Diagrama de Estructura Mejorado y Extendido para el modelado de software Cliente Visual/Servidor Relacional

Esteban E. Tripodi Gauthier
Universidad Tecnológica Nacional
Facultad Regional de Concepción del Uruguay
Departamento de Ingeniería en Sistemas de Información
Grupo de Investigación sobre Metodologías de Desarrollo de Sistemas de Información (GIMSI)

Abstract.
En el diseño de software, las técnicas para el modelado de procesos más populares están basadas en un único paradigma: el Diagrama de Estructura (DE) permite representar la estructura de un programa bajo el paradigma imperativo-estructurado, y los distintos diagramas que provee UML soportan el modelado de software orientado a objetos. Pero la realidad es que actualmente, gran parte del software Cliente/Servidor que se desarrolla para los Sistemas de Información de una organización hace uso de al menos dos paradigmas: el imperativo-estructurado o el orientado a objetos (o una combinación de ambos) y el relacional. Esto es común cuando el cliente se codifica en alguna herramienta visual no orientada a la web (Delphi, C++ Builder, Visual C#, NetBeans, PowerBuilder, Visual Basic, etc.), y el servidor es un DBMS relacional. Ante este escenario, las técnicas anteriormente mencionadas resultan inadecuadas. Por este motivo en esta publicación se presenta el Diagrama de Estructura Mejorado y Extendido (DEME), el cual permite modelar la estructura de un software Cliente Visual/Servidor Relacional de manera integral. Además se realiza una comparación entre el DEME, el DE, y los diagramas de Clases y Comunicación de UML, analizando las ventajas y desventajas que presentan cada uno de ellos al aplicarlos al modelado de procesos de esta clase de software.

Palabras Clave.
Modelado de software. Cliente/Servidor. Cliente Visual/Servidor Relacional. Diagrama de Estructura. UML. DEME.

Introducción.

El software orientado al procesamiento información se ha convertido en una parte integral de los Sistemas de Información de cualquier organización, independientemente del tamaño de la misma. La adaptación del mismo a los continuos cambios a los que se ven sometidas las reglas de negocios de una organización, es una tarea continua y de vital importancia.

En este contexto, la documentación generada en todas las etapas de desarrollo de software es especialmente importante, ya que no solo contribuye a la detección de fallas sino que también reduce los costos de mantenimiento y ayuda a la comprensión por parte de los desarrolladores, lo que se traduce en un incremento en la productividad del proceso. A pesar de esto, es mucho el software que carece de una adecuada documentación, especialmente la relacionada al diseño, donde la misma generalmente se reduce al modelo de datos y a alguna descripción de la funcionalidad requerida. [1, 2]

A lo anterior hay que agregar que en la actualidad gran parte del software existente hace uso de la arquitectura Cliente/Servidor, que cuando el cliente no está orientado a la web, se codifica en algún lenguaje visual (por ejemplo, Delphi, PowerBuilder, C++ Builder, Visual Basic, Visual C#, etc.) y el servidor es una base de datos relacional (por ejemplo, Oracle, MySQL, Interbase, Firebird, Informix, etc.). Esto implica que la programación de este tipo de software se hace en al menos dos lenguajes diferentes, el lenguaje del cliente y el del servidor, que no solo difieren en su sintaxis y su semántica, sino también en su paradigma. En general se combinan los paradigmas imperativo-estructurado, orientado a objetos y relacional, ya sea por el uso de más de un lenguaje o porque el lenguaje utilizado soporta más de un paradigma. Por ejemplo,

Delphi y PowerBuilder permiten definir procedimientos y funciones aisladas (paradigma imperativo-estructurado) o que formen parte de una clase (orientación a objetos), y además, para comunicarse con el servidor, se utiliza algún dialecto de SQL.
Ante esta realidad se presenta el inconveniente de que las técnicas para el modelado de procesos mayormente utilizadas -Diseño Estructurado y UML para el Diseño Orientado a Objetos- están basadas cada una de ellas en un único paradigma, por lo que no resultan adecuadas para modelar el tipo de software mencionado. Por este motivo, en este trabajo se presenta el Diagrama de Estructura Mejorado y Extendido (DEME), el cual permite modelar la estructura de un software Cliente Visual/Servidor Relacional. Posteriormente se compara esta técnica con el Diagrama de Estructura, y los diagramas de Clases y Comunicación de UML.

Un caso de estudio.
Para llevar a cabo la investigación dentro de la que se enmarca este trabajo, se tomó como base el Sistema de Préstamos de Biblioteca de la Facultad Regional de Concepción del Uruguay - Universidad Tecnológica Nacional. Dicho sistema posee una arquitectura Cliente/Servidor, donde el cliente está programado en PowerBuilder, y se utiliza el RDBMS Informix para el servidor. El cliente hace uso de los paradigmas imperativo-estructurado, orientado a eventos y relacional. En el servidor, solo está presente el paradigma relacional.
La documentación del diseño de este sistema está formada por un Diagrama de Entidad-Relación, una especificación de la funcionalidad requerida, y por modelos de interfaces y reportes básicos. Para obtener los diagramas a evaluar se aplicó ingeniería inversa sobre su código fuente. Los diagramas obtenidos fueron: Diagrama de Estructura por un lado, y los diagramas de Clases, Casos de Uso, Actividad, Secuencia y Comunicación de UML.

