DISSENY
DE LA LÒGICA DE L'APLICACIÓ

Tema 5

Enginyeria del Programari
ETS Enginyeria Informàtica
DSIC – UPV

Curs 2021-2022

Objectius

- Comprendre el disseny programari com un conjunt d'objectes que interactuen entre ells i que administren el seu propi estat i operacions.

- Com derivar un disseny a partir del diagrama de classes.

Continguts

1. Introducció
2. Disseny d'Objectes
3. Disseny de Constructors
4. Disseny Arquitectònic

INTRODUCCIÓ

Introducció

Modelat Conceptual *(Anàlisi)*

És el procés de construcció d'un **model** / d'una especificació detallada del **problema del món real** al que ens enfrontem.
Està **desproveït** de consideracions de ***disseny*** i ***implementació***.

Modelat = Disseny?

NO

Introducció

Modelatge vs. **Disseny**

Modelatge

Orientat al **Problema**

És un procés que **estén**, **refina** i **reorganitza** els aspectes detectats en el procés de modelatge conceptual, per a generar una **especificació rigorosa** del sistema d'informació sempre **orientada a l'obtenció de la solució** del sistema programari.

Disseny

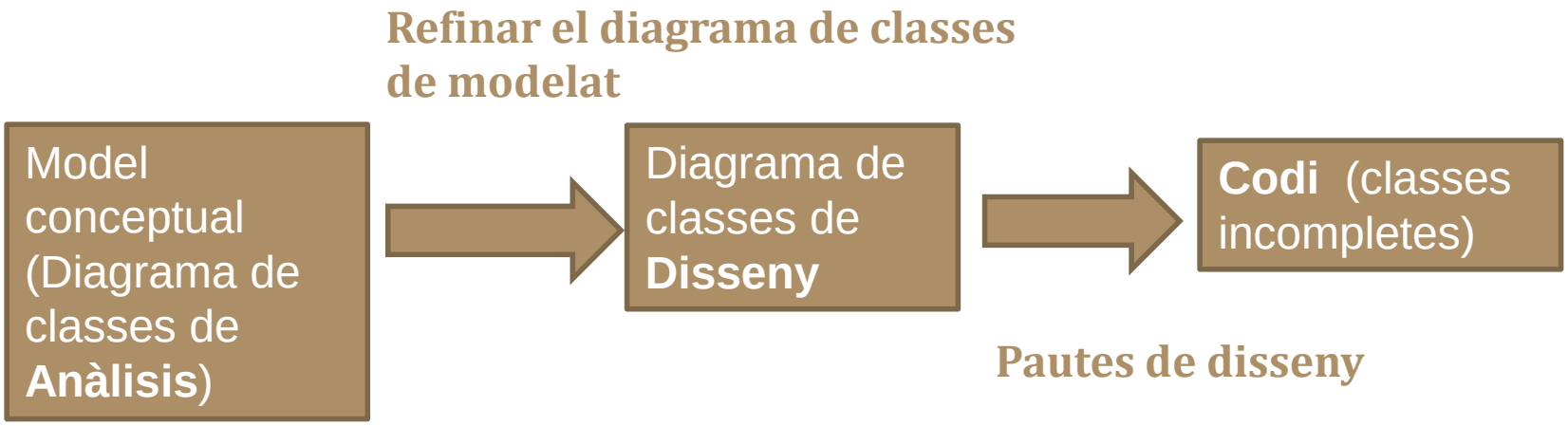
Orientat a la **Solució**

El disseny afig l'entorn de desenvolupament (i llenguatge d'implementació) com un nou element a considerar.

DISSENY D'OBJECTES

Disseny d'Objectes

- Entrada: Modelatge Conceptual
- Eixida: Disseny – Classes dissenyades en llenguatge OO



Decisions i pautes de Disseny

- Refinament del diagrama de classes
  - Crear noves classes
  - Esborrar classes i/o fusionar-les amb altres
  - Crear noves relacions entre classes
  - Modificar relacions existents
    - Restringir la navegabilitat
    - . . .
- Pautes per a...
  - Disseny de Classes
  - Disseny d'Associacions
  - Disseny d'Agregacions
  - Disseny d'Especialitzacions

Diagrama de Classes de Disseny

Diagrama de Classes – Cas d'estudi EcoScooter

Model conceptual (Diagrama de classes de **Anàlisis**)

