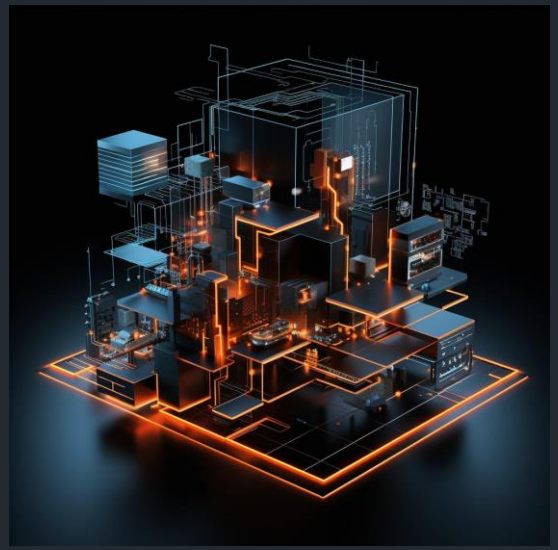
COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

Introducción a la Arquitectura del Software

Arquitectura del Software - **Introducción**

La arquitectura del software abarca la estructura y el diseño de alto y bajo nivel de un programa, estableciendo cómo se construirá y funcionará el software como parte de un **proceso** que toma un problema y crea una solución



Arquitectura del Software - **Introducción**

Características Clave	Arquitectura de Software	Arquitectura de Edificios
Estructura Organizada	Organización de componentes	Distribución de habitaciones y espacios
Módulos y Componentes	Partes principales del software y su interacción	Habitaciones, puertas y ventanas
Flujo de Datos y Control	Cómo se maneja la información y el control	Flujo de agua y electricidad
Escalabilidad y Mantenibilidad	Capacidad para crecer y adaptarse a cambios futuros, facilidad de mantenimiento	Capacidad de expansión, facilidad de mantenimiento y adaptación
Desempeño y Eficiencia	Cómo se garantiza que el software funcione de manera eficiente	Eficiencia energética y uso de recursos

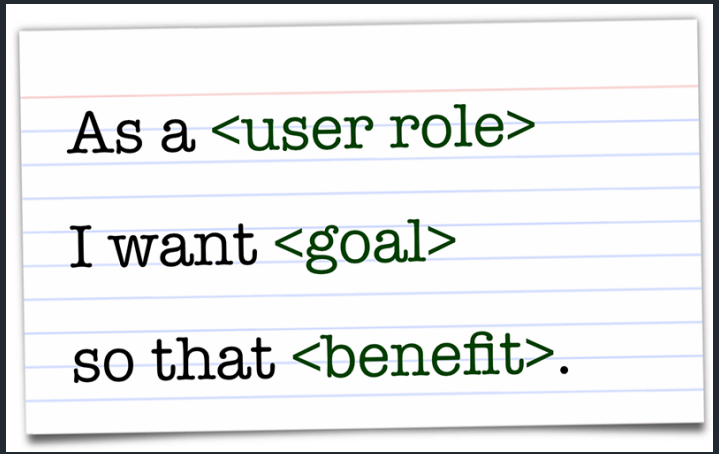
Arquitectura del Software - **Ciclo de vida del desarrollo de sistemas (SDLC)**

Fases comunes del SDLC:

1. **Requisitos** → Captura de necesidades y restricciones
2. Análisis → Comprender a fondo los requisitos
3. **Diseño** → Planificación y estructura del software
4. Implementación → Creación del código real
5. Pruebas → Validación de funcionalidad y calidad
6. Despliegue → Instalación en entorno de producción
7. Mantenimiento → Corrección, mejora y actualización continua
8. Documentación → Registro de detalles y decisiones clave

Arquitectura del Software – SDLC - **Requisitos**

- **¿Qué debe hacer el software?**
- Técnica: “Historias de Usuario”
  - **Como** [rol del usuario], **quiero** [acción] **para** [beneficio]
    - **Como** comprador en línea, **quiero** añadir un artículo a mi cesta de la compra **para** poder comprarlo



