BUSINESS LOGIC DESIGN

Chapter 5

Software Engineering

Computer Science School

DSIC – UPV

Goals

- Understand the software design as a set of objects that interact with each other and manage their own state and operations.
- Learn how to derive a design model from a class diagram.
- Learn how to derive methods from sequence diagrams.

Contents

Introduction

Conceptual Modeling *(Analysis)*

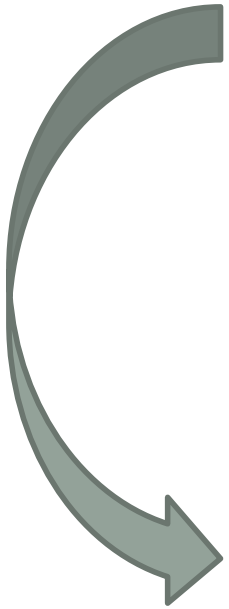
It is the process of constructing a **model** / of a detailed specification
of
A problem of the real world we are confronted with.
It does not **contain** ***design and implementation*** elements

Modeling = Design?

NO

Introduction

Modeling *vs.* Design



Modeling

Problem Oriented

A process that **extends**, **refines** and **reorganizes** the aspects detected in the process of conceptual modeling to generate a **rigurous specification** of the information system always **oriented to the final solution** of the software system.

Design

Solution Oriented

The design adds the development environment and the implementation language as elements to consider.

OBJECTS DESIGN

Objects Design

- Input: Conceptual Modeling – **Class diagram**

** **Refine Analysis class diagram**

- Output: Design – **C# Design**

Design of Classes

Design of Associations

Design of Aggregations

Design of Specializations

Objects Design

** Refine analysis class diagram

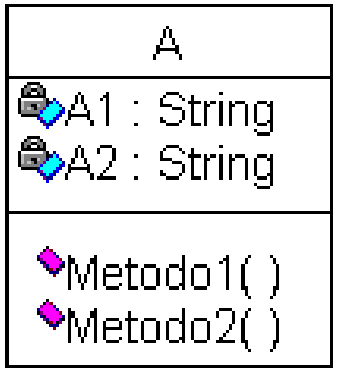
Design decisions

- ✓ Create new classes
- ✓ Remove/Join classes
- ✓ Create new relationships
- ✓ Modify existing relationships
  - ✓ Restrict navegability
- ✓ . . .

Design Class Diagram

Design Patterns. Classes

Conceptual Modeling



Design

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

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

 public void setA1(String a) {…}
 public void setA2(String a) {…}
 public String getA1() {…}
 public String geA2() {…}

}
```

Classes

Methods

```
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;
}
```

Implementation O.O in language

C# Properties

A way of def. an attribute ⇒ we use the name of attribute for operations rather than methods

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

Accesors

Default implementation

```
A a;
…
//set
a.A1="Hello World";
//get
Console.WriteLine($"Value is
{a.A1}");
```

class A

Sets the attribute of a, A1, to "Hello World"

Classes (Properties)

```
using System;

class TimePeriod
{
   private double seconds;

   public double Hours
   {
      get { return seconds / 3600; }
      set {
         if (value < 0 || value > 24)
            throw new ArgumentOutOfRangeException(
                  $"{nameof(value)} must be between 0 and 24.");

         seconds = value * 3600;
      }
   }
}
```

Hidden

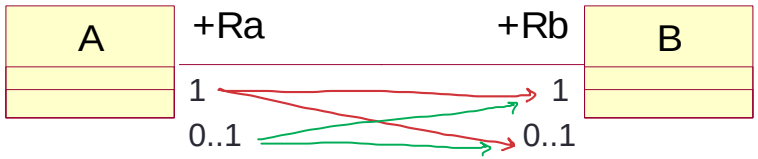
Own implementation

```
class Program
{
   static void Main()
   {
       TimePeriod t = new TimePeriod();
       // The property assignment causes the 'set' accessor to be called.
       t.Hours = 24;