En las siguientes secciones se presentan las diferentes dificultades encontradas al modelar el Sistema de Préstamos de Biblioteca utilizando las técnicas tradicionales comentadas anteriormente, y considerando únicamente el aspecto organizacional o estructural del software, ya que el aspecto lógico-algorítmico no se analiza en esta publicación. Por este motivo, los diagramas utilizados son aquellos que en la técnica correspondiente se utilizan para modelar las características estructurales de los procesos, o que brindan información acerca de las mismas.

Modelado utilizando el Diagrama de Estructura.
La técnica seleccionada para modelar la estructura del Sistema de Préstamos de Biblioteca de acuerdo al paradigma imperativo-estructurado fue el Diagrama de Estructura o Diagrama de Constantine [3, 4, 5], ya que es la más utilizada dentro del Diseño Estructurado. Un Diagrama de Estructura permite ilustrar la división del sistema en módulos, mostrando la jerarquía, organización y comunicación entre los mismos.
A pesar de que esta técnica es muy eficiente para el modelado de sistemas imperativo-estructurados, la misma presenta los siguientes problemas al aplicarla a software Cliente Visual/Servidor Relacional:

1. Dificultad para diseñar software interactivo, ya que no se pueden representar elementos importantes de la interfaz tales como ventanas, componentes visuales y eventos. Esto se debe a que el Diagrama de Estructura está pensado principalmente para el procesamiento por lotes, donde hay una entrada, un proceso y una salida, en ese orden.
2. No se pueden diferenciar los procesos que residen y se ejecutan en el cliente, de aquellos que residen y se ejecutan en el servidor.
3. No se pueden distinguir entre los módulos predefinidos por el lenguaje y

los módulos reusables definidos por el programador.
4. Solo se pueden representar archivos, sin hacer referencia a si son locales al cliente o si representan tablas de la base de datos, y en este último caso, a que base de datos pertenecen.
5. No se pueden representar los triggers asociados a las tablas.
6. No permite especificar los procesos que forman parte de una transacción.

Modelado utilizando UML.

Los trece diagramas que posee la versión 2 del Lenguaje de Modelado Unificado (UML) [6, 7, 8] permiten modelar los aspectos estructurales y de comportamiento de un software orientado a objetos desde distintas perspectivas. De los trece diagramas, seis sirven para modelar el aspecto estructural del software, a saber: Diagramas de Clases, de Objetos, de Componentes, de Estructura Compuesta, de Despliegue y de Paquetes.

A pesar de que esta técnica ha sido creada en la primera mitad de la década pasada, y de que ha evolucionado enormemente a partir de ese momento, se han encontrado las siguientes dificultades en el modelado de sistemas del tipo Cliente Visual/Servidor Relacional:

1. No existe un diagrama que permita modelar los datos y los procesos por separado.
2. No se puede mostrar la estructura de invocación de los procesos (mensajes en la terminología de objetos) en forma independiente del flujo de control.

El punto 1 se debe a que UML ha sido concebido para el modelado de software desarrollado únicamente bajo el paradigma de orientación a objetos, y no para una combinación de paradigmas, por lo que no puede ser aplicado directamente al modelado de procesos del tipo de software planteado en esta publicación. Esto trae el inconveniente de que al tener que modelar todo con objetos y clases -que naturalmente agrupan mensajes, métodos y atributos-, no se puedan representar procedimientos y funciones de manera aislada (elementos comunes en el tipo de software planteado). Una consecuencia directa de lo anterior es que intentar automatizar, o al menos facilitar, el modelado orientado a objetos de este tipo de sistemas a través de herramientas, es poco factible. Por ejemplo, en el caso de querer realizar ingeniería inversa en forma automática, se tendría que realizar un programa que comprenda la semántica de los distintos procedimientos y funciones aisladas del software para poder agruparlos en distintas clases.

Lo que implica el segundo punto es que no alcanza con los diagramas estructurales que provee UML para modelar la estructura de invocación de los procesos. Para hacerlo hay que utilizar alguno de los Diagramas de Interacción (entre los que se encuentran los diagramas de Comunicación y Secuencia), ya que sirven para complementar la información que brinda un Diagrama de Clases sobre este aspecto. El inconveniente que esto presenta es que se necesita conocer la secuencia en la que se pasan cada uno de los mensajes, lo que implica que se deben saber detalles del aspecto lógico-algorítmico de los procesos para poder graficar correctamente cualquiera de los Diagramas de Interacción.

En esta publicación se ha optado por utilizar los diagramas de Clases y Comunicación, ya que el primero es uno de los diagramas más utilizados por las distintas metodologías que aplican UML para representar la estructura estática del software, y el segundo permite modelar la invocación entre los distintos procesos.

Diagrama de Estructura Mejorado y Extendido (DEME).

