Separación Dinámica del Aspecto de Persistencia mediante Reflectividad Computacional

Benjamín López, Francisco Ortin, Juan Manuel Cueva

Object-Oriented Technologies Laboratory research group (OOTLab),
Departamento de Informática, Universidad de Oviedo, C/ Calvo Sotelo s/n,
33007, Oviedo, España
{benja, ortin, cueva}@lsi.uniovi.es

Abstract. El principio de la separación de incumbencias se centra en la capacidad de modularizar aquellas partes diferentes de una aplicación relevantes a un concepto, objetivo, tarea o propósito específico. Una separación apropiada de los distintos aspectos de un sistema reduce la complejidad del software, mejora su comprensión y facilita la reutilización de código. Considerando la persistencia como una incumbencia típica en la mayoría de las aplicaciones, la separación de ésta del código principal del sistema conlleva el desarrollo de programas sin tener en cuenta sus requisitos persistentes, añadiendo éstos en fases posteriores. Esta separación permite al desarrollador manejar la persistencia de los programas de forma independiente a su funcionalidad. Tras analizar distintas alternativas existentes para conseguir una separación total del aspecto de persistencia, nos hemos percatado de que, si bien unas ofrecen mayor transparencia que otras, ninguna permite desarrollar una aplicación cuyo código no posea manejo alguno de características persistentes. La reflectividad computacional es una técnica que permite adaptar la estructura y comportamiento de un sistema en tiempo de ejecución. Sobre una plataforma reflectiva no restrictiva implementada previamente, hemos desarrollado un sistema de persistencia en el que se comprueba cómo este mecanismo puede ser empleado para ofrecer una separación total de la incumbencia de la persistencia. Adicionalmente, ofrece un elevado nivel de adaptabilidad que permite cambiar dinámicamente las características persistentes de un programa en ejecución.

Palabras Clave. Persistencia, Reflectividad Computacional, Separación de Incumbencias, Persistencia Ortogonal, Separación de Aspectos.

1 Introducción

Actualmente la mayoría de las aplicaciones que manejan objetos persistentes son diseñadas con uso explícito de Sistemas Gestores de Bases de Datos (SGBD). Éstos pueden ser orientados a objetos o relacionales, empleando en el segundo caso algún mecanismo de traducción. Por tanto, el modo más común de desarrollar aplicaciones persistentes es entremezclando el código de la aplicación con sentencias OQL o SQL de control de persistencia. El no separar el código principal de la aplicación de las sentencias de gestión de la persistencia provoca una serie de inconvenientes:

1. Legibilidad y mantenibilidad. Puesto que el código de persistencia está enmarañado con el funcional, la comprensión, modificación y depuración de la lógica principal del programa se hacen más complejas.
2. Portabilidad. Existe una dependencia directa entre el mecanismo de persistencia y la implementación de la aplicación. Así, requerir cambios relativos a la persistencia provocarán cambios en la implementación del programa.
3. Reutilización de rutinas de persistencia. Es común encontrarse repetidamente con rutinas persistentes similares en las que lo único que varían es la estructura de los datos que manipulan. Este código podría estar factorizado y reutilizado independientemente de los objetos que manejen.
4. Adaptabilidad. El adaptar características persistentes de una aplicación requiere modificar el código fuente y recompilar éste. No es posible adaptar la configuración del aspecto de persistencia en tiempo de ejecución sin tener que modificar y recompilar la aplicación.

El principio de la separación de incumbencias o intereses (*separation of concerns*) surge para superar estos problemas comunes, existentes a lo largo ciclo de vida del software [1]. La idea de este principio se basa en identificar y separar diferentes incumbencias de una aplicación, ortogonales entre sí. Siguiendo este principio, la persistencia de una aplicación debería poder añadirse por separado a una aplicación una vez ésta haya sido desarrollada. El código fuente de la misma no debería verse modificado.

En este artículo analizaremos las distintas alternativas existentes para obtener una separación de la persistencia de una aplicación de su código principal. Comenzamos con las tenencias dominantes en la actualidad (sección 2), para pasar a los sistemas de persistencia ortogonal (sección 3) y los estudios realizados con programación orientada a aspectos (sección 4). En la sección 5 identificaremos la reflectividad computacional como técnica adecuada para obtener los beneficios enunciados en este punto, mostrando posteriormente la implementación realizada y una aplicación de ejemplo. Finalmente se detalla el trabajo futuro y las conclusiones.

