ISW NOTES

UNIT 1: INTRODUCTION TO SOFTWARE ENGINEERING

1.-INTRODUCTION

In the latest decades software has overcome hardware as a **critical factor for success** and in the last decade, as a result of the success of the Web as a platform and the use of mobile devices, the software industry has experienced a **revolution**. Software is now **more expensive**.

2.-SOFTWARE

Software is:

- **Instructions** that provide an expected function and behaviour when executed.
- **Data structures** that allow programs to adequately manipulate information.
- **Documents** that describe the operation and use of programs.

Software is a **logical element** that it is **developed**. It doesn't break down, it **deteriorates** as a result of changes. Most of it is tailored for **specific purposes**.

High quality software is an agreement with:

- **Functional** and **non-functional requirements**.
- The **documented development standards**.
- The **expected features** exhibited by any software developed professionally.

**(TYPICAL QUESTION IN THE EXAM)

The classification of the quality factors of software takes into account three important aspects of a software product.

1. Its operational features.
2. Its capability to support updates.
3. Its adaptability to new environments.

These must be measured direct or indirectly during the whole development process.

The **quality factors** are:

- **Operational features**:
    - **Correctness**: Does it do what I want?
    - **Reliability**: Is it reliable all the time?
    - **Efficiency**: Will it run in the HW platform efficiently?
    - **Integrity**: Is it safe?
    - **Usability**: Is it designed to be used?

- **Supporting updates**:
  - **Maintainability**: May it be corrected?
  - **Flexibility**: May it be easily changed?
  - **Testability**: May it be verified?
- **Adaptability to new environments**:
  - **Reusability**: Is the software reusable?
  - **Portability**: May it be used in another HW or OS?
  - **Interoperability**: May it interact with another system?

*

The most important **software industry problems** are:

- Products are low quality.
- High maintenance and development costs.
- Delivery delays.

Solutions to these problems are:

- **Education**:
  - Formal methods.
  - New development methods and new lifecycles.
- **Diffusion of technological advancements**:
  - New programming paradigms.
  - Architectures, protocols, computation models.
- **Tools investment**:
  - Modern development environments.
  - Documentation generation engines.

3.-SOFTWARE ENGINEERING

**(POSSIBLE QUESTION IN THE EXAM)

Software engineering is a discipline that integrates **methods**, **tools** and **procedures** for the development of Software.

*

The **software process** is a framework for the development of software.

The **management** of a software project is the **first level** of a software development process and it covers all the development process.

UNIT 2: THE SOFTWARE PROCESS

1.-INTRODUCTION. THE SOFTWARE PROCESS

The **software process** is a framework for the development of software. In general, it is associated to the production process, but it includes the management process.

The **development process** is a collection of activities towards the development or evolution of software. It is also known as **Lifecycle**.

The generic activities that are always carried out are:

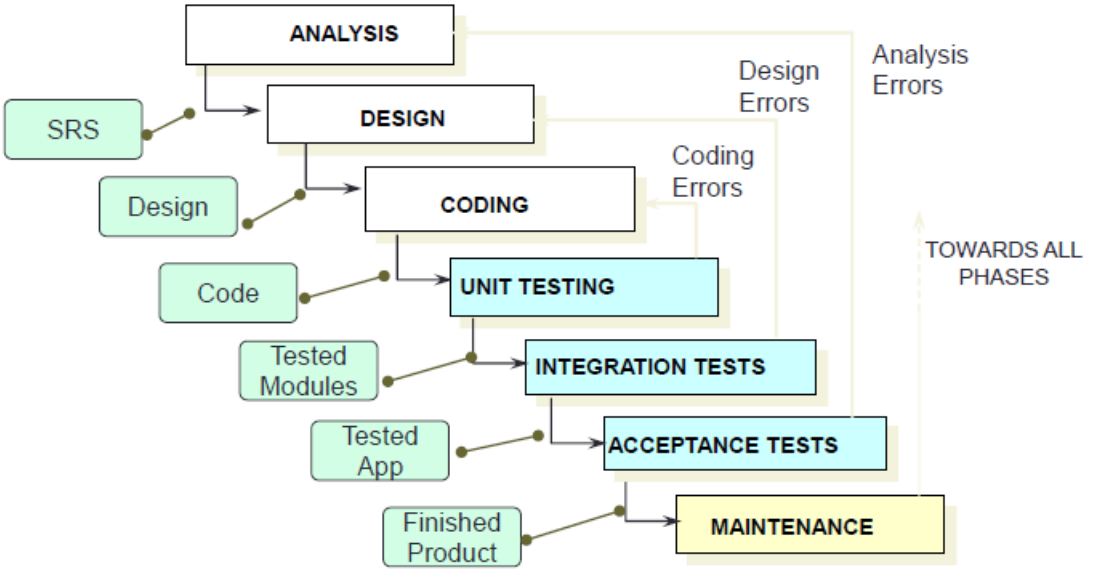
- Specification.
- Development.
- Validation.
- Evolution.
- Analysis.
- Design.
- Implementation.
- Testing.
- Maintenance.

