UNIVERSIDAD AUTÓNOMA METROPOLITANA

Casa abierta al tiempo

División de Ciencias Básicas e Ingeniería - Unidad Azcapotzalco.

Ingeniería en Computación.

Descripción de requerimientos de software.

Aplicación de tecnologías de cómputo móvil y sistemas de información para la mejora en el proceso de negocio dentro de un restaurante.

Miércoles 18 de Junio de 2008. Trimestre 08-I.

Integrantes:

Sergio Rubio Ugalde (203304654).

Tonatihu Díaz Alegría (203305736).

Mtra. Rafaela Blanca Silva López.

Asesora del proyecto.

Índice general

Índice de figuras

Resumen

En este proyecto, se hace un recorrido por algunos de los conceptos, principios y técnicas más recurrentes en el desarrollo de aplicaciones empresariales, así como algunas de las tecnologías que permiten acelerar la inclusión de estos conceptos en la implementación final de un sistema de información.

Estos conceptos, comprenden el almacenamiento y la recuperación de datos, la estructuración de la lógica de la aplicación, la distribución/consumo de la lógica de la aplicación y finalmente la presentación.

Casi la totalidad de los conceptos y principios utilizados para la implementación de aplicaciones empresariales, son independientes tanto de los lenguajes de programación como del entorno de ejecución; sin embargo, cada tecnología aporta puntos de vista muy diversos sobre la correcta aplicación de un mismo concepto o principio.

En este trabajo se exploran principios de programación como el principio de sustitución de Liskov, el principio de inversión de control, etc., así como los principales elementos estructurales de una aplicación empresarial, desde el punto de vista de diversas tecnologías y aplicados al desarrollo de un sistema de información real: el administrador de restaurantes Restomatic.

Parte I

Fundamentos de aplicaciones empresariales.

Capítulo 1

Servicios primarios.

Los dominios de aplicación del software son prácticamente ilimitados. Una categoría de software de aplicación que destaca, si no en importancia, al menos en cantidad de implementaciones[Pre01], es la categoría de las aplicaciones empresariales [1]. Aunque carece de una definición ampliamente aceptada, se reconocen[Fow03] como aplicaciones empresariales a los sistemas de control de nómina, inventario, contabilidad, finanzas, etc.

Algunos ejemplos que no corresponden con aplicaciones empresariales son: los sistemas operativos, compiladores, controladores de dispositivo, procesadores de texto, hojas de cálculo, etc.

Si bien las aplicaciones empresariales, en su mayoría, no ejecutan cálculos complejos y tienen principios de operación bastante simples, históricamente han sido casos de estudio notables por su alta frecuencia de fallos, alto costo, gran esfuerzo y considerable tiempo de implementación[Bro95].

No obstante, su estudio ha derivado en técnicas, métodos y herramientas de gran utilidad no sólo para las futuras implementaciones de aplicaciones empresariales, sino para otros tipos de software.

Objetivos básicos.

La funcionalidad específica está determinada por los usuarios, los dueños y el dominio de aplicación. Sin embargo, existen un conjunto de objetivos comunes que todos los sistemas llevan a cabo de una forma u otra durante su operación cotidiana:

- **Obtiene** una abundante cantidad de datos de su dominio de aplicación. Deben poder recibir su entrada de fuentes diversas, ya sean usuarios u otros sistemas, cuya ubicación es incierta o variable.
- **Procesa** grandes volúmenes de datos por unidad de tiempo, sin que dicho procesamiento interfiera de forma perceptible con las actividades de los usuarios.
- **Almacena** parte de los datos que procesa por un periodo de tiempo indefinido, cuya variación va desde algunos milisegundos hasta varios años.
- **Presenta** a sus usuarios información relevante para la toma de decisiones en su ámbito de aplicación.
- **Evoluciona** al ritmo de los cambios de su ámbito de aplicación y su entorno de operación.

Para que un sistema de información sea capaz de llevar a cabo sus objetivos básicos, los desarrolladores requieren implementar una serie de servicios necesarios para casi la totalidad de las implementaciones; debido a ello estos servicios se denominan servicios primarios[2].

La constante necesidad de estos servicios, conduce al desarrollo de tecnologías que encapsulan estos servicios y permiten reutilizarlos para la construcción de diversos sistemas de información. En la segunda parte de este trabajo se analizan de forma detallada algunas de estas tecnologías; el objetivo de esta sección es describir la importancia de estos servicios.

[1] El termino *sistema de información* puede emplearse de forma intercambiable

[2] En el contexto del contenedor de EJBs, Bill Burke[Bur06] considera como *servicios primarios* a los siguientes: (1) concurrencia, (2) persistencia, (3) transacciones, (4) objetos distribuidos, (5) *enterprise messaging*, (6) calendarización, (7) nombrado y (8) seguridad. En este trabajo no se consideran a (6), (7) y (8) como servicios primarios; adicionalmente, (5) y (6) se consideran estilos de intergración[Hoh04] y se agrupan bajo el nombre de integración.

Concurrencia.

Es bastante probable que muchos de los datos en el dominio de la aplicación necesiten ser compartidos entre varios usuarios. La concurrencia tiene que ver con la habilidad del sistema de información para brindar servicio a más de un cliente de forma simultánea. Este requerimiento obliga al sistema a crear mecanismos como los *threads* que permiten ejecutar de forma independiente flujos de ejecución dentro del programa, lo que ayuda al sistema de información a entregar más trabajo por unidad de tiempo, a diferencia de los sistemas de procesamiento por lotes, donde un trabajo se ejecuta en forma secuencial, uno al término de otro.

La solución al problema de la concurrencia siempre crea nuevos problemas que a su vez necesitan ser resueltos:

- La integridad y consistencia de datos. Es común que cuando más de un cliente manipula el mismo conjunto de datos, algunos cambios se pierdan como resultado de la superposición de operaciones. Este problema es indeseable especialmente en dominios de aplicación como el sector financiero, donde las más mínimas inconsistencias puden ocasionar pérdidas económicas de enorme magnitud.

- El bloqueo de recursos. Una forma de asegurar la integridad, es haciendo uso de bloqueos exclusivos (*exclusive lock*) de recursos; sin embargo esta solución a su vez crea problemas cuando dichos bloqueos se hacen de forma descoordinada y con una granularidad inadecuada, resultando por un lado en bloqueos permanentes (*deadlock*), y por otro en un desempeño general bastante pobre.

- Administración de recursos. Cuando la cantidad de clientes es muy alta, el volumen de datos que maneja el sistema de información puede llegar a exceder el espacio de memoria primaria de la máquina sobre la que se hospeda; cuando esto ocurre, los sistemas operativos generalmente transfieren bloques de memoria principal a memoria secundaria. La memoria secundaria es, en la mayoría de los casos, varias órdenes de magnitud más lenta que la memoria primaria. Si la concurrencia es sostenida, el acceso a memoria secundaria inevitablemente crea una contención de acceso, que degrada de forma considerable el rendimiento de la aplicación.

En los sistemas orientados a objetos una forma de mitigar el uso excesivo de recursos tanto de memoria, como de procesador, es limitando el número de instancias de las clases de servicios que se encuentran en un instante del tiempo.

En esta sección se exponen de forma breve, algunas técnicas utilizadas para disminuir el impacto de la alta concurrencia en el desempeño, desde el punto de vista del número de instancias de servicio.

Algunos criterios como (1) la frecuencia en las llamadas de servicio, (2) la duración de la interacción entre el cliente y el servicio, (3) la necesidad de mantener una conversación utilizando datos de estado y (4) la escasez o dificultad en la obtención de los recursos administrados son algunos criterios que en ocasiones hacen más efectiva a una técnica que a otra.

Singleton.

Frecuencia de acceso:	baja - alta
Periodo de interacción:	corto - largo
Control del estado del cliente:	sin estado
Recursos adicionales para prestar el servicio:	recursos intensivos

En estas circunstancias el patrón de diseño *singleton*[Wak06] asegura la existencia de una sola instancia y un único punto de acceso para atender una cantidad muy grande de clientes.

Cada llamada a un método del *singleton*, contiene en sus parámetros toda la información necesaria para llevar a cabo el servicio; debido a que no se mantienen datos de estado, es que una instancia es capaz de atender a una gran cantidad de clientes de forma concurrente, con mínimo impacto en cuanto a la cantidad de memoria empleada. No obstante, todos los recursos que administra el *singleton*, a su vez, deben contar con pocos o ningún dato de estado que ocasione problemas de acceso concurrente.

Instancia por cliente.

Frecuencia de acceso:	media - alta
Periodo de interacción:	medio - largo
Control del estado del cliente:	con estado
Recursos adicionales para prestar el servicio:	recursos no intensivos

Cuando se requiere sostener una interacción de tipo *conversación* con un cliente, es necesario mantener variables de estado; una opción razonable es crear una instancia de la clase de servicio por cada cliente. Emplear una instancia por cliente es una técnica muy utilizada en aplicaciones web, que necesitan mantener una sesión por un periodo de tiempo corto y un contexto de seguridad determinado.

La precondición más importante es que los recursos que administra cada instancia deben ser fáciles de obtener, no ocupar mucha memoria mientras no están en uso y tener bajo consumo de CPU. Los recursos administrados son los que en última instancia imponen el límite de cuántos clientes pueden ser atendidos de forma concurrente.

Cuando los clientes son pocos y la frecuencia de acceso es baja, esta técnica puede no ser muy eficiente en cuanto al gasto de recursos, ya que las instancias de servicio pueden ocupar recursos por largos periodos de tiempo, de forma completamente innecesaria; para éstas circunstancias se han diseñado mecanismos para sacar de la memoria primaria a aquellas instancias no utilizadas, guardar los datos de estado en memoria secundaria y activarlas (traerlas de vuelta a memoria principal) cuando los clientes realicen otra petición de servicio.

Instancia por petición.

Frecuencia de acceso:	baja - media
Periodo de interacción:	corto - medio
Control del estado del cliente:	sin estado
Recursos adicionales para prestar el servicio:	recursos no intensivos

En el caso en que no se requieran mantener datos de estado y la frecuencia de acceso sea baja, una opción es crear una instancia de la clase de servicio por cada petición del cliente.

Crear una instancia por petición puede resultar muy eficiente en términos de memoria, y soportar una enorme cantidad de clientes concurrentes. Para que este enfoque sea capaz de sostener una alta concurrencia, es necesario que los recursos administrados por cada instancia sean muy fáciles de obtener: poca memoria y poco uso de CPU.

Contenedor de instancias.

Frecuencia de acceso:	media - alta
Periodo de interacción:	corto - medio
Control del estado del cliente:	con estado
Recursos adicionales para prestar el servicio:	recursos intensivos

Cuando se requiere un objeto de servicio dedicado a un cliente por periodos de tiempo cortos, pero la escases o la dificultad de la obtención de recursos hace inviable crear una instancia por cliente, existe la alternativa del *contenedor de instancias*[3].

Antes de comenzar a atender las peticiones de clientes, se crean una cantidad inicial de instancias de clases de servicio y se colocan en un *contenedor*. El contenedor, es una construcción lógica que generalmente administra el ciclo de vida de cada uno de los objetos contenidos y coordina la asignación de objetos de servicio a los clientes.

Cuando un cliente requiere iniciar la conversación con la clase de servicio, el contenedor aparta una de las instancias y la reserva de forma exclusiva para ese cliente por la duración total de la conversación. Luego cuando la interacción finaliza, la instancia se *reinicializa*[4] y se regresa al contenedor.

El número de instancias en el contenedor es lo que determina la cantidad máxima de clientes concurrentes que el sistema puede atender. Todos aquellos clientes que realicen una petición de servicio cuando el contenedor esté vacío, deberán esperar indefinidamente a que una de las instancias esté disponible.

Transacciones.

Las transacciones son un mecanismo fundamental para asegurar la consistencia de datos en sistemas de alta concurrencia. Una transacción es un conjunto de tareas que se ejecutan de forma agrupada. Es común referirse a cuatro propiedades fundamentales de las transacciones, conocidas como propiedades ACID[5]:

- *Atomicity*: el conjunto de tareas es una unidad de trabajo, o bien se completan todas o bien no se completa ninguna. El nombre de esta propiedad indica que la ejecución de las tareas es indivisible.
- *Consistency*: el conjunto de operaciones realizadas en el contexto de una transacción deben dejar el modelo de datos en un estado consistente, ya sea si la transacción se completa o se rechaza.
- *Isolation*: las transacciones son independientes unas de otras, el resultado de una transacción no debe influir sobre el resultado de ninguna otra.

[3] En la literatura en inglés sobre aplicaciones empresariales se conoce como *instance pool*.

[4] Se reestablecen a sus valores iniciales, todas las variables de estado utilizadas por el cliente anterior.

[5] Los textos sobre bases de datos relacionales como [Sum07], abordan de forma más extensa estas propiedades y las consecuencias de la alta concurrencia, particularmente los conflictos de lectura y escritura.

- *Durability*: los cambios realizados por las tareas pertenecientes a la transacción deben ser almacenados de forma permanente, sobreviviendo incluso a una falla del entorno de ejecución.

Aún cuando las transacciones son responsabilidad última del almacén de datos, el contexto de ejecución de una transacción puede propagarse hasta la lógica de dominio. Es en estas circunstancias cuando algunas tecnologías permiten establecer de forma clara, ya sea programáticamente o declarativamente, los límites de una transacción dentro de la lógica del dominio.

Persistencia.

La persistencia atiende la necesidad de las aplicaciones de almacenar datos que trasciendan en contexto de ejecución de la aplicación, de forma suficientemente confiable como para sobrevivir a errores de hardware y por periodos de tiempo muy largos.

En esta sección se exponen algunos criterios que deben ser considerados para implementar la solución de persistencia en una aplicación empresarial.

Representación de datos.

Los datos con los que trabaja una aplicación empresarial están representados utilizando jerarquías de clases en tiempo de diseño y grafos de objetos en tiempo de ejecución. Sin embargo, en el almacén de datos, la representación utilizada para almacenar los datos no siempre coincide con la representación de la aplicación. Algunas representaciones usuales para almacenar datos son:

- En forma de grafos: representación utilizada en las bases de datos orientadas a objetos[6], donde los grafos de objetos se guardan utilizando referencias similares a los apuntadores de memoria.
- En forma jerárquica: representación utilizada en formatos de intercambio de datos como XML, donde una entidad se representa como un documento en forma de árbol.
- En forma tabular: la representación más ampliamente utilizada, sus exponentes por excelencia son las bases de datos relacionales[7] que almacenan los datos en tablas que contienen tanto los datos, como sus relaciones[8].

La mayoría de las aplicaciones empresariales de hoy en día asumen almacenes de datos con representación tabular: las bases de datos relacionales.

Ubicación del almacén de datos.

Algunos ejemplos de ubicaciones son la memoria RAM, los discos magnéticos y unidades de red. Cada ubicación del almacén de datos representa problemas distintos; la memoria RAM tiene como propiedades que es rápida (acceso), escasa (capacidad) y volátil (durabilidad), por lo que no es una opción adecuada

[6] Referidas como OODBMS, *Object-Oriented Database Management Systems*.

[7] Referidas como RDBMS, *Relational Database Management System*.

[8] El modelo relacional no toma su nombre de las relaciones (*relationship*) que pueden establecerse entre las tablas, sino de la relación (*relation*) que guardan los atributos de una entidad; algunos expertos en el ámbito académico siguen utilizando el nombre de *relación* en lugar de *tabla*, señalando que el nombre de tabla es más un detalle de implementación que un auténtico concepto. Para estudiar algunas interpretaciones incorrectas del modelo relacional, puede consultarse el capítulo 19 del libro *Date on Database, Writings 2000 - 2006* de C.J. Date, publicado por Apress.

para aplicaciones que no toleran pérdidas de datos; los discos magnéticos son lentos (acceso), abundantes (capacidad) y no-volátil (durabilidad), pero una gran cantidad de accesos siempre repercute negativamente en el desempeño global de la aplicación. Debido a estas propiedades, los algoritmos y las estructuras de datos que se utilizan para el almacenamiento y la recuperación son muy distintos[9] y son un factor que se debe tomar como factor de desempeño en toda aplicación empresarial. La mayoría de las aplicaciones empresariales asumen almacenes de datos ubicados en memoria secundaria.

Disparidad de representación entre el almacén de datos y la aplicación.

Cuando se consumen datos, se deben tomar en cuenta las formas de representación del almacén de datos y la aplicación, así como la ubicación de éstos. Cuando la representación difiere hay básicamente dos caminos para el consumo por parte de la aplicación:

- Convertir la representación del almacén de datos a su forma orientada a objetos.
- Crear en la aplicación, estructuras análogas a la representación nativa del almacén de datos y consumirlas mediante adaptadores.

Hoy en día la mayoría de las aplicaciones se construyen utilizando, por un lado, un entorno de programación orientado a objetos y por otro, una base de datos relacional. La disparidad en la representación de datos entre estos dos modelos ha impactado negativamente el desarrollo de aplicaciones empresariales, especialmente en cuanto a complejidad y a rendimiento. Como se verá en la segunda parte, hay tres alternativas (cada una estudiada con una tecnología distinta) que permiten, si bien no eliminar, al menos reducir los efectos negativos de la disparidad de representaciones en el diseño e implementación de los sistemas de información.

Duración de la interacción con el almacén de datos.

La ubicación del almacén tiene una segunda dimensión, que corresponde con el tiempo de duración de la interacción desde la aplicción. Mantener conexiones abiertas a una base de datos relacional, por ejemplo, es una operación costosa en cuanto a recursos, por lo que se intenta de forma generalizada, reducir el tiempo en que dicha conexión permanece abierta y cerrarla tan pronto como se terminen de enviar o recibir los datos involucrados. También a este respecto hay dos caminos:

- Trabajar con los datos en modo *on-line*: que requiere una conexión permanente con el almacén de datos; a pesar de que es costoso en cuanto a recursos, ofrece como ventaja una mínima desincronización entre el almacén de datos y la aplicación.
- Trabajar con los datos en modo *off-line*: que requiere una copia local de los datos y una conexión temporal. Cada cambio desincroniza la copia local, por lo que al final es necesario reconectarse y actualizar el almacén de datos. En la mayoría de los casos es el camino más adecuado, aunque volúmenes de datos muy grandes hacen imposible mantener copias locales en memoria; para este caso además, debe tenerse especial cuidado en condiciones de alta concurrencia.

Integración.

El crecimiento de una aplicación puede conducirla eventualmente a la necesidad de interactuar con otros sistemas de información. Un caso bastante común hoy en día lo constituye el sistema bancario;

[9] Una extraordinaria referencia al respecto es el libro *File Structures*, de Michael J. Folk, editado por Addison Wesley.

diversos bancos tienen en operación sistemas de información muy diversos, con arquitecturas distintas, con tecnologías distintas y con requerimientos distintos. Sin embargo, es posible utilizar el sistema de un banco para acceder a una cuenta perteneciente a otro banco (administrada por un sistema distinto), hacer transferencias interbancarias, etc. La integración es una necesidad en casi todas las aplicaciones empresariales, incluso cuando la interacción con otros sistemas no sea un requerimiento en la especificación inicial de dicho sistema.

Existen una gran cantidad de enfoques y tecnologías orientadas a resolver el problema de la integración de aplicaciones, sin embargo se han podido identificar[Hoh04] cuatro grandes estilos de integración de aplicaciones, clasificados en función de si la integración es sólo a nivel de datos o a nivel de datos y comportamiento; dicha clasificación se muestra en la figura 1.1.

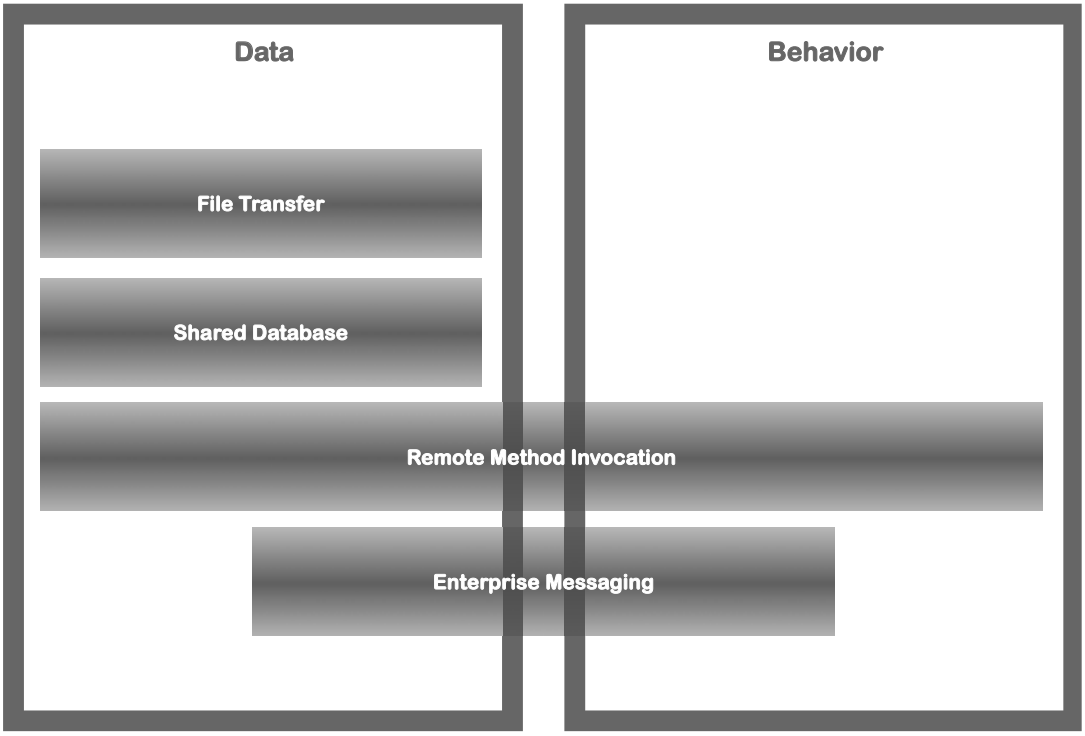


Figura 1.1: Estilos de integración.

Transferencia de archivos (*file transfer*).

Una de las formas más antiguas de integración entre aplicaciones es el intercambio de archivos. Todos los sistemas operativos ofrecen buen soporte para producir y consumir datos almacenados en archivos; en este estilo de integración una de las aplicaciones exporta una porción de su conjunto de datos y otra aplicación los importa, utilizando como intermediario el sistema de archivos.

Este estilo requiere buena coordinación entre las aplicaciones integradas y ofrece pocas posibilidades de automatización. Factores a considerar al utilizar este estilo son:

- Formato del archivo y de los tipos de datos almacenados.
- Periodicidad y caducidad de los datos intercambiados.

Un problema evidente en este estilo de integración consiste en la gran variedad de formatos propietarios de archivos, que introducen mucha resistencia si las aplicaciones no están inicialmente diseñadas para interoperar de esta forma. Una alternativa consiste en la utilización de formatos estándar como XML, que facilitan tanto la producción como el consumo, pero cuya principal desventaja es el gran tamaño de los archivos producidos en comparación con otros formatos binarios.

Dependiendo de las necesidades particulares del dominio, puede ser intolerable una desincronización en los conjuntos de datos; tal requerimiento hace inviable la utilización de archivos.

La ventaja más importante consiste en el enorme grado de desacoplamiento entre las aplicaciones que participan en la integración, haciendo posible que éstas evolucionen de forma independiente. Los procedimientos de extracción y carga son bien conocidos y relativamente sencillos; muchas bases de datos cuentan con herramientas para la generación de archivos de datos y están diseñadas para importar o exportar grandes volúmenes de datos de forma muy eficiente. Por ello, si la desincronización de archivos no es un factor relevante, entonces el uso de archivos es una opción bastante razonable.

Base de datos común (*database sharing*).

Otra técnica de integración consiste en la utilización de un sólo conjunto de datos para todas las aplicaciones participantes en la integración. Dicho conjunto de datos normalmente se concentra en un motor de bases de datos relacional, al que tienen acceso las aplicaciones en cuestión.

En comparación con el uso de archivos, no es necesario preocuparse por el formato y los tipos de datos empleados, además de que no existe desincronización.

Algunas características adicionales presentes en este estilo de integración, se heredan de la utilización de un gestor de base de datos[Sum07]:

- Disponibilidad: los datos están disponibles para una gran cantidad de usuarios y de aplicaciones. Es posible agregar más aplicaciones al esquema de integración sin afectar a las existentes.
- Integridad: a diferencia de los archivos, los gestores de bases de datos cuentan con mecanismos para garantizar en todo momento la integridad del conjunto de datos, incluso cuando existen fallas de hardware.
- Seguridad: a diferencia de los archivos, es posible brindar acceso a información sensible sólo a determinadas aplicaciones y usuarios.
- Independencia: de acuerdo a las necesidades de la aplicación integrada, se pueden ofrecer distintas abstracciones del mismo conjunto de datos, independientemente de la forma en la que se almacenan físicamente, además de que dichos conjuntos son accesibles mediante un lenguaje de consulta estándar: SQL.

La desventaja más importante de este estilo consiste en el alto acoplamiento que resulta entre las aplicaciones integradas. El cambio en los requerimientos de una de ellas podría requerir adecuaciones al esquema de base de datos. La modificación realizada en favor de una de las aplicaciones puede afectar

negativamente al resto de los participantes en el esquema de integración.

Por lo anterior, el uso de una base de datos común puede resultar una técnica exitosa sólo para la integración de aplicaciones dentro de una misma organización, donde es más probable que las aplicaciones evolucionen a un ritmo aproximadamente igual.

Invocación remota (*remote method invocation*).

Los estilos estudiados anteriormente, solo atienden el problema de la integración a nivel de datos; cuando se agrega la necesidad de exportar comportamiento, ninguno de los enfoques anteriores resulta adecuado.

En una aplicación orientada a objetos, los clientes utilizan los servicios que proporciona una clase, sin considerar la ubicación de ésta, pues se asumen accesos locales. La clase expone su funcionalidad a través de los métodos de su interfaz pública, que reciben información en forma de parámetros y generalmente devuelven un resultado derivado del trabajo que realizan.

La técnica de invocación remota permite exportar tanto datos como comportamiento a clientes localizados en ubicaciones accesibles a través de una red, extendiendo éste modelo de programación.

Para extender el modelo de programación orientado a objetos a un ambiente de acceso remoto, son necesarios dos grupos de componentes:

- Un conjunto de clases que implementan la funcionalidad que desea ser exportada. Estas clases normalmente no reflejan en su diseño el requerimiento de accesos remotos y de hecho, la mayor parte de las veces son utilizadas también en un entorno local.
- Clases de infraestructura, cuya responsabilidad consiste en comunicar a los clientes remotos con las clases de servicio, brindando una abstracción sobre las tareas de comunicación de bajo nivel.

El segundo grupo de componentes es el que de hecho hace posible la invocación remota. Esta infraestructura coloca dos intermediarios entre el cliente remoto y los servicios, con el objeto de abstraer del modelo de programación todas las tareas de conexión de bajo nivel. Estos intermediarios se conocen de forma genérica con el nombre de *stub* y *skeleton*[Hoh04]; por convención operan del lado del cliente y del lado del servidor, respectivamente.

Los *stub*s y *skeleton*s normalmente se construyen utilizando el patrón de diseño de *proxy*[Wak06]. El objetivo último del segundo grupo de componentes, es brindar a los clientes remotos el mismo modelo de programación que tendrían si interactuasen con objetos locales. Su responsabilidad se puede dividir en tres subsistemas:

- Infraestructura de conexión: para establecer y administrar la conexión de red entre el cliente y el servidor, además de informar sobre excepciones relacionadas con la comunicación.
- Infraestructura de invocación: que ofrece del lado del cliente la misma interfaz pública de los servicios, con el objeto de crear un modelo de programación lo más parecido posible a utilizar objetos locales; de forma absolutamente transparente para el cliente, utiliza los servicios que brinda la infraestructura de conexión para propagar las llamadas hasta los objetos residentes en el servidor y traer de vuelta los resultados al cliente.
- Infraestructura de serialización: que se encarga de preparan los objetos utilizados para la invocación (parámetros) para ser enviados a través de la red y luego reconstruidos del lado del servi-

dor. De igual manera con los valores de retorno que se originan en el servidor y tiene que viajar de regreso al cliente.

Una desventaja evidente de invocación remota, es que muchas veces crea un alto acoplamiento entre los diversos sistemas que participan en la integración. En aplicaciones donde se comparte un sistema de tipos (alto acoplamiento respecto a las clases), es especialmente difícil la evolución, ya que en algunas plataformas como Java, diversas versiones de una misma clase no son compatibles a nivel binario y por lo tanto no pueden ser serializadas/deserializadas de forma adecuada. Por otro lado, de una forma un tanto menos evidente, Java RMI restringe su funcionamiento a aplicaciones que también utilicen Java como plataforma de desarrollo (alto acoplamiento respecto a la plataforma de desarrollo).

Para que la técnica de invocación remota pueda ser utilizada en un esquema de integración amplio, se deben establecer convenciones mínimas respecto a los subsistemas de invocación y serialización, a manera de minimizar el acoplamiento entre las aplicaciones que conforman el esquema de integración.

Dos aspectos que tienen especial importancia para implementar invocación remota, son la descripción de los servicios que se ofrecen a los clientes remotos y la representación utilizada para serializar los datos:

- En cuanto a la descripción de los servicios, existen mecanismos como IDL (*Interface Definition Language*) y WSDL (*Web Service Description Language*); el segundo es extensamente utilizado y ha demostrado ser muy exitoso al minimizar e incluso desaparecer el acoplamiento respecto a la plataforma de desarrollo.
- Con ayuda de formatos de representación como XML (*eXtensible Markup Language*) o JSON (*JavaScript Object Notation*) se facilita la representación de estructuras complejas y grafos de objetos[10], y minimizan las barreras que un sistema de tipos (y una plataforma de desarrollo) impone en un esquema de integración[11].

Con ayuda de estas convenciones y estándares, es posible construir esquemas de integración basados en invocación remota, como es el caso de las tecnologías de *web services*, cuyos participantes están bien desacoplados y pueden evolucionar independientemente.

Uno de los aspectos más negativos de la invocación remota es, paradójicamente, la transparencia de ubicación que se ofrece al cliente. Debido a que el cliente no es capaz de distinguir entre un objeto local y un objeto remoto, es posible que se sienta libre de realizar una cantidad indiscriminada de llamadas a métodos remotos, degradando de forma muy considerable el rendimiento global del sistema[12].

Mensages asíncronos (*enterprise messaging*).

Pareciera que algunas formas de invocación remota, como *web services*, ofrecieran una solución definitiva al problema de la integración de aplicaciones; sin embargo, existen detalles del modelo de pro-

[10] Un problema latente con estas formas de representación es la gestión de referencias circulares; los objetos se representan como grafos, que permiten referencias circulares, mientras que los documentos XML más parecidos a un árbol, por definición no tienen referencias circulares.

[11] Cualquier forma de representación en modo texto, ofrece algunos inconvenientes, particularmente un mayor esfuerzo de procesamiento y un mayor consumo de memoria respecto a sus equivalentes binarios. Sin embargo, una gran cantidad de desarrolladores consideran que las ventajas rebasan a los inconvenientes, de forma tal que el uso de estas representaciones es una práctica muy extendida en estos días.

[12] A diferencia de una invocación a un objeto local, que ocurre en el orden de los nanosegundos, una invocación remota puede ser tanto o más costosa que un acceso a disco, en el orden de los milisegundos y puede ser incluso mayor dependiendo de las condiciones de la red.

gramación del estilo de invocación remota, que son susceptibles de mejora y en ese sentido es que se expone el estilo de mensajes asíncronos.

Ya se han analizado algunos inconvenientes sobre invocación remota, tal es el caso del alto acoplamiento y del bajo desempeño resultante de una gran cantidad de llamadas remotas. Sin embargo, invocación remota tiene limitaciones más sutiles:

- La impredecible disponibilidad de la red puede ocasionar que las aplicaciones (clientes) que utilizan invocación remota tengan tasas de fallos más altos, o en el mejor de los casos, código de gestión de excepciones muy extenso. Es decir, invocación remota asume que la red está disponible de forma permanente.
- No es posible establecer esquemas de comunicación multidireccional (*multicast* o *broadcast*) entre los clientes y el servidor, ya que esencialmente toda interacción esta limitada entre pares.
- El modelo de programación es inherentemente síncrono[13]: los clientes deben absorber los tiempos de latencia del modelo de petición - respuesta, lo que puede crear un cuello de botella en sistemas de alta concurrencia.

Los mensajes asíncronos son una forma de evitar los inconvenientes del estilo de invocación remota, a la vez que permiten cumplir con el objetivo de integrar aplicaciones a nivel de datos y comportamiento; aún más, brinda casi el mismo grado de desacoplamiento que la transferencia de archivos, sin los inconvenientes de los formatos, la desincronización y la caducidad de los datos.

Para implementar una solución de integración que utilice mensajes asíncronos, es necesario contar con un intermediario entre el productor de servicios y el consumidor de servicios; dicho intermediario se conoce con el nombre de MOM (*Message-Oriented Middleware*). Las peticiones de servicio de un cliente se mandan a una ubicación lógica dentro del intermediario y éste a su vez envía el mensaje al proveedor de servicio; el mismo esquema se puede aplicar con los valores de retorno, si es que existe alguno.

Este nivel de indirección añadido permite superar las limitaciones comentadas con anterioridad:

- Puesto que las peticiones de servicio se envían a través de un intermediario, no es necesario asumir que la conexión es permanente. El MOM en muchas ocasiones puede configurarse para entregar los mensajes a sus destinatarios cuando la comunicación con éste se re-establezca[14].
- Los mensajes producidos por un cliente no necesariamente tienen que ser entregados a un sólo destinatario; la existencia del intermediario y el concepto de ubicaciones lógicas, permite al productor enviar mensajes a una ubicación lógica, luego el MOM puede permitir a varios consumidores suscribirse a las notificaciones de llegada de mensajes a cierta ubicación lógica, de forma que se establezca una comunicación multidireccional[15]. Incluso pueden configurarse una entrega selectiva de mensajes a los consumidores, basada en el contenido del mismo.
- El modelo de programación se simplifica, ya que no es necesario para el cliente esperar a que la petición de servicio se ejecute, para continuar con el resto del flujo del programa[16]; debido a ello,

[13] Aunque algunas tecnologías de *web services* permiten crear interfaces de servicio con comportamiento asíncrono.

[14] El almacenamiento temporal y la confirmación de entrega de mensajes, entre otras cosas ofrecen una comunicación *confiable* (*reliable messaging*).

[15] Las ubicaciones lógicas que permiten comunicación *peer-to-peer* se conocen como colas (*queues*) y las ubicaciones lógicas multidireccionales se conocen como publicaciones (*topics*).

[16] Esta característica se denomina *fire & forget*.

puede incrementar el desempeño de sistemas de alta concurrencia sobre otras opciones como invocación remota.

- Las aplicaciones que participan en un esquema de integración están bien desacopladas y pueden evolucionar de forma independiente. Incluso cuando los sistemas de tipos no son compatibles, se puede configurar al MOM para interceptar los mensajes y redirigirlos a traductores de mensajes (*message translator*[Hoh04]) antes de ser enviados a los consumidores.

Aunque las ventajas que ofrece el estilo de mensajes asíncronos son muy extensas, es uno de los modelos más complejos en cuanto a implementación. Debido a que las más recientes tecnologías de *web services* cuentan con características de este modelo (por lo tanto, muchas de sus ventajas), es que en este trabajo no se profundiza en el estudio de tecnologías de mensajes asíncronos.

Otros servicios no considerados.

Dentro de la gran cantidad de servicios de propósito general que pueden ser de ayuda para implementar aplicaciones empresariales, se mencionan tres servicios más, aparte de los expuestos, que sin ser considerados primarios, son utilizados por muchas aplicaciones empresariales.

Calendarización.

La mayoría de las aplicaciones están conducidas por la interacción con sus usuarios; sin embargo, existen algunas tareas que deben ser conducidas por temporizadores, es decir, elementos programables que generan eventos en intervalos de tiempo regulares.

Algunas tecnologías como los gestores de bases de datos y la mayoría de las plataformas de desarrollo, cuentan con puntos de extensión donde es posible configurar eventos disparados por un temporizador y asignar un conjunto de acciones cuando dicho temporizador expire.

A pesar de que no todas las aplicaciones empresariales incluyen este tipo de requerimientos, la calendarización de eventos se ha vuelto muy común en estos días; no obstante, la calendarización es un servicio que lejos está de ser indispensable en buena parte de los nuevos sistemas de información y por ello no se consideran dentro de los servicios primarios.

Seguridad.

En el caso de la seguridad se ha hecho una consideración muy particular sobre esta categoría de servicios: la seguridad es un requerimiento que debe ser considerado desde el inicio en el diseño de todo sistema de información; en la implementación en cambio, el aspecto de la seguridad puede ser adicionado posteriormente a la codificación del resto de la lógica del dominio, especialmente mediante el uso de AOP (*Aspect-Oriented Programming*).

Es por ello que sin desestimar la importancia de la seguridad en etapas tempranas del ciclo de vida, en este trabajo, cuyo punto central son las tecnologías de implementación, no se considera a la seguridad como un servicio primario.

Registros y nombrado.

Existen algunos componentes de las aplicaciones empresariales que hacen uso de un espacio global de objetos; este espacio brinda la funcionalidad mínima de registro y localización de objetos en ámbitos

de ejecución muy amplios; usualmente estos registros exponen la interfaz de una tabla de dispersión, donde la llave más utilizada es una cadena de caracteres.

Aún y cuando para muchas aplicaciones la existencia de un servicio de nombrado y directorio facilita considerablemente la implementación, para el caso de este trabajo no se considera como un servicio primario.

Capítulo

2

Conceptos básicos de arquitectura.

Una vez hecho un resumen de los servicios primarios, se estudiará uno de los patrones de arquitectura más ampliamente utilizados en el desarrollo de aplicaciones empresariales: arquitectura en capas. En esta sección se estudian los denominados servicios primarios, su ubicación dentro de la estructura general de una aplicación empresarial y las funciones agrupadas de cada capa.

Patrones de organización de la lógica de dominio.

La parte central de toda aplicación empresarial está constutuida por los requerimientos y reglas que se enmarcan en su ámbito de dominio. Esta parte es la que verdaderamente otorga valor agregado a las organizaciones que utilizan los sistemas de información.

Existen distintas formas de organizar la lógica de dominio en la estructura general de una aplicación empresarial, cada una de ellas es más o menos conveniente dependiendo del tamaño y la complejidad de la aplicación; lo cierto es que cuando se selecciona un modelo de organización de la lógica de dominio, tal decisión afecta la forma en la que construyen y se comunican otras partes del sistema, particularmente la parte encargada de los servicios de persistencia.

En esta sección se estudian algunos de los patrones más utilizados de acuerdo a la taxonomía propuesta por Martin Fowler en PoEAA[Fow03][1]: *transaction script*, *domain model* y *service layer*[2].
Para clarificar el impacto a nivel de código entre *transaction script* y *domain model*, se considera la funcionalidad mínima de un sistema de órdenes: agregar una lista de productos a una orden y remover un producto de la lista. En comparación con una aplicación empresarial típica, el ejemplo seleccionado para esta sección raya en lo trivial, pero ayuda a subrayar algunos puntos que se enumeran en la descripción de estos patrones.

Transaction Script.

Aún cuando se asume un entorno de programación orientado a objetos, el patrón *transaction script* organiza la lógica en procedimientos, que se inician generalmente por eventos que ocurren en la interfaz gráfica del sistema.

En este patrón, se considera que las clases que representan los objetos del dominio no contienen mucho comportamiento; la lógica de dominio se encapsula en procedimientos (es usual encontrar *singleton*s o métodos estáicos) que operan directamente con el estado de los objetos de dominio, realizan operaciones y dirigen los servicios de persistencia. Las transacciones tienen límites perfectamente bien definidos, demarcados por el inicio de la ejecución de un método (seguido de cualquier cadena de invocaciones a otras subrutinas) y el punto de retorno.

La simplicidad y eficiencia de este patrón es una de sus mayores ventajas, aunque sólo se recomienda cuando la aplicación contiene relativamente poca lógica de dominio. En cuanto a los servicios de persistencia, este patrón de diseño permite utilizar los datos en la representación nativa del almacén de datos, o bien, construir clases relativamente símples (mucho estado, poco comportamiento), cuya dificultad de conversión con la representación del almacén de datos es mínima.

[1] Se utilizarán los nombres tal y como fueron propuestos por el autor en el texto original, escrito en inglés.

[2] Existe otro patrón de organización de lógica de dominio denominado *table module*, se omite en este trabajo debido al alto grado de acoplamiento respecto a la base relacional, al esquema físico y a los objetos que modelan dicho esquema del lado de la aplicación (*record sets*).

Uno de los mayores inconvenientes de este patrón es que no fomenta una correcta encapsulación de la funcionalidad, incluso tareas comunes entre los *scripts* tiende a duplicarse. Por otro lado, no encaja bien en el paradigma de programación orientado a objetos, ya que en la mayoría de los casos se manipula con toda libertad el estado de los objetos del dominio, lo que puede resultar en muchos comportamientos inconsistentes, si el mismo conjunto de datos se utiliza como entrada para distintos *transaction script*.

Por lo anterior, queda claro que es un patrón conveniente para sistemas relativamente estáticos y sin cambios importantes en la lógica de dominio; en otras circunstancias, la complejidad que resulta de realizar cambios en código potencialmente duplicado, hacen de este patrón una elección con alto riesgo de fallas en la etapa de mantenimiento de la aplicación.

El código del ejemplo del sistema de órdenes para el caso de *transaction script* se muestra en la figura 2.1.

Domain Model.

Este patrón de organización de la lógica de dominio es la forma ideal de construir sistemas orientados a objetos. El modelado de cada clase intenta es una astracción fiel de los objetos equivalentes en el dominio de la aplicación. Tales objetos están compuestos tanto por datos, como por comportamiento.

Con *domain model* la lógica de dominio se divide en pequeñas unidades de comportamiento asignadas a cada objeto (los métodos administran la vista del estado para los clientes de la clase). La mayoría de las veces es necesaria la interacción entre una gran cantidad de objetos para implementar comportamientos complejos. La ventaja es que cada objeto tiene asignada una cantidad finita y bien determinada de trabajo, su estado está correctamente encapsulado a través de la interfaz pública y por lo tanto, los clientes de las clases del modelo de dominio están bien protegidas de los cambios en la implementación interna.

