# Implementando Acceso Directo y Secuencial a Colecciones de Datos mediante Aspectos¥

Jordi Marco[1], Xavier Franch[1], Jordi Àlvarez[2]

[1]Universitat Politècnica de Catalunya (UPC)
c/ Jordi Girona 1-3 (Campus Nord, C6) 08034 Barcelona
{jmarco, franch}@lsi.upc.es
http://www.lsi.upc.es/~gessi/
[2]Universitat Oberta de Catalunya (UOC)
Av. Tibidabo 39-43, 08035 Barcelona
jalvarezc@uoc.edu

**Resumen**. Las bibliotecas de colecciones de datos juegan un papel importante en el desarrollo de software basado en componentes. Las colecciones contenidas en las bibliotecas de este tipo (JCF, STL, LEDA, etc.) implementan un modelo matemático que define uno o más métodos de acceso a los elementos (acceso por clave, acceso al último elemento almacenado, etc.). Además, la mayoría de estas bibliotecas permiten un tipo de acceso diferente, más eficiente, a los elementos, que puede ser acceso directo (e.g., mediante la posición obtenida en el momento de insertar el elemento) o acceso secuencial (normalmente usando el concepto de iterador). Este tipo de acceso eficiente presenta ciertos riesgos respecto a criterios tales como precisión y adecuación funcional, que no son resueltos adecuadamente en las bibliotecas actuales. En este artículo, se presentan sendos patrones de diseño que proporcionan una solución genérica al problema, y su implementación mediante aspectos. Los patrones introducen nuevos tipos de datos y nuevas operaciones que dotan a las bibliotecas de una uniformidad total y un alto grado de extensibilidad. El uso de aspectos permite disociar estos tipos de acceso y la funcionalidad misma de la colección, concentrando en los aspectos la gestión de la persistencia de las posiciones, la modificación controlada durante las iteraciones, etc. La propuesta se implementa mediante AspectJ.

## 1. Introducción

La mayoría de lenguajes de programación orientados a objetos incluyen algunas bibliotecas estándar de componentes reutilizables como parte de su definición. Una clase habitual de tales bibliotecas son las bibliotecas de colecciones de datos. Una *colección* (también denominada *contenedor* en algunos contextos) puede definirse como un objeto que contiene otros objetos. Algunos ejemplos de colecciones son los conjuntos, las tablas y las secuencias. Las bibliotecas de colecciones se han utilizado en las últimas décadas en diferentes campos (computación geométrica, ingeniería mecánica, inteligencia artificial, etc.) y siguen mostrando su utilidad en dominios emergentes tales como los sistemas de información geográficos, la bioinformática y la infraestructura de claves públicas. Algunas bibliotecas de colecciones representativas son la Java Collection Framework (JCF) [AGH00], la Standard Template Library (STL) [MS96], la Booch Components (BC) [BV90, BWW99] y la Library of Efficient Data types and Algorithms (LEDA) [MN99].

¥ Este trabajo se ha desarrollado en el marco del proyecto CICYT TIC2001-2165.

Las colecciones contenidas en las bibliotecas de este tipo implementan un modelo matemático que define uno o más métodos de acceso a los elementos (acceso por clave, acceso al último elemento almacenado, acceso al elemento menor según un criterio de ordenación, etc.). Además, la mayoría de estas bibliotecas permiten un tipo de acceso diferente a los elementos, que puede ser acceso directo (por ejemplo, mediante la posición obtenida en el momento de insertar el elemento) o acceso secuencial (normalmente usando el concepto de iterador). A este tipo de acceso lo denominamos *acceso eficiente*, pues la eficiencia es el motivo de su existencia. El acceso eficiente presenta ciertos riesgos, especialmente respecto a criterios tales como precisión y adecuación funcional, que no son resueltos adecuadamente en las bibliotecas actuales. En concreto, su uso puede impactar negativamente en los siguientes factores de calidad (que hemos reconocido como los más significativos en el dominio de las bibliotecas de colecciones [FM03]:

- Adecuación funcional: Algunas operaciones, como la actualización de la colección durante un acceso secuencial, pueden no ser permitidas. Otras pueden ser restringidas a ciertos tipos de colecciones.
- Adaptabilidad: Algunas implementaciones, en particular aquéllas que reubican elementos durante la vida de las colecciones (por ejemplo, algunas estrategias de resolución de colisiones en tablas de dispersión [Knu73]), pueden ser descartadas como objeto del acceso eficiente.
- Cambiabilidad: Puede ser complicado modificar o extender la biblioteca debido a ciertas suposiciones para el correcto funcionamiento de este tipo de acceso.
- Precisión: Generalmente existen ciertas situaciones que no se controlan totalmente, por ejemplo el acceso directo a elementos que han sido borrados de la colección.
- Calidad interna: El diseño e implementación pueden ser inadecuados desde el punto de vista de la reutilización de código y del acoplamiento introducido entre las clases.

