */
    public static readonly uint l1 =  (uint) DateTime.Now.Ticks;

    /* Referencia a objeto como constante. */
    readonly Object o = new Object();

    /* Variable de solo lectura no inicializada. */
    readonly float f;

    ConstantTest() {
    // Esta tiene que ser inicializada en el constructor.
    f = 17.21f;
    }
}
```

Java Code

```
import java.util.*;

public class ConstantTest{

    /* Constantes en tiempo de compilación */
    final int i1 = 10;   // Variable de instancia.
    static final int i2 = 20; // Variable de clase.

    /* Constantes en tiempo de compilación. */
    public static final long l1 = new Date().getTime();

    /* Referencia a objeto como constante. */
    final Vector v = new Vector();

    /* Variable final no inicializada. */
    final float f;

    ConstantTest() {
    // Esta tiene que ser inicializada en el constructor.
    f = 17.21f;
    }
}
```

- ***Métodos virtuales.***

C# Code

```
using System;

public class Parent {
        public virtual void DoStuff(string str) {
                Console.WriteLine("In Parent.DoStuff: " + str);
        }
}
```

```
public class Child: Parent {
          public void DoStuff(int n) {
                    Console.WriteLine("In Child.DoStuff: " + n);
          }

          public override void DoStuff(string str) {
                    Console.WriteLine("In Child.DoStuff: " + str);
          }
}

public class VirtualTest {
          public static void Main(string[] args) {

                    Child ch = new Child();

                    ch.DoStuff(100);
                    ch.DoStuff("Test");
                    // Hace referencia al método redefinido en la clase hija.
                    ((Parent) ch).DoStuff("Second Test");
          }
}//VirtualTest
```

Java Code

```
class Parent {
          public void DoStuff(String str){
          System.out.println("In Parent.DoStuff: " + str);
          }
}

class Child extends Parent {
          public void DoStuff(int n) {
          System.out.println("In Child.DoStuff: " + n);
          }

          public void DoStuff(String str) {
          System.out.println("In Child.DoStuff: " + str);
          }
}

public class VirtualTest {
          public static void main(String[] args) {
                    Child ch = new Child();

                    ch.DoStuff(100);
                    ch.DoStuff("Test");
                    ((Parent) ch).DoStuff("Second Test");
          }
}//VirtualTest
```

- ***Propiedades, índices y sobrecarga de operadores.***

Las propiedades permiten acceder a campos privados de forma controlada usando sintaxis de asignación. Alternativa a métodos **`get/set`** utilizados en Java.

Los índices permiten acceder a un array encapsulado en una clase con el nombre de la clase.

C# Code

```
// Propiedades.
public int Impropiedad
{
          get { return MiVar; }
```

```
            set { MiVar = value; }
}

// Índices.
public string this[int index]
{
            get { return MiArray[index]; }

            set { MyArray[index] = value; }
}

// Sobrecarga de operadores.
public static MiClase operador + (MiClase a, MiClase b)
{
            return new MiClase(a.MiVar + b.MiVar);
}
```

- ***Paso por referencia.***

En Java, los argumentos pasados a métodos lo son por valor, trabajando el método sobre copias de los mismos. En C# se puede decidir si pasar una copia o una referencia directamente a él.

Java Code

```
class PassByRefTest {
          public static void changeMe(String s) {
                    s = "Changed";
          }

          public static void swap(int x, int y) {
                    int z = x;
                    x = y;
                    y = z;

          }

          public static void main(String[] args) {
                    int a = 5, b = 10;
                    String s = "Unchanged";

                    swap(a, b);
                    changeMe(s);

                    System.out.println("a := " + a + ", b := " + b + ", s = " + s);
          }
}
```

OUTPUT

```
a := 5, b := 10, s = Unchanged
```

C# Code

```
using System;

class PassByRefTest {
          public static void ChangeMe(out string s) {
                    s = "Changed";
          }

          public static void Swap(ref int x, ref int y) {
                    int z = x;
                    x = y;
                    y = z;
          }
```

```
        public static void Main(string[] args) {
                int a = 5, b = 10;
                string s;

                Swap(ref a, ref b);
                ChangeMe(out s);

                Console.WriteLine("a := " + a + ", b := " + b + ", s = " + s);
        }
}
```

OUTPUT

```
a := 10, b := 5, s = Changed
```

- ***Arrays multidimensionales y "dentados".***

C# Code

```
using System;

public class ArrayTest {
       public static void Main(string[] args) {
          int[,] multi = { {0, 1}, {2, 3}, {4, 5}, {6, 7} };  // Array multidimensional.

          for(int i=0, size = multi.GetLength(0); i < size; i++)
               for(int j=0, size2 = multi.GetLength(1); j < size2; j++)
                    Console.WriteLine("multi[" + i + "," + j + "] = " + multi[i,j]);

          int[][] jagged = new int[4][];  // Array dentado.
          jagged[0] = new int[2]{0, 1};
          jagged[1] = new int[2]{2, 3};
          jagged[2] = new int[2]{4, 5};
          jagged[3] = new int[2]{6, 7};

          for(int i=0, size = jagged.Length; i < size; i++)
                for(int j=0, size2 = jagged[1].Length; j < size2; j++)
                    Console.WriteLine("jagged[" + i + "][" + j + "] = " + jagged[i][j]);
       }
} // ArrayTest
```

Las dificultades de traducción vienen dadas casi siempre por las llamadas a los paquetes pre-existentes y al uso de los frameworks que vienen incorporados al lenguaje. No obstante, si hay en algunos casos diferencias importantes, de las cuales se van a explicar dos: el manejo de eventos y la creación de hilos.

- ***Manejo de eventos.***

Vamos a comparar el manejo de eventos. Como muestra vamos a centrarnos en cómo ambos lenguajes actúan al presionar los botones **`Empezar`** y **`Abortar`**.

Java Code

```
…
btEmpezar.addActionListener(this);
btAbortar.addActionListener(this);
…
```

```
public void actionPerformed(ActionEvent ae)
{
          try {
                    if (ae.getSource() == btEmpezar)
                    {
                              ca.empezar();
                              return;
                    }

                    if (ae.getSource() == btAbortar)
                    {
                              ca.parar();
                              return;
                    }
          }
          catch(TransicionIlegalException e) {
                    e.printStackTrace();
          }
}
…
```

C# Code

```
…
this.btEmpezar.Click += new System.EventHandler(this.btEmpezar_Click);
this.btAbortar.Click += new System.EventHandler(this.btAbortar_Click);
…
private void btEmpezar_Click(object sender, System.EventArgs e)
{
          try
          {
                    ca.empezar();
                    this.setEstadoBotones();
                    return ;
          }
          catch(TransicionIlegalException tie)
          {
                    Console.Error.WriteLine( tie.StackTrace );
          }
}

private void btAbortar_Click(object sender, System.EventArgs e)
{
          try
          {
                    ca.parar();
                    this.setEstadoBotones();
                    return ;
          }
          catch(TransicionIlegalException tie)
          {
                    Console.Error.WriteLine( tie.StackTrace );
          }
}
…
```

En el caso de Java nos limitamos a exponer el código, mientras que en el caso de C# nos fijamos en las dos primeras líneas:

```
this.btEmpezar.Click += new System.EventHandler(this.btEmpezar_Click);
this.btAbortar.Click += new System.EventHandler(this.btAbortar_Click);
```

Aquí se le está indicando al programa que al hacer clic sobre el botón **Empezar** (un botón puede generar eventos de diversos tipos), este evento sea controlado por el método que se le pasa como argumento, e igualmente con el botón **Abortar**.

"**+=**" significa que se agrega un controlador de eventos a la cadena de eventos. Para eliminarlo se utiliza "**-=**".

.NET Framework incluye un tipo de delegado integrado llamado **EventHandler**, que se puede utilizar para declarar controladores de eventos en los que no se necesita información adicional. Un delegado es un objeto que puede hacer referencia a un método. Por tanto, cuando se crea un delegado, se crea un objeto que puede guardar la referencia a un método. Más aún, al método se le puede llamar mediante esta referencia. De esta manera, un delegado puede invocar al método al que hace referencia.

En la segunda parte de este código están los métodos que contienen el código que controla el evento del objeto correspondiente.

- ***Hilos.***

En Java, todas las clases, **wait()**, **notify()** and **notifyAll()** heredan de **java.lang.Object** para operaciones relacionadas con hilos. Los métodos equivalentes en C# son **Wait()**, **Pulse()** y **PulseAll()**, respectivamente, de la clase **System.Threading.Monitor**.

C# Code

```
public void workCompleted(WorkerThread worker){
          try {

                    lock(this) {
                        while(worker.getIDNumber() != NextInLine()){

                              try {
                                        Monitor.Wait(this, Timeout.Infinite);

                              } catch (ThreadInterruptedException e) {}

                        }//while

                RemoveNextInLine();

                Monitor.PulseAll(this);

        }

    }catch(SynchronizationLockException){ }
}
```

```
public synchronized void workCompleted(WorkerThread worker){
          while(worker.getIDNumber().equals(nextInLine())==false) {
                    try {
                              wait();
```

```
                    } catch (InterruptedException e) {}

          }//while

          removeNextInLine();

