Diseño de Sistemas Software

Arquitectura .NET de una Lógica Orientada a Objetos

Índice

- **Objetivos**
- **Caso de Estudio**
- **Arquitectura del proyecto**
  - Componente de Proceso
  - Componente de Negocio
  - Entidad de Negocio o Data Transfer Object
  - Componente de Acceso a Datos

Objetivos

- Dar a conocer los diferentes componentes que constituyen la arquitectura de una lógica de negocio orientada a objetos del proyecto a desarrollar

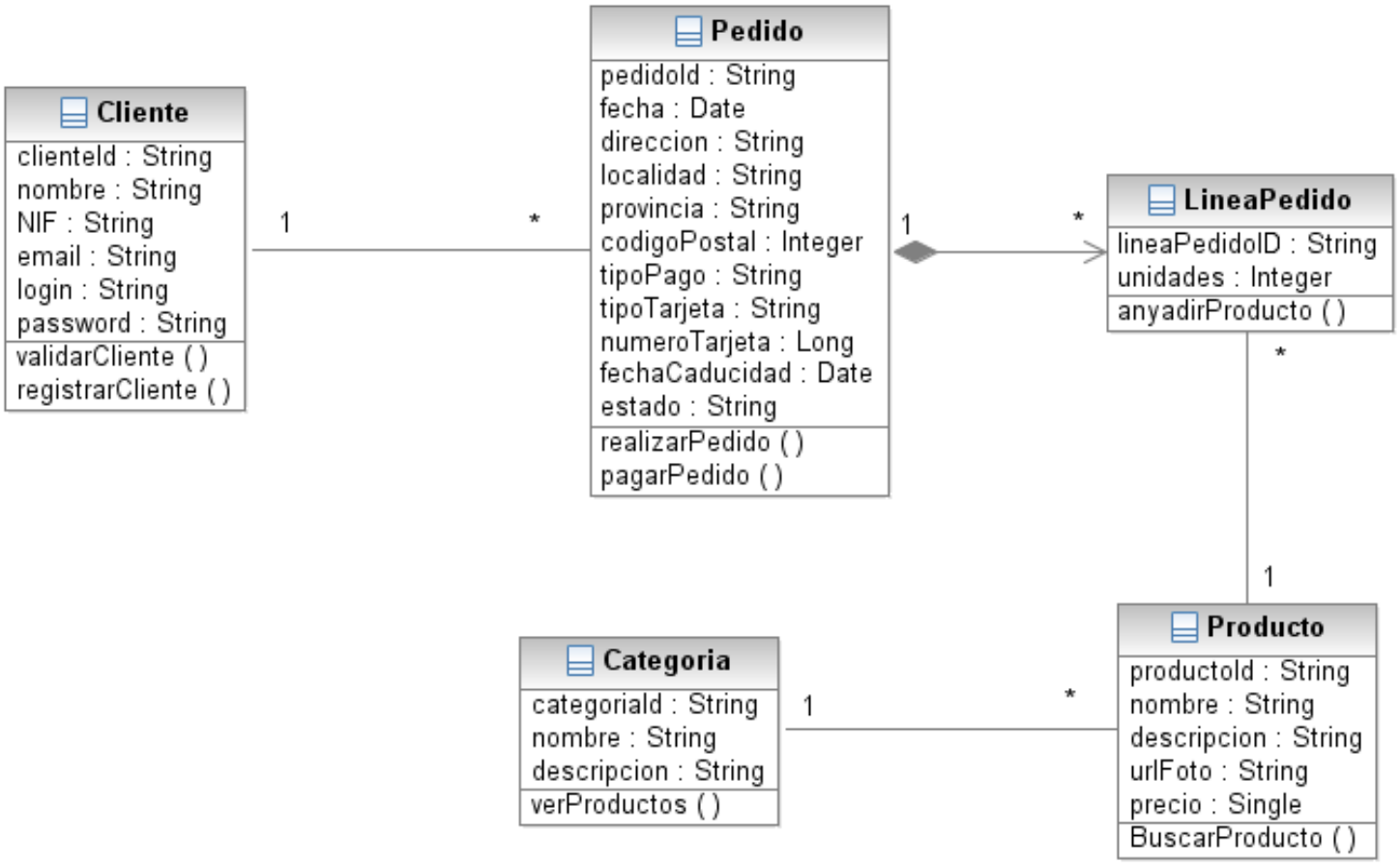
- Proveer de una guía que ayude a elegir el modo más apropiado de exponer, representar y convertir el modelo de dominio en las entidades de negocio

- Ser capaz de implementar la lógica de negocio y las capa de datos mediante el framework NHibernate

Caso de Estudio: Petstore

- Conocida aplicación de comercio electrónico diseñada por SUN microsystems
- Petstore usa las mejores prácticas y guías de diseño para una arquitectura Web
- Usada para realizar pruebas de rendimiento entre las plataformas .NET vs JavaEE