Un modelo de dominio generalmente resulta en un conjunto de objetos independientes de otros componentes estructurales de la aplicación, como puede ser el caso de los servicios de persistencia o la presentación. Cuando se explotan todas las posibilidades del modelo de programación orientado a objetos, que involucran la herencia, la composición y diversos patrones de diseño[Wak06], en ocasiones resulta difícil implementar los servicios de persistencia debido al siempre presente problema de la disparidad en la representación de datos, entre el modelo orientado a objetos y el modelo nativo del almacén de datos.

Un inconveniente de *domain model* radica en que la lógica de dominio dispersa en varios objetos de dominio, es difícil de enmarcarse en una transacción, puesto que tales objetos están diseñados para no depender de la infraestructura de otros servicios primarios.

Ningún patrón está excento de inconvenientes, sin embargo, en el caso de *domain model* se pueden diseñar sistemas de información con mucha lógica de dominio con un modelo de objetos relativamente fácil de mantener; éste es otro patrón donde las ventajas superan por mucho a las desventajas.

El código del ejemplo del sistema de órdenes para el caso de *domain model* se muestra en la figura 2.2.

Service Layer.

Existen sistemas de información, cuya lógica de dominio es consumida por una variedad de sistemas heterogéneos externos, tal es el caso de las aplicaciones que participan como el punto central de un sis-

```
class Product
{
    public string Name { get; set; }
    public decimal Price { get; set; }
}

class Order
{
    public List<Product> Products { get; protected set; }
    public decimal Total { get; set; }

    public Order() { Products = new List<Product>(); }
}

static class OrderProcessor
{
    public static Order CreateOrder(List<Product> selectedProducts)
    {
        Order order = new Order();

        foreach (Product product in selectedProducts)
        {
            order.Products.Add(product);
            order.Total += product.Price;
        }

        return order;
    }

    public static void RemoveItem(Order order, Product product)
    {
        if (order.Products.Contains(product))
        {
            order.Products.Remove(product);
            order.Total -= product.Price;
        }
    }
}
```

Figura 2.1: Implementación de un sistema de órdenes según *transaction script* (C#).

```
class Product
{
    public string Name { get; set; }
    public decimal Price { get; set; }
}

class Order
{
    protected List<Product> ProductList { get; set; }
    public IEnumerable<Product> Products { get { return ProductList; } }
    public decimal Total { get; protected set; }

    public Order() { ProductList = new List<Product>(); }

    public Order(List<Product> selectedProducts) : this()
    {
        SetSelectedProducts(selectedProducts);
    }

    public void SetSelectedProducts(List<Product> selectedProducts)
    {
        foreach (Product product in selectedProducts)
        {
            AddProduct(product);
        }
    }

    public void AddProduct(Product product)
    {
        if (!ProductList.Contains(product))
        {
            ProductList.Add(product);
            Total += product.Price;
        }
    }

    public void RemoveProduct(Product product)
    {
        if (ProductList.Contains(product))
        {
            ProductList.Remove(product);
            Total -= product.Price;
        }
    }
}
```

Figura 2.2: Implementación de un sistema de órdenes según *domain model* (C#).

tema de integración o aquellas aplicaciones con múltiples interfaces gráficas de usuario (por ejemplo, modo consola, aplicación de escritorio y aplicación web).

Service Layer permite crear un límite del sistema con sus potenciales clientes (aplicaciones integradas o interfaces de usuario), mediante un conjunto bien definido de operaciones explícitas. Dichas operaciones se organizan en un nivel de abstracción distinto a otros patrones de organización de lógica de dominio. Una aplicación empresarial que implementa *service layer* puede organizar su lógica de dominio utilizando tanto *transaction script* como *domain model* y aún así no revelar a sus clientes de esta decisión de arquitectura.

La utilización de *service layer* junto con *domain model* es una combinación muy utilizada en la implementación de aplicaciones empresariales[Fow03] con fuertes requerimientos de integración. Generalmente las clases que integran *service layer* proveen a sus clientes de aquella funcinalidad establecida directamente por los casos de uso.

Este patrón brinda un punto de acceso a la lógica contenida en el modelo de dominio y coordina la respuesta de la aplicación a cada petición hecha por los clientes; algunas de las responsabilidades añadidas que pueden atenderse mediante *service layer* son las siguientes:

- Coordinar el flujo de trabajo que involucra la participación de otros objetos del dominio.
- Notificar sobre el avance de una tarea determinada.
- Convertir los datos a una representación útil para la entidad consumidora.
- Delimitar el contexto de ejecución de las transacciones.
- Imponer restricciones de seguridad por medio de autenticación y autorización.
- Permitir la segregación de la interfaz de servicios, dependiendo de la ubicación del cliente: local o remota.

De acuerdo a PoEAA, existen dos variantes de implementación para *service layer*:

1. *Domain Façade*: que utiliza una serie de clases de fachada con poca lógica de dominio, que escencialmente delegan responsabilidades sobre las clases del modelo de dominio.
2. *Operation Script*: que utiliza una serie de clases de fachada que implementan una buena cantidad de lógica de dominio, pero delegan buena parte a las clases del modelo de dominio[3].

   La desventaja es que para buena parte de las aplicaciones empresariales, *service layer* contiene operaciones directamente relacionadas con el servicio de persistencia[4] y por lo tanto, no agregan ningun valor a la estructura general de la aplicación; por el contrario, una capa de abstracción que sólo redirecciona llamadas al modelo de dominio, de hecho disminuye el desempeño de la aplicación[5].

[3] Esta opción de implementación sugiere un enfoque combinado de otros patrones de organización: para la lógica de alto nivel (*workflow*) se utiliza *transaction script*; la lógica de bajo nivel se delega a *domain model*.

[4] Se denominan operaciones CRUD(*Create, Readm Update, Delete*).

[5] A penas de forma perceptible, sobre todo si se utiliza en ambiente distribuido.

Patrones de acceso a datos.

Los servicios de persistencia representan un constante reto de implementación en el desarrollo de aplicaciones empresariales; algunos aspectos como la complejidad de los datos de dominio y los modelos de organización de la lógica, ofrecen varios caminos para diseñar los servicios de acceso a datos.

Debido a que los almacenes de datos generalmente se encuentran ubicados en memoria secundaria, y su incorrecta implementación siempre ocasiona *cuellos de botella* en el rendimiento de una aplicación, es que se requiere prestar especial atención a las distintas formas de acceso a datos.

En esta sección se estudian dos patrones de acuerdo a la taxonomía de Martin Fowler[Fow03]: *active record* y *data mapper*. Se intentará, mediante fragmentos de código, resaltar algunas de las ventajas y desventajas de estos patrones y mostrar las combinaciones más frecuentes de patrones de persistencia y organización de la lógica de dominio.

Active Record.

Como se ha mencionado anteriormente, el almacén de datos más empleado para desarollar aplicaciones empresariales es la base de datos relacional. La idea detrás de *active record* es la de proporcionar una estructura de datos parecida a la representación nativa del almacén de datos.

En el caso de la base de datos relacional, un *active record* es un objeto que tiene una relación 1:1 respecto a las tuplas contenidas en una tabla de la base de datos: cada atributo en la tabla (columna) corresponde exactamente con un atributo (propiedad) en el objeto.

Cada instancia de *active record* es responsable de administrar su estado persistente y es común encontrar métodos para guardar la tupla, actualizar la tupla, borrar la tupla, etc. Los métodos de recuperación y búsqueda de instancias de muchas veces se implementan utilizando métodos estáticos, que regresan colecciones de objetos *active record*.

En general, existe una clase *active record* por cada tabla en la base de datos y una instancia por cada tupla. En caso de que se combine con *domain model* estos objetos pueden tener comportamiento, que corresponde con un parte de la lógica de dominio[6].

La principal ventaja de *active record* es que es extremadamente fácil de implementar y de utilizar.

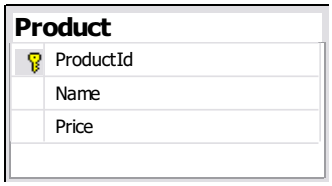
No obstante, como se verá más adelante, los objetos están fuertemente acoplados al esquema de base de datos y por lo tanto, cualquier cambio en el esquema impactará negativamente a todos los clientes de la clase. En presencia de una gran cantidad de lógica de dominio, las clases de persistencia estan sobrecargadas de responsabilides, lo que las hace frágiles ante el cambio.

Un inconveniente adicional consiste en que, como la persistencia está gestionada de forma individual, trabajar con una gran cantidad de instancias puede resultar en una contención de acceso a disco, lo que

[6] En PoEAA[Fow03], Martin Fowler expone dos patrones distintos de acceso a datos a nivel de tupla: *row data gateway* y *active record*, cuya única diferencia es que el segundo contiene lógica de dominio, mientras que el primero no. No se consideró adecuado incluir dos patrones de persistencia con diferencias tan sutiles, de forma que para evitar confusión, en este trabajo la discusión se limita a *active record*.

impactará considerablemente el rendimiento de la aplicación.

En resumen, cuando las entidades del dominio son simples, *active record* es una buena alternativa de persistencia, muy fácil de utilizar e implementar, con una interfaz pública realmente intuitiva. En las figura 2.3 se muestra un ejemplo sencillo donde se puede apreciar el gran parecido del esquema de la base de datos con la clase *active record*.



(a) Esquema de base de datos.

```
CREATE TABLE [Product]
(
    [ProductId] INT NOT NULL,
    [Name] VARCHAR(64) DEFAULT '',
    [Price] MONEY DEFAULT 0.0,
    CONSTRAINT [PK] PRIMARY KEY([ProductId])
)
```

(b) Esquema de base de datos (SQL).

```
class Product
{
    // Properties
    public int ProductId { get; protected set; }
    public string Name { get; set; }
    public decimal Price { get; set; }

    // Constructor
    public Product(int productId) { ProductId = productId; }

    // Domain Logic Methods
    public void DoublePrice() { Price = 2 * Price; }

    // Data Access Methods
    public void Save() { /* Data source specific */ }
    public void Delete() { /* Data source specific */ }
    public void Update() { /* Data source specific */ }

    // Finder Methods
    public static Product GetById(int productId) { /* ... */ }
    public static List<Product> FindByName(string name) { /* ... */ }
}
```

(c) Código de la clase (C#).

Figura 2.3: Persistencia con *Active Record*.

Data Mapper.

Considerando algunas de las desventajas presentadas para el patrón *active record*, queda claro que es necesario un mecanismo alternativo de persistencia.

Data Mapper es un patrón que permite desacoplar al modelo de dominio de los detalles de representación del almacén de datos[7]. Un *data mapper* se encarga de:

- Encapsular la responsabilidad de persistencia en objetos distintos del modelo de dominio[8].
- Convertir la representación nativa del almacén de datos a la representación del modelo de dominio.
- Convertir las relaciones complejas entre objetos, como es el caso de la herencia, la cardinalidad de las relaciones de composición, las colecciones de objetos etc., a la representación nativa del almacén de datos.
- Aislar al modelo de dominio de los cambios del esquema de base de datos.
- Aislar el esquema de base de datos se su representación orientada a objetos.

Puesto que el *data mapper* es el encargado de convertir representaciones de datos desde la aplicación y hacia el almacén de datos, debe de tener conocimiento sobre la estructura de ambos modelos. Asumiendo una base de datos relacional, el caso más común, se requiere conocer el esquema de la base de datos (nombre de tablas, nombre de campos, tipos de datos, etc.) y de la clase correspondiente en el modelo de dominio (las propiedades).

Un correcto diseño de las clases del modelo de dominio, generalmente establece la no utilización de propiedades públicas, por lo que todo estado persistente del objeto almacenado en variables de instancia con visibilidad restringida, es claramente un obstáculo para la implementación de *data mapper*.

En estos casos usualmente se utilizan los mecanismos de introspección sobre tipos[9] para descubrir la información relativa a los nombres y tipos de datos de los objetos persistentes y establecer los valores en tiempo de ejecución, independientemente de su visibilidad declarada. En el caso del esquema de base de datos, la información se proporciona de manera externa, ya sea por medio de un archivo de configuración o por cualquier mecanismo de anotaciones que la plataforma de desarrollo disponga, y que sobreviva en tiempo de ejecución.

Una ventaja de implementar el servicio de persistencia utilizando *data mapper*, es que éste fácilmente puede utilizarse con distintos tipos de almacén de datos, si el servicio de persistencia está diseñado a base de abstracciones de alto nivel.
En la figura 2.4 se muestra la implementación de un objeto de dominio y la abstracción del servicio de persistencia utilizando un *data mapper*.
Debido a que mucha de la responsabilidad de un *data mapper* es básicamente la misma, independientemente del objeto de dominio, es posible construir código genérico que sea capaz de convertir de cualquier tabla (base de datos) a cualquier tipo (aplicación). Aprovechando algunas características de los modernos lenguajes de programación es posible construir tipos genéricos, que luego sólo necesitan ser personalizados en tiempo de ejecución, un ejemplo de ésto se aprecia en la figura 2.5.
Todas las características comentadas hacen que *data mapper* como servicio de persistencia, sea un excelente complemento para *domain model*.

[7] Este atributo de los modelos de dominio se refiere como *persistence ignorance*.
[8] Responsabilidad que en *active record* por ejemplo, estaría delegada a los objetos del modelo de dominio.
[9] Lo que en inglés se conoce como *reflection*.

```
class Product
{
    // Properties
    public int ProductId { get; protected set; }
    public string Name { get; set; }
    public decimal Price { get; set; }

    // Constructor
    public Product(int productId) { ProductId = productId; }

    // Domain Logic Methods
    public void DoublePrice() { Price = 2 * Price; }
}

interface IProductMapper
{
    // Data Access Methods
    void Save(Product product);
    void Delete(Product product);
    void Update(Product product);

    // Finder Methods
    Product GetById(int productId);
    List<Product> FindByName(string name);
}
```

Figura 2.4: Persistencia con *Data Mapper* (C#).

```
interface IGenericMapper<Entity, Key>
{
    // Data Access Methods
    void Save(Entity entity);
    void Delete(Entity entity);
    void Update(Entity entity);

    // Finder Methods
    Entity GetById(Key id);
}
```

Figura 2.5: *Data Mapper* con tipos genéricos (C#).

Aspect Oriented Programming.

Esta sección se dedica a hacer un breve estudio de AOP (*Aspect Oriented Programming*), un paradigma de programación de creciente importancia en el desarrollo de aplicaciones empresariales. AOP no es un reemplazo de la programación orientada a objetos, sino un paradigma complementario.

AOP permite reducir la duplicidad de código; encapsular funcionalidad dispersa y aplicar (reutilizar) de forma transparente esta funcionalidad en distintas partes de la aplicación. AOP es especialmente útil en la implementación de algunos servicios como seguridad, *profiling*, transacciones, etc.

AOP se creo a mediados de la década de 1990, por Gregor Kiczakes en el *Palo Alto Research Center* de Xerox, como un intento para reducir la duplicidad de código en los sistemas orientados a objetos[Nil06].La abstracción básica en este paradigma son los *aspectos*.

Problemas que originan la necesidad de AOP.

Dispersión de funcionalidad (*code scattering*).

Un servicio requerido frecuentemente durante el desarrollo (y pruebas) de aplicaciones empresariales, consiste en medir y reportar el tiempo de ejecución de algunos componentes.

El código de la figura 2.6 muestra una clase donde se mezcla la medición del tiempo de ejecución junto con el resto de la lógica del componente. Si se observa el ejemplo se podrá notar que la cantidad de código duplicado es proporcional a la cantidad de partes (métodos) de un componente (clase) que se deseen medir.

En un sistema orientado a objetos se busca encapsular el código duplicado para centralizarlo en un punto sensible. Una forma de hacer esto es crear una clase cuya responsabilidad sea capturar el tiempo de ejecución de una cierta parte de un componente (presumiblemente un método), como lo muestra la clase `SimpleProfiler` del ejemplo 2.7.

Ahora sólo resta aplicar esta funcionalidad a todos los componentes que requieran *profiling*. Una opción es utilizando el patrón de diseño *command*[Wak06], de forma que todas las clases que requieran *profiling* se encapsularán en un objeto comando. De esta forma, se abstrae el contexto de ejecución de un método y es trivial crear una versión especializada de un *profiler* que trabaje exclusivamente con *commands*, como se muestra en la figura 2.8.

El uso del patrón de diseño *command*, no obstante, obliga al desarrollador a refactorizar una porción considerable del código para encapsular el contexto de invocación; si se toma en cuenta la cantidad de factores que hay que abstraer, como los parámetros formales, valores de retorno y excepciones lanzadas, quizá se llegue a la conclusión de que todo este esfuerzo podría resultar en un esfuerzo excesivo y finalmente innecesario, pues *profiling* es un servicio complementario, y que con muchas probabilidades, será deshabilitado cuando el sistema de información llegue a su etapa de producción.

AOP es precisamente un paradigma de programación que brinda una solución elegante, fácil de implementar, fácil de habilitar y deshabilitar, para problemas que de otra forma requerirían un gran esfuerzo de implementación.

```
public class ProductMapper
{
    public <T> T get(Object id, Class<T> clazz)
    {
        long milliseconds = System.currentTimeMillis();

        /* Lookup code */

        milliseconds = System.currentTimeMillis() - milliseconds;
        System.out.println(milliseconds);

        return null;
    }

    public void delete(Object product)
    {
        long milliseconds = System.currentTimeMillis();

        /* Delete code */

        milliseconds = System.currentTimeMillis() - milliseconds;
        System.out.println(milliseconds);
    }
}
```

Figura 2.6: Clase con evidente duplicidad de código (Java).

```
public class SimpleProfiler
{
    void profile()
    {
        long milliseconds = System.currentTimeMillis();

        /* Execute domain logic here (But how?) */

        milliseconds = System.currentTimeMillis() - milliseconds;
        System.out.println(milliseconds);
    }
}
```

Figura 2.7: SimpleProfiler: encapsulación del código de *profiling* (Java).

```
public interface Command
{
    void execute();
}
```

```
public class CommandProfiler
{
    void profile(Command command)
    {
        long milliseconds = System.currentTimeMillis();

        command.execute();

        milliseconds = System.currentTimeMillis() - milliseconds;
        System.out.println(milliseconds);
    }
}
```

Figura 2.8: CommandProfiler: encapsulación del código de *profiling* junto con el patrón *command* (Java).

Crosscutting functionalities.

Para este problema[10] es necesario considerar un sistema de información que administre un catálogo de productos.

Para simplificar el problema se asume que sólo existen dos componentes: la clase que representa a los productos (`Product`) y la clase que representa el catálogo de productos (`Catalog`), organizados de acuerdo al patrón *domain model* para lógica de dominio y *active record* para persistencia, cuya estructura se bosqueja en el código del ejemplo 2.9.

La funcionalidad estudiada en esta sección consiste en verificar la integridad referencial, específicamente: cuando se borre un producto es necesario verificar que no existan entradas en el catálogo para ese producto, o bien que tenga cero existencias.

Aunque en un primer momento quizá parezca una buena idea agregar esta lógica al código del *active record* `Product`, ésto dificulta la reutilización de esa clase en un contexto distinto a la aplicación del catálogo de clientes; quizá se pueda concluir que finalmente, esta responsabilidad no pertenece a la clase `Product`.

La alternativa en este sistema de dos clases es agregar esa lógica a la clase `Catalog`, pero tampoco resulta muy adecuado, pues es necesario interceptar las llamadas al método `Product.delete()`, lo que no parece ni fácil de hacer ni muy intuitivo. Después de todo, la conclusión es la misma: esta responsabilidad no pertenece a la clase `Catalog`.

[10] Desafortunadamente no existe un témino simple en español que tenga el mismo significado.

```
class Product
{
    private String model;
    private double price;

    public void delete() { /* Delete code */ }
    /* ... */
}

class Catalog
{
    private Map<Product, Integer> content;

    /* ... */
}
```

Figura 2.9: Diseño de clases del sistema de catálogo de productos (Java).

Esta situación ilustra que en ocasiones el paradigma de programación orientado a objetos, no soporta la implementación de cierto tipo de lógica que abarca a más de una clase.

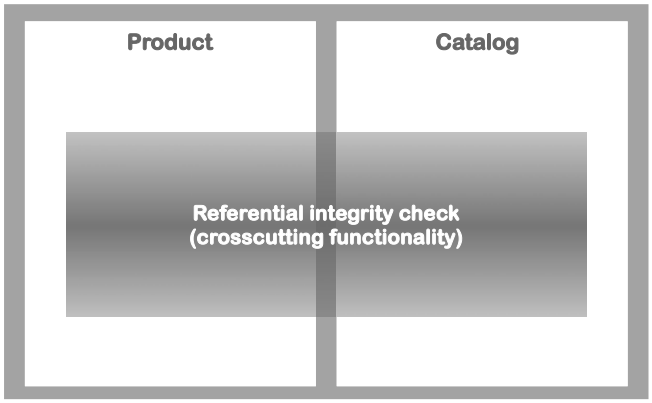


Figura 2.10: *Crosscutting functionality*: problema originante de AOP.

En muchas aplicaciones empresariales, la integridad referencial se implementa en una de las dos clases que itervienen en la relación, lo que introduce como efecto negativo, alto acoplamiento. El alto acoplamiento es indeseable, no obstante, la falta de una mejor solución orilla al desarrollador de aplicaciones a utilizar una de las variantes aquí expuestas.

Terminología de AOP.

El primer concepto que lógicamente llama la atención es el de *aspecto*, cuya definición es abstracta de la misma forma que el concepto de clase. Un **aspecto** es una unidad de programación donde se encapsula funcionalidad dispersa(*crosscutting functionalities*)[Paw05].

Una aplicación basada en el paradigma AOP consiste de clases y aspectos, donde las clases implementan la funcionalidad básica de la aplicación (*dimensión estructural*) y los aspectos implementan aquella funcionalidad dispersa (*dimensión operacional*).

Para obtener una aplicación con la funcionalidad provista tanto por las clases como por los aspectos, es necesario un segmento de código que combine los módulos que contienen la dimensión estructural y la dimensión operacional. A la acción de introducir *aspectos* dentro del código de las clases se le denomina *aspect weaving* y al programa que realiza esta combinación se le denomina *weaver*.

Cuando se habla de combinar las clases con aspectos, es necesario considerar aquellos puntos en la ejecución del programa donde se aplicará la funcionalidad provista por los aspectos. Si bien en teoría, cada instrucción de un programa escrito en un programa de alto nivel, es un potencial receptor de un aspecto, en la práctica, sólo es posible aplicarlos en una pequeña cantidad de construcciones de alto nivel. Estas ubicaciones se denominan *joinpoints* y los representantes por excelencia son los métodos.

De todo el universo de *joinpoints* disponibles en un componente, es indispensable que el desarrollador de aplicaciones pueda aplicar selectivamente los aspectos a un subconjunto específico de *joinpoints*, definido a conveniencia. El conjunto de todos los *joinpoints* donde será aplicado un aspecto se denomina *pointcut*.

El último concepto relevante para esta discusión, corresponde al código que implementa la funcionalidad específica provista por un aspecto, que se conoce como *advisor*[11]. Este código generalmente está contenido en un conjunto de métodos pertenecientes a una o varias clases.

Funcionalidad encapsulada mediante AOP.

A continuación se replantea el problema del servicio de *profiling*, descrito al inicio de esta sección para la clase `ProductMapper` y se proporciona una solución basada en la aplicación de los conceptos de AOP.

Para codificar una solución utilizando AOP, es necesario primero identificar en el problema de *profiling* las distintas partes de la solución de acuerdo a la terminología de AOP:

- El código que mide la duración del tiempo de ejecución es el *advisor*.
- El universo de todos los métodos de una clase (todos los posibles métodos de una clase que podrían utilizar el servicio de profiling) son los *joinpoints*.
- El subconjunto de los métodos de una clase a los cuáles se desea aplicar el servicio de *profiling* son el *pointcut*.

[11] Existen al menos tres tipos de *advisor*[Paw05] dependiendo de la posición relativa respecto al *joinpoint*: *before*, que se ejecuta inmediatamente antes del *joinpoint*; *after* que se ejecuta inmediatamente después del *joinpoint*; y *arround* que contiene código que rodea al *joinpoint*.

- El código que aplica el servicio de *profiling* a los métodos que conforman el *pointcut* es el *weaver*.

```java
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Aspect { }
```

```java
public interface InvocationContext
{
    public Object getTarget();
    public Method getMethod();
    public Object[] getParameters();
    public Object proceed() throws Exception;
}
```

```java
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Advice { }
```

Figura 2.11: *Profiling* mediante AOP (I) (Java).

El código escrito en lenguaje Java que implementa el servicio de *profiling*, contenido en los ejemplos 2.11, 2.12 y 2.13, muestra varios elementos importantes:

- Aunque no es indispensable, en este ejemplo se consideró importante proveer un mecanismo para identificar los aspectos: la anotación `@Aspect`.
- Es indispensable ubicar a los métodos que proveen la implementación del *advisor*, para ello se usa la anotación `@Advice`.
- De una forma similar al patrón de diseño *command*, se requiere una abstracción del contexto de ejecución del método objetivo y dicha abtracción se modela mediante la interfaz `InvocationContext`. Esta interfaz provee información del método objetivo, como son sus parámetros formales, la instancia específica sobre la que se invoca el método, etc. Especialmente importante es el método `InvocationContext.proceed()`, el cuál no sólo determina el tipo de *advisor* (*after*, *before* o *around*), sino que permite señalizar la ejecución del cuerpo del método objetivo.
- El aspecto se representa por la clase `ProfilingAspect` y el código del *advisor* está contenido en `ProfilingAdvisor.profile()`, el cuál utiliza la abstracción `InvocationContext`. La posición de la llamada a `proceed()` indica que el *advisor* es de tipo *around*.
- El *pointcut* consiste en todos los métodos anotados con `@AspectizedBy` que además obligan a especificar el aspecto (una clase anotada con `@Aspect`) que contiene la funcionalidad deseada.
- Finalmente se muestra el código objetivo al que será aplicado el aspecto de *profiling*: una versión reducida de la clase `ProductMapper` estudiada con anterioridad.

```java
@Aspect
public class ProfilingAspect
{
    @Advice
    public Object profile(InvocationContext context) throws Exception
    {
        long milliseconds = System.currentTimeMillis();

        Object result = context.proceed();

        milliseconds = System.currentTimeMillis() - milliseconds;
        System.out.println(milliseconds);

        return result;
    }
}
```

Figura 2.12: *Profiling* mediante AOP (II) (Java).

```java
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface AspectizedBy
{
    public Class<?> value();
}
```

```java
public class ProductMapper
{
    @AspectizedBy(ProfilingAspect.class)
    public <T> T get(Object id, Class<T> clazz)
    {
        /* Lookup code */
        return null;
    }

    @AspectizedBy(ProfilingAspect.class)
    public void delete(Object product)
    {
        /* Delete code */
    }
}
```

Figura 2.13: *Profiling* mediante AOP (III) (Java).

La única pieza que no se mostrará es el código que implementa el *aspect weaver*. Es evidente que el código de este componente dista de ser trivial, pues de deben recuperar los metadatos provistos por las anotaciones, instanciar un objeto del *advisor*, interceptar las llamadas en la lógica estructural (*pointcut*), crear el contexto de invocación para los métodos objetivo (`InvocationContext`) y un largo etc.

Existen *frameworks* y herramientas que permiten hacer uso de AOP en lenguajes y plataformas orientados a objetos y que proveen implementaciones muy elaboradas del *aspect weaver* en sus dos variantes: en tiempo de compilación o en tiempo de ejecución. De esta forma, el trabajo del desarrollador de aplicaciones consiste en crear sólo el codigo del *advisor* y definir el *pointcut*, tareas con una complejidad similar a los ejemplos mostrados en esta sección.

Algunas tecnologías como *Spring Framework* y *Enterprise JavaBeans* utilizan variantes del patrón *interceptor*[12] para proveer AOP. La limitante del patrón *interceptor* es que sólo permite utilizar métodos como *joinpoints*; en el caso de que se requiera utilizar *joinpoints* como constructores o propiedades, es necesario recurrir a una herramienta más sofisticada como el caso de *AspectJ*.

Los ejemplos ilustrados en esta sección, son una variante de implementación del mecanismo de intercepción de EJB; es probablemente la forma más fácil de utilizar AOP en un ambiente administrado (contenedor), por lo que suele llamarsele *lightweight AOP*.

Nótese que aún en ejemplos tan sencillos como los aquí expuestos, la reducción de la complejidad y la eliminación de código duplicado, representan una clara ventaja a favor de AOP; la flexibilidad del código resultante y la facilidad de agregar o remover comportamiento a otras partes de la aplicación, difícilmente se podría lograr sin el uso de aspectos.

Mediante variaciones en la implementación de AOP, se puede lograr una flexibilidad aún mayor si se externalizan los metadatos que definen los aspectos; guardar los metadatos en un archivo de configuración, por ejemplo mediante XML, permite agregar o eliminar aspectos en una gran cantidad de lugares, sin necesidad de recompilar un sólo componente de toda la base de código de la aplicación.

Demarcación de transacciones a nivel de código.

En esta sección se estudiarán dos patrones a nivel de código que se utilizan para establecer los límites de una transacción.

Control programático.

Este patrón proporciona un control explícito y bien definido de los límites de una transacción, que se agrupan en forma de una abstracción de servicio y se utilizan en la lógica de dominio, por el código que se encarga del control de los flujos de trabajo. La abstracción de servicio (véase la figura 2.14), tiene por lo general, tres comandos básicos:

- *begin*: marca el inicio de la transacción.

[12] Para una descripción detallada del patrón *interceptor* puede consultarse el libro *Pattern-Oriented Software Architecture*[Sch00]. Para *Java Enterprise Edition* se utiliza con frecuencia una variante conocida como *intercepting filter*, cuyo representante más conocido son los filtros de Servlets. Para más información sobre los patrones específicos de Java puede consultarse *Core J2EE Patterns*[Alu03].

- *commit*: marca el fin satisfactorio de la transacción y la permanencia de todos los cambios realizados.
- *rollback*: marca el fin insatisfactorio de la transacción y la restitución del estado previo al inicio de la transacción.

```
interface Transaction
{
    void begin();
    void commit();
    void rollback();
}
```

Figura 2.14: Abstracción de una transacción con control programático.

El control programático de transacciones es muy utilizado junto con código de gestión de excepciones, como se muestra en la figura 2.15.

```
Transaction transaction = null; // Create or obtain.

try
{
    transaction.begin();

    // Workflow code.

    transaction.commit(); // Save changes.
}
catch(Exception e)
{
    // Exception handling

    transaction.rollback(); // Undo changes.
}
```

Figura 2.15: Control programático de transacciones.

Control declarativo.

El control declarativo se basa en metadatos para establecer el contexto de la transacción, con una granularidad menor que la versión programática, y de una forma mucho menos invasiva.

Las transacciones declarativas por lo regular tienen una granularidad a nivel de método; la transacción inicia (*begin*) al mismo tiempo que la ejecución del método y finaliza satisfactoriamente (*commit*) junto con la devolución del control por parte del método. Cuando se genera una excepción que escapa

al código de gestión del cuerpo del método, la transacción se cosidera como finalizada insatisfactoriamente (*rollback*).

En el ejemplo 2.16 se muestra un fragmento de código que ilustra la utilización de anotaciones para brindar información sobre el contexto de la transacción en forma declarativa. En el ejemplo, el método `OrderProcessor.createOrder()` está marcado con una anotación `Transaction`; la anotación contiene metadatos que manejan el comportamiento de la transacción; en el ejemplo se puede ver que en la anotación se puede configurar el nivel de bloqueo de recursos (*isolation level*) del almacén de datos subyacente y las excepciones que, de no gestionarse en el cuertpo del método, ocasionan que la transacción se aborte.

```
public enum IsolationLevel
{
    READ_COMMITTED,
    READ_UNCOMMITED,
    SERIALIZABLE
}
```

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Transaction
{
    IsolationLevel isolationLevel() default IsolationLevel.READ_COMMITTED;
    Class<?>[] rollbackExceptions() default Exception.class;
}
```

```
public class OrderProcessor
{
    @Transaction(isolationLevel = SERIALIZABLE)
    public Order createOrder(List<Product> product)
    {
        /* ... */
    }
}
```

Figura 2.16: Control declarativo de transacciones.

Es necesario señalar que el comportamiento descrito anteriormente para el control declarativo no se logra símplemente anotando los métodos con `Transaction`; es necesario que una una rutina recupere los metadatos en tiempo de ejecución (presumiblemente mediante *reflection*), e intercepte la llamada al método objetivo, para posteriormente:

1. Iniciar la transacción (*begin*).
2. Ejecutar el código del cuerpo del método.

3. Si no hay exceptiones, completar la transacción (*commit*).

4. Si hay excepciones abortar la transacción (*rollback*).

El control declarativo de transacciones es exactamente el tipo de funcionalidad que se puede implementar mediante AOP: el método que ejecute los 4 pasos descritos es el *advisor*; el método `createOrder()` es un ejemplo particular de *joinpoint*; y el conjunto de métodos anotados con `@Transaction` son el *pointcut*.

Arquitectura de aplicaciones empresariales.

Agrupamiento lógico de funciones en capas.

En la disciplina de las redes de computadoras desde principios de la década de 1980, se creó un modelo conceptual que entonces ayudaba a comprender las funciones básicas involucradas en la comunicación entre dos nodos de una red. Dicho modelo, el modelo OSI, agrupaba las funciones en construcciones conceptuales llamadas capas[13], que se apilaban una sobre otra.

Una capa es un conjunto de funciones fuertemente relacionadas, que brinda servicios a la capa inmediatamente superior y utiliza los servicios de la capa inmediatamente inferior[Doh08]. Esta arquitectura de red basada en capas tiene las mismas ventajas que buscan algunos de los principios de diseño orientados que se tratarán posteriormente: alta cohesión y bajo acoplamiento!.

De una forma similar a las redes de computadoras, la arquitectura de aplicaciones empresariales se divide en capas. A diferencia de las 7 capas del modelo OSI, los sistemas de información no tienen definido un modelo estandarizado, ni en cuanto a las características comunes de la funcionalidad de una capa, ni en cuanto al número de éstas.

En esta sección se presentan los diversos componentes de las aplicaciones empresariales, utilizando un modelo basado en capas para descomponer los bloques de funcionalidad común presentes en la mayoría de los desarrollos.

Arquitectura de tres capas.

A pesar de que no existe un modelo estandarizado, como en el caso de las redes de computadoras, los desarrolladores de aplicaciones empresariales convienen en asumir la presencia de tres capas principales[Fow03]:

- *Data Access*: Que concentra la funcionalidad relacionada con las tareas de almacenamiento y recuperación de datos desde y hacia la aplicación empresarial.
- *Domain Logic*: Agrupa la funcionalidad relacionada con la organización de la lógica de dominio y la interfaces de servicio de la aplicación con todos sus clientes potenciales (interfaces de usuario, aplicaciones integradas, etc.).
- *Presentation*: Encargada de todos los servicios de visualización y *entrega* del conjunto de datos de la aplicación empresarial, que son relevantes para los usuarios finales.

En la figura 2.17 se presenta un marco de arquitectura de aplicaciones empresariales. En el resto de esta sección se abordan los diversos componentes que integran este modelo; también se detallará el lugar

[13] El término original en inglés es *layer*.

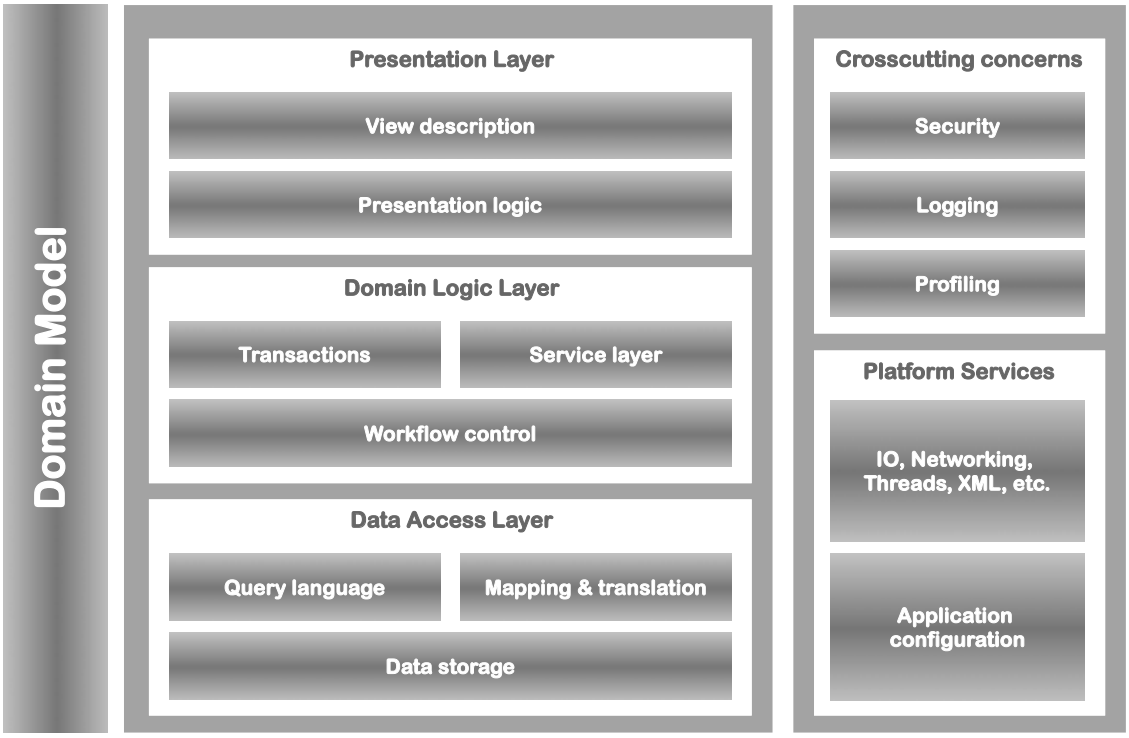


Figura 2.17: Estructura general de una aplicación empresarial.

que los servicios primarios, la organización de la lógica de dominio, los servicios complementarios y los servicios de la plataforma de desarrollo, ocupan en el contexto general de la arquitectura de tres capas.

Data Access Layer.

Desde el punto de vista de los servicios primarios, esta capa está fuertemente relacionada con los servicios de **persistencia**, **transaccciones** y en menor medida, **concurrencia**.

Si bien la mayor parte de las veces la demaración de una transacción se inicia desde la lógica de dominio, es el almacén de datos es el que finalmente tiene que crear los mecanismos para asegurar que se lleven a cabo operaciones donde se verifiquen las propiedades ACID, y son los objetos de la capa de acceso a datos los que ordenan la iniciación y terminación de una transacción.

Como se comentó en el capítulo anterior, muchos de los *cuellos de botella* de las aplicaciones empresariales tienen que ver con una estrategia inadecuada de acceso a memoria secundaria, particularmente los discos magnéticos. Es por ello que se considera a la capa de acceso a datos como un factor importante en el volumen máximo de concurrencia de un sistema de información. En esta capa se pueden encontrar algunos objetos que proveen mecanismos de carga anticipada, *caché*, *batch update*, etc., que ayudan a reducir las posibles contenciones de acceso a disco y que por lo tanto incrementan el rendimiento global de la aplicación.

Se considera parte de esta capa a todos los posibles almacenes de datos de la aplicación empresarial (**Data storage**); los representantes por excelencia son los motores de bases de datos relacionales, pero no se limitan a éstos.

En esta capa se ubican todos los objetos cuya responsabilidad sea intermediar entre el almacén de datos y la aplicación, ya sea administrando las conexiones, haciendo solicitudes de almacenamiento o recuperación, transformando la representación del almacén a la representación de la aplicación, etc. En caso de que la representación de la aplicación sea distinta de la del almacén de datos, en esta capa se ubican los (meta) datos que establecen las reglas de asociación y transformación entre las estructuras de representación en conflicto (**Mapping & transformation**).

Un elemento más consiste en los mecanismos de consulta del conjunto de datos del sistema de información. La existencia de representaciones distintas entre el almacén y la aplicación, sugiere la existencia de dos mecanismos de consulta diferentes (**Query language**). De la misma forma que con las representaciones de datos, los mecanismos de consulta requieren una traducción, o bien, dotar a la aplicación de una interfaz para realizar consultas en el lenguaje del almacén de datos. Para algunos tipos de almacenes de datos existen estándares de consulta, como es el caso de las bases de datos relacionales y el lenguaje SQL.

Domain Logic Layer.

En la capa de lógica de dominio se concentra la implementación de los servicios de transacciones, integración y concurrencia. En esta capa también se concetra parte de la lógica de dominio, de acuerdo con el patrón de organización seleccionadao.

- En el caso de *domain model*, en esta capa residirán los puntos de entrada para las aplicaciones integradas (**Service Layer**) y la orquestación del flujo de trabajo (**Workflow**).

- De otra forma, en esta capa se ubicarán todas las clases que contengan *transaction script*s y sólo se utilizarán los objetos del modelo de dominio para obtener y modificar datos.

Servicios de la plataforma.

Ya se han discutido algunos de los servicios que, desde el punto de vista de las aplicaciones empresariales se denominan servicios primarios, por su importancia capital en la implementación de los modernos sistemas de información.

Existe otra categoría de servicios que se pueden denominar como secundarios o complementarios, tal es el caso de:

- *Networking* de bajo nivel (sockets).
- Contenedores y algoritmos para manipular colecciones de objetos.
- Algoritmos criptográficos.
- Manipulación de texto y expresiones regulares.
- Procesamiento de XML.

- Gestion de concurrencia y *threads*.

Aunque estos servicios son de propósito general y muchas aplicaciones empresariales en efecto no los utilizan (al menos directamente), cuando éstos están presentes es imposible ubicarlos dentro de una sola capa; debido a su naturaleza, podrían usarse para la implementación de cualquier otro servicio de la aplicación y por lo tanto, en cualquier capa.

Lo mejor es que estén a disposición de los desarrolladores de aplicaciones, pues siempre existe la posibilidad de que en algún momento sean requeridos. Afortunadamente, muchos de estos servicios se implementan como parte de la biblioteca de clases base de las principales plataformas de desarrollo, tal es el caso de *Microsoft .NET Framework* y *Java Platform Standard Edition*.

Crosscutting functionality.

Anteriormente se discutió un ejemplo simplista sobre **profiling** para ilustrar el concepto de *crosscutting functionality* y su implementación mediante AOP. De forma completamente análoga, otros servicios como la **seguridad**, las **transacciones** y **logging** pueden encontrarse dispersos en prácticamente cualquier componente del sistema de información. Tal circunstancia hace imposible ubicar a estos servicios dentro de alguna de las tres capas principales.

Aún así, existe funcionalidad de la lógica de dominio, particularmente en el caso de las reglas de integridad referencial, que es considerada como *crosscutting functionality* y en este caso se puede ubicarse dentro de *domain logic*.

Presentation Layer.

La capa de presentación concentra la responsabilidad de brindar al usuario mecanismos para interactuar con el software, tareas tales como indicar el inicio de la ejecución de una tarea, introducir datos, solicitar datos, seleccionar datos de un conjunto, etc.

Desde el punto de vista del usuario final, la interfaz gráfica es la parte más importante del sistema de información, y en algunos casos, la única que importa. Debido a que la interfaz de usuario representa la frontera de interacción entre una persona y un programa de computadora, es en este componente donde se presentan algunas de las mayores dificultades de implementación.

La implementación de la capa de presentación generalmente se divide en dos etapas:

- Una primera etapa comprende la descripción de la vista, es decir, la disposición de los diversos elementos de interacción, como campos de texto, listas de selección, botones, etc., en una determinada área de la pantalla de un dispositivo de visualización.

- La segunda etapa comprende la lógica necesaria para manipular los elementos de la interfaz, es decir, el código que interviene para asociar un campo de texto con un atributo de una clase, o el código que responde a los distintos eventos generados por el usuario en su interacción con los elementos de la interfaz gráfica.

Históricamente, ambas etapas en la construcción de la interfaz gráfica de usuario se producen programáticamente. Uno de los problemas de las primeras aplicaciones construidas con herramientas de programación visual, tal es el caso de Borland Delphi o Microsoft Visual Basic, era que no había una separación entre la descripción de la vista y la lógica de manipulación de la interfaz.

Sin embargo, uno de los mayores problemas surgidos con el uso de estas herramientas era, no sólo la borrosa frontera entre la descripción y la lógica de la vista, sino que permitían introducir toda clase de lógica en los espacios destinados al manejo de eventos de la interfaz de usuario.

De esta forma, era posible insertar todo el código de acceso a datos en la rutina de control de eventos de un botón, lo que desde luego, tenía enormes desventajas desde el punto de vista de arquitectura. Debido a ello, en la actualidad, las herramientas y lenguajes utilizados para la construcción de interfaces gráficas de usuario promueve de forma enérgica, la separación entre la lógica de dominio, la lógica de manejo de la interfaz y la descripción de la vista.

Adobe Flex, Microsoft Windows Presentation Foundation e incluso Sun JavaFX, han introducido lenguajes declarativos para representar la descripción de la interfaz; tales lenguajes generalmente están basados en XML[14], o bien son lenguajes híbridos como JavaFX Script, donde es posible especificar la descripción de la vista de forma declarativa y al mismo tiempo, codificar las rutinas de manejo de eventos de forma programática.

La gran cantidad de innovaciones recientes en el campo de las interfaces de usuario y diseño de interacción con dispositivos digitales, permite observar la enorme importancia que la capa de presentación tiene, no sólo en el terreno de las aplicaciones empresariales, sino prácticamente en toda clase de software.

[14] Los ejemplos más notables en la actualidad son XAML en el caso de WPF y MXML en el caso de Flex.

Capítulo

3

Principios de diseño.

En esta sección se estudiarán algunos de los principios de diseño más comunes en el desarrollo de aplicaciones empresariales. Las tecnologías estudiadas más adelante, están basadas y promueven muchos de estos principios de diseño.

La aplicación de estos principios de diseño se considera fundamental, debido a que su aplicación generalmebte produce un incremento notable en la calidad del código fuente resultante, haciendo a éste fácilmente extensible, y reduciendo el costo del futuro mantenimiento.

En este trabajo, estos principios de diseño se estudian desde el punto de vista del paradigma de programación orientada a objetos. Entender estos principios es el último paso antes de comenzar a estudiar algunas de las decisiones de implementación que se utilizaron en el sistema Restomatic, particuarmente desde el punto de vista de las tecnologías.

Principio SLR (Single Responsibility Principle).

En el diseño de un sistema de información se busca modelar una abstracción del mundo real, con suficiente detalle como para representar el problema que se quiere resolver. Estas abstracciones resultan eventualmente en clases, que constan de estado y de comportamiento.

Añadir funcionalidad muy variada a una clase, tiene efectos negativos al momento de realizar cambios en el comportamiento.

En el ejemplo 3.1 se tiene una clase que representa la abstracción de una orden en un sistema de comercio electrónico. Esta clase contiene sólo dos variables de estado y cuatro métodos.

Una clase tiene una responsabilidad cuando las acciones que integran su comportamiento, guardan una relación cercana entre sí. Este no es el caso con los tres últimos métodos, pues hacen responsable a la clase de tres actividades más, que nada tienen que ver entre sí:

- Registrarse ante el sistema de procesamiento de órdenes.
- Almacenarse en un medio persistente.