          notifyAll();
}
```

Capítulo 3.
Diseño General.

3.1. Introducción.

Para llevar a cabo los objetivos de este proyecto, a continuación se explican las estrategias llevadas a cabo para conseguirlo, como lo son el uso de middleware de comunicaciones para llevar a cabo la distribución, la estructura del programa determinada por el patrón Modelo-Vista-Controlador. El diseño de partida del proyecto es el establecido en el proyecto [1], el cual hemos utilizado y explicaremos a continuación.

3.2. Uso de middleware.

Para lograr la distribución se opta por utilizar middleware de comunicaciones. El uso de middleware permite precisamente conseguir homogeneidad entre llamadas locales y remotas permitiendo distribuir el sistema por toda la maqueta.

El middleware nos ofrece las siguientes ventajas:

- Permite usar máquinas de poca capacidad.
- Permite usar hardware ya existente, evitando comprar hardware nuevo.
- Ayuda a mantener y mejorar el tiempo de respuesta.
- Es rápido de implementar.

Si bien hay que contar también con algunos inconvenientes:

- Alto consumo de recursos, tanto de CPU como de memoria RAM.
- Debe desarrollarse de forma específica.
- Costo de desarrollo, implementación
- Capacidad limitada, no entrega todas las soluciones
- Requiere cierta complejidad logística a la hora de instalar en cada equipo.

Existen otras formas para cumplir estos requisitos, como hacer un Proxy a medida (oculta la información) o un pequeño Broker. Pero como .NET incluye un middleware comercial que satisface nuestros requisitos no merece la pena acudir a un desarrollo a medida. Ejemplos de middlewares para implementar sistemas distribuidos son CORBA, .NET Remoting, Java RMI, etc.

3.3. Uso del patrón MVC.

Se ha optado por la utilización del patrón MVC (Modelo-Vista-Controlador). Este patrón es una de las herramientas básicas de cualquier desarrollador de GUI, estableciendo una independencia entre los módulos de la aplicación, para facilitar la mantenibilidad y la sustitución de diferentes componentes gráficos. Además, este patrón nos ofrece la posibilidad de componer diferentes vistas simultáneas sincronizadas de un mismo modelo.

De esta manera, se mantiene desacoplada la parte de las vistas y la parte del modelo, por lo que no hay que cambiar el modelo por el hecho de cambiar las vistas. La sustitución y extensión de los módulos de la aplicación no es tarea difícil, facilitando mucho los cambios y extensiones de la GUI sin afectar al modelo. La utilización de este patrón presenta algunos inconvenientes, debido a que requiere una implementación compleja, es difícil diseñar la jerarquía de componentes, y tanto la vista como el controlador están muy fuertemente acoplados, si bien no supone un gran problema ya que ambos módulos suelen tener una comunicación directa en esta aplicación.

De acuerdo con lo dicho, la aplicación se divide en tres áreas: procesamiento, entrada y salida, que se corresponden con los tres tipos de módulos del patrón arquitectónico base utilizado (modelo, vista y controlador). La capa del modelo mantiene referencias de todas las vistas y les notifica los cambios de estado, producidos por la solicitud de un servicio por medio de un controlador o de algún evento sincronizado interno, mediante el patrón Observador. MVC se apoya a su vez en los patrones Observador y Estrategia, utilizándolos de manera implícita.

Para agilizar la lectura del Proyecto y evitar la consulta de otras fuentes, se incluye una descripción de estos patrones en el Anexo B “Patrones de Diseño”, situado al final de esta documentación. Se recomienda empezar la lectura por los patrones Estrategia y Observador, para entender el funcionamiento de MVC. A continuación se exponen ciertas características que afectan a la implementación de la aplicación, no explicadas anteriormente, sobre la utilización de estos patrones:

- Se definen familias de vistas/controladores relacionadas, pudiendo definirse por diferentes modelos.
- Se limitan el número de clases derivadas.
- Puede cambiarse la vista/controlador (estrategia) dinámicamente.
- Se eliminan o reducen el número de sentencias condicionales.

Como puntos negativos cabe destacar que:

- El modelo debe conocer las vistas/controladores (estrategias).
- Existe una necesidad de comunicación entre el modelo y la vista/controlador (estrategia).
- Se incrementan el número de objetos.

3.4. Tipos de datos del modelo.

Sus elementos principales son los siguientes, que serán los tipos de datos definidos en la aplicación:

1. Las vías.
2. Las agujas.
3. Los sensores.
4. Las vistas, de las vías, agujas y sensores.
5. Los trenes.

Estos elementos, representan a su vez, a todos los elementos simples que conforman la maqueta, de manera independiente, tomados de entre los disponibles en el catálogo de la casa Märklin.

A cada uno de los elementos de la maqueta del simulador, correspondientes con su respectivo elemento de la maqueta real, se le establece su modelo (con cierta funcionalidad), su vista asociada, y si dispone de él, su controlador, para poder interactuar con el modelo.

Se puede apreciar que tanto los tramos de vía, como las agujas y los sensores que componen la vía férrea del simulador, en relación con la maqueta real, tienen una vista asociada. Cada vista es un objeto estrategia y por tanto todas tienen una interfaz común (aquí es donde entran en juego los patrones Estrategia y Observador). Se usan diferentes vistas (estrategias) para la representación de los datos del modelo. La disposición de las vistas en función del modelo es gestionada a través de la interfaz común. Además, en cualquier momento es posible cambiar una vista asociada a un modelo. Por lo tanto, realmente las vistas son observadores de un modelo y éste las actualiza (si se han suscrito) mediante el patrón Observador.

Con los controladores ocurre lo mismo. Son objetos estrategia, y todos tienen una interfaz común. Dependiendo del controlador, cambia la implementación.

No se genera una vista para los trenes, sino para los elementos de la vía férrea, es decir, los trenes no tienen vista asociada porque no interesa visualizar el tren, sino por dónde pasa, es

decir, la localización del tren en cada instante. Esta es una característica importante de diseño.

3.5. Implementación de la vía férrea y los trenes.

Tanto la vía férrea como los trenes son implementados como listas enlazadas o grafos ya que facilitan mucho el cambio de topología y la adición/eliminación de elementos de la maqueta.

En este Proyecto, al necesitar establecer una relación de avance y retroceso de los trenes sobre los nodos, la vía férrea se ha implementado mediante listas doblemente enlazadas. Los nodos se organizan, de modo que cada uno apunta al siguiente y al anterior. El grafo resultante representa a la maqueta.

Además en la teoría de la computación existe mucha algoritmia relacionada con el manejo de datos, lo cual puede estar muy bien a la hora de ampliar este Proyecto. Los grafos son un clásico de la ciencia de la computación.

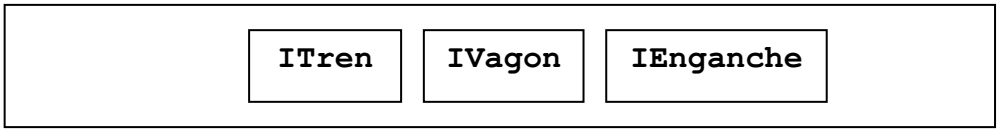
La aguja se implementará con un tipo de nodo especial porque en vez de tener un sucesor, puede tener dos posibles sucesores.

3.6. Interfaces

Otro criterio de diseño general importante es proporcionar las interfaces que se han definido. Los elementos antes mencionados se modelan en las siguientes interfaces y clases:

- ✓ **Trenes**

El comportamiento de un tren está definido en la interfaz **ITren**. Para la aplicación, es necesario definir esta interfaz, porque el tren puede ser un tren real o un tren simulado, por lo que la implementación cambia por completo, pero los dos trenes, tanto el real como el simulado tienen que tener la misma interfaz.



Interfaces ***ITren***, ***IVagon***, ***IEnganche***.

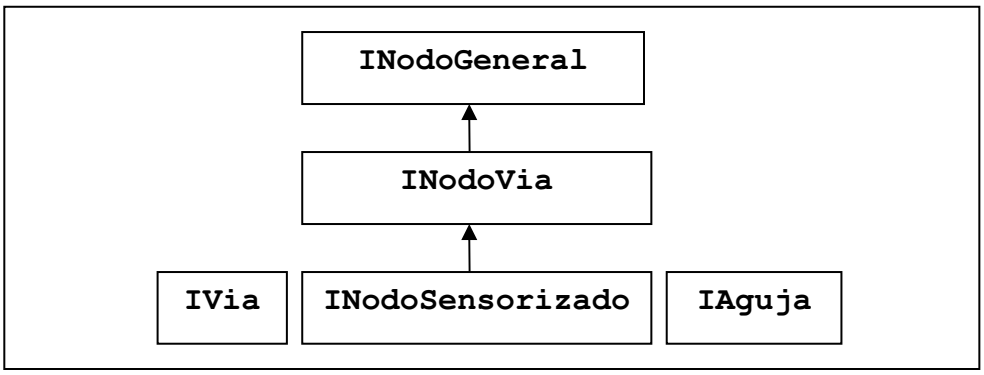
Resumen de las interfaces **`ITren`**, **`IVagon`** *e* **`IEnganche`**	
`ITren`: Interfaz común para modelar un tren, definiendo sus funcionalidades básicas.	
Servicios	**Descripción**
`void avanzar( );`	Permite avanzar a un objeto Tren.
`void decVelocidad( );`	Decrementa la velocidad de un objeto Tren.
`void fijarVelocidad( double v );`	Establece la velocidad que tendrá cada Tren.
`void incVelocidad( );`	Incrementa la velocidad de un objeto Tren.
`double obtenerVelocidad( );`	Devuelve la velocidad que tiene cada Tren.
`void parar( );`	Método para parar el movimiento del Tren.
`void ponerVagon( IEnganche v );`	Método para agregar un vagón al Tren.
`void quitar( IObservadorTren obs );`	Permite eliminar observadores de cada Tren.
`void retroceder( );`	Permite retroceder a un objeto Tren.
`void suscribir( IObservadorTren obs );`	Permite suscribir a los observadores de cada Tren.
`IVagon`: Interfaz común para modelar un vagón de tren, definiendo sus funcionalidades básicas	
Servicios	**Descripción**
`IVagon concatenar( IEnganche vg );`	Permite enganchar un vagón con otro
`IEnganche inicio( );`	Obtiene el enganche frontal de un vagón
`IEnganche fin( );`	Obtiene el enganche trasero de un vagón
`IEnganche`: Interfaz de marca para que ocultar la implementación que hay por debajo	

Cabe destacar que **`IVagon`** presenta la funcionalidad usada para establecer la longitud de un determinado tren.

Por otra parte, la interfaz **`IEnganche`** se ha definido para modelar el tipo de datos que permite enganchar desde los métodos de concatenación de vagones de cada tren.

- ✓ **Vías, agujas y sensores**

Otros elementos principales son las vías, las agujas y los sensores. Existe una interfaz que representa cada uno de ellos. Tanto el comportamiento de una vía, una aguja y un sensor está definido en las interfaces **`INodoGeneral`** e **`INodoVia`**. Estas interfaces definen la funcionalidad común que tienen estos elementos, sea cual sea cada uno de ellos. A su vez, el comportamiento de cada elemento está definido también por otra interfaz, ya más específica, de acuerdo a las características de funcionalidad que ofrezca el elemento. La interfaz de una vía es **`IVia`**, la de la aguja es **`IAguja`** y la de un sensor es **`INodoSensorizado`**.



*Interfaces **INodoGeneral**, **INodoVia**, **IVia**, **IAguja** y **INodoSensorizado**.*

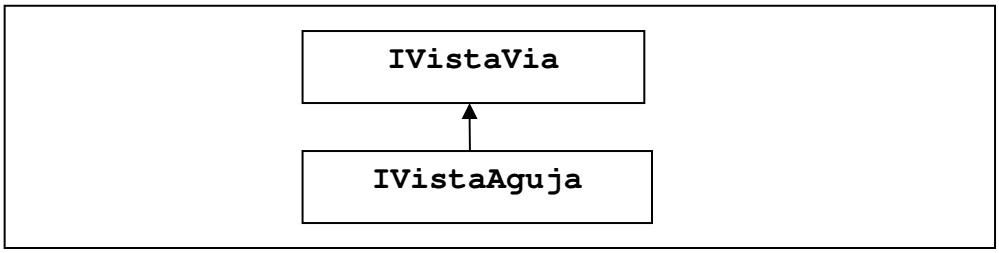
A continuación se presenta una breve descripción, tipo resumen, de las interfaces anteriores y su contenido:

*Resumen de las interfaces **INodoGeneral**, **INodoVia**, **IVia**, **IAguja** y **INodoSensorizado***	
INodoGeneral: Interfaz común para modelar los objetos de las clases que representan los elementos principales que forman la vía férrea de la maqueta. Funcionalidades básicas.	
Servicios	**Descripción**
`INodoGeneral anterior( );`	Método que devuelve el nodo anterior al actual, es decir, al nodo en el que se encuentra el tren.
`INodoGeneral concatenar( INodoGeneral ns );`	Método que concatena dos nodos.
`INodoGeneral siguiente( );`	Método que devuelve el nodo siguiente al actual, es decir, al nodo en el que se encuentra el tren.
INodoVia: Interfaz común para modelar los objetos de las clases que representan los elementos principales que forman la vía férrea de la maqueta. Funcionalidades más específicas.	
Servicios	**Descripción**
`bool hayNodoTren( );`	Método para indicar si hay algún objeto tren ya colocado sobre un elemento de la vía férrea.
`void ponerNodoTren( INodoTren nt );`	Método para mostrar la vista que tendrá el objeto tren en la nueva posición de la vía férrea.
`void quitarNodoTren( );`	Método para eliminar la vista del objeto tren cuando se mueva, es decir, cuando pase a otro elemento de la vía férrea.
`IVistaVia VistaVia{ get; set; };`	Propiedad de lectura y escritura de la vista que utiliza la vía.
`void suscribir( IObservadorVia obs );`	Suscribimos observadores a lista de observadores de la vía.
`void quitar( IObservadorVia obs );`	Eliminamos observados de la lista de observadores de la vía.
IVia: Interfaz común para modelar los objetos de las clases que representan los tramos de vía de la maqueta.	
Servicios	**Descripción**
`bool Ocupado{ get; set; };`	Propiedad de lectura y escritura del estado de un nodo de vía
`IVistaVia VistaVia{ get; set; };`	Propiedad de lectura y escritura de la

	vista que utiliza la vía
`void quitar( IObservadorVia obs );`	Eliminamos observadores de la lista de observadores de la vía
`void suscribir( IObservadorVia obs );`	Suscribimos observadores a la lista de observadores de la vía
`IAguja:` Interfaz común para modelar los objetos de las clases que representan las agujas que se encuentran en la vía férrea de la maqueta.	
Servicios	**Descripción**
`bool Estado{ get; };`	Propiedad de lectura del estado en el que se encuentra la aguja, recto o curvo.
`IVistaAguja VistaAguja{ get; set; };`	Propiedad de lectura y escritura de la vista que utiliza la aguja.
`void quitar( IObservadorAguja obs );`	Eliminamos observadores de la lista de observadores de un nodo aguja.
`void setCurvo( );`	Método para enlazar la aguja con el siguiente tramo de la vía, con estado curvo.
`INodoVia Curvo{ set; };`	Propiedad de escritura del siguiente nodo de Vía que tendrá una aguja si está en estado curvo.
`void setRecto( );`	Método para enlazar la aguja con el siguiente tramo de la vía, con estado recto.
`INodoVia Recto{ set; };`	Propiedad de escritura del siguiente nodo de Vía que tendrá una aguja si está en estado recto.
`void setSentido( int s, int ud )`	Fija el sentido de la aguja, y la orientación que tendrá cuando esté en estado curvo.
`void suscribir( IObservadorAguja obs );`	Suscribimos observadores a la lista de observadores de un nodo aguja.
`INodoSensorizado:` Interfaz común para modelar los objetos de las clases que representan los sensores que se encuentran en la vía férrea de la maqueta.	
Servicios	**Descripción**
`void activate( );`	Se encarga de activar el sensor.
`void deactivate( );`	Se encarga de desactivar el sensor.
`bool estaActivo( );`	Se encarga de devolver el estado del sensor.
`IVistaVia VistaSensor{ get; set; };`	Propiedad de lectura y escritura de la vista que utiliza el objeto sensor.

✓ Vistas

Por último, el último grupo de elementos principales que queda por definir, son las vistas. La interfaz definida para las vistas es **`IVistaVia`**, y presenta la funcionalidad que tendrán las vistas de los diferentes elementos que tiene la vía férrea: tramos de vía, agujas y sensores. La particularidad que diferencia a la interfaz **`IVistaAguja`** de **`IVistaVia`**, es que define nuevas funcionalidades además de las que tiene una vía.



*Interfaces **IVistaVia** e **IVistaAguja***

Tal y como se ha realizado con los anteriores elementos principales se presenta una breve descripción, tipo resumen, de las interfaces anteriores y su contenido.

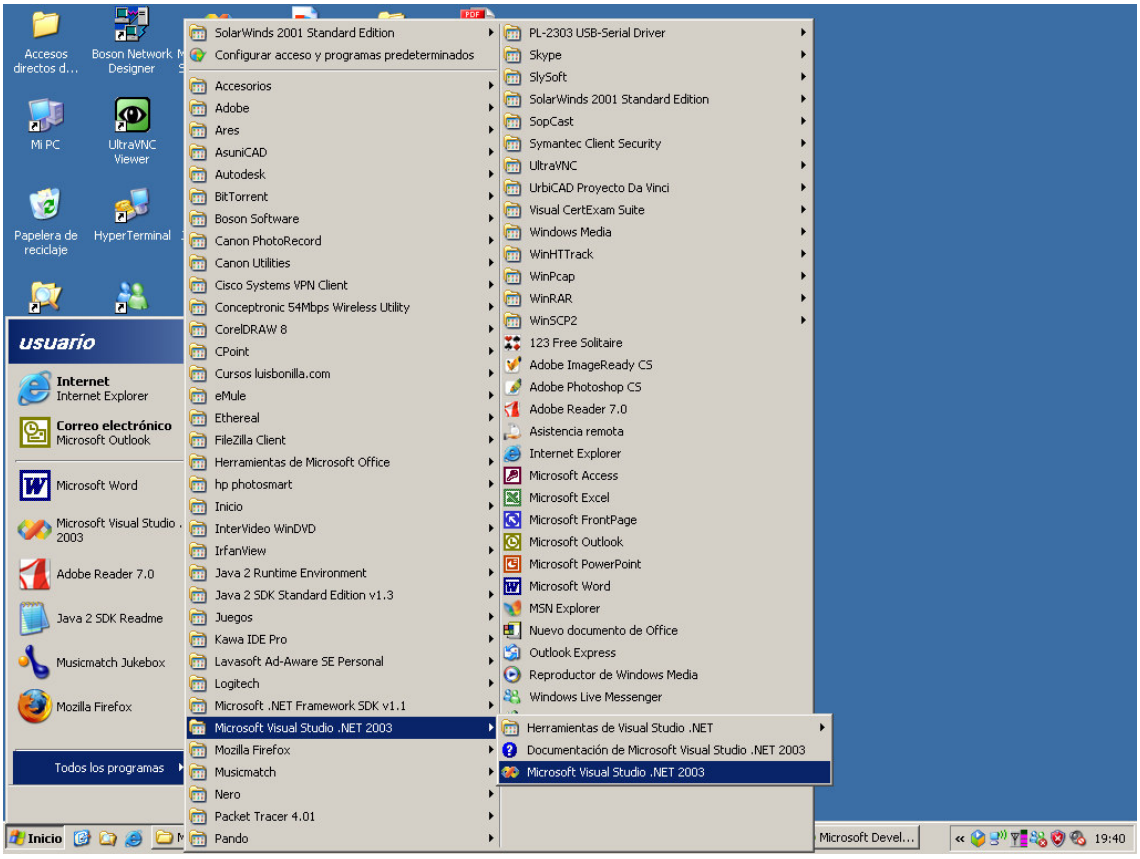
*Resumen de las interfaces **IVistaVia** e **IVistaAguja.***	
IVistaVia: Interfaz común para modelar las distintas vistas de los elementos de la maqueta que permite.	
Servicios	**Descripción**
`void dibujar( Graphics g );`	Método para dibujar cada vista del elemento de la maqueta dependiendo si está ocupado o no.
`void reset( );`	Método para mostrar que el tren ya no se encuentra sobre el elemento vía.
`void set(int x, int y);`	Método para establecer las coordenadas de la vista en la vía.
`void update( bool ocupado );`	Actualiza si el elemento de la maqueta está ocupado o no.
IVistaAguja: Interfaz común para modelar las vistas que posee una aguja	
Servicios	**Descripción**
`bool Estado{ set; }`	Propiedad de escritura del estado de cada aguja, es decir, la vista inicial
`void updateAguja( bool recto, int sentido, int updown );`	Actualiza si la aguja está recta o curva, si es ADerechas o AIzquierdas, y si girará hacia arriba o hacia abajo.

Capítulo 4.
Traducción del código.

4.1. Traducción automática.

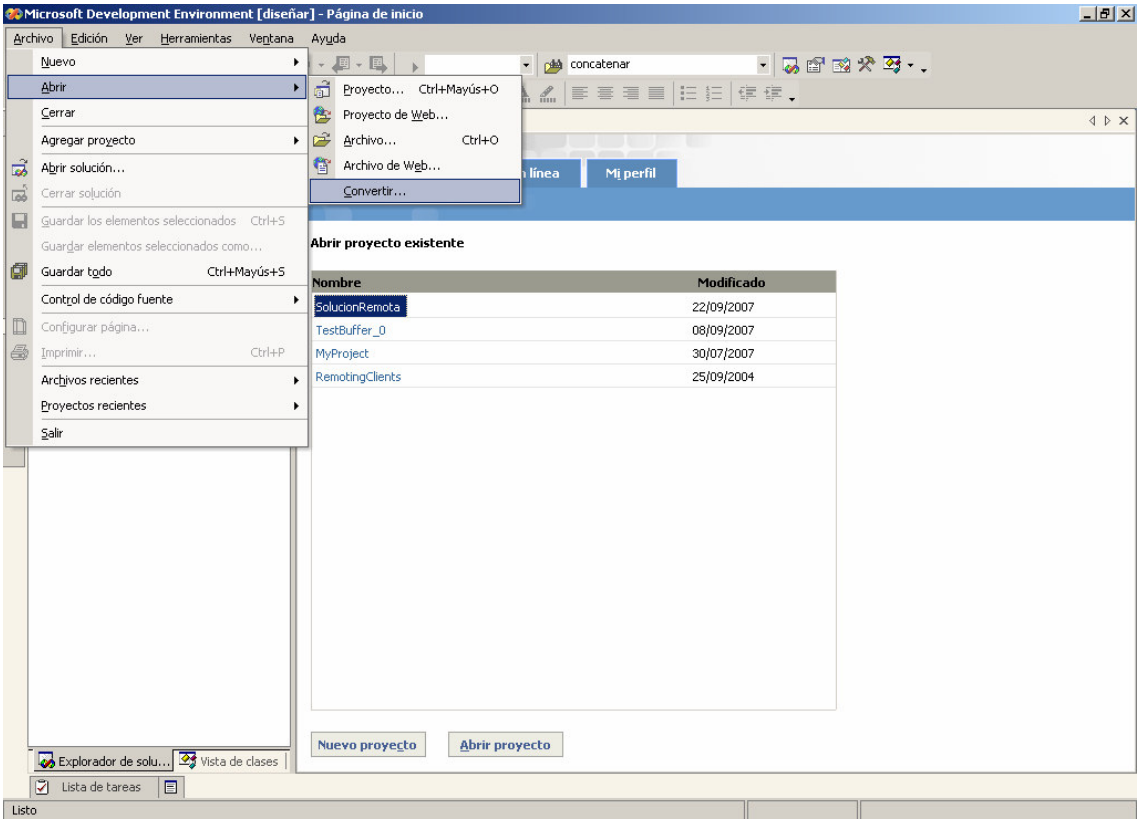
Uno de los objetivos de este proyecto es traducir a C# la aplicación Java de control de la maqueta de trenes. Para esto hemos partido de un proyecto [1]. Para obtener una primera visión global de la estructura del programa en el lenguaje C# hemos utilizado el recurso disponible por Microsoft Visual Studio .NET 2003, "Ayudante para la conversión del lenguaje Java", el resultado del uso de este recurso nos ofrece un boceto del proyecto en C# desde el cual partir hasta conseguir un programa compilable y ejecutable, siendo este el proceso más laborioso.

Para ello abrimos el programa **Microsoft Visual Studio .NET 2003**, que es el entorno de desarrollo que hemos utilizado. Como se muestra en la figura.



Arrancado Microsoft Visual Studio .NET 2003.

Una vez abierto el programa nos vamos a **`Archivo/Abrir/Convertir`**. Como se observa en la figura.



Accediendo a la herramienta "Ayudante para la conversión del lenguaje Java".

Al pulsar, nos aparece un cuadro que nos da a elegir el convertidor, de entre los disponibles que hay, el que lleva a cabo nuestro objetivo. En nuestro caso dejamos seleccionado *"Ayudante para la conversión del lenguaje Java"*, y pulsamos *Aceptar*.

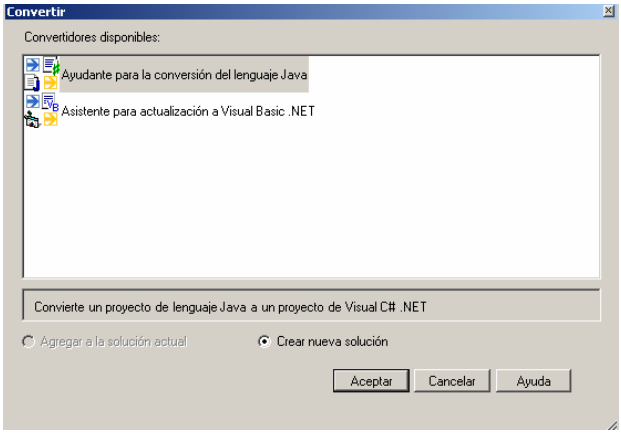
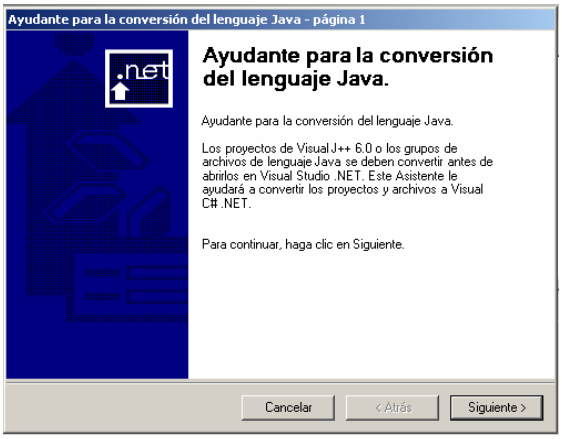


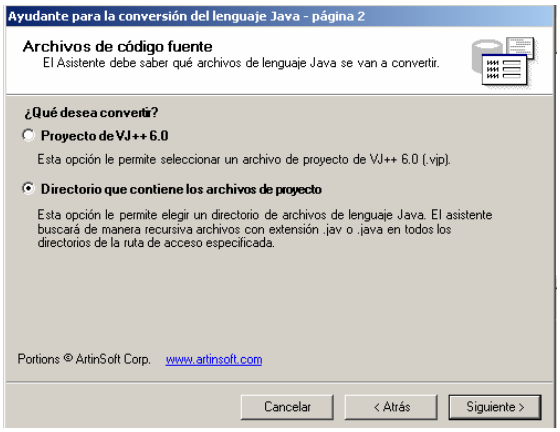
Figura 4.3. *Convertidores disponibles.*

Al pulsar **`Aceptar`** accedemos al inicio del **`"Ayudante para la conversión del lenguaje Java"`**.



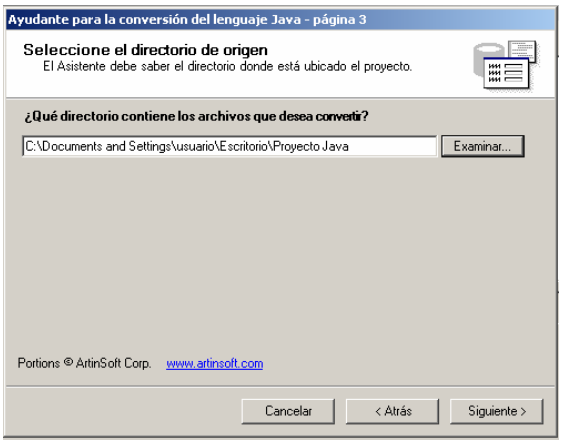
"Ayudante para la conversión del lenguaje Java" página 1.

Al pulsar **`Siguiente`**, le vamos a indicar un **`"Directorio que contiene los archivos de proyecto"`**, que vamos a convertir.



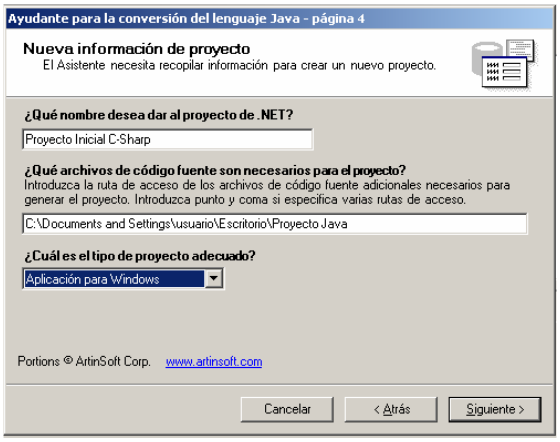
"Ayudante para la conversión del lenguaje Java" página 2.

Al pulsar **`Siguiente`**, le indicamos el directorio que contiene los archivos a convertir.



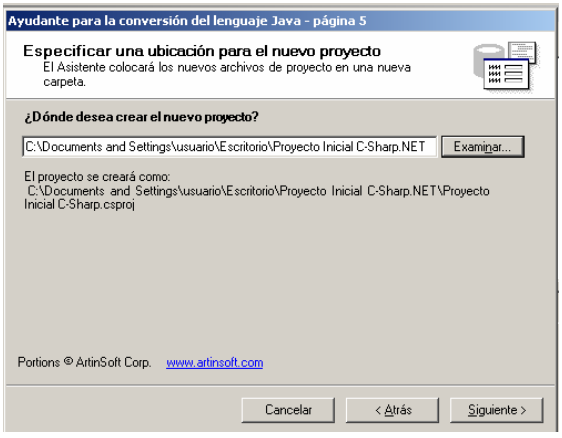
"Ayudante para la conversión del lenguaje Java" página 3

Al pulsar **`Siguiente`**, le indicamos el nombre que vamos a dar al proyecto de .NET, los archivos de código fuente que son necesarios para el proyecto y el tipo de proyecto que en nuestro caso indicamos **`Aplicación para Windows`**.



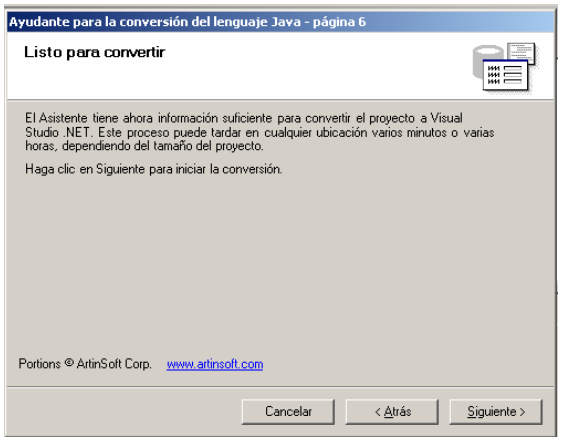
"Ayudante para la conversión del lenguaje Java" página 4.

Al pulsar **`Siguiente`**, le indicamos la ruta donde se va a crear el nuevo proyecto.



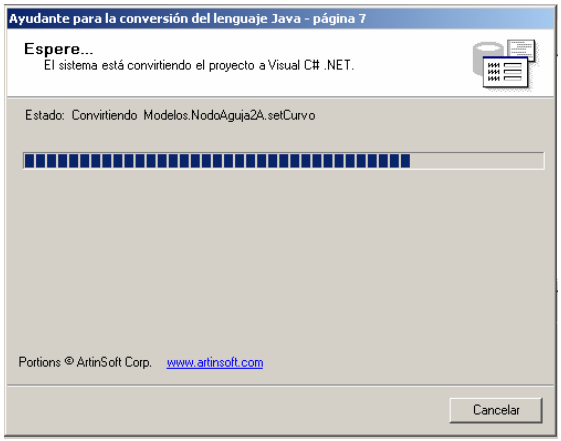
"Ayudante para la conversión del lenguaje Java" página 5.

Al pulsar **Siguiente**, el asistente nos informa de que tiene información suficiente para convertir el proyecto a Visual Studio .NET.



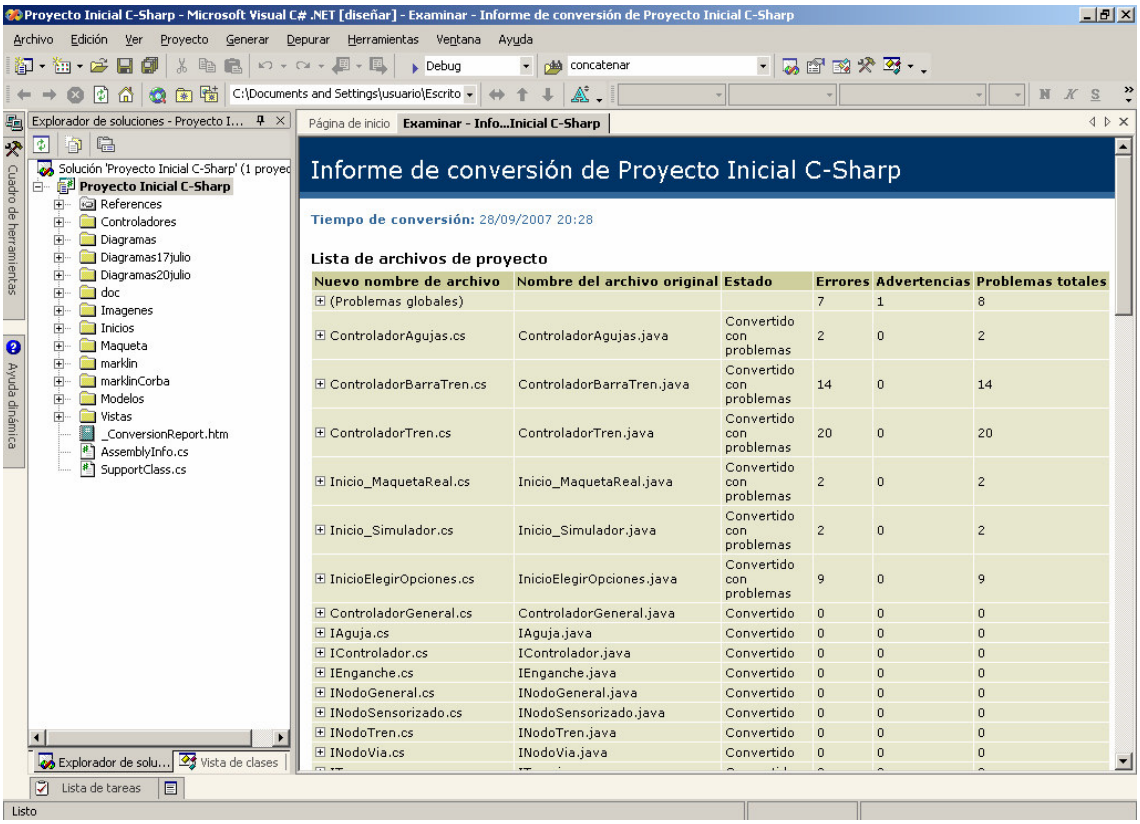
"Ayudante para la conversión del lenguaje Java" página 6.

Al pulsar **Siguiente**, se inicia el proceso de conversión y se muestra mediante la barra de estado activa.



"Ayudante para la conversión del lenguaje Java" página 7.

Al finalizar el proceso, disponemos del proyecto en C# (un proyecto no compilable) en el entorno de desarrollo. También, lo primero que nos muestra es el informe de conversión del proyecto en el cual, como se observa, una lista de archivos del proyecto, y el estado de la conversión. Y es aquí dónde empieza la adaptación manual del código.

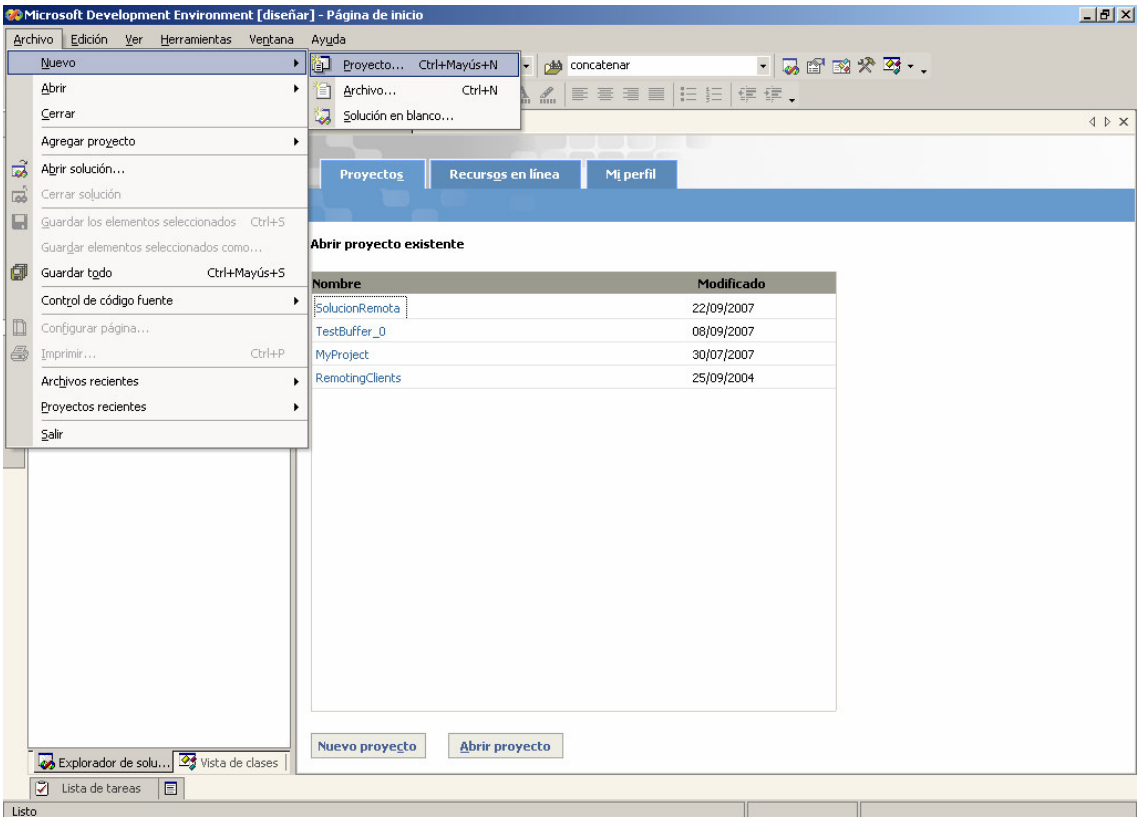


Informe de conversión resultante.

4.2. Adaptación manual del código.

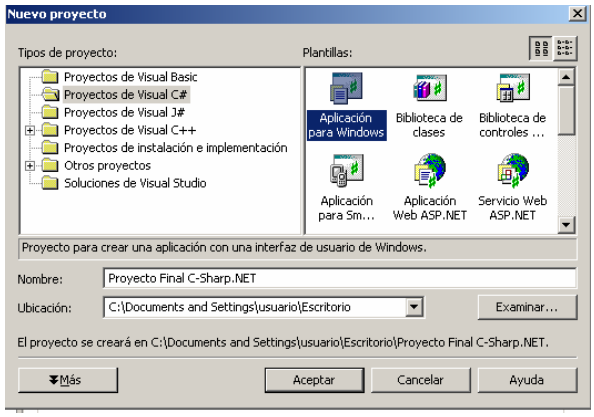
Volvemos a abrir el programa **Microsoft Visual Studio .NET 2003**. Como se mostró anteriormente.

Una vez abierto el programa nos vamos a **`Archivo/Nuevo/Proyecto`**. Como se observa en la figura.



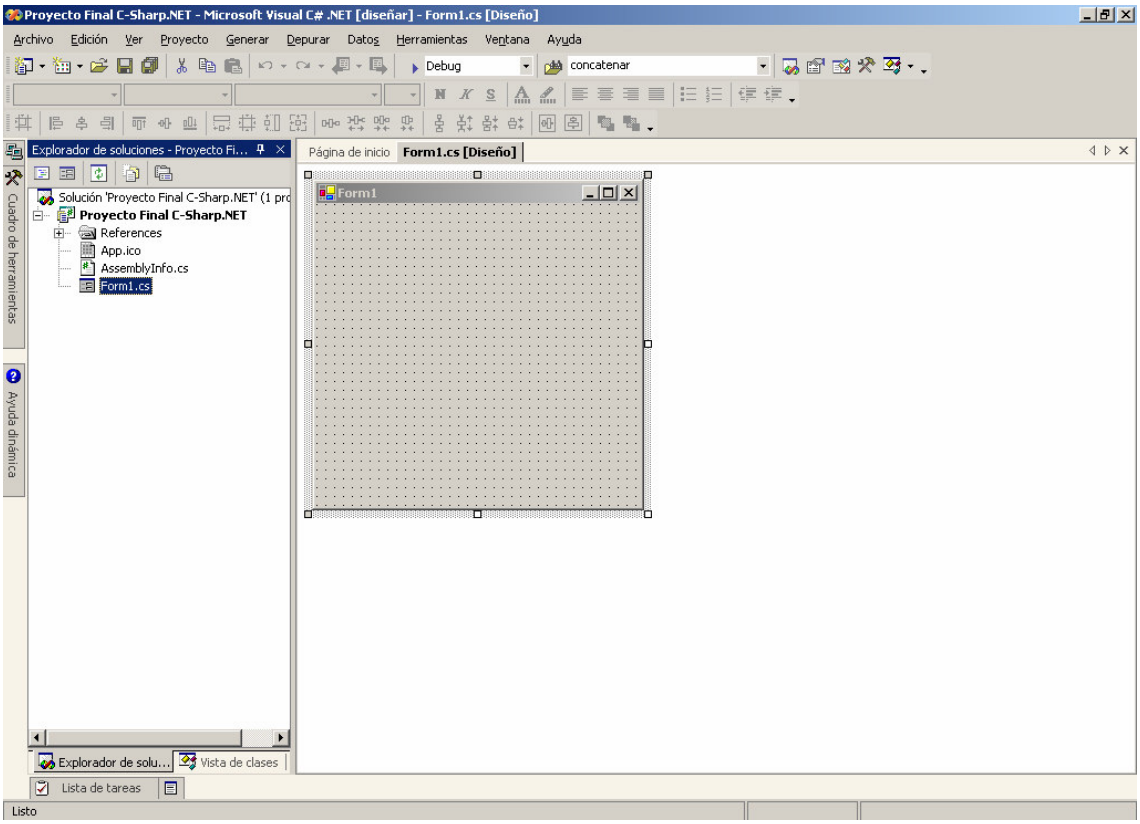
Creando un nuevo proyecto.

Al pulsar, nos aparece un cuadro que nos da a elegir el tipo de proyecto, **"Proyectos de Visual C#"**, el tipo de plantilla, "**Aplicación para Windows**" y nombre y ubicación del proyecto que vamos a crear.



Creando un nuevo proyecto.

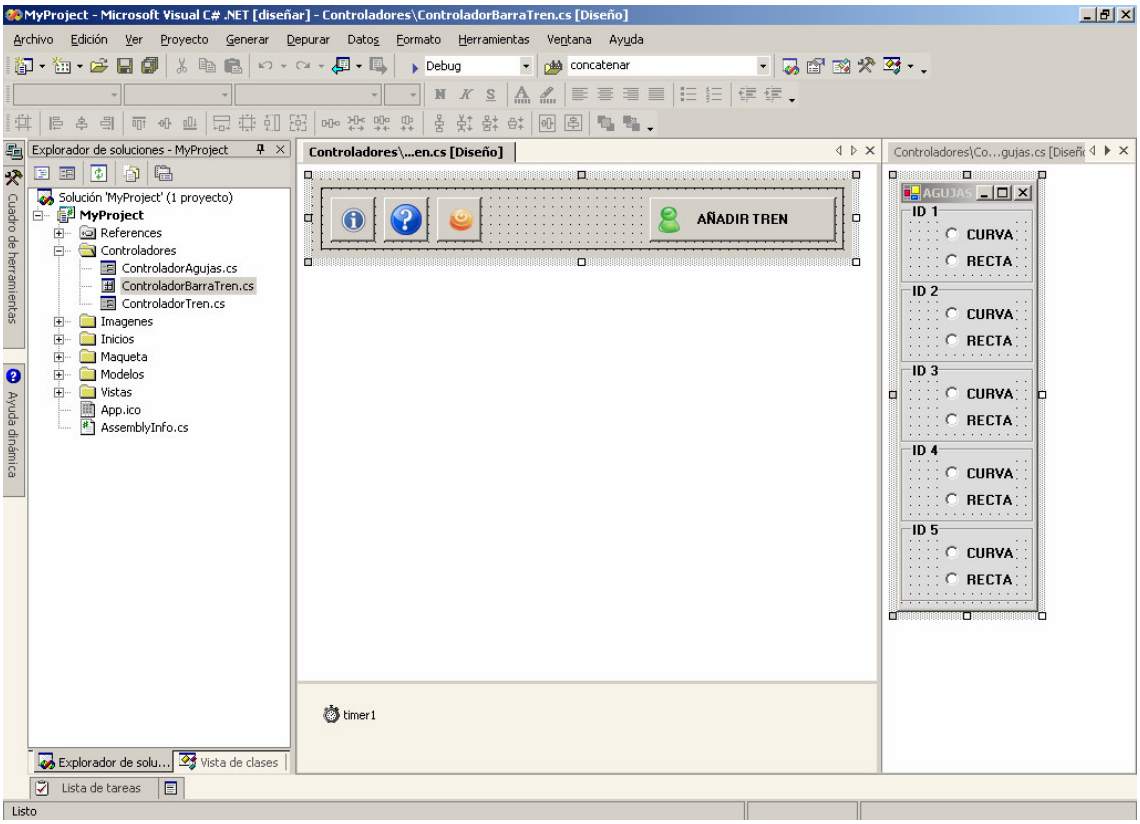
Al pulsar **"Aceptar"**, ya tenemos el punto de partida del proyecto.



Creando un nuevo proyecto.

A este proyecto le añadiremos las clases revisadas que podemos reutilizar del proyecto traducido. La estructura de espacios de nombres, clases, interfaces y respectivas variables que vamos a seguir es la misma que la del proyecto Java.

Como ya se comentó en el capítulo 2, la primera gran diferencia con la que nos enfrentamos es con la creación de la interfaz gráfica, por eso hemos utilizado el Diseñador de Windows Forms, mediante el cual seleccionamos de forma gráfica los componentes que integran la aplicación. Automáticamente disponemos del código generado por la acción anterior. A partir de este código añadiremos funcionalidad a los componentes. Este es el método que hemos utilizado para desarrollar el programa, como se observa en la figura.



Creando la interfaz gráfica.

La adaptación manual del código de todas las clases lo hacemos a partir del informe de conversión. De aquí obtenemos información del resultado de las clases convertidas de las cuales, como se observa, existe una gran parte que presentan problemas, o sea, que no se consiguió una adaptabilidad total del código Java al código C#. Como ejemplo, en la siguiente figura mostramos el resultado de una clase convertida con problemas de la cual se observan los diferentes tipos de errores asociados. El tipo de errores viene asociado con las bibliotecas de java, que no son existen en .NET Framework, siendo esta la forma de actuar, trasladando llamadas a funciones utilizadas en Java a funciones de .NET Framework que realizan la misma función.

⊟ ControladorTren.cs	ControladorTren.java	Convertido con problemas	20	0	20

Problemas de conversión de Controladores.ControladorTren.ControladorTren (Controladores.ControladorBarraTren,java.lang.String,int,Modelos.TrenSimulado_0,Modelos.TrenSimulado_0):

#	Tipo	Gravedad	Descripción
1	Error de compilación	1	No se convirtió Method 'java.awt.Container.setLayout'.
2	Error de compilación	1	No se convirtió Constructor 'java.awt.BorderLayout.BorderLayout'.
3	Error de compilación	1	No se convirtió Constructor 'java.awt.BorderLayout.BorderLayout'.
4	Error de compilación	1	No se convirtió Field 'java.awt.BorderLayout.NORTH'.
5	Error de compilación	1	No se convirtió Constructor 'java.awt.BorderLayout.BorderLayout'.
6	Error de compilación	1	No se convirtió Field 'java.awt.BorderLayout.NORTH'.
7	Error de compilación	1	No se convirtió Field 'java.awt.BorderLayout.SOUTH'.
8	Error de compilación	1	No se convirtió Field 'java.awt.BorderLayout.CENTER'.
9	Error de compilación	1	No se convirtió Field 'java.awt.BorderLayout.SOUTH'.
10	Error de compilación	1	No se convirtió Field 'java.awt.BorderLayout.CENTER'.

Problemas de conversión de Controladores.ControladorTren.ControladorTren (Controladores.ControladorBarraTren,java.lang.String,int,Modelos.TrenSimulado_0,Modelos.TrenSimulado_0,marklin.TrenM

#	Tipo	Gravedad	Descripción
1	Error de compilación	1	No se convirtió Method 'java.awt.Container.setLayout'.
2	Error de compilación	1	No se convirtió Constructor 'java.awt.BorderLayout.BorderLayout'.
3	Error de compilación	1	No se convirtió Constructor 'java.awt.BorderLayout.BorderLayout'.
4	Error de compilación	1	No se convirtió Field 'java.awt.BorderLayout.NORTH'.

Errores de conversión de una clase cualquiera.

Capítulo 5.
Código resultante.

5.1. Paquetes y librerías.

5.1.1. Introducción.

La motivación de este apartado es la presentación de los diferentes paquetes que conforman el software desarrollado en este Proyecto y su porqué. Surge de la necesidad de realizar un control de la estructura general que adopta el programa. Esto permite tener información de relevancia y global acerca del modelado aunque no tanto de la implementación.

5.1.2. Diferentes paquetes desarrollados.

En la siguiente figura se ilustran los accesos y dependencias entre los paquetes que componen la parte **"Modo Simulador"** de este Proyecto.

Cada paquete proporciona un mecanismo de agrupación para organizar las clases e interfaces de la aplicación, además de proporcionar un espacio de nombrado sobre cada parte en la que está dividido el Proyecto. Al querer dar con la mayor claridad y elegancia posible el cumplimiento de características propias del patrón MVC los paquetes en los que se organiza el código tienden a reflejar la estructura de dicho patrón.

Dichos paquetes son los siguientes:

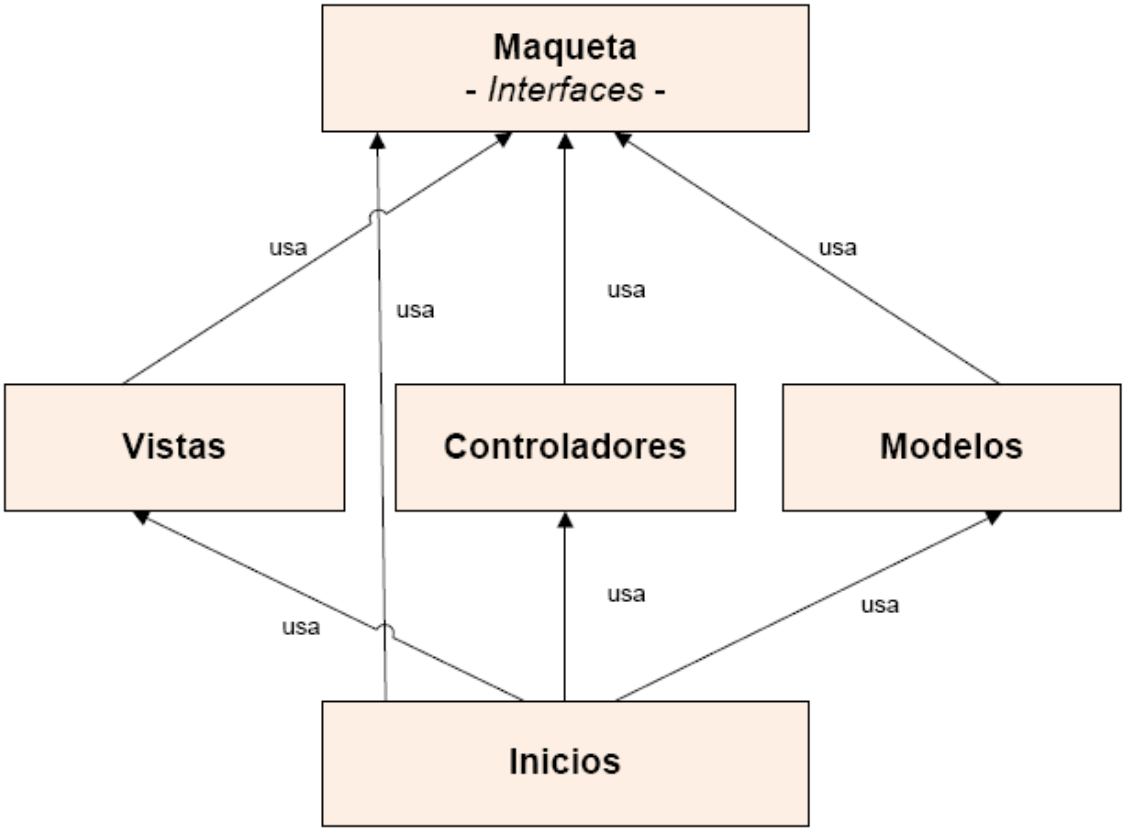
- ✓ Paquete **Maqueta**: contiene las interfaces y clases abstractas de la aplicación.

- ✓ Paquete **Inicios**: contiene las aplicaciones principales que modelan el simulador y la maqueta del laboratorio.

- ✓ Paquete **Vistas**: contiene las distintas vistas de los elementos de una vía, además de las interfaces gráficas que se muestran al usuario.

✓ Paquete **`Controladores`**: contiene los distintos controladores en los que el usuario podrá interactuar con el simulador o con la maqueta real.

✓ Paquete **`Modelos`**: contiene la funcionalidad de la aplicación, es decir, la implementación de cada uno de los tipos de elementos, así como el tratamiento de excepciones.



Representación de los paquetes en "Modo Simulador".

Se ha definido un paquete **`Maqueta`** para favorecer la ocultación de la implementación lo máximo posible. En **`Maqueta`** se encuentran todas las interfaces del Proyecto en este modo. De esta manera, el resto de paquetes lo usan para implementar la funcionalidad que mejor les convenga para desempeñar sus funciones sin que se establezcan dependencias mutuas entre implementaciones.

La biblioteca de clases para el iniciar el simulador, denominada **`Inicios`**, tiene una correspondencia directa con el resto de bibliotecas de clases, tanto para el control de la maqueta, como para la interacción del usuario, como para la visualización del estado de la maqueta. Éste paquete utiliza a todos los demás, debido a que además de contener el código de inicialización del simulador, contiene las clases que modelan la maqueta global creando

sus componentes según el patrón MVC. Por último, crea también un controlador y le pasa una referencia del modelo y de la vista. Este último caso es una variante usada en este Proyecto del patrón, debido a que puede implementarse de diferentes formas, dependiendo de cómo mejor convenga al desarrollador, y siempre y cuando, sea efectivo. Se puede decir que este paquete constituye la aplicación principal, donde se ejecuta y crea los vínculos correspondientes con el resto de paquetes para gestionar las funciones del patrón MVC.

El paquete **Modelos**, tal y como su nombre referencia, se corresponde con la funcionalidad que presenta cada modelo. Está compuesto por diferentes clases para poder hacer posible llevar a cabo la funcionalidad de esta aplicación, registrar las vistas y controladores dependientes, y notificar a los componentes dependientes sobre los cambios de datos. Se intenta establecer lo más independiente posible, por lo que no tiene referencia directa alguna sobre los paquetes **Controladores** y **Vistas**, que contienen implementaciones, sino con **Maqueta** que contiene las interfaces que implementan las clases del paquete. La idea es depender de interfaces, pero no de implementaciones.

Por otra parte el paquete **Vistas**, es aquel que incluye todas las vistas de cada uno de los elementos de los que está formada la maqueta de trenes del laboratorio, además de proporcionarnos una interfaz gráfica intuitiva para monitorizar el estado de cada tren, sensor, y aguja sobre la maqueta. Además, está formada por clases que a parte de realizar lo anteriormente reflejado, implementan el paquete **Maqueta** realizando el procedimiento de actualización y recuperación de los datos procedentes del modelo, según el patrón MVC. Este paquete tiene referencias a las interfaces de **Maqueta**, para la creación de los elementos de la vía férrea de la maqueta, sin necesidad de saber qué elementos de **Modelos** tendrán relación con este paquete.

En última instancia, queda el paquete **Controladores**, el cual gestiona las entradas del usuario para acceder a la funcionalidad del modelo. Sus clases reflejan los controladores de los distintos elementos de la maqueta, con los que se puede interactuar, tales como, un controlador de agujas para cada una de ellas, un controlador general para incorporar trenes al simulador de la maqueta, y uno también para cada uno de los trenes, para interactuar con la velocidad. Con respecto a la relación que hay entre este paquete y el paquete **Modelos** es que reflejan una correspondencia directa unilateral por medio de interfaces, para poder informar a las clases del modelo, los cambios que se deben llevar a cabo. También usa el paquete **Maqueta** para implementar una interfaz de su tipo.

5.2. Descripción de la aplicación.

5.2.1. Introducción.

El objetivo de este apartado es realizar un estudio detallado sobre el modelado y la implementación de los componentes. Se pretende distinguir el código desde dos puntos de vista: una vista estática, que englobará la estructura de clases, representados mediante los diagramas de clases, y una vista dinámica, que representa el comportamiento de la aplicación en tiempo de ejecución, representado mediante los diagramas de secuencia UML. Desde el punto de vista del funcionamiento, deben distinguirse tres partes, totalmente diferenciadas pero que cooperan entre sí: el cliente, el servidor y la maqueta. El que haya tres partes no significa que deban ejecutarse todas en máquinas distintas. Todas las partes se podrán ejecutar en la misma máquina (si se activa el modo simulador o el modo maqueta), o cada parte podría ejecutarse en una máquina distinta (si se ejecuta la aplicación en modo remoto).

5.2.2. Visión estática del código.

Tal y como se ha explicado en el capítulo 3, la funcionalidad de cada uno de los componentes de la aplicación de este Proyecto se modela en una interfaz, proporcionándose además, para cada una de ellas, una o varias clases que implementan el comportamiento por defecto común de cada tipo de componente. A continuación se explicará la funcionalidad, e implementación para cada una de las clases implementadas en este Proyecto.

5.2.2.1. Bloques constructivos fundamentales. Los nodos.

En este Proyecto se necesita establecer una relación de avance y retroceso de los trenes sobre los nodos, por lo que la vía férrea se ha implementado mediante listas doblemente enlazadas o grafos, facilitando el cambio de topología y la adición/eliminación de elementos de la maqueta. Estos nodos se organizan, de modo que cada uno apunta al siguiente y al anterior nodo. El grafo resultante de enlazar cada uno de los tramos representa a la maqueta global.

La aguja se implementa como un tipo de nodo especial ya que en vez de tener un sucesor, puede tener dos posibles sucesores o anteriores, dependiendo del tipo de aguja que sea.

Los grafos son un clásico de la teoría de la computación. La organización de este tipo de bloques constructivos mediante listas o grafos permite ampliar el Proyecto fácilmente, debido a la gran cantidad de algoritmia relacionada con el manejo de datos que existe en la ciencia de la computación.

Tal y como se mencionó en el capítulo 3, existe una interfaz que representa a cada uno de los nodos, definiendo la funcionalidad común que tiene cada uno de ellos. El significado de los métodos de las clases que implementan de sus interfaces genéricas y particulares está explicado en el capítulo 3. La siguiente figura muestra las clases e interfaces de los nodos.

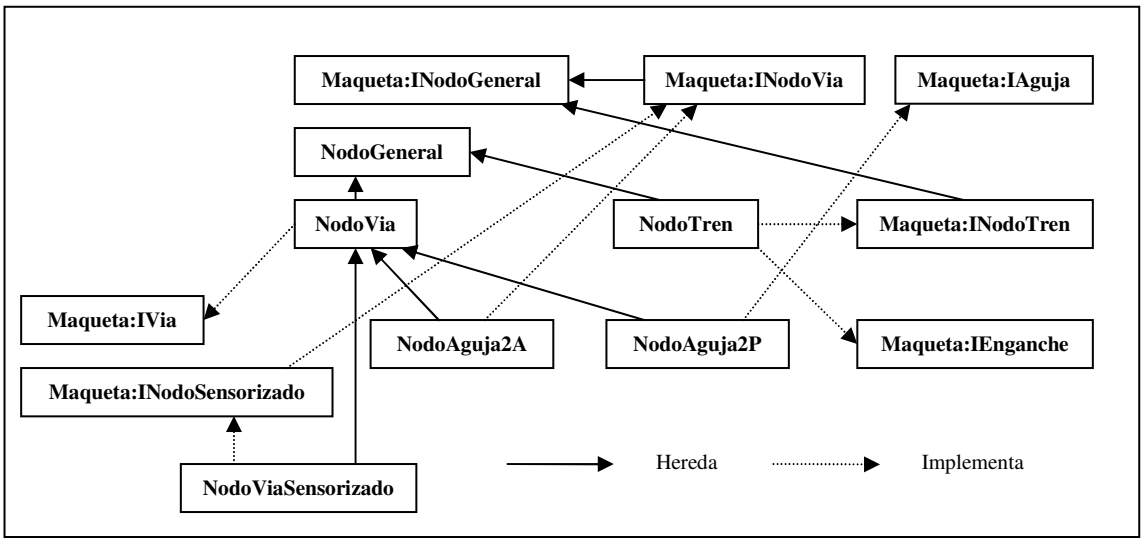
Obsérvese que existe una clase **`NodoGeneral`** que implementa de **`INodoGeneral`**. Éstos definen la funcionalidad común que tienen los nodos, sea cual sea cada uno de ellos. En este caso será la implementación de una lista doblemente enlazada, por lo que de ahí vienen sus métodos.

Las clases de los nodos de vía son **`NodoVia`**, **`NodoAguja2A`**, **`NodoAguja2P`** y **`NodoViaSensorizado`**. Éstos son los elementos de la vía férrea que componen la maqueta. Cabe destacar que tanto las clases **`NodoViaSensorizado`** como **`NodoAguja2A`** y **`NodoAguja2P`** heredan directamente de **`NodoVia`**.

La clase **`NodoVia`** permite modelar la posición de un tren sobre la vía de la maqueta. Además de heredar la funcionalidad de una lista, de **`NodoGeneral`**, e implementar la interfaz **`IVia`**, recogiendo la funcionalidad básica de todo **`NodoVia`**, añade las siguientes funcionalidades: indicar si hay un tren colocado sobre la maqueta, incorporarlo y eliminarlo de ella y notificar a los observadores de la vía los cambios que se hayan producido. Esta funcionalidad se suele utilizar para un simulador de una maqueta de tren.

Un **`NodoViaSensorizado`** adquiere la funcionalidad recogida en **`NodoVia`**, al heredar de ella, e implementa la funcionalidad básica de un sensor, de la interfaz **`NodoSensorizado`**, tal como activar y desactivar un sensor, comprobar su estado, etc.

Como tipo de nodo especial se encuentran las agujas. Se definen dos clases para su implementación: **`NodoAguja2A`** (cuando la aguja tiene dos nodos anteriores) y **`NodoAguja2P`** (para cuando tiene dos nodos posteriores). Al formar parte también de una vía, adquiere la funcionalidad de **`NodoVia`**, y por consiguiente, hereda de esta clase.



Clases e Interfaces de los nodos.

Destacar también la clase **`NodoTren`**, utilizada para ayudar a los objetos de las clases de los elementos de la vía férrea, proporcionándoles una serie de métodos para el modelado de la posición del tren sobre la maqueta del simulador, tales como indicar y fijar el nodo de la maqueta en el que se encuentra el tren en un momento determinado. Notar que hereda de la clase **`NodoGeneral`** para gestionar la funcionalidad indicada en el siguiente apartado, **`Tren`**.

En la tabla se muestra la información correspondiente a cada una de las clases que conforman este bloque. Únicamente se da una descripción a los métodos que no han sido explicados en las tablas de las interfaces que se encuentran en el capítulo 3.

Resumen de las clases de los nodos.	
`NodoVia:` Clase para modelar los elementos de los tramos de vía de una maqueta.	
Servicios	**Descripción**
`void actualizarObservadores( )`	Actualiza la lista de observadores de un nodo de una vía, indicando si está ocupado.
`boolean hayNodoTren( )`	Indica si hay un tren ya colocado sobre el nodo
`void ponerNodoTren(INodoTren nt)`	Establece el nodo en el que se encuentra el tren como ocupado y actualiza indirectamente la vista del simulador
`void quitarNodoTren( )`	Establece el nodo que va a abandonar el tren como no ocupado, actualizando indirectamente la vista del simulador
`NodoViaSensorizado:` Clase para modelar los elementos de los tramos de vía que están sensorizados.	
Servicios	**Descripción**
`void actualizarObservadores( )`	Actualiza su lista de observadores indicando si está ocupado.
`void ponerNodoTren(INodoTren nt)`	Establece el nodo en el que se encuentra el tren como ocupado, activa el sensor y actualiza indirectamente la vista del simulador.

void quitarNodoTren( )	Establece el nodo que va a abandonar el tren como no ocupado, desactiva el sensor y actualiza indirectamente la vista del simulador.
NodoAguja2A: Clase para modelar los elementos de las agujas con dos nodos anteriores	
Servicios	**Descripción**
void actualizarObservadores( )	Actualiza su lista de observadores indicando si está ocupada.
void actualizarObservadores2( )	Actualiza su lista de observadores indicando su sentido y su forma.
void ponerNodoTren(INodoTren nt)	Establece la aguja en el que se encuentra el tren como ocupado y actualiza indirectamente la vista del simulador.
void quitarNodoTren( )	Establece la aguja que va a abandonar el tren como no ocupado y actualiza indirectamente la vista del simulador.
NodoAguja2P: Clase para modelar los elementos de las agujas con dos nodos posteriores.	
Servicios	**Descripción**
void actualizarObservadores( )	Actualiza su lista de observadores indicando si está ocupada.
void actualizarObservadores2( )	Actualiza su lista de observadores indicando su sentido y su forma.
void ponerNodoTren(INodoTren nt)	Establece la aguja en el que se encuentra el tren como ocupado y actualiza indirectamente la vista del simulador.
void quitarNodoTren( )	Establece la aguja que va a abandonar el tren como no ocupado y actualiza indirectamente la vista del simulador.

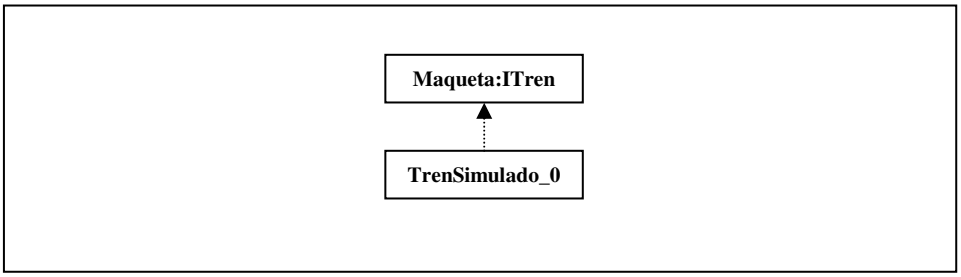
La idea general al desarrollar este bloque ha sido establecer una serie de interfaces para ocultar la implementación desarrollada y favorecer la extensibilidad de este Proyecto, para el caso de que se quiera utilizar otra técnica que no sean listas enlazadas o grafos. Para ello existen interfaces comunes y otras más específicas de cada tipo de nodo en concreto. Los nodos se han desarrollado bajo una relación de herencia acorde con la estructuración de los elementos que componen una maqueta de trenes.

5.2.2.2. El tren.

El comportamiento de un tren está definido en la interfaz **`ITren`**. Es necesario definir esta interfaz ya que el tren puede ser un tren real o un tren simulado, cambiando la implementación por completo.

La clase que representa el tren simulado, **`TrenSimulado_0`**, sí que implementa esta interfaz. Por lo tanto sí que realiza el comportamiento de un tren en el **"Modo Simulador"**.

En la figura se muestra la interfaz y las clases que se han utilizado para definir el comportamiento de un tren.



Clases e Interfaces de los trenes.

En la tabla se recoge un resumen de la funcionalidad de cada uno de los métodos no explicados en el capítulo 3, en el apartado de interfaces.

Resumen de clases de los trenes.	
`TrenSimulado_0`: Clase que define el comportamiento de un tren simulado	
Servicios	**Descripción**
`void actualizarObservadores( )`	Actualiza la lista de observadores de un tren, útil para el paso de mensajes específicos que se actualizará en la vista.
`void ponerTren(int nroTrenes, AreaDibujo area)`	Permite colocar el tren en un nodo de la maqueta.
`String Mensajes{set;}`	Establece un mensaje que se propagará hasta la vista principal para mostrarlo al usuario.
`int NumTrenes{set;}`	Establece el número de trenes que se propagará hasta la vista principal para mostrarlo al usuario.
`void tick( )`	Permite realizar una determinada acción sobre el tren, en cada instante de tiempo en el que se ejecute, como avanzar, retroceder, parar…

Dos de los métodos más importantes que definen el comportamiento de un tren, y por lo tanto, los implementan, son **`avanzar()`** y **`retroceder()`**. Se realiza un estudio detallado sobre la implementación de cada uno de ellos.

✓ Método `avanzar()`.

El código desarrollado en este método es el siguiente:

```
public virtual void  avanzar()
{
        NodoTren indice = (NodoTren) primero;

        while (indice != null){
                indice.Nodo.quitarNodoTren();

                try
                {
                        NodoVia nv = (NodoVia) indice.Nodo.siguiente();
                        nv.ponerNodoTren(indice);
                }
                catch(ColisionTrenesException e)
                {
                        Console.Out.WriteLine("Colisión");
                        parar();
                        MessageBox.Show(
                                "Se ha producido una colisión entre trenes.\n" + "Todos
                                los trenes implicados se detienen hasta que llegue la
                                Policía y \nAmbulancia paraestimar" + " los daños y
                                atender a las personasheridas",
                                "COLISIÓN DE TRENES",
                                MessageBoxButtons.OK,
                                MessageBoxIcon.Exclamation);
                }

                indice.Nodo = (NodoVia) (indice.Nodo.siguiente());

                indice = (NodoTren) indice.siguiente();

        }
}
```

La explicación de esta implementación es la siguiente:

- Hay un nodo de referencia que define la posición actual del tren en la vía y a partir del cual se avanza o retrocede. Este nodo, llamado **`indice`**, inicialmente es el primer nodo donde se coloca el tren en la maqueta.

- Al avanzar, y por tanto, cambiar de un nodo a su siguiente, se le quita la asociación directa que tiene con el nodo vía en el que se encuentra, actualizando su vista para que se refleje que ya ha abandonado ese nodo de la vía.

- Posteriormente se necesita obtener una referencia del nodo de la vía que está concatenado con el que estaba antes, que será el que se encuentra inmediatamente después. Los métodos de la clase **`NodoGeneral`** permiten realizar esto.

- Al obtener esta referencia, el nodo del tren se coloca en ese mismo nodo de la vía y se actualiza la vista para reflejar que el tren se ha colocado en dicho nodo. En el caso de que se intente ocupar un nodo por dos trenes al mismo tiempo, se produce una excepción, que se trata parando los trenes.

- Para continuar con el ciclo que permite ir pasando de un nodo de vía a otro nodo de vía colocando el tren, se obtiene una referencia del nodo siguiente al actual, donde se encuentra el tren, y se le establece al nodo utilizado en la condición (al nodo de referencia). Por lo tanto, de esta manera, siempre se van obteniendo referencia al nodo siguiente, hasta que no haya ninguno, y por lo tanto, el tren no pueda avanzar más.

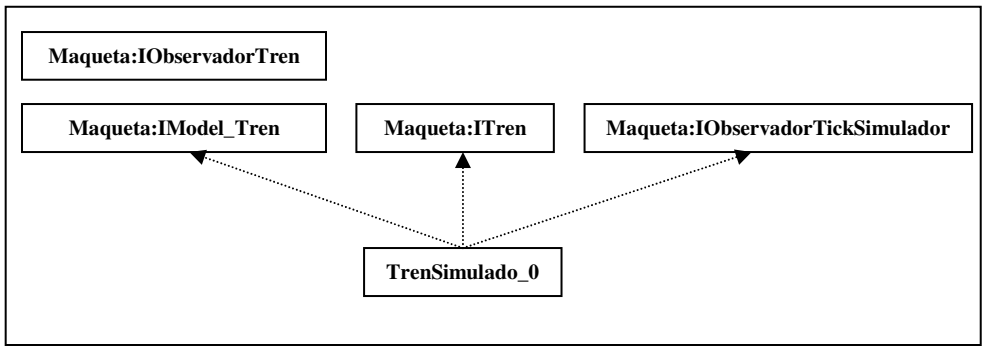
✓ Método `retroceder()`.

Se expone a continuación el código desarrollado en este método:

```
public virtual void  retroceder()
{
       NodoTren indice = (NodoTren) ultimo;

       while (indice != null)
       {
              indice.Nodo.quitarNodoTren();

              try
              {
                     NodoVia nv = (NodoVia) indice.Nodo.anterior();
                     nv.ponerNodoTren(indice);
              }
              catch (ColisionTrenesException e)
              {
                     Console.Out.WriteLine("Colisión");
                     parar();
                     MessageBox.Show(
                              "Se ha producido una colisión entre trenes.\n" + "Todos
                              lostrenes implicados se detienen hasta que llegue la
                              Policía y\nAmbulancia para estimar" + " los daños y
                              atender a las personas heridas",
                            "COLISIÓN DE TRENES",
                            MessageBoxButtons.OK,
                            MessageBoxIcon.Exclamation);
              }

              indice.Nodo = (NodoVia) (indice.Nodo.anterior());