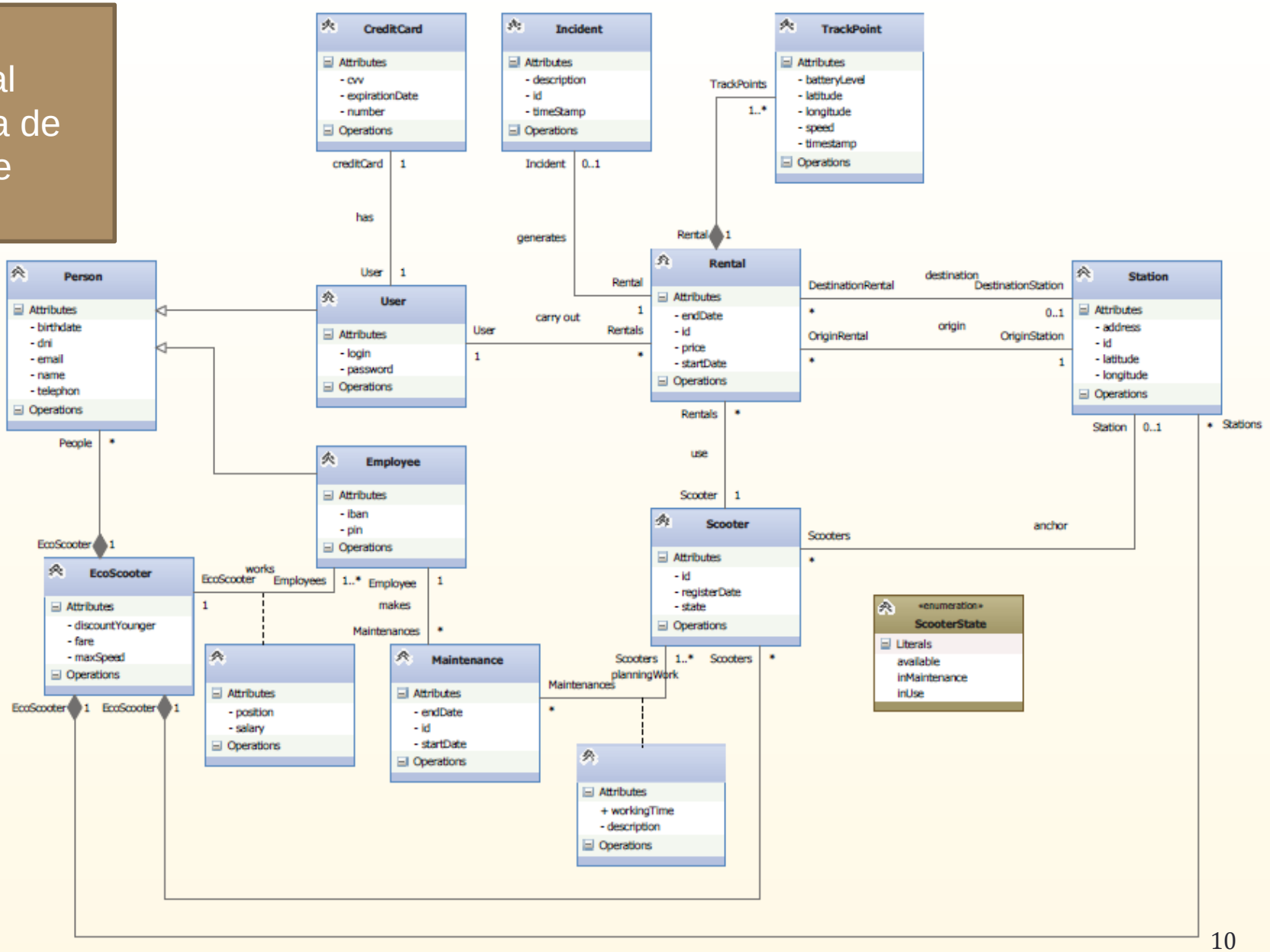


Diagrama de Classes – Cas d'estudi EcoScooter

Diagrama de classes de **Disseny**

Decisions de disseny:

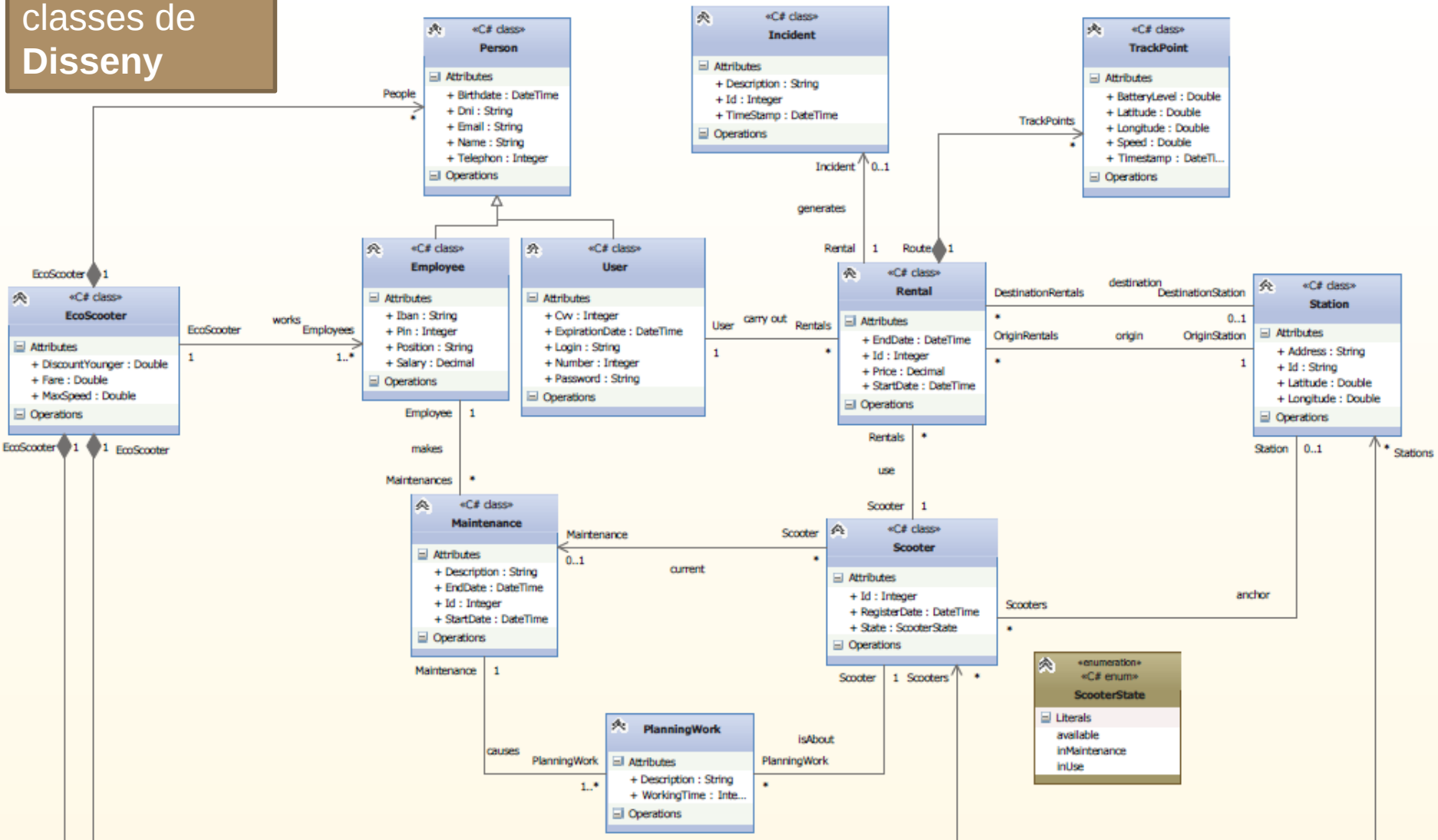
• La classe CreditCard s'elimina i els seus atributs pasan a User.
• El atribut d'enllaç works (entre EcoScooter i Employee) pasa a la classe Employee
• S'inclou una associació entre Scooter i Maintenance per facilitar la realització d'un informe