Arquitectura del Software – SDLC - **Requisitos**

- Existen 2 tipos de requisitos:
  - **Funcionales**:
    - **Acciones y características que un sistema debe realizar**
    - **Ejemplo**: *"El sistema debe permitir a los usuarios iniciar sesión utilizando su nombre de usuario y contraseña."*
  - **No funcionales**:
    - Definen aspectos de calidad y cómo debe comportarse el sistema
    - Incluyen **reutilización**, **flexibilidad** y **mantenimiento**
    - **Ejemplo:** *"El sistema debe responder a cualquier solicitud de inicio de sesión en menos de 2 segundos."*

Arquitectura del Software – SDLC - **Análisis**

- **¿Cómo se hará lo que el software debe hacer?**
- Análisis de “Historias de Usuario”:
  - Identificar el Objetivo → Clarificar objetivo principal
  - Descomposición de la Historia → Dividir en tareas
  - Requisitos y Restricciones → Identificar condiciones
  - Casos de Uso y Escenarios → Escenarios y casos
  - Interacciones del Usuario → Flujo de usuario
  - Diseño y Arquitectura → Evaluar Integración y diseño
  - Pruebas y Validación → Criterios de validación
  - Estimación de Esfuerzo → Estimación del tiempo
  - Priorización → Determinar importancia

- Análisis de la “Historia de Usuario”: Como comprador en línea, quiero añadir un artículo a mi cesta de la compra para poder comprarlo
  - **Identificar el Objetivo**:
    - Permitir a los compradores agregar productos a la cesta antes de comprar
  - **Descomposición de la Historia**:
    - Tareas:
      - Crear una interfaz de usuario para seleccionar productos
      - Diseñar un sistema de almacenamiento temporal para la cesta de compra
      - Permitir a los usuarios ver los elementos en su cesta y gestionar su contenido.

Arquitectura del Software – SDLC - **Análisis**

- **Requisitos y Restricciones**:
  - Requisitos de **seguridad** para garantizar que la información de la cesta de compra sea segura
  - Restricciones sobre la **cantidad de artículos** que se pueden agregar o sobre la **disponibilidad** de productos
- **Casos de Uso y Escenarios**:
  - Agregar, eliminar o cambiar cantidad de productos en la cesta
- **Interacciones del Usuario**:
  - Selección de productos, visualización de cesta y proceso de compra
- **Diseño y Arquitectura**:
  - Gestión de sesiones
  - Base de datos para gestionar cestas de compra temporales

- **Pruebas y Validación**:
  - Criterios de aceptación:
    - Capacidad de agregar y eliminar productos de la cesta
    - Ver los elementos en la cesta
    - La cesta de compra debe mostrar con precisión la selección del usuario
- **Estimación de Esfuerzo**:
  - 2 a 3 semanas de desarrollo
- **Priorización**:
  - Alta (funcionalidad básica de una plataforma de compras en línea)

Arquitectura del Software – SDLC - **Diseño**

- **Diseño conceptual**:
  - Reconoce los **componentes**, **conexiones** y **responsabilidades** apropiados del producto de software
  - Conceptos de más **alto nivel**
  - **Maquetas** conceptuales (**Mock-ups**)
- **Diseño técnico**:
  - Se basa en el diseño conceptual
  - Dividir los componentes en otros más pequeños que sean lo suficientemente **específicos** como para diseñarlos en detalle
  - Detalles técnicos de **bajo nivel**
  - **Diagramas** técnicos

Arquitectura del Software – SDLC – **Diseño Conceptual**

- Aplicar el pensamiento orientado a objetos a las “Historias de Usuario” rellenas para desglosarlas.
  - Por lo general, los **sustantivos** corresponden a **objetos** del SW
    - Como **comprador en línea**, quiero añadir un **artículo** a mi **cesta de la compra** para poder comprarlo
      - OnlineShopper
      - Item
      - ShoppingCart
  - Los **verbos** ayudan a identificar los requisitos que pueden tener los objetos
    - Como comprador en línea, quiero **añadir** un artículo a mi cesta de la compra para poder **comprar**lo
      - **Add()** - Responsabilidad de ShoppingCart
      - **Buy()** - Responsabilidad de ShoppingCart

