Universidad Autónoma del Estado de Hidalgo

Instituto de Ciencias Básicas e Ingeniería

Escuela Superior de Tlahuelilpan

ARQUITECTURA DE MICROSOFT .NET UTILIZANDO LINQ

Monografía que para obtener el título de Licenciado en Sistemas Computacionales

Presenta

P.L.S.C Armando Rufino Martínez

Asesor

Ing. Armando Hernández del Castillo

TLAHUELILPAN, HIDALGO, 20 DE NOVIEMBRE DE 2009

MEXICO

Resumen

En este trabajo se muestra el uso de la herramienta LINQ (Lenguaje de Consultas integradas) que es el mapeador de objetos relacional (ORM, Object Relational Mapping) de la plataforma de Microsoft .NET, dicha herramienta encargada del mapeo entre objetos de Bases de Datos y objetos de un lenguaje de programación, mostrando así una alternativa más de desarrollo de software, fundamentado en los modelos y conceptos del desarrollo de software en general, incluyendo los temas necesarios como programación orientada a objetos, lenguaje de programación C#, Bases de Datos, ingeniería de software, lenguaje de marcado extensible(XML) y los términos generales de los ORM.

En los dos últimos capítulos se explican los conceptos, la estructura y características de LINQ. Además de un ejemplo del uso de la herramienta LINQ, se utiliza como lenguaje de programación C# y entorno de desarrollo Visual Studio 2008.

Índice

Introducción

En la actualidad las innovaciones en las tecnologías de la información, llámense software y hardware, han facilitado el desarrollo de aplicaciones más robustas y sencillas para el tratamiento de los datos. Si bien los métodos tradicionales de desarrollo y modelado no han cambiado significativamente, las herramientas y técnicas si lo han hecho, cuando se programaba sin el uso de estas herramientas, háblese de una controladora de datos (una clase encargada de modificar, actualizar o eliminar) el desarrollador debía elaborar el código para que cumpliera con sus funciones, para agilizar el desarrollo de aplicaciones de software surgieron los mapeadores de objetos relacionales, ORM(Object-Relational Mapping) , estos se encargan del mapeo entre objetos de las Bases de Datos(Tablas, Procedimientos almacenados, etc.) y los objetos del lenguaje de programación(clases en C#), estos simplifican la creación de dichas controladoras de datos, ya que generan las funciones necesarias para la modificación o actualización de las Bases de Datos entre otras funciones, además de que en conjunto con los marcos de trabajo de los Sistemas Operativos(.NET Framework para Microsoft) agilizan el procesamiento de los datos. Un ejemplo de estas herramientas y técnicas se aplica en la plataforma de Microsoft .Net con su mapeador de objetos relacionales LINQ (Language-Integrated Query) o Lenguaje de consultas integrado.

En este documento se describe lo que es la programación orientada a objetos (POO), el manejo de las Bases de Datos y sus lenguajes de consulta, así como los modelos de desarrollo de software como el Modelo Vista Controlador (MVC), el ciclo de vida de software, el lenguaje de programación C#, LINQ como una herramienta de desarrollo de software, así como un ejemplo con C# y LINQ.

Identificación del problema

No existe documentación que se enfoque al tema Arquitectura de Microsoft .Net utilizando LINQ en detalle, y el alumno de Licenciatura en Sistemas Computaciones no tiene acceso a estas técnicas.

Objetivo General

Elaborar un documento dirigido a los alumnos de la Lic. En Sistemas Computacionales para que conozcan y utilicen la arquitectura de LINQ, para facilitar el desarrollo de sistemas de información.

Objetivos Específicos

- Documentar la Arquitectura de desarrollo de Software.
- Diseñar el modelo de datos de un sistema.
- Explicar la Arquitectura LINQ.
- Ejemplificar el uso de la herramienta LINQ.

Justificación del Tema

Este trabajo da una visión general del desarrollo de software y en especifico del mapeador de objetos relacional ORM (Object-Relational Mapping), para este tema LINQ (Language-Integrated Query, Lenguaje de Consultas Integrado) que es una herramienta más en el desarrollo de aplicaciones de software, siendo estos de gran utilidad para agilizar su elaboración, y que permiten un manejo más robusto de las operaciones a las Bases de Datos.

Así este documento queda como una alternativa para quien se interese en aplicar esta forma de desarrollo de software con sus técnicas y herramientas. Además de que se ejemplifica una aplicación para que los alumnos de la Lic. En Sistemas Computacionales manejen este concepto y lo puedan incorporar en su actividad laboral en donde se requiere de sistemas de información de gran tamaño.

CAPÍTULO 1. Marco Teórico

1.1.-Programación Orientada a Objetos

La programación orientada a objetos, POO (OOP, Object Oriented Programming), entiende a la actividad de desarrollar software basados en el paradigma de orientación a objetos. Dentro este ámbito, la programación es adquisición de conocimiento de la realidad que se desea modelar, en favor de construir un modelo computacional de la misma. Se asocian entes de la realidad, objetos del mundo computacional con el objetivo de construir un modelo de simulación de la misma (BARNES & KOLLING, 2007, pág. 3).

La programación orientada a objetos es un "modo de ver las cosas", por supuesto que se puede desarrollar alguna aplicación para resolver un problema sin utilizar POO, sin embargo se ha demostrado que la POO aporta grandes beneficios al proceso de desarrollo y mantenimiento de software.

Si se utiliza POO lo que se hace es representar objetos de mundo real mediante código, un ejemplo:

Se tiene un cliente que es distribuidor de Relojes, y desea tener control de los relojes que tiene en existencia. Identificando los objetos del mundo real entre ellos el objeto "Reloj", y se debe representar en la aplicación (mediante código) con las características que vayan a utilizar, ver figura 1.

Figura 1: Representación de un Objeto

Los objetos representados en la aplicación pueden tener propiedades (también llamadas atributos) y/o métodos (también llamados funciones u operaciones), de modo sencillo las propiedades son las características del objeto y los métodos son las acciones que este puede realizar, en nuestro ejemplo del objeto “Reloj” los datos marca, modelo, color y precio son propiedades y cada reloj tiene un valor determinado para ellas, algunas de las acciones que puede realizar un reloj son:

1. Activar Alarma
2. Detener Alarma.

Sin embargo aquí hay un detalle, dichas acciones que el objeto real tiene no son necesarias para nuestra aplicación de control de inventarios, por lo cual no es necesario representarlas, por lo tanto no todas las propiedades o métodos serán necesarios, por otra parte así como se quita también puede agregar algunas cosas, seguramente al cliente le interesara saber la fecha de entrada del objeto al inventario, el precio en que se adquirió y el precio de venta.

La programación orientada a objetos implica entre otros beneficios, gran capacidad de reuso. Dado que en la realidad se resuelven problemas a través de que las diferentes entidades de la misma colaboren, una vez que se ha encontrado un mecanismo para resolver un problema, se utiliza éste para alcanzar el mismo resultado exitoso una y otra vez. Dado que el modelo computacional es una simulación de esta realidad, el reuso se presenta en ambos sentidos.

Se puede definir un objeto como un conjunto complejo de datos y programas que poseen estructura y forman parte de una organización.

Esta definición especifica varias propiedades importantes de los objetos. En primer lugar, un objeto no es un dato simple, sino que contiene en su interior cierto número de componentes bien estructurados. En segundo lugar, cada objeto no es una entidad aislada, sino que forma parte de una organización jerárquica o de otro tipo.

1.1.1.-Clase

Las clases son declaraciones de objetos, también se podrían definir como abstracciones de objetos (JOYANES AGUILAR, 1996, pág. 75). Esto quiere decir que la definición de un objeto es la clase. Cuando se programa un objeto y se define sus características y funcionalidades en realidad lo que se hace es programar una clase.

La clasificación se basa en un comportamiento y atributos comunes. Permite crear un vocabulario estandarizado para comunicarse y pensar dentro del equipo de trabajo.

Una clase es una construcción estática que describe un comportamiento común y atributos (que toman distintos estados) (JOYANES AGUILAR, 1996, pág. 76). Su formalización es a través de una estructura de datos que incluye datos y funciones, llamadas métodos. Los métodos son los que definen el comportamiento.

Toda clase tiene, de manera implícita o explícita, dos métodos. El constructor y el destructor.

El constructor es llamado cuando la clase comienza a ocupar un lugar en memoria, es decir, cuando comienza a ser utilizada, al ser instanciada. Se encarga de inicializar valores, e incluso muchas veces de llamar a otros constructores de otras clases.

El destructor es un método que se llama al finalizar la vida de la instancia de la clase, para liberar recursos.

Cualquiera de los dos métodos tienen que formar parte de las clases, existen constructores y destructores por defecto, para las clases más simples. En clases más complejas, el desarrollador deberá implementar estos métodos. Muchas veces, los constructores reciben parámetros con los valores con los que la clase debe inicializarse. Las llamadas a constructores y destructores pueden ser implícitas o explicitas, dependiendo de la clase, la aplicación, etc.

1.1.2.-Objeto

Los objetos son instancias de una clase. Cuando se crea una instancia debe especificarse la clase a partir de la cual se creará. Esta acción de crear un objeto a partir de una clase se llama instanciar (JOYANES AGUILAR, 1996, págs. 70-71).

Por ejemplo, un objeto de la clase Cliente es por ejemplo Juan. El concepto o definición de Cliente sería la clase, pero cuando se habla de un Cliente en concreto Pedro, Adrián o cualquier otro, será un objeto.

Identidad: El principio de identidad se refiere a la relación única entre cada objeto del modelo computacional y el dominio de problema de la realidad que representa. En el modelo computacional esto se representa a través de un identificador único en el modelo.

Comportamiento que le permite realizar tareas específicas, como a todos los objetos de su misma clase.

Estado que se determina a través de cierta información almacenada, que puede ser fija o variable.

1.2.-Características de la Programación Orientada a Objetos

Dado que la POO se basa en la idea natural de la existencia de un mundo lleno de objetos y que la resolución del problema se realiza en términos de objetos, un lenguaje se dice que es orientado a objetos, si los elementos esenciales de construcción del software son objetos y mensajes. Para realizar esta representación se entienden los siguientes conceptos que la POO.

1.2.1.-Abstracción

La *abstracción* de datos permite no preocuparse de los detalles no esenciales. Existe en casi todos los lenguajes de programación. Las estructuras de datos y los tipos de datos son un ejemplo de abstracción. Los procedimientos y funciones son otro ejemplo (JOYANES AGUILAR, 1996, págs. 10-11,17).

Es la capacidad de un objeto de cumplir sus funciones independientemente del contexto en el que se lo utilice; o sea, un objeto “cliente” siempre expondrá sus mismas propiedades y dará los mismos resultados a través de sus eventos, sin importar el ámbito en el cual se lo haya creado. Es poder generalizar un objeto como tipo de dato, con sus características y comportamientos comunes.

Es la característica que permite usar objetos sin preocuparse por el modo en que funcionan internamente, el ejemplo clásico del Auto, uno puede tener un auto y utilizarlo sin saber ni preocuparse por el modo en que funciona, en eso precisamente consiste la abstracción, el objeto puede cambiar los valores de sus propiedades, ejecutar algún tipo de trabajo o método y/o comunicarse con otros objetos sin revelar el modo en que internamente lo hace, de este modo el programador que lo utiliza conoce de modo “abstracto” las propiedades y métodos que puede realizar un objeto, pero no necesita preocuparse entender el funcionamiento real del objeto (muchas personas utilizan este principio con sus autos).

1.2.2.-Encapsulamiento

Esta característica es la que denota la capacidad del objeto de responder a peticiones a través de sus métodos sin la necesidad de exponer los medios utilizados para llegar a brindar estos resultados (JOYANES AGUILAR, 1996, pág. 18). Ejemplo se tiene el Objeto Auto, este tiene métodos como Acelerar, Frenar, Velocidad, ver figura 2. Si el auto Acelera siempre aumentará la velocidad, sin necesidad de tener conocimiento de cuáles son los recursos que ejecuta para llegar a brindar este resultado.

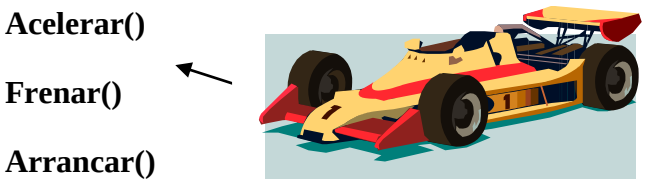

Figura 2: Encapsulamiento.

Es común confundir esta característica con el "Principio de ocultación", el encapsulamiento consiste en colocar dentro de un objeto todas sus características y funcionalidad correspondiente, el objeto agrupa o "encapsula" esas características, a nuestro objeto Auto no se debe ponerle un método que se llame "volar" puesto que los autos no vuelan, el encapsulamiento nos permite mantener una "alta cohesión", esto significa que toda la información del objeto se encuentra precisamente en el objeto puesto que es información que está relacionada y en conjunto define al objeto.

Los objetos deben comunicarse solo a través de su protocolo de modo que la responsabilidad en administrar tanto el comportamiento como el respetar los cambios de clase sea solo del objeto que implementa dicho comportamiento. Esto además es una condición necesaria para permitir el polimorfismo.

La utilidad del encapsulamiento va por la facilidad para manejar la complejidad, ya que las Clases se ven como cajas negras donde sólo se conoce el comportamiento pero no los detalles internos, y esto es conveniente porque solo interesará conocer qué hace la Clase pero no será necesario saber cómo lo hace.

El encapsulamiento también es llamado "ocultamiento de la información", esto asegura que los objetos no pueden cambiar el estado interno de otros objetos de maneras inesperadas; solamente los propios métodos internos del objeto pueden acceder a su estado. Cada tipo de objeto expone una *interfaz* a otros objetos que especifica cómo

otros objetos pueden interactuar con él. Algunos lenguajes permiten un acceso directo a los datos internos del objeto de una manera controlada y limitando el grado de abstracción (JOYANES AGUILAR, 1996, pág. 18).

1.2.3.-Principio de ocultación

Esta característica permite que el objeto defina que características o métodos pueden ser vistos y ejecutados desde el exterior por otros objetos, y "oculta" a los objetos externos las características y métodos que sean necesarios solamente para uso interno, por ejemplo el auto tiene un método llamado "arrancar" que puede ser ejecutado por un objeto externo (el chofer) sin embargo puede tener un método llamado "inyectar combustible" que puede ser invocado solo por el mismo objeto, en este último caso el método puede ser establecido con acceso privado, aunque también se puede establecer de tipo protected o según sea el caso y de este modo nadie más podrá ejecutarlo.

1.2.4.-Polimorfismo

Cuando los lenguajes orientados a objetos son fuertemente tipados, esto es que se especifica el tipo de argumento de entrada y salida de los objetos en forma implícita o explícita, se puede decidir para cada conjunto de tipos de argumento (o cantidad de argumentos) de entrada un método diferente que resuelve dicho mensaje, el compilador decide cual método aplica en función de esta información de tipos sobre los argumentos, a esto se lo conoce como sobrecarga.

Cuando el objeto cliente desconoce la clase concreta asociada al objeto servidor, permitiéndonos intercambiar diferentes servidores que aplican diferentes métodos para resolver un problema, esto es polimorfismo.

El término de polimorfismo también define la capacidad de que más de un objeto puedan crearse usando la misma clase de base para lograr dos conceptos de objetos diferentes, en este caso se puede citar el ejemplo de los teléfonos, los cuales se basan en un teléfono base, con la capacidad de hacer ring y tener un auricular, para luego obtener un teléfono digital, inalámbrico, con botonera de marcado y también, tomando la misma base, construir un teléfono analógico y con disco de marcado (JOYANES AGUILAR, 1996, pág. 19).

Si en el momento de la compilación de un programa se conoce la clase concreta del objeto que se usará, las llamadas a sus métodos quedarán fijadas mediante lo que se conoce como "enlace estático o temprano" (early binding), si no, habrá de determinarse

la llamada adecuada en tiempo de ejecución, efectuándose entonces un "enlace dinámico o tardío" (late binding).

El polimorfismo en orientación a objetos “es hacer algo de diferentes formas”, es una forma muy sencilla de explicarlo, el polimorfismo permite tener comportamientos diferentes con el mismo nombre en objetos distintos, al tener el objeto “Auto eléctrico” y el objeto “Auto de combustión” (ahí están los objetos distintos), ambos cuentan con un método llamado “arrancar” (ahí está el método con el mismo nombre en ambos objetos), sin embargo el método “arrancar” del “Auto eléctrico” provoca que inicie el funcionamiento de un motor eléctrico, y el método “arrancar” del “Auto de combustión” provoca la inyección de gasolina, pone a girar el motor e inicia la combustión para dejarlo funcionando.

1.2.5.-Herencia

La herencia es uno de los conceptos más importantes en la POO, básicamente consiste en que una clase puede heredar sus variables y métodos a varias subclases, la clase que hereda es llamada superclase o clase padre (JOYANES AGUILAR, 1996, págs. 84-86). Esto significa que una subclase, aparte de los atributos y métodos propios, tiene incorporados los atributos y métodos heredados de la superclase, de esta manera se crea una jerarquía de herencia, ver figura 3.

Relación “es un” significa que la clase hija (o heredera), es, además, lo mismo que su padre, es decir, un auto “es un” transporte, un caballo “es un” animal, etc.

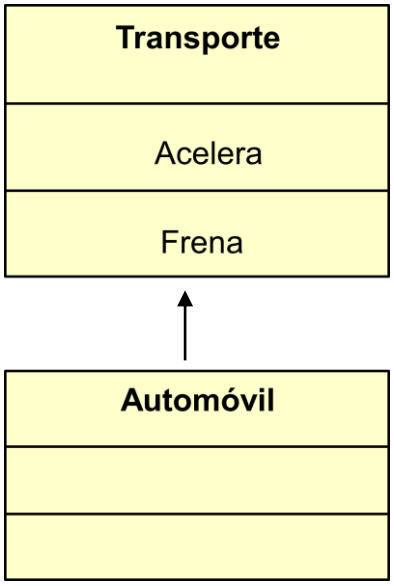

Figura 3: Herencia

Estos pueden compartir (y extender) su comportamiento sin tener que reimplementar su comportamiento. Esto suele hacerse habitualmente agrupando los objetos en clases y las clases en árboles o enrejados que reflejan un comportamiento común.

La herencia permite definir objetos generales y posteriormente crear otros objetos que desciendan de ellos por lo tanto contienen sus características y definen algunas otras que los diferencian de los demás objetos descendientes del mismo objeto padre de este modo los objetos forman una jerarquía de clasificación en la cual conforme desciende se encuentran objetos cada vez más detallados, por ejemplo volviendo al ejemplo del Auto, dicho objeto puede contener las características y comportamiento comunes de todos los autos y posteriormente crear otros objetos que hereden de Auto como serian "Auto eléctrico" y "Auto de combustión", cada uno de ellos heredara las propiedades y comportamientos comunes de un auto y a su vez agregaran las características que los hacen especiales a cada uno de ellos.

1.3.-Base de Datos

Una base de datos es la representación integrada de los conjuntos de entidades instancia correspondiente a las diferentes entidades tipo del y de sus interrelaciones. Esta representación informática (o conjunto estructurado de datos) debe poder ser utilizada de forma compartida por muchos usuarios de distintos tipos.

En la estructura de la Base de Datos se encuentra el modelo de datos: una colección de herramientas conceptuales para describir los datos, las relaciones, la semántica y las restricciones de consistencia. Para mostrar el concepto de un modelo de datos, se describen dos modelos de datos: el modelo entidad-relación y el modelo relacional. Los diferentes modelos de datos que se han propuesto se clasifican en tres grupos diferentes: modelos lógicos basados en objetos, modelos lógicos basados en registros y modelos físicos (SILBERSCHATZ, KORTH, & SUDARSHAN, 2002, págs. 1-2).

1.3.1.-Modelo entidad-relación

El modelo de datos entidad-relación (E-R) está basado en una percepción del mundo real que consta de una colección de objetos básicos, llamados entidades, y de relaciones entre estos objetos. Una entidad es una «cosa» u «objeto» en el mundo real que es distinguible de otros objetos (SILBERSCHATZ, KORTH, & SUDARSHAN, 2002, pág. 5). Por ejemplo, cada persona es una entidad, y las cuentas bancarias pueden ser consideradas entidades.

Las entidades se describen en una Base de Datos mediante un conjunto de atributos. Por ejemplo, los atributos número-cuenta y saldo describen una cuenta particular de un banco y pueden ser atributos del conjunto de entidades cuenta. Análogamente, los atributos nombre-cliente, calle-cliente y ciudad-cliente pueden describir una entidad cliente.

Un atributo extra, id-cliente, se usa para identificar unívocamente a los clientes (dado que puede ser posible que haya dos clientes con el mismo nombre, dirección y ciudad. Se debe asignar un identificador único de cliente a cada cliente.

Una relación es una asociación entre varias entidades. Por ejemplo, una relación impositor asocia un cliente con cada cuenta que tiene. El conjunto de todas las entidades del mismo tipo, y el conjunto de todas las relaciones del mismo tipo, se denominan respectivamente conjunto de entidades y conjunto de relaciones.

La estructura lógica general de una Base de Datos se puede expresar gráficamente mediante un diagrama ER, que consta de los siguientes componentes:

- Rectángulos, que representan conjuntos de entidades.
- Elipses, que representan atributos.
- Rombos, que representan relaciones entre conjuntos de entidades.
- Líneas, que unen los atributos con los conjuntos de entidades y los conjuntos de entidades con las relaciones.

Cada componente se etiqueta con la entidad o relación que representa. Como ilustración, considérese parte de una Base de Datos de un sistema bancario consistente en clientes y cuentas que tienen esos clientes. En la figura 4 se muestra el diagrama E-R correspondiente. El diagrama E-R indica que hay dos conjuntos de entidades cliente y cuenta, con los atributos descritos anteriormente. El diagrama también muestra la relación impositor entre cliente y cuenta.

Además de entidades y relaciones, el modelo E-R representa ciertas restricciones que los contenidos de la Base de Datos deben cumplir. Una restricción importante es la correspondencia de cardinalidades, que expresa el número de entidades con las que otra entidad se puede asociar a través de un conjunto de relaciones (SILBERSCHATZ, KORTH, & SUDARSHAN, 2002, pág. 8). Por ejemplo, si cada cuenta puede pertenecer sólo a un cliente, el modelo puede expresar esta restricción.

El modelo entidad-relación se utiliza habitualmente en el proceso de diseño de Bases de Datos.

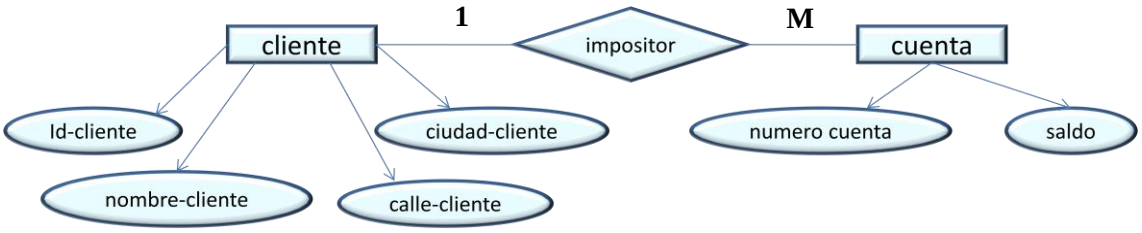

Figura 4: Modelo Entidad-Relación

1.3.1.1- Relaciones

Las entidades no están aisladas sino que están relacionadas entre sí. Estas relaciones pueden ser de tres tipos diferentes, ver figura 5:

- 1 a 1
- 1 a muchos (1 a N)
- Muchos a muchos (M a N)

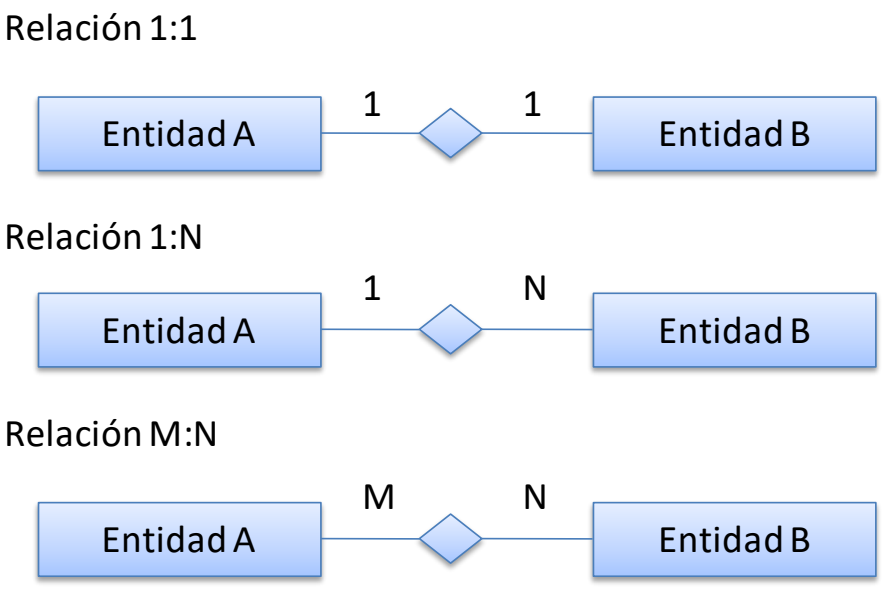

Figura 5: Relaciones

1.3.2.-Modelo Relacional

En el modelo relacional se utiliza un grupo de tablas para representar los datos y las relaciones entre ellos. Cada tabla está compuesta por varias columnas, y cada columna tiene un nombre único.

El modelo relacional es un ejemplo de un modelo basado en registros. Los modelos basados en registros se denominan así porque la Base de Datos se estructura en registros de formato fijo de varios tipos (SILBERSCHATZ, KORTH, & SUDARSHAN, 2002, pág. 6). Cada tabla contiene registros de un tipo particular. Cada tipo de registro define un número fijo de campos, o atributos. Las columnas de la tabla corresponden a los atributos del tipo de registro.

No es difícil ver cómo se pueden almacenar las tablas en archivos. Por ejemplo, un carácter especial (como una coma) se puede usar para delimitar los diferentes atributos de un registro, y otro carácter especial (como un carácter de nueva línea) se puede usar para delimitar registros. El modelo relacional oculta tales detalles de implementación de bajo nivel a los desarrolladores de Bases de Datos y usuarios.

El modelo de datos relacional es el modelo de datos más ampliamente usado, y una amplia mayoría de sistemas de Bases de Datos actuales se basan en el modelo relacional.

1.3.3.-Estructura de un Sistema de Base de Datos

Un sistema de Bases de Datos se divide en módulos que se encargan de cada una de las responsabilidades del sistema completo. Los componentes funcionales de un sistema de Bases de Datos se pueden dividir a grandes rasgos en los componentes, gestor de almacenamiento y procesador de consultas (SILBERSCHATZ, KORTH, & SUDARSHAN, 2002, pág. 10). El gestor de consultas es importante porque las Bases de Datos requieren normalmente una gran cantidad de espacio de almacenamiento. Las Bases de Datos corporativas tienen un tamaño de entre cientos de gigabytes y, para las mayores Bases de Datos, terabytes de datos.

Debido a que la memoria principal de las computadoras no puede almacenar esta gran cantidad de información, esta se almacena en discos. Los datos se trasladan entre el disco de almacenamiento y la memoria principal cuando es necesario. Como la transferencia de datos a y desde el disco es lenta comparada con la velocidad de la unidad central de procesamiento, es fundamental que el sistema de Base de Datos

estructure los datos para minimizar la necesidad de movimiento de datos entre el disco y la memoria principal.

El procesador de consultas es importante porque ayuda al sistema de Bases de Datos a simplificar y facilitar el acceso a los datos. Las vistas de alto nivel ayudan a conseguir este objetivo. Con ellas, los usuarios del sistema no deberían ser molestados innecesariamente con los detalles físicos de implementación del sistema. Sin embargo, el rápido procesamiento de las actualizaciones y de las consultas es importante. Es trabajo del sistema de Bases de Datos traducir las actualizaciones y las consultas escritas en un lenguaje no procedimental, en el nivel lógico, en una secuencia de operaciones en el nivel físico.

1.3.3.1.-Gestor de almacenamiento

Un gestor de almacenamiento es un módulo de programa que proporciona la interfaz entre los datos de bajo nivel en la Base de Datos y los programas de aplicación y consultas emitidas al sistema. El gestor de almacenamiento es responsable de la interacción con el gestor de archivos. Los datos en bruto se almacenan en disco usando un sistema de archivos, que está disponible habitualmente en un sistema operativo convencional. El gestor de almacenamiento traduce las diferentes instrucciones LMD (Lenguaje de Manipulación de Datos) a órdenes de un sistema de archivos de bajo nivel. Así, el gestor de almacenamiento es responsable del almacenamiento, recuperación y actualización de los datos en la Base de Datos (SILBERSCHATZ, KORTH, & SUDARSHAN, 2002, págs. 11-12).

Los componentes del gestor de almacenamiento incluyen:

- Gestor de autorización e integridad, que comprueba que se satisfagan las restricciones de integridad y la autorización de los usuarios para acceder a los datos.

- Gestor de transacciones, que asegura que la Base de Datos quede en un estado consistente (correcto) a pesar de los fallos del sistema, y que las ejecuciones de transacciones concurrentes ocurran si conflictos.

- Gestor de archivos, que gestiona la reserva de espacio de almacenamiento de disco y las estructuras de datos usadas para representar la información almacenada en disco.

• Gestor de memoria intermedia, que es responsable de traer los datos del disco de almacenamiento a memoria principal y decidir qué datos tratar en memoria caché. El gestor de memoria intermedia es una parte crítica del sistema de Bases de Datos, ya que permite que la Base de Datos maneje tamaños de datos que son mucho mayores que el tamaño de la memoria principal.

El gestor de almacenamiento implementa varias estructuras de datos como parte de la implementación física del sistema:

• Archivos de datos, que almacenan la Base de Datos en sí.

• Diccionario de datos, que almacena metadatos acerca de la estructura de la Base de Datos, en particular, el esquema de la Base de Datos.

• Índices, que proporcionan acceso rápido a elementos de datos que tienen valores particulares.

1.3.3.2.-Procesador de consultas

Los componentes del procesador de consultas incluyen:

• Intérprete del LDD o Lenguaje de definición de datos, que interpreta las instrucciones del LDD y registra las definiciones en el diccionario de datos.

• Compilador del LMD o Lenguaje de manipulación de datos, que traduce las instrucciones del LMD en un lenguaje de consultas a un plan de evaluación que consiste en instrucciones de bajo nivel que entiende el motor de evaluación de consultas.

Una consulta se puede traducir habitualmente en varios planes de ejecución alternativos que proporcionan el mismo resultado. El compilador del LMD (Lenguaje de manipulación de datos) también realiza optimización de consultas, es decir, elige el plan de evaluación de menor coste de entre todas las alternativas.

• Motor de evaluación de consultas, que ejecuta las instrucciones de bajo nivel generadas por el compilador del LMD (Lenguaje de manipulación de datos).

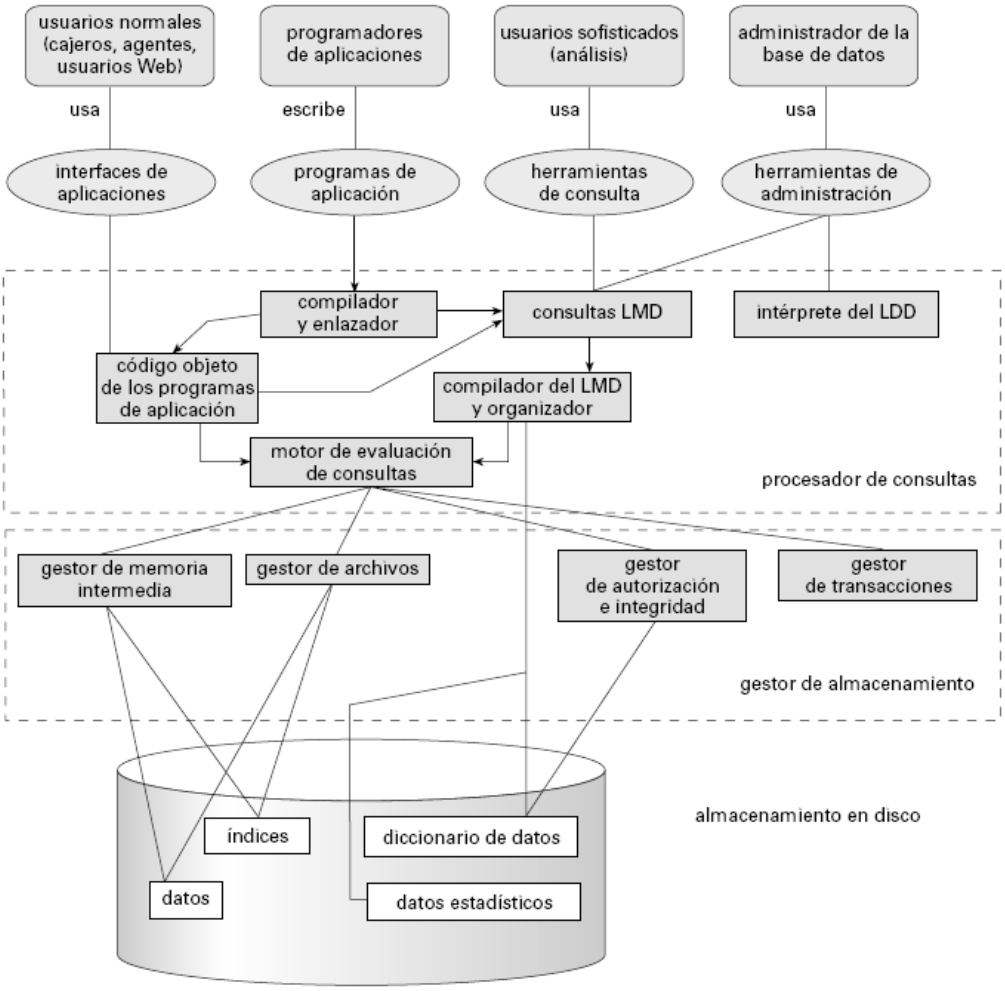

Figura 6: Esquema de evaluación de consultas

La mayoría de usuarios de un sistema de Bases de Datos no están situados actualmente junto al sistema de Bases de Datos, sino que se conectan a él a través de una red. Se puede diferenciar entonces entre las máquinas cliente, en donde trabajan los usuarios remotos de la Base de Datos, y las máquinas servidor, en las que se ejecuta el sistema de Bases de Datos.

Las aplicaciones de Bases de Datos se dividen usualmente en dos o tres partes, como se ilustra en la figura 7. En una arquitectura de dos capas, la aplicación se divide en un componente que reside en la máquina cliente, que llama a la funcionalidad del sistema de Bases de Datos en la máquina servidor mediante instrucciones del lenguaje de consultas. Los estándares de interfaces de programas de aplicación como ODBC y JDBC se usan para la interacción entre el cliente y el servidor.

En cambio, en una arquitectura de tres capas, la máquina cliente actúa simplemente como frontal y no contiene ninguna llamada directa a la Base de Datos. En su lugar, el cliente se comunica con un servidor de aplicaciones, usualmente mediante una interfaz de formularios.

El servidor de aplicaciones, a su vez, se comunica con el sistema de Bases de Datos para acceder a los datos. La lógica de negocio de la aplicación, que establece las acciones a realizar bajo determinadas condiciones, se incorpora en el servidor de aplicaciones, en lugar de ser distribuida a múltiples clientes. Las aplicaciones de tres capas son más apropiadas para grandes aplicaciones, y para las aplicaciones que se ejecutan en World Wide Web.

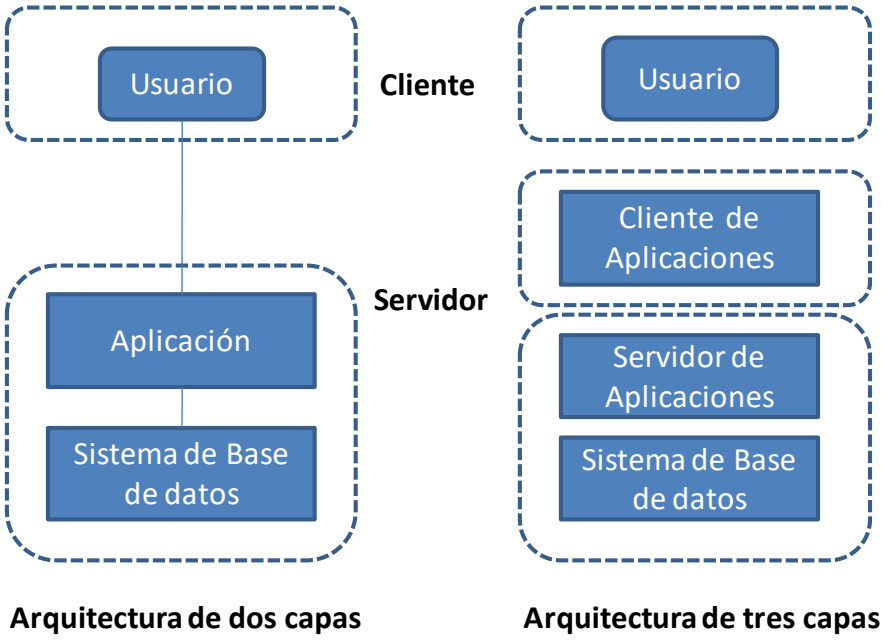

Figura 7: Lógica de negocio de una aplicación dos y tres capas.

1.3.4.-Normalización

Para que un modelo sea correcto debe cumplir tres normas, que se conocen como las tres formas de normalización:

1.3.4.1.-Primera Forma Normal (1FN)

La primera forma normal si y solo si todas las columnas de todas las tablas contienen solo valores atómicos, es decir solo un único valor (SILBERSCHATZ, KORTH, & SUDARSHAN, 2002, pág. 161).

1.3.4.2.-Segunda Forma Normal (2FN)

Todos los atributos que no forman parte de la clave, deben depender de ella en conjunto y nunca de un subconjunto de la misma (SILBERSCHATZ, KORTH, & SUDARSHAN, 2002, pág. 176).

1.3.4.3.-Tercera Forma Normal (3FN)

Todos los atributos que no forman parte de la clave primaria deben de ser independientes entre sí (SILBERSCHATZ, KORTH, & SUDARSHAN, 2002, pág. 177).

1.3.5.- Ventajas e inconvenientes de los sistemas de bases de datos

Los sistemas de bases de datos presentan numerosas ventajas que se pueden dividir en dos grupos: las que se deben a la integración de datos y las que se deben a la interface común que proporciona el SGBD (Sistema de Gestión de Base de Datos).

1.3.5.1.-Ventajas por la integración de datos

- **Control sobre la redundancia de datos**. Los sistemas de ficheros almacenan varias copias de los mismos datos en ficheros distintos. Esto hace que se desperdicie espacio de almacenamiento, además de provocar la falta de consistencia de datos. En los sistemas de bases de datos todos estos ficheros están integrados, por lo que no se almacenan varias copias de los mismos datos. Sin embargo, en una base de datos no se puede eliminar la redundancia completamente, ya que en ocasiones es necesaria para modelar las relaciones entre los datos, o bien es necesaria para mejorar las prestaciones (CAMPS PARÉ, CASILLAS SANTILLAN, & COSTAL COSTA, 2005, pág. 7).
- **Consistencia de datos**. Eliminando o controlando las redundancias de datos se reduce en gran medida el riesgo de que haya inconsistencias. Si un dato está almacenado una sola vez, cualquier actualización se debe realizar sólo una vez, y está disponible para todos los usuarios inmediatamente. Si un dato está duplicado y el sistema conoce esta redundancia, el propio sistema puede encargarse de garantizar que todas las copias se mantienen consistentes (SILBERSCHATZ, KORTH, & SUDARSHAN, 2002, págs. 403-404).

- **Compartición de datos**. En los sistemas de ficheros, los ficheros pertenecen a las personas o a los departamentos que los utilizan. Pero en los sistemas de bases de datos, la base de datos pertenece a la empresa y puede ser compartida por todos los usuarios que estén autorizados. Además, las nuevas aplicaciones que se vayan creando pueden utilizar los datos de la base de datos existente.
- **Mantenimiento de estándares**. Gracias a la integración es más fácil respetar los estándares necesarios, tanto los establecidos a nivel de la empresa como los nacionales e internacionales. Estos estándares pueden establecerse sobre el formato de los datos para facilitar su intercambio, pueden ser estándares de documentación, procedimientos de actualización y también reglas de acceso.

1.3.5.2.-Ventajas por la existencia del SGBD (Sistema de Gestión de Base de Datos)

- **Mejora en la integridad de datos**. La integridad de la base de datos se refiere a la validez y la consistencia de los datos almacenados. Normalmente, la integridad se expresa mediante restricciones o reglas que no se pueden violar. Estas restricciones se pueden aplicar tanto a los datos, como a sus relaciones, y es el SGBD quien se debe encargar de mantenerlas (CAMPS PARÉ, CASILLAS SANTILLAN, & COSTAL COSTA, 2005, pág. 18).
- **Mejora en la accesibilidad a los datos**. Muchos SGBD proporcionan lenguajes de consultas o generadores de informes que permiten al usuario hacer cualquier tipo de consulta sobre los datos, sin que sea necesario que un programador escriba una aplicación que realice tal tarea.
- **Mejora en la productividad**. El SGBD proporciona muchas de las funciones estándar que el programador necesita escribir en un sistema de ficheros. A nivel básico, el SGBD proporciona todas las rutinas de manejo de ficheros típicas de los programas de aplicación.
- **Mejora en el mantenimiento gracias a la independencia de datos**. En los sistemas de ficheros, las descripciones de los datos se encuentran inmersas en los programas de aplicación que los manejan. Esto hace que los programas sean dependientes de los datos, de modo que un cambio en su estructura, o un cambio en el modo en que se almacena en disco, requiere cambios importantes en los programas cuyos datos se ven afectados.
- **Aumento de la concurrencia**. En algunos sistemas de ficheros, si hay varios usuarios que pueden acceder simultáneamente a un mismo fichero, es posible que el acceso interfiera entre ellos de modo que se pierda información o, incluso, que se pierda la integridad. La mayoría de los SGBD gestionan el acceso concurrente a la base de datos y garantizan que no ocurran problemas de este tipo.

