Tipo de Datos Abstractos y Programación Orientada a Objetos

Materia: Análisis Comparativo de Lenguajes

Departamento de Informática
Universidad Nacional de San Luis (UNSL)
San Luis. Argentina. Noviembre de 2019

1. Introducción

Casi todos los lenguajes de programación proveen, hoy en día, algún soporte para la *Programación Orientada a Objetos* (POO) (incluyendo a Lisp y Cobol). La integración de la POO en estos lenguajes, ha adoptado diferentes formas que van desde incorporar características de la POO en *lenguajes funcionales* (por ejemplo CLOS) y en *lenguajes imperativos* (por ejemplo C++) al extremo de tener lenguajes completamente basados en los principios de la POO (por ejemplo SMALLTALK). La POO tiene sus raíces en SIMULA-67, un lenguaje que ya en el año 1967 introduce conceptos como *clases*, *objetos*, *herencia*, etc. Sin embargo, se considera que los conceptos de la POO no fueron completamente desarrollados hasta el surgimiento de SMALLTALK 80, para muchos, el único lenguaje de POO "puro".

Se considera que un lenguaje orientado a objetos debe proveer soporte para tres características claves:

- Tipos de Datos Abstractos.
- Herencia.
- Ligadura dinámica de mensajes a métodos.

Los Tipos de Datos Abstractos (TDA's) constituyen el eslabón final en la evolución del concepto de tipo de datos. Inicialmente no existía la concepción de la definición del tipo de un dato sino que se utilizaba un *conjunto de valores* que una variable podía tomar durante la ejecución de un programa. Por ejemplo en FORTRAN la variable `A` era una estructura de 100 valores reales: `REAL A(100)`. Luego, la *definición de tipo* especificando la estructura del objeto de dato y aplicable a una o más variables significó un paso más en la evolución de los lenguajes como Pascal. Finalmente, al extender el concepto de tipo para considerar todas las *operaciones que manipulan* esos objetos de datos, más la provisión de encapsulamiento dio origen al tipo de datos abstracto. La idea de *encapsulamiento* en estos casos, jugaba un rol fundamental en el diseño de abstracciones definidas por el programador. La posibilidad de definir TDAs es una característica provista por distintos lenguajes (como ADA83 y Modula-2) y cuando el lenguaje sólo soporta esta característica, se suele hablar de lenguajes *basados* en objetos[1].

[1]Complementar con el material del libro: pgs 234-240, Cap. 6, Programming Languages - Design and Implementation. Pratt y Zelkowitz. Cuarta Edición.

Si bien los TDAs han sido propuestos como las *unidades de reúso* candidatas para incrementar la producción de software, su uso (aislado de otras características como la herencia) también conlleva algunos problemas:

- el reúso de TDA's normalmente involucraba su modificación (y entendimiento en detalle).
- los TDA's, son independientes y se encuentran todos al mismo nivel $\Rightarrow$ dificulta reflejar organización en categorías del dominio (jeraquías).

El concepto de *herencia* que se explica en la siguiente sección, permite abordar ambos tipos de problemas.

2. Herencia

La *herencia* permite reusar un TDA existente sin modificarlo. El programador sólo debe seleccionar un TDA adecuado, declarar una subclase de éste que herede la funcionalidad y estructura necesaria y "moldearla" para alcanzar los nuevos requerimientos del problema. Esto además permite definir jerarquías que reflejan el espacio del problema. Así, por ejemplo, la Figura 1 podrá representar una jerarquía de figuras correspondiente a un sistema gráfico hipotético.