• Restriccions de navegació:

o De Rental a Incident
o De EcoScooter a Person
o De EcoScooter a Station
o De EcoScooter a Scooter
o De EcoScooter a Employee
o De Rental a TrackPoint

Diagrama de Classes – Cas d'estudi EcoScooter

Diagrama de classes de **Disseny**

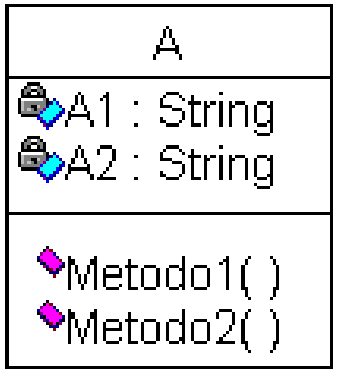


PAUTES PER AL DISSENY D'OBJECTES

Amb exemples en C#

Pautes de Disseny. Classes

Modelatge Conceptual



Disseny en C#

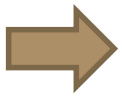
```
public class A
{
 private String A1;
 private String A2;

 public int Metodo1(){…}
 public String Metodo2() {…}

 public void assignar_A1(string a) {…}
 public void assignar_A2(string a) {…}
 public String obtenir_A1() {…}
 public String obtenir_A2() {…}

}
```

Nota: Els mètodes consultors i modificadors (Assignar() /obtenir()) els denominarem *set/get*

Classes (utilitzant propietats C#)

Mètodes clàsics

```
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;
}
```



Propietats de C#

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

Pautes de Disseny. Associacions

Relació un-a-un

Modelatge Conceptual

Disseny en C#

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

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

Associacions

Relació un-a-molts

Modelatge Conceptual

Disseny en C#

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

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

Associacions

Alternativa: mètodes específics per accés a les col·leccions

```
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));
    }
}
```

Col·leccions en C#

- Genèriques
  - List<T>, LinkedList<T>, SortedList<K,V>
  - Stack<T>, Queue<T>
  - Dictionary<K,V>, SortedDictionary<K,V>
  - HashSet<T>, SortedSet<T>
- No genériques
  - Array, ArrayList, SortedList
  - Hashtable
  - Queue, Stack
- Concurrents i altres

Colecciones y estructuras de datos (Ayuda .NET Framework)

Elegir Col·leccions en C#

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>

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

Associacions

Relació molts-a-molts

Modelatge Conceptual

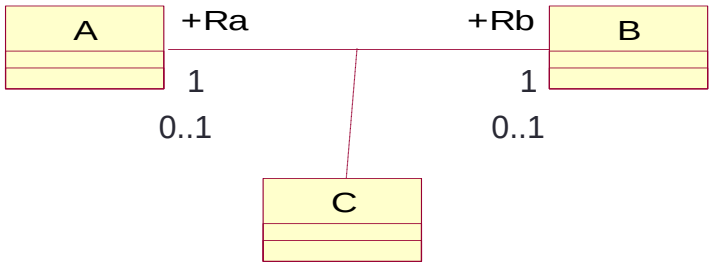
Disseny en C#

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

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

Associacions

Relació un-a-un (Classe Associació)

Modelatge Conceptual



Disseny en C#

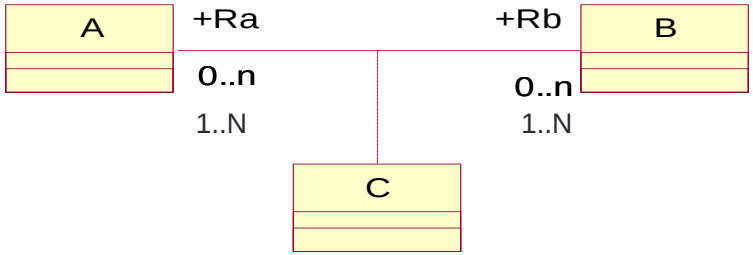
```
public class A
{
  public C Rc {
        get;
        set;
        }
}
public class B
{
  public C Rc {
        get;
        set;
        }
}
```

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

Associacions

Relació molts-a-molts (Classe Associació)

Modelatge Conceptual



Disseny en C#

```
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;
         }
}
```