2 Desarrollo Habitual de Aplicaciones Persistentes

El modelo de datos dominante en la actualidad es mayoritariamente el modelo relacional, representado a nivel práctico por el lenguaje SQL. Tomando el lenguaje Java como ejemplo, el programador suele utilizar SQL, ya bien sea de un modo directo (utilizando JDBC o SQLJ) o indirecto (empleando algún sistema de traducción objeto-relacional, o un framework como EJB).

Otro mecanismo para dar persistencia a una aplicación es utilizar un sistema de almacenamiento basado en ficheros. Centrándonos en Java como ejemplo, esta plataforma incluye una tecnología de serialización de objetos. Adicionalmente, XML se está empleando como formato popular de formato de ficheros.

Cuando un programador elige un lenguaje orientado a objetos, la necesidad de traducir a SQL o XML un grafo de objetos es un requisito adicional a los existentes en el dominio del problema, además del coste computacional existente en tiempo de ejecución. Además, esta desadaptación de impedancias [2] requiere que se escriban senten-

cias de código explícitas para hacer a los objetos persistir. La utilización de un sistema en el que la gestión de la persistencia se haga de un modo transparente es sin duda ventajosa, ya que mejora su mantenimiento, legibilidad, adaptabilidad y reusabilidad [3].

3 Persistencia Ortogonal

Un paso adelante en la obtención de sistemas persistentes transparentes fue la aparición de la persistencia ortogonal en los años 90 [4]. El principal objetivo de estos sistemas es proporcionar un modelo computacional sencillo y uniforme para todos los aspectos de una aplicación que traten con datos persistentes. Esta capacidad se define con tres principios.

- Ortogonalidad respecto al tipo. Todos los objetos podrán ser persistentes, independientemente de su tipo.
- Independencia de la persistencia. La forma de un programa es independiente de la longevidad de los datos que manipulan.
- Persistencia por alcance. El ciclo de vida de cada objeto viene determinado por el alcance desde un conjunto de objetos raíz.

Los dos primeros principios conforman el objetivo principal de un sistema completamente transparente: mismo código para tratar objetos persistentes y temporales, independientemente del tipo de éstos.

La tercera regla especifica un mecanismo para implementar persistencia transparente. Sin embargo, este principio se centra en establecer la incumbencia de la persistencia en el propio código de la aplicación. No obstante, si pensamos en persistencia como un una incumbencia que ha de estar separada del código fuente de una aplicación, el tercer principio de la persistencia ortogonal es un mero ejemplo de un conjunto de implementaciones. Podría ser interesante en ciertos escenarios la utilización de otros criterios, implementados aparte de la aplicación principal –como elegir un único conjunto de objetos de una aplicación como persistentes.

Existen distintos ejemplos de sistemas persistentes ortogonales. En el caso de Java, PJama [4] y PEVM [5] son las dos implementaciones más conocidas de la especificación persistente de la plataforma de Java (OPJ) [6]. El principal inconveniente de ambas implementaciones es que el principio de persistencia por alcance hace que no cumplan de un modo completo el criterio de independencia de la persistencia [7, 8]. La persistencia no es tenida en cuenta como una incumbencia totalmente separada de la lógica de la aplicación.

4 Persistencia en Programación Orientada a Aspectos

El desarrollo de software orientado a aspectos (AOSD) es una evolución de la programación orientada a aspectos (AOP) [3]. AOP es una técnica de implementación que proporciona un soporte a nivel de lenguaje de programación para modularizar las incumbencias ortogonales a la funcionalidad de la aplicación. Los aspectos expresan

funcionalidad que "atraviesa" el sistema de un modo modular, permitiendo diseñar un sistema como composición de aspectos separados. La separación de los distintos aspectos de una aplicación y su código principal hace que éstos dejen de estar entremezclados, facilitando su depuración y mantenimiento [9].

El la literatura existente en AOSD, es común ver la persistencia como una funcionalidad candidata a ser tenida en cuenta como un aspecto [10]. Teóricamente, tendría que ser posible:

- Modularizar la persistencia de una aplicación como un aspecto, empleando técnicas propias de la AOP.
- Reutilizar los aspectos propios de la persistencia de un modo independiente al tipo de aplicación.
- Desarrollar programas de un modo independiente a la naturaleza persistente de sus datos.