       // Retrieving the property causes the 'get' accessor to be called.
       Console.WriteLine($"Time in hours: {t.Hours}");
   }
}
// The example displays the following output:
//    Time in hours: 24
```

Design Patterns. Associations

1-to-1 Relationship

Conceptual Modeling



Design

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

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

Associations

1-to-Many Relationship

Conceptual Modeling

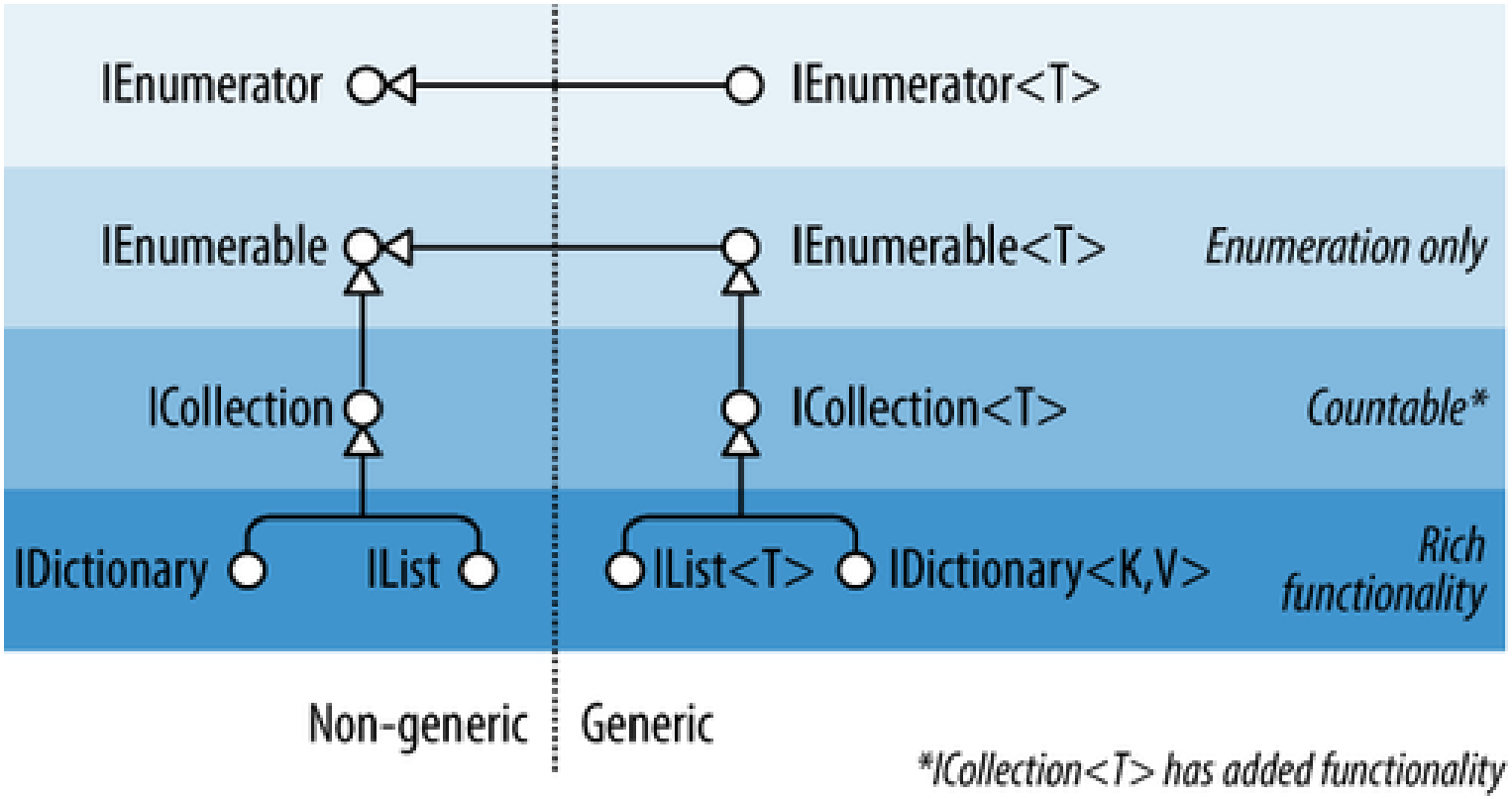