El DEME ha sido definido para superar los inconvenientes que presentan las distintas técnicas estructuradas y orientadas a objetos al aplicarlas en un entorno Cliente/Servidor, donde el cliente sea desarrollado en un lenguaje visual que combina los paradigmas imperativo-estructurado, orientado a eventos, relacional y en algunos casos la

orientación a objetos; y el servidor sea un RDBMS.

Características.
Para el diseño del DEME se ha tomado como base el Diagrama de Estructura, adaptando y extendiendo el mismo para que cumpla con las siguientes características:

1. Integración: los modelos representados mediante el diagrama deben poder integrarse con modelos que representan otras partes del sistema, en especial con el modelo de interfaces (ventanas, botones, etc.) y con el modelo de datos.
2. Eventos: los eventos asociados a los distintos componentes visuales se deben poder representar.
3. Paradigmas: debe permitir combinar los paradigmas imperativo-estructurado, orientado a objetos y relacional.
4. Cliente/Servidor: se deben poder diferenciar los procesos que forman parte del cliente de los que forman parte del servidor.
5. Capas: debe facilitar la identificación de capas lógicas, especialmente las capas de interfaz, lógica de negocios y acceso a datos.
6. Base de Datos: debe ser posible representar vistas, triggers, tablas pertenecientes a distintas bases de datos, archivos locales al cliente y transacciones.
7. Decisiones de diseño: debe facilitar el análisis y rediseño del software, permitiendo, entre otras cosas, la identificación de errores estructurales, problemas de cohesión y acoplamiento, duplicación de procesos, entre otros.
8. Ingeniería inversa: debe permitir automatizar la creación de modelos a partir del código fuente de software ya construido.
9. Interfaz: debe permitir representar los componentes que forman parte de la interfaz, como mínimo, las ventanas.
10. Ausencia de estructuras de control: el diagrama solamente debe representar la estructura del software, o sea la división del programa en módulos (procedimientos y funciones) y las relaciones de invocación existentes entre ellos. Esta decisión se tomó porque se considera que la estructura de un programa no debe mostrar el flujo de control utilizado en cada módulo, ya que sería una violación al principio de ocultamiento de la información, uno de los pilares de la buena programación. Este es un problema que poseen algunas versiones del Diagrama de Estructura, ya que permiten representar el control del flujo (condicional e iterativo), que forma parte de la lógica de cada módulo en particular, y no de la estructura del software en general.

Descripción.
El Diagrama de Estructura Mejorado y Extendido permite representar la estructura de un software Cliente Visual/Servidor Relacional en forma de módulos y de las relaciones de invocación existentes entre ellos. También permite definir la estructura de descomposición de los procesos, facilitando el control de los niveles de cohesión y acoplamiento de éstos.
Los símbolos que utiliza el DEME son:

1. Módulo Cliente: representa un procedimiento, función o rutina que reside en el cliente. Puede contener código SQL aunque el mismo se ejecute en el servidor.

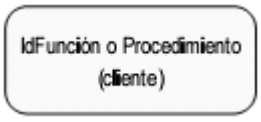

Figura 1: Módulo Cliente.
"IdFunción o Procedimiento" es el nombre (físico o lógico) de la función o procedimiento que representa.

2. Módulo Servidor: representa un procedimiento, función o rutina que reside en el servidor, en particular, un procedimiento almacenado.

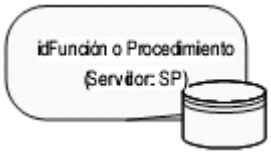

Figura 2: Módulo Servidor.
"IdFunción o Procedimiento" es el nombre (físico o lógico) del procedimiento almacenado que representa.

3. Relación de invocación: representa el llamado de un módulo a otro. La flecha apunta hacia el módulo que se invoca. Se acompaña de los parámetros de entrada al módulo invocado (representados por una secuencia de identificadores entre paréntesis, separados por comas), y el o los resultados (representados por una segunda lista de identificadores entre paréntesis).

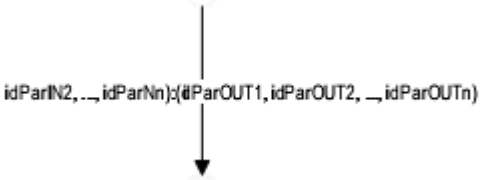

Figura 3: Relación de invocación.
La flecha debe tener un módulo del que parte y otro al que llega.

4. Ventana: representa una ventana o "form". Está asociada a la ventana del mismo nombre que forma parte del modelo de interfaces. Esta representación incluye a los componentes visuales que constituyen la ventana real. Puede tener asociados uno o más eventos, ya sean propios o de los componentes visuales que contiene.

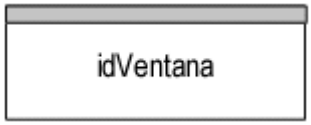

Figura 4: Ventana.
"IdVentana" es el nombre con el cual se identifica a la ventana en el modelo de interfaces.

5. Módulo Evento: representa el código asociado a un evento. Los módulos de este tipo solo pueden ser invocados desde las ventanas.

