Fecha de confección:	08/08/2007
Título:	Arquitectura Tecnológica Simple
Temática:	Aplicativos .NET
Confeccionado por:	Marcelo Oviedo
Revisado por:	Sebastian Biglia
Aprobado por	Diego Raspanti
Clasificación:	Público
Auditorio:	Desarrollo Action Line
Fecha de Impresión:	07/10/2008 10:21:11
Versión	1.0

Alcance

Esta normativa es aplicable a todo desarrollo de aplicaciones en .NET, ya sea internas de Grupo Action Line o Tercerizadas.

Elaboración / Revisión / Aprobación

El presente documento es elaborado por el Arquitecto Líder de la División QA, Revisado por el Responsable de División QA y aprobado por la Gerencia de Desarrollo.

Roles y Responsabilidades

Rol	Responsabilidad
Gerencia de desarrollo	Aprobar el estándar de Desarrollo.
Arquitecto Líder	Definir Estándares y auditar el estándar.
Responsable División QA	Revisión general del estándar.

Introducción

Las Soluciones de Negocio Corporativas deben, actualmente, enfrentar muchas complejidades tecnológicas y responder a altas expectativas de sus usuarios. No solo tienen que responder a variables relacionadas con Alta Disponibilidad, Escalabilidad y Alto Rendimiento, muchas veces en un contexto poco predecible, sino que también tienen que dar respuestas a un negocio altamente cambiante sin afectar la calidad de las soluciones y manteniendo el mejor nivel de productividad posible.

Las mejores soluciones son aquellas compuestas por un conjunto de simples mecanismos que resuelven escenarios recurrentes en forma confiable y eficiente. Durante el proceso de desarrollo de grandes soluciones, la combinación de estos simples mecanismos permite la creación de sistemas con requerimientos complejos.

Este documento define una *Arquitectura para Aplicaciones Corporativas* como la suma de patrones que resuelven problemas simples, aplicando a su vez un conjunto de buenas prácticas, probadas e implementadas en sistemas de gran envergadura.

Por otro lado, se ha elegido utilizar Plataforma .NET (Windows 2005, Framework .NET 2.0, VS.NET, Etc.) para soportar toda la problemática tecnológica que deriva de los tres punto enunciados con anterioridad.

Propósito

El presente documento tiene como propósito definir el diseño de la arquitectura tecnológica sobre la cual se implementará toda solución de Action Line.
Dicho documento definirá un modelo prescriptivo que permita a Action Line desarrollar con una arquitectura basada en las siguientes variables:

- Alta disponibilidad
- Escalabilidad
- Productividad
- Performance

Es importante destacar que a efectos de contar con una adecuada transferencia de conocimientos, la elaboración de dicho documento es producto de un trabajo en conjunto de los distintos equipos de trabajo de Action Line y del área QA que centraliza la información de una forma unificada.

Audiencia

Dicho documento debe ser material de lectura para los siguientes equipos de trabajo:

- Equipo de arquitectura de Action Line SA, compuesto por recursos de la compañía y de socios de negocio.
- Equipo dedicado a llevar a cabo el diseño lógico del Sistema en cuestión.
- Equipo de desarrollo compuesto por recursos de la compañía y de socios de negocio.
- Program Manager de la solución que se está desarrollando.
- Todo recurso técnico que Action Line considere necesario.

Estructura del documento

El documento esta compuesto por una sección que define el diseño de arquitectura conceptual y tecnológica. La sección que define el diseño de la arquitectura de software se encuentra bajo el nombre *"Arquitectura para aplicaciones corporativas"*.

Definiciones, acrónimos y abreviaturas

La siguiente tabla contiene la lista de definiciones, acrónimos y abreviaturas que se aplicarán en el documento.

Definición / Acrónimo / Abreviatura	Descripción
MS	Microsoft
BC	Business Components / Componentes de negocio
SSC	Service Support Component
BSC	Business Support Component
BE	Business Entities / Entidades de negocio
SC	Servicio de consulta.
ST	Servicio transaccional.
DALC	Data Access Layer Components / Capa de componentes de acceso a los datos.
UI	User Interface / Interfase de usuario
SP	Stored Procedure / Procedimiento almacenados en la Base de Datos que permiten el acceso y mantenimiento de los datos.
DER	Diagrama de Entidad-Relación (Modelo de datos)
DB	Database / Base de datos
Roundtrip	Envío de requerimientos y espera de respuestas entre las diferentes capas físicas de la arquitectura. Ejemplos: Accesos a la DB, Ejecución de servicios de consulta o transaccionales.
DS	DataSet, estructura de datos que se utilizará para representar entidades.
DS Tipificados	Son DSs que contienen propiedades, métodos y eventos que están fuertemente ligados a una entidad
Request	Define al conjunto de datos que será enviado a la capa de servicios, desde la capa física de presentación y que corresponderá al parámetro de entrada de un servicio transaccional o de consulta.
Response	Define al conjunto de datos retornado por un servicio transaccional o de consulta.
CRUD	Acrónimo (inglés) que sintetiza las operaciones básicas sobre una entidad: C = Crear, R = Leer, U = Actualizar, D = Borrar.
ACID	Acrónimo (inglés) utilizado para definir el nivel de integridad y consistencia de datos que garantiza un sistema. A = Atomicidad, C = Consistencia, I = Aislamiento, D = Durabilidad.
MVC	Acrónimo (inglés) utilizado para definir el modelo de interacción entre la capa de presentación y el modelo del negocio. M = Modelo, V = Vista, C = Controlador.
OLTP	On-Line Transactional Processing, este concepto hace referencia al sistema transaccional de un sistema de información.
DAC	Data Access Component – Componentes de Acceso a Datos.
SOA	Service Oriented Applications – Aplicaciones orientadas a la ejecución de servicios de negocio.
SN	Servicio de Negocio – Se define como servicio de negocio a toda clase .NET que hereda de BusinessComponent , convirtiendo de esta manera a la clase en un servicio que expone lógica de negocio a los diferentes canales de acceso.
QA	Quality assurance (QA)

Arquitectura para aplicaciones corporativas

La gran mayoría de los sistemas se basan en la obtención de datos de un repositorio, presentarlo a través de una determinada tecnología al usuario y luego impactar los cambios realizados por el usuario al repositorio de datos. Dado que el comportamiento de los sistemas de información se acerca al supuesto mencionado, es posible que los arquitectos de soluciones se inclinen a acoplar la capa de datos a la de presentación. Si bien esto es posible trae algunos inconvenientes:

La lógica de presentación cambia con mayor frecuencia que la lógica de negocio.

Cada vez es mas necesario presentar la misma información en diferentes tipos de tecnologías / dispositivos.

Existen necesidades de conocimientos técnicos diferentes para desarrollar lógica de presentación respecto al desarrollo de lógica de negocio.

Complejidad para detectar errores dentro de la solución.

Dada la distribución de capas lógicas, encontramos que el patrón MVC (Model – View – Controller), es el mas adecuado para dar soporte a las necesidades y requerimientos del sistema transaccional. Bajo este modelo la presentación de datos al usuario representa a la *Vista*, el encargado de ejecutar una acción en función de un evento de entrada es el *Controlador*, y el conjunto de capas lógicas definidas bajo los Servicios de Negocio representan al *Modelo*.

DISEÑO CONCEPTUAL DEL MODELO MVC

En esta sección se proveen las definiciones básicas correspondientes al modelo MVC del sistema transaccional del Sistema.

Modelo

El modelo esta compuesto por una serie de servicios y componentes conceptualmente bien identificados y con objetivos y responsabilidades diferentes.

- División en 3 áreas
  - Modelo: Encapsula los datos y la información
  - Vista: Muestra la información al usuario
  - Controlador: Interpreta las acciones del usuario

- Ventajas
  - Separación total entre lógica de negocio y presentación.
  - A esto se le pueden aplicar opciones como el multilenguaje
  - Distintos diseños de presentación.

El siguiente gráfico visualiza las capas lógicas que conforman dicho modelo:

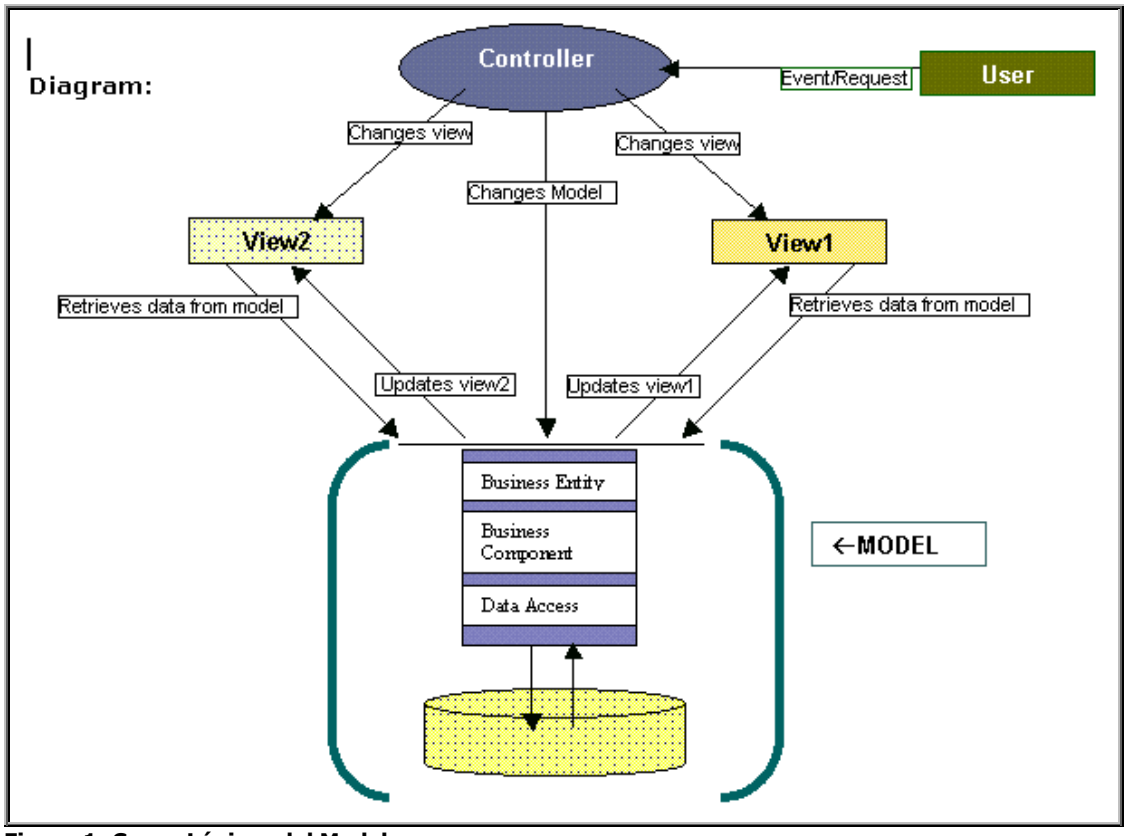


Figura 1: Capas Lógicas del Modelo

Tipos de componentes

El análisis de la mayoría de las soluciones empresariales basadas en modelos de componentes por capas muestra que existen varios tipos de componentes habituales. En la figura 2 se muestra una ilustración completa en la que se indican estos tipos de componentes.[1]

Aunque la lista que se muestra en la figura 2 no es completa, representa los tipos de componentes de software más comunes encontrados en la mayoría de las soluciones distribuidas. Describiremos en profundidad cada uno de estos tipos.

[1] **Nota:** El término componente hace referencia a una de las partes de la solución total, como los componentes de software compilado (por ejemplo, los ensamblados de Microsoft .NET) y otros elementos de software, como las páginas Web y otros programas de Microsoft® como por Ejemplo Microsoft® BizTalk® Server Orchestration.