```
public class A
{
 public B Rb {// one-to-one association
        get;
        set;
    }

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

Associations

Alternative: specific methods to access collections

```
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));
    }
}
```

Collections in C#



Collections in C#

- Generic
  - List<T>, LinkedList<T>, SortedList<K,V>
  - Stack<T>, Queue<T>
  - Dictionary<K,V>, SortedDictionary<K,V>
  - HashSet<T>, SortedSet<T>
- Non generic
  - Array, ArrayList, SortedList
  - Hashtable
  - Queue, Stack

Collections and data structures (Help .NET Framework)

Associations

Many-to-Many Relationship

Conceptual Modeling

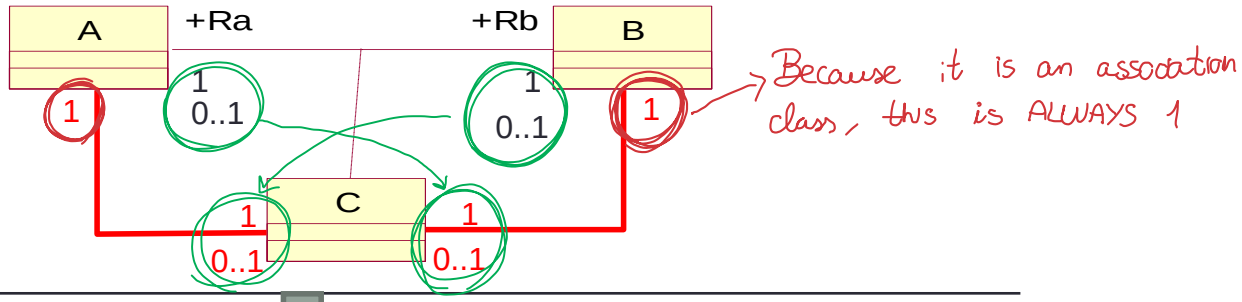
Design

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

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

Associations

1-1 Association (Association Class)

Conceptual Modelling




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

A is not directly connected to B →

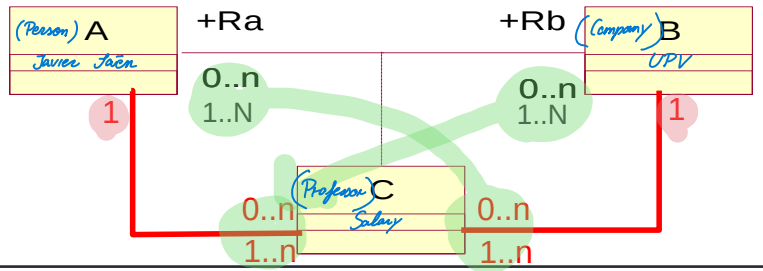
→ A has no B property

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

Associations

Many-to-Many Association (Association Class)

Conceptual Modeling



C is always related to 1 A and 1 B, thus its cardinality is 1 towards A and B

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

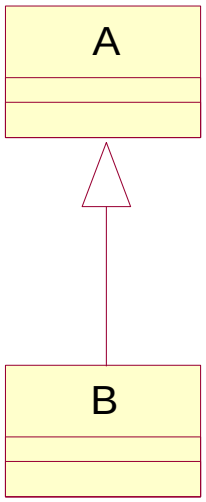
```
public class C
{
    public A Ra {
         get;
         set;
         }
    public B Rb {
         get;
         set;
         }
}
```