               indice = (NodoTren) indice.anterior();
       }
}
```

El procedimiento implementado en este método, es igual que el que se ha explicado para el método **`avanzar()`**, con la salvedad de que ahora el nodo de referencia no tiene que saber cual es su nodo siguiente, sino su nodo anterior, provocando así que cada nodo enlace con su nodo anterior concatenado, haciendo que el tren pueda retroceder satisfactoriamente.

En la figura se puede observar con detalle cómo se ha organizado la estructura de los trenes.



Estructura general para el comportamiento de un tren.

Para el uso de la aplicación, como se ha dicho antes, tanto en un modo como en otro se va a realizar una simulación. Para ello la clase **`TrenSimulado_0`** implementa la interfaz **`IObservadorTickSimulador`**, la cual presenta el método **`tick()`**, que realizará las funciones de un tren según el temporizador. Esta interfaz e implementación es el motor, el corazón del simulador. Cada 50 milisegundos se gestionarán todas las actualizaciones mediante el uso de la clase **`Timer`**.

A su vez, la clase **`TrenSimulado_0`** implementa la interfaz **`IModel_Tren`**. Esta interfaz es específica del tren del simulador y no se han incorporado las funcionalidades a la interfaz **`ITren`**, por el hecho de obtener una mayor especificidad para cada tipo de tren. Permite establecer los mensajes y el número de trenes que se vayan a actualizar para ser reflejados en la vista principal, de cara al usuario, implementando la interfaz **`IObservadorTren`**. Además, en **`IModel_Tren`** se pueden añadir funcionalidades, en función de las características que vaya a tener el tren en concreto. Es decir, está condicionado y estructurado para lograr una mantenibilidad y adaptabilidad a nuevos servicios o tendencias de la aplicación.

En la tabla se presenta de forma resumida la funcionalidad de las interfaces **`IModel_Tren`**, **`IObservadorTickSimulador`**, y **`IObservadorTren`**.

Interfaces utilizadas para definir el comportamiento de un tren.	
`IModel_Tren`: Interfaz común para modelar los objetos de un tren simulado, definiendo sus funcionalidades.	
Servicios	**Descripción**
`String Mensajes{ set; }`	Propiedad para establecer un mensaje que se visualizará al usuario.
`int NumTrenes{ set; }`	Propiedad para establecer el número de trenes que hay sobre la maqueta.
`IObservadorTickSimulador`: Interfaz común para modelar los objetos de un tren simulado, definiendo lo que será el motor o corazón del simulador, cada ciertos ciclos periódicos de reloj para gestionar los eventos.	
Servicios	**Descripción**

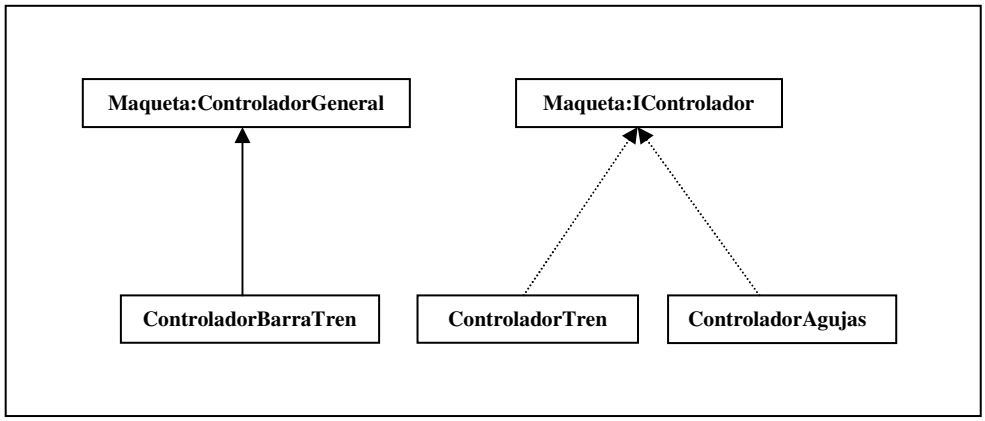
`void tick ( );`	Método que permite realizar una determinada acción sobre el tren simulado (avanzar, retroceder, etc.) en cada tick de reloj.
`IObservadorTren:` Interfaz común para modelar los objetos de un tren simulado, para actualizar tanto los mensajes que se quieran mostrar al usuario, como el número de trenes que hay en la maqueta.	
Servicios	**Descripción**
`void actualizar (int nroTrenes, String mensaje);`	Método que permite actualizar, según MVC, tanto el número de trenes que hay sobre la maqueta como los mensajes a mostrar al usuario.

5.2.2.3. Los controladores.

Los controladores se encargan de aceptar entradas del usuario para acceder a la funcionalidad del modelo, es decir, proporciona la funcionalidad (servicios) de la aplicación.

Se ha definido una interfaz común para los controladores de las agujas, **`ControladorAgujas`**, y de los trenes, **`ControladorTren`**, llamada **`IControlador`**, y una clase, **`ControladorGeneral`**, para poder utilizar el patrón MVC, recibiendo una referencia del tren del modelo y de la vista que lo llame. Destacar también que la funcionalidad del método abstracto que presenta esta clase, la implementa su clase hija, **`ControladorBarraTren`**, añadiendo además otras implementaciones, tal y como se describe durante este apartado.

En la figura se presentan las clases y la interfaz anteriormente mencionadas, junto con su organización general.



Interfaces y clases de los controladores.

Básicamente, la clase **`ControladorBarraTren`** se desarrolla creando una interfaz gráfica y manejando los eventos de los botones creados. De esta manera el usuario puede

interactuar cada vez que quiera añadir un tren a la maqueta pulsando el botón **"Añadir Tren"**. Además se crea una barra de botones en la parte superior de la aplicación principal, con funcionalidades de ayuda, **"acerca de"** y "**salir**". Destacar también la existencia de dos constructores, que se diferencian en su lista de parámetros por la referencia de la clase **TrenMaqueta_0**. Esta referencia es utilizada para el caso **"Modo Maqueta"**, en el que además de arrancar el simulador, permite invocar operaciones a la maqueta de trenes real, tal como avanzar y parar un tren. Además, en esta clase, para cada evento proporcionado por un objeto de la clase **Timer**, comprueba el estado de todos trenes y mediante el método **tick()** se realizan las funciones que tiene pendientes cada tren, además de actualizar las vistas.

Por otra parte, la clase **ControladorTren** genera la interfaz gráfica de los controladores de un tren, manejando los eventos de los botones **"Velocidad +"**, **"Velocidad –"**, y **"Parar"**. Se ha desarrollado para que el usuario pueda interactuar con el simulador y la maqueta independientemente para cada tren. Para ello se crea un **UserControl** para cada tren, representando un controlador de tren para cada uno de ellos, manejando su funcionalidad independientemente. Cabe destacar también la existencia de dos constructores, al igual que en el caso anterior, para diferenciar, si al pulsar uno de los botones del controlador, va a simular solamente, o por el contrario, va a simular y ejecutar en la maqueta del laboratorio. Al manejar los eventos de los botones se comprueba si existe una referencia de **TrenMaqueta_0** recibida en el constructor apropiado. Si no hay, se simula directamente, y si por el contrario, se ha pasado una referencia, se simula y se ejecuta la operación correspondiente.

Por último, destacar la clase **ControladorAgujas**. Se encarga de aceptar entradas de usuario, el cual elige el tipo de aguja y su estado (curvo o recto), para posteriormente acceder a la funcionalidad del modelo de la aguja. Se crea un **UserControl** que aparece directamente al arrancar la aplicación, junto con la vista principal. Éste **UserControl** está dividido en cinco grupos de botones. Cada uno de ellos representa a cada una de las agujas, y dentro de cada grupo existen dos botones específicos de cada aguja, los cuales determinan el estado en el que se encontrará dicha aguja.

En la tabla, se hace un resumen de la funcionalidad que adquiere cada una de las clases anteriormente descritas.

Resumen de las clases de los controladores.	
ControladorGeneral: Clase general controladora de un tren. Se suscribe al modelo del tren	
Servicios	**Descripción**
ControladorGeneral(TrenSimulado_0 tren, VistaGeneral v1)	Constructor que recibe una referencia del modelo y de la vista

	que lo llame, y se suscribe el controlador al modelo
`void actualizar(int nro, String mens);`	Método para actualizar, según MVC.

`ControladorBarraTren:` Clase para que el usuario interactúe tanto añadiendo un tren, como con la barra de botones.

Servicios	Descripción
`ControladorBarraTren(TrenSimulado_0 tr, VistaGeneral v1)`	En el constructor es donde va la GUI. Se sigue el patrón MVC, tal y como se indica en el constructor de su clase padre. Además se inicia el objeto Timer, el cual llevará el motor del Simulador.
`ControladorBarraTren(TrenSimulado_0 tr, VistaGeneral v1, TrenMaqueta_0 sm)`	En el constructor es donde va la GUI. Se sigue el patrón MVC, tal y como se indica en el constructor de su clase padre. Además se inicia el objeto Timer, el cual llevará el motor del Simulador. La referencia de TrenMaqueta_0 se utiliza para su gestión con la maqueta real.
`void activarControlador(TrenSimulado_0 train)`	Método para activar un UserControl para cada Tren que se incorpore a la maqueta, y a partir de ahí, controlar cada uno de manera independiente.
`void activarControlador(TrenSimulado_0 train, TrenMaqueta_0 sm)`	Método para activar un UserControl para cada Tren que se incorpore a la maqueta, y a partir de ahí, controlar cada uno de manera independiente, incorporando una referencia de TrenMaqueta_0 para utilizar la maqueta real
`void actualizar(int nro, String mens)`	Método para actualizar, tanto el número de trenes sobre la maqueta, como los mensajes que se quieran mostrar en la Vista Principal.
`void botonAyuda_Click(object sender, EventArgs e)`	Delegado que controla el evento generado por el botón Ayuda.
`void botonAcerca_Click(object sender, EventArgs e)`	Delegado que controla el evento generado por el botón Acerca.
`void botonSalir_Click(object sender, EventArgs e)`	Delegado que controla el evento generado por el botón Salir.
`void bt_añadirTren_Click(object sender, EventArgs e)`	Delegado que controla el evento generado por el botón Añadir Tren.
`void timer1_Tick(object sender, EventArgs e)`	Delegado que controla el evento generado por el tick del temporizador.

ControladorTren: Clase que genera la interfaz gráfica de los Controladores de Tren, que manejan los eventos de los botones "Velocidad +", "Velocidad -", y "Parar",	
Servicios	**Descripción**
`ControladorTren(ControladorBarraTren cbt, String id, int pos, TrenSimulado_0 t, TrenSimulado_0 tren)`	Constructor donde se establece la GUI.
`ControladorTren(ControladorBarraTren cbt, String id, int pos, TrenSimulado_0 t, TrenSimulado_0 tren, TrenMaqueta_0 sm )`	Constructor donde se establece la GUI., recibiendo además, una referencia de la clase TrenMaqueta_0 para sus gestiones con la maqueta real.
`void btIncVel_Click(object sender, EventArgs e)`	Delegado que controla el evento generado por el botón Incrementar velocidad.
`void btDecVel_Click(object sender, EventArgs e)`	Delegado que controla el evento generado por el botón Decrementar velocidad.
`void btParar_Click(object sender, EventArgs e)`	Delegado que controla el evento generado por el botón Parar.

ControladorAgujas: Clase controladora de las agujas. Acepta entradas de usuario eligiendo el tipo de aguja y su estado (curvo o recto), para posteriormente acceder a la funcionalidad del modelo de la Aguja.	
Servicios	**Descripción**
`ControladorAgujas(IAguja na[])`	Establece la Interfaz Gráfica del controlador de agujas, con la que interactúa el usuario.
`void radioButton[i]_CheckedChanged( Object sender, EventArgs e)`	Delegado que controla el evento generado por el cambio de posición de aguja, "i" es el indice.

5.2.2.4. Las vistas.

Las vistas se encargan de desplegar en una interfaz de usuario la información procedente del modelo.

Se han definido dos grupos de vistas. Por un lado se encuentran las vistas de los elementos de la maqueta y por otro lado las vistas más generales para mostrar al usuario, las cuales engloban un conjunto de vistas de elementos de la maqueta.

Para el primer grupo, se ha definido una interfaz común, denominada **`IVistaVia`**, la cual presenta la funcionalidad que tendrán las vistas de los diferentes elementos que tiene la vía férrea, tales como nodos de vía, agujas y sensores. La interfaz **`IVistaAguja`**, al ser hija de **`IVistaVia`**, añade nuevas funcionalidades además de las que tiene una vía.

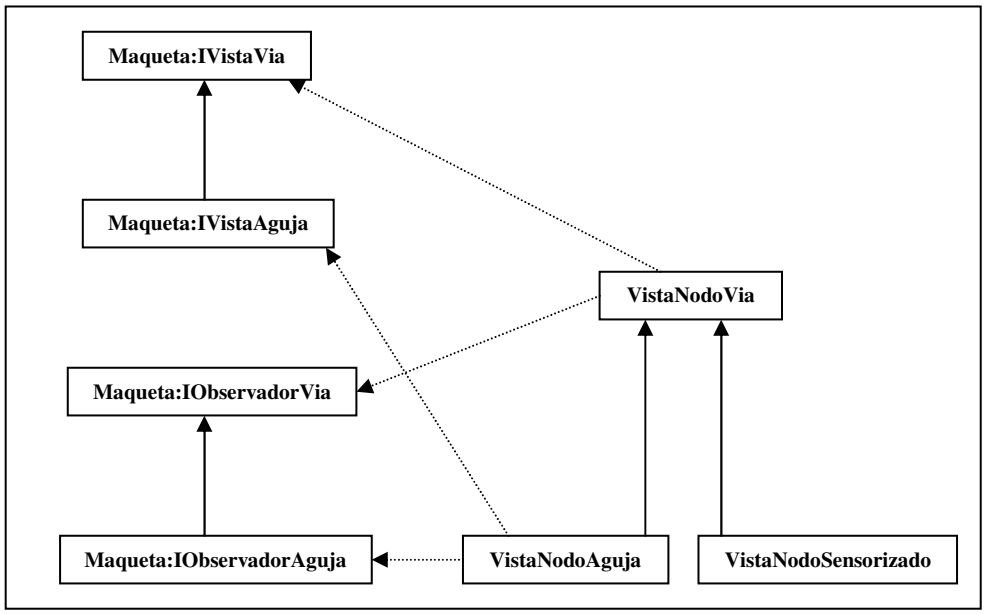
En cuanto a la estructura de clases, hay una clase **`VistaNodoVia`** que implementa **`IVistaVia`** y es la que desarrolla la funcionalidad general de cualquier nodo. De manera

más específica, existen las clases **`VistaNodoAguja`** y **`VistaNodoSensor`**, las cuales son clases hijas de **`VistaNodoVia`**, añadiendo nuevas implementaciones de acuerdo a las características que cada clase debe ofrecer. A su vez **`VistaNodoAguja`** implementa de **`IVistaAguja`**, desarrollando sus funcionalidades.

Todo esto se ve mas claro en la siguiente figura, estableciéndose la estructura que se ha seguido para las vistas de los elementos que conforman una maqueta de trenes.

Resumiendo, las vistas son objetos estrategia, y deben tener una interfaz, cada una definiendo diferentes implementaciones. Además, las vistas son observadores del modelo. Por lo tanto entran en juego los patrones Estrategia y Observador. Las vistas de los elementos de una maqueta implementan independientemente otra interfaz, para poder llevar a cabo el patrón MVC, al ser observadores del modelo. Los tres elementos de la maqueta implementan la interfaz **`IObservadorVia`**, mientras que como excepción, las agujas implementan **`IObservadorAguja`** también, debido a que es una especialización de **`IObservadorVia`**, presentando otras funcionalidades.

Cabe destacar que no se genera una vista para los trenes, sino para los elementos de la vía férrea. Los trenes no tienen una vista asociada porque no interesa visualizar el tren, sino por dónde pasa, indicando la localización del tren en cada instante.



Clases e Interfaces de los elementos de la maqueta.

En la tabla se recoge la información de los constructores propios de cada clase, ya que el resto se encuentra explicado en el capítulo 3.

Resumen de clases e interfaces de los elementos de la maqueta.	
VistaNodoVia: Clase que permite implementar la vista de los nodos, en particular los nodos de vía.	
Servicios	**Descripción**
`VistaNodoVia( )`	Constructor por defecto, para obtener el lienzo donde se genera la vía férrea.
`VistaNodoVia(INodoVia nv)`	Constructor con argumentos para suscribir la vista a su Modelo NodoVia.
VistaNodoAguja: Esta clase permite implementar la vista de las agujas, de una forma personalizada, dependiendo de las características de cada una de ellas.	
Servicios	**Descripción**
`VistaNodoAguja( )`	Constructor por defecto, para obtener el lienzo donde se genera la vía férrea.
`VistaNodoAguja(IAguja na)`	Constructor con argumentos para suscribir la vista a su Modelo NodoAguja.
VistaNodoSensorizado: Esta clase permite implementar la vista de los sensores.	
Servicios	**Descripción**
`VistaNodoSensorizado( )`	Constructor por defecto, para obtener el lienzo donde se genera la vía férrea.
`VistaNodoSensorizado(NodoViaSensorizado ns)`	Constructor con argumentos para suscribir la vista a su Modelo NodoViaSensorizado.
IObservadorVia: Interfaz común para modelar los objetos de las clases que la implementan. Se utiliza para actualizar siguiendo el patrón MVC, para indicar el estado de la vía (si está ocupado o no por un tren).	
Servicios	**Descripción**
`void update(boolean ocupado);`	Actualiza indicando para cada nodo vía si está ocupado por un tren o no
IObservadorAguja: Interfaz común para modelar los objetos de las clases que la implementan. Se utiliza para actualizar siguiendo el patrón MVC, para indicar el estado de la aguja, su sentido y su orientación cuando se mueva.	
Servicios	**Descripción**
`void updateAguja(boolean recto, int sentido, int updown);`	Actualiza indicando para cada aguja, su estado, sentido y orientación al moverse

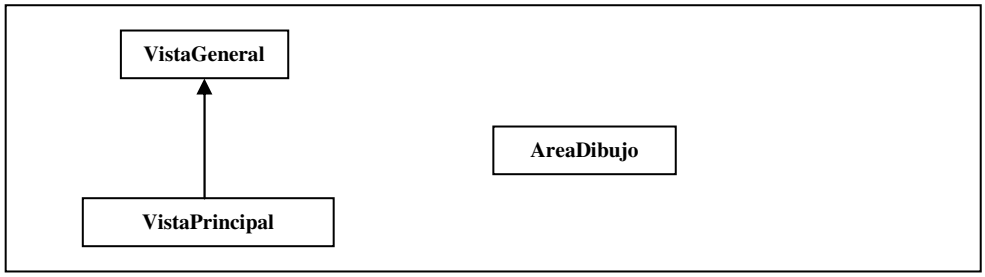
Como segundo grupo se encuentran las vistas generales para mostrar al usuario, entre las que se incluyen la interfaz gráfica general de la aplicación. Se compone de una clase abstracta, **`VistaGeneral`**, creada como clase padre para la clase **`VistaPrincipal`**, para utilizar el patrón MVC, suscribiendo esta vista general al modelo del tren. **`VistaPrincipal`** es la clase que establece la interfaz gráfica de usuario, formada por un panel donde se localiza el lienzo de la vía férrea, un área de texto para mostrar los mensajes de actualización, y un campo de texto para incluir comentarios locales, siempre siguiendo el patrón arquitectónico de este Proyecto (MVC). Independientemente de estas clases, se

crea otra llamada **`AreaDibujo`**, encargada de establecer la estructura de la maqueta en el simulador, es decir, la vía férrea (a escala de la maqueta real), actualizando cada elemento cambiante.

En la siguiente figura, se muestran los diagramas de clases y su estructura para este grupo. Además, en la tabla se muestra un resumen de cada uno de los métodos que proporciona cada clase.

Resumen de la funcionalidad de las clases VistaGeneral, VistaPrincipal y AreaDibujo.	
`VistaGeneral:` Clase general para mostrar la Vista General. Se suscribe al Modelo del Tren (Tren).	
Servicios	**Descripción**
`VistaGeneral (ITren t)`	Establece la estructura que debe tener una vista siguiendo MVC.
`void actualizar(int nro, String mens)`	Método para actualizar, tanto el número de trenes sobre la maqueta, como los mensajes que se quieran mostrar en la Vista Principal.
`void displayEstado();`	Método para mostrar todo lo actualizado, según MVC.
`VistaPrincipal:` Esta clase permite implementar la Vista Principal, formada por un UserControl donde se localiza el lienzo de la vía férrea, un área de texto para mostrar los mensajes de actualización y un cuadro de texto para incluir comentarios locales, siguiendo todo el Patrón de Diseño MVC.	
Servicios	**Descripción**
`VistaPrincipal(TrenSimulado_0 t)`	Constructor en el que establece la Interfaz Gráfica del simulador, en la que el usuario verá los resultados de sus acciones determinadas en los controladores.
`AreaDibujo Lienzo{get;}`	Devuelve el lienzo donde se encuentra la vía férrea de la maqueta.
`void displayEstado()`	Método para mostrar todo lo actualizado, según MVC. En este caso se trata de los mensajes que se vayan actualizando procedentes de alguno de los Modelos, y el número de trenes que hay en cada momento incorporado en la maqueta.
`AreaDibujo:` Clase para establecer la estructura de la maqueta en el simulador (a escala de la real), actualizar cada elemento cambiante, y pasar las referencias al Modelo apropiado, de cada Vista, según MVC.	
Servicios	**Descripción**
`AreaDibujo()`	Constructor por defecto

static AreaDibujo Lienzo{get;}	Devuelve el interconexionado de NodoVia, NodoAguja y NodoViaSensorizado (Via Ferrea) que se ha creado. En este caso con forma de la maqueta del laboratorio DSIE en escala
static Point alinearVistasEnRectaHor(IVistaVia [] nodosVia, Point org, int sentido, int delta)	Método para establecer los tramos de NodoVia rectos
static Point alinearVistasEnRectaHorSensor(IVistaVia nodoVia, Point org, int sentido, int delta)	Método para establecer los tramos de sensor en tramo recto
static Point alinearVistasEnArco(IVistaVia [] nodosVia, Point org, Point ctr, int delta, int sentido)	Método para establecer los tramos de vía en tramo curvo
static Point alinearVistasEnArcoSensor(IVistaVia nodoVia, Point org, Point ctr, int delta, int sentido)	Método para establecer los tramos de sensor en tramo curvo
static Point alinearVistasEnRectaAguja(IVistaVia nodoAguja, Point org, int sentido, int delta)	Método para establecer los tramos de aguja en tramo recto.
static Point alinearVistasEnRectaDiag_Inf(IVistaVia [] nodosVia, Point org, int sentido, int delta, int pendiente)	Método para establecer los tramos de vía en tramo recto pero en diagonal hacia abajo.
static Point alinearVistasEnRectaDiag_Sup(IVistaVia [] nodosVia, Point org, int sentido, int delta, int pendiente)	Método para establecer los tramos de vía en tramo recto pero en diagonal hacia arriba.
Void generarViaFerrea()	Método en el que se genera la vía férrea de la maqueta.
Void concatenar ()	Método para concatenar cada tramo de vía de la maqueta (vía férrea).
INodoVia[] getNodoVia()	Devuelve el conjunto de todos los NodoVia que tiene la vía férrea.
Void anadirVista(IVistaVia v)	Método para añadir los elementos que hay que dibujar en la vía férrea.
Void OnPaint(PaintEventArgs e)	Método para realizar el contexto gráfico, es decir, dibujar cada uno de los elementos de la vía férrea en el lienzo.
Void refrescar()	Método para refrescar el lienzo. Es el que llama a Refresh ().



Clases VistaGeneral, VistaPrincipal y AreaDibujo.

5.2.2.5. Las clases de Inicio.

Las clases de inicio son las que establecen el inicio de la aplicación desarrollada en este Proyecto.

Se han definido dos clases: **`InicioElegirOpciones`** e **`Inicio_Simulador`**. Son independientes pero relacionadas entre si.

La clase **`InicioElegirOpciones`** es la encargada de mostrar una ventana para interactuar con el usuario. Éste debe elegir entre **`"Modo Simulador"`** y **`"Modo Maqueta"`**. Si se elige el primer modo, se invoca a la clase **`Inicio_Simulador`**, y sigue el flujo que determina esta clase. El segundo modo no ha sido implementado, dejándose para futuros proyectos.

En la tabla se muestra un resumen sobre la funcionalidad de cada uno de los métodos de cada clase anteriormente definida.

Resumen de las clases de inicio de la aplicación	
`InicioElegirOpciones:` Clase principal, que permite elegir dentro del Simulador, el tipo de modo a utilizar, ya sea "Modo Local" o "Modo Remoto".	
Servicios	**Descripción**
`InicioElegirOpciones()`	Establece la Interfaz Gráfica con la que interactúa el usuario en primera instancia.
`void button1_Click(object sender, EventArgs e)`	Delegado que controla el evento de pulsar el botón “Modo Simulador”.
`void button2_Click(object sender, EventArgs e)`	Delegado que controla el evento de pulsar el botón “Modo Maqueta”.
`static void main (String args[])`	Método Principal. Aquí es donde comienza el programa del Proyecto.
`Inicio_Simulador:` Clase principal en “Modo Simulador”. Inicia el Simulador y realiza la creación de componentes según el Patrón de Diseño "Modelo-Vista-Controlador"	
Servicios	**Descripción**
`Inicio_Simulador ()`	Constructor inicializador del Modelo-Vista-Controlador para crear los componentes.
`static void main (String args[])`	Método Principal. Aquí es donde puede comenzar el programa del Proyecto si directamente sólo se quiere simular.

5.2.3. Visión dinámica del código.

La mejor manera de explicar el funcionamiento del código es mediante los diagramas de secuencia. A continuación se muestra uno que presenta el funcionamiento básico del **"Modo Simulador"**.

5.2.3.1. Modo Simulador.

En la siguiente figura se muestra el funcionamiento del programa en este modo. Se recoge la funcionalidad básica, incluyendo los aspectos más importantes. Obsérvese que todos los objetos descritos en la figura son discretos, a excepción de **"Modelo de las Vias"** y **"Vista de las Vias"**, que son agregados tanto de los modelos que representan los elementos de una vía férrea, como de sus vistas, respectivamente. Se ha hecho esta distinción, para evitar confusiones sobre el diagrama. Nótese cómo la aplicación se inicia tanto creando el modelo principal como la vista principal, y suscribiendo unos a otros. En la clase **AreaDibujo** se crean todos los modelos de los nodos y sus vistas, suscribiéndose también. Todo el tema de la simulación va dirigido por un **Timer**, que se ejecuta periódicamente, haciendo que cada tren avance o retroceda. Esto se notifica al modelo correspondiente y de ahí se actualizan los cambios. La activación de un controlador de tren para incrementar y decrementar la velocidad, o parar el tren, no se ha contemplado en este esquema, ya que es el mismo procedimiento y dificultaría la visión general del esquema. En este caso al pulsar el botón **"Añadir Tren"** definido en la clase **ControladorBarraTren**, se accede a la funcionalidad del modelo **TrenSimulado_0** para su incorporación a la vía. A su vez se activa un controlador para dicho tren, que se implementa en la clase **ControladorTren**, estableciendo la interfaz gráfica para poder interactuar con la funcionalidad del tren, que no se actualizará hasta que reciba un **tick()**. Al producirse, se invoca a los métodos **avanzar()** o **retroceder()** de esta clase modelo, **TrenSimulado_0**, cuyas funcionalidades ya se han explicado en este capítulo, notificando a sus modelos de vía de los cambios para actualizar su respectiva vista.

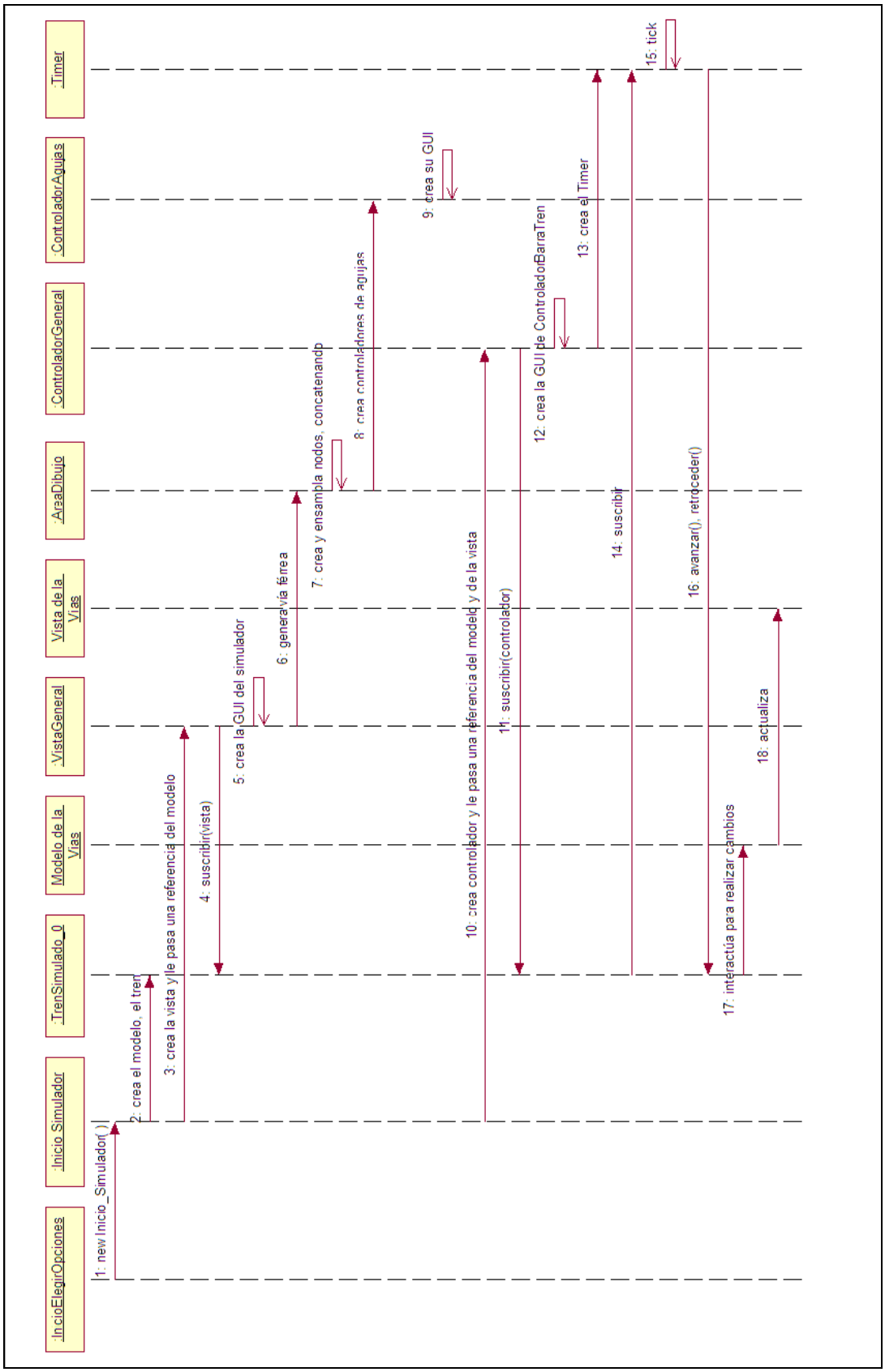


Diagrama UML de Secuencia, sobre el comportamiento general en "`Modo Simulador`".

Capítulo 6.
Código resultante con distribución.

6.1. Introducción.

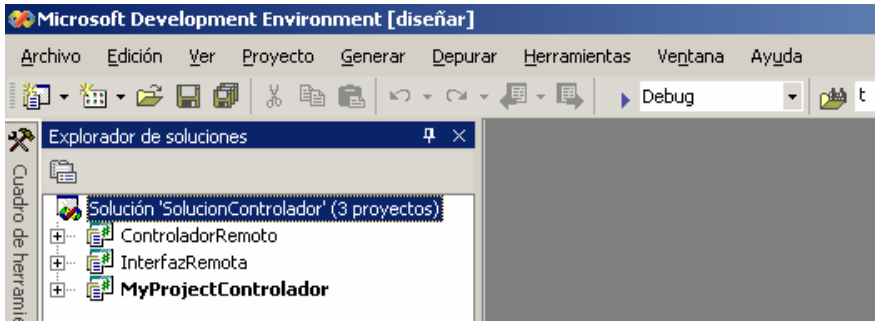
En este capítulo del proyecto, tras la presentación de la aplicación en los capítulos 4 y 5, vamos a explicar como ha sido modificado el código para distribuir la aplicación. El capítulo va a presentar dos aplicaciones, una distribución normal y otra distribución que sigue el patrón MVC. Cabe destacar que para lograr la distribución hemos utilizado **.NET Remoting**, ya que permite crear fácilmente aplicaciones ampliamente distribuidas, tanto si los componentes de las aplicaciones están todos en un equipo como si están repartidos por el mundo. Se recomienda la lectura del anexo C, en el cual se explica detalladamente .NET Remoting.

6.2. Aplicación distribuida.

En el capítulo siguiente se explica el manejo de la aplicación, aquí nos centramos en los puntos clave del diseño del código del proyecto.

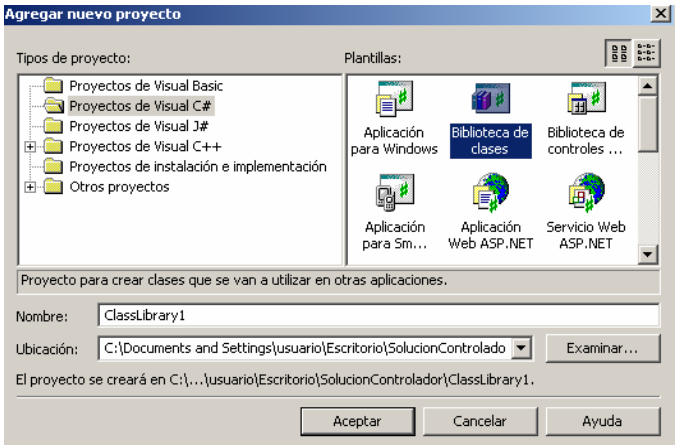
- **Solución** `SolucionControlador.`

Para la creación de la aplicación distribuida, hemos creado la solución **`SolucionControlador`**. Una solución es un conjunto de proyectos que conforman una aplicación.



"Explorador de soluciones".

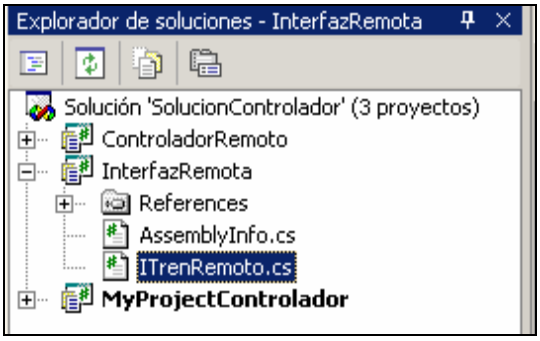
Esta solución está formada por tres proyectos, **`ControladorRemoto`**, **`MyProjectControlador`** e **`InterfazRemota`**, los dos primeros son del tipo "Aplicación para Windows" y el tercero es del tipo "Biblioteca de clases".



"Aplicación para Windows" y "Biblioteca de clases".

- **Biblioteca de clases `InterfazRemota`.**

La finalidad de la biblioteca de clases **`InterfazRemota`** es contener la definición de la interfaz **`ITrenRemoto`**, que será implementada por el objeto a distribuir.



Los métodos del objeto serán invocados remotamente mediante un objeto del tipo de esta interfaz.

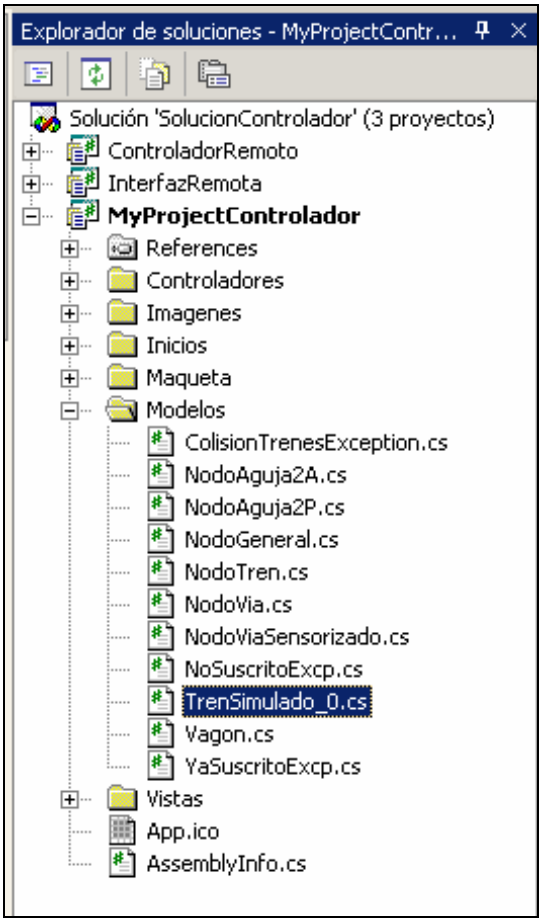
```
using System;

