Calderón Castro Alan. Enseñanza de la programación orientada a objetos mediante patrones de diseño. **Tecnología en Marcha**. Vol 14 N° 1.

Enseñanza de la programación orientada a objetos mediante patrones de diseño

Alan Calderón Castro[1]

Resumen

Se muestra cómo los patrones de diseño pueden ser útiles para generar situaciones de aprendizaje que permitan el tratamiento didáctico apropiado de conceptos y habilidades básicas de la programación por objetos. Se analiza un caso específico desarrollado con estudiantes de la carrera de Computación e Informática de la Universidad de Costa Rica.

Abstract

Design patterns can be used to create the appropriate conditions where basic concepts and skills of object programming can be effectively learned. An specific case is analyzed, which took place at the Computer Science department of University of Costa Rica.

Palabras clave: programación por objetos, enseñanza de la programación por objetos, patrones de diseño, diseño de clase.

Introducción

El aprendizaje de la programación orientada a objetos supone la comprensión de ciertos conceptos básicos cuyo tratamiento didáctico no es fácil. Al enseñar la programación orientada a objetos usando C++, algunos ejemplos de conceptos difíciles de abordar didácticamente son:

1. Polimorfismo (de datos[2] y de funciones).
2. Clase abstracta (en contraposición a clase concreta[3]).

1 Profesor de la Escuela de Computación e Informática de la Universidad de Costa Rica. Dirección electrónica: calderon@anubis.ecci.ucr.ac.cr

2 Por polimofismo de datos se entiende la posibilidad que tienen objetos contenedores (como una pila, una lista o una cola) de almacenar objetos de diferentes clases a la vez.

3 Una clase concreta en C++ sería aquella que no tiene funciones miembro virtuales puras; es decir, una clase que no es abstracta.

3. Funciones virtuales puras y funciones genéricas[4] (en contraposición a funciones *concretas*[5]).

Para enseñar estos conceptos, no basta con dar una definición clara ni mostrar ejemplos de su aplicación; es necesario exponer al aprendiz a un problema que lo involucre en su utilización.

Por otro lado, hay también ciertas habilidades difíciles de tratar como:

1. Especificación y documentación de clases.
2. Reutilización de bibliotecas de clases.
3. Identificación de opciones de diseño.
4. Análisis de las ventajas y desventajas; es decir, selección de las opciones de diseño dado un problema.

De acuerdo con la experiencia del autor en la enseñanza de la programación orientada a objetos, particularmente en C++, es difícil formular problemas que de manera natural permitan al docente tratar estos conceptos y habilidades. En este trabajo se expone la manera cómo la implantación de ciertos patrones de diseño[6] podría llevar de manera natural al tratamiento didáctico apropiado de dichos conceptos y habilidades. También se tocan tangencialmente algunos aspectos pedagógicos.

La experiencia que se describe y se analiza en este trabajo se desarrolló en la Escuela de Computación e Informática de la Universidad de Costa Rica, con estudiantes de segundo año de la carrera de bachillerato en Computación e Informática. Se trata de 23 estudiantes que, previo a este curso, habían aprobado un curso de programación en Pascal. Algunos tuvieron una introducción a la programación orientada a objetos en su primer curso. Ninguno conocía C++ antes de entrar en el curso en el que se desarrolló la experiencia analizada aquí.

Patrones

En Johnson (1992) se trata por primera vez el concepto de patrón en el contexto de la ingeniería de *software*[7]. Posteriormente, en Gamma (1995) se desarrolla ampliamente el concepto de patrón de diseño. Gamma *et al.* ofrecen un magnífico catálogo de patrones de diseño y de paso proponen lo que podría denominarse como un **meta-patrón** que sirve para especificar y documentar patrones de diseño.

Por otro lado, en Hay (1996) se vislumbra lo que podría considerarse *patrones de análisis*, específicamente de modelos de datos. Es posible que a nivel de análisis la variedad de categorías de patrones sea mayor, de manera que el interesante trabajo de Hay se centra solamente en una de estas categorías: patrones de modelos de datos. Por ejemplo, Rumbaugh (1996) introduce el concepto de arquitectura de aplicación, mientras que en el contexto del análisis de dominio Macala (1996) introduce el concepto de arquitectura de dominio. Todos estos pueden considerarse *patrones de análisis* en la medida en que constituyen marcos conceptuales que han mostrado ser

---

4 Una función genérica f ( ) de una clase A, se define como una función que se aplicará sobre instancias de clases derivadas de A y que invoca al menos una función virtual f ' ( ), cuyo enlace se realiza en forma dinámica.

5 En este trabajo se usa el término de función concreta para una función que no es ni virtual pura ni genérica.

6 Véase Gamma, 1995.

7 Johnson hace referencia a un trabajo previo que inspiró el suyo. Se trata del trabajo pionero de Christopher Alexander, en el dominio del diseño arquitectónico.

útiles en la comprensión, el modelado y la especificación general de la funcionalidad de familias enteras de sistemas de aplicación.

