DISEÑO DE LA LÓGICA DE LA APLICACIÓN

Tema 5

Ingeniería del Software

ETS Ingeniería Informática

DSIC – UPV

Objetivos

- Comprender el diseño software como la especificación de la manera en que un conjunto de objetos interactúan entre ellos y administran su propio estado y operaciones

- Cómo derivar un diseño a partir del diagrama de clases

Contenidos

1. Introducción
2. Diseño de Objetos
3. Diseño de Constructores
4. Diseño Arquitectónico

INTRODUCCIÓN

Introducción

Modelado Conceptual (Análisis)

Es el proceso de construcción de un **modelo** / especificación detallada del **problema del mundo real** al que nos enfrentamos.
Está **desprovisto** de consideraciones de ***diseño*** e ***implementación***.

¿Modelado = Diseño?

NO

Modelado vs Diseño

Modelado

Orientado al **Problema**

proceso que **extiende**, **refina** y **reorganiza** los aspectos detectados en el proceso de modelado conceptual, para generar una **especificación rigurosa orientada a la obtención de la solución** del sistema software.

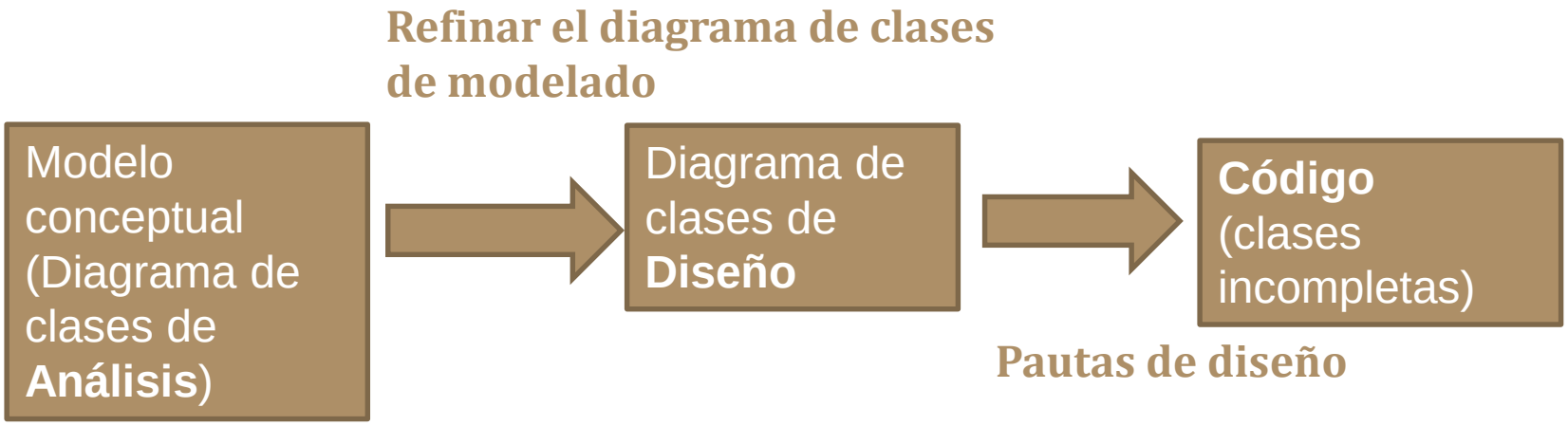
Diseño

Orientado a la **Solución**

El diseño añade el entorno de desarrollo (y lenguaje de implementación) como un nuevo elemento a considerar.

Diseño de Objetos

- Entrada: Modelo Conceptual
- Salida: Diseño – Clases diseñadas en lenguaje OO



Decisiones y pautas de Diseño

- Refinamiento del diagrama de clases
  - Crear nuevas clases
  - Eliminar clases y/o fusionarlas con otras
  - Crear nuevas relaciones entre clases
  - Modificar relaciones existentes
    - Restringir la navegabilidad
    - . . .
- Pautas para
  - Diseño de Clases
  - Diseño de Asociaciones
  - Diseño de Agregaciones
  - Diseño de Especializaciones

Modelo conceptual (Diagrama de clases de **Análisis**)