Agregació /Composició

Agregació un-a-un

Modelatge Conceptual

Agregació un-a-molts

Modelatge Conceptual

Agregació molts-a-molts

Modelatge Conceptual

```
....  // se segueix les pautes ja vistes
```

Atributs d'enllaç (Associació un-a-molts)

Modelatge Conceptual


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

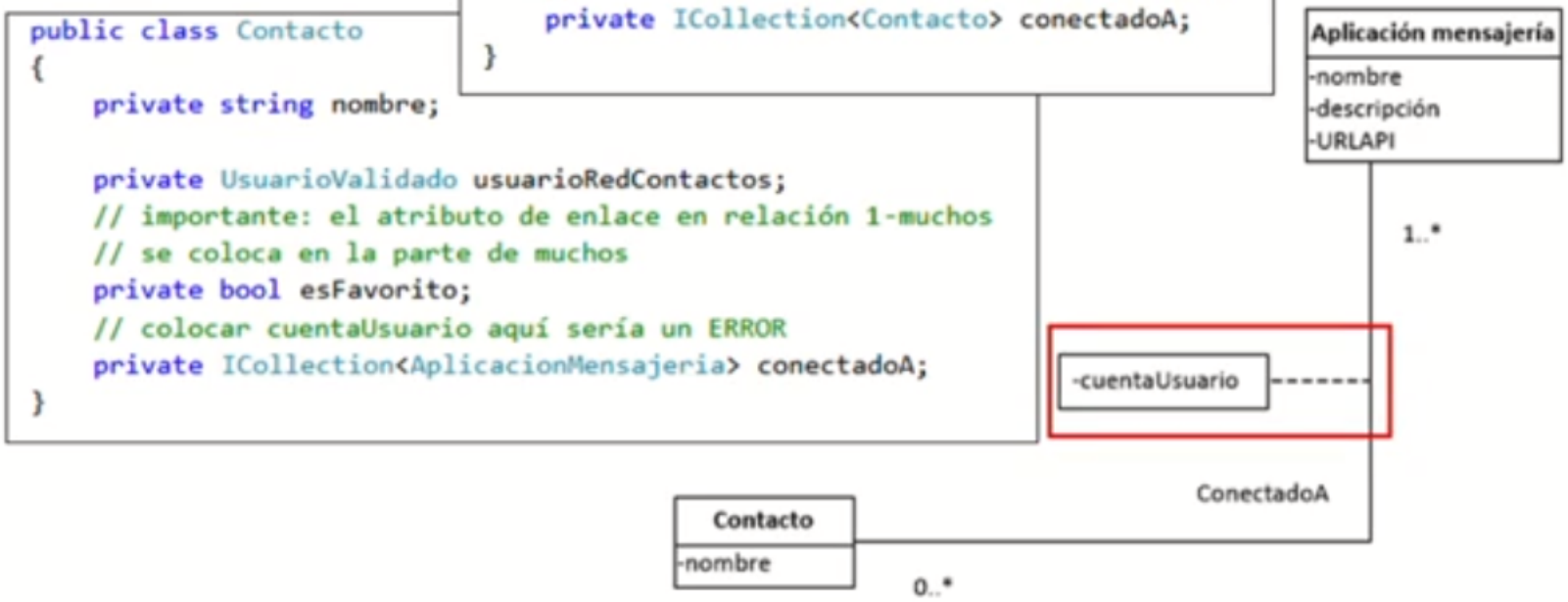
```
public class Contacto
{
    private string nombre;

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

Atributs d'enllaç (Associació molts-a-molts)

Modelatge Conceptual

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

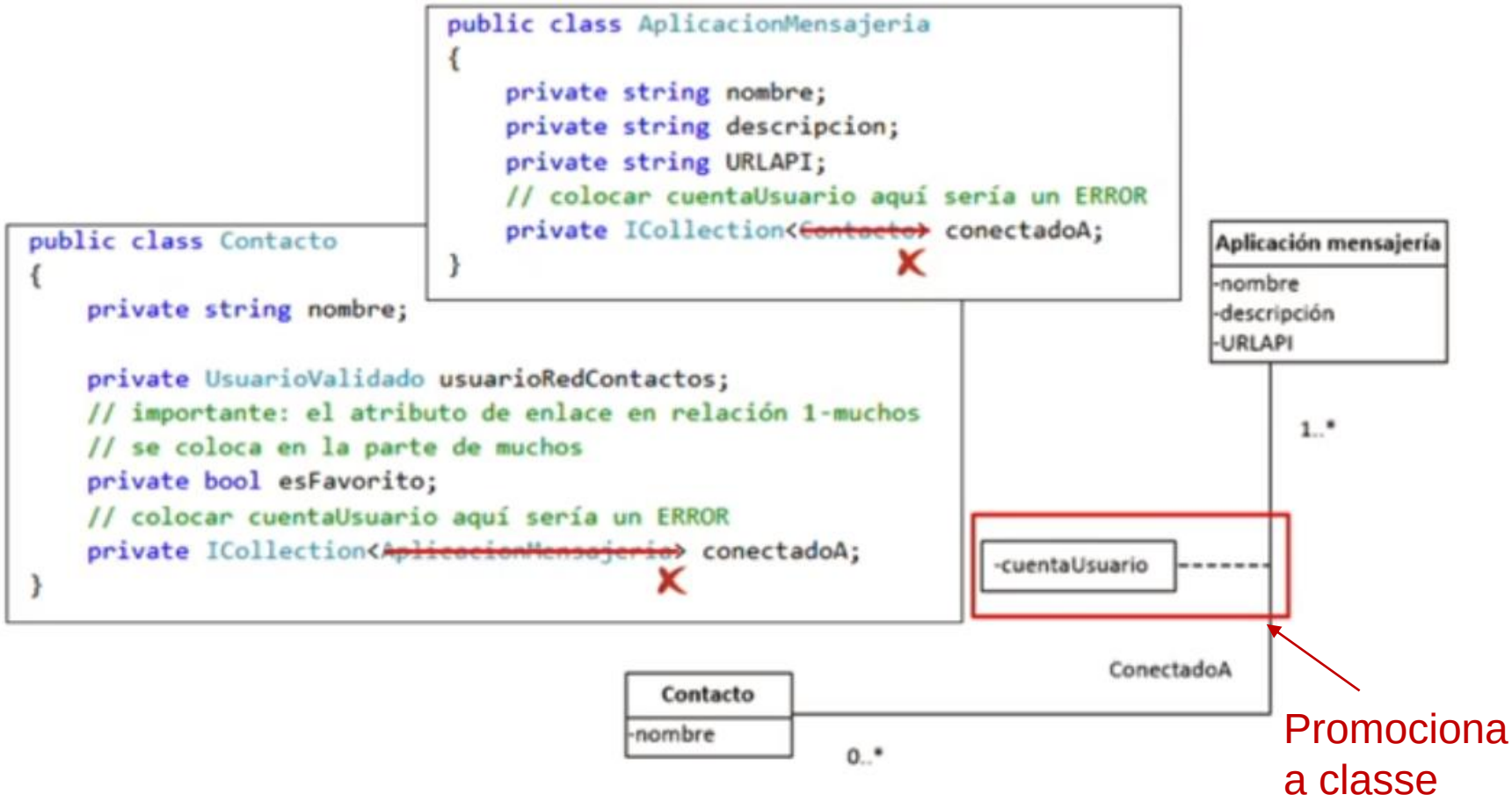
```
public class Contacto
{
    private string nombre;