```
class Order
{
    public Item[] Items { get; set; }
    public decimal Total { get; set; }

    public void ClearItems() { /* .. */ }

    public void Place() { /* .. */ }
    public void Save() { /* ... */ }
    public void Render() { /* ... */ }
}
```

Figura 3.1: Diseño de clase con varias responsabilidades.

- Desplegarse en una interfaz de usuario.

Cada responsabilidad asignada a una clase, es potencialmente un eje de cambio:

- Si se actualiza el procedimiento para registrar las ordenes, será necesario abrir la clase para reflejar el nuevo procedimiento.
- Hacer a la clase responsable de su propia persistencia, limita la reutilización de la clase en escenarios donde no se tenga disponible, o no sea necesario hacer persistente el estado de la orden.
- El método de visualización es quizá el problema más grande, pues restringe a la orden a una sola forma de representación, quizá contemplando un único dispositivo o tecnología de visualización.

En este sencillo ejemplo, se pueden delegar las responsabilidades de procesamiento de órdenes, persistencia y visualización a clases secundarias. El principio de única responsabilidad intenta disminuir la frecuencia de cambio asignando a una clase una única razón para cambiar, de ahí el nombre.

El principio SRP, o principio de única responsabilidad, fue un concepto ideado a finales de la década de 1970, y se conoce con el nombre alternativo de **cohesión**[DeM79].

Una forma alternativa de distribuir las responsabilidades, de acuerdo con el principio de única responsabilidad puede observarse en el ejemplo 3.2.

Principio OPC (Open/Closed Principle).

Cualquier funcionalidad entregada por una aplicación empresarial, requiere de la interacción entre varios componentes para lograr un objetivo específico. Cuando un componente consume la funcionalidad de otro se dice que el primero es un cliente del servicio, y el segundo es un proveedor de servicio.

En general, muchos de los servicios ofrecidos en una aplicación empresarial tienden a cambiar conforme transcurre el tiempo. Cuando un cliente utiliza una implementación concreta de un servicio, está sujeta a las variaciones de esta implementación, y aún más, dificulta la sustitución del proveedor de servicio, obligando al cliente a cambiar todas las referencias y llamadas a la interfaz pública del proveedor.

El principio OPC es una herramienta útil para flexibilizar el código y disminuir significativamente el impacto del cambio sobre la implementación de un servicio. El principio OPC se le atribuye por primera vez a Bertrand Meyer en su libro *Object oriented software construction* en 1988, donde establece que todo artefacto de software debe ser:

- Abierto para extensión (*Open for extension*): debe ser posible modificar el comportamiento del artefacto, utilizando puntos de extensión.
- Cerrado para modificación (*Closed for modification*): las modificaciones en el comportamiento no deben implicar cambios en el código existente del artefacto.

Los puntos de extensión: programar contra abstracciones.

Un caso común donde se puede apreciar el impacto de programar contra abstracción de un servicio, es la persistencia del estado de una aplicación. Para un objeto determinado del dominio, normalmente existe otra clase cuya responsabilidad es guardar el estado del objeto y recuperarlo del almacén de datos.

```
class Order
{
    public Item[] Items { get; set; }
    public decimal Total { get; set; }

    public void ClearItems() { /* .. */ }
}

class OrderProcessor
{
    public void Place(Order order) { /* .. */ }
    /* Other methods */
}

class OrderRepository
{
    public void Save(Order order) { /* ... */ }
    /* Other methods */
}

class OrderRenderer
{
    public void Render(Order order) { /* ... */ }
    /* Other methods */
}
```

Figura 3.2: Diseño de clase con responsabilid única.

Los efectos negativos de programar contra una implementación se pueden apreciar en el ejemplo 3.3.

La clase `CustomerApp` es el cliente y la clase `CustomerDbRepository` es el proveedor de servicio. El cliente conoce detalles de implementación de su proveedor de servicios: si el servicio falla, el cliente debe capturar una excepción de tipo `SQLException`[1].

Debido a que el cliente está atado a los detalles de una implementación concreta del proveedor de servicio, no puede utilizar el mismo código en presencia de un proveedor distinto, por ejemplo el que se ilustra en la figura 3.4.

Casi siempre es más conveniente modelar al servicio en un nivel de abstracción mayor, de forma que el cliente no tenga la necesidad de conocer los detalles de implementación del proveedor de servicio; así, en el ejemplo, al cliente debería serle indistinto si el almacén de datos es el sistema de archivos o una base de datos relacional, lo único que realmente importa es almacenar y recuperar el estado de un objeto utilizando el servicio. Es responsabilidad del proveedor encargarse de los detalles específicos del

[1] Si el cliente requiere cambiar de proveedor de persistencia, entonces deberá cambiar por completo la rutina de manejo de excepciones. En este ejemplo las excepciones se utilizaron para ilustrar el problema, pero no son de ninguna forma, el único síntoma de que el cliente está fuertemente acoplado con su proveedor de servicios.

```
import java.sql.SQLException;

public class CustomerDbRepository
{
    public void save(Customer customer) throws SQLException
    {
        /* Database specific implementation */
    }
}
```

```
import java.sql.SQLException;

public class CustomerApp
{
    private CustomerDbRepository repository;

    public void setRepository(CustomerDbRepository repository)
    {
        this.repository = repository;
    }

    public void run()
    {
        Customer customer = new Customer();

        try
        {
            repository.save(customer);
        }
        catch(SQLException e) { }
    }
}
```

Figura 3.3: Programando contra la implementación.

almacén de datos.

Cuando el cliente se programa contra una abstracción lo único que importa es el trabajo que el proveedor efectúa para el cliente. De esta forma, el cliente puede permanecer invariante si posteriormente se necesita cambiar de proveedor de servicio.

Algunos lenguajes de programación modernos, permiten modelar proveedores de servicios utilizando construcciones abstractas, conocidas como interfaces[2]. Una interfaz consiste en un agrupamiento lógi-

[2]En lenguajes como Java o C#, las interfaces se crean utilizando una palara reservada `interface`, aunque

```
import java.io.IOException;

public class CustomerIoRepository
{
    public void save(Customer customer) throws IOException
    {
        /* File system specific implementation */
    }
}
```

Figura 3.4: Programando contra la implementación (II).

co de definiciones de métodos que actúan como un contrato entre el cliente y el proveedor de servicio [Low05].

En la figuras 3.5 y 3.6 se pueden apreciar los cambios necesarios, tanto en el cliente, como en el proveedor de servicios de forma que se aplique el principio OPC.

Al programar utilizando interfases, se aisla a los clientes del posible cambio en las implementaciones de los proveedores (cerrado), así como la sustitución completa de un proveedor por otro (abierto), muchas veces sin necesidad de recompilar.

Cuando un cliente consume un servicio utilizando para ello una abstracción, se dice que entre ellos hay *bajo acomplamiento*. Esta característica hace al código más flexible al cambio, y permite reutilizar algunas porciones de código en la construcción de otros sistemas de información[3].

Contratos de servicio.

Debido a que las interfaces son abstracciones, tanto el cliente como los proveedores deben estar de acuerdo sobre el efecto que produce el servicio; a ésto se le conoce como *contrato de servicio*.

Los dos elementos básicos del contrato del servicio son:

- Las precondiciones. Que son aquellas condiciones que se deben satisfacer con anterioridad a que el cliente consuma el servicio.

- Las postcondiciones. Que son aquellas condiciones que se deben satisfacer posteriormente a que el proveedor brinde el servicio.

también es posible programar contra una interfaz utilizando clases abstractas, por medio de la palabra reservada `abstract`. En lenaguajes como C++, donde no existe un equivalente, se puede hacer uso de una clase, con un bloque de visibilidad pública que contenga únicamente funciones virtuales puras (`function(...) = 0;`). Es decir, las interfaces son conceptos que trascienden las construcciones de un lenguaje de programación específico.

[3] Un artículo clásico sobre el principio OPC se publicó en 1996 por Robert C. Martin bajo el nombre de *The Open Closed Principle*; en este artículo se expone una jerarquía de clases relacionadas con figuras geométricas (`Shape`, `Circle`, etc.) que aún hoy en día se utiliza como ejemplo para exponer el concepto de polimorfismo en cursos básicos de programación orientada a objetos.

```
public interface ICustomerRepository
{
    void save(Customer customer) throws PersistenceException;
}
```

```
public class CustomerDbRepository implements ICustomerRepository
{
    public void save(Customer customer) throws PersistenceException
    {
        /* Database specific implementation */
    }
}
```

```
public class CustomerIoRepository implements ICustomerRepository
{
    public void save(Customer customer) throws PersistenceException
    {
        /* File system specific implementation */
    }
}
```

Figura 3.5: Programando contra la interfaz (Proveedores de servicio).

Si las precondiciones no son satisfechas por el cliente, entonces el proveedor muy probablemente será incapaz de llevar a cabo el servicio. Si las postcondiciones no son satisfechas, entonces el servicio se llevó a cabo de forma incorrecta. Una relación importante entre las interfaces y las implementaciones de un servicio es que, las últimas pueden tener precondiciones menos estrictas y postcondiciones más estrictas[Mey97].

En algunos lenguajes de programación, la sintaxis de la interfaz ayuda a documentar el contrato de servicio. En al caso de los lenguajes con verificación estática de tipos, la interfaz de servicio puede limitar los tipos de datos sobre los que puede operar; aún más, en casos como el lenguaje Java, la sintaxis de la interfaz puede documentar incluso, las condiciones de excepción que el cliente puede esperar al solicitar el servicio.

El la figura 3.7 se muestra una implementación de un algoritmo cuadrático de ordenamiento de colecciones implementado utilizando Python.

Los puntos clave en este ejemplo corresponden a las líneas 1 y 6. Debido a que Python es un lenguaje interpretado sin verificación estática de tipos, la interfaz de servicio (en este caso, la definición de función en la línea 1) debe indicar a través de su documentación, los tipos de datos válidos sobre los que opera. En esta definición, el primer parámetro debe ser de tipo lista, de otra forma, todos los lugares en la función donde se hace uso del indizador de listas provocarán un error en tiempo de ejecución; el segundo

```
public class CustomerApp
{
    private ICustomerRepository repository;

    public void setRepository(ICustomerRepository repository)
    {
        this.repository = repository;
    }

    public void run()
    {
        Customer customer = new Customer();

        try
        {
            repository.save(customer);
        }
        catch(PersistenceException e) { }
    }
}
```

Figura 3.6: Programando contra la interfaz (Cliente).

```
 1 def sort(collection, comparator):
 2     n = len(collection)
 3
 4     for i in xrange(n):
 5         for j in xrange(i + 1, n, 1):
 6             if comparator(collection[i], collection[j]) > 0:
 7                 element = collection[i]
 8
 9                 collection[i] = collection[j]
10                 collection[j] = element
```

Figura 3.7: Algoritmo de ordenamiento (Python).

parámetro es una referencia a otra función, que en este caso se utiliza para comparar los elementos de la lista. La documentación será extensa, pues `sort` además de la detallar su contrato, deberá incluir la descripción del contrato de la función de comparación `comparator`.

Los lenguajes con verificación estática de tipos, hacen más claro el contrato y menos extensa la documentación. En Java, por ejemplo, se puede hacer mucho más claro el método de comparación si se diseña un contrato específico para esto, como se muestra en la figura 3.8.

```
public interface Comparator<T>
{
    int compare(T left, T right) throws IllegalArgumentException;
}
```

Figura 3.8: Servicio de comparación (Java).

Independientemente del lenguaje de programación utilizado, se debe documentar de alguna forma el contrato de servicio de forma que:

- El cliente pueda determinar la utilidad del servicio.
- El proveedor tenga información sobre la funcionalidad que debe brindar el servicio.
- El cliente sea informado cuando el servicio no pueda llevarse a cabo.
- El proveedor informe al cliente de condiciones de fallo (sin exponer detalles de implementación).
- El cliente pueda realizar pruebas para verificar el funcionamiento del servicio (verificar las post-condiciones).
- El proveedor pueda verificar las precondiciones.

Principio DRY (Don't Repeat Yourself).

El principio DRY tiene que ver con la correcta ubicación de datos y responsabilidades en una aplicación. Se trata de evitar en la medida de lo posible, la duplicación de código.

La duplicación de código se evita factorizando aquellos elementos comunes y colocándolos en una sola ubicación. De esta forma, cualquier cambio que involucre a este código deberá ejecutarse y probarse en un solo lugar; dicho principio fomenta la reutilización y aumenta la flexibilidad del código en cuestión.

Ejemplo de duplicidad de comportamiento en Java EE.

Una tarea fundamental en el desarrollo de aplicaciones empresariales consiste en la verificación de los objetos del modelo de dominio. Para atender este problema se han diseñado una gran cantidad de soluciones, como las siguientes:

- La tecnología de presentación en web JavaServer Faces, incluye la validación como un paso de su ciclo de petición - respuesta por medio de la interfaz `javax.faces.validator.Validator`.
- El framework de inyección de dependencias Spring, permite realizar validación de objetos de dominio, por medio de la interfaz `org.springframework.validation.Validator`.
- El proveedor de persistencia Hibernate posee dentro de su conjunto de herramientas, Hibernate Validator, que permite la verificación de objetos del dominio a través de anotaciones.

Si una aplicación hace uso de las tres tecnologías descritas anteriormente, muy probablemente se encuentre duplicando mucho del código de validación, desde la persistencia hasta la interfaz de usuario.

Algunos riesgos de tener duplicado el código de validación en los objetos de dominio se pueden observar en secuencia:

1. Toda regla de validación para un objeto determinado tendría que implementarse en tres ubicaciones distintas.

2. Si llegase a haber alguna modificación en la regla de validación, el cambio tendría que sincronizarse en cada una de las implementaciones del objeto que ejecuta la validación, de forma que no se generen varias reglas.

3. Si existen distintas reglas dependiendo del API de validación, esto puede provocar que un objeto pase la validación frente a una implementación, pero falle ante otra. Esto generaría reglas de validación inconsistentes.

4. Si el criterio de validación es inconsistente y se utilizan dos implementaciones sobre un mismo objeto en una misma pila de llamadas, es posible que la discrepancia en la regla oculte defectos de código difíciles de depurar, sólo visibles como fallas en tiempo de ejecución.

Si en lugar de requerir distintas implementaciones, el comportamiento de validación se pudiese encapsular en una sola ubicación sensible, entonces todos los problemas descritos serían perfectamente evitables.

Es sorprendente que a pesar de que el principio DRY data de hace cerca de 10 años[Hun99], en la plataforma Java, bastantes servicios comunes se encuentran dispersos en una multitud de lugares, siendo la validación sólo uno de muchos ejemplos[4].

Principio LSP (Liskov Substitution Principle).

Uno de los mecanismos de reutilización que son posibles en la programación orientada a objetos, consiste en la herencia. El principio LSP está precisamente relacionado con la herencia y se utiliza en el diseño de jerarquías de clase.

El principio LSP fue propuesto por Barbara Liskov en 1988 en un artículo titulado *Data Abstraction and Hierarchy*. La idea fundamental de este principio[5] es que para una jerarquía de clase, cualquier código que esté diseñado para operar sobre una instancia de una clase base B, debe operar correctamente en presencia de una instancia de una clase derivada D.

Una violación del principio LSP deja de manifiesto un error en el diseño de una jerarquía de clase. Una de las razones por las que existe la herencia es para extender la funcionalidad de un tipo existente, de forma que el conjunto de datos y comportamiento de una clase base B, es un subconjunto de cualquier

[4] Tras casi 10 años de la aparición del libro *The Pragmatic Programmer*, existe un esfuerzo de estandarización de un framework de validación para la plataforma Java, conocido JSR-303 Bean Validation, para más información se recomienda visitar el sitio `http://jcp.org/en/jsr/detail?id=303`.

[5] El enunciado original es bastante más preciso: *"What is wanted here is something like the following substitution property: if for each object o_1 of type S there is an object o_2 of type T such that all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2 then S is a subtype of T "*.

especialización o clase derivada *D*.

Violación evidente del principio de sustitución de Liskov.

Algunos lenguajes de programación con verificación estática de tipos, previenen al programador de violaciones evidentes al principio LSP.

```
public abstract class AbstractButton
{
    public abstract void doClick();
}
```

```
public class ToggleButton extends AbstractButton
{
    protected void doClick() { /* Method body */ }
}
```

Figura 3.9: Jerarquía de clases - violación de LSP (Java).

El código del ejemplo 3.9 produce un error de compilación. La razón es que en Java, un tipo derivado en una jerarquía de clase no puede reescribir un método de su clase base, proporcionando un modificador de acceso más restrictivo[6]. En este caso, el compilador previene al programador que accidental o intencionalmente se viole el principio de sustitución de Liskov.

Violación no evidente al principio LSP.

Uno de los ejemplos más utilizados para enseñar el concepto de herencia consiste en la utilización de la jerarquía entre dos figuras geométricas conocidas: el rectángulo y el cuadrado. En esta jerarquía, es natural modelar al cuadrado como un caso particular de rectángulo, donde tanto el ancho como el largo tienen la misma dimensión. Esta jerarquía se puede ver en la figura 3.10.

Aún cuando para la mayoría de los casos, esta jerarquía de herencia produce resultados deseables, el ejemplo 3.11 hace visible la violación al principio LSP.

Tal demostración[7] sugiere bastante cuidado al diseñar jerarquías de clase, pues existen casos de frontera donde al utilizar un tipo derivado en sustitución de su clase base, puede producir condiciones de excepción, siendo estos errores súmamente difíciles de depurar.

[6] En otros lenguajes como C#, es imposible reescribir un método cambiando el modificador de acceso original especificado en la clase base.

[7] La idea original del ejemplo, así como una explicación más detallada se pueden consultar en [Mar07].

```
class Rectangle
{
    public virtual decimal Width { get; set; }
    public virtual decimal Height { get; set; }

    public decimal Area { get { return Width * Height; } }
}
```

```
class Square : Rectangle
{
    public override decimal Height
    {
        set { base.Height = base.Width = value; }
    }

    public override decimal Width
    {
        set { base.Width = base.Height = value; }
    }
}
```

Figura 3.10: Jerarquía de clases - violación de LSP (C#).

```
class RectangleTest
{
    public void Test(Rectangle rectangle)
    {
        rectangle.Height = 20;
        rectangle.Width = 100;

        Debug.Assert(rectangle.Area == 20D * 100D);
    }
}
```

Figura 3.11: Comprobación de la violación al LSP (C#).

Principio IoC (Inversion of Control).

Dos de los principios que se han tratado hasta ahora, el principio OPC y LSP sugieren una preferencia sobre el uso de abstracciones, como regla general en los puntos donde un componente tiene dependencia con otro.

Aún y cuando programar contra interfaces tiene grandes beneficios, existe a menos un punto dentro del código del programa donde necesariamente se requiere utilizar clases concretas: la instanciación.

```
 1 public class OrderProcessor
 2 {
 3     private ICustomerRepository customerRepository;
 4
 5     public void setCustomerRepository(ICustomerRepository
customerRepository)
 6     {
 7         this.customerRepository = customerRepository;
 8     }
 9
10     public OrderProcessor()
11     {
12         setCustomerRepository(new CustomerDbRepository());
13     }
14
15     /* Other methods */
16 }
```

Figura 3.12: Dependencia sobre clase concreta (Java).

En el ejemplo 3.12 : línea 12, es evidente que aún cuando las dependencias se almacenan en referencias a abstracciones (`ICustomerRepository`), en el constructor de la clase se tiene una dependencia a una clase concreta (`CustomerDbRepository`); este código no está cerrado contra modificaciones.

Este problema tiene soluciones particularmente interesantes, que se basan en el principio SRP; de acuerdo con SRP, es posible tomar la responsabilidad de crear los proveedores de servicio, encapsularla en otra abstracción y utilizar ésta para ligar las instancias de los proveedores de servicio con los clientes.

Solución #1: *Simple Factory.*

Aunque no es estrictamente un patrón de diseño[Fre04], *Simple Factory* es una forma de encapsular la responsabilidad de la creación de objetos en una clase externa al cliente.

Si se observa el ejemplo 3.13 se ve que el cliente ya no depende en una implementación concreta del proveedor de servicios; aparentemente, el problema sólo se ha movido de lugar, a la clase `CustomerRepositoryFactory`

[8]La simplicidad de este ejemplo quizá no permita apreciar la ventaja de *Simple Factory* sobre una instancia creada a través de `new`. Pero existen otros lugares, como el paquete `javax.swing` de la plataforma Java, donde dicha ventaja es más clara: un ejemplo es `Border`(su extensa jerarquia) y `BorderFactory`.

```
public class CustomerRepositoryFactory
{
    public static ICustomerRepository getDefaultInstance()
    {
        return new CustomerDbRepository();
    }
}
```

```
public class OrderProcessor
{
    private ICustomerRepository customerRepository;

    public void setCustomerRepository(ICustomerRepository
customerRepository)
    {
        this.customerRepository = customerRepository;
    }

    public OrderProcessor()
    {
        setCustomerRepository(CustomerRepositoryFactory.getDefaultInstance());
    }

    /* Other methods */
}
```

Figura 3.13: Dependencia sobre un Simple Factory (Java).

Solución #2: Registros globales.

Las soluciones basadas en registros implican, de una forma u otra, objetos de ámbito global dentro de algún contexto de ejecución[9].

Para el caso de objetos *singleton*, existe un momento determinado, usualmente el inicio de la aplicación, donde se crean los objetos globales y se registran. Posteriormente se utiliza el API del registro u otro patrón de adquisición como *Service Locator*[Alu03] para recuperar los objetos cuando estos se requieran.

En el ejemplo 3.14 se puede ver una implementación trivial de un registro. Para evitar que sea el registro el que esté fuertemente acoplado al proveedor de servicios, la información referente al tipo concreto se puede externalizar en un archivo de configuración y después utilizar ésta para crear la instancia en tiempo de ejecución, utilizando *reflection*.

[9]Ejemplos de contextos comunes son los *threads*, los *application domains*(.NET), las sesiones dentro de una aplicación web, etc.

```
import java.util.*;

public class Registry
{
    private static Map<Class<?>, String> defaultImpl;
    private static Map<Class<?>, Object> defaultSing;

    static
    {
        defaultImpl = new HashMap<Class<?>, String>();
        defaultSing = new HashMap<Class<?>, Object>();

        /* Init configuration should be externalized */
        defaultImpl.put(ICustomerRepository.class, "CustomerDbRepository");
    }

    public static <T> T get(Class<T> clazz) throws Exception
    {
        if (!defaultSing.containsKey(clazz))
        {

            String typeName = defaultImpl.get(clazz);
            Class<?> targetType = Class.forName(typeName);

            register(clazz, targetType.newInstance());
        }

        return (T) defaultSing.get(clazz);
    }

    public static void register(Class<?> iface, Object impl)
    {
        if (!defaultSing.containsKey(iface))
            defaultSing.put(iface, impl);
    }
}
```

Figura 3.14: Implementación simple de un registro (Java).

El siguiente código ilustra el cambio que debería aplicarse al ejemplo 3.12 : línea 12, de forma que obtenga sus dependencias utilizando un registro:

```
ICustomerRepository repository = Registry.get(ICustomerRepository.class);
```

Aunque esta solución resulta ser más adecuada que los *factories*, trabajar con registros puede resultar poco atractivo para algunos desarrolladores[Fow03].

En particular, esta solución no permite que se utilicen de forma simultánea, más de un proveedor para una determinada abstracción de servicio; sin embargo, este código está bastante mejor cerrado contra modificaciones, puesto que el cambio de proveedores se puede efectuar de forma trivial, directamente en la configuración, sin necesidad de recompilar el código.

Solución #3: Inyección de dependencias.

Puede considerarse como una responsabilidad innecesaria el hecho de que un objeto tenga que recuperar sus propias dependencias, como es el caso del código que utiliza *Simple Factory* y *Registry*; utilizando de nuevo SRP, se puede abstaer otra entidad cuya responsabilidad sea *conocer* las dependencias entre un conjunto de clases e *introducirlas* cuando sea requerido, en base a una serie de *reglas* flexibles.

El punto clave a notar en esta tercera solución, es que a diferencia de las dos anteriores, el cliente no tiene que buscar sus dependencias; es el entorno, quien toma al cliente y le provee de éstas, y de ahí el nombre de inyección de dependencias[10].

Para utilizar inyección de dependencias, el objeto sólamente declara sus dependencias (utilizando abstracciones) y un mecanismo externo se encarga de proveerlas en tiempo de ejecución[11], tal como se aprecia en el ejemplo 3.15.

```
public class OrderProcessor
{
    private ICustomerRepository customerRepository;

    public void setCustomerRepository(ICustomerRepository
customerRepository)
    {
        this.customerRepository = customerRepository;
    }

    /* Other methods */
}
```

Figura 3.15: Clase preparada para DI (Java).

El proceso de construir los objetos y proporcionarle sus dependencias se conoce como *wiring*; a pesar de que puede hacerse desde un bloque muy externo de la aplicación, por ejemplo el punto de entrada del programa, usualmente lo realiza un subsistema conocido como el contenedor de IoC, utilizando configuración externa y *reflection*, de forma parecida al ejemplo 3.14.

[10] La inyección de dependencias (DI, *dependency injection*), es un caso particular dentro de otro concepto más general denominado Inversión de Control. El nombre de DI fue sugerido inicialmente por Martin Fowler en su artículo *Inversion of Control Containers and the Dependency Injection pattern*, que puede consultarse en la dirección `http://www.martinfowler.com/articles/injection.html`.

[11] Este mecanismo externo muchas veces se implementa en forma de un *contenedor*IoC, como es el caso de PicoContainer, Spring, Google Guice, etc.; pero éste no es de ninguna manera es un requisito para la inyección de dependencias.

El contenedor de IoC: The Spring Framework.

Spring es uno de los muchos contenedores IoC que existen[12], en donde la configuración de las dependencias se almacena de forma externa a la aplicación mediante un archivo XML. El ejemplo 3.16 ilustra un ejemplo de dicho archivo de configuración.

```
<beans>
    <bean id="customerRepository" class="CustomerDbRepository"/>

    <bean id="orderProcessor" class="OrderProcessor">
        <property name="customerRepository" ref="customerRepository"/>|
    <bean>
<beans>
```

Figura 3.16: Archivo de configuración de Spring.

Desde luego, aún existirá dentro del código de la aplicación un lugar para instanciar el contenedor y un API para solicitar los objetos configurados dentro del contenedor[13]. Dichos objetos, al ser recuperados del contenedor, tendrán configuradas todas sus dependencias, independientemente del tamaño final del grafo de objetos. En ocasiones, el contenedor es lo suficientemente *inteligente* como para detectar referencias cíclicas, o bien, dependencias con tipos de datos incompatibles, entre muchos otros mecanismos de seguridad.

El contenedor de IoC: Google Guice.

En el caso de Spring, la configuración de las dependencias y de los objetos en el contenedor se realiza por medio de un archivo XML, de forma completamente declarativa. XML es un formato fácil de producir y consumir por las herramientas de los entornos de desarrollo; sin embargo, también es cierto que tales herramientas no pueden verificar la validez de los tipos de datos entre las dependencias, antes de la ejecución del programa. Aún más, tal vez la principal desventaja de la configuración en XML sea su fragilidad ante la refactorización (la dificultad de sincronizar los cambios hechos en el código).

Algunos contenedores como Guice utilizan al extremo las características del lenguaje de programación Java(especialmente los tipos genéricos y las anotaciones), para brindar un tipo de configuración de carácter programático[14], como se aprecia en el ejemplo 3.17. Este tipo de configuración puede verificarse estáticamente por el compilador y es extremadamente fácil de refactorizar.

Por supuesto que la configuración programática no está exenta de tener desventajas, pues una vez más, los cambios en la implementación de un servicio requieren recompilación.

[12] Aunque DI es sólo una de las muchas cosas que se obtienen al utilizar Spring.

[13] Este proceso se conoce como *bootstrapping*.

[14] Spring de hecho también provee configuración declarativa en base a anotaciones como `@Autowire`; de hecho la especificación de la configuración está desacoplada del resto del contenedor de Spring.

```
import com.google.inject.*;

public class OrderProcessor
{
    private ICustomerRepository customerRepository;

    @Inject
    public void setCustomerRepository(ICustomerRepository
customerRepository)
    {
        this.customerRepository = customerRepository;
    }

    /* Other methods */
}
```

```
import com.google.inject.*;

public class CustomerRepositoryModule extends AbstractModule
{
    @Override
    protected void configure()
    {
        bind(ICustomerRepository.class)
        .to(CustomerDbRepository.class);
    }
}
```

Figura 3.17: Configuración de inyección de dependencias con Guice.

Puntos de extensión y ventajas de DI.

Ninguna de las soluciones aquí descritas están necesariamente limitadas a la *creación de instancias*; debido a que son puntos centrales para la obtención de objetos y servicios, su responsabilidad puede extenderse para abarcar un concepto más genérico llamado *administración de instancias*. Es aquí donde se pueden incluir características como la cantidad de proveedores (*singleton*, *instance pool*, nueva instancia por lamada, etc.), control sobre el ámbito de la instancia, dependiendo del contexto de ejecución (*thread*, global, por sesión, por petición, etc.) y manejo del ciclo de vida (creación, eliminación, salida de memoria primaria, etc.).

En cuanto a las soluciones estudiadas, es claro que la inyección de dependencias es el único enfoque no invasivo, en el sentido de que los clientes no tienen dependencia ni siquiera en aquellas clases auxiliares, que de otra forma serían necesarias para buscar sus dependencias.

Las dos formas de configurar la inyección de dependencias tienen ventajas y desventajas muy impor-

tantes; sin embargo, entre la configuración declarativa en XML y la configuración declarativa/programática que se logra utilizando anotaciones, se puede apreciar que la primera es mucho menos invasiva que la segunda, en el sentido de que los clientes ni siquiera necesitan conocer las clases (y anotaciones) de su contenedor de IoC, es decir, están desacoplados del mismo contenedor.

En suma, la inversión de control y particularmente la inyección de dependencias, permiten programar código mucho más limpio y desacoplado que otras alternativas como *factory* y *registry*, con muy pocas desventajas añadidas.

Parte II

Implementación de la aplicación Restomatic.

Capítulo

4

Panorama general de implementación.

En este capítulo se introducirá la aplicación de prueba *Restomatic*, así como algunas de las decisiones de diseño más importantes, la elección de las tecnologías y algunos de los problemas más sobresaientes en la implementación de dicha aplicación de prueba.

La aplicación *Restomatic* es una aplicación empresarial de gestión de carta digital para restaurantes, acotada a un subconjunto de los requerimientos solicitados para una aplicación de estas características, pero suficientemente extensa como para utilizar muchos de los fundamentos de arquitectura y principios de diseño introducidos en la primera parte de este trabajo.

Estructura general de la aplicación.

Restomatic, es una aplicación cliente-servidor diseñada para interacción con múltples interfaces de usuario a través de una capa de servicios. En base a los requerimientos, se dividió lógicamente la aplicación en 4 componentes estructurales de alto nivel:

- Terminales móviles.
- Terminal de facturación.
- Terminal de monitoreo y notificación.
- Servidor de aplicaciones.

La estructura física de la aplicación puede apreciarse en la figura 4.1. La especificación de requerimientos puede consultarse directamente en el apendice A.

La implementación de la aplicación se dividió lógicamente en los siguientes subproyectos:

- RestomaticDB: Que contiene los scripts de creación del esquema de base de datos. El código fuente del esquema de base de datos se encuentra disponible en el apéndice C.
- RestomaticServer: Que contiene la funcionalidad del **servidor de aplicaciones**. El código fuente del servidor de aplicaciones se muestra parcialmente en los apéndices B, D, E y F.
- RestomaticPocket: Que contiene la funcionalidad de las **terminales móviles**.
- RestomaticRichClient: Que contiene la funcionalidad tanto de la **terminal de monioreo** como de la **terminal de facturación**[1].

Patrones de organización de la aplicación en `RestomaticServer`.

Para esta parte del sistema, se siguió de forma rigurosa la arquitectura de tres capas comentada en la primera parte de este trabajo. En cuanto a los principios de diseño, existe una violación evidente al principio DRY en la implementación de los Web Services; sin embargo, dicha decisión se tomó de forma razonada para facilitar la inclusión de servicios de autenticación y autorización, que de cualquier forma hubiese requerido la creación de una capa de abstracción encima de la lógica de dominio.

[1] Debido a un incremento no estimado en la complejidad de las interfaces de usuario, la funcionalidad de ambas terminales no se concluyó y por lo tanto, el código relativo a dicho proyecto no se incluye en los apéndices

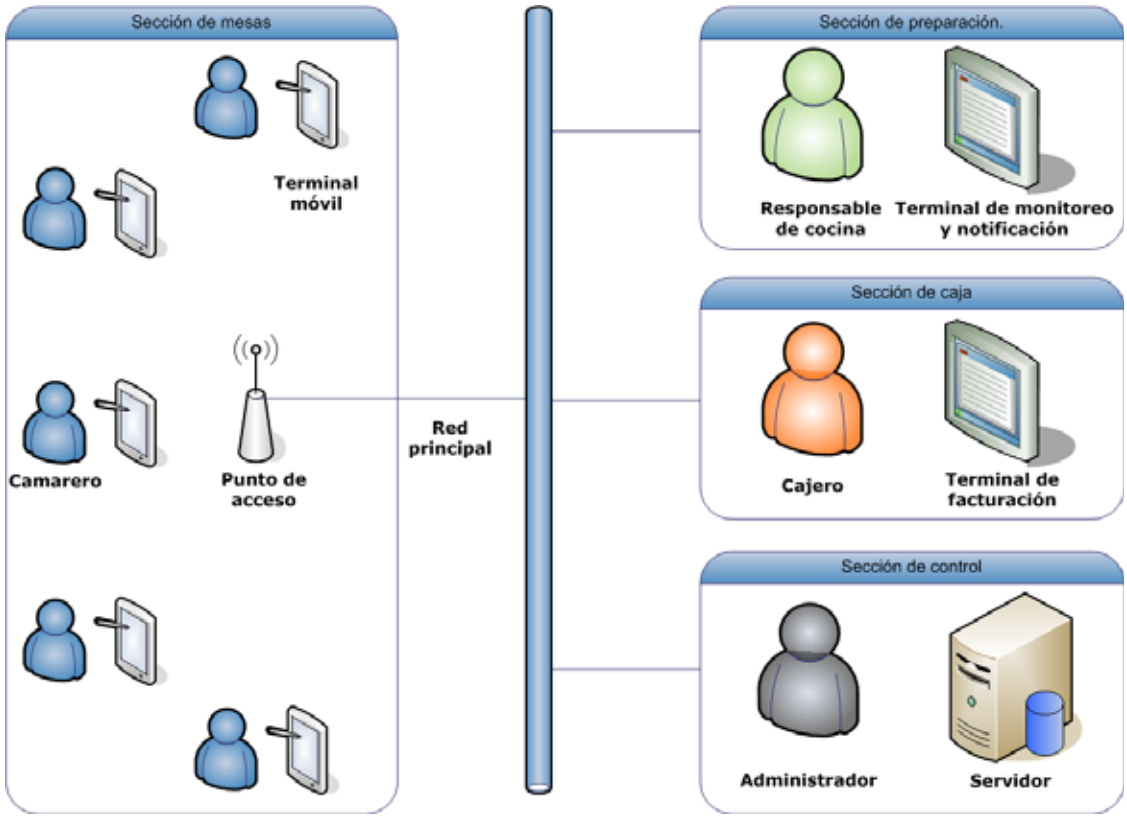


Figura 4.1: Diagrama estructural del sistema.

Lógica de dominio.

Para organizar la lógica de dominio, se utilizó *domain model*, debido principalmente a que el modelo de objetos puede ser reutilizado en el desarrollo de otras aplicaciones, o bien modificaciones o adaptaciones a esta aplicación; de esta forma se buscó aumentar al máximo las capacidades de reutilización del modelo de dominio en otros contextos.

Debido a que la aplicación está concebida para distintos tipos de interfaces de usuario y puesto que la mayoría de éstas interfaces interactúan de forma remota con la aplicación, se utilizó *service layer* y la variante *domain façade* para implementar el estilo de integración RMI.

Aunque en general, los Web Services no agregan ninguna funcionalidad, se encargan de gestionar la concurrencia de los clientes mediante un esquema combinado de instancia por petición y por sesión. Los servicios luego delegan la prestación de servicios a clases administradas, que implementan manejo de transacciones declarativamente a través de AOP.

Acceso a datos.

Para ser consistentes con la elección de *domain model* como patrón de organización de lógica de dominio, se optó por utilizar *data mapper* como patrón de acceso a datos. La utilización de *active record*

supondría acoplar innecesariamente el modelo de dominio a una capa de acceso a datos, lo que coarta de entrada, cualquier posibilidad de reutilizar dicho modelo en otros contextos, donde es posible que no se contemple la necesidad de los servicios de persistencia.

Data mapper resultó ser en el caso de esta aplicación, una de las mejores elecciones para implementar los servicios de persistencia, ya que no se presentaron condiciones excepcionales al momento de especificar la información de mapeo entre el modelo orientado a objetos y el esquema de base de datos relacional.

Elección de la plataforma de desarrollo.

Para implementar la aplicación, esto incluye a la totalidad de los subproyectos, se utilizó la platarma de desarrollo de Microsoft, el .NET Framework en su versión 3.5. Originalmente, la parte central de la aplicación, `RestomaticServer` fue implementada completamente en la plataforma Java Enterprise Edition de Sun Microsystems.

Sin embargo, debido a algunas limitantes del stack de Web Services de Java (específicamente, la carencia de un mecanismo de notificación al cliente o *callback*) y al considerable *overhead* que suponía el despliegue de la aplicación dentro de un servidor de aplicaciones, se decidió reescribir la aplicación desde cero, utilizando los servicios de la plataforma .NET.

Comparativamente hablando, las dos plataformas tienen características muy similares, desde la utilización de entornos administrados de máquina virtual, hasta la utilización de lenguajes de programación con la misma variante sintáctica, derivada de C.

La creación de WebServices e interfaces de usuario son algunas de las actividades más demandantes de la aplicación y generalmente consumen una cantidad grande de tiempo. En el caso de Java, no existen herramientas que permitan hacer inferfaces gráficas para dispositivos móviles, al menos no, de una forma sencilla. La plataforma .NET en su más reciente versión ofrece una biblioteca de clases que ayuda a reducir el tiempo de desarrollo de la interfaz gráfica en el cliente móvil, utilizando el API de Windows Forms. El soporte de la herramienta Visual Studio, permite diseñar visualmente las pantallas, definir la disposición de los elementos de la interfaz y proveerles del código necesario para responder a los eventos desencadenados por el usuario.

El lenguaje C# a diferencia de Java, posee una cantidad de construcciones sintácticas que hacen particularmente fácil la implementación de notificaciones tipo *callback*, manejo de colecciones, un dialecto de consulta de objetos independiente del orígen de datos, etc. Dichas facilidades del lenguaje permiten a los desarrolladores despreocuparse de tareas de bajo nivel que no aportan ningún valor al resultado final.

En las figuras 4.2 y 4.3 se pueden observar sutiles diferencias entre la forma de programar eventos en las plataformas Java y .NET Framework. Como se puede observar, los lenguajes de programación comparten una sintaxis muy parecida, aunque varía la aplicación de algunos conceptos. En los ejemplos señalados la diferencia más significativa consiste en la programación del mecanismo de notificación de eventos; mientras que en Java es necesario el uso de interfaces y colecciones, en .NET se emplean los delegados, construcciones invocables que encapsulan llamadas a métodos, y que soportan *broadcast*.
A pesar de que Java es la plataforma de desarrollo más madura y la calidad de las herramientas de código fuente disponibles es excelente, muchas de estas herramientas están portadas directamente a

```
 1 import java.util.ArrayList;
 2 import java.util.List;
 3
 4 interface ActionListener {
 5     void actionPerformed(ActionEvent event);
 6 }
 7
 8 class Button {
 9     private String label;
10     private List<ActionListener> listeners = new
ArrayList<ActionListener>();
11
12     public String getLabel() { return label; }
13     public void setLabel(String label) { this.label = label; }
14
15     public void addActionListener(ActionListener listener){
16         if(!listeners.contains(listener)){
17             listeners.add(listener);
18         }
19     }
20
21     public void removeActionListener(ActionListener listener){
22         if(listeners.contains(listener)){
23             listeners.remove(listener);
24         }
25     }
26
27     protected void raiseActionEvent(){
28         for(ActionListener listener : listeners){
29             ActionEvent event = new ActionEvent(this);
30             listener.actionPerformed(event);
31         }
32     }
33 }
34
35 class ActionEvent {
36     private Object source;
37
38     public Object getSource(){
39         return source;
40     }
41
42     public ActionEvent(Object source){
43         this.source = source;
44     }
45 }
```

Figura 4.2: Boceto de implementación de un componente UI y el patrón Observador en Java por medio de interfaces.