2.-LIFECYCLES

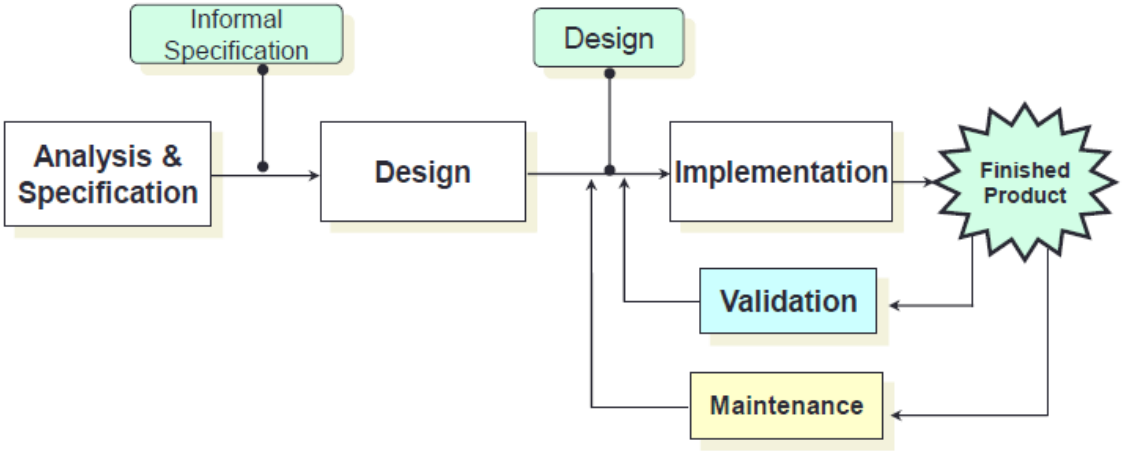
Lifecycle models:

- **Code-and-fix**.
- Classic or **waterfall**.
- Classic with **prototyping**.
- **Automatic Code Generation**.
- Evolution models:
    - **Incremental**.
    - **Spiral**.

2.1.-CLASSIC OR WATERFALL



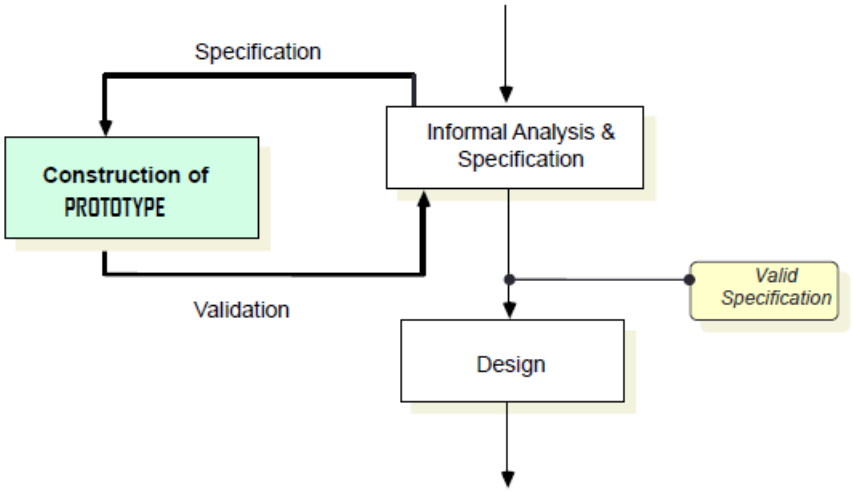
Really, all the validation and maintenance are performed on the source code:



2.2.-CLASSIC WITH PROTOTYPING

A **prototype** is a first version of a product in which only some features are integrated or all of them are featured but unfinished. There are two types of prototypes:

- **Vertical**: some functionality of the system is fully developed.
- **Horizontal**: all views of the system are shown (simulated).



This model helps customers to clearly establish the requirements and helps developers to improve their products.

Advantages:

- It **reduces** the **risk of patching** on the final product code maintenance is not avoided.
- It **helps** both **customers** and **developers** to understand the requirements.

Drawbacks:

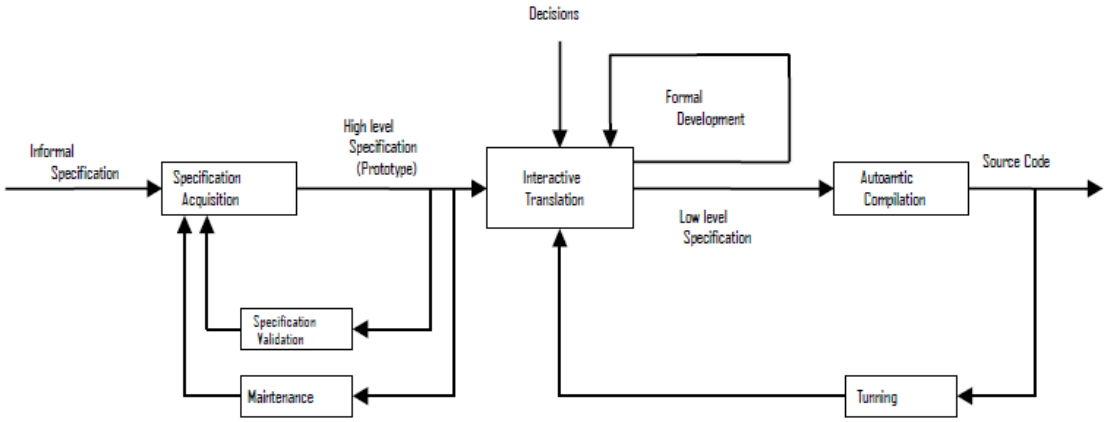
- The customer sees a version of the final product not assuming it is **not robust** and **incomplete**.
- It requires an **additional investment** the invested time may result in loosing market opportunity.
- **Bad decisions** taken during a rapid development of the prototype are usually transferred to the **final product**.

