Persistencia Automática de Objetos .NET

Fernández, Gonzalo Rafael
Universidad Tecnológica Nacional, Facultad Regional Córdoba

Abstract

Este proyecto encara el problema de la persistencia de objetos en repositorios tradicionales, como los motores de bases de datos relacionales, a través del concepto de abstracción para lograr producir un código más puro en cuanto a paradigma. Propone un aprovechamiento de las técnicas de "reflección" de los lenguajes modernos para implementar una persistencia automática y más inteligente que ahorre tiempos de desarrollo y permita lograr código más claro, mantenible y Type-Safe.
Además, amplía las posibilidades en la implementación de modelos al soportar estructuras más complejas y arbitrarias como pueden ser los grafos.

Palabras Clave

Programación Orientada a Objetos. Persistencia. Base de Datos. Type-Safe. Modelo Relacional. Caching. Late Binding. Reflection.

Introducción

En el moderno mundo de la programación, el Paradigma Orientado a Objetos (POO) es el que domina la metodología de desarrollo y es complementado con nuevas estrategias de arquitectura como la Orientada a Servicios (SOA), la Orientada a Aspectos (AOP), etc. También encontramos al POO como base de procesos de desarrollo, como el Proceso Unificado de Desarrollo de Software (RUP), y lenguajes de modelado como el Lenguaje Unificado (UML).

Este paradigma es apto para cualquier tipo de sistemas. Tanto para los que manejan y generan muchos datos a través del tiempo como para los que invierten su procesamiento en cálculos con datos temporales. En aquellos que utilizan grandes volúmenes de datos persistentes, generalmente en bases de datos relacionales, el problema se encuentra al querer almacenar los objetos generados en dicho repositorio. Este es el tan nombrado problema del *Diferencial de Impedancia* (Impedance Mismatch), generalmente entre la forma en que los objetos "viven" y son creados en memoria y la forma de almacenar los datos por parte de los Motores de Bases de Datos Relacionales (RDBMS).

Figura 1

En la figura anterior, podemos ver los conceptos que integran el POO y las características que se requieren en la Persistencia de Objetos.

Se pueden encontrar muchos intentos de solución al respecto, sin que ninguno pueda considerarse definitivo; por lo tanto, es un problema que amerita ser investigado y tratado. La mayoría de éstos encara el problema con técnicas de mapeo, y son los llamados Mapeadores Objeto-Relacional (ORM). Este sistema consiste en identificar cómo puede un objeto "encajar" en las estructuras de un repositorio que no soporta

objetos y configurar el sistema para que la persistencia de los objetos (guardar) y la materialización (recuperar) se haga a través de dicho "mapa" de correspondencias. Un producto muy popular que implementa ORM es Hibernate.

Uno de los problemas de los ORM es, por un lado, el trabajo previo necesario para construir los mapas de correspondencia (el tiempo ahorrado al no emplear directamente SQL, se pierde en la definición de los mapeos). Otro inconveniente es la limitación a la que somete al programador a la hora de implementar el POO: la herencia, la identidad de objetos, los tipos complejos, las asociaciones de multiplicidad arbitraria, las asociaciones arbitrarias (grafos), etc. no pueden formar parte del diseño de su sistema porque el repositorio que en última instancia persistirá la información no lo soporta.

Otra solución con un enfoque diferente, es la que propone serializar los objetos directamente en algún tipo de archivo como los de formato XML. Este mecanismo no sería apto para grandes volúmenes de datos y para aquellos que requieran consultas complejas. Esta técnica toma a los objetos y construye una representación ya sea binaria o como cadena de caracteres que puede ser guardada en un archivo en disco. Un producto muy popular que hace uso de este sistema es db4o.

El proyecto descripto en este artículo encara el problema desde un punto de vista diferente. Se propone como objetivo lograr el máximo nivel de abstracción posible en cuanto al repositorio de datos, de modo de poder aplicar la mayor cantidad de características del POO sin más restricciones que las que el propio dominio del problema imponga. Por lo tanto, se pueden implementar modelos que reflejan mejor la realidad del problema. Como objetivos secundarios, persigue el aumento de productividad al eliminar cualquier trabajo referido a la persistencia de datos, la seguridad en cuanto a tipos a la hora de programar (Type-Safe) y una mayor mantenibilidad y claridad de código.