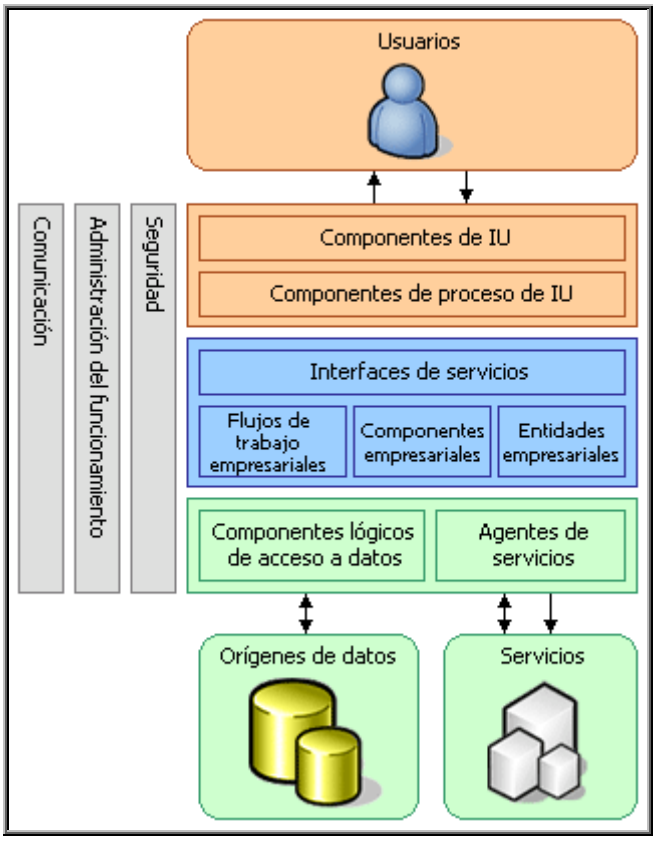


Figura 2. Tipos de componentes utilizados en el escenario comercial de ejemplo

Los tipos de componentes identificados en el escenario de diseño de ejemplo son:

1. **Componentes de interfaz de usuario (IU)**. La mayor parte de las soluciones necesitan ofrecer al usuario un modo de interactuar con la aplicación. En el ejemplo de aplicación comercial, un sitio Web permite al cliente ver productos y realizar pedidos, y una aplicación basada en el entorno operativo Microsoft Windows® permite a los representantes de ventas escribir los datos de los pedidos de los clientes que han telefoneado a la empresa. Las interfaces de usuario se implementan utilizando formularios de Windows Forms, páginas Microsoft ASP.NET, controles u otro tipo de tecnología que permita procesar y dar formato a los datos de los usuarios, así como adquirir y validar los datos entrantes procedentes de éstos.
2. **Componentes de proceso de usuario**. En un gran número de casos, la interactuación del usuario con el sistema se realiza de acuerdo a un proceso predecible. Por ejemplo, en la aplicación comercial, podríamos implementar un procedimiento que permita ver los datos del producto. De este modo, el usuario puede seleccionar una categoría de una lista de categorías de productos disponibles y, a continuación, elegir uno de los productos de la categoría seleccionada para ver los detalles correspondientes. Del mismo modo, cuando el usuario realiza una compra, la interactuación sigue un proceso predecible de recolección de datos por parte del usuario, por el cual éste en primer lugar proporciona los detalles de los productos que desea adquirir, a continuación los detalles de pago y, por último, la información para el envío. Para facilitar la sincronización y organización de las interactuaciones con el usuario, resulta útil utilizar componentes de proceso de usuario individuales. De este modo, el flujo del proceso y la lógica de administración de estado no se incluye en el código de los elementos de la interfaz de usuario, por lo que varias interfaces podrán utilizar el mismo "motor" de interactuación básica.

3. **Flujos de trabajo empresariales**. Una vez que el proceso de usuario ha recopilado los datos necesarios, éstos se pueden utilizar para realizar un proceso empresarial. Por ejemplo, tras enviar los detalles del producto, el pago y el envío a la aplicación comercial, puede comenzar el proceso de cobro del pago y preparación del envío. Gran parte de los procesos empresariales conllevan la realización de varios pasos, los cuales se deben organizar y llevar a acabo en un orden determinado. Por ejemplo, el sistema empresarial necesita calcular el valor total del pedido, validar la información de la tarjeta de crédito, procesar el pago de la misma y preparar el envío del producto. El tiempo que este proceso puede tardar en completarse es indeterminado, por lo que sería preciso administrar las tareas necesarias, así como los datos requeridos para llevarlas a cabo. Los flujos de trabajo empresariales definen y coordinan los procesos empresariales de varios pasos de ejecución larga y se pueden implementar utilizando herramientas de administración de procesos empresariales, como BizTalk Server Orchestration.
4. **Componentes empresariales**. Independientemente de si el proceso empresarial consta de un único paso o de un flujo de trabajo organizado, la aplicación requerirá probablemente el uso de componentes que implementen reglas empresariales y realicen tareas empresariales. Por ejemplo, en la aplicación comercial, deberá implementar una funcionalidad que calcule el precio total del pedido y agregue el costo adicional correspondiente por el envío del mismo. Los componentes empresariales implementan la lógica empresarial de la aplicación.
5. **Agentes de servicios**. Cuando un componente empresarial requiere el uso de la funcionalidad proporcionada por un servicio externo, tal vez sea necesario hacer uso de código para administrar la semántica de la comunicación con dicho servicio. Por ejemplo, los componentes empresariales de la aplicación comercial descrita anteriormente podría utilizar un agente de servicios para administrar la comunicación con el servicio de autorización de tarjetas de crédito y utilizar un segundo agente de servicios para controlar las conversaciones con el servicio de mensajería. Los agentes de servicios permiten aislar las idiosincrasias de las llamadas a varios servicios desde la aplicación y pueden proporcionar servicios adicionales, como la asignación básica del formato de los datos que expone el servicio al formato que requiere la aplicación.
6. **Interfaces de servicios**. Para exponer lógica empresarial como un servicio, es necesario crear interfaces de servicios que admitan los contratos de comunicación (comunicación basada en mensajes, formatos, protocolos, seguridad y excepciones, entre otros) que requieren los clientes. Por ejemplo, el servicio de autorización de tarjetas de crédito debe exponer una interfaz de servicios que describa la funcionalidad que ofrece el servicio, así como la semántica de comunicación requerida para llamar al mismo. Las interfaces de servicios también se denominan *fachadas empresariales*.
7. **Componentes lógicos de acceso a datos**. La mayoría de las aplicaciones y servicios necesitan obtener acceso a un almacén de datos en un momento determinado del proceso empresarial. Por ejemplo, la aplicación empresarial necesita recuperar los datos de los productos de una base de datos para mostrar al usuario los detalles de los mismos, así como insertar dicha información en la base de datos cuando un usuario realiza un pedido. Por tanto, es razonable abstraer la lógica necesaria para obtener acceso a los datos en un capa independiente de componentes lógicos de acceso a datos, ya que de este modo se centraliza la funcionalidad de acceso a datos y se facilita la configuración y el mantenimiento de la misma.
8. **Componentes de entidad empresarial**. La mayoría de la aplicaciones requieren el paso de datos entre distintos componentes. Por ejemplo, en la aplicación comercial es necesario pasar una lista de productos de los componentes lógicos de acceso a datos a los componentes de la interfaz de usuario para que éste pueda visualizar

dicha lista. Los datos se utilizan para representar entidades empresariales del mundo real, como productos o pedidos.

9. **Componentes de seguridad, administración operativa y comunicación**. La aplicación probablemente utilice también componentes para realizar la administración de excepciones, autorizar a los usuarios a que realicen tareas determinadas y comunicarse con otros servicios y aplicaciones.

Visión conceptual de la arquitectura del Sistema transaccional

Antes de comenzar con las definiciones técnicas de la arquitectura propuesta, es necesario entender cuales son los conceptos básicos planteados en dicha arquitectura. A efecto de comprender los bloques conceptuales que definen la *Arquitectura Tecnológica*, a continuación se visualiza un gráfico exponiendo los diferentes componentes que integran la Visión Conceptual de la Arquitectura Simple.

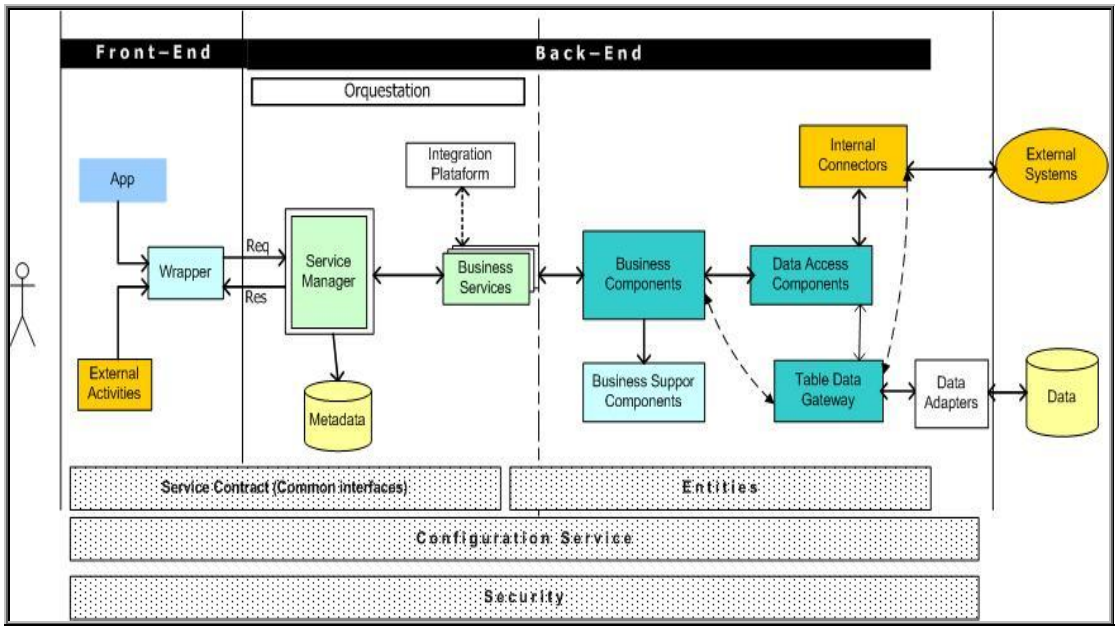


Figura 3. Tipos de componentes utilizados en el escenario comercial de ejemplo

A efectos de cumplimentar esta premisa básica, hemos decidido diseñar un sistema basado en capas (Layers). Bajo este patrón, denominado Múltiples Capas (Multilayer), se definen dos capas bien diferenciables:

Front End System
Back End System.

A continuación iremos describiendo cada uno de los componentes visualizados en el gráfico anterior. Los componentes se describirán comenzando de izquierda a derecha, en función de las capa físicas que conforman la arquitectura.

FRONT-END O CAPA DE PRESENTACIÓN

La capa de presentación es la encargada de brindar al usuario un entorno de trabajo, en el cual se listan sus actividades y tareas. Por otro lado, es quien interactúa con la capa de Middleware para ejecutar servicios de consulta o transaccionales.
Es la capa que contiene las siguientes funcionalidades de los aplicativos:

Obtener datos del Sistema de Información, a través de servicios de consulta.
Actualizar el Sistema de Información a través de servicios transaccionales
Presentación de datos al usuario a través de formularios, utilizando diferentes tipos de tecnologías.
Piezas de software que permiten la interacción entre los formularios y los servicios de negocio.

Actividades propias del sistema

Las actividades representan a una o mas acciones que interactúan con usuarios del sistema de información. Un usuario del sistema de información puede estar representado por una persona física, una aplicación, un proceso, una máquina de estado, etc.

Las actividades forman parte del sistema de información y comúnmente corresponden a una actividad de un proceso de negocio.

Las actividades pueden ser ejecutadas desde diferentes tecnologías / dispositivos: Web Browsers, Celulares, Web Services, Etc..

Actividades Externas

Las actividades externas son aquellas que no forman parte directa del sistema de información, sino que son extensiones al mismo. Un ejemplo de una actividad externa podría ser un requerimiento de datos, desde el aplicativo de un proveedor.

Lo importante a destacar en este punto es que las actividades externas acceden al sistema de información de la misma forma que las actividades de los procesos de negocio del sistema de información.

Wrapper de aplicación

El Wrapper es el único que puede interactuar con la capa de servicios de negocio para resolver requerimientos de negocio. Dicha interacción se realiza a través de un adaptador de un canal, enviando un Request y esperando la respuesta en un Response.

Los adaptadores de canal son piezas de código que permiten desacoplar el envío de un requerimiento a un servicio de negocio, con su correspondiente respuesta, de la tecnología de transporte necesaria para acceder a dicho servicio de negocio (de consulta o transaccional).

El sistema de adaptadores de canal es extensible, de esta forma se podrían ampliar los tipos de canales de acceso a los servicios. A través de ellos se podrían ejecutar servicios desde Web Services, Dispositivos Real-Time, Páginas WEB, Aplicaciones Winform, Etc.

BACKEND

El Back-End es la capa física que cumple tres funciones básicas:

Brindar soporte al sistema transaccional (OLTP).
Brindar soporte a los servicios de negocio.
Brindar soporte a las problemáticas de integración.

Por lo tanto expone servicios a los aplicativos del Front-End. Es la intermediaria entre los requerimientos de los aplicativos y los componentes del Back-End.