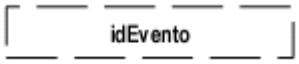

Figura 5: Módulo Evento.
"IdEvento" es el nombre del evento.

6. Tabla o Archivo: representa una tabla propia (base de datos de la aplicación), una tabla externa (base de datos de otra aplicación) o un archivo local (residente en el cliente).

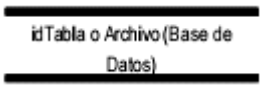

Figura 6: Tabla o Archivo.
"IdTabla o Archivo" es el nombre de la tabla en el modelo de datos, o de un archivo local. El nombre de la base de datos se escribe solo si la tabla es externa. En caso de un archivo local, entre paréntesis se escribe "Local".

7. Vista: representa una vista definida en la base de datos.

Figura 7: Vista.
"IdVista" es el nombre de la vista en la base de datos. Si la vista pertenece a una base de datos externa, entonces a la derecha del nombre se escribe entre paréntesis el nombre de dicha base de datos.

8. Transacción: indica que el proceso contiene una o más transacciones. Se representa agregando una **T** al módulo correspondiente.

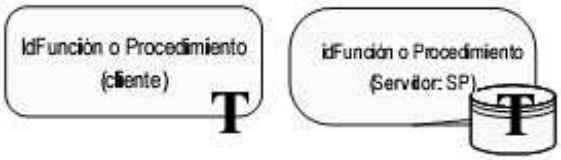

Figura 8: Transacción.
Transacciones en un Módulo Cliente y en un Módulo Servidor.

9. Trigger: representa un evento generado desde la base de datos. Debe contener como origen una Tabla y como destino un Módulo Servidor.

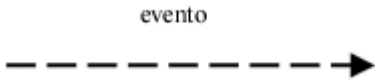

Figura 9: Trigger.
"Evento" es el nombre de la acción (insert, delete, update) que produce la ejecución del módulo asociado.

10. Mensaje a Objeto: representa un mensaje enviado a un objeto y la ejecución de su método correspondiente. A través de estos módulos se extiende la programación estructurada al paradigma de objetos, permitiendo la combinación de ambos.

Figura 10: Mensaje a Objeto.
"Objeto" es el nombre del objeto receptor del mensaje, mientras que "Mensaje" es el nombre del mensaje que se le pasa al objeto.

Ejemplo: Prestar ejemplar.

El ejemplo presentado en esta sección forma parte del proceso de préstamo de un ejemplar. El mismo ha sido modelado utilizando el DEME, el Diagrama de Estructura y los diagramas de Clases y Comunicación.