```
 1 using System;
 2
 3 class Button
 4 {
 5     public string Label { get; set; }
 6     private event ActionListener click;
 7
 8     public event ActionListener Click
 9     {
10         add
11         {
12             click += value;
13         }
14
15         remove
16         {
17             click -= value;
18         }
19     }
20
21     protected void RaiseEvent()
22     {
23         if (click != null)
24         {
25             click(new ActionEvent(this));
26         }
27     }
28 }
29
30 delegate void ActionListener(ActionEvent actionEvent);
31
32 class ActionEvent
33 {
34     public object Source { get; protected set; }
35
36     public ActionEvent(object source)
37     {
38         Source = source;
39     }
40 }
```

Figura 4.3: Boceto de implementación de un componente UI y el patrón Observador en C# por medio de delegados.

la plataforma .NET y presentan una funcionalidad ligeramente reducida, más suficiente para las necesidades de este proyecto; en algunos casos la reconfiguración y utilización de dichas herramientas es prácticamente indistinguible entre ambas plataformas.

En suma, todos estos factores condujeron a la decisión de utilizar al .NET Framework, como la plataforma de desarrollo objetivo de esta aplicación. La decisión, trasciende aspectos subjetivos, ya que de hecho, mucha de la funcionalidad fué implementada primero en una plataforma y posteriormente reescrita en otra, por lo que la evaluación de ambas tecnologías resultó más favorable a la plataforma .NET, al menos para el desarrollo de esta aplicación.

Tecnologías de acceso a datos.

El patrón de acceso a datos utilizado en la aplicación Restomatic es *Data Mapper*. En la plataforma .NET de forma convencional se utiliza una combinación de patrones de arquitectura: *Table Module* como patrón de organización de lógica de dominio y *Table Data Gateway* como patrón de acceso a datos[2]. El exponente por excelencia de acceso a datos en .NET es la tecnología ADO.NET 2.0, que con ayuda del entorno de desarrollo Visual Studio permite generar de forma automática, casi todos los artefactos necesarios para realizar acceso a datos, entre ellos los *Data Set*s, los *Data Table*s y los *Table Adapter*s.

Aunque ADO.NET y los *Table Adapter*s son tecnologías establecidas para el desarrollo de los servicios de persistencia, tienen algunas limitantes que añaden complejidad a la aplicación y que rebasan los beneficios obtenidos por el generador de código. Vale la pena hacer una exploración un poco más detallada de tales problemas.

Los problemas inherentes del estándar de acceso a datos en .NET: ADO.NET.

En especial, cuando se está trabajando con un modelo orientado a objetos por medio de *Domain Model*, ADO.NET no ayuda en ninguna medida a resolver el problema del desfase entre las representaciones relacional y orientada a objetos. En estas circunstancias, el esquema de base de datos se manipula directamente desde el código de la aplicación, lo que en general, sólo cambia la ubicación del problema *object/relational mismatch***??** del servidor de base de datos a la aplicación.

En ADO.NET, la manipulación de las tablas de la base de datos se efectúa directamente por medio de construcciones conocidas como *Data Table*s y *Data Set*s, que son objetos que imitan el esquema de la tabla en la aplicación. A pesar de que dichos objetos son autogenerados y fáciles de regenerar en el caso de un cambio al esquema original, no es posible trabajar a nivel de tupla[3]; para ello, es necesario agregar una capa adicional de objetos de dominio, que contienen los mismos datos de estado, añaden comportamiento y soportan encapsulación, herencia y polimorfismo. Esta capa adicional de objetos, tiene un impacto en la cantidad de código, pues no sólo es necesario duplicar la representación de datos de una

[2] Por motivos de tiempo, no se incluyó una descripción de estos patrones de arquitectura en capitulos anteriores, sin embargo, puede consultarse *Patterns of Enterprise Application Architecture*, de Martin Fowler para una descripción detallada.

[3] Esta apreciación no es enteramente cierta, debido a la existencia de *Data Row*s; no obstante, éstos no permiten emplear ninguno de los principios básicos de la programación orientada a objetos, y su utilidad dentro de las diversas capas de la aplicación es cuestionable. Por lo anterior, cabe hacer la precisión sobre la sentencia: **no es posible trabajar a nivel de tupla, sin la presencia de objetos intrusivos que acoplan la aplicación innecesariamente al esquema de base de datos**.

entidad, sino que es necesario proveer de objetos que transformen la representación orientada a objetos (*domain model*) a las representaciones menos granulares como los *Data Table*s y *Data Set*s (o bien, *Data Row*s). Un ejemplo típico de un *DataTable* puede observarse en la figura 4.4.

```
public partial class BillTablesDataTable : TypedTableBase<BillTablesRow>
{
    private DataColumn columnBillId;
    private DataColumn columnTableId;

    // Lots of code ...

    public DataColumn BillIdColumn
    {
        get { return this.columnBillId; }
    }

    public DataColumn TableIdColumn
    {
        get { return this.columnTableId; }
    }

    public int Count
    {
        get { return this.Rows.Count; }
    }

    // Lots of code ...

    public BillTablesRow this[int index]
    {
        get { return ((BillTablesRow)(this.Rows[index])); }
    }

    // Lots of code ...

    public event BillTablesRowChangeEventHandler BillTablesRowChanging;
    public event BillTablesRowChangeEventHandler BillTablesRowChanged;
    public event BillTablesRowChangeEventHandler BillTablesRowDeleting;
    public event BillTablesRowChangeEventHandler BillTablesRowDeleted;

    // Lots of code ...
}
```

Figura 4.4: Código recortado de un objeto *Data Table* autogenerado por Visual Studio.

Es posible optar por no añadir esta capa de objetos del modelo de dominio a la aplicación, pero en-

tonces, la arquitectura de la aplicación quedaría completamente acoplada al esquema de base de datos. Aún más debido al bajo nivel de abstracción, los objetos de ADO.NET sólo permiten hacer operaciones de bajo nivel sobre el conjunto de datos. Las consultas que involucran relaciones entre tablas, muchas veces son realizadas a través de mecanismos de recuperación manual, que son complicados de programar y propensos a errores, especialmente en lo referente a la conservación de la integridad referencial.

Dichos problemas se conocen, se han documentado**??** y ha mostrado ser una causa común del aumento de complejidad en las aplicaciones escritas para la plataforma .NET. Es común encontrar *Data Set*s directamente en el código que implementa la lógica de dominio y aún hasta en la capa de presentación; y si bien esto generalmente no es malo en aplicaciones pequeñas, inflexibiliza innecesariamente las aplicaciones de gran escala*Para un tratamiento más detallado sobre la correcta utilización de ADO.NET, aplicada a las interfaces de usuario en clientes ricos, puede consultarse el libro de Brian Noyes:* Data Binding with Windows Forms 2.0, *editado por Addison Wesley*, haciéndolas más resistentes al cambio y por lo tanto, más costosas de mantener.
Para tratar de superar los problemas ocasionados con la utilización de ADO.NET, la tendencia general sugiere la utilización de esquemas no intrusivos, donde la capa de acceso a datos aisle al desarrollador de aplicaciones en la medidad de lo posible, de las representaciones nativas del almacén de datos; a la característica de diseñar modelos de dominio desacoplados de la representación nativa del almacén de datos se les denomina *persistence ignorance*.

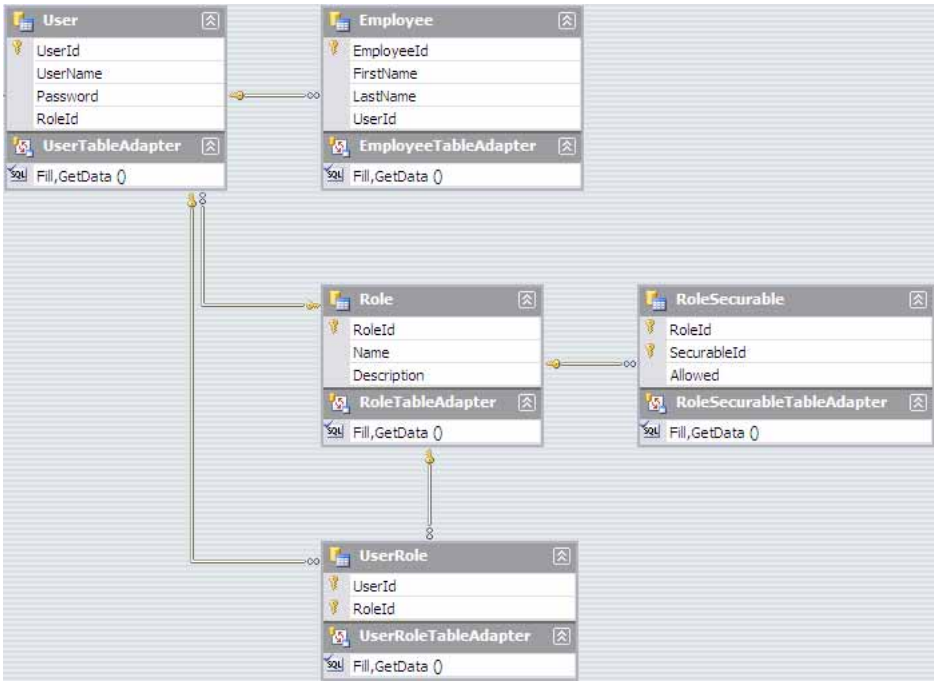


Figura 4.5: Data Set correspondiente al esquema RestomaticDB.

En la figura 4.5 se puede apreciar una vista de un conjunto de *Data Set*s y sus correspondientes *Table Adapter*s generados a partir del modelo de dominio mostrado en el apéndice C. Se puede observar que dichas representaciones son equivalentes, la diferencia radica en que los objetos de ADO.NET se pueden consumir directamente por el código de la aplicación, como se aprecia en la figura 4.6.

```
BillTableAdapter billAdapter = new BillTableAdapter();
RestomaticDBDataSet.BillDataTable billTable = billAdapter.GetData();

foreach(RestomaticDBDataSet.BillRow billRow in billTable.Rows)
{
    if (billRow.State == 0) // New
    {
        billRow.FinishedTime = DateTime.Now;
        billRow.State = 2; // Completed
    }
}

billAdapter.Update(billTable);
```

Figura 4.6: Modelo de programación mediante ADO.NET, utilizando Data Sets.

Object-Relational Mapping.

En otras plataformas más consolidadas como es el caso de la plataforma Java, se tuvieron problemas similares, aunque más graves en relación al alto acoplamiento entre el modelo de dominio y la capa de acceso a datos; la técnica más empleada y que ha demostrado tener un éxito modesto pero importante en este aspecto, es ORM (*Object-Relational Mapping*).

Desde 2001, se gestaron en la plataforma Java distintos motores de persistencia basados en el concepto de ORM, siendo uno de los exponentes más populares el proyecto Hibernate, conducido por Gavin King. Esta herramienta permitía, casi desde sus inicios, organizar la lógica de la aplicación utilizando objetos con un excelente nivel de desacoplamientos de la base de datos relacional.

Sin embargo, en la plataforma .NET no se dió un debate tan intenso como en el caso de Java sino hasta hace poco tiempo, cuando los problemas resultantes de la utilización de ADO.NET 2.0 fueron más evidentes y la necesidad de buscar alternativas se hizo más fuerte. Debido al considerable éxito de la herramienta Hibernate en la plataforma Java, se ha intentado portar la herramienta a la plataforma .NET, bajo el nombre de NHibernate; la cercanía que existe entre las plataformas Java y .NET, entre los lenguajes Java y C#, junto con la extensa documentación y la amplia base de desarrolladores, hacen a NHibernate una de las herramientas más utilizadas en la plataforma .NET para implementar servicios de persistencia, basados en el patrón de arquitectura *Data Mapper*.

A diferencia de lo que ocurre con ADO.NET, donde normalmente el diseño de los artefactos de persistencia es un paso posterior a la definición del esquema de base de datos, las herramientas ORM permiten invertir o conservar este orden**??**. Puesto que el proceso de diseño de clases del modelo de dominio está cercanamente relacionado al nivel diseño conceptual del modelo de base de datos, permite evitar la duplicidad de información en el proceso de diseño de ambos modelos. Aún más, es posible construir el esquema físico a partir del modelo de dominio[4] y también a la inversa[5]; en cualquier caso, las herramientas ORM no suplen el proceso de diseño del esquema de base de datos, ni eliminan la necesidad

[4] Sin embargo, el modelo resultante es de una calidad generalmente inferior a un script escrito por un desarrollador de base de datos.

[5] En este caso, la calidad del código autogenerado es mucho peor que en el otro caso, ya que la herramienta de

de normalizar el esquema resultante.

En general, ORM es una técnica muy adecuada cuando el diseño y la implementación de ambos modelos están bajo el control del desarrollador de aplicaciones, pero aún en casos en que el esquema de base de datos esté fijo, es posible construir un modelo de dominio desacoplado, auque en algunos casos la información de mapeo es muy extensa. Ya se ha mencionado que el orden de producción del modelo de dominio y el esquema es una elección del desarrollador, lo cierto es que se necesitan al menos cuatro ingredientes para implementar los servicios de persistencia a nivel de entidad:

- La definición de clase del objeto de dominio.
- La información de mapeo.
- El esquema físico de la base de la entidad que representa la clase.
- Una implementación de *Data Mapper*.

Las figuras 4.10, 4.8, 4.9 muestran respectivamente, los tres primeros ingredientes, aplicados a un objeto de dominio que representa una orden el sistema Restomatic.

```
[Serializable]
public class Order
{
    public virtual long OrderId { get; set; }
    public virtual DateTime RequestedTime { get; set; }
    public virtual DateTime StartedTime { get; set; }
    public virtual DateTime FinishedTime { get; set; }
    public virtual DateTime DeliveryTime { get; set; }
    public virtual OrderState State  { get; set; }
    public virtual OrderPriority Priority { get; set; }
    public virtual Dish Dish  { get; set; }
    public virtual string Instructions  { get; set; }

    public virtual decimal Price
    {
        get { return Dish != null ? Dish.Price : 0M; }
    }

    public Order() : this() { }

    // Domain Logic
}
```

Figura 4.7: Modelo de dominio.

mapeo no es capaz de inferir el uso de colecciones, distinguir entre diversas formas de implementar cardinalidades, elegir un tipo de datos de entre varias opciones de la plataforma, utilizar correctamente esquemas de nombrado, etc.

```
CREATE TABLE [Bills].[Order]
(
    [OrderId] BIGINT NOT NULL IDENTITY(1, 1),
    [RequestedTime] DATETIME NOT NULL DEFAULT GETDATE(),
    [StartedTime] DATETIME NULL,
    [FinishedTime] DATETIME NULL,
    [DeliveryTime] DATETIME NULL,
    [State] TINYINT NOT NULL,
    [DishId] INT NOT NULL,
    [Instructions] NVARCHAR(256) NULL DEFAULT '',
    [Priority] TINYINT NOT NULL,
    CONSTRAINT [PK_Order] PRIMARY KEY CLUSTERED([OrderId]),
    CONSTRAINT [FK_Order_Dish] FOREIGN KEY([DishId])
        REFERENCES [Menu].[Dish]([DishId]),
    CONSTRAINT [CK_Order_State] CHECK([State] IN (0, 1, 2, 3, 4, 5)),
    CONSTRAINT [CK_Order_Priority] CHECK([Priority] IN (0, 1, 2))
)
GO
```

Figura 4.8: Esquema de base de datos.

En este ejemplo se aprecia una forma de especificar la información de mapeo, utilizando configuración externa provista al mapeador mediante un doumento XML. El documento resultante es casi tan grande como la información de estado de la clase, y bastante más grande que el esquema de base de datos. Sin embargo, existen alternativas de especificación de la información de mapeo, que reducen la cantidad de información que necesita ser especificada manualmente por el desarrollador de aplicaciones. En el caso de la plataforma Java, la tecnología de persistencia estándar conocida como *Java Persistence API* provee de un conjunto de anotaciones para especificar la información de mapeo; una característica de JPA que se conoce como configuración por convención, permite al desarrollador omitir, por ejemplo, los nombres y tipos de datos de las columnas y dejar al mapeador inferir por medio de *reflection* el nombre del campo de clase y el tipo de dato SQL compatible. En la figura **??** permite ver una implementación en Java de la misma clase utilizando anotaciones como medio de especificación de información de mapeo, dicho código es equivalente al código combinado de las figuras 4.10 y 4.9.

El mapeador NHibernate permite utilizar en la plataforma .NET atributos para espeficicar la información de mapeo de una forma similar a JPA en Java; sin embargo, cabe señalar que dicha elección tiene el gran inconveniente de que, a diferencia del documento XML, para hacer un cambio menor en el nombrado del esquema, es necesario recompilar el código de la aplicación. Lo que es más grave, la información de mapeo, en el caso de esquemas no normalizados o fuera del control del desarrollador de aplicaciones puede ser muy extensa, al grado de ocupar una porción considerable del código fuente, lo que en última instancia dificulta la legibilidad del código resultante. Otro inconveniente radica en que, un modelo de dominio donde la información de mapeo está contenida directamente en el código fuente, reduce la posibilidad de reutilizar el código en otro contexto de persistencia, derivado del hecho de que no es posible dotar a la clase de dos juegos de anotaciones distintas.

Por tal motivo, para la implementación de los servicios de persistencia se eligió la configuración externa

```
<hibernate-mapping
       xmlns="urn:nhibernate-mapping-2.2"
       assembly="Restomatic.DomainModel"
       namespace="Restomatic.DomainModel.Bills">

   <class name="Order" table="[Order]" schema="Bills">
       <id name="OrderId" column="OrderId" type="System.Int64"
       unsaved-value="0" access="property">
           <generator class="identity"/>
       </id>

       <property name="RequestedTime" column="RequestedTime"
           type="System.DateTime" not-null="true" update="false"
           access="property"/>

       <property name="StartedTime" column="StartedTime"
           type="System.DateTime" not-null="false" update="true"
           access="property"/>

       <property name="FinishedTime" column="FinishedTime"
           type="System.DateTime" not-null="false" update="true"
           access="property"/>

       <property name="DeliveryTime" column="DeliveryTime"
           type="System.DateTime" not-null="false" update="true"
           access="property"/>

       <property name="State" column="State" not-null="true"
           update="true" access="property"/>

       <many-to-one name="Dish" column="DishId"
           class="Restomatic.DomainModel.Menu.Dish" not-null="true"
           cascade="none" fetch="select" lazy="false"/>

       <property name="Instructions" column="Instructions"
           type="System.String" length="256" not-null="false"
           update="false" access="property"/>

       <property name="Priority" column="Priority" not-null="true"
           update="true" access="property"/>
   </class>

</hibernate-mapping>
```

Figura 4.9: Información de mapeo.

```
 @Entity
@Table(name = "[Order]", schema = "[Bills]")
public class Order implements Serializable
{
    @Id @GeneratedValue
    private long orderId = 0L;

    @Temporal(TemporalType.TIME)
    private Date requestedTime;

    @Temporal(TemporalType.TIME)
    private Date startedTime;

    @Temporal(TemporalType.TIME)
    private Date finishedTime;

    @Temporal(TemporalType.TIME)
    private Date deliveryTime;

    @Enumerated(EnumType.ORDINAL)
    private OrderState state;

    @ManyToOne
    @JoinColumn(name = "dishId")
    private Dish dish;

    @Basic
    private String instructions = "";

    @Enumerated(EnumType.ORDINAL)
    private OrderPriority priority;

    // Domain Logic
}
```

Figura 4.10: Modelo de dominio e información de mapeo con anotaciones.

en XML para dejar al modelo de dominio completamente independiente de la información de mapeo, como puede apreciarse en los apéndices D y B.

Escribir un modelo de dominio del tipo *persistence ignorance* junto con una información de mapeo externa, permite simplificar considerablemente el código de la capa de persistencia. Un ejemplo del modelo de programación[6] que deriva del uso de las herramientas ORM se puede apreciar en la figura 4.11, que debe contrastarse con las construcciones de bajo nivel de la figura 4.6.

```
IBillRepository repository = new BillRepository(...);
IList<Bill> bills = repository.FindAll();

foreach (Bill bill in bills)
{
    if(bill.State == BillState.New)
    {
        bill.FinishedTime = DateTime.Now;
        bill.State = BillState.Completed;

        repository.Update(bill);
    }
}
```

Figura 4.11: Modelo de programación derivado del uso de herramientas ORM.

A pesar de que, al momento de realizar este trabajo la herramienta ORM de NHibernate es una de las mejores opciones disponibles, es posible elevar aún más el nivel de abstracción mediante otras herramientas, particularmente mediante el nuevo estándar de acceso a datos en .NET 3.5 denominado *ADO.NET Entity Framework* y el dialecto de consulta conocido como *Language INtegrated Query* (LINQ). Aunque los beneficios son particularmente sustanciales desde el punto de vista del desarrollador, no representan una mejora considerable, ni brindan funcionalidad nueva sobre las herramientas ORM existentes; por desgracia, hasta el momento de publicación de este trabajo, el EF no estaba listo para aplicaciones en producción y la documentación al respecto, era limitada. Sin embargo, para contrastar brevemente los resultados esperados de esta nueva tecnología de persistencia, en la figura 4.12 se muestra el modelo de programación resultante de la aplicación tanto del *ADO.NET Entity Framework*, como de LINQ[7].
Es evidente que mientras más se eleva el nivel de abstracción en el modelo de programación de la capa de acceso a datos, más se permite al desarrollador de aplicaciones concentrarse en implementar

[6] La decisión de recuperar todos los objetos de la clase *Bill* mediante el método *FindAll()* y posteriormente filtrarlos, es ciertamente una elección muy pobre; se eligió así para este ejemplo con el objeto de no complicarlo al introducir el lenguaje HQL o el *API Criteria* de NHibernate. El uso de estos medios permite recuperar solo los objetos que cumplan con una determinada característica, lo que disminuye considerablemente el uso de memoria y el tiempo de ejecución.

[7] El uso de LINQ no se restringe al *Entity Framework* como herramienta de ORM; de hecho, la arquitectura de proveedores de LINQ no excluye la utilización de otras tecnologías alternativas al EF. Actualmente, se está desarrollando un proveedor que en el futuro cercano, permitirá realizar consultas nativas mediante LINQ utilizando NHibernate.

```
using (RestomaticDataContext context = new RestomaticDataContext())
{
    // LINQ
    var bills = from Bill bill in context.Bills
                where bill.State == BillState.New
                select bill;

    foreach (Bill bill in bills)
    {
        bill.FinishedTime = DateTime.Now;
        bill.State = BillState.Completed;
    }