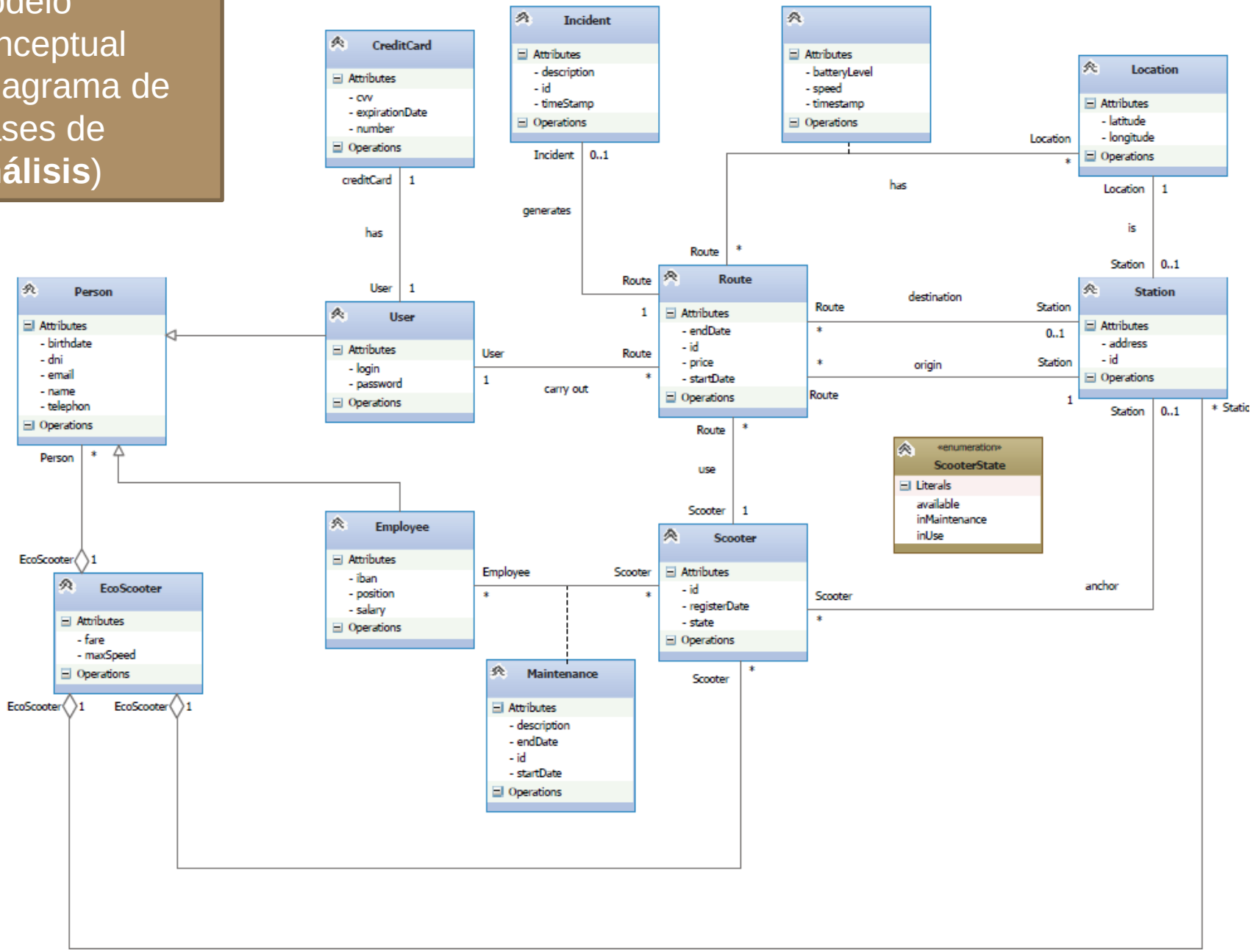
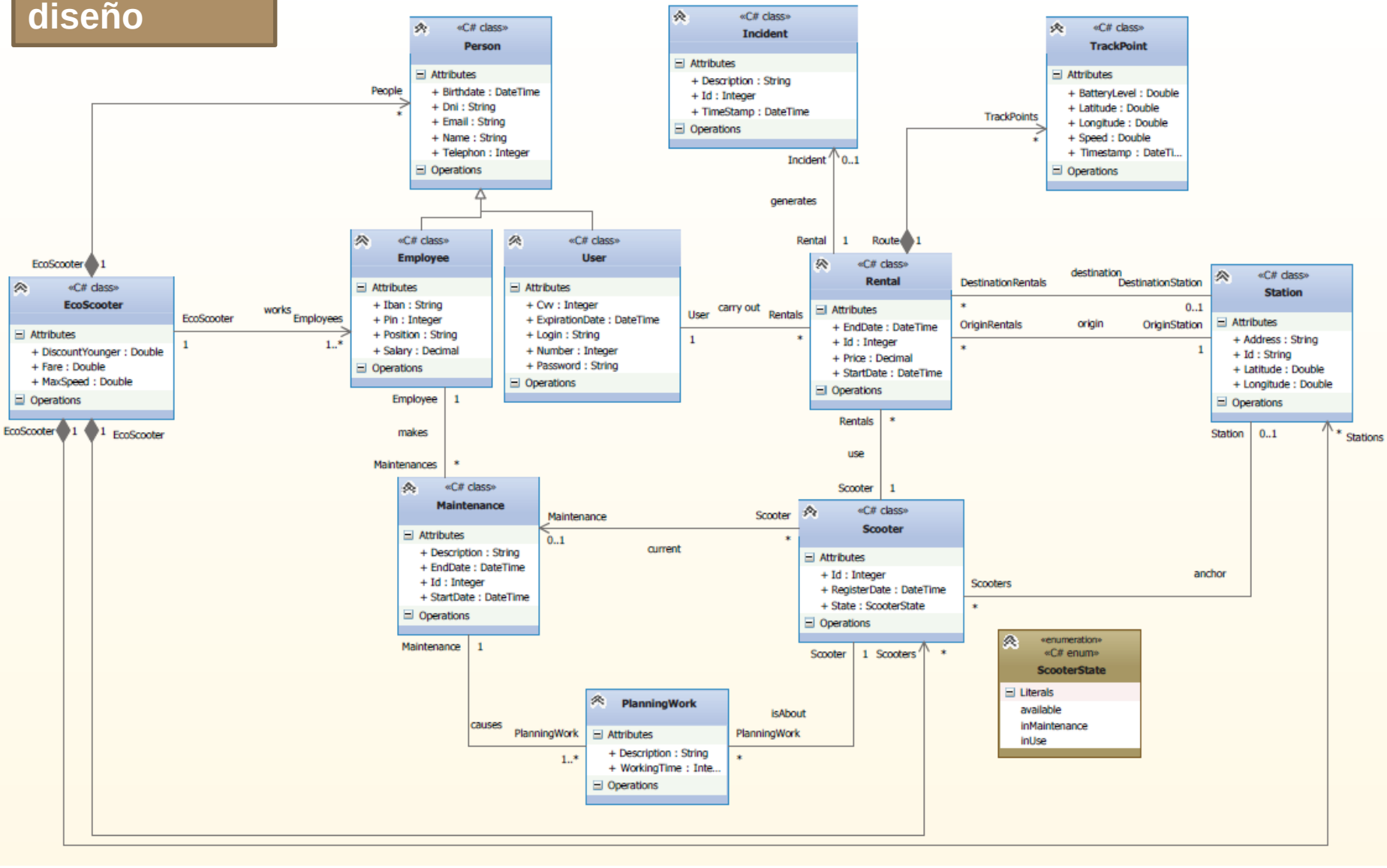