Elementos del Trabajo y metodología

El proyecto consiste en una Librería de clases .NET, que puede usarse con cualquier lenguaje basado en CLR (C#, Visual Basic, J#, Boo, etc.) y se utiliza a la hora de desarrollar Clases de Negocio (Business Classes). Esta librería provee a nuestras propias clases de funcionalidad suficiente para ser persistidas automáticamente sin configuración alguna (Cero SQL).

En cuanto a la materialización de objetos que ya han sido persistidos, el objetivo de *abstracción* impide que se usen técnicas propias del repositorio para esta tarea. Para esto implementa un conjunto de funciones que permiten construir en forma genérica la mayoría de las consultas habituales, con la ventaja de que es Type-Safe casi en su totalidad (a diferencia del uso de cadenas SQL) y que no requiere más conocimiento que el modelo de Clases de Negocio (Cero SQL, otra vez).

También se brinda soporte a técnicas como el *Late Binding,* o Enlace Tardío, que es posible gracias a implementar el concepto de i*dentidad de objetos* que el POO define. Esta técnica bastante conocida permite que al cargar un objeto en memoria, el resto de los objetos asociados se carguen solo en el momento en que se deseen utilizarlos.

El proyecto incluye otras funcionalidades destinadas a restar tiempos de desarrollo, que son las típicas destinadas al manejo de usuarios y caching de datos automático.

Desde el punto de vista del programador, la Librería intenta dar la sensación de que los objetos con los que trabaja el sistema se encuentran siempre en memoria y nunca salieron de allí. Por lo tanto, para usar

cualquier objeto tanto para mostrar sus datos como para modificarlos, simplemente hay que utilizarlo obteniendo nuevamente una referencia (o puntero) a él como si nunca hubiese salido de la memoria principal sin copiar datos de una tabla a un objeto o al revés (la Librería hace esto automática y transparentemente).

Hablando de un caso práctico, si el modelo diseñado con UML (o cualquier otro lenguaje de modelado) le indica al programador que debe implementar una clase `Libro` con las propiedades `Nombre` y `Editorial`, comenzará creando una clase con los dos atributos privados (que definen su *estado*) y encapsulará los mismos mediante dos propiedades públicas. La clase en este momento está terminada y ya posee funcionalidad para que sus objetos sean persistidos y recuperados en cualquier momento sin más código. La identidad de los objetos está garantizada y no requiere de datos artificiales (como claves primarias). Esta implementación refleja el modelo real. Por ejemplo, en la práctica podemos contar con dos libros iguales que salieron de la misma edición. Éstos libros no poseen ninguna clave o número de identificación propia cada uno, como el que tendría una implementación basada en el modelo relacional de datos. ¿Con cuál de estos dos objetos trabajamos en un momento dado? La elección es igual que en la vida real, tomamos el que esté más a mano porque ninguno tiene hasta el momento ninguna característica que nos haga preferirlo. Éste es un ejemplo de uso del concepto de *identidad de objetos* en el POO.

Las asociaciones se manejan con esta misma simplicidad. La Librería provee opciones para aplicar integridad en las asociaciones y unicidad de datos (equivalente a claves únicas del modelo relacional).

La forma en que estos objetivos se llevan a cabo, es mediante la descomposición de los objetos al momento de persistirlos, y la reintegración de los mismos al momento de recuperarlos. Se inspecciona cada objeto con técnicas de *reflección* y se guarda su estado interno, así como las asociaciones a otros objetos. Las llamadas a persistir son recursivas, por lo que una compleja estructura de objetos en un momento dado puede ser persistida con una sola instrucción.

La arquitectura empleada en el proyecto, permite que la Librería pueda ser extendida para utilizar otro tipo de repositorio aparte de los RDBMS. En este momento se puede trabajar con motores como Firebird, SQL Server, Oracle y cualquiera que pueda ser accedido mediante OleDb u ODBC.

Resultados

El producto final es un conjunto de librerías DLL en .NET, compiladas mediante el compilador de C# de Microsoft y probadas tanto en el framework .NET (bajo Windows) como en Mono (bajo Linux), permitiendo el desarrollo multiplataforma. El programador se encuentra frente a una herramienta para agilizar el desarrollo. Le ahorra el trabajo de realizar manualmente las tareas de persistencia de sus datos, además de evitarle la posibilidad de errores en la misma. Permite obtener un código más organizado y mantenible, más fácil de transmitir a otros programadores por ser más claro.