namespace Maqueta
{
        public interface ITrenRemoto
        {
                void  parar();

                void  incVelocidad();
```

```
                void  decVelocidad();

                int NumTrenes { set; get; }

                String Mensajes { set; }
        }
}
```

*Código: Interfaz **ITrenRemoto.***

- **Proyecto `MyProjectControlador.`**

El proyecto **`MyProjectControlador`** contiene la definición del objeto a distribuir, **`TrenSimulado_0`**, este proyecto es el que va a cumplir el papel de **servidor**, puesto que contiene la definición del objeto y la generación de este tipo de objetos.



*Explorador de Soluciones: Proyecto **MyProjectControlador.***

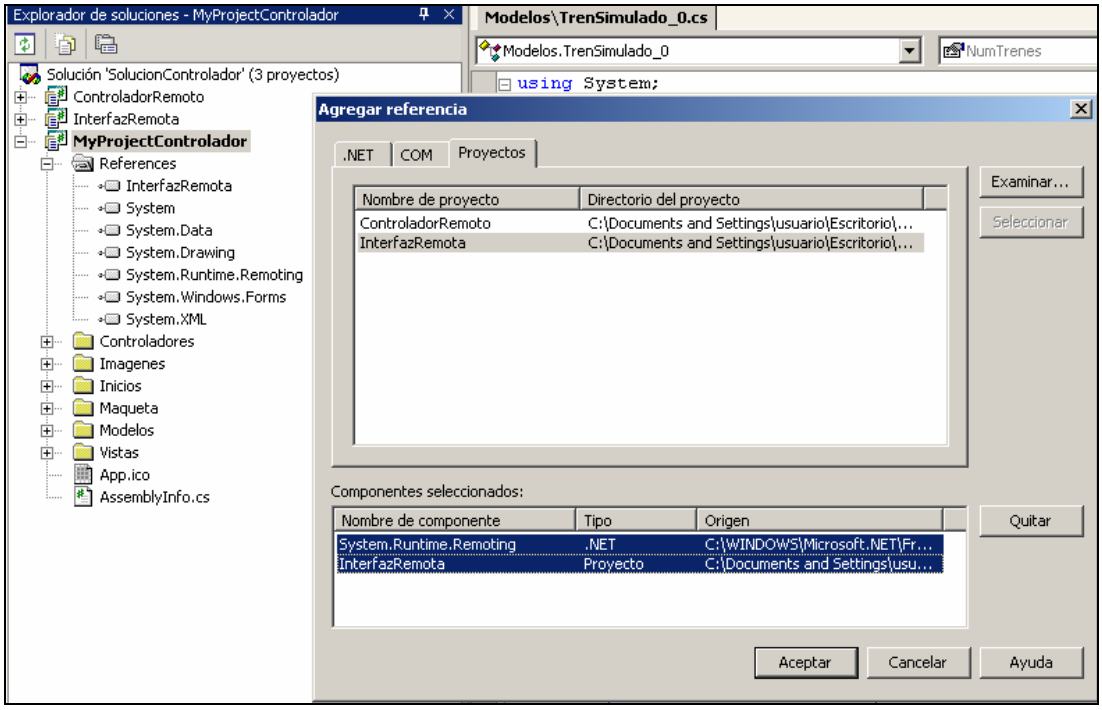
Para que la definición de este objeto sea accesible de forma remota:

- ✓ Tiene que hacer accesible el namespace **Maqueta** mediante la directiva **using**, puesto que es aquí donde está definida la interfaz **ITrenRemoto** y la clase **TrenSimulado_0** está definida en un namespace distinto.
- ✓ La clase **TrenSimulado_0** tiene que heredar de la clase **MarshalByRefObject**.
- ✓ La clase **TrenSimulado_0** tiene que implementar la interfaz **ITrenRemoto**.

```
…
using Maqueta;

namespace Modelos
{
        public class TrenSimulado_0 : MarshalByRefObject, ITrenRemoto, ITren,
                                              IObservadorTickSimulador, IModel_Tren
        {
                …
        }
}
```

*Código: Clase **TrenSimulado_0.***

Puesto que la interfaz **ITrenRemoto** se encuentra en un proyecto distinto, en la biblioteca de clases **InterfazRemota**, se tiene que agregar esta referencia al proyecto (en la pestaña "Proyectos" del gráfico). No nos tenemos que olvidar que también tenemos que agregar la referencia **System.Runtime.Remoting** al proyecto, puesto que esta no viene agregada por defecto (en la pestaña ".NET" del gráfico).



Agregar referencia.

Para que un objeto creado en la aplicación que actúa de servidor esté disponible de forma remota:

- ✓ Al iniciar la aplicación se tiene que implementar un canal emisor-receptor, en nuestro caso implementamos un canal que utiliza el protocolo **http**.
- ✓ En ese canal implementado indicamos el **puerto** por el cual ofrece el servicio la aplicación, en nuestro caso es el puerto 8085. Este puerto tiene que ser conocido por el cliente.
- ✓ Lo siguiente es **registrar** en el sistema el canal implementado.
- ✓ Para llevar a cabo lo anterior, utilizamos unas clases definidas en namespaces disponibles a partir de la referencia **`System.Runtime.Remoting`**, agregada anteriormente.

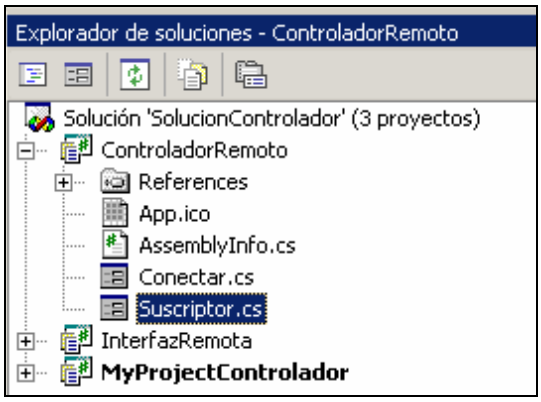
```
…
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
…
HttpChannel theChannel = new HttpChannel(8085);
ChannelServices.RegisterChannel(theChannel);
…
```

- ✓ Una vez que tenemos identificado en el código del programa el objeto creado que tenemos que hacer disponible remotamente, llamamos a la función **`Marshal()`**, pasándole como argumento el objeto a distribuir y la dirección URI con la que se calculan las referencias del objeto distribuido. Con esta función nos aseguramos de que tanto la aplicación cliente como servidor tienen la referencia al mismo objeto, así que cuando se produce cualquier cambio en el objeto ambas aplicaciones tendrán conocimiento de la nueva situación.

```
…
RemotingServices.Marshal( t, "TrenSimulado_0["+(pos).ToString()+"].soap" );
…
```

- **Proyecto `ControladorRemoto.`**

El proyecto **`ControladorRemoto`** contiene la clase que va a hacer uso del objeto **`TrenSimulado_0`** de forma remota, este proyecto es el que va a cumplir el papel de **cliente**. Por los motivos explicados anteriormente, a este proyecto agregaremos las mismas referencias.



*Explorador de Soluciones: Proyecto **ControladorRemoto.***

Para poder hacer referencia a un objeto que está en el servidor desde el cliente:

- ✓ Igual que hicimos en el servidor, al iniciar la aplicación implementamos y registramos el canal. Aquí no tenemos que registrar el puerto, eso solo lo hace el servidor.

```
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
…
HttpChannel theChannel = new HttpChannel();
ChannelServices.RegisterChannel(theChannel);


…
```

- ✓ Una vez que tenemos identificado en el código del programa el lugar donde efectuar la llamada al objeto remoto, hacemos la siguiente llamada a la función **`GetObject()`**.