Tal como la denominan Gamma *et al.*, la *comunidad de desarrolladores de software interesados en los patrones* procura llevar el concepto de reutilización hasta el conocimiento mismo: de lo que se trata es de sistematizar conocimiento y hacerlo de fácil acceso para su *re*-utilización en la solución de problemas.

Desde la perspectiva del autor, la reutilización de conocimiento está implícita en la reutilización de funciones, clases, plantillas ("templates" en C++) y marcos de referencia[8] (o "frameworks"). La reutilización de una clase implica saber para qué sirve, cómo puede ser usada en el contexto de un problema específico, así como algunos detalles clave de su estructura. Al menos estos temas deben ocupar la documentación de una clase, una plantilla o un marco de referencia[9]. Esta documentación funciona como un contrato entre el programador usuario y el programador del componente reutilizable (ya sea función, clase, plantilla o marco de referencia). Por otro lado, la especificación (la actividad de diseño conceptual) de una clase (y lo mismo puede decirse de una plantilla y de un marco de referencia) conlleva la elaboración de un contrato que sintetiza la funcionalidad del componente y explicita el contexto en que puede ser usado. Es por esto que el autor ha desarrollado un esquema que sirve de guía a sus estudiantes, a la hora de especificar/documentar clases. Dado que las clases son componentes cuya utilización se repite una y otra vez en distintos problemas de programación, el esquema propuesto puede verse como un ***meta-patrón*** para especificar/documentar patrones de clase. Un contrato o patrón de clase puede implantarse de muy diversas maneras –al igual que un patrón de diseño– pero define claramente la funcionalidad de la clase, tanto para el programador de la herramienta (en la fase de implantación de la clase) como para el programador de aplicaciones que intente luego usar la clase.

El meta-patrón para especificar/documentar clases ha servido para centrar la atención del estudiante en los aspectos clave del diseño de una clase. Esta estrategia de trabajo implica que se va de lo abstracto a lo concreto, partiendo de la especificación hasta llegar a la implantación, sin que se descarten las iteraciones naturales. Aun así, algunos estudiantes demuestran tener preferencia clara por una estrategia de abajo hacia arriba: parten de un bosquejo del archivo de encabezados de la clase y la construyen en sucesivas iteraciones, luego la afinan, para culminar con la documentación del diseño. Esta opción siempre debe quedar abierta en una estrategia de enseñanza de la programación que esté centrada en el aprendiz.

Planteamiento del problema

El esquema del curso en que se desarrolló la experiencia que ocupa este trabajo se basó en cuatro tareas programadas. La primera tarea programada tuvo como meta que los estudiantes transfirieran, a C++, sus esquemas básicos de iteración, bifurcación, entrada/salida de datos, manejo de memoria dinámica y manejo de tipos aprendidos en Pascal. La segunda tarea programada tuvo como meta que

---

8 Véase Booch, 1996.

9 La especificación y documentación de funciones ha sido muy bien tratada por Liskov y Guttag en Liskov, 1986.

los estudiantes se familiarizaran con la biblioteca de clases contenedoras provista por el ambiente en que se trabajó[10] y de esta manera desarrollaran habilidades básicas de exploración de una biblioteca de clases. La tercera tarea programada, que es la que dio lugar a la experiencia que se analiza aquí, tuvo como meta que los estudiantes profundizaran en conceptos básicos de programación por objetos, así como que desarrollaran habilidades para la especificación/documentación de clases, la identificación de opciones de diseño y el análisis correspondiente de ventajas y desventajas[11]. De manera que al empezar a trabajar en el problema que a continuación se describe, los estudiantes ya conocían, aunque en forma superficial, conceptos como *clase, instancia, herencia simple, herencia múltiple* y *funciones virtuales*; sin embargo, no habían

El esquema de almacenamiento que supone un conjunto de asociaciones (llave, valor) direccionables a partir de la llave, es un esquema muy útil. Existe una gran cantidad de programas de aplicación que requieren este tipo de esquema de almacenamiento, desde programas procesadores de transacciones, orientados típicamente a las aplicaciones administrativas; hasta los compiladores. En el primer caso, el conjunto de asociaciones suele ser muy grande, y el acceso rápido es muy importante. Comparativamente, en el segundo caso, el conjunto de asociaciones es más bien pequeño, pero también es importantísimo un acceso rápido. A su vez, en el primer caso, el conjunto de asociaciones suele estar en continuo crecimiento, mientras que el tamaño de la tabla de símbolos de un compilador se mantiene relativamente estático. En otros casos, es muy importante poder recorrer en orden las asociaciones almacenadas.