Diagrama de clases de **diseño**

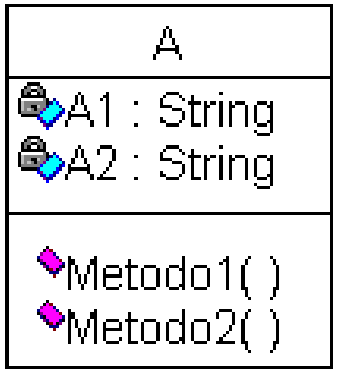


PAUTAS PARA EL DISEÑO DE OBJETOS

Con ejemplos en C#

Clases

Diagrama de Clases



Diseño en C#

```
public class A
{
 private string A1;
 private string A2;

 public void setA1(string a) {…}
 public void setA2(string a) {…}
 public string getA1() {…}
 public string getA2() {…}

 public int Metodo1(){…}
 public string Metodo2() {…}
}
```

Nota: Los métodos consultores y modificadores los denominaremos *set/get*

Clases (usando propiedades C#)

Métodos clásicos

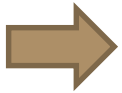
```
private string A1;
private string A2;

public void setA1(string a){
   A1=a;
}

public void setA2(string a){
   A2=a;
}

public string getA1(){
   return A1;
}

public string getA2(){
   return A2;
}
```



Propiedades de C#

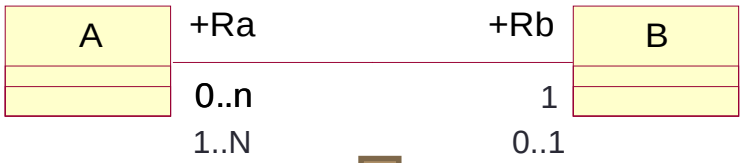
```
public string A1 {
   get;
   set;
}
public string A2 {
   get;
   set;
}
```

Asociaciones Uno - Uno


```
public class A
{
    public B Rb {
        get;
        set;
    }
}
```

```
public class B
{
    public A Ra {
        get;
        set;
    }
}
```

Asociaciones Uno - Muchos




```
public class A
{
 public B Rb {// asociación uno-uno
        get;
        set;
   }

public class B
{
   public ICollection<A> Ra {
        get;
        set;
   }
}
```

Asociaciones Uno – Muchos (Bis)

Alternativa: métodos específicos para acceso a las colecciones

```
public class B
{
    private ICollection<A> Ra;
    public void AddA(A a){
        Ra.Add(a);
    }
    public void RemoveA (A a){
        Ra.Remove(a);
    }
    public A GetA(object idA){
        foreach (A a in Ra) if (a.Id == id) return a;
        return null;
    }
    public void RemoveA(object idA){
        RemoveA(GetA(idA));
    }
}
```

Colecciones en C#

- Genéricas
  - List<T>, LinkedList<T>, SortedList<K,V>
  - Stack<T>, Queue<T>
  - Dictionary<K,V>, SortedDictionary<K,V>
  - HashSet<T>, SortedSet<T>
- No genéricas (almacenan object y requieren conversión)
  - Array, ArrayList, SortedList
  - Hashtable
  - Queue, Stack
- Concurrentes y otras

Colecciones y estructuras de datos (Ayuda .NET Framework)

Elegir colección

Deseo...	Opciones de colección genérica	Opciones de colección no genérica	Opciones de colección de subprocesos o inmutable
Almacenar elementos como pares clave/valor para una consulta rápida por clave	Dictionary<TKey,TValue>	Hashtable (Colección de pares clave/valor que se organizan en función del código hash de la clave).	ConcurrentDictionary<TKey,TValue> ReadOnlyDictionary<TKey,TValue> ImmutableDictionary<TKey,TValue>
Acceso a elementos por índice	List<T>	Array ArrayList	ImmutableList<T> ImmutableArray
Utilizar elementos FIFO (el primero en entrar es el primero en salir)	Queue<T>	Queue	ConcurrentQueue<T> ImmutableQueue<T>
Utilizar datos LIFO (el último en entrar es el primero en salir)	Stack<T>	Stack	ConcurrentStack<T> ImmutableStack<T>
Acceso a elementos de forma secuencial	LinkedList<T>	Sin recomendación	Sin recomendación
Recibir notificaciones cuando se quitan o se agregan elementos a la colección. (implementa INotifyPropertyChanged y INotifyCollectionChanged)	ObservableCollection<T>	Sin recomendación	Sin recomendación
Una colección ordenada	SortedList<TKey,TValue>	SortedList	ImmutableSortedDictionary<TKey,TValue> ImmutableSortedSet<T>
Un conjunto de funciones matemáticas	HashSet<T> SortedSet<T>	Sin recomendación	ImmutableHashSet<T> ImmutableSortedSet<T>

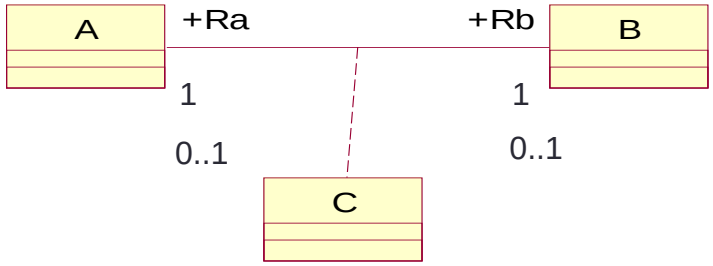
Fuente: https://docs.microsoft.com/es-es/dotnet/standard/collections/index#choosing-a-collection