** WHAT IS THE MAIN DIFFERENCE BETWEEN CW AND CP? *

2.3.-AUTOMATIC CODE GENERATION

The goal of **Automatic Code Generation** is to automatize the software development process. Its basic features are:

- Use of **formal specification languages**.
- The **specification** is a **prototype** of the product.
- The **requirements** are discussed by **running** the **specification**.
- The **application** is **derived semi automatically**.



Classic Prototyping vs. Automatic Generation:

CLASSIC Prototyping

- Informal Specification
- Non standard prototype
- Prototype manually built
- Prototype discarded
- Manual implementation
- Code must be tested
- Maintenance on the code

AUTOMATIC GENERAT.

- Formal specification
- Standard prototype
- The specification is the prototype
- It evolves towards the final product
- Automatic Implementation
- No testing
- Maintenance on the specification

Advantages:

- It helps **reducing human errors**.
- It **reduces development costs**.

Its main **drawback** is that it is difficult to use formal languages.

** WHAT IS THE MAIN DIFFERENCE BETWEEN CP AND ACG? *

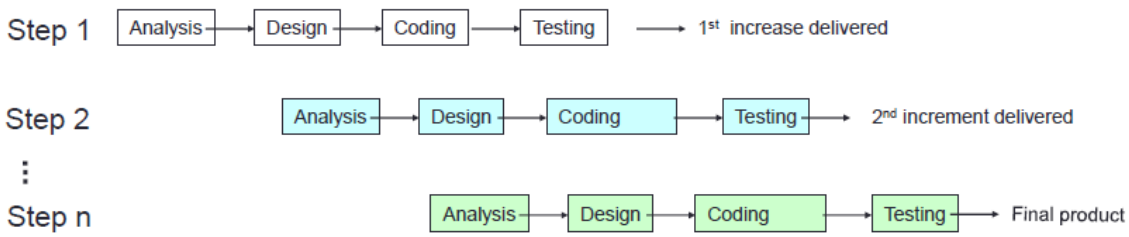
2.4.-EVOLUTIONARY DEVELOPMENT

The **Evolutionary Development** is adaptable to changing requirements. More elaborated versions are built at each iteration.

** WHAT TYPE OF EVOLUTIONARY DEVELOPMENT DO YOU KNOW? *

2.4.1.-INCREMENTAL

The **Incremental Model** is a sequence of applications of the classical model. Each iteration produces a delta of the product. It ends when the final product is delivered.



Advantages:

- **Useful** when **not enough human resources** for a complete deliverable.
- Each **deliverable** may be **evaluated** by the **customer**, so it is highly interactive.

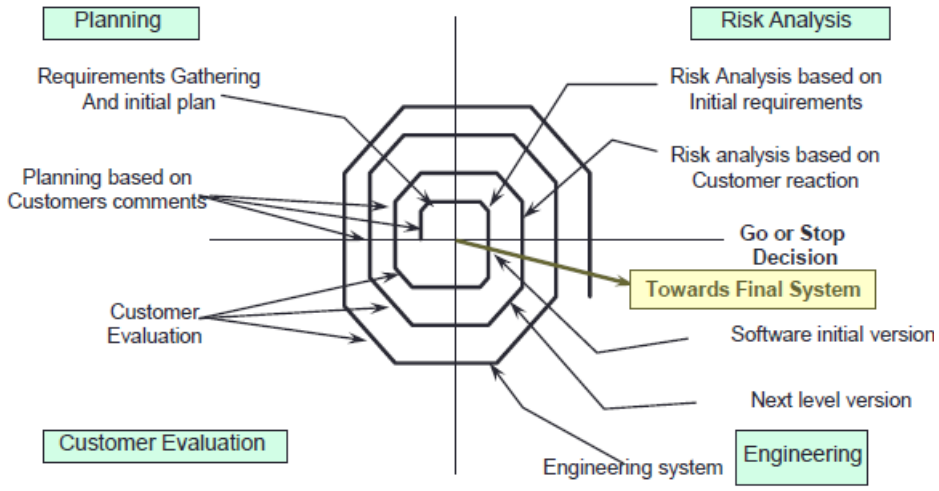
The main **drawback** is that it is **difficult to know the required increase** for each iteration.

2.4.2.-SPIRAL

The **approach** is:

1. **Iterative**.
2. **Interactive**.
3. **Evolutive**.

It introduces **risks analysis** in the development process.



Advantages:

- **Each time more complete versions** of the product are obtained.
- Each version is **evaluated by the customer**, so it is highly interactive.

Drawbacks:

- It is **difficult to assess risks**.
- **Hard to guarantee path** towards the **final product**.

The engineering phase may be adapted to new requirements.

3-METHODOLOGIES

In a software development project, the methodology defines: **Who / What / How / When**. It defines an explicit process of software development.

There is no universal software methodology:

- **Object oriented methodologies**: RUP.
- **Agile methodologies**: XP.
- **Structure methodologies**.

** IT IS THE SAME THING A METHODOLOGY AND A SOFTWARE PROCESS? *

3.1.-RUP

3.2.-AGILE METHODOLOGIES

Its **main features** are:

- The customer is part of the development team (on-site).
- Small teams (<10 members). Working at the same place.
- Few artifacts.
- Few roles.
- Low emphasis on the architecture.
- Heuristic.
- Tolerant with updates.
- Internally imposed (by the team).
- Less controlled process, with few principles.
- No traditional contract or at least very flexible.