En la biblioteca de BC++ 3.1 existe básicamente una clase que permite implantar directamente este concepto; sin embargo, esta implantación no es la más adecuada en todas las circunstancias. El uso de una "tabla-hash" es más apropiado en el caso de una tabla de símbolos de un compilador, pero es poco apropiado en el caso de un conjunto de asociaciones como, por ejemplo, (#Carné, Estudiante). No solo el tamaño de ambos conjuntos (de símbolos y de estudiantes) puede tener un comportamiento dinámico muy diferente, sino que además el conjunto de estudiantes frecuentemente será recorrido siguiendo algún orden, mientras que raras veces se requerirá un recorrido completo sobre una tabla de símbolos, y menos aún, con un orden específico.

Es por esto que surge la idea de crear un diccionario inteligente (**DiccIntel**). Esta clase de diccionario permitirá que:

1. se almacenen asociaciones con llaves y valores de tipo Object;
2. se construya de un diccionario apropiado a las necesidades del programa de aplicación (**Cliente**), para lo cual se basará en tres criterios: *tamaño aproximado* (pequeño, mediano o grande), *variabilidad* (poca, mediana, mucha) y *tipo de acceso* (solo al azar, solo en orden, ambos);
3. el diccionario se represente por medio de alguna de las siguientes clases: **Btree** o **Dictionary**, dependiendo de los parámetros de construcción;
4. la escogencia de alguna de estas dos estructuras de representación sea invisible para el programador usuario, de manera que el **Cliente** no dependa de esta escogencia;
5. eventualmente se agreguen nuevas posibles clases de implantación, sin que esto afecte a los programas **Cliente**.

Figura 1
Descripción del problema

10 Versión 3.1 de Borland C++ y su biblioteca de clases contenedoras, en su modalidad para DOS.

11 Cabe anotar que este esquema de tareas programadas lo he utilizado en tres semestres, en el curso de Programación II, con resultados que, a mi juicio, son muy satisfactorios.

nombre de la clase _____
[¿Es abstracta o concreta?]
¿Para qué sirve?:
¿Cómo se usa?:
1. Archivos asociados:
2. Requiere especialización
3. ¿Requiere que el usuario genere clases auxiliares?:
4. Ejemplos de su uso:

Estructura de la clase:
1. Ancestros.
2. Descendientes:
3. Atributos relevantes:
4. Constructores y Destructores
   4.1 Constructores Primarios.
   4.2 Constructores Secundarios.
   4.3 Destructores.
5. Funciones miembro públicas.
   5.1 Funciones/Operadores Mutadoras(es).
   5.2 Funciones/Operadores Observadoras(es).
6. Otras características importantes.
   6.1 Atributos globales de clase:
   6.2 Funciones globales de clase:

Otras relaciones importantes:
1. Esta clase es amiga de:
2. Las siguientes clases son amigas de esta clase:
3. Las siguientes funciones son amigas de esta clase:

Figura 2
Patrón para especificar/documentar clases en C++

trabajado con suficiente intensidad en la aplicación de estos conceptos.

En la Figura 1 aparece el planteamiento del problema tal como se expuso a los estudiantes. En la Figura 2 aparece el esquema que guía la especificación/documentación de las clases por desarrollar.

Hay varias metas implícitas en el problema propuesto:

1. Para el programador usuario, aunque la escogencia de Btree o Dictionary se le hace explícita a la hora de construir un objeto de tipo DiccIntel, el uso de uno u otro es en cambio transparente, pues la interfaz de DiccIntel debe ser homogénea, suficientemente general y, a la vez, ofrecer la totalidad (o la mayoría posible) de los servicios de ambas clases.

2. La interfaz de DiccIntel debe ser muy general como para asegurar que, si en el futuro se desea agregar otra clase de implementación como "SortedArray"[12], esto sea posible sin tener que cambiar la interfaz de DiccIntel y aprovechando la totalidad (o la mayoría posible) de los servicios de "SortedArray".

3. El mecanismo mediante el cual se garantiza la posibilidad de hacer uso de los servicios de cualquier clase de implantación (Btree, Dictionary o SortedArray u otra que resulte adecuada) debe ser totalmente transparente para el programador usuario de DiccIntel[13].

Hay tres tipos principales de componentes que entran en juego en este problema:

1. La propia clase DiccIntel en la que se concentra el mayor esfuerzo de diseño, por tanto se denomina ***clase objetivo***.

2. El conjunto de ***clases de implantación o adaptadas*** (Btree, Dictionary y otras posibles como SortedArray, List, Double

12 "SortedArray" es una clase de la biblioteca de clases contenedoras de BC++ 3.1 que permite guardar objetos en un arreglo siguiendo un cierto orden previamente definido.

13 Cabe hacer notar que cualquier mecanismo que se implante debe sacar provecho del enlace dinámico de funciones que C++ ofrece a través de las funciones virtuales.

List, u otras clases que podrían agregarse a la biblioteca en el futuro) que no es posible conocer en su totalidad. Cada una de estas clases debe ser analizada para poder diseñar una interfaz suficientemente general para DiccIntel.

3. Los mecanismos que operacionalizan la interfaz homogénea de DiccIntel, los cuales están llamados a ser clases, por el contexto en que se ha planteado el problema. Estas se denominan ***clases adaptadoras***.