Asociaciones Muchos - Muchos


```
public class A
{
   public ICollection<B> Rb {
        get;
        set;
        }
}

public class B
{
   public ICollection<A> Rb {
        get;
        set;
        }
}
```

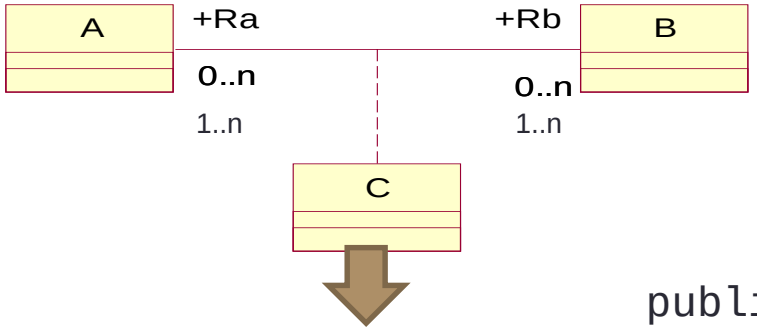
Clase Asociación Uno - Uno




```
public class A
{
  public C Rc {
        get;
        set;
        }
}
public class B
{
  public C Rc {
        get;
        set;
        }
}
```

```
public class C
{
   public A Ra {
        get;
        set;
   }
   public B Rb {
        get;
        set;
        }
}
```

Clase Asociación Muchos - Muchos




```
public class A
{
  public ICollection<C> Rc {
        get;
        set;
        }
}
public class B
{
 public ICollection<C> Rc {
        get;
        set;
        }

}
```

```
public class C
{
   public A Ra {
        get;
        set;
        }
   public B Rb {
        get;
        set;
        }
}
```

Agregación /Composición

Agregación uno-a-uno

Agregación uno-a-muchos

Agregación muchos-a-muchos

Sigue las mismas pautas vistas para las asociaciones

Asociaciones y atributo de enlace uno a muchos


```
public class UsuarioValidado : UsuarioBasico
{
    private string nombre;
    private string numeroTelefono;
    // colocar esFavorito aquí sería un ERROR
    private ICollection<Contacto> redContactos;
}
```

```
public class Contacto
{
    private string nombre;

    private UsuarioValidado usuarioRedContactos;
    // importante: el atributo de enlace en relación 1-muchos
    // se coloca en la parte de muchos
    private bool esFavorito;
}
```

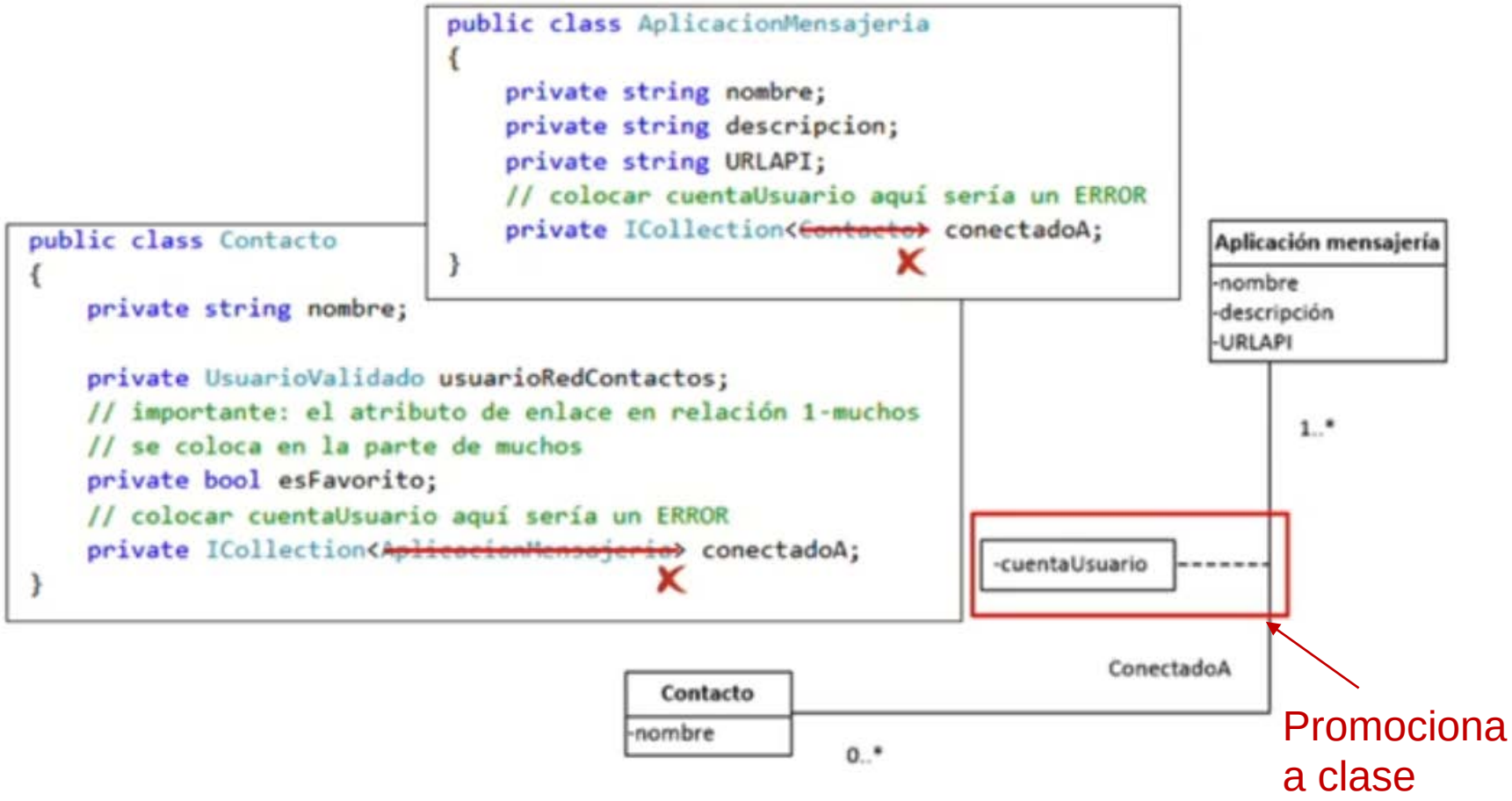
Asociaciones y atributo de enlace muchos a muchos

```
public class AplicacionMensajeria
{
    private string nombre;
    private string descripcion;
    private string URLAPI;
    // colocar cuentaUsuario aquí sería un ERROR
    private ICollection<Contacto> conectadoA;
}
```

```
public class Contacto
{
    private string nombre;

    private UsuarioValidado usuarioRedContactos;
    // importante: el atributo de enlace en relación 1-muchos
    // se coloca en la parte de muchos
    private bool esFavorito;
    // colocar cuentaUsuario aquí sería un ERROR
    private ICollection<AplicacionMensajeria> conectadoA;
}
```