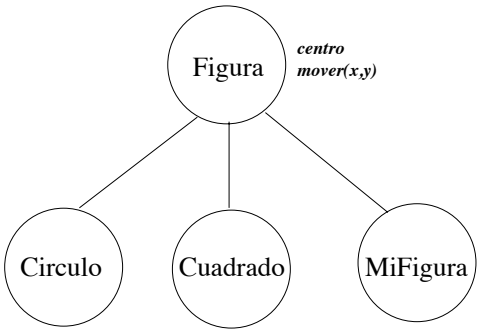


Figura 1: Una jerarquía de figuras

Cuando se utiliza la herencia para relacionar dos clases, de la manera mostrada en la Figura 2, se introducen una serie de conceptos, que son referenciados de distintas maneras en la comunidad de POO. Así por ejemplo, y siguiendo la terminología de SIMULA-67, el término *clase* se utiliza en POO para referenciar a los populares TDAs. La Tabla 1 muestra en su primera columna, algunos de los términos frecuentemente usados en POO con una breve descripción de cada uno de ellos o su analogía con otros términos usados para TDAs.

Continuando con otros conceptos básicos en POO, un elemento esencial de este paradigma es la idea de *mensaje*. Un mensaje está constituido esencialmente por 3 partes:

1. un objeto receptor
2. el nombre del mensaje
3. los argumentos

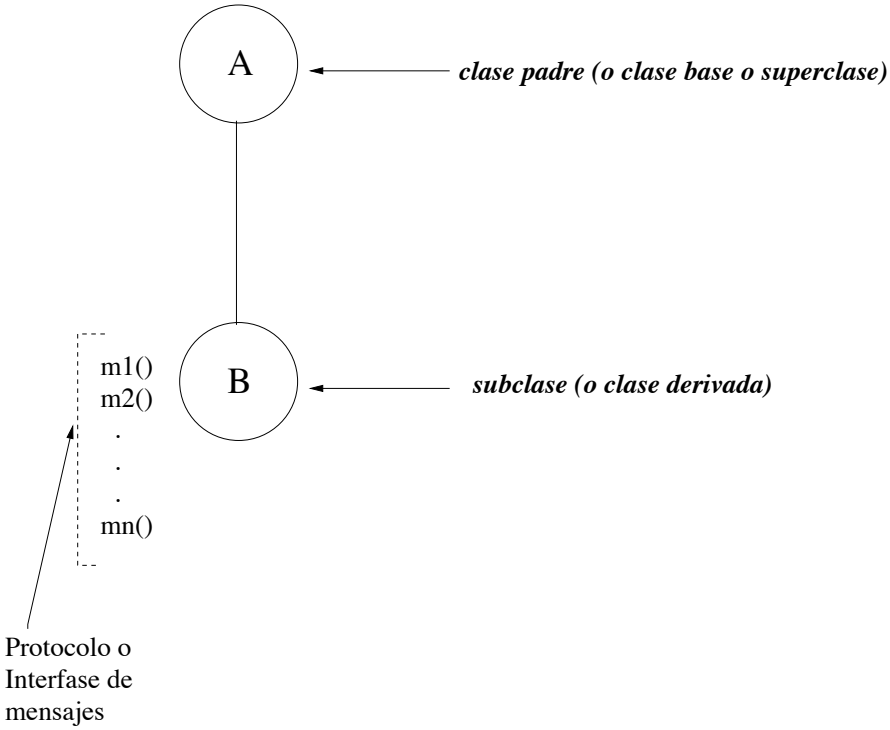


Figura 2: Una jerarquía de figuras.

Término	Descripción
Clase	Nueva forma de referenciar a los TDA
Objeto	Instancia de una clase (TDA)
Subclase o clase derivada	Clase definida desde otra clase a partir de la herencia
Superclase o clase padre	Clase a partir de la cual se crea una nueva clase
Método	Subprograma que describe operación para objetos de una clase
Mensaje	Invocación (o llamada) a un método.
Protocolo (interface) de mensaje	Colección completa de métodos de un objeto

Tabla 1: Terminología usual en POO.