UNIT 3: SOFTWARE ARCHITECTURE

1.-INTRODUCTION

Throughout history of software development different strategies to manage complexity, usually related with design at different levels of abstraction, have been used.

A **structure diagram** is a system is partitioned in modules that invoke or provide service to other modules, possibly with data passing in both directions. Modules are a part of a program that implements part of the functionality. A module may be decomposed in terms of other modules of a lower level.

Modular architecture has two designs**:**

- **Preliminary design**: structuring the system in terms of modules (black box), that is, building a structure diagram.
- **Detailed design**: description of process that are implemented by modules (white box).

In **Object Oriented Architectures**:

- **Classes** act like **decomposition units** (structure and behaviour in a module).
- The **structure** of classes is **propagated to the code**. New classes are incorporated when lowering the abstraction level.
- Packages are a way of grouping classes in self-contained components.

The problems that have these architectures are:

- **Approaches based on modules and objects** are **low level** ones.
- They **do not divide the application** in terms of functional blocks but they are mere groupings of code.

2.-THE SOFTWARE ARCHITECTURE

The **software architecture** has to do with the design and implementation of high level structures It is the outcome after assembling a number of different architectural elements in order to adequately satisfy both functional and non-functional requirements such as trustability, scalability, portability and availability.

In the description phase of the **Software Architecture** the system must be organized in terms of **subsystems**. Many times, the architecture is based on other similar previously developed systems by means of **architectonic patterns**.

Types of **systems**:

- **Distributed systems**: a software system in which information processing is distributed among different computing nodes.
- **Personal systems**: non distributed systems that are designed to be run in a personal computer or workstation.
- **Embedded systems**: information systems (hardware + software), usually real time ones integrated in a more general engineering system that perform functions of control, processing and/ or monitoring.

Distributed Systems Architectures:

Multi-processing architectures: the system consists of multiple processes that may or may not be run in different processors.

Client /Server architectures: the system is seen as a set of services that are provided to client applications by server applications. Client and server applications are handled separately.

Distributed objects architectures: the system is seen as a set of interacting objects whose location is not relevant. There is no distinction between a provider of a service and a consumer.

3.-CLIENT-SERVER ARCHITECTURE

Client-server architecture divides an application into two components which are run in one or more devices:

- The **server** (S) is a service provider.
- The **client** (C) is a consumer of services.

C and S interact by means of a message passing mechanism, like service requests or answers.

**What types of layered architectures do you know*

4.-MULTI-LAYERED ARCHITECTURE

A **layered system** is a sorted set of subsystems each one defined in terms of the ones located below them and providing the implementation base of the systems above.

The **objects in each layer may be independent** recommended although there use to be some dependencies between objects of different layers.

There is a relationship **client /server** between the lower layers providing services and the upper layers using those services.

Layered architectures may be **open** or **closed** depending on the dependencies between layers:

- **Open**: a layer may use characteristics of any layer.
- **Closed**: a layer may only use characteristics of its adjacent lower layer.

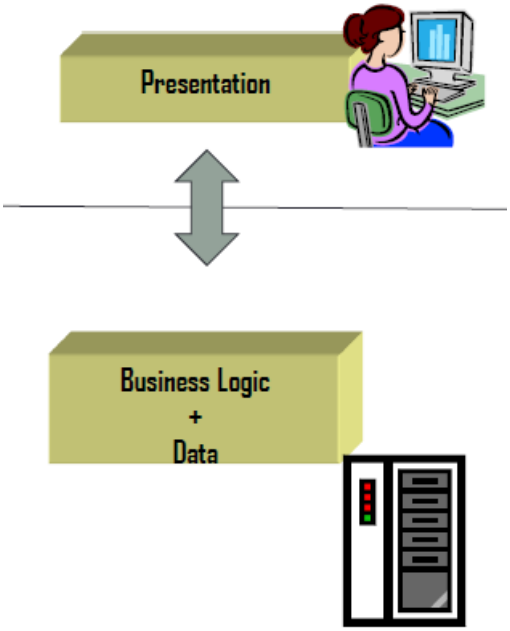
It is recommended to use closed architectures, because there are fewer dependencies between layers and because it is easier to apply changes because the interface of a layer only affects to its immediate upper layer.

There are two types of **2-layers architectures**:

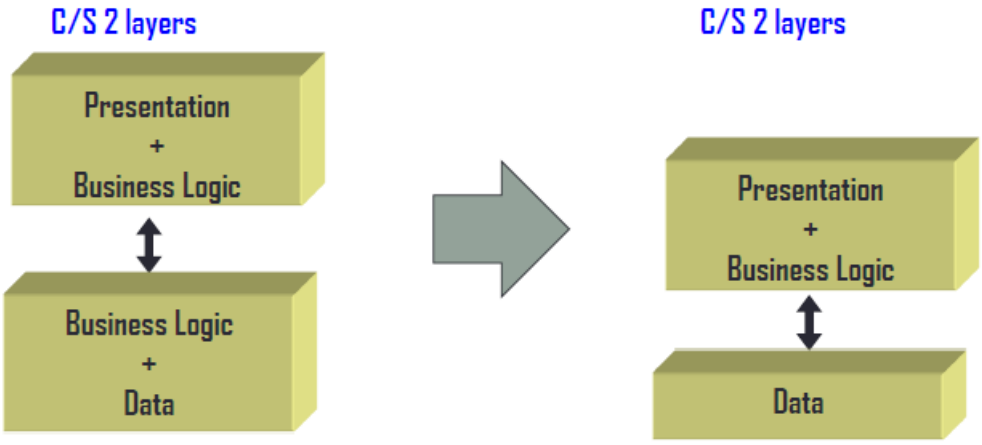
- **Thin clients**.
- **Fat clients**.