Asociaciones y atributo de enlace muchos a muchos



Asociaciones y atributo de enlace muchos a muchos

```
public class AplicacionMensajeria
{
    private string nombre;
    private string descripcion;
    private string URLAPI;
    // colocar cuentaUsuario aquí sería un ERROR
    private ICollection<Contacto> conectadoA;
}
```

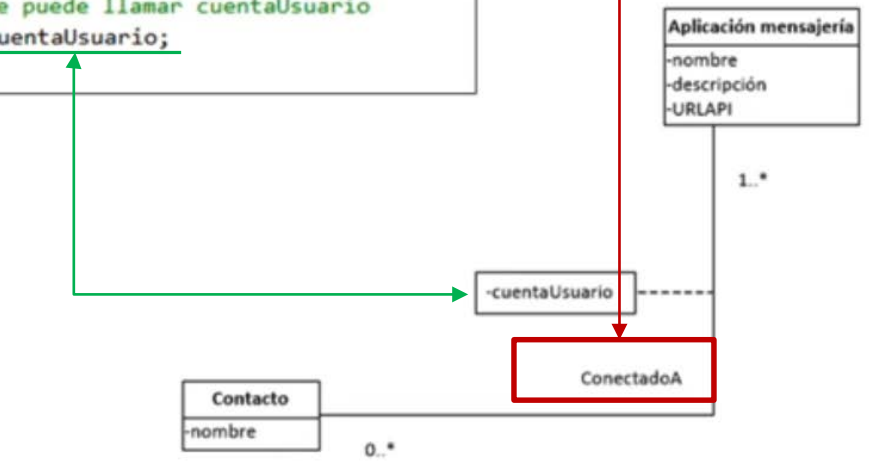
```
public class Contacto
{
    private string nombre;

    private UsuarioValidado usuarioRedContactos;
    // importante: el atributo de enlace en relación 1-muchos
    // se coloca en la parte de muchos
    private bool esFavorito;
    // colocar cuentaUsuario aquí sería un ERROR
    private ICollection<AplicacionMensajeria> conectadoA;
}
```

Asociaciones y atributo de enlace muchos a muchos

```
public class ConectadoA
{
    private Contacto conectadoAContacto;
    private AplicacionMensajeria conectadoAAplicacion;
    // la clase NO se puede llamar cuentaUsuario
    private string cuentaUsuario;
}
```

Clase con el nombre de la relación




```
public class AplicacionMensajeria
{
    private string nombre;
    private string descripcion;
    private string URLAPI;
    // tener una colección de Contacto sería un error
    private ICollection<ConectadoA> conectadoA;
}
```

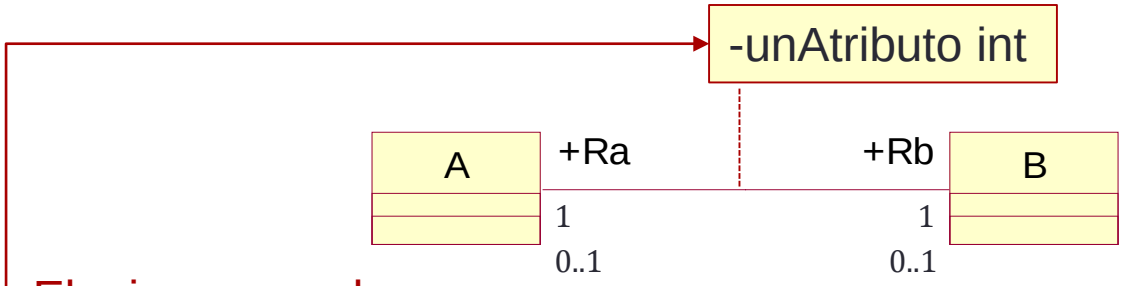
✔

```
public class Contacto
{
    private string nombre;

    private UsuarioValidado usuarioRedContactos;
    // importante: el atributo de enlace en relación 1-muchos
    // se coloca en la parte de muchos
    private bool esFavorito;
    // tener una colección de AplicacionMensajeria sería un error
    private ICollection<ConectadoA> conectadoA;
}
```