Así, por ejemplo, si en C++ una variable `lis` es un puntero a objetos de la clase `Lista` que tiene un método `insertar` con dos argumentos, uno indicando el elemento a insertar y el otro la posición, la siguiente será una expresión de mensaje válida en este lenguaje:

```
lis->insertar(elem,3);
```

donde `lis` es el objeto receptor, `insertar` es el nombre del mensaje y `elem` y `3` son los argumentos. Un ejemplo similar de mensaje en el lenguaje SMALLTALK, tendría la forma

```
lis insertar: elem enPos: 3
```

donde `lis` nuevamente es el objeto receptor, `elem` y `3` son los argumentos y el nombre del mensaje es `insertar:enPos:`.

Otro concepto básico en POO son las formas de *control de acceso* disponibles para permitir al diseñador del programa ocultar partes de su TDA a los clientes del TDA. Así, por ejemplo, es común que los *datos internos* de un TDA (datos miembros en C++, variables de instancia en SMALLTALK) sean declarados como *privados* (`private` en C++) mientras que los métodos del TDA sean declarados como *públicos* (`public` en C++). De esta forma, los usuarios de un TDA tendrán acceso a los métodos de la interface, pero éstos serán los únicos autorizados para modificar los datos internos. Sin embargo, si consideramos que las subclases (o clases derivadas) de un TDA son otros potenciales clientes, algunos lenguajes de POO (como C++ y Java) incluyen una tercera categoría de control de acceso, denominada *protegida* (`protected` en C++) para permitir el acceso a las clases derivadas. Es importante notar que si bien los lenguajes modernos de POO como C++ o Java permiten especificar qué tipo de control de acceso usar en cada caso, lenguajes de POO tradicionales como SMALLTALK trabajan con formas de control por defecto. Por ejemplo, las variables de instancia de SMALLTALK son *protegidas* y los métodos *públicos*.

Respecto a los tipos de datos y métodos que podemos definir en una clase, los mismos se pueden dividir en métodos y variables de *instancia*, o de *clase*.

Las *variables de instancia* son aquellos datos internos propios de cada instancia (objeto) que se crean cada vez que se crea un nuevo objeto. Los *métodos de instancia* son los métodos usuales definidos en el TDA que se pueden enviar a cualquier instancia (objeto) de la clase. Como ejemplos de variables y métodos de instancia podemos mencionar a los datos y funciones miembros tradicionales de C++.

Las variables de clase son aquellos datos internos que no pertenecen a ningún objeto en particular, sino que pertenecen a la clase y son por lo tanto compartidos por todos los objetos que se creen de la misma. Los métodos de clase por su parte, son aquellos que pueden ser invocados haciendo referencia directamente al nombre de la clase, sin necesidad de enviarle el mensaje a ningún objeto (instancia) particular. Para obtener este tipo de variables y métodos en C++ se usa el calificador `static` en la definición de los datos y funciones miembro.

En SMALLTALK por su parte, el carácter de instancia y de clase de una variable o un método se especifica directamente en la interface al momento de crear la clase. Sin embargo, es sencillo identificar en un trozo de código SMALLTALK si un método es de clase o de instancia, dependiendo de si el mensaje está siendo enviado a una clase o a una instancia. Si tomamos, por ejemplo, el siguiente código SMALLTALK

```
p <- Persona nueva:'Juan Perez' edad:25.
p darnombre.
```

Podríamos reconocer que el método `nueva:edad:` es un método de clase que se le envía a la clase `Persona` y cuyo efecto es retornar un nuevo objeto de esta clase con el nombre y edad especificado en el envío del mensaje. Una vez que este objeto es asignado a `p`, se le podrán enviar cualquiera de los métodos de instancia válidos definidos para `Persona`, como por ejemplo `darnombre`.

3. Polimorfismo y ligadura dinámica

La tercera característica (además de los TDAs y la herencia) que distingue a la POO es una forma de *polimorfismo* provista por la *ligadura dinámica* de funciones a definiciones de métodos. Con este concepto referenciamos a la idea de que la decisión de cuál es el método que efectivamente se ejecutará como respuesta al envío de un mensaje es demorada hasta el momento de la ejecución efectiva de la sentencia durante la corrida del programa.