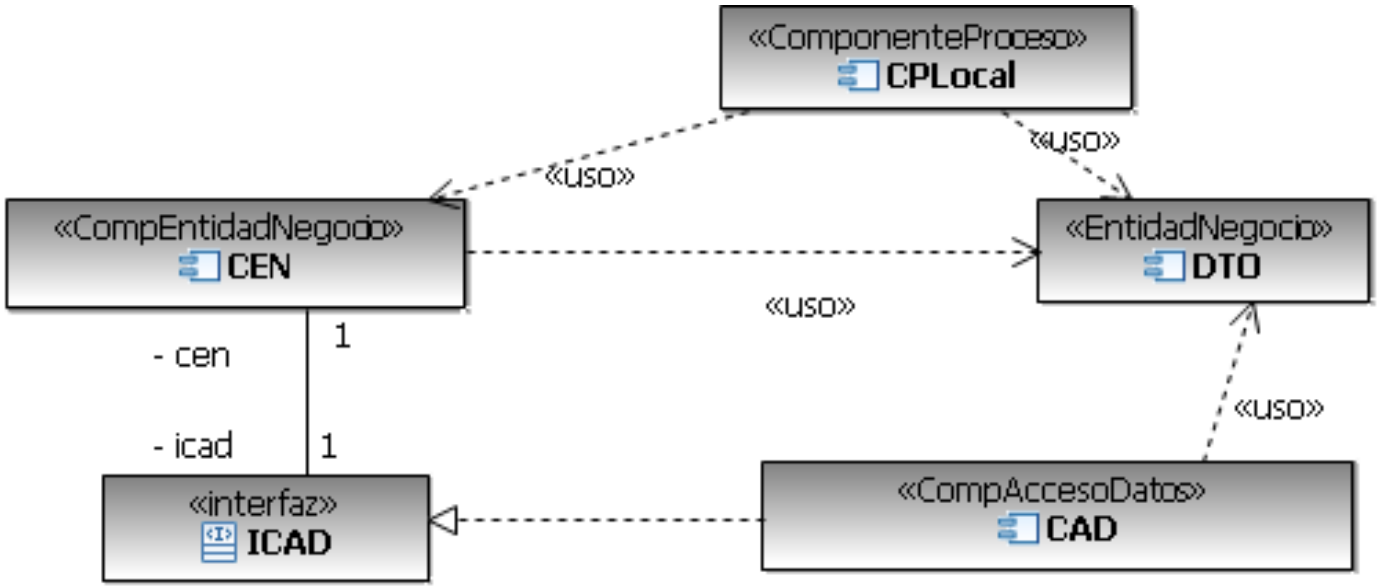
Modelo de Dominio de Petstore



Características de la Arquitectura

- Seguimos una tipificación de componentes propuesta por el libro Enterprise Solution Patterns (Microsoft, 2003)
- Adaptamos los componentes de la arquitectura para soportar la utilización del framework objeto-relacional Nhibernate
- Se aplican patrones de arquitectura y diseño que proporcionan un mejor acoplamiento y mayor cohesión entre los componentes de la lógica de negocio y los datos
- Se proporciona soporte para las transacciones complejas mediante Componentes de Proceso

Arquitectura de la lógica de negocio con NHibernate



Componentes Entidad de Negocio

Arquitectura .NET de una Lógica Orientada a Objetos

Componente Entidad de Negocio

- Un modelo de dominio contiene múltiples clases con relaciones y debemos decidir como mapear las clases en diferentes Componentes Entidad de Negocio (CEN), en inglés Business Objects (BO)

- Se debe identificar el núcleo de CENs que encapsularán la funcionalidad de la aplicación (p.e. un solo CEN puede contener una agregación o una jerarquía de herencia)

Distinción entre CEN y EN

- Hablamos de CEN (Componente Entidad de Negocio o BO), cuando dicho componente representa a las entidades de dominio localizadas en la capa de lógica de negocio

- Hablamos de EN (Entidad de Negocio o DTO) cuando es un componente que representa a las entidades de dominio que pueden almacenarse y transitar por diferentes capas

Distinción entre CEN y EN

- Los CEN requieren de lógica de negocio y de persistencia aunque esta puede contener el almacenamiento en BBDD mediante el CRUD (alta, baja, modificación y consulta) o sin CRUD

- Las EN pueden ofrecer lógica de forma opcional para que sea evaluada en la capa de cliente. Solo son obligatorios los atributos y las propiedades

Implementación de los CEN

- Se implementa un componente CEN por cada clase de dominio (o composición o jerarquía de herencia)<-cuidado con la cohesión!!
- Existen diferentes tipos de CEN:
  - Contienen atributos (y propiedades publicas) y las operaciones de la clase de dominio
  - Contienen exclusivamente las operaciones y separamos en una clase a parte los atributos y propiedades llamada EN o DTO (Patrón **Data Transfer Object**, Fowler, 2002)

Implementación de los CEN

- Cada CEN referenciará a un Componente de Acceso a Datos (Patrón ***Data Access Object***) que permite independizar la lógica de negocio de la persistencia

- En lugar de referenciar directamente a la clase, lo hará mediante la interfaz del CAD aplicando el patrón de **Dependency Injection** (Fowler, 2002)