✔

Asociaciones y atributo de enlace Uno - Uno

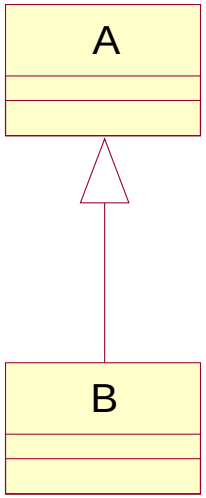


Elegimos una de las dos clases

```
public class A
{   public int UnAtributo{
        get;
        Set;
    }
    public B Rb {
        get;
        set;
    }
}
```

```
public class B
{
    public A Ra {
        get;
        set;
    }
}
```

Especialización/Generalización



```
public class A
{
  ...
}

public class B : A
{
  ...
}
```

Se puede jugar con la visibilidad de los atributos y métodos según si queremos maximizar la facilidad de extensión o la encapsulación

Ojo porque en C# los modificadores no son los mismos que en Java y tiene algunas peculiaridades importantes

Modificadores de acceso (Referencia de C#)

- Palabras clave que se usan para especificar la accesibilidad declarada de un miembro o un tipo.
  En c#: [Ejemplo]

  - `public`
  - `protected`
  - `internal`
  - `private`

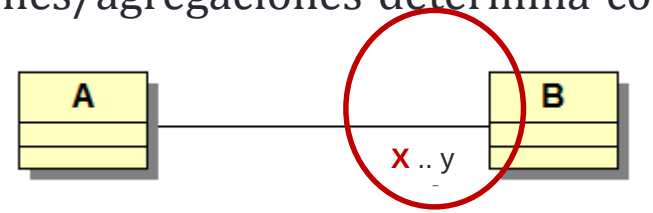
- `public`: el acceso no está restringido.
- `protected`: el acceso está limitado a la clase contenedora o a los tipos derivados de la clase contenedora.
- `internal`: el acceso está limitado al ensamblado actual.
- `protected internal`: el acceso está limitado al ensamblado actual o a los tipos derivados de la clase contenedora.
- `private`: el acceso está limitado al tipo contenedor.
- `private protected`: el acceso está limitado a la clase contenedora o a los tipos derivados de la clase contenedora que hay en el ensamblado actual.

Fuente: https://docs.microsoft.com/es-es/dotnet/csharp/language-reference/keywords/access-modifiers

DISEÑO DE CONSTRUCTORES

Diseño general de constructores

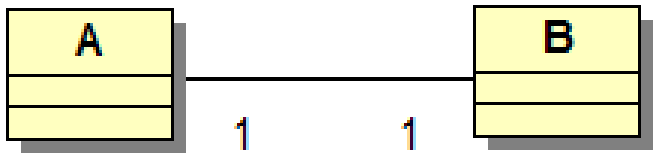
- *Inicializar un objeto* supone dar valores tanto a sus **atributos** como a los **enlaces con objetos de otras clases**, si los hubiere.
- La **multiplicidad mínima** de las asociaciones/agregaciones determina cómo se realiza la inicialización



x	y	Declaración en A	Constructor de A
0	1	`public B Rb { get; set; }`	`public A(…){…};`
1	1		`public A(…, B b, …) { this.Rb = b; … }`
0	N	`public ICollection<B> Rb { get; set; }`	`public A(…){ Rb=new List<B>; … }`
1	N		`public A(…, B b, …){ Rb = new List<B>; Rb.Add(b); … }`

Constructores en relaciones uno-uno

- Cuando en ambos extremos de una asociación, la **multiplicidad mínima** es 1, se crea una dependencia circular que no puede resolverse en un paso.
- Debe implementarse una inicialización en varios pasos en modo "**transaccional**"…. (*detalle implementación - Tema 8*).



Tenemos que garantizar por código que la restricción se cumple

```
public class A {
 B el_B;
 public A(...)
 {
 ...
 }
 ...
 }
```

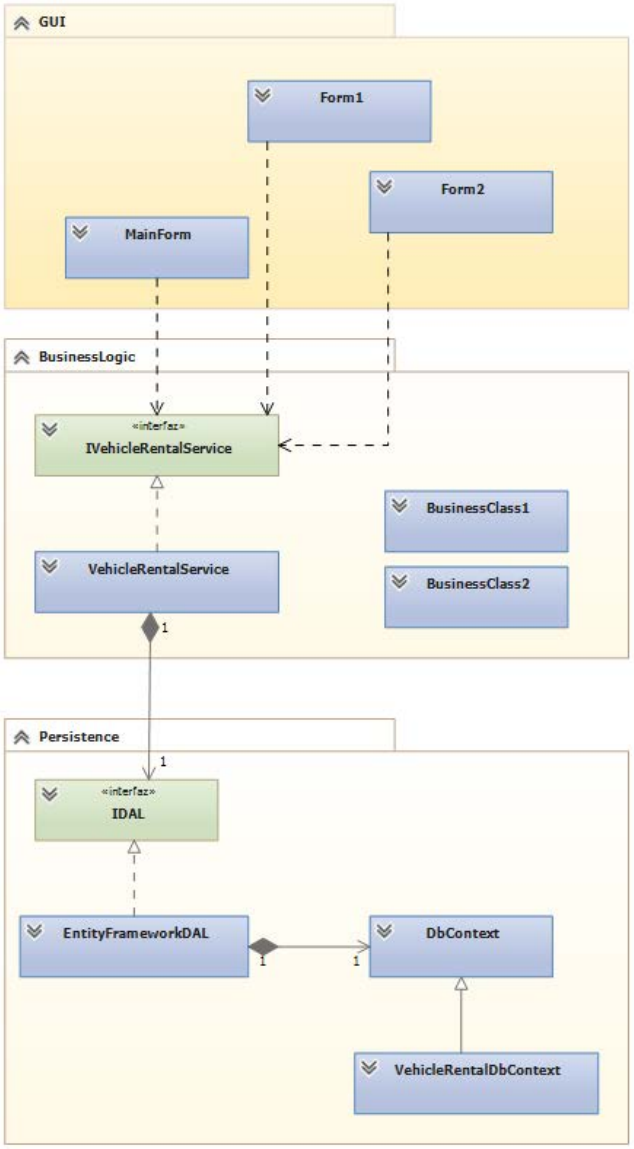
```
public class B {
 A el_A;
 public B(... A el_A)
 {
   this.el_A=el_A;
 }
 ...
 }
```

```
...
//se debe ejecutar como un todo
A un_A=new A(...);          // un A
B un_B = new B(un_A);       // un B
un_A.setEl_B(un_B);         // 1..1
...
```