El Back-End se encuentra conformado por las siguientes subsistemas y capas lógicas:

Dispatcher (SOA – Service Oriented Applications)
Business Services
Servicios sincrónicos / asincrónicos
Servicios transaccionales
Servicios de consulta
Componentes de negocio y de soporte al negocio (BC y BSC)
Componentes de acceso a datos (DAC)
Plataforma de integración

Dispatcher de servicios

Es el encargado de recibir los requerimientos realizados por los diferentes clientes (*Request*), ubicar el servicio requerido dentro del catálogo de acciones, ejecutarlo bajo un contexto controlado y condicionado por la configuración establecida en el catálogo, y retornar al cliente el resultado de esta ejecución (a través de un objeto de tipo *Response*). A través del *Dispatcher* se exponen todas los servicios de negocio (de consulta y transaccionales) del Sistema.

El Dispatcher es el único punto de entrada a los servicios de negocio (transaccionales o de consulta) del Sistema.

El Dispatcher brinda un contexto de ejecución para los servicios y provee funcionalidades como:

1- Entorno transaccional

2- Control de ejecución de servicios

3- Sistema de mensajes y configuración

4- Manejo de excepciones

5- Sistema de conectores para acceder a plataformas no integradas nativamente al sistema de información. (Ej. SAP, BizTalk, Otros Servicios nuevos o preexistentes del cliente)

Business Services

Los Business Services representan a cualquier tipo de servicio, administrados por el Dispatcher ya sean transaccionales o de consulta, sincronos o asíncronos. Los servicios se pueden clasificar de la siguiente manera:

Servicios sincrónicos: se deben utilizar, en los casos en los cuales exista interacción con el sistema de base de datos del Sistema o con sistema transaccionales cuyos de tiempos de respuestas sean óptimos.

Servicios asincrónicos: se deben utilizar en los casos en los cuales se requiera interactuar con sistemas no transaccionales, o en los cuales los tiempos de procesamiento no son óptimos o todo el procesamiento de un determinado paso de la transacción se realiza en sistemas externos. Un ejemplo que muestra la necesidad de utilizar servicios asincrónicos podría ser una transacción que interactúa con un mainframe bajo un patrón de diseño de Request / Response es decir el servicio envía un mensaje al mainframe y queda a la espera de una respuesta.

Servicios transaccionales: son aquellos que actualizan las bases de datos del sistema. La interacción con dichos subsistemas se realizará a través de los Componentes de Negocio o Componentes de Soporte a Servicios.

Servicios de consulta: son aquellos que solo obtienen datos del sistema de información. En este caso, el acceso al sistema de información también es realizado por los mismos mecanismos.

Plataforma de integración

La plataforma de integración se compone por un conjunto de componentes y servicios que permiten la interacción entre el sistema de información y sistemas externos .

La plataforma de integración puede recibir eventos provenientes del Dispatcher y es responsable de entregarlos en forma confiable a los sistemas externos. Por otro lado, tiene la capacidad de recibir eventos provenientes de sistemas externos que deban impactar en una base de datos del sistema o ejecutar un servicio transaccional o de consulta.

A continuación se describen algunos de los servicios provistos por la plataforma de integración:

- Envío confiable de mensajes
- Firma / Validación de certificados digitales
- Encriptación / Desencriptación de mensajes
- Auditoria de los mensajes intercambiados
- Validación del formato de los mensajes
- Transformación de un tipo de mensaje entrante a un tipo de mensaje saliente
- Sistema de adaptadores a sistemas externos extensibles, a través de la exposición de APIs.
- Herramientas para la administración de la plataforma y el diseño de las interfases.

Nota: En este documento es todo lo que se detallara sobre plataforma de integración ya que es un amplio tema que escapa a los fines prácticos del objetivo principal de una propuesta de arquitectura simple. Además se explicaran solo a nivel conceptual los servicios asíncronos y los conectores que utiliza la plataforma de integración, la implementación técnica de los mismos serán detallados en un documento anexo a la propuesta de arquitectura simple.

Business components (BC)

Las BC (Business components) contiene la lógica del negocio de la Solución. A través de los componentes de negocio se realizan validaciones, se obtienen datos del sistema de persistencia, se actualiza el sistema de información, etc.
Los componentes de negocio no acceden a la base de datos en forma directa, lo hacen a través de los componentes de acceso a datos (DAC).

Connectores

Los conectores son piezas de software destinados a interactuar con los sistemas externos en aquellas interfases basadas en un patrón de tipo Request / Response.

El Framework de Action Line provee un subsistema de conectores que permite ejecutar servicios de sistemas externos a través de una misma interfaz de programación. Facilitando de esta manera la tarea de desarrollar servicios de negocio que requieren integrarse en-línea con sistemas externos. De esta manera, toda la complejidad de integración se encuentra

encapsulada en piezas de software a la cual denominamos Conectores del Framework de Action Line.

Como ejemplo de un conector del Framework podemos citar al que permite ejecutar requerimientos sincrónicos a SAP a través de RFCs.

Data Access Logic Components (DALC) o DAC

DALC es el acrónimo de Data Access Logic Components (componentes de lógica de acceso a datos). Dichos componentes son los que tienen el conocimiento de cómo interactuar con el sistema de persistencia (Ej. Base de Datos SQL Server).

Data Adapters

Los adaptadores para el acceso a datos son piezas de software que permiten interactuar a los componentes de acceso a datos (DAC) con la Base de Datos. ADO.NET es un ejemplo de un adaptador para el acceso a datos.

Database

La base de datos es el repositorio de datos para el Sistema de Información. Es una base de datos relacional que contiene el modelo de entidad-relación correspondiente a dicho sistema de información.

External Systems

Los sistemas externos son aquellos que no constituyen un componente nativo del sistema de información pero que son requeridos para la operación diaria del negocio.

La interacción con los sistemas externos puede ser Punto a punto asincrónica sin espera de respuesta, Punto a punto asincrónica con espera de respuesta, Punto a punto sincrónica.

No se debe confundir una interfase asincrónica con un proceso batch. Por ejemplo, una interfase de tipo *Punto a punto asincrónica sin espera de respuesta* podría agregar registros a un archivo de texto plano que será procesado a una determinada hora del día.

Especificaciones técnicas

En esta sección se especifican con mayor nivel de detalle lo aspectos técnicos de los componentes definidos en el modelo anterior.

Como se puede visualizar en el documento el único punto de entrada a los servicios de negocio es el Dispatcher. Así como también se puede apreciar en el diagrama que el único punto de acceso al sistema de persistencia del Sistema son los Componentes de Acceso a Datos (DAC).

El resto de las clases del diagrama intentan representar de forma genérica como serán implementados los diferentes escenarios de uso definidos para la capa de Servicio & Dominio.

Nota: Todos los nombres de componentes, clases y servicios, así como también los Namespaces visualizados en los ejemplos de código, son ficticios. El sistema de nomenclatura de Namesapces formará parte de este documento, las normas con respecto a componentes, clases, servicios se definirán en un documento separado.

CLIENTES

Las aplicaciones clientes pueden ser aplicaciones web, win 32 o incluso algún otro servicio que requiera hacer peticiones de algún servicio al Dispatcher.

Desde el punto de vista del cliente las piezas más importantes son:

Request
Response
Wrapper

Requests y Responses

Los Request y los Response son clases de .Net que están especialmente diseñadas para cada servicio en particular. Es decir toda estructura de un Request se adapta especialmente a cumplir los requisitos que necesita un determinado servicio para poder ser ejecutado y por otro lado, todo estructura Response representa la forma en que dicho servicio retornara información de respuesta al cliente.

Por lo tanto todo servicio dispone de un Request y un Response.

Tanto los Request como los Response implementan una interfase `IServiceContract` como se muestra a continuación. Esta implementación se hace implícitamente cuando heredan de las clases bases : `Request<T>` y `Response<T>` respectivamente.

```
ClaseRequest : Action Line.Framework.Bases.BackEnd.Request<T>
ClaseResponse : Action Line.Framework.Bases.BackEnd.Response<T>
Y
Request<T> : ServiceContractBase <T> where T : IEntity
Response<T> : ServiceContractBase <T> where T : IEntity
```

Done la clase **T** que recibe de forma generica es cualquier entidad o clase que implemente de `IEntity.`

Esta clase, que implemeta `IEntity,` es la que realmente tiene toda la estructura de negocio que nececita el servicio.

Las estructuras de las clases `Entity/Entities` que reciben los `Request` o `Responses` seran reprecentados por esquemas XSD. Esto es para disponer de una vista mas rapida y comoda de estos objetos y ademas para que puedan ser utilizados para generar clases automaticamente.

Los `Request` y `Response` son clases que no tienen comportamiento ni atributos alguno en la definicion de su cuerpo ya que las funcionalidades mas abituales estan absorvidas por los metodos heredados de sus clases bases.

Queda a criterio del desarrollador incluir alguna funcionalidad adicional a estos objetos.
Alguno de los metodos o atributos heredados son :

Método o Atributo	**Descripción**
BusinessData	Este atributo representa la clase contenida dentro del Request o Response que hereda de Entity y contiene toda información funcional o de negocio.
SetXml(string pXMLService)	Rellena el Request o Response con la información del xml. Contiene tanto información de contexto como la funcional.
GetXml()	Obtiene el xml del Request o response.. Contiene tanto información de contexto como la funcional
ContextInformation	Información de contexto acerca del Request o response del servicio.
SetContextInformationXml()	Inicializa los datos contexto que pertenecen al Request con el contenido del xml.-
GetContextInformationXml()	Retorna el xml del Información de contexto que pertenece al Request o Response.-
InitializeServerContextInformation()	Establece la información de contexto del Request o Response del lado del despachador de servicios.-
InitializeHostContextInformation()	Establece la información de contexto del Request o Response del lado del cliente.-
SetBusinessDataXml(string pXMLData)	Inicializa los datos de negocio que pertenecen al Param o Result con el contenido del xml, dependiendo de si se trata de un Request o Response respectivamente.
GetBusinessDataXml()	Retorna el xml del Param o Result que pertenece al Request o Response respectivamente.-
Errors	Solo valido para los objetos que heredan de Response. Y representa cualquier tipo de error ocurrido desde que el despachador de servicio lanzo la ejecución de algún BusinessService

Context Information

El atributo `ContextInformation` es muy importante para obtener alguna informacion extra acerca del estado del servicio.

La siguiente tabla lista los atributos que dispone:

Propiedad	**Descripción**
HostName	Indica el host que inicio la peticion del servicio .
ServerName	Indica el server que atendio el servicio servicio .
UserName	Indica el usuario logueado en el host que inicio el servicio.
ServerTime	Indica fecha y hora de inicio o fin del servicio del lado del Servidor o despachador de servicio. Para un Request : fecha y hora de inicio del lado del server . Para un Response: fecha y hora de finalizacion del lado del server .

Propiedad	Descripción
HostTime	Indica fecha y hora de inicio o fin del servicio del lado del Cliente. Para un Request: fecha y hora de inicio del lado del cliente (horario de inicio desde el host). Para un Response: fecha y hora de finalizacion del lado del cliente (horario de llegada al host).

Wrapper

Esta pieza de software esta ubicada del lado del cliente y es la encargada de recibir todas las peticiones que los clientes hacen al servidor o distpatcher.

Este componente tiene la inteligencia para establecer la informacion de contexto del lado del cliente de los servicios y ademas se encarga de la serializacion automatica de los parametros Request o Responce para enviarlos al servidor.

Por otro lado detecta la configuracion de canal para poder ubicar al dispatcher y abstrae al cliente de toda comlejidad de comunicación ante un dispatcher montado en un WebService o un Servicio de windows remoting.

El Wrapper dispone de un metodo **`ExecuteService`** que es publico para los clientes y permite ejecutar un servicio de negocio.

La interfaz generica de este metodo es la siguiente:

```
public     TResponse     ExecuteService<TRequest,     TResponse>(string
pServiceName, TRequest pData)
            where TRequest : IServiceContract
            where TResponse : IServiceContract, new()
```

Es un metodo que recibe como parametros el nombre del servicio mas un objeto Request.
Como notaran este objeto debe implemetar la interfaz IServiceContract y esto lo hgace automaticamente heredando de la clase Request explicada anteriormente.

Por otro lado retorna un objeto Responce que implementa tambien la interfaz IServiceContract, que de la misma manera al heredar de la clase base Response tambien implementa esta interfaz.

Ejemplo de implementacion

Mostraremos a continuación un ejemplo practico de como se ven implementados los objetos `Request` y `Response` y como son llamados desde una aplicación cliente.