La combinación de metas y componentes principales da como resultado un conjunto de opciones de diseño, y una serie de secuencias de decisiones de diseño, que hacen que este sea un problema complejo. Es por esto que la estrategia didáctica para sacar provecho a este problema requiere de un enfoque cooperativo en el que no solo los aprendices se apoyen mutuamente en grupos pequeños, sino que también deben tener acceso al soporte cognoscitivo del instructor. De acuerdo con la experiencia docente del autor, para que este tipo de problemas se conviertan en ricas situaciones de aprendizaje, es necesario que el instructor participe activamente en la búsqueda de soluciones y en el soporte a los aprendices. De lo que se trata es de llevar el conocimiento y las habilidades para la resolución de problemas de los aprendices a niveles superiores. Rogoff (1999:93) ofrece un marco de referencia que sustenta el papel del instructor en estas situaciones, y algunas de las características que señala de su comportamiento son:

1. Estimula el interés del aprendiz en la tarea tal como ha sido definida por el instructor.

2. Reduce apropiadamente las fases en que se debe desarrollar la tarea, a fin de que el aprendiz pueda comprender cómo algunos de los aspectos cruciales de la solución se ajustan a los requerimientos de la tarea.

3. Destaca aspectos críticos entre lo que el aprendiz va generando como solución y la solución ideal.

4. Controla la frustración y el riesgo de fracaso durante la solución del problema.

Es por eso que, ante el planteamiento de un problema como el discutido, conviene ofrecer a los aprendices un bosquejo de solución posible, sin descartar que en el camino surjan otras soluciones, o que los detalles para completar el esquema inicial puedan variar sustancialmente la solución tentativa. Todas estas posibilidades deben ser consideradas con cuidado por el instructor durante el proceso, centrándose siempre en el proceso de los aprendices, que podrán trabajar en grupos de dos o tres.

Análisis y comparación de soluciones

Primera alternativa: generalización de "Adapter" por composición.

El primer bosquejo de solución que se ofreció se basa en el patrón denominado "Adapter" de Gamma *et al.*, que se describe en la Figura 3.

El **Cliente** es un programa de aplicación que usa la clase **DiccIntel**. **DiccIntelBtree** y **DiccIntelDictionary** son las ***clases adaptadoras*** que sirven de intermediarias y transforman el protocolo genérico de DiccIntel a los servicios de cualquiera de las clases de implantación o ***clases adaptadas*** (**Btree** y **Dictionary**). La fila de adaptadores puede aumentar en el futuro si se desea agregar nuevas clases adaptadas, como, por ejemplo, SortedArray, List o DoubleList. Por esta razón, en este bosquejo inicial se sugirió que DiccIntel fuera una clase abstracta, pues su función principal es definir una interfaz genérica que restrinja la interfaz de futuros adaptadores. Este es uno de los usos convencionales que se da a las clases abstractas en C++.

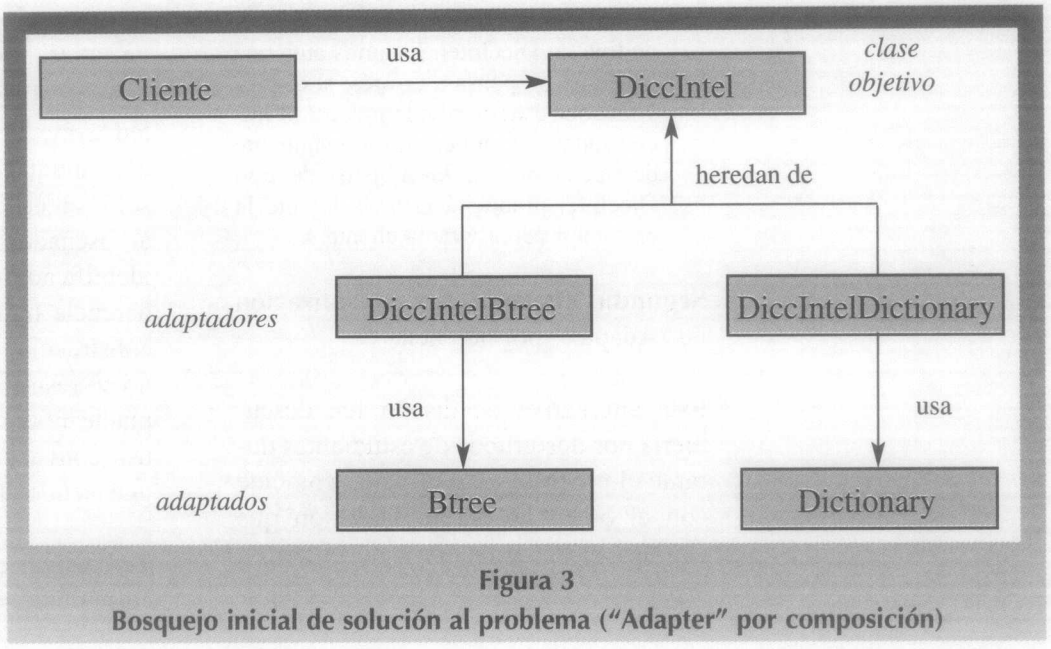

Figura 3
Bosquejo inicial de solución al problema ("Adapter" por composición)