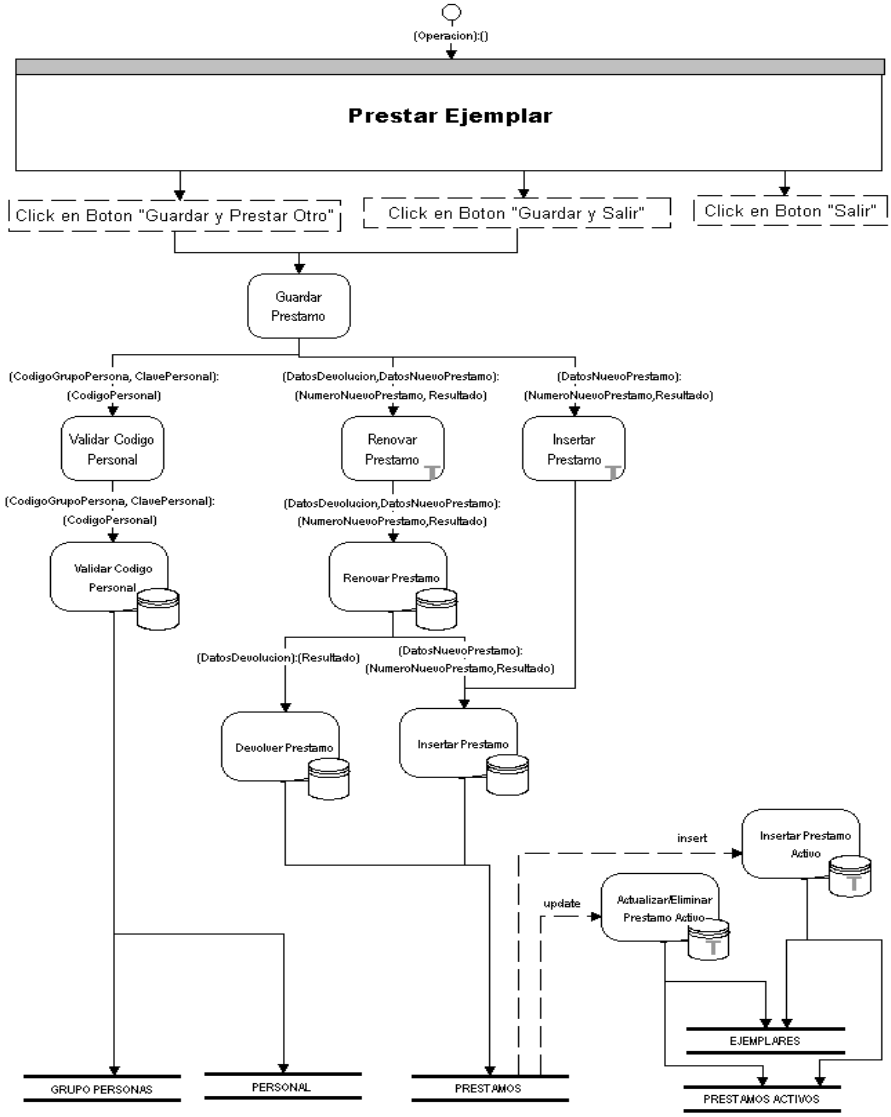

Figura 11: DEME correspondiente a parte del proceso de préstamo de un ejemplar.

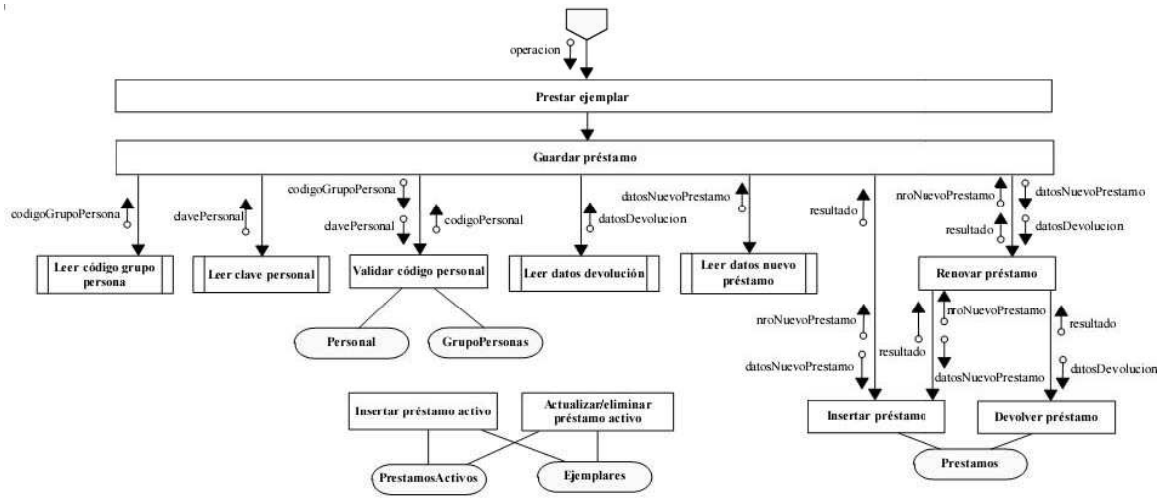

Figura 12: Diagrama de Estructura que representa parte del proceso de préstamo de un ejemplar.

Descripción del proceso.

La ventana principal, “Prestar ejemplar”, es invocada recibiendo como entrada una variable llamada “operación”, que indica si el usuario está realizando un nuevo préstamo o una renovación. La ventana responde, entre otros, a los eventos generados por los botones “Guardar y Prestar Otro”, “Guardar y Salir” y “Salir”. En este caso los dos primeros eventos invocan al mismo módulo (“Guardar Préstamo”), mientras que el módulo evento “Click en Botón Salir” no realiza ninguna llamada, si no que cumple con su función a través de su propio código. El módulo “Guardar Préstamo” valida el código del usuario que está registrando el préstamo invocando a “Validar Código Personal”. Este último es un módulo cliente, que lo único que hace es encapsular el llamado al procedimiento almacenado correspondiente y devolver los resultados. Luego invoca a “Renovar Préstamo” o “Insertar Préstamo”, dependiendo de si se trata, respectivamente, de un nuevo préstamo o la renovación de uno existente. Estos dos módulos son transaccionales. “Insertar Préstamo” abre una transacción, ejecuta el procedimiento almacenado “Insertar Préstamo”, y luego la cierra. El módulo cliente “Renovar Préstamo” encapsula la llamada al procedimiento almacenado “Renovar Préstamo”, que a su vez invoca a los módulos residentes en el servidor “Devolver Préstamo” e “Insertar Préstamo”. Cuando se inserta un préstamo, se dispara el trigger “insert”, que invoca a “Insertar Préstamo Activo”, el cual registra el nuevo préstamo en la tabla “Préstamos Activos”. Esta tabla se utiliza duplicando datos de la tabla de préstamos por razones de eficiencia. El módulo también actualiza la tabla “Ejemplares”, indicando que el ejemplar está prestado. En el caso de una renovación, primero se devuelve el préstamo actual disparando un “update” que invoca al módulo “Actualizar/eliminar préstamo activo” para eliminar el préstamo de la tabla de préstamos activos y actualizar el estado del ejemplar.

Análisis de las tres técnicas de modelado.