Esta forma de polimorfismo requiere fundamentalmente de dos mecanismos:

- ligadura dinámica de mensajes a definiciones de métodos.
- variables polimórficas.

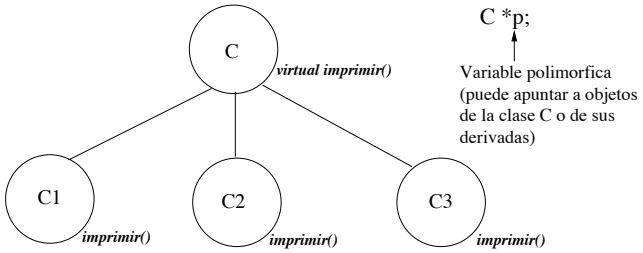
La primera característica, es esencial en cualquier lenguaje de POO, y constituye la forma estándard de implementar el envío de mensajes en lenguajes de POO puros. Es interesante notar, por ejemplo, que una expresión del tipo `2 + 3` en SMALLTALK también sigue esta filosofía para la implementación del envío de mensajes, y de esta manera es interpretada como el envío de un mensaje común (`+`), al objeto receptor `2`, tomando como argumento el `3`, y el método que corresponde ejecutar en este caso ser resuelto en ejecución como cualquier método común. Otros lenguajes no toman un enfoque tan extremo, y el uso de la ligadura dinámica de mensajes deber ser explícitamente requerido para lograr este efecto. Esto se logra por ejemplo en C++, especificando una función miembro (método) como `virtual`.

Cuando se habla de variables polimórficas por su parte, se habla de aquellas variables del tipo de la clase padre, que son capaces de referenciar a objetos de cualquiera de las subclases. La clase padre define métodos que son luego sobreescritos por las subclases. Cuando se combina las variables polimórficas con la ligadura dinámica de mensajes, un mensaje enviado a una variable polimórfica es ligado dinámicamente al método de la clase correspondiente (la del objeto siendo referenciado en ese momento). Existen dos enfoques usuales respecto al tratamiento de las variables polimórficas:

- las variables polimórficas corresponden a una clase común (Object) y todos los mensajes se ligan dinámicamente (Java y SMALLTALK).
- especificar la clase de la variable polimórfica y los métodos con ligadura dinámica (C++)

Como ejemplo del segundo enfoque, veamos la forma en que C++ provee polimorfismo mediante la ligadura dinámica de mensajes y variables polimórficas, en el trozo de programa que se muestra a continuación

```
C *p;
p = new C1;
p->imprimir(); /* el de C1 */
p = new C2;
p->imprimir(); /* el de C2 */
C *ar[3];
ar[0] = new C1;
ar[1] = new C2;
ar[2] = new C3;
for(int i=0;i <3;i++)
 ar[i]->imprimir();
```



En este ejemplo, `p` (la variable polimórfica del tipo de la clase base `C`), puede referenciar a objetos de cualquiera de las sub-clases (clases derivadas) de `C`, es decir, a objetos de las clases `C1`, `C2` y `C3`. La misma capacidad tienen cada una de las componentes del arreglo `ar`. Por otra parte, al definir el método `imprimir()` como `virtual` en `C` y redefinirlo en cada una de las sub-clases, el método

`imprimir()` que se ejecute será aquel de la sub-clase de `C` que corresponda al objeto que esté siendo referenciado en el momento del envío del mensaje. Esto queda claro en el ejemplo, donde los dos mensajes `imprimir()` que se envían a `p` correspondern al método `imprimir()` de `C1` en el primer caso, y al de `C2` en el segundo. Una situación similar se repite en la iteración siguiente, donde se envía un mensaje `imprimir()` a cada uno de los componentes del arreglo `ar`. En este caso, se ejecutarn los métodos `imprimir()` de `C1`, `C2` y `C3`, en ese orden.