En este artículo, se presentan sendos patrones de diseño que proporcionan una solución genérica al problema. Los patrones introducen nuevos tipos de datos y un conjunto de operaciones que dotan a las bibliotecas de una uniformidad total y un alto grado de extensibilidad. Además, proponemos implementar estos patrones en el marco de la orientación a aspectos [AOS04]. El uso de aspectos permite disociar el acceso eficiente y la funcionalidad misma de la colección, concentrando en los aspectos la gestión de la persistencia de las posiciones, la modificación controlada durante las iteraciones, etc. La propuesta se implementa mediante AspectJ [Kic+01].

El resto del artículo se estructura de la manera siguiente. En la sección 2 resumimos las características más habituales del acceso eficiente que podemos encontrar en las bibliotecas de colecciones habituales. En las secciones 3 y 4 introducimos dos patrones de diseño para el acceso directo y secuencial a las colecciones, respectivamente. En la sección 5 mostramos la implementación de los patrones mediante aspectos. En la sección 6 evaluamos la propuesta mediante los criterios identificados en la sección 2. Finalmente, en la sección 7 proporcionamos las conclusiones.

## 2. Problemas habituales en los mecanismos de acceso eficiente

El acceso eficiente por posición puede implementarse mediante dos estrategias diferentes: usando directamente el concepto de apuntador proporcionado por el lenguaje

de programación, o bien introduciendo un concepto nuevo de posición que abstraiga el anterior y efectúe un control más o menos estricto en el acceso. La primera estrategia la siguen entre otras JCF y BC, mientras que la segunda la adoptan STL y LEDA.

El primer caso es especialmente grave, pues no existe ningún tipo de control de acceso a la colección. Pero incluso en el segundo caso, el control acostumbra a no ser exhaustivo y por ello pueden darse algunos inconvenientes, entre los que destacamos:

- Inexistencia de información que permita discernir cuándo se accede a un elemento que ya se ha borrado de la estructura.
- Obsolescencia cuando los elementos se reubican en la estructura a causa de inserciones o supresiones que exigen cambios internos.
- Inexistencia de información que permita asegurar que los accesos se realizan siempre a la colección que corresponde.
- Ausencia de operaciones de control para detectar las situaciones mencionas u otras igualmente perniciosas.

Por lo que se refiere al acceso secuencial, casi siempre se plasma mediante iteradores. Las estrategias de implementación de éstos son básicamente dos: puede haber un concepto de iterador totalmente diferente al del acceso directo, o se puede utilizar el mismo concepto de posición para los iteradores y para las posiciones (e.g., STL o LEDA). Sobre su uso, algunas propuestas (e.g., JCF) presentan la limitación de que son unidireccionales. Pero especialmente, nos encontramos con problemas parecidos al caso del acceso directo, agravado con otro tipo de inconveniente: la posible interferencia de las actualizaciones durante un recorrido secuencial. Destacamos:

- La inserción de elementos que van a parar a la parte todavía no recorrida; o bien la supresión de elementos de esta parte no recorrida. Algunas propuestas actualizan el iterador y tienen en cuenta estas modificaciones, otras no.
- Inserciones o supresiones que provocan movimientos de los elementos en la colección y que pueden hacer desaparecer elementos de la parte pendiente de recorrer.
- En particular, la supresión mediante un acceso directo del elemento actual en el recorrido secuencial puede tener consecuencias gravísimas.

En algunas colecciones, estas situaciones no se controlan (e.g., STL en el caso de reorganizaciones internas) o bien se controlan mediante excepciones pero no a priori (e.g., JCF). En otros casos, se identifican los problemas pero no se controlan, siendo responsabilidad del usuario el uso correcto de la biblioteca (e.g., LEDA).

Por último, muchas de las bibliotecas mencionadas presentan ciertas deficiencias estructurales que perjudican la calidad interna y afectan a criterios de calidad como la cambiabilidad y la adaptabilidad. Estas deficiencias se deben sobre todo a que el código se concentra muchas veces en las partes bajas de la jerarquía de clases (o directamente en las hojas, como el caso de JCF), de manera que la adición de nuevas colecciones, o nuevas implementaciones de colecciones existentes, exige la escritura de más código del que se hubiera escrito con una estructuración diferente. Con frecuencia, los iteradores se implementan partiendo de cero en cada implementación abstracta que existe en la jerarquía de colecciones.

Como resumen, podemos decir que las propuestas existentes presentan algunos inconvenientes que pueden dificultar su uso adecuado. Estos inconvenientes se refieren tanto al grado de adecuación funcional y precisión de los mecanismos ofrecidos como a la calidad de la estructura interna que no asegura un buen comportamiento respecto la adaptabilidad y cambiabilidad de estas bibliotecas. Por ello se justifica la formulación de nuevas propuestas que solucionen los inconvenientes mencionados.