A **Thin clients** is useful for:

- **Legacy systems** in which the separation between processes and data management is not feasible.
- **Data intensive applications** (queries and navigation on a DB) with little processing.
- **One application with N platforms**.



In **Flat clients**, part of the logic is moved to the client.



**Difference between layers and tiers*

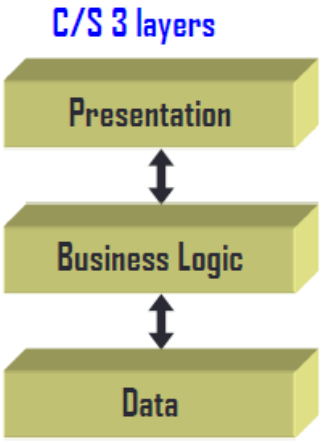
Layer refers to a logical segmentation of the solution whereas **tier** refers to a physical segmentation or location.

The solution to the problems of a **2-layers architecture** with **flat clients** is a **3-layers architecture**. It is structured as:

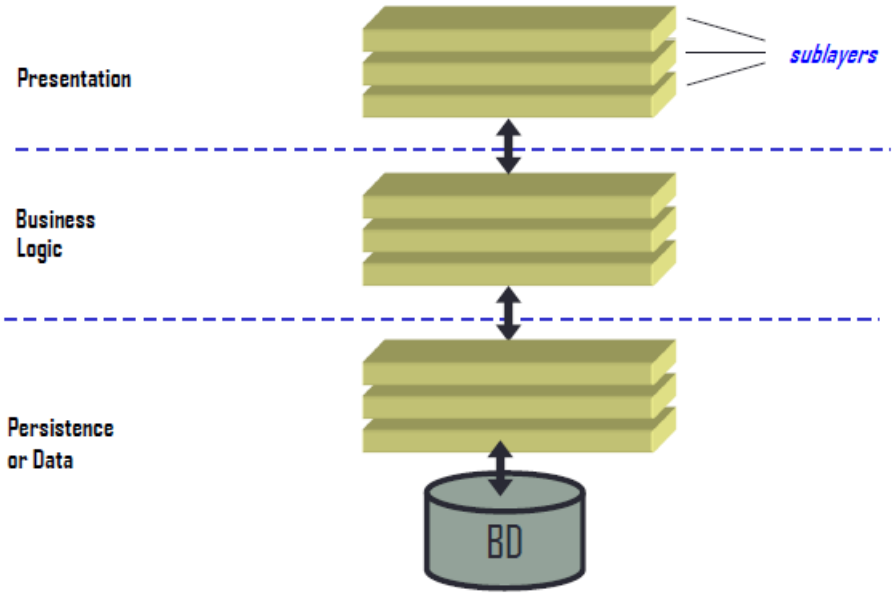
- **Presentation**: presentation of computation results to the user and user input detection.
- **Business Logic**: provide the functionality of the application.
- **Data**: provide persistence to data by means of databases or files.

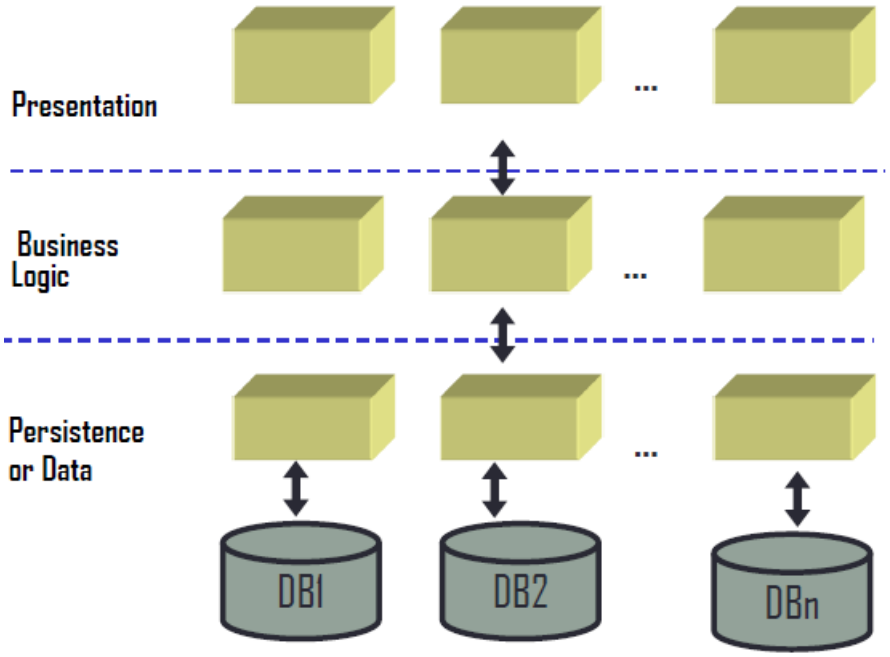
Its main **advantages** are:

- Isolate business logic in a separate component.
- Distribution of layers in different machines or processes.
- Possible parallel development.
- Assigning resources to each layer.
- SOFTWARE REUSE.