Analizando distintas implementaciones de persistencia como aspectos, nos hemos dado cuenta de que dichos objetivos no se han alcanzado en implementaciones reales. PersAJ [11] es un sistema que implementa un prototipo para almacenar aspectos en una base de datos orientada a objetos. Kielze y Guerraoui [12] proporcionaron una evaluación de AOP para separar la concurrencia y tolerancia a fallos de un sistema distribuido. Rashid y Chitchyan se centraron en desarrollar un sistema de persistencia con AspectJ [13]. La conclusión de todos estos trabajos indicó que el desarrollo del aspecto de persistencia no puede ser llevado a cabo de forma totalmente separada a la lógica del programa, empleando las herramientas existentes de AOP –el almacenamiento y actualización pueden llevarse a cabo de un modo transparente, pero no la obtención y eliminación de objetos persistentes [13].

En comparación con los mecanismos estudiados, mostraremos cómo la reflectividad computacional es una técnica más apropiada para desarrollar los aspectos persistentes de cualquier aplicación, de un modo separado a su código.

5 El sistema Reflectivo nitrO

La reflectividad de un sistema computacional es la capacidad de razonar y actuar sobre sí mismo, adaptándose a condiciones surgidas dinámicamente [14]. Su dominio computacional es ampliado con su propia representación, teniendo la capacidad de manipular su estructura y semántica (comportamiento) como si de datos se tratase.

Aunque existen múltiples clasificaciones de reflectividad [15], nos centraremos en la reflectividad computacional en tiempo de ejecución: aquélla que permite modificar la estructura y comportamiento de un sistema dinámicamente.

La reflectividad computacional es unta técnica mucho más potente que el nivel de reflectividad ofrecido hoy en día por la mayoría de los lenguajes comerciales (como Java o C#). Éstos simplemente ofrecen reflectividad estructural de lectura, también conocida como introspección. La reflectividad computacional no sólo añade la capacidad de modificar esta estructura, sino que ofrece la posibilidad de modificar la semántica del lenguaje.

Los protocolos de meta-objetos (MOPs) son los mecanismos más utilizados para obtener reflectividad computacional en tiempo de ejecución [16]. Poseen dos incon-

venientes importantes: ofrecen un conjunto de primitivas (protocolo) demasiado reducido para desarrollar sistemas altamente adaptables y emplean un único lenguaje de programación [17]. Por este motivo, desarrollamos nuestro propio sistema reflectivo no restrictivo denominado nitrO [17]. Este sistema ofrece una mayor adaptabilidad que los MOPs existentes, siendo posible su programación en cualquier lenguaje de programación. Ha sido desarrollado en Python 2.2.

El diseño de nitrO se centró en la definición teórica de reflectividad [18]. Las aplicaciones que se están ejecutando en nitrO pueden acceder al intérprete (meta nivel) en tiempo de ejecución, modificando su estructura y adaptando la semántica de su lenguaje (su comportamiento). El sistema se centra en un intérprete genérico capaz de interpretar cualquier lenguaje de programación, previa especificación del mismo (figura 1).

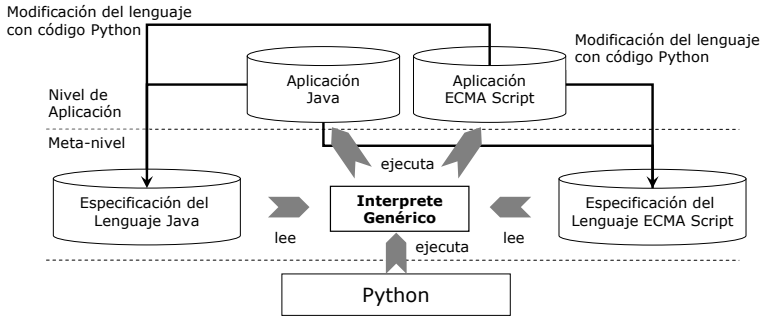

Fig. 1 Arquitectura del sistema nitrO

La descripción del los lenguajes en nitrO es llevada a cabo mediante ficheros de especificación que utilizan un sistema de procesamiento descendente muy similar al empleado por la herramienta JavaCC. La especificación léxica (sección *scanner*) y sintáctica (sección *parser*) son descritas mediante producciones libres de contexto. Las rutinas semánticas se describen mediante código Python al final de cada producción. Hemos especificado Python, ECMAScript, un subconjunto de Java (eliminando sus tipos primitivos para simplificar su implementación) y una serie de lenguajes de propósito específico.