# 3. El patrón de diseño *Atajo*

Definimos un *atajo* [FM02] como un camino alternativo de acceso a los objetos almacenados en una colección, que cumple las siguientes propiedades:

- Eficiencia: El acceso a los datos mediante los atajos se realiza en tiempo asintótico constante $O(1)$.
- Precisión: No es posible acceder a datos inesperados. Los accesos indebidos son controlados y evitados.
- Abstracción: El acceso a los datos se realiza sin saber cómo se almacenan los objetos en la estructura de datos utilizada para implementar la colección.
- Robustez. Un acceso directo a los objetos se considera robusto si:
  - Está asociado a un único objeto.
  - No cambia mientras el objeto permanece en la colección, incluso si la estructura de datos subyacente cambia la localización de dicho objeto.
  - Es posible conocer si el objeto asociado está todavía en la colección.
- Evolución: Los atajos se crean y se invalidan a medida que se insertan y se borran los objetos de la estructura.

Estas propiedades nos permiten afirmar que los atajos son un método de acceso a los objetos almacenados en una colección más adecuado que los métodos de acceso que proporcionan directamente los lenguajes de programación. Tomemos por ejemplo las referencias de Java, y veamos los inconvenientes que presentan:

- No se pueden invalidar todas las referencias externas cuando se borra un objeto de la colección y por lo tanto, una referencia externa no permite discernir si el objeto se encuentra aún en la colección o no.
- Se necesita conocer la implementación de la localización del objeto y por lo tanto son totalmente dependientes de la implementación, perdiendo completamente la propiedad de abstracción de los atajos.
- En estructuras de datos donde los objetos pueden ser reubicados, las referencias no son persistentes.
- Tampoco son precisas, ya que se puede acceder a una localización que ha sido liberada e incluso reutilizada, provocando un acceso indebido.
- Por último, el acceso a la estructura de datos mediante referencias puede modificar el comportamiento funcional de la misma.

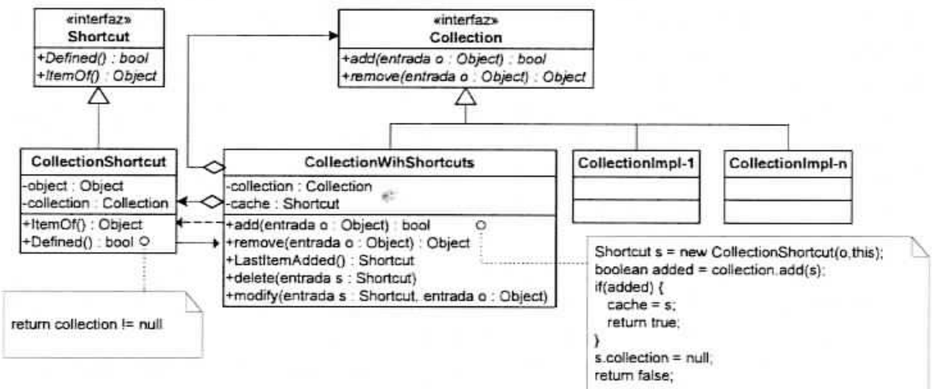


**Fig. 1.** El patrón de diseño *Atajo*.

El patrón de diseño *Atajo* [MF03] muestra los elementos necesarios para incorporar los atajos a las colecciones. La figura 1 muestra la estructura de este patrón. Se puede observar que la jerarquía de colecciones está organizada como una jerarquía de agregación (mediante la aplicación del patrón *Composite* [GHJ+95]):

- `Collection`. Define el interfaz genérico de la colección. Sin pérdida de generalidad, podemos suponer que toda colección proporciona como mínimo un método de inserción y un método de supresión.
- `Shortcut`. Define el interfaz correspondiente al acceso directo a los objetos. En concreto, define los métodos: `Defined()`, que permite saber si el objeto esta todavía en la colección o no; e `ItemOf()`, que retorna el objeto asociado al atajo en caso de que éste sea válido.
- `CollectionShortcut`. Esta clase implementa el interfaz `Shortcut` como un par de referencias: una referencia a la colección en la que se encuentra el objeto, y una referencia al objeto mismo. La primera de estas referencias permite controlar que el atajo se usa siempre para acceder al contenedor adecuado. En caso de que el atajo esté invalidado, la referencia a la colección tiene el valor `null`.
- `CollectionWithShortcuts`. Añade al interfaz `Collection` los métodos correspondientes a los atajos: `LastItemAdded()`, que retorna el atajo asociado al último objeto insertado en la colección (almacenado en el atributo `cache`); `delete()`, que permite borrar un objeto mediante su atajo asociado; y `modify()`, que permite sustituir en la colección el objeto asociado a un atajo por otro. Esta clase mantiene una referencia a una colección que se corresponde a una implementación concreta de la colección, `CollectionImpl-k`; así, cada vez que se inserta un objeto en la colección se crea un nuevo objeto `CollectionShortcut`, que mantiene la referencia a la colección junto con la referencia al objeto insertado, que se almacena en la implementación concreta de la colección. De forma similar al borrar un objeto, primero se borra de la implementación concreta el objeto `CollectionShortcut` asociado al mismo y luego se invalida el `CollectionShortcut` (i.e., la referencia a la colección se pone a `null`).
- `CollectionImpl-k`. Estas clases se corresponden a las diferentes implementaciones de las diferentes colecciones.