DISEÑO ARQUITECTÓNICO

Diseño de la separación de capas

- Seguimos una arquitectura multi-capa con:
  - Presentación (IGU)
  - Lógica de negocio
  - Persistencia para acceso a la fuente de datos

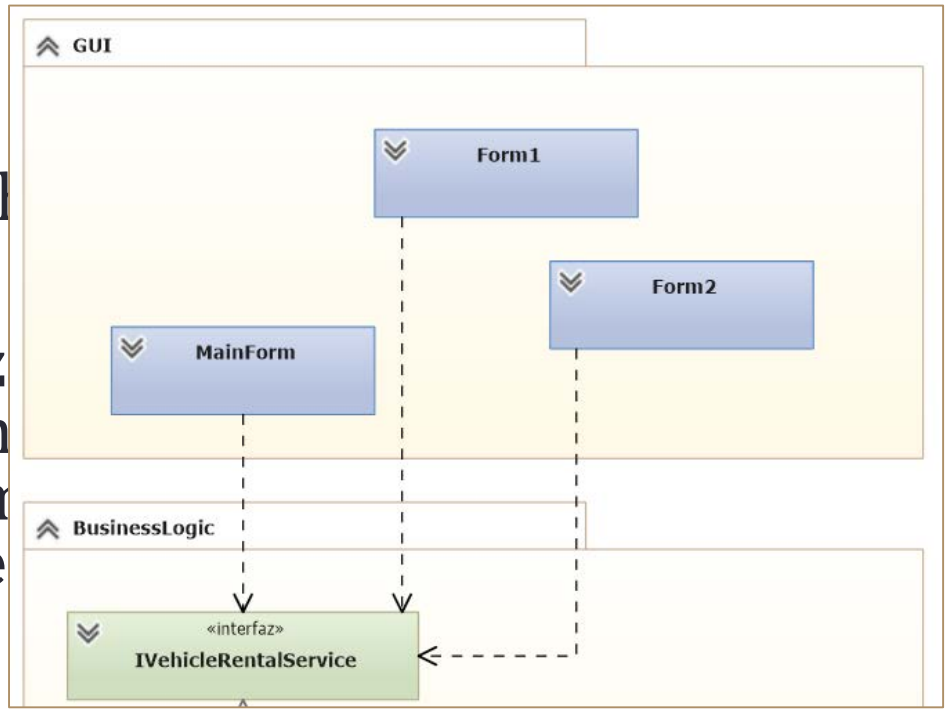


Separación de capas. Presentación

- Conjunto de formularios (uno de ellos el **MainForm**)
- **Todos** los formularios accederán a los servicios que ofrece la lógica de negocio (en el ejemplo, vía VehicleRentalService)
- Por lo tanto, el **constructor** de **todos** los formularios necesita una referencia a VehicleRentalService
- Para incrementar la **reutilización** definimos una **interfaz** IVehicleRentalService que indica el **qué**, no el **cómo**. Así, se podrán adoptar **distintas implementaciones** y la capa de presentación **no se verá afectada**

Separación de capas. Presentación

- Conjunto de formularios (uno de ellos el **MainForm**)
- **Todos** los formularios accederán a los servicios que ofrece la lógica de negocio (en el ejemplo, vía VehicleRentalService)
- Por lo tanto, el **constructor** necesita una referencia a Vel
- Para incrementar la **reutiliz** IVehicleRentalService que in podrán adoptar **distintas im** presentación **no se verá afe**