- **Mejora en los servicios de copias de seguridad y de recuperación ante fallos**. Muchos sistemas de ficheros dejan que sea el usuario quien proporcione las medidas necesarias para proteger los datos ante fallos en el sistema o en las aplicaciones. Los usuarios tienen que hacer copias de seguridad cada día, y si se produce algún fallo, utilizar estas copias para restaurarlos. En este caso, todo el trabajo realizado sobre los datos desde que se hizo la última copia de seguridad se pierde y se tiene que volver a realizar. Sin embargo, los SGBD actuales funcionan de modo que se minimiza la cantidad de trabajo perdido cuando se produce un fallo (CAMPS PARÉ, CASILLAS SANTILLAN, & COSTAL COSTA, 2005, pág. 18).

Inconvenientes de los sistemas de bases de datos

- **Complejidad**. Los SGBD son conjuntos de programas muy complejos con una gran funcionalidad. Es preciso comprender muy bien esta funcionalidad para poder sacar un buen partido de ellos.
- **Tamaño**. Los SGBD son programas complejos y muy extensos que requieren una gran cantidad de espacio en disco y de memoria para trabajar de forma eficiente.
- **Coste económico del SGBD**. El coste de un SGBD varía dependiendo del entorno y de la funcionalidad que ofrece. Además, hay que pagar una cuota anual de mantenimiento que suele ser un porcentaje del precio del SGBD.
- **Prestaciones**. Un sistema de ficheros está escrito para una aplicación específica, por lo que sus prestaciones suelen ser muy buenas. Sin embargo, los SGBD están escritos para ser más generales y ser útiles en muchas aplicaciones, lo que puede hacer que algunas de ellas no sean tan rápidas como antes (CAMPS PARÉ, CASILLAS SANTILLAN, & COSTAL COSTA, 2005, págs. 7,8,9,22).
- **Vulnerable a los fallos**. El hecho de que todo esté centralizado en el SGBD hace que el sistema sea más vulnerable ante los fallos que puedan producirse.

1.4.- SQL (Lenguaje de Consulta Estructurado)

El lenguaje de consulta estructurado (SQL) es un lenguaje de Base de Datos normalizado, utilizado por el motor de Base de Datos de Microsoft Jet. SQL se utiliza para crear objetos QueryDef, como el argumento de origen del método OpenRecordSet y como la propiedad RecordSource del control de datos. También se puede utilizar con el método Execute para crear y manipular directamente las Bases de Datos Jet y crear consultas SQL de paso a través para manipular Bases de Datos remotas cliente - servidor.

1.4.1.-Componentes SQL

El lenguaje SQL está compuesto por comandos, cláusulas, operadores y funciones de agregado. Estos elementos se combinan en las instrucciones para crear, actualizar y manipular las Bases de Datos (Manual Basico del Lenguaje SQL, 2007).

1.4.2.-Comandos

Existen dos tipos de comandos SQL:

- DLL (Dynamic Linking Library) o Librería dinámica de enlaces que permiten crear y definir nuevas Bases de Datos, campos e índices.
- DML (Data Manipulation Language) o Lenguaje de manipulación de datos que permiten generar consultas para ordenar, filtrar y extraer datos de la Base de Datos.

Comandos DLL

Comando	Descripción
CREATE	Utilizado para crear nuevas tablas, campos e índices.
DROP	Empleado para eliminar tablas e índices.
ALTER	Utilizado para modificar las tablas agregando campos o cambiando la definición de los campos.

Tabla 1: Comandos DLL

Comandos DML

Comando	Descripción
SELECT	Utilizado para consultar registros de la Base de Datos que satisfagan un criterio determinado.
INSERT	Utilizado para cargar lotes de datos en la Base de Datos en una única operación.
UPDATE	Utilizado para modificar los valores de los campos y registros Especificados.
DELETE	Utilizado para eliminar registros de una tabla de una Base de Datos.

Tabla 2: Comandos DML

1.4.3.-Cláusulas

Las cláusulas son condiciones de modificación utilizadas para definir los datos que desea seleccionar o manipular (Manual Basico del Lenguaje SQL, 2007).

Cláusula	Descripción
FROM	Utilizada para especificar la tabla de la cual se van a seleccionar los registros.
WHERE	Utilizada para especificar las condiciones que deben reunir los registros que se van a seleccionar.
GROUP BY	Utilizada para separar los registros seleccionados en grupos específicos.
HAVING	Utilizada para expresar la condición que debe satisfacer cada grupo.
ORDER BY	Utilizada para ordenar los registros seleccionados de acuerdo con un orden específico.

Tabla 3: Clausulas

1.4.4.-Operadores Lógicos

Operador	Uso
AND	Es el "y" lógico. Evalúa dos condiciones y devuelve un valor de verdad sólo si ambas son ciertas.
OR	Es el "o" lógico. Evalúa dos condiciones y devuelve un valor de verdad si alguna de las dos es cierta.
NOT	Negación lógica. Devuelve el valor contrario de la expresión.

Tabla 4: Operadores Lógicos

1.4.5.-Operadores de Comparación

Operador	Uso
<	Menor que
>	Mayor que
<>	Distinto de
<=	Menor ó Igual que
>=	Mayor ó Igual que
=	Igual que
BETWEEN	Utilizado para especificar un intervalo de valores.
LIKE	Utilizado en la comparación de un modelo.
In	Utilizado para especificar registros de una Base de Datos.

Tabla 5: Operadores de Comparación

1.4.6.-Funciones de Agregado

Las funciones de agregado se usan dentro de una cláusula SELECT en grupos de registros para devolver un único valor que se aplica a un grupo de registros (Manual Basico del Lenguaje SQL, 2007).

Función	Descripción
AVG	Utilizada para calcular el promedio de los valores de un campo determinado
COUNT	Utilizada para devolver el número de registros de la selección.
SUM	Utilizada para devolver la suma de todos los valores de un campo determinado.
MAX	Utilizada para devolver el valor más alto de un campo especificado
MIN	Utilizada para devolver el valor más bajo de un campo especificado.

Tabla 6: Funciones de Agregado

1.4.7.-Consultas de Selección

Las consultas de selección se utilizan para indicar al motor de datos que devuelva información de las Bases de Datos, esta información es devuelta en forma de conjunto de registros que se pueden almacenar en un objeto recordset. Este conjunto de registros es modificable (Manual Basico del Lenguaje SQL, 2007).

1.4.7.1.-Consultas básicas

La sintaxis básica de una consulta de selección es la siguiente:

```
SELECT Campos FROM Tabla;
```

En donde campos es la lista de campos que se deseen recuperar y tabla es el origen de los mismos, por ejemplo:

```
SELECT Nombre, Cuenta FROM Clientes;
```

Esta consulta devuelve un recordset con el campo nombre y cuenta de la tabla clientes.

1.4.7.2.-Ordenar los registros

Adicionalmente se puede especificar el orden en que se desean recuperar los registros de las tablas mediante la cláusula ORDER BY Lista de Campos. En donde

Lista de campos representa los campos a ordenar. Ejemplo:

```
SELECT CodigoPostal, Nombre, Cuenta FROM Clientes ORDER BY Nombre;
```

Esta consulta devuelve los campos CodigoPostal, Nombre, Cuenta de la tabla Clientes ordenados por el campo Nombre.

Se pueden ordenar los registros por más de un campo, como por ejemplo:

```
SELECT CodigoPostal, Nombre, Cuenta FROM Clientes ORDER BY CodigoPostal, Nombre;
```

Incluso se puede especificar el orden de los registros: ascendente mediante la cláusula (ASC -se toma este valor por defecto) o descendente (DESC)

```
SELECT CodigoPostal, Nombre, Cuenta FROM Clientes ORDER BY CodigoPostal DESC , Nombre ASC;
```

1.4.7.3.-Consultas con Predicado

El predicado se incluye entre la cláusula y el primer nombre del campo a recuperar, los posibles predicados son:

Predicado	Descripción
ALL	Devuelve todos los campos de la tabla
TOP	Devuelve un determinado número de registros de la tabla.
DISTINCT	Omite los registros cuyos campos seleccionados coincidan totalmente.
DISTINCTROW	Omite los registros duplicados basándose en la totalidad del registro y no sólo en los campos seleccionados.
ALL	Si no se incluye ninguno de los predicados se asume ALL. El Motor de Base de Datos selecciona todos los registros que cumplen las condiciones de la instrucción SQL. No se conveniente abusar de este predicado ya que obliga al motor de la Base de Datos a analizar la estructura de la tabla para averiguar los campos que contiene, es mucho más rápido indicar el listado de campos deseados.

Tabla 7: Consultas con predicado

1.4.7.4.-Consultas de Acción

Las consultas de acción son aquellas que no devuelven ningún registro, son las encargadas de acciones como añadir y borrar y modificar registros.

DELETE

Crea una consulta de eliminación que elimina los registros de una o más de las tablas listadas en la cláusula FROM que satisfagan la cláusula WHERE. Esta consulta elimina los registros completos, no es posible eliminar el contenido de algún campo en concreto. Su sintaxis es:

```
DELETE Tabla.* FROM Tabla WHERE criterio
```

DELETE es especialmente útil cuando se desea eliminar varios registros. En una instrucción DELETE con múltiples tablas, debe incluir el nombre de tabla (Tabla.*). Si especifica más de una tabla desde la que eliminar registros, todas deben ser tablas de muchos a uno. Si desea eliminar todos los registros de una tabla, eliminar la propia tabla es más eficiente que ejecutar una consulta de borrado (Manual Basico del Lenguaje SQL, 2007).

Se puede utilizar DELETE para eliminar registros de una única tabla o desde varios lados de una relación uno a muchos. Las operaciones de eliminación en cascada en una consulta únicamente eliminan desde varios lados de una relación.

Por ejemplo, en la relación entre las tablas Clientes y Pedidos, la tabla Pedidos es la parte de muchos por lo que las operaciones en cascada solo afectaran a la tabla Pedidos. Una consulta de borrado elimina los registros completos, no únicamente los datos en campos específicos. Si desea eliminar valores en un campo especificado, crear una consulta de actualización que cambie los valores a Null.

Una vez que se han eliminado los registros utilizando una consulta de borrado, no puede deshacer la operación. Si desea saber qué registros se eliminarán, primero examine los resultados de una consulta de selección que utilice el mismo criterio y después ejecute la consulta de borrado. Mantenga copias de seguridad de sus datos en todo momento. Si elimina los registros equivocados podrá recuperarlos desde las copias de seguridad.

```
DELETE * FROM Empleados WHERE Cargo = 'Vendedor';
```

INSERT INTO

Agrega un registro en una tabla. Se la conoce como una consulta de datos añadidos. Esta consulta puede ser de dos tipos: Insertar un único registro o Insertar en una tabla los registros contenidos en otra tabla.

Para insertar un único Registró:

En este caso la sintaxis es la siguiente:

```
INSERT INTO Tabla (campo1.., campoN) VALUES (valor1, valor2, ..., valorN)
```

Esta consulta graba en el campo1 el valor1, en el campo2 y valor2 y así sucesivamente. Hay que prestar especial atención a acotar entre comillas simples (') los valores literales (cadenas de caracteres) y las fechas indicarlas en formato mm-dd-aa y entre caracteres de almohadillas (#).

Para insertar Registros de otra Tabla:

En este caso la sintaxis es:

```
INSERT INTO Tabla [IN base_externa] (campo1, campo2, ..., campoN)

SELECT TablaOrigen.campo1, TablaOrigen.campo2, ..., TablaOrigen.campoN
FROM TablaOrigen
```

En este caso se seleccionarán los campos 1,2, ..., n de la tabla origen y se grabarán en los campos 1,2,.., n de la Tabla. La condición SELECT puede incluir la cláusula WHERE para filtrar los registros a copiar. Si Tabla y TablaOrigen poseen la misma estructura se puede simplificar la sintaxis a:

```
INSERT INTO Tabla SELECT TablaOrigen.* FROM TablaOrigen
```

De esta forma los campos de TablaOrigen se grabarán en Tabla, para realizar esta operación es necesario que todos los campos de TablaOrigen estén contenidos con igual nombre en Tabla. Con otras palabras que Tabla posea todos los campos de TablaOrigen (igual nombre e igual tipo).

En este tipo de consulta hay que tener especial atención con los campos contadores o autonuméricos puesto que al insertar un valor en un campo de este tipo se escribe el

valor que contenga su campo homólogo en la tabla origen, no incrementándose como le corresponde.

Se puede utilizar la instrucción INSERT INTO para agregar un registro único a una tabla, utilizando la sintaxis de la consulta de adición de registro único tal y como se mostró anteriormente. En este caso, su código específica el nombre y el valor de cada campo del registro. Debe especificar cada uno de los campos del registro al que se le va a asignar un valor así como el valor para dicho campo. Cuando no se especifica dicho campo, se inserta el valor predeterminado o Null. Los registros se agregan al final de la tabla.

También se puede utilizar INSERT INTO para agregar un conjunto de registros pertenecientes a otra tabla o consulta utilizando la cláusula SELECT ... FROM como se mostró anteriormente en la sintaxis de la consulta de adición de múltiples registros. En este caso la cláusula SELECT especifica los campos que se van a agregar en la tabla destino especificada.

La tabla destino u origen puede especificar una tabla o una consulta.

Si la tabla destino contiene una clave principal, hay que asegurarse que es única, y con valores no-Null; si no es así, no se agregarán los registros. Si se agregan registros a una tabla con un campo Contador, no se debe incluir el campo Contador en la consulta. Se puede emplear la cláusula IN para agregar registros a una tabla en otra Base de Datos.

Se pueden averiguar los registros que se agregarán en la consulta ejecutando primero una consulta de selección que utilice el mismo criterio de selección y ver el resultado. Una consulta de adición copia los registros de una o más tablas en otra. Las tablas que contienen los registros que se van a agregar no se verán afectadas por la consulta de adición. En lugar de agregar registros existentes en otra tabla, se puede especificar los valores de cada campo en un nuevo registro utilizando la cláusula VALUES. Si se omite la lista de campos, la cláusula VALUES debe incluir un valor para cada campo de la tabla, de otra forma fallará INSERT.

```
INSERT INTO Clientes SELECT Clientes_Viejos.* FROM Clientes_Nuevos;

INSERT INTO Empleados (Nombre, Apellido, Cargo) VALUES ('Luis', 'Sánchez', 'Díaz');

INSERT INTO Empleados SELECT Vendedores.* FROM Vendedores WHERE Fecha_Contratacion < Now() - 30;
```

UPDATE

Crea una consulta de actualización que cambia los valores de los campos de una tabla especificada basándose en un criterio específico. Su sintaxis es:

```
UPDATE Tabla SET Campo1=Valor1, Campo2=Valor2, ... CampoN=ValorN
WHERE Criterio;
```

UPDATE es especialmente útil cuando se desea cambiar un gran número de registros o cuando éstos se encuentran en múltiples tablas. Puede cambiar varios campos a la vez. El ejemplo siguiente incrementa los valores Cantidad pedidos en un 10 por ciento y los valores Transporte en un 3 por ciento para aquellos que se hayan enviado a México:

```
UPDATE Pedidos SET Pedido = Pedidos * 1.1, Transporte = Transporte * 1.03
WHERE PaisEnvío = 'MX';
```

UPDATE no genera ningún resultado. Para saber qué registros se van a cambiar, hay que examinar primero el resultado de una consulta de selección que utilice el mismo criterio y después ejecutar la consulta de actualización.

```
UPDATE Empleados SET Grado = 5 WHERE Grado = 2;

UPDATE Productos SET Precio = Precio * 1.1 WHERE Proveedor = 8 AND
Familia = 3;
```

Si en una consulta de actualización se suprime la cláusula WHERE todos los registros de la tabla señalada serán actualizados.

```
UPDATE Empleados SET Salario = Salario * 1.1
```

1.4.7.5.-Consultas de Unión Internas

Las vinculaciones entre tablas se realizan mediante la cláusula INNER que combina registros de dos tablas siempre que haya concordancia de valores en un campo común. Su sintaxis es:

```
SELECT campos FROM tb1 INNER JOIN tb2 ON tb1.campo1 comp tb2.campo2
```

En **tbl1** y **tbl2** son los nombres de las tablas desde las que se combinan los registros.

Campo1 y **campo2** son los nombres de los campos que se combinan. Si no son numéricos, los campos deben ser del mismo tipo de datos y contener el mismo tipo de datos, pero no tienen que tener el mismo nombre.

Y comp es cualquier operador de comparación relacional: =, <, >, <=, >=, o <>.

Se puede utilizar una operación INNER JOIN en cualquier cláusula FROM. Esto crea una combinación por equivalencia, conocida también como unión interna. Las combinaciones aquí son las más comunes; éstas combinan los registros de dos tablas siempre que haya concordancia de valores en un campo común a ambas tablas. Se puede utilizar INNER JOIN con las tablas Departamentos y Empleados para seleccionar todos los empleados de cada departamento. Por el contrario, para seleccionar todos los departamentos (incluso si alguno de ellos no tiene ningún empleado asignado) se emplea LEFT JOIN o todos los empleados (incluso si alguno no está asignado a ningún departamento), en este caso RIGHT JOIN.

Si se intenta combinar campos que contengan datos Memo u Objeto OLE, se produce un error. Se pueden combinar dos campos numéricos cualesquiera, incluso si son de diferente tipo de datos. Por ejemplo, puede combinar un campo Numérico para el que la propiedad Size de su objeto Field está establecida como Entero, y un campo Contador.

El ejemplo siguiente muestra cómo podría combinar las tablas Categorías y Productos basándose en el campo IDCategoria:

```
SELECT Nombre_Categoría, NombreProducto

FROM Categorias INNER JOIN Productos

ON Categorias.IDCategoria = Productos.IDCategoria;
```

En el ejemplo anterior, IDCategoria es el campo combinado, pero no está incluido en la salida de la consulta ya que no está incluido en la instrucción SELECT. Para incluir el campo combinado, incluir el nombre del campo en la instrucción SELECT, en este caso, Categorias.IDCategoria.

También se pueden enlazar varias cláusulas ON en una instrucción JOIN, utilizando la sintaxis siguiente:

```
SELECT campos
FROM tabla1 INNER JOIN tabla2
ON tb1.campo1 comp tb2.campo1 AND
ON tb1.campo2 comp tb2.campo2) OR
ON tb1.campo3 comp tb2.campo3)];
```

También puede anidar instrucciones JOIN utilizando la siguiente sintaxis:

```
SELECT campos
FROM tb1 INNER JOIN
(tb2 INNER JOIN [( ]tb3
[INNER JOIN [( ]tablax [INNER JOIN ...)]
ON tb3.campo3 comp tbx.campox)]
ON tb2.campo2 comp tb3.campo3)
ON tb1.campo1 comp tb2.campo2;
```

Un LEFT JOIN o un RIGHT JOIN puede anidarse dentro de un INNER JOIN, pero un INNER JOIN no puede anidarse dentro de un LEFT JOIN o un RIGHT JOIN.

Si se utiliza la cláusula INNER en la consulta se seleccionarán sólo aquellos registros de la tabla de la que haya escrito a la izquierda de INNER JOIN que contengan al menos un registro de la tabla que haya escrito a la derecha. Para solucionar esto se tienen dos cláusulas que sustituyen a la palabra clave INNER, estas cláusulas son LEFT y RIGHT.

LEFT toma todos los registros de la tabla de la izquierda aunque no tengan ningún registro en la tabla de la izquierda. RIGHT realiza la misma operación pero al contrario, toma todos los registros de la tabla de la derecha aunque no tenga ningún registro en la tabla de la izquierda.

1.4.8. Microsoft Jet

El motor de base de datos Microsoft Jet proporciona un modelo relacional basado en el Lenguaje de Consulta Estructurado (SQL) estándar, con pocas diferencias existentes con respecto al lenguaje SQL ANSI, utilizándose fundamentalmente para ejecutar consultas de manipulación y definición de datos mediante los objetos correspondientes de las bibliotecas de DAO y ADO, bien para recuperar un conjunto de registros de la base de datos, bien para diseñar o modificar directamente la estructura de una base de datos Microsoft Jet (MONTEJO, 2004).

1.4.8.1.- Tipos de datos del motor Microsoft Jet

A la hora de crear una tabla, es importante definir el tipo de dato que utilizaran las columnas o campos para almacenar valores de la tabla, al fin de asegurarse que usan la misma cantidad posible de espacio de almacenamiento.

Los tipos de datos SQL del motor de base de datos Microsoft Jet se compone de una serie de datos primarios definidos por el propio motor, y de varios tipos de datos sinónimos reconocidos por aquellos.

El motor de Microsoft Jet contiene todos los componentes antes mencionados en el tema de SQL, las clausulas, comandos, etc.

1.5.-Ingeniería de Software

La ingeniería de software es una disciplina de la ingeniería cuya meta es el desarrollo costeable de software (SOMMERVILLE, 2005, pág. 6). Esta definición de Sommerville, describe un de los principales propósitos de la Ingeniera de Software que consiste mediante un buen análisis el desarrollo de aplicaciones con mejor planificación, para ahorrar costos y tiempo.

1.5.1.-Ciclo de Vida

Todo proyecto de ingeniería tiene unos fines ligados a la obtención de un producto, proceso o servicio que es necesario generar a través de diversas actividades. Algunas de estas actividades pueden agruparse en fases porque globalmente contribuyen a obtener un producto intermedio, necesario para continuar hacia el producto final y facilitar la gestión del proyecto (SOMMERVILLE, 2005, págs. 8-10). Al conjunto de las fases empleadas se le denomina "ciclo de vida".

La forma de agrupar las actividades, los objetivos de cada fase, los tipos de productos intermedios que se generan, etc. pueden ser muy diferentes dependiendo del tipo de producto o proceso a generar y de las tecnologías empleadas.

La complejidad de las relaciones entre las distintas actividades crece exponencialmente con el tamaño. De esta forma la división de los proyectos en fases sucesivas es un primer paso para la reducción de su complejidad, tratándose de escoger las partes de manera que sus relaciones entre sí sean lo más simples posibles (PRESUMAN, 2002).

La definición de un ciclo de vida facilita el control sobre los tiempos en que es necesario aplicar recursos de todo tipo (personal, equipos, suministros, etc.) al proyecto. Si el proyecto incluye subcontratación de partes a otras organizaciones, el control del trabajo subcontratado se facilita en la medida en que esas partes encajen bien en la estructura de las fases. El control de calidad también se ve facilitado si la separación entre fases se hace corresponder con puntos en los que ésta deba verificarse (mediante comprobaciones sobre los productos parciales obtenidos).

De la misma forma, la práctica acumulada en el diseño de modelos de ciclo de vida para situaciones muy diversas permite beneficiarse de la experiencia adquirida utilizando el enfoque que mejor de adapte a nuestros requerimientos.

1.5.1.1.-Elementos de un Ciclo de Vida

Un ciclo de vida para un proyecto se compone de fases sucesivas compuestas por tareas planificables. Según el modelo de ciclo de vida, la sucesión de fases puede ampliarse con bucles de realimentación (PRESUMAN, 2002), de manera que lo que conceptualmente se considera una misma fase se pueda ejecutar más de una vez a lo largo de un proyecto, recibiendo en cada pasada de ejecución aportaciones de los resultados intermedios que se van produciendo, ver figura 8.

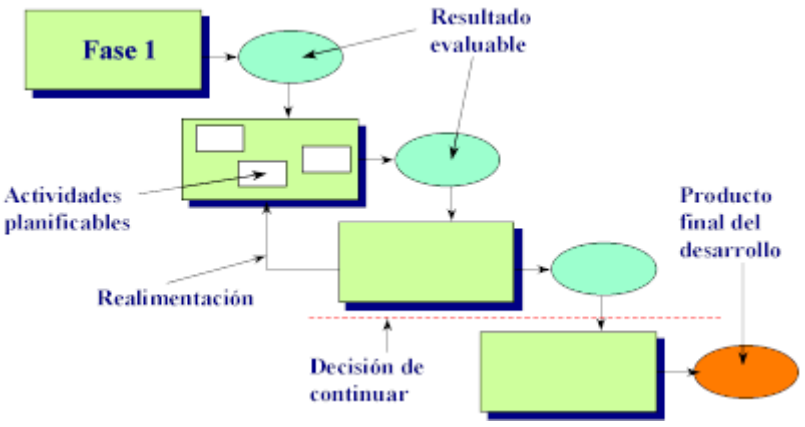

Figura 8: Elementos del Ciclo de Vida

Para un adecuado control de la progresión de las fases de un proyecto se hace necesario especificar con suficiente precisión los resultados evaluables, o sea, productos intermedios que deben resultar de las tareas incluidas en cada fase.

Aquí se muestran los distintos elementos que integran un ciclo de vida:

Fases. Una fase es un conjunto de actividades relacionadas con un objetivo en el desarrollo del proyecto, ver figura 9. Se construye agrupando tareas (actividades elementales) que pueden compartir un tramo determinado del tiempo de vida de un proyecto. La agrupación temporal de tareas impone requisitos temporales correspondientes a la asignación de recursos (humanos, financieros o materiales).

Cuanto más grande y complejo sea un proyecto, mayor detalle se necesitará en la definición de las fases para que el contenido de cada una siga siendo manejable. De esta forma, cada fase de un proyecto puede considerarse un "micro-proyecto" en sí mismo, compuesto por un conjunto de micro-fases.

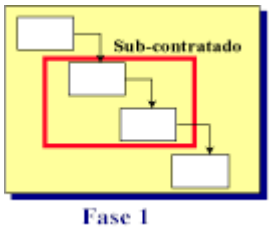

Figura 9: Fases

Otro motivo para descomponer una fase en subfases menores puede ser el interés de separar partes temporales del proyecto que se subcontraten a otras organizaciones, requiriendo distintos procesos de gestión.

Cada fase viene definida por un conjunto de elementos observables externamente, como son las actividades con las que se relaciona, los datos de entrada (resultados de la fase anterior, documentos o productos requeridos para la fase, experiencias de proyectos anteriores), los datos de salida (resultados a utilizar por la fase posterior, experiencia acumulada, pruebas o resultados efectuados) y la estructura interna de la fase, ver figura 10.

Figura 10: Esquema general de operación de una fase

Entregables ("deliverables"). Son los productos intermedios que generan las fases. Pueden ser materiales (componentes, equipos) o inmateriales (documentos, software). Los entregables permiten evaluar la marcha del proyecto mediante comprobaciones de su adecuación o no a los requisitos funcionales y de condiciones de realización previamente establecidos. Cada una de estas evaluaciones puede servir, además, para la toma de decisiones a lo largo del desarrollo del proyecto.

1.5.2.-Tipos de Modelos de un Ciclo de Vida

Las principales diferencias entre distintos modelos de ciclo de vida están en:

- El alcance del ciclo dependiendo de hasta dónde llegue el proyecto correspondiente. Un proyecto puede comprender un simple estudio de viabilidad del desarrollo de un producto, o su desarrollo completo o, llevando la cosa al extremo, toda la historia del producto con su desarrollo, fabricación, y modificaciones posteriores hasta su retirada del mercado.
- Las características (contenidos) de las fases en que dividen el ciclo. Esto puede depender del propio tema al que se refiere el proyecto (no son lo mismo las tareas que deben realizarse para proyectar un avión que un puente), o de la organización (interés de reflejar en la división en fases aspectos de la división interna o externa del trabajo).
- La estructura de la sucesión de las fases que puede ser lineal, con prototipado, o en espiral.

1.5.2.1.-Ciclo de Vida Lineal

Es el más utilizado, siempre que es posible, precisamente por ser el más sencillo. Consiste en descomponer la actividad global del proyecto en fases que se suceden de manera lineal, es decir, cada una se realiza una sola vez, cada una se realiza tras la anterior y antes que la siguiente. Con un ciclo lineal es fácil dividir las tareas entre equipos sucesivos, y prever los tiempos, ver figura 11.

Requiere que la actividad del proyecto pueda descomponerse de manera que una fase no necesite resultados de las siguientes, aunque pueden admitirse ciertos supuestos de realimentación correctiva. Desde el punto de vista de la gestión (para decisiones de planificación), requiere también que se sepa bien de antemano lo que va a ocurrir en cada fase antes de empezarla.

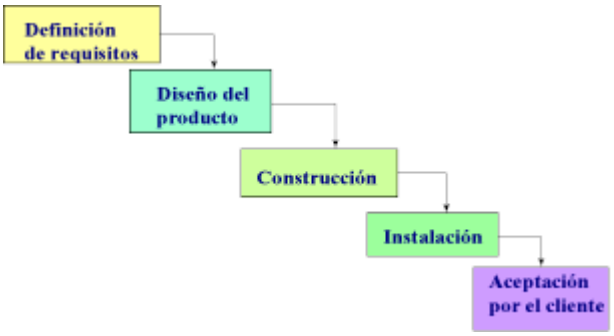

Figura 11: Ciclo lineal para un proyecto de construcción

1.5.2.2.-Ciclo de vida con Prototipo

A menudo ocurre en desarrollos de productos con innovaciones importantes, o cuando se prevé la utilización de tecnologías nuevas o poco probadas, que las incertidumbres sobre los resultados realmente alcanzables, o las ignorancias sobre el comportamiento de las tecnologías, impiden iniciar un proyecto lineal con especificaciones cerradas.

Si no se conoce exactamente cómo desarrollar un determinado producto o cuáles son las especificaciones de forma precisa, suele recurrirse a definir especificaciones iníciales para hacer un prototipo, o sea, un producto parcial (no hace falta que contenga funciones que se consideren triviales o suficientemente probadas) y provisional (no se va a fabricar realmente para clientes, por lo que tiene menos restricciones de coste y/o prestaciones). Este tipo de procedimiento es muy utilizado en desarrollo avanzado.

La experiencia del desarrollo del prototipo y su evaluación deben permitir la definición de las especificaciones más completas y seguras para el producto definitivo.

A diferencia del modelo lineal, puede decirse que el ciclo de vida con prototipo repite las fases de definición, diseño y construcción dos veces: para el prototipo y para el producto real (SOMMERVILLE, 2005, págs. 373-376).

Figura 12: Desarrollo de prototipo

1.5.2.3.-Ciclo de Vida en Espiral

El ciclo de vida en espiral puede considerarse como una generalización del anterior para los casos en que no basta con una sola evaluación de un prototipo para asegurar la desaparición de incertidumbres y/o ignorancias, ver figura 13. El propio producto a lo largo de su desarrollo puede así considerarse como una sucesión de prototipos que progresan hasta llegar a alcanzar el estado deseado (PRESUMAN, 2002). En cada ciclo (espirales) las especificaciones del producto se van resolviendo gradualmente.

A menudo la fuente de incertidumbres es el propio cliente, que aunque sepa en términos generales lo que quiere, no es capaz de definirlo en todos sus aspectos sin ver como unos influyen en otros. En estos casos la evaluación de los resultados por el cliente no puede esperar a la entrega final y puede ser necesaria repetidas veces.

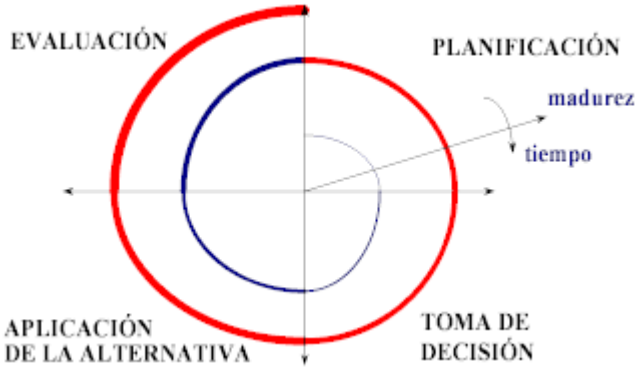

Figura 13: Bucle en espiral

El esquema del ciclo de vida para estos casos puede representarse por un bucle en espiral, donde los cuadrantes son, habitualmente, fases de especificación, diseño, realización y evaluación (o conceptos y términos análogos).

En cada vuelta el producto gana en "madurez" (aproximación al final deseado) hasta que en una vuelta la evaluación lo apruebe y el bucle pueda abandonarse.

1.5.3.-Modelos de Procesos de Software

"Un modelo de procesos del software es una descripción simplificada de un proceso del software que presenta una visión de ese proceso" (SOMMERVILLE, 2005, pág. 8). La mayor parte de los modelos de procesos de software se basan en uno de los tres modelos generales de desarrollo de software, que a continuación se describen.

1.5.3.1.-Modelo de Cascada

En Ingeniería de software el desarrollo en cascada, también llamado modelo en cascada, es el enfoque metodológico que ordena rigurosamente las etapas del ciclo de vida del software, de forma tal que el inicio de cada etapa debe esperar a la finalización de la inmediatamente anterior.

Un ejemplo de una metodología de desarrollo en cascada es:

1. Análisis de requisitos
2. Diseño
3. Codificación
4. Pruebas
5. Implantación
6. Mantenimiento

De esta forma, cualquier error de diseño detectado en la etapa de prueba conduce necesariamente al rediseño y nueva programación del código afectado, aumentando los costes del desarrollo. La palabra cascada sugiere, mediante la metáfora de la fuerza de la gravedad, el esfuerzo necesario para introducir un cambio en las fases más avanzadas de un proyecto.

Análisis de requisitos

Se analizan las necesidades de los usuarios finales del software para determinar qué objetivos debe cubrir. De esta fase surge una memoria llamada SRD (Documento de Especificación de Requisitos), que contiene la especificación completa de lo que debe hacer el sistema sin entrar en detalles internos.

Diseño

Se descompone y organiza el sistema en elementos que puedan elaborarse por separado, aprovechando las ventajas del desarrollo en equipo. Como resultado surge el SDD (Documento de Diseño del Software), que contiene la descripción de la estructura global del sistema y la especificación de lo que debe hacer cada una de sus partes, así como la manera en que se combinan unas con otras.

Codificación

Es la fase de programación propiamente dicha. Aquí se desarrolla el código fuente, haciendo uso de prototipos así como pruebas y ensayos para corregir errores.

Pruebas

Los elementos, ya programados, se ensamblan para componer el sistema y se comprueba que funciona correctamente antes de ser puesto en explotación.

Implantación

El software obtenido se pone en producción.

Mantenimiento

Durante la explotación del sistema software pueden surgir cambios, bien para corregir errores o bien para introducir mejoras. Todo ello se recoge en los Documentos de Cambios.

Variantes

Existen variantes de este modelo; especialmente se destaca la que hace uso de prototipos y en la que se establece un ciclo antes de llegar a la fase de mantenimiento, verificando que el sistema final esté libre de fallos.

1.5.3.2.-Modelo de Riesgo y Espiral

Este modelo fue desarrollado por B. Boehm, la idea es Desarrollo Evolutivo, usando el Modelo de Cascada para cada etapa; está orientado a evitar riesgos de trabajo. No define

en detalle el sistema completo a la primera. Se tendrían que definir solamente las más altas prioridades. Definir e implementarlas y entonces obtener un feedback(retroalimentación) de los usuarios (tal y como feedback distingue desarrollo "evolutivo" de "incremental"). Con este conocimiento, deberían entonces retroceder o volver al punto de partida para definir e implementar más y mejores partes.

El Modelo Espiral mejora el Modelo de Cascada denotando la naturaleza iterativa del proceso de diseño. Eso introduce un ciclo de prototipo iterativo. En cada iteración, las nuevas expresiones que son obtenidas transformando otras dadas son examinadas para ver si representan progresos hacia el objetivo (PRESUMAN, 2002).

Este método está basado en dos importantes principios:

1. La práctica de diseñó profesional es caracterizar en términos de conocer, actuar en situaciones, conversación con la situación y reflexión en acción. Hay un distinto medio de proceso - orientación en esta aproximación al diseño. Es raro que el diseñador tenga el diseño en su cabeza por adelantado y que después meramente lo transcriba. Gran parte del tiempo del diseñador esta inmiscuido en una progresiva relación con su entorno
2. La necesidad para diseñadores de tomar la práctica de trabajo seriamente, de supervisar las formas en las que el trabajo se está haciendo, en el sentido de una solución abierta y desplegada para aumentar la complejidad de una situación que el diseñador solo entiende parcialmente. El hecho por el cual se está tratando con "actores humanos". Los sistemas necesitan tratar o estar en contacto con las preocupaciones del usuario.

Una visión general del Modelo de Cascada puede ser la siguiente, ver figura 14:

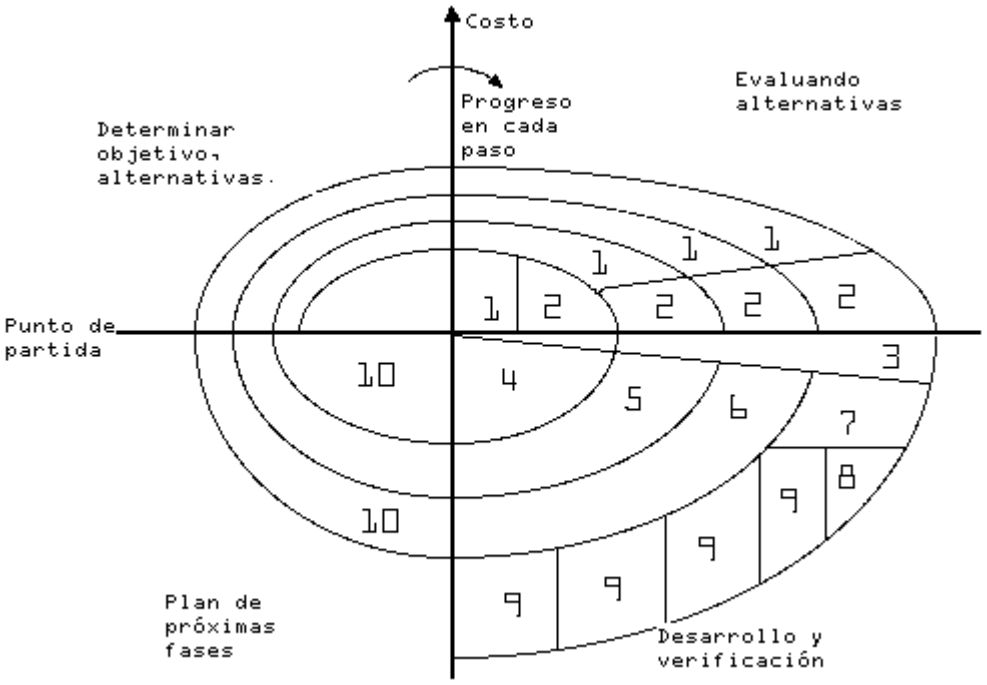

Figura 14: Proceso en Espiral

1.5.4.-Fundamentos del Análisis de Requerimientos

Es la etapa más crucial del desarrollo de un proyecto de software.

La IEEE los divide en funcionales y no funcionales:

- **Funcionales:** Condición o capacidad de un sistema requerida por el usuario para resolver un problema o alcanzar un objetivo (SOMMERVILLE, 2005, pág. 110).
- **No Funcionales:** Condición o capacidad que debe poseer un sistema para satisfacer un contrato, un estándar, una especificación u otro documento formalmente impuesto (SOMMERVILLE, 2005, pág. 112).

Para realizar bien el desarrollo de software es necesario realizar una especificación completa de los requerimientos de los mismos. Independientemente de lo bien diseñado o codificado que esté, un programa pobremente especificado decepcionará al usuario y hará fracasar el desarrollo.

El papel del análisis de los requerimientos es un proceso de descubrimiento y refinamiento, El ámbito del programa, establecido inicialmente durante la ingeniería del

sistema, es refinado en detalle. Se analizan y asignan a los distintos elementos de los programas las soluciones alternativas.

Tanto el que desarrolla el software como el cliente tienen un papel activo en la especificación de requerimientos. El cliente intenta reformular su concepto, algo nebuloso, de la función y comportamiento de los programas en detalles concretos, El que desarrolla el software actúa como interrogador, consultor y el que resuelve los problemas.

El análisis y especificación de requerimientos puede parecer una tarea relativamente sencilla, pero las apariencias engañan. Puesto que el contenido de comunicación es muy alto, abundan los cambios por mala interpretación o falta de información.

1.5.4.1.-Análisis de Requerimientos

El análisis de requerimientos es la actividad que plantea la asignación de software a nivel de sistema y el diseño de programas. El análisis de requerimientos facilita al ingeniero de sistemas especificar la función y comportamiento de los programas, indicar la interfaz con otros elementos del sistema y establecer las ligaduras de diseño que debe cumplir el programa (PRESUMAN, 2002). El análisis de requerimientos permite al ingeniero refinar la asignación de software y representar el dominio de la información que será tratada por el programa. El análisis de requerimientos de al diseñador la representación de la información y las funciones que pueden ser traducidas en datos, arquitectura y diseño procedimental.

La especificación de requerimientos suministra al técnico y al cliente, los medios para valorar la calidad de los programas, una vez que se haya construido.

1.5.4.2.-Tareas del Análisis

El análisis de requerimientos puede dividirse en cuatro áreas:

1. Reconocimiento del problema
2. Evaluación y síntesis
3. Especificación
4. Revisión.

Inicialmente, el analista estudia la especificación del sistema y el plan de proyecto. Es importante comprender el contexto del sistema y revisar el ámbito de los programas que

se usó para generar las estimaciones de la planificación. Debe establecerse la comunicación necesaria para el análisis, de forma que se asegure el reconocimiento del problema.

