

Diseño de los Componentes de Proceso

Diseño de Sistemas Software

- Componentes de Proceso
 - Diseño e Implementación
 - Transacciones
 - Estrategia basada en Session Façade
 - Estrategia basada en Application Façade

- La parte más importante de la aplicación es la funcionalidad que ésta proporciona
- Una aplicación realiza un proceso de negocio que puede constar de una o varias tareas
- Cuando existen tareas complejas que requieren varios pasos y transacciones se necesita de un mecanismo para organizar y almacenar el estado hasta que el proceso se haya completado

- El Componente de Proceso (CP) se encarga de exponer al interfaz de usuario los métodos de mayor nivel de la lógica de negocio
- Permite independizar la funcionalidad ofertada de cómo y donde esté realizada la implementación de los métodos invocados

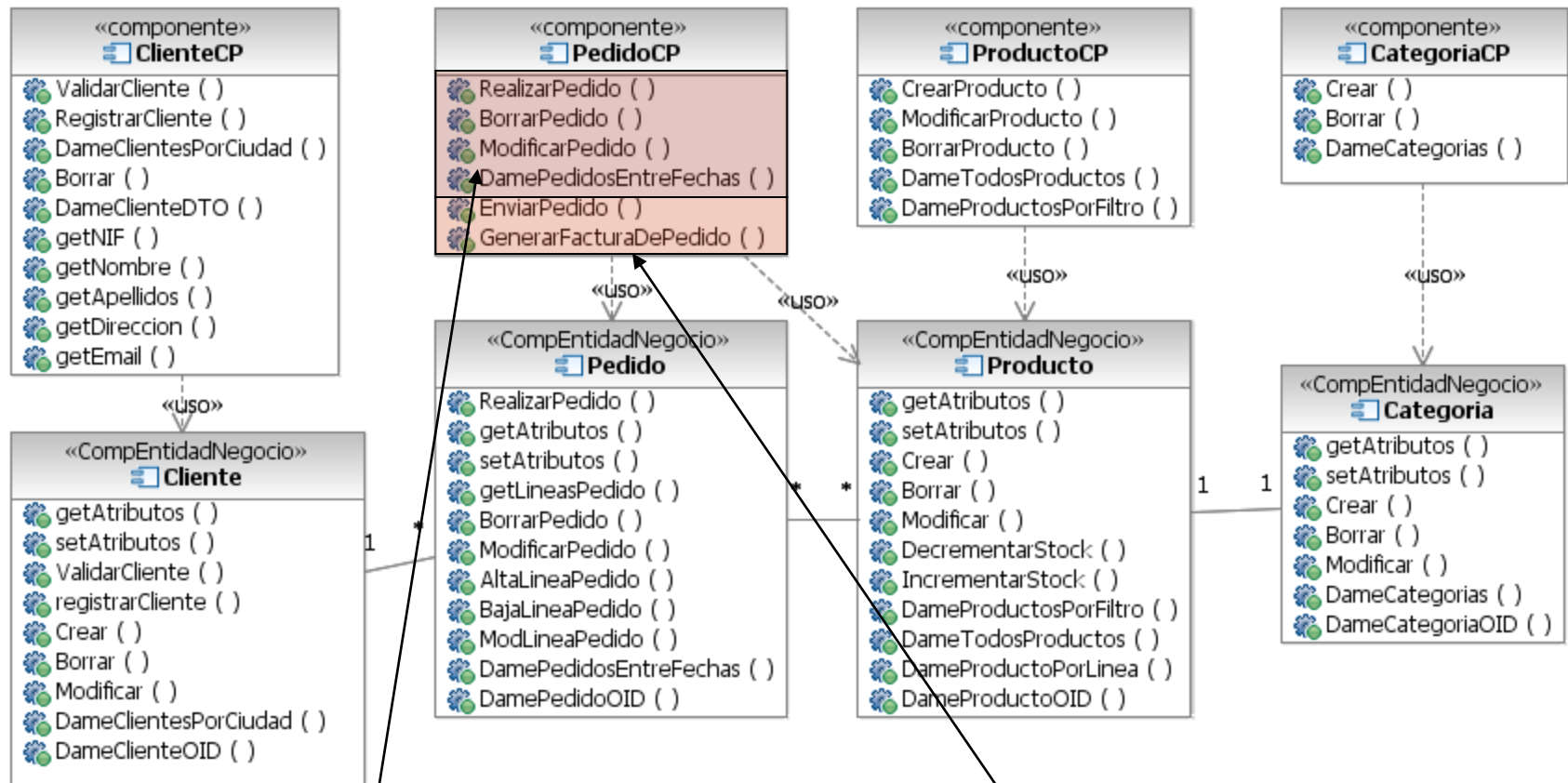
- Un CP contiene la lógica de negocio que involucra a procesos complejos a diferencia del CEN que contiene la lógica de procesos simples (o locales a una entidad)
- Inicia y finaliza las **transacciones** de manera que los Componentes Entidad de Negocio y Componentes de Acceso a Datos estén dentro del contexto estos servicios