En tiempo de ejecución cualquier aplicación podrá acceder y modificar la especificación de cualquier lenguaje, utilizando para ello toda la expresividad de un metalenguaje: Python. De forma contraria a los sistemas convencionales reflectivos, no existe un protocolo previo que restrinja el abanico de primitivas susceptibles de ser modificadas. Las modificaciones dinámicas realizadas en el meta-nivel serán automáticamente reflejadas en la ejecución de la aplicación, puesto esas estructuras son las que el intérprete utiliza para evaluar la semántica de los programas (figura 1).

6 El sistema de Persistencia de nitrO

Utilizando las características reflectivas de nitrO, hemos desarrollado un sistema de persistencia que ofrece una separación completa del aspecto (incumbencia) de la persistencia. El sistema se compone principalmente de tres subsistemas:

1. Aplicación. Este módulo ofrece la representación de la estructura de los programas en ejecución (sus métodos, clases, objetos…). Puede ser reutilizado con independencia del lenguaje seleccionado, siempre que éste utilice un modelo orientado a objetos. La modificación de estos objetos dinámicamente producirá reflectividad estructural del lenguaje interpretado.
2. Intérprete. Es el responsable de llevar a cabo el análisis y ejecución de cada aplicación. En nuestro caso hemos desarrollado un único intérprete de un subconjunto del lenguaje Java –la principal simplificación ha sido la eliminación de los tipos primitivos, para facilitar su implementación. Su modificación dinámica supondrá reflectividad computacional (de comportamiento).
3. Persistencia. Este es el subsistema principal. Empleando reflectividad, este módulo se encarga de adaptar dinámicamente las aplicaciones en ejecución para hacerlas persistentes de un modo transparente. Ha sido diseñada para que se puedan emplear distintos sistemas de almacenamiento, mecanismos de indexación y políticas de actualización de objetos, adaptables dinámicamente.

El sistema ha sido desarrollado al mismo nivel que el intérprete genérico, en el metanivel, empleando el lenguaje Python. Su código emplea las características reflectivas de la plataforma, personalizando la estructura y comportamiento de los programas en ejecución. Cualquier programa podrá hacer persistente a otro, sin necesidad de cambiar su código fuente original.

6.1 Subsistema Intérprete

NitrO toma la especificación del lenguaje Java y automáticamente generará el árbol sintáctico de la aplicación a ser ejecutada. Entonces, se ejecutará (siguiendo el patrón de diseño *Command* [19]) la rutina semántica especificada al final de cada producción. Este proceso devuelve el árbol sintáctico abstracto (AST) del programa de entrada.

El intérprete toma el AST y lleva a cabo su interpretación. El método *parse* toma un AST, analiza la estructura de sus nodos y llama al método de visita apropiado. Siguiendo este esquema se realizan las fases de análisis semántico, traducción de código (en memoria) y ejecución del mismo. La modificación de los métodos de visita de la fase de interpretación conllevará la reflectividad computacional de la aplicación ejecutada.

6.2 Subsistema Aplicación

La figura 2.a muestra el diagrama de clases de los elementos que conforman una aplicación Java en tiempo de ejecución. Las clases (*JClass*) están constituidas por atributos (*JField*), métodos (*JMethod*) y constructores (*JConstructor*); los dos últimos ele-

mentos se agrupan mediante instancias de *JMethodGroup*. *JRef* denota una referencia a una instancia.

Un elemento importante del diseño de este módulo es el empleo de la interfaz *Instance* encargado de indicar el conjunto de operaciones que son necesarias implementar para hacer un elemento persistente. En nuestro diseño, sólo los objetos son persistentes ya que las clases (el código) están almacenadas en el sistema de archivos.