Construcción de las interfases o contratos del servicio

Lo primero que debemos crear es una esquema que reprecente funcionalmente los datos que seran enviados al servicio. Supongamos un servicio que cree una factura en un sistema, lo llamaremos `CrearFacturaService`. Mas adelante detallaremos como se implementa un servicio.
Dseñamos el esquema correspondiente al Request:

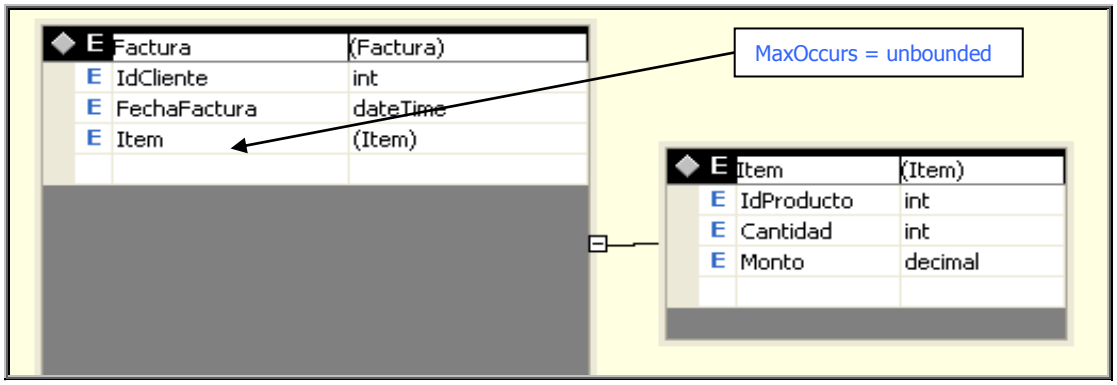


Figura 4. CrearFacturaREQ.xsd

En base a este esquema creamos la clase `FacturaBE, ItemBE y ItemBECollection` que heredan de `Entity` y corresponden a los datos funcionales de entrada al servicio. Notemos aquí que la primer clase `FacturaBE` es la principal y la que encapsula todos los datos de negocio y es la que tiene la posibilidad de hacer llamadas a las demas subclasess clases que aparecen en el esquema (en este caso solo la colección de Items).

La clase `FacturaBE` como es la cabecera de todas las demás es la que va a ser llamada desde el objeto `CrearFacturaRequest` principal atravez del atributo heredado `"BusinessData":` Ej: CrearFacturaRequest.BusinessData

Por lo tanto, una vez que tenemos diseñadas las clase de negocio, solo nos falta construir una clase que sea la que va a heredar de la clase `Request, es decir CrearFacturaRequest:`

```
public class CrearFacturaRequest : Request< FacturaBE >
{
}
```

A contunuacion mostramos como nos queda el codigo del Request completo:

```
using System;
using System.Collections.Generic;
using Action Line.Framework.Bases.BackEnd;
using System.Xml.Serialization;

namespace Promi.SistemaContable.InterfaseServices.Facturas
{
   public class CrearFacturaRequest : Request<FacturaBE >
   {
   }

   [XmlInclude(typeof(FacturaBE)), Serializable]
   public class FacturaBE : Entity
   {
      private int? _IdCliente;
      private DateTime? _FechaFactura;
      private ItemBECollection _ItemBECollection = new ItemBECollection();

      public int? IdCliente
      {
         get { return _IdCliente; }
         set { _IdCliente = value; }
      }
```

```
        public DateTime? FechaFactura
        {
            get { return _FechaFactura; }
            set { _FechaFactura = value; }
        }
        public ItemBECollection ItemBECollection
        {
            get { return _ItemBECollection; }
            set { _ItemBECollection = value; }
        }
    }

    public class ItemBECollection : Entities<ItemBE>{}

    [XmlInclude(typeof(ItemBE)), Serializable]
    public class ItemBE : Entity
    {
        private int? mi_IdProducto;
        private int? mi_Cantidad;

        public int? IdProducto
        {
            get { return mi_IdProducto; }
            set { mi_IdProducto = value; }
        }

        public int? Cantidad
        {
            get { return mi_Cantidad; }
            set { mi_Cantidad = value; }
        }

    }
}
```

Esta clase `CrearFacturaRequest` es la clase que realmente recibe el Dispatcher y contiene tanto información técnica o de contexto como información funcional o de negocio. Gráficamente a un objeto `Request` lo podemos ver de la siguiente manera:

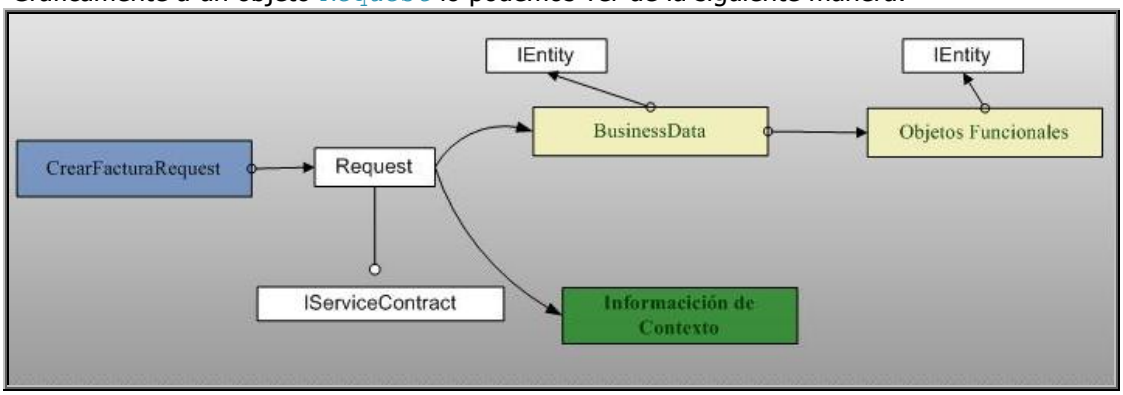


Figura 5. Modelo Request de un Servicio

Interfaz de respuesta

De manera similar necesitamos construir una interfaz que represente el resultado del servicio, es decir, el Response.
Entonces construimos el esquema XSD que lo represente, como se muestra a continuación:

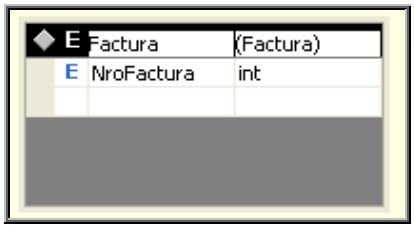


Figura 6: CrearFacturaRES.xsd

Luego desarrollamos el objeto de negocio, `FacturaBE`, que representa al esquema de arriba viéndose como se muestra en sniped debajo.

```
using System;
using System.Collections.Generic;
using Action Line.Framework.Bases.BackEnd;
using System.Xml.Serialization;

namespace Promi.SitemaContable.InterfaseServices.Facturas
{
   /// <summary>
   /// Request CrearFacturaResponse .-
   /// </summary>
   public class CrearFacturaResponse : Response< FacturaBE >
   {
   }

   public class FacturaBE : Entity
   {
      #region [Private Members]
      private int? mi_NroFactura;

      #endregion
      #region [NroFactura]
      [XmlElement(ElementName = "NroFactura", DataType = "int")]
      public int? NroFactura
      {
         get { return mi_NroFactura; }
         set { mi_NroFactura = value; }
      }
      #endregion

    }
```

Por ultimo solo nos resta construir la clase CrearFacturaResponse que la agregamos en el mismo namespace donde agrego la clase FacturaBE. Cuandodo creamos esta clase lo que estamos haciendo vemos que le pasamos como parametro generico, a la clase base Response, un tipo de dato definido por nosotros que es FacturaBE y esta misma clase va a ser la sera el contenido del **BusinessData** del Response.

Entonces cuando se quiera acceder al valor de negocio retornado por el servicio se lo haga de la siguiente manera:

```
CrearFacturaResponse wResponse = null;
lblFactura.Text = "Factura N°: " + wResponse.BusinessData.NroFactura;
```

Tipo genérico funcional recibido por un Request o Response (IEntity)

Como habran notado las dos clases tanto `Request` como `Response` recientemente construidas son definidas como una clase que hereda de una clase base que recibe como parámetro genérico cualquier clase que implemente `IEntity`. Es decir cualquier clase que sea de tipo `Entity` (clase escalar) o `Entities` (clase vectorial de la clase `Entity`). Estas clases bases están definidas en el Framework de Action Line la construcción de objetos de estos tipos serán detallados en la sección.

Por lo tanto si en nuestra clase del ejemplo `CrearFacturaRequest : Request<FacturaBE >`, que como vemos es un request que recibe un objeto de tipo factura, quisiéramos que de ahora en adelante pueda recibir una colección de la misma, simplemente bastaría con definir una clase de tipo `Entities<Entity>` llamémosla `FacturaCollectionBE` que reciba una clase `FacturaBE`. Es decir crearíamos una lista contenedora de objetos `FacturaBE`:

public class FacturaBECollection : Entities<FacturaBE>{}

Y luego adaptar nuestra clase `CrearFacturaRequest` para que reciba tal colección de modo que se vea de la siguiente manera:

```
public class CrearFacturaRequest : Request< FacturaBECollection >
{
}
```

Esto es posible dado que la firma de las clases Requests o Responses base es:

```
:Request<T> where T : IEntity
:Response<T> where T : IEntity

Y
Entity : IEntity
Entities : IEntity
```

Utilización de las interfaces del lado del Cliente:

Tanto los Requests como los Responses creados en la sección anterior son utilizados en el Back-End por los servicios y en el Front-End por los mismos clientes para llamar los servicios o bien para utilizar sus resultados.

La utilización de estas clases es muy sencilla, simplemente basta con crear una instancia de la clase `Request` y rellenarla con la información necesaria de acuerdo las reglas de negocios exigidas y luego pasársela al método del **Wrapper** ***ExecuteService (..).***

La mejor forma de entender esto es a través de un ejemplo:

Supongamos un método dentro de nuestro formulario de Windows que rellene el objeto `CrearFacturaRequest` apartir un control `DataGridView.` Entonces para encapsular la complejidad dejamos que toda esta tarea sea llevada a acabo en un método separado, lo llamaremos `BuildRequest`.

Nota: Con respecto a al informacion de contexto, no es necesario que el desarrollador se esfuerce en establecer sus valores, se debe dejar este trabajo para que sea realizado por la logica del wrapper.

```
private CrearFacturaRequest BuildRequest()
                {
        CrearFacturaRequest wRequest = new CrearFacturaRequest();


        wRequest.BusinessData.IdCliente = (int) txtNumeroCliente.Text;
        wRequest.BusinessData.FechaFactura = cmbFecha.Value;
        foreach (DataGridViewRow wRow in grdDetalle.Rows)
                        {
            ItemBE wItemBE = new ItemBE();

            wItemBE.Cantidad = Convert.ToInt32(wRow.Cells["Cantidad"].Value);
            wItemBE.IdProducto = Convert.ToInt32(wRow.Cells["IdArticulo"].Value);

            wRequest.BusinessData.ItemBECollection.Add(wItemBE);
                        }

                        return wRequest;
                }
```

Luego, una vez que llenamos el Request con la información necesaria, lo que nos resta es llamar al servicio pasándole este y esperar el resultado.
Para esto solo tenemos que enviar esta información al dispatcher y esto lo hacemos a través del **Wrapper** que esta embebido en la clase base (FrmBase) que heredan los WinForms.

```
private void btnCrear_Click(object sender, EventArgs e)
{
CrearFacturaResponse wResponse= null;

try
  {
wRequest = BuildRequest();
wResponse =
base.ExecuteService<CrearFacturaRequest,CrearFacturaResponse>("CrearFacturaService",
wRequest);
MessageBox.Show("Se creó la Factura N°: " + wResponse.Result.NroFactura);
}
catch (Exception ex)
 {
    MessageBox.Show(ExceptionHelper.ProcessException(ex).Message);
}
                                        }
```

Nomenclaturas

A fin de unificar un poco los criterios para el nombrado de servicios e interfases utilizaremos esta tabla con los patrones de nombres:

Objeto	Patrón
Servicios	[Nombre Servicio] + Service → [Class]

Request	[Nombre Servicio] + Request → [Class]
Response	[Nombre Servicio] + Response → [Class]
Esquema para modelar el objeto de negocio de un Request	[Nombre Servicio] + REQ.xsd → [File]
Esquema para modelar el objeto de negocio de un Response	[Nombre Servicio] + RES.xsd→ [File]

SERVICIOS

Servicios Sincrónicos

Los servicios de negocio sincrónicos (transaccionales o de consulta) representan a *servicios* simples del Framework.