- Existen básicamente 2 alternativas:
 - Definir un componente de proceso por cada entidad de negocio de manera que encapsulamos tanto operaciones simples como complejas (solo Session Façade)
 - Definir únicamente un componente de proceso para las operaciones complejas delegando a los CEN las operaciones simples (Application Façade)

- Session Façade (Core J2EE, 2001) tienen la responsabilidad de definir las transacciones y además se le incorpora la responsabilidad de la distribución
- Permite realizar una distribución de los componentes de proceso de forma individual permitiendo así un mayor balanceo de carga y escalabilidad
- Requerido en aplicaciones con lógica de negocio complejas

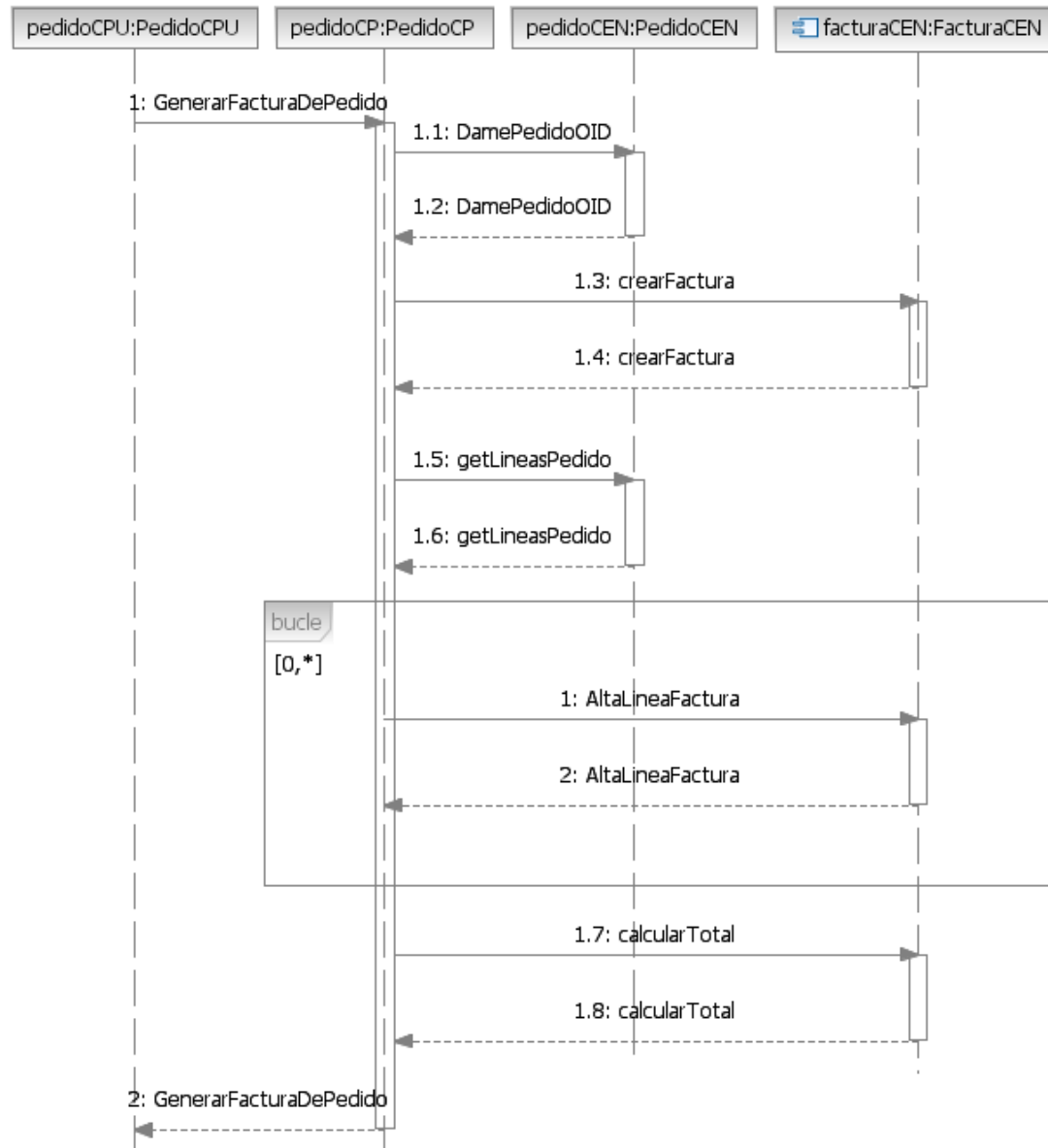
1ª estrategia de Componentes de Proceso

9



Operaciones Simples

Operaciones
Complejas



- Ventajas:

- Mayor escalabilidad soportando la distribución a nivel de componente
- Soporta transacciones distribuidas en diferentes fuentes de datos

- Desventajas:

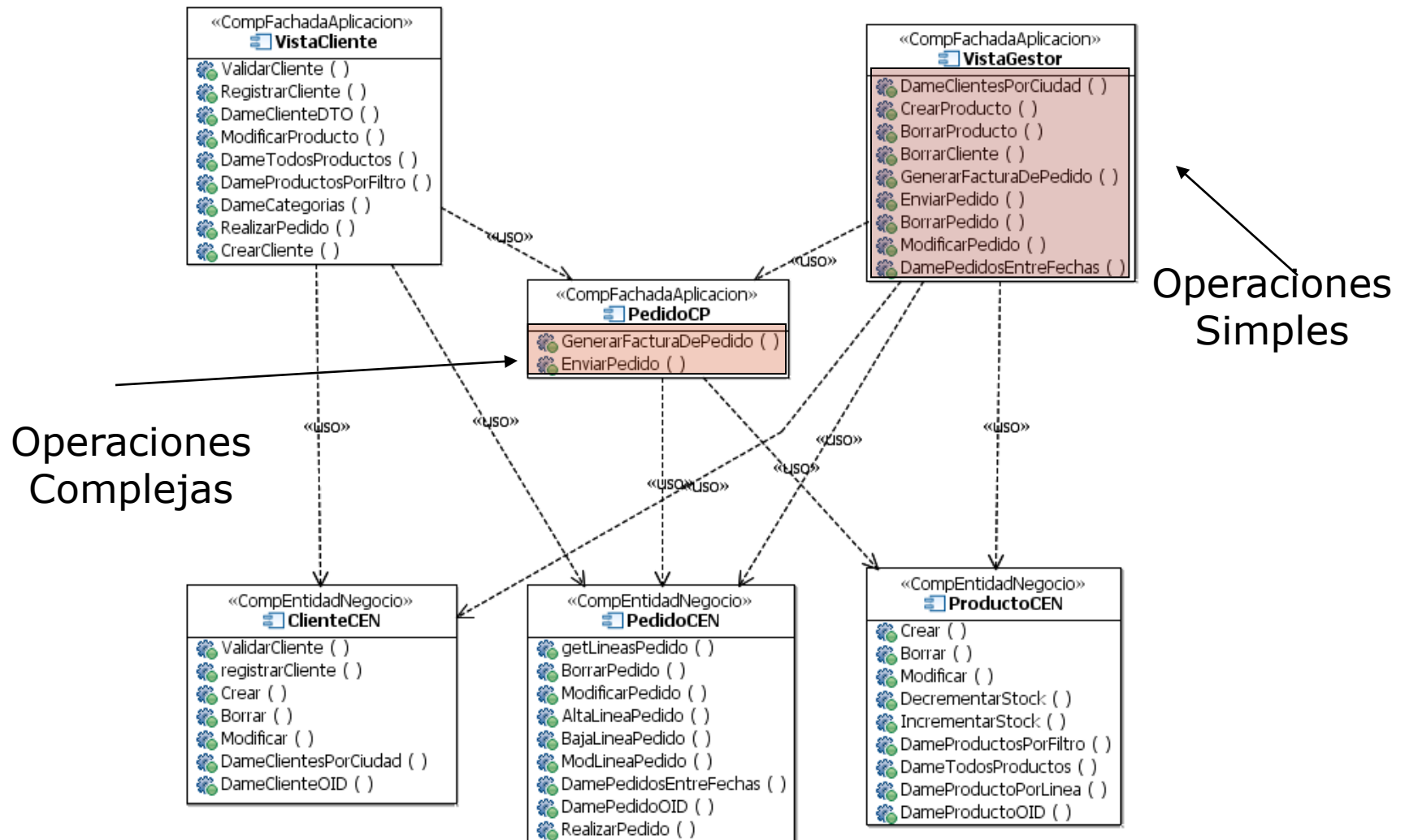
- Mayor acoplamiento en las responsabilidades de distribución y transaccionalidad (reduce el reuso)
- Introduce mayor retardo en las llamadas distribuidas entre componentes reduciendo el rendimiento en operaciones simples

