Capítulo

5

Espacios de Nombres y Librerías

Los Espacios de Nombres

Cuando conversamos con un grupo reducido de amigos es común que utilicemos sólo sus nombres, o apodos, para referirnos a cada uno de ellos. En estos casos es poco común que dos amigos tengan el mismo nombre, pero cuando esto ocurre utilizamos modificaciones de los nombres para aclarar a quién nos estamos refiriendo. Sin embargo, cuando el grupo al que nos dirigimos crece, el número de personas con el mismo nombre tiende a aumentar y con ello la dificultad de utilizar nombres cortos. En un entorno más formal se tiende a utilizar los nombres y los apellidos para aclarar a quién nos referimos en un momento dado. Aún así, siempre existe la posibilidad de encontrar homónimos.

Un problema similar ocurre cuando programamos. Por ejemplo, si dentro de un método definimos una variable con el mismo nombre que otra definida en ámbito de clase, la primera oculta a la segunda. Por tanto, cuando deseamos referirnos a la segunda requerimos utilizar un discriminante, esto es, una sintaxis especial que permita indicarle al compilador a quién nos queremos referir. Desde el punto de vista de los nombres que se le dan a los tipos de datos ocurre lo mismo, sobre todo si utilizamos librerías implementadas por terceros.

Este problema es atacado mediante la definición de espacios de nombres. Un espacio de nombres es un ámbito donde es posible declarar elementos de programación, como variables, métodos y tipos de datos, y dentro del cual se utilizan nombres únicos para cada uno de estos elementos.

Los ámbitos de una función, un método y una clase definen implícitamente un espacio de nombres. Pero es posible definir espacios de nombres explícitos adicionales que permitan organizar nuestros elementos de programación en una estructura jerárquica donde cada elemento tenga un nombre único.

Definición de un Espacio de Nombres

C++ y C# permiten definir espacios de nombres explícitos mediante la palabra reservada “namespace”. La sintaxis básica en ambos lenguajes es:

```
namespace <nombre> {
    <declaración de elementos de programación>
}
```

Java por su lado, une el concepto de espacio de nombres con el de librería, por lo que los detalles acerca de su definición los veremos más adelante, en la sección de librerías.

El siguiente ejemplo corresponde al uso de un espacio de nombres en C++:

```
namespace general {
    int a, b;
    void fn() {
        a = 20;
        b = a * 2;
    }
}
void main() {
    general::a = 10;
    general::b = general::a;
}
```

Nótese que dentro del namespace "general" es posible acceder directamente a todos los elementos declarados en él, esto es, utilizar su nombre corto. Fuera del namespace "general" requerimos utilizar su nombre completo o largo. El nombre del namespace forma parte del nombre de los elementos declarados dentro de él.

El siguiente ejemplo corresponde al uso de un espacio de nombres en C#:

```
namespace General {
    class A {}
    class B {}
}

public class PruebaDeNamespace {
     public static void Main() {
        General.A objA = new General.A();
        General.A objB = new General.A();
     }
}
```

Nótese que mientras en C++ se pueden definir variables, funciones o tipos de datos en un espacio de nombres en C++, en C# **sólo** pueden definirse tipos de datos y como se verá más adelante, otros espacios de nombres.

Anidamiento de Espacios de Nombres

Ahora bien, los elementos no declarados dentro de un namespace explícito, como la función "main" del ejemplo anterior, forman parte del namespace global, el cuál es anónimo. Más aún, el namespace "general" forma parte de éste, lo que implica que los namespaces pueden anidarse. El siguiente ejemplo en C++ muestra la anidación de espacios de nombres y cómo hacer referencia a los elementos del namespace global de manera explícita.

```
#include <iostream.h>

namespace Espacio1 {
    namespace Espacio2 {
        const int var = 10;
    }
    int var = Espacio2::var * 2;
}
float var = Espacio1::Espacio2::var + Espacio1::var;
namespace Espacio3 {
    double var = ::var * 2;
}

void main() {
    int var = (int)Espacio3::var + 100;
    // imprimimos todas las variables "var"
    cout << "Espacio1::Espacio2::var = " << Espacio1::Espacio2::var << endl;
    cout << "Espacio1::var = " << Espacio1::var << endl;
    cout << "::var = " << ::var << endl;
    cout << "Espacio3::var = " << Espacio3::var << endl;
    cout << "var = " << var << endl;
}
```

Nótese cómo, conforme se sale del namespace "Espacio1", se requiere utilizar de un nombre cada vez más largo para acceder a sus elementos desde fuera de él. Nótese además como los nombres de los elementos son cada vez más largos conforme la anidación de los namespace crece en profundidad.

El siguiente ejemplo corresponde al uso del anidamiento de espacios de nombres en C#:

```
using System;
namespace Espacio1 {
    namespace Espacio2 {
        class A {}
    }
    class A {}
}
class A {}
public class PruebaDeNamespace {
     public static void Main() {
        Espacio1.Espacio2.A obj1 = new Espacio1.Espacio2.A();
        Espacio1.A obj2 = new Espacio1.A();
        A obj3 = new A();
     }
}
```

Nótese que en el espacio de nombres global están definidas dos clases, A y PruebaDeNamespace, y un espacio de nombres, Espacio1.

Publicación de un Espacio de Nombres

Para hacer referencia a un elemento de un namespace desde fuera del mismo, utilizando su nombre corto, se utiliza la palabra reservada **"using"**. Esta palabra permite hacer públicos los elementos declarados dentro de un namespace, en otro namespace. Se pueden publicar elementos individuales o todos los elementos a la vez. El siguiente ejemplo en C++ muestra el uso de "using".

```
#include <iostream.h>

namespace Espacio1 {
    int var1 = 10;
    int var2 = 20;
}
namespace Espacio2 {
    int var3 = 30;
    int var4 = 40;
}

// Se publica todo el namespace Espacio1 en el espacio de nombres global
using namespace Espacio1;
// Se publica solo var3 de Espacio2 en el espacio de nombres global
using Espacio2::var3;

void main() {
    cout << "var1 = " << var1 << endl;
    cout << "var2 = " << var2 << endl;
    cout << "var3 = " << var3 << endl;

    cout << "var4 = " << var4 << endl; // ERROR: variable no reconocida en el
    presente espacio de nombres
    cout << "Espacio2::var4 = " << Espacio2::var4 << endl; // Corrección

}
```

El siguiente ejemplo en C# muestra el uso de "using".

```
using System;
using Espacio1.Espacio2;

namespace Espacio1 {
    namespace Espacio2 {
        class A {}
```

```
        class B {}
    }
}

public class PruebaDeNamespace {
    public static void Main() {
        A obj1 = new A();
        B obj2 = new B();
    }
}
```

En el caso de C#, la directiva "using" **sólo** puede utilizarse antes de la declaración de cualquier espacio de nombres. Note que "System" es realmente un espacio de nombres, no una librería. Dentro de System están definidos todos los tipos de datos primitivos, tipos para el manejo de la entrada y salida estándar como la clase Console, entre otros tipos y espacios de nombres que forman la librería estándar de .NET.