Son clases .NET que heredan de una clase base abstracta llamada `BusinessService`, provista por el Framework de Action Line, dicha clase base tiene un método que se debe implementar denominado `Execute` y exige que se pace un `Request` y retorne un `Response`.

```
public abstract TResponse Execute(TRequest pServiceRequest);
```

La clase base `BusinessService` Es la clase de la que deben heredar todas aquellas clases que sean implementaciones de servicios de negocio.

El Dispatcher recibe un objeto de tipo `Request` especializado (Ej. CrearClienteRequest) mas el nombre del servicio (Ej.: "CrearCliente") . Tal cual como se muestra con el ejemplo citado en "Utilización de las interfaces del lado del Cliente" cuando se dispara el servicio de *CrearFactura*.

El `Request` contiene la información necesaria para dar de alta un cliente más información de contexto del sistema que requiere el Dispatcher para identificar el servicio a ejecutar. El Dispatcher localiza la información necesaria para identificar el servicio a trabes de una meta data que puede ser un XML de configuración o una base de datos de SQL..
Una vez que el Dispatcher identifica el servicio a ejecutar, instancia la clase, visualizada anteriormente, y ejecuta el método *Execute*, pasando como parámetro un objeto de tipo `Request` especializado.

Una vez que el servicio ejecuta su propia lógica retorna un objeto de tipo Response especializado (Ej.: CrearClienteResponse).

Los servicios sincronos simples pueden ser transaccionales o de consulta, a continuación describiremos más detalladamente ambos y se mostraran ejemplos con pseudo código para su mayor comprensión.

Servicios de consulta (SC)

Los servicios de consulta solo recuperan datos del sistema de información y/o de sistemas externos. Estos interactuarán con uno o más BCs que accederán a los datos a través de los componentes de acceso a datos.

Los servicios de consulta retornarán DataSets no tipificados, dichos DataSets podrán ser retornados por BCs. Los datos contenidos en el DataSets podrán sufrir transformaciones, si fueran necesarias, realizadas en los o DACs, BCs.

A continuación se visualiza un pseudo ejemplo de un servicio sincrónico de consulta que utiliza directamente un BC para obtener los datos.

```
using System;
using Framework.Core;
using Promi.SitemaContable.Cliente.Backend.BusinessComponents
using Promi.SitemaContable.Cliente.InterfaseServices;

namespace Promi.SitemaContable.Cliente.Backend.BusinessServices
{
   public class ConsultarClientesPorFiltroService : BusinessService<
  ConsultarClientesPorFiltroRequest,        ConsultarClientesPorFiltroResponse>
  {
        ConsultarClientesPorFiltroResponse
  Execute(ConsultarClientesPorFiltroRequest req )
        {
              ConsultarClientesPorFiltroResponse  res = new
  ConsultarClientesPorFiltroResponse  ();

            ClienteBC wClienteBC = new ClienteBC();
             DataSet  wDts =
  wClienteBC.ConsultarClientesPorFiltro(req.Param.Nombre,req. Param.Apellido);

              this.MappingResult (wDts, out res )

              return res;
        }

  }
}
```

Conclusiones:

- Son el punto de entrada para cualquier consulta que se quiera realizar sobre el sistema de información.
- Los servicios de consulta utilizarán BCs directamente y/o SSCs, ambos retornan directamente DataSets, DataTables o entidades de negocios que heredan de las clases bases de framework.
- Informar, a través del sistema de excepciones del framework (excepciones técnicas o funcionales), cualquier error que pudo haber surgido. El detalle de cómo implementar este mecanismo de excepciones se explica en otro capítulo de este documento.
- Transformar los datos del Request a los datos requeridos por el BC o SSC, y de agregar el resultado retornado por los BCs y/o SSCs al Response.
- La interfaz de entrada al SC es un request del Framework de Promiente. La interfaz de salida del SC son responses del Framewok.
- Los tipos de SC identificados son:
  - Servicio de pre-llenado de formularios
  - Servicio de búsqueda.
  - Servicio de Validación.

Servicios transaccionales

Los servicios transaccionales son aquellos que actualizan o mantienen el Entorno. También interactúan con uno o más BCs para ejecutar la transacción.

A continuación se visualiza la estructura básica de un servicio sincrónico que crea un cliente en el sistema.

```
using System;
using System.Data;
using Action Line.Framework.Bases.BackEnd;
using Promi.SitemaContable.Cliente.BackEnd.BusinessEntities;
using Promi.SitemaContable.Cliente.BackEnd.BusinessComponents;
using Promi.SitemaContable.Cliente.InterfaseServices;

namespace Promi.SitemaContable.Cliente.Backend.BusinessServices
{
 public class CrearClienteService : BusinessService<CrearClienteRequest,
CrearClienteResponse>
        {
                CrearClienteServiceResponse Execute(CrearClienteServiceRequest req)
                {
                        CrearClienteResponse res = new CrearClienteResponse ();

                        // Ejecutar lógica del servicio.

                        return res;
                }
  }
                                                }
```

Conclusiones:

- Los servicios transaccionales son utilizados cuando se requiere actualizar / mantener el sistema de información / sistemas externos involucrados.
- Los servicios transaccionales utilizan uno o más BCs para resolver su lógica.
- Son encargados de transformar / relacionar los datos del Request a los datos requeridos por los BCs, y de transformar / relacionar el resultado retornado por los BCs al Response.
- Informar, a través del sistema de excepciones del framework (técnicas o funcionales), cualquier error que puede haber surgido.
- La interfaz de entrada y salida de los STs son request y response especializados.
- La interfaz de entrada para interactuar con BCs son id's y/o BEs simples o colecciones.

Servicios Asíncronos

Los servicios asincrónicos están preparados para ejecutar varios pasos en secuencia sin que se consuman los recursos que administra el del framework del Dispatcher. Cuando un servicio se está ejecutando y realiza una operación que puede tomar un largo tiempo, se consumen recursos innecesariamente ya que el servicio está mucho tiempo esperando que otros componentes externos terminen de realizar sus tareas para retornar el valor final al Front-End.

Utilizando el sistema de conectores, el Dispatcher libera los recursos consumidos por un servicio, mientras el conector se encarga de ejecutar la transacción con un sistema externo y de esperar que la transacción finalice. Cuando esto ocurre, el conector comunica al *Dispatcher* para que continúe con la ejecución del servicio con los resultados obtenidos a través del conector. De esta forma mientras el conector realiza su tarea el Dispatcher puede

atender otros servicios ya que tiene recursos libres para ello. De esta forma logramos mejorar el nivel de escalabilidad del sistema.

Los servicios asincrónicos interactúan con el sistema de conectores y a diferencia de los servicios sincrónicos, están constituidos por dos métodos, Execute y ExecuteStep. Los servicios asincrónicos tienen un método llamado Execute donde comienza la ejecución y el servicio prepara la llamada al sistema externo.

Cuando el conector finaliza la ejecución, devuelve los resultados al *Dispatcher*, que procede a ejecutar el paso siguiente llamando al método ExecuteStep. De esta forma el servicio conoce que se trata de la continuación de la ejecución a uno o varios conectores.
Los servicios asincrónicos heredan de la clase base BusinessAsyncService, a diferencia de los sincrónicos que heredan de BusinessService.

Conclusiones:

- Los servicios asincrónicos se utilizan solo en escenarios en los cuales se requiere ejecutar consultas o transacciones en-línea contra sistemas externos o para ejecutar servicios de consulta o transaccionales cuyo tiempo de respuesta es elevado. En estos casos existe mucho tiempo de I/O dado que el procesamiento de la transacción no consume recursos del Dispatcher, sino del sistema en el cual se ejecuta dicha transacción o consulta.

- Dado que un servicio asincrónico puede involucrar uno o más pasos, es posible que se ejecuten consultas o transacciones contra sistemas externos y, en el mismo servicio, consultar o ejecutar una actualización al sistema propio.

- Para utilizar servicios asincrónicos es necesario desarrollar previamente los conectores a los sistemas externos necesarios.

Conectors

Uno de los objetivos del sistema de conectores externos es unificar la interfaz de programación para los servicios que acceden a sistemas externos. A efectos de cumplimentar dicho objetivo, se ha definido un mecanismo por el cual los desarrolladores de servicios puedan interactuar con otros sistemas sin la necesidad de aprender una nueva interfaz de programación para interactuar con cada uno de ellos.

De esta forma los servicios pueden solicitar la ejecución de una transacción o consulta a un sistema externo utilizando siempre una misma interfaz. Este soporte de conectores es también extensible de manera tal que los usuarios pueden crear nuevos conectores que interactúen con sistemas externos propios.

El sistema de conectores está pensado para que se puedan crear nuevos conectores cuando sean necesarios. Los conectores son clases .NET que implementan la interfaz IServiceConnector que se define en el Framework de Action Line.

Conclusiones:

- Los conectores son necesarios para ejecutar transacciones *en-línea* o consultas a través de servicios asincrónicos.

- Los conectores deben ser piezas de código eficientes que conocen con mucho nivel de detalle los sistemas externos con los cuales se encuentran interactuando.

- Dado que los conectores son utilizados en el sistema *en-línea*, deben estar orientados a minimizar el tiempo de respuesta. Ejemplo, si sabemos que la conexión a SAP toma bastante tiempo, es responsabilidad del conector de manejar un pool de conexiones.

- La transaccionalidad cuando se interactúa con sistemas externos a través de conectores esta dada por mecanismos de compensación / reversas.

- El subsistema de conectores define una interfaz programática común, para ser utilizada en los servicios de negocio en los escenarios en los cuales se requiera acceso a sistemas externos o tecnologías / rutinas con tiempos altos de respuesta.

- La comunicación con el sistema de conectores internos es a través de las DACs. Es decir que si los componentes BCs necesitan hacer una llamada a un servicio de un sistema externo utilizarán algún componente DAC que exponga la interfaz que necesite el conector a dicho sistema.

Componentes de Negocio (BC)

Los componentes de negocio contienen la lógica de negocio del sistema de información. Dichos componentes son los únicos encargados de actualizar o consultar la base de datos a través de su Componente de Acceso a Datos (DAC).

Los componentes de negocio no contienen datos, solo comportamiento. Los datos los obtienen por parámetro a través de valores escalares (representativos del negocio, por ejemplo ClienteID) o Entidades.

Los componentes de negocio no pueden interactuar con otros componentes de negocio. Por ejemplo, el componente de negocio ClienteBC, no puede interactuar con el componente de negocio PersonaBC a efectos de registrar una persona y luego registrar un cliente. Para tal propósito se deberá diseñar una componente de negocio de más alto nivel que sea orquestadora de las demás componentes.

A continuación se muestran dos ejemplos:

- El primero permite registrar un cliente, recibiendo una entidad de negocio como parámetro.
- El segundo permite retornar una colección de clientes.

Ejemplo 1

```
// Componente de negocio que simula
// crear un cliente.
namespace Promi.SitemaContable.Cliente.BackEnd.BusinessComponents
   {
        public class ClienteBC : BaseBC
    {
                public bool AgregarCliente(ClienteBE pClienteBE, TelefonosClienteBE
pTelefonosClienteBE )
                {
                        ClienteDAC wClienteDAC = new ClienteDAC();

                        int wIdCliente  = wClienteDAC.AgregarCliente(pCliente);

                 TelefoniaDAC.CrearTelefonosCliente(wIdCliente  , pTelefonosClienteBE);


                        return wIdCliente  ;
                         }
     }
 }

// Componente que crea un cliente en
// la base de datos.
```

```
namespace Promi.SitemaContable.Cliente.BackEnd.DataAccessComponents
{
                        public class ClienteDAC : BaseDAC
        {
                public bool AgregarCliente(ClienteBE pCliente)
                {
                                        int retVal = SqlHelper.ExecuteNonQuery(
                                                conn,
                                                CommandType.StoredProcedure,
                                                "[Cliente_i]",
                                                new SqlParameter[]
{new SqlParameter( "@ Name ", pCliente.Name)});
                        return true;
                }
        }
}

Ejemplo Nro. 2
// Componente que consulta los clientes
// de un vendedor en la base de datos.
namespace Promi.SitemaContable.Vendedor.BackEnd.BusinessComponents
{
                        public class VendedorBC : BaseBC
        {
                                public DataSet ConsultarClientes(int vendedorID)
                {
                                         VendedorDAC wVendedorDAC = new VendedorDAC();
                                        ClienteSBE wClienteSBE = null;

                   DataSet  wDtsCli  =VendedorDAC.ObtenerClientes(wVendedorID,
wClientes);

 wClienteSBE  = Helpér.MappingCliente(wDtsCli  )
                }
        }
}

// Componente que obtiene los clientes
// de un vendedor desde la base de datos.
namespace Promi.SitemaContable.Vendedor.BackEnd.DataAccessComponents
{
                                public class VendedorDAC : BaseDAC
        {
   public DataSet ObtenerClientes(int vendedorID, ClienteSBE clientes)
                {

          Database wDataBase = null;
                        DbCommand wCmd = null;

                        try
                        {
                                wDataBase = DatabaseFactory.CreateDatabase();
                                wCmd = wDataBase.GetStoredProcCommand("Articulos s");

                                return wDataBase.ExecuteDataset("Articulos_s");
                        }

                        finally
                        {
                                wCmd.Dispose();
                                wCmd = null;
                                wDataBase = null;
                        }

                }
         }
}
```

Conclusiones:

No será posible tener referencias circulares entre componentes de negocio.