- El CP tiene la responsabilidad únicamente de la transaccionalidad delegando en un componente Application Façade la distribución
- Se definen las operaciones complejas en el CP y las operaciones simples son realizadas en el CEN

- Se define un componente CP para contener todas las operaciones complejas de una entidad de negocio.

Presenta las siguientes características:

- Inicia y termina las transacciones que involucran a más de una entidad de negocio
- Permite reutilizar la lógica de negocio de dichos componentes de proceso por diferentes fachadas de negocio
- Invoca a las CEN que contienen la lógica simple o incluso puede invocar a su vez a otras CPs
- No contiene estados entre diferentes llamadas de la capa cliente




- Optamos por esta segunda estrategia cuando las aplicaciones tienen una lógica de negocio sencilla
- Para ello los CP son **clases de librería** que invocan a los CEN, y que inician las sesiones y transacciones de Nhibernate
- Permite el lazy feching = true en cada entidad EN, ya que en este contexto podemos mantener una sesión durante todas las llamadas de la operación

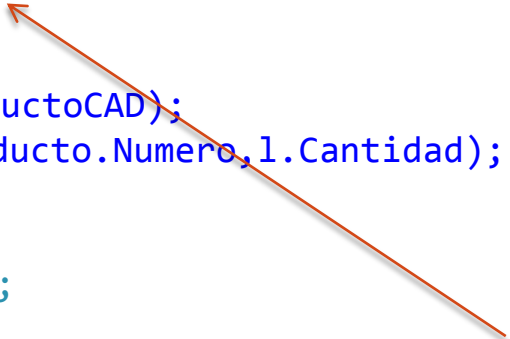
Implementación de la operación “EnviarPedido”

```
public String EnviarPedido(DateTime p_fecha, int p_idPedido)
{
    PedidoCEN pedidoCEN = null;
    ProductoCEN articuloCEN = null;
    try
    {
        SessionInitializeTransaction();
        PedidoCAD pedidoCAD = new PedidoCAD (session);
        ProductoCAD productoCAD = new ProductoCAD (session);
        PedidoEN pedido = pedidoCAD.ReadOIDDefault(p_idPedido);
        foreach (LineaPedidoEN l in pedido.LineaPedido)
        {
            ProductoEN producto = l.Producto;
            ProductoCEN = new ProductoCEN(productoCAD);
            productoCEN.decrementarStock (producto.Numero, l.Cantidad);
        }
        pedido.Fecha = p_fecha;
        pedidoCEN = new PedidoCEN (pedidoCAD);
        pedidoCEN.Modificar (pedido);
        SessionCommit();
    }
    catch (Exception ex)
    {
        SessionRollBack();
        ...
    }
}
```

Se mantiene la
misma sesión para
todas las operaciones
permitiendo Lazy
fetching



Se puede navegar por los
roles de un EN



- Ventajas:

- Reducción del código al solo poner en el CP las operaciones complejas
- Permite una mayor reutilización de los CPs para diferentes fachadas al separar la distribución de la transaccionalidad

- Desventajas

- La granularidad de la distribución se eleva a nivel de capa (mayor acoplamiento funcional de la fachada CP con los componentes CEN)
- Menos posibilidades de balancear la carga

- Una transacción es un conjunto de operaciones realizadas en una única unidad de trabajo
- Su ejecución de forma atómica asegura la consistencia y fiabilidad del sistema
- Todas las operaciones de la transacción han de ser completadas con éxito para que su ejecución se materialice

- Solo existen 2 posibles casos en una transacción:
- Si todo ha ido bien:
 - Commit: Se materializan los cambios realizados por todas las invocaciones
- Si ha habido algún error:
 - Rollback: Deshacen todos los cambios realizados desde el inicio de la transacción