Claramente se trata solo de un bosquejo de solución, pues quedan problemas por resolver no necesariamente triviales:

- ¿Qué clase de la biblioteca de contenedores ha de heredar DiccIntel?
- ¿Cuál debe ser el conjunto de operaciones de DiccIntel?
- ¿Cuáles de las funciones de DiccIntel pueden ser abstractas puras y cuáles deben ser genéricas?
- ¿Cómo se implantará la construcción de objetos de tipo DiccIntel?
- ¿La derivación de las clases adaptadoras a partir de DiccIntel debe ser privada o pública?
- ¿Qué mecanismos alternativos existen para la implantación de la relación de uso entre las clases adaptadoras y las clases adaptadas?, ¿cómo escoger uno? Existen básicamente dos mecanismos: el de composición y el de herencia. En esta primera aproximación de solución se está suponiendo que la relación de uso entre las clases adaptadoras y las adaptadas se representa por composición; es decir, en las adaptadoras aparece un puntero, referencia o variable del tipo adaptado (por ejemplo, en DiccIntelBtree aparece un puntero, referencia o variable privada de tipo Btree). El uso de herencia da lugar al otro esquema de solución que se discute brevemente en el apartado siguiente.
- ¿Cómo deberá el usuario implantar la relación de uso entre su programa de aplicación y la clase DiccIntel?

La mayor dificultad o desventaja a la hora de implantar este bosquejo de solución aparece en la construcción de objetos de tipo DiccIntel; es decir, en la implantación de la relación entre el **Cliente** y la **clase objetivo**. En el patrón "Adapter" de Gamma *et al.*, la clase objetivo no es abstracta y por lo tanto admite la construcción directa de objetos. El uso convencional de las clases abstractas en C++ para representar una interfaz genérica y extensible (la cual servirá de base restrictiva para futuras clases adaptadoras), motivó este planteamiento inicial en que la clase objetivo es abstracta y que tiene los siguientes problemas:

1. Por ser la clase objetivo una clase abstracta, el programa cliente no puede declarar objetos de tipo DiccIntel, más bien debe contentarse con punteros o referencias. Esto introduce un elemento distorsionante que se aleja de las soluciones estándar en C++.

2. Al no poder declararse un constructor para DiccIntel, el programa cliente debe invocar un inicializador que asegure un

estado inicial correcto para los objetos de tipo de DiccIntel. Estamos ante una situación propensa a errores: si el programador usuario olvida invocar al inicializador, la aplicación subsiguiente de operaciones a los objetos de tipo DiccIntel producirá errores durante la ejecución del programa cliente.

Segunda alternativa: generalización de "Adapter" por herencia.

Esta alternativa de diseño fue descubierta por dos grupos de estudiantes durante el proceso. Esta solución coincide con el otro tipo de implantación que Gamma *et al.* proponen para el patrón "Adapter". La Figura 4 describe este tipo de solución.

La diferencia principal con el esquema de la Figura 3 es que en este caso las clases adaptadoras se implementan por medio de una relación de herencia múltiple que otorga a las adaptadoras todo el protocolo de las adaptadas. Al tratarse de una versión generalizada del "Adapter" original de Gamma *et al.*, este esquema de solución no ofrece muchas ventajas sobre el anterior. En este problema lo que en realidad interesa es la interfaz de la clase objetivo DiccIntel; cuyo diseño está comprometido no solo con las clases adaptadas Btree y Dictionary, sino con otras candidatas a serlo en el futuro (SortedArray, DoubleList, etc.). Por esta razón, aunque se trata de un diseño viable, solo parece complicar las cosas al obligar al diseñador a lidiar con la repetición de identificadores, como consecuencia de la herencia múltiple. Los problemas en la construcción de objetos de tipo DiccIntel no se resuelven, pues de acuerdo con el planteamiento de los estudiantes y con el uso convencional en C++, DiccIntel sigue siendo abstracta.

Tercera alternativa: una variante con clase objetivo concreta.