Una manera de evitar las referencias cruzadas es crear un BC a un nivel más alto como intermediario. Ejemplo, si en el componente de negocio de clientes necesito utilizar lógica del componente de negocio de direcciones y viceversa, se utilizará este nuevo BC que ejecuta al componente de negocio de direcciones y luego al de cliente.

- Los componentes de negocio no acceden a los datos directamente, lo hacen a través de su componente de acceso a datos.
- Existe un componente de acceso a datos por componente de negocio.
- Los componentes de negocio pueden interactuar con mas de una entidad, es decir, no hay una relación uno a uno entre las entidades de negocio y los componentes de negocio.
- En el caso de que la cantidad de componentes de negocio sea muy extensa, se debe fragmentar el assembly. Esto mejora el trabajo en equipo y disminuye el impacto en la implementación de un assembly.

- Los componentes de negocio se definen de acuerdo a la funcionalidad que realizan. A continuación se definen las reglas para decidir cuando crear un componente de negocio en Entorno:

- Existe un único componente de negocio por cada entidad fuerte y sus relaciones débiles o relaciones de muchos a muchos. Ejemplos:

    1. Existe la relación Clientes tiene Perfiles. Clientes y Perfiles son entidades fuertes, y existe una entidad intermedia que registra la relación. En este caso existe un único componente de negocio llamado Cliente administra sus perfiles.

    2. Existe la relación Empleado tiene Hijos. Hijo es una entidad débil mientras que Empleado es una entidad fuerte. En este caso existe un único componente de negocio llamado Empleado que administra la entidad débil.

- En el caso del patrón Tipo – Supertipo, se debe definir un componente de negocio para el tipo y uno para el subtipo. El subtipo conoce información de su tipo. Ejemplos:

    1. En una relación Persona -> PersonaNatural y Persona -> PersonaJuridica. En este caso existe un componente de negocio para Persona, otro para PersonaNatural (que conoce los datos de Persona) y otro para PersonaJuridica (que también conoce los datos de Persona).

- Los componentes de negocio no deben requerir identidad de usuario para su ejecución, la identidad del usuario (autorización) será requerida por el Dispatcher, antes de ejecutar cualquier tipo de servicio.

- La interfaz de entrada para los métodos de actualización de los BCs son id's y/o entidades (simples o colecciones).

- La interfaz de salida de los BCs pueden ser:

    1. DataSets no tipificados u objetos que implementen IDataReader.

    2. Un único escalar, por ejemplo un saldo dado un id de cuenta.

    3. Una o mas entidades simple o colección, por ejemplo:

```
void LeerClienteConDirecciones(int pIdCliente, out ClienteBE pCliente, out[2]
DireccionSBE pDirecciones)
```

- El nombre de los métodos de los BCs, se definen utilizando la siguiente nomenclatura/patrón:

[Operación][Entidad][Aspecto]

Ejemplos:

- CrearClienteConDirección, donde Crear es la operación, Cliente es la entidad y ConDirección es el aspecto.
- ActualizarClienteDatosBásicos, donde Actualizar es la operación, Cliente es la entidad y DatosBásicos el aspecto.

Componentes de Soporte al Negocio (BSC)

Los componentes de soporte al negocio contienen funciones que son utilizadas por más de un componente de negocio. No son simples utilitarios, sino que están ligados a las problemáticas del negocio, pero su funcionalidad brinda servicios que son comunes para mas de un componente de negocio.

Dado que los componentes de soporte no tienen ninguna particularidad con respecto al modelo, son solo componentes con complejidad técnica, no se expondrá ningún ejemplo al respecto.

Conclusiones:

- Si bien los componentes de soporte al negocio no son más que lógica reutilizable agrupada en piezas de software, el hecho de conceptualizar a este tipo de componentes permite mejorar el entendimiento de las partes que conforman al modelo.

- Para los componentes de soporte al negocio aplican las mismas conclusiones que para los componentes de negocio, con las siguientes variaciones:

  - Podrán acceder a la base de datos a través de su DAC, para obtener metadata propia de su función.
  - Solo pueden ser utilizados por componentes de negocio.
  - La interfaz de entrada de los BSCs son valores escalares y/o vectoriales.
  - La interfaz de salida de los BSCs son escalares, vectoriales (DataSets), o entidades pasadas como referencia.

Componentes de Acceso a Datos (DAC) y Table Data Gategay (TDG)

Los componentes de acceso a datos son el único punto de entrada a la base de datos. Por otro lado, son los que conocen la lógica de datos y como optimizar el acceso y la actualización de los mismos.

Los DAC permiten desacoplar las problemáticas de acceso a datos de la lógica de negocio.

TableDataGateway*:* Opcionalmente y teniendo en cuenta la complejidad del proyecto o la situación a desarrollar se puede incluir en la DAC una capa de mas bajo nivel llamada TDG.

[2] ***Nota****: el indicador out en C# para entidades desarrolladas por los programadores o el generador de código no es necesario debido siempre la entidad pasa por referencia. Solo se pone este indicador a fin de dejar mas claro el código e indicar que el método modificara totalmente o en parte la clase pasada por parámetro. Es decir luego de llamar al método anterior es altamente probable de que la clase ya no sea la misma que antes de llamada.*

De esta forma la DAC solo tendría la inteligencia orquestadora de llamadas a procedimientos ubicadas en las TDG y los procedimientos de las TDG son los que en fin tienen todo el conocimiento de acceso a datos.

Por ejemplo podemos agregar una TDG en el proyecto DAC cuando la DAC es la encargada en algunos casos de hacerle cierta transformación a los datos que retornan los SPs a través de la TDG que agreguemos:

```
using System;
using System.Data;
using System.Data.Common;
using Microsoft.Practices.EnterpriseLibrary.Data;
using Action Line.Framework.Bases.BackEnd;
using Promi.SitemaContable.Facturacion.BackEnd.BusinessEntities;

namespace Promi.SitemaContable.Facturacion.BackEnd.DataAccessComponents
{
        /// <summary>
        /// Data access component para Factura.
        /// </summary>
        /// <Date>2006-04-19T14:11:34</Date>
        /// <Author>moviedo</Author>
                        public class FacturaDAC : DataAccessComponent
                                                {
          /// <summary>
               /// Crear una factura y sus detalles ParaClienteEspecial.-
               /// </summary>
               /// <param name="pFacturaBE">FacturaBE</param>
               /// <returns>void</returns>
               /// <Date>2007-04-19T14:11:34</Date>
               /// <Author>moviedo</Author>
               public void CrearFacturaParaClienteEspecial (FacturaBE pFacturaBE)
               {
                        FacturaTDG wFacturaTDG = null;

                        DetalleFacturaTDG wDetalleFacturaTDG = null;


                        try
                        {
                                wFacturaTDG = new FacturaTDG();
                                wDetalleFacturaTDG = new DetalleFacturaTDG();

                                wFacturaTDG.Insert( pFacturaBE);

                                foreach (DetalleFacturaBE wDetalleFacturaBE in
pFacturaBE.DetalleFacturaList)
                                {
                                        wDetalleFacturaBE.Numero = pFacturaBE.Numero;

                                        wDetalleFacturaTDG.Insert(wDetalleFacturaBE);

                                }

                        }
                        finally
                        {
                                wFacturaTDG = null;
                                wDetalleFacturaTDG = null;

                        }
                                                }
```

Como habrán notado en ningún momento se paso como parámetro una entidad Fatura y otra DetalleLista. Esto es así por que las entidades no siempre representan exactamente una tabla del dominio de BD.

En ocasiones es necesario pasar como parámetros ciertos aspectos de negocio que surgen de una simple serializacion de una entidad Request que envía el Servicio a los BC. Este tema es

muy cuestionable y dependerá de un buen análisis de la situación por parte del equipo de desarrollo.

Continuando con el ejemplo lo que necesitamos ahora es crear dos clases que sean especializadas en acceso a datos para que cumplan con el objetivo de crear la cabecera de la factura por un lado y los detalles por el otro.

```
using System;
using System.Data;
using System.Data.Common;
using Microsoft.Practices.EnterpriseLibrary.Data;
using Action Line.Framework.Bases.BackEnd;
using Promi.SitemaContable.Facturacion.BackEnd.BusinessEntities;

namespace Promi.SitemaContable.Facturacion.BackEnd.DataAccessComponents
{
        /// <summary>
        /// TableDataGateway para Factura.
        /// </summary>
        /// <Date>2006-04-19T14:11:34</Date>
        /// <Author>moviedo</Author>
        internal class FacturaTDG : TableDataGateway
        {
                /// <summary>
                /// Crear cabecera Factura cliente especial
                /// </summary>
                /// <param name="pFacturaBE">FacturaBE</param>
                /// <returns>void</returns>
                /// <Date>2006-04-19T14:11:34</Date>
                /// <Author> moviedo </Author>
                public void Crear(FacturaBE pFacturaBE)
                {
                        Database wDataBase = null;
                        DbCommand wCmd = null;

                        try
                        {
                                wDataBase = DatabaseFactory.CreateDatabase();
                                wCmd = wDataBase.GetStoredProcCommand("Factura i");

                                /// Fecha
wDataBase.AddInParameter(wCmd, "Fecha", System.Data.DbType.DateTime,
pFacturaBE.Fecha);
                                /// Numero
                                wDataBase.AddOutParameter(wCmd, "Numero",
System.Data.DbType.Int32, 4);
                                /// Monto
wDataBase.AddInParameter(wCmd, "Monto", System.Data.DbType.Double, pFacturaBE.Monto);
                                /// Cliente
wDataBase.AddInParameter(wCmd, "Cliente", System.Data.DbType.String, pFacturaBE.
Cliente);

                                wDataBase.ExecuteNonQuery(wCmd);
                                pFacturaBE.Numero = (System.Int32)
wDataBase.GetParameterValue(wCmd, "Numero");
                        }
                        finally
                        {
                                wCmd.Dispose();
                                wCmd = null;
                                wDataBase = null;
                        }
                }

   }
                                                }
```

y por otro lado la TDG para los detalles:

```
using System;
using System.Data;
using System.Data.Common;
```

```
using Microsoft.Practices.EnterpriseLibrary.Data;
using Action Line.Framework.Bases.BackEnd;
using Promi.SitemaContable.Facturacion.BackEnd.BusinessEntities;

namespace Promi.SitemaContable.Facturacion.BackEnd.DataAccessComponents
{
        /// <summary>
        /// TableDataGateway para DetalleFactura.
        /// </summary>
        /// <Date>2006-04-19T14:11:34</Date>
        /// <Author>moviedo</Author>
        internal class DetalleFacturaTDG : TableDataGateway
        {
                /// <summary>
                /// Crear
                /// </summary>
                /// <param name="pDetalleFacturaBE">DetalleFacturaBE</param>
                /// <returns>void</returns>
                /// <Date>2006-04-19T14:11:34</Date>
                /// <Author>moviedo</Author>
                public void Insert(DetalleFacturaBE pDetalleFacturaBE)
                {
                        Database wDataBase = null;
                        DbCommand wCmd = null;

                        try
                        {
                                wDataBase = DatabaseFactory.CreateDatabase();
                                wCmd =
wDataBase.GetStoredProcCommand("DetalleFactura i");

                                /// Numero
                                wDataBase.AddInParameter(wCmd, "Numero",
System.Data.DbType.Int32, pDetalleFacturaBE.Numero);
                                /// IdArticulo
        wDataBase.AddInParameter(wCmd, "IdArticulo", System.Data.DbType.Int32,
pDetalleFacturaBE.IdArticulo);
                                /// Cantidad
        wDataBase.AddInParameter(wCmd, "Cantidad", System.Data.DbType.Int32,
pDetalleFacturaBE.Cantidad);

                                wDataBase.ExecuteNonQuery(wCmd);

                        }
                        finally
                        {
                                wCmd.Dispose();
                                wCmd = null;
                                wDataBase = null;
                        }
                }

        }
                                                }
```

Nota: Las TDG no son obligatorias y serán desarrollada en caso de creerse conveniente por el equipo de desarrollo junto su líder técnico.