Uso de un Alias

También se puede declarar un alias para hacer referencia a un namespace dentro de otro namespace. El siguiente programa en C++ muestra un ejemplo de esto.

```
#include <iostream.h>

namespace Espacio {
    const double var = 3.1416;
}

namespace EspacioX = Espacio; // EspacioX es el nuevo alias de Espacio

void main() {
    cout << "EspacioX::var = " << EspacioX::var << endl;
}
```

El siguiente programa en C# muestra un ejemplo de ésto.

```
using System;
using UnAlias = Espacio1.Espacio2;

namespace Espacio1 {
    namespace Espacio2 {
        class A {}
        class B {}
    }
}

public class PruebaDeNamespace {
    public static void Main() {
        UnAlias.A obj1 = new UnAlias.A();
            UnAlias.B obj2 = new UnAlias.B();
    }
}
```

Definición por Partes

Los espacios de nombres no requieren ser declarados en un solo bloque, éstos pueden declararse en varios bloques independientes, incluso en bloques en archivos y librerías distintas. El siguiente programa en C++ muestra esta definición por partes.

```
#include <iostream.h>

namespace Espacio {
    int var1 = 10;
}
namespace Espacio {
    int var2 = 20;
}
```

```
int main(void) {
    cout << "Espacio::var1 = " << Espacio::var1 << endl;
    cout << "Espacio::var2 = " << Espacio::var2 << endl;
    cin.ignore();
    return 0;
}
```

Se dice que el segundo bloque namespace "Espacio" extiende el espacio de nombres definido por el primer bloque namespace "Espacio". Nótese que "var1" y "var2" pertenecen al mismo espacio de nombres, aún cuando son declarados en bloques distintos. Los bloques pudieron incluso haberse colocado en archivos distintos. Si esto fuese así, el compilador iría completando el espacio de nombres conforme fuera compilando los archivos fuente de un proyecto.

El siguiente programa en C# muestra esta definición por partes.

```
namespace Espacio1 {
    namespace Espacio2 {
        class A {}
    }
}
namespace Espacio1.Espacio2 {
    class B {}
}

public class PruebaDeNamespace {
     public static void Main() {
        Espacio1.Espacio2.A obj1 = new Espacio1.Espacio2.A();
        Espacio1.Espacio2.B obj2 = new Espacio1.Espacio2.B();
     }
}
```

En el ejemplo anterior, las clases "A" y "B" forman parte del espacio de nombres "Espacio1.Espacio2".

Las Librerías

En el contexto de los lenguajes de programación, una librería es una unidad de agrupación de los elementos de programación en una forma tal que puede ser distribuido para su reutilización desde otros programas. Los elementos del lenguaje que pueden formar parte de una librería, así como la forma de crearla y utilizarla dependen de cada lenguaje.

Las siguientes secciones describen el enfoque utilizado y la creación de librerías en C/C++, Java y C#.

Librerías en C/C++

En C/C++ se definen dos tipos de librerías:

- Las librerías estáticas.
- Las librerías dinámicas.

Las librerías **estáticas** (comúnmente, con extensión LIB) son archivos con código máquina ya enlazado (similar a los archivos OBJ, solo que estos últimos no están enlazados) que se utilizan durante el proceso de enlace de archivos OBJ's para formar un programa final, ya sea un ejecutable u otra librería. En este sentido, utilizar librerías estáticas significa tener los compilados de un conjunto de archivos fuente, C y CPP, con lo que se ahorra tiempo al momento de compilar un programa que las utilice, dado que para dichos archivos ya no se requiere pasar por un proceso de compilación. Si una librería es extensa, este tiempo de compilación ahorrado puede ser significativo.

Las librerías **dinámicas** (comúnmente, con extensión DLL, abreviatura de Dinamic Link Library) son archivos con código máquina ya enlazado y datos que permiten a otro programa, en tiempo de ejecución, obtener la ubicación de sus funciones, para llamarlas. Los programas que hacen uso de estas librerías deben seguir un proceso, asistido por el sistema operativo, para cargar dicho código a memoria, buscar la ubicación en memoria de las funciones y llamar a éstas. La DLL no forma parte del archivo del programa que las utiliza. Si dos o más programas en ejecución solicitan una DLL, ésta se carga una sola vez a memoria, en la primera solicitud, con lo que se ahorra espacio de memoria.

Dado que las librerías no se ejecutan directamente sino a través de otros programas que llaman a sus funciones, no requieren implementar una función de entrada, como una función main o WinMain.

Cuando una función de una librería es declarada de manera tal que ésta pueda ser llamada desde otro programa, se dice que dicha función es exportada por la librería. Sólo las funciones exportadas por una librería pueden ser accedidas desde un programa que utilice dicha librería.

Las librerías estáticas son más sencillas de utilizar que las dinámicas. En contraparte, las librerías dinámicas permiten evitar la duplicidad de código. En consecuencia, se suele utilizar librerías estáticas cuando:

- Estas son significativamente pequeñas respecto a la cantidad de memoria disponible en las computadoras donde correrán los programas que las usen.
- Son pocos los programas, instalados en una misma computadora, que las utilizan.

En el resto de casos, se prefiere utilizar librerías dinámicas.

Las librerías pueden ser utilizadas tanto por programas ejecutables, como por otras librerías. Una librería puede contener, además de las funciones exportadas y no-exportadas, recursos como imágenes, audio, video, textos, etc.

Librerías Estáticas

La creación y manejo de una librería estática es sencillo:

- Se compila como librería estática los archivos fuente que la conforman. Si se está trabajando con un IDE, comúnmente se deberá crear el proyecto del tipo “para creación de una librería estática”. Por ejemplo, en Microsoft Visual C++ 6.0, el tipo de proyecto es “Win32 Static Library”.
- El proceso de compilación generará el archivo de la librería, con extensión LIB.
- Se incluye el archivo LIB dentro del proyecto del programa que utilizará esta librería. Si se está utilizando un IDE, comúnmente basta con agregar el LIB como un archivo más del proyecto, al igual que los archivos fuente.