    // Better abstraction level!
    context.SaveChanges();
}
```

Figura 4.12: Modelo de programación derivado del uso del ADO.NET EF y LINQ.

la lógica de dominio, actividad que aporta la mayor cantidad de valor a la aplicación. No obstante, es inevitable que también se pierda control sobre los detalles de bajo nivel, lo que en muchas ocasiones implica un impacto en el desempeño.

En cualquier caso, las herramientas ORM están lo bastante consolidadas como para considerarse una opción, en muchos casos excelente, en la implementaión de los servicios de persistencia, tal cual puede corroborarse en los trabajos de Martin Fowler**??** y Jimmy Nilsson**??** y bajo este razonamiento se incluyeron en el desarrollo de Restomatic.

Parte III

Especificación de requerimientos y código fuente.

Apéndice

A

Especificación de requerimientos.

Introducción y objetivos.

En el presente documento se detallarán el conjunto de requerimientos que serán incorporados a la implementación final del sistema de información, así como la estructura general y los componentes que lo conforman.

Versión simplificada del proceso del restaurante.

La interacción con los clientes la llevan a cabo los camareros en el área de mesas. El camarero recibe las órdenes de los comensales y genera minutas. Una vez que el cliente verifica y aprueba las minutas, éstas son entregadas a la sección de preparación del restaurante. Una vez que los platillos contenidos en las minutas se han preparado, el encargado de la sección de preparación, notifica al camarero la existencia de platillos listos para entrega y procede a la entrega de los platillos contenidos en las órdenes.

Cuando el cliente ha terminado, solicita al camarero el total de los cargos generados por su consumo. El camarero entrega la información de las minutas en la sección de caja del establecimiento, para que éste coteje las minutas contra la lista de precios del restaurante, calcule el total, emita un recibo y lo entregue al camarero, quién a su vez lo presenta ante el cliente.

El cliente puede acudir directamente a la sección de caja para realizar su pago o bien, puede realizar el pago directamente con el camarero.

Objetivo principal de la implementación del sistema

Eficientar y facilitar la operación de las tareas en el proceso del restaurante.

Objetivos específicos de la implementación del sistema.

- Reducir el tiempo requerido para capturar las órdenes solicitadas por los clientes.
- Minimizar el tiempo de entrega de las minutas a la sección de preparación.
- Eliminar la posibilidad de pérdida o extravío de las minutas antes y después de su entrega al área de preparación.
- Brindar en todo momento certeza sobre el estado de una orden emitida.
- Eliminar desplazamientos innecesarios del camarero hasta el área de preparación o el área de caja.
- Agilizar la notificación y entrega de órdenes preparadas.
- Eliminar los errores aritméticos y omisiones al integrar los cargos por consumo.
- Facilitar la gestión del menú.

Estructura funcional y organización del sistema.

Buscando reducir la complejidad y los conflictos de dependencias que puedan presentarse en la implementación del sistema, se ha dividido el sistema en subsistemas cohesivos, que implementan funcionalidades comunes de acuerdo a los siguientes grupos de tareas:

- Gestión de cuentas y emisión de órdenes.
- Procesamiento de minutas y preparación de órdenes.
- Facturación por consumo.
- Administración del menú.

Subsistema de administración de menú.

Consiste en una aplicación, desplegada en un cliente pesado, que se conecta directamente al servidor y que permite realizar tareas relacionadas con la composición del menú, la descripción de sus elementos, los precios y la disponibilidad de los platillos:

- Agregación modificación o eliminación de elementos del menú.
- Gestión de disponibilidad de los platillos del menú.

Subsistema de gestión de cuentas y emisión de órdenes.

Consiste en una aplicación desplegada en un dispositivo móvil tipo Tablet PC, que permite realizar el grupo de tareas que abarca las interacciones entre el cliente y el camarero:

- Apertura de cuentas.
- Cancelación de cuentas.
- Consulta del subtotal de una cuenta.
- Elaboración de minutas.
- Envío de minutas.
- Cancelación de minutas.
- Consulta del estado de una minuta.
- Movimiento de minutas entre cuentas.
- Búsqueda de platillos en el menú.
- Consulta de los platillos del menú.
- Agregación y cancelación de órdenes.
- Movimiento de órdenes entre cuentas.
- Consulta del estado de una orden.

Subsistema de procesamiento de minutas y preparación de órdenes.

Consiste en una aplicación desplegada sobre un cliente pesado que permite llevar a cabo las tareas presentes en la sección de preparación del restaurante:

- Gestión del estado de minutas.
- Gestión del estado de órdenes.
- Monitoreo de órdenes pendientes.
- Notificación de órdenes preparadas.

Subsistema de facturación por consumo.

Consiste en una aplicación desplegada en un cliente pesado, que permite realizar las tareas que abarcan la interacción entre el cliente y el cajero:

- Cierre de cuentas.
- Consulta de los datos de una cuenta.
- Gestión del pago de cuentas.
- Elaboración de notas de consumo.

Componente de administración de sesiones.

Adicionalmente a las tareas propias de la operación del restaurante, existen tareas relacionadas con la identificación de los empleados ante el sistema, que básicamente suponen la utilización de una política de autenticación basada en password.

Debido a que la funcionalidad proporcionada por esta parte del sistema no puede operar de forma autónoma, no se le consideró como un subsistema, sino como un componente, que será reutilizado en el resto de los subsistemas para gestionar las sesiones de trabajo, correspondientes a cada uno de los empleados que interaccionan con el sistema.

Requerimientos funcionales.

Subsistema de administración de sesiones.

El sistema deberá permitir al usuario realizar las siguientes operaciones:

Clave	Descripción
M01RF01	Iniciar una sesión de trabajo, mediante el uso de un nombre de usuario y una contraseña. La sesión deberá guardar los datos de la hora de inicio y la hora de cierre de sesión, así como mantener un registro de todos los movimientos realizados durante la duración de la sesión.
M01RF02	Cerrar una sesión de trabajo.
M01RF03	Solicitar cambio de contraseña. Para cambiar la contraseña, el sistema deberá solicitar la contraseña anterior, y la nueva contraseña. Deberá haber un mecanismo que permita reasignar la contraseña en caso de pérdida.

Subsistema de administración del menú.

Administración del menú a la carta.

El sistema deberá permitir al usuario realizar las siguientes operaciones:

Clave	Descripción
M02RF01	Dar de alta categorías de platillos, introduciendo el nombre de la categoría y una fotografía genérica que distinga a la categoría.
M02RF02	Brindar la opción de cambiar la categoría de un platillo de forma individual.
M02RF03	Eliminar una categoría de platillos, brindando al usuario la posibilidad de eliminar todos los platillos de la categoría, o bien moverlos a otras categorías antes de proceder a la eliminación.
M02RF04	Dar de alta platillos individuales en el menú a la carta, deberá ser posible introducir el nombre del platillo, asignarle una categoría, brindar una breve descripción, establecer un precio y agregar una fotografía representativa del platillo preparado. En caso de no contar con una fotografía del platillo, el sistema deberá proveer por defecto, una imagen genérica dependiendo de la categoría seleccionada.
M02RF05	Habilitar o deshabilitar un platillo, para mostrarlo u ocultarlo en el menú a la carta, pero sin la necesidad de eliminarlo.
M02RF06	Modificar la información de cada platillo.
M02RF07	Imprimir el menú a la carta.
M02RF08	Buscar los platillos del menú a la carta. La búsqueda deberá permitir introducir como criterios, los nombres, las descripciones y rangos de precio.

Administración del menú del día.

El sistema deberá permitir al usuario realizar las siguientes operaciones:

Clave	Descripción

M03RF01	Generar el menú del día, mediante la selección de platillos individuales del menú a la carta y asignarles un precio agrupado. Deberá ser posible buscar los platillos del menú a la carta durante la generación del menú del día. La búsqueda deberá permitir introducir como criterios, los nombres, las descripciones y rangos de precio. El menú del día deberá ser vigente por un periodo determinado de tiempo y una vez expirado ese tiempo debe inhabilitarse automáticamente.
M03RF02	Generar más de un menú del día.
M03RF03	Guardar el menú del día de forma que pueda ser reutilizado en el futuro.
M03RF04	Imprimir el menú del día.

Administración del menú de paquetes.

El sistema deberá permitir al usuario realizar las siguientes operaciones:

Clave	Descripción
M04RF01	Seleccionar platillos del menú a la carta y agruparlos para integrar un paquete.
M04RF02	Guardar los paquetes de forma que puedan ser reutilizados en el futuro.
M04RF03	Imprimir el menú de paquetes.

Subsistema de gestión de cuentas y emisión de órdenes.

Consulta remota del menú.

El sistema deberá permitir al usuario realizar las siguientes operaciones:

Clave	Descripción
M05RF01	Consultar una lista con todos los platillos del menú a la carta, con la opción de agrupar por categorías.
M05RF02	Consultar una lista con los menús del día, que estén actualmente disponibles.
M05RF03	Consultar una lista con todos los paquetes que estén actualmente disponibles.
M05RF04	Buscar los platillos en el menú a la carta por algún criterio, como nombre, descripción o rangos de precio.

Gestión de cuentas.

El sistema deberá permitir al usuario realizar las siguientes operaciones:

Clave	Descripción
M06RF01	Consultar una lista con todas las cuentas abiertas que no tengan asignado a ningún camarero.
M06RF02	Establecer una cuenta activa para una sesión, de forma que se asigne automáticamente un camarero a esa cuenta, con base en la información de la sesión.
M06RF03	Abrir una cuenta nueva. La asignación del camarero se hará con base en la información de la sesión. Para crear la cuenta será necesario indicar el número de comensales, así como la mesa o las mesas que serán cubiertas por esa cuenta.
M06RF04	Cancelar una cuenta.
M06RF05	Consultar los datos de una cuenta.

Emisión de minutas.

El sistema deberá permitir al usuario realizar las siguientes operaciones:

Clave	Descripción
M07RF01	Crear una nueva minuta.
M07RF02	Cancelar una minuta existente. Una minuta solo podrá ser cancelada si ninguno de los platillos que contiene está siendo preparado.
M07RF03	Consultar el estado de una minuta. Se deberá mostrar la información de cada una de las órdenes contenidas en la minuta, así como su estado actual.
M07RF04	Enviar una minuta al área de preparación. La minuta sólo debe ser enviada si contiene al menos una orden.

Control de órdenes.

El sistema deberá permitir al usuario realizar las siguientes operaciones:

Clave	Descripción
M08RF01	Agregar una orden a una minuta existente. Deberá seleccionarse un platillo a la carta, un menú del día o un paquete para poder agregar la orden a la minuta. Deberá ser posible especificar indicaciones adicionales para cada platillo, antes de ser enviado a la sección de preparación.
M08RF02	Cancelar una orden. Será posible cancelar una orden, incluso después de ser enviada a la sección de preparación, con la restricción de que la orden aún no esté siendo preparada.
M08RF03	Mover una orden. Será posible mover una orden siempre que no haya sido cancelada, entre cuentas distintas.
M08RF04	Entregar una orden. Esta operación lógica se realiza cuando una orden ha sido entregada al cliente.

Subsistema de procesamiento de minutas y preparación de órdenes.

El sistema deberá permitir al usuario realizar las siguientes operaciones:

Clave	Descripción
M09RF01	Consultar una lista con todas las órdenes enviadas al área de preparación.
M09RF02	Filtrar y ordenar la lista de órdenes de acuerdo al estado de las mismas.
M09RF03	Ordenar los elementos de la lista de órdenes de acuerdo a la prioridad y al instante de llegada.
M09RF04	Seleccionar una orden individual y alterar su estado para indicar que dicha orden ha comenzado su preparación.
M09RF05	Seleccionar una orden individual y alterar su estado para indicar que dicha orden ha finalizado su preparación (está lista).

El sistema deberá realizar de forma automática las siguientes operaciones:

Clave	Descripción
M09RF06	Actualizar el estado de una minuta individual, cuando cualquiera de las órdenes contenidas haya iniciado su preparación, para indicar que dicha minuta ha comenzado a ser procesada.

M09RF07	Actualizar el estado de una minuta individual, cuando todas las órdenes contenidas estén listas, para indicar que dicha minuta ha sido completada.
M09RF08	Generar una notificación para el subsistema de gestión de cuentas y emisión de órdenes, cuando se actualice el estado de una orden individual para indicar que está lista.

Subsistema de facturación por consumo.

El sistema deberá permitir al usuario realizar las siguientes operaciones:

Clave	Descripción
M10RF01	Cerrar una cuenta. No deberá permitirse agregar minutas a las cuentas cerradas.
M10RF02	Consultar los datos de una cuenta.
M10RF03	Modificar el estado de una cuenta para indicar que ha sido pagada. Sólo se podrán aceptar pagos sobre cuentas previamente cerradas.
M10RF04	Imprimir nota de consumo. Deberá mostrar el total a pagar y un detalle de todos los platillos que fueron incluidos en la cuenta.

Sección de definiciones.

En esta sección se presenta a manera de glosario, una serie de conceptos así como su significado particular en el contexto de este documento.

Concepto	**Definición**
Platillo	Alimento en presentación individual que se ofrece en el menú.
Categoría	Clasificación que se asigna a los platillos del menú de acuerdo a su naturaleza. Las siguientes son ejemplos de categorías de platillos: Bebidas, Entremeses, Platos fuertes, Postres.
Orden	Elemento que se utiliza para solicitar un platillo del menú. Adicionalmente al platillo, en la orden se pueden hacer indicaciones adicionales sobre el platillo solicitado.
Minuta	Elemento que se utiliza para agrupar un conjunto de órdenes.
Cuenta	Es una lista en la que se asientan todas las órdenes solicitadas por un cliente en una comida, el costo de cada platillo individual y el costo de todos los platillos en conjunto.
Estado de una orden	Un valor discreto asignado a una orden de acuerdo a su situación en el tiempo. Los valores posibles para el estado de una orden son los siguientes: Creada, Recibida, Preparándose, Lista, Entregada y Cancelada.
Estado de una minuta	Un valor discreto asignado a una minuta de acuerdo a su situación en el tiempo. Los valores posibles para el estado de una minuta son los siguientes: Creada, Recibida, Preparándose, Completada y Cancelada.

Estado de una cuenta	Un valor discreto asignado a una cuenta de acuerdo a su situación en el tiempo. Los valores posibles para el estado de una minuta son los siguientes: Abierta, Cerrada, Pagada, Cancelada.
Cliente	Sustantivo que se utiliza para generalizar a toda persona que utiliza los servicios del restaurante.
Comensal	Término utilizado para referirse a un cliente del restaurante, que se encuentra en el comedor.
Menú a la carta (*a la carte*)	Es una lista de todos los platillos que se pueden preparar y servir, que se cocinan justo al ordenarse. En el menú a la carta se ofrecen los precios de cada platillo de forma individual.
Menú del día (*table de hôte*)	Es una lista que contiene un grupo de platillos que generalmente abarca todas las categorías, que se cocinan anticipadamente y se sirven al momento. En el menú del día se ofrece un precio único para todo el conjunto de platillos. El menú del día normalmente nunca es el mismo en días consecutivos.
Paquete	Conjunto establecido de platillos que generalmente abarcan todas las categorías y que ofrecen bajo un único precio.
Menú de paquetes	Es una lista que contiene todos los paquetes ofrecidos por el restaurante, que pueden prepararse y servirse y que se cocinan al ordenarse.
Área de mesas	Es la sección del restaurante donde se ubica a los clientes al momento de servir. Se puede utilizar de forma intercambiable con *comedor*.
Área de preparación	Es la sección del restaurante donde se preparan los alimentos. Se puede utilizar de forma intercambiable con *cocina*.
Área de caja	Es la sección del restaurante que se encarga de procesar el pago derivado del consumo de un cliente. Se puede referir sencillamente como *caja*.
Camarero	Empleado del restaurante que tiene como función, tomar las órdenes del cliente, enviarlas a la sección de preparación y devolvérselas al cliente cuando los platillos se hayan preparado. Se puede referir coloquialmente como *mesero*.
Cajero	Empleado del restaurante que tiene como función procesar los pagos de la cuenta de un cliente.
Encargado de cocina	Empleado del restaurante que tiene como función, dar seguimiento a las órdenes entregadas por el camarero a la cocina para su preparación e informar al camarero cuando las órdenes estén listas para ser entregadas.
Empleados del restaurante	Término genérico con el que puede referirse a los camareros, cajeros y encargados de cocina, pero no se limita a estos.
Sesión	Unidad de trabajo en un sistema de información, mediante la cuál se pueden realizar operaciones de forma controlada y bajo una política de autenticación.
Facturación	Proceso mediante el cual, se informa al cliente del total del cargo generado por su consumo. Generalmente este proceso se limita a agregar los precios individuales de cada platillo o paquete, pero no excluye necesariamente el gravamen de impuestos.

Modelo de casos de uso.

Administración de sesiones.

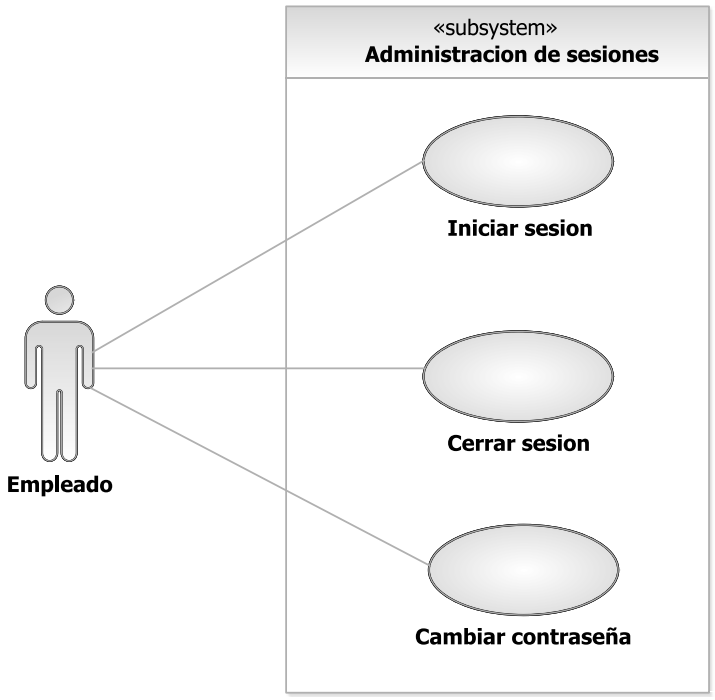


Figura A.1: Diagrama de casos de uso del subsistema de administración de sesiones.

Iniciar sesión.

Narrativa de caso de uso.	
Caso de uso:	Iniciar sesión.
Actores:	Empleado.
Descripción:	Crea una sesión de trabajo para el empleado dentro del sistema.
Precondiciones:	El empleado tiene asignadas previamente las credenciales utilizadas para autenticarse ante el sistema.
Postcondiciones:	El sistema ha creado una nueva sesión de trabajo para el empleado autenticado.

Estímulo del actor	**Respuesta del sistema**
1 El empleado introduce su nombre de usuario y su contraseña.	
	2 El sistema verifica que las credenciales del usuario sean válidas.
	3 El sistema crea una sesión de trabajo y registra la hora de inicio.

Condiciones excepcionales

Excepción: Paso 2.
Causa: Las credenciales proporcionadas son incorrectas.
Acción del sistema: Abandonar el caso de uso. Informar la causa por la cual no fue posible iniciar la sesión.
Respuesta esperada del actor: Repetir el procedimiento desde el Paso 1.

Cerrar sesión.

Narrativa de caso de uso.	
Caso de uso:	Cerrar sesión.
Actores:	Empleado.
Descripción:	Cerrar una sesión de trabajo en el sistema.
Precondiciones:	Deberá existir previamente una sesión de trabajo abierta.
Postcondiciones:	El sistema ha concluido la sesión de trabajo.

Estímulo del actor	**Respuesta del sistema**
1 El empleado solicita el cierre de la sesión actual.	
	2 El sistema cierra la sesión de trabajo.

Cambiar contraseña.

Narrativa de caso de uso.	
Caso de uso:	Cambiar contraseña.
Actores:	Empleado.
Descripción:	Cambiar los datos de inicio de sesión de un empleado, mediante el reemplazo de la contraseña.
Precondiciones:	Deberá existir previamente una sesión de trabajo abierta.
Postcondiciones:	Se ha guardado cualquier cambio en los datos de inicio de sesión.

Estímulo del actor	**Respuesta del sistema**
1 El empleado solicita al sistema el cambio de su contraseña.	
	2 El sistema solicita autenticación del usuario.
3 El empleado introduce sus datos de inicio de sesión.	
	4 El sistema verifica que las credenciales sean válidas.
	5 El sistema solicita al empleado su nueva contraseña.
6 El empleado realiza el cambio de contraseña.	
	7 El sistema solicita la confirmación de los cambios.
	8 El sistema guarda permanente los cambios.

Condiciones excepcionales

Excepción: Paso 4.
Causa: Las credenciales no son válidas.
Acción del sistema: Abandonar el caso de uso. Informar la causa del error.
Respuesta esperada del actor: Repetir el procedimiento desde el Paso 1.

Administración del menú a la carta.

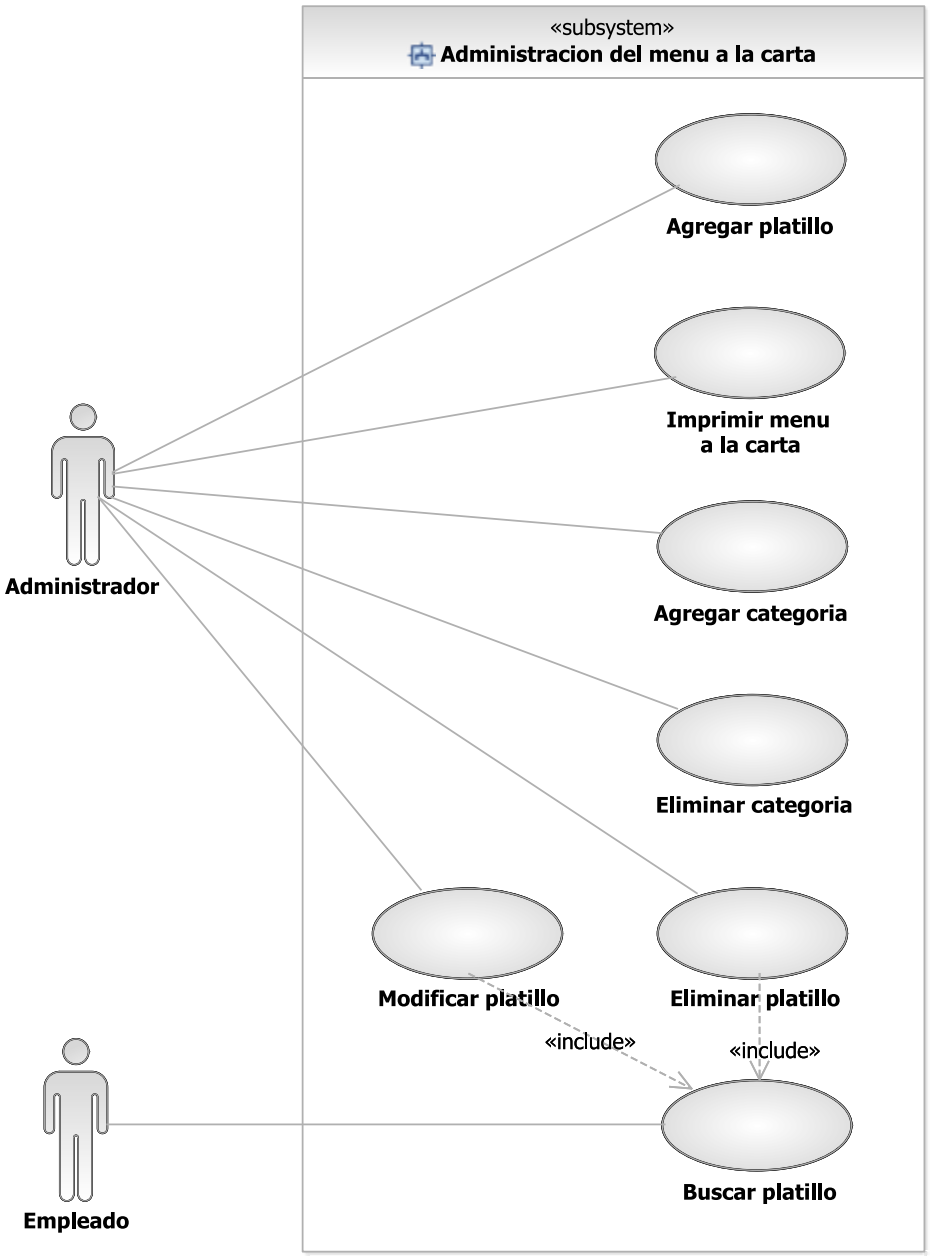


Figura A.2: Diagrama de casos de uso del subsistema de administración del menú a la carta.

Agregar categoría.

Narrativa de caso de uso.	
Caso de uso:	Agregar categoría.
Actores:	Administrador.
Descripción:	El caso de uso comienza cuando el administrador solicita la creación de una nueva categoría en el menú a la carta.
Precondiciones:	El administrador tiene abierta una sesión de trabajo en el sistema.
Postcondiciones:	Se ha agregado una nueva categoría al menú a la carta.

Estímulo del actor	**Respuesta del sistema**
1 El administrador introduce el nombre de la categoría.	
	2 El sistema crea una nueva categoría.

Condiciones excepcionales

Excepción: Paso 2.
Causa: El nombre de categoría ya existe.
Acción del sistema: Abortar el caso de uso. Notificar la causa del error.
Respuesta esperada del actor: Reiniciar el procedimiento a partir del Paso 1.

Eliminar categoría.

Narrativa de caso de uso.	
Caso de uso:	Eliminar categoría.
Actores:	Administrador.
Descripción:	El caso de uso comienza cuando el administrador solicita la eliminación de una cagtegoría del menú a la carta.
Precondiciones:	El administrador tiene abierta una sesión de trabajo en el sistema.
Postcondiciones:	Se ha eliminado la categoría del menú a la carta.

Estímulo del actor	**Respuesta del sistema**
1 El administrador selecciona una categoría del menú a la carta.	
	2 El sistema elimina la categoría seleccionada.

Condiciones excepcionales

Excepción: Paso 2.
Causa: La categoría seleccionada contiene al menos un platillo del menú a la carta.
Acción del sistema: Abandonar el caso de uso. Notificar la causa del error.
Respuesta esperada del actor: Reiniciar el procedimiento desde el Paso 1.

Agregar platillo.

Narrativa de caso de uso.	
Caso de uso:	Agregar platillo.
Actores:	Administrador.
Descripción:	El caso de uso inicia cuando el administrador solicita la adición de un nuevo platillo al menú a la carta.
Precondiciones:	El administrador tiene una sesión de trabajo abierta en el sistema.
Postcondiciones:	Se ha creado un nuevo platillo en el menú la carta.

Estímulo del actor	**Respuesta del sistema**
1 El empleado introduce los datos del platillo.	
	2 El sistema crea un nuevo platillo en el menú a la carta.

Condiciones excepcionales

Excepción: Paso 2.
Causa: El nombre del platillo ya existe en el menú a la carta.
Acción del sistema: Abandonar el caso de uso. Notificar la causa del error.
Respuesta esperada del actor: Reiniciar el procedimiento desde el Paso 1.

Buscar platillo.

Narrativa de caso de uso.	
Caso de uso:	Buscar platillo.
Actores:	Empleado.
Descripción:	Permite encontrar un platillo del menú a la carta, realizando una búsqueda basada en las propiedades de los platillos individuales.
Precondiciones:	El administrador tiene una sesión de trabajo abierta en el sistema.
Postcondiciones:	*Ninguna*

Estímulo del actor	**Respuesta del sistema**
1 El empleado introduce los datos de la búsqueda.	
	2 El sistema muestra una lista de los platillos que coinciden con los datos de búsqueda.

Notas

Los datos de búsqueda son un subconjunto de los campos del registro que representa a la entidad platillo en el modelo de datos.

Eliminar platillo.

Narrativa de caso de uso.	
Caso de uso:	Eliminar platillo.
Actores:	Administrador.
Descripción:	El caso de uso inicia cuando el administrador solicita la remoción de un platillo del menú a la carta.
Precondiciones:	El administrador tiene una sesión de trabajo abierta en el sistema.
Postcondiciones:	Se ha eliminado el platillo seleccionado del menú la carta.

Estímulo del actor	**Respuesta del sistema**
1 De ser necesario, el administrador invoca al caso de uso **Buscar platillo**.	
2 El administrador selecciona el platillo a eliminar.	
	3 El sistema elimina el platillo del menú a la carta.

Condiciones excepcionales

Excepción: Paso 3.
Causa: El platillo forma parte de algún paquete o menú del día.
Acción del sistema: Abandonar el caso de uso. Informar la causa del error.

Notas

Para poder eliminar un platillo es necesario que éste no se encuentre contenido dentro de ningún paquete o menú del día.

Modificar platillo.

Narrativa de caso de uso.	
Caso de uso:	Modificar platillo.
Actores:	Administrador.
Descripción:	Permite modificar los datos de un platillo del menú.
Precondiciones:	El administrador tiene una sesión de trabajo abierta en el sistema.
Postcondiciones:	Se han actualizado los datos del platillo modificado.

Estímulo del actor	**Respuesta del sistema**
1 De ser necesario, el administrador invoca al caso de uso **Buscar platillo**.	
2 El administrador selecciona el platillo a ser modificado.	
3 El administrador modifica los datos del platillo.	
	4 El sistema confirma los cambios realizados.
	5 El sistema actualiza los cambios realizados.

Condiciones excepcionales

Excepción: Paso 4.
Causa: El administrador rechaza los cambios realizados.
Acción del sistema: Abandonar el caso de uso. Restaurar los datos del platillo hasta antes de su modificación.

Imprimir menú a la carta.

Narrativa de caso de uso.	
Caso de uso:	Imprimir menú a la carta.
Actores:	Administrador.
Descripción:	El caso de uso comienza cuando el administrador solicita la impresión del menú a la carta. La lista con todos los platillos disponibles del menú a la carta, se imprime sobre un medio físico.
Precondiciones:	El administrador tiene abierta una sesión de trabajo en el sistema.
Postcondiciones:	*Ninguna*

Estímulo del actor	**Respuesta del sistema**
1 El administrador solicita la impresión del menú a la carta.	
	2 El sistema imprime la lista de platillos del menú a la carta sobre un medio físico.

Administración del menú de paquetes.

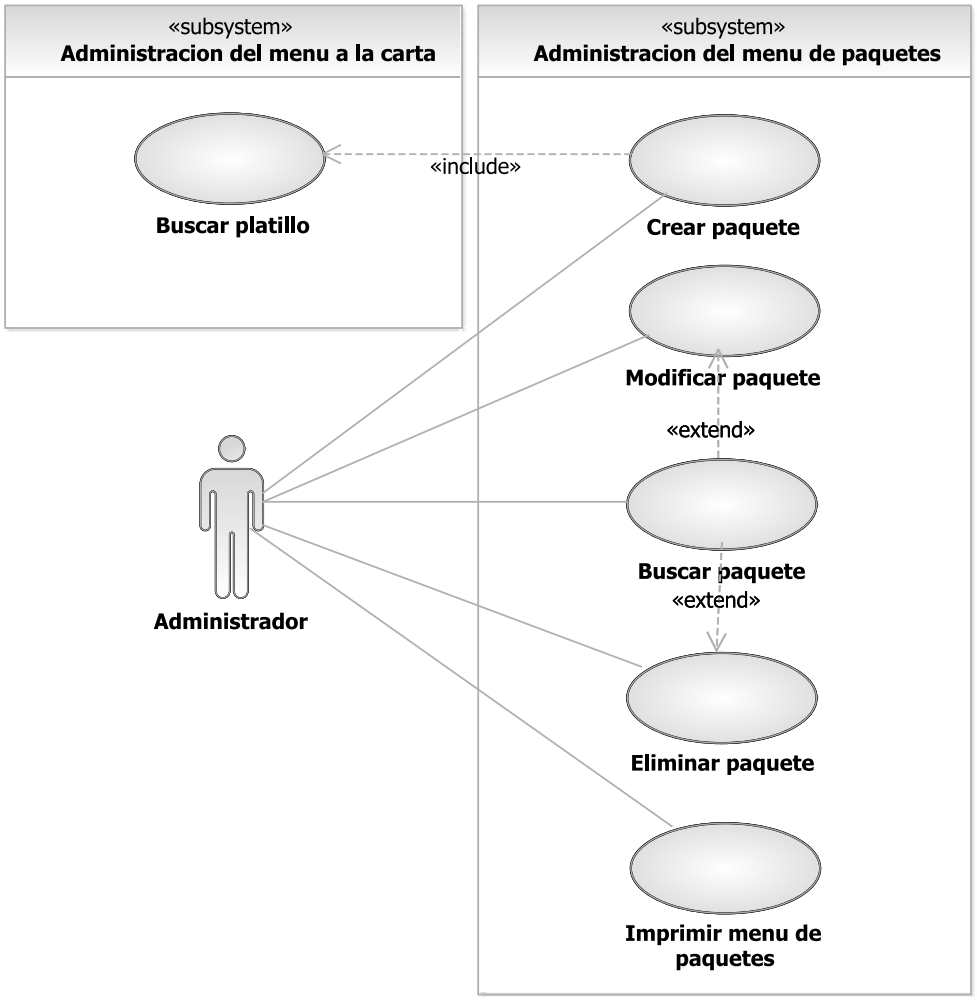


Figura A.3: Diagrama de casos de uso del subsistema de administración del menú de paquetes.

Crear paquete.

Narrativa de caso de uso.	
Caso de uso:	Crear paquete.
Actores:	Administrador.
Descripción:	Permite agrupar un conjunto de platillos del menú.
Precondiciones:	El administrador tiene abierta una sesión de trabajo en el sistema.
Postcondiciones:	Se ha agregado una entrada al menú de paquetes.

Estímulo del actor	**Respuesta del sistema**
1 El administrador solicita la creación de un paquete.	
	2 El sistema crea una lista de platillos vacía.
3 El administrador opcionalmente invoca al caso de uso **Buscar platillo**.	
4 El administrador selecciona un platillo y lo agrega a la lista de platillos.	
5 El administrador establece los datos del paquete.	
	6 El sistema crea el paquete a partir de la lista de platillos.

Rutas alternativas.

Posterior al Paso 4 (Estímulo del actor): El administrador elimina un elemento de la lista de platillos.
Posterior al Paso 5 (Estímulo del actor): El administrador cancela la creación del paquete.

Condiciones excepcionales

Excepción: Paso 6.
Causa: La lista de platillos está vacía.
Acción del sistema: Informar la causa del error.
Respuesta esperada del actor: Reiniciar el procedimiento a partir del Paso 3.

Notas

La secuencia formada por el Paso 3 y Paso 4, puede repetirse tantas veces sea necesario.

Buscar paquete.

Narrativa de caso de uso.	
Caso de uso:	Buscar paquete.
Actores:	Administrador.
Descripción:	Permite encontrar una lista de entradas en el menú de paquetes.
Precondiciones:	El administrador ha iniciado una sesión de trabajo en el sistema.
Postcondiciones:	*Ninguna.*
Estímulo del actor	**Respuesta del sistema**
1 El administrador introduce los datos de búsqueda.	
	2 El sistema muestra la lista de todos los paquetes que coinciden con los datos de búsqueda.

Modificar paquete.

Narrativa de caso de uso.	
Caso de uso:	Modificar paquete.
Actores:	Administrador.
Descripción:	Permite modificar los datos y la composición de un elemento del menú de paquetes.
Precondiciones:	El administrador ha iniciado una sesión de trabajo en el sistema.
Postcondiciones:	La información del paquete ha sido actualizada.

Estímulo del actor	**Respuesta del sistema**
1 El administrador opcionalmente invoca al caso de uso **Buscar paquete**.	
2 El administrador selecciona un paquete.	
	3 El sistema muestra la información del paquete.
4 El administrador realiza las modificaciones.	
	5 El sistema solicita la confirmación de los cambios.
	6 El sistema guarda los cambios.

Rutas alternativas.

Posterior al Paso 4 (Estímulo del actor): El administrador cancela los cambios realizados.

Eliminar paquete.

Narrativa de caso de uso.		
Caso de uso:	Eliminar paquete.	
Actores:	Administrador.	
Descripción:	Permite borrar una entrada del menú de paquetes.	
Precondiciones:	El administrador ha iniciado una sesión de trabajo en el sistema.	
Postcondiciones:	Se ha eliminado el paquete seleccionado.	
Estímulo del actor		**Respuesta del sistema**
1 El administrador opcionalmente invoca al caso de uso **Buscar paquete**.		
2 El administrador selecciona un paquete.		
		3 El sistema muestra todos los datos del paquete.
4 El administrador confirma la eliminación del paquete.		
		5 El sistema elimina el paquete.
Rutas alternativas.		
Posterior al Paso 3 (Estímulo del actor): El administrador cancela la eliminación del paquete.		

Imprimir menú de paquetes.

Narrativa de caso de uso.	
Caso de uso:	Imprimir menú de paquetes.
Actores:	Administrador.
Descripción:	El caso de uso comienza cuando el administrador solicita laimpresión del menú de paquetes. La lista con todos los paquetes disponibles se imprime sobre un medio físico.
Precondiciones:	El administrador ha iniciado una sesión de trabajo en el sistema.
Postcondiciones:	*Ninguna.*

Estímulo del actor	**Respuesta del sistema**
1 El administrador solicita la impresión del menú de paquetes.	
	2 El sistema imprime la lista de todos los paquetes disponibles.

Gestión de cuentas.

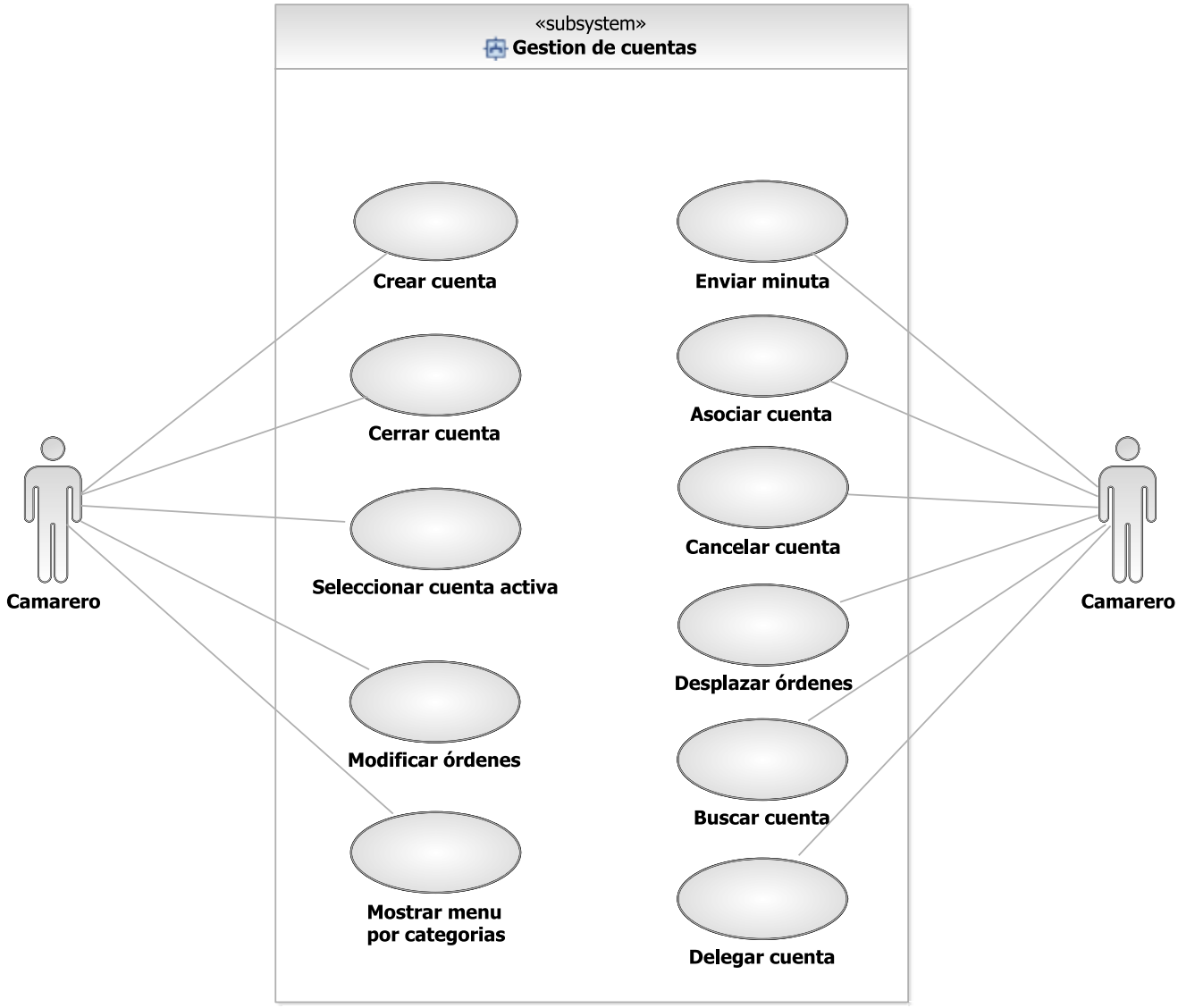


Figura A.4: Diagrama de casos de uso del subsistema de gestión de cuentas.

Buscar cuenta.

Narrativa de caso de uso.	
Caso de uso:	Buscar cuenta
Actores:	Camarero, Cajero.
Descripción:	Permite realizar una búsqueda sobre el conjunto de cuentas creadas.
Precondiciones:	El camarero/cajero ha iniciado una sesión en el sistema.
Postcondiciones:	*Ninguna.*
Estímulo del actor	**Respuesta del sistema**
1 El camarero/cajero introduce los datos de búsqueda de la cuenta.	
	2 El sistema muestra una lista de todas las cuentas que coincidan con los datos de búsqueda.
Notas	
Los datos de búsqueda de la orden incluyen: ▪ El rango de fecha o de horas. ▪ El estado de la cuenta. ▪ El número de mesa.	

Enviar minuta.

Narrativa de caso de uso.	
Caso de uso:	Enviar minuta
Actores:	Camarero.
Descripción:	Este caso de uso se inicia cuando el cliente se dispone a ordenar sus platillos. El camarero recibe cada una de las órdenes del cliente y las agrupa en una minuta. Una vez que la minuta ha sido concluida, el camarero confirma el contenido con el cliente, antes de suscribirla al área de preparación.
Precondiciones:	El camarero ha iniciado una sesión en el sistema. La cuenta a la que será agregada la minuta deberá estar abierta.
Postcondiciones:	Se ha asociado un conjunto de órdenes asociadas a la cuenta activa. Se ha establecido el estado de cada orden en enviada.

Estímulo del actor	**Respuesta del sistema**
	1 El sistema presenta una lista con los platillos del menú.
2 El camarero opcionalmente invoca al caso de uso *Buscar platillo*.	
3 El camarero selecciona un platillo del menú y anota las indicaciones requeridas por el cliente.	
	4 El sistema agrega a la minuta la orden creada por el camarero.
5 El camarero confirma con el cliente el contenido de la minuta.	
	6 El sistema actualiza la cuenta con la información de cada una de las órdenes contenidas en la minuta.

Condiciones excepcionales

Excepción: Paso 5.
Causa: El cliente rechaza el contenido de la minuta.
Respuesta esperada del actor: Eliminar la orden de la minuta y reiniciar el caso de uso a partir del paso 2.

Excepción: Paso 6. **Causa:** No se han agregado órdenes a la minuta. **Respuesta esperada del actor:** Informar al camarero a causa del error. Reiniciar el caso de uso a partir del paso 2.
Notas
La secuencia de pasos 2 y 3 se pueden repetir tantas veces sea necesario.

Modificar órdenes.

Narrativa de caso de uso.	
Caso de uso:	Modificar órdenes.
Actores:	Camarero.
Descripción:	Auxilia para rectificar los datos de una orden emitida por un cliente, en caso de que éste cambie de opinión.
Precondiciones:	El camarero ha iniciado una sesión en el sistema. La cuenta activa debe de estar abierta. La orden que desea modificarse debe encontrarse en estado de enviada.
Postcondiciones:	Se han actualizado todos los datos de la orden modificada.

Estímulo del actor	**Respuesta del sistema**
	1 El sistema presenta las órdenes solicitadas dentro de la cuenta activa.
2 El camarero selecciona una orden de la lista de órdenes y realiza las modificaciones.	
3 El camarero confirma las modificaciones con el cliente.	
	4 El sistema actualiza los datos de la orden.

Notas

Las modificaciones posibles que se pueden aplicar a una orden consisten en agregar o cambiar las indicaciones de una orden y eliminar una orden. Los cambios solo se podrán realizar sobre las órdenes que aun no hayan comenzado a prepararse.

La secuencia de pasos 2, 3 y 4 se pueden repetir tantas veces sea necesario.

Desplazar órdenes.

Narrativa de caso de uso.	
Caso de uso:	Desplazar órdenes.
Actores:	Camarero.
Descripción:	El caso de uso se origina del hecho de que un cliente puede absorber los cargos derivados del consumo de otra cuenta. El caso de uso inicia, cuando un cliente solicita al camarero absorber parcial o totalmente la cuenta de otro cliente. Cada una de las órdenes seleccionadas, contenidas en la cuenta de origen, se trasladan a la cuenta activa.
Precondiciones:	El camarero ha iniciado una sesión en el sistema. La cuenta de origen contiene al menos una orden y no debe de encontrarse pagada. La cuenta de destino no debe encontrarse pagada.
Postcondiciones:	Se han movido las órdenes de la cuenta de origen a la cuenta activa. Si la cuenta de origen no contiene al menos una orden, esta cuenta se cancela. Los cambios se guardan en forma permanente.

Estímulo del actor	**Respuesta del sistema**
1 El camarero opcionalmente invoca al caso de uso *Buscar cuenta*.	
2 El camarero selecciona la cuenta de origen.	
	3 El sistema presenta la lista de todas las órdenes contenidas en la cuenta de origen.
4 El camarero selecciona una o mas cuentas de la cuenta de origen.	
5 El camarero confirma con el cliente las órdenes que serán absorbidas.	
	6 El sistema transfiere las órdenes seleccionadas a la cuenta activa.
	7 El sistema actualiza los datos de la cuenta de origen y la cuenta activa.

Notas

La cuenta de origen solo será cancelada si al final de la operacion de transferencia, no contiene al menos una orden, en cualquier otro caso, la cuenta de origen no cambia su estado.

La secuencia de pasos 2, 3 y 4 puede repetirse tantas veces sea necesario.

Mostrar menú por categorías.

Narrativa de caso de uso.	
Caso de uso:	Mostrar menú por categorías.
Actores:	Camarero.
Descripción:	Permite consultar la lista de platillos del menú, agrupados por categoría.
Precondiciones:	El camarero ha iniciado una sesión en el sistema.
Postcondiciones:	*Ninguna.*

Estímulo del actor	**Respuesta del sistema**
1 El camarero solicita al sistema el menú por categorías.	
	1 El sistema presenta al camarero una lista de platillos agrupados por categoría.

Crear cuenta.

Narrativa de caso de uso.	
Caso de uso:	Crear cuenta.
Actores:	Camarero.
Descripción:	Permite crear una cuenta para dar seguimiento al consumo de un cliente que recién acaba de ocupar el comedor.
Precondiciones:	El camarero ha iniciado una sesión en el sistema.
Postcondiciones:	Se ha creado una nueva cuenta en el sistema. La cuenta se ha asociado a la sesión actual. La cuenta se ha promovido como cuenta activa en la sesión actual.

Estímulo del actor	**Respuesta del sistema**
1 El camarero introduce el número de mesa y la cantidad de comensales.	
	1 El sistema abre una nueva cuenta y la asocia a la sesión actual.

Delegar cuenta.

Narrativa de caso de uso.	
Caso de uso:	Delegar cuenta.
Actores:	Camarero.
Descripción:	Permite a un camarero, abandonar su sesión de trabajo y desacoplarse de sus cuentas asociadas, de forma que otros camareros puedan atenderlas.
Precondiciones:	El camarero ha iniciado una sesión en el sistema. Existe al menos una cuenta asociada a la sesión actual.
Postcondiciones:	El sistema guarda la información de todas las cuentas abandonadas por el camarero.

Estímulo del actor	**Respuesta del sistema**
1 El camarero selecciona una o más cuentas de su lista de cuentas asociadas.	
	2 El sistema agrega las cuentas seleccionadas a la lista de cuentas abandonadas.

Asociar cuenta.

Narrativa de caso de uso.	
Caso de uso:	Asociar cuenta.
Actores:	Camarero.
Descripción:	Permite asociar a un camarero, una o más cuentas de la lista de cuentas abandonadas.
Precondiciones:	El camarero ha iniciado una sesión en el sistema. Existe al menos una cuenta en la lista de cuentas abandonadas.
Postcondiciones:	La cuenta seleccionada se ha agregado a la lista de cuentas asociadas del camarero.

Estímulo del actor	**Respuesta del sistema**
	1 El sistema presenta la lista de cuentas abandonadas.
2 El camarero selecciona una o mas cuentas de la lista de cuentas abandonadas.	
3 El camarero confirma la asociación de las cuentas seleccionadas.	
	4 El sistema asocia las cuentas seleccionadas a la sesión actual.

Seleccionar cuenta activa.

Narrativa de caso de uso.	
Caso de uso:	Seleccionar cuenta activa.
Actores:	Camarero.
Descripción:	Permite al camarero seleccionar de su lista de cuentas asociadas, aquella cuenta que estará atendiendo de forma exclusiva.
Precondiciones:	El camarero ha iniciado una sesión en el sistema. El camarero tiene al menos una cuenta asignada.
Postcondiciones:	Se ha establecido una cuenta activa en la sesión actual.

Estímulo del actor	Respuesta del sistema
	1 El sistema presenta una lista de las cuentas asociadas a la sesión actual.
2 El camarero selecciona una cuenta asociada.	
	3 El sistema registra la cuenta seleccionada como la cuenta activa de la sesión actual.

Cerrar cuenta.

Narrativa de caso de uso.	
Caso de uso:	Cerrar cuenta
Actores:	Camarero, Cajero.
Descripción:	Permite bloquear una cuenta de forma que ya no se puedan agregar más órdenes.
Precondiciones:	El camarero/cajero ha iniciado una sesión en el sistema. El camarero/cajero tiene una cuenta activa al momento de invocar al caso de uso.
Postcondiciones:	El estado de la cuenta activa ha sido cambiado a cerrada.

Estímulo del actor	Respuesta del sistema
1 El actor solicita al sistema el cierre de la cuenta activa.	
	2 El sistema cambia el estado de la cuenta activa a Cerrado.

Cancelar cuenta.

Narrativa de caso de uso.	
Caso de uso:	Cancelar cuenta
Actores:	Camarero.
Descripción:	El caso de uso inicia cuando un camarero se ve obligado a cancelar la cuenta de un cliente.
Precondiciones:	El camarero ha iniciado una sesión en el sistema. El camarero tiene una cuenta activa al momento de invocar al caso de uso.
Postcondiciones:	La cuenta activa del camarero se ha marcado como cancelada. Todas las minutas contenidas en la cuenta, así como todas las órdenes contenidas en cada una de las minutas han sido marcadas como canceladas. El sistema guarda los cambios de forma permanente.

Estímulo del actor	**Respuesta del sistema**
1 El camarero introduce en el sistema la causa de la cancelación de la cuenta.	
	2 El sistema cancela la cuenta y todas las minutas y órdenes asociadas.

Apéndice

B

Modelo de dominio.

Diagrama general de clases del modelo de dominio.

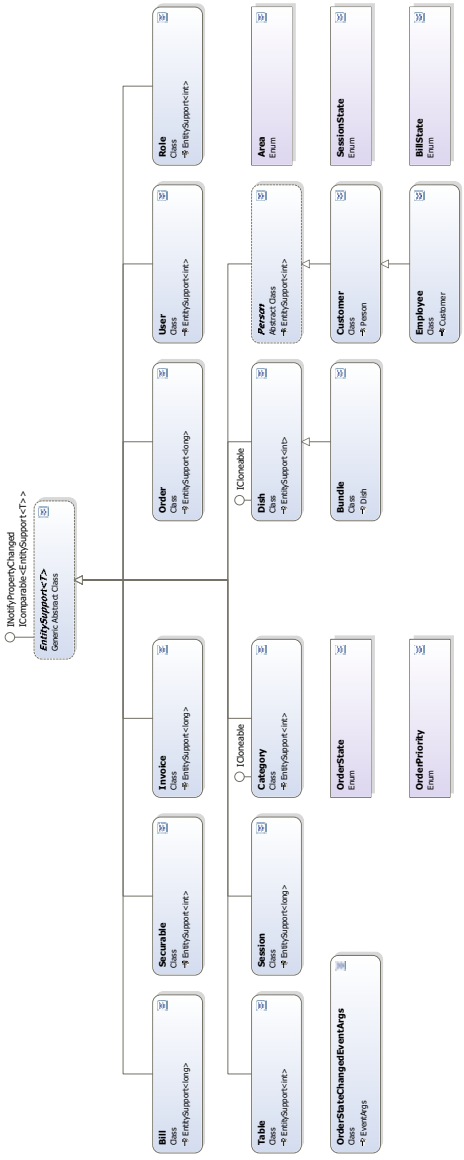
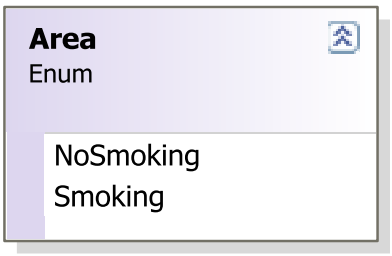


Figura B.1: Modelo de dominio de la aplicación Restomatic.

Código fuente C# de las clases del modelo de dominio.

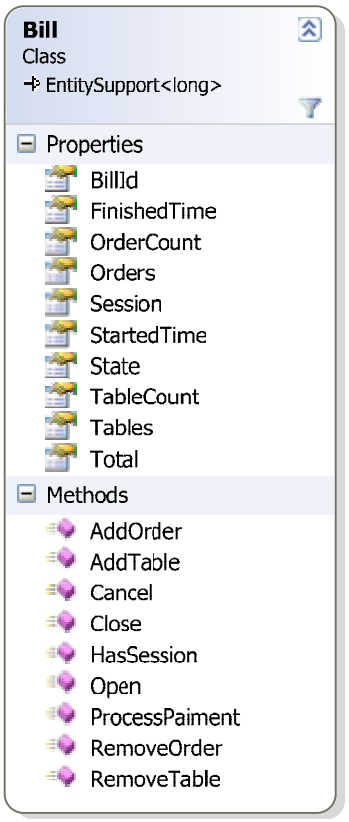
Enum `Restomatic.DomainModel.Common.Area`.



Area.cs

```
 1 using System;
 2 using System.Runtime.Serialization;
 3
 4 namespace Restomatic.DomainModel.Common
 5 {
 6     [DataContract(Namespace = "http://restomatic.azc.uam.mx/")]
 7     public enum Area
 8     {
 9         [EnumMember]
10         NoSmoking,
11
12         [EnumMember]
13         Smoking
14     }
15 }
```

Class Restomatic.DomainModel.Bills.Bill.



Bill.cs

```
1  using System;
2  using System.Runtime.Serialization;
3  using System.Collections.Generic;
4
5  using Iesi.Collections.Generic;
6
7  using Restomatic.DomainModel.Common;
8  using Restomatic.DomainModel.Security;
9
10 namespace Restomatic.DomainModel.Bills
11 {
12     [Serializable]
13     [DataContract(Namespace = "http://restomatic.azc.uam.mx/")]
14     public class Bill : EntitySupport<long>
15     {
16         #region Properties
17
18         [DataMember]
19         public virtual long BillId
20         {
21             get
22             {
23                 return Id;
```

```
24              }
25
26              protected set
27              {
28                  Id = value;
29                  NotifyChanged("BillId");
30              }
31          }
32
33          private DateTime startedTime;
34
35          [DataMember]
36          public virtual DateTime StartedTime
37          {
38              get
39              {
40                  return startedTime;
41              }
42
43              set
44              {
45                  startedTime = value;
46                  NotifyChanged("StartedTime");
47              }
48          }
49
50          private DateTime finishedTime;
51
52          [DataMember]
53          public virtual DateTime FinishedTime
54          {
55              get
56              {
57                  return finishedTime;
58              }
59
60              set
61              {
62                  finishedTime = value;
63                  NotifyChanged("FinishedTime");
64              }
65          }
66
67          private BillState  state;
68
69          [DataMember]
70          public virtual BillState State
71          {
72              get
73              {
74                  return state;
75              }
76
77              set
78              {
79                  state = value;
80                  NotifyChanged("State");
81              }
82          }
83
```

```
84          private ISet<Table> tableSet;
85
86          [DataMember(Name = "Tables")]
87          protected virtual ISet<Table> TablesSet
88          {
89              get
90              {
91                  if (tableSet == null)
92                  {
93                      lock (this)
94                      {
95                          tableSet = new HashedSet<Table>();
96                      }
97                  }
98
99                  return tableSet;
100             }
101
102             set
103             {
104                 tableSet = value != null ? new HashedSet<Table>(value) :
105                            null;
106
107                 NotifyChanged("TablesSet");
108                 NotifyChanged("Tables");
109                 NotifyChanged("TableCount");
110             }
111         }
112
113         public virtual IEnumerable<Table> Tables
114         {
115             get { return TablesSet; }
116         }
117
118         private decimal total;
119
120         [DataMember]
121         public virtual decimal Total
122         {
123             get
124             {
125                 return total;
126             }
127
128             set
129             {
130                 total = value;
131                 NotifyChanged("Total");
132             }
133         }
134
135         private ISet<Order> orderSet;
136
137         protected virtual ISet<Order> OrdersSet
138         {
139             get
140             {
141                 if (orderSet == null)
142                 {
143                     lock (this)
```

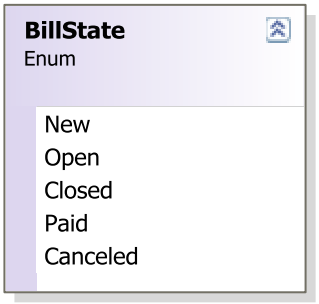
```
144                     {
145                         orderSet = new HashedSet<Order>();
146                     }
147                 }
148 
149                 return orderSet;
150             }
151 
152             set
153             {
154                 orderSet = value != null ? new HashedSet<Order>(value) :
155                            null;
156 
157                 NotifyChanged("OrdersSet");
158                 NotifyChanged("Orders");
159                 NotifyChanged("OrderCount");
160             }
161         }
162 
163         [DataMember]
164         public virtual IEnumerable<Order> Orders
165         {
166             get
167             {
168                 return OrdersSet;
169             }
170 
171             protected set
172             {
173                 OrdersSet = new HashedSet<Order>(new
List<Order>(value));
174             }
175         }
176 
177         private Session session;
178 
179         [DataMember]
180         public virtual Session Session
181         {
182             get
183             {
184                 return session;
185             }
186 
187             set
188             {
189                 session = value;
190                 NotifyChanged("Session");
191             }
192         }
193 
194         public virtual int TableCount
195         {
196             get { return TablesSet.Count; }
197         }
198 
199         public virtual int OrderCount
200         {
201             get { return OrdersSet.Count; }
202         }
```

```
203
204         public virtual bool HasSession()
205         {
206             return Session != null;
207         }
208
209         #endregion
210
211         #region Constructors
212
213         public Bill()
214         {
215             StartedTime = DateTime.Now;
216             FinishedTime = DateTime.Now.AddHours(12);
217
218             State = BillState.New;
219         }
220
221         #endregion
222
223         #region Private State Management Support Methods
224
225         private void CheckNewState()
226         {
227             if (State != BillState.New)
228             {
229                 throw new InvalidOperationException("State is not New");
230             }
231         }
232
233         private void CheckOpenState()
234         {
235             if (State != BillState.Open)
236             {
237                 throw new InvalidOperationException("State is not Open");
238             }
239         }
240
241         private void CheckClosedState()
242         {
243             if (State != BillState.Closed)
244             {
245                 throw new InvalidOperationException("State is not
Closed");
246             }
247         }
248
249         #endregion
250
251         #region State Management Methods
252
253         public virtual void Close()
254         {
255             CheckOpenState();
256             State = BillState.Closed;
257             FinishedTime = DateTime.Now;
258         }
259
260         public virtual void Open()
261         {
```