Esta alternativa fue generada por uno de los dos mejores estudiantes del grupo. Su diseño pretende solventar el problema que afecta al esquema propuesto inicialmente. En el esquema de la Figura 5, DiccIntel ha dejado de ser una clase abstracta. Se ha factorizado la interfaz común a todos los adaptadores en una nueva clase que el estudiante denominó DiccIntelImpl (DiccIntel-Implementación) la cual sí es abstracta. El programa cliente usa directamente a DiccIntel y puede crear instancias

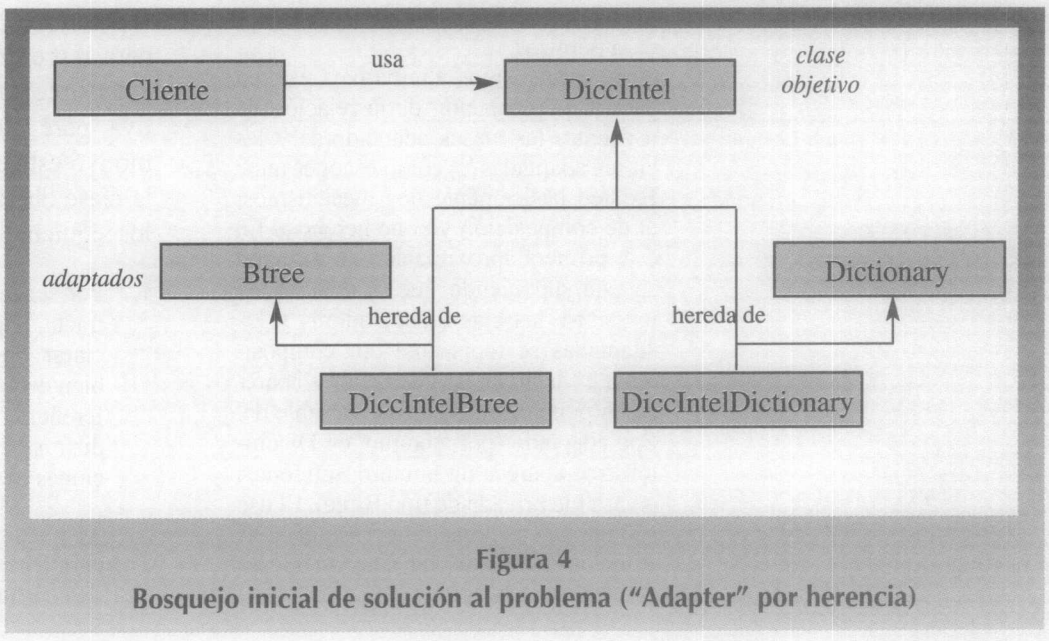

Figura 4
Bosquejo inicial de solución al problema ("Adapter" por herencia)

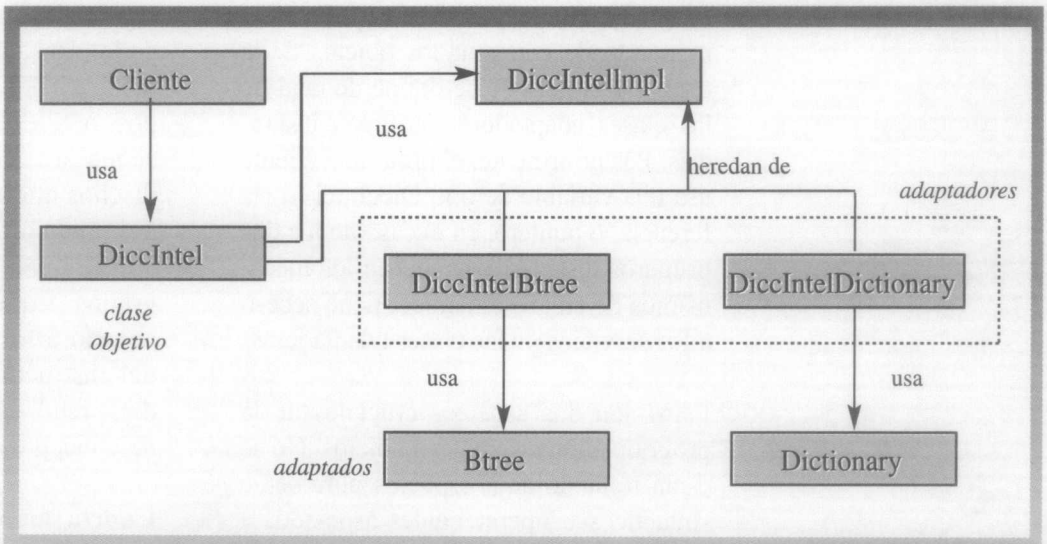

Figura 5
Variante A con clase objetivo concreta

usando los constructores estándar. Al haber en DiccIntel un atributo privado con el tipo *DiccIntelImpl**, que implementa la relación de uso entre DiccIntel y DiccIntelImpl, el constructor puede invocar la construcción de objetos de la clase adaptadora apropiada.

La diferencia fundamental de este diseño con el propuesto inicialmente consiste en que la clase objetivo se divide, representándose de dos formas alternas y complementarias:

1. Como clase abstracta **DiccIntelImpl** permite representar la interfaz genérica extensible.

2. Como clase concreta, **DiccIntel** permite la construcción estándar de objetos al programa cliente.

Hasta aquí todos los diseños se pueden extender agregando otras clases adaptadas, y coinciden en dos aspectos importantes:

1. Una clase abstracta siempre es usada para representar interfaces genéricas que seguramente serán extendidas en el futuro, en este caso, al agregar nuevas clases adaptadoras.