- Existen tres modos de implementarlas en .NET:
 - Utilizando procedimientos almacenados
 - **Transacciones manuales** con ADO.NET (incluido uso frameworks como **Nhibernate** o Enterprise Library)
 - Transacciones automáticas con COM+

- Transacción manual escrita en el T-SQL (SQLServer) encapsula las operaciones dentro de BEGIN TRANSACTION y COMMIT/ROLLBACK TRANSACTION
- Solo permite lanzar transacciones de forma local a un gestor de BBDD
- Se obtiene un excelente rendimiento permitiendo lanzar una transacción compleja con solo una invocación al gestor

```
CREATE PROCEDURE Proc1  
AS
```

```
-- Se inicia la transacción
```

```
BEGIN TRANSACTION
```

```
-- Se realizan las llamadas SQL
```

```
-- Se comprueba si hay algún error
```

```
If @@Error <> 0
```

```
-- Rollback de la transacción
```

```
ROLLBACK TRANSACTION
```

```
...
```

```
-- Si todo ha ido bien se  
realiza un Commit de la  
Transacción
```

```
COMMIT TRANSACTION
```

- Ventajas

- Alto rendimiento al ser local al gestor y solo requerir una llamada a la BBDD
- Proporciona flexibilidad para explicitar el ámbito de la transacción

- Desventajas:

- Aprender Transact-SQL o el lenguaje del gestor, aumenta la curva de aprendizaje
- Poco portable ya que el código es dependiente del gestor
- La escalabilidad está limitada por los servidores de datos


- ADO.NET proporciona un objeto transacción (SQLTransaction) usado para comenzar la transacción y para controlar explícitamente si debe hacerse commit o rollback
- Dicho objeto esta ligado a una conexión de BBDD, así que en nuestra arquitectura se ha de implementar en una clase CAD

- Ventajas:
 - Fáciles de programar
 - Flexibilidad para controlar la transacción con instrucciones explícitas
 - Soluciones como NHibernate permiten mantener en el CP el ámbito de dichas transacciones
- Desventajas:
 - Necesita más invocaciones al gestor que en el procedimiento almacenado
 - Solo admite transacciones en un único gestor de BBDD

- Los CPs heredan de basicCP.cs que tiene el atributo booleano `sessionStarted` indicando si se ha iniciado la sesión o no
- Además se implementan los métodos `SessionInitializeTransation`, `SessionCommit` y `SessionRollback`
- Permite crear o no una transacción en un CP
- Permite que un CP llame a otro CP y se conserve la misma session y transacción de NHibernate

```
public class PedidoCP: basicCP
{
    public String EnviarPedidoYGenerarFactura(DateTime p_fecha, int p_idPedido)
    {
        PedidoCEN pedidoCEN = null;
        ProductoCEN articuloCEN = null;
        try
        {
            SessionInitializeTransaction();
            FacturaCP facturaCP = new CFacturaCP (session);
            sessionStarted = false;
            EnviarPedido (p_fecha, p_idPedido);
            sessionStarted = true;
            PedidoEN pedido = pedidoCEN.ReadOID(p_idPedido);
            facturaCP.GenerarFactura (pedido);
            SessionCommit();
        }
        catch (Exception ex)
        {
            SessionRollBack();
            ...
        }
    }
}
```

Se mantiene el atributo sesión entre diferentes llamadas



Ejemplo CP que invoca a otros CPs

```
public int RestarStockEnviarPedido(String p_descripcion, Nullable<DateTime> p_fechaRealizacion,
Ilist<LineaPedidoEN> p_lineaPedido, String p_cliente, Nullable<DateTime> p_fechaEnvio, String
p_estado) {

    try {

        ...

        SessionInitializeTransaction();           //Llamada a CP de restar stock de los articulos
        articuloCP = new ArticuloCP(session); //Le pasamos la sesión así al CP invocado
        articuloCP.RestarStock(p_lineaPedido);

        _IPedidoCAD = new PedidoCAD(session);

        pedidoCEN = new PedidoCEN(_IPedidoCAD);    //Creo el pedido

        sessionStarted = false; // Al llamar a otra operación dentro del mismo CP tenemos que
poner la variable sessionStarted a false, para que no finalice la transacción.

        oid = CrearPedidoCP(p_descripcion, p_fechaRealizacion, p_lineaPedido, p_cliente,
p_fechaEnvio, p_estado);

        sessionStarted = true; // Volvemos ponerla a true para haga commit.

        SessionCommit();

        ....
    }
```