## 4. El patrón de diseño *Iterador*

El patrón de diseño *Iterador* [GHJ+95] proporciona un mecanismo de acceso secuencial a los objetos almacenados en una colección. En este trabajo utilizamos la propuesta presentada en [MF03] que propone una estructura diferente a [GHJ+95] y que permite definir iteradores bidireccionales totalmente independientes de la implementación de la colección. Tanto la bidireccionalidad como la independencia se consiguen manteniendo los objetos en una lista doblemente enlazada y almacenando una referencia al nodo de la lista en la colección. Esta propuesta permite entre otras ventajas que se pueda iterar en cualquier orden simplemente aplicando un algoritmo genérico de ordenación a la colección. Este algoritmo ordena la lista doblemente enlazada sin modificar la implementación concreta de la colección. La figura 2 muestra la estructura de este patrón que es muy similar a la del patrón *Atajo*. A continuación se presenta brevemente cada uno de los participantes de esta estructura que no aparecían en el patrón anterior:

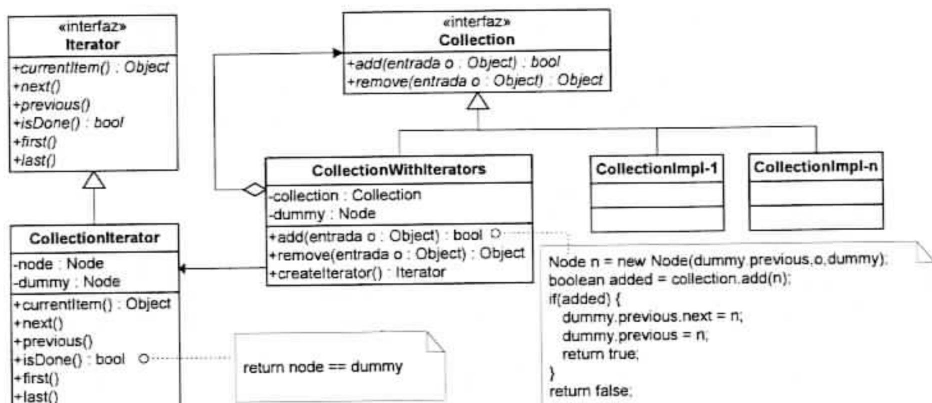


**Fig. 2.** El patrón de diseño *Iterador*.

- `Iterator`. Define el interfaz correspondiente al acceso secuencial a los objetos.
- `CollectionIterator`. Esta clase implementa el interfaz `Iterator` como una referencia al nodo actual de la lista doblemente enlazada (clase `Node`, no incluida en la figura), junto a una referencia a un nodo (`dummy`) que representa el elemento anterior al primero y posterior al último. El iterador se encuentra al final cuando el nodo actual es igual a `dummy`.
- `CollectionWithIterators`. Añade al interfaz `Collection` el método `createIterator()` que retorna un iterador apuntando al primer elemento. Igual que la clase `CollectionWithShortcuts` del patrón *Atajo*, esta clase mantiene una referencia a una colección que se corresponde a una implementación concreta; de esta manera cada vez que se inserta un objeto se crea un nuevo objeto `Node`, que se enlaza con el último elemento y con el `dummy`. De forma similar cuando se borra un objeto, primero se borra el `Node` asociado al mismo de la implementación concreta y después se desenlaza de la lista.

## 5. Implementación de los patrones *Atajo* e *Iterador* mediante aspectos

En esta sección se explica como implementar los patrones de diseño *Atajo* e *Iterador* en el marco del Diseño de Software Orientado a Aspectos (DSOA) [AOS04] utilizando el lenguaje AspectJ [Kic+01]. El uso de aspectos permite capturar en una única unidad el comportamiento que se repite en diferentes partes de la jerarquía de clases (y de manera transversal a ésta). La posibilidad de capturar regularidades transversales en unidades independientes evita tener el código correspondiente a los aspectos disperso entre las distintas clases de la jerarquía (*scattering*). Además, el resto de la aplicación queda totalmente libre del código correspondiente a dichos aspectos, evitando así también el código enmarañado que este tipo de situaciones acostumbran a producir (*tangling*). La interacción entre clases y aspectos se define utilizando puntos de encuentro (*joint points*) dentro del flujo de ejecución; por ejemplo la llamada a una operación, el retorno desde su ejecución, o la lectura o escritura de un atributo. La composición de aspectos y clases, que recibe el nombre de *weaving*, se puede realizar o bien en tiempo de compilación o bien en tiempo de ejecución dependiendo del sistema o lenguaje de aspectos utilizado. En el caso de AspectJ, que es el lenguaje usado