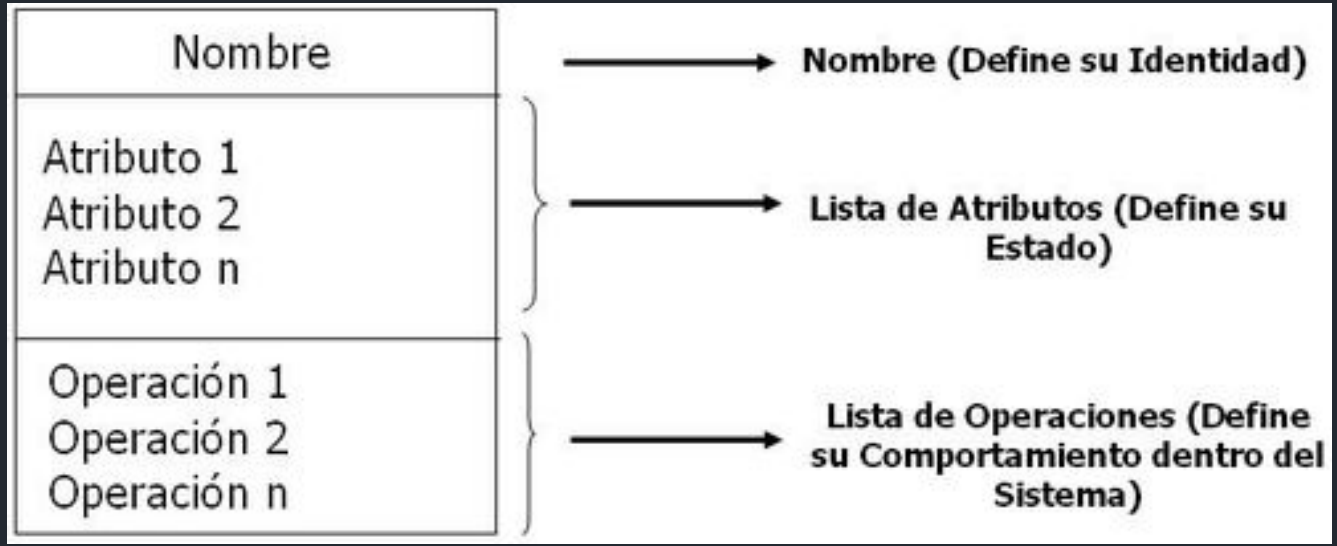
- El último punto es un poco más sutil, pero las “Historias de Usuario” también puede ayudar a descubrir **conexiones** entre objetos
  - Un **comprador en línea** suele **estar asociado** a una **cesta de la compra**
  - La **cesta de la compra** debe **tener capacidad** para varios **artículos**

- Cuatro principios de diseño:
  - **Abstracción:**
    - Simplificar sistemas ocultando detalles complejos y exponiendo solo lo esencial
  - **Encapsulación:**
    - Agrupar datos y métodos en una clase, restringiendo el acceso directo a los datos
  - **Descomposición:**
    - Desglosar un sistema complejo en partes más pequeñas y manejables
  - **Generalización:**
    - Definir una jerarquía de clases para reutilizar y compartir características comunes