Design Patterns. Aggregation/Composition

Same patterns as with associations

Specialization/Generalization

Conceptual Modeling



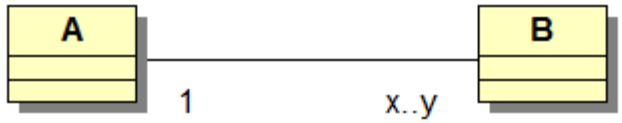
Design

```
public class A
{
 ...
}

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

DESIGN OF CONSTRUCTORS

Considerations about constructors

- Initializing an object results in giving values not only to attributes but also to links with objects of other classes.
- The minimum cardinality of associations/aggregations determines how the initialization is done.



Will determine the constructor

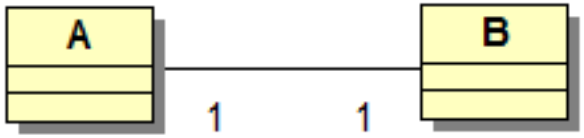
x	y	Declaration in A	Constructor of A
0	1	`public B Rb { get; set; }`	`public A(…){…};` → NOTHING: A doesn't need a B always, we don't put it
1	1		`public A(…, B b, …) { this.Rb = b; … }` → A NEEDS a B inside
0	N	`public ICollection<B> RbRb { get; set; }`	`public A(…){ Rb=new List<B>; … }`
1	N		`public A(…, B b, …){ Rb = new List<B>; Rb.Add(b); … }`

Constructors in one-to-one associations