    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;
}
```



Atributs d'enllaç (Associació molts-a-molts)

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

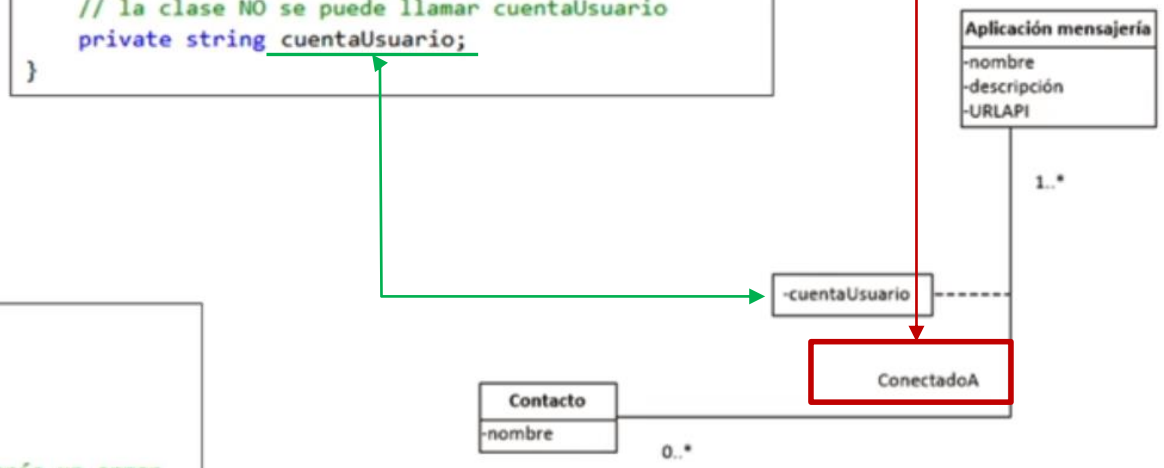
```
public class Contacto
{
    private string nombre;

    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;
}
```



Atributs d'enllaç (Associació molts-a-molts)

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

Classe nova



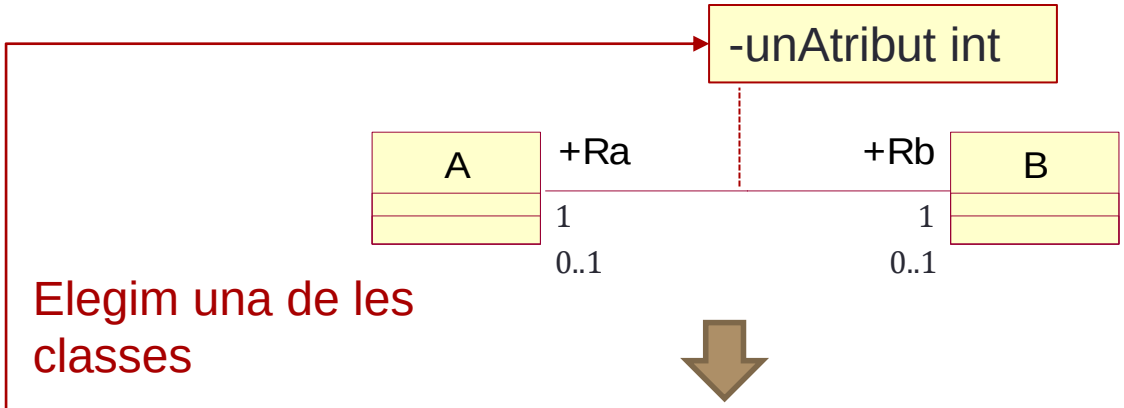