Arquitectura del Software – SDLC – **Diseño Técnico**

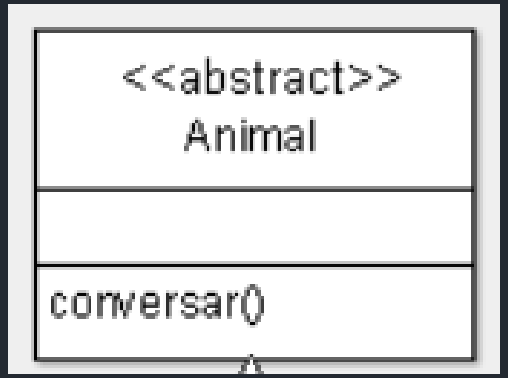
- Diagramas gráficos para visualizar y diseñar sistemas software
- UML (Unified Modeling Language)
- Dos tipos principales de diagramas UML:
  - Diagramas de **estructura**
  - Diagramas de **comportamiento**
- Diagramas principales
  - **Diagrama de clases** (estructura)
  - Diagrama de secuencia (comportamiento)
  - Diagrama de casos de uso (comportamiento)
  - Diagrama de actividad (comportamiento)
  - Diagrama de estados (comportamiento)
- Todos los diagramas

- Diagrama de clases:
  - Representa clases y relaciones en la estructura de un sistema SW
  - **Abstracción**:



- Diagrama de clases:
  - **Abstracción**:
    - Nombre: el mismo que el nombre de la clase de C#
      - OnlineShopper
    - Lista de atributos: **<nombre de variable>: <tipo de variable>**
      - userName: string
    - Lista de operaciones: **<nombre>(<lista de parámetros>) : <tipo de retorno>**
      - AddProduct(product: Product) : void

- Diagrama de clases:
  - **Abstracción**:
    - Clases abstractas:

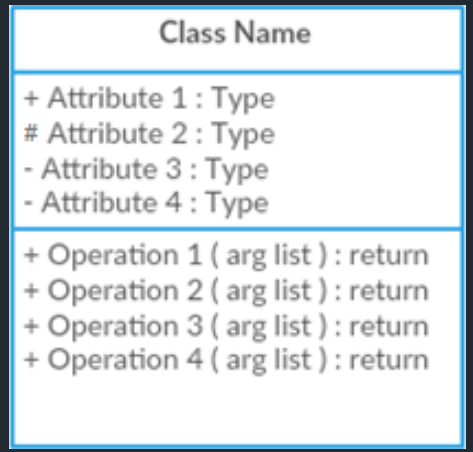


OnlineShopper
userName: string password: string
Login() : void SelectItem() : void AddItemToCart(item: Item) : void RemoveItemFromCart(item: Item) : void Checkout() : void

Item
name: string price: double stockQuantity: int
GetName() : string GetPrice() : double GetStockQuantity() : int

ShoppingCart
items: List<Item>
AddItem(item: Item) : void RemoveItem(item: Item) : void CalculateTotalPrice() : double

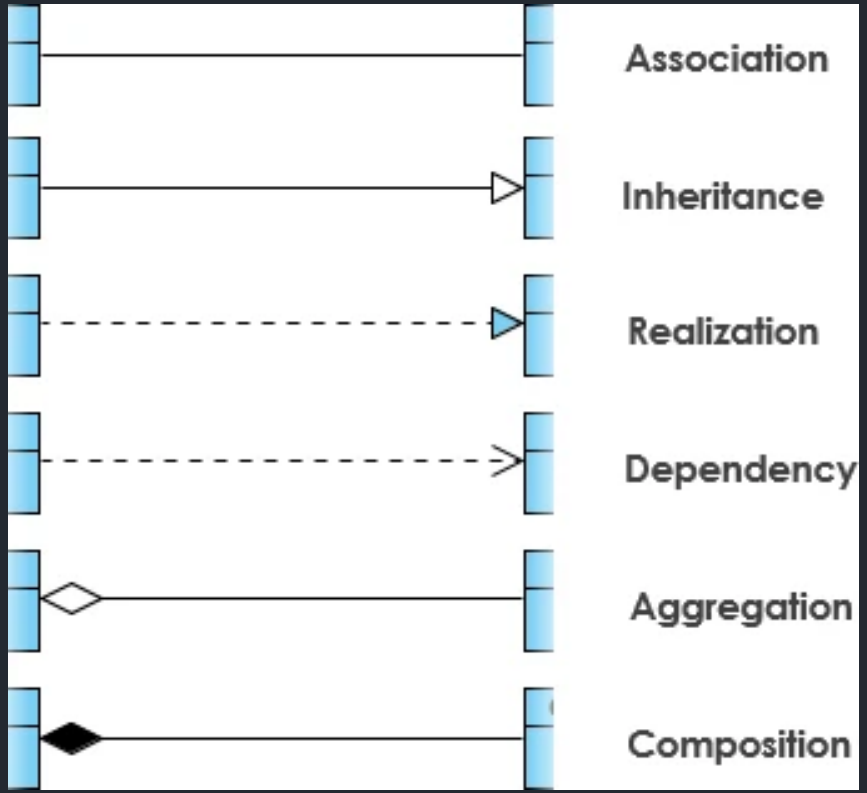
- Diagrama de clases:
  - **Encapsulación**:
    - private → **-**
    - protected → **#**
    - public → **+**