Implementación de los CEN

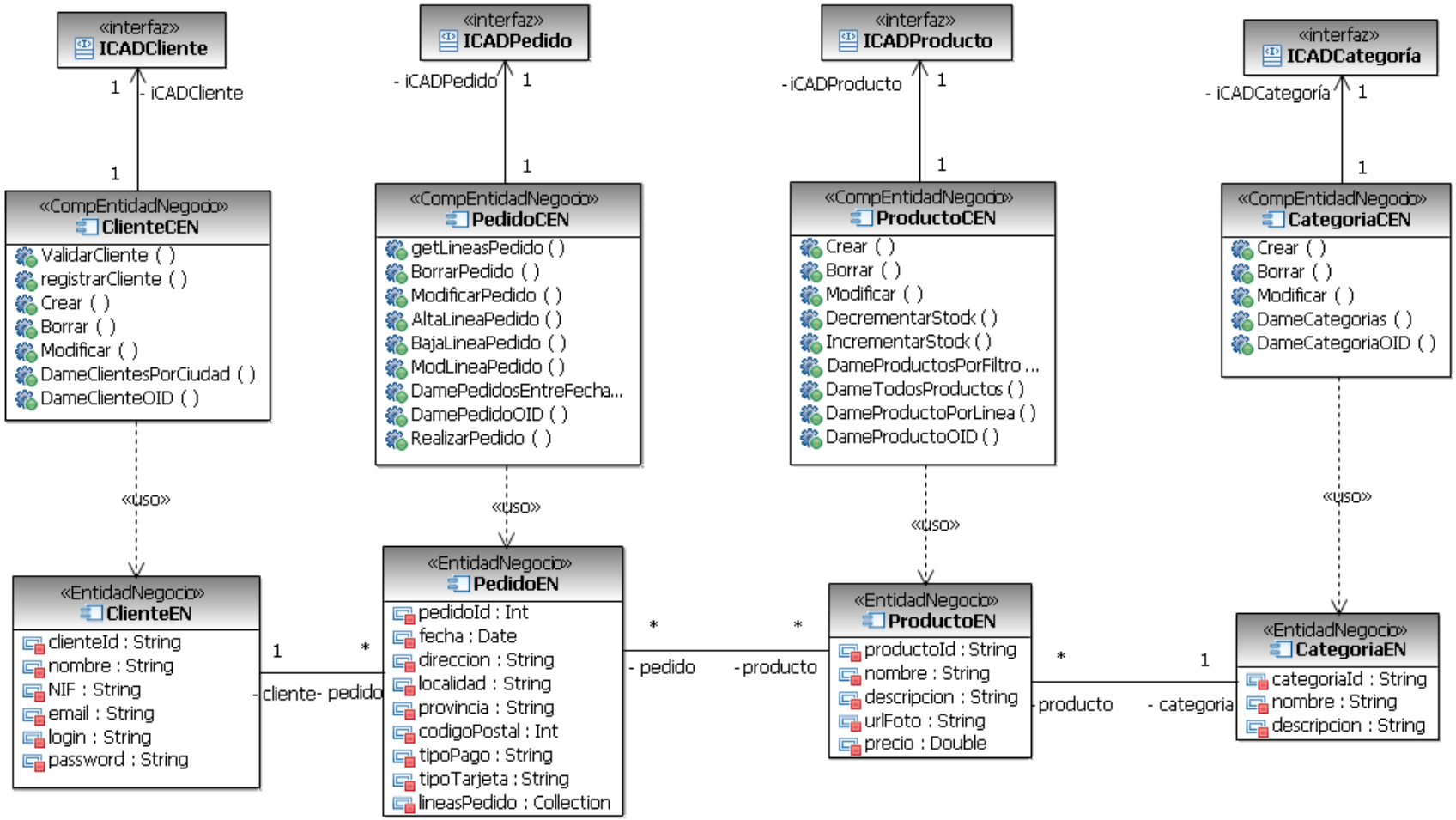
- Los CEN con DTO separado no mantienen estado, en cada operación se instancia el EN (que contiene el estado) y se opera sobre el mismo

- Se mantiene así la necesidad de que las operaciones que trabajen sobre objeto/s reciban sus oid/s para trabajar

Componentes Entidad de Negocio en Petstore

- Basándose en el Modelo de Dominio de Petstore, se detecta una composición entre Pedido y Línea de Pedido que permite unir las clases en una CEN
- Así podemos definir cuatro CEN lógicas que controlen la aplicación:
  - Un Cliente
  - Un Pedido que contendrá sus líneas de pedido
  - Un Producto y una Categoría

Lógica de Negocio de Petstore



Implementación del CEN de Producto

```
public class ProductoCEN
{
        private IProductoCAD _iProductoCAD;

         public ProductoCEN() // Constructor por defecto
        {
                        _iProductoCAD = new ProductoCAD();
        }
         public ProductoCEN(IClienteCAD iProductoCAD) // Patrón inyección dependencia
        {
                _iProductoCAD = iProductoCAD;
        }

         //Métodos CRUD (Alta, baja, modificación y consultas)
         public String Crear (String p_id, String p_nombre, Double p_precioUnit,
                                        String p_descripción)
        {…}
}
```

Implementación del CEN de producto

```
public String Crear (String p_id, String p_nombre,
                     Double p_precioUnit, String p_descripción)
    {
        ProductoEN en = new ProductoEN ();
        String oid;
        en.id = p_id;
        en.nombre = p_nombre;
        ....
        oid = iProductoCAD.Crear (en);       //<- Se crea un DTO
                                             // y se envía a la capa de datos
        return oid;
     }
```

Implementación CEN Producto

```
// Operaciones que realizan algún procesamiento
// Siempre reciben el oid del objeto a manipular

public void IncrementarPrecioUnidadPor (String id, decimal p_cantidad)
{
        ENProducto en = _IProductoCAD.BuscarPorOID(id);

        en.precio += p_cantidad;

        icad.Actualizar (en);
}
```

Modelo de ejecución de una operación

- Se pretende unificar la forma de plasmar las operaciones de dominio en la implementación
- Nos basamos en el diseño dirigido por contrato para definir las operaciones (Meyer, 1992)
- Se deben de cumplir las precondiciones y postcondiciones antes y después del cuerpo de las operaciones

Diseño dirigido por contrato

- Los contratos software se especifican mediante la utilización de expresiones lógicas denominadas aserciones. Dichas aserciones siempre deben cumplirse para poder ejecutar una operación
- Existen 2 tipos de aserciones:
  - **Precondiciones**: Asegura que se cumple un determinado estado de partida para ejecutar la operación
  - **Postcondiciones**: Verifica que el trabajo realizado por la operación se ha cumplido correctamente
- Su incumplimiento invalida totalmente el contrato, indicando que existe un error en la operación (se lanza una excepción)

Diseño por contrato de una operación

```
public void operación (tipoOID p_oid)
{
      ENClase en = icad.buscarPorOID (en);
      //Precondicion   (opcional)
       if (!condicion) throw new ModelException("...");
      // Cuerpo (obligatorio)
       Puede modificar solo el objeto actual
      //Poscondicion (opcional)
       if (!condicion) throw new ModelException("..");
      icad.actualizar(en); //Actualización en persistencia
}
```

Ejemplo de operación

```
public void decrementarStock(int cantidadDecr, int idProducto)
{
        ENProducto en = icad.buscarPorOID (idProducto);
        // precondición: stock > cantidadDecr
        if (!(stock > cantidadDecr))
            throw new ModelException("No se cumple la precondicion:
                                    La cantidad a decrementar es superior al stock");
         // Cuerpo
        int stockpre = this.stock;
        stock = stock - cantidadDecr;
         //postcondición: self.stock = self.stock@pre - cantidadDecr
        if (!(stock == stockpre – cantidadDecr)) throw new
ModelException("No se cumple la postcondicion de decrementarStock...");

       icad.Modificar(en);
}
```

Entidades de Negocio

Arquitectura .NET de una Lógica Orientada a Objetos

Implementando las Entidades de Negocio

- Las EN pueden ser serializables, para hacer persistente el estado actual de las entidades. P.e pueden ser guardadas en el disco duro local, en una BBDD de escritorio si la aplicación trabaja sin conexión, etc.