```
262             CheckNewState();
263             State = BillState.Open;
264             StartedTime = DateTime.Now;
265         }
266
267         public virtual void ProcessPaiment()
268         {
269             CheckClosedState();
270             State = BillState.Paid;
271         }
272
273         public virtual void Cancel()
274         {
275             if (State != BillState.Canceled)
276             {
277                 State = BillState.Canceled;
278                 FinishedTime = DateTime.Now;
279             }
280         }
281
282         #endregion
283
284         #region Tables Collection Management Methods
285
286         public virtual void AddTable(Table table)
287         {
288             CheckOpenState();
289
290             if (!TablesSet.Contains(table))
291             {
292                 TablesSet.Add(table);
293             }
294         }
295
296         public virtual void RemoveTable(Table  table)
297         {
298             CheckOpenState();
299
300             if (TablesSet.Contains(table))
301             {
302                 TablesSet.Remove(table);
303             }
304         }
305
306         #endregion
307
308         #region Orders Collection Management Methods
309
310         public virtual void AddOrder(Order order)
311         {
312             CheckOpenState();
313
314             if (OrdersSet.Contains(order))
315             {
316                 OrdersSet.Add(order);
317                 Total += order.Price;
318             }
319         }
320
321         public virtual void RemoveOrder(Order order)
```

```
322         {
323             CheckOpenState();
324 
325             if (OrdersSet.Contains(order))
326             {
327                 OrdersSet.Remove(order);
328                 Total -= order.Price;
329             }
330         }
331 
332         #endregion
333 
334         #region Context Equality
335 
336         protected override bool ContextAwareEquals(object obj)
337         {
338             Bill bill = obj  as Bill;
339 
340             return bill != null
341                 && bill.StartedTime == StartedTime
342                 && bill.State == State;
343         }
344 
345         #endregion
346 
347         #region ICloneable Members
348 
349         public Bill CopyTo(Bill bill)
350         {
351             if (bill == null)
352             {
353                 throw new ArgumentException("Null parameter not
allowed");
354             }
355 
356             bill.BillId = BillId;
357             bill.FinishedTime = FinishedTime;
358             bill.OrdersSet = OrdersSet;
359             bill.Session = Session != null ?
360                 Session.Clone() as Session
361                 : null;
362 
363             bill.StartedTime = StartedTime;
364             bill.State = State;
365             bill.TablesSet = TablesSet;
366             bill.Total = Total;
367 
368             return bill;
369         }
370 
371         public override object CopyTo(object template)
372         {
373             return CopyTo(template as Bill);
374         }
375 
376         #endregion
377     }
378 }
```

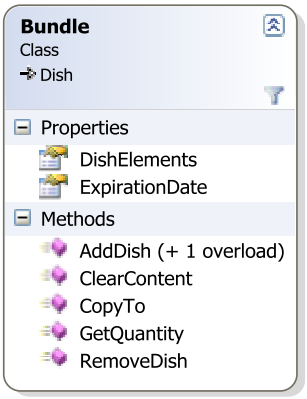
Enum `Restomatic.DomainModel.Bills.BillState.`



BillState.cs

```
 1 using System;
 2 using System.Runtime.Serialization;
 3
 4 namespace Restomatic.DomainModel.Bills
 5 {
 6     [DataContract(Namespace = "http://restomatic.azc.uam.mx/")]
 7     public enum BillState : byte
 8     {
 9         [EnumMember]
10         New,
11
12         [EnumMember]
13         Open,
14
15         [EnumMember]
16         Closed,
17
18         [EnumMember]
19         Paid,
20
21         [EnumMember]
22         Canceled
23     }
24 }
```

Class Restomatic.DomainModel.Menu.Bundle.



Bundle.cs

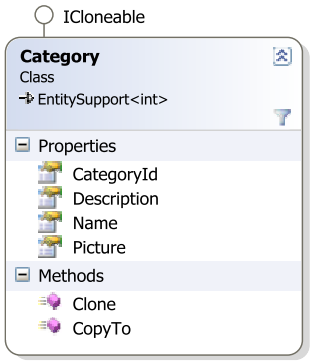
```
 1 using System;
 2 using System.Runtime.Serialization;
 3 using System.Collections.Generic;
 4
 5 namespace Restomatic.DomainModel.Menu
 6 {
 7     [Serializable]
 8     [DataContract(Namespace="http://restomatic.azc.uam.mx/")]
 9     public class Bundle : Dish
10     {
11         #region Properties
12
13         private DateTime expirationDate;
14
15         [DataMember]
16         public virtual DateTime ExpirationDate
17         {
18             get
19             {
20                 return expirationDate;
21             }
22
23             set
24             {
25                 expirationDate = value >= DateTime.Today ? value :
DateTime.Today;
26                 NotifyChanged("ExpirationDate");
27             }
28         }
29
30         private Dictionary<Dish, int> content;
31
32         [DataMember]
33         protected virtual Dictionary<Dish, int> Content
34         {
35             get
36             {
37                 if (content == null)
38                 {
```

```
39                      lock (this)
40                      {
41                          content = new Dictionary<Dish, int>();
42                      }
43                  }
44
45                  return content;
46              }
47
48              private set
49              {
50                  content = value != null ? new Dictionary<Dish,
int>(value) : null;
51
52                  NotifyChanged("Content");
53                  NotifyChanged("DishElements");
54              }
55          }
56
57          public virtual IEnumerable<Dish> DishElements
58          {
59              get { return Content.Keys; }
60          }
61
62          #endregion
63
64          #region Constructors
65
66          public Bundle() : this("") { }
67
68          public Bundle(string name) : this(name, "") { }
69
70          public Bundle(string name, string description)
71              : this(name, description, 0M) { }
72
73          public Bundle(string name, string description, decimal price)
74              : base(name, description, price) { }
75
76          public Bundle(Dictionary<Dish, int> dishes)
77              : this()
78          {
79              Content = dishes;
80
81              foreach (Dish dish in dishes.Keys)
82              {
83                  Price += dishes[Dish] * Dish.Price;
84              }
85          }
86
87          #endregion
88
89          #region Public Methods
90
91          public virtual void AddDish(Dish dish)
92          {
93              AddDish(dish, 1);
94          }
95
96          public virtual void AddDish(Dish dish, int quantity)
97          {
```

```
98              if(Content.ContainsKey(dish))
99              {
100                 RemoveDish(dish);
101             }
102
103             Price += dish.Price * quantity;
104             Content.Add(dish, quantity);
105         }
106
107         public virtual int GetQuantity(Dish dish)
108         {
109             return Content.ContainsKey(dish) ? Content[dish] : 0;
110         }
111
112         public virtual void RemoveDish(Dish dish)
113         {
114             if (Content.ContainsKey(dish))
115             {
116                 int q = Content[dish];
117                 Price -= Dish.Price * q;
118
119                 Content.Remove(dish);
120             }
121         }
122
123         public virtual int this[Dish dish]
124         {
125             get
126             {
127                 return GetQuantity(dish);
128             }
129
130             set
131             {
132                 RemoveDish(dish);
133                 AddDish(dish, value);
134             }
135         }
136
137         public virtual void ClearContent()
138         {
139             Content.Clear();
140             Price = 0M;
141         }
142
143         #endregion
144
145         #region Context Equality
146
147         protected override bool ContextAwareEquals(object obj)
148         {
149             Bundle bundle = obj as Bundle;
150
151             return base.ContextAwareEquals(obj)
152                 && bundle != null
153                 && bundle.Content.Keys.Equals(Content.Keys);
154         }
155
156         #endregion
157
```

```
158         #region ICloneable Members
159
160         public virtual Bundle CopyTo(Bundle bundle)
161         {
162             // Dish members
163             bundle.DishId = DishId;
164             bundle.Name = Name;
165             bundle.Description = Description;
166             bundle.Category = Category != null
167                 ? Category.Clone() as Category
168                 : null;
169
170             bundle.Price = Price;
171             bundle.Visible = Visible;
172             bundle.Picture = Picture != null
173                 ? Picture.Clone() as DishPicture
174                 : null;
175
176             // Bundle members
177             bundle.ExpirationDate = ExpirationDate;
178             bundle.Content = Content;
179
180             return bundle;
181         }
182         #endregion
183     }
184 }
```

Class Restomatic.DomainModel.Menu.Category.



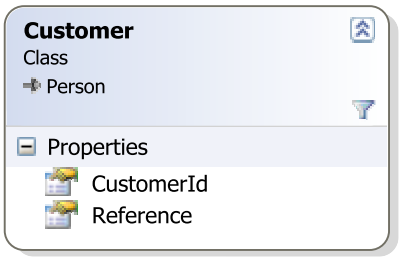
Category.cs

```
1  using System;
2  using System.Runtime.Serialization;
3  using System.Collections.Generic;
4
5  using Restomatic.DomainModel.Common;
6
7  namespace Restomatic.DomainModel.Menu
8  {
9      [Serializable]
10     [DataContract(Namespace="http://restomatic.azc.uam.mx/")]
11     public class Category : EntitySupport<int>
12     {
13         #region Properties
14
15         [DataMember]
16         public virtual int CategoryId
17         {
18             get
19             {
20                 return Id;
21             }
22
23             protected set
24             {
25                 Id = value;
26                 NotifyChanged("CategoryId");
27             }
28         }
29
30         private string name;
31
32         [DataMember]
33         public virtual string Name
34         {
35             get
36             {
37                 if (name == null)
38                 {
39                     name = "";
40                 }
41
42                 return name;
43             }
```

```
44
45              set
46              {
47                  name = value;
48                  NotifyChanged("Name");
49              }
50          }
51
52          private string description;
53
54          [DataMember]
55          public virtual string Description
56          {
57              get
58              {
59                  if (description == null)
60                  {
61                      description = "";
62                  }
63
64                  return description;
65              }
66
67              set
68              {
69                  description = value;
70                  NotifyChanged("Description");
71              }
72          }
73
74          private CategoryPicture picture;
75
76          [DataMember]
77          public virtual CategoryPicture Picture
78          {
79              get
80              {
81                  return picture;
82              }
83
84              set
85              {
86                  picture = value;
87                  NotifyChanged("Picture");
88              }
89          }
90
91          #endregion
92
93          #region Constructors
94
95          public Category() : this("") { }
96
97          public Category(string name)
98          {
99              Name = name;
100             Picture = new CategoryPicture();
101         }
102
103         #endregion
```

```
104
105         #region ContextEquality
106
107         protected override bool ContextAwareEquals(object obj)
108         {
109             Category category = obj as Category;
110             return category != null && category.Name == Name;
111         }
112
113         #endregion
114
115         #region ICloneable Members
116
117         public Category CopyTo(Category category)
118         {
119             if (category == null)
120             {
121                 throw new ArgumentException("Null parameter is not
allowed");
122             }
123
124             category.CategoryId = CategoryId;
125             category.Name = Name;
126             category.Description = Description;
127             category.Picture = Picture != null
128                 ? Picture.Clone() as CategoryPicture
129                 : null;
130
131             return category;
132         }
133
134         public override object CopyTo(object template)
135         {
136             return CopyTo(template as Category);
137         }
138
139         #endregion
140     }
141 }
```

Class `Restomatic.DomainModel.Common.Customer.`

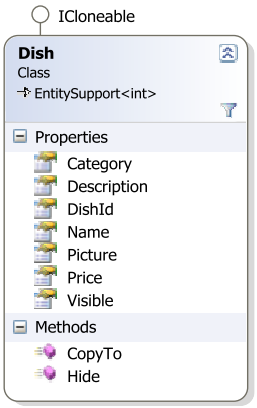


`Customer.cs`

```
 1 using System;
 2 using System.Runtime.Serialization;
 3 using System.Collections.Generic;
 4
 5 namespace Restomatic.DomainModel.Common
 6 {
 7     [Serializable]
 8     [DataContract(Namespace="http://restomatic.azc.uam.mx/")]
 9     public class Customer : Person
10     {
11         #region Properties
12
13         [DataMember]
14         public virtual int CustomerId
15         {
16             get
17             {
18                 return Id;
19             }
20
21             protected set
22             {
23                 Id = value;
24                 NotifyChanged("CustomerId");
25             }
26         }
27
28         private string reference;
29
30         [DataMember]
31         public virtual string Reference
32         {
33             get
34             {
35                 if (reference == null)
36                 {
37                     reference = "";
38                 }
39
40                 return reference;
41             }
42
43             set
44             {
```

```
45                  reference = value;
46                  NotifyChanged("Reference");
47              }
48          }
49
50          #endregion
51
52          #region Constructors
53
54          public Customer() : this("", "") { }
55
56          public Customer(string name, string lastName)
57              : base(name, lastName) { }
58
59          #endregion
60
61          #region ICloneable Members
62
63          public Customer CopyTo(Customer customer)
64          {
65              if (customer == null)
66              {
67                  throw new ArgumentException("Null parameter not
allowed");
68              }
69
70              customer.CustomerId = CustomerId;
71              customer.LastName = LastName;
72              customer.Name = Name;
73              customer.Reference = Reference;
74
75              return customer;
76          }
77
78          public override object CopyTo(object template)
79          {
80              return CopyTo(template as Customer);
81          }
82
83          #endregion
84      }
85  }
```

Class `Restomatic.DomainModel.Menu.Dish.`



Dish.cs

```
 1 using System;
 2 using System.Runtime.Serialization;
 3 using System.Collections.Generic;
 4
 5 using Restomatic.DomainModel.Common;
 6
 7 namespace Restomatic.DomainModel.Menu
 8 {
 9     [Serializable]
10     [DataContract(Namespace="http://restomatic.azc.uam.mx/")]
11     public class Dish : EntitySupport<int>
12     {
13         #region Properties
14
15         [DataMember]
16         public virtual int DishId
17         {
18             get
19             {
20                 return Id;
21             }
22
23             protected set
24             {
25                 Id = value;
26                 NotifyChanged("DishId");
27             }
28         }
29
30         private string name;
31
32         [DataMember]
33         public virtual string Name
34         {
35             get
36             {
37                 if (name == null)
38                 {
39                     name = "";
```

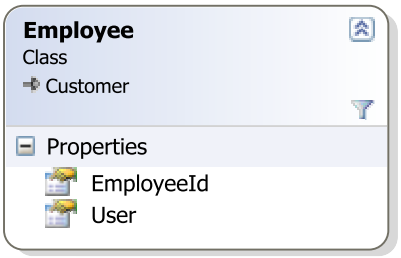
```
40                  }
41
42                  return name;
43              }
44
45              set
46              {
47                  name = value;
48                  NotifyChanged("Name");
49              }
50          }
51
52          private string description;
53
54          [DataMember]
55          public virtual string Description
56          {
57              get
58              {
59                  if(description == null)
60                  {
61                      description = "";
62                  }
63
64                  return description;
65              }
66
67              set
68              {
69                  description = value;
70                  NotifyChanged("Description");
71              }
72          }
73
74          private Category category;
75
76          [DataMember]
77          public virtual Category Category
78          {
79              get
80              {
81                  return category;
82              }
83
84              set
85              {
86                  category = value;
87                  NotifyChanged("Category");
88              }
89          }
90
91          private decimal price;
92
93          [DataMember]
94          public virtual decimal Price
95          {
96              get
97              {
98                  return price;
99              }
```

```
100
101             set
102             {
103                 price = value;
104                 NotifyChanged("Price");
105             }
106         }
107
108         private bool visible;
109
110         [DataMember]
111         public virtual bool Visible
112         {
113             get
114             {
115                 return visible;
116             }
117
118             protected set
119             {
120                 visible = value;
121                 NotifyChanged("Visible");
122             }
123         }
124
125         private DishPicture picture;
126
127         [DataMember]
128         public virtual DishPicture Picture
129         {
130             get
131             {
132                 return picture;
133             }
134
135             set
136             {
137                 picture = value;
138                 NotifyChanged("Picture");
139             }
140         }
141
142         #endregion
143
144         #region Constructors
145
146         public Dish() : this("") { }
147
148         public Dish(string name) : this(name, "") { }
149
150         public Dish(string name, string description)
151             : this(name, description, 0M) { }
152
153         public Dish(string name, string description, decimal price)
154         {
155             Name = name;
156             Description = description;
157             Price = price;
158             Visible = true;
159             Picture = new DishPicture();
```

```
160         }
161
162         #endregion
163
164         #region Other Methods
165
166         public virtual void Hide()
167         {
168             Visible = false;
169         }
170
171         #endregion
172
173         #region Context Equality
174
175         protected override bool ContextAwareEquals(object obj)
176         {
177             Dish dish = obj as Dish;
178
179             return dish != null
180                 && dish.Name == Name
181                 && dish.Price == Price
182                 && dish.Visible == Visible;
183         }
184
185         #endregion
186
187         #region ICloneable Members
188
189         public virtual Dish CopyTo(Dish dish)
190         {
191             if (dish == null)
192             {
193                 throw new ArgumentException("Null parameter is not
allowed");
194             }
195
196             dish.DishId = DishId;
197             dish.Name = Name;
198             dish.Description = Description;
199             dish.Category = Category != null
200                 ? Category.Clone() as Category
201                 : null;
202
203             dish.Price = Price;
204             dish.Visible = Visible;
205             dish.Picture = Picture != null
206                 ? Picture.Clone() as DishPicture
207                 : null;
208
209             return dish;
210         }
211
212         public override object CopyTo(object template)
213         {
214             return CopyTo(template as Dish);
215         }
216
217         #endregion
```

```
218     }
219 }
```

Class Restomatic.DomainModel.Common.Employee.

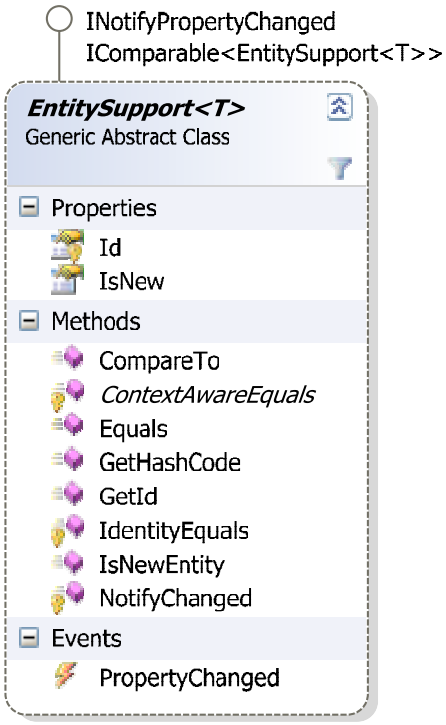


Employee.cs

```
 1 using System;
 2 using System.Runtime.Serialization;
 3 using System.Collections.Generic;
 4
 5 using Restomatic.DomainModel.Security;
 6
 7 namespace Restomatic.DomainModel.Common
 8 {
 9     [Serializable]
10     [DataContract(Namespace="http://restomatic.azc.uam.mx/")]
11     public class Employee : Person
12     {
13         #region Properties
14
15         [DataMember]
16         public virtual int EmployeeId
17         {
18             get
19             {
20                 return Id;
21             }
22
23             protected set
24             {
25                 Id = value;
26                 NotifyChanged("EmployeeId");
27             }
28         }
29
30         private User user;
31
32         [DataMember]
33         public virtual User User
34         {
35             get
36             {
37                 return user;
38             }
39
40             set
41             {
42                 user = value;
43                 NotifyChanged("User");
44             }
```

```
45          }
46
47          #endregion
48
49          #region Constructors
50
51          public Employee() : this("", "") { }
52
53          public Employee(string name, string lastName)
54              : base(name, lastName) { }
55
56          #endregion
57
58          #region ICloneable Members
59
60          public Employee CopyTo(Employee employee)
61          {
62              if (employee == null)
63              {
64                  throw new ArgumentException("Null parameter not
allowed");
65              }
66
67              employee.EmployeeId = EmployeeId;
68              employee.LastName = LastName;
69              employee.Name = Name;
70              employee.User = User != null
71                  ? User.Clone() as User
72                  : null;
73
74              return employee;
75          }
76
77          public override object CopyTo(object template)
78          {
79              return CopyTo(template as Employee);
80          }
81
82          #endregion
83      }
84  }
```

Class Restomatic.DomainModel.EntitySupport.



EntitySupport.cs

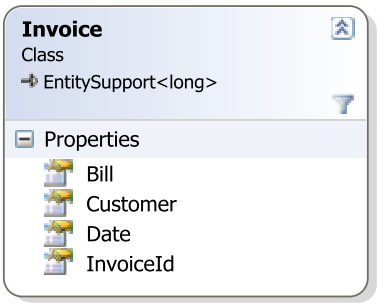
```
    1 using System;
    2 using System.Collections.Generic;
    3 using System.Runtime.Serialization;
    4 using System.ComponentModel;
    5 using System.Reflection;
    6
    7 namespace Restomatic.DomainModel
    8 {
    9     [Serializable]
   10     [DataContract(Namespace = "http://restomatic.azc.uam.mx/")]
   11     public abstract class EntitySupport<T> : INotifyPropertyChanged,
ICloneable, IComparable<EntitySupport<T>>
   12         where T : struct, IConvertible, IComparable<T>
   13     {
   14         #region Static Methods
   15
   16         public static bool IsNewEntity(object entity)
   17         {
   18             try
   19             {
   20                 Type KeyType =
entity.GetType().BaseType.GetGenericArguments()[0];
   21                 Type EntitySupportType =
typeof(EntitySupport<>).MakeGenericType(KeyType);
   22
   23                 if (entity != null &&
entity.GetType().IsSubclassOf(EntitySupportType))
```

```
   24                 {
   25                     PropertyInfo property =
entity.GetType().GetProperty("IsNew");
   26                     return (bool)property.GetValue(entity, null);
   27                 }
   28             }
   29             catch { }
   30
   31             throw new ArgumentException("The object must be a subclass of
EntitySupport");
   32         }
   33
   34         public static object GetId(object entity)
   35         {
   36             try
   37             {
   38                 Type KeyType =
entity.GetType().BaseType.GetGenericArguments()[0];
   39                 Type EntitySupportType =
typeof(EntitySupport<>).MakeGenericType(KeyType);
   40
   41                 if (entity != null &&
entity.GetType().IsSubclassOf(EntitySupportType))
   42                 {
   43                     PropertyInfo property =
entity.GetType().GetProperty("Id");
   44                     return property.GetValue(entity, null);
   45                 }
   46             }
   47             catch { }
   48
   49             throw new ArgumentException("The object must be a subclass of
EntitySupport");
   50         }
   51
   52         #endregion
   53
   54         #region Property
   55
   56         private T id;
   57
   58         protected virtual T Id
   59         {
   60             get { return id; }
   61
   62             set { id = value; }
   63         }
   64
   65         public virtual bool IsNew
   66         {
   67             get
   68             {
   69                 return Id.Equals(default(T));
   70             }
   71         }
   72
   73         #endregion
   74
   75         #region Equality and HashCode
   76
```

```
77          public override bool Equals(object obj)
78          {
79              if (obj == null || !GetType().Equals(obj.GetType()))
80              {
81                  return false;
82              }
83
84              return IdentityEquals(obj as EntitySupport<T>) ||
ContextAwareEquals(obj);
85          }
86
87          public override int GetHashCode()
88          {
89              return Convert.ToInt32(Id);
90          }
91
92          protected virtual bool IdentityEquals(EntitySupport<T> entity)
93          {
94              return !Id.Equals(default(T)) && Id.Equals(entity.Id);
95          }
96
97          protected abstract bool ContextAwareEquals(object obj);
98
99          #endregion
100
101         #region INotifyPropertyChanged Members
102
103         protected void NotifyChanged(string property)
104         {
105             try
106             {
107                 if (PropertyChanged != null)
108                 {
109                     PropertyChanged(this, new
PropertyChangedEventArgs(property));
110                 }
111             }
112             catch { }
113         }
114
115         public event PropertyChangedEventHandler PropertyChanged;
116
117         #endregion
118
119         #region IComparable<EntitySupport<T>> Members
120
121         public int CompareTo(EntitySupport<T> other)
122         {
123             T currentId = Id;
124             T otherId = (T)GetId(other);
125
126             return currentId.CompareTo(otherId);
127         }
128
129         #endregion
130
131         #region ICloneable Members
132
133         public abstract object CopyTo(object template);
134
```

```
135         public object Clone()
136         {
137             return CopyTo(Activator.CreateInstance(GetType()));
138         }
139 
140         #endregion
141     }
142 }
```

Class Restomatic.DomainModel.Bills.Invoice.



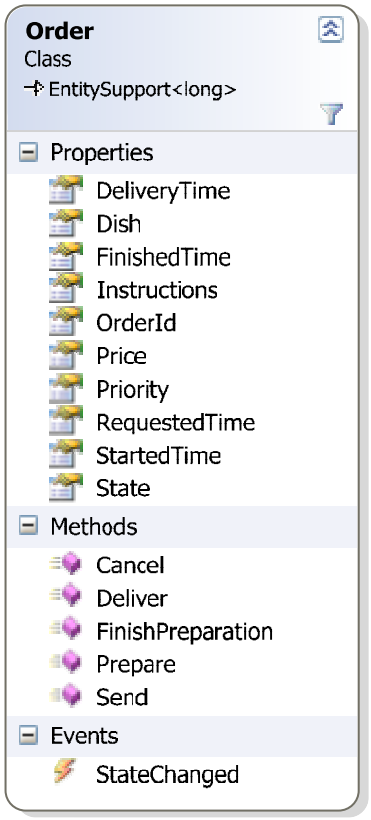
Invoice.cs

```
 1 using System;
 2 using System.Runtime.Serialization;
 3 using System.Collections.Generic;
 4
 5 using Restomatic.DomainModel.Common;
 6
 7 namespace Restomatic.DomainModel.Bills
 8 {
 9     [Serializable]
10     [DataContract(Namespace="http://restomatic.azc.uam.mx/")]
11     public class Invoice : EntitySupport<long>
12     {
13         #region Properties
14
15         [DataMember]
16         public virtual long InvoiceId
17         {
18             get
19             {
20                 return Id;
21             }
22
23             protected set
24             {
25                 Id = value;
26                 NotifyChanged("InvoiceId");
27             }
28         }
29
30         private Bill bill;
31
32         [DataMember]
33         public virtual Bill Bill
34         {
35             get
36             {
37                 return bill;
38             }
39
40             set
41             {
42                 bill = value;
43                 NotifyChanged("Bill");
```

```
44              }
45          }
46
47          private Customer customer;
48
49          [DataMember]
50          public virtual Customer Customer
51          {
52              get
53              {
54                  return customer;
55              }
56
57              set
58              {
59                  customer = value;
60                  NotifyChanged("Customer");
61              }
62          }
63
64          private DateTime date;
65
66          [DataMember]
67          public virtual DateTime Date
68          {
69              get
70              {
71                  return date;
72              }
73
74              set
75              {
76                  date = value;
77                  NotifyChanged("Date");
78              }
79          }
80
81          #endregion
82
83          #region Constructors
84
85          public Invoice() { }
86
87          public Invoice(Customer customer, Bill bill)
88          {
89              Customer = customer;
90              Bill = bill;
91          }
92
93          #endregion
94
95          #region ContextEquality
96
97          protected override bool ContextAwareEquals(object obj)
98          {
99              Invoice invoice = obj as Invoice;
100
101             return invoice != null
102                 && invoice.Customer != null
103                 && invoice.Customer.Equals(Customer)
```

```
104                 && invoice.Bill != null
105                 && invoice.Bill.Equals(Bill);
106         }
107
108         #endregion
109
110         #region ICloneable Members
111
112         public Invoice CopyTo(Invoice invoice)
113         {
114             if (invoice == null)
115             {
116                 throw new ArgumentException("Null parameter not
allowed");
117             }
118
119             invoice.Bill = Bill != null
120                 ? Bill.Clone() as Bill
121                 : null;
122
123             invoice.Customer = Customer != null
124                 ? Customer.Clone() as Customer
125                 : null;
126
127             invoice.Date = Date;
128             invoice.InvoiceId = InvoiceId;
129
130             return invoice;
131         }
132
133         public override object CopyTo(object template)
134         {
135             return CopyTo(template as Invoice);
136         }
137
138         #endregion
139     }
140 }
```

Class `Restomatic.DomainModel.Bills.Order`.



Order.cs

```
 1 using System;
 2 using System.Runtime.Serialization;
 3 using System.Collections.Generic;
 4
 5 using Restomatic.DomainModel.Common;
 6 using Restomatic.DomainModel.Menu;
 7
 8 namespace Restomatic.DomainModel.Bills
 9 {
10     [Serializable]
11     [DataContract(Namespace="http://restomatic.azc.uam.mx/")]
12     public class Order : EntitySupport<long>
13     {
14         #region Properties
15
16         [DataMember]
17         public virtual long OrderId
18         {
19             get
20             {
21                 return Id;
22             }
23
```

```
24              protected set
25              {
26                  Id = value;
27                  NotifyChanged("OrderId");
28              }
29          }
30
31          private DateTime requestedTime;
32
33          [DataMember]
34          public virtual DateTime RequestedTime
35          {
36              get
37              {
38                  return requestedTime;
39              }
40
41              set
42              {
43                  requestedTime = value;
44                  NotifyChanged("RequestedTime");
45              }
46          }
47
48          private DateTime startedTime;
49
50          [DataMember]
51          public virtual DateTime StartedTime
52          {
53              get
54              {
55                  return startedTime;
56              }
57
58              set
59              {
60                  startedTime = value;
61                  NotifyChanged("StartedTime");
62              }
63          }
64
65          private DateTime finishedTime;
66
67          [DataMember]
68          public virtual DateTime FinishedTime
69          {
70              get
71              {
72                  return finishedTime;
73              }
74
75              set
76              {
77                  finishedTime = value;
78                  NotifyChanged("FinishedTime");
79              }
80          }
81
82          private DateTime deliveryTime;
83
```

```
84          [DataMember]
85          public virtual DateTime DeliveryTime
86          {
87              get
88              {
89                  return deliveryTime;
90              }
91
92              set
93              {
94                  deliveryTime = value;
95                  NotifyChanged("DeliveryTime");
96              }
97          }
98
99          private OrderState state;
100
101         [DataMember]
102         public virtual OrderState State
103         {
104             get
105             {
106                 return state;
107             }
108
109             set
110             {
111                 state = value;
112                 NotifyChanged("State");
113             }
114         }
115
116         private OrderPriority priority;
117
118         [DataMember]
119         public virtual OrderPriority Priority
120         {
121             get
122             {
123                 return priority;
124             }
125
126             set
127             {
128                 priority = value;
129                 NotifyChanged("Priority");
130             }
131         }
132
133         private Dish dish;
134
135         [DataMember]
136         public virtual Dish Dish
137         {
138             get
139             {
140                 return dish;
141             }
142
143             set
```

```
144             {
145                 dish = value;
146 
147                 NotifyChanged("Dish");
148                 NotifyChanged("Price");
149             }
150         }
151 
152         private string instructions;
153 
154         [DataMember]
155         public virtual string Instructions
156         {
157             get
158             {
159                 if (instructions == null)
160                 {
161                     instructions = "";
162                 }
163 
164                 return instructions;
165             }
166 
167             set
168             {
169                 instructions = value;
170                 NotifyChanged("Instructions");
171             }
172         }
173 
174         public virtual decimal Price
175         {
176             get
177             {
178                 return Dish != null ? Dish.Price : 0M;
179             }
180         }
181 
182         #endregion
183 
184         #region Events
185 
186         private event EventHandler<OrderStateChangedEventArgs>
stateChanged;
187 
188         public virtual event EventHandler<OrderStateChangedEventArgs>
StateChanged
189         {
190             add
191             {
192                 stateChanged += value;
193             }
194 
195             remove
196             {
197                 stateChanged -= value;
198             }
199         }
200 
201         #endregion
```

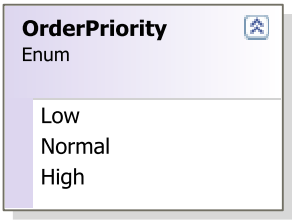
```
202
203         #region Constructors
204
205         public Order() : this("") { }
206
207         public Order(string suggestion)
208             : this(suggestion, OrderPriority.Normal)
209         { }
210
211         public Order(OrderPriority priority) : this("", priority) { }
212
213         public Order(string suggestion, OrderPriority priority)
214         {
215             RequestedTime = DateTime.Now.AddMinutes(1);
216             StartedTime = DateTime.Now.AddMinutes(10);
217             FinishedTime = DateTime.Now.AddMinutes(30);
218             DeliveryTime = DateTime.Now.AddMinutes(35);
219
220             Instructions = suggestion;
221             Priority = priority;
222             State = OrderState.New;
223         }
224
225         #endregion
226
227         #region Private State Management Support Methods
228
229         private void CheckCanceledOrder()
230         {
231             if (State == OrderState.Canceled)
232             {
233                 throw new InvalidOperationException("State is not
Canceled");
234             }
235         }
236
237         private void CheckNewState()
238         {
239             if (State != OrderState.New)
240             {
241                 throw new InvalidOperationException("State is not New");
242             }
243         }
244
245         private void CheckSentState()
246         {
247             if (State != OrderState.Sent)
248             {
249                 throw new InvalidOperationException("State is not Sent");
250             }
251         }
252
253         private void CheckPreparingState()
254         {
255             if (State != OrderState.Preparing)
256             {
257                 throw new InvalidOperationException("State is not
Preparing");
258             }
259         }
```

```
260
261         private void CheckPreparedState()
262         {
263             if (State != OrderState.Prepared)
264             {
265                 throw new InvalidOperationException("State is not
Prepared");
266             }
267         }
268
269         #endregion
270
271         #region State Management Methods
272
273         public virtual void Send()
274         {
275             CheckNewState();
276             CheckCanceledOrder();
277             State = OrderState.Sent;
278             RequestedTime = DateTime.Now;
279             RaiseEvent(State, OrderState.Sent);
280         }
281
282         public virtual void Prepare()
283         {
284             CheckSentState();
285             CheckCanceledOrder();
286             State = OrderState.Preparing;
287             StartedTime = DateTime.Now;
288             RaiseEvent(State, OrderState.Preparing);
289         }
290
291         public virtual void FinishPreparation()
292         {
293             CheckPreparingState();
294             CheckCanceledOrder();
295             State = OrderState.Prepared;
296             FinishedTime = DateTime.Now;
297             RaiseEvent(State, OrderState.Prepared);
298         }
299
300         public virtual void Deliver()
301         {
302             CheckPreparedState();
303             CheckCanceledOrder();
304             State = OrderState.Delivered;
305             DeliveryTime = DateTime.Now;
306             RaiseEvent(State, OrderState.Delivered);
307         }
308
309         public virtual void Cancel()
310         {
311             State = OrderState.Canceled;
312             FinishedTime = DateTime.Now;
313             RaiseEvent(State, OrderState.Canceled);
314         }
315
316         #endregion
317
318         #region Other Support Methods
319
```

```
  320         private void RaiseEvent(OrderState oldState, OrderState newState)
  321         {
  322             if (stateChanged != null)
  323             {
  324                 try
  325                 {
  326                     stateChanged(this,
  327                         new OrderStateChangedEventArgs(oldState,
newState));
  328                 }
  329                 catch { }
  330             }
  331         }
  332 
  333         #endregion
  334 
  335         #region Context Equality
  336 
  337         protected override bool ContextAwareEquals(object obj)
  338         {
  339             Order order = obj as Order;
  340 
  341             return order != null
  342                 && order.RequestedTime == RequestedTime
  343                 && order.State == State
  344                 && order.Priority == Priority
  345                 && order.Dish != null
  346                 && order.Dish.Equals(Dish);
  347         }
  348 
  349         #endregion
  350 
  351         #region ICloneable Members
  352 
  353         public Order CopyTo(Order order)
  354         {
  355             if (order == null)
  356             {
  357                 throw new ArgumentException("Null parameter not
allowed");
  358             }
  359 
  360             order.DeliveryTime = DeliveryTime;
  361             order.Dish = Dish != null
  362                 ? Dish.Clone() as Dish
  363                 : null;
  364 
  365             order.FinishedTime = FinishedTime;
  366             order.Instructions = Instructions;
  367             order.OrderId = OrderId;
  368             order.Priority = Priority;
  369             order.RequestedTime = RequestedTime;
  370             order.StartedTime = StartedTime;
  371             order.State = State;
  372 
  373             return order;
  374         }
  375 
  376         public override object CopyTo(object template)
  377         {
```

```
378             return CopyTo(template as Order);
379         }
380
381         #endregion
382     }
383 }
```

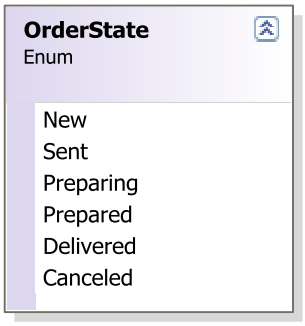
Enum `Restomatic.DomainModel.Bills.OrderPriority`.



`OrderPriority.cs`

```
 1 using System;
 2 using System.Runtime.Serialization;
 3
 4 namespace Restomatic.DomainModel.Bills
 5 {
 6     [DataContract(Namespace = "http://restomatic.azc.uam.mx/")]
 7     public enum OrderPriority
 8     {
 9         [EnumMember]
10         Low,
11
12         [EnumMember]
13         Normal,
14
15         [EnumMember]
16         High
17     }
18 }
```

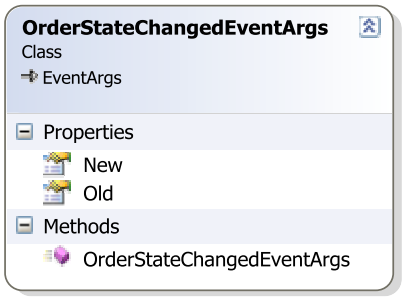
Enum Restomatic.DomainModel.Bills.OrderState.



OrderState.cs

```
 1 using System;
 2 using System.Runtime.Serialization;
 3
 4 namespace Restomatic.DomainModel.Bills
 5 {
 6     [DataContract(Namespace = "http://restomatic.azc.uam.mx/")]
 7     public enum OrderState
 8     {
 9         [EnumMember]
10         New,
11
12         [EnumMember]
13         Sent,
14
15         [EnumMember]
16         Preparing,
17
18         [EnumMember]
19         Prepared,
20
21         [EnumMember]
22         Delivered,
23
24         [EnumMember]
25         Canceled
26     }
27 }
```

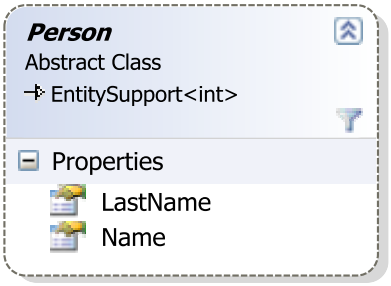
Class `Restomatic.DomainModel.OrderStateChangedEventArgs.`



`OrderStateChangedEventArgs.cs`

```
 1 using System;
 2 using System.Collections.Generic;
 3
 4 using Restomatic.DomainModel.Bills;
 5
 6 namespace Restomatic.DomainModel
 7 {
 8     public class OrderStateChangedEventArgs : EventArgs
 9     {
10         public OrderState Old { get; protected set; }
11         public OrderState New { get; protected set; }
12
13         public OrderStateChangedEventArgs(OrderState o, OrderState n)
14         {
15             Old = o;
16             New = n;
17         }
18     }
19 }
```

Enum Restomatic.DomainModel.Common.Person.

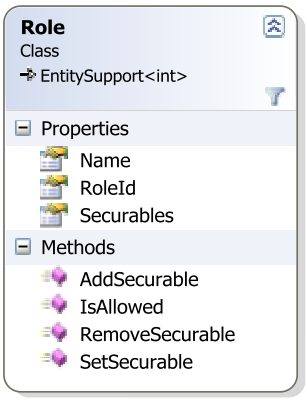


Person.cs

```
 1 using System;
 2 using System.Runtime.Serialization;
 3 using System.Collections.Generic;
 4
 5 namespace Restomatic.DomainModel.Common
 6 {
 7     [Serializable]
 8     [DataContract(Namespace="http://restomatic.azc.uam.mx/")]
 9     public abstract class Person: EntitySupport<int>
10     {
11         #region Properties
12
13         private string name;
14
15         [DataMember]
16         public virtual string Name
17         {
18             get
19             {
20                 if (name == null)
21                 {
22                     name = "";
23                 }
24
25                 return name;
26             }
27
28             set
29             {
30                 name = value;
31                 NotifyChanged("Name");
32             }
33         }
34
35         private string lastName;
36
37         [DataMember]
38         public virtual string LastName
39         {
40             get
41             {
42                 if (lastName == null)
43                 {
```

```
44                      lastName = "";
45                  }
46
47                  return lastName;
48              }
49
50              set
51              {
52                  lastName = value;
53                  NotifyChanged("LastName");
54              }
55          }
56
57          #endregion
58
59          #region Constructor
60
61          public Person() : this("", "") { }
62
63          public Person(string name, string lastName)
64          {
65              Name = name;
66              LastName = lastName;
67          }
68
69          #endregion
70
71          #region Context Equality
72
73          protected override bool ContextAwareEquals(object obj)
74          {
75              Person person = obj as Person;
76
77              return person != null
78                  && person.Name == Name
79                  && person.LastName == LastName;
80          }
81
82          #endregion
83      }
84  }
```

Class `Restomatic.DomainModel.Security.Role`.



Role.cs

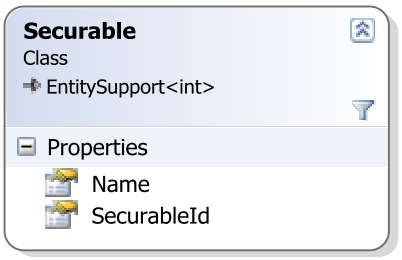
```
 1 using System;
 2 using System.Runtime.Serialization;
 3 using System.Collections.Generic;
 4
 5 using Restomatic.DomainModel.Common;
 6
 7 namespace Restomatic.DomainModel.Security
 8 {
 9     [Serializable]
10     [DataContract(Namespace="http://restomatic.azc.uam.mx/")]
11     public class Role : EntitySupport<int>
12     {
13         #region Properties
14
15         [DataMember]
16         public virtual int RoleId
17         {
18             get
19             {
20                 return Id;
21             }
22
23             protected set
24             {
25                 Id = value;
26                 NotifyChanged("RoleId");
27             }
28         }
29
30         private string name;
31
32         [DataMember]
33         public virtual string Name
34         {
35             get
36             {
37                 if (name == null)
38                 {
39                     name = "";
```