El proceso anterior se pudo modelar sin dificultades utilizando el DEME (figura 11). Sin embargo, al utilizar el Diagrama de Estructura (figura 12) y los diagramas de Clases (figuras 13 y 14) y Comunicación (figura 15) se presentaron inconvenientes que en algunos casos pudieron superarse parcialmente y en otros directamente no, quedando aspectos sin modelar o modelados de forma irreal.

Observando el Diagrama de Estructura de figura 12 se puede ver que faltan representar los eventos, elementos que no pueden ser simbolizados usando esta técnica. La ventana está representada por el módulo “Prestar ejemplar”, pero esta no se distingue del resto de los módulos que no representan componentes visuales. Los

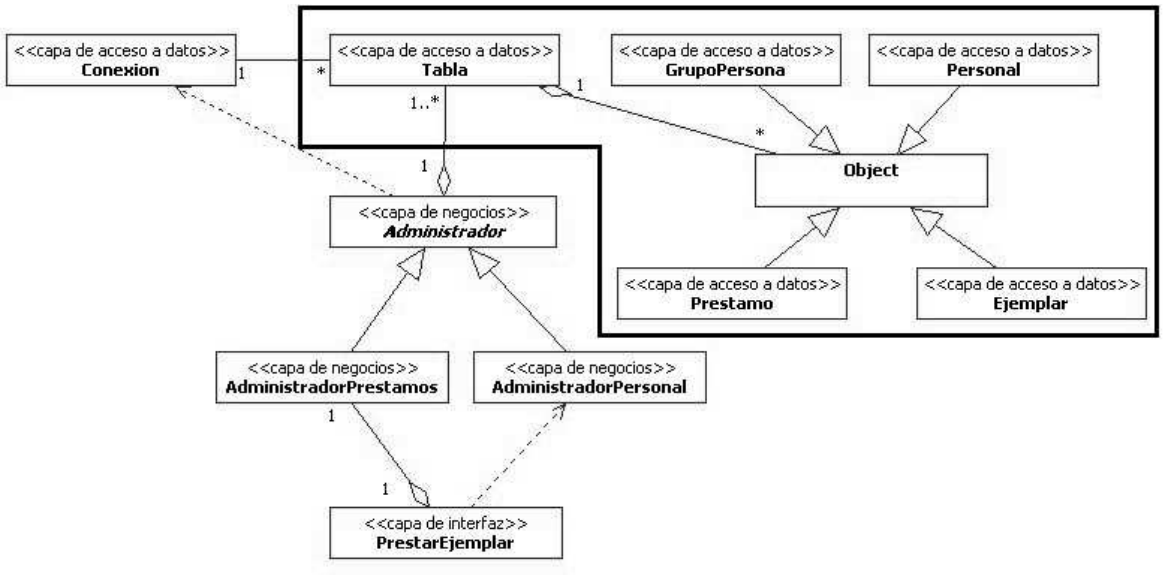

Figura 13: Diagrama de Clases utilizado para modelar parte del proceso de préstamo de un ejemplar. Las clases que están dentro del recuadro tuvieron que ser creadas para modelar algunos aspectos del proceso, pero en realidad no existen. Este es uno de los problemas que se presentaron al modelar el proceso con UML.

módulos "Insertar préstamo", "Renovar préstamo" y "Devolver préstamo" serían tanto los módulos cliente como los procedimientos almacenados (módulos servidor), ya que en este diagrama no se pueden diferenciar entre los dos tipos de módulos, ni tampoco se pueden tener módulos con nombres duplicados (lo que ayudaría a modelar con mayor precisión, pero reduciría la comprensión del diagrama). En cambio, los módulos "Insertar préstamo activo" y "Actualizar/eliminar préstamo activo" serían ambos módulos servidor, pero como se puede observar, quedan totalmente aislados del resto de los módulos, por lo que no se conoce quién los invoca ni con qué parámetros. Los triggers que ejecuta la tabla préstamos tampoco quedan plasmados utilizando este diagrama, así como tampoco los módulos que inician transacciones.

Los diagramas de las figuras 13, 14 y 15 solamente representan modelos del cliente, y no del servidor. Esto es así porque el servidor es de naturaleza relacional, y no se puede modelar con una técnica orientada a objetos.

En la figura 13 se pueden observar las relaciones entre las distintas clases que se utilizan en el proceso. Las clases que están dentro del recuadro son un artificio creado para que el modelo cierre, pero en el sistema real eso no es así. Los objetos correspondientes a las clases "GrupoPersona", "Personal", "Prestamo" y "Ejemplar" serían una tupla de las tablas "GrupoPersonas", "Personal", "Prestamos", "PrestamosActivos" y "Ejemplares" respectivamente. Los objetos de la clase Tabla son los que representan una colección de objetos de alguna de dichas clases, o sea que serían las tablas en sí.

En la figura 14 se pueden observar los atributos y operaciones de cada una de las clases presentes en la figura 13, pero sin las relaciones entre ellas.