El analista debe establecer contacto con el equipo técnico y de gestión del usuario/cliente y con la empresa que vaya a desarrollar el software. El gestor del programa puede servir como coordinador para facilitar el establecimiento de los caminos de comunicación. El objetivo del analista es reconocer los elementos básicos del programa tal como lo percibe el usuario/cliente (SOMMERVILLE, 2005, pág. 116).

La evaluación del problema y la síntesis de la solución es la siguiente área principal de trabajo del análisis. El analista debe evaluar el flujo y estructura de la información, refinar en detalle todas las funciones del programa, establecer las características de la interface del sistema y descubrir las ligaduras del diseño, Cada una de las tareas sirve para descubrir el problema de forma que pueda sintetizarse un enfoque o solución global.

Las tareas asociadas con el análisis y especificación existen para dar una representación del programa que pueda ser revisada y aprobada por el cliente. En un mundo ideal el cliente desarrolla una especificación de requerimientos del software completamente por sí mismo. Esto se presenta raramente en el mundo real. En el mejor de los casos, la especificación se desarrolla conjuntamente entre el cliente y el técnico.

Una vez que se hayan descrito las funcionalidades básicas, comportamiento, interface e información, se especifican los criterios de validación para demostrar una comprensión de una correcta implementación de los programas. Estos criterios sirven como base para hacer una prueba durante el desarrollo de los programas. Para definir las características y atributos del software se escribe una especificación de requerimientos formal. Además, para los casos en los que se desarrolle un prototipo se realiza un manual de usuario preliminar. (SOMMERVILLE, 2005, págs. 117-118)

Puede parecer innecesario realizar un manual de usuario en una etapa tan temprana del proceso de desarrollo, Pero de hecho, este borrador del manual de usuario fuerza al analista a tomar el punto de vista del usuario del software. El manual permite al usuario / cliente revisar el software desde una perspectiva de ingeniería humana.

Los documentos del análisis de requerimiento (especificación y manual de usuario) sirven como base para una revisión conducida por el cliente y el técnico. La revisión de los requerimientos casi siempre produce modificaciones en la función, comportamiento, representación de la información, ligaduras o criterios de validación. Además, se realiza

una nueva apreciación del plan del proyecto de software para determinar si las primeras estimaciones siguen siendo validas después del conocimiento adicional obtenido durante el análisis.

1.5.4.3.-Principios del Análisis

En la pasada década, se desarrollaron varios métodos de análisis y especificación del software. Cada método de análisis tiene una única notación y punto de vista. Sin embargo, todos los métodos de análisis están relacionados por un conjunto de principios fundamentales:

El dominio de la información, así como el dominio funcional de un problema debe ser representado y comprendido.

El problema debe subdividirse de forma que se descubran los detalles de una manera progresiva (o jerárquica).Deben desarrollarse las representaciones lógicas y físicas del sistema.

Aplicando estos principios, el analista enfoca el problema sistemáticamente. Se examina el dominio de la información de forma que pueda comprenderse su función más completamente. La partición se aplica para reducir la complejidad. La visión lógica y física del software, es necesaria para acomodar las ligaduras lógicas impuestas por los requerimientos de procesamiento, y las ligaduras físicas impuestas por otros elementos del sistema.

1.5.5. Partición

Normalmente los problemas son demasiado grandes y complejos para ser comprendidos como un todo.

Por esta razón, se tiende a particionar (dividir) tales problemas en partes que puedan ser fácilmente comprendidas, y establecer interfaces entre las partes, de forma que se realice la función global.

Durante el análisis de requerimientos, el dominio funcional y el dominio de la información del software pueden ser particionados. Y se una representación jerárquica de la función o información y luego se parte del elemento superior mediante:

1. Incrementando los detalles, moviéndonos verticalmente en la jerarquía

2. Descomponiendo funcionalmente el problema, moviéndonos horizontalmente en la jerarquía.

1.5.6.-Visiones Lógicas y Físicas

La visión lógica de los requerimientos del software presenta las funciones que han de realizarse y la información que ha de procesarse independientemente de los detalles de implementación.

La visión física de los requerimientos del software presenta una manifestación del mundo real de las funciones de procesamiento y las estructuras de información. En algunos casos se desarrolla una representación física como el primer paso del diseño del software. Sin embargo la mayoría de los sistemas basados en computador, se especifican de forma que se dictan ciertas recomendaciones físicas.

1.5.7.-Construcción de Prototipos de Software

En análisis debe ser conducido independientemente del paradigma de ingeniería de software aplicado. Sin embargo, la forma que ese análisis tomara puede variar. En algunos casos es posible aplicar los principios de análisis fundamental y derivar a una especificación en papel del software desde el cual pueda desarrollarse un diseño. En otras situaciones, se va a una recolección de los requerimientos, se aplican los principios de análisis y se construye un modelo de software, llamado un prototipo, según las apreciaciones del cliente y del que lo desarrolla. Hay circunstancias que requieren la construcción de un prototipo al comienzo del análisis, puesto que el modelo es el único mediante el que los requerimientos pueden ser derivados efectivamente.

1.5.8.-Especificación

No hay duda de que la forma de especificar tiene mucho que ver con la calidad de la solución. Los ingenieros de software que se han esforzado en trabajar con especificaciones incompletas, inconsistentes o mal establecidas han experimentado la frustración y confusión que invariablemente se produce. Las consecuencias se padecen en la calidad, oportunidad y completitud del software resultante.

Se ha visto que los requerimientos de software pueden ser analizados de varias formas diferentes. Las técnicas de análisis pueden conducir a una especificación en papel que contenga las descripciones graficas y el lenguaje natural de los requerimientos del

software. La construcción de prototipos conduce a una especificación ejecutable, esto es, el prototipo sirve como una representación de los requerimientos. Los lenguajes de especificación formal conducen a representaciones formales de los requerimientos que pueden ser verificados o analizados.

1.5.9.-Métodos de Análisis de Requerimientos

Las metodologías de análisis de requerimientos combinan procedimientos sistemáticos con una notación única para analizar los dominios de información y funcional de un problema de software; suministra un conjunto de heurísticas para subdividir el problema y define una forma de representación para las visiones lógicas y físicas. En esencia, los métodos de análisis de requerimientos del software, facilitan al ingeniero de software aplicar principios de análisis fundamentales, dentro del contexto de un método bien definido (PRESUMAN, 2002).

La mayoría de los métodos de análisis de requerimientos son conducidos por la información. Estos es, el método suministra un mecanismo para representar el dominio de la información. Desde esta representación, se deriva la función y se desarrollan otras características de los programas.

El papel de los métodos de análisis de requerimientos, es asistir al analista en la construcción de “una descripción precisa e independiente” del elemento software de un sistema basado en computadora.

1.5.10.-Metodologías de Análisis de Requerimientos

Las metodologías de análisis de requerimientos facilitan al analista la aplicación de los principios fundamentales del análisis de una manera sistemática (PRESUMAN, 2002).

Aunque cada método introduce nueva notación y heurística de análisis, todos los métodos pueden ser evaluados en el contexto de las siguientes características comunes:

1. Mecanismos para el análisis del dominio de la información
2. Método de representación funcional
3. Definición de interfaces
4. Mecanismos para subdividir el problema
5. Soporte de la abstracción
6. Representación de visiones físicas y lógicas

Aunque el análisis del dominio de la información se conduce de forma diferente en cada metodología, pueden reconocerse algunas guías comunes. Todos los métodos se enfocan (directa o indirectamente) al flujo de datos y al contenido o estructura de datos. En la mayoría de los casos el flujo se caracteriza en el contexto de las transformaciones (funciones) que se aplican para cambiar la entrada en la salida.

El contenido de los datos puede representarse explícitamente usando un mecanismo de diccionario o, implícitamente, enfocando primero la estructura jerárquica de los datos.

Las funciones se describen normalmente como transformaciones o procesos de la información. Cada función puede ser representada usando una notación específica. Una descripción de la función puede desarrollarse usando el lenguaje natural, un leguaje procedimental con reglas sintácticas informales o un lenguaje de especificación forma.

1.5.11.-Diagramas de Flujos de Datos

Conforme con la información se mueve a través del software, se modifica mediante una serie de transformaciones. Un diagrama de flujos de datos (DFD), es una técnica grafica que describe el flujo de información y las transformaciones que se aplican a los datos, conforme se mueven de la entrada a la salida.

1.5.12.-Diccionario de Datos

Se ha propuesto el diccionario de datos como una gramática casi formal para describir el contenido de los elementos de información.

El diccionario de datos contiene las definiciones de todos los datos mencionados en el DFD, en una especificación del proceso y en el propio diccionario de datos. Los datos compuestos (datos que pueden ser además divididos) se definen en términos de sus componentes; los datos elementales (datos que no pueden ser divididos) se definen en términos del significado de cada uno de los valores que puede asumir. Por tanto, el diccionario de datos está compuesto de definiciones de flujo de datos, archivos (datos almacenados) y datos usados en los procesos.

1.5.13.-Descripciones Funcionales

Una vez que ha sido representado el dominio de la información (usando un DFD y un diccionario de datos), el analista describe cada función (transformación) representada, usando el lenguaje natural o alguna otra notación. Una de tales notaciones se llama

ingles estructurado (también llamado lenguaje de diseño del programa o proceso (LDP)). Incorpora construcciones procedimentales básicas –secuencia, selección y repetición- junto con frases del lenguaje natural, de forma que pueden desarrollarse descripciones procedimentales precisas de las funciones representadas dentro de un DFD.

1.5.14.-Métodos Orientados a la Estructura de Datos

Los métodos de análisis orientados a la estructura de datos representan los requerimientos del software enfocándose hacia la estructura de datos en vez de al flujo de datos. Aunque cada método orientado a la estructura de datos tiene un enfoque y notación distinta, todos tienen algunas características en común:

- Asisten al analista en la identificación de los objetos de información clave (también llamados entidades o ítems) y operaciones (también llamadas acciones o procesos).
- Suponen que la estructura de la información es jerárquica;
- Requiere que la estructura de datos se represente usando la secuencia, selección y repetición.
- Dan un conjunto de pasos para transformar una estructura de datos jerárquica en una estructura de programa.

Como los métodos orientados al flujo de datos, los métodos de análisis orientados a la estructura de datos proporcionan la base para el diseño de software. Siempre puede extenderse un método de análisis para que abarque el diseño arquitectural y procedimental del software.

1.6.-Proceso Unificado de Desarrollo de Software

El Proceso Unificado como metodología es un proceso de software genérico que puede ser utilizado para una gran cantidad de tipos de sistemas de software, para diferentes áreas de aplicación, diferentes tipos de organizaciones, diferentes niveles de competencia y diferentes tamaños de proyectos (JACOBSON, BOOCH, & RUMBAUG, 2000, pág. 4).

Es parte de un esquema para la asignación de tareas y responsabilidades dentro de una organización de desarrollo. Su principal objetivo es asegurar la producción de software de muy alta calidad que satisfaga las necesidades de los usuarios finales, dentro de un calendario y presupuesto predecible.

El Proceso Unificado tiene dos dimensiones:

- Un eje horizontal que representa el tiempo y muestra los aspectos del ciclo de vida del proceso a lo largo de su desenvolvimiento
- Un eje vertical que representa las disciplinas, las cuales agrupan actividades de una manera lógica de acuerdo a su naturaleza.

La primera dimensión representa el aspecto dinámico del proceso conforme se va desarrollando, se expresa en términos de fases, iteraciones e hitos.

La segunda dimensión representa el aspecto estático del proceso: cómo es descrito en términos de componentes del proceso, disciplinas, actividades, flujos de trabajo, artefactos y roles.

El Proceso Unificado se basa en componentes (component-based), lo que significa que el sistema en construcción está hecho de componentes de software interconectados por medio de interfaces bien definidas (well-defined interfaces).

El Proceso Unificado usa el Lenguaje de Modelado Unificado (UML) en la preparación de todos los planos del sistema. De hecho, UML es una parte integral del Proceso Unificado, fueron desarrollados a la par (JACOBSON, BOOCH, & RUMBAUG, 2000, pág. 4).

Los aspectos distintivos del Proceso Unificado están capturados en tres conceptos clave: dirigido por casos de uso (use-case driven), centrado en la arquitectura (architecture-centric), iterativo e incremental. Esto es lo que hace único al Proceso Unificado.

1.6.1.-El Proceso Unificado es dirigido por casos de uso

Los sistemas computacionales o de software se crean para servir a sus usuarios. Por lo tanto, para construir un sistema exitoso se debe conocer qué es lo que quieren y necesitan los usuarios prospectos.

Cuando se habla de un usuario no solamente se refiere a los usuarios humanos, sino a otros sistemas. En este contexto, el término usuario representa algo o alguien que interactúa con el sistema por desarrollar.

Un caso de uso es un elemento importante en la funcionalidad del sistema que le da al usuario un resultado de valor. Los casos de uso capturan los requerimientos funcionales. Todos los casos de uso juntos constituyen el modelo de casos de uso el cual describe la funcionalidad completa del sistema. Este modelo reemplaza la tradicional especificación funcional del sistema. Los casos de uso no son solamente una herramienta para especificar los requerimientos del sistema, también dirigen su diseño, implementación y pruebas, esto es, dirigen el proceso de desarrollo (JACOBSON, BOOCH, & RUMBAUG, 2000, pág. 5).

Los casos de uso no son elegidos de manera aislada aun a pesar de que se encargan de dirigir procesos. Son desarrollados a la par con la arquitectura del sistema, esto es, los casos de uso dirigen la arquitectura del sistema y la arquitectura del sistema influencia la elección de los casos de uso. Por lo tanto, la arquitectura del sistema y los casos de uso maduran conforme avanza el ciclo de vida.

1.6.2.- El Proceso Unificado está centrado en la arquitectura

El papel del arquitecto de sistemas es similar en naturaleza al papel que el arquitecto desempeña en la construcción de edificios. El edificio se mira desde diferentes puntos de vista: estructura, servicios, plomería, electricidad, etc. Esto le permite al constructor ver una radiografía completa antes de empezar a construir. Similarmente, la arquitectura en un sistema de software es descrita como diferentes vistas del sistema que está siendo construido.

El concepto de arquitectura de software involucra los aspectos estáticos y dinámicos más significativos del sistema. La arquitectura surge de las necesidades de la empresa, tal y como las interpretan los usuarios y como están reflejadas en los casos de uso. Sin embargo, también está influenciada por muchos otros factores, tales como la plataforma de software en la que se ejecutará, la disponibilidad de componentes reutilizables,

consideraciones de instalación, sistemas legados, requerimientos no funcionales (ej. desempeño, confiabilidad). La arquitectura es la vista del diseño completo con las características más importantes hechas más visibles y dejando los detalles de lado. Ya que lo importante depende en parte del criterio, el cual a su vez viene con la experiencia, el valor de la arquitectura depende del personal asignado a esta tarea. Sin embargo, el proceso ayuda al arquitecto a enfocarse en las metas correctas, tales como claridad (understandability), flexibilidad en los cambios futuros (resilience) y reuso (JACOBSON, BOOCH, & RUMBAUG, 2000, págs. 5-6).

Los casos de uso y arquitectura tienen función y forma. Uno sólo de los dos no es suficiente. Estas dos fuerzas deben estar balanceadas para obtener un producto exitoso. En este caso función corresponde a los casos de uso y forma a la arquitectura. Existe la necesidad de intercalar entre casos de uso y arquitectura. Por una parte, los casos de uso deben, cuando son realizados, acomodarse en la arquitectura. Por otra parte, la arquitectura debe proveer espacio para la realización de todos los casos de uso, hoy y en el futuro. En la realidad, ambos arquitectura y casos de uso deben evolucionar en paralelo.

1.6.3.-El Proceso Unificado es Iterativo e Incremental

Para elaborar un software de manera comercial es una tarea enorme que puede continuar por varios meses o años. Es práctico dividir el trabajo en pequeños pedazos o mini-proyectos. Cada mini-proyecto es una iteración que finaliza en un incremento. Las iteraciones se refieren a pasos en el flujo de trabajo, los incrementos se refieren a crecimiento en el producto. Para ser más efectivo, las iteraciones deben estar controladas, esto es, deben ser seleccionadas y llevadas a cabo de una manera planeada (JACOBSON, BOOCH, & RUMBAUG, 2000, págs. 6-7).

Los desarrolladores basan su selección de qué van a implementar en una iteración en dos factores. Primero, la iteración trata con un grupo de casos de uso que en conjunto extienden la usabilidad del producto. Segundo, la iteración trata con los riesgos más importantes. Las iteraciones sucesivas construyen los artefactos del desarrollo a partir del estado en el que fueron dejados en la iteración anterior.

En cada iteración, los desarrolladores identifican y especifican los casos de uso relevantes, crean el diseño usando la arquitectura como guía, implementan el diseño en componentes y verifican que los componentes satisfacen los casos de uso. Si una iteración cumple sus metas el desarrollo continúa con la siguiente iteración. Cuando la

iteración no cumple con sus metas, los desarrolladores deben revisar sus decisiones previas y probar un nuevo enfoque.

1.7.-UML (Lenguaje de Modelado Unificado)

UML (Unified Modeling Language) es un lenguaje estándar que sirve para escribir los planos del software, puede utilizarse para visualizar, especificar, construir y documentar todos los artefactos que componen un sistema con gran cantidad de software. UML puede usarse para modelar desde sistemas de información hasta aplicaciones distribuidas basadas en Web, pasando por sistemas empotrados de tiempo real. UML es solamente un lenguaje por lo que es sólo una parte de un método de desarrollo software, es independiente del proceso aunque para que sea optimo debe usarse en un proceso dirigido por casos de uso, centrado en la arquitectura, iterativo e incremental (SCHMULLER, 2003, págs. 5-7).

UML es un lenguaje por que proporciona un vocabulario y las reglas para utilizarlo, además es un lenguaje de modelado lo que significa que el vocabulario y las reglas se utilizan para la representación conceptual y física del sistema.

UML es un lenguaje que nos ayuda a interpretar grandes sistemas mediante gráficos o mediante texto obteniendo modelos explícitos que ayudan a la comunicación durante el desarrollo ya que al ser estándar, los modelos podrán ser interpretados por personas que no participaron en su diseño (e incluso por herramientas) sin ninguna ambigüedad. En este contexto, UML sirve para especificar, modelos concretos, no ambiguos y completos.

Debido a su estandarización aunque no sea un lenguaje de programación, UML se puede conectar de manera directa a lenguajes de programación como Java, C++ o Visual Basic, esta correspondencia permite lo que se denomina como ingeniería directa (obtener el código fuente partiendo de los modelos) pero además es posible reconstruir un modelo en UML partiendo de la implementación, o sea, la ingeniería inversa.

UML proporciona la capacidad de modelar actividades de planificación de proyectos y de sus versiones, expresar requisitos y las pruebas sobre el sistema, representar todos sus detalles así como la propia arquitectura. Mediante estas capacidades se obtiene una documentación que es válida durante todo el ciclo de vida de un proyecto.

El lenguaje UML se compone de tres elementos básicos, los bloques de construcción, las reglas y algunos mecanismos comunes.

Los bloques de construcción se dividen en tres partes: Elementos, que son las abstracciones de primer nivel, Relaciones, que unen a los elementos entre sí, y los Diagramas, que son agrupaciones interesantes de elementos.

Existen cuatro tipos de elementos en UML, dependiendo del uso que se haga de ellos: elementos estructurales, elementos de comportamiento, elementos de agrupación y elementos de anotación.

Las relaciones, a su vez se dividen para abarcar las posibles interacciones entre elementos que se nos pueden presentar a la hora de modelar usando UML, estas son: relaciones de dependencia, relaciones de asociación, relaciones de generalización y relaciones de realización.

Se utilizan diferentes diagramas dependiendo de qué, nos interese representar en cada momento, para dar diferentes perspectivas de un mismo problema, para ajustar el nivel de detalle, por esta razón. UML soporta un gran número de diagramas diferentes aunque, en la práctica, sólo se utilicen un pequeño número de combinaciones.

1.7.1.-Bloques de construcción de UML

A continuación se describen los elementos que componen los bloques estructurales de UML, así como su notación y se vaya generando un esquema conceptual sobre UML.

1.7.1.1.-Elementos Estructurales

Los elementos estructurales en UML, es su mayoría, son las partes estáticas del modelo y representan cosas que son conceptuales o materiales.

Clases

Una clase es una descripción de un conjunto de objetos que comparten los mismos atributos, operaciones, relaciones y semántica. Una clase implementa una o más interfaces. Gráficamente se representa como un rectángulo que incluye su nombre, sus atributos y sus operaciones, ver figura 15 (SCHMULLER, 2003, pág. 8).

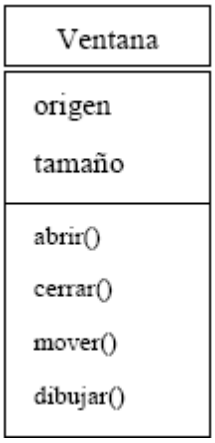

Figura 15: Clase

Interfaz

Una interfaz es una colección de operaciones que especifican un servicio de una determinada clase o componente. Una interfaz describe el comportamiento visible externamente de ese elemento, puede mostrar el comportamiento completo o sólo una parte del mismo (SCHMULLER, 2003, pág. 61). Una interfaz describe un conjunto de especificaciones de operaciones (o sea su signatura) pero nunca su implementación. Se representa con un círculo, como se puede ver en la figura 16, y rara vez se encuentra aislada sino que más bien conectada a la clase o componente que realiza.

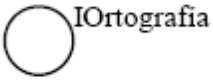

Figura 16: Interfaz

Colaboración

Define una interacción y es una sociedad de roles y otros elementos que colaboran para proporcionar un comportamiento cooperativo mayor que la suma de los comportamientos de sus elementos. Las colaboraciones tienen una dimensión tanto estructural como de comportamiento (LEYTON G., 2002). Una misma clase puede participar en diferentes colaboraciones. Las colaboraciones representan la implementación de patrones que forman un sistema. Se representa mediante una elipse con borde discontinuo, ver figura 17.

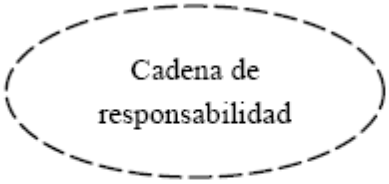

Figura 17: Colaboración

Casos de Uso

Un caso de uso es la descripción de un conjunto de acciones que un sistema ejecuta y que produce un determinado resultado que es de interés para un actor particular. Un caso de uso se utiliza para organizar los aspectos del comportamiento en un modelo (LEYTON G., 2002). Un caso de uso es realizado por una colaboración. Se representa como en la figura 18, una elipse con borde continuo.

Figura 18: Casos de Uso

Componentes

Un componente es una parte física y reemplazable de un sistema que conforma con un conjunto de interfaces y proporciona la implementación de dicho conjunto. Un componente representa típicamente el empaquetamiento físico de diferentes elementos lógicos, como clases, interfaces y colaboraciones.

Nodos

Un nodo es un elemento físico que existe en tiempo de ejecución y representa un recurso computacional que, por lo general, dispone de algo de memoria y, con frecuencia, de capacidad de procesamiento. Un conjunto de componentes puede residir en un nodo.

Estos siete elementos vistos son los elementos estructurales básicos que se pueden incluir en un modelo UML. Existen variaciones sobre estos elementos básicos, tales como actores, señales, utilidades (tipos de clases), procesos e hilos (tipos de clases

activas) y aplicaciones, documentos, archivos, bibliotecas, páginas y tablas (tipos de componentes).

1.7.1.2.-Elementos de comportamiento

Los elementos de comportamiento son las partes dinámicas de un modelo. Se podría decir que son los verbos de un modelo y representan el comportamiento en el tiempo y en el espacio. Los principales elementos son los dos que siguen.

Interacción

Es un comportamiento que comprende un conjunto de mensajes intercambiados entre un conjunto de objetos, dentro de un contexto particular para conseguir un propósito específico. Una interacción involucra otros muchos elementos, incluyendo mensajes, secuencias de acción (comportamiento invocado por un objeto) y enlaces (conexiones entre objetos) (LEYTON G., 2002). La representación de un mensaje es una flecha dirigida que normalmente con el nombre de la operación, ver figura 19.

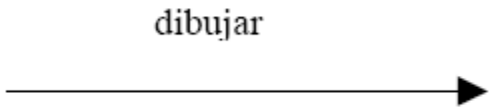

Figura 19: Interacción

Maquinas de estados

Es un comportamiento que especifica las secuencias de estados por las que van pasando los objetos o las interacciones durante su vida en respuesta a eventos, junto con las respuestas a esos eventos. Una maquina de estados involucra otros elementos como son estados, transiciones (flujo de un estado a otro), eventos (que disparan una transición) y actividades (respuesta de una transición), ver figura 20.

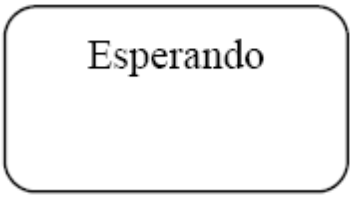

Figura 20: Estados

1.7.1.3.-Elementos de agrupación

Forman la parte organizativa de los modelos UML. El principal elemento de agrupación es el paquete, que es un mecanismo de propósito general para organizar elementos en grupos. Los elementos estructurales, los elementos de comportamiento, incluso los propios elementos de agrupación se pueden incluir en un paquete.

Un paquete es puramente conceptual (sólo existe en tiempo de desarrollo). Gráficamente se representa como una carpeta conteniendo normalmente su nombre y, a veces, su contenido, ver figura 21.

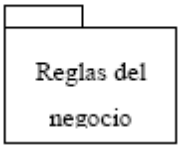

Figura 21: Elementos de agrupación

1.7.1.4.-Elementos de anotación

Los elementos de anotación son las partes explicativas de los modelos UML. Son comentarios que se pueden aplicar para describir, clasificar y hacer observaciones sobre cualquier elemento de un modelo.

El tipo principal de anotación es la nota que simplemente es un símbolo para mostrar restricciones y comentarios junto a un elemento o un conjunto de elementos, ver figura 22.

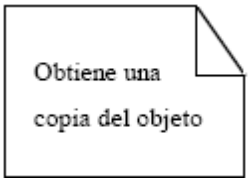

Figura 22: Elementos de anotación

1.7.2.-Relaciones

Existen cuatro tipos de relaciones entre los elementos de un modelo UML. Dependencia, asociación, generalización y realización, estas se describen a continuación:

1.7.2.1.-Dependencia

Es una relación semántica entre dos elementos en la cual un cambio a un elemento (el elemento independiente) puede afectar a la semántica del otro elemento (elemento dependiente). Se representa como una línea discontinua, ver figura 23, posiblemente dirigida, que a veces incluye una etiqueta (SCHMULLER, 2003, pág. 54).

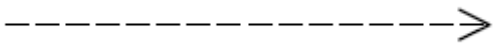

Figura 23: Dependencia

1.7.2.2.-Asociación

Es una relación estructural que describe un conjunto de enlaces, los cuales son conexiones entre objetos. La agregación es un tipo especial de asociación y representa una relación estructural entre un todo y sus partes (SCHMULLER, 2003, pág. 46). La asociación se representa con una línea continua, posiblemente dirigida, que a veces incluye una etiqueta. A menudo se incluyen otros adornos para indicar la multiplicidad y roles de los objetos involucrados, como se puede ver en la figura 24.

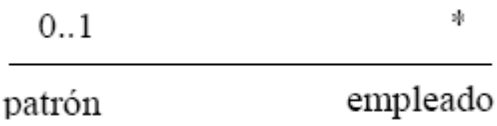

Figura 24: Asociación

1.7.2.3.-Generalización

Es una relación de especialización / generalización en la cual los objetos del elemento especializado (el hijo) pueden sustituir a los objetos del elemento general (el padre). De esta forma, el hijo comparte la estructura y el comportamiento del padre. Gráficamente, la generalización se representa con una línea con punta de flecha vacía, ver figura 25.

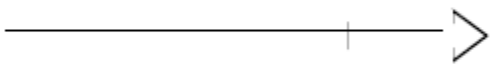

Figura 25: Generalización

1.7.2.4.-Realización

Es una relación semántica entre clasificadores, donde un clasificador especifica un contrato que otro clasificador garantiza que cumplirá. Se pueden encontrar relaciones de realización en dos sitios: entre interfaces y las clases y componentes que las realizan, y entre los casos de uso y las colaboraciones que los realizan (SCHMULLER, 2003). La realización se representa como una mezcla entre la generalización (figura 25) y la dependencia (figura 23), esto es, una línea discontinua con una punta de flecha vacía, ver figura 26.

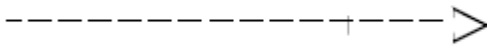

Figura 26: Realización

1.7.3.-Diagramas

Los diagramas se utilizan para representar diferentes perspectivas de un sistema de forma que un diagrama es una proyección del mismo. UML proporciona un amplio conjunto de diagramas que normalmente se usan en pequeños subconjuntos para poder representar las cinco vistas principales de la arquitectura de un sistema.

1.7.3.1.-Diagramas de Clases

Muestran un conjunto de clases, interfaces y colaboraciones, así como sus relaciones, ver figura 27. Estos diagramas son los más comunes en el modelado de sistemas orientados a objetos y cubren la vista de diseño estática o la vista de procesos estática (sí incluyen clases activas) (SCHMULLER, 2003, pág. 8).

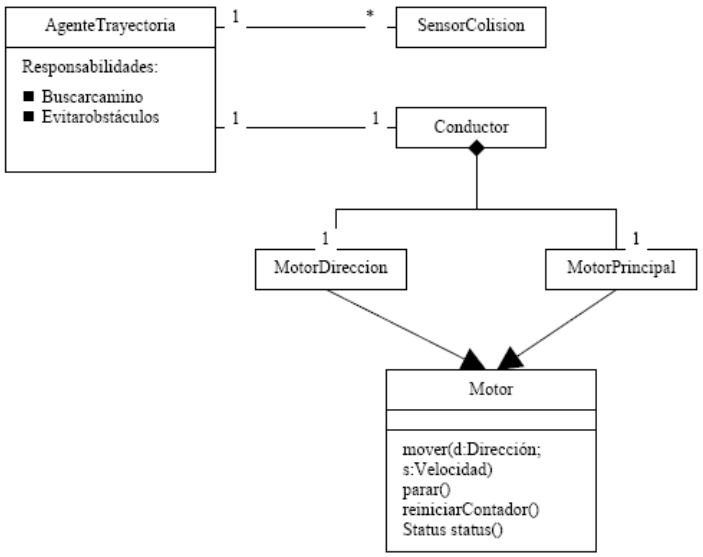

Figura 27: Diagrama de clases con colaboraciones simples

1.7.3.2.-Diagramas de Objetos

Muestran un conjunto de objetos y sus relaciones, son como fotos instantáneas de los diagramas de clases y cubren la vista de diseño estática o la vista de procesos estática desde la perspectiva de casos reales o prototípicos, ver figura 28 (SCHMULLER, 2003, pág. 9).

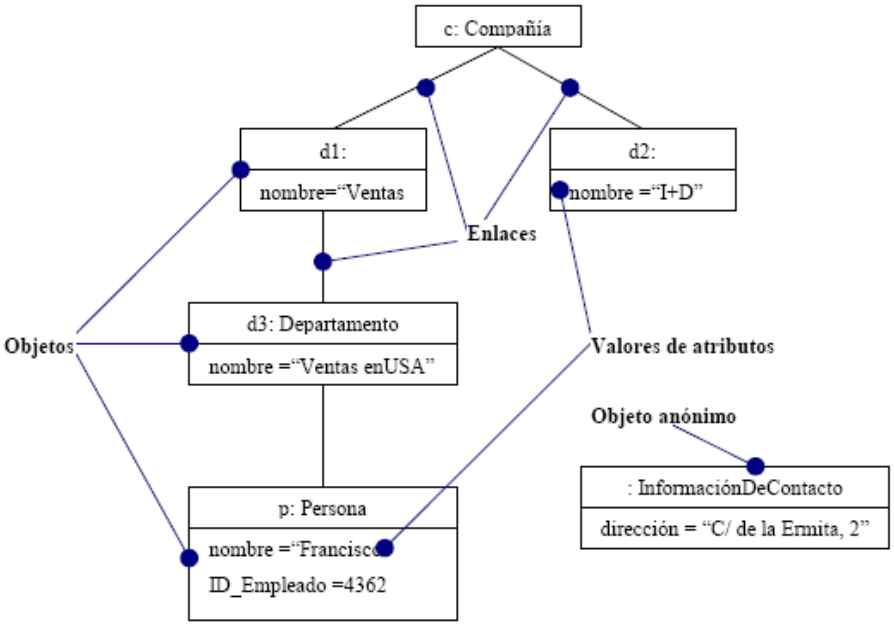

Figura 28: Diagrama de Objetos

1.7.3.3.-Diagramas de Casos de Usos

Muestran un conjunto de casos de uso y actores (tipo especial de clases) y sus relaciones, ver figura 29. Cubren la vista estática de los casos de uso y son especialmente importantes para el modelado y organización del comportamiento (SCHMULLER, 2003, pág. 10).

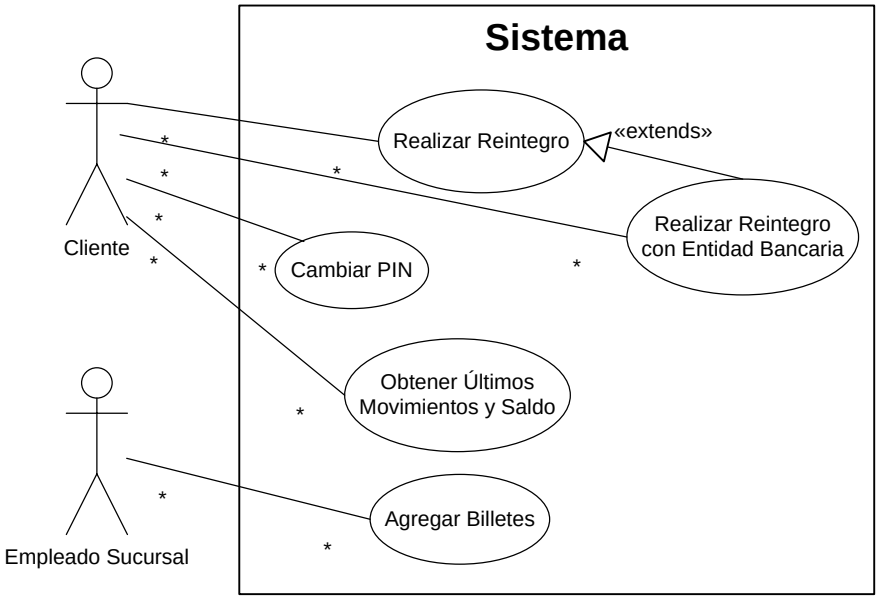

Figura 29: Diagrama Casos de uso

1.7.3.4.-Diagramas de Secuencia y de Colaboración

Tanto los diagramas de secuencia como los diagramas de colaboración son un tipo de diagramas de interacción. Constan de un conjunto de objetos y sus relaciones, incluyendo los mensajes que se pueden enviar unos objetos a otros. Cubren la vista dinámica del sistema. Los diagramas de secuencia enfatizan el ordenamiento temporal de los mensajes mientras que los diagramas de colaboración muestran la organización estructural de los objetos que envían y reciben mensajes, ver figura 30 y 31. Los diagramas de secuencia se pueden convertir en diagramas de colaboración sin pérdida de información, lo mismo ocurren en sentido opuesto (SCHMULLER, 2003, págs. 11-12).

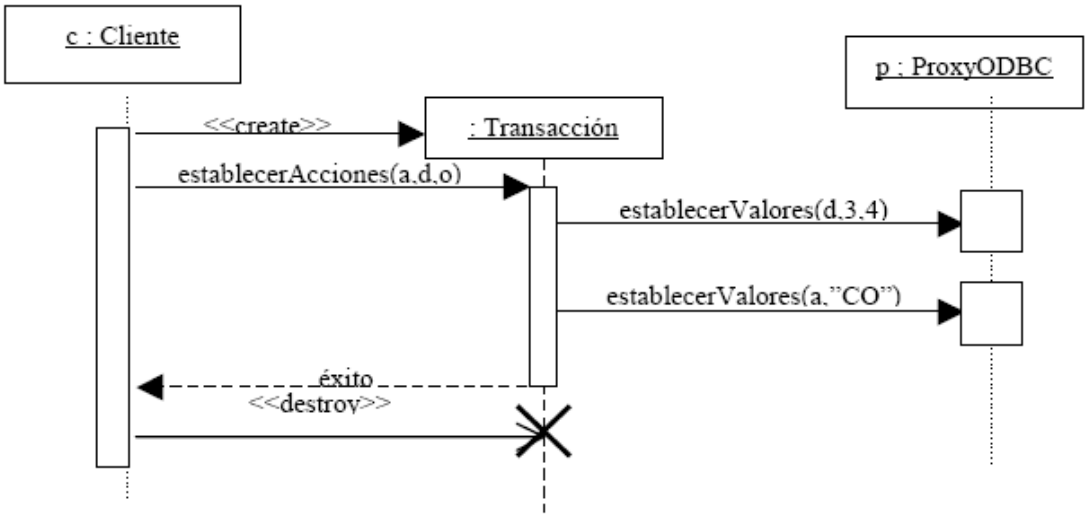

Figura 30: Diagrama de Secuencia

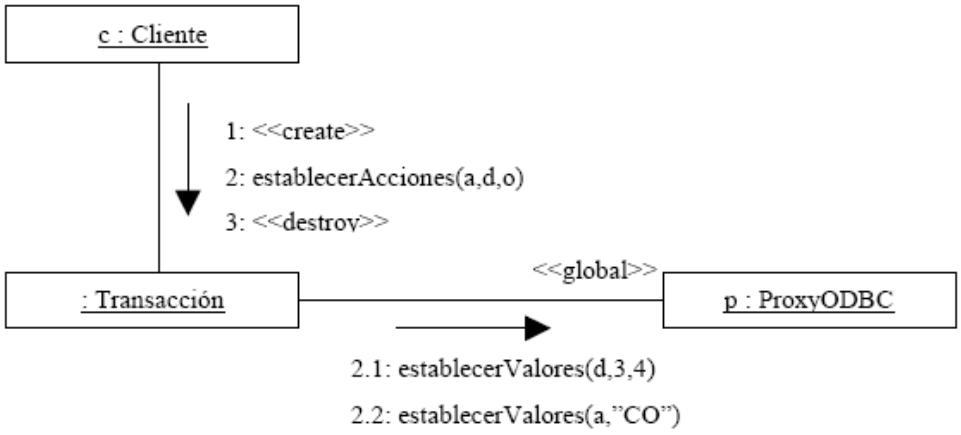

Figura 31: Diagrama de Colaboración

1.7.3.5.-Diagramas de Estados

Muestran una maquina de estados compuesta por estados, transiciones, eventos y actividades, ver figura 32. Estos diagramas cubren la vista dinámica de un sistema y son muy importantes a la hora de modelar el comportamiento de una interfaz, clase o colaboración (SCHMULLER, 2003, pág. 10).

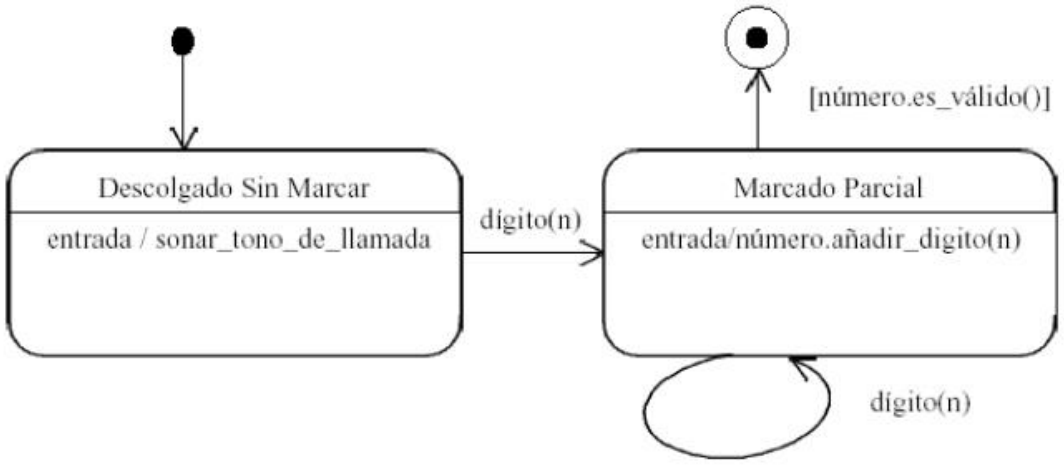

Figura 32: Diagrama de Estados

1.7.3.6.-Diagramas de Actividades

Son un tipo especial de diagramas de estados que se centra en mostrar el flujo de actividades dentro de un sistema, ver figura 33. Los diagramas de actividades cubren la parte dinámica de un sistema y se utilizan para modelar el funcionamiento de un sistema resaltando el flujo de control entre objetos (SCHMULLER, 2003, pág. 12).