para este trabajo, se realiza en tiempo de compilación. Además de la integración del comportamiento de los aspectos mediante puntos de encuentro, en AspectJ es posible también definir comportamiento de manera estática (*structural crosscuting*). Esta posibilidad permite definir nuevas clases e interfaces, añadir métodos y atributos a clases ya existentes, y modificar la jerarquía de clases original para adaptarla a los comportamientos transversales que se definen mediante los aspectos.

La intención de los patrones de diseño *Atajo* e *Iterador* es, básicamente, añadir nuevas funcionalidades a la jerarquía de clases de la colección. Para ello proponemos utilizar las *introductions* de AspectJ que permiten añadir estáticamente operaciones y atributos a una clase sin necesidad de modificar su definición original. Por otro lado los dos patrones tienen una estructura similar que se basa en guardar los objetos en una localización independiente de la colección para posteriormente guardar dicha localización en la colección correspondiente. En otras palabras, los dos se basan en un concepto más elemental que denominamos *Localización* para el que definimos un nuevo aspecto, que utilizamos para definir el comportamiento transversal básico que permita a los aspectos *Atajo* e *Iterador* incorporar el comportamiento esperado. Ello posibilita además que el comportamiento de *Atajo* e *Iterador* pueda ser añadido a una colección de forma independiente, y que no haya ninguna dependencia entre ambos. La figura 3 muestra la estructura resultante de los patrones *Atajo* e *Iterador*.

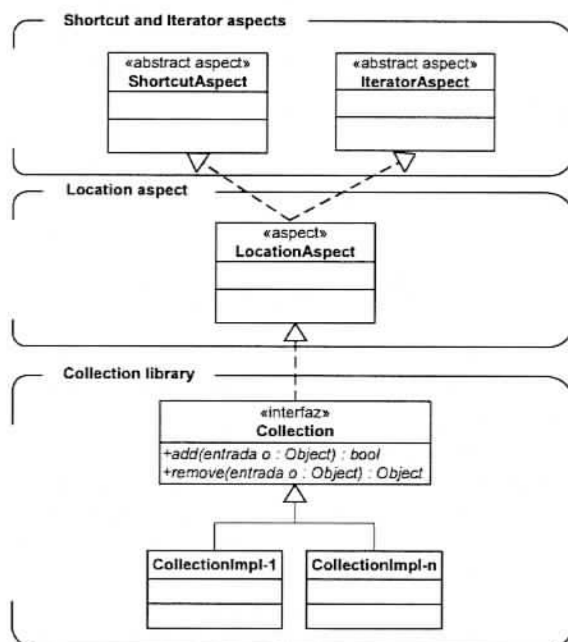


**Fig. 3.** Los patrones de diseño *Atajo* e *Iterador* usando aspectos.

El aspecto *Localización* se encarga de añadir a las operaciones de inserción y supresión de la colección el comportamiento necesario para poder crear localizaciones y trabajar con ellas. Para ello añade a la propia colección operaciones que se ejecutarán previamente y posteriormente a las operaciones de inserción y borrado, tal y como se puede apreciar en la figura 4. Los aspectos *Atajo* e *Iterador* utilizan la estructura añadida por el aspecto mencionado anteriormente para añadir el comportamiento corres-

pondiente a sendos patrones de diseño. En la figura 4 se puede observar gráficamente cómo *Atajo* usa la estructura añadida por *Localización*. La figura 5 muestra un extracto de la definición de este aspecto en AspectJ.