- Las EN no inician transacciones, estas son iniciadas por CAD o por los componentes de proceso

Tipos de Representaciones de los EN en .NET

- XML: Se usa una cadena XML o un objeto XML DOM (Document Object Model). XML es un formato abierto y flexible que puede ser usado para integrar diversos tipos de aplicaciones
- DataSet: representa una cache de las tablas de una BBDD relacional
- DataSet tipado: clase que hereda del DataSet de ADO.NET que proporciona métodos fuertemente tipados asociados a una tabla concreta

Tipos de representaciones de una EN

- Clase Personalizada: se define una clase por cada entidad de negocio. Contiene los atributos y propiedades para exponer la información a la aplicación cliente
- Representan las relaciones de asociación (incluidas sus subtipos agregación y composición) y las de herencia (el CEN NO!!)
- No contiene los métodos CRUD para invocar a los CAD, sino que es el CEN quien lo hace

Representación de un EN

```
public class ENProducto
{
// Campos privados para mantener el
// estado de la Entidad Producto
private int idProducto;
private string nombre;
private string cantidadPorUnidad;
private decimal precioUnitario;
private int unidadesStock;
private int stockMinimo;
// Propiedades publicas para exponer
//el estado del producto
public int IdProducto
{
get { return idProducto; }
set { idProducto = value; }
}
```

```
public string Nombre {
  get { return nombre; }
  set { nombre = value; }
}
public string
CantidadPorUnidad
{
get { return
cantidadPorUnidad; }
set { cantidadPorUnidad =
value; }
}
public decimal
PrecioUnitario
{
get { return precioUnitario; }
set { precioUnitario =
value; }}
…
```

Relaciones de Asociación

```
public class ENPedido
{
// Atributos de Pedido
private int idPedido;
private ENCliente cliente; //Un Cliente
private DateTime fechaPedido;
// Atributo que contiene las Lineas de
Pedido
private Ilist<LineaPedidoEN>
lineasPedido; // Muchas LineasPedido
// Propiedades públicas
public int IdPedido
{
  get { return idPedido; }
  set { idPedido = value; }
}
```

```
public ENCliente Cliente
{
get { return cliente; }
set { cliente = value; }
}
public DateTime FechaPedido
{
get { return fechaPedido; }
set { fechaPedido = value;}
}
public
Ilist<LineaPedidoEN>
LineasPedido
{
      get { return
      lineasPedido; }
      set { lineasPedido = value;
      }
}
```

Relaciones de Asociación

- La diferencia a nivel de asociación, con composición no se refleja a nivel de EN sino de CAD (creación y borrado de las entidades contenidas)

- Cuidado con la navegabilidad, si una relación no es navegable en un sentido, no se genera ni el rol ni la propiedad en el EN

Relación de Herencia

- Se representa mediante una herencia entre el clases C#

  no soporta herencia múltiple (compatible con Nhibernate)

```
public class ClienteExternoEN :  ClienteEN
{
   // Solo contiene los atributos especializados
    String pais;
    String ciudad;
   // Solo contiene las propiedades a atributos especializados
       ...
}
```

Serialización de una Entidad de Negocio en XML

```
using System.Xml.Serialization; // Espacio de nombres del XMLSerializer


...
// Creamos un objeto XmlSerializer object, para serializar Pedidos
XmlSerializer serializer = new XmlSerializer(typeof(ENPedido));

// Serializamos un objeto pedido en el fichero “MiXmlENPedido.xml"
TextWriter writer = new StreamWriter("MiXmlENPedido.xml");
serializer.Serialize(writer, pedido);
writer.Close();
```

Representando una entidad con XML

- El siguiente ejemplo muestra un producto representado con XML

```
<?xml version="1.0"?>
<Producto>
<IdProducto>1</IdProducto>
<Nombre>Caramelos de Menta </Nombre>
<Cantidad>10 cajas x 20 bolsas </Cantidad>
<PrecioUnitario>18.00</PrecioUnitario>
<UnidadesEnStock>39</UnidadesEnStock>
</Producto>
```

Representando EN compuestos con XML

```
<Pedido>
 <PedidoID>10248</PedidoID>
 <ClienteID>VINET</ClienteID>
 <Fecha>1996-07-04</Fecha>
 <FechaEnvio>1996-07-16</FechaEnvio>
 <LineasPedido>
     <LineaPedido>
          <ProductoID>11</ProductoID>
          <PrecioUnitario>14.00</PrecioUnitario>
          <Cantidad>12</Cantidad>
    </LineaPedido>
    <LineaPedido>
          <ProductoID>42</ProductoID>
          <PrecioUnitario>9.80</PrecioUnitario>
          <Cantidad>10</Cantidad>
     </LineaPedido>
 </LineasPedido>
</Pedido>
```

Representando una EN con XML

- **Ventajas:**
  - Utilización de un estándar de la W3C para la representación de datos
  - Flexibilidad. XML permite representar jerarquías y colecciones de información
  - Interoperabilidad. XML es una opción ideal para intercambiar información con sistemas legados (externos), socios comerciales, de forma independiente de plataforma.
- **Desventajas:**
  - La correctitud del tipo no es preservada en XML
  - La validación de un XML es lenta
  - No permite el uso de campos privados
  - Requiere de un proceso complejo su ordenación