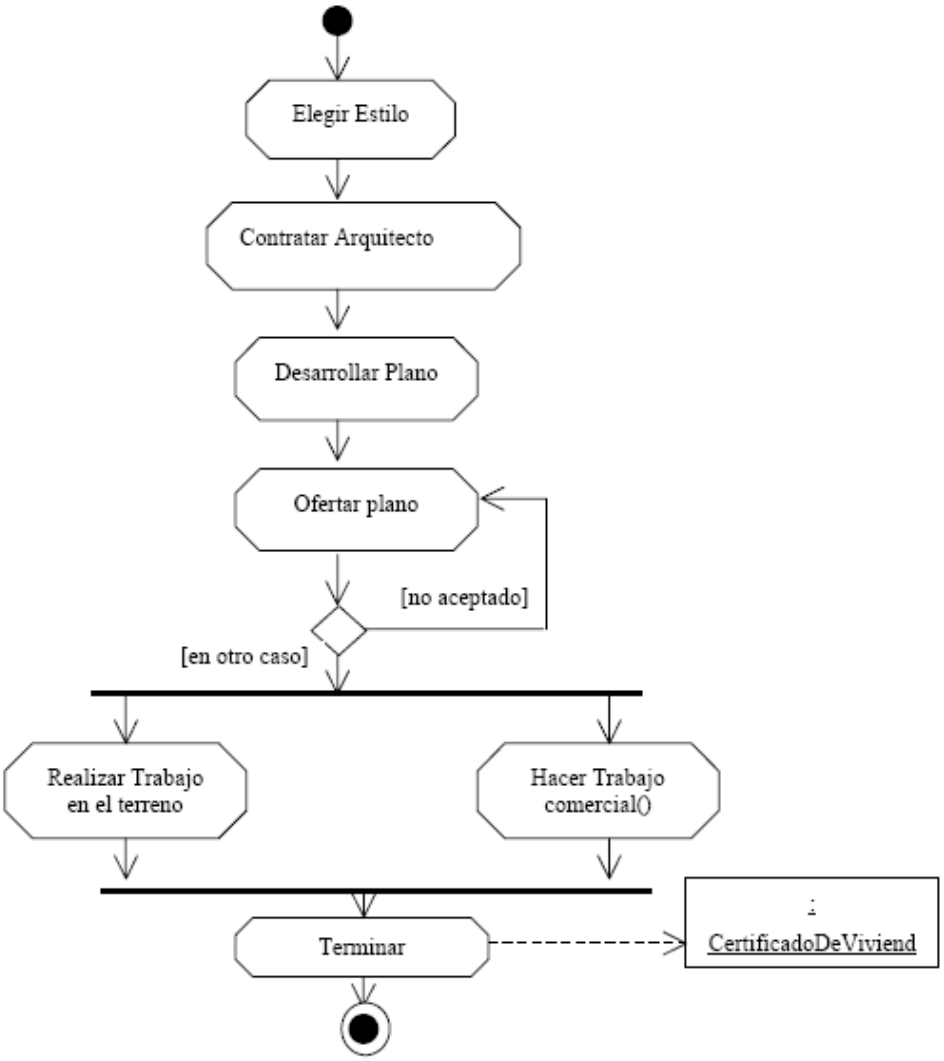

Figura 33: Diagrama de Actividades

1.7.3.7.-Diagramas de Componentes

Muestra la organización y las dependencias entre un conjunto de componentes. Cubren la vista de la implementación estática y se relacionan con los diagramas de clases ya que en un componente suele tener una o más clases, interfaces o colaboraciones (SCHMULLER, 2003, pág. 13).

1.7.4.-Arquitectura

El desarrollo de un sistema con gran cantidad de software requiere que este sea visto desde diferentes perspectivas. Diferentes usuarios (usuario final, analistas, desarrolladores, integradores, jefes de proyecto...) siguen diferentes actividades en diferentes momentos del ciclo de vida del proyecto, lo que da lugar a las diferentes vistas del proyecto, dependiendo de qué interese más en cada instante de tiempo.

La arquitectura es el conjunto de decisiones significativas sobre:

- La organización del sistema
- Selección de elementos estructurales y sus interfaces a través de los cuales se constituye el sistema.
- El Comportamiento, como se especifica las colaboraciones entre esos componentes.
- Composición de los elementos estructurales y de comportamiento en subsistemas progresivamente más grandes.
- El estilo arquitectónico que guía esta organización: elementos estáticos y dinámicos y sus interfaces, sus colaboraciones y su composición.

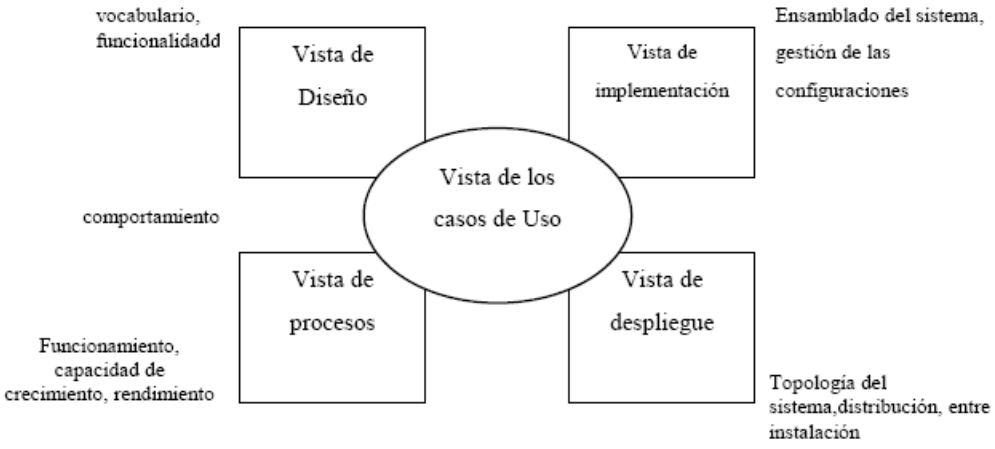

Figura 34: Vista general de un proyecto.

La una arquitectura que no debe centrarse únicamente en la estructura y en el comportamiento, sino que abarque temas como el uso, funcionalidad, rendimiento, capacidad de adaptación, reutilización, capacidad para ser comprendida, restricciones,

compromisos entre alternativas, así como aspectos estéticos. Para ello se sugiere una arquitectura que permita describir mejor los sistemas desde diferentes vistas, figura 18, donde cada una de ellas es una proyección de la organización y la estructura centrada en un aspecto particular del sistema (LEYTON G., 2002).

La **vista de casos de uso** comprende la descripción del comportamiento del sistema tal y como es percibido por los usuarios finales, analistas y encargados de las pruebas y se utilizan los diagramas de casos de uso para capturar los aspectos estáticos mientras que los dinámicos son representados por diagramas de interacción, estados y actividades.

La **vista de diseño** comprende las clases, interfaces y colaboraciones que forman el vocabulario del problema y de la solución. Esta vista soporta principalmente los requisitos funcionales del sistema, o sea, los servicios que el sistema debe proporcionar. Los aspectos estáticos se representan mediante diagramas de clases y objetos y los aspectos dinámicos con diagramas de interacción, estados y actividades.

La **vista de procesos** comprende los hilos y procesos que forman mecanismos de sincronización y concurrencia del sistema cubriendo el funcionamiento, capacidad de crecimiento y el rendimiento del sistema. Con UML, los aspectos estáticos y dinámicos se representan igual que en la vista de diseño, pero con el énfasis que aportan las clases activas, las cuales representan los procesos y los hilos.

La **Vista de implementación** comprende los componentes y los archivos que un sistema utiliza para ensamblar y hacer disponible el sistema físico. Se ocupa principalmente de la gestión de configuraciones de las distintas versiones del sistema. Los aspectos estáticos se capturan con los diagramas de componentes y los aspectos dinámicos con los diagramas de interacción, estados y actividades.

La **vista de despliegue** de un sistema contiene los nodos que forman la topología hardware sobre la que se ejecuta el sistema. Se preocupa principalmente de la distribución, entrega e instalación de las partes que constituyen el sistema. Los aspectos estáticos de esta vista se representan mediante los diagramas de despliegue y los aspectos dinámicos con diagramas de interacción, estados y actividades.

1.7.5.-Ciclo de Vida

Se entiende por ciclo de vida de un proyecto software a todas las etapas por las que pasa un proyecto, desde la concepción de la idea que hace surgir la necesidad de diseñar un

sistema software, pasando por el análisis, desarrollo, implantación y mantenimiento del mismo y hasta que finalmente muere por ser sustituido por otro sistema.

Aunque UML es bastante independiente del proceso, para obtener el máximo rendimiento de UML se debería considerar un proceso que fuese:

- Dirigido por los casos de uso, o sea, que los casos de uso sean un artefacto básico para establecer el comportamiento del deseado del sistema, para validar la arquitectura, para las pruebas y para la comunicación entre las personas involucradas en el proyecto.
- Centrado en la arquitectura de modo que sea el artefacto básico para conceptuar, construir, gestionar y hacer evolucionar el sistema.
- Un proceso iterativo, que es aquel que involucra la gestión del flujo de ejecutables del sistema, e incremental, que es aquel donde cada nueva versión corrige defectos de la anterior e incorpora nueva funcionalidad. Un proceso iterativo e incremental se denomina dirigido por el riesgo, lo que significa que cada nueva versión se ataca y reducen los riesgos más significativos para el éxito del proyecto.

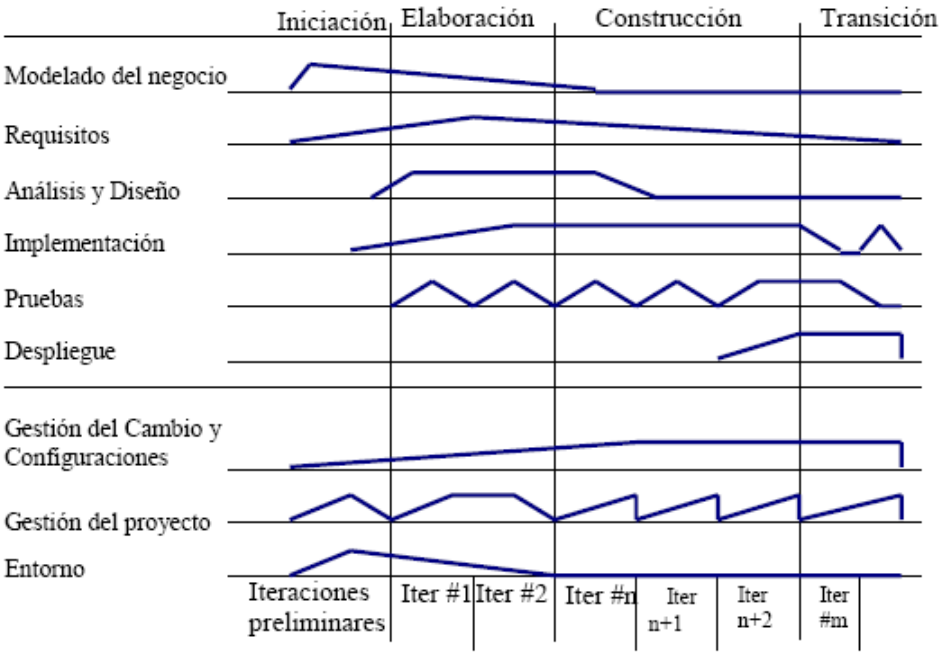

Figura 35: Ciclo de vida de proyecto de Software

Este proceso, dirigido a los casos de uso, centrado en la arquitectura, iterativo e incremental pude descomponerse en fases, donde cada fase es el intervalo de tiempo entre dos hitos importantes del proceso, cuando se cumplen los objetivos bien definidos, se completan los artefactos y se toman decisiones sobre si pasar o no a la siguiente fase. En el ciclo de vida de un proyecto software existen cuatro fases, ver en la figura 30. La iniciación, que es cuando la idea inicial está lo suficientemente fundada para poder garantizar la entrada en la fase de elaboración, esta fase es cuando se produce la definición de la arquitectura y la visión del producto. En esta fase se deben determinar los requisitos del sistema y las pruebas sobre el mismo. Posteriormente se pasa a la fase de construcción, que es cuando se pasa de la base arquitectónica ejecutable hasta su disponibilidad para los usuarios, en esta fase se reexaminan los requisitos y las pruebas que ha de soportar. La transición, cuarta fase del proceso, que es cuando el software se pone en mano de los usuarios.

Raramente el proceso del software termina en la etapa de transición, incluso durante esta fase el proyecto es continuamente reexaminado y mejorado erradicando errores y añadiendo nuevas funcionalidades no contempladas. Un elemento que distingue a este proceso y afecta a las cuatro fases es una iteración, que es un conjunto de bien definido de actividades, con un plan y unos criterios de evaluación, que acaban en una versión del producto, bien interna o externa.

1.8.-XML(Lenguaje de Marcado Extensible)

XML (Extensible Markup Language) es un metalenguaje de etiquetas, lo que significa que es un lenguaje que se utiliza para definir lenguajes de etiquetas (POSADAS, 2000). A cada lenguaje creado con XML se le denomina vocabulario XML, y la documentación generada por el compilador de C# está escrita en un vocabulario de este tipo.

XML como: El formato universal para documentos y datos estructurados en Internet, y se pueden explicar las características de su funcionamiento a través de 7 puntos importantes, tal y como recomienda W3C.

XML es un estándar para escribir datos estructurados en un fichero de texto. Por datos estructurados se entiende como tipos de documentos que van desde las hojas de cálculo, o las libretas de direcciones de Internet, hasta parámetros de configuración, transacciones financieras o dibujos técnicos. Los programas que los generan, utilizan normalmente formatos binarios o de texto. XML es un conjunto de reglas, normas y convenciones para diseñar formatos de texto para tales tipos de datos, de forma que produzca ficheros fáciles de generar y de leer, que carezcan de ambigüedades y que

eviten problemas comunes, como la falta de extensibilidad, carencias de soporte debido a características de internacionalización, o problemas asociados a plataformas específicas (POSADAS, 2000).

XML está en formato texto, pero no para ser leído. Esto le da innumerables ventajas de portabilidad, depuración, independencia de plataforma, e incluso de edición, pero su sintaxis es más estricta que la de HTML: una marca olvidada o un valor de atributo sin comillas convierten el documento en inutilizable. No hay permisividad en la construcción de documentos, ya que esa es la única forma de protegerse contra problemas más graves.

El Modelo de Objetos de Documento (DOM) es un conjunto estándar de funciones para manipular documentos XML (y HMTL) mediante un lenguaje de programación. XML Namespaces, es una especificación que describe cómo puede asociarse una URL a cada etiqueta de un documento XML, otorgándoles un significado adicional. Y finalmente, XML-Schemas es un modo estándar de definir los datos incluidos en un documento de forma más similar a la utilizada por los programadores de Bases de Datos, mediante los metadatos asociados. Y hay otros en desarrollo, pero todos están basados en el principal: XML.

XML no requiere licencias, es independiente de la plataforma, y tiene un amplio soporte. La selección de XML como soporte de aplicaciones, significa entrar en una comunidad muy amplia de herramientas y desarrolladores, y en cierto modo, se parece a la elección de SQL respecto a las Bases de Datos. Todavía hay que utilizar herramientas de desarrollo, pero la tranquilidad del uso del estándar y de su formato, hacen que las ventajas a la larga sean notables.

XML es un meta-lenguaje de marcas, es decir, permite que el usuario diseñe sus propias marcas (tags) y les dé el significado que se le antoje, con tal de que siga un modelo coherente. La primera gran ventaja de XML es el hecho de que los datos se auto-definen a sí mismos y esa definición pueden encontrase en la propia página XML (ó en otra separada a la que se hace referencia), suministrando así a cualquier programa que abra ese fichero la información necesaria para el manejo de su contenido. De hecho, un fichero XML correctamente escrito requiere dos cualidades que establecen su grado de conformidad con las reglas establecidas:

Fichero XML bien formado (well formed) es aquel que se ha escrito de acuerdo con el estándar.

Fichero XML válido es aquel que -cumpliendo con la definición del estándar- está, además, lógicamente bien estructurado y define en su totalidad cada uno de sus contenidos sin ambigüedad alguna.

Por si esto fuera poco, su formato (texto plano) permite su transporte y lectura bajo cualquier plataforma o herramienta y le da un valor de universalidad que no se puede conseguir con ningún formato nativo, por mucha aceptación que tenga. Incluso su convivencia con el código HTML actual está garantizada gracias a que existe una marca HTML definida con la sintaxis <XML>Fichero XML</XML>, que permite definir una ubicación de un fichero XML externo que puede ser embebido en el documento y tratado de forma conjunta. Durante largo tiempo (...), se prevé una coexistencia de ambos, pero bien entendido, que HTML sólo es un caso particular de XML, donde su DTD define cómo representar los documentos en un navegador.

Recientemente, se ha definido un nuevo estándar por parte de la W3C que recibe el nombre de

XHTML. Se trata, utilizando su propia definición, de “una reformulación del lenguaje HTML usando la sintaxis XML”. Esto supone que aquellos documentos escritos en XHTML deberán de guardar las restricciones del lenguaje que impone XML, aunque el lenguaje en sí, siga siendo HTML. Esto quiere decir que no se permiten etiquetas sin cerrar, atributos sin entrecomillar, etc. Su propósito es normalizar más fuertemente el lenguaje HTML de cara a las nuevas generaciones de navegadores, permitiendo la integración de los dos lenguajes y evitando las ambigüedades existentes en la actualidad.

1.8.1.-Las DTD (Definición de Tipo de Documento)

Si se quiere tener la seguridad de que un fichero XML pueda ser perfectamente interpretado por cualquier herramienta, se puede (aunque no es obligatorio) incluir una definición de su construcción que preceda a los datos propiamente dichos. Este conjunto de meta-datos recibe el nombre de DTD ó Definición de Tipo de Documento. Esta definición sí que debe basarse en una normativa rígida, que es la definida por XML.

Así, si una herramienta es capaz de interpretar XML, eso significa que posee un analizador sintáctico (Parser) que es capaz de contrastar la definición dada por el autor del documento contra la especificada por la normativa, indicando si hay errores, y, de no ser así, presentando el documento de la forma adecuada. Esto es por ejemplo lo que hace Internet Explorer 5.0, o Netscape 6.0, cuando abren un documento XML.

Por lo tanto, las DTD's son la clave de la auto-descripción: siguiendo el modelo establecido por XML, permiten definir al usuario qué significa exactamente cada una de las marcas que va a incluir a continuación para identificar los datos. De ella, se derivan otras ventajas, como la posibilidad de soslayar el modelo simétrico de datos (el formato clásico de datos tiene que tener forma rectangular: filas y columnas), permitiendo diseñar modelos de datos totalmente jerárquicos que adopten la forma de árboles asimétricos si ello describe el contenido de forma más precisa, y evitando repeticiones.

La estructura de árbol se presenta, pues, más rica en posibilidades de representación que la tabular, permitiendo representar la información de forma más exacta.

Ejemplo XML

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Mensaje [ <!ELEMENT Contenido (#PCDATA)> ]>
<!-- este es un comentario -->
<Contenido>¡Hola, mundo!</Contenido>
```

En el ejemplo se pueden observar 3 líneas clave: La primera, es la definición general. Nos indica que lo que viene a continuación es un documento XML (las de inicio y fin son el carácter obligatorio que delimita esa definición. Además, se observan dos tributos: versión -que se establece a 1.0- que nos indica que el intérprete de XML debe de utilizar las normas establecidas en Febrero/98 y enconding, asignado a “UTF-8”, y que el estándar recomienda incluir siempre, aunque algunos navegadores (como Explorer 5) no lo exijan de forma explícita.

Téngase en cuenta que XML debe soportar características internacionales, por tanto se dice que, tras su interpretación, todo documento XML devuelve Unicode. El valor por defecto es “UTF-8”.

La segunda línea es una DTD muy simple. Consta de la declaración de tipo de documento mediante !DOCTYPE seguido del nombre genérico que va a recibir el objeto que se defina a continuación (mensaje), e indica que sólo va a contener un elemento (!ELEMENT) que también se denominará mensaje y que está compuesto de texto (#PCDATA).

Finalmente, la cuarta línea (la tercera es un simple comentario) contiene la información en sí. Dentro de dos etiquetas de apertura y cierre con el nombre definido en la línea 2,

se incluye la información propiamente dicha. Si se visualiza el documento en Internet Explorer 5.0 o superior se obtendrá una salida como la de figura 31.

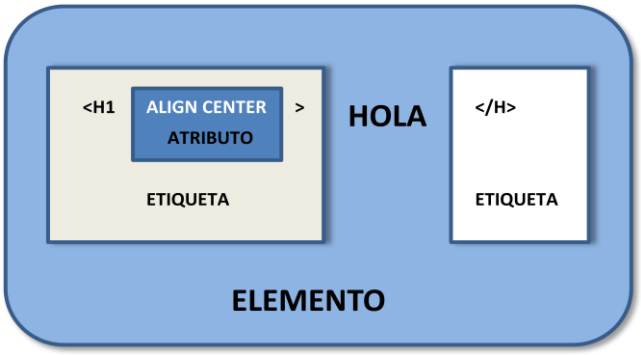

Figura 36: Ejemplo XML

1.8.2.-El concepto de elemento en XML

Por elemento se entiende el conjunto de fragmentos que componen la etiqueta o tag HTML: La etiqueta de cabecera o identificador, los atributos, el cuerpo o contenido y la etiqueta de cierre.

1.8.3.-Restricciones sintácticas del lenguaje XML

1. No se permite la anidación incorrecta de elementos (Deben cerrarse correctamente, según el grado de profundidad de la anidación).

2. No se permiten elementos sin etiqueta de cierre (etiquetas cuyo cierre era opcional en HTML).

3. Los elementos que no poseen contenido (etiquetas del tipo <HR SIZE="3">), deben de utilizar una etiqueta de cierre, o usar la abreviatura permitida en XML, consistente en incluir una barra vertical (/) antes del carácter de cierre (>). Por ejemplo, en el caso anterior se tendría que expresar mediante <HR SIZE="3" />.

4. Todos los atributos deben de ir entre comillas dobles (como en el ejemplo anterior).

5. XML diferencia entre mayúsculas y minúsculas (quizás se trata de la característica más incómoda al principio, pues produce errores en los analizadores sintácticos, que pueden ser difíciles de detectar.

Por otro lado, la implementación del parser HTML de Internet Explorer 4.01/5.0 soporta la inclusión de una etiqueta <XML> (con su correspondiente etiqueta de cierre), en la cual se puede incluir documentos enteros escritos en XML y que serán tratados adecuadamente de forma separada del Código HTML, si bien lo más normal es que -si incluya una etiqueta XML en un documento HTML- se haga para referenciar un documento externo XML que será cargado en memoria y procesado, lo que es preferible por razones de claridad.

1.8.4.-Características de las DTD

El estándar define un conjunto limitado de identificadores para la creación de estructuras de datos: DOCTYPE, ELEMENT, y ATTLIST.

Como todo documento XML tiene que incluir un nodo raíz (recuerde el lector que XML extiende la metáfora de estructura tabular -filas y columnas-, a una estructura jerárquica), DOCTYPE define el nombre del nodo raíz del árbol jerárquico, así como el punto de inicio del documento XML. También sirve para la declaración del origen del propio DTD.

El identificador ELEMENT sirve para declarar cada uno de los elementos de un DTD. Sirve tanto para la descripción de elementos simples como de elementos compuestos (de otros elementos). Dispone de un grupo de símbolos asociados que permiten al diseñador indicar la cardinalidad de los elementos (si puede existir uno ó más de uno, uno o ninguno, etc.), así como de especificaciones consistentes en palabras reservadas, que informan acerca del tipo de dato, si es requerido ó no, si tiene un valor por defecto, si puede adoptar cualquier otra forma, etc. Finalmente, el identificador ATTLIST permite la definición de listas de atributos para los elementos, pudiendo también incluir datos sobre los valores aceptables para ese elemento, y valores por defecto. En el Código Fuente del ejemplo de abajo se implementa a partir de la Base de Datos Pedidos utilizada como origen de registros (POSADAS, 2000).

Sintaxis:

```
<?xml version="1.0" encoding="UTF-8"?>
<Clientes>
<Cliente>
<IdCliente>1</IdCliente>
<Empresa>Tecnología SA DE CV</Empresa>
<Contacto>Pedro Pérez</Contacto>
<Ciudad>Pachuca</Ciudad>
<Estado>Hidalgo</Estado>
```

CAPÍTULO 2.- Conceptos de la Plataforma .NET y C#

2.1.-FrameWork

El uso de los frameworks en las implementaciones de software se ha convertido en las bases para el desarrollo de software, un framework, según su traducción literal es un marco de trabajo. Este marco de trabajo, ofrece a quien lo utiliza, una serie de herramientas para facilitarle la realización de determinada tarea.

Un framework puede estar compuesto por librerías de clases, documentación y ayuda, ejemplos, tutoriales e incluso foros de discusión. Es posible que se utilicen varios frameworks a la vez, o incluso que algunos sean soporte de otros.

.NET Framework es un componente Windows que soporta el desarrollo y ejecución de aplicaciones Windows y Web Services. El propósito de este componente es proveer al usuario con un entorno de programación orientada a objetos consistente, donde el código pueda estar almacenado localmente o de manera remota.

Intenta minimizar los conflictos con el deploy y versionado de software y promueve la ejecución de código segura. El propósito es hacer que la experiencia del desarrollador sea consistente entre una gran variedad de aplicaciones y plataformas y crear una comunicación estándar que ayude a las aplicaciones (MSDN Library Framework, 2009).

El .NET Framework tiene dos componentes principales. La Common Runtime (CLR) y la Class Library.

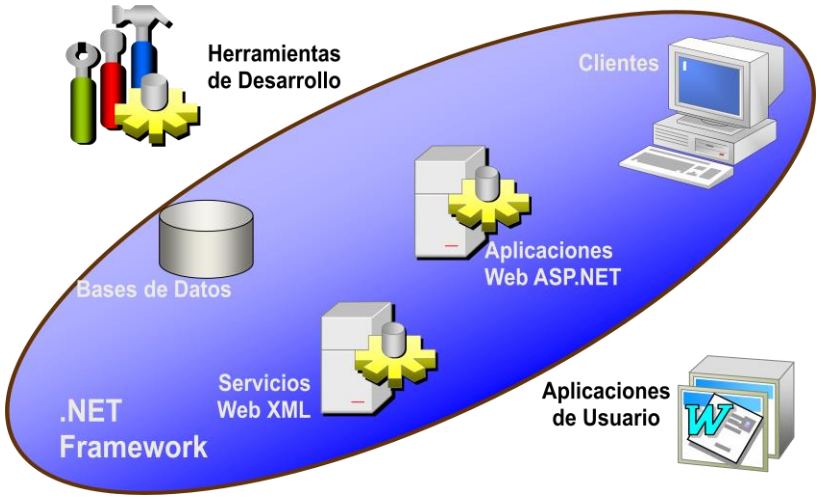

Figura 37: .NET Framework

El .NET Framework 2.0

El .NET Framework provee las herramientas necesarias en run-time y compile-time para construir y ejecutar aplicaciones basadas en .NET

Application Services

El .NET Framework expone servicios de aplicaciones a través de clases de la .NET Framework Class Library.

Common Language Runtime 2.0

La Common Language Runtime (CLR) simplifica el desarrollo de aplicaciones, provee un entorno de ejecución robusto y seguro, soporta varios lenguajes y simplifica el despliegue y la administración. La CLR es un entorno administrado (managed), en el cual los servicios comunes, como garbage collection y seguridad, son provistos automáticamente.

.NET Framework Class Library 2.0

La librería de clases de .NET Framework expone características en tiempo de ejecución y provee otros servicios útiles para todos los desarrolladores. Las clases simplifican el desarrollo basado en .NET. Los desarrolladores pueden extenderlas creando sus propias librerías de clases. Las librerías de clases base implementan el .NET Framework. Todas las aplicaciones (web, windows, web services) acceden a las mismas clases base. Estas están almacenadas en namespaces. Los diferentes lenguajes acceden a las mismas librerías (MSDN Library Framework, 2009).

ADO.NET 2.0

ADO.NET provee soporte para modelos de programación desconectada. Además proveen soporte para XML enriquecido.

ASP.NET 2.0

Microsoft ASP.NET es un framework de programación que está montado sobre la CLR. ASP.NET puede ser utilizado sobre un servidor para construir poderosas aplicaciones web. ASP.NET Web Forms provee un poderoso y sencillo método para construir Interfaces de usuario (UI) dinámicas.

XML Web Services

Componentes Web programables que pueden ser compartidos entre aplicaciones, sobre Internet o una intranet. El .NET Framework provee herramientas y clases para desarrollo, testeo y distribución de XML Web Services.

User Interfaces

El .NET Framework soporta tres tipos de Interfaces de usuario: Web Forms, Windows Forms, Aplicaciones de Consola.

Lenguajes

Cualquier lenguaje que sea acorde a la Common Language Specification (CLS) puede ejecutarse sobre la CLR. En .NET Framework, Microsoft provee Visual Basic, Visual C++, Visual C#, Visual J#. Terceros pueden proveer nuevos lenguajes.

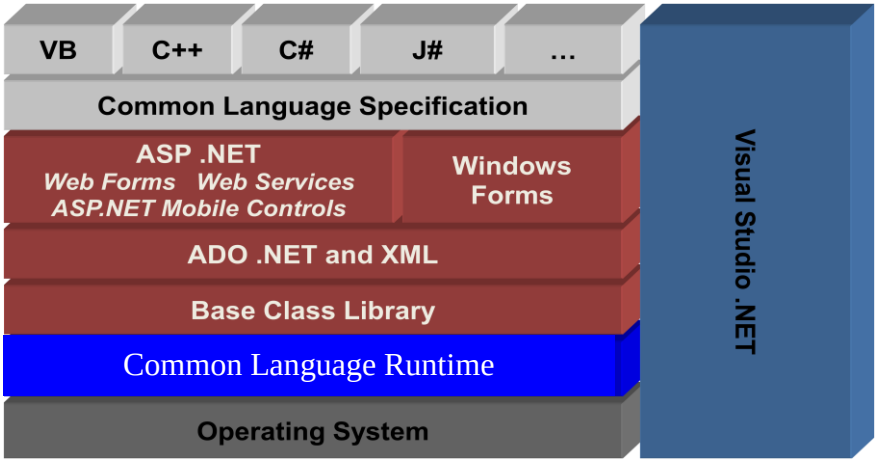

Figura 38. Nivel de ejecución de una aplicación

2.2.-CLR (Common Language Runtime)

El .NET Framework resuelve la comunicación entre distintas aplicaciones, permite la reusabilidad de módulos, brinda la posibilidad a los desarrolladores de utilizar el lenguaje en el que más cómodos se sienten.

La Common Language Runtime es el corazón del .NET Framework. Los compiladores y herramientas exponen funcionalidad en tiempo de ejecución y permiten escribir código con el beneficio de un entorno de ejecución administrado. El código que se desarrolla con un compilador de lenguaje que trabaja con el runtime se llama código administrado

(managed code) (MSDN Library Framework, 2009). Esto permite beneficios como integración y manejo de excepciones entre distintos lenguajes, seguridad mejorada, versionamiento y soporte para despliegue. Además de un modelo simplificado para interacción de componentes y servicios de debugging y profiling.

Para permitir al runtime proveer servicios al código administrado, los compiladores deben emitir metadata (información adicional) que describe tipos, miembros y referencias en el código. La metadata se almacena con el código. Cada archivo que la CLR puede cargar contiene metadata. El runtime la utiliza para localizar y cargar las clases, mantener las instancias en memoria, resolver el llamado de métodos, generar código nativo, mejorar la seguridad y definir las fronteras del contexto de ejecución.

CLR administra la memoria utilizada por las aplicaciones, evitando pérdidas de memoria que podrían estar originadas por errores en el código escrito. El entorno de ejecución brinda además un entorno de ejecución que permitirá y administrará la conversión de tipos de los valores con los que operan las aplicaciones, la inicialización de las variables, el control de overflows, etc.

Permite además que convivan diferentes versiones de una misma dll, sin que se generen conflictos.

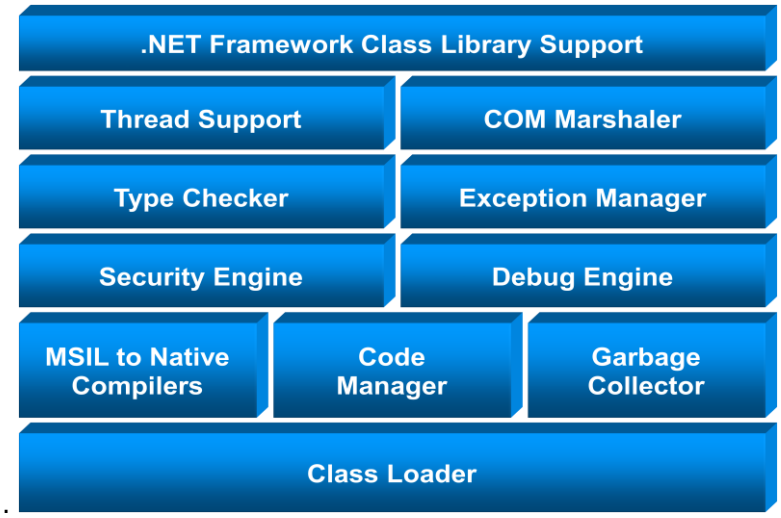

Figura 39: Entorno Administrado

La Common Language Runtime simplifica el desarrollo de aplicaciones, brindando un entorno de ejecución seguro y robusto, con soporte para múltiples lenguajes, tal como se muestra en la figura 39. Este entorno se conoce generalmente como entorno

administrado o managed environment, en el cual son provistos automáticamente los servicios comunes, como garbage collector y security.

Class loader

Administra metadata (información provista con los archivos), carga y disponibilidad de las clases.

Microsoft intermediate language (MSIL) to native compiler

Convierte MSIL a código nativo (JIT)

Code manager

Administra la ejecución de código.

Garbage collector (GC)

Provee la administración automática del ciclo de vida de todos los objetos.

Security engine

Provee la seguridad basada en el origen de código y en el usuario que lo ejecuta.

Debug engine

Permite realizar el debug de la aplicación a partir de un trazado del código que está siendo ejecutado.

Type checker

Evita que se realicen casteos inseguros o se utilicen variables no inicializadas.

Exception manager

Provee una estructura de manejo de excepciones, la cual se integra con Windows Structured Exception Handling.

Thread support

Provee clases e interfaces para trabajar con programación multihilos.

COM marshaler

Provee interoperabilidad entre .NET y COM

Base Class Library (BCL)

Integra el código con el runtime que soporta BCL

2.3.-CTS (Common Type System)

CTS (Common Type System) o Sistema de Tipo Común se define como la forma en la que los tipos deben ser declarados, utilizados y administrados en el runtime, ver figura 40. Además es una parte del runtime para el soporte en la integración de varios lenguajes.

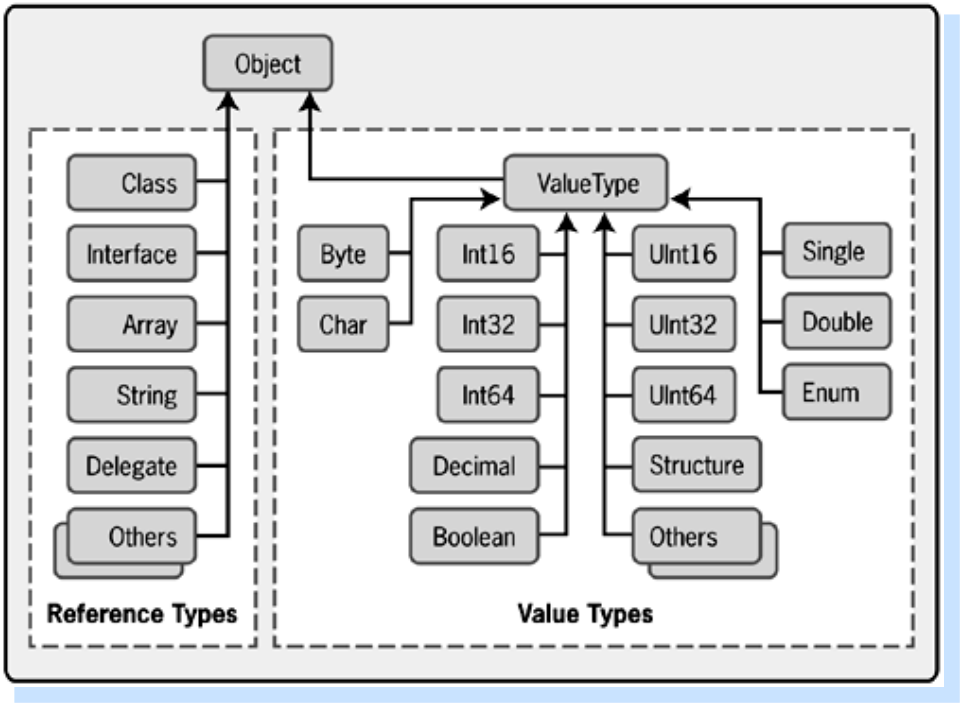

Figura 40: CTS

El sistema de tipos comunes realiza las siguientes funciones:

- Establecer un framework para soporte de integración de múltiples lenguajes, seguridad de tipos y alta performance en la ejecución de código.
- Provee un modelo orientado a objetos que soporta la implementación de varios lenguajes de programación.
- Define las reglas que debe seguir un lenguaje, lo que asegura que distintos lenguajes puedan interactuar sin problemas.

Clasificación de tipos

CTS soporta dos categorías generales de tipos, cada una de las cuales se divide en subcategorías:

1. **Value types:** Directamente contienen sus datos. Las instancias de los value types son alocadas en la stack o de manera inline en una estructura. Pueden ser incorporados (implementados por el runtime), definidos por el usuario o enumeraciones.
2. **Reference types**: almacenan una referencia a una dirección de memoria con un valor, y son alocados en la heap. Los Reference types pueden tipos auto-descriptivos, punteros, o interfaces.

Todos los tipos derivan de System.Object, que es el tipo base.

2.4.-MSIL (Microsoft Intermediate Lenguage)

Cuando se compila código soportado en .NET Framework, el compilador convierte el código fuente en Lengua intermedio de Microsoft (MSIL), que es un conjunto de instrucciones independiente de la CPU que se pueden convertir de forma eficaz en código nativo.

MSIL incluye instrucciones para cargar, almacenar, inicializar y llamar a métodos en los objetos, así como instrucciones para operaciones lógicas y aritméticas, flujo de control, acceso directo a la memoria, control de excepciones y otras operaciones.

Antes de poder ejecutar código, se debe convertir MSIL al código específico de la CPU, normalmente mediante un compilador JIT (Just in time).

Common Language Runtime proporciona uno o varios compiladores JIT para cada arquitectura de equipo compatible, por lo que se puede compilar y ejecutar el mismo conjunto de MSIL en cualquier arquitectura compatible.

Cuando el compilador produce MSIL, también genera información adicional sobre el código. Esta información describe los tipos que aparecen en el código, incluidas las definiciones de los tipos, las firmas de los miembros de tipos, los miembros a los que se hace referencia en el código y otros datos que el motor de tiempo de ejecución utiliza en tiempo de ejecución.

El lenguaje intermedio de Microsoft (MSIL) y los datos adicionales, conocidos como MetaData, se incluyen en un archivo ejecutable portable (PE), que se basa y extiende el

PE de Microsoft publicado y el formato Common Object File Format (COFF) utilizado tradicionalmente para contenido ejecutable. Este formato de archivo, que contiene código MSIL o código nativo así como metadata, permite al sistema operativo reconocer imágenes de Common Language Runtime (MSDN Library Framework, 2009).

La presencia de metadatos junto con el Lenguaje intermedio de Microsoft (MSIL) permite crear códigos autodescriptivos, con lo cual las bibliotecas de tipos y el Lenguaje de definición de interfaces (IDL) son innecesarios.

El motor de tiempo de ejecución localiza y extrae los metadatos del archivo cuando son necesarios durante la ejecución.

2.5.-Lenguaje C#

El Lenguaje de programación C# fue creado con la misma idea de los lenguajes de C y C++. Esto explica en gran medida su fácil curva de aprendizaje y lo interesante de sus prestaciones. C# fue creado desde cero. Por esta razón Microsoft elimino algunas de las prestaciones más pesadas, como los punteros (FERGUSON, PATERSON, & BERES, 2003, págs. 44-50).

Microsoft diseño C# de forma que retuviera casi toda la sintaxis de C y C++, así los programadores que están familiarizados con C y C++, puedan programar en este ámbito. Sin embargo la gran ventaja de C# consiste en que los diseñadores decidieron no hacerlo compatible con los anteriores C y C++ y así eliminar las deficiencias que tenían estos lenguajes.

C# lenguaje presentado en .NET Framework procede de C++. Sin embargo, C# es un lenguaje orientado a objetos desde el inicio.

2.5.1.-Características del Lenguaje

C# está totalmente integrado con .NET Framework, por tanto los tipos de datos, estructuras y funciones que se utilizan con este lenguaje están definidos en este "marco de trabajo", en los siguientes temas se describen cada una de las características de este lenguaje de programación.

2.5.1.1.-Variables

RAM que se puede utilizar en el código. En .NET todas las variables deben ser obligatoriamente declaradas y tener un tipo, pero no es necesario declarar todas las variables al inicio del programa. Como el ambiente .NET trata de código seguro, eso asegura la integridad de la variable, o sea, que los tipos asignados siempre estén de acuerdo a lo definido. El contenido de una variable podrá cambiar a lo largo del programa, pues eso es común. No se puede asignar un valor a una variable sin definirla, pues eso causará un error. Las variables son usadas en situaciones en las que se necesita almacenar en la memoria (TROELSEN, 2008, págs. 81-90).

C#: el tipo de variable precede al identificador, ejemplo:

```
int x;
decimal y;
rectangle z;
Cliente cli;
```

En la declaración de una variable se deberá informar su nombre y su tipo. Una variable no puede ser declarada sin su respectivo tipo, ya que .NET no lo permite. Se declara el tipo y a continuación el nombre de la variable seguido de su respectivo contenido.

En Visual Basic.NET se hace de la manera inversa. Primero se utilizar la palabra clave DIM seguida del nombre de la variable, y luego la palabra reservada AS seguida del tipo. Es una práctica común asignar el valor de la variable directamente en la misma línea, aunque esto no siempre será posible debido al flujo de información del programa. Una variable puede contener el resultado de otras. Si se declara en la misma línea dos variables con el mismo tipo, ambas serán de ese tipo. Además se puede declara en la misma línea diversas variables con tipos diferentes, y la regla es que todas estas serán del tipo declarado luego de la palabra clave AS.

2.5.1.2.-Tipos de Variables

La mayoría de las variables manejadas en versiones pasadas de C y C++siguen estando presentes. El espacio de nombres System es el espacio de nombres de la raíz de los tipos fundamentales de .NET Framework. Este espacio de nombres contiene clases que representan los tipos de datos base que se utilizan en todas las aplicaciones: Object (raíz

de la jerarquía de herencia), Byte, Char, Array, Int32, String, etc. Muchos de estos tipos se corresponden con los tipos de datos primitivos que utiliza el lenguaje de programación. Cuando se escribe código utilizando tipos de .NET Framework se puede utilizar la palabra clave correspondiente del lenguaje cuando se espera un tipo de datos base de .NET Framework (FERGUSON, PATERSON, & BERES, 2003, págs. 75-95).

Categoría	Nombre de la clase	Descripción	Tipo de datos en Visual Basic	Tipo de datos en C#	Tipo de datos de C++	Tipo de datos en JScript
Integer	Byte	Entero de 8 bits sin signo.	**Byte**	**byte**	**char**	**Byte**
	SByte	Entero de 8 bits con signo. No cumple CLS.	**SByte**	**sbyte**	**signed char**	**SByte**
	Int16	Entero de 16 bits con signo.	**Tipo Short**	**short**	**short**	**short**
	Int32	Entero de 32 bits con signo.	**Integer**	**int**	**int** O bien **long**	**int**
	Int64	Entero de 64 bits con signo.	**Tipo Long**	**long**	**__int64**	**long**
	UInt16	Entero de 16 bits sin signo. No cumple CLS.	**UShort**	**ushort**	**unsigned short**	**UInt16**
	UInt32	Entero de 32 bits sin signo. No cumple CLS.	**UInteger**	**uint**	**unsigned int** O bien **unsigned long**	**UInt32**
	UInt64	Entero de 64 bits sin signo. No cumple CLS.	**ULong**	**ulong**	**unsigned __int64**	**UInt64**
Punto flotante	Single	Número de punto flotante (32 bits) de precisión simple.	**Sencillo**	**float**	**float**	**float**
	Double	Número de punto flotante (64 bits) de doble precisión.	**Tipo Double**	**double**	**double**	**double**
Lógico	Boolean	Valor booleano (verdadero o falso).	**Booleano**	**bool**	**bool**	**bool**
Otros	Char	Carácter Unicode (16 bits).	**Tipo Char**	**char**	**wchar_t**	**char**
	Decimal	Valor decimal (128 bits).	**Decimal**	**decimal**	**Decimal**	**Decimal**
	IntPtr	Entero con signo cuyo tamaño depende de la plataforma subyacente (valor de 32 bits en una plataforma de 32 bits y valor de 64 bits en una plataforma de 64 bits).	**IntPtr** No dispone de un tipo integrado.	**IntPtr** No dispone de un tipo integrado.	**IntPtr** No dispone de un tipo integrado.	**IntPtr**
	UIntPtr	Entero sin signo cuyo tamaño depende de la plataforma subyacente (valor de 32 bits en una plataforma de 32 bits y valor de 64	**UIntPtr** No dispone de un tipo integrado.	**UIntPtr** No dispone de un tipo integrado.	**UIntPtr** No dispone de un tipo integrado.	**UIntPtr**
Objetos de clase	Object	Base de la jerarquía de objetos.	**Objeto**	**object**	**Object ***	**Objecto**
	String	Cadena inmutable de longitud fija de caracteres Unicode.	**Cadena**	**string**	**String***	**Cadena**

Tabla 8: Variables

2.5.1.3.-Nomenclaturas

La nomenclatura de las variables depende de la metodología y del patrón cada empresa utiliza.

Hay distintas formas de trabajar, una bastante común es que cada variable comience con el tipo de la variable y luego el nombre.

La facilidad proporcionada por esta nomenclatura ayuda en códigos muy extensos, pero no influye en ninguna forma sobre la performance o el desempeño.

Otra forma es haciendo referencia al lo que representa la variable y el ámbito en el que esta, como en el ejemplo.

Es importante adoptar un patrón, sobre todo para el trabajo con grupos numerosos de desarrolladores o en proyectos medianos y grandes.