Librerías Dinámicas

Las DLL son librerías a las que los programas acceden en tiempo de ejecución. Dado que estas librerías no forman parte de dichos programas, tampoco son cargadas a memoria automáticamente al ejecutarse éstos. Por ello, si un programa requiere ejecutar una función exportada por una DLL, deberá solicitar al sistema operativo que cargue el archivo DLL a memoria, averigüe la dirección de la función exportada de interés y llamarla. Al proceso de averiguar la ubicación de una función para luego llamarla, se le conoce como **enlace dinámico**, de allí el nombre de este tipo de librería.

Las DLL no tienen un punto de entrada para un hilo primario, como sucede con los archivos ejecutables (función main o WinMain), debido a que el sistema operativo no crea un hilo de ejecución para ellas. Son los hilos de los procesos que hacen uso de una librería, los que ejecutan el código de ésta.

Windows está formado en gran parte por librerías dinámicas, desde donde se comparte la funcionalidad que otras aplicaciones necesitan importar para interactuar con el sistema operativo. Como ejemplo tenemos algunas de las DLL típicamente utilizados por las aplicaciones de Windows:

- KRNL386.EXE { Nótese que es un ejecutable}
- GDI.EXE
- USER.EXE
- KEYBOARD.DRV
- SOUND.DRV
- Winmm.dll
- Msvcrt40.dll, Msvcrt20.dll, Msvcrt.dll

Aunque es un uso poco frecuente, un archivo ejecutable también puede ser utilizado como una DLL, siempre que éste contenga funciones exportadas y datos que permitan a otros programas ubicarlas. Más adelante veremos cuáles son estos datos y cómo se crean.

Las DLL y los procesos que las llaman se cargan a memoria en tiempos y lugares distintos, por lo que originalmente tienen espacios de direccionamiento distintos. Sin embargo, el sistema operativo "mapea" las llamadas a las funciones de las DLL's dentro de los espacios de direccionamiento de los procesos de los hilos llamadores. Esto significa que la dirección de la función exportada obtenida por el hilo llamador corresponde a un valor dentro del espacio de direccionamiento de su proceso, pero cuando es utilizada para llamar a la función, el sistema operativo "mapea" dicha dirección de forma que se acceda a la ubicación real de dicha función y se pueda ejecutar su código. Por tanto, nunca se rompe la regla de que "los hilos de un proceso no pueden acceder a direcciones en memoria fuera de su espacio de direccionamiento".

Estructura Interna

Cuando un conjunto de archivos fuente es compilado y enlazado como una DLL, al archivo resultante se le agrega al inicio, una tabla de exportación. Esta tabla contiene los datos que permiten a los programas que hacen uso de una DLL, obtener la ubicación de una función exportada. La estructura de esta tabla es, de manera simplificada, la siguiente:

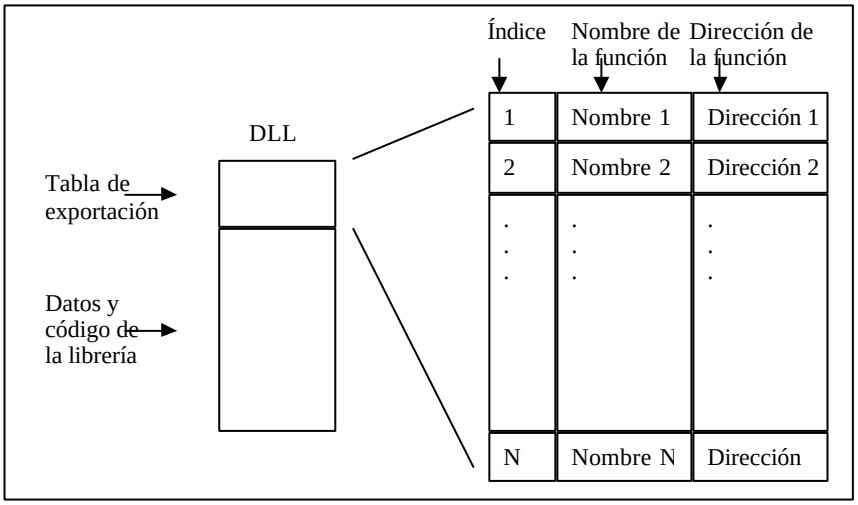

Figura 5.1. Estructura Interna de una Librería

Cuando un programa carga a memoria una DLL, mediante funciones del API de Windows, obtiene un handle a dicha librería. Mediante este handle el programa puede obtener la dirección mapeada de cualquiera de las funciones exportadas por la DLL, igualmente, mediante funciones del API de Windows.

Estas funciones del API de Windows realizan una búsqueda sobre la tabla de exportación, según los parámetros que se le pasen, bien por el nombre de la función o directamente utilizando un índice. Si se encuentra la función, se retorna su dirección mapeada. La búsqueda por nombre es más lenta que utilizando un índice, pero ofrece la ventaja de asegurar al programa usuario que la dirección devuelta corresponde a la función correcta. Como contraparte, la búsqueda por índice es más rápida pero, dado que la posición en la tabla de exportación para los datos de una función exportada puede variar entre versiones de una misma DLL, la dirección retornada puede no ser de la función buscada.

Todas las funciones dentro de una DLL que son declaradas para exportación, estarán incluidas en esta tabla. El resto de funciones son privadas de la librería, pero pueden ser llamadas desde las funciones exportadas.

Espacio de Direccionamiento

Cada proceso activo en el sistema tiene un espacio de direccionamiento virtual privado. El espacio de direccionamiento es un rango de direcciones virtuales, lo cual significa que dos procesos podrían tener punteros con el mismo valor pero estar realmente apuntando a lugares de memoria diferentes. La dirección a la que realmente se apunta es determinada por el sistema operativo mediante tablas de direccionamiento. Este esquema de trabajo permite al sistema operativo sacar de memoria (copiando sus datos a disco) procesos, o parte de ellos, que no se estén ejecutando para colocar, en la misma ubicación de memoria real, otros procesos que deban ejecutarse. De esta forma, el sistema puede simular que se está trabajando con una memoria mucho mayor de la que realmente existe. A la técnica de bajar y subir a memoria bloques de datos de procesos se le llama **SWAPING**. Al archivo en disco que se utiliza para esto se le llama archivo de SWAP.