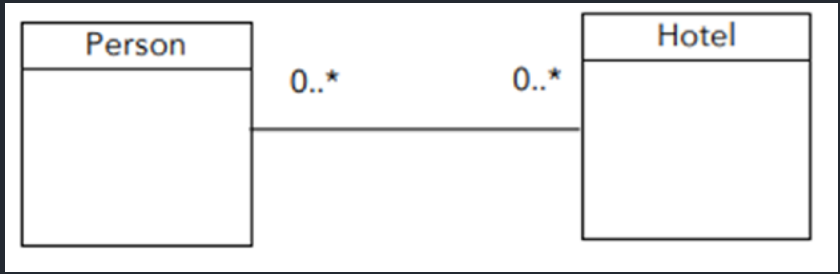
OnlineShopper
- userName: string - password: string
+ Login() : void + SelectItem() : void + AddItemToCart(item: Item) : void + RemoveItemFromCart(item: Item) : void + Checkout() : void

Item
- name: string - price: double - stockQuantity: int
+ GetName() : string + GetPrice() : double + GetStockQuantity() : int

ShoppingCart
- items: List<Item>
+ AddItem(item: Item) : void + RemoveItem(item: Item) : void + CalculateTotalPrice() : double


Association
Inheritance
Realization
Dependency
Aggregation
Composition

- **Descomposición**
  - **Asociación**
    - **Relación poco estrecha** entre dos objetos que pueden interactuar entre sí durante algún tiempo
    - **No son dependientes entre sí**: si un objeto se destruye, el otro puede seguir existiendo.

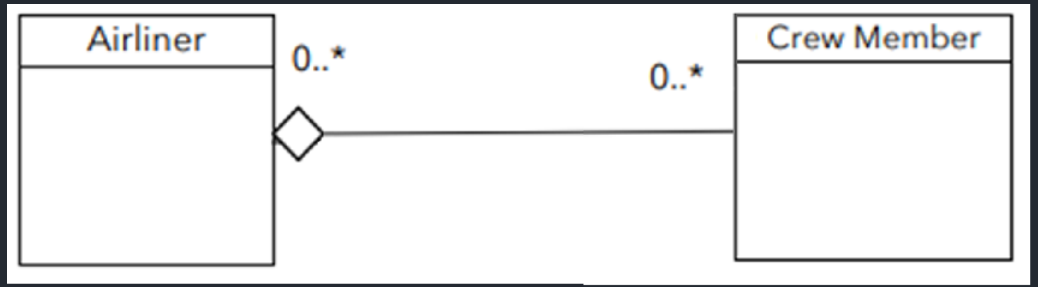



```
public class Person
{
    ...

    public void CheckIn(Hotel hotel) { ... }
}
```

- **Descomposición**
  - **Dependencia**
    - Es una relación de **"usa"** o **"necesita"**
    - Indica que una clase depende de otra para funcionar correctamente


```
public class Printer
{
    public void Print(Document document)
    {
        /*
        Printer uses the Document to
        perform the print operation
        */
        ...
    }
    ...
}
```

- **Descomposición**
  - **Agregación**
    - Es una **relación "tiene-un" débil**
    - Una clase contiene a otra, pero ambas pueden existir de forma independiente