```
private string tituloVentana="Hola";
private decimal pagoCliente=12.34;
private int contadorPersona;
private object Persona;
```

2.5.1.4.-Alcance

El tiempo de vida de una variable es el lugar de código donde podrá ser visualizada y utilizada dentro de la estructura del programa. No necesariamente todas las variables deben ser declaradas en el inicio de un programa, ya que algunas de ellas nunca necesitarán ser usadas a lo largo del mismo debido a su flujo, generalmente esto es un error conceptual grave.

Por ejemplo, si dentro de un bloque condicional se necesita utilizar una variable, solo en ese lugar, entonces ésta debería declararse dentro del bloque.

En el ejemplo en pseudocódigo se declaran **cliente** fuera y **lista**. La variable dentro, solo puede ser vista dentro del **if** en el que se declaro, las variables fuera pueden ser accedidas desde cualquier lado, ya que se declararon fuera del bloque.

Ejemplo:

```
gcli_cliente cliente = this.ejecuta(new gcli_cliente(), Modo.Nuevo);
if (cliente != null)
   {
      List<gcli_cliente> lista = new List<gcli_cliente>();
      lista.Add(cliente);
   }
```

2.5.1.5.-Case Sensitive

Case Sensitive significa que el compilador distinguirá entre los caracteres escritos en mayúscula y en minúscula. Es decir, que no será lo mismo A que a. Dos variables podrán llamarse igual, y solo distinguirse por su primera letra, o alguna de ellas, en mayúscula. Además la terminación de línea se realiza con "**;**".

2.5.1.6.-Comentarios

Comentar el código es una tarea fundamental del desarrollador, para permitir que su trabajo sea legible por otros desarrolladores y mantenible fácilmente. Existen convenciones sobre la forma en la que el código debe ser comentado. Independientemente de eso, la sintaxis de comentario entre los lenguajes, difiere de la siguiente manera:

En C# se puede comentar con "//", esto implica que toda la línea será de comentarios. El código que se encuentre entre "/*", y "*/" también será tomado por el compilador como comentario, por lo que lo salteara. Esta última forma suele ser cómoda a la hora de debugear, ya que es posible evitar el ingreso a líneas de código momentáneamente, sin tener que eliminarlas.

Ejemplo:

```
//Verifica si se cancelo el proceso.
If (this.ProcesoCancelado ¡= null)
{
  /* Identifica los */
  this.ProcesoCancelado(this, new EventArgs());
}
```

2.5.1.7.-Operadores Lógicos

Los operadores lógicos permiten armar una cláusula compuesta en la estructura de decisión. Permiten relacionar los distintos valores de las variables. Si las condiciones se cumplen, arrojará verdadero, de lo contrario, falso, ver tabla 9.

Operador	Descripción
&&	Operador Lógico Y
||	Operador Lógico O
!	Negación Lógica
==	Igual
!=	Distinto

Tabla 9: Operadores Lógicos

2.5.2.-Estructuras de Decisión IF

La estructura de decisión es la más utilizada en la programación, su objetivo es analizar una condición y dirigir el flujo del programa hacia un determinado cálculo, rutina, desvió, función, etc., según el resultado de la misma. El primer componente de esta estructura es la palabra IF.

IF "Condición" THEN entonces haga algo. De lo contrario, el flujo continuará por el statement dentro del bloque ELSE. Es posible anidar varios ELSE IF hasta que el programa encuentre el resultado correcto. Cabe resaltar que los statements pueden ser instrucciones simples o bloques de código.

Ejemplo:

```
if (elementoEvento.EstaSeleccionado == false)
{
    elementoEvento.EstaSeleccionado = true;
}
else
{
    elementoEvento.EstaSeleccionado = false;
}
```

2.5.3.-Estructuras de Decisión Case

Las estructuras de SWITCH permiten obtener un código más claro y más comprensible.

Su funcionamiento es el siguiente: La expresión es evaluada dentro del SWITCH y para cada posible valor que pueda tomar la condición existe un Estado CASE donde ésta se validará. En el caso de que un CASE sea verdadero, el bloque de código que le corresponde será ejecutado. Si ninguno de los Cases fuera evaluado como verdadero se puede usar la instrucción DEFAULT y se ejecutara el código correspondiente.

Es necesario declarar una sentencia BREAK por cada CASE para que el programa no siga evaluado todos los posibles casos luego de que uno se haya cumplido.

Ejemplo:

```
switch(titulo)
case "Empleado":
    return this.titulo;
break;
default
    return "Principal";
break;
```

2.5.4.-Arreglos

Los arreglos o arrays son estructuras muy comunes en los lenguajes de programación. El array, arreglo o vector es un conjunto de valores o variables del mismo tipo, que es almacenado en posiciones de memoria generalmente consecutivas. Pueden ser de 1 dimensión, o de varias (llamados matrices). La forma de acceder a los elementos es directa, es decir, a partir de un índice (TROELSEN, 2008, págs. 114-121).

Ejemplo:

```
string[] valores;
valores= new string[]{"1","2","3"};
```

2.5.5.-Estructuras de Iteración For

El uso de estructura de iteración (o "looping") es una práctica común, pues en muchos casos es necesario recorrer una determinada colección de datos, un conjunto de registros, valores de matrices, etc. El For permite realizar una serie predeterminada de ciclos, que permite, por ejemplo, recorrer los elementos de un array.

Ejemplo:

```
for (int valor = 1; i <= 10; i++)
{
    Console.WriteLine(i);
}
```

2.5.6.-Estructura de Iteración For/Each

Permite recorrer arreglos y colecciones. La idea de colecciones es muy útil, porque no se necesita saber cuántos registros existen en una colección de datos ya que el ForEach se encarga de ello.

Ejemplo:

```
string[] valores = new string[]{ "1","2","3"};
foreach (string valor in valores)
{
   predicate = p => p.strPerfil.Contains(valor);
   listaExpres.Add(this.obtenerPerfil(valor, claveEstadoPerfil));
}
```

2.5.7.-Estructura de Iteración While

El bucle While es ejecutado siempre asociado a una condición, o sea que en cada pasaje por el loop la condición es evaluada. Dependiendo de la situación, la condición debe ser colocada al inicio o al final del bucle. Si se la coloca al inicio será evaluada por primera

vez antes de entrar a los estados; si se la coloca al final, el bucle será ejecutado por lo menos una vez, ya que la evaluación se lleva a cabo luego de ejecutar los estados.

Ejemplo:

```
int i = 1;
while (i <= 10)
{
   Console.WriteLine(i);
   i++;
}
```

2.5.8.-Clases

En C# las clases se definen usando la palabra clave class seguida del nombre de la clase, el cuerpo de la clase estará encerrada entre un par de llaves, que por otra parte es la forma habitual de definir bloques de código en C# (FERGUSON, PATERSON, & BERES, 2003, págs. 205-206).

En el siguiente ejemplo se define una clase llamada Cliente que tiene dos campos públicos.

```
class Cliente
{
    public string Nombre;
    public string Apellidos;
}
```

Una vez definida la clase se puede agregar los elementos (o miembros) que se crean convenientes.
En el ejemplo anterior, para simplificar, se agregaron dos campos públicos, aunque también se podrían haber definido cualquiera de los miembros permitidos en las clases.

2.5.8.1.-Los miembros de una clase

Una clase puede contener cualquiera de estos elementos (miembros):

- Enumeraciones
- Campos
- Métodos (funciones o procedimientos)

- Propiedades
- Eventos

Las enumeraciones, se pueden usar para definir valores constantes relacionados, por ejemplo para indicar los valores posibles de cualquier "característica" de la clase.

Los campos son variables usadas para mantener los datos que la clase manipulará.

Los métodos son las acciones que la clase puede realizar, normalmente esas acciones serán sobre los datos que contiene. Los métodos pueden devolver valores como resultado de la "acción" realizada o también pueden devolver un valor void, que significa que realmente no devuelve nada.

Las propiedades son las "características" de las clases y la forma de acceder "públicamente" a los datos que contiene. Por ejemplo, se puede considerar que el nombre y los apellidos de un cliente son dos características del cliente.

Los eventos son mensajes que la clase puede enviar para informar que algo está ocurriendo en la clase.

2.5.9.-Características de los métodos y propiedades

Es importante identificar dos conceptos muy importantes en el lenguaje de C#, accesibilidad y ámbito son dos conceptos que están estrechamente relacionados. Aunque en la práctica tienen el mismo significado, ya que lo que representan es la "cobertura" o alcance que tienen los miembros de las clases e incluso de las mismas clases que se definieron (FERGUSON, PATERSON, & BERES, 2003, págs. 149-150).

2.5.9.1.-Ámbito

Es el alcance que la definición de un miembro o tipo puede tener. Es decir, cómo se puede acceder a ese elemento y desde dónde se puede accederlo.
El ámbito de un elemento de código está restringido por el "sitio" en el que se ha declarado (FERGUSON, PATERSON, & BERES, 2003, pág. 260). Estos sitios pueden ser:

- **Ámbito de bloque:** Disponible únicamente en el bloque de código en el que se ha declarado.

- **Ámbito de procedimiento**: Disponible únicamente dentro del procedimiento, (función o propiedad), en el que se ha declarado.
- **Ámbito de módulo**: Disponible en todo el código de la clase o la estructura donde se ha declarado.
- **Ámbito de espacio de nombres**: Disponible en todo el código del espacio de nombres.

2.5.9.2.-Accesibilidad

A los distintos elementos de nuestro código (ya sean clases o miembros de las clases) se les puede dar diferentes tipos de accesibilidad. Estos tipos de "acceso" dependerán del ámbito que se desea tengan, es decir, desde dónde se accederá.

Los modificadores de accesibilidad son:

- **public**: Acceso no restringido.
- **protected**: Acceso limitado a la clase contenedora o a los tipos derivados de esta clase.
- **interna**l: Acceso limitado al proyecto actual.
- **protected internal**: Acceso limitado al proyecto actual o a los tipos derivados de la clase contenedora.
- **private**: Acceso limitado al tipo contenedor.

Por ejemplo, se pueden declarar miembros privados a una clase, en ese caso, dichos miembros solamente se podrán acceder desde la propia clase, pero no desde fuera de ella.

En C# cuando se declara una variable sin indicar el modificador de accesibilidad, el ámbito será privado.

Una vez que se tiene una clase definida, lo único de lo que se dispone es de una especie de plantilla o molde a partir del cual se puede crear objetos en memoria.

Para crear esos objetos en Visual C# se puede hacer de dos formas, pero siempre será mediante la instrucción **new** que es la encargada de crear el objeto en la memoria y asignar la dirección del mismo a la variable usada en la parte izquierda de la asignación.

2.5.10.-Declarar variables en C#

Lo primero que se tiene que hacer es declarar una variable del tipo que se instanciara, esto se hace de la misma forma que con cualquier otro tipo de datos:

```
Cliente cli1;
```

Con esta línea de código lo que se indica a C# es que se usa una variable llamada cli1 para acceder a una clase de tipo **Cliente**. Esa variable, cuando llegue el momento de usarla, sabrá todo lo que hay que saber sobre una clase **Cliente**, pero hasta que no tenga una "referencia" a un objeto de ese tipo no podrá usarse.

La asignación de una referencia a un objeto Cliente se hace usando la instrucción **new** seguida del nombre de la clase:

```
cli1 = new Cliente();
```

A partir de este momento, la variable **cli1** tiene acceso a un nuevo objeto del tipo Cliente, por tanto podrá usar para asignarle valores y usar cualquiera de los miembros que ese tipo de datos contenga:

```
cli1.Nombre = "Antonio";
cli1.Apellidos = "Ruiz Rodríguez";
cli1.Saludar();
```

La otra forma de instanciar una clase es haciéndolo al mismo tiempo que se declara. En C# esto se hace como si uniera la declaración y la instanciación en una sola instrucción:

```
Cliente cli2 = new Cliente();
```

De esta forma se asignará a la variable **cli2** una referencia a un nuevo objeto creado en la memoria, el cual será totalmente independiente del resto de objetos creados con esa misma clase.

2.5.10.1.-Constructor

Cada vez que se crea un nuevo objeto en memoria se llama al constructor de la clase. En Visual C# Studio el constructor es una especie de método que se llama de la misma forma que la clase.

En el constructor de una clase se puede incluir el código que sea conveniente, aunque solamente se debería incluir el que realice algún tipo de inicialización, en caso de que no necesite realizar ningún tipo de inicialización, no es necesario definir el constructor, ya que el propio compilador lo hará. Esto es así porque todas las clases deben implementar un constructor, por tanto si no se define, lo hará el compilador de C#.

Si la clase **Cliente** tiene un campo para almacenar la fecha de creación del objeto se puede hacer algo como esto:

```
class Cliente
{
    public string Nombre;
    public string Apellidos;
    public DateTime FechaCreacion;

    public Cliente()
    {
        FechaCreacion = DateTime.Now;
    }
}
```

De esta forma se creara un nuevo **Cliente** y acto seguido comprobar el valor del campo FechaCreacion para saber la fecha de creación del objeto.

En los constructores también se puede hacer las inicializaciones que, por ejemplo permitan a la clase a conectarse con una Base de Datos, abrir un fichero o cargar una imagen gráfica, etc.

2.5.10.2.-Miembros compartidos (estáticos)

Por otro lado, los miembros compartidos, (o miembros estáticos), de una clase o tipo, son elementos que no pertenecen a una instancia o copia en memoria particular, sino que pertenecen al propio tipo y por tanto siempre están accesibles o disponibles, dentro del nivel del ámbito y accesibilidad que se le haya aplicado, y su tiempo de vida es el mismo que el de la aplicación.

2.5.10.3.-Parámetros

En C#, los métodos de una clase, (funciones), pueden recibir parámetros. Esos parámetros pueden estar explícitamente declarados, de forma que indiquen cuantos argumentos pasaran por el método, o se puede usar un array de parámetros opcionales, en los que se puede indicar un número variable de argumentos al llamar a la función que los define.

En C#, solo se pueden indicar parámetros en los métodos no en las propiedades. En el ejemplo siguiente de una función que recibe dos parámetros y devuelve la suma de esos dos parámetros:

```
int suma(int uno, int dos)
{
    return uno + dos;
}
```

Los dos parámetros definidos son de tipo int y el valor devuelto también es de ese tipo. El valor devuelto, es la suma de los dos parámetros, y ese resultado se devuelve mediante la instrucción return.

Para usar esta función se puede hacer algo como esto:

```
int total = suma(10, 25);
```

En este caso, los argumentos que se han pasado a la función son constantes literales, pero pueden usar cualquier tipo de argumento que genere un valor del tipo adecuado, en este caso int o cualquiera que produzca un valor implícitamente convertible en int. Por tanto se pueden usar expresiones, variables o constantes. El valor resultante será el que se pase a la función y lo verá como un valor simple.

2.5.11.-Genéricos

Los tipos genéricos se agregaron a la versión 2.0 del lenguaje C# y Common Language Runtime (CLR). Estos tipos agregan el concepto de parámetros de tipo a .NET Framework, lo cual permite diseñar clases y métodos que aplazan la especificación de uno o más tipos hasta que el código de cliente declara y crea una instancia de la clase o del método (MSDN Library C#, 2009). Por ejemplo, mediante la utilización de un parámetro de tipo genérico T, se puede escribir una clase única que otro código de

cliente puede utilizar sin generar el costo o el riesgo de conversiones en tiempo de ejecución u operaciones de conversión boxing, como se muestra a continuación:

```
using System;
using System.Collections.Generic;
using System.LINQ;
using System.LINQ.Expressions;
using System.Text;
using System.Data.LINQ;

namespace Generico.Comun
{
    public class CtlCatalogo<T> : TablaBase<T> where T : BaseCatalog, new()
    {

        public IQueryable<T> getCatalog(DataContext  dc)
        {
            return base.select( dc);
        }
```

Las clases y los métodos genéricos combinan reusabilidad, seguridad de tipos y eficacia de una manera que sus homólogos no genéricos no pueden. Los tipos genéricos se utilizan frecuentemente con las colecciones y los métodos que funcionan en ellas. La versión 2.0 de la biblioteca de clases de .NET Framework proporciona un nuevo espacio de nombres, System.Collections.Generic, que contiene varias clases nuevas de colección basadas en tipos genéricos (MSDN Library C#, 2009). Se recomienda que todas las aplicaciones cuyo destino es la versión 2.0 utilicen las nuevas clases de colección genéricas en lugar de sus homólogas no genéricas anteriores como ArrayList. En este caso se usa **List<T>.**

```
private void cargarControles()
{
   List<BaseCatalog> listaCat = new List<BaseCatalog>();
   listaCat.AddRange(new CtlCatalogo<estado alumno>().getBaseCatalog(this.dataContext));
   listaCat.AddRange(new CtlCatalogo< ocupacion>().getBaseCatalog(this.dataContext));
   listaCat.AddRange(new CtlCatalogo<sexo>().getBaseCatalog(this.dataContext));
   listaCat.AddRange(new CtlCatalogo<estado_civil>().getBaseCatalog(this.dataContext));
   this.dgvFiltros.DataSource = listaCatalogos;
}
```

También se pueden crear tipos y métodos genéricos personalizados para proporcionar soluciones generalizadas propias y diseñar modelos eficaces que tengan seguridad de tipos. El parámetro de tipo T se emplea en diversas ubicaciones donde generalmente se utilizaría un tipo concreto para indicar el tipo del elemento almacenado en la lista.

En una definición de tipo o método genérico, un parámetro de tipo es un marcador para un tipo especificado por un cliente al crear una instancia de una variable del tipo genérico. Una clase genérica, como **GenericList<T>**, no se puede utilizar tal cual,

porque no es realmente un tipo, sino el plano de un tipo. Para utilizar **GenericList<T>**, el código de cliente debe declarar y crear una instancia de un tipo construido especificando un argumento de tipo dentro de los corchetes angulares. El argumento de tipo para esta clase determinada puede ser cualquier tipo reconocido por el compilador. Se puede crear cualquier número de instancias del tipo construido, cada una de ellas con un argumento de tipo diferente (MSDN Library C#, 2009).

En cada una de estas instancias de **GenericList<T>,** cada aparición de T en la clase se sustituirá en tiempo de ejecución con el argumento de tipo. Mediante esta sustitución, se han creado tres objetos independientes y eficaces con seguridad de tipos utilizando una sola definición de clase. Este parámetro se utiliza de las formas siguientes:

```
public List<BaseCatalog> getBaseCatalog(DataContext _dc)
{
   List<BaseCatalog> list = new List<BaseCatalog>();
   IQueryable<T> iQueryable = this.getCatalog(_dc);
   foreach (T baseCatalog in iQueryable)
   {
      list.Add(baseCatalog);
   }
   return list;
}
public T obtenerPorId(DataContext  dc, T  entity)
{
   Expression<Func<T, bool>> predicate = p => (p.id == _entity.id);

   foreach (T  temp in base.select( dc, predicate))
   {
      return  temp;
   }

   return null;
}
public T obtenerPorNombre(DataContext  dc, String  name)
{
Expression<Func<T, bool>> predicate = p =>p.strClave.ToUpper().Equals(_name.ToUpper());

   foreach (T  temp in base.select( dc, predicate))
   {
      return  temp;
   }

   return null;
}
```

Para este ejemplo se puede ver que cualquier entidad o valor que se envié adquirirá el valor adecuado, ya sea el de Estado, Sexo o cualquier catalogo que se pase como entidad.

Los tipos genéricos proporcionan la solución a una limitación de las versiones anteriores de Common Language Runtime y del lenguaje C#, en los que se realiza una

generalización mediante la conversión de tipos a y desde el tipo base universal Object. Con la creación de una clase genérica, se puede crear una colección que garantiza la seguridad de tipos en tiempo de compilación.

2.5.12.-Interfaz

Son clases que no tienen implementación. Sirven como tipos de otras clases. Todos sus métodos son abstractos. Una clase puede implementar varias interfaces. Implican un contrato, una "protocolo" para que una clase y otra, que es utilizada por esta, se puedan comunicar.

Una interface define un contrato. Una clase o estructura que implementa una interface se adhiere a ese contrato. Una interface puede heredar de varias interfaces base, y una clase o estructura puede implementar varias interfaces (FERGUSON, PATERSON, & BERES, 2003, págs. 305-322).

Las interfaces pueden contener métodos, propiedades, eventos e indexadores. Por si misma no provee implementaciones para los miembros que define. Solamente especifica los miembros que deben ser suministrados por las clases que la implementan.

En C#, las implementaciones se definen siguiendo al nombre de la clase por un ":", en VB.NET, debe utilizarse Implements.

- Los métodos son implícitamente públicos
- Los métodos no tienen cuerpo (implementación)
- No se declaran "access modifiers"
- Estándar se les agrega el prefijo "I"

```
interface ISeleccionarElemento
{
    void Obtener();
    bool Publicar(string s);
}
```

Una clase puede implementar cero, una o más interfaces. También deben implementarse todos los métodos heredados de la interfaz.

Implementación de una Interfaz

```
interface ISeleccionarElemento
{
    void Obtener();
    bool Publicar(string s);
}
class Cliente : ISeleccionarElemento
{
    public void Obtener() { }
    public bool Publicar(string s) { }
}
```

2.5.13.-Delegados

Los eventos y los delegados están muy unidos en C#. De hecho no se pueden definir eventos sino se definen previamente un delegado; ya que por medio de ese delegado se podrán crear o asignar el método que se encargará de interceptar el evento.

Por esa razón se da una explicación de que son los delegados para de esta forma comprender mejor todo lo que se tiene hacer para definir eventos en nuestras clases y desde los objetos creados a partir de ellas asociarlos a un método.

2.5.13.1.-Definición Delegado

Algunas de las definiciones de la documentación de Visual Studio sobre los delegados:

"Un delegado es una clase que puede contener una referencia a un método. A diferencia de otras clases, los delegados tienen un prototipo (firma) y pueden guardar referencias únicamente a los métodos que coinciden con su prototipo." (MSDN Library C#, 2009).

De esta definición se obtiene que los delegados son clases especiales que pueden tener referencias a un método, y que ese método debe cumplir con el prototipo definido por el delegado. Y que esa referencia que contienen es como los punteros de otros lenguajes, pero que están enfocados a ser utilizados desde el punto de vista de .NET, es decir, a ser usados desde una perspectiva orientada a objetos.

Por tanto, cuando se define un delegado, se está definiendo la firma que debe tener una función (o método), con idea de que se pueda crear un puntero a dicha función, pero de

una forma controlada por el CLR de .NET. Por tanto, solo se admitirán punteros a funciones que concuerden con la definición que ha hecho el delegado (FERGUSON, PATERSON, & BERES, 2003, pág. 346).

En la definición del delegado System.EventHandler definido por .NET para los eventos, se muestra que ese delegado realmente está definiendo la "forma" que se debe de declarar el método que intercepte un evento basado en ese delegado.

Se analiza la definición tanto del delegado EventHandler como del evento TextChanged.

El delegado está definido de la siguiente forma:

```
public delegate void EventHandler(object sender, EventArgs e);
```

Si se quita public delegate, queda una definición de un método que bien podría incluirse en una interfaz, ya que lo que hace es indicar que ese método debe ser de tipo void (no devuelve ningún valor), y que tiene dos parámetros, el primero de tipo object y el segundo de tipo EventArgs.

Esta es la definición del evento TextChanged:

```
public event EventHandler TextChanged;
```

Esta es una declaración típica de C#, en la que el tipo de datos es EventHandler (el delegado, que al fin y al cabo es una clase de tipo especial), seguida de "la variable" que define el evento. Lo que se muestra aquí es que no aparece por ningún lado los parámetros que hay que usar, y esa es una de las características de los delegados, y posiblemente lo que complica más su entendimiento.

De la siguiente forma se asigna el método al evento del control:

```
this.textBox1.TextChanged += new
System.EventHandler(this.textBox1_TextChanged);
```

El evento se conecta por medio del constructor del delegado, el cual espera como parámetro un método, y lo que se le pasa es "un puntero al método", es decir, se le indica dónde está ese método. Este método debe cumplir con las especificaciones indicadas por el delegado, algo que se puede comprobar si se ve el código del método que se utilizará cuando el evento se produzca:

```
private void textBox1_TextChanged(object sender, EventArgs e)
{...}
```

CAPÍTULO 3. MVC (Modelo Vista Controlador) y ORM (Mapeo Relacional Objeto)

3.1.- MVC (Modelo Vista Controlador)

MVC (Model View Controller) o Modelo Vista Controlador es un framework metodológico que divide la implementación de una aplicación en tres roles: modelos, vistas y controladores (Modelo Vista Controlador, 2009).

Los **modelos** de una aplicación basada en MVC son los componentes responsables de mantener el estado. Normalmente el estado se guarda en una Base de Datos (por ejemplo: al tener una clase Producto que se usa para representar los datos de una tabla Productos en SQL).

Las **vistas** son los componentes responsables de mostrar la interfaz de usuario de la aplicación. Normalmente esta interfaz de usuario se crea a parte del modelo de datos. Por ejemplo: se podría crear una vista "Edit" que muestre textboxes, dropdowns y checkboxes dependiendo del estado actual de un objeto Producto).

Los **controladores** son los encargados de administrar la interacción con el usuario final, manipular el modelo, y en último lugar elegir una vista para ver la interfaz de usuario. En una aplicación MVC la vista sólo muestra la información - es el controlador el que administra y responde a las entradas de usuario y a las interacciones.

Uno de los beneficios de usar una metodología MVC es que ayuda a mantener una separación limpia entre modelos, vistas y controladores en la aplicación. Manteniendo una separación clara de conceptos hace que el testing de las aplicaciones sea mucho más fácil, ya que los contratos entre diferentes componentes de la aplicación están mejor definidos y articulados.

El patrón MVC también nos ayuda a realizar un desarrollo basado en test (Test Driven Development, TDD) - donde se implementan test unitarios automáticos, que definen y verifican los requerimientos del nuevo código, antes de que se escriba el código.

3.2.- ORM (Mapeo Relacional Objetos)

El avance de la tecnología en el desarrollo de software propicia el uso de nuevas técnicas y herramientas para acceder de forma efectiva a la Base de Datos desde un contexto

orientado a objetos, es necesaria una interfaz que traduzca la lógica de los objetos a la lógica relacional, esta interfaz se llama ORM (object-relational mapping) o "mapeo de objetos a Bases de Datos", y está formada por objetos que permiten acceder a los datos y que contienen en sí mismos el código necesario para hacerlo.

La principal ventaja que aporta el ORM es la reutilización, permitiendo llamar a los métodos de un objeto de datos desde varias partes de la aplicación e incluso desde diferentes aplicaciones. La capa ORM también encapsula la lógica de los datos (MEHTA, 2008, pág. 5).

La persistencia de la información es la parte más crítica en una aplicación de software. Si la aplicación está diseñada con orientación a objetos, la persistencia se logra por: serialización del objeto o almacenando en una Base de Datos. El modelo de objetos difiere en muchos aspectos del modelo relacional. La interface que une esos dos modelos se llama marco de mapeo relacional-objeto (ORM). Marcos de trabajo como Java o .Net han popularizado el uso de modelos de objetos (UML) en el diseño de aplicaciones dejando de lado el enfoque monolítico de una aplicación.

Las Bases de Datos más populares hoy en día son relacionales. Oracle, SQLServer, Mysql, Postgress son los DBMS más usados. En el momento de persistir un objeto, normalmente, se abre una conexión a la Base de Datos, se crea una sentencia SQL parametrizada, se asignan los parámetros y recién allí se ejecuta la transacción (Mapeo objeto-relacional, 2009).

Un buen porcentaje de desarrollo de software está dedicado al mapeo entre objeto y su correspondiente relación. Como se ve, la incongruencia entre los 2 modelos aumenta a medida que crece el modelo de objetos. Hay varios puntos por considerar:

- Carga perezosa.
- Referencia Circular.
- Caché.
- Transacciones.

El ORM debería resolver la mayoría de las cargas. Un buen ORM permite:

- Mapear clases a tablas: propiedad a columna, clase a tabla.

- Persistir objetos. A través de un método Orm.Save(objeto). Encargándose de generar el SQL correspondiente.
- Recuperar objetos persistidos. A través de un método objeto = Orm.Load(objeto.class, clave_primaria).
- Recuperar una lista de objetos a partir de un lenguaje de consulta especial. A través de un método.
- ListObjetos = Orm.Find("Objeto FROM MyObject WHERE Objeto.Propiedad=5"), o algo más complejo ListObjetos = Orm.Find("Objeto FROM MyObject WHERE Objeto.Relacion1.Relacion2.Propiedad2=5"), y el ORM transformará a través de varios joins de tablas.

3.2.1.-Diferencia entre el modelo relacional y el de objetos

Las tablas tienen atributos simples, o sea, tipo definidos previamente por los arquitectos del software. Por otro lado, un objeto tiene tanto atributos simples como aquellos definidos por el usuario, que en sí es otro objeto más.

La incongruencia entre el modelo relacional y el de objetos es la diferencia en la forma de representar atributos de los 2 modelos. Así en uno se tiene una representación tabular, mientras que en otro se tiene una representación jerárquica. La incongruencia entre la tecnología de objetos y la relacional, fuerza al programador a mapear el esquema de objetos a un esquema de datos. Los objetos deberían almacenarse en una Base de Datos relacional (DOUGLAS, 2009).

Ahora, una tabla mantiene relacionados los atributos que contiene. Un modelo de objetos tiene una jerarquía en árbol. Para ello se usa una capa extra muy fina pero suficiente para servir como un puente entre los 2 modelos. Para implementar esos mapeos, se necesita agregar código a los objetos de negocios, código que impacta en la aplicación.

Para la mayoría de las aplicaciones, almacenar y recuperar información implica alguna forma de interacción con una Base de Datos relacional. Esto ha representado un problema fundamental para los desarrolladores porque algunas veces el diseño de datos relacionales y los ejemplares orientados a objetos comparten estructuras de relaciones muy diferentes dentro de sus respectivos entornos.

Las Bases de Datos relacionales están estructuradas en una configuración tabular y los ejemplares orientados a objetos normalmente están relacionados en forma de árbol. Esta

'diferencia de impedancia' ha llevado a los desarrolladores de varias tecnologías de persistencia de objetos a intentar construir un puente entre el mundo relacional y el mundo orientado a objetos. El marco de trabajo Enterprise JavaBeans (EJB) proporciona uno de los muchos métodos para reducir esta distancia.

3.2.2.-Terminología

Modelo de objetos:

- Identidad de objeto. Propiedad por la que cada objeto es distinguible de otros aún si ambos tienen el mismo estado (o valores de atributos).
- Atributo. Propiedad del objeto al cual se le puede asignar un valor.
- Estado.
- Comportamiento. Es el conjunto de interfaces del objeto.
- Interface. Operación mediante la cual el cliente accede al objeto.
- Encapsulación. Es el ocultamiento de los detalles de implementación de las interfaces del objeto respecto al cliente.
- Asociación. Es la relación que existe entre dos objetos.
- Clase. Define como será el comportamiento del objeto y como almacenará su información. Es responsabilidad de cada objeto ir recordando el valor de sus atributos.
- Herencia. Especifica que una clase usa la implementación de otra clase, con la posible sobreescritura de la implementación de las interfaces.

Modelo relacional:

- Base de Datos. Conjunto de relaciones variables que describen el estado de un modelo de información. Puede cambiar de estado (valores) y puede responder preguntas sobre su estado.
- Relación variable (Tabla). Mantiene tuplas relacionadas a lo largo del tiempo, y puede actualizar sus valores. Esquematiza como están organizados los atributos para todas las tuplas que contiene.

- Tupla (fila). Es un predicado de verdad que indica la relación entre todos sus atributos.
- Atributo (columna). Identifica un nombre que participa en una relación y especifica el dominio sobre el cual se aplican los valores.
- Valor de atributo (valor de columna). Valor particular del atributo con el dominio especificado.
- Dominio. Tipos de datos simples.

3.2.3.-Mapeo de Objetos.

Un objeto está compuesto de propiedades y métodos. Como las propiedades, representan a la parte estática de ese objeto, son las partes que son persistidas. Cada propiedad puede ser simple o compleja. Por simple, se entiende que tiene algún tipo de datos nativos como por ejemplo entero, de coma flotante, cadena de caracteres. Por complejo se entiende algún tipo definido por el usuario ya sea objetos o estructuras.

En el modelo relacional, cada fila en la tabla se mapea a un objeto, y cada columna a una propiedad.

Normalmente, cada objeto de nuestro modelo, representa una tabla en el modelo relacional. Así que cada propiedad del objeto se mapea a cero, 1 o, más de 1 columna en una tabla. Cero columnas, pues se puede dar el caso de propiedades que no necesitan ser persistidas, el ejemplo más sencillo, es el de las propiedades que representan cálculos temporarios. Lo más común que sucede es que cada propiedad del objeto se mapea a una única columna, se debe tener en cuenta el tipo de datos. Una propiedad puede ser mapeada en más de una columna, por ejemplo, las propiedades de cálculos temporarios.

También se da el caso contrario de que en la tabla se tienen otros atributos que no se representan en el objeto real. Estos atributos suelen estar disponibles para el manejo de la concurrencia, auditorías, etcétera.

Cada columna de la tabla debería respetar el tipo de datos con su correspondiente en la propiedad del objeto. Aunque a veces, por optimización, es necesario algunos ajustes en la Base de Datos relacional. Una de las razones principales, es que la performance aumenta considerablemente si se trabaja con valores numéricos que con caracteres. En caso de no poder representarse el tipo de datos se debe tener en cuenta la pérdida de

información. Por ejemplo si almacena un valor de punto flotante como cadena de caracteres, al reconstruirlo en propiedad, es posible la pérdida de información.

Para acceder a las propiedades normalmente se usan métodos especiales llamados getters y setters, o mutadores y accesores.

3.2.3.1.-Identidad del Objeto.

Para distinguir cada fila de las otras, se necesita un identificador de objetos (OID) que es una columna más. Este identificador no es necesario en memoria, porque la unicidad del objeto queda representada por la unicidad de la posición de memoria que ocupa. El OID siempre representa la clave primaria en la tabla. Normalmente es numérico por razones de performance (DOUGLAS, 2009).

Solo se mapea la parte estática de un objeto. Así cada clase se mapea a una tabla. Cada objeto se mapea a una fila en la tabla. Cada propiedad del objeto se mapea por lo general a una columna. Existe una columna especial que identifica al objeto en la tabla llamada OID.

3.2.4.-Mapeo de Relaciones

Por relación se entiende asociación, herencia o agregación. Cualquiera sea el caso, para persistir las relaciones, se usan transacciones, ya que los cambios pueden incluir varias tablas. En el caso de que la transacción falle, se aumenta la probabilidad de éxito integridad referencial.

3.2.4.1.-Asociaciones

Una regla general para el mapeo es respetar el tipo de multiplicidad en el modelo de objetos, y en el modelo relacional. Así una relación 1-1 en el modelo de objetos, deberá corresponder a una relación 1-1 en el modelo relacional.

Las asociaciones, a su vez, están divididas según su multiplicidad y su navegabilidad. Según su multiplicidad, pueden existir asociaciones 1-1, 1-n, m-n. Según su navegabilidad, se tiene unidireccional o bidireccional. Se puede dar las seis combinaciones posibles. Una aclaración importante, es que en las Base de Datos relacionales, todas las asociaciones son bidireccionales, también es un factor de la incongruencia del modelos.

3.2.4.2.-Asociación Clave Foránea

Para mapear las relaciones, se usan los identificadores de objetos (OID). Estos OID se agregan como una columna más en la tabla donde se quiere establecer la relación. Dicha columna es una clave foránea a la tabla con la que se está relacionada. Así, queda asignada la relación. Recordar que las relaciones en el modelo relacional son siempre bidireccionales. El patrón se llama Foreign Key Mapping.

3.2.5.-Agregación y Composición

Una asociación es una relación débil e independiente entre 2 objetos. Una agregación es una relación más fuerte que una asociación pero aún independiente. Una composición es a la vez una relación fuerte y dependiente entre 2 objetos.

Con esto se define que tanto una asociación, agregación o composición se mapea en el modelo relacional como una relación.

La agregación normalmente se mapea como una relación n-m, o sea que hay una tabla auxiliar para mapear la relación. La composición puede mapearse como una relación 1-n. Donde los objetos compuestos mantienen una relación con el objeto compositor. A nivel relacional, indica que la tabla de los objetos compuestos tienen una columna con la clave foránea al objeto que los compuso. En el modelo objetos, tanto las agregaciones como las composiciones se corresponden con un array o una colección de objetos.

3.2.6.-Herencia

Las asociaciones funcionan en ambos modelos, objeto y relacional. Para el caso de la herencia se presenta el problema que las Base de Datos relacionales no la soportan. Así que se tiene que modelar como se verá la herencia en el modelo relacional (DOUGLAS, 2009). Una regla a seguir es que se debe minimizar la cantidad de joins posibles. Existen 3 tipos de mapeos principales: modelar la jerarquía a una sola tabla, modelar la jerarquía concreta en tablas, modelar la jerarquía completa en tablas. La decisión estará basada en el performance y, en la escalabilidad del modelo.

3.2.6.1.-Mapeo de la Jerarquía a una tabla.

Se mapean todos los atributos, de todas las clases del árbol de herencia en una única tabla. En el mapeo, se agregan 2 columnas de información oculta: la clave primaria de la tabla OID y, el tipo de clase que es cada registro. El tipo de clase se resuelve con 1 columna carácter o numérica entera. Para los casos más complejos se necesita de varias columnas booleanas (si/no).

Por ejemplo, la jerarquía Persona (que es abstracta), Cliente y Empleado quedarían en una única tabla llamada persona (es recomendable colocarle el nombre de la raíz de la estructura). A la tabla se le agrega el tipo de clase que es, así, se tiene la columna TipoPersona donde C será cliente, E empleado, D para aquellos que son clientes y empleados al mismo tiempo.

Las ventajas son:

- Aproximación simple.
- Cada nueva clase, simplemente se agregan columnas para datos adicionales.
- Soporta el polimorfismo cambiando el tipo de fila.
- El acceso a los datos es rápido porque los datos están en una sola tabla.
- El reporte es fácil porque los datos están en una tabla.

Las desventajas son:

- Mayor acoplamiento entre las clases. Un cambio en una de ellas puede afectar a las otras clases ya que comparten la misma tabla.
- Se desperdicia espacio en la Base de Datos (por lo tanto disminuye performance), para aquellas columnas en las que son de las clases derivadas.
- La tabla crece rápidamente a mayor jerarquía.

3.2.6.2.-Mapeo de cada clase concreta a una tabla.

Cada clase concreta es mapeada a una tabla. Cada tabla incluye los atributos heredados más los implementados en la clase. La clase base abstracta entonces, es mapeada en cada tabla de las derivadas.

Ventajas:

- Reportes fáciles de obtener ya que los datos necesarios están en una sola tabla.
- Buena performance para acceder a datos de un objeto.

Desventajas:

- Pobre escalabilidad, si se modifica la clase base, se debe modificar en todas las tablas de las clases derivadas para reflejar ese cambio. Por ejemplo, si a la clase Persona se le agrega el atributo de EstadoCivil, se debe agregarlo en las tablas Cliente y en Empleado.
- Actualización compleja, si un objeto cambia su rol, se le asigna un nuevo OID y se mueven los datos a la tabla correspondiente.
- Se pierde integridad en los datos, para el caso en que el objeto tenga los 2 roles. Por ejemplo Cliente y Empleado a la vez.

3.2.6.3.-Mapeo de cada clase a su propia tabla.

Se crea una tabla por cada clase de la herencia, aún la clase base abstracta. Se agregan también las columnas para el control de la concurrencia o versión a cualquiera de las tablas.

Cuando se necesita leer el objeto heredado se unen (join) las 2 tablas de la relación o se leen las 2 tablas en forma separada. Las claves primarias de todas las tablas heredadas, será la misma que la tabla base. A su vez, también serán claves foráneas hacia la tabla base.

Para simplificar las consultas, a veces será necesario agregar una columna en la tabla base indicando los subtipos de ese elemento, o, agregando varias columnas booleanas. El mismo efecto se logra, y mucho más eficiente, a través de vistas.

Ventajas:

- Fácil de entender, porque es un mapeo uno a uno.
- Soporta muy bien el polimorfismo, ya que tiene almacenado los registros en la tabla correspondiente.

- Fácil escalabilidad, se pueden modificar atributos en la superclase que afectan a una sola tabla. Agregar subclases es simplemente agregar nuevas tablas.

Desventajas:

- Hay muchas tablas en la Base de Datos.
- Menor performance, pues se necesita leer y escribir sobre varias tablas.
- Reportes rápidos difíciles de armar, a menos que se tengan vistas.

3.2.7.-Persistencia

Al existir una incongruencia entre el mundo orientado a objetos y las Base de Datos relacionales. Para reducir esa incongruencia se recurre a una capa auxiliar que mapeará entre los 2 mundos, adaptándolo según las especificaciones hechas (MEHTA, 2008, págs. 12-13). Esa capa auxiliar se denomina ORM, object relational mapping.

Los ORM nacieron a mediados de la década del 90, se hicieron masivos a partir de uso de Java. Por esa razón, los frameworks más populares hoy en día en .Net son adaptaciones del modelo pensado para Java.

3.2.7.1.-Patrón CRUD

Acrónimo de Create-Read-Update-Delete. Esta es una capa de acceso que describe que cada objeto debe ser creado en la Base de Datos para que sea persistente. Una vez creado, la capa de acceso debe tener una forma de leerlo para poder actualizarlo o simplemente borrarlo (MEHTA, 2008, págs. 4-5).

Teóricamente el borrado de objetos debería quedar a cargo de la misma Base de Datos. Pero un recolector de objetos “basura” (garbage collector) en una Base de Datos gigante afecta en gran medida la performance. Por ello es que la tarea de borrado queda delegada al programador.

3.2.7.2.-Caché

Hay un conjunto de datos dinámicos que son relevantes a todos los usuarios del sistema, y por lo tanto accedido con más frecuencia. Las aplicaciones empresariales de sincronización de caché normalmente necesitan escalarse para manejar grandes cargas

transaccionales, así múltiples instancias pueden procesar simultáneamente. Es un problema serio para el acceso a datos desde la aplicación, especialmente cuando los datos involucrados necesitan actualizarse dinámicamente a través de esas instancias. Para asegurar la integridad de datos, la Base de Datos comúnmente juega el rol de árbitro para todos los datos de la aplicación. Es un rol muy importante dado que los datos representan la proporción de valor más significante de una organización. Desafortunadamente, este rol también no está fácilmente distribuido sin introducir problemas significantes, especialmente en un entorno transaccional.

Es común para la Base de Datos usar replicación para lograr datos sincronizados, pero comúnmente ofrece una copia offline del estado de los datos más que una instancia secundaria activa. Es posible usar Base de Datos que puedan soportar múltiples instancias activas, pero se pueden volver caras en cuanto a performance y escalabilidad, debido a que introducen el bloqueo de objetos y la latencia de distribución. La mayoría de los sistemas usan una única Base de Datos activa, con múltiples servidores conectada directamente a ella, soportando un número variables de clientes (DOUGLAS, 2009).

En esta arquitectura, la carga en la Base de Datos incrementará linealmente con el número de instancias de la aplicación en uso, a menos que se emplee alguna caché. Pero implementando un mecanismo de caché en esta arquitectura puede traer muchos problemas, incluso corrupción en los datos, porque la caché en el servidor 1 no sabrá sobre los cambios en el servidor 2.

3.2.7.3.-Carga de las relaciones

Uno de los problemas que surgen después del mapeo es si siempre se cargan todos los objetos relacionados a uno principal. Lo más posible es que no, porque las redes de relaciones tienden a ser más compleja y la cadena de relación tiende a ser más larga en la vida real. Supóngase que por defecto se carga cada relación de un objeto. En el caso de una gran Base de Datos se volverá grandísimo y habrá pérdida de performance. La solución para este problema es conocida como “Carga retardada” de las relaciones, y se implementa por algún tipo de objetos con “Patrón Proxy” que lanzan la carga cuando se acceden (punteros inteligentes en C++).

Para lograrlo, el objeto tiene un método acceso (“Get”) cuyo único propósito es proveer el valor de un atributo simple, que verifica a ver si el atributo ha sido inicializado y si no es así lo lee desde la Base de Datos.

Otro uso común de cargas retardadas es la generación de reporte y objetos que se dan como resultados de una búsqueda, casos en los cuales se necesita solo un subconjunto de datos del objeto.

Lo mismo sucede para aquellos campos grandes y menos usados. Por ejemplo si se almacena la foto de una Persona ocupará alrededor de 100k mientras que el resto de los atributos no llegan, en total, a 1k; y, raramente son accedidas.

Pero también, habrá veces en la que la "Carga Directa" de las relaciones se prefiera.

Con ella, cada vez que se cargue un objeto, se desearía tener algunas de sus relaciones cargadas sin necesidad de volver a consultar la Base de Datos.

3.2.7.4.-Transacción

Los datos almacenados en una Base de Datos necesitan ser protegidos por una transacción. Esto permite múltiples inserciones, modificaciones y borrados con la seguridad de que todo o se ejecuta o falla, como si fuera una sola entidad coherente.

Las transacciones también pueden ofrecer protección de concurrencia; el bloqueo pesimista de tuplas mientras los usuarios están trabajando en ellos, evita que otros usuarios comiencen con sus cambios (DOUGLAS, 2009).

Sin embargo, el mecanismo de transacción de la Base de Datos tiene algunas limitaciones:

- Cada transacción requiere una sesión separada, para permitir que los usuarios abran ventanas relacionadas a su trabajo requeriría de licencias extras o que las ventas estén limitada a acceso de solo lectura.

- Una alta aislación en la transacción y bloqueo basado en páginas puede evitar que otros usuarios accedan a los datos que deberían estar legítimamente permitidos.

Una alternativa es que los datos sean puestos en un buffer en la capa de persistencia, con todas las escrituras que se harán hasta que el usuario lo confirme. Esta transacción de Base de Datos se necesita solo durante la operación de escritura en masa (bulk operation), permitiendo ser compartida entre múltiples ventanas. Esto requiere un esquema de bloqueo optimista donde las tuplas son chequeadas mientras se escriben.

Obviamente, esta escritura en masa corre protegida por la integridad referencial. Esas restricciones especifican los requerimientos lógicos según el caso: la tupla debe existir antes de que se la relacione y, las tuplas relacionadas a otra deben ser borradas antes de que se borre la tupla objetivo. Esto se logra mediante un mecanismo en la Base de Datos que mantiene correctamente las dependencias entre las tuplas.

3.2.7.5.-Concurrencia

La capa de persistencia debe permitir que múltiples usuarios trabajen en la misma Base de Datos y proteger los datos de ser escritos equivocadamente. También es importante minimizar las restricciones en su capacidad concurrente para verla y acceder.

La integridad de datos es un riesgo cuando 2 sesiones trabajan sobre la misma tupla: la pérdida de alguna actualización está asegurada. También se puede dar el caso, cuando una sesión está leyendo los datos y la otra los está editando: una lectura inconsistente es muy probable.

Hay dos técnicas principales para el problema: bloqueo pesimista y bloqueo optimista. Con el primero, se bloquea todos acceso desde que el usuario empieza a cambiar los datos hasta que COMMIT la transacción. Mientras que en el optimista, el bloqueo se aplica cuando los datos son aplicados y se van verificando mientras los datos son escritos.

CAPÍTULO 4. Arquitectura LINQ

LINQ(Language Integrated Query) es un ORM(object relational mapping, mapeador de objetos relacionales) que viene con la versión del .NET Framework, y permite modelar Bases de Datos relacionales con clases de .NET. Una de las funciones es consultar Bases de Datos, así como actualizar/añadir/borrar datos de ellas.

Una consulta es una expresión que recupera datos de un origen de datos. Las consultas normalmente se expresan en un lenguaje de consultas especializado. A lo largo del tiempo se han ido desarrollando lenguajes diferentes para los distintos tipos de orígenes de datos, como SQL para las Bases de Datos relacionales y XQuery para XML. LINQ simplifica esta situación, proporcionando un modelo coherente para trabajar con datos de distintos tipos de formatos y orígenes de datos, ver figura 41. En una consulta LINQ, siempre se trabaja con objetos. Se utilizan los mismos modelos de codificación básicos para consultar y transformar datos de documentos XML, Bases de Datos SQL, conjuntos de datos ADO.NET, colecciones .NET y cualquier otro formato para el que haya disponible un proveedor LINQ (KUMAR, 2007, págs. 5-8).

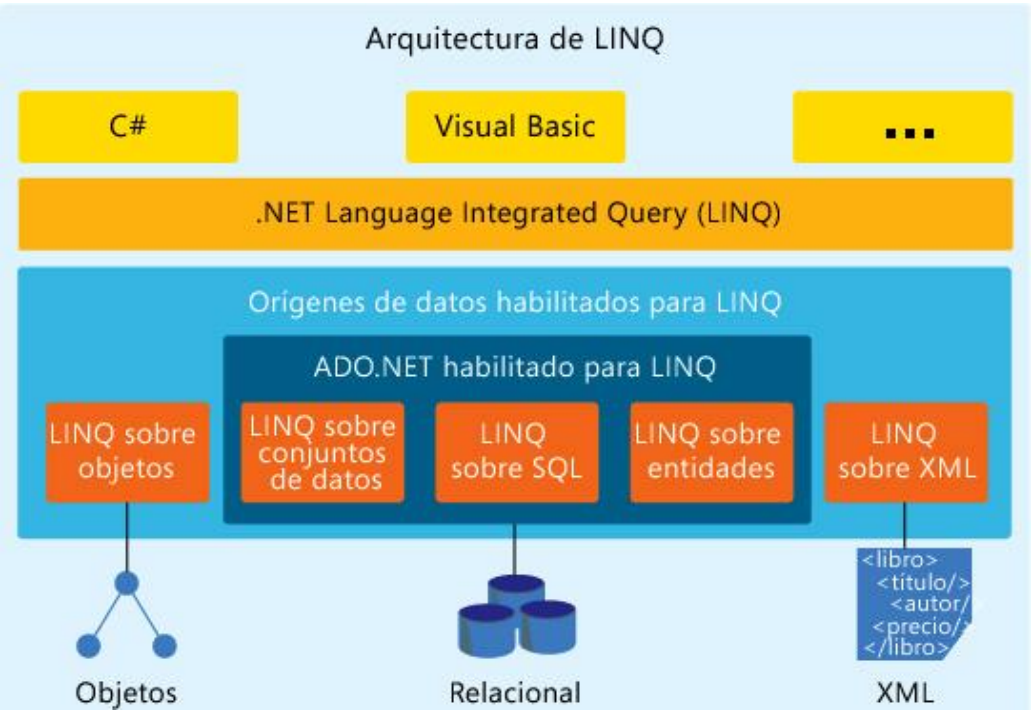

Figura 41: Modelo .Net FrameWork 3.5 con LINQ

Consultas integradas en los lenguajes (Language Integrated Query - LINQ). Es una característica que viene con Visual Studio 2008 que extiende las capacidades de consultas, usando C# y Visual Basic.

Se utiliza el término consultas integradas en los lenguajes para indicar que las consultas son una característica integrada del lenguaje de programación principal del desarrollador (por ejemplo C#, Visual Basic). Las consultas integradas en los lenguajes permiten que las expresiones de consulta se beneficien de los metadatos ricos, verificación de sintaxis en tiempo de compilación, tipado estático y ayuda IntelliSense que antes estaban disponibles solo para el código. Las consultas integradas en los lenguajes también hacen posible aplicar una única facilidad declarativa de propósito general a toda la información en memoria, y no solo a la información proveniente de fuentes externas.

Las consultas integradas en los lenguajes .NET definen un conjunto de operadores de consulta estándar de propósito general que hacen posible que las operaciones de recorrido, filtro y proyección sean expresadas de una manera directa pero declarativa en cualquier lenguaje de programación. Los operadores de consulta estándar permiten aplicar las consultas a cualquier fuente de información basada en **IEnumerable<T>**. LINQ permite que terceros fabricantes aumenten el conjunto de operadores de consulta estándar, añadiendo los operadores de dominio específico que sean apropiados para el dominio o la tecnología de destino. Más importante aún es que terceros fabricantes también pueden reemplazar los operadores de consulta estándar con sus propias implementaciones que ofrezcan servicios adicionales como la evaluación remota, traducción de consultas, optimización, etc. Al adherirse a los convenios del patrón LINQ, tales implementaciones gozarán de la misma integración en los lenguajes y soporte de herramientas que los operadores de consulta estándar (KUMAR, 2007, págs. 5,7).

La extensibilidad de la arquitectura de consultas es aprovechada por LINQ para ofrecer implementaciones que operan sobre datos XML y SQL. Los operadores de consulta sobre XML (XLINQ) utilizan una facilidad de XML en memoria interna eficiente y fácil de usar para ofrecer funcionalidad XPath/XQuery dentro del lenguaje de programación huésped (KUMAR, 2007, págs. 7-8). Los operadores de consulta sobre datos relacionales (DLINQ) se apoyan en la integración de definiciones de esquemas basadas en SQL en el sistema de tipos del CLR. Esta integración ofrece un fuerte control de tipos sobre los datos relacionales, a la vez que mantiene la potencia expresiva del modelo relacional y el rendimiento de la evaluación de las consultas directamente en el almacén de datos subyacente.

Todas las operaciones de consulta LINQ se componen de tres acciones distintas:

1. Obtención del origen de datos.
2. Creación de la consulta.
3. Ejecución de la consulta.

En LINQ, la ejecución de la consulta es distinta de la propia consulta; en otras palabras, no se recuperan datos con la simple creación de la variable de consulta (MSDN Library LINQ, 2009).

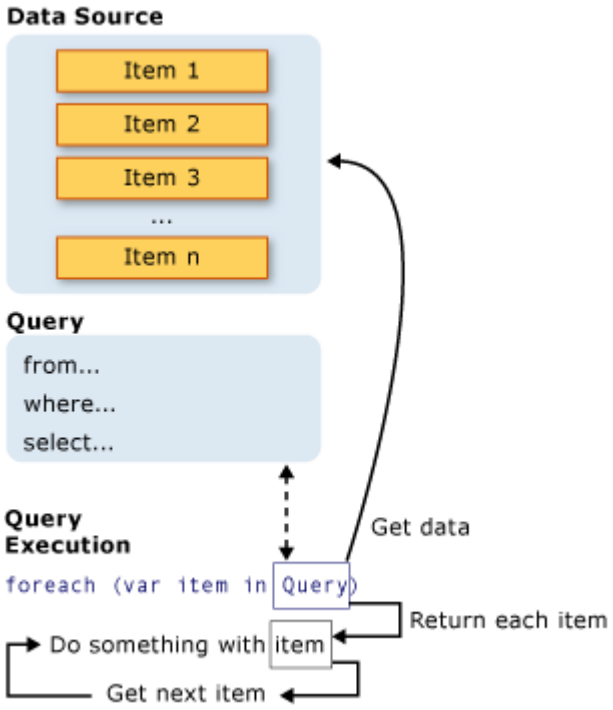

Figura 42: Interfaz Genérica IEnumerable

4.1.-El origen de datos

En la figura 42 el origen de datos es una matriz, se admite implícitamente la interfaz genérica **IEnumerable(T).** Este hecho implica que se puede consultar con LINQ. Una consulta se ejecuta en una instrucción **foreach** y **foreach** requiere **IEnumerable** o **IEnumerable(T)**. Los tipos que admiten **IEnumerable(T)** o una interfaz derivada, como la genérica **IQueryable(T)**, se conocen como *tipos que se pueden consultar*.

Un tipo que se puede consultar no requiere ninguna modificación o tratamiento especial para servir como origen de datos LINQ. Si los datos de origen aún no están en memoria como tipo que se puede consultar, el proveedor LINQ debe representarlos como tales. Por ejemplo, LINQ to XML carga un documento XML en un tipo XElement que se puede consultar:

```
// Crear data source de un documento XML.
// using System.Xml.LINQ;
XElement contacts = XElement.Load(@"c:\ListaContactos.xml");
```

Con LINQ to SQL, primero se crea una asignación relacional de objetos en tiempo de diseño, ya sea manualmente o mediante el Diseñador relacional de objetos (Diseñador R/O). Después, se escriben las consultas en los objetos y, en tiempo de ejecución, LINQ to SQL controla la comunicación con la Base de Datos. En el ejemplo siguiente, Cliente representa una tabla concreta de la Base de Datos y **Table<Cliente>** admite la interfaz genérica **IQueryable(T)**, que se deriva de **IEnumerable(T)**.

```
// Crea un data source de una Base de Datos de Sql.
// using System.Data.LINQ;
DataContext db = new DataContext(@"c:\ControlEscolar.mdf");
```

Un origen de datos LINQ es cualquier objeto que admite la interfaz genérica IEnumerable(T) o una interfaz que herede de ella.

4.2.-La consulta.

La consulta especifica qué información se va a recuperar de uno o varios orígenes de datos. Una consulta también especifica cómo debería ordenarse, agruparse y darse forma a esa información antes de ser devuelta. Una consulta se almacena en una variable de consulta y se inicializa con una expresión de consulta. Para simplificar la escritura de consultas, C# incluye nueva sintaxis de consulta.

La consulta del ejemplo anterior devuelve todos los números pares de la matriz de enteros. La expresión de consulta contiene tres cláusulas: from, where y select. (Si está familiarizado con SQL, habrá observado que el orden de las cláusulas se invierte respecto al orden de SQL.) La cláusula from especifica el origen de datos, la cláusula where aplica el filtro y la cláusula select especifica el tipo de los elementos devueltos.

Aquí, lo importante es que, en LINQ, la propia variable de consulta no realiza ninguna acción ni devuelve datos. Simplemente almacena la información necesaria para generar los resultados cuando la consulta se ejecute posteriormente.

4.3.-Ejecución de Consulta

La ejecución de consultas toma un plan de evaluación, lo ejecuta y devuelve su respuesta a la consulta. En los siguientes temas se muestran los tipos de ejecución.

4.3.1.-Ejecución diferida

Como se ha explicado anteriormente la variable de consulta sólo almacena los comandos de la consulta. La ejecución real de la consulta se aplaza hasta que se procese una iteración en la variable de consulta, en una instrucción foreach. Este concepto se conoce como ejecución diferida y se muestra en el ejemplo siguiente:

```
// Query execution.

foreach (int num in numQuery)

{    Console.Write("{0,1} ", num);

}
```

La instrucción foreach es también donde se recuperan los resultados de la consulta. Por ejemplo, en la consulta anterior, la variable de iteración num contiene cada valor (de uno en uno) en la secuencia devuelta.

Dado que la propia variable de consulta nunca contiene los resultados de la consulta, se puede ejecutar tantas veces como se desee. Por ejemplo, puede que una aplicación independiente actualice continuamente una Base de Datos. En su aplicación, podría crear una consulta que recuperase los datos más recientes, y podría ejecutarla repetidamente cada cierto tiempo para recuperar cada vez resultados diferentes.

4.3.2.-Forzar la ejecución inmediata

Las consultas que realizan funciones de agregación en un intervalo de elementos de origen primero deben recorrer en iteración dichos elementos. Algunos ejemplos de esas

consultas son Count, Max, Average y First. Se ejecutan sin una instrucción foreach explícita porque la propia consulta debe utilizar foreach para devolver un resultado. Debe saber también que estos tipos de consultas devuelven un solo valor, no una colección **IEnumerable**. La consulta siguiente devuelve un recuento de los números pares de la matriz de origen:

```
var evenNumQuery =
    from num in numeros
    where (num % 2) == 0
    select num;
int evenNumCount = evenNumQuery.Count();
```

Para forzar la ejecución inmediata de cualquier consulta y almacenar en memoria caché sus resultados, puede llamar al método ToList(TSource) o ToArray (TSource).

4.4.-Integración con SQL.

Las consultas de LINQ se traducen a consultas de SQL, para ser ejecutadas en la Base de Datos y los resultados son de nueva cuenta traducidos a objetos para ser usados por la aplicación. LINQ usa la misma conexión y transacción que ofrece .NET Framework, para conectar a la Base de Datos y manipular las transacciones. Se puede hacer uso de Intellisense para validar las consultas de LINQ.

Para representar la relación de los datos, se necesitan crear las clases para las entidades. Para crear las clases de la entidad, se tiene que especificar los atributos de la Clase. Los objetos de la entidad deben de tener propiedades similares a los objetos de la Base de Datos.

4.5.-Tipos anónimos

Utilice un tipo anónimo en una expresión de consulta cuando se cumplan las dos condiciones siguientes:

- Sólo desea devolver algunas propiedades de cada elemento de origen.
- No tiene que almacenar los resultados de la consulta fuera del ámbito del método en el que se ejecuta la consulta.

Si sólo desea devolver una propiedad o campo de cada elemento de origen, puede utilizar simplemente el operador de punto en la cláusula select.

Hay que tener en cuenta que el tipo anónimo utiliza los nombres del elemento de origen para sus propiedades si no se especifican nombres.

4.6.-Inicializador de Objetos

Los tipos del .NET Framework tienen claramente definido el uso de propiedades. Cuando se instancia y usan clases nuevas, es muy común escribir código como este:

```
Persona persona = new Persona();
   persona.Nombre = “Juan”;
   persona.APaterno = “Pérez”;

   persona.AMaterno = “Hernández”
   persona.Edad = 28;
```

Con los compiladores para C# y VB, se puede usar una característica también llamada “inicializadores de objetos” que nos permite reducir el código, como se muestra en el ejemplo de abajo:

```
Persona persona = new Person { Nombre=“Juan”, APaterno
=“Pérez”, AMaterno=”Hernández”, Edad=28 };
```

El compilador generará automáticamente el código de inicialización necesario guardando el mismo significado semántico que como en el primer ejemplo. Aunque no sólo nos permite inicializar un objeto con valores simples, sino también nos permite poner valores más complejos como propiedades anidadas. Por ejemplo, si en la clase Persona también se tiene una propiedad llamada “Direccion” del tipo “Direccion”. Se puede escribir el siguiente código para crear un objeto del tipo Persona y poner sus propiedades de la siguiente forma:

```
Persona persona = new Persona {
    Nombre = “Juan”,
    APaterno = “Pérez”
    AMaterno = “Hernández”
    Edad = 28,
    Direccion = new Direccion {
      Calle = “Av. Hidalgo”,
      Ciudad = “Tlahuelilpan”,
      Estado = “Hidalgo”,
      Cp = 42780
    }
  };
```

Los inicializadores de objetos simplifican algunos pasos, y hacen mucho más claro añadir objetos a colecciones de objetos. Por ejemplo, si se quiere añadir tres personas a una lista genérica de tipo Persona. Se puede escribir de la siguiente forma:

```
List<Persona> gente = new List<Persona>();
gente.Add( new Persona { Nombre = “Juan”, APaterno = “Pérez”, Edad = 18 } );
gente.Add( new Persona { Nombre = “Lucia”, APaterno = “Escobar”, Edad = 30 } );
gente.Add( new Persona { Nombre = “Pedro”, APaterno = “Sánchez”, Edad = 20 } );
```

Los compiladores para C# y VB 3.0 permiten mucho más, y soportan los “inicializadores de colecciones” que permiten evitar tener varias llamadas al método Add, y ahorrar aún más código:

```
List<Persona> gente = new List<Persona> {
      new Persona { Nombre = “Juan”, APaterno = “Pérez”, Edad = 18 },
      new Persona { Nombre = “Lucia”, APaterno = “Escobar”, Edad = 30 },
      new Persona { Nombre = “Pedro”, APaterno = “Sánchez”, Edad = 20 }
   };
```

Cuando el compilador encuentra un código como este último, generará el código necesario para añadir los diferentes elementos a la colección.

4.7.-Extensiones

Los métodos de extensión permiten a los desarrolladores añadir nuevo métodos al contrato público de un tipo ya existente en el CLR, sin tener que recompilar el tipo original. Los métodos de extensión permiten mezclar la flexibilidad del soporte "duck typing" de los lenguajes dinámicos de hoy con el rendimiento y la validación en tiempo de compilación de los lenguajes fuertemente tipados (KUMAR, 2007, págs. 15-16).

Los métodos de extensión posibilitan la aparición de una gran variedad de escenarios, y ayudan a hacer posible el poder del framework en LINQ. En el ejemplo siguiente se muestra algo de los métodos de extensión

```
if (enUsuario != null)
{
    baseEntity.gseg_usuario = enUsuario;
    resultado = adminEmpleado.insertaEmpleado(this.dataContext,
this.baseEntity);
}
else
{
    resultado = adminEmpleado.insertaEmpleado(this.dataContext,
this.baseEntity);
}
```

En el código siguiente se muestra la extensión del código insertaEmpleado.