There are some **variations** of this architecture:




Presentation
Business
Logic
Persistence
or Data
DB1
DB2
DBn

UNIT 4: UML STRUCTURAL MODELS

1.-INTRODUCTION

The **objects model** or the **classes diagrams** reflect the static structure of the system. It is the **main tool** of most OO methods. An object model contains **interrelated classes** by means of associations organized as aggregation and generalization and specialization hierarchies. Sometimes it is useful to use diagrams that contain objects, **instances diagrams**.

The **object identifier** (oid) is a feature to differentiate two occurrences that have the same state. During analysis it is assumed that objects have an identity. During implementation there must be an **identification mechanism**:

- **Memory addresses programming languages**.
- **Combination of attributes values**.
- **Unique names** ("surrogates").

2.-CLASSES

A **class** is the description of a group of objects with similar structure, behaviour and relationships.

A **class** is an abstraction. An **object** is a concrete realization of this abstraction.

3.-ATTRIBUTES

An **attribute** is a property of a class identified with a name and describes a range of values. They may be represented showing only their names.

The general definition **schema** is:

[visibility] Name [: Type] [=initial value]

Allowed types are (integer, real, char, string, etc.), no object types. Attributes do not include references to other objects, these references are represented as links. In the objects model attributes acting as objects identifiers should not be present.

4.-OPERATIONS

An **operation** corresponds to a service that may be required to any object of the class. It is a function or transformation that may be applied to objects. A **method** is the implementation of an operation.

Operations are defined as follows:

[visibility] Name([parameters]) [: return_type]

Operations that **change** the **state** of **objects** are defined as having **side effects**. **Operations** that do **not** have **side effects** and just **calculate** a **functional value** are called **queries**. Queries return the values of attributes.

An attribute is derived if its value is obtained in terms of the values of other attributes. The notation for derived attributes is:

/Attribute_Name : Type

5.-ASSOCIATIONS AND LINKS

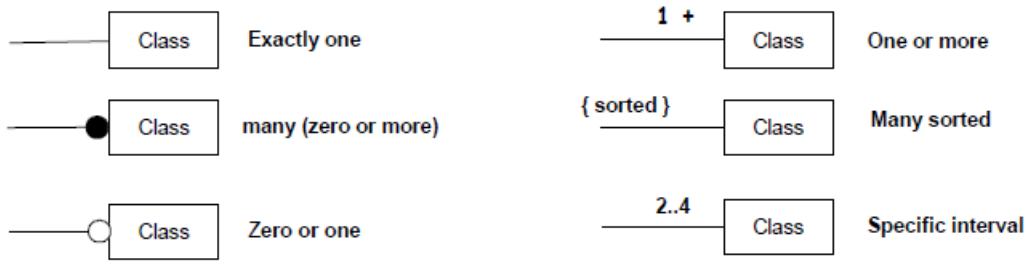
A **link** is a physical or conceptual connection between objects. An **association** is a structural relationship to show that the objects of an element are linked to the objects of another element. Associations are represented in class diagrams whereas links appear in instances diagrams.

Each association in a class diagram corresponds to a collection of links in the instances diagram. Given an association between two classes, it may be navigated in both directions. Binary associations are those connecting two classes.

During analysis, references to objects are represented as links. Similar links are represented as associations.

6.-CARDINALITY

The **cardinality** defines the number of instances of another class that may be related to an instance of a given class.



Cardinality in OMT

1	One and only one
5	Exactly five
0..1	zero or one
M..N	from M to N
*	from 0 to many
n	from 0 to many
0..*	from 0 to many
1..*	from 1 to many

Cardinality in UML

7.-ROLES

Roles are names that define the role played by a class in a relationship. Roles are used to navigate associations. They are mandatory to distinguish reflexive associations. They are mandatory to distinguish associations between same pair of classes.

**NO QUALIFIERS*

8.-ASSOCIATION AS CLASSES

In an association between two classes the relationship itself may have attributes. These are represented as **association classes**, that always have **cardinality equal one**.

9.-AGRREGATION

An **aggregation** is a type of relationship with additional semantics. This relationship is used to model the semantics "**part_of consists_of**". The properties of the aggregation are:

- **Transitive** (if A is part of B and B is part of C then A is part of C).
- **Antisymmetric** (if A is part of B then B may not be part of A).

** Which is the difference between a physical aggregation and a referential aggregation? *

There may be several nesting levels. There are **two types of aggregations**:

- **Inclusive or physical**: each component may belong at most to one container. The destruction of the container implies the destruction of its parts.
- **Referential or catalogue**: components may belong to several containers. Their lifetimes are not correlated.

** What is the difference between generalization and specialization? *

** What is the difference between specialization and inheritance? *

10.-GENERALIZATION AND SPECIALIZATION

Hierarchies of classes allow the management of the complexity in terms of taxonomic classifications. Starting from classes that have in common a number of attributes and operations, using generalization, a more generic class (super class) can be obtained from these initial classes (subclasses). Shared attributes and operations are placed in the superclass.

The **specialization** is the opposite relationship. Starting from a superclass the subclasses are obtained. Subclasses inherit attributes and operations defined in the superclass. Each instance in a subclass is also an instance of the parent class. The specialization relationship is used for conceptual modelling whereas inheritance is a code reusing mechanism during the implementation phase.