Esta forma de polimorfismo que la POO provee de una manera directa y elegante, facilita el trabajo con colecciones de datos heterogéneos, como se pueden encontrar por ejemplo, en las interfaces gráficas. En estos casos, la incorporación de un nuevo tipo de ventana a ser mostrada en la interface, sólo involucrará la definición de la clase asociada a la ventana como sub-clase de una clase de ventana genérica, debiendo luego redefinir aquellos métodos que deban ser manejados en forma dinámica de acuerdo a la nueva información que incorpora la sub-clase. Lograr un efecto similar utilizando un lenguaje imperativo similar es una tarea costosa no sólo desde el punto de vista de la programación, sino también del mantenimiento ante la necesidad de incorporar nuevos tipos heterogéneos. El análisis de la complejidad de estas tareas respecto al enfoque adoptado por la POO, queda como trabajo para el lector.

4. Aspectos del diseño de los lenguajes Orientados a objetos

A la hora de disear las características de los lenguajes de programación para proveer soporte para la herencia y la ligadura dinámica de mensajes, existen distintos aspectos que deben ser considerados, como por ejemplo los siguientes:

- Exclusividad de objetos
- Herencia simple y múltiple
- Asignación y liberación de objetos
- Ligadura estática y dinámica

El resto de estas notas de clase, estarán dedicadas a desarrollar cada uno de estos puntos.

4.1. Exclusividad de objetos

El primer aspecto a considerar es el grado en que un lenguaje de POO adhiere al paradigma, considerando si su modelo de objetos absorve a los restantes conceptos de tipo o no. En el caso más extremo (más POO "puro") desde el entero más pequeño hasta un sistema de software completo es considerado un objeto. Este enfoque, adoptado por lenguajes como SMALLTALK, provee gran elegancia y uniformidad al lenguaje y su uso. Sin embargo, su principal desventaja es que *todas* las operaciones (incluso las aritméticas más elementales como un simple `2 + 3`) se realizan mediante el proceso de pasaje de mensajes. Este enfoque, es significativamente más lento que el utilizado en lenguajes imperativos tradicionales, donde este tipo de operaciones se suele implementar directamente por instrucciones de máquina. En este modelo de POO puro, todos los tipos son clases y no existe diferencia entre clases pre-definidas y definidas por el usuario. De hecho, todas las clases son tratadas de la misma manera, y todas las computaciones se realizan a partir del pasaje de mensajes.

En algunos lenguajes, como JAVA, se consideró que un enfoque OO tan extremo como el planteado previamente, conducía a algunas ineficiencias inaceptables, al menos en lo referido al manejo de los tipos escalares primitivos (booleanos, caracteres y numéricos). Por este motivo, en el caso particular de JAVA no se usan objetos de forma exclusiva, sino que conviven los objetos con los tipos escalares primitivos que son manejados de la forma tradicional. Esta decisión, permite realizar las operaciones vinculadas a estos tipos de manera mucho más eficiente. Sin embargo, en aquellos casos en los que se deben combinar "objetos" con "no objetos", se pierde la uniformidad en el lenguaje y se adicionan algunas construcciones necesarias para "convertir" los no-objetos en objetos. Este enfoque es el utilizado en JAVA, donde las clases "wrapper" (fundas/envoltorio) permiten que valores escalares primitivos (no objetos) sean almacenados en estructuras que sólo admiten objetos. Por ejemplo, una estructura `Vector` en JAVA sólo puede contener objetos, y si deseáramos almacenar un entero primitivo en el mismo, el mismo debe ser previamente guardado en un objeto. Así, si quisiéramos guardar un `10` en el objeto `Vector` referenciado por la variable `miVector`, podríamos hacerlo mediante la siguiente sentencia:

```
miVector.addElement(new Integer(10));
```

donde `addElement` es un método de `Vector` que inserta un nuevo elemento e `Integer` es la clase "funda" para **int**.

Finalmente, un tercer enfoque respecto al sistema de tipos en un POO, es el que está basado en un lenguaje imperativo (reteniendo el modelo de tipos imperativo completo) y simplemente le agrega el modelo de objetos mediante algunas construcciones específicas a tal fin. Este enfoque, utilizado en C++ resulta en un lenguaje más grande cuya estructura de tipos es, en general, bastante más compleja de entender.