```
model_tren[i] = (ITrenRemoto)Activator.GetObject(
                typeof(ITrenRemoto),
                "http://"+DirIPServ+":8085/"+"TrenSimulado_0["+(i).ToString()+"].soap");
```

Con la llamada a la función **`GetObject()`** de la clase **`Activator`** obtenemos una referencia al proxy del objeto remoto que le indicamos. Como primer argumento le indicamos el tipo del objeto del cual vamos a tener la referencia, en este caso **`ITrenRemoto`**, como segundo argumento le pasamos un String, formado por el **protocolo** de intercambio de mensajes (http), **dirección IP** del servidor al que nos conectamos (valor introducido por el usuario), **puerto** (8085) y la **dirección URI** con la que se calculan las referencias del objeto distribuido, definida en el servidor. La función devuelve un objeto del tipo **`Object()`**, por eso la conversión al tipo que nos interesa.

6.3. Aplicación distribuida siguiendo el patrón MVC.

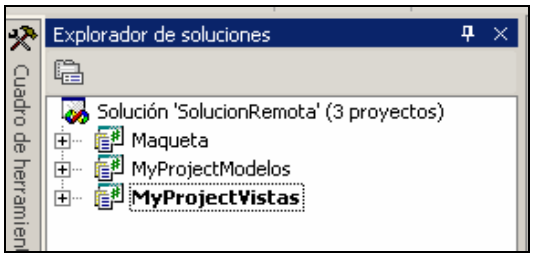
Otra parte del proyecto era lograr la distribución siguiendo el patrón Modelo-Vista-Controlador, del cual se han sentado las bases para alcanzarlo, no pudiéndose materializar en este proyecto.

La idea es que un programa hiciese de servidor de los modelos y cliente de las vistas de los objetos, y otro programa hiciese de cliente de los modelos y servidor de las vistas.

La distribución de los objetos y el procedimiento para lograrlo utiliza es el mismo que el presentado en el apartado anterior. Por eso obviaremos esa explicación en este apartado. Centrándonos en explicar la visión global del desarrollo de la aplicación.

- **Solución** **`SolucionRemota.`**

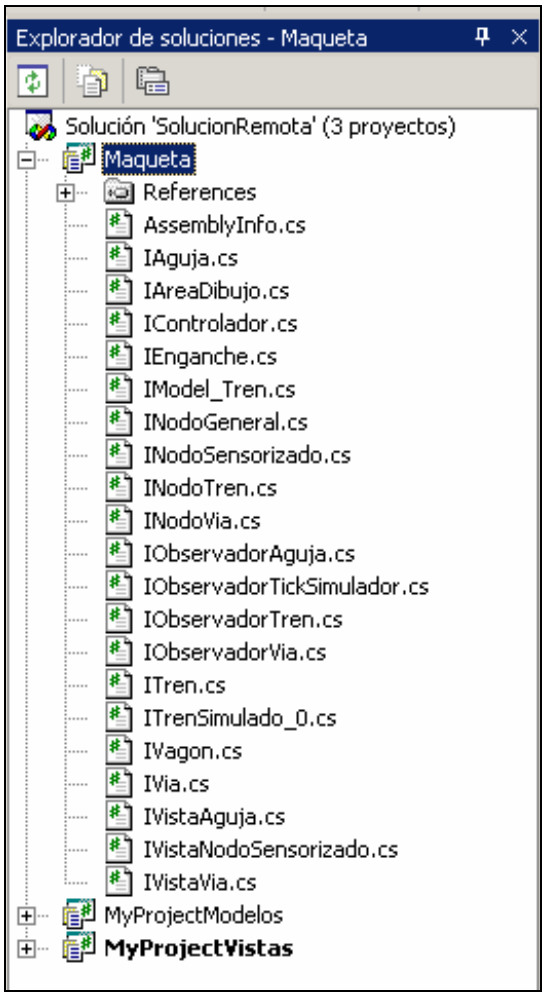
Para hacer esto se creó la solución **`SolucionRemota`**. Esta solución está formada por tres proyectos, **`MyProjectModelos`**, **`MyProjectVistas`** y **`Maqueta`**, los dos primeros son del tipo "Aplicación para Windows" y el tercero es del tipo "Biblioteca de clases".



Explorador de soluciones".

- **Biblioteca de clases `Maqueta`.**

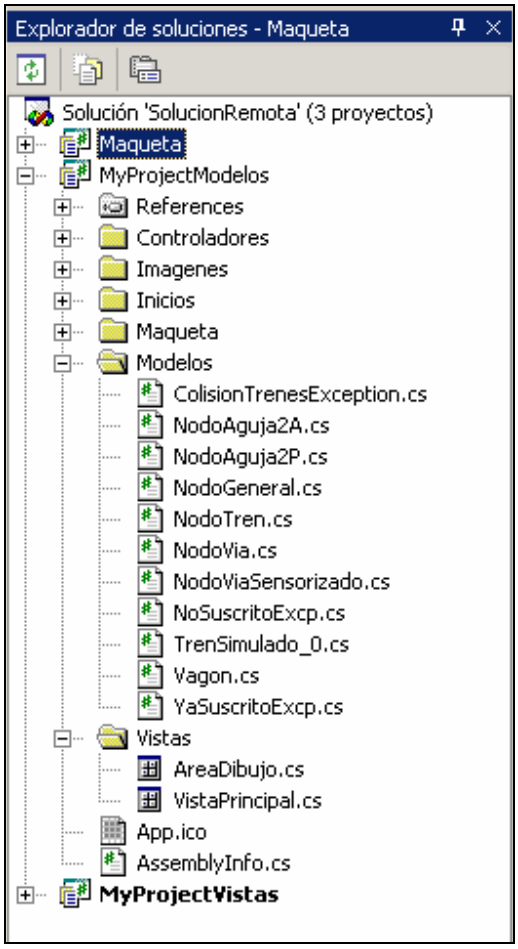
La finalidad de esta biblioteca es la definición de las respectivas interfaces que implementan los objetos que hacen de modelos o vistas. Esta biblioteca será agregada como referencia por ambos proyectos, del mismo modo que el explicado en el apartado anterior. Esta biblioteca contiene todas las interfaces que implementan todos los objetos de la aplicación. Para que los objetos cuya definición no es conocida de forma local sean accesibles de forma remota mediante las interfaces.



*Explorador de Soluciones: Biblioteca de clases **Maqueta.***

- **Proyecto `MyProjectModelos.`**

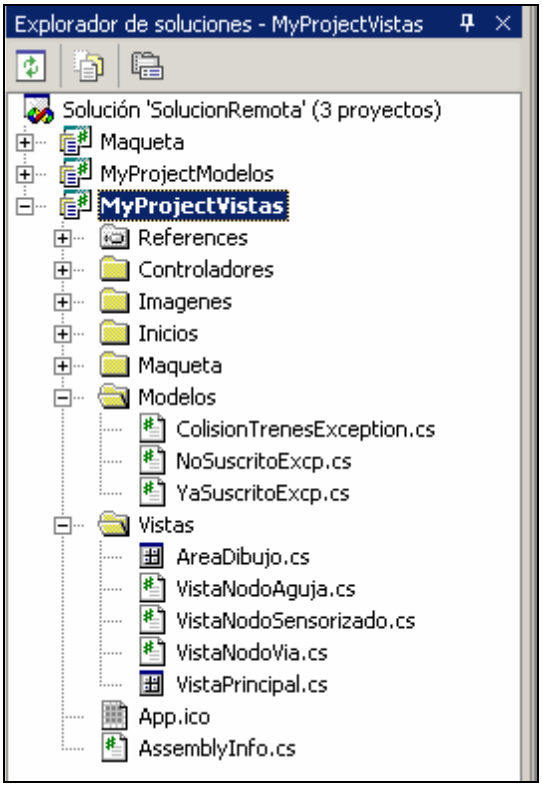
Este proyecto es el que contiene las definiciones de los objetos que hacen de modelos, y se encuentran en el namespace **Modelos**, como se puede observar. Esta clase no contiene las definiciones de los objetos que hacen de vistas, también como se puede observar.



*Explorador de Soluciones: Proyecto **MyProjectModelos.***

- **Proyecto `MyProjectVistas.`**

Este proyecto es el que contiene las definiciones de los objetos que hacen de vistas, y se encuentran en el namespace **`Vistas`**, como se puede observar. Esta clase no contiene las definiciones de los objetos que hacen de modelos, también como se puede observar.



*Explorador de Soluciones: Proyecto **MyProjectVistas.***

La gran parte del código encargada de llevar a cabo la distribución es la clase **`AreaDibujo`**, en el método **`generarViaFerrea`** una clase implementada en ambos proyectos. En el proyecto **`MyProjectModelos`** se crean los modelos y se les hace disponer de forma remota, y se accede de forma remota a las vistas. Mientras que en el proyecto **`MyProjectVistas`**, se crean las vistas y se les hace disponer de forma remota, y se accede de forma remota a los modelos.

No se alcanzó el propósito de distribuir la aplicación bajo el patrón MVC. Aunque la base de la distribución queda presentada en este proyecto. Como se puede observar en el código del proyecto.

Capítulo 7.
Manual de usuario.

7.1. Introducción.

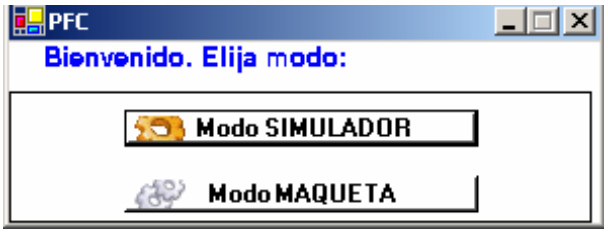
Vamos a mostrar las indicaciones que se deben seguir para ejecutar los diversos programas que componen el proyecto, así como explicar su funcionamiento. Para estas ejecuciones no es necesario instalar ningún tipo de programa, puesto que al haber trabajado con **Microsoft Visual Studio .NET 2003** en la generación del proyecto crea los correspondientes archivos de extensión **`.exe`**, con los que haciendo doble clic en el icono se lleva a cabo su ejecución. Tal y como se pudo comprobar en la ejecución de la aplicación correspondiente al capítulo 2 de este proyecto.

Todos los archivos necesarios se encuentran en carpetas de la carpeta **`Ejecutables`** del CD que acompaña a este proyecto.

7.2. Ejecutable del proyecto.

Este ejecutable se encuentra en la carpeta **`Capítulo 5`**, el cual se llama **`MyProject.exe`**.

Una vez ejecutada la aplicación, haciendo doble clic en el ejecutable, aparece una ventana de bienvenida como la de la figura.



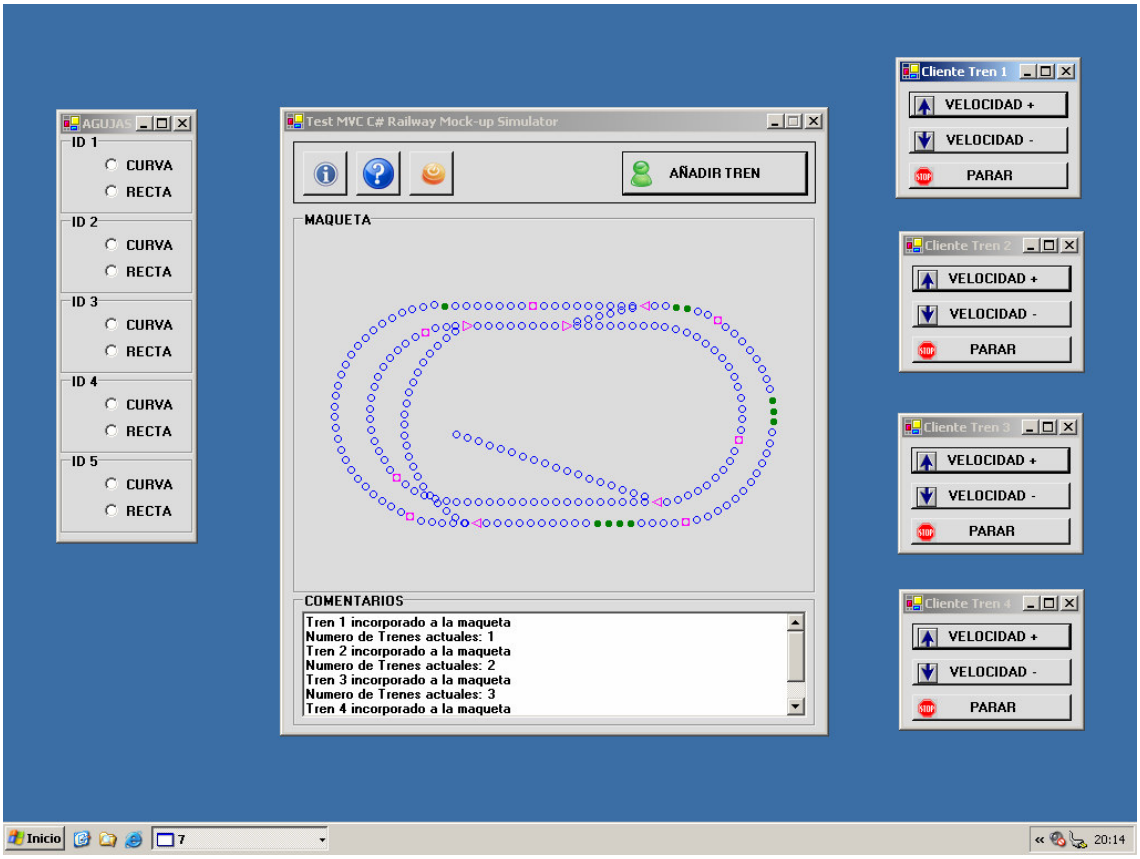
En esta ventana, hay que elegir el modo en el que se quiere trabajar:

- ✓ **Modo SIMULADOR**: Para ejecutar el simulador de la maqueta de trenes.

- ✓ **Modo MAQUETA**: Para ejecutar el simulador y la maqueta en función de las órdenes que se envíen mediante los controladores, esta opción no ha sido implementada en este proyecto, dejándose para futuros.

En la siguiente figura se muestra la disposición actual de los componentes de la interfaz gráfica del simulador. Aquí se obtiene una visión más generalizada. Se distinguen tres partes en la imagen de izquierda a derecha: controlador de agujas, área principal del simulador y controladores de trenes.

Destacar que se monitoriza la posición de cada tren en cada instante mediante los círculos rellenos de verde. Se distingue también cada tren por el número de vagones que lleva incorporados de acuerdo al número de tren.



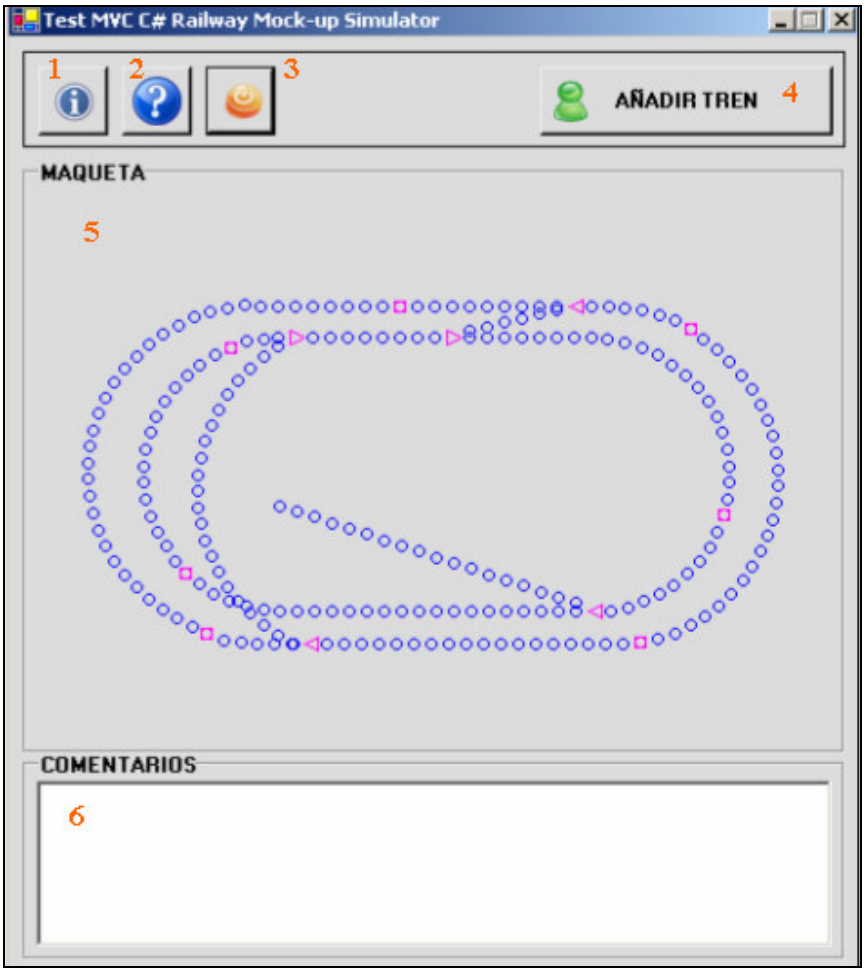
Interfaz Gráfica General de la aplicación.

A continuación se explica cada componente y la forma de utilización. La siguiente figura muestra la vista principal del simulador. Cada número hace referencia a un elemento útil para el usuario. Se realiza un estudio más detallado en la siguiente tabla:

Identificador	Descripción
1	Botón para activar o desactivar la ayuda.
2	Botón “acerca de” que muestra información del Proyecto.
3	Botón para decidir si se sale de la aplicación o no.
4	Botón para añadir un tren a la maqueta del simulador.

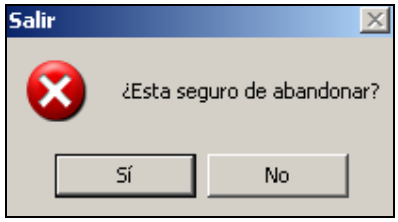
5	Maqueta a escala de la real, con sus correspondientes elementos, para monitorizar el estado de cada elemento y la posición de cada tren sobre ella.
6	Área de mensajes para mostrar información sobre el número de trenes que hay en la maqueta y otra información cualquiera.

Descripción de los identificadores mostrados.



Resultado de pulsar el botón identificado con el número 2.

Resultado de pulsar el botón identificado con el número 1.

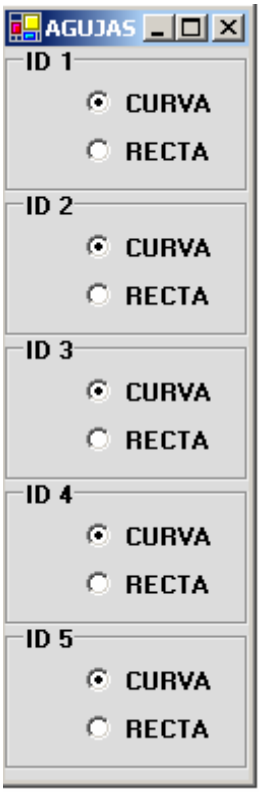


Destacar también, que al pulsar el botón **"Añadir Tren",** crea un controlador para el tren que se ha añadido, para poder controlarlo específicamente.

La siguiente figura muestra el controlador de las agujas. Se recoge el significado de cada identificador en la siguiente tabla.

Identificador	Descripción
R	Orientación recta de la aguja específica.
ID	Identificador de la aguja. Aguja 1, Aguja 2, etc.
C	Orientación curva de la aguja específica.

Descripción de los identificadores mostrados en la figura.

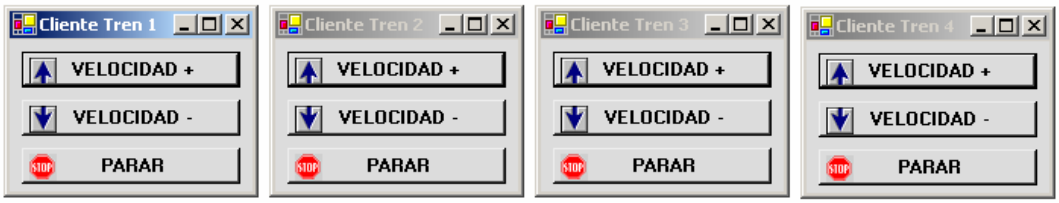


Controladores de las Agujas

En el simulador, si se pincha el botón “C” de ID 1, gira la aguja 1, si estuviese en estado recto, a curvo. La siguiente figura muestra el controlador de cada tren. Al igual que antes, en la siguiente tabla se recoge la información acerca de la funcionalidad.

Identificador	Descripción
Velocidad +	Incrementa un nivel la velocidad de un tren determinado.
Velocidad -	Decrementa un nivel la velocidad de un tren determinado.
Parar	Para un tren determinado.

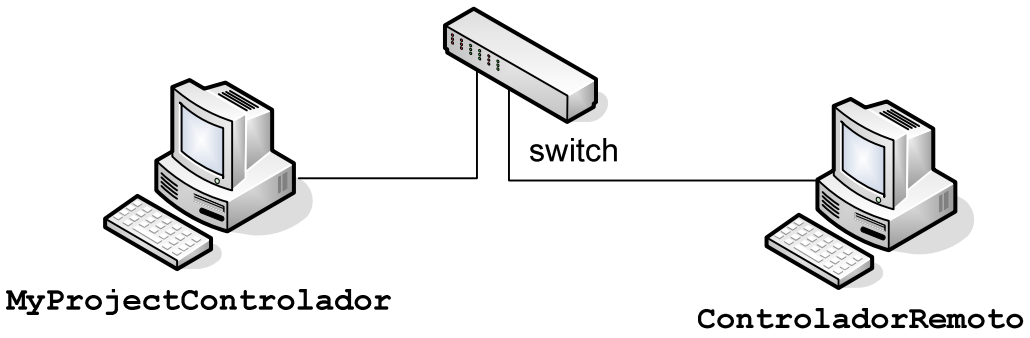
Descripción de los identificadores mostrados en la siguiente figura.



Controladores de los Trenes.

7.3. Ejecutables del proyecto distribuido.

Al ser un proyecto distribuido vamos a presentar la ejecución en dos computadores que se encuentran en la misma red.



En una computadora vamos a ejecutar la aplicación que hace de servidor. El ejecutable de esta aplicación se encuentra en la carpeta **`Capítulo 6`**, y se llama **`MyProjectControlador.exe.`** En otro computador se va a ejecutar la aplicación **`ControladorRemoto.exe`**, que hace de servidor. La condición para que se ejecuten ambas aplicaciones es que en los directorios desde donde se ejecutan estas aplicaciones se encuentre el archivo **`InterfazRemota.dll`**. Que es el archivo de la librería que relaciona ambas aplicaciones distribuidas, como se explicó en el anterior capítulo.

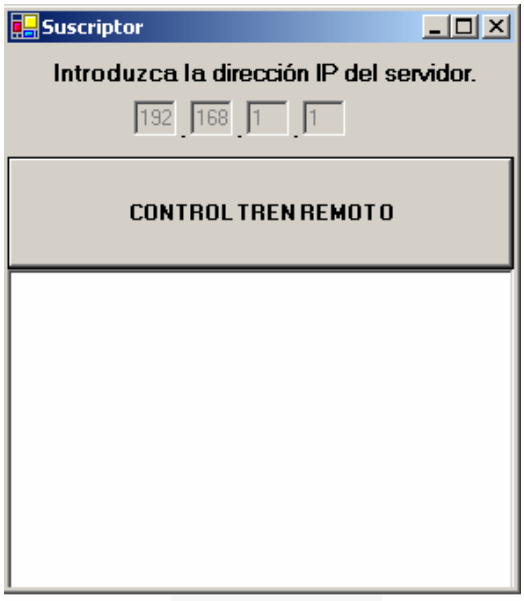
- **Ejecución de `MyProjectControlador.exe`.**

La ejecución es exactamente igual que la explicada en el apartado anterior.

- **Ejecución de `ControladorRemoto.exe`.**

En esta aplicación, al ejecutar nos va a salir la siguiente pantalla.

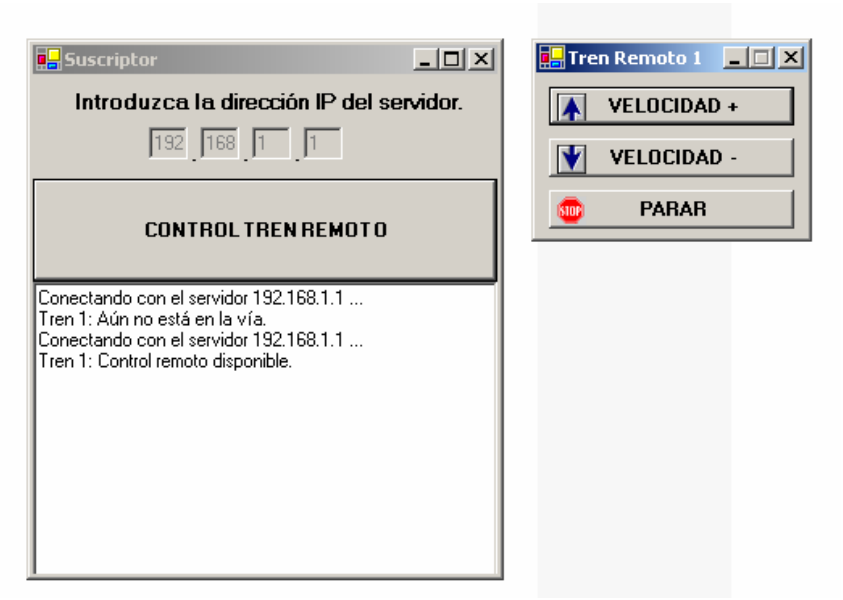
Se nos solicita la dirección IP de la computadora que está ejecutando la aplicación que hace de servidor, para esta explicación la dirección IP del servidor escogida es la 192.168.1.1. Pulsamos el botón.



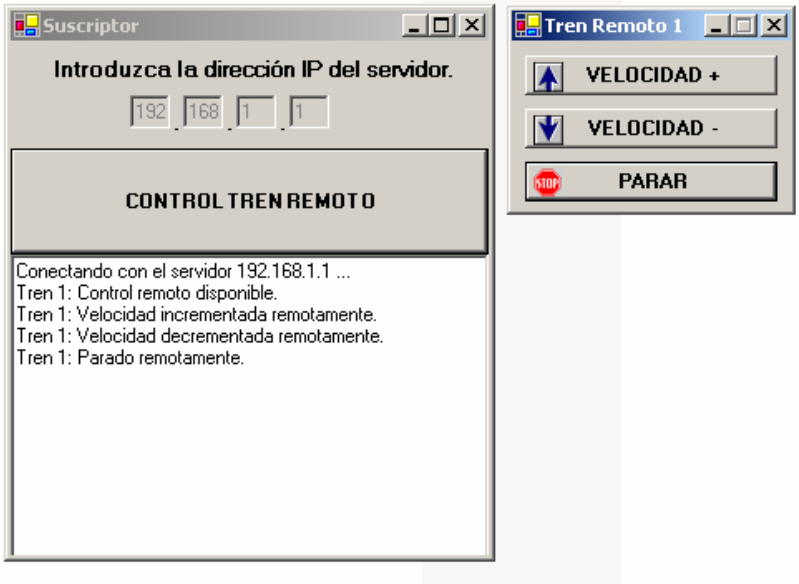
Si intentamos conectarnos antes de que el servidor esté activo nos lo indicará en la pantalla, y nos volverá a pedir una dirección IP.

Si nos conectamos después de que el servidor esté activo pero aún no hay ningún tren en la vía se nos informará de ello.

En el instante en el cual el usuario de la aplicación servidora añade un tren, este ya podrá ser disponible de forma remota, pulsamos el botón "**CONTROL TREN REMOTO**".



Al pulsar el botón "VELOCIDAD +" el tren se pondrá en funcionamiento de forma remota. Esto se comprueba observando la maqueta de tren que aparece en la maqueta de la aplicación del computador servidor. También podemos decrementar la velocidad del tren y pararlo mediante los otros botones.



A medida que desde el servidor se van añadiendo trenes a la maqueta estos pueden ser accesibles de forma remota, como acabamos de explicar. El número máximo de trenes es cuatro.

Capítulo 8.
Conclusiones.

8.1. Objetivos de partida.

Los objetivos iniciales con los que nació este Proyecto Final de Carrera eran:

1. Introducir la tecnología .Net y el lenguaje C# en el control de la maqueta de trenes.
2. Proporcionar ejemplos didácticos para la docencia.
3. Traducir a C# la aplicación Java de control de la maqueta de trenes.
4. Distribuir dichas aplicaciones mediante .Net Framework Remoting.

8.2. Resultados obtenidos.

El primer objetivo se ha conseguido tras un estudio de la plataforma .Net y el lenguaje C#, y su comparación con Java, ambos orientados a objetos, con sus semejanzas y diferencias.

Con el segundo objetivo se ha alcanzado una aplicación sencilla pero que engloba las características del lenguaje del proyecto, el lenguaje C# frente al lenguaje Java.

El tercer objetivo fue uno de los más laboriosos sino el que más, partimos de la base de un proyecto anterior en otro lenguaje, Java. Del que intentamos seguir la misma estructura. Lográndose este objetivo al salvar las diferencias que había principalmente en la adaptación de las clases de las bibliotecas. Quedando una aplicación ejecutable que cumple con los requisitos deseados.

El cuarto objetivo era el reto de lograr la distribución de la aplicación. Partimos de dos enfoques, uno complicado, el que debía seguir el patrón MVC, del cual hemos sentado las bases para su posterior logro, no alcanzándolo en este. Y otro de distribución libre, una aplicación que demuestra las capacidades de la distribución de .Net Remoting en la aplicación de la simulación de la maqueta.

Capítulo 9.
Bibliografía y referencias.

[1] *Simulador Java de una maqueta de trenes.* José Andrés Martínez Muñoz. Proyecto Fin de Carrera. Escuela Técnica Superior de Ingeniería de Telecomunicaciones. Universidad Politécnica de Cartagena, 07-2007.

[2] *C#. Manual de referencia.* Herbert Schildt. Ed. McGraw-Hill.

[3] *C# for Java Programmers.* Brian Bagnall, Philip Chen, Stephen Goldberg, Jeremy Faircloth y Harold Cabrera. Ed. Syngress.

[4] *The C# Programming Language.* Anders Hejlsberg, Scout Wiltamuth y Meter Golde. Ed. Addison-Wesley. Pearson Education.

[5] *Remoting with C# and .NET. RemoteObjects for Distributed Applications.* David Conger. Ed. Wiley.

[6] *Advanced .NET Remoting.* Ingo Rammer y Mario Szpuszta. Ed. Apress.

[7] *Pattern Oriented Software Architecture Vol1: A system of patterns: Pattern languages of program design* F. Buschmann et al John Wiley & Son Ltd

[8] *Design Patterns, Elements of Reusable Object-Oriented Software* Erik Gamma, Richard Helm, Ralph Johnson, John Vlissides Addison Wesley, 1995

[9] *UML y Patrones.- Introducción al análisis y diseño orientado a objetos* Craig Larman Prentice Hall 1ª Edición, 1999

[10] *Diseño e Implementación de una Interfaz Java para Comunicaciones con una Maqueta de Trenes.* Ricardo Jover Acosta. Proyecto Fin de Carrera, Escuela Técnica Superior de Ingeniería de Telecomunicaciones. Universidad Politécnica de Cartagena, 04-2006.

[11] *Modelado y control de una maqueta ferroviaria.* Javier del Palacio Paredes. Proyecto Fin de Carrera, Escuela Técnica Superior de Ingeniería de Telecomunicaciones. Universidad Politécnica de Cartagena, 01-2006.

Anexo A.
Enunciado de la práctica de Complementos de Informática.

A.1. Objetivos y enunciado de la práctica.

Los principales objetivos de esta práctica son:

- Implementar programas con varios hilos de control.
- Ofrecer ejemplos de uso de los patrones **`Observador`** y **`Modelo-Vista-Controlador`**.

Se trata de:

1. Implementar un hilo productor de enteros **`Productor_0`** y un hilo consumidor de enteros **`Consumidor_0`** que implementen la interfaz **`common.ComponenteActivo`** y se comuniquen a través de la clase **`p3.bufferCompartido.Buffer_0`**.

2. A partir de las clases **`ComponenteActivoView_0`** y **`ComponenteActivoController_0`** definir una ventana **`UIComponenteActivo_0`** que constituya la interfaz de usuario de un componente activo.

3. Utilizar **`UIComponenteActivo`** para proporcionar interfaces de usuario a **`Productor_0`** y **`Consumidor_0`**.

4. Implementar un programa **`TestBuffer_0`** de prueba de las clases implementadas en los apartados anteriores.

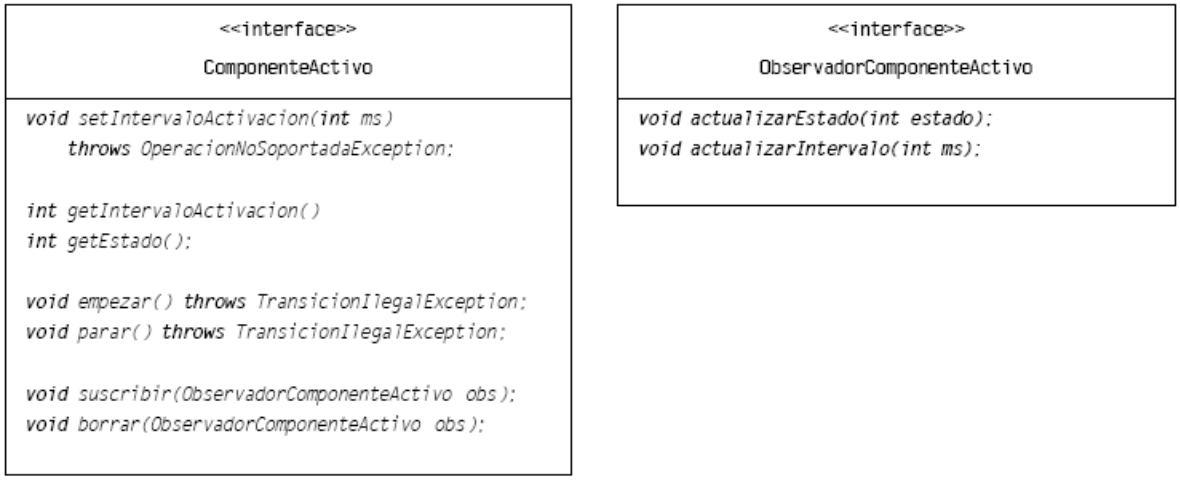
A.2. Interfaces proporcionadas.

A.2.1. Interfaces ComponenteActivo y ObservadorComponenteActivo.

La interfaz **`ComponenteActivo`** (figura B.1) define una interfaz común para todos aquellos objetos que:

- Deben ejecutarse en su propio hilo de control realizando una acción de forma periódica.
- Puedan ser detenidos y rearrancados.
- Propagan su estado a los observadores interesados en sus cambios. Este estado consiste en el valor de su periodo de activación y en su estado de funcionamiento (detenido o en marcha).

La interfaz **`ObservadorComponenteActivo`** (figura B.1) define la interfaz que deben exponer los objetos que deseen suscribirse a un **`ComponenteActivo`**.



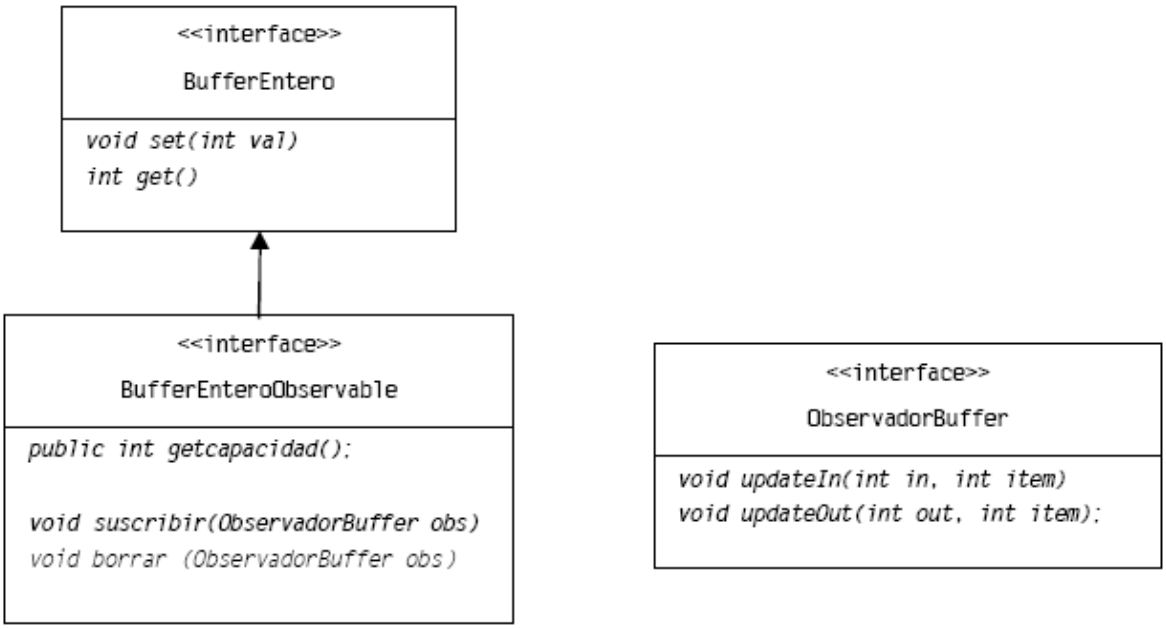
Interfaces ComponenteActivo y ObservadorComponenteActivo.

A.2.2. Interfaces BufferEntero, BufferObservable y ObservadorBuffer.

La interfaz **`BufferEntero`** define una interfaz común para todos los contenedores de valores enteros.

La interfaz **`BufferObservable`** extiende **`BufferEntero`** para que los contenedores puedan propagar cambios en su estado según un patrón observador. Finalmente, la interfaz

ObservadorBuffer define una interfaz para los observadores de un **BufferObservable** (véase figura B.2).

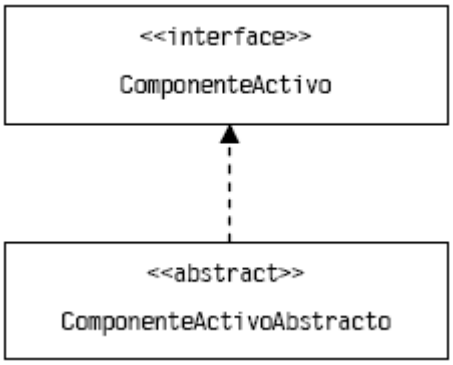


Interfaces BufferEntero, BufferEnteroObservable y ObservadorBuffer.

A.3. La clase ComponenteActivoAbstracto.

La clase **ComponenteActivoAbstracto** proporciona una implementación por defecto de todos los métodos de **ComponenteActivo** excepto de **run()**. De esta forma, una clase que extienda **ComponenteActivoAbstracto** tiene solucionada:

- La gestión del arranque y parada del componente activo, salvo el código de terminación en el método **run** (véanse comentarios en código fuente).
- La suscripción y borrado de los observadores del componente activo y su actualización cuando cambia su estado o su intervalo de activación.
- La actualización del intervalo de activación.



A.4. Vistas y controladores proporcionados.

Vistas y Controladores Abstractos.

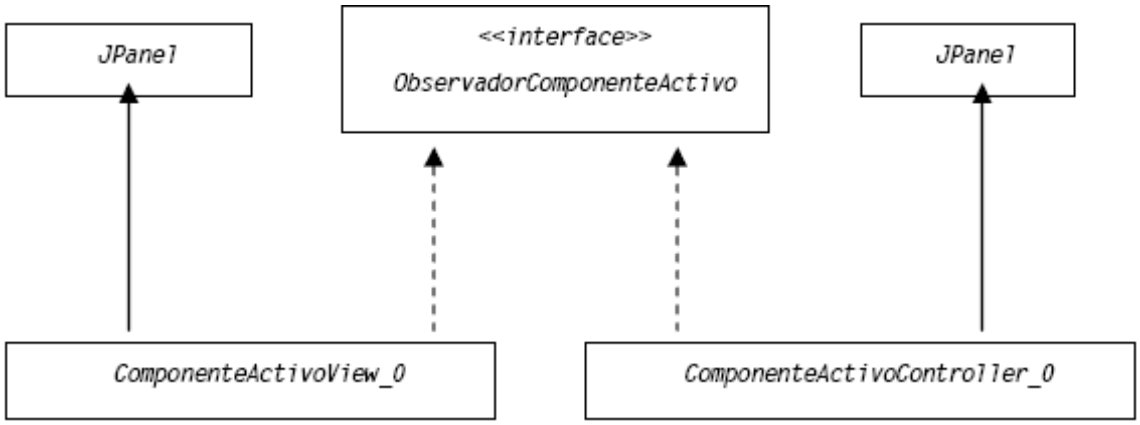
Tal y como define el patrón **MVC**:

- La **`vista`** despliega en una interfaz de usuario la información del modelo.
- El **`controlador`** acepta entradas del usuario para acceder a la funcionalidad del modelo.
- Vistas y controladores son observadores del modelo.

Vistas y Controladores de componentes activos.

Se proporcionan 2 clases que implementan respectivamente una vista y un controlador por defecto para un **`ComponenteActivo`**. Dichas clases son:

- **`ComponenteActivoView_0.`**
- **`ComponenteActivoController_0.`**


JPanel
<<interface>>
ObservadorComponenteActivo
JPanel
ComponenteActivoView_0
ComponenteActivoController_0

Anexo B.
Patrones de Diseño

B.1. Introducción

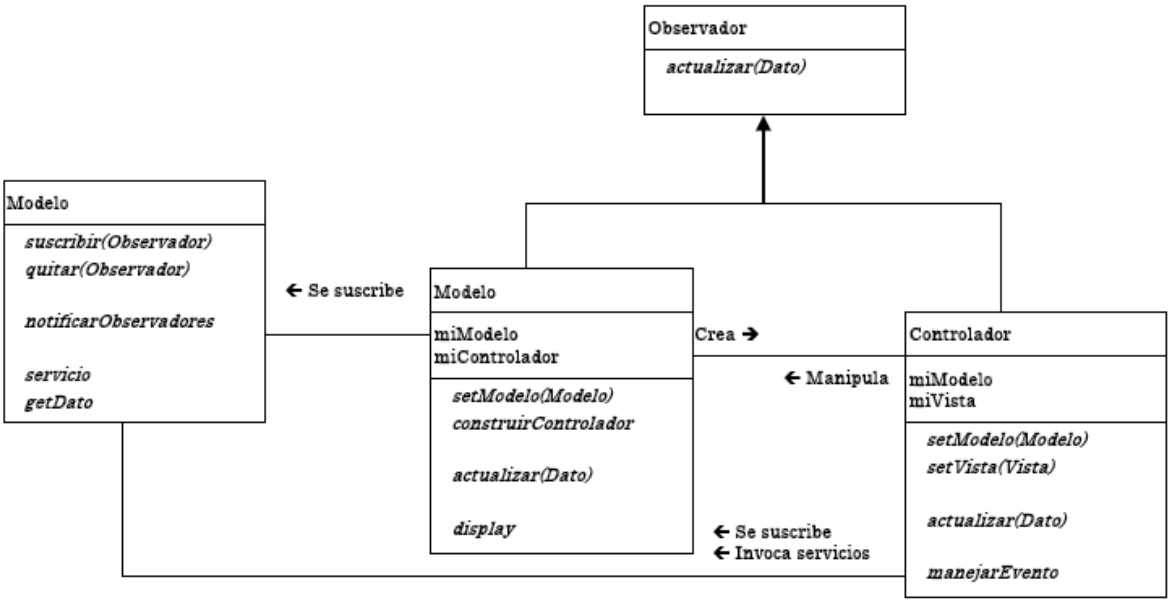
A continuación se expone una descripción de cada uno de los patrones de diseño que se han utilizado en este Proyecto.

B.2. Patrones de diseño utilizados

Un patrón es una solución recurrente a un problema estándar [8]. Los patrones de diseño capturan las estructuras estáticas y dinámicas de soluciones que funcionan de forma satisfactoria dentro de ciertos contextos o aplicaciones. Cada patrón define ciertos componentes y las reglas mediante las cuales pueden relacionarse con objeto de resolver un problema software específico. La ingeniería del software dispone ya de un conjunto muy extenso y convenientemente catalogado de patrones que describen soluciones probadas a problemas comunes. Se han aplicado ciertos patrones de diseño software. Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno y proporciona una solución a tal problema, de manera que es posible utilizar dicha solución muchísimas veces sin hacer exactamente lo mismo. La utilización y aplicación de patrones de diseño que se adapten a los requisitos del Proyecto no han sido consecuencia de un capricho, sino de la misma naturaleza de adaptabilidad a los requisitos que este Proyecto, a causa de su motivación, debe soportar. Un cambio en las necesidades o en los requisitos acarrea un cambio en el sistema, y a mayor robustez del sistema menores habrán de ser los cambios. Este proyecto trata de alcanzar una buena robustez. En este Proyecto se han utilizado sobre todo los patrones de diseño Modelo-Vista-Controlador [7], Observador [8] y Estrategia [8].

B.2.1. MVC (Modelo-Vista-Controlador).

El objetivo es separar la interfaz de usuario de una aplicación de su lógica interna y permitir que ambos evolucionen por separado. Se suele aplicar para el diseño de aplicaciones interactivas con interfaces gráficas de usuario. Este patrón, permite dividir la aplicación en tres áreas: procesamiento, entrada y salida, que se corresponden con los tres tipos de componentes que define: modelo, vista y controlador, cuyas relaciones se resumen en la siguiente figura



Patrón MVC. Relaciones estructurales.

Componentes:

- ✓ El modelo representa la funcionalidad y los datos de la aplicación.
- ✓ La vista despliega en una interfaz de usuario la información del modelo.
- ✓ El controlador acepta entradas del usuario para acceder a la funcionalidad del modelo.

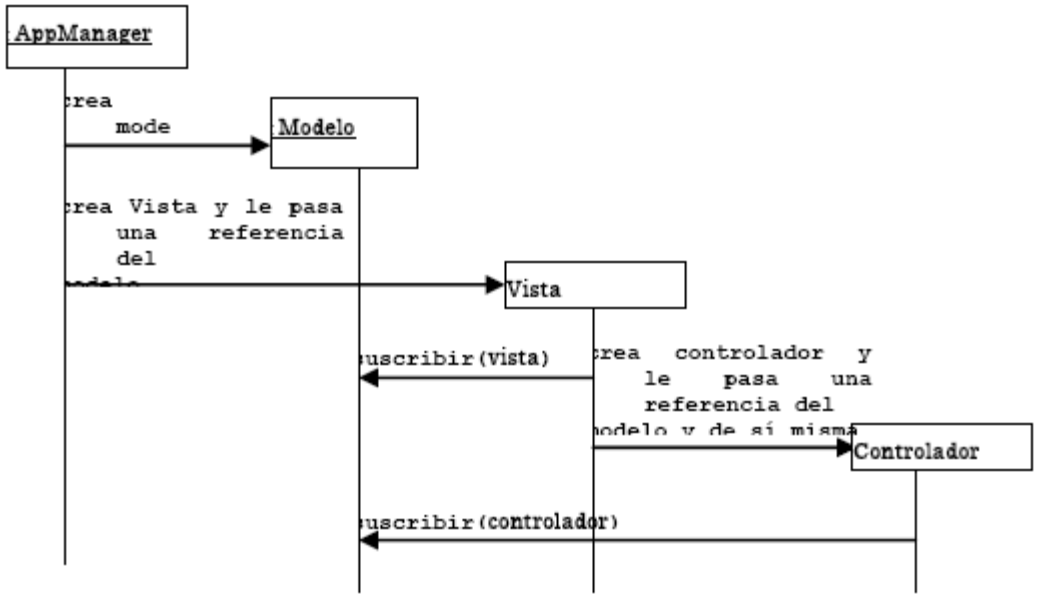
A continuación se detalla la funcionalidad que sigue este patrón:

1. El controlador proporciona la funcionalidad (servicios) de la aplicación.

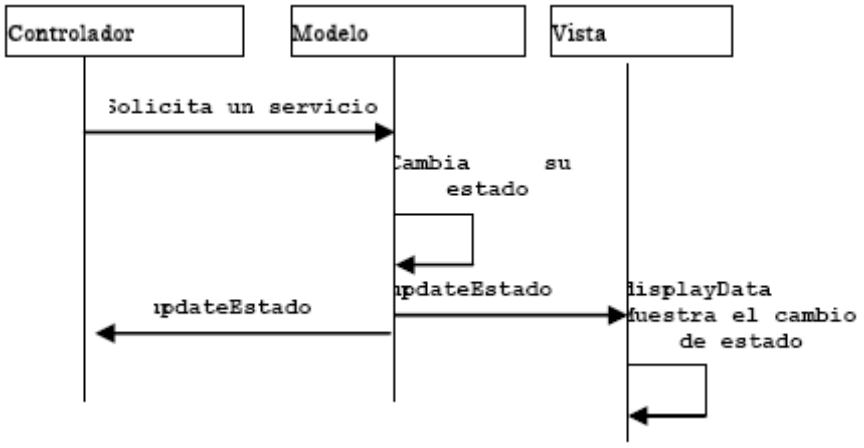
2. La vista y el controlador conforman la interfaz de usuario del modelo, el cual propaga sus cambios a los mismos siguiendo un patrón Observador.

3. *Creación y arranque de los componentes* (véase figura): Habitualmente un programa principal o un gestor de aplicación crea el modelo y la vista, la cual a su vez crea al controlador. Este esquema puede modificarse ligeramente cuando conviene.

4. *Funcionamiento* (véase figura): El controlador solicita un servicio al modelo. Como consecuencia de este servicio el modelo se modifica y notifica el cambio a su vista y a su controlador, produciéndose la actualización de la interfaz de usuario.



Patrón MVC. Creación de componentes.



Patrón MVC. Funcionamiento.

A continuación cabe destacar las siguientes ventajas al utilizar este patrón:

- ✓ Es fácil sustituir una interfaz de usuario por otra.

✓ Es fácil sustituir un componente gráfico por otro.

✓ Existe la posibilidad de definir bibliotecas de componentes para construir interfaces de usuario.

✓ Posibilidad de componer diferentes vistas simultáneas sincronizadas de un mismo modelo.

Pero como todo patrón, también tiene sus inconvenientes:

✓ Implementación compleja.

✓ Es difícil diseñar la jerarquía de componentes.

✓ La vista y el controlador están muy fuertemente acoplados.

B.2.2. Observador

Este patrón define una relación de dependencia uno a muchos entre un sujeto y sus observadores, de forma que cada vez que cambie el estado del sujeto, todos sus observadores sean notificados y actualizados automáticamente.

El objetivo de este patrón es desacoplar la clase de los objetos clientes del objeto, aumentando la modularidad del lenguaje, así como evitar bucles de actualización (espera activa o polling). A continuación se muestra la aplicabilidad que tiene usar este patrón, es decir, se utiliza cuando:

✓ Una abstracción tiene dos aspectos dependientes. Por ejemplo, cuando existe una parte que captura datos y otra u otras que los procesan. Se encapsulan estos aspectos en objetos separados y permite desacoplarlos de forma que ambos evolucionen separadamente y se puedan reutilizar las partes en aplicaciones diferentes.

✓ Un cambio en un objeto implica un cambio en otros objetos que no es posible conocer a priori, de manera que cada observador puede reaccionar de forma diferente a la misma notificación.

✓ Un objeto debe ser capaz de informar de su estado a otros objetos sin saber cómo son los receptores y sin conocer a priori cuántos son.

Los componentes participantes en la utilización de este patrón son:

✓ Sujeto: Es el origen de los datos, proporcionando métodos para que los observadores se suscriban a la información que produce. Mantiene una lista de observadores y cada vez que sus datos cambian, avisa a sus observadores. También puede proporcionar métodos de acceso a sus datos.

✓ Observador: Es el objeto interesado en los datos del Sujeto, proporcionando métodos para que el Sujeto pueda avisarle (una vez avisado invoa los métodos de acceso del sujeto) y enviarle los datos.

Las ventajas que produce la utilización de este patrón son:

✓ Bajo acoplamiento entre sujeto y observadores. El sujeto no necesita conocer detalles de los observadores. Todo lo que necesita saber es la signatura de sus métodos de actualización.

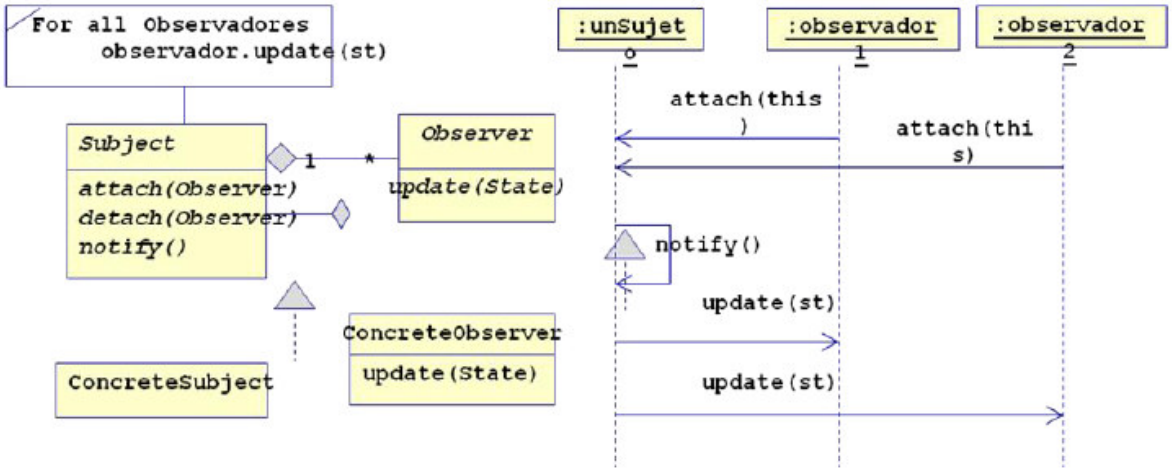
✓ Permite la difusión (broadcast), ya que el número de observadores sólo está limitado por el tamaño de la lista que mantiene el sujeto.

✓ Es posible añadir y eliminar observadores dinámicamente, es decir, en tiempo de ejecución.

Algunos inconvenientes son los siguientes:

✓ Si se permite que los observadores influyan en el estado del sujeto puede provocarse un aluvión de mensajes que sobrecargue el sistema.

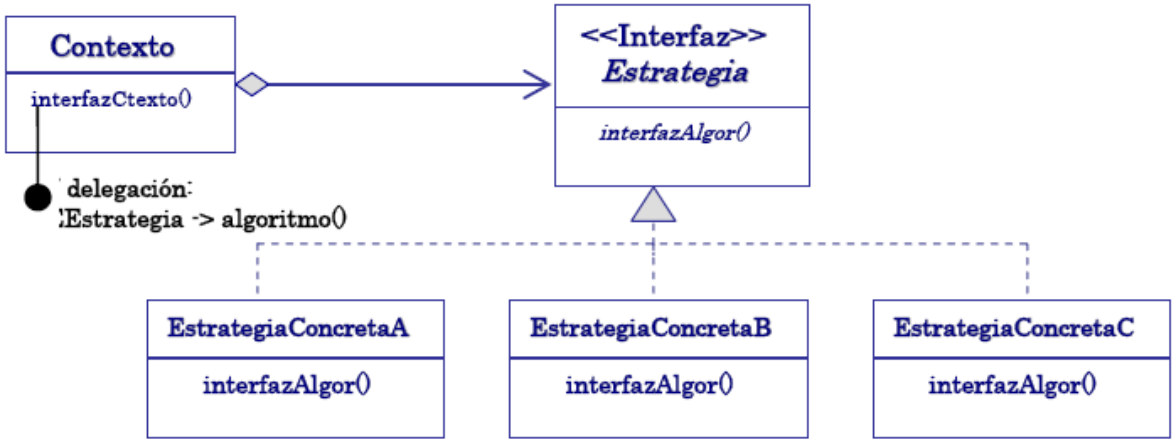
✓ Según la política de notificación que se utilice puede haber problemas de rendimiento y de falta de predecibilidad.



Estructura y colaboración del Patrón Observador.

B.2.3. Estrategia

Este patrón define una familia de algoritmos, los encapsula y los hace intercambiables. Permite que los algoritmos cambien independientemente de los clientes que los usan. La figura muestra las relaciones estructurales al usar este patrón.



Patrón Estrategia. Relaciones estructurales.

A continuación se muestra la aplicabilidad que tiene usar este patrón:

✓ Si existen muchas clases estrechamente relacionadas que sólo difieren en sus comportamientos, aplicar Estrategia permite configurar la clase con uno de entre muchos comportamientos.

✓ Para cuando se necesitan diferentes variantes de un mismo algoritmo.

✓ Para cuando el algoritmo usa datos que el cliente no necesita o no debe conocer.

✓ Si una clase define muchos comportamientos, y estos aparecen como múltiples sentencias condicionales, en lugar de usar tantos condicionales, es mejor utilizar Estrategia.

Los componentes participantes en la utilización de este patrón son:

✓ Estrategia: Define una interfaz común para todos los algoritmos. Contexto usa esta interfaz para llamar a la estrategia concreta que le haga falta.

✓ EstrategiaConcreta: Implementa la interfaz definida en Estrategia.

✓ Contexto:

- Se configura con un objeto EstrategiaConcreta.
- Mantiene una referencia a una EstrategiaConcreta.
- Puede definir una interfaz que deje a Estrategia acceder a sus datos.

Las consecuencias de utilización de este patrón son:

✓ Permite definir familias de algoritmos relacionados. Pueden definirse familias de algoritmos utilizables por diferentes contextos.

✓ Limita el número de clases derivadas.

✓ Puede cambiarse el algoritmo (la estrategia) dinámicamente.

✓ Elimina o reduce el número de sentencias condicionales.

✓ El cliente (contexto) debe conocer las estrategias.

✓ Necesidad de comunicación entre contexto y estrategia.

✓ Incremente el número de objetos.

Anexo C.
Distribución.

C.1. Comunicación entre procesos.

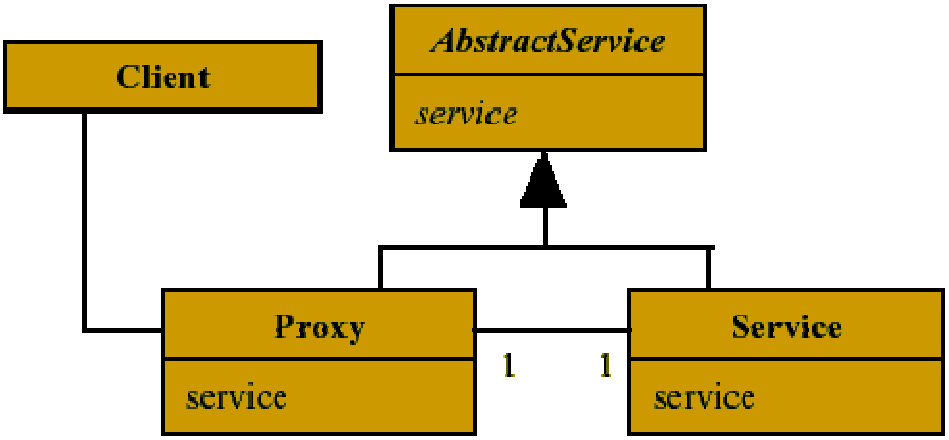
Cuando en un sistema tenemos distintos procesos independientes ejecutándose, necesitamos disponer de mecanismos que hagan posible la comunicación entre ellos. Se pueden utilizar distintos mecanismos de comunicación entre procesos [**IPC**: InterProcess Communication]:

- El **portapapeles de Windows** [Windows Clipboard], que, como almacén central de datos, nos permite que distintas aplicaciones intercambien datos "copiando y pegando".
- **DDE** [Dynamic Data Exchange], un protocolo que permite a las aplicaciones intercambiar datos de una forma más general que el portapapeles.
- **OLE** [Object Linking and Embedding], basado en COM, nos permite manipular documentos compuestos de datos que pueden provenir de distintas aplicaciones.
- **ActiveX** es otra tecnología basada en COM que permite la comunicación entre componentes independientemente del lenguaje en el que estén implementados.
- **COM** [Component Object Model] establece un estándar binario mediante el cual se puede acceder a los servicios de un componente utilizando uno o varios conjuntos de funciones relacionadas (interfaces).
- **DCOM** [Distributed COM] extiende el modelo de programación COM para que distintos componentes puedan comunicarse a través de una red.
- **.NET Remoting**, el mecanismo que sustituye a DCOM en la plataforma .NET.
- **NetBIOS** [Network Basic Input/Output System], un protocolo para redes de área local de PCs creado por IBM en la época del MS-DOS que es similar a Novell Netware (IPX/SPX).
- **Sockets**, para los cuales Windows incluye un API más sofisticado que el API tradicional que proviene del BSD UNIX [Berkeley Software Distribution], el API que se ha utilizado habitualmente para transmitir datos utilizando la familia de protocolos TCP/IP.
- **Pipes anónimos** [Anonymous pipes]: Permiten redireccionar la entrada o salida estándar de un proceso (utilizando | en la línea de comandos, por ejemplo).
- **Pipes con nombre** [Named pipes], similares a las colas FIFO de POSIX [Portable Operating System Interface for Computer Environments. IEEE 1003.1, 1988] pero no compatibles con ellas, permiten a dos procesos intercambiar mensajes utilizando una sección de memoria compartida [pipe].

- **Mailslots**, en Win32 y OS/2: Proporcionan un mecanismo de comunicación entre procesos unidireccional y no fiable que puede ser útil para difundir mensajes cortos a múltiples procesos en un sistema distribuido.
- **WM_COPYDATA**, un mensaje de Windows que se puede utilizar para transmitir datos de un proceso a otro utilizando la infraestructura del propio sistema operativo.
- **Ficheros mapeados en memoria**: Permiten que dos procesos de una misma máquina compartan un fichero que pueden manipular como si fuese un bloque de memoria en su espacio de direcciones, si bien para acceder a él correctamente deberán utilizar algún mecanismo de exclusión mutua (vg: semáforos).
- **Semáforos**, **eventos**, **Mutex** y otras **primitivas de sincronización** que permiten comunicar la ocurrencia de alguna acción o de algún cambio de estado.
- **RPC** [Remote Procedure Call]: Llamadas a procedimientos remotos. Permiten realizar la comunicación entre procesos como si se tratase de llamadas a funciones. El RPC de Windows cumple con el estándar OSF DCE [Open Software Foundation Distributed Computing Environment], lo que permite la comunicación entre procesos que se ejecuten en sistemas operativos diferentes a Windows.
- **CORBA** [Common Object Request Broker Architecture], estándar del OMG [Object Management Group] para el desarrollo de sistemas distribuidos.
- **MPI** [Message Passing Interface]: Estándar de paso de mensajes muy utilizado en clusters y supercomputadores.
- **PVM** [Parallel Virtual Machine]: Otro estándar de paso de mensajes utilizado en multiprocesadores y multicomputadores.
- **Servicios web** [Web services]: Conjunto de estándares que facilitan el paso de mensajes en entornos heterogéneos.

Como se puede ver, disponemos de una amplia variedad de mecanismos de comunicación entre procesos. Algunos de ellos facilitan la división del trabajo entre distintos procesos de una máquina, otros permiten dividir el trabajo entre procesos que se ejecutan en distintos ordenadores de un sistema distribuido.

Independientemente del mecanismo de comunicación entre procesos que decidamos emplear, nuestra aplicación debería acceder a los recursos externos de la misma forma que accede a recursos locales. Para encapsular el acceso a recursos externos se suelen emplear **proxies** o **gateways**:



Además, lo ideal es que intentemos que en nuestra aplicación pueda modificar el mecanismo de comunicación entre procesos utilizado con el menor esfuerzo posible, algo que no siempre facilitan los estándares existentes.

En cualquier caso, lo que siempre debemos tener en cuenta a la hora de desarrollar sistemas distribuidos es que los mecanismos de comunicación no siempre son fiables (algunos paquetes se pierden), la comunicación entre procesos consume tiempo (la latencia no es cero), la capacidad del canal de comunicación no es infinita (el ancho de banda es un recurso muy valioso) y las comunicaciones no siempre se realizan a través de medios seguros.

Normalmente, las aplicaciones que utilizan mecanismos de comunicación entre procesos suelen clasificarse como clientes o servidores, si bien pueden desempeñar ambos roles en distintos momentos. El cliente es el que solicita acceder a algún servicio proporcionado por otro proceso. El servidor es el que atiende las peticiones de los clientes.

C.2. .NET Remoting.

C.2.1. Inicios.

En el desarrollo de sistemas complejos usando técnicas de orientación a objetos, una interfaz simple a nivel de bytes como la de los sockets no resulta del todo apropiada. Por eso existen distintas tecnologías mediante las cuales un objeto puede exponer sus interfaces al público para facilitar su utilización a un nivel de abstracción mayor.

En el modelo **COM** [Component Object Model], todos los objetos han de implementar la interfaz **`IUnknown`** mediante la cual se controla su ciclo de vida (contando las referencias existentes a un objeto se puede saber cuándo puede ser éste eliminado) y se pueden explorar sus características (consultando los interfaces que implementa o, más bien, preguntando si el objeto soporta un interfaz dado):