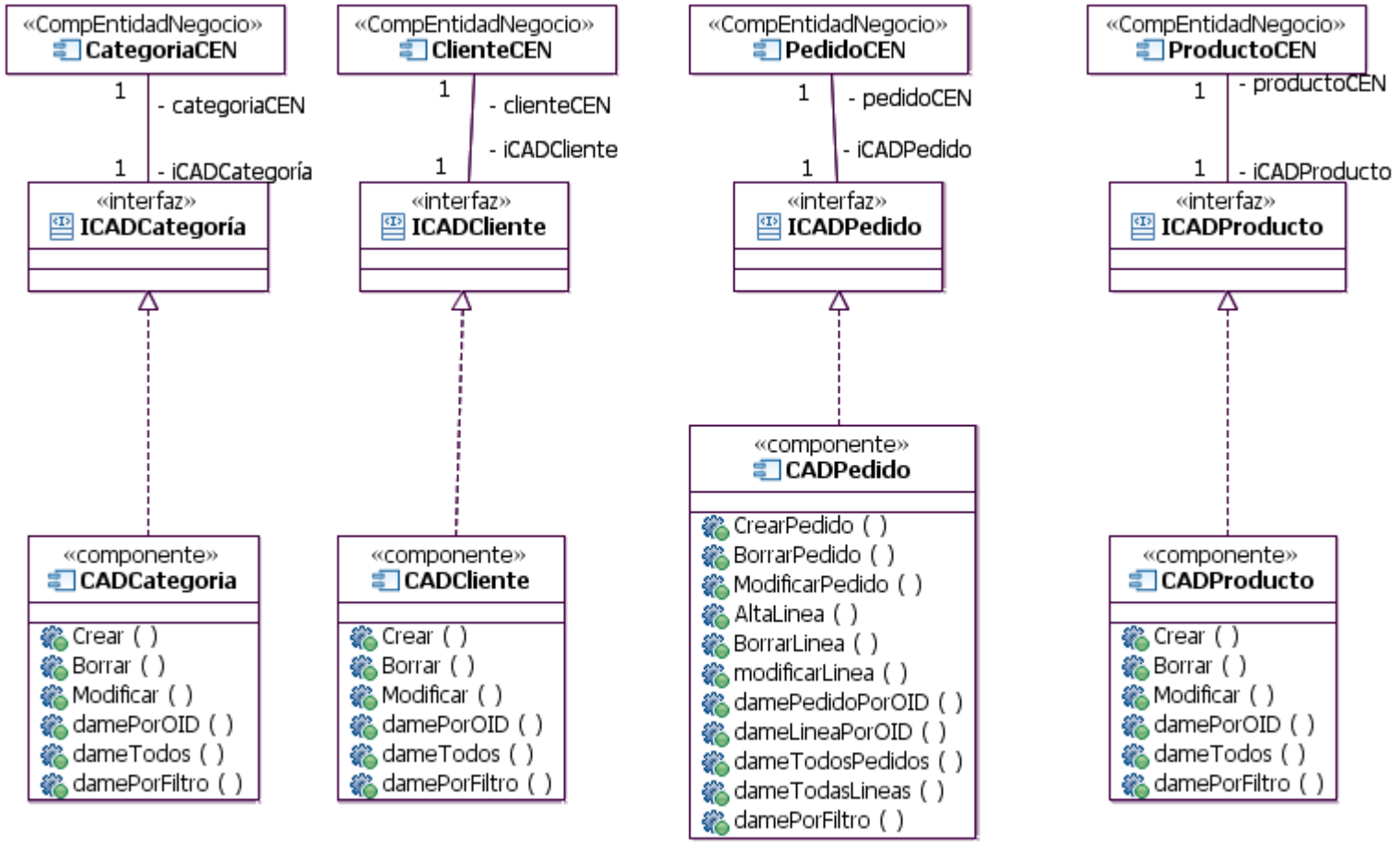
Componente de Acceso a Datos

Arquitectura .NET de una Lógica Orientada a Objetos

Componente de Acceso a Datos

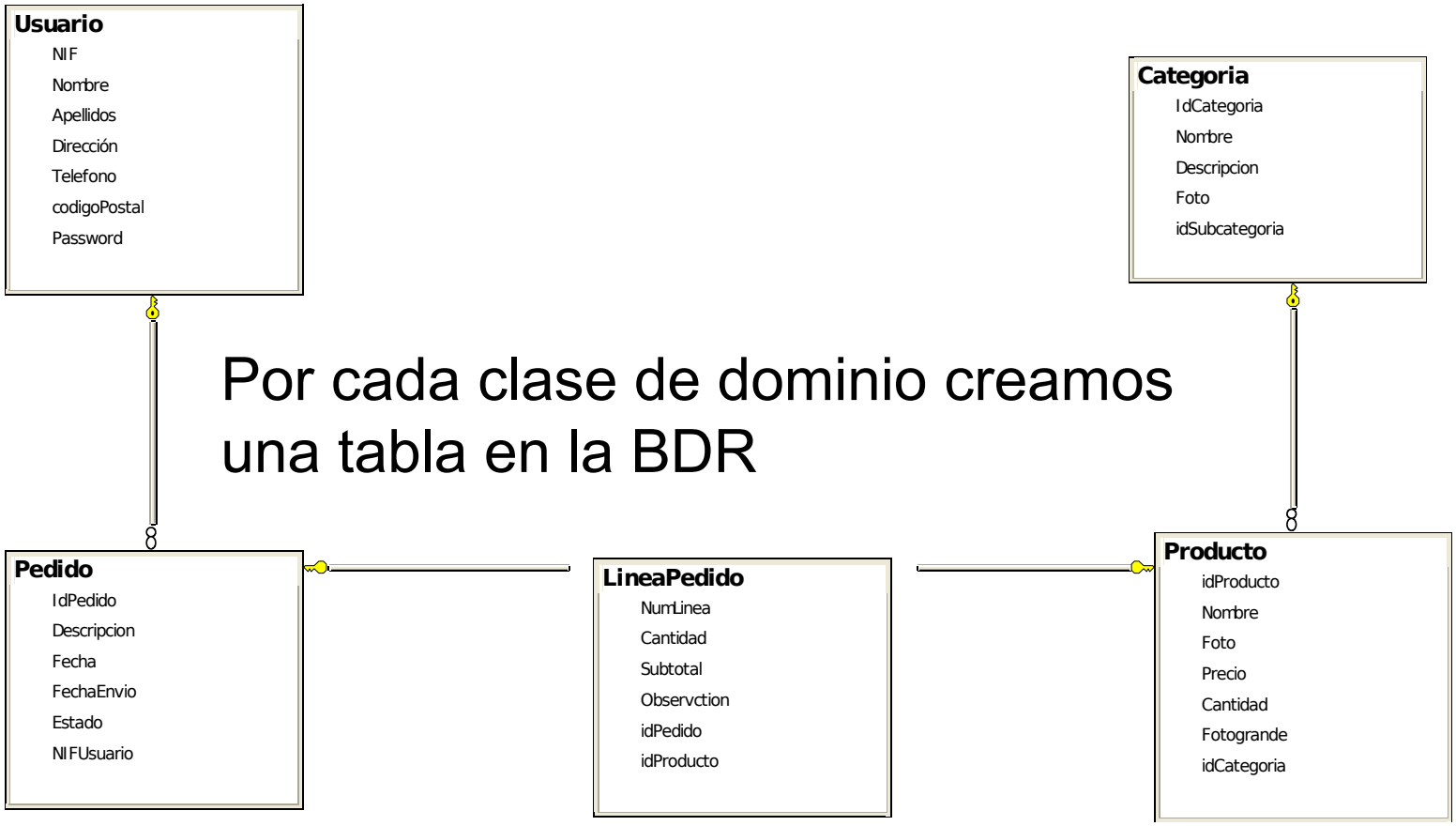
- Para cada CEN, definimos un CAD que será definido como sigue:
  - ClienteCAD: Esta clase provee los servicios para recuperar y modificar los datos de las tablas Cliente
  - PedidoCAD: Esta clase provee los servicios para recuperar y modificar los datos de las tablas Pedido y LineaPedido
  - ProductoCAD: Esta clase provee los servicios para recuperar y modificar los datos de la tabla Producto
  - CategoríaCAD:Esta clase provee los servicios para recuperar y modificar los datos de la tabla Categoria

Diseño de los Componentes de Acceso a Datos



Recordar!! Unificar Pedido y Línea es opcional

Base de Datos Relacional



Mapeo Objeto-Relacional

- Definir todos los métodos que devuelven un tipo concreto de Entidad de Negocio en un solo CAD para mantener una cohesión alta
  - Por ejemplo, si se están recuperando todos los pedidos de un determinado cliente, implementar la función en PedidoCAD llamada *DamePedidosPorCliente* que devuelva los pedidos filtrando por un idCliente
  - Contrariamente, si estás recuperando todos los clientes que han realizado un pedido específico, implementa la función en ClienteCAD *DameClientesPorPedido*

Mapeo Objeto-Relacional

- Se han de mantener las relaciones de asociación mediante métodos de modificación, creación y borrado de relaciones
- Relaciones 1 a muchos, 1 a 1, requieren de métodos de update en la BBDD
- Relaciones muchos a muchos requieren de métodos de añadir y borrar relaciones en una tabla intermedia.

Implementando los CADs

- Un CAD es utilizado sin estado, es decir, todos los mensajes intercambiados pueden ser interpretados independientemente