There are two types of restrictions:

- **Complete**: all children classes are specified in the model.
- **Incomplete**: not all children specified.

A specialization is dynamic if an object may change to a different class within the hierarchy.

11.-ABSTRACT CLASSES

An **abstract class** has no instances. Its descendant classes have them. It has at least one method without code (undefined methods). Abstract classes are used to define the operations that are inherited by the descendant classes. They provide the protocol (interface) but without giving a concrete implementation.

All concrete subclasses must provide an implementation for an abstract method.

** NOT IN THE EXAM *

12.-RESTRICTIONS

Restrictions are functional relationships between entities in the model. Usually expressed in a declarative form but also natural language can be used. These restrictions may refer to values of the attributes of an object. They are also used between association relationships.

13.-EXCLUSIVE ASSOCIATIONS

An **exclusive association** (or association) consists of set of associations that relate an initial class (source) with several destination classes. Taking an object of the source class, it is at most related with one object of a destination class.

UNIT 5: BUSINESS LOGIC DESING

1.-INTRODUCTION

Conceptual Modelling (Analysis)is the process of constructing a **model** of a detailed specification of **a problem of the real world** we confronted with. It does not contain **design and implementation** elements.

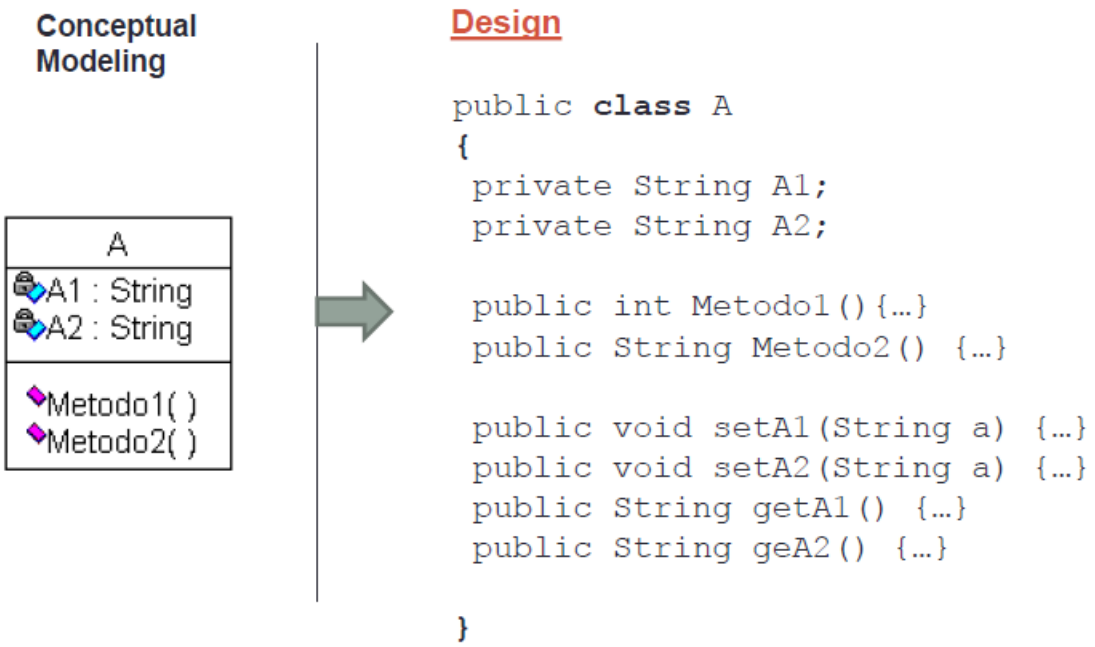
Design is a process that **extends**, **refines** and **reorganizes** the aspects detected in the process of conceptual modelling to generate a rigorous specification of the information system always **oriented to the final solution** of the software system. The design adds the development environment and the implementation language as elements to consider.

2.-OBJECTS DESIGN

The input is the conceptual modelling (**class diagram**) and the output is the design (**C# Design**)

Design patterns:

- **Classes**:



We should change the methods by **C# Properties**:

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;
}
```

C# Properties

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

Accesors

```
A a;
…
//set
a.A1="Hello World";

//get
Console.WriteLine($"Value is
{a.A1}");
```

- **Associations**:

Conceptual Modeling

1-to-1 Relationship

Design

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

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

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;
    }
}
```

Many-to-Many Relationship

Conceptual Modeling

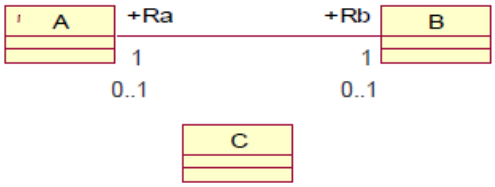
Design

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

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

1-1 Association (Association Class)

Conceptual Modeling



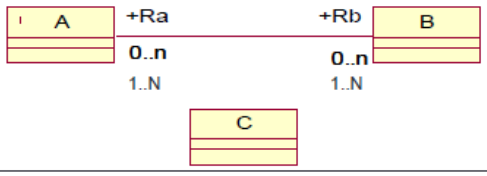

```
public class A
{
  public C Rc {
        get;
        set;
        }
}
public class B
{
  public C Rc {
        get;
        set;
        }
}
```

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

18

Many-to-Many Association (Association Class)