Los demás aspectos del proceso han podido ser modelados en mayor o menor medida con UML: los eventos forman parte de operaciones privadas de la clase "PrestarEjemplar"; los módulos que realizan transacciones o invocan a procedimientos almacenados pueden detectarse a través del Diagrama de Comunicación (figura 15); y la estructura de invocación entre los procesos puede verse analizando conjuntamente los Diagramas de Clases y Comunicación.

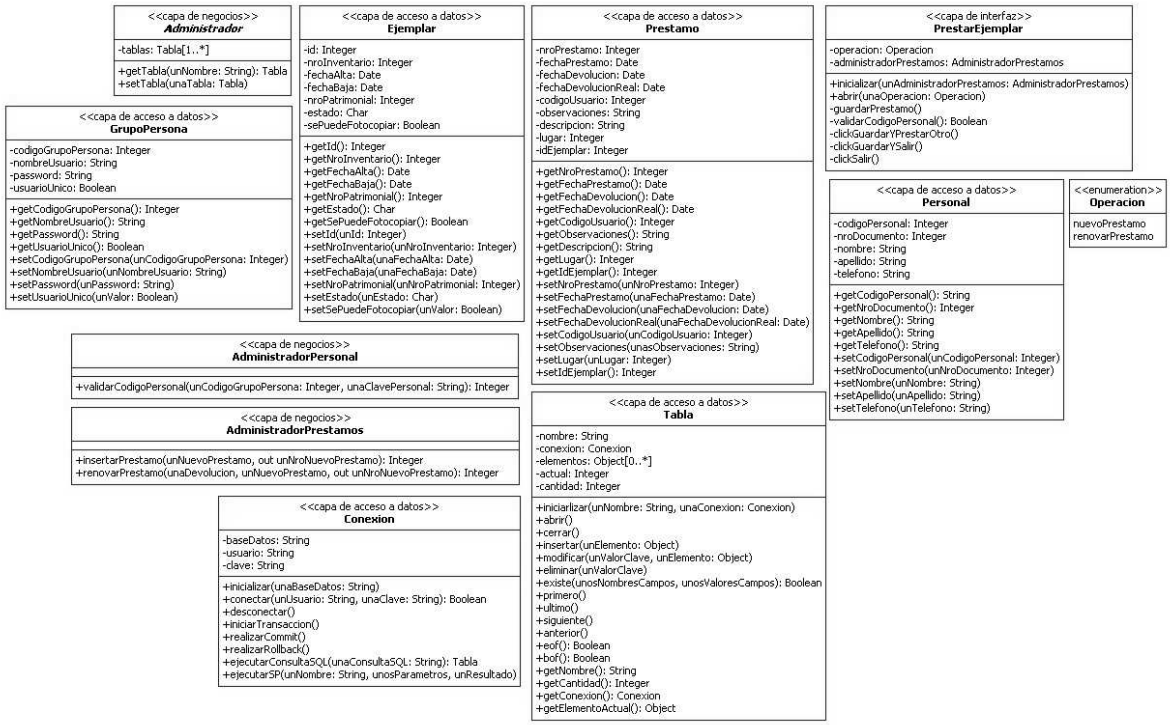

Figura 14: Diagrama de Clases con atributos y operaciones.
En este Diagrama de Clases se muestran algunos de los atributos y las operaciones de las clases presentes en el diagrama de la figura 13.

Algo que permite hacer UML es el uso de notas y estereotipos. Estos últimos son utilizados en los diagramas 13 y 14 para indicar las capas lógicas (layers) a las que pertenecen cada una de las clases, algo que aún no se puede realizar utilizando el DEME. La notas podrían ser utilizadas para aclarar bien cuáles son los eventos que posee la clase "PrestarEjemplar".

En lo que concierne a la detección de errores de diseño, analizando el diagrama de la figura 11 se puede identificar un error en el manejo de transacciones. Los módulos cliente "Renovar Préstamo" e "Insertar Préstamo" realizan la gestión de la transacción. Al realizar modificaciones en la base de datos se disparan los triggers insert y update sobre la tabla "Préstamos", los cuales invocan a los módulos servidor "Actualizar/Eliminar Préstamo Activo" e "Insertar Préstamo Activo" que también abren y cierran una transacción, por lo que se tienen transacciones anidadas, las cuales pueden producir inconsistencias en la base