```
[uuid(00000000-0000-0000-C000-000000000046)]

interface IUnknown
{

        HRESULT QueryInterface (
                [in] const IID iid,
                [out, iid_is(iid)] IUnknown iid );

        unsigned long AddRef();

        unsigned long Release();
}
```

La forma de hacer referencia a un interfaz COM es a través de su identificador **IID** [Interface IDentifier], un número de 128 bits único a nivel global [GUID: Globally Unique IDentifier].

Hay que mencionar que COM es un estándar binario que ni requiere ni impide el uso de orientación a objetos en el diseño de objetos COM. De hecho, COM se limita a la especificación de interfaces y no permite herencia de implementación (lo que puede considerarse algo positivo en el desarrollo de componentes software, donde es preferible usar composición en vez de herencia), si bien sí permite herencia simple de interfaces (aunque también es cierto que esta característica tampoco se usa mucho, ya que se prefieren definir categorías que no son más que conjuntos de interfaces).

Una vez publicado una interfaz con su IID, su especificación no puede modificarse bajo ninguna circunstancia (un componente puede implementar distintas versiones de un interfaz pero éstas se tratan en realidad como interfaces diferentes implementados por el componente, lo que, por otro lado, evita los conflictos de nombres que se producirían en un modelo orientado a objetos convencional).

DCOM [Distributed COM] se limita a ampliar el modelo COM de forma transparente mediante la utilización interna de un mecanismo de comunicación entre procesos basado en el uso de llamadas a procedimientos remotos usando un estándar binario definido por DCOM.

COM/DCOM se utiliza mediante la generación automática de proxies en el cliente y stubs en el servidor que se encargan de encapsular la comunicación entre procesos. De cara al

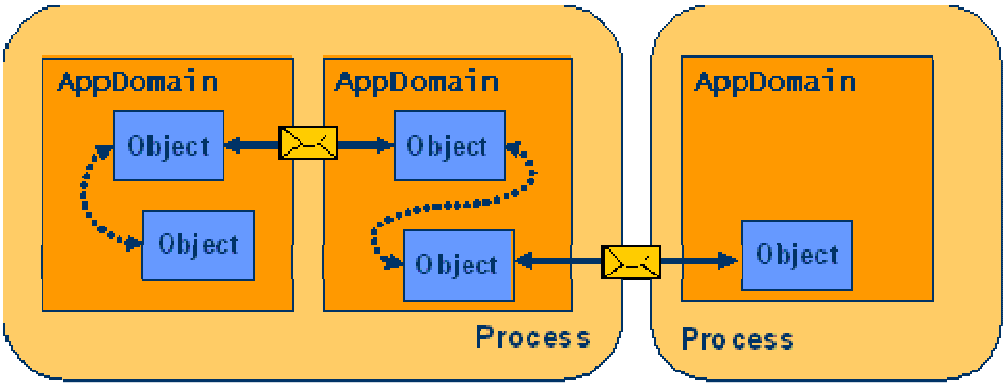
programador, ésta se realiza de forma transparente. Los proxies y los stubs se generan automáticamente a partir de la especificación de los interfaces en MIDL [Microsoft COM Interface Definition Language], un ejemplo de la cual aparece arriba en la definición de **`IUnknown`**.

COM+ es otra extensión de COM que apareció en el año 2000 con Windows 2000 Server. Su primera versión, COM+ 1.0, integra COM con distintas tecnologías, tales como el procesamiento de transacciones (con MTS [Microsoft Transaction Server]) o el envío de mensajes asíncronos (mediante MSMQ [Microsoft Message Queue server]), entre otras. Su segunda versión, COM+ 2.0, es la plataforma .NET.

La principal innovación que supone COM+ (y, por ende, la plataforma .NET) en la evolución de los productos de Microsoft es la introducción de atributos declarativos. Estos atributos permiten separar distintos aspectos en el mismo sentido en que el desarrollo de software orientado a aspectos permite identificar y aislar asuntos compartidos por distintos componentes. Para entender esto de forma intuitiva, digamos que los aspectos sirven para centralizar código que de otra forma aparecería duplicado y esparcido por distintas partes de una aplicación (p.ej. control de acceso, serialización, sincronización, transacciones...).

C.2.2. Dominios de aplicación.

El **CLR** [Common Language Runtime] de la plataforma .NET divide cada proceso en uno o varios dominios de aplicación. Dichos dominios aíslan los objetos que contienen de todos los demás que queden fuera del dominio, de forma que para que dos objetos de distintos dominios se puedan comunicar es necesario utilizar "**marshalling**" (el mecanismo mediante el cual se empaquetan datos para su transmisión, también conocido como serialización):



El CLR se encarga de permitir la realización de llamadas que atraviesen los límites de un dominio, de un proceso y de una máquina. Cuando los objetos están en el mismo dominio,

la llamada es local. En cualquier otro caso, la llamada es remota (de ahí lo de .NET Remoting), incluso cuando los objetos estén en el mismo proceso pero en diferente dominio. Las llamadas locales son inmediatas (como en cualquier invocación a una función en un lenguaje de programación tradicional), mientras que las llamadas remotas involucran el uso de "marshalling" a través de proxies.

El siguiente fragmento de código muestra el nombre del dominio de aplicación actual y de los assemblies que se hallan en él:

```
using System.Reflection;
using System.Runtime.Remoting;
...

AppDomain domain = AppDomain.CurrentDomain;

Console.WriteLine("Dominio actual: " +  domain.FriendlyName);

Assembly[] loadedAssemblies = domain.GetAssemblies();