En conclusión, el sistema puede trabajar con tantos procesos a la vez como espacio de memoria tenga sumando la memoria RAM más el espacio disponible en disco para el archivo de SWAP.

La técnica de mapeado de las direcciones de las funciones exportadas por las DLL's cargadas a memoria, permite no violar la regla de que los hilos de un proceso no pueden acceder a direcciones fuera del espacio de direccionamiento de su proceso.

Creación de una DLL

Se deben seguir los siguientes pasos:

1. Se crea un nuevo proyecto del tipo "quiero crear una DLL". Por ejemplo, en Microsoft Visual C++ 6.0, el tipo del proyecto es "Win32 Dynamic-Link Library".

2. Se agrega los archivos y el código necesario.

3. Se compila y se genera el archivo de la librería con extensión DLL y un archivo para enlace en modo implícito con extensión LIB. El uso de éste último se verá más adelante.

Una DLL comúnmente contiene los siguientes archivos:

1. Un archivo **cabecera** (*.H) donde se declaran las funciones exportadas.

2. Los archivos **fuentes** (*.C o *.CPP) y otros archivos de cabecera. Una de las fuentes deberá implementar la función DllMain. Las fuentes deberán implementar las funciones exportadas, junto con el resto de funciones utilizadas por éstas.

3. Un archivo de **definición** (*.DEF) donde se declaren qué funciones serán las que se exporten.

Como ejemplo, se explicarán los archivos correspondientes a una DLL que sólo exporte una función. Llamaremos a esta DLL "MiLibreria.DLL".

EL ARCHIVO CABECERA MILIBRERIA.H

Contiene los prototipos de las funciones a exportar. Los prototipos deben especificar además el tipo de convención de llamada que se usará. Cada convención de llamada provoca una decoración particular del nombre de cada función, de forma que quién llame a dicha función sepa distinguir a qué convención se refiere. Nosotros podemos dejar que el compilador coloque dichos adornos internamente (al momento de compilar) por nosotros o colocar dichas decoraciones manualmente.

La convención de llamada de una función determina la forma en que el código máquina que se genere, al momento de compilar las fuentes, realice la llamada a dichas funciones.

Si una función exportada por una DLL tiene una determinada convención de llamada, el programa que utilice dicha función deberá declararla con la misma convención de llamada. Luego, la manera más sencilla de salvar estos problemas es dejar que el compilador decida la convención de llamada por nosotros, tanto cuando se compila el DLL como el programa que lo utilizará.

Sin embargo, el compilador de C++ agrega adornos adicionales al nombre de las funciones exportadas, lo que no realiza el compilador de C. La manera más sencilla de solucionar este problema, si el programa y la DLL son compilados con compiladores distintos, es forzar a que las funciones de exportación sean declaradas de una misma forma, por ejemplo, de C. Para hacer esto, se debe declarar los prototipos de estas funciones, tanto en el programa como en el DLL, dentro de la sentencia:

```
extern "C" {
    // Aquí se colocan los prototipos de las funciones exportadas
}
```

Todo lo que esté dentro de los corchetes se compila con el compilador de C. Si el fichero es agregado a un archivo *.CPP, este código se compila igualmente como C, y el resto del archivo fuente se compila con el compilador de C++.

Como ejemplo, se desea declarar el archivo cabecera de la DLL para exportar una función que muestre un mensaje de saludo. El archivo contendrá:

```
//////////////////////////
// Archivo MiLibreria.h
//////////////////////////

#include <windows.h>

#ifdef __cplusplus
  extern "C" {
#endif

void WINAPI Saludame(char * szNombre);

#ifdef __cplusplus
  }
#endif
```

Las directivas #ifdef y #endif permiten que el código entre ellas sea tomado en cuenta por el compilador únicamente si éste es el compilador de C++. La macro WINAPI permite especificar la convención de llamada de la función a la del estándar utilizado por las librerías dinámicas de Windows.

EL ARCHIVO FUENTE MILIBRERIA.CPP

Contiene la implementación de las funciones cuyos prototipos hemos declarado en el archivo cabecera. Adicionalmente, este archivo debe contener la definición de la función DllMain. Esta función es llamada cada vez que el sistema operativo recibe una solicitud de carga o descarga de dicha DLL por parte de algún proceso. Esta función no es llamada nunca directamente por el proceso. Su prototipo y estructura típica es:

```
BOOL WINAPI DllMain ( HANDLE hModule, DWORD dwReason, LPVOID lpReserved ) {
  switch ( dwReason ) {
    case DLL_PROCESS_ATTACH:
      // Código ejecutado la primera vez que un hilo de un proceso carga la DLL
      break;
    case DLL_THREAD_ATTACH:
      // Código ejecutado las siguientes veces que un hilo de un proceso carga la
DLL
      break;
    case DLL_THREAD_DETACH:
      // Código ejecutado cuando un hilo de un proceso descarga la DLL
      break;
    case DLL_PROCESS_DETACH:
      // Código ejecutado cuando el último hilo de un proceso descarga la DLL
      // En este caso, al retornar de esta función el sistema descarga de memoria la
DLL
      break;
  }
  return TRUE;
}
```

Donde:

- hModule : Es el handle para la instancia del DLL cargada en memoria.
- dwReason : Es la razón por la que se llama a la función.
- LPVOID lpReserved : Esta reservado para uso del sistema.

Si la función no se define, el entorno de Visual C++ agrega a nuestro código compilado la declaración de una función DllMain por defecto. Si la función devuelve FALSE significa que la carga / descarga falló, por lo que la función, en el hilo del proceso desde dónde se llamó a la carga / descarga de la librería, recibirá un valor de error como resultado.

El resto del archivo fuente deberá contener la implementación de las funciones exportadas así como otras de uso interno. Para nuestro ejemplo, el código sería:

```
//////////////////////////
// Archivo MiLibreria.cpp
//////////////////////////

#include <windows.h>

BOOL WINAPI DllMain ( HANDLE hModule, DWORD dwReason, LPVOID lpReserved ) {
   // Aquí va lo indicado arriba
}

#include "MiLibreria.h"

void WINAPI Saludame(char * szNombre) {
   MessageBox(NULL, szNombre, "Hola", MB_OK);
}
```

EL ARCHIVO DE DEFINICIÓN MILIBRERIA.DEF

Existen 3 métodos para exportar una definición:

- Utilizar la palabra-clave __declspec(dllexport) en la declaración del prototipo de la función a exportar.
- Utilizar la sentencia EXPORTS en un archivo con extensión DEF.
- Utilizar la opción /EXPORT como parte de la información pasada al enlazador.