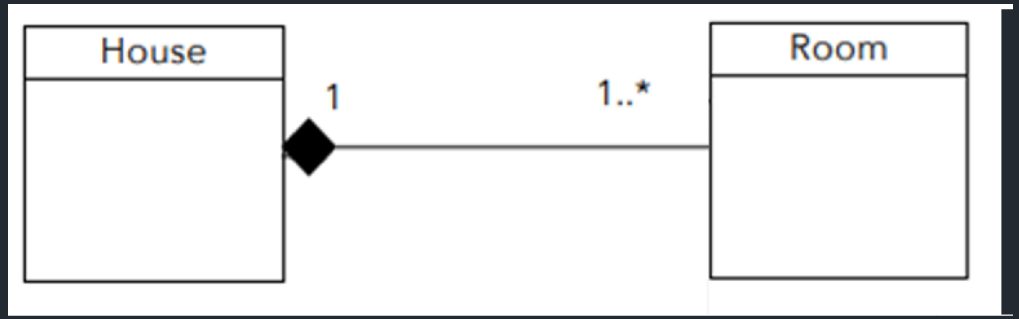


- **Descomposición**
  - **Agregación**

```
public class Airliner
{
    private List<CrewMember> crew;

    public void AddCrewMember(CrewMember crewMember)
    {
        Crew.Add(crewMember);
    }
    ...
}
```

```
// Create an airliner
Airliner airliner = new Airliner("Boeing 737");

// Create crew members
CrewMember pilot = new CrewMember("John Doe", "Pilot");

// Add crew members to the airliner (Aggregation)
airliner.AddCrewMember(pilot);
```

- **Descomposición**
  - **Composición**
    - Es una **relación "tiene-un" fuerte**
    - Una clase contiene a otra, pero si la clase contenedora se elimina (*House*), también se elimina la contenida (*Room*).

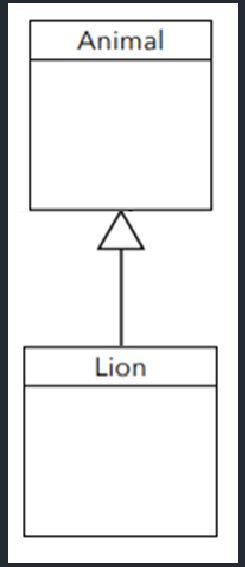


- Descomposición
  - **Composición**

```
public class House
{
    private List<Room> rooms;

    public void AddRoom(string roomName)
    {
        Room room = new Room(roomName);
        rooms.Add(room);
    }
    ...
}
```

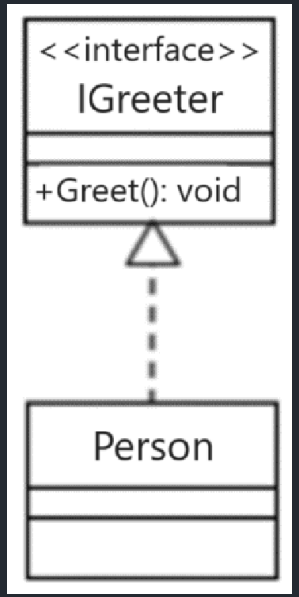
```
// Create a house
House house = new House("123 Maple Street");

// Add rooms to the house (Composition)
house.AddRoom("Living Room");
```

- **Generalización**
  - **Herencia**
    - Es una relación una relación **"es un"** entre una clase base (*Animal*) y una clase derivada (*Lion*)