2. El polimorfismo de funciones, y más específicamente el mecanismo de enlace dinámico, es imprescindible en todos los diseños. Este mecanismo se usa para que el programa cliente construya un objeto de tipo DiccIntel pero que, de manera transparente, en realidad use instancias de las clases adaptadoras que le permiten accesar los servicios de las clases adaptadas Btree y Dictionary.

Stroustrup expone el primer principio de la siguiente forma:

> "El mecanismo de clase abstracta apoya la noción de un concepto general, como una figura, del cual solo se pueden usar en la práctica variantes más concretas, como círculo y cuadrado. Las clases abstractas también pueden servir para definir una interfaz de la que las clases derivadas representan implantaciones" (Stroustrup, 1993:590).

El uso del polimorfismo a través del mecanismo de enlace dinámico coincide también con el uso estándar. Al usar la clase DiccIntel, el programa cliente en realidad estará usando, de manera transparente, instancias de las clases adaptadoras, las cuales les dan acceso a los servicios provistos por las clases

adaptadas (Btree y Dictionary). Es en el momento de construir los objetos de la clase DiccIntel que se define de cuál de las clases adaptadoras se crean instancias. Posteriormente, el programa cliente usa una variable de tipo DiccIntel (o referencia, o puntero, en el caso de los dos primeros diseños) y, gracias a los mecanismos de enlace dinámico, tiene acceso a los servicios de las clases adaptadoras.

Estos son dos aspectos cruciales de la programación orientada a objetos. No se están tratando aquí aspectos puramente sintácticos u operacionales asociados a estos mecanismos; en realidad, se trata de que el aprendiz comprenda cómo deben ser usados estos mecanismos a la hora de resolver problemas. La definición operacional y la exposición de ejemplos ayuda a aproximar estos conceptos, pero su uso en contextos específicos permite analizar sus consecuencias en el diseño, sus ventajas y sus desventajas. Este es el conocimiento clave que debe ser objeto de un curso de programación orientada a objetos a este nivel.

Paralelamente, los aprendices deben especificar y documentar su diseño siguiendo el meta-patrón que se les ha dado. Descubren por sus propios medios, por los compañeros de otros grupos, o con el soporte del instructor, diseños alternativos. Comparan estos diseños alternativos y pueden decidirse por uno u otro. Esta actividad de diseño cooperativo, a dos niveles: el de los grupos pequeños y el de toda la clase, aunado a los distintos tipos de soporte del instructor, constituye un ambiente de aprendizaje apropiado para este tipo de conceptos y habilidades.

Cuarta alternativa: otra variante con clase objetivo concreta.

Otra variante a la propuesta inicial consiste en no usar una clase abstracta para DiccIntel, de manera que el programa cliente pueda construir objetos de tipo DiccIntel de manera estándar. Esto, sin embargo, obliga a que el constructor de DiccIntel invoque directamente los constructores de las clases adaptadoras que derivan de DiccIntel y por tanto aparecen después en el archivo de encabezados. Esta situación impide que se puedan invocar los constructores de las clases adaptadoras desde el constructor

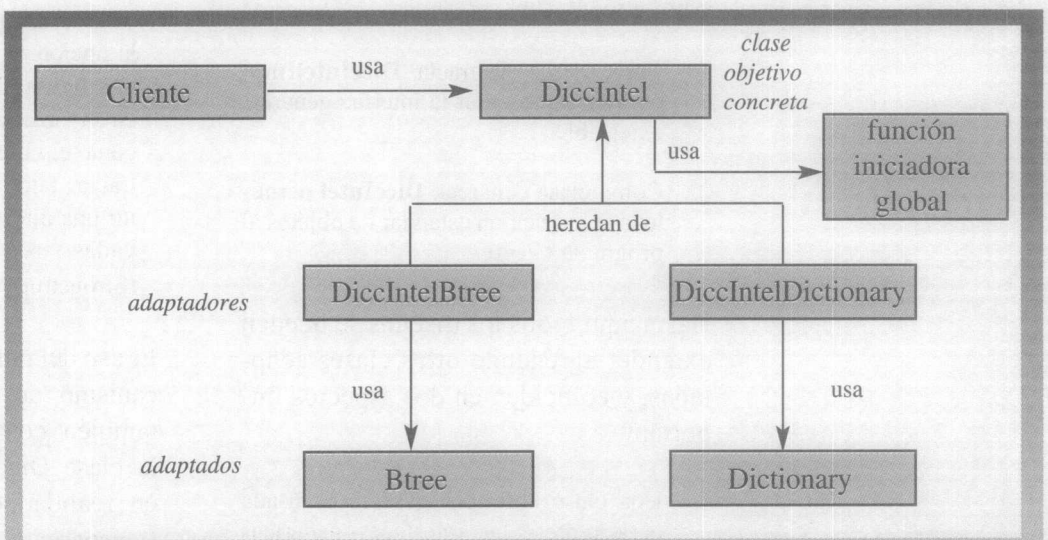

Figura 6
Variante B con clase objetivo concreta

de la clase DiccIntel, lo cual obliga a invocar una función que no sea miembro de DiccIntel ni de ninguna de las adaptadoras. Esta función puede declararse como amiga de DiccIntel en la sección privada, de manera que el usuario no tiene necesariamente que conocer de su existencia. De hecho, el usuario no requiere conocer esta función, pues de su invocación oportuna se hace cargo el constructor de DiccIntel.