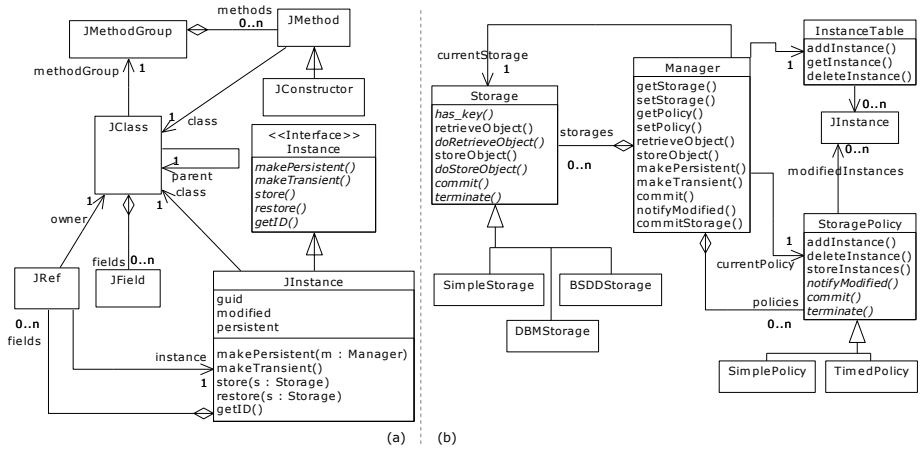

Fig. 2. Diagrama de clases del subsistema aplicación (a) y del subsistema persistencia (b).

6.2.1 ID de Objetos Persistentes

La creación de un identificador (ID) único para todo objeto persistente es una de las tareas de todo sistema persistente. Puesto que los objetos persistentes de una aplicación van a sobrevivir a su ejecución, las referencias a éstos (sus direcciones de memoria) no serán válidas en posteriores ejecuciones. Por tanto, deberemos asignar un identificador único y global a todo objeto persistente.

El diseño del sistema de persistencia obliga a que todo elemento persistente deba retornar su ID en la invocación de su método *getID*. La implementación de dicho método en la clase *JInstance* devuelve la concatenación de los siguientes valores: la dirección IP de la máquina, el PID del proceso, el UID del usuario, el TID del hilo activo y los milisegundos transcurridos desde el 1 de enero de 1970. Esta implementación de referencia utiliza IDs tan amplios para evitar la colisión de identificadores en futuras versiones distribuidas del sistema.

6.3 Subsistema Persistencia

La figura 2.b muestra el modulo de persistencia. La clase *Manager* es la *Fachada* y ha sido implementada mediante un *Singleton* [19]. Ofrece servicios de persistencia al programador, permitiendo modificar el comportamiento de éste dinámicamente mediante la selección de objetos derivados de *Storage* y *StoragePolicy*.

Nuestro sistema permite utilizar distintos sistemas de almacenamiento y de políticas de actualización de objetos. Las clases abstractas *Storage* y *StoragePolicy* ofrecen implementaciones parciales de tales funcionalidades, facilitando así la adición de nuevas clases derivadas. Las clases derivadas de *Storage* representan distintas formas de almacenar información persistente, así como distintos sistemas de indexación. En el caso de *StoragePolicy*, se especifica la frecuencia con la que los objetos tengan que ser actualizados en el almacenamiento seleccionado. La selección, intercambio y modificación dinámica de estas dos variables puede ser llevada a cabo de un modo programático.

Hemos desarrollado tres almacenamientos de referencia:

- *SimpleStorage*: Es un sencillo diccionario que es salvado y cargado de disco. Es el almacenamiento por omisión seleccionado por el gestor de persistencia.
- *BSDDBStorage*: Proporciona el acceso a una *Berkeley DB Library*. El usuario puede crear sistemas de almacenamiento basados en técnicas de hashing extendido lineal, árboles B+ o registros de longitud variables, en función de los parámetros pasados a su constructor. Este sistema de almacenamiento puede ser usado para hacer cambiar dinámicamente el mecanismo de indexación, en función de contextos y condiciones surgidos en tiempo de ejecución.
- *DBMStorage*: Esta clase ofrece una librería del tipo *Unix (n)dbm*. El sistema de almacenamiento se comporta como una memoria asociativa persistente.

También hemos desarrollado dos políticas distintas de actualización de objetos:

- *SimplePolicy*: La actualización de los objetos persistentes en el sistema de almacenamiento (implementada por el método *commit*) se producirá siempre que el estado de un objeto haya sido modificado un número especificado de veces. Ésta es la política por omisión, con una única modificación necesaria para actualizar el objeto.
- *TimedPolicy*: Se emplea un temporizador parametrizado por un número determinado de milisegundos.

Cada uno de los parámetros mencionados en las políticas y almacenamientos anteriores puede ser modificado dinámicamente, en función de requisitos surgidos durante la ejecución de la aplicación.

6.3.1 Almacenamiento de Objetos

En los sistemas de almacenamiento de referencia implementados, hemos utilizado el módulo *pickle* de Python capaz de serializar objetos. Aunque este módulo puede ser utilizado para almacenar objetos, no aborda el problema de asignar IDs a los objetos persistentes. Es por ello por lo que hemos implementado nuestro propio sistema de identificadores, descrito en la sección 6.2.1. El proceso de convertir los IDs de objetos persistentes a referencias en memoria se denomina *swizzling*; el proceso contrario es llamado *unswizzling*.