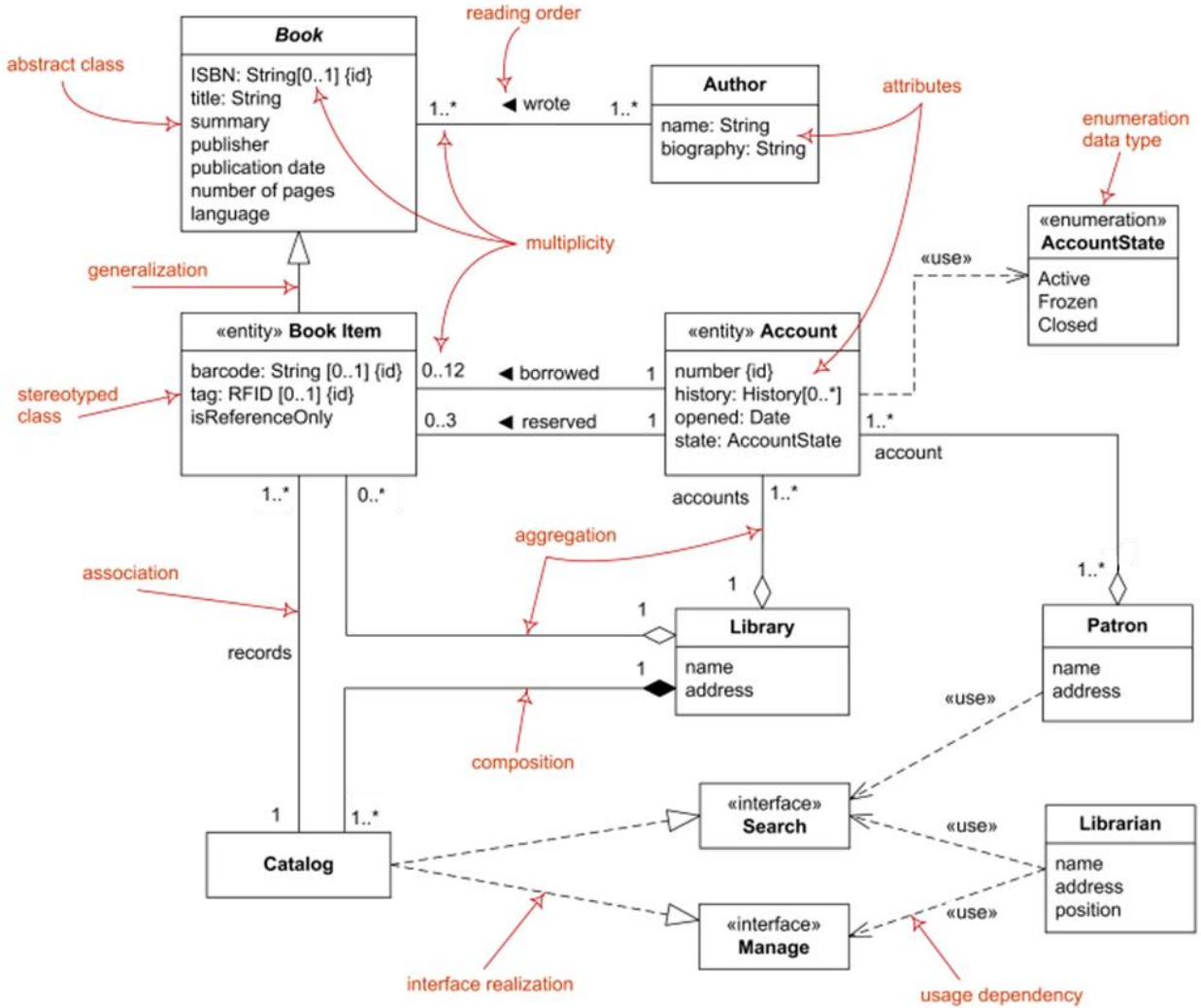



```
public class Lion : Animal
{
    ...
}
```