Conclusiones:

- Los componentes de acceso a datos exponen métodos para actualizar y consultar datos en el sistema de información. Proveen funciones que permiten ejecutar consultas dinámicas. Brindan soporte al sistema de paginación (en el caso de consultas que recuperen gran volumen de registros).

- Los componentes de acceso a datos pueden interactuar con una o más tablas del modelo de datos. Por ejemplo, el componente que graba una factura, graba tanto sus datos fijos como secciones simples y secciones múltiples.

- Es muy importante, que en los SPs ejecutados por los componentes de acceso a datos, se utilice el adecuado nivel de aislamiento (ISOLATION LEVEL). Dado que los bloqueos de datos de una tabla son una de las mayores causas de contención en la DB, es necesario bloquear en forma exclusiva sólo los datos que se requiera por problemáticas de negocio.

- Analizar la posibilidad de ejecutar consultas utilizando el nivel de aislamiento READ UNCOMMITTED, siempre que la problemática de negocio lo permita.

- Evitar los niveles SERIALIZABLE y REPEATABLE READ.

- Cada vez que se necesite bloquear exclusivamente un registro (XLOCK), se deberá analizar cuidadosamente el impacto, inclusive se deberá analizar si no existe otro mecanismo de implementación. En un entorno con mucha concurrencia sobre los mismos registros, el XLOCK genera mucha contención y provoca tiempos de respuestas altos.

- Desde un componente de acceso a datos no se puede invocar a otro componente de acceso a datos.

- Los componentes de acceso a datos son los encargados de encriptar / desencriptar los datos en los casos que fuera necesario.

- Los componentes de acceso a datos deben ser desarrollados de forma homogénea en todo el sistema de información. Es decir, mas allá de que interactúen con conjuntos de datos diferentes, el comportamiento de un componente de acceso a datos debe ser similar en todos los casos:

    - Ejecutan Stored Procedures en la DB. La única excepción es la ejecución de selects construidos dinámicamente para resolver servicios de búsqueda.

      Ej.: Una consulta avanzada que basada en templates de consultas según la particularidad de la búsqueda.

      Ej. concreto: Hay situaciones donde se obtienen listado de ciertas tablas pero de acuerdo a lo que el usuario selecciona desde distintos ComboBox con un patrón de filtro en cascada. Tal combinación puede que requiera incluir además condiciones distintas en la cláusula Where, distintas Inner Join con otras tablas. Para tal problemática es posible que se decida armar template de script de SP para las distintas situaciones.

    - Deben ejecutar sentencias SQL en forma Batch a efectos de disminuir la cantidad de Roundtrips al servidor de base de datos. Los componentes de la arquitectura deben estar orientados a disminuir los roundtrips a la DB, a través del uso de esta técnica. Ahora, esto no significa que solo se pueda realizar un único roundtrip por SC o ST. El objetivo es intentar no superar los 5 o 6 roundtrips a la DB por SC o ST. Ejemplo, en el caso de tener que grabar los ítems de una factura, es recomendable que se envíe un lote de sentencias armadas con la llamada al SP correspondiente pero con diferentes parámetros.

En el siguiente ejemplo se puede ilustrar el uso de un batch para crear todos los medios de contacto de un vendedor determinado:

```
/// <summary>
/// Crea en Batch los medios de contacto de un vendedor
/// </summary>
/// <param name="pFacturaBE">FacturaBE</param>
/// <returns>void</returns>
/// <Date>2006-04-19T14:11:34</Date>
```

```
/// <Author>moviedo</Author>
internal void CrearMediosDeContacto(MedioContactoSBE pMedioContactoSBE,int
pIdVendedor)
                {
                        using (SqlConnection wCnn = new SqlConnection(mCnnString))
                        using (SqlCommand wCmd = new SqlCommand() )
                        {
                                try
                                {
                                        wCnn.Open();
                                        wCmd.CommandType = CommandType.Text;
                                        wCmd.Connection = wCnn;
                                StringBuilder BatchCommandText = new StringBuilder();

                                        BatchCommandText.Append("DECLARE @RetVal INT; ");
                                        foreach(MedioContactoBE wMedioContactoBE in
pMedioContactoSBE)
                                        {
                                                BatchCommandText.Append("EXEC
MedioContacto_i ");

                                                /// IdTipoMedioContacto.
                                                BatchCommandText.Append("@pIdVendedor = "
);
                                                if
(wMedioContactoBE.IsIdTipoMedioContacto == null)
                                                {
                                                        BatchCommandText.Append("NULL");
                                                }
                                                else
{
BatchCommandText.Append(wMedioContactoBE.IdTipoMedioContacto);
                                                }
                                                BatchCommandText.Append( ", ");

                                                /// IdRelacionCategoria.

        BatchCommandText.Append("@IdRelacionCategoria = " );

        BatchCommandText.Append(pIdRelacionCategoria);
                                                BatchCommandText.Append( ", ");

                                                /// CodigoPais.
                                                BatchCommandText.Append("@CodigoPais = "
);
                                                if (wMedioContactoBE.IsCodigoPais ==
null)
                                                {
                                                        BatchCommandText.Append("NULL");
                                                }
                                                else
                                                {
                                                        BatchCommandText.Append("'");

        BatchCommandText.Append(wMedioContactoBE.CodigoPais);
                                                        BatchCommandText.Append("'");
                                                }
                                                BatchCommandText.Append( ", ");


                                                /// EsPrincipal.
                                                BatchCommandText.Append("@EsPrincipal = "
);
                                                if (wMedioContactoBE.IsEsPrincipalNull)
                                                {
                                                        BatchCommandText.Append("NULL");
                                                }
                                                else
                                                {
                                                        if(wMedioContactoBE.EsPrincipal ==
true)
                                                        {

        BatchCommandText.Append("1");
```

```
                                                }
                                                else
                                                {

        BatchCommandText.Append("0");

                                                }
                                        }
                                        BatchCommandText.Append( ", ");



                                        /// IdMedioContacto (OUTPUT).
                                        BatchCommandText.Append("@IdMedioContacto
= @RetVal; ");
                                }
                                if (BatchCommandText.Length > 0)
                                {
                                        wCmd.CommandText =
BatchCommandText.ToString();
                                        wCmd.ExecuteNonQuery();
                                        wCnn.Close();
                                }
                        }
                        catch (Exception ex)
                        {
                                throw ExceptionHelper.ProcessException(ex, this);
                        }
                }
        }
```

- A efectos de lograr que esto sea factible todos los componentes de la arquitectura deben tener en cuenta el manejo de datos empaquetados. Por ejemplo, si es necesario grabar cinco documentos y por cada documento se debe validar el saldo del cliente. Es conveniente que el Servicio Transaccional realice, a través de componentes de negocio, primero las cinco validaciones en forma batch y luego las cinco creaciones del documento. Grabando los datos del tipo de documento (un roundtrip), los datos de las secciones simples (en forma batch) y los datos de las carteras (en forma batch).

- Acceder a la DB con un mismo usuario (string de conexión) a efectos de poder utilizar las ventajas de Connection Pooling.

- Utilizar la Base Class SQLClient para obtener mayor eficiencia en el acceso a base de datos SQL Server.

- Liberar la conexión a la DB lo antes posible.

- La interfaz de entrada para los métodos de actualización de los DACs son id's y/o BEs simples o colecciones. En el caso de usar id's estos pueden ser escalares o colecciones de id's de un mismo tipo de entidad.

- La interfaz de entrada para los métodos de lectura de los DACs serán valores escalares. En el caso de consultas dinámicas el método del DAC contendrá todos los posibles valores y aceptará que algunos de ellos estén en blanco o nulls. Es responsabilidad del DAC el componer un string de selección adecuado a partir de estos parámetros.
  EJ: Suponga una consulta de clientes por los siguientes parámetros
  - Nombre
  - Apellido
  - EstadoCivil

La idea es desarrollar un método en DAC que pase por parámetro o bien una entidad o los escalares mismos que contengan todos los parámetros necesarios para la consulta sin importar se pueda filtrar por estos o no.

Es decir la DAC al recibir por ejemplo Nombre determinara si pasar un null o su valor a la interfaz del SP y el SP también debe estar preparado para aceptar nulos por defecto y tomar una acción distinta para cada una de las combinaciones que se puedan dar.

- La interfaz de salida de los DACs serán:
    - DataSets no tipificados o DataReaders en caso de consultas de solo avance en el cual no se necesite modificar o reordenar los resultados.-
    - Un único escalar, por ejemplo un saldo dado un id de cuenta.
    - Una BE simple o colección, obtenido como parámetro por referencia y completado por el DAC. Por ejemplo un ClienteBE. Por ejemplo:

```
void LeerClienteConDirecciones(int pIdCliente, out ClienteBE
pCliente, out DireccionSBE pDirecciones)
```

Entidades

Las entidades representan a la fachada de una o más tablas del modelo de entidad relación o por otro lado pueden estar incluidas unas peticiones Request a un web service. Es decir con una entidad podemos representar una tabla Cliente o asta el patrón Cabecera Detalle de una factura.

Dicha fachada especifica de forma fuertemente tipificada las propiedades que corresponden a los atributos de las tablas en cuestión o de los atributos combinados de distintas tablas que correspondan a un proceso de negocio representado por un Request/Result.

Las entidades son utilizadas por los componentes de negocio o por los servicios de negocio (transaccionales o de consulta). Por dicha razón se encuentra en color GRIS en contacto con estos tres módulos.

Entidades Simples

Las *entidades simples* contienen datos escalares de una o mas tablas o pueden contener datos de una colección de entidades o asta una entidad. Dichas entidades están compuestas por una serie de propiedades Get y Set que permiten el acceso a dichos datos.

Colecciones

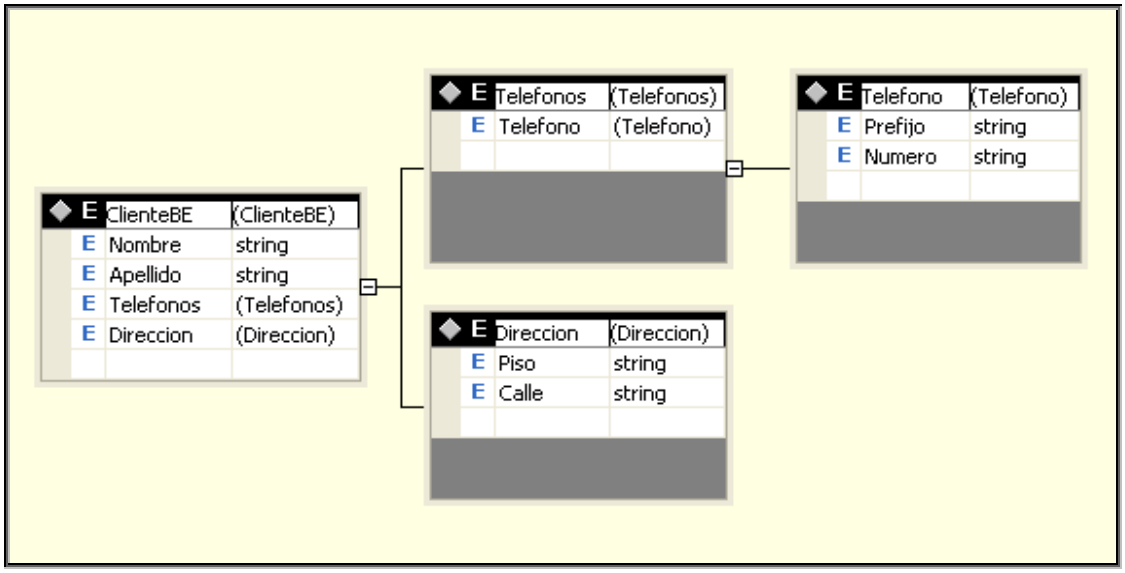
Las *entidades colecciones* son un conjunto de entidades simples. Las mismas están compuestas por una serie de métodos que me permitan realizar operaciones sobre la colección a efectos de permitir:

- Agregar una entidad de negocio simple
- Buscar una entidad de negocio
- Obtener un subconjunto de entidades de negocio
- Iterar por la colección
- Retornar xml o DataSet de la colección misma.

Ejemplo que detalla las distintas formas de relacionar colecciones con una entidad:
Podríamos tener una entidad Cliente que tenga una colección de de teléfonos como propiedad. Al mismo tiempo, si el negocio exige, la entidad de Cliente necesita almacenar la dirección, pero resulta que los clientes tienen siempre una sola dirección entonces el diseñador (un tanto limitado en este caso) decide establecer una propiedad a la clase Cliente de tipo complex type Direccion, y no utiliza una colección de direcciones. En este caso la clase cliente contiene como propiedad una entidad . (Direccion).