- Sin embargo, cuando la transacción se mantiene en el Componente de Proceso se mantiene la cache (Isession) y la transacción (Itransaction) entre llamadas
- Uno de los principales objetivos de un CAD es encapsular o esconder la idiosincrasia de la invocación y el formato de la BD a la lógica de negocio

Operaciones de un CAD

- Un CAD debería proveer los métodos para realizar las siguientes tareas sobre la BD:
  - Crear registros en la BD
  - Leer registros en la BD y devolver las entidades de negocio al CEN
  - Actualizar registros en la BD, usando entidades de negocio proporcionadas por el CEN
  - Borrar registros de la BD
- Estos métodos son llamados CRUD, acrónimo de “Create, Read, Update and Delete”

Operaciones de un CAD (II)

- Los CAD pueden contener también métodos que realizan algún filtro. Por ejemplo, un CAD puede tener un método para encontrar el producto más vendido en un catalogo durante un mes
- Un CAD accede a una única BD y encapsula las operaciones relacionadas con una única tabla o un grupo de tablas vinculadas de la BD.
- Por ejemplo, podréis definir un CAD que controle las tablas de Cliente y sus Direcciones, y otro CAD que controle los Pedidos y las LineasDePedidos (debido a que son composiciones)

Implementación de un CAD

- **Alternativas para su implementación en .NET:**
  - ADO.NET
  - Enterprise Library: Data Access Application Block
  - LinQ
  - Nhibernate
  - Entity Framework

CADs con NHibernate

- Framework Object-Relational Mapping (ORM)
- Permite simplificar el almacenamiento y recuperación de los objetos en la BBDD
- Simplifica el código de los CADs reduciendo las operaciones ofertadas al mínimo
- Las navegaciones por rol y herencia son realizadas de forma implícita por el propio framework
- Simplifica las consultas SQL orientadas a datos, utilizando el lenguaje HQL (Hibernate Query Language)

¿Por qué Nhibernate?