Usaremos el archivo con extensión DEF. Éste es un archivo de texto con un conjunto de sentencias:

- NAME
- LIBRARY
- DESCRIPTION
- STACKSIZE
- SECTIONS
- EXPORTS
- VERSION

La sentencia EXPORTS es la única obligatoria y marca el inicio de una lista de definiciones de exportación. Cada definición tiene el siguiente formato:

```
entryname[=internalname] [@ordinal[NONAME]] [DATA] [PRIVATE]
```

Para nuestro ejemplo, el archivo DEF contendrá:

```
////////////////////////////////////////////////////
// Archivo MyDll.def
// Este comentario no debe incluirlo en el archivo DEF
////////////////////////////////////////////////////

LIBRARY SALUDAMEDLL
DESCRIPTION "Implementación de un saludo."
EXPORTS
 Saludame @1
```

La inclusión de un archivo DEF en el proyecto también permite la creación, por parte de IDE, de un archivo LIB, el cual podrá agregarse a los demás proyectos C/C++ desde donde deseemos usar la librería enlazándola en modo implícito.

Cuando se utiliza la palabra-clave __declspec(dllexport), ésta es la que le indica al IDE la creación del archivo LIB. Si para el ejemplo anterior deseáramos no utilizar un DEF, tendríamos que declarar el prototipo de la función a exportar de la forma:

```
__declspec(dllexport) void WINAPI Saludame(char * szNombre);
```

Utilización de una DLL

Para poder llamar a una DLL, el programa llamador debe enlazarse con ella. En este contexto, la palabra "enlazarse" significa "cargar a memoria la DLL y obtener las direcciones mapeadas de sus funciones exportadas. Dado que el "enlace" se realiza en tiempo de ejecución, se le llama **dinámico**, de allí el nombre de este tipo de librerías.

En enlace dinámico puede realizarse de dos formas:

- De modo implícito.
- De modo explícito.

EL MODO IMPLÍCITO

El enlace en modo implícito se consigue utilizando el archivo LIB generado cuando se compiló la DLL. Existen además programas utilitarios que permiten obtener un LIB directamente de un DLL ya generado, como el programa implib.exe de Borland. Este archivo LIB contiene código compilado que es agregado a nuestro programa. Dicho código se ejecutará al momento de iniciar el programa, realizando el enlace dinámico por nosotros. Adicionalmente, este código es sumamente eficiente y nos permite utilizar los prototipos de las funciones como si éstas fueran codificadas dentro de nuestro programa al momento de compilarlo.

El archivo LIB se utiliza durante el proceso de enlace del programa que hará uso de la librería. Si se usa un IDE, se tendrá que configurar el proyecto para que utilice los LIB's de las DLL's que deseamos utilizar. Como ejemplo, para agregar un archivo LIB a un proyecto en Microsoft Visual C++ 6.0, se colocan el nombre de éste en:

```
Project => Settings => Link => Category:General => Object/Library modules:
```

Luego, dentro del código de nuestro programa llamador podemos realizar la llamada a la función exportada de la siguiente manera:

```
#include <windows.h>
#include "..\MiLibreria\MiLibreria.h"

int WINAPI WinMain( HINSTANCE hInstance, HINSTANCE hInstancePrev, LPSTR lpCmdLine,
int nCmdShow ) {
    Saludame("JUAN");
    return 0;
}
```

Nótese que se utiliza el mismo archivo de cabecera de la DLL, MiLibreria.h.

EL MODO EXPLÍCITO

El enlace en modo explícito no requiere el uso del archivo LIB. Esto es útil cuando no disponemos de éste o cuando sabemos el nombre del DLL sólo después que la aplicación se está ejecutando, por ejemplo, siendo ingresado por el usuario.

Para este caso, necesitamos realizar los siguientes pasos:

- Solicitarle al sistema operativo que cargue la librería a memoria: **LoadLibrary**.
- Obtener la dirección de la función: **GetProcAddress**
- Solicitarle al sistema operativo que descargue la librería: **FreeLibrary**

Los prototipos de las funciones del API de Windows indicadas son:

```
HMODULE LoadLibrary(    // HMODULE es un typedef de HINSTANCE
LPCTSTR lpszModuleName  // Nombre del archivo DLL
);

BOOL FreeLibrary(
HINSTANCE hInstLib      // Handle a la DLL, devuelta por AfxLoadLibrary
);

FARPROC GetProcAddress(
HMODULE hModule,        // Handle a la DLL
LPCSTR lpProcName       // Nombre de la función
);
```

Se debe de usar la macro MAKEINTRESOURCE para el parámetro lpProcName cuando se desea obtener la dirección de la función exportada en base a su índice en la tabla de exportación.

Además debemos declarar una variable de tipo "puntero a función" para cada función que deseemos utilizar de dicha librería. Para nuestro ejemplo, la declaración del puntero a función sería:

```
typedef void (WINAPI * PFUNC) (char *);
```

Luego de definir el tipo podemos declarar una variable de dicho tipo:

```
PFUNC pfnSaludo;
```

La que utilizaremos para ejecutar la función en la DLL. Nuestro código de ejemplo quedaría de la siguiente forma:

```
typedef void (WINAPI * PFUNC) (char *);
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hInsPrev, LPSTR lpCLine, int
nCShow) {
    PFUNC pfnSaludo;
    HINSTANCE hDll;

    hDll = LoadLibrary("MyDll.dll");
    if ( hDll != NULL ) {
        pfnSaludo = (PFUNC)GetProcAddress( hDll, "Saludame" );
        if( pfnSaludo != NULL ) {
            // cualquiera de los dos formatos siguientes es válido
            ( *pfnSaludo ) ( "OTTO" );
            pfnSaludo ( "OTTO" );
        }
        FreeLibrary( hDll );
    }
    return 0;
}
```

Las funciones de carga y descarga lo que hacen es manejar un contador, mantenido por el sistema operativo, del uso de una DLL. Cuando ese contador regresa a cero (una carga lo aumenta en uno, una descarga lo baja en uno) la librería es descargada de memoria dado que ya nadie la está utilizando.