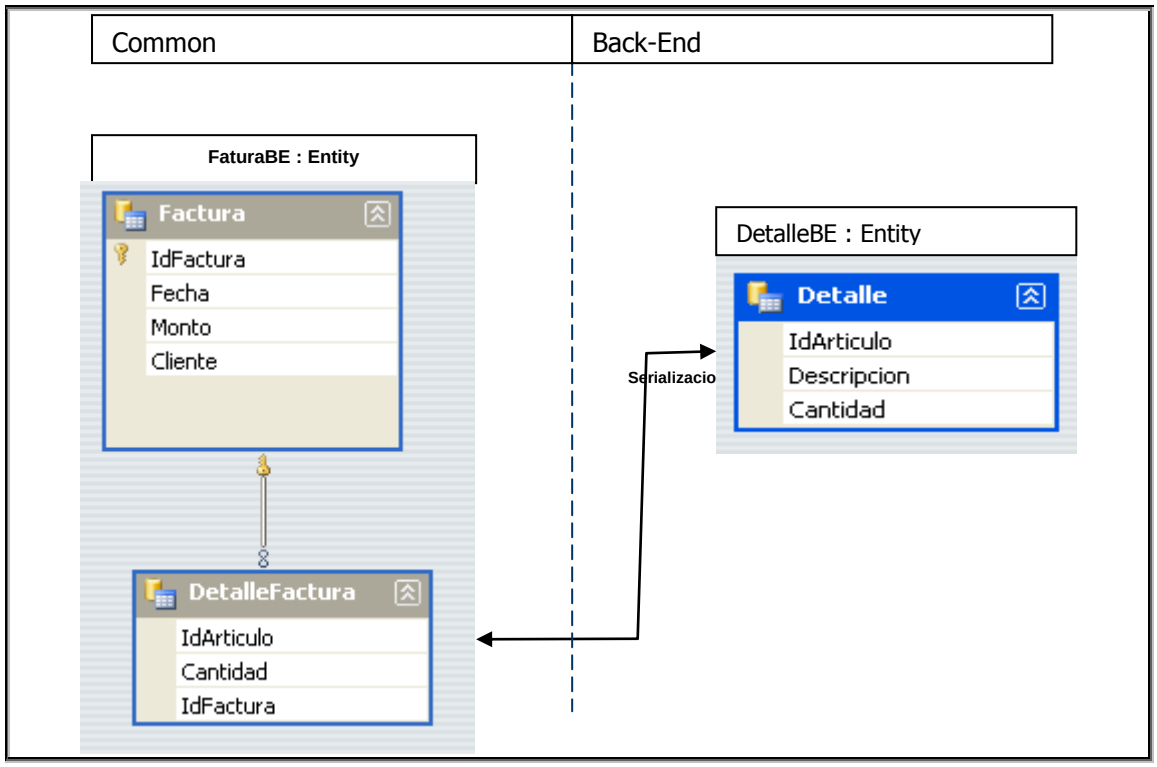
De esta forma podemos usar la clase cliente de la siguiente manera:

```
 ClienteEnt wCli = new  ClienteEnt();

        WCli.Nombre = "Marcelo";
        WCli.Nombre.Direccion.Calle = "Sarmiento";
        WCli.Nombre.Direccion.Piso = 2;
        WCli.Nombre.Telefonos[1].Numero = 1231244;

Donde:
     ClienteEnt : Entity
Dirección  : Entity
 Telefonos  : Entities
```



Serializacion

Otra de las características de las entidades es que son Serializables lo cual permite obtener un DataSet o un xml de la misma muy sencillamente. Además nos permite generar entrara para otra entidad más compleja que requiera el formato de la entidad en cuestión, como se muestra en la figura siguiente:



Entidades de relación (ER)

Las entidades de relación son colecciones utilizadas para representar relaciones de muchos a muchos, del modelo de datos. Si bien se comportan de la misma forma que las colecciones de entidades de negocio, se las define de diferente manera dado que no representan a una entidad de negocio en si misma. Es decir siempre serán utilizadas en relación con otras entidades.

EJ: ContratoComercialCondicionesComercialesBE

En esta entidad relación esta claro que relaciona dos entidades ContratoComercialBE y CondicionesComercialesBE

Por que tendríamos la necesidad de definir una ER ? Pues es necesario en algunos casos almacenar o mantener información de la relación de ambas tablas a las que se refieren las dos entidades anteriores. Podría ser que no solo se necesite los IDs de ambas tablas sino otra información que provenga del seteo de valores obtenidos por el sistema o configurados en una pantalla por el usuario.

Clases Base Entity/Entities

Todas las entidades heredan de una clase llamada Entity. Dicha clase contiene funcionalidad genérica para todos los tipos de entidades simples (un ejemplo de funcionalidad genérica puede ser GetXml() que retorna el xml de la entidad).
Todas las entidades poseen una entidad contenedora que hereda de Entities del framework de Action Line. Esta clase contenedora es la colección de la que hablamos arriba.
A continuación se visualiza un ejemplo de una entidad y su colección:

```
using System;
using System.Xml.Serialization;
using System.Collections.Generic;
```

```
using Action Line.Framework.Bases.BackEnd;
using helper = Action Line.Framework.HelperFunctions;
using System.ComponentModel;
namespace Promi.SitemaContable.Facturacion.BackEnd.BusinessEntities
{
        // Entidad de tipo colección.
[XmlRoot("ClienteCollection"), SerializableAttribute]
        public class ClienteCollection : :Entities<ClienteBE>
        {
         //Métodos que sean útiles como:
         //buscar por Nombre,  por Id etc.-
        }

// Clase base de las colecciones.
  [XmlInclude(typeof(ClienteBE)), Serializable]
    public class ClienteBE:Entity
    {

         #region [Atributes]

         private DateTime? mdt_FechaNacimiento ;
         private string msz_Nombre ;
         private string msz  Apellido;
         private int? msz  Edad;

         #endregion


         #region [Properties]

         [XmlElement(ElementName = "FechaNacimiento", DataType = "dateTime")]
         public DateTime? FechaNacimiento
         {
             get { return dmt FechaNacimiento; }
             set { mdt FechaNacimiento = value;  }
         }

         [XmlElement(ElementName = "Nombre", DataType = "string")]
         public string Nombre
         {
             get { return msz Nombre; }
             set { msz Nombre = value; }
         }

         [XmlElement(ElementName = "Apellido", DataType = "string")]
         public string Apellido
         {
             get { return msz  Apellido; }
             set { msz _Apellido = value;  }
         }

         [XmlElement(ElementName = "Edad", DataType = "int")]
         public int? Edad
         {
             get { return msz _Edad; }
             set { msz _Edad = value;  }
         }

          #endregion
  }
}
```

Conclusiones acerca de las entidades simples:

- Las entidades heredan de la clase Entity. Dicha clase encapsula funcionalidad genérica para todas sus derivadas.
- Las entidades de tipo simple solo contienen datos, todo comportamiento de negocio se encuentra en los componentes de negocio.

- Todas las entidades son serializables. Al igual que los request y response de los servicios de consulta o transaccionales.
- Todas las entidades pueden exponer sus datos a través de GetXml() o GetDataSet() .
- Solo contienen datos y comportamiento técnico, no contienen comportamientos funcionales de negocio.

Conclusiones acerca de las entidades tipo colecciones:

- Las colecciones exponen sus ítems como si fueran entidades simples tipificadas. Ejemplo, GetClienteById retorna una entidad simple de tipo Cliente.
- Son serializables.
- Debido al que retornan DataSet, las colecciones exponen métodos de búsqueda optimizados y ya desarrollados.
- Heredan de una clase base llamada Entities. Las colecciones se basan en listas bajo la siguiente definición:

  Entities<T> : List<T> where T : Entity

- Las clases bases Entities al heredar de List<T> poseen todas las ventajas de las listas genéricas.- http://msdn2.microsoft.com/en-us/library/s6hkc2c4.aspx
- Todas las colecciones deberían mantener un mismo patrón, es decir existe un conjunto de métodos comunes a todas. Ejemplo, el método GetClienteByName es un método común a todas las colecciones (solo cambia Cliente por alguna otra entidad).

```
// Entidad de tipo colección.
        public class ClienteCollection : Entities<ClienteBE>
        {
         public Cliente GetClienteByName
{
                        //Implementacion del metodo.-
}
        }
```

Sistema de espacios de nombres (Namespaces)

Los *Namespaces* son utilizados para crear un sistema destinado organizar la forma de exponer la funcionalidad de un aplicativo. Dicha funcionalidad se expresa en clases .NET.

El esquema de *Namespaces* es jerárquico, es decir permite definir un sistema de nombres comenzando por un mayor nivel de abstracción hasta llegar a niveles más específicos.

A continuación se visualiza la regla general para nombrar *Namespaces*:

<Compañía>.<[Tecnología][Sistema]>[.<Módulo>][.Diseño]

Dado que nuestros aplicativos se encuentran basados en un sistema de múltiples capas, inclusive capas físicamente bien diferenciadas, la regla que utilizaremos para definir *Namespaces* es:

<Compañía>.<[Tecnología][Sistema]>[.Capa Física][.< Módulo >][.Diseño]

A continuación citamos algunos ejemplos:

```
Action Line.Tools
Action Line.Framework.Bases.BackEnd
Action Line.SistemaFacturacion
Action Line.SistemaFacturacion.Cliente.Frontend
Action Line.SistemaFacturacion.Cliente.InterfaseServices
Action Line SistemaFacturacion.Cliente.Backend.BusinessServices
```

Como se puede visualizar en los ejemplos, de acuerdo al área de competencia de la clase, se utiliza un *namespace* de mayor o menor nivel de abstracción o detalle. Es decir, si existen clases de tipo utilitarios que pueden ser utilizadas por más de un aplicativo, se utilizará el namespace `Action Line.Tools` o `Action Line.Framework.` mientras que si el utilitario solo puede ser utilizado dentro del `Sistema de Facturacion`, el namespace a utilizar sería `SistemaFacturacion.SCA.Tools`.

En los ejemplos también podemos visualizar la existencia de dos capas físicas bien diferenciadas: Front-End y Back-End. Existe una nomenclatura que permite definir el namespaces para clases que son compartidas entre ambas capas: "`InterfaseServices`". Un ejemplo de cuando utilizar `InterfaseServices` es cuando se definen los Request del Framework de `Action Line.Framework`, se utilizan tanto en el Front-End como en el Backend.

De Front-End se desprenden los namespaces utilizados para componentes como: Páginas ASPX, UIComponents, Piezas propias del Front Controller, Etc.

De Back-End se desprenden los namespaces utilizados para representar a todos los servicios y las capas lógicas que se encuentran detrás: Componentes de Negocio, Componentes de Soporte, Entidades de Negocio, Entidades de Soporte, Etc.

A efectos de no generar confusión entre los nombres de los namespaces y las clases contenidas en dichos namespaces, se recomienda utilizar el siguiente sistema de sufijos (en la lista no están todos los tipos de componentes visualizados en la jerarquía de namespaces, solo aquellos que se utilizan para desarrollar las problemáticas del negocio):

Tipo de componente	Sufijo	Ejemplo archivo	Ejemplo Clase
BusinessServices	Service	ProductosService.cs	TraerListaProductosService

ServiceSupportComponents	SSC	NotaVentaSSC.cs	CrearNotaVentaSSC
BusinessComponents	BC	CirculacionBC.cs	CrearSuscriptorBC
SupportComponents	SC	FormuladorSC.cs	AplicarFormulaSC
DataAccessComponents	DAC	CirculacionDAC.cs	CrearSuscriptorDAC
TableDataGateway	TDG	CirculacionTDG.cs	CirculacionTDG
BusinessEntities	BE	ClienteBE.cs	ClienteBE / ClientesBE
DataEntities	DE	CuerpoSeccionesDE.cs	CuerpoSeccionDE
Request ()	Request	DatosClientesRequest.cs	DatosClientesRequest
Response ()	Response	DatosClienteResponse.cs	DatosClienteResponse
Proyectos donde se almacenan los Requests y Responses	ISVC	ModuloISVC.DLL	El contenidos son las clases: DatosClientesREQ DatosClientesRES DatosClienteResponse DatosClientesRequest Mas los esquemas correspondientes.
Proyectos donde se almacenan los BusinessServices	SVC	ModuloSVC.DLL	Contenido: TraerListaProductosService

Tabla de Contenidos