```
public class AplicacionMensajeria
{
    private string nombre;
    private string descripcion;
    private string URLAPI;
    // tener una colección de Contacto sería un error
    private ICollection<ConectadoA> conectadoA;
}
```

✔

```
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;
}
```

✔

Atributs d'enllaç (Associació un-a-un)



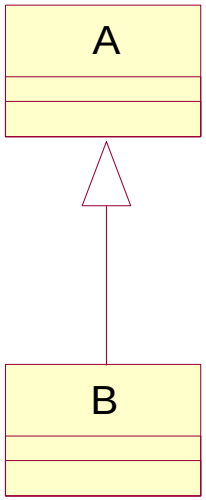

```
public class A
{   public int UnAtribut{
        get;
        Set;
    }
    public B Rb {
        get;
        set;
    }
}
```

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

Pautes de Disseny. Especialització/Generalització

Especialització Simple

Modelatge Conceptual



Disseny en C#

```
public class A
{
  ...
}

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

Es pot jugar amb la visibilitat dels atributs i mètodes segons si volem maximitzar la facilitat d'extensió o l'encapsulació

Compte perquè en C# els modificadors no són els mateixos que en Java i té algunes peculiaritats importants

Modificadors d'Access (C#)

- Paraules clau que es gasten per a especificar l'accesibilitat declarada d'un membre o un tipus.

- `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.

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

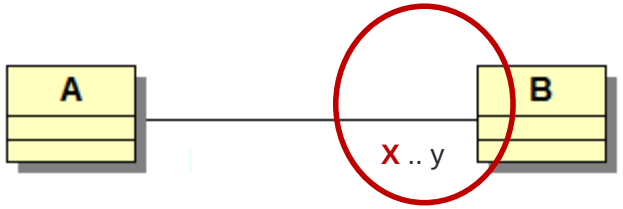
CONSTRUCTORS

Implementació ...

Consideracions sobre els constructors (1/2)

- *Inicialitzar un objecte* suposa donar valors tant al seus **atributs** com als **enllaços amb altres objectes d'altres classes**, si els haguera.

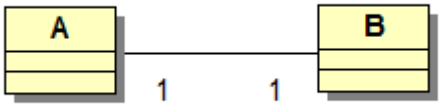
- La **multiplicitat mínima** de les associacions/agregacions determina cóm es realitza la inicialització



x	y	Declaració 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); … }`

Consideracions sobre els constructors (2/2)

- El cas 1/1
  - Quan en els dos extrem d'una associació, la multiplicitat mínima es 1, es crea una dependència circular que no es pot resoldre en un pas.
  - S´ha de implementar-se una inicialització en mes d'un pas de forma "transaccional"...




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

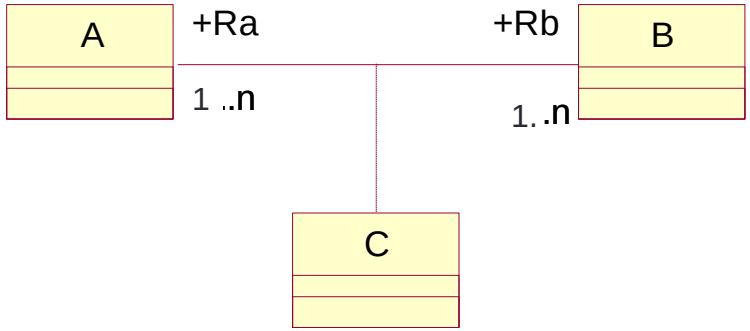
```
public class B {
 A el_A;
 public B(... A el_A)
 {
   this.el_A=el_A;
 }
 ...
 }
```

```
...
//es deu executar com un tot
A un_A=new A(...);          // un A
B un_B = new B(un_A);       // un B
un_A.setEl_B(un_B);         // 1..1
...
```

Constructors en classe associació amb 1/1

- En el cas de tindre una clase associació amb multiplicitat mínima 1 ..... es crea una dependencia circular que es resol de forma similar:

```
public class A {
 public ICollection<C> Rc {get; set;}
 public A(...);
 }

public class B {
 public ICollection<C> Rc {get; set;}
 public B(...);
 }

public class C {
 public A Ra {get; set;}
 public B Rb {get; set;}
 public C(...,A el_A,B el_B);
}
```