```
public class CtlEmpleado : IValidaEntidad<gemp_empleado>, IControl
{
    public bool insertaEmpleado(DataContext _dataContext, gemp_empleado _empleado)
    {
        this.lastMessage = string.Empty;
        this.lastException = null;
        try
        {
            if (this.validaEntidad( empleado))
            {
                return true;
            }
        }
        catch (Exception _ex)
        {
            this.lastException =  ex;
        }
        return false;
    }
```

4.8.-Expresiones

Las expresiones de consulta se escriben en una sintaxis de consulta declarativa que se dio a conocer en C# 3.0. Al usar la sintaxis de consulta, con apenas código se pueden realizar operaciones complejas de filtrado, clasificación y agrupación en orígenes de datos. Los mismos patrones de expresión de consulta básicos se usan para consultar y transformar datos de Bases de Datos de SQL, conjuntos de datos de ADO.NET, secuencias y documentos XML, y colecciones de .NET (MSDN Library LINQ, 2009).

4.8.1.-Expresiones Lambda

Las expresiones lambda no se utilizan directamente en sintaxis de consulta, sino que se utilizan en llamadas a métodos, y las expresiones de consulta pueden contener llamadas a métodos. De hecho, algunas operaciones de consulta sólo se pueden expresar en sintaxis de método.

Algunos de los operadores de consulta estándar más frecuentemente utilizados poseen una sintaxis de palabras clave dedicadas del lenguaje C# y Visual Basic que les permite ser llamados como parte de una expresión de consulta. Una expresión de consulta constituye una forma diferente de expresar una consulta, más legible que su equivalente basado en método. Las cláusulas de las expresiones de consulta se traducen en llamadas a los métodos de consulta en tiempo de compilación (KUMAR, 2007, págs. 15-17).

4.8.2.-Árboles de Expresiones

Los árboles de expresiones representan el código de nivel del lenguaje en forma de datos. Los datos se almacenan en una estructura con forma de árbol. Cada nodo del árbol de expresión representa una expresión, por ejemplo, una llamada al método o una operación binaria, como x < y.

En la figura 43 se muestra un ejemplo de una expresión y su representación en forma de un árbol de expresión. También se muestran los diferentes tipos de los nodos del árbol de expresión.

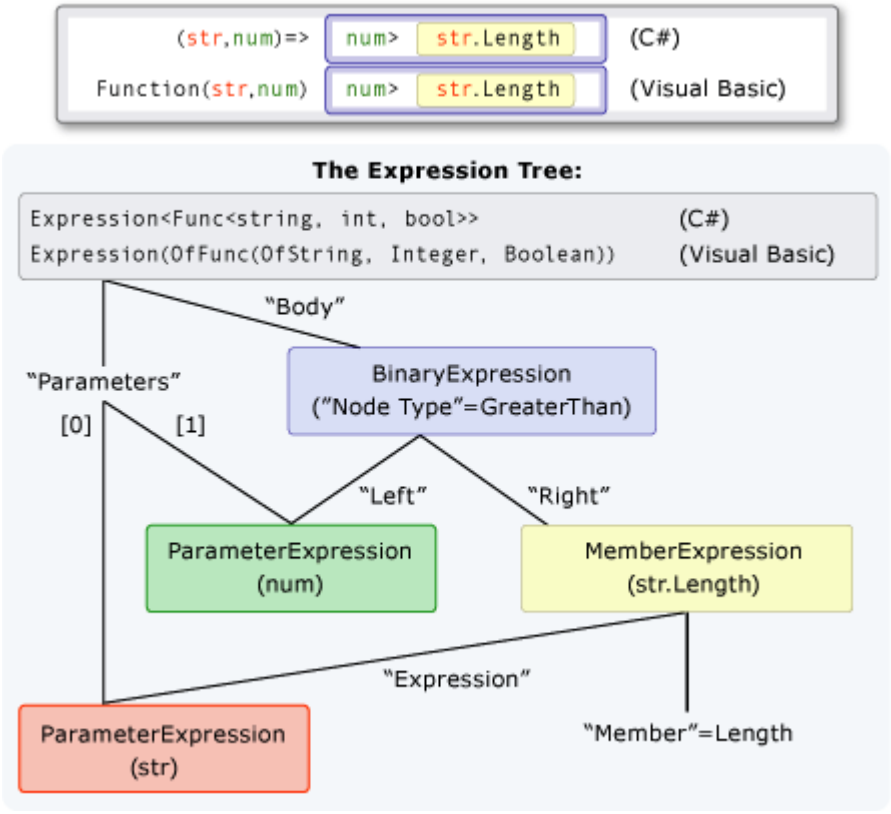

Figura 43: Arboles de Expresión

CAPÍTULO 5. Conceptos Genérales LINQ

LINQ soporta un modelo de extensibilidad muy rico que facilita la creación de operadores eficientes para fuentes de datos. La versión de LINQ del .NET Framework viene con librerías que habilitan LINQ sobre objetos, XML y Bases de Datos.

5.1.-LINQ TO OBJECTS

Con LINQ se puede hacer consultas a las colecciones y estructuras de datos en memoria., el término "LINQ to Objects" hace referencia al uso de consultas LINQ con cualquier colección **IEnumerable** o **IEnumerable<(Of <(T>)>)** directamente, sin utilizar ninguna API o proveedor LINQ intermedio, como LINQ to SQL o LINQ to XML. Puede utilizar LINQ para consultar cualquier colección enumerable, como **List<(Of <(T>)>)**, Array o **Dictionary<(Of <(TKey, TValue>)>)**. La colección puede estar definida por el usuario o ser devuelta por una API de .NET Framework (KUMAR, 2007, págs. 25-28).

Básicamente, LINQ to Objects representa una nueva forma de ver las colecciones. De la manera convencional, es necesario escribir bucles foreach complejos que especifican cómo recuperar los datos de una colección. Según LINQ, se escribe código declarativo que describe lo que se desea recuperar.

Además, las consultas LINQ ofrecen tres ventajas principales respecto a los bucles foreach tradicionales:

1. Son más concisas y legibles, sobre todo al filtrar varias condiciones.
2. Proporcionan funcionalidad eficaz de filtrado, ordenación y agrupación con código de aplicación mínimo.
3. Se pueden trasladar a otros orígenes de datos con pocas o ningunas modificaciones.

En general, cuanto más compleja sea la operación que se deba realizar con los datos, observará un número mayor de ventajas al utilizar LINQ en lugar de las técnicas de iteración convencionales.

5.2.-LINQ TO XML

Es un nuevo método para crear y manipular datos en XML. Las Propiedades y métodos de LINQ ayudan en la navegación y manipulación de elementos y atributos en XML.

LINQ to XML proporciona una interfaz de programación XML en memoria que aprovecha las características de .NET Language-Integrated Query (LINQ) Framework. LINQ to XML utiliza las características más recientes del lenguaje .NET Framework y es comparable a una actualizada y rediseñada interfaz de programación XML para el Modelo de objetos de documento (DOM).

La familia de tecnologías de LINQ proporciona un completo entorno de creación de consultas para objetos (LINQ), Bases de Datos relacionales (LINQ to SQL) y XML (LINQ to XML).

XML se ha adoptado ampliamente como un modo de formatear datos en diversos contextos. Puede encontrar XML en la Web, en archivos de configuración, en archivos de Microsoft Office Word y en Bases de Datos.

LINQ to XML es un método actualizado y rediseñado para la programación con XML. Proporciona capacidades de modificación de documento en memoria de Document Object Model (DOM), y es compatible con expresiones de consulta LINQ. Aunque estas expresiones de consulta difieren sintácticamente de XPath, proporcionan una funcionalidad similar con un establecimiento inflexible de tipos superior (KUMAR, 2007, págs. 33-36).

Tal vez sea tan significativo como las capacidades LINQ de LINQ to XML el hecho de que LINQ to XML proporciona una interfaz de programación XML mejorada. Mediante LINQ to XML, puede hacer todo lo que espera de la programación con XML, incluido lo siguiente:

- Cargar XML a partir de archivos o secuencias.
- Serializar XML a archivos o secuencias.
- Crear árboles XML desde cero mediante la construcción funcional.
- Realizar consultas de XML con ejes de tipo XPath.
- Manipular el árbol XML en memoria con métodos como Add, Remove, ReplaceWith y SetValue.

- Validar árboles XML mediante XSD.
- Usar una combinación de estas características para transformar las formas de los árboles XML.

5.2.1.-Desarrollo de LINQ to XML

LINQ to XML se dirige a diversos desarrolladores. Para el desarrollador medio que sólo desea completar una tarea, LINQ to XML simplifica el código XML al proporcionar una experiencia de consulta similar a SQL. Con un poco de estudio, los programadores pueden aprender a escribir consultas sucintas y eficaces.

Se puede usar LINQ to XML para aumentar considerablemente la productividad. Con LINQ to XML, se puede escribir menos código, que a su vez resulte más expresivo, compacto y eficaz. Pueden usar expresiones de consulta de distintos dominios de datos simultáneamente.

LINQ to XML es una interfaz de programación XML en memoria y habilitada para LINQ que permite trabajar con XML desde los lenguajes de programación de .NET Framework.

Se parece a Document Object Model (DOM) en lo que respecta a la inserción del documento XML en la memoria. Puede consultar y modificar el documento; una vez modificado, puede guardarlo en un archivo o serializarlo y enviarlo a través de una conexión. Sin embargo, LINQ to XML difiere de DOM: proporciona un nuevo modelo de objetos más ligero con el que se trabaja más fácilmente y que aprovecha las mejoras de lenguaje de Visual C# 2008.

La ventaja más importante de LINQ to XML radica en su integración con Language-Integrated Query (LINQ). Esta integración permite escribir consultas en el documento XML en memoria para recuperar colecciones de elementos y atributos. Las capacidades de consulta de LINQ to XML son comparables en cuanto a funcionalidad (aunque no en sintaxis) a XPath y XQuery. La integración de LINQ en Visual C# 2008 proporciona una escritura más rápida, comprobación en tiempo de compilación y una compatibilidad mejorada con el depurador.

La capacidad de usar los resultados de la consulta como parámetros en constructores de objetos XElement y XAttribute habilita un método eficaz para crear árboles XML. Este

método, denominado *construcción funcional*, permite que los desarrolladores transformen fácilmente árboles XML de una forma a otra (KUMAR, 2007, págs. 36-37).

La capacidad LINQ de LINQ to XML permite ejecutar consultas en XML. Por ejemplo, es posible que tenga un pedido de compra XML típico, tal como se describe en Archivo XML de muestra: pedido de compra típico (LINQ to XML).

5.2.3.-Diferencias entre LINQ to XML y XmlReader

XmlReader es un analizador rápido, de sólo avance y sin almacenamiento en caché. LINQ to XML se implementa sobre XmlReader y ambos están estrechamente integrados. No obstante, también puede usar XmlReader de forma independiente (KUMAR, 2007, pág. 40).

Aunque se superponen, LINQ to XML y XmlReader tienen diferentes escenarios de uso. Por ejemplo, se está creando un servicio web que analizará cientos de de documentos XML por segundo y los documentos tienen la misma estructura, por lo que sólo tiene que escribir una implementación de código para analizar el XML. En ese caso, probablemente deseará usar XmlReader de forma independiente.

Por el contrario, si está creando un sistema que analiza varios documentos XML más pequeños y cada uno es diferente, quizás desee aprovechar las mejoras de productividad que LINQ to XML proporciona.

5.2.4.-Diferencias entre LINQ to XML y XSLT

Tanto LINQ to XML como XSLT proporcionan amplias capacidades de transformación de documentos XML. XSLT es un enfoque declarativo basado en reglas. Los programadores avanzados de XSLT escriben XSLT en un estilo de programación funcional que resalta un enfoque sin estado, factorizando funciones puras que se implementan sin efectos secundarios (KUMAR, 2007, pág. 41). Este enfoque basado en reglas o funcional no es familiar para muchos desarrolladores y puede requerir bastante trabajo para su dominio.

XSLT puede ser un sistema muy productivo que produce aplicaciones de alto rendimiento. Algunas grandes compañías web utilizan XSLT como forma de generar HTML a partir de XML extraído de varios almacenes de datos. El motor XSLT administrado compila código XSLT a CLR y tiene un rendimiento incluso mejor en algunos escenarios que el motor XSLT nativo.

No obstante, XSLT no aprovecha el conocimiento de C# y Visual Basic que muchos desarrolladores tienen. Requiere que los desarrolladores escriban código en un lenguaje de programación complejo y diferente. Usar dos sistemas de desarrollo diferentes y no integrados como C# (o Visual Basic) y XSLT tiene como resultado sistemas de software que son más difíciles de desarrollar y mantener.

Cuando se dominen las expresiones de consulta LINQ to XML, las transformaciones de LINQ to XML son una tecnología eficaz y fácil de usar. Básicamente, se forman documentos XML utilizando construcciones funcionales, extrayendo datos de varios orígenes, construyendo objetos XElement dinámicamente y ensamblando el conjunto en un nuevo árbol XML. La transformación puede generar un documento completamente nuevo. Construir transformaciones en LINQ to XML es relativamente sencillo e intuitivo y el código resultante es legible. Esto reduce los costos de desarrollo y mantenimiento.

LINQ to XML no se ha diseñado para sustituir a XSLT. XSLT sigue siendo la herramienta preferida de transformaciones XML complicadas y basadas en documentos, especialmente si la estructura del documento no está bien definida.

XSLT tiene la ventaja de ser un estándar de W3C. Si tiene el requisito de usar solamente tecnologías que son estándares, XSLT puede ser más adecuado.

XSLT es XML y, por lo tanto, se puede manipular mediante programación.

5.2.5.-Diferencias entre LINQ to XML y MSXML

MSXML es la tecnología basada en COM utilizada para procesar XML y que se incluye con Microsoft Windows. MSXML ofrece una implementación nativa de DOM, que es compatible con XPath y XSLT. También contiene el analizador basado en eventos y sin almacenamiento en caché SAX2 (KUMAR, 2007, págs. 41-43).

MSXML tiene un buen rendimiento, es seguro de forma predeterminada en la mayoría de casos y se puede tener acceso a él en Internet Explorer para realizar procesamiento XML en el cliente en aplicaciones de estilo AJAX. MSXML se puede usar en cualquier lenguaje de programación que admita COM, incluyendo C++, JavaScript y Visual Basic 6.0.

No se recomienda el uso de MSXML en código administrado basado en Common Language Runtime (CLR).

5.2.6.-Diferencias entre LINQ to XML y XmlLite

XmlLite es un analizador de extracción sin almacenamiento en caché, de sólo avance. Los desarrolladores utilizan principalmente XmlLite con C++. No se recomienda usar XmlLite con código administrado (MSDN Library LINQ, 2009).

La principal ventaja de XmlLite es que es un analizador XML rápido y ligero que es seguro en la mayoría de casos. Su área de exposición a amenazas es muy pequeña. Si debe analizar documentos que no son de confianza y desea protegerse de ataques tales como denegación de servicio o exposición de datos, XmlLite puede ser una buena opción.

XmlLite no se integra con Language-Integrated Query (LINQ). No proporciona al programador mejoras de productividad que sean la fuerza motivadora detrás de LINQ.

5.3.-LINQ TO SQL

LINQ to SQL es una implementación de O/RM(object relational mapping, mapeador de objetos relacionales) y nos permite modelar Bases de Datos relacionales con clases de .NET (MEHTA, 2008, pág. 47). Se pueden consultar Bases de Datos con LINQ, así como actualizar/añadir/borrar datos de ellas.

LINQ to SQL es un componente de .NET Framework 3.5 que proporciona una infraestructura en tiempo de ejecución para administrar los datos relacionales como objetos.

En LINQ to SQL, el modelo de datos de una Base de Datos relacional se asigna a un modelo de objetos expresado en el lenguaje de programación del programador. Cuando la aplicación se ejecuta, LINQ to SQL convierte a SQL las consultas integradas en el lenguaje en el modelo de objetos y las envía a la Base de Datos para su ejecución. Cuando la Base de Datos devuelve los resultados, LINQ to SQL los vuelve a convertir en objetos con los que pueda trabajar en su propio lenguaje de programación (MEHTA, 2008, págs. 47-48).

Los programadores de Visual Studio normalmente utilizan el Object Relational Designer, que proporciona una interfaz de usuario para implementar muchas de las características de LINQ to SQL.

5.3.1.-Modelo de Objetos LINQ to SQL

En LINQ to SQL, el modelo de datos de una Base de Datos relacional se asigna a un modelo de objetos expresado en el lenguaje de programación del desarrollador. Así, las operaciones con los datos se realizan según el modelo de objetos.

En este escenario, no ejecuta comandos de Base de Datos (como INSERT) en la Base de Datos. En su lugar, cambia los valores y ejecuta los métodos de su modelo de objetos. Si desea consultar la Base de Datos o enviar cambios, LINQ to SQL convierte sus solicitudes en los comandos SQL correctos y los envía a la Base de Datos.

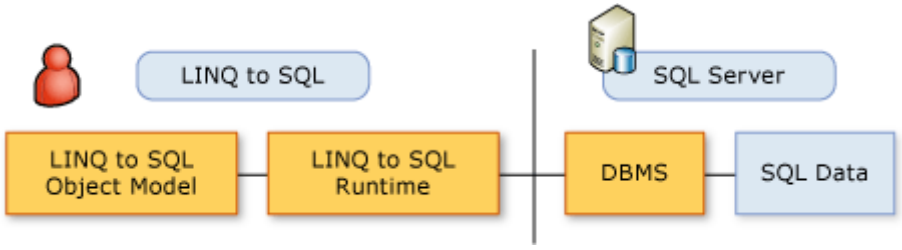

Figura 44: Modelado de Datos LINQ to SQL

En la tabla siguiente se resumen los elementos más fundamentales del modelo de objetos de LINQ to SQL y su relación con los elementos del modelo de datos relacionales.

Modelo de objetos de LINQ to SQL	Modelo de datos relacionales
Clase de entidad	Tabla
Miembro de clase	Columna
Asociación	Relación de clave externa
Método	Procedimiento almacenado o función

Tabla 10: Modelado de Datos en LINQ

5.4.-LINQ TO DATASET

LINQ to DataSet facilita y acelera las consultas en datos almacenados en caché en un objeto DataSet. En concreto, LINQ to DataSet simplifica la consulta permitiendo a los desarrolladores escribir consultas a partir del lenguaje de programación mismo, en lugar de utilizar un lenguaje de consulta diferente (KUMAR, 2007, pág. 141). Esto resulta especialmente útil para desarrolladores de Visual Studio, que ahora pueden aprovechar la comprobación de sintaxis en tiempo de compilación, los tipos estáticos y la compatibilidad con IntelliSense que proporciona Visual Studio en las consultas.

LINQ to DataSet también se puede utilizar para consultar en datos que se han consolidado de uno o más orígenes de datos. Esto permite muchos casos que requieren flexibilidad en la forma de representar y controlar los datos, como consultar datos agregados localmente y almacenar en caché en el nivel medio en aplicaciones web. En concreto, las aplicaciones de Business Intelligence, análisis e informes genéricos requieren este método de manipulación.

La funcionalidad LINQ to DataSet se expone principalmente mediante métodos de extensión en las clases DataRowExtensions y DataTableExtensions. LINQ to DataSet se basa y utiliza la arquitectura existente ADO.NET 2.0, y no está destinada a reemplazar ADO.NET 2.0 en el código de aplicación. El código de ADO.NET 2.0 existente continuará funcionando en una aplicación LINQ to DataSet (KUMAR, 2007, págs. 141-142). La relación de LINQ to DataSet con ADO.NET 2.0 y los datos almacenados se muestran en el diagrama siguiente.

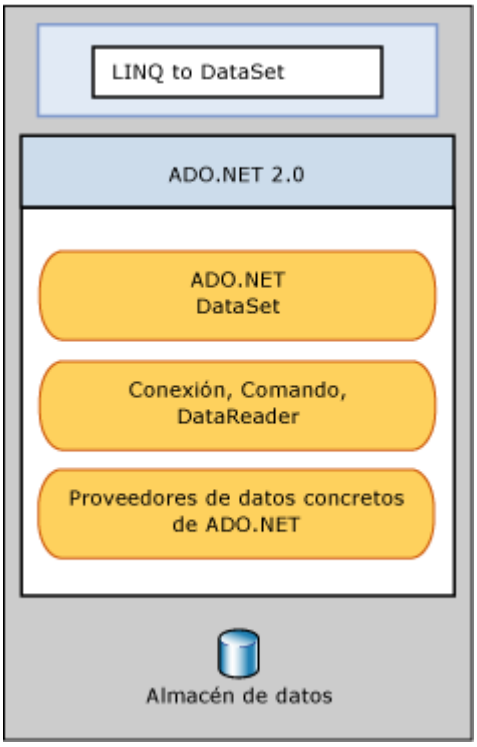

Figura 45: Almacenamiento de datos Dataset

DataSet es uno de los componentes más ampliamente utilizados de ADO.NET. Es un elemento fundamental del modelo de programación desconectado en el que se basa ADO.NET y permite almacenar explícitamente en caché datos de diferentes orígenes de

datos. Para el nivel de presentación, DataSet está estrechamente integrado en los controles de GUI para el enlace de datos. Para el nivel medio, proporciona una caché que conserva la forma relacional de los datos e incluye servicios de exploración de jerarquías y consultas rápidos y sencillos. Una técnica común que se usa para reducir el número de solicitudes de una Bases de Datos consiste en utilizar DataSet para el almacenamiento en caché en el nivel medio. Por ejemplo, piense en una aplicación web de ASP.NET controlada por datos. A menudo una parte importante de los datos de aplicación no cambia frecuentemente y es común entre sesiones o usuarios. Estos datos se pueden conservar en memoria o en un servidor web, lo que reduce el número de solicitudes en la Base de Datos y acelera las interacciones del usuario. Otro aspecto útil de DataSet es que permite que una aplicación lleve subconjuntos de datos de uno o más orígenes de datos al espacio de la aplicación. La aplicación puede manipular los datos en memoria mientras retiene su forma relacional (MSDN Library LINQ, 2009).

A pesar de su importancia, DataSet tiene capacidades de consulta limitadas. El método Select se puede usar para filtrar y ordenar y los métodos GetChildRows y GetParentRow se pueden usar para la exploración de jerarquías. Sin embargo, para cualquier tarea más compleja, el programador debe escribir una consulta personalizada. Esto puede tener como resultado aplicaciones con un bajo rendimiento y con un mantenimiento difícil.

LINQ to DataSet facilita y acelera las consultas en datos almacenados en caché en un objeto DataSet. Esas consultas se expresan en el lenguaje de programación mismo, en lugar de como literales de cadena incrustadas en el código de la aplicación. Esto significa que los desarrolladores no tienen que aprender un lenguaje de consultas diferente. Adicionalmente, LINQ to DataSet permite a los desarrolladores de Visual Studio trabajar de forma más productiva, ya que la IDE de Visual Studio proporciona comprobación de sintaxis en tiempo de compilación, tipos estáticos y compatibilidad de IntelliSense con LINQ. LINQ to DataSet también se puede usar para consultar los datos que se han consolidado de uno o más orígenes de datos. Esto permite muchos casos que requieren flexibilidad en la forma de representar y controlar los datos. En concreto, las aplicaciones de inteligencia empresaria, análisis e informes genéricos requieren este método de manipulación.

5.4.1.-Consultar conjuntos de datos usando LINQ to DataSet

Antes de empezar a consultar un objeto DataSet usando LINQ to DataSet, se debe rellenar el DataSet. Existen varias formas de cargar datos en un DataSet, como usar la clase DataAdapter o LINQ to SQL. Cuando se han cargado los datos en un objeto

DataSet, se puede empezar a realizar consultas en él. La formulación de consultas usando LINQ to DataSet es similar a usar Language-Integrated Query (LINQ) en otros orígenes de datos habilitados para LINQ. se pueden realizar consultas de LINQ en tablas únicas en un DataSet o en más de una tabla usando los operadores de consulta estándar Join y GroupJoin (KUMAR, 2007, págs. 144-146).

Se admiten consultas de LINQ en objetos DataSet con tipo y sin tipo. Si el esquema de DataSet es desconocido en tiempo de diseño, se recomienda un DataSet con tipo. En un DataSet con tipo, las tablas y las filas tienen miembros con tipo para cada una de las columnas, lo que hace que las consultas sean más sencillas y legibles.

Además de los operadores de consulta estándar implementados en System.Core.dll, LINQ to DataSet agrega varias extensiones específicas de DataSet que hacen que realizar consultas en un conjunto de objetos DataRow sea sencillo. Estas extensiones específicas de DataSet incluyen operadores para comparar secuencias de filas, así como métodos que proporcionan acceso a los valores de columna de un DataRow.

5.4.2.-Aplicaciones con n niveles y LINQ to DataSet

Las aplicaciones de datos con n niveles son aplicaciones centradas en datos separadas en varias capas lógicas (o niveles). Una aplicación con n niveles típica incluye un nivel de presentación, un nivel medio y un nivel de datos. Al separar los componentes de la aplicación en diferentes niveles, se aumenta el mantenimiento y la escalabilidad de la aplicación. Para obtener más información acerca de las aplicaciones de datos con n niveles, vea N-Tier Data Applications.

En las aplicaciones con n niveles, a menudo se utiliza DataSet en el nivel medio para almacenar en caché información para una aplicación web. La funcionalidad de consulta de LINQ to DataSet se implementa mediante los métodos de extensión y DataSet de ADO.NET 2.0 existente.

En el siguiente diagrama se muestra cómo se relaciona LINQ to DataSet con el DataSet y cómo encaja en una aplicación de nivel n:

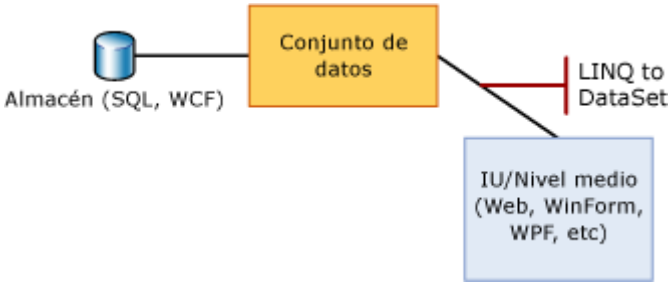

Figura 46: Relación LINQ a DataSet

.

5.5.-Operadores Estándar de Consulta

Los operadores de consulta estándar son los métodos que forman el modelo de Language-Integrated Query (LINQ). La mayoría de estos métodos funciona en secuencias, donde una secuencia es un objeto cuyo tipo implementa la interfaz **IEnumerable<(Of <(T>)>)** o la interfaz **IQueryable<(Of <(T>)>)**. Los operadores de consulta estándar proporcionan capacidades de consulta que incluyen filtrado, proyección, agregación, ordenación y otras (KUMAR, 2007, pág. 171).

Hay dos conjuntos de operadores de consulta estándar de LINQ, uno que funciona sobre objetos de tipo **IEnumerable<(Of <(T>)>)** y otro que funciona sobre objetos de tipo **IQueryable<(Of <(T>)>).** Los métodos que constituyen cada conjunto son miembros estáticos de las clases Enumerable y Queryable, respectivamente. Se definen como métodos de extensión del tipo sobre el que operan. Esto significa que se pueden llamar utilizando sintaxis del método estático o sintaxis del método de instancia.

Además, varios métodos de operador de consulta estándar funcionan con tipos distintos de los que se basan en **IEnumerable<(Of <(T>)>)** o **IQueryable<(Of <(T>)>)**. El tipo Enumerable define dos de esos métodos, que operan sobre objetos de tipo IEnumerable. Estos métodos, **Cast<(Of <(TResult>)>)(IEnumerable)** y **OfType<(Of <(TResult>)>)(IEnumerable)**, permiten que una colección no parametrizada, o no genérica, pueda ser consultada en el modelo de LINQ. Esto lo consiguen creando una colección de objetos con establecimiento inflexible de tipos. La clase Queryable define dos métodos similares, **Cast<(Of <(TResult>)>)(IQueryable)** y **OfType<(Of <(TResult>)>)(IQueryable)**, que operan sobre objetos de tipo Queryable (KUMAR, 2007, págs. 183-204).

Los operadores de consulta estándar difieren en el momento de su ejecución, dependiendo de si devuelven un valor singleton o una secuencia de valores. Los métodos que devuelven un valor singleton (por ejemplo, Average y Sum) se ejecutan inmediatamente. Los métodos que devuelven una secuencia retrasan la ejecución de la consulta y devuelven un objeto enumerable.

En el caso de los métodos que operan sobre colecciones en memoria, es decir, aquellos métodos que extienden **IEnumerable<(Of <(T>)>)**, el objeto enumerable devuelto captura los argumentos que se pasaron al método. Cuando se enumera ese objeto, se emplea la lógica del operador de consulta y se devuelven los resultados de la consulta.

En contraste, los métodos que extienden **IQueryable<(Of <(T>)>)** no implementan cualquier comportamiento de consulta, sino que generan un árbol de expresión que representa la consulta que se va a realizar. El procesamiento de la consulta es administrado por el objeto **IQueryable<(Of <(T>)>)** del origen.

Las llamadas a métodos de consulta se pueden encadenar juntas en una sola consulta, lo cual permite hacer consultas arbitrariamente complejas.

CAPÍTULO 6. Ejemplo de Desarrollo .NET con LINQ

En este Capítulo se muestra un ejemplo sencillo y practico del uso de LINQ a SQL, la herramienta de desarrollo es Visual Studio 2008, y el manejador de Base de Datos Microsoft SQL Server. El ejemplo es de un sistema de Control Escolar, partiendo del modelado con MVC, el diseño y creación de la Base de Datos con Microsoft SQL Server, así como la conexión entre la Base de Datos y la herramienta LINQ.

6.1.-Modelado con MVC del sistema de Control Escolar

Como se ha mencionado en temas anteriores el modelado de los sistemas es importante para su correcta implementación, en la figura 47 se muestran los roles para el ejemplo de Control Escolar.

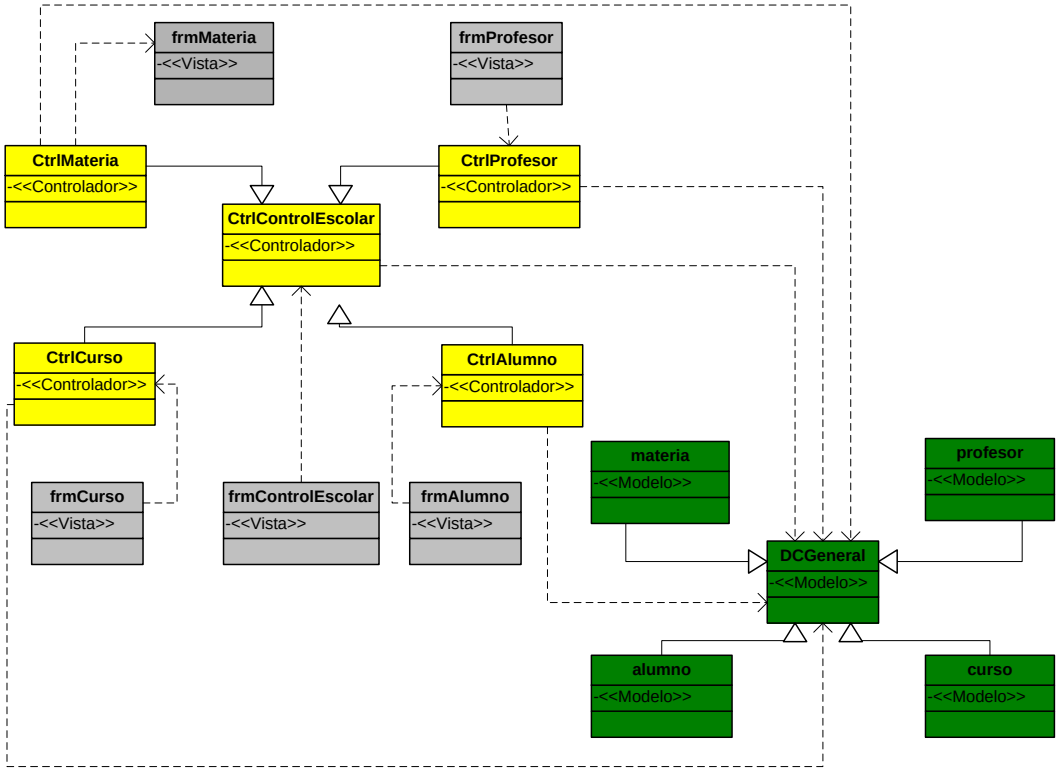

Figura 47: Diagrama de Clases Control Escolar en MVC

Los **modelos** de una aplicación basada en MVC son los componentes responsables de mantener el estado. Normalmente el estado se guarda en una Base de Datos para el ejemplo del sistema de Control Escolar, se tiene la entidad de materia, profesor, alumno, curso y DCGeneral que para el caso del uso de la herramienta LINQ es la que concentra todas estas, si bien podría solo mostrarse DCGeneral no es así ya que pueden utilizarse como extensiones para funciones que no son adecuadas con el formato que se necesite, así también para demostrar que están de manera implícita en dicha clase.

Las **vistas** son los componentes responsables de mostrar la interfaz de usuario de la aplicación. Esta interfaz se crea a partir del modelo de datos es decir, es la representación que el usuario visualiza, así dependiendo de lo que contenga las entidades serán los valores que pueden tener, en este ejemplo son los formularios de frmAlumno, frmCurso, frmProfesor, frmMateria y frmControlEscolar, que a su vez tendrán cajas de texto, etiquetas, etc.

Los **controladores** son los componentes responsables de la interacción con el usuario final, manipular el modelo y por último elegir una vista para interactuar. En las aplicaciones del MVC la vista solo muestra la información, es el controlador el que administra y responde a las peticiones del usuario y a las interacciones, en este ejemplo los controladores son CtrlControlEscolar, CtrlAlumno, CtrlProfesor, CtrlCurso y CtrlMateria.

En los siguientes temas se volverá a retomar este modelo ya implementado con Visual Studio 2008, después del mapeo de la Base de Datos.

6.2.- Elaboración de la Base de Datos de Control Escolar

Para continuar con la implementación del ejemplo se tiene que diseñar las tablas de la Base de Datos y posteriormente crear sus relaciones, el motor de Base de Datos en el que tienen que estar es Microsoft SQL Server de la versión 2005 o posteriores, las tablas a generar son las de Alumno, Profesor, Materia y Curso en la Base de Datos de CONTROLESCOLAR, ver figura 40.

Para crear la Base de Datos se utiliza la interfaz grafica Microsoft SQL Server Management Server para realizar el proceso en un ambiente visual, aunque se puede usar cualquier otro administrador de Base de Datos que soporte el motor de SQL Server.

En la figura en la figura 48 se puede ver como se crea una Base de Datos desde el administrador de Microsoft SQL Server, aquí se establecerá el nombre de la Base de Datos (CONTROLESCOLAR) y como no se tienen otros requerimientos se deja con los valores por defecto.

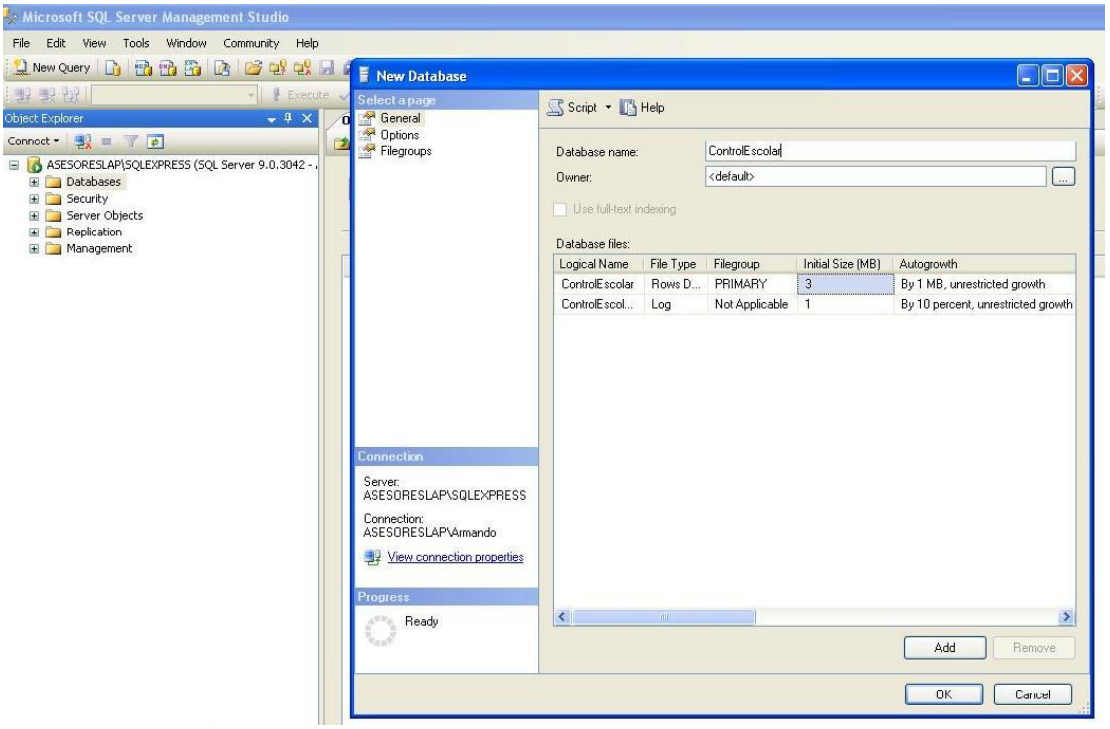

Figura 48: Creación de la Base de Datos Control Escolar

Una vez creada la Base de Datos y teniendo propiamente el diseño de esta se procede a generar las tablas establecidas. Se deben de considerar los aspectos que en capítulos anteriores se mencionaron, específicamente en los que hacen referencia a la elaboración y diseño de la Base de Datos, ya que LINQ trabaja con Bases de Datos relacionales.

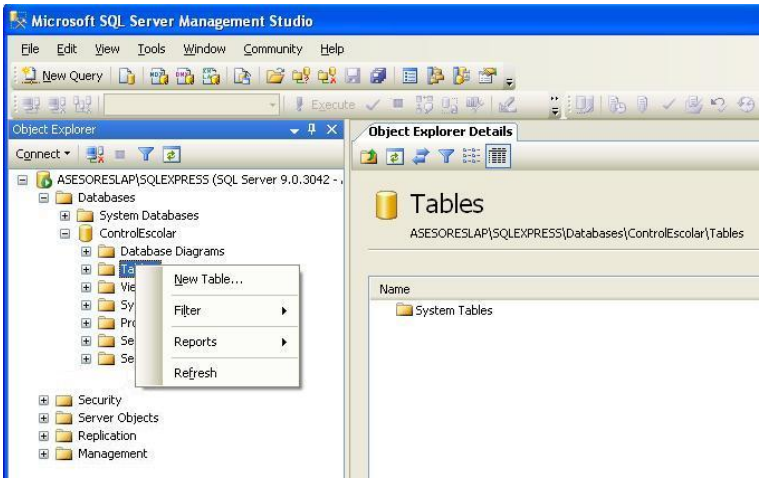

Figura 49: Agregar tablas a la Base de Datos Control Escolar

Al crear las tablas se agregan los campos que en el diseño se estipularon, ver figura 50, es bueno considerar a futuro que tanto crecerá la Base de Datos y en especifico los valores de las tablas, para asignarle el tipo de dato correcto, basado en el tamaño de los datos que nuestro sistema ingresara ya que puede suceder que no sea suficiente.

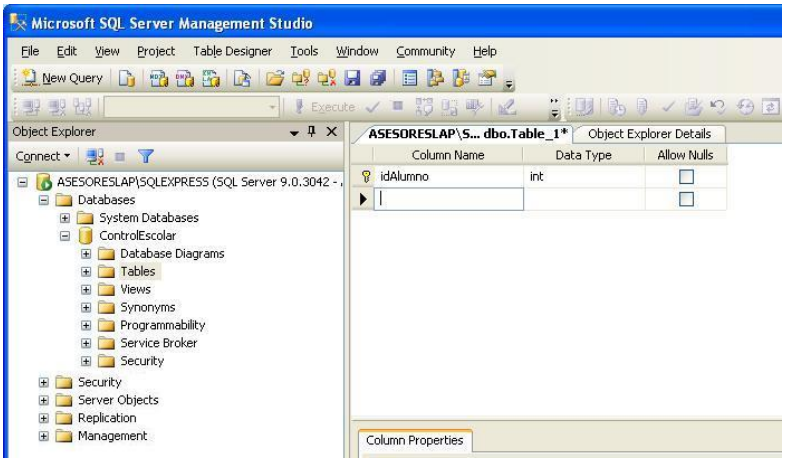

Figura 50: Campos en las tablas de la Base de Datos Control Escolar

Después de que se elaboraron las tablas de alumno, curso, profesor y materia, se establecen las relaciones, LINQ las ocupara para generar las asociaciones. Es importante tener en cuenta los aspectos que involucran las Bases de Datos relacionales, debido a que si no cumplen con los requerimientos de un buen diseño, pueden generar errores al momento de ejecución de la aplicación, como los referentes a integridad. En la figura 51 se puede ver las tablas con sus correspondientes relaciones.

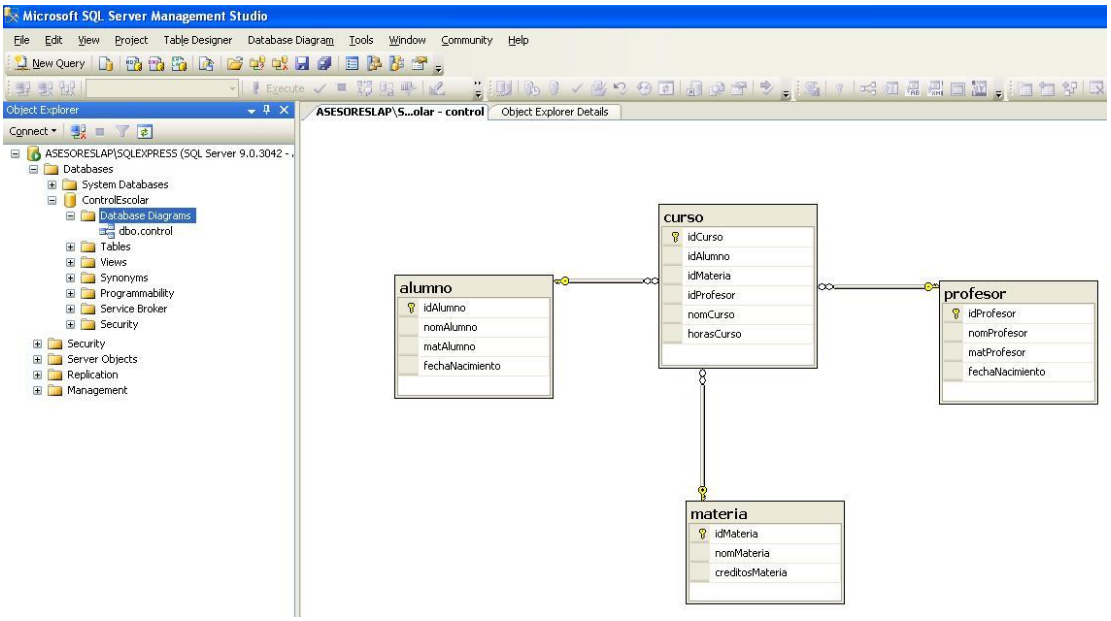

Figura 51: Diagrama de las tablas en la Base de Datos Control Escolar.

6.3.- Mapeo de la Base de Datos de Control Escolar

En la figura 52 se muestran las tablas a mapear y teniendo activa la Base de Datos, se procede a realizar el mapeo de los objetos (alumno, curso, profesor y materia) de la Base de Datos a los objetos de nuestro sistema que está contenido en el entorno de Visual Studio 2008.

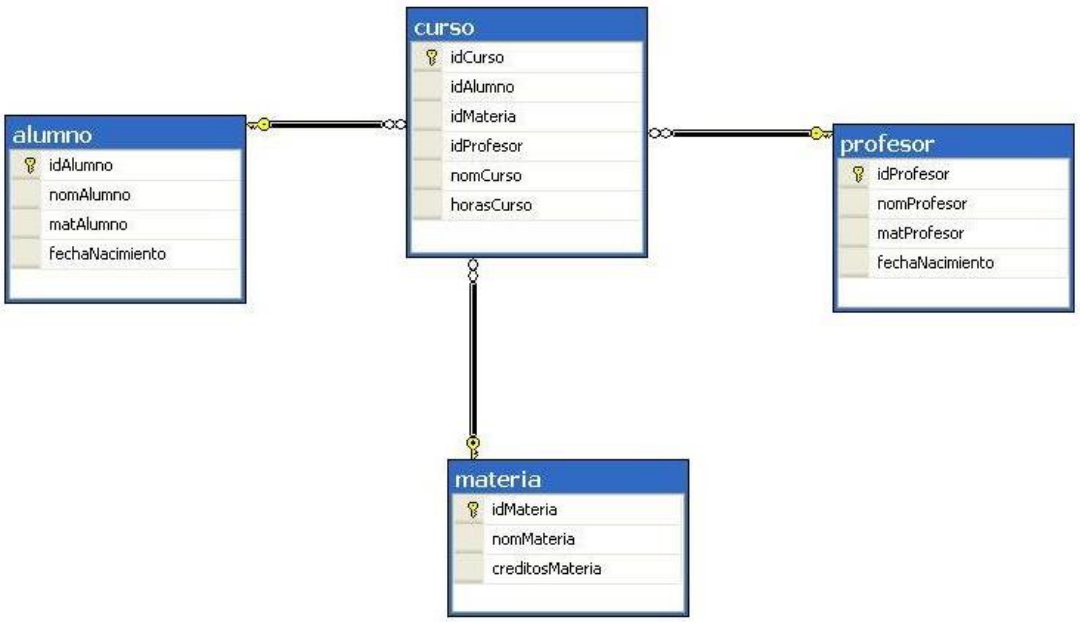

Figura 52. Diagrama de Tablas Control Escolar

Se utilizara el Diseñador relacional de objetos (Object Relational Designer) que proporciona una superficie de diseño visual para crear clases de entidad y asociaciones (relaciones). Es decir, el Diseñador relacional de objetos se usa para crear un modelo de objetos en una aplicación que se asigna a los objetos de una Base de Datos. También genera una clase DataContext con establecimiento inflexible de tipos que se usa para enviar y recibir datos entre las clases de entidad y la Base de Datos.

El Diseñador relacional de objetos también proporciona la funcionalidad para asignar los procedimientos almacenados y funciones a los métodos de DataContext con el fin de devolver datos y rellenar las clases de entidad, para este ejemplo no aplican pero cabe mencionarlos si es que en algún momento se tuvieran. El Diseñador relacional de objetos permite diseñar relaciones de herencia entre las clases de entidad.

A partir de la versión de Visual Studio 2008 ya se incluyen las librerías y controladoras de LINQ, así que no se tiene que hacer más que agregar al proyecto las referencias. En versiones anteriores como Visual Studio 2005 no están incluidas por lo cual se tiene que agregar a dicha versión.

Es muy importante considerar el modelo creado en el tema de modelado con MVC, ver figura 47, ya que es la estructura la cual llevara el sistema, se agregaran los proyectos de **ControlEscolar.Control** que es el contendor de los controladores, el proyecto de **ControlEscolar.Data** que es contendor de los modelos y el proyecto de **ControlEscolar.View** que es el contendor de las vistas, en la figura53, se pueden ver dichos proyectos agregados.

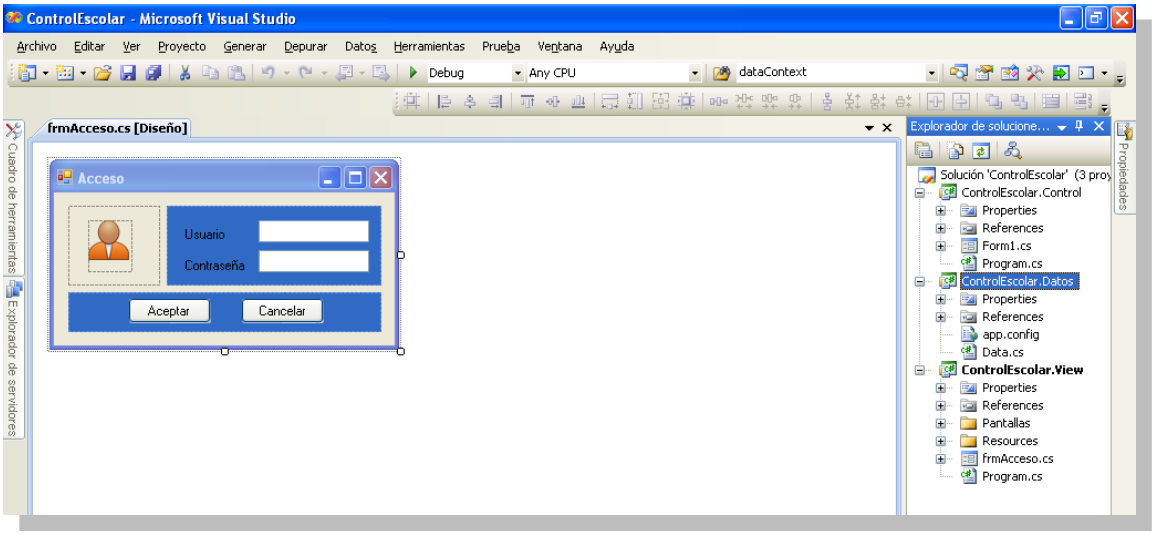

Figura 53. Entorno Visual Studio generación de proyecto

En el modulo de datos (ControlEscolar.Data) se agregan las extensiones de las entidades y principalmente el archivo .DBML que es el que contiene las entidades mapeadas de la Base de Datos. En la figura 54 se muestra como agregar dicho elemento, dentro de las opciones presentadas se selecciona el elemento **Clases de LINQ to SQL**.

Figura 54. Creación de Archivo DBML

La generación del código del archivo .DBML se da en tres etapas que son:

1. El Extractor de DBML extrae información del esquema de la Base de Datos y vuelve a ensamblar la información en un archivo DBML con formato XML.
2. El Validador de DBML examina el archivo DBML en busca de errores.
3. Si existen errores, el archivo se pasa al Generador de código.

Una vez generado el archivo se realiza el mapeo, agregando una conexión de Base de Datos en el **explorador de servidores** de Visual Studio 2008, este se muestra al agregar el archivo .DBML, cuando se establece la conexión se solicita los datos de autenticación (usuario y contraseña) para posteriormente utilizarlos cada que sea necesario, después de realizar la conexión exitosamente se seleccionan las tablas que esta Base de Datos contiene, ver figura 55.

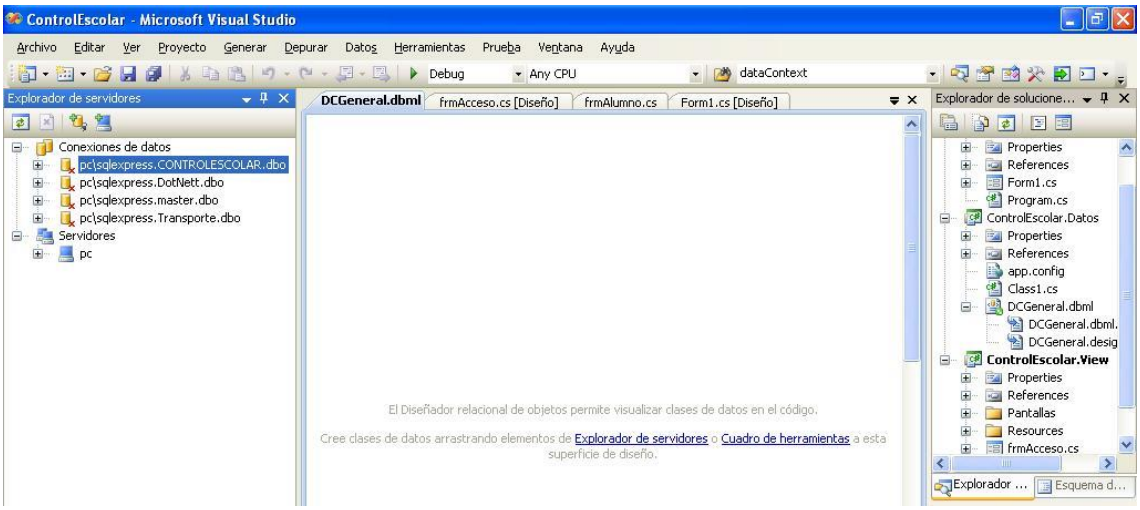

Figura 55: Explorador de Servidores

Al identificar y seleccionar las tablas, se arrastran al diseñador relacional de objetos ver figura 56. Se deben elegir las tablas necesarias para el funcionamiento del modulo requerido y dependiendo el diseño que se tenga del sistema se pueden elegir todas o solo algunas, para este ejemplo se hace uso de todas.

Figura 56 : Selección de Tablas en BD desde Explorador de Servidores

x

Al concluir con dicho procedimiento se puede ver en el Diseñador relacional de objetos(O/R Designer), ver figura 57. Al generar el archivo .DBML no solo mapeara las tablas, si en dicha Base de Datos existen procedimientos almacenados estos también se

generaran en el código. El Diseñador relacional de objetos permite asignar clases de LINQ to SQL a las tablas de una Base de Datos. Estas clases asignadas también se denominan clases de entidad. Las propiedades de una clase de entidad se asignan a las columnas de la tabla y pueden enlazar los datos a los controles de un formulario Windows Forms.

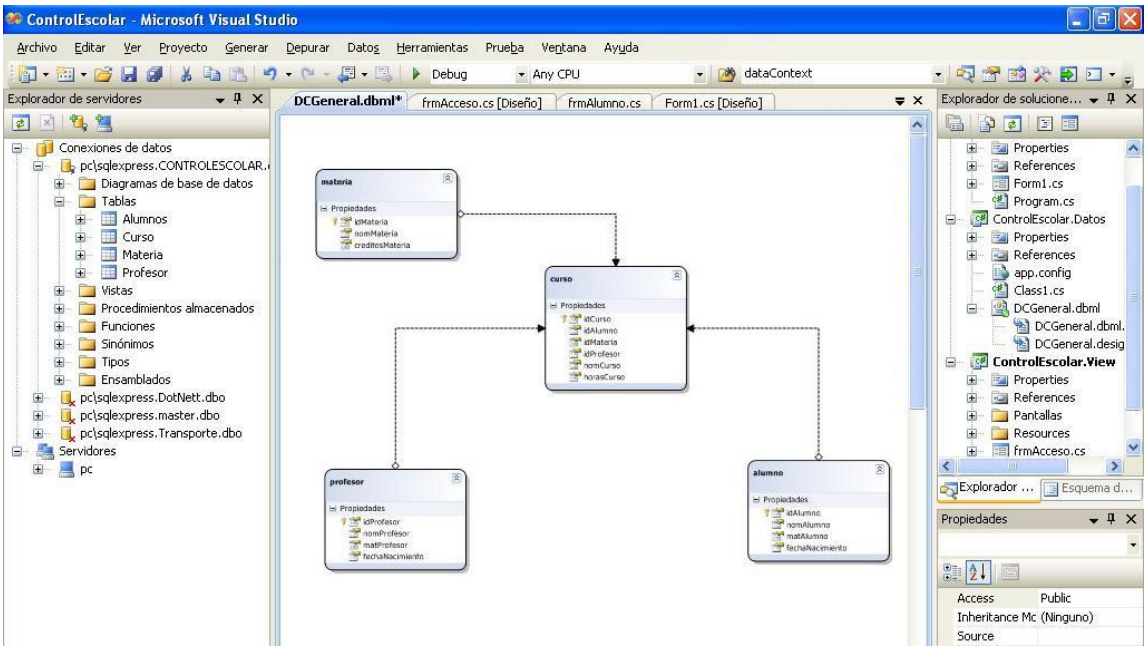

Figura 57: Diseñador Relacional de Objetos

Después de generar el código fuente se forman distintas entidades, entre ellas la más importante el DataContext que es el origen de todas las entidades asignadas en una conexión de Base de Datos, ver figura 58. Realiza un seguimiento de los cambios realizados en todas las entidades recuperadas y mantiene una "memoria caché de identidad" que garantiza que las entidades que se recuperan más de una vez se representan utilizando la misma instancia de objeto.

En general, una instancia de DataContext está diseñada para que dure una "unidad de trabajo", es decir mientras se mantiene en ejecución. DataContext es un objeto ligero cuya creación no es costosa. Una aplicación típica crea instancias de DataContext en el ámbito de métodos o como miembro de clases de duración corta que representan un conjunto lógico de operaciones de Base de Datos relacionadas.