- In this case a circular dependency is created that cannot be resolved in one step
- An initialization in two steps is implemented (transactional)

WHY? Every A needs 1 B, and every B needs 1 A. Which should be initialized first??

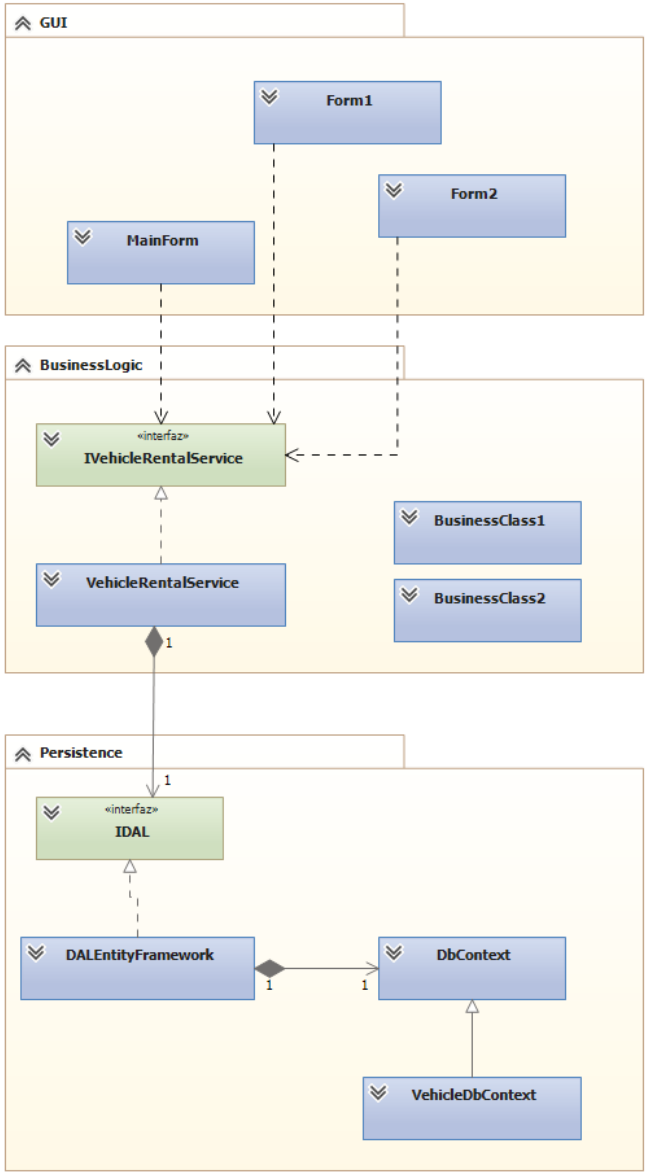
- Homework: How is this initialization done?



ARCHITECTURAL DESIGN

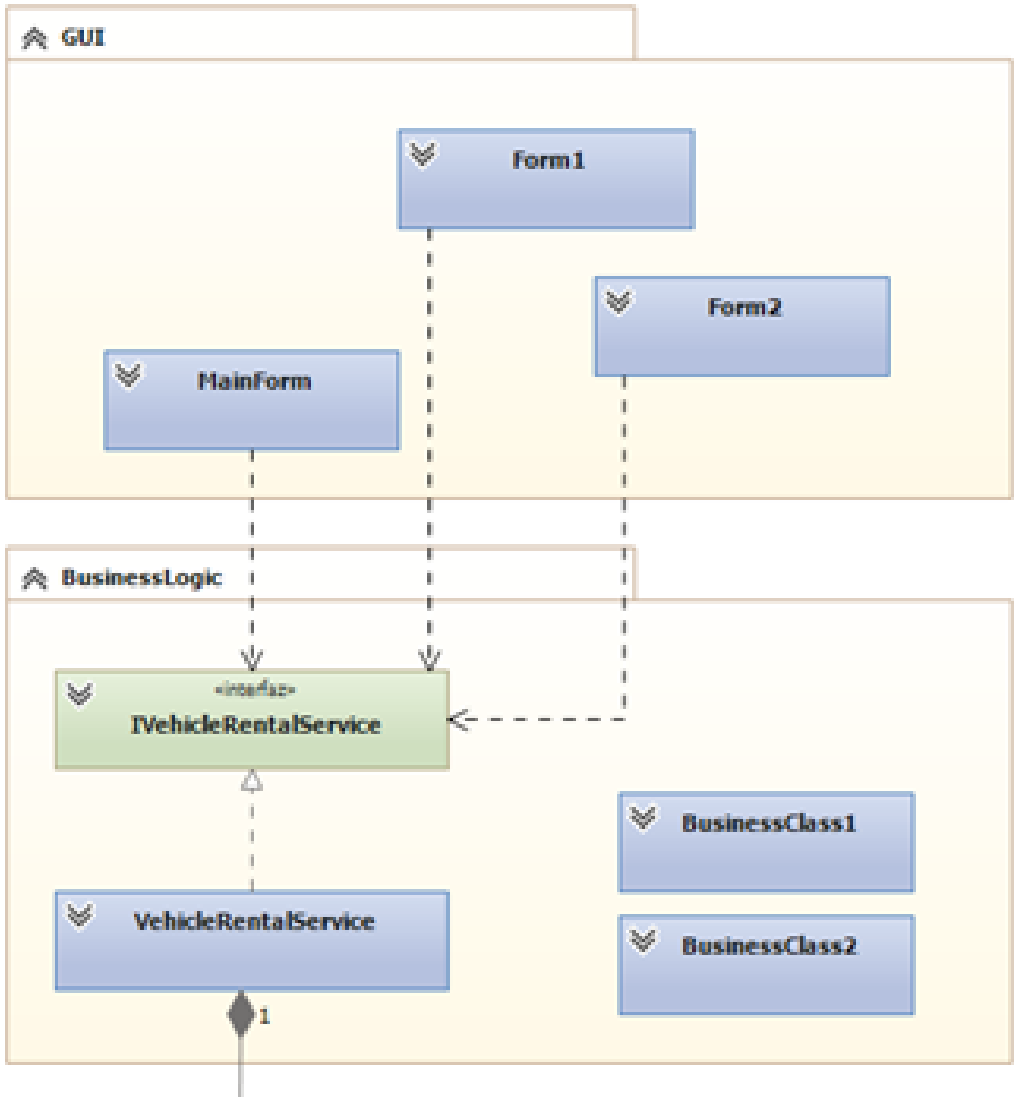
Designing Layers Separation

- We follow a multi-layered architecture with:
  - Presentation (GUI)
  - Business Logic
  - Persistence to access data sources



Layers Separation. Presentation

- The **constructors** of **all** forms need a reference to an object providing business logic services
- To increase software **reuse** we define an **interface (**IVehicleRentalService) indicating **what** (services offered), but not **how** (actual implementations). This way if business logic provides a different implementation in the future, the presentation layer **will not be affected**

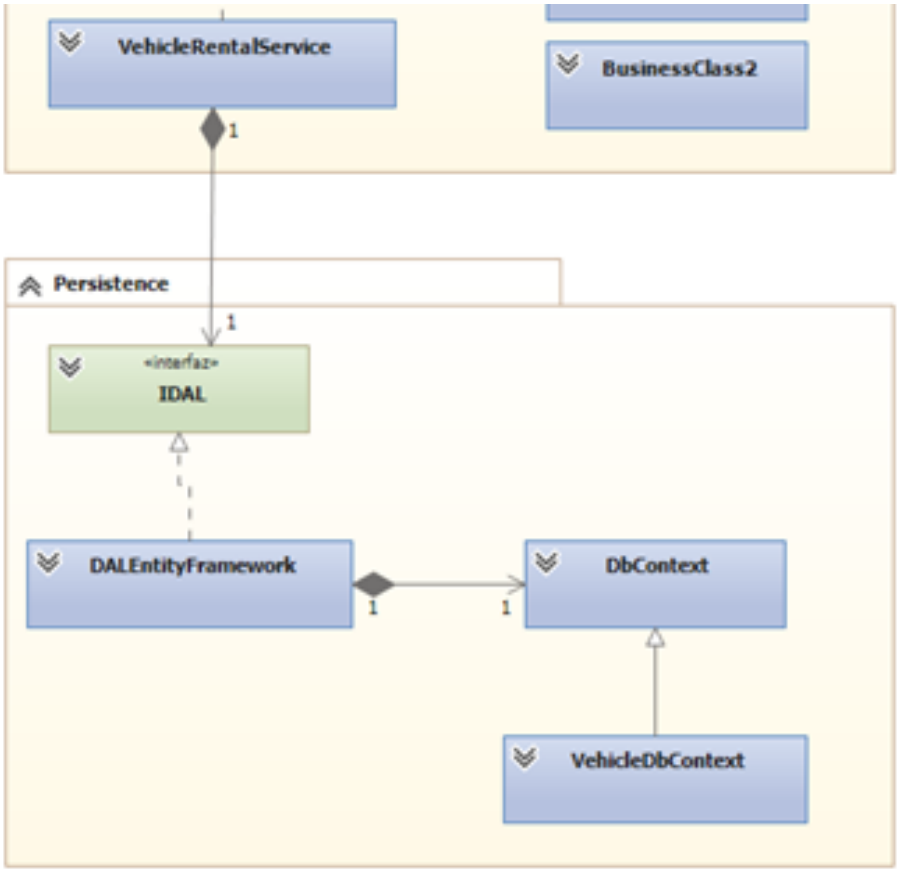


Layers Separation. Business Logic

- Provides all the **services** of our App (**use cases**)
- These services are specified as an **interface** (IVehicleRentalService)
- **Different implementations** of these services may be provided (e.g. VehicleRentalService or in the future VehicleRentalService2, VehicleRentalService3…)
  - The implemented services will handle objects belonging to classes of our model/domain (e.g. Vehicle, Customer, etc.)

Layers Separation. Persistence

- It provides **access to a data source** (relational DB, OODB, XML files, etc.)
- The services provided by the persistence layer are specified again as an **interface** (e.g. IDAL)
- **Different implementations** of the interface may be given depeding on the concrete data source (e.g. DALEntityFramework, DALHibernate, DALXML, etc.)
  - By using an interface any change in the implementation of IDAL **does not affect** the business logic layer



References

- 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.