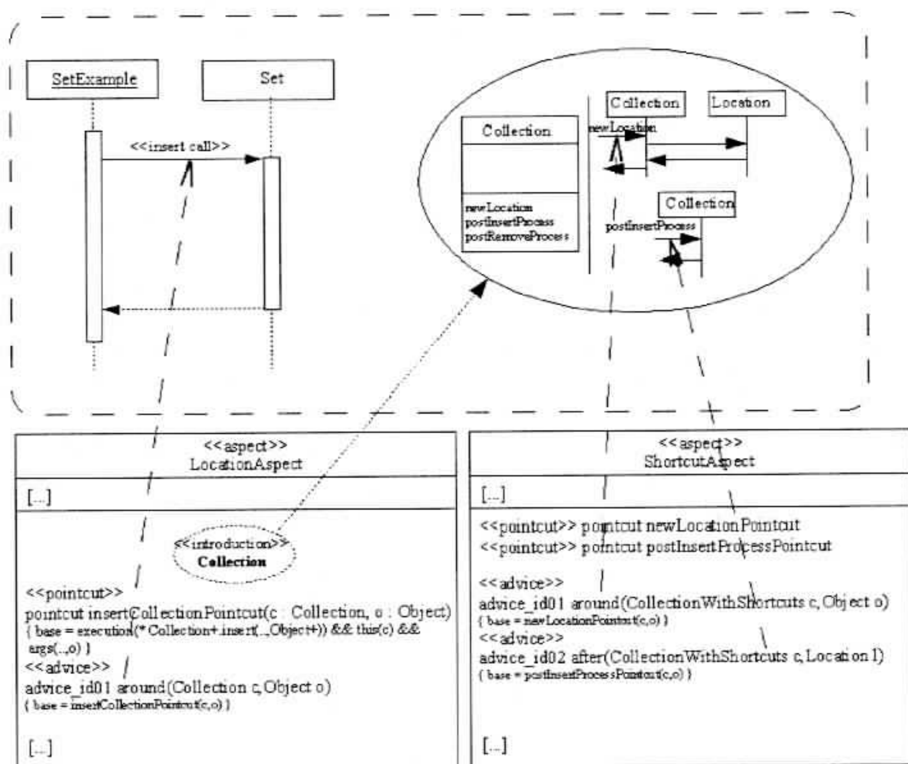


**Fig. 4.** Interacción entre una colección (*Set*) y los patrones *Localización* y *Atajo*.

Los aspectos *Atajo* e *Iterador* están definidos como abstractos. Esto permite proporcionar el comportamiento de manera totalmente independientemente del tipo de colección (conjunto, lista, etc.) al que vamos a añadir dicho comportamiento.

La situación es análoga a la definida por [HK02], donde se presenta la implementación de algunos patrones de diseño de [GHJ+95] mediante aspectos abstractos. Los detalles de la aplicación de los patrones a un dominio concreto se proporcionan en aspectos concretos que extienden el aspecto abstracto que implementa el patrón.

En el caso de los patrones *Atajo* e *Iterador*, el dominio concreto de aplicación se puede equiparar en realidad con la una colección concreta de la biblioteca a la que pretendemos añadir las funcionalidades correspondientes. Entonces, la adaptación para una colección concreta del comportamiento general definido en los aspectos abstractos `ShortcutAspect` e `IteratorAspect`, se realiza mediante sendos aspectos concretos que extienden a los dos anteriores.

Los aspectos abstractos, a su vez, proporcionan un comportamiento estándar para los aspectos concretos. Con ello, en algunas ocasiones únicamente será necesario definir los aspectos concretos, sin añadir comportamiento adicional, ya que el proporcionado por los aspectos abstractos será suficiente. Por ejemplo, en el caso de aplicar el aspecto *Iterador* a un conjunto no hace falta añadir ningún comportamiento adicio-

nal, mientras que en el caso del aspecto *Atajo* es necesario cambiar el comportamiento del método `modify()` para evitar que pueda producir duplicados en un conjunto. Las figuras 6 y 7 muestran respectivamente el comportamiento general del método `modify()` definido en el aspecto *Atajo* (`ShortcutAspect`) y su concreción en el caso de un conjunto (`ShortcutSetAspect`).

```
public privileged aspect LocationAspect {

 Location Collection.NewLocation(Object o){
       return new Location(o);
 }

 void Collection.postInsertProcess(Location l, boolean added){
 }

 void Collection.postRemoveProcess(Location l){
 }

 pointcut insertCollectionPointCut(Collection c, Object o):
              execution(* Collection+.insert(.., Object+)) &&
              this(c) && args(..,o);

 boolean around(Collection c, Object o):
                     insertCollectionPointCut(c, o){
       Location l = c.NewLocation(o);
       boolean added = proceed(c,l);
       c.postInsertProcess(l,added);
       return added;
 }

 pointcut removeCollectionPointCut(Collection c):
              execution(* Collection+.remove(..)) && this(c);

 Object around(Collection c) : removeCollectionPointCut(c){
       Location l = (Location) proceed(c);
       if(l==null) return null;
       c.postRemoveProcess(l);
       return l.object;
 }
...
}
```

**Fig. 5.** El aspecto *LocationAspect* en AspectJ (extracto)

```
public abstract aspect ShortcutAspect {
...

  void CollectionWithShortcuts.modify(Shortcut s, Object o){
       if (((CollectionShortcut)s).collection == this)
              ((Location)s).setObject(o);
       else undefinedShortcut((CollectionShortcut)(s));
  }

declare parents: CollectionWithShortcuts extends LocationAspect.Collection;
...
}
```

**Fig. 6.** El método `modify()`: definición general.

```
public privileged aspect ShortcutSetAspect extends ShortcutAspect {

  declare parents : Set extends CollectionWithShortcuts;
...
  void Set.modify(Shortcut s, Object o){
       if (((CollectionShortcut)s).collection == this){
              if(!belongs(o)) ((Location)s).setObject(o);
              else
                 System.out.println("Other Shortcut to Object " +
                 o + " is in this set");
              }
              else undefinedShortcut((CollectionShortcut)(s));
       }
...
}
```