Conceptual Modeling

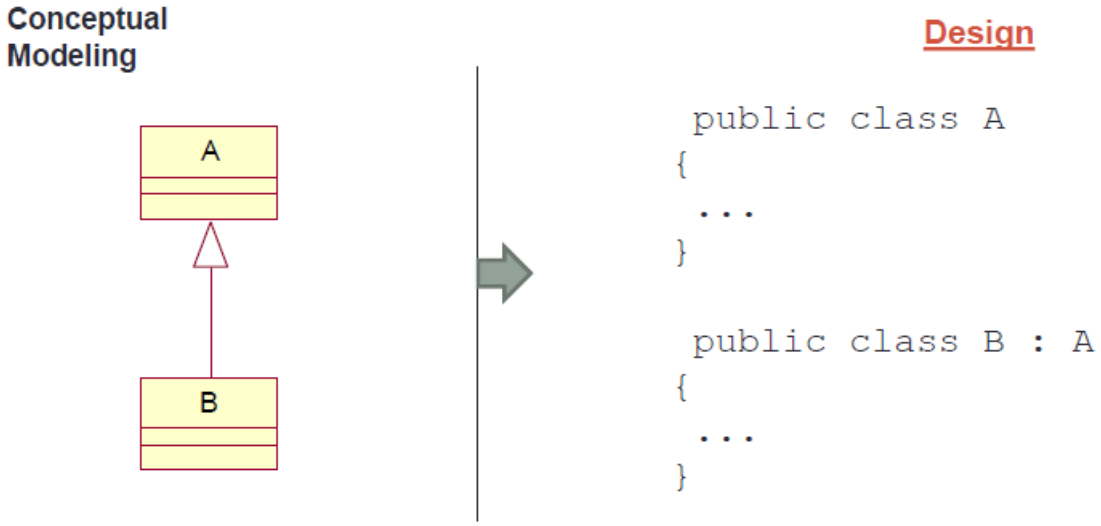



```
public class A
{
  public ICollection<C> Rc {
        get;
        set;
        }
}
public class B
{
 public ICollection<C> Rc{
        get;
        set;
        }
}
```

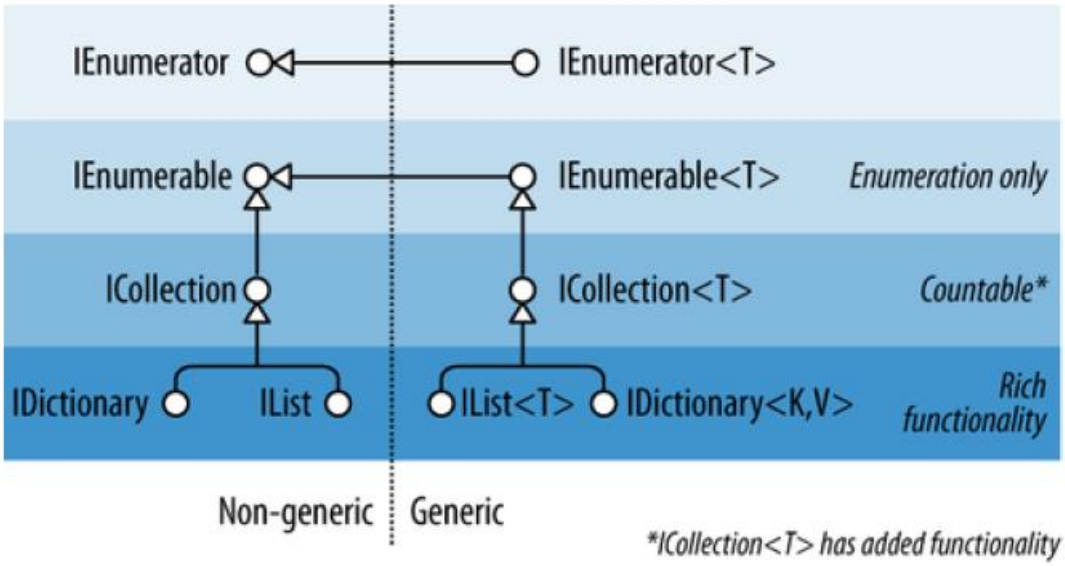
```
public class C
{
   public A Ra {
        get;
        set;
        }
   public B Rb {
        get;
        set;
        }
}
```

- **Aggregation/Composition**: same pattern as with associations.

- **Specialization/Generalization**:



Collections in C#:



3.-DESIGN OF 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.

x	y	Declaration in A	Constructor of A

0	1	public B Rb {
		  get;
		  set;
		}
			public A(…){…};

1	1		public A(…, B b, …) {
			  this.Rb = b;
			  … }

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 have a circular dependency is created that cannot be resolved in one step. An initialization in two steps is implemented transactional.

4.-ARCHITECTURAL DESIGN

We follow a **multi-layered architecture** with:

- **Presentation** (GUI).
- **Business Logic**.
- **Persistence** to access data sources.

In the presentation, the **constructors** of **all** forms need a reference to an object providing business logic services. To increase software **reuse** we define an **interface** 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**.

Business Logic provides all the **services** of our App (**use cases**). These services are specified as an **interface. Different implementations** of these services may be provided. The implemented services will handle objects belonging to classes of our model domain.

Persistence provides **access to a data source**. The services provided by the persistence layer are specified again as an **interface**. **Different implementations** of the interface may be given depending on the concrete data source. By using an interface any change in the implementation of IDAL **does not affect** the business logic layer.