El gestor de persistencia implementa un sistema de (un)swizzling perezoso. En el caso de la traducción de referencias a IDs persistentes, unswizzling, es realizado justo antes de almacenar los objetos en el sistema de persistencia. Si el objeto tiene referencias a otros objetos persistentes, este proceso de traducción es efectuado de un modo recursivo de forma paralela a la serialización de los objetos.

El proceso inverso (swizzling) se desarrolla en dos fases. El objeto demandado es buscado inicialmente en el sistema de almacenamiento a partir de su ID. En este paso,

la secuencia de bytes que representa a este objeto es extraída del sistema de almacenamiento y convertida en un objeto. Posteriormente se lleva a cabo el swizzling, recuperando los enlaces originales entre los objetos.

El proceso descrito se logra empleando la clase *InstanceTable* de la figura 2.b. Esta tabla es un diccionario de referencias débiles. Cuando un objeto se hace persistente, se añade una entrada a éste en la tabla. Si se necesita un objeto que tiene una entrada en la tabla se obtendrá de esta su referencia, actuando así como una caché.

Si el objeto persistente deja de ser referenciado en la aplicación, el recolector de basura podrá eliminarlo. Cuando un objeto persistente es requerido y no está en la tabla de instancias, el gestor de persistencia lo obtendrá del sistema de almacenamiento seleccionado.

7 Aplicación de Ejemplo

En esta sección presentaremos un ejemplo de aplicación de autores, tomada de la información almacenada en el servidor web DBLP [20]. El modelo de datos está compuesto de un conjunto de elementos bibliográficos (revistas, conferencias, libros, actas y artículos), editoriales, autores, localizaciones y editores.

La aplicación ha sido desarrollada en su totalidad en el lenguaje de programación Java, siendo ésta no persistente inicialmente. El programa se ejecutará permitiendo al usuario crear objetos y enlazarlos entre sí. Una instancia de la clase *AplicacionBibliografica* posee distintas colecciones de las editoriales, autores, editores, localizaciones y elementos bibliográficos. Al finalizar la ejecución de la aplicación, la memoria de estas colecciones es liberada –no son persistentes.

Aparte de la aplicación bibliográfica, desarrollamos otro programa que controla la incumbencia de la persistencia de la primera aplicación, permitiendo asignar, eliminar y modificar dinámicamente los aspectos propios de persistencia del primer programa. Esta implementación será posible gracias a la utilización de reflectividad, utilizando código Python dentro de instrucciones *reify*.

Mediante un menú gráfico el controlador de persistencia permitirá al usuario hacer persistente la aplicación bibliográfica, no persistente inicialmente. Por otra parte, permite cambiar en tiempo de ejecución el sistema de almacenamiento, política de actualización de objetos y el sistema de indexación utilizado.

La figura 3 muestra un escenario de ejemplo en el que los objetos de la aplicación son recuperados de una ejecución previa. Las dos ventanas superiores muestran el programa bibliográfico con su menú gráfico, estando la lista de elementos mostrada inicialmente vacía (ventana superior izquierda). Mediante el empleo del *shell* de nitrO lanzamos el controlador de persistencia (dos ventanas inferiores) permitiendo asignar y modificar la persistencia del primer programa. Seleccionando la opción *Recuperar Estado*, la aplicación bibliográfica recuperará indirectamente un conjunto de objetos persistentes procedentes de una ejecución previa. En ese momento, si se vuelven a mostrar las distintas colecciones de elementos, veremos la lista de la figura 3.