Mecanismo de Búsqueda de una DLL

El enlace estático utiliza el siguiente mecanismo de búsqueda en directorios para encontrar el archivo de la DLL:

1. En el directorio donde se encuentra el ejecutable de la aplicación.
2. En el directorio de trabajo actual
3. En el directorio System. Si es NT o Windows 2000, en el directorio System32.
4. En el directorio de Windows

5. En la lista de directorios de la variable PATH

De igual forma, si en el enlace dinámico no se especifica una ruta en el nombre del archivo pasado a la función AfxLoadLibrary, ésta utilizará el mismo mecanismo de búsqueda anterior.

Librerías en Java

Las librerías en Java se conocen con el nombre de paquetes. Un paquete Java es realmente un directorio en algún parte de nuestro disco. Todos los archivos CLASS bajo un mismo directorio pertenecen a un mismo paquete. Los sub-directorios dentro del directorio de un paquete corresponden a otros paquetes que generalmente están relacionados a éste. Por ejemplo, la siguiente estructura de directorios corresponde a parte de la librería estándar de Java.

```
<Directorio raíz de las clases de Java>
|
---- java
|     |
|     ---- awt
|        |
|        ---- applet
|              |
|              ---- event
---- javax
      |
      ---- swing
         |
         ---- JoptionPane
```

La estructura indica que el paquete (directorio) “java” contiene otros 2 paquetes: “awt” y “applet”. El paquete (directorio) “applet” contiene a su vez al paquete “event”. De igual forma, el paquete (directorio) “javax” contiene al paquete “swing” el cual contiene la clase “JOptionPane”.

Uso de un Paquete

El siguiente programa hace uso de la clase JOptionPane.

```
public class PruebaDeInterfaces {
    public static void main(String[] args) {
       javax.swing.JOptionPane.showMessageDialog(null, "Hola",
          "Un Mensaje", JOptionPane.ERROR_MESSAGE);
       System.exit(0);
    }
}
```

Nótese que el nombre completo de una clase indica el directorio donde se encuentra el archivo CLASS de ésta. Nótese además como Java une el concepto de espacio de nombres con el de librería. Un paquete define un espacio de nombres y todo espacio de nombres es un paquete.

También es posible publicar el contenido de un paquete, en otro paquete, utilizando la palabra reservada “import”, de forma que se puedan utilizar los nombres cortos de los elementos del paquete. El siguiente programa modifica el anterior utilizando “import”.

```
import javax.swing.*;
public class PruebaDeInterfaces {
    public static void main(String[] args) {
       JOptionPane.showMessageDialog(null, "Hola", "Un Mensaje",
          JOptionPane.ERROR_MESSAGE);
       System.exit(0);
    }
}
```

El uso de "import" es equivalente al uso de "using" en C++ y C#. Adicionalmente Java permite publicar elementos individuales de un paquete dentro de otro paquete. La primera línea del programa anterior pudo haberse escrito de la siguiente manera:

```
import javax.swing.JOptionPane;
```

Sin embargo, si se utilizan muchos elementos definidos dentro de un paquete, resulta más conveniente publicar todo el paquete en lugar de publicar cada elemento individualmente.

Java no permite la definición de alias para los paquetes.

Ubicación de un Paquete

El directorio raíz bajo el que se encuentran todos los paquetes (directorios) de Java instalados en una máquina se indica mediante una variable de entorno guardada en algún archivo de configuración del sistema operativo. En el caso de Windows, esta variable de entorno se llama CLASSPATH. Para Windows 95/98/Millenium, ésta y otras variables de entorno se suelen colocar dentro del archivo autoexec.bat, donde CLASSPATH es inicializado de la siguiente forma:

```
set CLASSPATH = <rutas iniciales separadas por ";">
```

Por ejemplo, supongamos que el directorio raíz estuviese especificado de la siguiente forma.

```
set CLASSPATH = C:\CLASES
```

Supongamos además que incluimos la instrucción siguiente en un programa:

```
import java.awt.Graphics;
```

La instrucción anterior le dirá al compilador que la clase Graphics la podrá encontrar en la ruta

```
C:\CLASES\JAVA\AWT
```

Java es sensitivo a las diferencias entre letras capitales y no-capitales, por tanto las siguientes importaciones son consideradas distintas por el compilador:

```
import java.awt.Graphics;
import java.awt.graphics;
```

Nótese que en los programas en Java se hace uso de algunas clases sin haber especificado su paquete, como la clase String y la clase System. Estas clases pertenecen al paquete "java.lang". Este paquete corresponde a la librería básica de java y contiene definiciones que son parte del mismo lenguaje, siendo importado automáticamente por el compilador por lo que no es necesario declarar una sentencia "import" para dicho paquete.

Creación de un Paquete

Para definir clases que formen parte de un paquete se requiere utilizar la directiva "package" al inicio del archivo donde se declaran estas clases. El siguiente programa define la clase A y B como parte del paquete P.

```
package P;
public class A {}
class B {}
```

Al compilar el archivo "A.java" conteniendo el código anterior, se generarán los archivos "A.class" y "B.class" correspondientes a las clases A y B respectivamente. La clase A es pública, por lo que podrá ser utilizada por otros paquetes fuera de "P", mientras que la clase B tiene el modificador de acceso por defecto "de-paquete", por lo que sólo podrá ser utilizada desde otras clases que pertenezcan al paquete

"P". Se pueden crear otros archivos que definan clases para el paquete "P", por lo que un paquete puede ser definido por partes, y no necesariamente en un solo archivo.

Cuando un archivo de Java no declara la directiva "package", las clases que define pertenecen al paquete global, el cual coincide con el directorio actual de ejecución del programa.

Utilización de un Nuevo Paquete

Para utilizar un paquete nuevo se requiere crear una estructura de directorios que coincida con la del nuevo paquete. Por ejemplo, si se tiene un paquete A con las clase A1, A2 y A3, y el subpaquete B con las clases B1 y B2, se requeriría crear un directorio A, colocar dentro los archivos A1.class, A2.class y A3.class, crear dentro el subdirectorio B y colocar dentro los archivos B1.class y B2.class.

La nueva estructura de directorios, para ser reconocida por el programa que requiere utilizarla, puede ir:

- Bajo el mismo directorio de los archivos "class" que lo utilizan.
- Bajo cualquier otro directorio en el computador, agregando dicho directorio a la lista de rutas especificadas en la variable de entorno utilizada por el compilador y el intérprete, CLASSPATH en el caso de Windows.