- **Programación Natural**- NHibernate soporta la OO de forma natural con herencia, polimorfismo, composición. Además soporta las colecciones con tipos primitivos
- **Simplifica los CADs**: Evita programar las navegaciones por rol y herencia, y su lenguaje HQL simplifica las consultas
- **Proporciona un mapping en doble sentido** (O/R) para recuperar objetos EN orientado a objetos como para almacenarlos ahorrándonos la conversión de datos a objetos y de objeto a datos

Método Insertar

```
public String CrearCliente (ClienteEN cliente)
{
    try
    {
        SessionInitializeTransaction ();
        session.Save (cliente);
        SessionCommit ();
    }
    catch (Exception ex) {
          SessionRollBack ();
          throw new DataLayerException ("Error in ClienteCAD.", ex);
    }
    finally
    {
       SessionClose ();
    }
    return cliente.Id; //<- Imprescindible para los id's autogenerados
 }
```

CADs con NHibernate

Método Borrar

```
public void BorrarCliente (String nif)
{
      try
      {
            SessionInitializeTransaction ();
             ClienteEN clienteEN =
            (ClienteEN)session.Load (typeof(ClienteEN), nif);
             session.Delete (clienteEN);
             SessionCommit();
      }
      catch (Exception ex) {
             SessionRollBack ();
             throw new DataLayerException ("Error in ClienteCAD.", ex);
      }
...
 }
```

CADs con Nhibernate. Modificar

```
public void Modify (ClienteEN cliente)
{
    try
    {
        SessionInitializeTransaction ();
        ClienteEN clienteEN = (ClienteEN)session.Load (typeof
(ClienteEN), cliente.DNI);
        clienteEN.Nombre = cliente.Nombre;
        ...
        session.Update (clienteEN);
        SessionCommit ();              }
    }
    catch (Exception ex) {
        ...
}
```

<- Sólo los Atributos propios de la clase

CADs con NHibernate

Método damePorOID

```
public ClienteEN ReadOIDDefault (String NIF)
{
      try
      {
            SessionInitializeTransaction ();
           clienteEN = (ClienteEN)session.Load (typeof(ClienteEN), NIF);
            SessionCommit ();
      }
      catch (Exception ex) {
             SessionRollBack ();
             throw new DataLayerException ("Error in ClienteCAD.", ex);
      }
      finally
      {
          SessionClose ();
      }
      return clienteEN;
 }
```

Método dameTodos

```
public System.Collections.Generic.IList<ClienteEN> DameTodos (int first, int size)
{
    System.Collections.Generic.IList<ClienteEN> result = null;
    try
    {
        SessionInitializeTransaction ();
        if (size > 0)
            result = session.CreateCriteria (typeof(ClienteEN)).
                SetFirstResult (first).SetMaxResults (size).List<ClienteEN>();
        else
          result = session.CreateCriteria (typeof(ClienteEN)).List<ClienteEN>();
        SessionCommit ();
    }

    catch (Exception ex) {
         SessionRollBack ();
         throw new DataLayerException ("Error in ClienteCAD.", ex);
    }
          ...
          return result;
     }
```

Creamos la BBDD

- Nhibernate solo crea las tablas así que nosotros debemos crear la BBDD y el schema primero
- (Ver el archivo createDB.cs en el proyecto generado por OOH4RIA)

```
 CreateDB.Create ("pruebasRelacionesGenNHibernate",
"nhibernateUser", "nhibernatePass");
 var cfg = new Configuration ();
 cfg.Configure ();
 cfg.AddAssembly (typeof(CamionEN).Assembly);
 new SchemaExport (cfg).Execute (true, true,
false);
```

Mapping Uno-a-Muchos

- Introducimos borrado en cascada solo en las relaciones de composición

- En los otros casos (asociación y agregación) daremos un error al intentar borrar un objeto que mantenga una relación

Mapping Uno-a-Muchos

```
<class name="CamionEN" table="Camion“>
 <id name="Id" column="IdCamion">
   <generator class="identity"/>
 </id>
 <bag name="Ruedas" cascade="all“> <-borrado en cascada!!
  <key>
   <column name=”FK_Camion"/>
  </key>
  <one-to-many class=“RuedasEN"/>
 </bag>
</class>
```

```
<class name="RuedasEN" table="Ruedas“>
  <id name="Id" column="IdRuedas">
    <generator class="identity"/>
</id>
 <many-to-one name="Camion"   class=“CamionEN“>
  <column name=“FK_Camion" not-null="false"/> <- cardinalidad
mínima 0.
 </many-to-one>
</class>
```

Mapping Uno-a-Uno

```
<class name="DespachoEN" table="Despacho”>
  <id name="Id" column="IdDespacho">
     <generator class="identity"/>
  </id>
   <property name="Ubicacion" />
     <one-to-one name="Profesor" class=“ProfesorEN"
property-ref="Despacho“/>
</class>
```

```
<class name="ProfesorEN" table="ProfesorEN">
     <id name="Id" column="IdProfesor">
          <generator class="identity"/>
     </id>
        <property name="Nombre" column="Nombre" />
        <many-to-one name="Despacho" class=“DespachoEN">
           <column name=”FK_id_despacho" unique="true" not-null="true"/>
       </many-to-one>
</class>
```

Se introduce el unique=“true” para forzar el one-to-one y la cardinalidad mínima es 1.