4.2. Herencia de implementación e interface

Una clase (TDA) ofrece la *interface* de sus facilidades a sus clientes pero oculta su *implementación*. De esta manera, podemos garantizar que todas las ventajas del *encapsulamiento* son efectivamente garantizadas. Sin embargo, un aspecto que surge naturalmente en estos casos, es cual es el criterio que debería ser adoptado con las *subclases*. Existen dos grandes enfoques a este interrogante: la herencia de *implementación* y la herencia de *interface*.

En la herencia de interface, la subclase es considerada como cualquier elemento externo a la clase que está siendo definida y, por tal motivo, sólo la interface del padre está visible a la clase derivada. En la herencia de implementación, la subclase es considerada como una componente privilegiada para el acceso de la representación interna del padre, y los detalles de implementación son por lo tanto visibles.

Las ventajas y desventajas de ambos enfoques son similares a los ya descriptos respecto a las ventajas y desventajas del encapsulamiento, con la diferencia que en este caso el encapsulamiento sólo es violado por las subclases. En este sentido, se puede observar que si las subclases utilizan herencia de implementación (tienen acceso a la parte "oculta" de sus padres), hace a las subclases dependientes de estos detalles, y cualquier modificación en la implementación de la clase padre, involucrará la recompilación/modificación de las subclases. Por otra parte, el acceso directo a las estructuras del padre en lugar de hacerlo exclusivamente a través de su interface permite usualmente un mayor grado de eficiencia, lo cual puede ser muy importante en aplicaciones críticas del mundo real.

Como se puede observar, no existe una respuesta general para aseverar que una u otra ventaja/desventaja sea más importante que la otra en todos los dominios. La mejor solución en estos casos,

suele ser que el diseñador del lenguaje provea ambas opciones de herencia y dejarle decidir al programador en cada caso cual opción es la mejor. En este sentido, es interesante observar que si bien SMALLTALK trabaja en general con herencia de implementación, otros lenguajes como JAVA o C++ dejan a criterio del programador el tipo de acceso a permitir en las subclases, dependiendo de cada caso particular.

4.3. Herencia simple y múltiple

Este aspecto tiene que ver con la posibilidad de que el lenguaje permita, además de herencia simple, la *herencia múltiple*. En la herencia múltiple una nueva clase puede heredar desde dos o más clases. Esto permite en muchos casos, organizar las clases de manera tal que reflejen mucho más adecuadamente las características del problema (por ejemplo, cuando efectivamente se hereda de más de una clase).

El uso de herencia múltiple, sin embargo, no suele ser gratuito ya que introduce algunos problemas de *complejidad* y *eficiencia*. Supongamos por ejemplo una situación como la mostrada en la Figura 3, donde la clase `C` hereda de las clases `A` y `B`, y ambas clases (`A` y `B`) definen un método heredable `dislpay`. Si `C` necesita referenciar ambas versiones de `dislpay`, el lenguaje deber proveer un mecanismo que resuelva esta ambigüedad, como lo hace por ejemplo C++, a partir del *operador de resolución de alcance* `::`. En este caso, la versión de `dislpay` a utilizar ser referenciada como `A::dislpay()` o `B::dislpay()`.

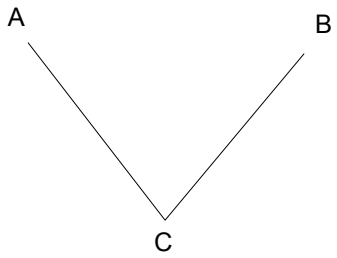


Figura 3: Un ejemplo de herencia múltiple.

Otro aspecto que aparece en este ejemplo con herencia múltiple, es el que se presentaría si tanto `A` como `B` heredan a su vez de una clase común `Z` dando origen a una situación de *herencia diamante* como la mostrada en la Figura 4.