Console.WriteLine("Assemblies en el dominio actual:");

foreach (Assembly assembly in loadedAssemblies)
Console.WriteLine (assembly.FullName);
```

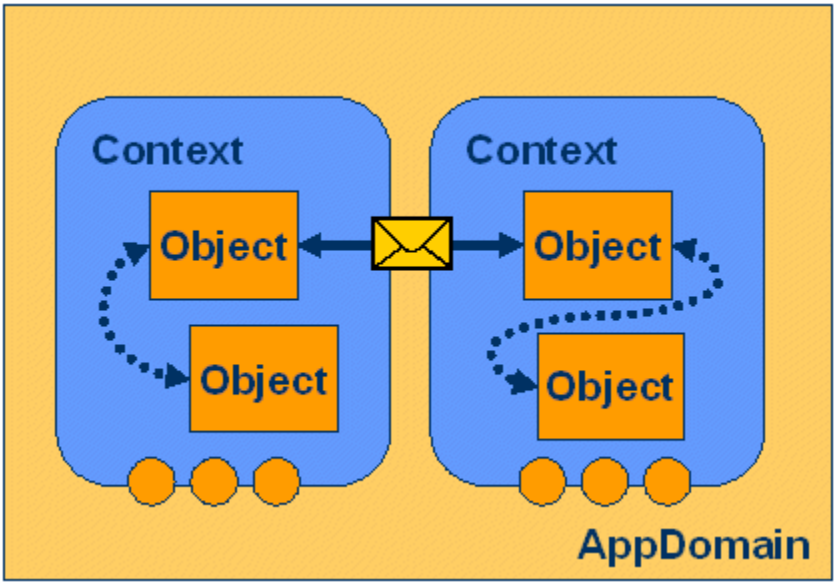
Los dominios de aplicación constituyen unidades aisladas en el CLR. En ellos, una aplicación puede ejecutarse o detenerse sin afectar a las aplicaciones que se ejecutan en otros dominios de aplicación. De hecho, una aplicación no puede acceder directamente a los recursos que se encuentren en un dominio de aplicación distinto del suyo. Gracias a ello, un fallo en una aplicación se puede mantener confinado en los límites de un dominio de aplicación de forma que, aunque distintos dominios de aplicación estén en un mismo proceso, los demás dominios de aplicación no se verán afectados por el fallo.

C.2.2.1. Contextos.

Los contextos definen una partición de los dominios de aplicación, de tal forma que los objetos pertenecientes a un contexto comparten las propiedades de su contexto. Los contextos en COM+ derivan de los apartamentos COM (el mecanismo mediante el cual se controla la sincronización entre hebras en COM) y de los contextos MTS (que separan objetos en función de su "dominio transaccional").

Un objeto puede ser ágil [**context-agile**] o estar ligado a un contexto [**context-bound**], lo cual viene predeterminado si el objeto deriva de la clase **`System.ContextBoundObject`**. Un objeto ágil A puede interactuar con un objeto B ligado a un contexto como si A estuviese en el mismo contexto que B. Esto es, puede llamarse a A desde cualquier contexto o dominio de aplicación libremente. Cualquier llamada que requiera pasar los

límites de un contexto es interceptada y, dependiendo de las propiedades del contexto, es preprocesada, postprocesada o, simplemente, rechazada.

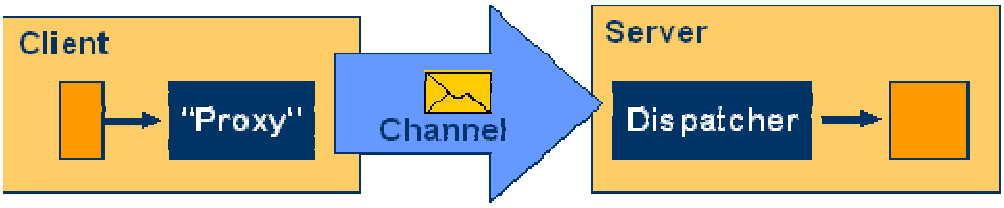


C.2.3. Arquitectura de .NET Remoting

Sobre la infraestructura definida por los contextos y empleando la capacidad de reflexión del **CLI** [Common Language Infraestructure], .NET Remoting (=CLR Object Remoting) proporciona las bases para construir una amplia variedad de estilos de comunicación: desde llamadas síncronas (como DCOM) hasta llamadas completamente asíncronas (con la posibilidad de sondeo y notificación de la terminación de la llamada), ya sea utilizando una codificación binaria sobre un canal TCP o empleando SOAP (en XML y, usualmente, sobre HTTP).

C.2.3.1. Canales

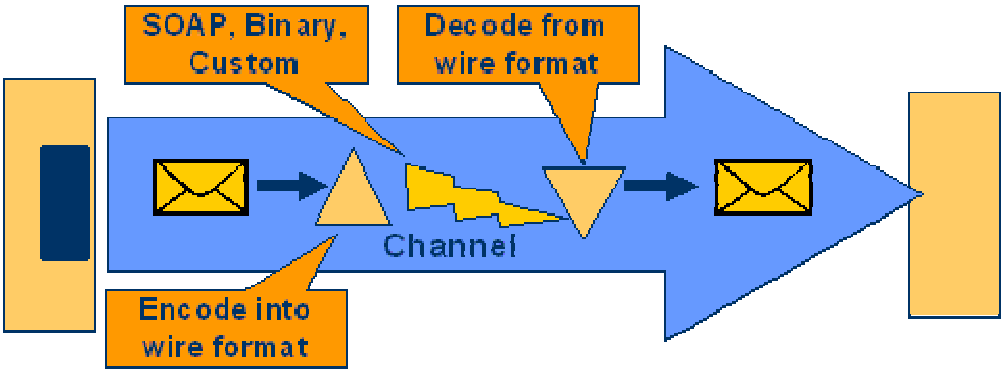
Los canales proporcionan el medio mediante el cual transmitir mensajes de extremo a extremo. Se encargan de transmitir mensajes entre dominios de aplicación, puede que en distintas máquinas conectadas a través de una red de ordenadores, entre distintos procesos concurrentes que se ejecutan en una máquina o, simplemente, entre dominios de aplicación dentro de un proceso. La plataforma .NET lleva incorporadas implementaciones de los canales para realizar la comunicación mediante sockets sobre **TCP** o **HTTP**.



Un canal normal, HTTP o TCP, no incorpora ninguna medida de seguridad a la hora de transmitir datos. Si queremos que la transmisión de datos se realice de forma segura, tendremos que crear nuestro propio canal (creando una clase que implemente los interfaces **`IChannelReceiver`**, **`IChannel`** e **`IChannelSender`**) o alojar nuestros objetos en el IIS, al cual se puede acceder usando HTTPS si lo configuramos adecuadamente.

C.2.3.2. Formateadores.

Los formateadores se encargan de serializar los objetos .NET para que puedan transmitirse a través de los canales de comunicación. La plataforma .NET puede serializar los objetos en binario y en SOAP/XML. El formateador binario es más eficiente, el que emplea XML resulta más cómodo a la hora de integrar sistemas heterogéneos. Además, podemos crear formateadores específicos que se adapten a nuestras necesidades.



La diferencia clave entre .NET Remoting y los Servicios Web se encuentra en la forma en que serializan los datos. Los Servicios Web emplean siempre XML, .NET Remoting puede emplear cualquier formateador que implemente la interfaz **`System.Runtime.Remoting.Messaging.IRemotingFormatter`**, como **`System.Runtime.Serialization.Formatters.Binary.BinaryFormatter`** o **`System.Runtime.Serialization.Formatters.Soap.SoapFormatter`**.

Por defecto, los canales HTTP utilizan el formateador **`SOAP`**, mientras que los canales TCP usan el formateador binario para acceder a objetos remotos, si bien todo es configurable en .NET Remoting. Cuando las llamadas remotas sólo cruzan los límites de un contexto pero

se realizan dentro de un dominio de aplicación, se utiliza un canal especial **`CrossContextChannel`** que está optimizado para trabajar dentro del espacio de memoria del proceso y no requiere formateador alguno.

C.2.3.3. Marshalling.

"Marshalling" es el mecanismo mediante el cual se empaquetan las llamadas entre dominios de aplicación para su transmisión (tanto el paso de parámetros como la devolución de resultados). Se puede acceder de forma remota a un objeto de dos formas diferentes:

- Por valor (**`MarshalByValue`**), haciendo una copia del objeto completo, que se transmite a través del canal perdiendo el enlace entre el original y la copia. El cliente dispondrá localmente de una copia completa del objeto remoto. Esta copia, obviamente, trabajará de forma independiente con respecto a la copia remota del objeto. En otras palabras, en el momento en el que se accede a un objeto remoto por valor, el objeto deja de ser remoto.
- Por referencia (**`MarshalByRef`**), pasando únicamente una referencia al objeto [ObjRef] y creando un "proxy" que sirve de enlace entre el cliente y el objeto remoto. Los objetos remotos siempre residen y se ejecutan en el servidor. El cliente se comunica con el objeto remoto a través del proxy, que sólo tiene una referencia al objeto remoto.

Los objetos ágiles (independientes del contexto) se transmiten por valor si no están ligados a un dominio de aplicación, mientras que se transmiten por referencia si están asociados a un dominio de aplicación y se accede a ellos desde fuera de ese dominio. Los objetos ligados a un contexto siempre se transmiten por referencia fuera de ese contexto.

El acceso a un objeto por valor siempre será más rápido una vez que se dispone de una copia local del objeto, si bien el tiempo que se tarda en obtener inicialmente esa copia puede ser considerable si el objeto es grande. En realidad, lo único que se hace al acceder por valor a un objeto remoto es "descargar" el objeto del servidor y trabajar con él localmente. Por el contrario, al acceder por referencia a un objeto remoto, nuestra aplicación es realmente una aplicación distribuida. Esto puede ser útil cuando los objetos remotos son demasiado grandes (lo que hace prohibitiva su transmisión hasta el cliente) o cuando los objetos remotos residen en un servidor desde el cual se puede acceder a recursos no disponibles directamente desde el cliente.

C.2.3.4. Proxy.

Un "proxy" [apoderado] es un objeto que actúa localmente en nombre de un objeto remoto. Desde el punto de vista del programador, el proxy acepta llamadas como si fuese el objeto real, si bien internamente lo único que hace es delegar en el objeto remoto para ejecutar las llamadas que recibe.

Cuando un cliente quiere acceder remotamente a un objeto del servidor, .NET Remoting crea automáticamente un proxy transparente que hace de servidor en el lado del cliente, de forma que el cliente trabaja con él como si del propio objeto remoto se tratase. El proxy implementa todos los métodos que aparecen en la interfaz del objeto remoto. Las llamadas que recibe las envía al objeto remoto, que es el que verdaderamente se encarga de hacer el trabajo.

C.2.3.5. Dispatcher.

El "dispatcher" se sitúa al otro extremo del canal, recibe los mensajes del proxy, invoca al método real en el objeto remoto, recoge la el resultado y devuelve un mensaje de respuesta.

C.2.4. Uso de .NET Remoting.

El modelo de activación de objetos de la plataforma .NET se parece más al de CORBA que al de COM:

- Si al otro extremo no se está escuchando, no se puede realizar ninguna conexión (en COM, la activación se produce bajo demanda).
- No existe ningún registro a modo de páginas amarillas (como sucede en RMI, por ejemplo).
- Los servidores no se activan remotamente (en este sentido, .NET Remoting se parece más a los sockets TCP que a los componentes COM).

Nota

En los proyectos que utilizan clases del espacio de nombres **`System.Runtime.Remoting`** es necesario agregar una referencia a la DLL **`System.Runtime.Remoting.dll`**.

C.2.4.1. Uso de dominios de aplicación dentro de un proceso.

Comencemos creando un servidor muy simple (`servidor.exe`):

```
using System;

namespace RemotingExample
{
        public class Servidor
        {
                [STAThread]
                static void Main(string[] args)
                {
                        Console.WriteLine ("Servidor cargado y ejecutado");
                }

                // Dominio en el que se ejecuta el servidor

                public string Domain
                {
                        get { return AppDomain.CurrentDomain.ToString(); }
                }
        }
}
```

Ahora intentamos usarlo desde un cliente que creamos como aplicación independiente (`cliente.exe`). Para que esta aplicación funcione correctamente, añadiremos una referencia al proyecto `servidor.exe` (algo que podemos hacer directamente desde el explorador de soluciones del Visual Studio).

Ya en el cliente, creamos un nuevo dominio de aplicación (dentro del mismo proceso) y cargamos en él el assembly `servidor.exe`:

```
using System;
using System.Reflection;
using System.Runtime.Remoting;

namespace RemotingExample
{
        public class Cliente
        {
                [STAThread]
                static void Main(string[] args)
                {
                        AppDomain    newDomain;
                        ObjectHandle o;
                        Servidor     s;

                        // Nuevo dominio

                        AnewDomain = AppDomain.CreateDomain("MiNuevoDominio");
                        newDomain.ExecuteAssembly("Servidor.exe", null, args);

                        // Acceso al servidor...

                        o = newDomain.CreateInstance ( "Servidor", "RemotingExample.Servidor");

                        s = o.Unwrap(); // ... para forzar la instanciación del objeto

                        Console.WriteLine(s + " @ " + s.Domain);
```

```
                    // Fin

                    Console.WriteLine("Pulse ENTER...");
                    Console.ReadLine();
                }
        }
}
```

`CreateInstance` recibe como parámetros el assembly en el que se encuentra nuestro "servidor" y el nombre completo de la clase a la que pertenece nuestro servidor.

Al ejecutar el programa anterior, salta una excepción porque el servidor no es un objeto al que se pueda acceder desde otro dominio de aplicación.

Para que el servidor sea accesible desde otro dominio de aplicación por valor, basta con marcarlo como **`serializable`**. Como para acceder al objeto por valor hay que construir localmente una copia exacta del objeto remoto, el objeto completo ha de transmitirse a través del canal existente entre dominios de aplicación diferentes, para lo cual es imprescindible que el objeto sea **`serializable`**.

```
using System;

namespace RemotingExample
{
        [Serializable]
        public class Servidor
        {
                [STAThread]
                static void Main(string[] args)
                {
                        Console.WriteLine ("Servidor cargado y ejecutado");
                }

                // Dominio en el que se ejecuta el servidor.
                public string Domain
                {
                        get { return AppDomain.CurrentDomain.ToString(); }
                }
        }
}
```

También podríamos controlar explícitamente cómo se serializa un objeto si, en vez de limitarnos a utilizar el atributo **`[Serializable]`**, implementásemos explícitamente la interfaz **`ISerializable`**.

Hacer que al objeto remoto se pueda acceder por referencia es casi tan simple. Basta con hacer que la clase `Servidor` derive de **`System.MarshalByRefObject`**.

```
using System;

namespace RemotingExample
```

```
{
        public class Servidor: System.MarshalByRefObject
        {
                [STAThread]
                static void Main(string[] args)
                {
                        Console.WriteLine ("Servidor cargado y ejecutado");
                }

                // Dominio en el que se ejecuta el servidor

                public string Domain
                {
                        get { return AppDomain.CurrentDomain.ToString(); }
                }
        }
}
```

C.2.4.2. Acceso a objetos remotos con .NET Remoting.

Una vez que ya tenemos los conocimientos básicos necesarios para crear un servidor real al que se acceda remotamente, creamos una biblioteca de clases (`Servidor.dll`) en la que incluimos la funcionalidad que nuestro servidor proporcionará a sus clientes:

```
public class Servidor: System.MarshalByRefObject
{
        public Servidor()
        {
                Console.WriteLine("Constructor");
        }

        ~Servidor()
        {
                Console.WriteLine("Destructor");
        }

        public string GetInfo ()
        {
                return AppDomain.CurrentDomain.FriendlyName;
        }
}
```

El servidor lo tenemos que alojar en un dominio de aplicación, para lo cual creamos una aplicación en modo de consola:

```
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using System.Runtime.Remoting.Channels.Tcp;
...

class Host
{
        static void Main(string[] args)
        {
                ChannelServices.RegisterChannel(
                        new HttpChannel(8888));

                RemotingConfiguration.RegisterWellKnownServiceType(
```

```
                typeof(RemotingExample.Servidor),
                "Servidor",
                WellKnownObjectMode.SingleCall);

            Console.WriteLine(
                "Servidor listo para aceptar mensajes...");
            Console.WriteLine(
                "Pulse INTRO para salir");
            Console.ReadLine();
        }
    }
```

Esta aplicación se encarga de crear y registrar un canal, ya que .NET Remoting exige que, para que los clientes puedan acceder a los objetos remotos, éstos estén ligados a un canal cuyo nombre sea conocido por el cliente. Por tanto, lo primero que hacemos es registrar un canal a través del cual se pueda acceder al objeto de forma remota.

Acto seguido, se registran los tipos de objetos a los que se podrá acceder desde fuera del dominio de aplicación del servidor, así como su URL de acceso. Los tipos registrados serán accesibles desde el exterior mientras que la aplicación que los aloja esté ejecutándose.

También hay que indicar el tipo de activación de los objetos a los que se accede de forma remota. Desde el servidor, sólo se permiten dos tipos de activación en .NET Remoting:

- **`SingleCall`**: Se crea una instancia de la clase para cada llamada realizada a través del canal (comunicación sin estado, como en el protocolo HTTP).
- **`Singleton`**: Existe una única instancia de la clase común para todos los clientes. Este singleton sirve de gateway a la lógica de la aplicación.

Para poder hacer referencia al servidor desde el cliente, obtenemos una referencia al proxy del objeto remoto mediante una llamada al método **`GetObject()`** de la clase **`System.Runtime.Remoting.Activator`**. Una vez que tenemos el proxy, podemos usar el servidor como cualquier otro objeto:

```
// Acceso remoto

ChannelServices.RegisterChannel(new HttpChannel());

Servidor remoto = (Servidor)Activator.GetObject (
                                typeof(RemotingExample.Servidor),
                                "http://localhost:8888/Servidor" );

Console.WriteLine( remoto.GetInfo() );

// Acceso local

Servidor local = new Servidor();

Console.WriteLine( local.GetInfo() );
```

El cliente, como es lógico, debe especificar el canal a través del cual se comunicará con el objeto remoto. Este objeto podría incluso ofrecer sus servicios a través de distintos canales, presumiblemente de distintos tipos. El cliente usará el canal que resulte más adecuado para comunicarse con el objeto remoto.

Los metadatos de la clase `Servidor` necesarios para poder compilar el cliente los podemos obtener del propio assembly (`Servidor.dll`) o de la referencia web **`http://localhost:8888/Servidor?wsdl`**. A partir de esa referencia se puede crear el ensamblado requerido por el cliente sin tener que distribuir el código completo del servidor. Para lograrlo, podemos usar la utilidad **`soapsuds`**:

```
soapsuds -url:http://localhost:8888/Servidor?wsdl
-oa:InterfazServidor.dll
```

Cuando un objeto se registra en un canal para que se pueda acceder a él de forma remota, los clientes podrán acceder a todas las propiedades, métodos y variables públicas no estáticas de la clase a la que corresponda el objeto.

En el ejemplo anterior, el ciclo de vida del objeto remoto se controla en el servidor. Cuando un cliente quiere acceder al objeto remoto, el cliente obtiene un proxy y en el servidor se crea una instancia de la clase del objeto remoto para atender única y exclusivamente a una petición del cliente. El objeto remoto sólo se instancia ("activa", si usamos la terminología de .NET Remoting) cuando el cliente llama a un método del proxy. Esto nos permite ahorrar una llamada inicial a través de la red para, simplemente, crear el objeto.

Si usamos el modo de activación **`SingleCall`**, el objeto remoto se instancia para atender una única petición, tras la cual el objeto se elimina. Por tanto, esos objetos no mantienen su estado entre distintas peticiones provenientes de un mismo cliente. Esto, que a primera vista es una seria limitación, permite construir aplicaciones distribuidas escalables, ya que el objeto sólo consume recursos del servidor temporalmente y, en el caso de usar un cluster en el servidor, la carga se puede distribuir con mayor facilidad (al no importar qué servidor atiende cada petición porque éstas son independientes unas de otras). En definitiva, este modo de activación es útil en aplicaciones que sólo requieren realizar una operación rápida e independiente del resto de la aplicación como puede ser consultar un dato concreto en una base de datos para mostrarlo en la interfaz de usuario.

Por el contrario, en el modo de activación **`Singleton`**, sólo tendremos una instancia del objeto que atenderá todas las peticiones que se reciban de distintos clientes. A diferencia de `SingleCall`, como el objeto existe de forma permanente, puede mantener el estado entre

distintas llamadas (estado que será global para todos los clientes que accedan a él). Por ejemplo, podríamos usar este modo de activación para implementar correctamente un servidor de Chat como un objeto remoto único (en vez de usar direcciones de broadcast).

Un objeto singleton es un objeto único en su tipo que proporciona un punto de acceso global a los servicios implementados por el objeto remoto. Cuando se hace una llamada al objeto remoto, sólo se creará una instancia del objeto si previamente no existe, algo difícil de lograr con DCOM, en el que los objetos remotos siempre se crean desde el cliente y hay que idear mecanismos artificiales que nos permitan comprobar si existe ya un objeto de ese tipo para no crear otro. En el caso de .NET Remoting, se simplifica la creación de "singletons", si bien su implementación seguirá siendo compleja. Los objetos remotos de este tipo tendremos que implementarlos como aplicaciones multihilo para que puedan atender concurrentemente las peticiones de múltiples clientes.

Este tipo de objetos remotos, **SAOs** [Server Activated Objects] (objetos activados por el servidor), proporcionan una funcionalidad limitada porque sólo se pueden instanciar usando constructores por defecto, sin parámetros. Pero no constituyen la única opción que tenemos a nuestra disposición en .NET Remoting, como veremos a continuación

C.2.4.3. Control desde el cliente.

En situaciones como las vistas hasta ahora, el servidor controla cuándo se crea un objeto al que se pueda acceder de forma remota. Como no podría ser menos, dada su flexibilidad, .NET Remoting también permite que el cliente sea el que controle el ciclo de vida de los objetos a los que accede de forma remota, al más puro estilo de COM/DCOM.

Este tipo de objetos, denominados **CAOs** [Client Activated Objects], se instancian en el servidor cuando el cliente lo solicita (no se espera a que se llame a un método, como sucedía con los SAOs). Esto permite utilizar constructores con parámetros si hace falta. La instanciación/activación de un objeto CAO se realiza de la siguiente forma:

- El cliente crea una instancia del objeto en el servidor mediante una solicitud de activación.
- El servidor crea una instancia de la clase (previamente registrada) y devuelve una referencia al objeto recién creado.
- El cliente utiliza esa referencia para crear un proxy mediante el cual pueda comunicarse con el objeto remoto.

Como consecuencia del proceso seguido, la instancia del objeto remoto atenderá únicamente las peticiones provenientes del cliente que creó el objeto (a diferencia de los SAOs Singleton, que son objetos compartidos entre todos los clientes). Por ejemplo, se usaría un objeto CAO cuando un cliente quiere realizar un pedido y la realización del pedido involucra ir pasando por una serie de etapas, para las cuales ha de mantenerse en todo momento el estado del pedido.

La implementación en sí de un objeto CAO no difiere de la de un objeto SAO. Tendremos que crear una clase que herede de **`System.MarshalByRefObject`**. Lo que sí tendremos que cambiar es la aplicación que aloja al objeto:

```
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using System.Runtime.Remoting.Channels.Tcp;
...

class Host
{
        static void Main(string[] args)
        {
                ChannelServices.RegisterChannel(new HttpChannel(8888));

                RemotingConfiguration.ApplicationName = "Servidor";
                RemotingConfiguration.RegisterActivatedServiceType (
                        typeof(RemotingExample.Servidor) );

                Console.WriteLine("Servidor listo para aceptar mensajes...");
                Console.WriteLine("Pulse INTRO para salir");
                Console.ReadLine();
        }
}
```

El cliente, como antes, deberá usar un canal HTTP para acceder al servidor. En esta ocasión no obstante, se ha de utilizar la llamada al método estático **`CreateInstance`** de la clase **`Activator`**. Este método devuelve una referencia a partir de la cual se puede obtener un proxy con el método **`Unwrap`** y dicho proxy lo usaremos para acceder al objeto de forma remota:

```
// Acceso remoto

ChannelServices.RegisterChannel(new HttpChannel());

object[] attrs = { new UrlAttribute("http://localhost:8888/Servidor")};

ObjectHandle handle =  Activator.CreateInstance(
                                "Servidor",  "RemotingExample.Servidor", attrs);

Servidor remoto = (Servidor) handle.Unwrap();

Console.WriteLine( remoto.GetInfo() );
```

En resumen, los CAOs ofrecen la máxima flexibilidad, mientras que los SAOs proporcionan una mayor escalabilidad.

Ciclo de vida de los objetos remotos.

En .NET Remoting, el ciclo de vida de los objetos remotos se controla mediante *leasing*, la realización de "contratos de alquiler", igual que en RMI o Jini en Java.

El *lease* determina el periodo de tiempo que el objeto estará activo en memoria antes de que el CLR lo elimine. En el caso de los objetos activados por el servidor de tipo `SingleCall`, estos sólo existen mientras dure una llamada a un método. En cambio, los SAOs de tipo `Singleton` y los CAOs vivirán en función de sus "contratos de alquiler".

Un objeto remoto puede definir su ciclo de vida redefiniendo el método **`InitializeLifeTimeService()`** de la clase base **`MarshalByRefObject`**. Cuando el objeto se crea, la duración de su contrato de alquiler se fija usando la propiedad **`InitialLeaseTime`** del contrato representado por un objeto que implementa la interfaz **`ILease`**. La duración predeterminada del contrato es de 300 segundos por defecto, aunque eso se puede modificar en la aplicación que aloja el objeto remoto (de forma global para todos los objetos de la aplicación) o en el propio objeto remoto (de forma local a cada objeto):

```
// En la aplicación del servidor.

public class Host
{
        static void Main(string[] args)
        {
                LifetimeServices.LeaseTime = TimeSpan.FromMinutes(1);
                LifetimeServices.RenewOnCallTime = TimeSpan.FromSeconds(30);
                ...
        }
        ...
}

// En el objeto remoto

public class ObjetoRemoto : MarshalByRefObject
{
        ...
        public override Object InitializeLifetimeService()
        {
                ILease lease = (ILease)base.InitializeLifetimeService();

                if (lease.CurrentState == LeaseState.Initial)
                {
                        lease.InitialLeaseTime = TimeSpan.FromMinutes(1);
                        lease.SponsorshipTimeout = TimeSpan.FromMinutes(2);
                        lease.RenewOnCallTime = TimeSpan.FromSeconds(30);
                }

                return lease;
        }
        ...
```

```
    }
```

Cada dominio de aplicación contiene un "gestor de alquileres" [lease manager], una hebra más, que elimina los objetos cuando sus "contratos de alquiler" expiran. El gestor de alquileres examina periódicamente los contratos para ver cuáles han caducado y eliminar los objetos correspondientes (cada 10 segundos).

Cada vez que un "patrocinador" del objeto renueva su contrato de alquiler, dicho contrato se amplía hasta el tiempo establecido por la propiedad **`RenewOnCallTime`** (120 segundos por defecto), siempre y cuando fuese a caducar antes de ese plazo. Cuando el contrato caduca, se espera el tiempo fijado por **`SponsorshipTimeout`** antes de eliminar físicamente el objeto, por si algún patrocinador desea mantener el objeto (de nuevo, 120 segundos por defecto).

El patrocinador del objeto puede ser el cliente que accede a él o cualquier otro objeto interesado en mantener al objeto remoto. Para renovar el contrato, el patrocinador debe llamar al método **`Renew()`** del contrato asociado al objeto remoto, el cual se obtiene a través de una llamada al método **`GetLifetimeService()`**. El patrocinador, además, recibirá una notificación cada vez que el lease vaya a caducar, para lo cual ha de implementar la interfaz **`ISponsor`** y registrarse como patrocinador del objeto:

```
remoto.GetLifeTimeService().Register ( patrocinador );
```

No obstante, el contrato se renueva automáticamente cada vez que se accede al objeto, por lo que usualmente no tendremos que preocuparnos por este asunto. Si el contrato llegase a caducar y el objeto remoto fuese eliminado, la llamada al método remoto generaría una excepción (en el caso de los objetos CAO) o volvería a instanciar otro objeto (en el caso de los objetos SAO Singleton, que son únicos en cualquier momento pero no siempre son los mismos).

C.2.4.4. Ficheros de configuración.

En los ejemplos vistos hasta ahora, el cliente debe conocer el tipo concreto del objeto remoto que activa. Eso implica que cualquier cambio en la configuración del servidor requiere recompilar todos los clientes.

Afortunadamente, esto no es necesario porque se pueden usar ficheros de configuración XML de forma que en el código no tengamos que saber de qué tipo concreto son los

objetos a los que se acceden (aunque siempre deberemos ser conscientes de si el objeto remoto es un SAO SingleCall, un SAO Singleton o un CAO).

En el servidor, una vez que tengamos el fichero de configuración, podemos registrar los canales oportunos y establecer los objetos a los que se puede acceder de forma remota con una simple llamada al método **`Configure`** de la clase **`RemotingConfiguration`**:

```
RemotingConfiguration.Configure ("Servidor.config");
```

donde **`Servidor.config`** es el fichero XML con todos los datos necesarios para la configuración del servidor.

Como es lógico, en el cliente también podemos usar un fichero de configuración análogo y no tener que especificar los datos relativos a la configuración del sistema (a costa de que un usuario malintencionado pueda acceder fácilmente a esos datos, disponibles en XML). Además, en el caso del cliente, ya no tenemos que usar **`Activator`** para instanciar un objeto remoto. Bastará con crear un objeto con **`new`** y la plataforma .NET se encargará automáticamente de instanciar correctamente el objeto, independientemente de si es un objeto local o un objeto remoto:

```
RemotingConfiguration.Configure ("Cliente.config");

Servidor remoto = new Servidor();
```

Ejemplo

Como ejemplo del uso de ficheros de configuración en .NET Remoting, crearemos un sencillo servidor al que se accederá por referencia:

```
using System;
using System.Net;
using System.Runtime.Remoting;

namespace RemotingExample
{

        public class Servidor: System.MarshalByRefObject
        {
                public Servidor()
                {
                        Console.WriteLine("Constructor");
                }

                ~Servidor()
                {
                        Console.WriteLine("Destructor");
                }

                public string GetHost ()
                {
                        return Dns.GetHostName();
```

```
                }

                public string GetApplication ()
                {
                        return RemotingConfiguration.ApplicationName;
                }

                public string GetAppDomain ()
                {
                        return AppDomain.CurrentDomain.FriendlyName;
                }
        }
    }
```

Este servidor lo alojaremos en un proceso al que se accederá remotamente (`Host.exe`):

```
using System;
using System.Runtime.Remoting;

namespace RemotingExample
{
        class Host
        {

                static void Main(string[] args)
                {

                        RemotingConfiguration.Configure("Servidor.config");

                        Console.WriteLine("Servidor listo para aceptar mensajes...");
                        Console.WriteLine("Pulse INTRO para salir");
                        Console.ReadLine();
                }
        }
}
```

El fichero de configuración **`Servidor.config`** especifica el canal que se utilizará para acceder al servidor (un canal **TCP** en el puerto **7777**) y el modo de activación del servidor (**`SingleCall`**):

`Servidor.config`

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.runtime.remoting>
    <application name="RemotingExample">

      <service>
        <wellknown mode="SingleCall"
          type="RemotingExample.Servidor, Servidor"
          objectUri="RemotingExample.rem" />
      </service>

      <channels>
        <channel port="7777" ref="tcp" />
      </channels>

    </application>
  </system.runtime.remoting>
</configuration>
```

De forma análoga, para el cliente también necesitamos otro fichero de configuración en formato XML:

Cliente.config

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.runtime.remoting>
    <application>

       <client>
         <wellknown type="RemotingExample.Servidor, Servidor"
                    url="tcp://localhost:7777/RemotingExample/RemotingExample.rem" />
       </client>

       <channels>
         <channel ref="tcp" />
       </channels>

     </application>
   </system.runtime.remoting>
</configuration>
```

A partir de este fichero, el acceso remoto al servidor desde el cliente resulta trivial:

```
using System;
using System.Runtime.Remoting;

namespace RemotingExample
{
        class Cliente
        {
                [STAThread]
                static void Main(string[] args)
                {
                        // Acceso remoto

                        RemotingConfiguration.Configure( "Cliente.config" );

                        Servidor remoto = new Servidor();

                        Console.WriteLine( remoto.GetAppDomain() );
                        Console.WriteLine( remoto.GetApplication() );
                        Console.WriteLine( remoto.GetHost() );

                        Console.ReadLine();
                }
        }
}
```

NOTA: Para asegurarnos de que los ficheros de configuración se distribuyen junto con nuestros ejecutables, tenemos que comprobar que la propiedad "***Copiar en el directorio de resultados***" de nuestros ficheros de configuración `.config` esté activada.

.NET Remoting también en IIS

Usando ficheros de configuración en XML, si nuestros objetos son de tipo SAO y utilizamos canales **HTTP**, podemos utilizar el Internet Information Server para alojar nuestros objetos remotos, sin necesidad de crear una aplicación que haga de servidor. Si alojamos la aplicación en un directorio virtual llamado **`MiServicio`** dentro del IIS, se podrá acceder al objeto remoto a través de la siguiente URL:

```
http://localhost/MiServicio/Servidor.soap
```