Mapping Muchos-a-Muchos

```
<class name="AlumnoEN" table="Alumno“ >

  <id name="Id" column="IdAlumno">
     <generator class="identity"/>
  </id>
  <property name="Nombre" />

   <bag name="Profesor" table="Alumno_Profesor“>
    <key>
      <column name=”FK_IdAlumno"/>
    </key>
   <many-to-many class=“ProfesorEN">
       <column name=”FK_idProfesor"/>
   </many-to-many>
  </bag>

</class>
```

Mapping muchos a muchos

```
<class name="ProfesorEN" table="Profesor">

  <id name="Id" column="IdProfesor">
      <generator class="identity"/>
  </id>
  <property name="Nombre" column="Nombre" />

 <bag name="Alumno" table="Alumno_Profesor" >
   <key>
      <column name=”FK_IdProfesor"/>
   </key>
    <many-to-many class="AlumnoEN">
          <column name=”FK_idAlumno" />
    </many-to-many>
  </bag>

 </class>
```

Mapping de herencia

```
<class name="ClienteEN" table="Cliente">
  <id name="DNI" column="DNICliente"/>
  <property name="Nombre" column="Nombre" />
  <property name="Direccion" column="Direccion" />
  <property name="Telefono" column="Telefono" />
</class>
```

```
<joined-subclass name="ClienteExternoEN" extends="ClienteEN">
   <key>
    <column name="DNI"/>
   </key>
     <property name="Pais" column="Country" />
</joined-subclass>
```

Cada subclase tiene una clave ajena a la clase padre.
Equivale a un mapping one-to-one

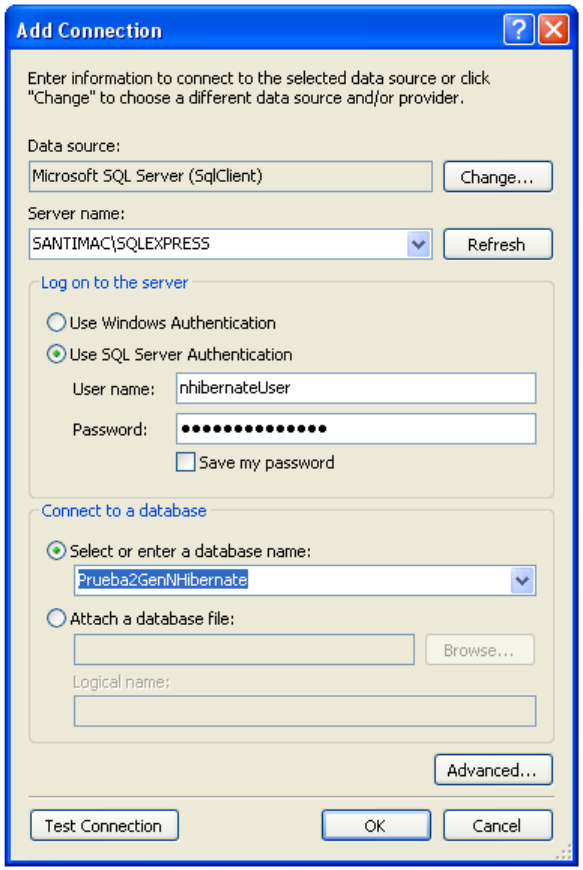
Aspectos de Arquitectura

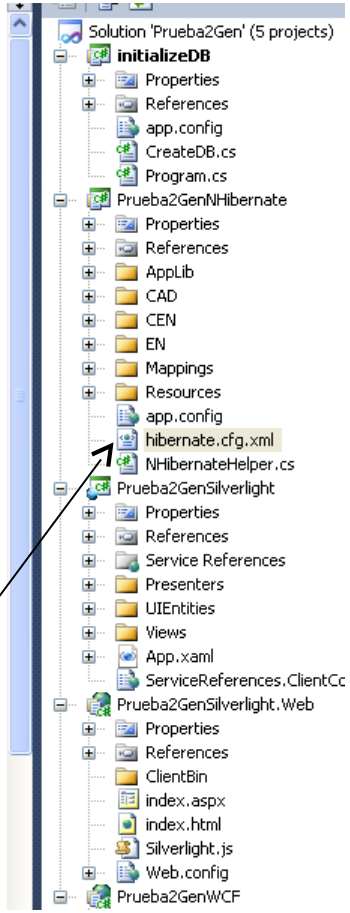
- Actualmente Nhibernate se configura con la estrategia “lazy fetching”, es decir, se trae el objeto y sus relaciones, solo bajo demanda
- Es decir, Nhibernate genera una consulta a la BBDD cada vez que hacemos **cliente.Pedidos**
- Esta estrategia obliga a que solo podemos navegar cuando tengamos una Session abierta
- Por lo tanto, solo dentro de los CPs o los CADs se puede navegar por lo objetos EN o DTOs. Los CEN solo podrán cuando son invocados por un CP

Añadir Conexión a la BBDD

```
onProvider</property>
</property>
 Id=nhibernateUser;Password=nhibernatePass;
```

- Se ha de ejecutar el archivo loginSQLUser.exe para habilitar usuarios SQL Server en el servidor SQLServer Express





La conexión de la aplicación se configura en hibernate.cfg.xml

Paginación de Datos

- La paginación de datos es un requisito común en las aplicaciones distribuidas
- Se hace imprescindible cuando trabajamos con poblaciones grandes de datos
  - P.e. El usuario quiere consultar los libros de una biblioteca, teniendo en cuenta que la tabla libro tiene miles de entradas
  - Si no hay paginación la consulta puede devolver un DTO enorme colapsando la parte cliente
- Fijamos la paginación pasando a todos los métodos de consulta de población del CAD el inicio de bloque y el tamaño de bloque como parámetros

Paginación con NHibernate

```
public IList<ClienteEN> dameClientePorCiudad
                    (string p_ciudad, int inicio, int tamaño)
   {
     using (ITransaction tx = session.BeginTransaction ())
     {
         if (size > 0) {
             IQuery q = sess.CreateQuery("from  Cliente cli where
                                                  cli.ciudad = :ciudad);
           q.SetString (“ciudad”, p_ciudad);
           q.SetFirstResult(first);  //<- Inicio de bloque
           q.SetMaxResults(size);  //<- tamaño de bloque
           return q.List();
          }
...  }
```