Si en este caso, `Z` definiera una variable heredable `suma`, no es claro si `C` debería heredar ambas versiones de `suma`, o sólo una, y en este último caso cual de ellas.

Finalmente, la herencia múltiple suele involucrar algún deterioro en la eficiencia que en muchos casos es más ficticio que real. Así, por ejemplo, el soporte de herencia múltiple en C++ requiere el uso de una operación de suma adicional en cada llamada de un método ligado dinámicamente (aún cuando no se esté usando herencia múltiple) lo cual puede ser considerado como un costo adicional muy pequeño.

4.4. Asignación y liberación de objetos

Un aspecto de diseño importante en un lenguaje de POO es el *lugar* donde los objetos son almacenados. Si se comportan como un TDA estándard, entonces potencialmente cualquiera sería un lugar válido, pudiéndose guardar tanto en el stack de ejecución o bien explícitamente creado desde el heap

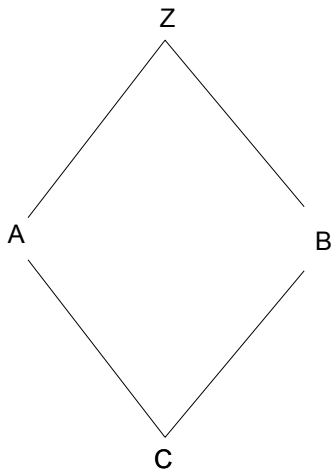


Figura 4: Un ejemplo de *herencia diamante*.

con un operador o función especial (por ejemplo **new** en C++). Otro enfoque, es que todos los objetos sean creados dinámicamente en el heap, lo cual ofrece como ventaja un método *uniforme* de creación y acceso mediante punteros o variables de referencia. Esto simplifica las asignaciones de objetos[2] y también la sintaxis de las referencias a los mismos[3].

El segundo aspecto se relaciona con el criterio adoptado para *liberar* los objetos asignados en el heap: liberación *implícita*, *explícita* o *ambas*. Los problemas que se presentan en este caso, son los mismos que se describieron al estudiarse los mecanismos de recuperación de memoria. De esta manera, si se trabaja con liberación implícita algún mecanismo de *contador de referencias* o *recolección de basura* serán necesarios (como en Java y SMALLTALK). Si en cambio se utiliza un enfoque de liberación de objetos explícita, se podrán presentar los conocidos problemas de *generación de basura* y *referencias desactivadas*.

4.5. Ligadura de mensajes a métodos

Ya vimos que la ligadura dinámica de mensajes a métodos constituye un aspecto clave en el soporte de polimorfismo que provee la POO. La pregunta que surge en este caso es si todas los envíos de mensajes deben ser resueltas en forma dinámica o no. Algunos lenguajes (como SMALLTALK), adoptan este enfoque extremo y ya vimos que hasta las más simples expresiones aritméticas son resueltas dinámicamente. En otros casos, y si consideramos que las ligaduras estáticas son más veloces, se le permite al usuario especificar si una ligadura particular es esttica o dinámica. Este es el enfoque adoptado por C++ que, a partir del calificador `virtual`, brinda al programador la posibilidad de especificar esta información a la hora de definir un método miembro.

En la siguiente tabla se presentan algunas de las principales características OO del lenguaje SMALLTALK. Queda como trabajo para el lector completar esta tabla y comparar con las características de los lenguajes Java y C++.

[2]En este caso sólo un valor puntero o referencia cambia.

[3]Estas refencias son implcitamente "desreferenciadas", pudiéndose observar esta simplificación si comparamos las referencias a objetos en Java con respecto a C++.

Lenguaje	Exclusividad de objetos	Herencia (simple o múltiple)	Herencia (implementación o interface)	Ubicación de objetos	Liberación de objetos	Ligadura de mensajes
SMALLTALK	SI	SIMPLE	IMPLEMENT.	HEAP	IMPLÍCITA	DINÁM.
Java						
C++						