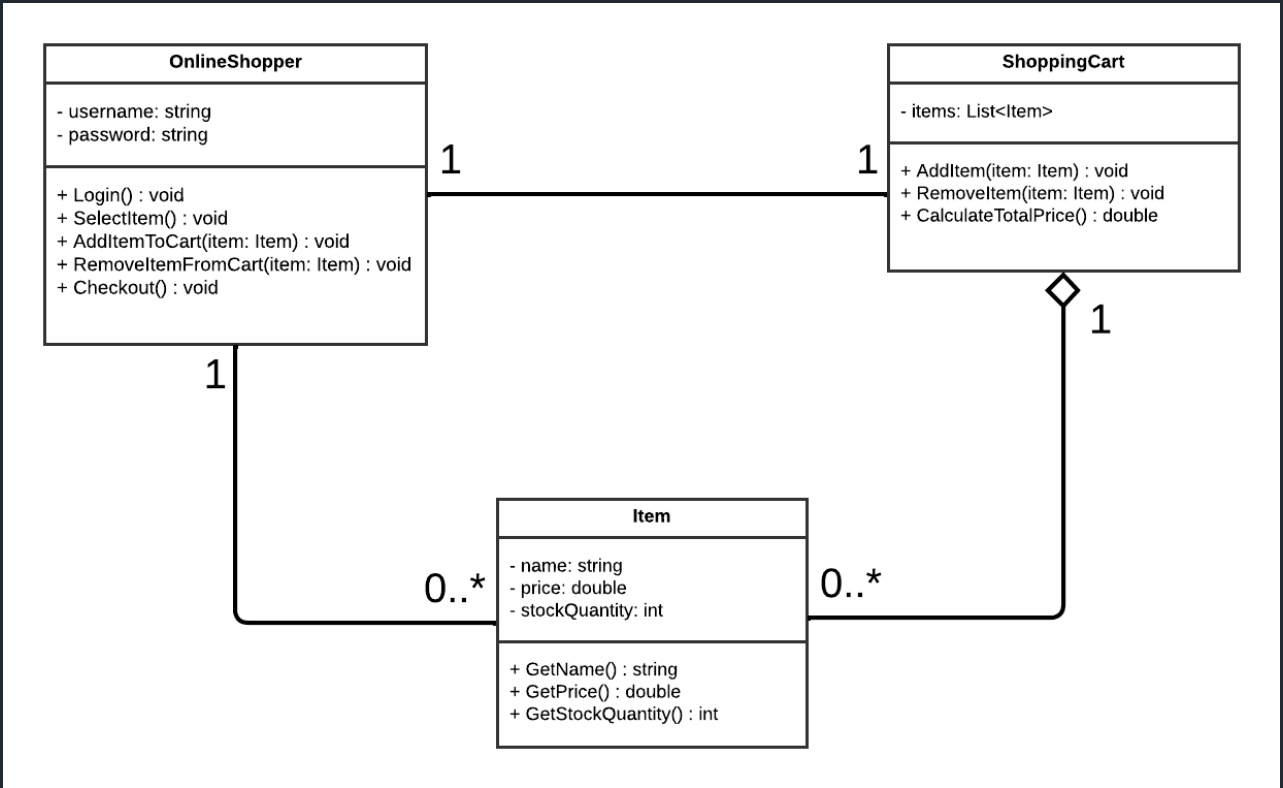
- **Generalización**
  - **Interfaces**:
    - Se utiliza para mostrar que una clase implementa una interfaz




```
public class Person : IGreeter
{
    public void Greet() { ... }
    ...
}
```


Book
ISBN: String[0..1] {id}
title: String
summary
publisher
publication date
number of pages
language
abstract class
reading order
wrote
1..*
1..*
Author
name: String
biography: String
attributes
enumeration data type
«enumeration» AccountState
Active
Frozen
Closed
«use»
multiplicity
generalization
«entity» Book Item
barcode: String [0..1] {id}
tag: RFID [0..1] {id}
isReferenceOnly
stereotyped class
0..12
borrowed
1
0..3
reserved
1
«entity» Account
number {id}
history: History[0..*]
opened: Date
state: AccountState
1..*
account
1..*
0..*
accounts
1..*
aggregation
association
1
1
Library
name
address
1..*
Patron
name
address
records
1
composition
«use»
1
1..*
Catalog
«interface» Search
«use»
Librarian
name
address
position
«interface» Manage
«use»
interface realization
usage dependency

Arquitectura del Software – SDLC – Diseño Técnico - **UML**



Enlaces de interés

- Documentación UML: https://www.uml-diagrams.org/
- Herramienta para crear diagramas UML: Draw.io