Si se revisa el contenido por defecto de la variable de entorno de ubicación de paquetes encontraremos que lo que indican es la ubicación de uno o más archivos con extensión "jar". Ésta es una forma alternativa de distribuir un paquete.

Los archivos JAR empaquetan, de la misma forma que un archivo de compresión como el ZIP, toda una estructura de directorios que forman un paquete, junto con los archivos incluidos en éstos. De esta forma, cuando el compilador o intérprete de Java busca una clase y encuentra la especificación de archivos con extensión JAR, continúa la búsqueda dentro de éstos de manera similar a como lo haría en un directorio.

La ventaja de utilizar archivos JAR es el ahorro de espacio y facilita la distribución de los paquetes, dado que sólo se requiere copiar un archivo y no crear toda una estructura de directorios en el computador donde se desea utilizar un paquete.

Para generar un archivo JAR se puede utilizar el programa utilitario, distribuido junto con el JDK, "jar.exe". La siguiente sintaxis corresponde a la llamada a este programa:

```
jar {ctxu}[vfm0Mi] [archivo-jar] [archivo-manifest] [-C dir] archivos
```

El detalle de lo que cada opción significa se puede obtener ejecutando el programa sin ningún parámetro. El siguiente comando es un ejemplo de creación de un paquete MiPaquete.jar con las clases contenidas en el directorio DirectorioRaiz:

```
jar cvf MiPaquete.jar -C DirectorioRaiz / .
```

Por ejemplo, asumiendo:

```
C:\trab\repositorio.class

C:\trab\miembros\GestorUsuario.class

C:\trab\miembros\Usuario.class
```

Con una ventana de comandos con directorio actual "c:\trab", ejecutar:

```
jar.exe –cvf Miembros.jar repositorio.class Miembros/GestorUsuario.class
Miembros/Usuario.class
```

```
java.exe  -cp  Miembros.jar;.  repositorio
```

Librerías en C#

Las DLL's presentan una carencia en su concepción: El sistema operativo no registra automáticamente que programas hacen uso de una DLL. Algunos casos típicos de problemas originados por esta carencia son:

- Si al desinstalar un programa, éste elimina una DLL que es utilizada por otro programa, éste último dejará de funcionar correctamente.
- Si un usuario cambia de posición el archivo de la DLL utilizada por un programa, éste quizá no lo encuentre, por lo que dejará de funcionar correctamente.

Adicionalmente, dado que las DLL's se ubican en el sistema de archivos, en directorios específicos, si se intenta copiar una nueva DLL, utilizada por ejemplo por un programa nuevo, y su nombre coincide con otra preexistente, se reemplazará el archivo de la DLL antigua. Por lo tanto, todos los programas que utilizaban la DLL reemplazada dejarán de funcionar correctamente. Aún en el caso que dicho reemplazo sea intencional, por ejemplo al actualizar la versión de una DLL, si la verificación de la versión de la DLL preexistente versus la nueva no se realiza correctamente, es posible que se reemplace una DLL más reciente con una más antigua. Incluso aún en el caso que el reemplazo sea realizado con una correcta verificación de las versiones, siempre es posible que un error de programación en la nueva versión haga que un programa que funcionaba correctamente con la antigua, deje de funcionar con la nueva.

Todos estos problemas son demasiado comunes, por lo que .NET desarrolla un nuevo concepto orientado a darles solución: **Los ensamblajes**.

Los Ensamblajes

Un ensamblaje es una unidad de instalación autodescriptiva. Todos los archivos generados en .NET son parte de algún ensamblaje. Por ejemplo, los ejecutables (*.EXE) son ensamblajes.

Un ensamblaje puede estar formado por uno o más archivos, los que en conjunto contienen los siguientes elementos:

- Metadata del ensamblaje
- Metadata de tipos
- Código MSIL
- Recursos

Los siguientes diagramas muestran dos ejemplos de distribución de estos elementos en los archivos de un ensamblaje. El primero, Ensamblaje1.dll, es un ensamblaje tipo librería (esto es, no existe un punto de entrada o método Main desde donde ejecutar un hilo primario) formado por un único archivo. El segundo es un ensamblaje tipo ejecutable (si existe un Main) formado por tres archivos. La Metadata del Ensamblaje del archivo Ensamblaje2.exe guarda la descripción exacta de los archivos que forman el ensamblaje. Fuera de la Metadata del Ensamblaje, el resto de elementos pueden existir en cada uno de los archivos que forman un ensamblaje.

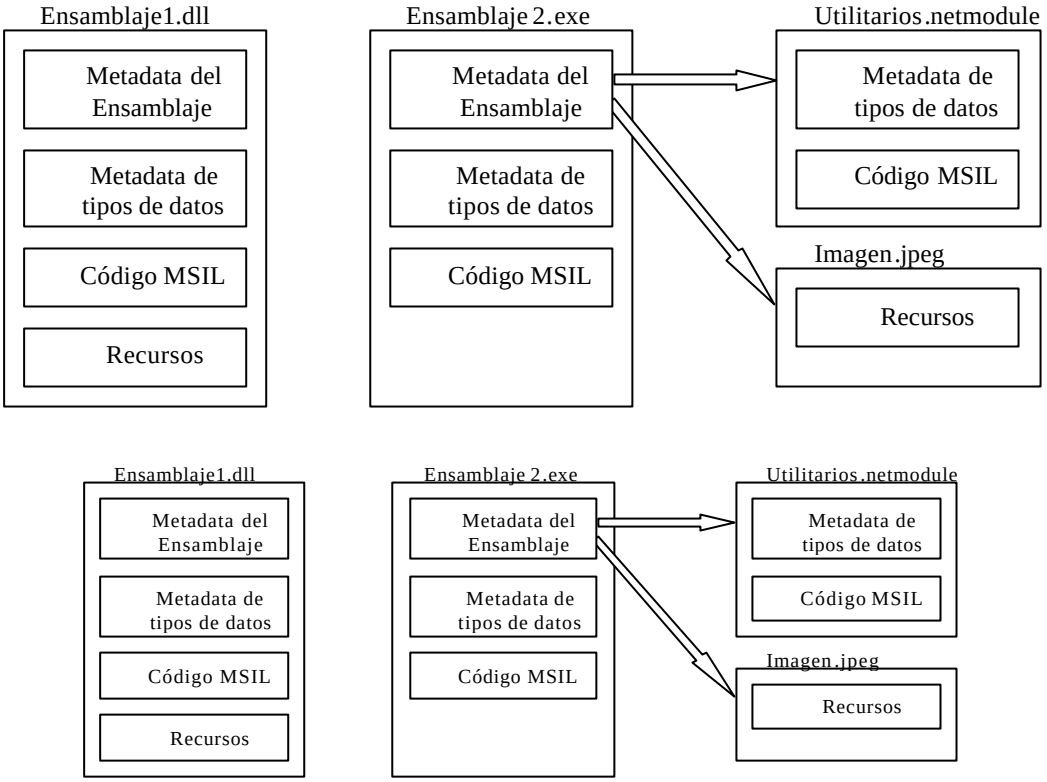

Figura 5.2. Ejemplo de Distribución de Elementos

Un ensamblaje puede estar formado por los siguientes tipos de archivos:

- Un archivo principal, EXE o DLL, donde se encuentra la Metadata del Ensamblaje, entre otros elementos.
- Cero, uno o más archivos de módulo de .NET, con extensión NETMODULE. Los módulos de .NET no contienen Metadata del Ensamblaje.
- Cero, uno o más archivos de recursos, como archivos de imagen, sonido, video, etc.

Si bien para el Ensamblaje2 anterior, éste es formado por tres archivos, es posible compilar el archivo principal, Ensamblaje2.exe, de forma que los demás archivos se incluyan dentro.

Las características más importantes de un ensamblaje son:

- Es autodescriptivo. Toda la información sobre la versión del ensamblaje, la descripción de los tipos de datos que contiene, los archivos que lo forman, etc., se encuentra dentro del propio ensamblaje, por lo que instalar un ensamblaje sólo requiere copiarlo. No se utiliza el registro de Windows.
- Registra sus dependencias a otros ensamblajes: Nombre, versión, etc.
- Pueden instalarse, en un mismo computador, diferentes versiones de un mismo ensamblaje, sin causar conflicto.

- Instalación sin impactos, es decir, el instalar un ensamblaje sólo puede afectar a los programas que lo utilizan. No hay posibilidad que un ensamblaje reemplace otro con el mismo nombre pero para otro uso, o con diferente versión. Los ensamblajes se identifican de manera única.

- Se ejecutan dentro de Dominios de Aplicación de un proceso.

Un ensamblaje en ejecución se le denomina aplicación. Dentro de un mismo proceso se pueden ejecutar varias aplicaciones, del mismo o distinto ensamblaje, cada uno en un Dominio de Aplicación distinto. Un Dominio de Aplicación es la frontera que aísla una aplicación del resto de aplicaciones. De esta manera, las fallas en una aplicación no pueden afectar a otra aplicación, aún perteneciendo ambas al mismo proceso. Para que un objeto de una aplicación acceda a uno en otra aplicación, requiere hacer uso de un objeto proxy, cuyo concepto es el mismo que el Stub de CORBA (ver capítulo 9).

Existen dos tipos de ensamblajes: Los **privados** y los **públicos** o **compartidos**.

Los ensamblajes privados son aquellos que se instalan junto con el programa que los utiliza, en el mismo directorio o en un subdirectorio de donde se encuentra este programa. No se manejan números de versión ni nombres únicos, dado que no se necesitan. Este tipo de ensamblaje puede causar conflictos (de versiones por ejemplo) pero sólo en la aplicación que lo utiliza, y como es natural se resuelven durante el proceso de desarrollo de dicho programa. Este tipo de ensamblaje no puede afectar otros programas.

Los ensamblajes públicos o compartidos pueden ser utilizados por más de un programa, por lo que se instalan en un lugar común. Estos ensamblajes deben seguir las siguientes reglas:

- Tener un nombre único (llamado nombre fuerte). Parte de este nombre es un número de versión mandatorio.

- Mayormente, estar instalado en la Global Assembly Cache, un directorio dentro del directorio de Windows (por ejemplo, C:\WINNT\assembly para Windows 2000).

El principal componente de la Metadata del Ensamblaje es el Manifiesto. Éste contiene:

- El nombre identificatorio para el ensamblaje, la versión, la cultura y una llave pública (una cadena de carácteres).

- Una lista de los archivos que forman el ensamblaje.

- Una lista de los ensamblajes referenciados por éste y por tanto, de los que depende para su ejecución.

- Un conjunto de Solicitudes de Permiso, que son los permisos necesarios para correr o utilizar el ensamblaje.

- Metadata de tipos de datos para aquellos tipos dentro de los archivos de módulo del ensamblaje.

Puede examinarse el contenido de un ensamblaje, incluyendo su manifiesto, utilizando el programa utilitario ILDASM.EXE.

El Ensamblaje Tipo Librería

Todos los programas hasta ahora generados son ensamblajes de un sólo archivo de tipo ejecutable. Para crear un ensamblaje tipo librería que sea utilizada por otro ensamblaje se debe de compilar como tal. Pongamos un ejemplo.

El siguiente código corresponde al archivo Ensamblaje1.cs:

```
using System;
public class Clase1 {
    public void Saludame(string nombre) {
        Console.WriteLine("Hola " + nombre);
    }
}
```

Para compilarlo como un ensamblaje tipo librería, utilizamos la opción /target::library del compilador csc.

```
csc /target:library Ensamblaje1.cs
```

Esto genera el archivo Ensamblaje1.dll. El siguiente código corresponde al archivo Ensamblaje2.cs, dentro del cual, se hace uso de la clase Clase1 definida dentro del ensamblaje Ensamblaje1.dll.

```
using System;
class Clase2 {
    public static void Main() {
        Clase1 obj = new Clase1();
        obj.Saludame("Juan");
    }
}
```

Para compilarlo como un ensamblaje tipo ejecutable, que hace referencia a elementos dentro del ensamblaje Ensamblaje1.dll utilizamos la opción /reference:Ensamblaje1.dll del compilador csc.

```
csc /reference:Ensamblaje1.dll Ensamblaje2.cs
```

Esto genera el archivo Ensamblaje2.exe. Para ejecutar este programa, ambos archivos, Ensamblaje1.dll y Ensamblaje2.exe, deben de estar en el mismo directorio. Este ejemplo corresponde a un ensamblaje privado. Los ensamblajes públicos van más allá de los alcances del presente curso.

Relación con los Espacios de Nombres

C# separa los conceptos de espacios de nombres y el de ensamblaje. Un espacio de nombres puede estar definido por partes en varios ensamblajes.