Se planea la inclusión de funciones para asistir a la validación de los datos antes de ser persistidos, así como facilidades para trabajar en ambientes distribuídos mediante el uso de Web Services o .NET Remoting. Esto último podrá permitir llevar la lógica y el procesamiento requerido en la persistencia en sí a equipos servidores con mayor capacidad.

Discusión

Otra posibilidad a la hora de trabajar con datos persistentes mediante el POO, es utilizar algún motor con soporte nativo de objetos. Los DBMS Orientados a Objetos (OODBMS) proveen soporte directo para lenguajes orientados a objetos y evitan los problemas de adaptación que enfrentan los programadores al trabajar con motores relacionales tradicionales.

Este tipo de motores, que parece ser el próximo paso lógico para este tipo de productos, lleva en este momento un largo tiempo de investigación pero sin ningún resultado exitoso aun que haga prever que el reemplazo tecnológico, tan obviamente necesario, ocurra en el corto plazo.

El proyecto aquí expuesto, hace lo posible por dar la sensación de estar trabajando frente a este tipo de motores de base de datos, simplemente implementando las funcionalidades deseadas en los mismos ya que no se cuenta con estándares difundidos y mundialmente aceptados a este respecto.

Un aspecto muy interesante y que ha generado muchas propuestas de solución, es el referido al modo en que se hacen las consultas para materializar objetos. Se pueden encontrar definiciones de lenguajes similares a lo que hoy es SQL, y que son implementadas por algunos de los productos ORM. En el caso de db4o, cuenta con las llamadas *Native Queries*, que es una opción muy interesante porque está integrada al lenguaje y es 100% Type-Safe. El único problema que puede presentar es el bajo rendimiento en consultas complejas. Hibernate provee algo similar pero que funciona convirtiendo las consultas a su versión SQL. En una base de objetos, que almacena complejas relaciones entre objetos, es importante cambiar el modo de pensar el acceso a datos, el cual sería más por Navegación que por Búsqueda.

Es importante ver que esta evolución que muestra el mundo del desarrollo de software, debe acompañarse con herramientas y lenguajes que sigan el ritmo del cambio. No solo al hablar de repositorios de datos aptos para el almacenamiento, sino en cuanto a los lenguajes en sí mismos. Debiéramos poder contar con lenguajes de programación extensibles que les permitan adaptarse a distintos tipos de entornos y puedan integrarse. Esto nos permitiría despegarnos de antiguas prácticas como el uso de cadenas de caracteres incrustadas en el código, tan propensas a fallas, para una tarea común como el Acceso a Datos. Ese sería el camino al código más seguro que cada día se busca más. El proyecto apunta directamente a este último concepto.

Conclusión

La orientación a objetos es el paradigma actual de programación aceptado y hemos visto que su evolución no es acompañada por las tecnologías de almacenamiento de información que son más acordes al estilo de programación estructurada del ayer. El proyecto encara el problema a través de un nivel de abstracción que independice la capa de clases de negocio del repositorio de datos.

Aporta un conjunto de funcionalidades para la consulta sobre conjunto de datos más acorde con el modelo de programación utilizado. Permitiendo ahorrar tiempos de desarrollo al evitar escribir código de acceso a datos de manera tradicional.

La intensión del proyecto es diferenciarse de soluciones conocidas en el sentido de evitar la sensación de adaptación entre los distintos paradigmas. Más bien hace uso de tecnologías incompatibles (RDBMS) solo mientras no interfieran con el objetivo de abstracción.

Agradecimientos

A Osvaldo Ocanto, por discutir, aportar ideas y encarar su propio proyecto referido al tema.

Referencias

[1] http://en.wikipedia.org/wiki/Object-Relational_impedance_mismatch: Artículo sobre el Diferencial de Impedancia
[2] www.db4o.com: Open Source Object Database
[3] www.hibernate.com: ORM
[4] Whitepaper: Bases de Objetos, por Germán Viscuso (db4o)
[5] www.dofactory.com: Patrones de Diseño
[6] The Object-Relational Impedance Mismatch, por Scott W. Ambler (www.agiledata.org)
[7] www.firebirdsql.org: Firebird: Open Source RDBMS

Datos de Contacto

Gonzalo Rafael Fernández. Laboratorio de Sistemas de la Universidad Tecnológica Nacional - Facultad Regional Córdoba
Maestro Marcelo Lopez, Esquina Cruz Roja Argentina- Ciudad Universitaria – Córdoba- CP 5016.- horcas@bbs.frc.utn.edu.ar