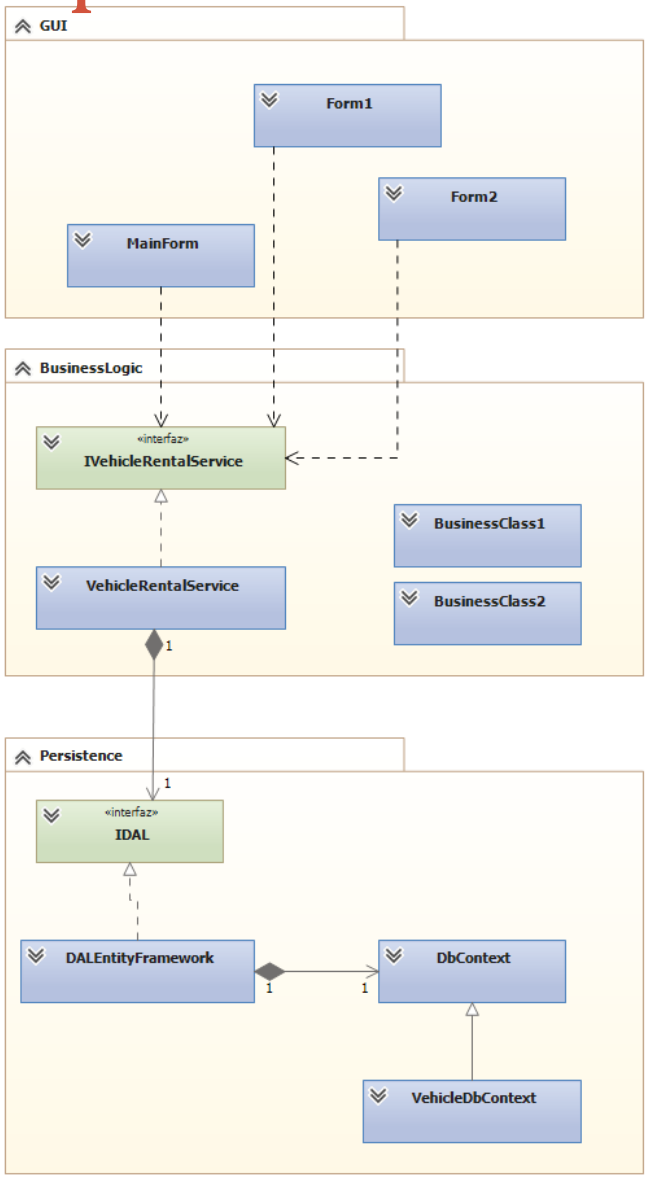



```
...
A un_A=new A(...);
B un_B=new B();
C un_C=new C(un_A,un_B);
un_A.addC(un_C);
un_B.addC(un_C);
...
```

DISSENY ARQUITECTÒNIC

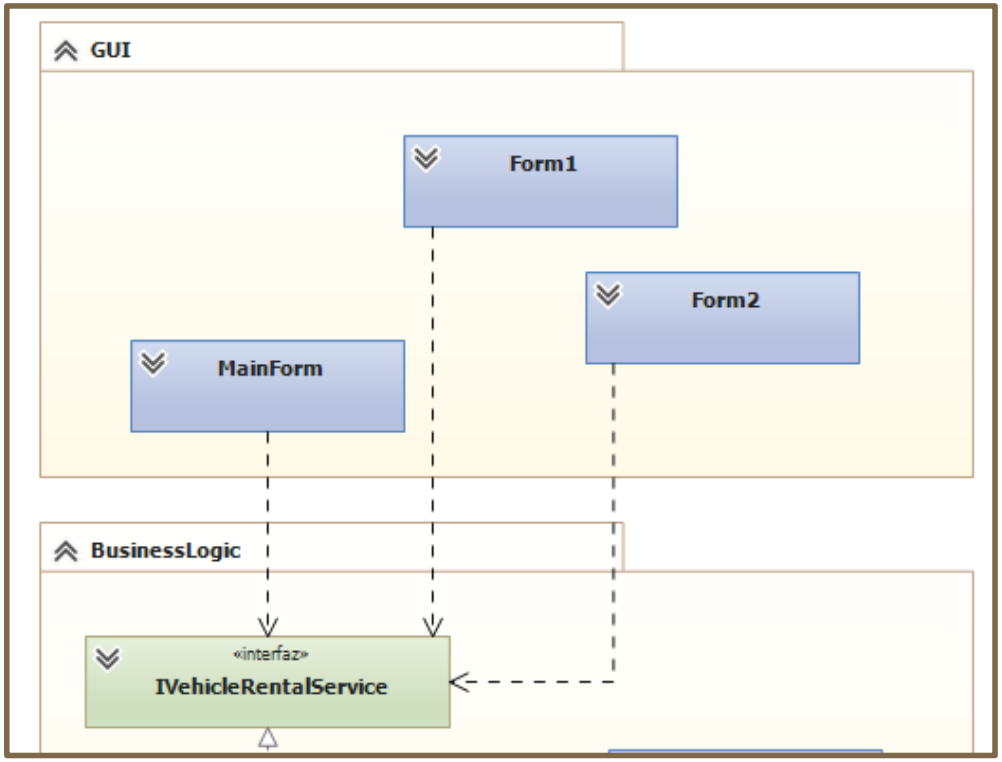
Disseny de la separació de capes

- Seguim una arquitectura multi-capa amb:
  - Presentació (IGU)
  - Lògica de negoci
  - Persistència per accés a la font de dades
- Cas d'estudi: ***VehicleRentalService***