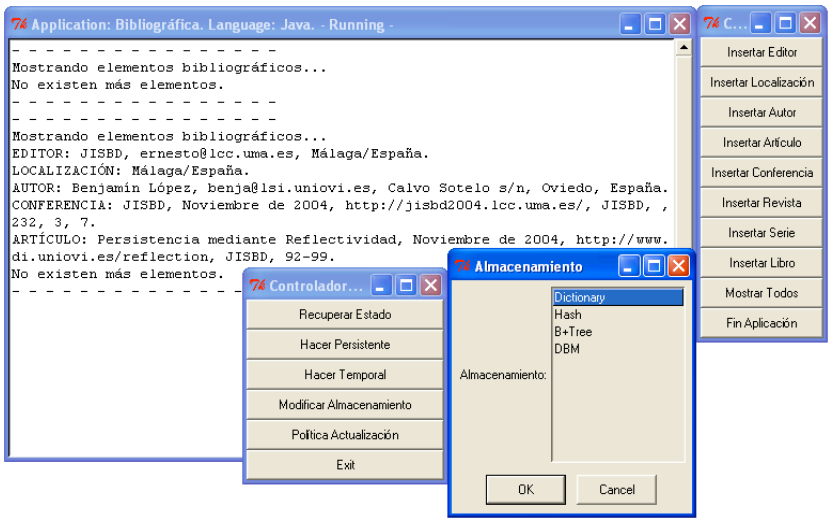

Fig. 3 Ejecución de la aplicación bibliográfica y su controlador de persistencia

Es posible hacer la aplicación persistente mediante el controlador, obteniendo así una sincronización totalmente transparente entre los objetos del programa y el sistema de almacenamiento. La figura 3 también muestra cómo los sistemas de almacenamiento (ventana de abajo a la derecha) y las políticas de actualización de objetos pueden ser modificadas dinámicamente.

Mediante el empleo de reflectividad computacional, este sistema ofrece una separación total de la incumbencia (aspecto) de persistencia. El sistema de persistencia es, además, adaptativo: permite desarrollar programas que adapten a otros programas en función de contextos surgidos dinámicamente.

8 Eficiencia

El mayor inconveniente la adaptabilidad dinámica de una aplicación es su eficiencia en tiempo de ejecución [21]. El proceso de adaptar un programa, sumado al hecho de utilizar reflectividad, supone un consumo de recursos adicional en la ejecución de la aplicación [22]. La adaptabilidad y la eficiencia son variables que normalmente son contrarias. En nuestra primera implementación hemos tratado de obtener el mayor grado de adaptabilidad en tiempo de ejecución, siguiendo fielmente el principio de la separación de incumbencias.

La razón principal de la caída de eficiencia de nuestra plataforma reflectiva es la interpretación de todos los lenguajes de programación. Hoy en día, es típico ver lenguajes interpretados empleados en la empresa (Java, C# o Python) debido a técnicas de optimización como compilación bajo demanda (JIT) o generación adaptativa de código nativo [23]. En futuras versiones de nuestra plataforma, emplearemos estas técnicas para optimizar la implementación del intérprete genérico.

9 Trabajo Futuro

El trabajo presentado muestra cómo la reflectividad computacional es un mecanismo ideal para separar el código funcional de una aplicación de sus distintas incumbencias de persistencia, pudiendo variar éstas dinámicamente de un modo programático. El trabajo futuro estará enfocado a desarrollar sistemas que ofrezcan servicios de persistencia al usuario final, empleando la plataforma presentada. Distintas alternativas son: mediante la selección de aspectos de persistencia con un explorador gráfico de objetos, como un entorno de programación de propósito específico, o como un sistema completo de persistencia ortogonal, en el que el almacenamiento, mecanismo de indexación, política de actualización y selección de objetos persistentes se lleven a cabo de un modo totalmente transparente.

Otras futuras mejoras a introducir en el sistema son la incorporación de especificaciones de nuevos lenguajes de programación y la utilización de un compilador JIT para acelerar su rendimiento.

10 Conclusiones

El principio de la separación de incumbencias se centra en ofrecer un mecanismo de modulación para los aspectos ortogonales de una aplicación, ofreciendo un conjunto de beneficios en el desarrollo software. Las alternativas para conseguir estos beneficios son esquemas de traducción de objetos al modelo relacional, sistemas de persistencia ortogonales, o desarrollo de software orientado a objetos. Sin embargo, ninguna de estas alternativas ofrece una separación total de la incumbencia de la persistencia, además de no representar aproximaciones adaptables dinámicamente.

Nosotros hemos empleado la reflectividad computacional como una técnica apropiada para poder evitar las deficiencias encontradas en los sistemas analizados. Utilizando nuestra plataforma reflectiva nitrO, hemos sido capaces de lograr una separación total del aspecto persistente de una aplicación, pudiendo además adaptar características como el sistema de almacenamiento, indexación y políticas de actualización de objetos.