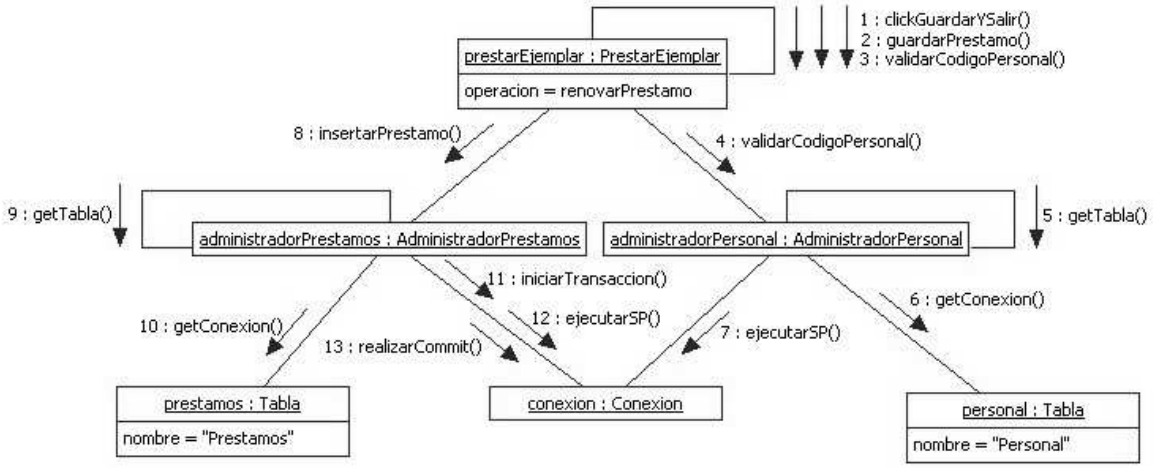

Figura 15: Diagrama de Comunicación.
En este diagrama muestra el paso de los mensajes entre los objetos, al momento de renovar un préstamo.

de datos. Al observar el Diagrama de Estructura de la figura 12 no se puede detectar este error, ya que las transacciones no se pueden representar en el mismo. Este error tampoco se puede divisar analizando los diagramas de UML, ya que a pesar de que en el Diagrama de Comunicación se observa el inicio de una transacción del lado del cliente, no se pueden graficar las transacciones que se inician del lado del servidor.

Otro problema que se puede detectar examinando el DEME es que el módulo "Guardar Préstamo" no recibe parámetros, sin embargo pasa información a los módulos que invoca obteniéndola directamente desde la ventana, lo que genera dependencias ocultas entre ambos módulos. Su diseño se podría mejorar haciendo explícito el intercambio de información a través de parámetros. A diferencia del problema anterior, este también puede ser detectado a través del Diagrama de Estructura. Sin embargo, observando el Diagrama de Clases no existiría tal error, ya que el método "guardarPrestamo" obtiene los datos del estado interno del mismo objeto al que pertenece, que es una instancia de la clase "PrestarEjemplar". Sin embargo, esto último no es así en el sistema real.

Conclusiones.

En la actualidad, la combinación de paradigmas es común en gran parte del software que se desarrolla para los Sistemas de Información de una organización. Otra característica importante de este software es que utiliza la arquitectura Cliente/Servidor, donde el cliente es codificado utilizando alguna herramienta de desarrollo visual, y el servidor es un DBMS relacional.

Las técnicas tradicionales del Diseño Estructurado y del Diseño Orientado a Objetos no permiten documentar correctamente este tipo de software, lo que resulta en una documentación incompleta que dificulta el proceso de adaptación al que continuamente está sometido el mismo. El DEME ha sido ideado para superar las falencias de dichas técnicas cuando se aplican a sistemas Cliente Visual/Servidor Relacional. Sin embargo, aún quedan muchas características importantes que deben ser definidas, tales como la forma de reflejar consideraciones arquitectónicas, el reconocimiento de patrones de software reutilizables, la identificación capas lógicas (layers), y la forma de relacionar módulos con reglas de negocios.

Referencias.

[1] W. Pannesi, M. Oloriz, J. Peri: Grado de Documentación del Proceso de Producción de Software. Jornaciti 2004. Universidad Nacional de Luján; pp. 8-8. (2004)

[2] W. Pannesi, M. Oloriz, C. Ortiz: Diagramas de Estructura como Mapa del Sistema. WICC. (2005)

[3] T. DeMarco: Structured Analysis and System Specification. Yourdon Press Computing Series. Prentice Hall. Englewood Cliffs, New Jersey. (1979)

[4] E. Yourdon and L. L. Constantine, Structured Design: Fundamentals of a Discipline of Computer Program and System Design. Yourdon Press Computing Series. Prentice Hall. Englewood Cliffs, New Jersey. (1979)

[5] M. Page-Jones: The Practical Guide to Structured Systems Design. Yourdon Press Computing Series. Prentice Hall. Englewood Cliffs, New Jersey. (1988)

[6] G. Booch, J. Rumbaugh, I. Jacobson: El Lenguaje Unificado de Modelado. Object Technology Series. Addison-Wesley. (1999)

[7] M. Fowler: UML Distilled. Object Technology Series. Addison-Wesley. (2003)

[8] O.M.G.: Unified Modeling Language: Superstructure 2.0 (formal/05-07-04). (2005)

Agradecimientos.

Les agradezco a Carolina Carrera, Noelia Vignola, Gastón Dall'Oglio y Andrés Pascal, quiénes revisaron, corrigieron y/o aportaron sugerencias para la presente publicación.

Datos de contacto.

Esteban E. Tripodi Gauthier.
Universidad Tecnológica Nacional.
Facultad Regional de Concepción del Uruguay.
Ing. Pereira 676.
(3260) Concepción del Uruguay – Entre Ríos.
etripodi@gmail.com

Docente tutor.

Ing. Andrés J. Pascal.
andrespascal2003@yahoo.com.ar