**Fig. 7.** El método `modify()`: definición para los conjuntos.

## 6. Evaluación de la propuesta

La implementación de los patrones *Atajo* e *Iterador* mediante aspectos permite definir de manera centralizada todo el comportamiento correspondiente a estos patrones, evitando la dispersión de este comportamiento entre las clases de la biblioteca de colecciones. Adicionalmente el comportamiento básico de las colecciones de la biblioteca y el comportamiento de los dos patrones antes mencionados queda definido como dos unidades independientes. Con ello, el comportamiento básico de la biblioteca queda libre del posible enmarañamiento causado por el código correspondiente a la implementación de los patrones *Atajo* e *Iterador*. Estas dos ventajas, inherentes al diseño de software orientado a aspectos, nos permiten en nuestro caso obtener otras ventajas adicionales especialmente importantes. En primer lugar, la implementación OO de nuestros dos patrones obligaba a tener en cuenta los patrones en el diseño de la biblioteca de colecciones. Es más, para añadir el comportamiento de los dos patrones a una biblioteca de colecciones ya existente, era necesario revisar su diseño y retocar su implementación con la inclusión del código correspondiente a ambos patrones.

El uso de la orientación a aspectos nos permite incorporar los aspectos *Atajo* e *Iterador* a cualquier biblioteca de colecciones sin necesidad de realizar ninguna modificación en ésta, ni a nivel de diseño ni a nivel de codificación. Como hemos visto en la sección anterior, la integración entre una biblioteca de colecciones cualquiera y el comportamiento general de los dos patrones definido en sus correspondientes aspectos abstractos, se realiza mediante la definición de aspectos concretos que extienden a estos. Es decir, en lugar de adaptar la biblioteca de colecciones para poder implementar los patrones *Atajo* e *Iterador* (versión OO), estamos especificando de una manera casi totalmente declarativa cómo aplicar los dos patrones para el caso de una biblioteca de colecciones concreta.

El hecho de poder aplicar nuestros dos patrones a cualquier biblioteca de colecciones sin necesidad de modificar esta última es práctico, eficiente y seguro. Se evita, evidentemente, la posibilidad de introducir errores en el código de la biblioteca.
Por otro lado, la versión de la biblioteca que incorpora los aspectos es totalmente intercambiable con la versión de la biblioteca que no los incorpora. Con ello, se puede proporcionar una versión de la biblioteca más "ligera" a aquellas aplicaciones que no necesiten las funcionalidades proporcionadas por los patrones *Atajo* e *Iterador*. Para

ello, únicamente hay que especificar al compilador de AspectJ si queremos o no incluir los aspectos en el proceso de *weaving*. En cambio, en el caso de la implementación orientada a objetos, al necesitar o bien un *wrapper* (ver figuras 1 y 2) o bien la redefinición de cada una de las colecciones, puede requerir algún cambio en la aplicación cliente.

La tabla 1 resume las ventajas de la propuesta en comparación con otras 4 bibliotecas existentes, JFC, STL, BC y LEDA (v. [MF00b] para una justificación). Las ventajas se deben tanto a la definición de los patrones (que impacta especialmente en la adecuación funcional y la precisión) como a la implementación por aspectos (que incide en los otros criterios). Podemos observar que nuestra propuesta favorece la mayoría de criterios identificados excepto la eficiencia (una indirección en las actualizaciones, espacio adicional para los atajos), si bien incluso con este criterio los inconvenientes no son insalvables. En [MF00b] pueden consultarse unos *benchmarks* sobre la eficiencia temporal y unos cálculos sobre la eficiencia espacial para una versión anterior de la propuesta de atajos implementada sin aspectos.

| | Atajos | JCF | STL | BC | LEDA |
|---|---|---|---|---|---|
| Iteradores | Bidireccionales<br>Persistentes<br>Actualización permitida | Hacia delante<br>No persistentes<br>Actualización no permitida | Bidireccionales<br>No persistentes<br>Actualización puede dañar | Hacia delante<br>No persistentes<br>Actualización no permitida | Bidireccionales<br>No persistentes<br>Actualización no permitida |
| Acceso directo | Persistente | Por referencias | No persistente | Por referencia | No persistente |
| Adaptabilidad | Fácil | Costosa | Costosa | Costosa | Costosa |
| Modularidad | Alta | Baja | Media | Baja y con acoplamiento | Media |
| Reusabilidad | Alta | Baja | Baja | Baja | Baja |
| Precisión | Total | No total | No total | No total | No total |
| Tiempo | Acceso óptimo<br>Actualización con indirección | Muy buena | Óptima | Muy buena | Óptima |
| Espacio | Estructura adicional para atajos | Muy buena | Óptima | Muy buena | Óptima |