La plataforma, el sistema de persistencia y el código fuente presentado en este artículo se pueden descargar gratuitamente de:

http://www.di.uniovi.es/reflection/lab/prototypes.html#persistence

References

[1] Hürsch, W.L., Lopes, C.V.: Separation of Concerns. Technical Report NU-CCS-95-03, College of Computer Science, Northeastern University (1995).

[2] Maier, D.: Why Isn't There an Object-Oriented Data Model? In: IFIP Wold Computer Congress, San Francisco, California (1989) 793–798.

[3] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect Oriented Programming. In: European Conference on Object-Oriented Programming (ECOOP), Springer-Verlag LNCS 1241 (1997) 220–242.

[4] Atkinson, M., Daynès, L., Jordan, M., Printezis, T., Spence, S.: An Orthogonally Persistent Java. SIGMOD Record, Vol. 25 No. 4 (1996) 68–75.
[5] Lewis, B., Mathiske, B., Gafter, N: Architecture of the PEVM: A High-Performance Orthogonally Persistent Java™ Virtual Machine. SMLI TR-2000-93 (2000).
[6] Jordan, M.J., Atkinson, M.P.: Orthogonal Persistence for the Java Platform —Specification. Sun Microsystems Laboratories, Palo Alto, CA 94303 (2000).
[7] García Perez-Schofield, J. B., García Roselló, E., Cooper, T.B., Pérez Cota, M.: Managing schema evolution in a container-based persistent system. Software, Practice & Expererience, Vol. 32, No. 14 (2002) 1395–1410.
[8] Jordan, M.: Early Experiences with Persistent Java. In: Proceedings of the First International Workshop on Persistence and Java. Glasgow, Scotland. Sun Microsystems Technical Report TR-96-58 (1996).
[9] Parnas, D.: On the Criteria to be Used in Decomposing Systems into Modules. Comunications of the ACM, Vol. 15, No. 12 (1972) 1053–1058.
[10] Suzuki, J., Yamamoto, Y.: Extending UML for Modelling Reflective Software Components. The Unified Modeling Language–Beyond the Standard. In: Second International Conference, Springer-Verlag LNCS 1723 (1999) 220–235.
[11] Rashid, A.: On to Aspect Persistence. In: GCSE Symposium, Erfurt, Germany, Springer-Verlag LNCS 2177, (2000) 26–36.
[12] Kienzle, J., Guerraoui, R.: AOP: Does it Make Sense? The Case of Concurrency and Failures. In: European Conference on Object-Oriented Programming (ECOOP), Springer-Verlag LNCS 2374 (2002) 37–61.
[13] A. Rashid, R. Chitchyan: Persistence as an Aspect. In: International Conference on Aspect-Oriented Software Development (AOSD), Boston, Massachusetts (2003) 120–129.
[14] Maes, P.: Computational Reflection. Technical Report 87_2, Artificial Intelligence Laboratory, Vrieje Universiteit Brussel (1987).
[15] Ortin, F., Cueva, J.M.: Non-Restrictive Computational Reflection. Elsevier Computer Standards and Interfaces, Vol. 25, No. 3 (2003) 241–251.
[16] Kiczales, G., Rivieres, J., Bobrow, D.G.: The Art of Meta-Object Protocol. MIT Press (1991).
[17] Ortin, F., Cueva, J.M.: Implementing a Real Computational-Environment Jump in order to Develop a Runtime-Adaptable Reflective Platform. ACM SIGPLAN Notices, Vol. 37, No. 8 (2002) 35–44.
[18] Smith, B.C.: Reflection and Semantics in a Procedural Language. Technical Report MIT-LCS-TR-272, MIT, Cambridge (1982).
[19] Gamma, E., Helm, R., Johnson, R., Vlisside, J.: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley (1995).
[20] Ley, M: DBLP: Digital Bibliography and Library Project. http://dblp.uni-trier.de/.
[21] Böllert, K.: On Weaving Aspects. In: European Conference on Object-Oriented Programming (ECOOP), Workshop on Aspect Oriented Programming, Lisbon, Portugal (1999) 301–302.
[22] Popovici, A., Gross, Th., Alonso, G.: Dynamic Homogenous AOP with PROSE. Technical Report, Department of Computer Science, ETH Zürich, Switzerland (2001).
[23] Hölzle, U., Ungar, D.: A Third-Generation SELF Implementation: Reconciling Responsiveness with Performance. In: Proceedings of the Object-Oriented Programming Languages, Systems and Applications (OOPSLA), Portland, Oregon (1994) 229–243.