```
40                  }
41
42                  return name;
43              }
44
45              set
46              {
47                  name = value;
48                  NotifyChanged("Name");
49              }
50          }
51
52          private IDictionary<Securable, bool> securableMap;
53
54          [DataMember(Name = "Securables")]
55          protected IDictionary<Securable, bool> SecurableMap
56          {
57              get
58              {
59                  if (securableMap == null)
60                  {
61                      lock (this)
62                      {
63                          securableMap = new Dictionary<Securable,
bool>();
64                      }
65                  }
66
67                  return securableMap;
68              }
69
70              set
71              {
72                  securableMap = value != null ? new Dictionary<Securable,
bool>(value) : null;
73
74                  NotifyChanged("SecurableMap");
75                  NotifyChanged("Securables");
76              }
77          }
78
79          public virtual IEnumerable<Securable> Securables
80          {
81              get { return SecurableMap.Keys; }
82          }
83
84          #endregion
85
86          #region Constructors
87
88          public Role() { }
89
90          public Role(string name) { Name = name; }
91
92          #endregion
93
94          #region Other Methods
95
96          public virtual void AddSecurable(Securable securable, bool
allowed)
```

```
   97          {
   98              if (SecurableMap.ContainsKey(securable))
   99              {
  100                  SecurableMap.Add(securable, allowed);
  101              }
  102          }
  103
  104          public virtual void RemoveSecurable(Securable securable)
  105          {
  106              if (SecurableMap.ContainsKey(securable))
  107              {
  108                  SecurableMap.Remove(securable);
  109              }
  110          }
  111
  112          public virtual void SetSecurable(Securable securable, bool
allowed)
  113          {
  114              if (SecurableMap.ContainsKey(securable))
  115              {
  116                  RemoveSecurable(securable);
  117                  AddSecurable(securable, allowed);
  118              }
  119              else
  120              {
  121                  throw new InvalidOperationException("Securable not
present");
  122              }
  123          }
  124
  125          public virtual bool IsAllowed(Securable securable)
  126          {
  127              if (SecurableMap.ContainsKey(securable))
  128                  return SecurableMap[securable];
  129
  130              return false;
  131          }
  132
  133          #endregion
  134
  135          #region Contex Equality
  136
  137          protected override bool ContextAwareEquals(object obj)
  138          {
  139              Role role = obj as Role;
  140              return role != null && role.Name == Name;
  141          }
  142
  143          #endregion
  144
  145          #region ICloneable Members
  146
  147          public virtual Role CopyTo(Role role)
  148          {
  149              if (role == null)
  150              {
  151                  throw new ArgumentException("Null parameter is not
allowed");
  152              }
  153
  154              role.Name = Name;
```

```
155             role.RoleId = RoleId;
156             role.SecurableMap = SecurableMap;
157 
158             return role;
159         }
160 
161         public override object CopyTo(object template)
162         {
163             return CopyTo(template as Role);
164         }
165 
166         #endregion
167     }
168 }
```

Class Restomatic.DomainModel.Security.Securable.

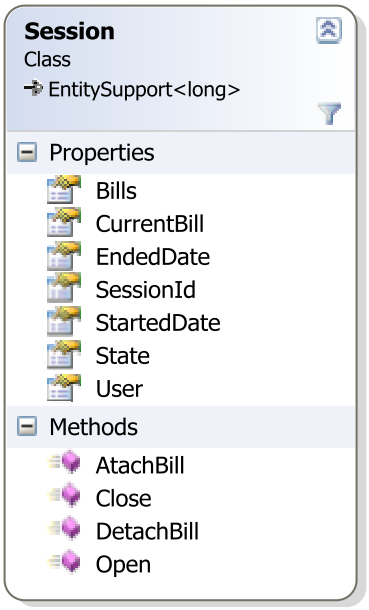


Securable.cs

```
 1 using System;
 2 using System.Runtime.Serialization;
 3 using System.Collections.Generic;
 4
 5 using Restomatic.DomainModel.Common;
 6
 7 namespace Restomatic.DomainModel.Security
 8 {
 9     [Serializable]
10     [DataContract(Namespace="http://restomatic.azc.uam.mx/")]
11     public class Securable : EntitySupport<int>
12     {
13         #region Properties
14
15         [DataMember]
16         public virtual int SecurableId
17         {
18             get
19             {
20                 return Id;
21             }
22
23             protected set
24             {
25                 Id = value;
26                 NotifyChanged("SecurableId");
27             }
28         }
29
30         private string name;
31
32         [DataMember]
33         public virtual string Name
34         {
35             get
36             {
37                 if (name == null)
38                 {
39                     name = "";
40                 }
41
42                 return name;
43             }
44
```

```
45              set
46              {
47                  name = value;
48                  NotifyChanged("Name");
49              }
50          }
51
52          #endregion
53
54          #region Constructors
55
56          public Securable() { }
57
58          public Securable(string name) { Name = name; }
59
60          #endregion
61
62          #region Context Equality
63
64          protected override bool ContextAwareEquals(object obj)
65          {
66              Securable securable = obj as Securable;
67              return securable != null && securable.Name == Name;
68          }
69
70          #endregion
71
72          #region ICloneable Members
73
74          public virtual Securable CopyTo(Securable securable)
75          {
76              if (securable == null)
77              {
78                  throw new ArgumentException("Null parameter is not
allowed");
79              }
80
81              securable.Name = Name;
82              securable.SecurableId = SecurableId;
83
84              return securable;
85          }
86
87          public override object CopyTo(object template)
88          {
89              return CopyTo(template as Securable);
90          }
91
92          #endregion
93      }
94  }
```

Class Restomatic.DomainModel.Security.Session.



Session.cs

```
 1 using System;
 2 using System.Runtime.Serialization;
 3 using System.Collections.Generic;
 4
 5 using Iesi.Collections.Generic;
 6
 7 using Restomatic.DomainModel.Common;
 8 using Restomatic.DomainModel.Bills;
 9
10 namespace Restomatic.DomainModel.Security
11 {
12     [Serializable]
13     [DataContract(Namespace="http://restomatic.azc.uam.mx/")]
14     public class Session : EntitySupport<long>
15     {
16         #region Properties
17
18         [DataMember]
19         public virtual long SessionId
20         {
21             get
22             {
23                 return Id;
24             }
25
26             protected set
27             {
28                 Id = value;
29                 NotifyChanged("SessionId");
30             }
31         }
```

```
32
33          private DateTime startedTime;
34
35          [DataMember]
36          public virtual DateTime StartedDate
37          {
38              get
39              {
40                  return startedTime;
41              }
42
43              set
44              {
45                  startedTime = value;
46                  NotifyChanged("StartedDate");
47              }
48          }
49
50          private DateTime endedDate;
51
52          [DataMember]
53          public virtual DateTime EndedDate
54          {
55              get
56              {
57                  return endedDate;
58              }
59
60              set
61              {
62                  endedDate = value;
63                  NotifyChanged("EndedDate");
64              }
65          }
66
67          private User user;
68
69          [DataMember]
70          public virtual User User
71          {
72              get
73              {
74                  return user;
75              }
76
77              set
78              {
79                  user = value;
80                  NotifyChanged("User");
81              }
82          }
83
84          private SessionState state;
85
86          [DataMember]
87          public virtual SessionState State
88          {
89              get
90              {
91                  return state;
```

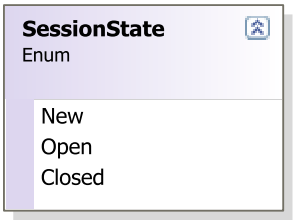
```
92              }
93
94              set
95              {
96                  state = value;
97                  NotifyChanged("State");
98              }
99          }
100
101         private ISet<Bill> billsCollection;
102
103         protected virtual ISet<Bill> BillsCollection
104         {
105             get
106             {
107                 if (billsCollection == null)
108                 {
109                     lock (this)
110                     {
111                         billsCollection = new HashedSet<Bill>();
112                     }
113                 }
114
115                 return billsCollection;
116             }
117
118             set
119             {
120                 billsCollection = value != null ? new
HashedSet<Bill>(value) : null;
121
122                 NotifyChanged("BillsCollection");
123                 NotifyChanged("Bills");
124             }
125         }
126
127         [DataMember]
128         public virtual IEnumerable<Bill> Bills
129         {
130             get
131             {
132                 return BillsCollection;
133             }
134
135             protected set
136             {
137                 BillsCollection = new HashedSet<Bill>(new
List<Bill>(value));
138             }
139         }
140
141         private Bill currentBill;
142
143         public virtual Bill CurrentBill
144         {
145             get
146             {
147                 return currentBill;
148             }
149
```

```
150             set
151             {
152                 currentBill = value;
153                 NotifyChanged("CurrentBill");
154             }
155         }
156 
157         #endregion
158 
159         #region Constructors
160 
161         public Session() : this(null) { }
162 
163         public Session(User user)
164         {
165             User = user;
166             StartedDate = DateTime.Now;
167             EndedDate = DateTime.MaxValue;
168         }
169 
170         #endregion
171 
172         #region Private State Management Support Methods
173 
174         private void CheckNewState()
175         {
176             if (State != SessionState.New)
177             {
178                 throw new InvalidOperationException("State is not New");
179             }
180         }
181 
182         private void CheckOpenState()
183         {
184             if (State != SessionState.Open)
185             {
186                 throw new InvalidOperationException("State is not Open");
187             }
188         }
189 
190         #endregion
191 
192         #region State Management Support Methods
193 
194         public virtual void Open()
195         {
196             CheckNewState();
197             State = SessionState.Open;
198             StartedDate = DateTime.Now;
199         }
200 
201         public virtual void Close()
202         {
203             CheckOpenState();
204             State = SessionState.Closed;
205             EndedDate = DateTime.Now;
206         }
207 
208         public virtual void AtachBill(Bill bill)
209         {
```

```
  210             if (bill != null && bill.HasSession() &&
!BillsCollection.Contains(bill))
  211             {
  212                 BillsCollection.Add(bill);
  213                 bill.Session = this;
  214             }
  215         }
  216
  217         public virtual void DetachBill(Bill bill)
  218         {
  219             if (bill != null && BillsCollection.Contains(bill))
  220             {
  221                 BillsCollection.Remove(bill);
  222                 bill.Session = null;
  223             }
  224         }
  225
  226         #endregion
  227
  228         #region Context Equality
  229
  230         protected override bool ContextAwareEquals(object obj)
  231         {
  232             Session session = obj as Session;
  233
  234             return session != null
  235                 && session.StartedDate == StartedDate
  236                 && session.State == State
  237                 && session.User != null
  238                 && session.User == User;
  239         }
  240
  241         #endregion
  242
  243         #region ICloneable Members
  244
  245         public virtual Session CopyTo(Session session)
  246         {
  247             if (session == null)
  248             {
  249                 throw new ArgumentException("Null parameter is not
allowed");
  250             }
  251
  252             session.BillsCollection = BillsCollection;
  253             session.CurrentBill = CurrentBill != null
  254                 ? CurrentBill.Clone() as Bill
  255                 : null;
  256
  257             session.EndedDate = EndedDate;
  258             session.SessionId = SessionId;
  259             session.StartedDate = StartedDate;
  260             session.State = State;
  261             session.User = User != null
  262                 ? User.Clone() as User
  263                 : null;
  264
  265             return session;
  266         }
  267
```

```
268         public override object CopyTo(object template)
269         {
270             return CopyTo(template as Session);
271         }
272 
273         #endregion
274     }
275 }
```

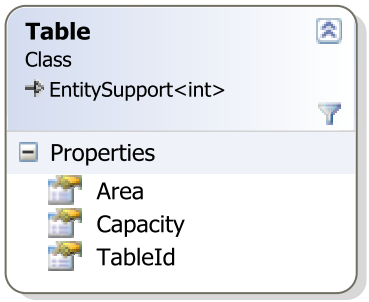
Enum Restomatic.DomainModel.Security.SessionState.



SessionState.cs

```
 1 using System;
 2 using System.Runtime.Serialization;
 3
 4 namespace Restomatic.DomainModel.Security
 5 {
 6     [DataContract(Namespace = "http://restomatic.azc.uam.mx/")]
 7     public enum SessionState
 8     {
 9         [EnumMember]
10         New,
11
12         [EnumMember]
13         Open,
14
15         [EnumMember]
16         Closed
17     }
18 }
```

Class `Restomatic.DomainModel.Common.Table.`



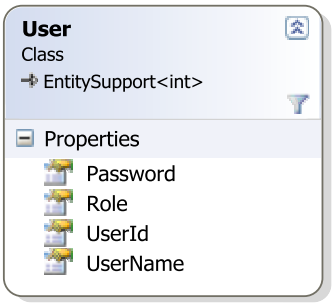
`Table.cs`

```
 1 using System;
 2 using System.Runtime.Serialization;
 3 using System.Collections.Generic;
 4
 5 namespace Restomatic.DomainModel.Common
 6 {
 7     [Serializable]
 8     [DataContract(Namespace="http://restomatic.azc.uam.mx/")]
 9     public class Table : EntitySupport<int>
10     {
11         #region Properties
12
13         [DataMember]
14         public virtual int TableId
15         {
16             get
17             {
18                 return Id;
19             }
20
21             protected set
22             {
23                 Id = value;
24                 NotifyChanged("TableId");
25             }
26         }
27
28         private Area area;
29
30         [DataMember]
31         public virtual Area Area
32         {
33             get
34             {
35                 return area;
36             }
37
38             set
39             {
40                 area = value;
41                 NotifyChanged("Area");
42             }
43         }
```

```
   44
   45          private int capacity;
   46
   47          [DataMember]
   48          public virtual int Capacity
   49          {
   50              get
   51              {
   52                  if (capacity < 1)
   53                  {
   54                      capacity = 1;
   55                  }
   56
   57                  return capacity;
   58              }
   59
   60              set
   61              {
   62                  capacity = value > 0 ? value : 1;
   63                  NotifyChanged("Capacity");
   64              }
   65          }
   66
   67          #endregion
   68
   69          #region Constructors
   70
   71          public Table() : this(Area.NoSmoking) { }
   72
   73          public Table(Area area) : this(area, 1) { }
   74
   75          public Table(int capacity) : this(Area.NoSmoking, capacity) { }
   76
   77          public Table(Area area, int capacity)
   78          {
   79              Area = area;
   80              Capacity = capacity;
   81          }
   82
   83          #endregion
   84
   85          #region Context Equality
   86
   87          protected override bool ContextAwareEquals(object obj)
   88          {
   89              Table table = obj as Table;
   90              return table != null && table.Area == Area && table.Capacity
== Capacity;
   91          }
   92
   93          #endregion
   94
   95          #region ICloneable Members
   96
   97          public Table CopyTo(Table table)
   98          {
   99              if (table == null)
  100              {
  101                  throw new ArgumentException("Null parameter not
allowed");
```

```
102             }
103
104             table.Area = Area;
105             table.Capacity = Capacity;
106             table.TableId = TableId;
107
108             return table;
109         }
110
111         public override object CopyTo(object template)
112         {
113             return CopyTo(template as Table);
114         }
115
116         #endregion
117     }
118 }
```

Class Restomatic.DomainModel.Security.User.



User.cs

```
 1 using System;
 2 using System.Runtime.Serialization;
 3 using System.Collections.Generic;
 4
 5 using Restomatic.DomainModel.Common;
 6
 7 namespace Restomatic.DomainModel.Security
 8 {
 9     [Serializable]
10     [DataContract(Namespace="http://restomatic.azc.uam.mx/")]
11     public class User : EntitySupport<int>
12     {
13         #region Properties
14
15         [DataMember]
16         public virtual int UserId
17         {
18             get
19             {
20                 return Id;
21             }
22
23             protected set
24             {
25                 Id = value;
26                 NotifyChanged("UserId");
27             }
28         }
29
30         private string userName;
31
32         [DataMember]
33         public virtual string UserName
34         {
35             get
36             {
37                 if (userName == null)
38                 {
39                     userName = "";
40                 }
41
42                 return userName;
43             }
```

```
44
45              set
46              {
47                  userName = value;
48                  NotifyChanged("UserName");
49              }
50          }
51
52          private string password;
53
54          [DataMember]
55          public virtual string Password
56          {
57              get
58              {
59                  if (password == null)
60                  {
61                      password = "";
62                  }
63
64                  return password;
65              }
66
67              set
68              {
69                  password = value;
70                  NotifyChanged("Password");
71              }
72          }
73
74          private Role role;
75
76          [DataMember]
77          public virtual Role Role
78          {
79              get
80              {
81                  return role;
82              }
83
84              set
85              {
86                  role = value;
87                  NotifyChanged("Role");
88              }
89          }
90
91          #endregion
92
93          #region Constructors
94
95          public User() : this("", "") { }
96
97          public User(string username, string password)
98              : this(username, password, null) { }
99
100         public User(string username, string password, Role role)
101         {
102             UserName = username;
103             Password = password;
```

```
104             Role = role;
105         }
106
107         #endregion
108
109         #region Context Equality
110
111         protected override bool ContextAwareEquals(object obj)
112         {
113             User user = obj as User;
114
115             return user != null
116                 && user.UserName != null
117                 && user.Password != null
118                 && user.UserName == UserName
119                 && user.Password == Password;
120         }
121
122         #endregion
123
124         #region ICloneable Members
125
126         public virtual User CopyTo(User user)
127         {
128             if (user == null)
129             {
130                 throw new ArgumentException("Null parameter is not
allowed");
131             }
132
133             user.Password = Password;
134             user.Role = Role != null
135                 ? Role.Clone() as Role
136                 : null;
137
138             user.UserId = UserId;
139             user.UserName = UserName;
140
141             return user;
142         }
143
144         public override object CopyTo(object template)
145         {
146             return CopyTo(template as User);
147         }
148
149         #endregion
150     }
151 }
```

Apéndice C

Esquema de base de datos.

Diagrama general del esquema de base de datos.

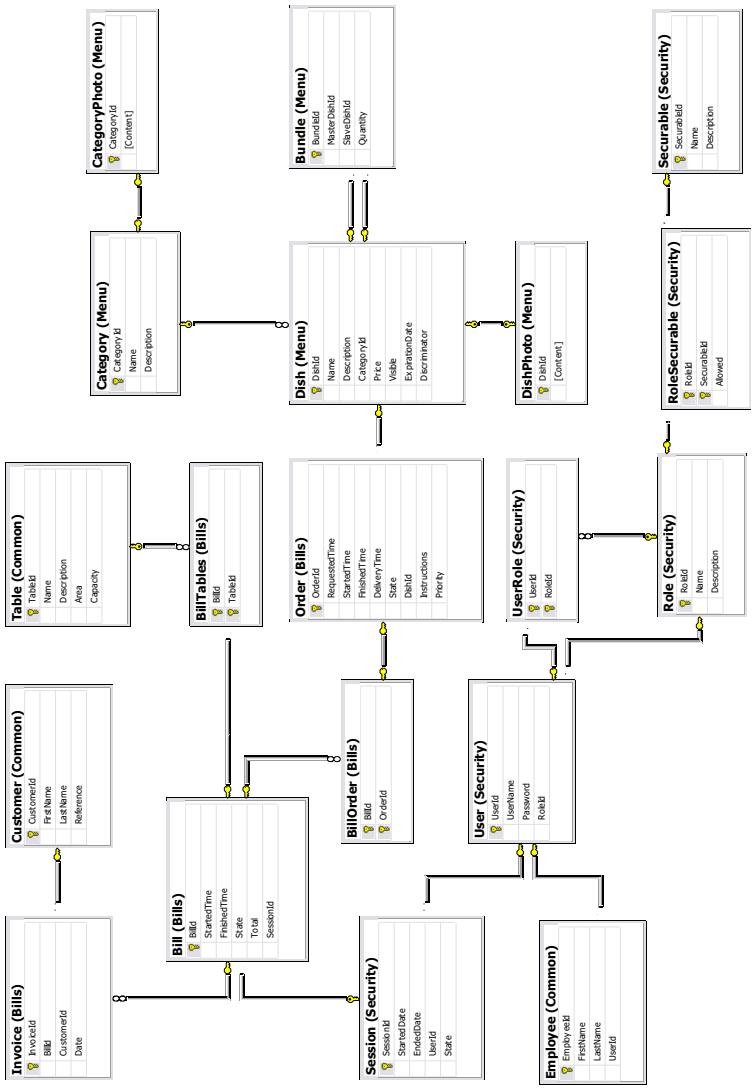


Figura C.1: Esquema de base de datos Restomatic.

Código fuente DDL-SQL del esquema de base de datos.

Entidad [Bills].[Bill].

```
CREATE TABLE [Bills].[Bill]
(
    [BillId] BIGINT NOT NULL IDENTITY(1,1),
    [StartedTime] DATETIME NOT NULL DEFAULT GETDATE(),
    [FinishedTime] DATETIME NULL DEFAULT GETDATE(),
    [State] TINYINT NOT NULL DEFAULT 0,
    [Total] MONEY NOT NULL DEFAULT 0,
    [SessionId] BIGINT NOT NULL,
    CONSTRAINT [PK_Bill] PRIMARY KEY CLUSTERED([BillId]),
    CONSTRAINT [FK_Bill_Session] FOREIGN KEY([SessionId])
        REFERENCES [Security].[Session]([SessionId])
)
GO
```

Tabla de relación [Bills].[BillOrder].

```
CREATE TABLE [Bills].[BillOrder]
(
    [BillId] BIGINT NOT NULL,
    [OrderId] BIGINT NOT NULL,
    CONSTRAINT [PK_BillOrder] PRIMARY KEY CLUSTERED([BillId], [OrderId]),
    CONSTRAINT [UQ_BillOrder] UNIQUE([OrderId]),
    CONSTRAINT [FK_BillOrder_Bill] FOREIGN KEY([BillId])
        REFERENCES [Bills].[Bill]([BillId]) ON DELETE CASCADE,
    CONSTRAINT [FK_BillOrder_Order] FOREIGN KEY([OrderId])
        REFERENCES [Bills].[Order]([OrderId]) ON DELETE CASCADE
)
GO
```

Tabla de relación [Bills].[BillTables].

```
CREATE TABLE [Bills].[BillTables]
(
    [BillId] BIGINT NOT NULL,
    [TableId] INT NOT NULL,
    CONSTRAINT [PK_BillTables] PRIMARY KEY CLUSTERED([BillId], [TableId]),
    CONSTRAINT [FK_BillTables_Bill] FOREIGN KEY([BillId])
        REFERENCES [Bills].[Bill]([BillId]) ON DELETE CASCADE,
    CONSTRAINT [FK_BillTables_Tables] FOREIGN KEY([TableId])
        REFERENCES [Common].[Table]([TableId])  ON DELETE CASCADE
)
GO
```

Entidad [Menu].[Bundle].

```
CREATE TABLE [Menu].[Bundle]
(
    [BundleId] INT NOT NULL IDENTITY(1, 1),
    [MasterDishId] INT NOT NULL,
    [SlaveDishId] INT NOT NULL,
    [Quantity] INT NOT NULL DEFAULT 1,
```

```
    CONSTRAINT [PK_Bundle] PRIMARY KEY CLUSTERED([BundleId]),
    CONSTRAINT [FK_Bundle_MasterDish] FOREIGN KEY([MasterDishId])
        REFERENCES [Menu].[Dish]([DishId]),
    CONSTRAINT [FK_Bundle_SlaveDish] FOREIGN KEY([SlaveDishId])
        REFERENCES [Menu].[Dish]([DishId]),
    CONSTRAINT [CK_Bundle_Quantity] CHECK([Quantity] > 0),
    CONSTRAINT [CK_Bundle_Relationship] CHECK([MasterDishId] <> [SlaveDishId])
)
GO
```

Entidad [Menu].[Category].

```
CREATE TABLE [Menu].[Category]
(
    [CategoryId] INT NOT NULL IDENTITY(1, 1),
    [Name] NVARCHAR(64) NOT NULL,
    [Description] NVARCHAR(256) NULL DEFAULT '',
    CONSTRAINT [PK_Category] PRIMARY KEY CLUSTERED([CategoryId]),
    CONSTRAINT [UQ_Category] UNIQUE([Name])
)
GO
```

Entidad [Common].[Customer].

```
CREATE TABLE [Common].[Customer]
(
    [CustomerId] INT NOT NULL IDENTITY(1, 1),
    [FirstName] NVARCHAR(64) NOT NULL,
    [LastName] NVARCHAR(128) NOT NULL,
    [Reference] NVARCHAR(32) NULL DEFAULT ''
    CONSTRAINT [PK_Customer] PRIMARY KEY CLUSTERED([CustomerId]),
    CONSTRAINT [UQ_Customer] UNIQUE([FirstName], [LastName])
)
GO
```

Entidad [Menu].[Dish].

```
CREATE TABLE [Menu].[Dish]
(
    [DishId] INT NOT NULL IDENTITY(1, 1),
    [Name] NVARCHAR(64) NOT NULL,
    [Description] NVARCHAR(256) NULL DEFAULT '',
    [CategoryId] INT NOT NULL,
    [Price] MONEY NOT NULL,
    [Visible] BIT NOT NULL,
    [ExpirationDate] DATETIME NULL DEFAULT '12/31/2100',
    [Discriminator] CHAR(1) DEFAULT 'D',
    CONSTRAINT [PK_Dish] PRIMARY KEY CLUSTERED([DishId]),
    CONSTRAINT [UQ_Dish] UNIQUE([Name]),
    CONSTRAINT [FK_Dish_Category] FOREIGN KEY([CategoryId])
        REFERENCES [Menu].[Category]([CategoryId]) ON DELETE NO ACTION,
    CONSTRAINT [CK_Dish_Price] CHECK ([Price] >= 0),
    CONSTRAINT [CK_Dish_Discriminator] CHECK([Discriminator] IN ('D', 'B'))
)
GO
```

Entidad [Common].[Employee].

```
CREATE TABLE [Common].[Employee]
(
    [EmployeeId] INT NOT NULL IDENTITY(1, 1),
    [FirstName] NVARCHAR(64) NOT NULL,
    [LastName] NVARCHAR(128) NOT NULL,
    [UserId] INT NULL,
    CONSTRAINT [PK_Employee] PRIMARY KEY CLUSTERED([EmployeeId]),
    CONSTRAINT [FK_Employee_User] FOREIGN KEY([UserId])
        REFERENCES [Security].[User]([UserId]) ON DELETE NO ACTION
)
GO
```

Entidad [Bills].[Invoice].

```
CREATE TABLE [Bills].[Invoice]
(
    [InvoiceId] BIGINT NOT NULL IDENTITY(1, 1),
    [BillId] BIGINT NOT NULL,
    [CustomerId] INT NOT NULL,
    [Date] DATETIME NOT NULL DEFAULT GETDATE(),
    CONSTRAINT [PK_Invoice] PRIMARY KEY CLUSTERED([InvoiceId]),
    CONSTRAINT [FK_Invoice_Bill] FOREIGN KEY([BillId])
        REFERENCES [Bills].[Bill]([BillId]) ON DELETE CASCADE,
    CONSTRAINT [FK_Invoice_Customer] FOREIGN KEY([CustomerId])
        REFERENCES [Common].[Customer]([CustomerId]) ON DELETE CASCADE
)
GO
```

Entidad [Bills].[Order].

```
CREATE TABLE [Bills].[Order]
(
    [OrderId] BIGINT NOT NULL IDENTITY(1, 1),
    [RequestedTime] DATETIME NOT NULL DEFAULT GETDATE(),
    [StartedTime] DATETIME NULL,
    [FinishedTime] DATETIME NULL,
    [DeliveryTime] DATETIME NULL,
    [State] TINYINT NOT NULL,
    [DishId] INT NOT NULL,
    [Instructions] NVARCHAR(256) NULL DEFAULT '',
    [Priority] TINYINT NOT NULL,
    CONSTRAINT [PK_Order] PRIMARY KEY CLUSTERED([OrderId]),
    CONSTRAINT [FK_Order_Dish] FOREIGN KEY([DishId])
        REFERENCES [Menu].[Dish]([DishId]),
    CONSTRAINT [CK_Order_State] CHECK([State] IN (0, 1, 2, 3, 4, 5)),
    CONSTRAINT [CK_Order_Priority] CHECK([Priority] IN (0, 1, 2))
)
GO
```

Entidad [Security].[Role].

```
CREATE TABLE [Security].[Role]
(
    [RoleId] INT NOT NULL IDENTITY(1, 1),
    [Name] NVARCHAR(32) NOT NULL,
    [Description] NVARCHAR(128) NULL DEFAULT '',
    CONSTRAINT [PK_Role] PRIMARY KEY CLUSTERED([RoleId]),
    CONSTRAINT [UQ_Role] UNIQUE ([Name])
)
GO
```

Tabla de relación [Security].[RoleSecurable].

```
CREATE TABLE [Security].[RoleSecurable]
(
    [RoleId] INT NOT NULL,
    [SecurableId] INT NOT NULL,
    [Allowed] BIT NOT NULL DEFAULT 0,
```

```
    CONSTRAINT [PK_RoleSecurable] PRIMARY KEY CLUSTERED([RoleId],
[SecurableId]),
    CONSTRAINT [FK_RoleSecurable_Role] FOREIGN KEY([RoleId])
        REFERENCES [Security].[Role]([RoleId]),
    CONSTRAINT [FK_RoleSecurable_Securable] FOREIGN KEY([SecurableId])
        REFERENCES [Security].[Securable]([SecurableId])
)
GO
```

Entidad [Security].[Securable].

```
CREATE TABLE [Security].[Securable]
(
    [SecurableId] INT NOT NULL IDENTITY(1, 1),
    [Name] NVARCHAR(32) NOT NULL,
    [Description] NVARCHAR(128) NULL DEFAULT '',
    CONSTRAINT [PK_Securable] PRIMARY KEY CLUSTERED([SecurableId]),
    CONSTRAINT [UQ_Securable] UNIQUE([Name])
)
GO
```

Entidad [Security].[Session].

```
CREATE TABLE [Security].[Session]
(
    [SessionId] BIGINT NOT NULL IDENTITY(1, 1),
    [StartedDate] DATETIME NOT NULL DEFAULT GETDATE(),
    [EndedDate] DATETIME NULL,
    [UserId] INT NOT NULL,
    [State] TINYINT NOT NULL DEFAULT 0,
    CONSTRAINT [PK_Session] PRIMARY KEY CLUSTERED([SessionId]),
    CONSTRAINT [FK_Session_User] FOREIGN KEY([UserId])
        REFERENCES [Security].[User]([UserId]),
    CONSTRAINT [CK_Session_State] CHECK ([State] IN (0, 1, 2))
)
GO
```

Entidad [Common].[Table].

```
CREATE TABLE [Common].[Table]
(
    [TableId] INT NOT NULL,
    [Name] NVARCHAR(5) NOT NULL,
    [Description] NVARCHAR(128) NULL DEFAULT '',
    [Area] TINYINT NOT NULL,
    [Capacity] SMALLINT NOT NULL,
    CONSTRAINT [PK_Table] PRIMARY KEY CLUSTERED([TableId]),
    CONSTRAINT [CK_Table_Area] CHECK([Area] IN (0, 1)),
    CONSTRAINT [CK_Table_Capacity] CHECK([Capacity] > 0)
)
GO
```

Entidad [Security].[User].

```
CREATE TABLE [Security].[User]
(
    [UserId] INT NOT NULL IDENTITY(1, 1),
    [UserName] NVARCHAR(32) NOT NULL,
    [Password] BINARY(40) NOT NULL,
    [RoleId] INT NOT NULL,
    CONSTRAINT [PK_User] PRIMARY KEY CLUSTERED([UserId]),
    CONSTRAINT [FK_User_Role] FOREIGN KEY([RoleId])
        REFERENCES [Security].[Role]([RoleId])
)
GO
```

Tabla de relación `[Security].[UserRole].`

```
CREATE TABLE [Security].[UserRole]
(
    [UserId] INT NOT NULL,
    [RoleId] INT NOT NULL,
    CONSTRAINT [PK_UserRole] PRIMARY KEY CLUSTERED([UserId], [RoleId]),
    CONSTRAINT [FK_UserRole_User] FOREIGN KEY([UserId])
        REFERENCES [Security].[User]([UserId]),
    CONSTRAINT [FK_UserRole_Role] FOREIGN KEY([RoleId])
        REFERENCES [Security].[Role]([RoleId])
)
GO
```

Apéndice D

Información de mapeo ORM para Hibernate.

Archivos de mapeo *.hbm.xml.

Mapeo de la clase `Bill` a la tabla `[Bills].[Bill]`.

```
<hibernate-mapping
        xmlns="urn:nhibernate-mapping-2.2"
        assembly="Restomatic.DomainModel"
        namespace="Restomatic.DomainModel.Bills">

    <class name="Bill" table="Bill" schema="Bills">
        <id name="BillId" type="System.Int64" column="BillId"
        unsaved-value="0" access="property">
            <generator class="identity"/>
        </id>

        <property name="StartedTime" column="StartedTime"
            type="System.DateTime" update="true" access="property"/>

        <property name="FinishedTime" column="FinishedTime"
            type="System.DateTime" update="true" access="property"/>

        <property name="State" column="State" update="true"
access="property"/>

        <set name="TablesSet" table="BillTables" cascade="none">
            <key column="BillId"/>
            <many-to-many class="Restomatic.DomainModel.Common.Table"
                    column="TableId"/>
        </set>

        <property name="Total" column="Total" type="System.Decimal"
            update="true" access="property"/>

        <set name="OrdersSet" table="BillOrder" cascade="all">
            <key column="BillId"/>
            <many-to-many class="Order" column="OrderId"/>
        </set>

        <many-to-one name="Session"
class="Restomatic.DomainModel.Security.Session"
                column="SessionId" cascade="none" fetch="select"
                update="true" access="property"/>
    </class>

</hibernate-mapping>
```

Mapeo de la clase `Category` a la tabla `[Menu].[Category]`.

```
<hibernate-mapping
        xmlns="urn:nhibernate-mapping-2.2"
        assembly="Restomatic.DomainModel"
        namespace="Restomatic.DomainModel.Menu">

    <class name="Category" table="Category" schema="Menu">
        <id name="CategoryId" column="CategoryId" type="System.Int32"
        unsaved-value="0" access="property">
            <generator class="identity"/>
        </id>

        <property name="Name" column="Name" type="System.String"
            length="64" not-null="true" access="property"/>
        <property name="Description" column="Description" type="System.String"
            length="256" not-null="false" access="property"/>

        <!--one-to-one name="Picture" class="CategoryPicture"
                lazy="false" cascade="all"/-->
    </class>

</hibernate-mapping>
```

Mapeo de la clase `Customer` a la tabla `[Common].[Customer]`.

```
<hibernate-mapping
        xmlns="urn:nhibernate-mapping-2.2"
        assembly="Restomatic.DomainModel"
        namespace="Restomatic.DomainModel.Common">

    <class name="Customer" table="Customer" schema="Common">
        <id name="CustomerId" column="CustomerId" type="System.Int32"
        unsaved-value="0" access="property">
            <generator class="identity"/>
        </id>

        <property name="Name" column="FirstName" type="System.String"
            length="64" update="true" access="property"/>

        <property name="LastName" column="LastName" type="System.String"
            length="128" update="true" access="property"/>

        <property name="Reference" column="Reference" type="System.String"
            length="32" update="true" access="property"/>
    </class>

</hibernate-mapping>
```

Mapeo de la clase `Dish` a la tabla `[Menu].[Dish]`.

```
<hibernate-mapping
        xmlns="urn:nhibernate-mapping-2.2"
        assembly="Restomatic.DomainModel"
        namespace="Restomatic.DomainModel.Menu">

    <class name="Dish" table="Dish" schema="Menu" discriminator-value="D">
        <id name="DishId" column="DishId" type="System.Int32"
        unsaved-value="0" access="property">
            <generator class="identity"/>
        </id>

        <discriminator column="Discriminator" type="System.String"/>

        <property name="Name" column="Name" type="System.String"
            length="64" update="true" access="property"/>

        <property name="Description" column="Description" type="System.String"
            length="128" update="true" access="property"/>

        <many-to-one name="Category" column="CategoryId" class="Category"
not-null="true"
                cascade="none" fetch="select" lazy="false"/>

        <property name="Price" column="Price" type="System.Decimal"
            update="true" access="property"/>

        <property name="Visible" column="Visible" type="System.Boolean"
            update="true" access="property"/>

        <!--one-to-one name="Picture" class="DishPicture" lazy="false"
cascade="all"/-->

        <subclass name="Bundle" discriminator-value="B">
            <property name="ExpirationDate" column="ExpirationDate"
                type="System.DateTime" update="true" access="property"/>

            <map name="Content" table="Bundle" schema="Menu" cascade="none"
            fetch="select" lazy="false" access="property">
                <key column="MasterDishId"/>
                <index-many-to-many column="SlaveDishId" class="Dish"/>
                <element column="Quantity" type="System.Int32"/>
            </map>
        </subclass>
    </class>

</hibernate-mapping>
```

Mapeo de la clase `Employee` a la tabla `[Common].[Employee]`.

```
<hibernate-mapping
        xmlns="urn:nhibernate-mapping-2.2"
        assembly="Restomatic.DomainModel"
        namespace="Restomatic.DomainModel.Common">

    <class name="Employee" table="Employee" schema="Common">
        <id name="EmployeeId" column="EmployeeId" type="System.Int32"
        unsaved-value="0" access="property">
            <generator class="identity"/>
        </id>

        <property name="Name" column="FirstName" type="System.String"
            length="64" update="true" access="property"/>

        <property name="LastName" column="LastName" type="System.String"
            length="128" update="true" access="property"/>

        <many-to-one name="User" column="UserId"
                class="Restomatic.DomainModel.Security.User"
                not-null="false" cascade="save-update" fetch="select"
                lazy="false"/>
    </class>

</hibernate-mapping>
```

Mapeo de la clase `Invoice` a la tabla `[Bills].[Invoice].`

```
<hibernate-mapping
        xmlns="urn:nhibernate-mapping-2.2"
        assembly="Restomatic.DomainModel"
        namespace="Restomatic.DomainModel.Bills">

    <class name="Invoice" table="Invoice" schema="Bills">
        <id name="InvoiceId" column="invoiceId" type="System.Int64"
        unsaved-value="0" access="property">
            <generator class="identity"/>
        </id>

        <many-to-one name="Bill" column="BillId" class="Bill" not-null="true"
                cascade="none" fetch="select" lazy="false"/>

        <many-to-one name="Customer" column="CustomerId"
                class="Restomatic.DomainModel.Common.Customer"
                not-null="true" cascade="none" fetch="select"
                lazy="false"/>

        <property name="Date" column="Date" type="System.DateTime"
            update="false" access="property"/>
    </class>

</hibernate-mapping>
```

Mapeo de la clase `Order` a la tabla `[Bills].[Order]`.

```
<hibernate-mapping
        xmlns="urn:nhibernate-mapping-2.2"
        assembly="Restomatic.DomainModel"
        namespace="Restomatic.DomainModel.Bills">

    <class name="Order" table="[Order]" schema="Bills">
        <id name="OrderId" column="OrderId" type="System.Int64"
        unsaved-value="0" access="property">
            <generator class="identity"/>
        </id>

        <property name="RequestedTime" column="RequestedTime"
            type="System.DateTime" not-null="true" update="false"
            access="property"/>

        <property name="StartedTime" column="StartedTime"
            type="System.DateTime" not-null="false" update="true"
            access="property"/>

        <property name="FinishedTime" column="FinishedTime"
            type="System.DateTime" not-null="false" update="true"
            access="property"/>

        <property name="DeliveryTime" column="DeliveryTime"
            type="System.DateTime" not-null="false" update="true"
            access="property"/>

        <property name="State" column="State" not-null="true"
            update="true" access="property"/>

        <many-to-one name="Dish" column="DishId"
            class="Restomatic.DomainModel.Menu.Dish" not-null="true"
            cascade="none" fetch="select" lazy="false"/>

        <property name="Instructions" column="Instructions"
            type="System.String" length="256" not-null="false"
            update="false" access="property"/>

        <property name="Priority" column="Priority" not-null="true"
            update="true" access="property"/>
    </class>

</hibernate-mapping>
```

Mapeo de la clase `Role` a la tabla `[Security].[Role].`

```
<hibernate-mapping
        xmlns="urn:nhibernate-mapping-2.2"
        assembly="Restomatic.DomainModel"
        namespace="Restomatic.DomainModel.Security">

    <class name="Role" table="[Role]" schema="Security">
        <id name="RoleId" column="RoleId" type="System.Int32"
        unsaved-value="0" access="property">
            <generator class="identity"/>
        </id>

        <property name="Name" column="Name" type="System.String"
            length="32" update="false" access="property"/>

        <map name="SecurableMap" table="RoleSecurable" schema="Security"
        cascade="all" fetch="select" lazy="false" access="property">
            <key column="RoleId"/>
            <index-many-to-many column="SecurableId" class="Securable"/>
            <element column="Allowed" type="System.Boolean"/>
        </map>
    </class>

</hibernate-mapping>
```

Mapeo de la clase `Securable` a la tabla `[Security].[Securable]`.

```
<hibernate-mapping
        xmlns="urn:nhibernate-mapping-2.2"
        assembly="Restomatic.DomainModel"
        namespace="Restomatic.DomainModel.Security">

    <class name="Securable" table="[Securable]" schema="Security">
        <id name="SecurableId" column="SecurableId" type="System.Int32"
        unsaved-value="0" access="property">
            <generator class="identity"/>
        </id>

        <property name="Name" column="Name" type="System.String"
            length="32" update="false" access="property"/>
    </class>

</hibernate-mapping>
```

Mapeo de la clase `Session` a la tabla `[Security].[Session].`

```
<hibernate-mapping
        xmlns="urn:nhibernate-mapping-2.2"
        assembly="Restomatic.DomainModel"
        namespace="Restomatic.DomainModel.Security">

    <class name="Session" table="[Session]" schema="Security">
        <id name="SessionId" column="SessionId" type="System.Int64"
        unsaved-value="0" access="property">
            <generator class="identity"/>
        </id>

        <property name="StartedDate" column="StartedDate"
            type="System.DateTime" access="property"/>

        <property name="EndedDate" column="EndedDate"
            type="System.DateTime" access="property"/>

        <many-to-one name="User" column="UserId" class="User"
                not-null="true" update="false" cascade="none"
                fetch="select" lazy="false"/>

        <property name="State" column="State" not-null="true"
            access="property"/>

        <set name="BillsCollection" cascade="save-update">
            <key column="SessionId"/>
            <one-to-many class="Restomatic.DomainModel.Bills.Bill"/>
        </set>
    </class>

</hibernate-mapping>
```

Mapeo de la clase `Table` a la tabla `[Common].[Table]`.