```
        private static System.Data.Linq.Mapping.MappingSource mappingSource = new AttributeMappingSource();

    #region Extensibility Method Definitions
    partial void OnCreated();
    partial void InsertAlumnos(Alumnos instance);
    partial void UpdateAlumnos(Alumnos instance);
    partial void DeleteAlumnos(Alumnos instance);
    partial void InsertCurso(Curso instance);
    partial void UpdateCurso(Curso instance);
    partial void DeleteCurso(Curso instance);
    partial void InsertMateria(Materia instance);
    partial void UpdateMateria(Materia instance);
    partial void DeleteMateria(Materia instance);
    partial void InsertProfesor(Profesor instance);
    partial void UpdateProfesor(Profesor instance);
    partial void DeleteProfesor(Profesor instance);
    #endregion

        public DCGeneralDataContext() :
                base(global::Generico.Datos.Properties.Settings.Default.CONTROLESCOLARConnectionString, mappin

        public DCGeneralDataContext(string connection) :
                base(connection, mappingSource)...

        public DCGeneralDataContext(System.Data.IDbConnection connection) :
                base(connection, mappingSource)...

        public DCGeneralDataContext(string connection, System.Data.Linq.Mapping.MappingSource mappingSource) :
                base(connection, mappingSource)...
#pragma warning disable 1591
//------------------------------------------------------------------------------
// <auto-generated>
//     Este código fue generado por una herramienta.
//     Versión del motor en tiempo de ejecución:2.0.50727.1433
//
//     Los cambios en este archivo podrían causar un comportamiento incorrecto y se perderán si
//     se vuelve a generar el código.
// </auto-generated>
//------------------------------------------------------------------------------

namespace Generico.Datos
{
    using System.Data.Linq;
    using System.Data.Linq.Mapping;
    using System.Data;
    using System.Collections.Generic;
    using System.Reflection;
    using System.Linq;
    using System.Linq.Expressions;
    using System.ComponentModel;
    using System;


    [System.Data.Linq.Mapping.DatabaseAttribute(Name="CONTROLESCOLAR")]
    public partial class DCGeneralDataContext : System.Data.Linq.DataContext
    {

        private static System.Data.Linq.Mapping.MappingSource mappingSource = new AttributeMappingSource();
```

Figura 58: DataContext en el Código

Una tabla de Base de Datos se representa mediante una clase de entidad. Una clase de entidad es como cualquier otra clase que se pueda crear, con la diferencia de que se anota utilizando información especial que asocia la clase a una tabla de Base de Datos. Para realizar esta anotación, se agrega un atributo personalizado (TableAttribute) a la declaración de clase, como en el ejemplo siguiente con alumno, ver figura 59.

Figura 59: Clases de Entidad LINQ to SQL

Además de asociar clases a tablas, se designan campos o propiedades para representar columnas de Base de Datos. Se define mediante el atributo ColumnAttribute, ver figura 60.

Figura 60: Miembros de Clase y columnas de Base de Datos en LINQ to SQL

Sólo los campos y las propiedades que estén asignados a columnas se conservan en la Base de Datos o se recuperan de ella. Si no se han declarado como columnas, se consideran partes transitorias de la lógica de aplicación.

El atributo ColumnAttribute tiene varias propiedades que se pueden utilizar para personalizar los miembros que representan columnas (por ejemplo, para designar un miembro como representativo de una columna de clave principal), ver figura 61.

```
        get
        {
            return this._nomAlumno;
        }
        set
        {
            if ((this._nomAlumno != value))
            {
                this.OnnomAlumnoChanging(value);
                this.SendPropertyChanging();
                this._nomAlumno = value;
                this.SendPropertyChanged("nomAlumno");
                this.OnnomAlumnoChanged();
            }
        }
    }

    [Column(Storage="_matAlumno", DbType="NVarChar(50) NOT NULL", CanBeNull=false)]
    public string matAlumno
    {
        get
        {
            return this._matAlumno;
        }
        set
        {
            if ((this._matAlumno != value))
            {
                this.OnmatAlumnoChanging(value);
                this.SendPropertyChanging();
```

Figura 61: Atributos de Columna

Las asociaciones de Base de Datos (como las relaciones de clave externa y clave principal) se representan aplicando el atributo AssociationAttribute. En el segmento de código siguiente, la clase profesor contiene una propiedad curso que tiene un atributo AssociationAttribute. Esta propiedad y su atributo proporcionan a la clase profesor una relación con la clase curso, ver figura 62. De esta manera también lo es para las demás tablas que están relacionadas.

```
[Association(Name="profesor_curso", Storage="_curso", OtherKey="idProfesor")]
public EntitySet<curso> curso
{
    get
    {
        return this._curso;
    }
    set
    {
        this._curso.Assign(value);
    }
}

public event PropertyChangingEventHandler PropertyChanging;

public event PropertyChangedEventHandler PropertyChanged;

protected virtual void SendPropertyChanging()
{
    if ((this.PropertyChanging != null))
    {
        this.PropertyChanging(this, emptyChangingEventArgs);
    }
}

protected virtual void SendPropertyChanged(String propertyName)
{
    if ((this.PropertyChanged != null))
    {

        if ((this.PropertyChanged != null))
        {
            this.PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
        }
    }

    private void attach_curso(curso entity)
    {
        this.SendPropertyChanging();
        entity.profesor = this;
    }

    private void detach_curso(curso entity)
    {
        this.SendPropertyChanging();
        entity.profesor = null;
    }
}
}
#pragma warning restore 1591
```

Figura 62: Asociaciones y relaciones de Base de Datos LINQ to SQL

LINQ to SQL admite los procedimientos almacenados y las funciones definidas por el usuario. Estas abstracciones definidas por la Base de Datos se asignan a objetos de cliente de tal forma que se pueda tener acceso a ellos con establecimiento inflexible de tipos desde el código de cliente. Las firmas de método guardan la máxima similitud con las firmas de los procedimientos y funciones que se definen en la Base de Datos.

Cuando se define el modelo se crean cuatro clases: Materia, Alumno, Profesor y Curso. Las propiedades de cada clase mapean las diferentes columnas de las tablas correspondientes en la Base de Datos. Cada instancia de cada clase es una entidad que representa una fila de cada tabla.

Cuando se define el modelo de datos, el diseñador LINQ to SQL creó la clase llamada DataContext que proporciona todo lo necesario para poder consultar/actualizar la Base de Datos.

Se pueden usar expresiones LINQ para consultar y obtener datos usando la clase DCGeneralDatacontext. LINQ to SQL traduce automáticamente estas expresiones LINQ al código SQL apropiado en tiempo de ejecución.

Al generar el archivo .DBML se agregan las demás entidades a cada modulo quedando como lo muestra la figura 63.

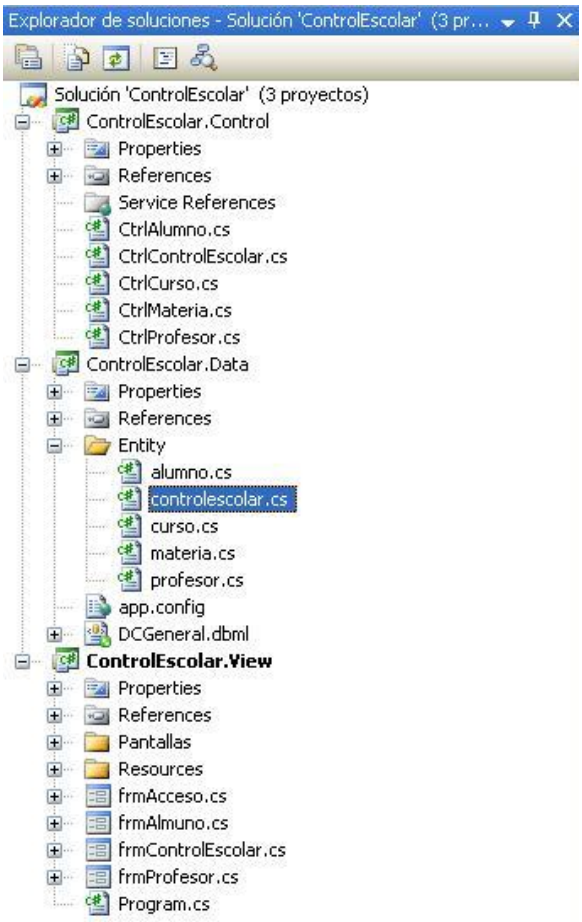

Figura 63: Modelo MVC en el Explorador de soluciones Visual Studio 2008

6.4.-Consulta, Actualización, Eliminación e Inserción en la Base de Datos con C# y LINQ to SQL

Continuando con el ejemplo de Control Escolar, se crea un formulario de **Alumno** en el modulo de **ControlEscolar.View**. En este formulario, ver figura 64, se agregaran las funciones básicas que usan las Bases de Datos, tales como insertar, consultar, actualizar y eliminar.

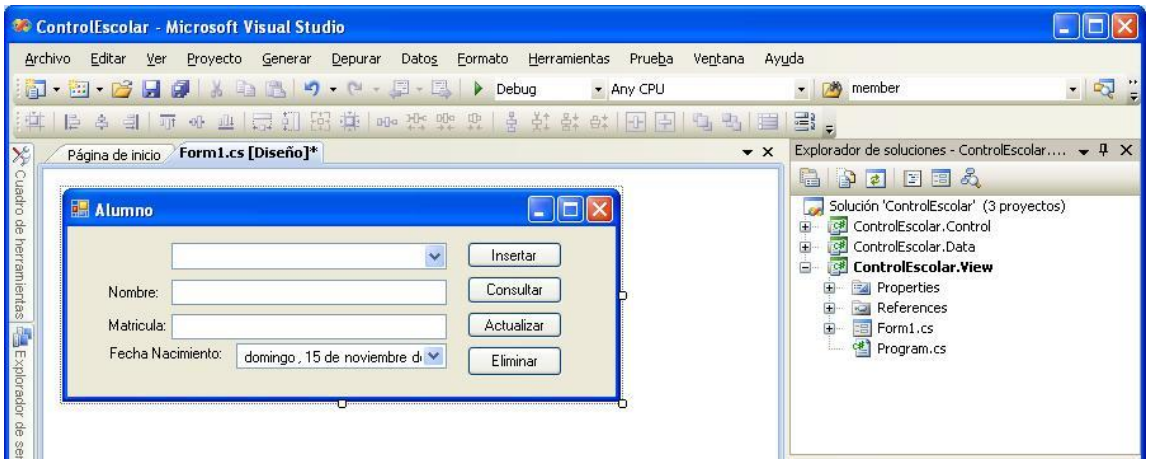

Figura 64: Formulario de Alumno

En el formulario, ver figura 64, se incluyen los botones que se encargan de llamar a la función correspondiente con su nombre, también se cuenta con las **cajas de texto**, un **control de fechas** y una **caja de selección**. En la **caja de selección** se tiene definido las acciones a realizar, así se habilitaran o deshabilitaran los controles según sea el caso. Por ejemplo, si se desea insertar un **Alumno**, se selecciona la acción y se habilitan las **cajas de texto** y el botón **insertar.**

Al agregar los controles en el formulario se procede a incluir las referencias, esto debido a que el modelo de datos se tiene en el proyecto de **ControlEscolar.Data**, si esta referencia no se incluye no se puede acceder a las clases ahí contenidas. Dicha referencia se agrega en la sección del **explorador de soluciones** en el proyecto **ControlEscolar.View** en la carpeta referencias, que se genera junto con el proyecto, ver figura 65.

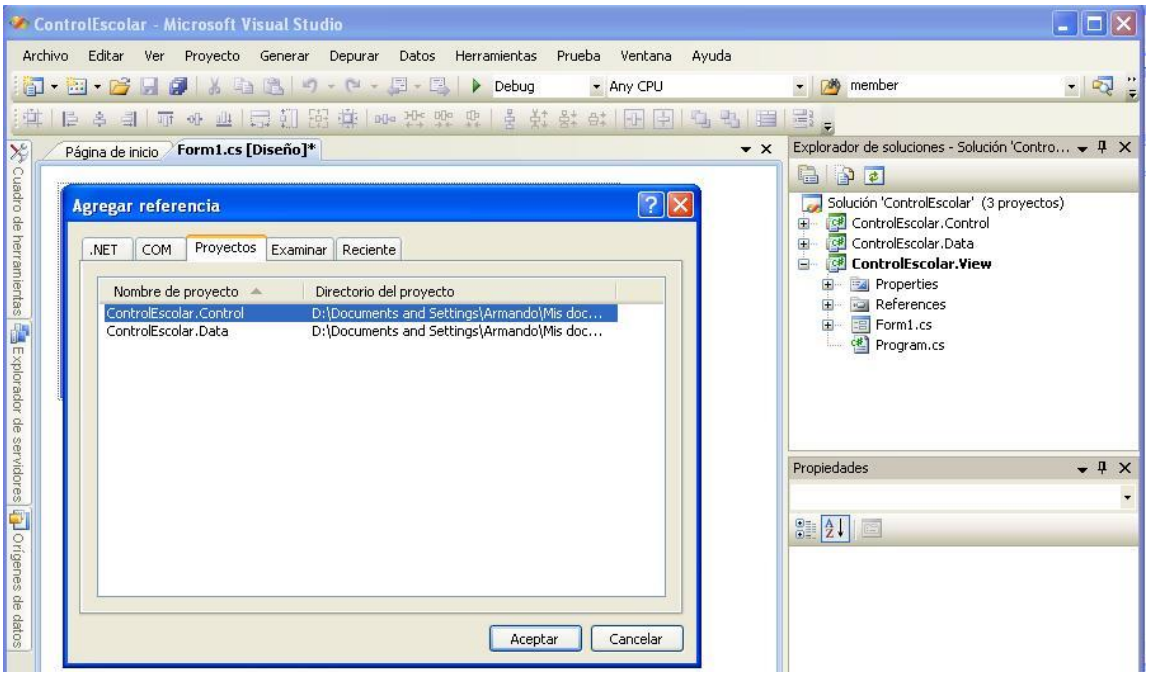

Figura 65: Agregar referencias entre proyectos

Una vez agregada la referencia en el proyecto, se agrega en el código del formulario de Alumno, debido a que solo sea indicado que se usara en el proyecto pero no en el formulario, también se tiene que agregar la librería System.Data.Linq ya que de esta se derivan las clases como DataContex necesarias para el desarrollo de la aplicación, ver figura 66.

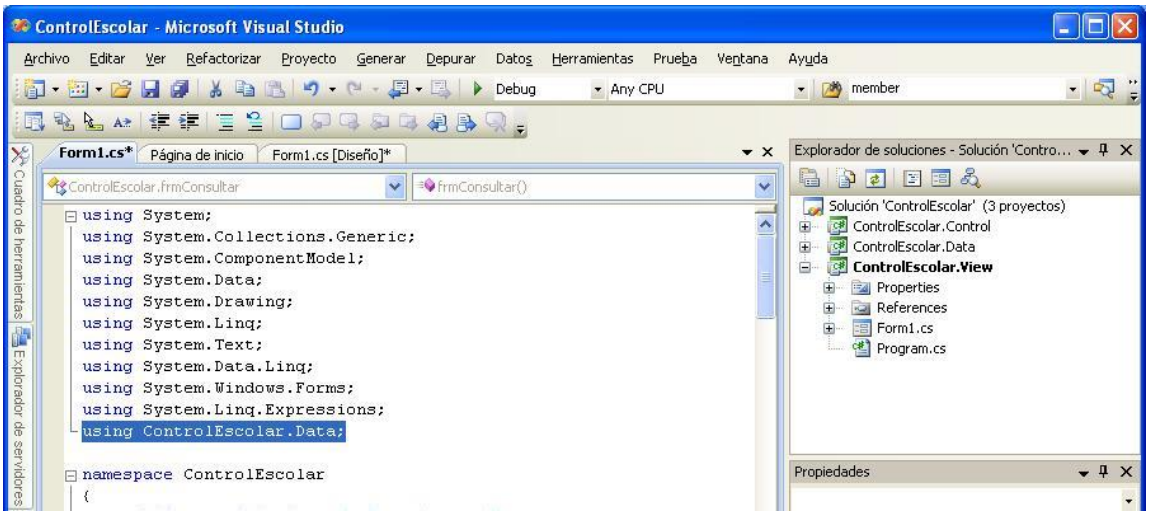

Figura 66: Agregar referencia en código

En la figura 67 se puede apreciar las funciones que se desarrollan para ejemplificar las acciones que usualmente se usan para la manipulación de la información en las Bases de Datos.

```
using System.Text;
using System.Data.Linq;
using System.Windows.Forms;
using System.Linq.Expressions;
using ControlEscolar.Data;

namespace ControlEscolar
{
    public partial class frmAlumno : Form
    {
        //Constructor
        public frmAlumno()...

        //Se crea el Contexto con la Base de Datos
        DCGeneralDataContext dt = new DCGeneralDataContext();

        //Funcion encargada de habilitar o deshabilitar los controles
        //dependiendo de la opcion seleccionada.
        private void cmbSeleccionar_SelectedIndexChanged(object sender, EventArgs e)...
        //Ejemplo de una consulta
        public void btnConsultar_Click(object sender, EventArgs e)...
        //Ejemplo de insertar un registro
        private void btnInsertar_Click(object sender, EventArgs e)...
        //Ejemplo de actualizar un registro
        private void btnActualizar_Click(object sender, EventArgs e)...
        //Ejemplo de eliminar un registro
        private void btnEliminar_Click(object sender, EventArgs e)...

    }
}
```

Figura 67: Estructura de las acciones en el formulario alumno

En esta estructura se tiene:

- Las librerías que son las clases útiles para definir variables, constantes y funciones dentro del programa, indicadas con **using**.
- Se define la clase principal **frmAlumno** que hereda de la clase form,
- En la clase se tiene el constructor **public frmAlumno ()**, que se genera en automático al crear el formulario.
- Se agrega el contexto con la Base de Datos **DCGeneralDataContext dt = new DCGeneralDataContext()**, la declaración del DataContext,
- Se tiene la función encargada de indicar la acción a realizar **cmbSeleccionar_SelectedIndexChanged()**.
- Y las funciones con las acciones consultar, insertar, actualizar y eliminar.

En la figura 68 se muestra la función de selección de la acción, así cada que se opte por una acción se habilitaran o deshabilitaran los controles.

```
private void cmbSeleccionar_SelectedIndexChanged(object sender, EventArgs e)
{
    txtMatricula.Enabled = false;
    dteFechaNacimiento.Enabled = false;
    btnActualizar.Enabled = false;
    btnInsertar.Enabled = false;
    btnEliminar.Enabled = false;
    btnConsultar.Enabled = false;

    if (cmbSeleccionar.SelectedItem.Equals("Actualizar"))
    {
        txtMatricula.Enabled = true;
        dteFechaNacimiento.Enabled = true;
        btnActualizar.Enabled = true;
    }
    if (cmbSeleccionar.SelectedItem.Equals("Eliminar"))
    {
        txtNombreAlumno.Enabled = false;
        txtMatricula.Enabled = true;
        btnEliminar.Enabled = true;
    }
    if (cmbSeleccionar.SelectedItem.Equals("Consultar"))
    {
        btnConsultar.Enabled = true;
    }
    if (cmbSeleccionar.SelectedItem.Equals("Insertar"))
    {
        txtMatricula.Enabled = true;
        dteFechaNacimiento.Enabled = true;
        btnInsertar.Enabled = true;
```

Figura 68: Función de selección de acción

LINQ tiene distintas alternativas de realizar consultas, en la figura 69 se muestra una de estas formas. Para el ejemplo, en el momento de ejecución de la aplicación se debe capturar en la Caja de Texto(**txtNombreAlumno**) el nombre del alumno que desea buscar, al dar clic sobre el botón de consulta, verificara mediante código que la caja de texto no esté vacía, si está vacía indicara que "no se capturo un nombre a consultar", sino lo está procede a realizar la consulta, el resultado de esta se almacena en la variable **alumnos,** el DataContext trae las tablas mapeadas y la consulta se realizara sobre este, es decir sobre **dt.alumno**, que es la tabla de alumno. La sintaxis de la consulta indica que seleccione **alum** de la entidad **dt.alumno** donde **alum** sea igual al dato capturado en la caja de texto, si existe un valor lo almacenará de lo contrario no asignara ningún valor, es importante mencionar que el contexto con la Base de Datos se genero al iniciar la ejecución de la aplicación.

```
//Ejemplo de una consulta
public void btnConsultar_Click(object sender, EventArgs e)
{

    if (txtNombreAlumno.Text.Trim().Length == 0)
    {
        MessageBox.Show("No se capturo nombre a consultar");
    }
    else
    {
        //Ejemplo de Consulta
        var alumnos = from alum in dt.alumno
                      where alum.nomAlumno == txtNombreAlumno.Text
                      select alum;
    }

}
```

Figura 69: Ejemplo de una consulta

Para insertar un nuevo registro, ver figura 70, se validara mediante código que las cajas de texto contenga un valor, al verificar que no estén vacios, se creara una entidad llamada **alum** que es de tipo **alumno**, esta es una clase que se genero durante el mapeo es por eso que no es necesario crearla, al menos que se quiera obtener algún valor especifico y no todas las propiedades de esta clase, pero aun así esta seguirá estando activa en el mapeo general. Se le asignara a la entidad **alum** cada una de las propiedades que la clase tiene y que están representadas en el formulario. De las cajas de texto se obtienen los datos del nuevo registro, en **alum.matAlumno** se le asignara el valor obtenido de la caja de texto (**txtMatricula**) que representa la matricula del alumno, así será para cada propiedad de la entidad. Ya que se tiene la entidad **alum** completa con sus propiedades asignadas, se inserta mediante la instrucción **dt.alumno.InsertSubmit(alum)** en la entidad alumno del Datacontext, de esta manera se tendrá los datos ya en el contexto solo falta realizar la acción en la Base de Datos y esto se realiza con la instrucción **dt.Submitchanges().**

```
//Ejemplo de insertar un registro
private void btnInsertar_Click(object sender, EventArgs e)
{

    if (txtNombreAlumno.Text.Trim().Length == 0 && txtMatricula.Text.Trim().Length == 0)
    {
        MessageBox.Show("No hay valor a insertar");
    }
    else
    {
        try
        {
            alumno alum = new alumno();
            alum.matAlumno = txtMatricula.Text;
            alum.fechaNacimiento = dteFechaNacimiento.Value;
            alum.nomAlumno = txtNombreAlumno.Text;
            dt.alumno.InsertOnSubmit(alum);
            dt.SubmitChanges();
            MessageBox.Show("Se agregaron correctamente los datos");

        }
        catch (Exception _e)
        {
            MessageBox.Show("Se ecnontro el siguiente error", _e.Message);
        }

    }
}
```

Figura 70: Ejemplo para insertar un registro

Para eliminar un registro, ver figura 71, se crea de igual forma una entidad **alum** de tipo **alumno** que es la que eliminara en el DataContext y posteriormente en la Base de Datos, para eliminarlo se debe de tener un criterio, en este caso se eliminara el alumno que contenga cierto número de matrícula. Este criterio implica realizar una consulta sobre el DataContext y buscar la entidad que corresponda al valor solicitado, esta consulta es; **dt.alumno.Single(r => r.matAlumno == txtMatricula.Text)**, que mediante la instrucción **Single** devuelve el elemento que cumpla con la condición especificada, es decir si en el DataContext encuentra el valor que se tiene en la caja de texto matricula (**r.matAlumno == txtMatricula.Text**) devolverá esa entidad(**r**) y la asignara a la entidad **alum**. Una vez asignada la entidad se agenda para eliminar con la instrucción **dt.alumno.DeleteOnSubmit(alum),** para realizarlo en la Base de Datos se hace con la instrucción **dt.SubmitChanges()**. En cada función de las ya mencionadas se agrega **catch** para capturar los posibles errores que se presenten al realizar la acción solicitada, enviando un mensaje con el error encontrado si es que existe.

```
//Ejemplo de eliminar un registro
private void btnEliminar_Click(object sender, EventArgs e)
{
    if (txtMatricula.Text.Trim().Length == 0)
    {
        MessageBox.Show("No se ha capturado el numero de Matricula a eliminar");
    }
    else
    {
        try
        {
            alumno alum = dt.alumno.Single(r => r.matAlumno == txtMatricula.Text);
            dt.alumno.DeleteOnSubmit(alum);
            dt.SubmitChanges();
            MessageBox.Show("Se elimino correctamente el registro");
        }
        catch (Exception _e)
        {
            MessageBox.Show(_e.Message);
        }
    }
}
```

Figura 71: Ejemplo para eliminar un registro

Para actualizar un registro, ver figura 72, se valida que las cajas de texto no estén vacías, además el criterio de actualización se realizara también mediante la búsqueda de la matricula para ello se realiza la misma consulta que en el ejemplo anterior **dt.alumno.Single(r => r.matAlumno == txtMatricula.Text)** al encontrar el valor a actualizar se le asigna a la entidad **alum**, para este ejemplo solo se actualizara el nombre del alumno (**alum.nomAlumno)** asignándole el valor que la caja de texto(**txtNombreAlumno**) tenga, para realizarlo en la Base de Datos se hace mediante **dt.SubmitChanges()**.

```
//Ejemplo de actualizar un registro
private void btnActualizar_Click(object sender, EventArgs e)
{
    if (txtNombreAlumno.Text.Trim().Length == 0 && txtMatricula.Text.Trim().Length == 0)
    {
        MessageBox.Show("Verifique sus datos a actualizar");
    }
    else
    {
        try
        {
            alumno alum = dt.alumno.Single(r => r.matAlumno == txtMatricula.Text);
            alum.nomAlumno = txtNombreAlumno.Text;
            dt.SubmitChanges();
            MessageBox.Show("Se actualizaron correctamente los datos");
        }
```

Figura 72: Ejemplo para actualizar un registro

Cuando se crean las consultas y se obtienen los objetos como en los ejemplos anteriores, LINQ to SQL estará pendiente de los cambios o actualizaciones que se le hagan a los objetos. Se pueden hacer tantas consultas y cambios como se deseen usando la clase DataContext de LINQ to SQL, sabiendo que dichos cambios serán supervisados a la vez.

El seguimiento de cambios de LINQ to SQL ocurre en el lado del consumidor y no en la Base de Datos. Es decir, no está consumiendo ningún recurso de la Base de Datos mientras se use, tampoco se tiene que cambiar/instalar nada en la Base de Datos para que esto funcione.

Conclusiones

Las herramientas para el desarrollo de software que van surgiendo día a día facilitan en buena medida la elaboración de sistemas de información, es por ello que hay que estar informado del uso de estas herramientas y técnicas para implementarlas en los nuevos desarrollos. LINQ como una herramienta de desarrollo contribuye a agilizar los procesos en el ámbito de programación y nivel de ejecución de aplicaciones. En este documento solo se muestra una de las opciones que da LINQ para el tratamiento de los datos, mediante LINQ to SQL, aunque se puede combinar con los distintos orígenes de datos ya sea con el uso de archivos XML, con LINQ to XML, con objetos, mediante LINQ to Objects y de esta forma elaborar sistemas más robustos.

En este documento se ejemplifico el manejo de los datos mediante LINQ to SQL, ya que es el que mapea la información de la Base de Datos al entorno de programación, y en la mayoría de los sistemas actuales tienen como contendor de la información a las Bases de Datos, es por ello la importancia del mapeo LINQ to SQL sobre los otros orígenes de datos que también maneja LINQ.

Tabla de Figuras

Tablas Descriptivas

Glosario de términos.

Biblioteca MFC [Microsoft Foundation Class library]

Biblioteca de clases de C++ que forma un contenedor orientado a objetos en torno a partes importantes de la API de Windows y proporciona un marco en el que se van a generar las aplicaciones.

Boxing

Conversión de una instancia de tipo de valor en un objeto, lo que implica que la instancia transportará información completa de tipos en tiempo de ejecución y se asignará en el montón. La instrucción box del conjunto de instrucciones del Lenguaje intermedio de Microsoft (MSIL) convierte un tipo de valor en un objeto; para ello, hace una copia del tipo de valor y la incrusta en un objeto recién asignado.

C#

Lenguaje de programación diseñado para crear aplicaciones empresariales que se ejecutan en .NET Framework. C#, que es una evolución de C y C++, garantiza la seguridad de tipos y está orientado a objetos.

Clave externa [foreign key]

Clave en una tabla de Base de Datos que procede de otra tabla. Esta clave hace referencia a una clave concreta, normalmente la clave principal, de la tabla que se está utilizando.

Common Language Runtime(CLR)

Motor que es el núcleo de la ejecución de código administrado. El motor en tiempo de ejecución proporciona al código administrado servicios como integración entre varios lenguajes, seguridad de acceso a código, administración de la duración de los objetos, y compatibilidad con la depuración y la generación de perfiles.

Common Language Specification(CLS)

Subconjunto de funciones del lenguaje admitidas por Common Language Runtime, incluyendo funciones comunes de varios lenguajes de programación

orientados a objetos. Se garantiza que las herramientas y los componentes compatibles con CLS pueden interoperar con otras herramientas y componentes compatibles con CLS.

Common Object File Format (COFF)

En programación de 32 bits, formato de archivos ejecutables (imagen) y de objeto que puede transportarse a distintas plataformas. La implementación de Microsoft se denomina formato de archivo ejecutable portable (PE).

Compatible con CLS [CLS-compliant]

Código que expone públicamente sólo las funciones de lenguaje incluidas en la especificación Common Language Specification. La compatibilidad con CLS puede aplicarse a clases, interfaces, componentes y herramientas.

Compilación JIT [JIT compilation]

Compilación que convierte el Lenguaje intermedio de Microsoft (MSIL) en código máquina cuando se requiere el código en tiempo de ejecución.

Consulta [query]

Expresión con forma de consulta o de consulta basada en método (o una combinación de las dos) que extrae información de un origen de datos.

Diseñador relacional de objetos [Object Relational Designer]

Herramienta que proporciona una superficie de diseño visual para crear clases de entidades y asociaciones (relaciones) de LINQ to SQL en función de los objetos de una Base de Datos. El Diseñador relacional de objetos también proporciona una funcionalidad mediante la que se asignan procedimientos almacenados y funciones.

Documento del esquema XML [XML Schema Document]

Especificación que describe los tipos complejos utilizados en un método Web y por tanto habilita la interoperabilidad entre clientes y servicios Web generados en distintas plataformas, mediante la adhesión a un sistema de tipos común.

Expresión de consulta [query expression]

Expresión que trabaja con tipos enumerables y puede generar estos tipos. La expresión se compone de cláusulas que son similares a las cláusulas de SQL y que se basan en palabras claves del lenguaje.

Expresión lambda [lambda expression]

Función inline de la tecnología Language-Integrated Query (LINQ) que utiliza el operador => para separar los parámetros de entrada del cuerpo de la función y que se puede convertir en tiempo de compilación en un delegado o un árbol de expresión.

Expresión regular [regular expression]

Notación concisa y flexible para buscar y reemplazar modelos de texto. Esta notación incluye dos tipos de caracteres básicos: caracteres de texto literales (normales), que indican texto que debe existir en la cadena de destino, y metacaracteres, que indican el texto que puede variar en la cadena de destino. Puede utilizar expresiones regulares para analizar rápidamente grandes cantidades de texto con el fin de buscar modelos de caracteres específicos, para extraer, modificar, reemplazar o eliminar subcadenas de texto, o para agregar las cadenas extraídas a una colección con el fin de generar un informe.

Genéricos [generics]

Característica de Common Language Runtime, conceptualmente similar a las plantillas de C++, que permite que las clases, estructuras, interfaces y métodos tengan marcadores de posición (parámetros de tipo genérico) para los tipos de datos que almacenan y manipulan. Los tipos genéricos son una forma de tipos parametrizados.

Inicializador de objeto [object initializer]

Construcción de lenguaje de Visual Basic y C# que habilita la inicialización de variables miembro de un objeto en la misma instrucción en la que se crea el objeto.

Interface

Tipo de referencia que define un contrato. Otros tipos implementan una interfaz para garantizar que admiten ciertas operaciones. La interfaz especifica los

miembros que las clases u otras interfaces que los implementan deben suministrar. Al igual que las clases, las interfaces pueden contener como miembros métodos, propiedades, indizadores y eventos.

Language Integrated Query (LINQ)

Sintaxis de consulta que define un conjunto de operadores de consulta que permiten expresar operaciones de cruce seguro, filtro y proyección de manera directa y declarativa en cualquier lenguaje de programación basado en .NET.

Lenguaje de marcado extensible (XML) [Extensible Markup Language (XML)]

Subconjunto del Lenguaje de marcado generalizado estándar (SGML) optimizado para su uso a través del Web. XML proporciona un método uniforme para describir e intercambiar datos estructurados que es independiente de las aplicaciones o los proveedores.

LINQ to ADO.NET

Tecnología que permite consultar cualquier objeto enumerable en ADO.NET utilizando el modelo de programación de LINQ. LINQ to ADO.NET se compone de dos tecnologías de LINQ relacionadas: LINQ to DataSet y LINQ to SQL.

LINQ to DataSet

Tecnología LINQ que facilita y acelera las consultas a los datos almacenados en la memoria caché de un objeto DataSet. Las consultas se expresan en el propio lenguaje de programación y no como literales de cadena incrustados en el código de aplicación.

LINQ to Objects

Uso de LINQ para consultar datos en memoria, como matrices y listas.

LINQ to SQL

Tecnología LINQ que proporciona una infraestructura en tiempo de ejecución para administrar datos relacionales como objetos. En LINQ to SQL, el modelo de datos de una Base de Datos relacional se asigna a un modelo de objetos expresado en el lenguaje de programación del desarrollador.

LINQ to XML

Interfaz de programación en memoria que permite trabajar con XML desde los lenguajes de programación de .NET Framework. Puede consultar y modificar un documento, y a continuación, después de modificarlo, guardarlo en un archivo o serializar el resultado y enviarlo a través de Internet.

Metadatos [metadata]

Información que describe todos los elementos administrados por Common Language Runtime: un ensamblado, el archivo cargable, el tipo, el método, etc. Esto puede incluir información necesaria para la depuración y la recolección de elementos no utilizados, así como atributos de seguridad, cálculo de referencias de datos, definiciones extendidas de clases y miembros, enlace de versión y otra información requerida por el motor en tiempo de ejecución.

Namespace

Esquema de nombres lógico para agrupar los tipos relacionados. .NET Framework utiliza un esquema de nombres jerárquico para agrupar los tipos en categorías lógicas de funcionalidad relacionada, como la tecnología ASP.NET o la funcionalidad de interacción remota.

OLE

Mecanismo para transferir y compartir información entre aplicaciones pegando la información creada en una aplicación en un documento creado en otra aplicación, como una hoja de cálculo o un archivo de un procesador de texto.

SELECT

Instrucción del lenguaje de consulta de WMI que se utiliza para recuperar información. SQL admite las consultas en varias tablas, pero WQL admite sólo consultas de clases únicas.

SubmitChanges()

Calcula el conjunto de objetos modificados que se van insertar, actualizar o eliminar, y ejecuta los comandos adecuados para implementar los cambios en la Base de Datos.

Siglario

Termino	Ingles	Español
ADO	Active Data Objects	Objetos activos de datos
ANSI	American National Standards Institute	Instituto de Normalización Nacional Americano
API	Application Programming Interface	Interfaz de programación de aplicaciones
BCL	Base Class Library	Librería base de clases
CLR	Common Language Runtime	Lenguaje Común de Ejecución
CLS	Common Language Specification	Especificación del Lenguaje Común
COFF	Common Object File Format	Sistema de Archivos de Objeto Común
CTS	Common Type System	Sistema de Tipo Común
DBML	Database Markup Language	Lenguaje de Marcado de Base de Datos
DBMS	Database Management system	Sistema Gestor de Bases de Datos (SGBD)
DOM	Document Object Model	Modelo de Objetos de Documento
DTD	Document Type Defination	Definición de Tipo de Documento
HTML	Hypertext Markup Language	Lenguaje de Marcas de Hipertexto
IDL	Interface Description Language	Lenguaje de Definición de Interfaces
IEEE	Institute for Electrical and Electronical Engineers	Instituto de Ingenieros Eléctricos y Electrónicos
JDBC	Java Database Connectivity	Conectividad con Bases de Datos en Java
JIT	Just In Time	Justo en Tiempo
LDD	Data Definition Language	Lenguaje de Definición de Datos
LINQ	Language Integrated Query	Lenguaje de Consultas Integradas
LMD	Data Manipulation Language	Lenguaje de manipulación de datos
MSIL	Microsoft Intermediate Language	Lenguaje Intermedio de Microsoft
MVC	Model View Controller	Modelo Vista Controlador
ODBC	Open Database Connectivity	Conectividad Abierta de Bases de Datos
OID	Object Identifier	Identificador de Objetos
OLE	Object Linking y Embedding	Vinculación e Incrustación de Objetos
ORM	Object Relational Mapping	Mapeo Relacional-Objeto
POO	Object oriented programming	Programación Orientada a Objetos
SQL	Structured query language	Lenguaje de Consulta Estructurado
UML	Unified Modeling Language	Lenguaje de Modelado Unificado
XML	Extensible Markup Language	Lenguaje de Marcado Extensible

Bibliografía

AGARWAL, V. V., HUDDLESTON, J., RAGHURAM, R., GILANI, S. F., PEDERSEN, J. H., & REID, J. (2008). *Beginning C# 2008 Databases From Novice to Professional* (Primera edición ed.). Apress.

BARNES, D. J., & KOLLING, M. (2007). *Programación Orientada a Objetos con Java* (Tercera edición ed.). (B. I. BRENTA, Trad.) Pearson.Prentice Hall.

CAMPS PARÉ, R., CASILLAS SANTILLAN, L. A., & COSTAL COSTA, D. (2005). *Bases de Datos* (Primera edición ed.). Eureca Media.

DOUGLAS, B. K. (2009). *Object-relational mapping articles*. Recuperado el 20 de Agosto de 2009, de Barry & Associates, Inc: http://www.service-architecture.com/object-relational-mapping/articles/

FERGUSON, J., PATERSON, B., & BERES, J. (2003). *La biblia C#* (Primera edición ed.). Anaya.

JACOBSON, I., BOOCH, G., & RUMBAUG, J. (2000). *El proceso unificado de desarrollo de software* (Primera edición ed.). (S. Sánchez, M. Á. Sicilia, C. Canal, & F. J. Duran, Trads.) Addison Wesly.

JOYANES AGUILAR, L. (1996). *Programación Orientada a Objetos* (Primera edición ed.). McGraw Hill.

KUMAR, N. S. (2007). *LINQ Quickly (A practical guide to programming Language Integrated Query with C#)* (Primera edición ed.). Packt Publishing.

LEYTON G., E. (Junio de 2002). *Ingeniería de Software con UML*. Recuperado el 23 de Mayo de 2009, de Eduardo Leyton.: http://www.eduardoleyton.com

Manual Basico del Lenguaje SQL. (2007). Recuperado el 1 de Abril de 2009, de Escuela de Ingenria de Colombia Julio Garivito: http://laboratorio.is.escuelaing.edu.co/labinfo/doc/Manual_Basico_del_Lenguaje_SQL.pdf

Mapeo objeto-relacional. (2009). Recuperado el 8 de Julio de 2009, de Wikimedia. La enciclopedia libre: http://es.wikipedia.org/wiki/Mapeo_objeto-relacional

MEHTA, V. P. (2008). *Pro LINQ Object Relational Mapping with C# 2008* (Primera edicion ed.). Apress.

Modelo Vista Controlador. (2009). Recuperado el 13 de Julio de 2009, de Wikipedia. La enciclopedia libre: http://es.wikipedia.org/wiki/Modelo_Vista_Controlador

MONTEJO, E. M. (28 de Marzo de 2004). *Manual del lenguaje SQL del motor Microsoft Jet*. Recuperado el Octubre de 2009, de MVP-ACCESS: http://www.mvp-access.es/softjaen/manuales/sql/index.htm

MSDN Library C#. (2009). Recuperado el 4 de Febrero de 2009, de Microsft Corporation: http://msdn.microsoft.com/en-us/vcsharp/default.aspx

MSDN Library Framework. (2009). Recuperado el 5 de Marzo de 2009, de Microsoft Corporation: http://msdn.microsoft.com/es-mx/library/zw4w595w.aspx

MSDN Library LINQ. (2009). Recuperado el 8 de Febrero de 2009, de Microsft Corporation: http://msdn.microsoft.com/es-mx/library/zw4w595w.aspx

PILAROSI, P., & RUSSO, M. (2007). *Introducing Microsoft LINQ* (Primera Edicion ed.). Microsoft Press.

POSADAS, M. (2000). *Introducción al Lenguaje XML* (Primera edición ed.).

PRESUMAN, R. S. (2002). *Ingeniería del Software. Un Enfoque Práctico* (Quinta edición ed.).

Proceso Unificado de Rational. (Julio de 2009). Recuperado el Julio de 2009, de Wikimedia. La enciclopedia libre: http://es.wikipedia.org/wiki/Proceso_Unificado_de_Rational

SCHMULLER, J. (2003). *Aprendiendo UML en 24 horas* (Primera edición ed.). Prentice Hall.

SILBERSCHATZ, A., KORTH, H. F., & SUDARSHAN, S. (2002). *Fundamentos de Base de Datos* (Cuarta edición ed.). (F. S. PÉREZ, Trad.) Mc Graw Hill.

SOMMERVILLE, I. (2005). *Ingeniería del Software* (Septima edición ed.). (A. B. Maria Isabel Galipenso, Trad.) Pearson.

TROELSEN, A. (2008). *Pro C# 2008 and the .NET 3.5 Plataform (Exploring the .NET universe using curly brackets)* (Cuarta edicion ed.). Apress.