**Tabla 1**: comparación de la propuesta con otras bibliotecas existentes

## 7. Conclusiones

En este artículo hemos presentado una solución al problema del acceso eficiente a los elementos contenidos en una colección. Las aportaciones más relevantes de la propuesta son:

- A nivel de diseño, hemos formulado unos patrones absolutamente general que puede aplicarse a cualquier marco de referencia (v. [MF00a] y [MF04] para propuestas en Ada95 y Java).
- Estos patrones reconcilian el acceso eficiente con otros criterios que en otras propuestas salen perjudicados, como la adecuación funcional, la precisión, la adaptabilidad y la cambiabilidad.
- Hemos implementado la propuesta mediante aspectos (usando AspectJ) lo que redunda en una mayor calidad de la estructura interna de la propuesta, mejorando la comprensión y permitiendo un uso versátil según se quiera incorporar o no los controles de precisión que caracterizan nuestra propuesta.

La implementación de los aspectos y su aplicación a un tipo de colección determinado, los conjuntos, está accesible en la dirección http://www.lsi.upc.es/~gessi/atajo.zip.

En la sección anterior hemos comparado nuestra propuesta con cuatro de las más extendidas en el área. Existen otras propuestas no tan difundidas que tratan algunas de las características objeto de nuestro estudio. Por ejemplo, algunas extensiones de STL como Safe STL o Boost Iterators Adaptors [Hor04] ofrecen operaciones para discernir si un iterador ha quedado obsoleto. Otra propuesta interesante es [TV01], que utiliza también patrones de diseño para estructurar la biblioteca que proponen, aunque no la implementan mediante aspectos. Además, estos patrones no son tan potentes como los aquí propuestos, concretamente: son dependientes del lenguaje y el concepto de iterador (denominado *position*) presenta algunos problemas típicos como: no son independientes de la implementación; cuando se insertan o suprimen elementos los iteradores pueden quedar obsoletos y el orden de iteración puede cambiar; y no es posible iterar en otro orden que el inherente a la estructura de datos.

## Referencias


[AGH00] K. Arnold, J. Gosling and D. Holmes. *The Java Programming Language*. Addison-Wesley, 3rd edition, 2000.

[AOS04] Aspect-Oriented Software Development, http://www.aosd.net. Last accessed June'04.

[BV90] G. Booch and M. Vilot. "The Design of the C++ Booch Components". In *Proceedings of OOPSLA '90*, volume 25 of *SIGPLAN Notices*, pages 1-11. ACM, 1990.

[BWW99] G. Booch, D.G. Weller and S. Wright. "The Booch Library for Ada 95 (version 1999)". Available at http://www.pogner.demon.co.uk/components/bc/.

[FM02] X. Franch and J. Marco. "Adding Alternative Access Paths to Abstract Data Types". In *Successful Software Reengineering*, chapter 18, pages 256-267. IRM Press, 2002.

[FM03] X. Franch and J. Marco. "A Quality Model for the Ada Standard Container Library". In *Reliable Software Technologies Ada-Europe 2003*, LNCS 2655, pp. 283-296, 2003.

[GHJ+95] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[HK02] J. Hannemann, G. Kiczales. "Design pattern implementation in Java and AspectJ". In *Proceedings of OOPSLA '02*, p. 161-173, 2002.

[Hor04] C.S. Horstmann. "Safe STL". Available at http://www.horstmann.com/safestl.html, last accessed June'04.

[Kic+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin. "Aspect-Oriented Programming". In *Procs. ECOOP'97*, LNCS 1241, pp. 220-241.

[Kic+01] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W. G. "An Overview of AspectJ". In *Proceedings of ECOOP'01*, LNCS 2072, 2001, 327-353.

[Knu73] D.E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 1973.

[MF00a] J. Marco and X. Franch. "Reengineering the Booch Component Library". In *Reliable Software Technologies Ada-Europe 2000*, LNCS 1845, Springer-Verlag, pp. 96-111, 2000.

[MF00b] J. Marco and X. Franch. "Bridging the Gap Between Design and Implementation of Component Libraries". Technical Report LSI-00-79-R, UPC, 2000.

[MF03] J. Marco and X. Franch. "A Framework for Designing and Implementing the Ada Standard Container Library". *In Proceedings of the ACM SIGAda Conf.*, pp. 49-61, 2003.

[MF04] J. Marco and X. Franch. "Adding Efficient and Reliable Access Paths to the JCF". Technical Report LSI-04-19-R, Universitat Politècnica de Catalunya, 2003.

[MN99] K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[MS96] D.R. Musser, A. Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, 1996.

[TV01] R. Tamassia, L. Vismara. "A case study in Algorithm Engineering for Geometric Computting". *Intl' Journal Computational Geometry and Applications*, 11(1):15-70, 2001.