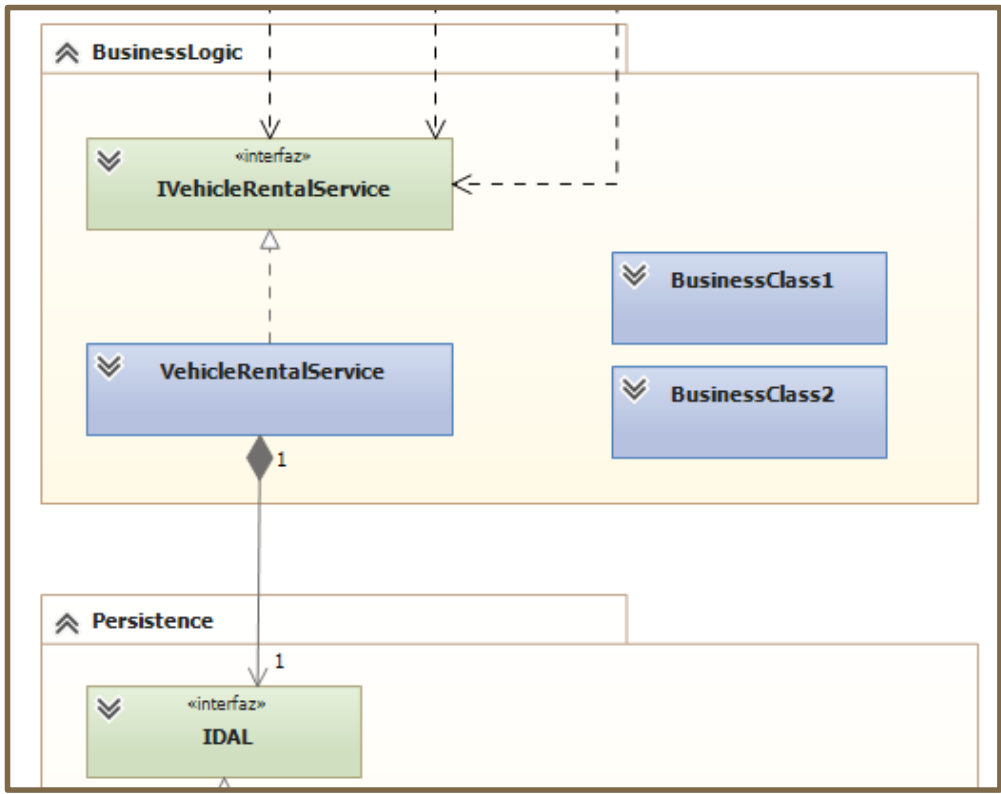
Separació de capes. Presentació

- Conjunt de formularis (un d'ells el **MainForm**)
- **Tots** els formularis accediran als servicis que ofereix la lògica de negoci (en el exemple, mitjançant VehicleRentalService)
- Per tant, el **constructor** de **tots** els formularis necessita una referència a VehicleRentalService
  - Per incrementar la **reutilització** definim una **interfície** IVehicleRentalService que indica el **què**, no el **com**. Així, es podran adoptar **distintes implementacions** i la capa de presentació **no se vorà afectada**



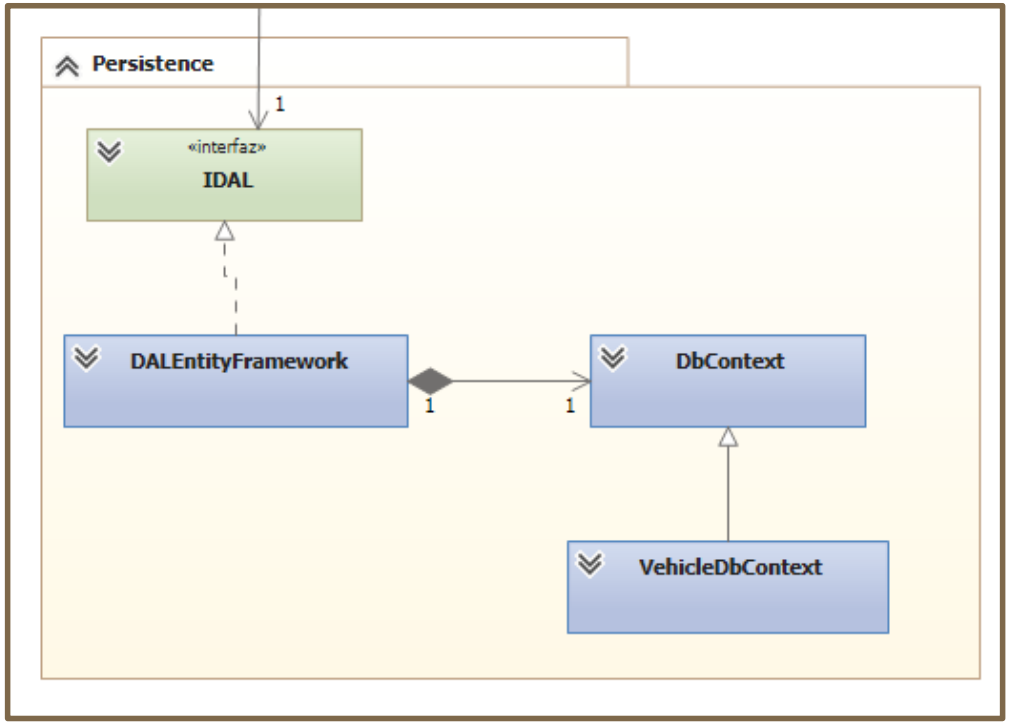
Separació de capes. Lògica de negoci

- Proporciona tots els **servicis** de la nostra aplicació (**casos d'ús**)
- Estos servicis son **identificats** en una **interfície** (en el exemple IVehicleRentalService)
- En poden proporcionar **distintes implementacions** dels servicis d'aquesta interfície (ex. VehicleRentalService o en el futur VehicleRentalService2, VehicleRentalService3…)
  - estes classes **treballaran amb la resta** de las classes de la lògica
  - cada **implementació** pot treballar amb una **capa d'accés a dades** diferent (**DAL**, Data Access Layer), modelada com interfície



Separació de capes. Persistència

- Proporciona l'accés **a la font de dades** (BD relacional, BDOO, arxiu de text, arxiu XML, etc.)
- L'accés a dades s'**identifica** mitjançant una **interfície** (en l'exemple IDAL)
- Es poden proporcionar **distintes implementacions** dels servicis d'aquesta interfície per accedir a les distintes fonts de dades (ej. DALEntityFramework que treballa amb el framework de BD de Visual Studio)
  - però en un futur es podria implementar un DALXML per treballar amb arxius XML i **la capa de lògica no estaria afectada**



Bibliografia

- https://msdn.microsoft.com/es-es. Ajuda on-line per desenvolupar programari OO amb Visual Studio i 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.