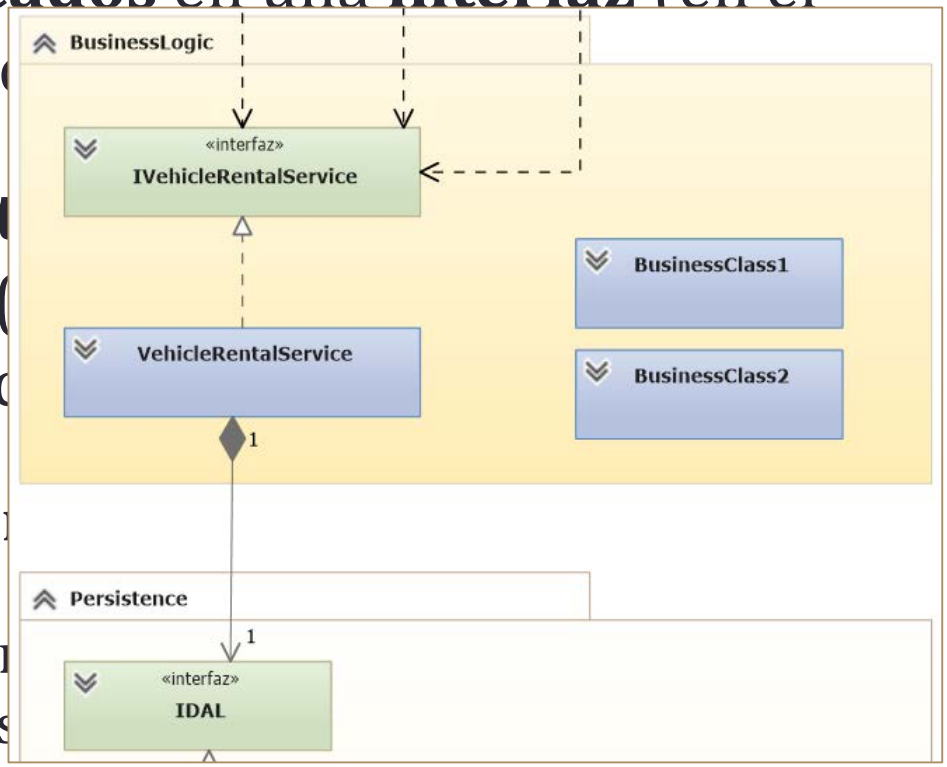


Separación de capas. Lógica de negocio

- Proporciona todos los **servicios** de nuestra aplicación (**casos de uso**)
- Estos servicios son **identificados** en una **interfaz** (en el ejemplo IVehicleRentalService)
- Se pueden proporcionar **distintas implementaciones** de los servicios de esa interfaz (ej. VehicleRentalService o en el futuro VehicleRentalService2, VehicleRentalService3…)
  - estas clases **trabajarán con el resto** de las clases de la lógica
  - cada **implementación** puede trabajar con una **capa de acceso a datos** distinta (**DAL**, Data Access Layer), modelada como interfaz

Separación de capas. Lógica de negocio

- Proporciona todos los **servicios** de nuestra aplicación (**casos de uso**)
- Estos servicios son **identificados** en una **interfaz** (en el ejemplo IVehicleRentalServi
- Se pueden proporcionar **dist** los servicios de esa interfaz (el futuro VehicleRentalServi
  - estas clases **trabajarán con el**
  - cada **implementación** puede t **datos** distinta (**DAL**, Data Acces

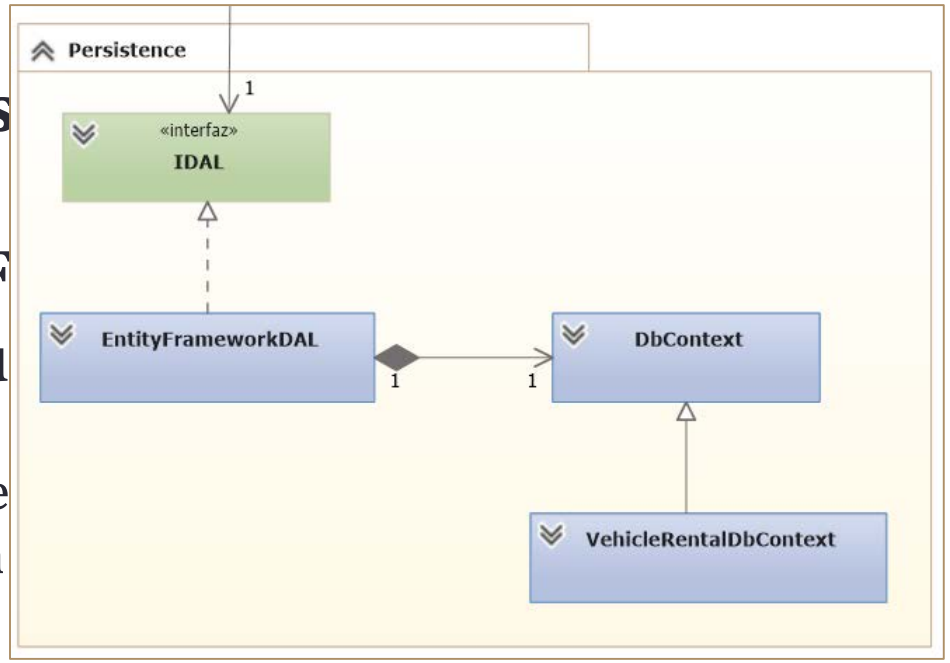


Separación de capas. Persistencia

- Proporciona el **acceso a la fuente de datos** (BD relacional, BDOO, archivo de texto, archivo XML, etc.)
- El acceso a datos se **identifica** mediante una **interfaz** (en el ejemplo IDAL)
- Se pueden proporcionar **distintas implementaciones** de los servicios de esa interfaz para acceder a las distintas fuentes de datos (ej. EntityFrameworkDAL que trabaja con el framework de BD de Visual Studio)
  - pero en un futuro se podría implementar un XMLDAL para trabajar con archivos XML y **la capa de lógica no se vería afectada**

Separación de capas. Persistencia

- Proporciona el **acceso a la fuente de datos** (BD relacional, BDOO, archivo de texto, archivo XML, etc.)
- El acceso a datos se **identifica** mediante una **interfaz** (en el ejemplo IDAL)
- Se pueden proporcionar **dis**
  los servicios de esa interfaz
  fuentes de datos (ej. EntityF
  el framework de BD de Visu
  - pero en un futuro se podría imple
    archivos XML y **la capa de lógica**



Bibliografía

- https://msdn.microsoft.com/es-es. Ayuda on-line para desarrollar software OO con Visual Studio y C#

- Doyle, B. C# Programming: From Problem Analysis to Program Design, Cengage Learning 2016

- Stevens, P., Pooley, R. Utilización de UML en Ingeniería del Software con Objetos y Componentes. Addison-Wesley Iberoamericana 2002.