```
<hibernate-mapping
        xmlns="urn:nhibernate-mapping-2.2"
        assembly="Restomatic.DomainModel"
        namespace="Restomatic.DomainModel.Common">

    <class name="Table" table="[Table]" schema="Common">
        <id name="TableId" column="TableId" type="System.Int32"
        unsaved-value="0" access="property">
            <generator class="identity"/>
        </id>

        <property name="Area" column="Area" not-null="true"
            update="true" access="property"/>

        <property name="Capacity" column="Capacity" type="System.Int32"
            not-null="true" update="true" access="property"/>
    </class>

</hibernate-mapping>
```

Mapeo de la clase `User` a la tabla `[Security].[User].`

```
<hibernate-mapping
        xmlns="urn:nhibernate-mapping-2.2"
        assembly="Restomatic.DomainModel"
        namespace="Restomatic.DomainModel.Security">

    <class name="User" table="[User]" schema="Security">
        <id name="UserId" column="UserId" type="System.Int32"
        unsaved-value="0" access="property">
            <generator class="identity"/>
        </id>

        <property name="UserName" column="UserName" type="System.String"
            length="32" update="false" access="property"/>

        <property name="Password" column="Password" type="System.String"
            length="40" update="true" access="property"/>

        <many-to-one name="Role" column="RoleId" class="Role"
            update="true" cascade="none" fetch="select" lazy="false"
            access="property"/>
    </class>

</hibernate-mapping>
```

Apéndice

E

Interfaces genéricas de la capa de servicios.

Código fuente de las interfaces de servicio en C#.

Clase `IBillLocator`.

```
using System;
using System.Collections.Generic;

using Restomatic.DomainModel.Bills;
using Restomatic.DomainModel.Common;

namespace Restomatic.DomainLogic.Contract.Bills
{
    public interface IBillLocator
    {
        IList<Bill> Find(BillState state);
        IList<Bill> Find(BillState state, DateTime startDate, DateTime
endDate);
        IList<Bill> FindWithoutSession();

        IList<Table> GetTables();
    }
}
```

Clase `IBillManager`.

```
using System;
using System.Collections.Generic;

using Restomatic.DomainModel.Bills;
using Restomatic.DomainModel.Common;

namespace Restomatic.DomainLogic.Contract.Bills
{
    public interface IBillManager
    {
        Bill Open(Bill bill);
        Bill Close(Bill bill);
        Bill Pay(Bill bill);
        Bill Cancel(Bill bill);
        Bill AssignTable(Bill bill, Table table);
        Bill RemoveTable(Bill bill, Table table);
        Bill AddOrder(Bill bill, Order order);
        Bill RemoveOrder(Bill bill, Order order);

        bool SplitBill(long billId, IDictionary<int, List<Order>> payload);
    }
}
```

Clase `IInvoiceManager`.

```
using System;
using System.Collections.Generic;

using Restomatic.DomainModel.Bills;
using Restomatic.DomainModel.Common;

namespace Restomatic.DomainLogic.Contract.Bills
{
    public interface IInvoiceManager
    {
        Invoice Create(Bill bill, Customer customer);
        Invoice Cancel(Invoice invoice);
        IList<Invoice> Find(Customer customer);
        IList<Invoice> Find(DateTime startDate, DateTime endDate);
    }
}
```

Clase `IMenuLocator`.

```
using System;
using System.Collections.Generic;

using Restomatic.DomainModel.Menu;

namespace Restomatic.DomainLogic.Contract.Menu
{
    public interface IMenuLocator
    {
        IList<Dish> GetDishList();
        IList<Dish> FindDish(string name);
        IList<Dish> FindDish(Category category);
        IList<Dish> FindDishByPrice(decimal lower, decimal higher);

        IList<Bundle> GetBundleList();
        IList<Bundle> FindBundle(string name);
        IList<Bundle> FindBundle(Category category);
        IList<Bundle> FindBundleByPrice(decimal lower, decimal higher);

        IList<Category> GetCategories();
        IDictionary<Category, IList<Dish>> GetMenu();
    }
}
```

Clase `IMenuManager`.

```
using System;
using System.Collections.Generic;

using Restomatic.DomainModel.Menu;

namespace Restomatic.DomainLogic.Contract.Menu
{
    public interface IMenuManager
    {
        Dish GetDish(int dishId);
        Dish CreateDish(Dish dish);
        Dish UpdateDish(Dish dish);
        Dish RefreshDish(Dish dish);
        Dish DeleteDish(Dish dish);

        Bundle GetBundle(int dishId);
        Bundle CreateBundle(Bundle bundle);
        Bundle UpdateBundle(Bundle bundle);
        Bundle RefreshBundle(Bundle bundle);
        Bundle DeleteBundle(Bundle bundle);

        Category GetCategory(int categoryId);
        Category CreateCategory(Category category);
        Category UpdateCategory(Category category);
        Category RefreshCategory(Category category);
    }
}
```

Clase `IOrderLocator`.

```
using System;
using System.Collections.Generic;

using Restomatic.DomainModel.Bills;

namespace Restomatic.DomainLogic.Contract.Bills
{
    public interface IOrderLocator
    {
        OrderState GetCurrentState(long orderId);

        IList<Order> Find(OrderState state);
        IList<Order> Find(OrderPriority priority);
        IList<Order> Find(OrderState state, OrderPriority priority);
        IList<Order> Find(DateTime startDate, DateTime endDate);
        IList<Order> Find(OrderState state, DateTime startDate, DateTime
endDate);
        IList<Order> Find(OrderPriority priority, DateTime startDate, DateTime
endDate);
        IList<Order> Find(OrderState state, OrderPriority priority, DateTime
startDate, DateTime endDate);
    }
}
```

Clase `IOrderManager`.

```
using System;
using System.Collections.Generic;

using Restomatic.DomainModel.Bills;

namespace Restomatic.DomainLogic.Contract.Bills
{
    public interface IOrderManager
    {
        Order Send(Order order);
        Order Prepare(Order order);
        Order Finish(Order order);
        Order Deliver(Order order);
        Order Cancel(Order order);
    }
}
```

Clase `ISecurityManager`.

```
using System;
using System.Collections.Generic;

using Restomatic.DomainModel.Security;

namespace Restomatic.DomainLogic.Contract.Security
{
    public interface ISecurityManager
    {
        Securable GetSecurableById(int securableId);
        void CreateSecurable(Securable securable);
        void DeleteSecurable(Securable securable);

        Role GetRoleById(int roleId);
        void CreateRole(Role role);
        void UpdateRole(Role role);
        void DeleteRole(Role role);

        User GetUserById(int userId);
        void CreateUser(User user);
        void DeleteUser(User user);
    }
}
```

Clase `ISecurityProvider`.

```
using System;
using System.Collections.Generic;

using Restomatic.DomainModel.Security;
using Restomatic.DomainModel.Bills;

namespace Restomatic.DomainLogic.Contract.Security
{
    public interface ISecurityProvider
    {
        Session OpenSession(string userName, string password);
        User GetUser(string userName, string password);
        Session CloseSession(Session session);

        Session Attach(Bill bill, Session session);
        Session Detach(Bill bill, Session session);

        bool IsAllowed(User user, Securable securable);

        ChangePasswordResult ChangePassword(User user, string newPassword);
        IList<Session> FindSessionByUser(User user);
    }
}
```

Apéndice F

Implementación de la capa de servicios: WCF.

Código fuente de las implementaciones de servicio en C#.

ServiceContract `BillLocatorService`.

BillLocatorService.cs

```
 1 using System;
 2 using System.Collections.Generic;
 3 using System.Linq;
 4 using System.Runtime.Serialization;
 5 using System.ServiceModel;
 6 using System.Text;
 7
 8 using Restomatic.DomainModel.Bills;
 9 using Restomatic.DomainLogic.Contract.Bills;
10 using Restomatic.DomainModel.Common;
11
12 namespace Restomatic.Services
13 {
14     [ServiceContract(Namespace="http://restomatic.azc.uam.mx/")]
15     public class BillLocatorService : IBillLocator
16     {
17         public IBillLocator BillLocator { get; protected set; }
18
19         #region IBillLocator Members
20
21         [OperationContract(Name="FindByState")]
22         public IList<Bill> Find(BillState state)
23         {
24             return BillLocator.Find(state);
25         }
26
27         [OperationContract]
28         public IList<Bill> Find(BillState state, DateTime startDate,
DateTime endDate)
29         {
30             return BillLocator.Find(state, startDate, endDate);
31         }
32
33         [OperationContract]
34         public IList<Bill> FindWithoutSession()
35         {
36             return BillLocator.FindWithoutSession();
37         }
38
39         [OperationContract]
40         public IList<Table> GetTables()
41         {
42             return BillLocator.GetTables();
43         }
44
45         #endregion
46     }
47 }
```

ServiceContract `BillManagerService.`

BillManagerService.cs

```
 1 using System;
 2 using System.Collections.Generic;
 3 using System.Linq;
 4 using System.Runtime.Serialization;
 5 using System.ServiceModel;
 6 using System.Text;
 7
 8 using Restomatic.DomainLogic.Contract.Bills;
 9 using Restomatic.DomainModel.Bills;
10 using Restomatic.DomainModel.Common;
11
12 namespace Restomatic.Services
13 {
14     [ServiceContract(Namespace = "http://restomatic.azc.uam.mx/")]
15     public class BillManagerService : IBillManager
16     {
17         public IBillManager BillManager { get; protected set; }
18
19         #region IBillManager Members
20
21         [OperationContract]
22         public Bill Open(Bill bill)
23         {
24             return BillManager.Open(bill);
25         }
26
27         [OperationContract]
28         public Bill Close(Bill bill)
29         {
30             return BillManager.Close(bill);
31         }
32
33         [OperationContract]
34         public Bill Pay(Bill bill)
35         {
36             return BillManager.Pay(bill);
37         }
38
39         [OperationContract]
40         public Bill Cancel(Bill bill)
41         {
42             return BillManager.Cancel(bill);
43         }
44
45         [OperationContract]
46         public Bill AssignTable(Bill bill, Table table)
47         {
48             return BillManager.AssignTable(bill, table);
49         }
50
51         [OperationContract]
52         public Bill RemoveTable(Bill bill, Table table)
53         {
54             return BillManager.RemoveTable(bill, table);
55         }
56
```

```
57          [OperationContract]
58          public Bill AddOrder(Bill bill, Order order)
59          {
60              return BillManager.AddOrder(bill, order);
61          }
62
63          [OperationContract]
64          public Bill RemoveOrder(Bill bill, Order order)
65          {
66              return BillManager.RemoveOrder(bill, order);
67          }
68
69          [OperationContract]
70          public bool SplitBill(long billId, IDictionary<int, List<Order>>
payload)
71          {
72              return BillManager.SplitBill(billId, payload);
73          }
74
75          #endregion
76      }
77  }
```

ServiceContract `InvoiceManagerService.`

InvoiceManagerService.cs

```
 1 using System;
 2 using System.Collections.Generic;
 3 using System.ServiceModel;
 4
 5 using Restomatic.DomainLogic.Contract.Bills;
 6 using Restomatic.DomainModel.Bills;
 7 using Restomatic.DomainModel.Common;
 8
 9 namespace Restomatic.Services
10 {
11     [ServiceContract(Namespace="http://restomatic.azc.uam.mx/")]
12     public class InvoiceManagerService : IInvoiceManager
13     {
14         public IInvoiceManager InvoiceManager { get; protected set; }
15
16         #region IInvoiceManager Members
17
18         [OperationContract]
19         public Invoice Create(Bill bill, Customer customer)
20         {
21             return InvoiceManager.Create(bill, customer);
22         }
23
24         [OperationContract]
25         public Invoice Cancel(Invoice invoice)
26         {
27             return InvoiceManager.Cancel(invoice);
28         }
29
30         [OperationContract(Name = "FindByCustomer")]
31         public IList<Invoice> Find(Customer customer)
32         {
33             return InvoiceManager.Find(customer);
34         }
35
36         [OperationContract(Name="FindByDate")]
37         public IList<Invoice> Find(DateTime startDate, DateTime endDate)
38         {
39             return InvoiceManager.Find(startDate, endDate);
40         }
41
42         #endregion
43     }
44 }
```

ServiceContract `MenuLocatorService.`

MenuLocatorService.cs

```
 1 using System;
 2 using System.Collections.Generic;
 3 using System.ServiceModel;
 4
 5 using Restomatic.DomainLogic.Contract.Menu;
 6 using Restomatic.DomainModel.Menu;
 7
 8 namespace Restomatic.Services
 9 {
10     [ServiceContract(Namespace="http://restomatic.azc.uam.mx/")]
11     public class MenuLocatorService : IMenuLocator
12     {
13         public IMenuLocator MenuLocator { get; protected set; }
14
15         #region IMenuLocator Members
16
17         [OperationContract]
18         public IList<Dish> GetDishList()
19         {
20             return MenuLocator.GetDishList();
21         }
22
23         [OperationContract]
24         public IList<Dish> FindDish(string name)
25         {
26             return MenuLocator.FindDish(name);
27         }
28
29         [OperationContract(Name="FindDishByCategory")]
30         public IList<Dish> FindDish(Category category)
31         {
32             return MenuLocator.FindDish(category);
33         }
34
35         [OperationContract]
36         public IList<Dish> FindDishByPrice(decimal lower, decimal
higher)
37         {
38             return MenuLocator.FindDishByPrice(lower, higher);
39         }
40
41         [OperationContract]
42         public IList<Bundle> GetBundleList()
43         {
44             return MenuLocator.GetBundleList();
45         }
46
47         [OperationContract]
48         public IList<Bundle> FindBundle(string name)
49         {
50             return MenuLocator.FindBundle(name);
51         }
52
53         [OperationContract(Name="FindBundleByCategory")]
54         public IList<Bundle> FindBundle(Category category)
55         {
```

```
56              return MenuLocator.FindBundle(category);
57          }
58
59          [OperationContract]
60          public IList<Bundle> FindBundleByPrice(decimal lower, decimal
higher)
61          {
62              return MenuLocator.FindBundleByPrice(lower, higher);
63          }
64
65          [OperationContract]
66          public IList<Category> GetCategories()
67          {
68              return MenuLocator.GetCategories();
69          }
70
71          [OperationContract]
72          public IDictionary<Category, IList<Dish>> GetMenu()
73          {
74              return MenuLocator.GetMenu();
75          }
76
77          #endregion
78      }
79  }
```

ServiceContract `MenuManagerService.`

MenuManagerService.cs

```
 1 using System;
 2 using System.Collections.Generic;
 3 using System.Linq;
 4 using System.Text;
 5 using System.ServiceModel;
 6
 7 using Restomatic.DomainLogic.Contract.Menu;
 8 using Restomatic.DomainModel.Menu;
 9
10 namespace Restomatic.Services
11 {
12     [ServiceContract(Namespace="http://restomatic.azc.uam.mx/")]
13     public class MenuManagerService : IMenuManager
14     {
15         public IMenuManager MenuManager { get; protected set; }
16
17         #region IMenuManager Members
18
19         [OperationContract]
20         public Dish GetDish(int dishId)
21         {
22             return MenuManager.GetDish(dishId);
23         }
24
25         [OperationContract]
26         public Dish CreateDish(Dish dish)
27         {
28             return MenuManager.CreateDish(dish);
29         }
30
31         [OperationContract]
32         public Dish UpdateDish(Dish dish)
33         {
34             return MenuManager.UpdateDish(dish);
35         }
36
37         [OperationContract]
38         public Dish RefreshDish(Dish dish)
39         {
40             return MenuManager.RefreshDish(dish);
41         }
42
43         [OperationContract]
44         public Dish DeleteDish(Dish dish)
45         {
46             return MenuManager.DeleteDish(dish);
47         }
48
49         [OperationContract]
50         public Bundle GetBundle(int dishId)
51         {
52             return MenuManager.GetBundle(dishId);
53         }
54
55         [OperationContract]
56         public Bundle CreateBundle(Bundle bundle)
```

```
57          {
58              return MenuManager.CreateBundle(bundle);
59          }
60
61          [OperationContract]
62          public Bundle UpdateBundle(Bundle bundle)
63          {
64              return MenuManager.UpdateBundle(bundle);
65          }
66
67          [OperationContract]
68          public Bundle RefreshBundle(Bundle bundle)
69          {
70              return MenuManager.RefreshBundle(bundle);
71          }
72
73          [OperationContract]
74          public Bundle DeleteBundle(Bundle bundle)
75          {
76              return MenuManager.DeleteBundle(bundle);
77          }
78
79          [OperationContract]
80          public Category GetCategory(int categoryId)
81          {
82              return MenuManager.GetCategory(categoryId);
83          }
84
85          [OperationContract]
86          public Category CreateCategory(Category category)
87          {
88              return MenuManager.CreateCategory(category);
89          }
90
91          [OperationContract]
92          public Category UpdateCategory(Category category)
93          {
94              return MenuManager.UpdateCategory(category);
95          }
96
97          [OperationContract]
98          public Category RefreshCategory(Category category)
99          {
100             return MenuManager.RefreshCategory(category);
101         }
102
103         #endregion
104     }
105 }
```

ServiceContract `OrderLocatorService.`

OrderLocatorService.cs

```
 1 using System;
 2 using System.Collections.Generic;
 3 using System.Linq;
 4 using System.Text;
 5 using System.ServiceModel;
 6
 7 using Restomatic.DomainLogic.Contract.Bills;
 8 using Restomatic.DomainModel.Bills;
 9 using Restomatic.Services.Callback;
10
11 namespace Restomatic.Services
12 {
13     [ServiceContract(Namespace = "http://restomatic.azc.uam.mx/")]
14     public class OrderLocatorService : IOrderLocator
15     {
16         public IOrderLocator OrderLocator { get; protected set; }
17
18         #region IOrderLocator Members
19
20         [OperationContract(Name="FindByState")]
21         public IList<Order> Find(OrderState state)
22         {
23             return OrderLocator.Find(state);
24         }
25
26         [OperationContract(Name = "FindByPriority")]
27         public IList<Order> Find(OrderPriority priority)
28         {
29             return OrderLocator.Find(priority);
30         }
31
32         [OperationContract(Name = "FindByStateAndPriority")]
33         public IList<Order> Find(OrderState state, OrderPriority
priority)
34         {
35             return OrderLocator.Find(state, priority);
36         }
37
38         [OperationContract(Name = "FindByDate")]
39         public IList<Order> Find(DateTime startDate, DateTime endDate)
40         {
41             return OrderLocator.Find(startDate, endDate);
42         }
43
44         [OperationContract(Name = "FindByStateAndDate")]
45         public IList<Order> Find(OrderState state, DateTime startDate,
DateTime endDate)
46         {
47             return OrderLocator.Find(state, startDate, endDate);
48         }
49
50         [OperationContract(Name = "FindByPriorityAndDate")]
51         public IList<Order> Find(OrderPriority priority, DateTime
startDate, DateTime endDate)
52         {
53             return OrderLocator.Find(priority, startDate, endDate);
```

```
54          }
55
56          [OperationContract]
57          public IList<Order> Find(OrderState state, OrderPriority
priority, DateTime startDate, DateTime endDate)
58          {
59              return OrderLocator.Find(state, priority, startDate,
endDate);
60          }
61
62          [OperationContract]
63          public OrderState GetCurrentState(long orderId)
64          {
65              return OrderLocator.GetCurrentState(orderId);
66          }
67
68          #endregion
69      }
70  }
```

ServiceContract `OrderManagerService.`

OrderManagerService.cs

```
 1 using System;
 2 using System.Collections.Generic;
 3 using System.Linq;
 4 using System.Text;
 5 using System.ServiceModel;
 6
 7 using Restomatic.DomainLogic.Contract.Bills;
 8 using Restomatic.DomainModel.Bills;
 9
10 namespace Restomatic.Services
11 {
12     [ServiceContract(Namespace="http://restomatic.azc.uam.mx/")]
13     public class OrderManagerService : IOrderManager
14     {
15         public IOrderManager OrderManager { get; protected set; }
16
17         #region IOrderManager Members
18
19         [OperationContract]
20         public Order Send(Order order)
21         {
22             return OrderManager.Send(order);
23         }
24
25         [OperationContract]
26         public Order Prepare(Order order)
27         {
28             return OrderManager.Prepare(order);
29         }
30
31         [OperationContract]
32         public Order Finish(Order order)
33         {
34             return OrderManager.Finish(order);
35         }
36
37         [OperationContract]
38         public Order Deliver(Order order)
39         {
40             return OrderManager.Deliver(order);
41         }
42
43         [OperationContract]
44         public Order Cancel(Order order)
45         {
46             return OrderManager.Cancel(order);
47         }
48
49         #endregion
50     }
51 }
```

ServiceContract `OrdersEventsSuscriberService.`

OrderEventsSuscriberService.cs

```
    1 using System;
    2 using System.Collections.Generic;
    3 using System.ServiceModel;
    4 using System.Diagnostics;
    5
    6 using Restomatic.Services.Callback;
    7 using Restomatic.DataAccess.Bills;
    8 using Restomatic.DataAccess.Support;
    9 using Restomatic.DomainModel.Bills;
   10
   11 namespace Restomatic.Services
   12 {
   13     [ServiceContract(Namespace = "http://restomatic.azc.uam.mx/",
CallbackContract = typeof(IOrderEventCallback), SessionMode =
   14                      SessionMode.Required)]
   15     [ServiceBehavior(InstanceContextMode =
InstanceContextMode.PerSession, ConcurrencyMode = ConcurrencyMode.Reentrant)]
   16     public class OrderEventsSuscriberService : IDisposable
   17     {
   18         #region Properties
   19
   20         public IOrderRepository OrderRepository { get; protected set; }
   21         private static IList<IOrderEventCallback> Clients = new
List<IOrderEventCallback>();
   22
   23         #endregion
   24
   25         #region Initial Configuration
   26
   27         public void Init()
   28         {
   29             try
   30             {
   31                 OrderRepository.Added +=
OrderEventsSuscriberService.OrderAdded;
   32                 OrderRepository.Updated +=
OrderEventsSuscriberService.OrderUpdated;
   33             }
   34             catch { }
   35         }
   36
   37         static void
OrderAdded(EntityPersistenceOperationEventArgs<Order> args)
   38         {
   39             List<IOrderEventCallback> unavailableClients = new
List<IOrderEventCallback>();
   40
   41             foreach (IOrderEventCallback callback in Clients)
   42             {
   43                 try
   44                 {
   45                     callback.OnOrderAdded(args.Source);
   46                 }
   47                 catch
   48                 {
   49                     unavailableClients.Add(callback);
```

```
50                 }
51             }
52
53             foreach (IOrderEventCallback callback in unavailableClients)
54             {
55                 Clients.Remove(callback);
56             }
57         }
58
59         static void
OrderUpdated(EntityPersistenceOperationEventArgs<Order> args)
60         {
61             List<IOrderEventCallback> unavailableClients = new
List<IOrderEventCallback>();
62
63             foreach (IOrderEventCallback callback in Clients)
64             {
65                 try
66                 {
67                     Order order = args.Source;
68
69                     if (order.State != OrderState.Canceled)
70                     {
71                         callback.OnOrderUpdated(order);
72                     }
73                     else
74                     {
75                         callback.OnOrderCanceled(order);
76                     }
77                 }
78                 catch
79                 {
80                     unavailableClients.Add(callback);
81                 }
82             }
83
84             foreach (IOrderEventCallback callback in unavailableClients)
85             {
86                 Clients.Remove(callback);
87             }
88         }
89
90         #endregion
91
92         [OperationContract(IsOneWay = true)]
93         public void Suscribe()
94         {
95             try
96             {
97                 IOrderEventCallback client =
OperationContext.Current.GetCallbackChannel<IOrderEventCallback>();
98
99                 lock (Clients)
100                {
101                    if (!Clients.Contains(client))
102                    {
103                        Clients.Add(client);
104                    }
105                }
106            }
107            catch { }
```

```
  108         }
  109
  110         [OperationContract(IsOneWay = true)]
  111         public void UnSuscribe()
  112         {
  113             try
  114             {
  115                 IOrderEventCallback client =
OperationContext.Current.GetCallbackChannel<IOrderEventCallback>();
  116
  117                 lock (Clients)
  118                 {
  119                     if (Clients.Contains(client))
  120                     {
  121                         Clients.Remove(client);
  122                     }
  123                 }
  124             }
  125             catch { }
  126         }
  127
  128         #region IDisposable Members
  129
  130         public void Dispose()
  131         {
  132             try
  133             {
  134                 IOrderEventCallback client =
OperationContext.Current.GetCallbackChannel<IOrderEventCallback>();
  135
  136                 lock (Clients)
  137                 {
  138                     if (Clients.Contains(client))
  139                     {
  140                         Clients.Remove(client);
  141                     }
  142                 }
  143             }
  144             catch { }
  145         }
  146
  147         #endregion
  148     }
  149 }
```

ServiceContract `SecurityManagerService`.

SecurityManagerService.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.ServiceModel;
6
7  using Restomatic.DomainLogic.Contract.Security;
8  using Restomatic.DomainModel.Security;
9
10 namespace Restomatic.Services
11 {
12     [ServiceContract(Namespace="http://restomatic.azc.uam.mx/")]
13     public class SecurityManagerService : ISecurityManager
14     {
15         public ISecurityManager SecurityManager { get; protected set; }
16
17         #region ISecurityManager Members
18
19         [OperationContract]
20         public Securable GetSecurableById(int securableId)
21         {
22             return SecurityManager.GetSecurableById(securableId);
23         }
24
25         [OperationContract(IsOneWay = true)]
26         public void CreateSecurable(Securable securable)
27         {
28             SecurityManager.CreateSecurable(securable);
29         }
30
31         [OperationContract(IsOneWay = true)]
32         public void DeleteSecurable(Securable securable)
33         {
34             SecurityManager.DeleteSecurable(securable);
35         }
36
37         [OperationContract]
38         public Role GetRoleById(int roleId)
39         {
40             return SecurityManager.GetRoleById(roleId);
41         }
42
43         [OperationContract(IsOneWay = true)]
44         public void CreateRole(Role role)
45         {
46             SecurityManager.CreateRole(role);
47         }
48
49         [OperationContract(IsOneWay = true)]
50         public void UpdateRole(Role role)
51         {
52             SecurityManager.UpdateRole(role);
53         }
54
55         [OperationContract(IsOneWay = true)]
56         public void DeleteRole(Role role)
```

```
57          {
58              SecurityManager.DeleteRole(role);
59          }
60
61          [OperationContract]
62          public User GetUserById(int userId)
63          {
64              return SecurityManager.GetUserById(userId);
65          }
66
67          [OperationContract(IsOneWay = true)]
68          public void CreateUser(User user)
69          {
70              SecurityManager.CreateUser(user);
71          }
72
73          [OperationContract(IsOneWay = true)]
74          public void DeleteUser(User user)
75          {
76              SecurityManager.DeleteUser(user);
77          }
78
79          #endregion
80      }
81  }
```

ServiceContract `SecurityProviderService`.

SecurityProviderService.cs

```
 1 using System;
 2 using System.Collections.Generic;
 3 using System.Linq;
 4 using System.Text;
 5 using System.ServiceModel;
 6
 7 using Restomatic.DomainLogic.Contract.Security;
 8 using Restomatic.DomainModel.Security;
 9 using Restomatic.DomainModel.Bills;
10
11 namespace Restomatic.Services
12 {
13     [ServiceContract(Namespace="http://restomatic.azc.uam.mx/")]
14     public class SecurityProviderService : ISecurityProvider
15     {
16         public ISecurityProvider SecurityProvider { get; protected set; }
17
18         #region ISecurityProvider Members
19
20         [OperationContract]
21         public Session OpenSession(string userName, string password)
22         {
23             return SecurityProvider.OpenSession(userName, password);
24         }
25
26         [OperationContract]
27         public User GetUser(string userName, string password)
28         {
29             return SecurityProvider.GetUser(userName, password);
30         }
31
32         [OperationContract(Name = "AttachBill")]
33         public Session Attach(Bill bill, Session session)
34         {
35             return SecurityProvider.Attach(bill, session);
36         }
37
38         [OperationContract(Name = "DetachBill")]
39         public Session Detach(Bill bill, Session session)
40         {
41             return SecurityProvider.Detach(bill, session);
42         }
43
44         [OperationContract]
45         public Session CloseSession(Session session)
46         {
47             return SecurityProvider.CloseSession(session);
48         }
49
50         [OperationContract]
51         public bool IsAllowed(User user, Securable securable)
52         {
53             return SecurityProvider.IsAllowed(user, securable);
54         }
55
56         [OperationContract]
```

```
57          public ChangePasswordResult ChangePassword(User user, string
newPassword)
58          {
59              return SecurityProvider.ChangePassword(user, newPassword);
60          }
61
62          [OperationContract]
63          public IList<Session> FindSessionByUser(User user)
64          {
65              return SecurityProvider.FindSessionByUser(user);
66          }
67
68          #endregion
69      }
70  }
```

Apéndice G

Interfaces gráficas: Windows Forms y WPF.

Interfaz gráfica de la terminal móvil: Windows Forms.

Figura G.1: Pantalla de inicio de sesión - Terminal Móvil.

Figura G.2: Configuración de la aplicación móvil - Terminal Móvil.

Figura G.3: Administración de cuentas - Terminal Móvil.

Figura G.4: Registro de nueva cuenta - Terminal Móvil.

Figura G.5: Administración de las órdenes de una cuenta - Terminal Móvil.

Figura G.6: Creación de una nueva orden - Terminal Móvil.

Figura G.7: Catálogo de platillos del Menú - Terminal Móvil.

Interfaz gráfica de administración: WPF.

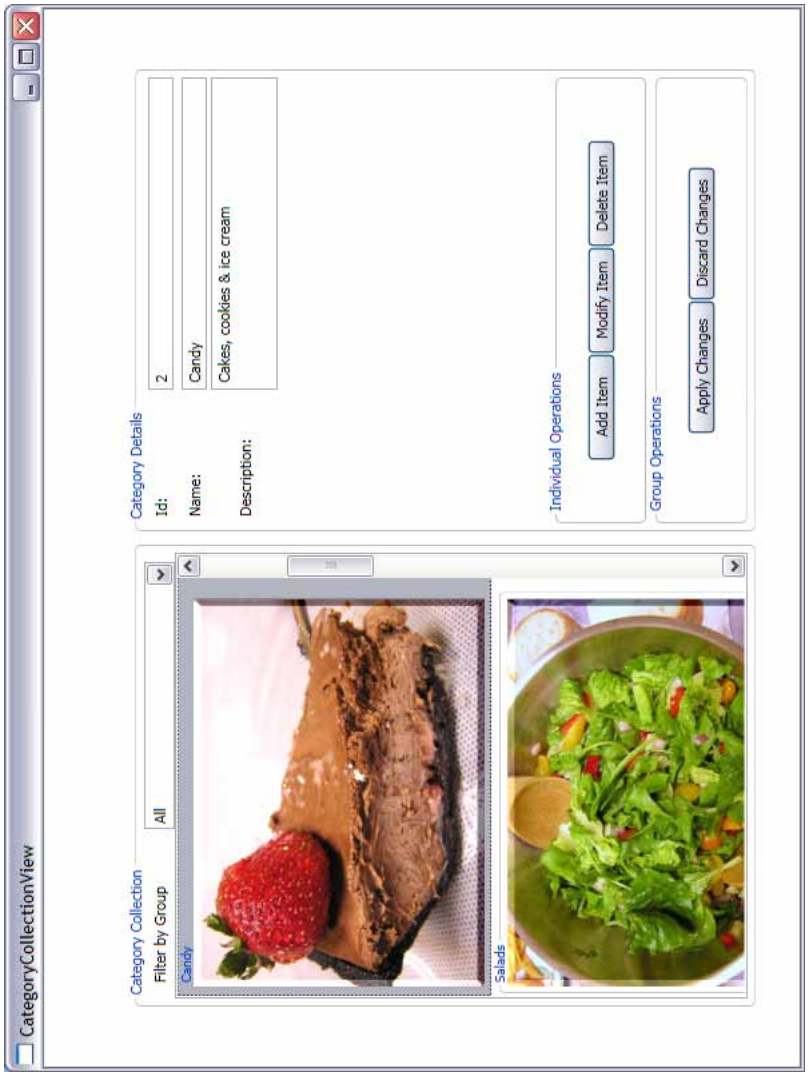


Figura G.8: Administración de categorías de platillos - Terminal de Administración.

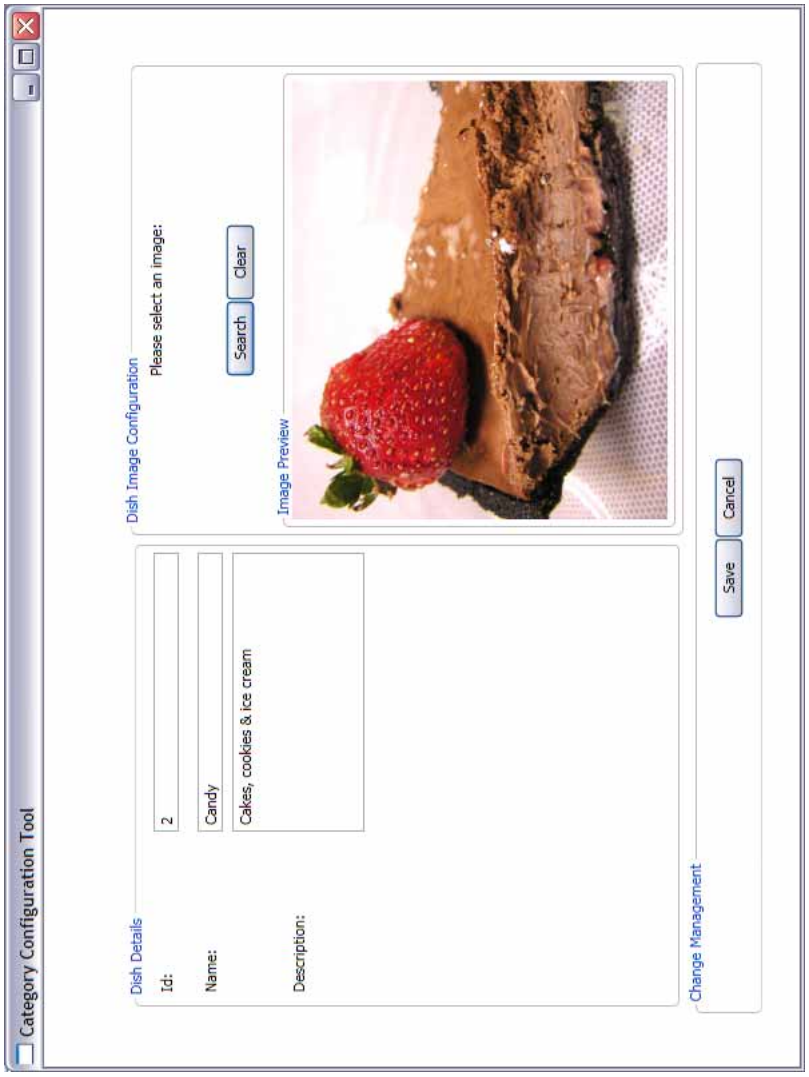


Figura G.9: Creación / edición de categorías de platillos - Terminal de Administración.

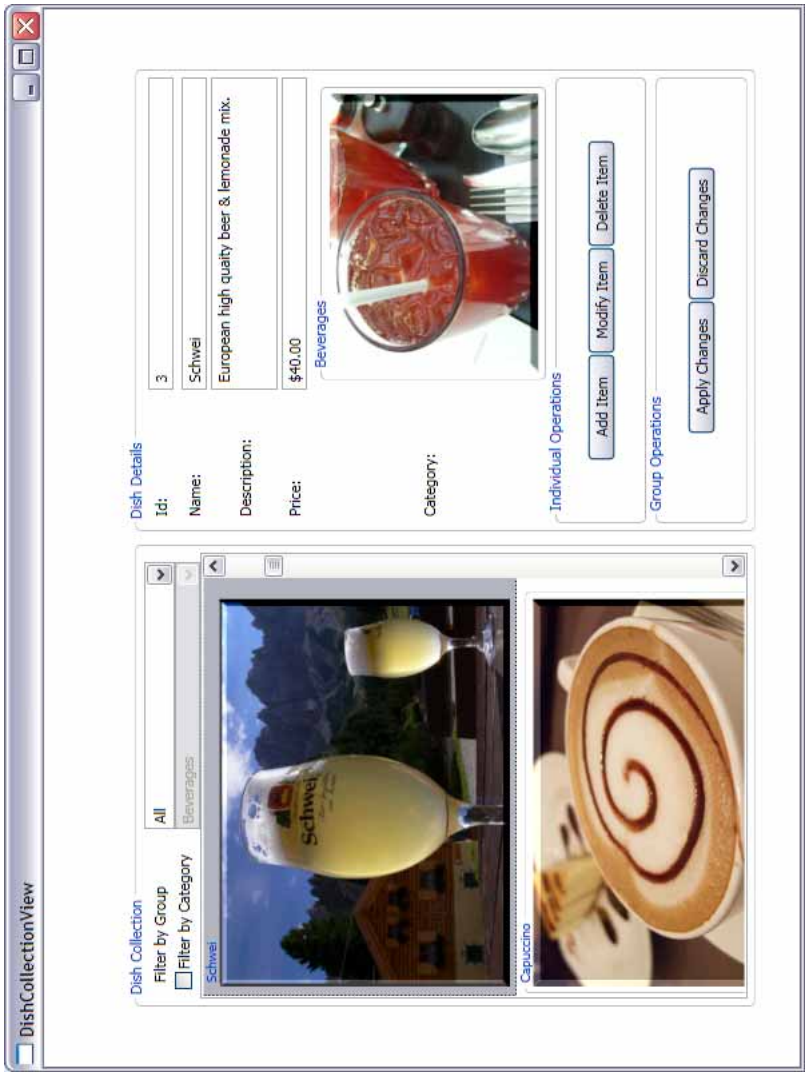


Figura G.10: Administración de platillos - Terminal de Administración.

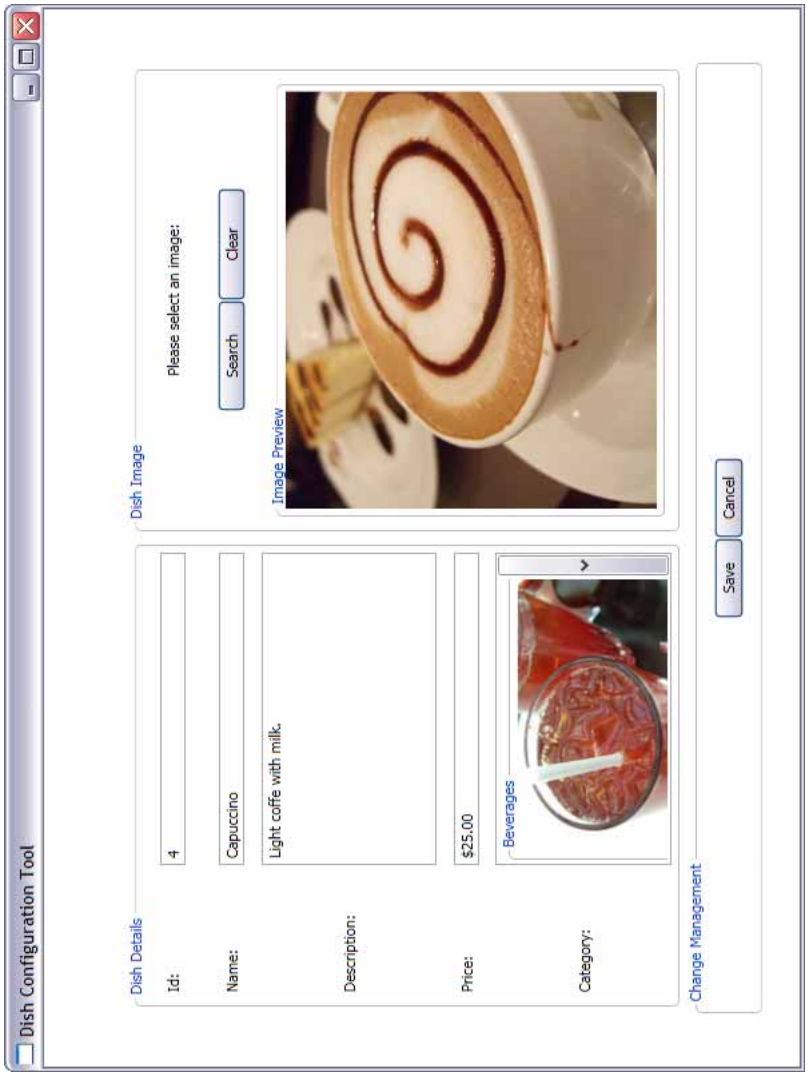


Figura G.11: Creación / edición de platillos - Terminal de Administración.

Bibliografía

[Alu03] Deepak Alur. *Core J2EE Patterns, Best Practices and Design Strategies.* Prentice Hall, 2003.

[Bro95] Frederick P. Brooks. *The Mythical Man-Month.* Addison Wesley, 1995.

[Bur06] Bill Burke. *Enterprise JavaBeans 3.0.* O'Reilly, 2006.

[DeM79] Tom DeMarco. *Structures Analysis and System Specification.* Yourdon Press, Computing Series, 1979.

[Doh08] Jim Doherty. *Cisco Networking Simplified.* Cisco Press, 2008.

[Fow03] Martin Fowler. *Patterns of Enterprise Application Architecture.* Addison Wesley, 2003.

[Fre04] Elisabeth Freeman. *Head First Design Patterns.* O'Reilly, 2004.

[Hoh04] Gregor Hohpe. *Enterprise Integration Patterns.* Addison Wesley, 2004.

[Hun99] Andrew Hunt. *The Pragmatic Programmer.* Addision Wesley, 1999.

[Low05] Juval Lowy. *Programming .NET Components.* O'Reilly, 2005.

[Mar07] Robert C. Martin. *Agile Principles, Patterns and Practices in C#.* Addision Wesley, 2007.

[Mey97] Bertrand Meyer. *Object oriented software construction.* Prentice Hall, 1997.

[Nil06] Jimmy Nilsson. *Applying Domain-Driven Design and Patterns.* Addison Wesley, 2006.

[Paw05] Renaud Pawlak. *Foundations of AOP for J2EE Development.* Apress, 2005.

[Pre01] Roger S. Pressman. *Sofware engineering: a practicioner's approach.* McGraw Hill, 2001.

[Sch00] Douglas Schmidt. *Pattern Oriented Software Architecture Vol 2: Patterns for Concurrent and Networked Objects.* Wiley, 2000.

[Sum07] S. Sumathi. *Fundamentals of Relational Database Management Systems.* Springer Verlag, 2007.

[Wak06] William C. Wake. *Design Patterns in Java.* Addison Wesley, 2006.