Esta variante genera un tipo de clase que queda a medio camino entre una clase abstracta y una clase concreta[14]. DiccIntel provee una interfaz genérica que restringe la interfaz de las clases adaptadoras actuales y futuras, pero a la vez permite la creación de instancias, por lo cual se le denominó en el curso como clase genérica. Además, se hace uso de un truco que permite unificar la clase abstracta de la Figura 5 (DiccIntelImpl) y la clase concreta (DiccIntel) en una sola. Este truco consiste en poner como atributo privado dentro de DiccIntel un puntero de tipo DiccIntel *. Este puntero permite hacer referencia internamente, dentro de DiccIntel, a instancias de cualquiera de las clases adaptadoras; y a la vez permite poner en funcionamiento los mecanismos de enlace dinámico que permiten, al programa cliente, accesar los servicios de las clases adaptadoras. Todas sus funciones son genéricas pues el cuerpo de cada una de ellas simplemente invoca las funciones correspondientes dentro de las clases adaptadoras.

Análisis de variantes A y B con clase objetivo concreta

Estas variantes resuelven el problema del esquema inicial en cuanto a la creación de objetos de tipo DiccIntel por parte del programa cliente. Al ser DiccIntel concreta en ambos casos, el programa cliente puede crear instancias de la clase y basarse en el constructor de la clase que garantiza la producción de objetos con un estado inicial correcto.

Desde el punto de vista de la extensibilidad, ambas soluciones permiten agregar fácilmente nuevas clases adaptadoras. En la variante A deben derivarse de DiccIntelImpl y en la variante B deben derivarse de DiccIntel. El mecanismo herencia otorga a una clase derivada de DiccIntelImpl o de DiccIntel las mismas funciones como interfaz restrictiva básica. El código que hay que introducir es prácticamente idéntico en ambas soluciones. La única diferencia estriba en el mecanismo de construcción. En la variante A, debe modificarse directamente el constructor de DiccIntel, en la variante B debe modificarse la función global inicializadora.

De la simplicidad del diseño, puede decirse que es levemente más simple la variante B al prescindir de una clase (DiccIntelImpl en la variante A). Finalmente, en cuanto a eficiencia:

1. El tiempo de resolución de la invocación de funciones es idéntico en ambas soluciones puesto que en ambos casos, la invocación por parte del cliente de una función sobre un objeto de tipo DiccIntel, implica el enlace dinámico con la función de la clase adaptadora correspondiente.

2. Uso de memoria, la variante A es levemente superior pues, dentro de la variante B, todas las clases adaptadoras heredan un atributo que no utilizan (el puntero de tipo DiccIntel).

Conclusiones

Se han mostrado cuatro posibles soluciones al problema planteado. Se han

[14] Es lo que se llamó en el curso una clase genérica.

analizado sus ventajas y desventajas. Es evidente que las dos últimas propuestas (variantes A y B con clase objetivo concretas) son las mejores, aunque todas son viables. Más aún, es posible que existan otras variantes interesantes. La argumentación y comparación de opciones, así como su implantación, supone la aplicación de conceptos clave de la programación orientada a objetos, tales como los expuestos en la introducción.

Por otro lado, este problema da la oportunidad para la confrontación de ideas en el aula; de esta discusión sobre posibilidades de diseño, de la consideración cuidadosa de sus ventajas y desventajas, emerge un ambiente de aprendizaje propicio para el tipo de habilidades meta. La habilidad más importante que se procura desarrollar es la generación de opciones de diseño y su análisis comparativo cuidadoso. Pero en un curso introductorio como este, es imprescindible el apoyo cognoscitivo del instructor. Este tipo de habilidad ha sido considerado por algunos investigadores en Psicología cognoscitiva como crucial en la resolución general de problemas (Véase por ejemplo, Johnson-Laird (1983; 1991).

La intención que subyace en todos los diseños alternativos es "Transformar la interfaz de una clase en otra interfaz que el cliente espera..." (Gamma 1995:139). Esta poderosa idea no solo es útil para resolver el problema planteado, sino también otros problemas como los descritos por Gamma *et al.* en su catálogo de patrones de diseño. Las posibilidades de implantación en cada contexto específico son diversas; esta característica de generalidad y flexibilidad es lo que permite que los patrones de diseño sean buenos candidatos para la definición de problemas suficientemente complejos como para provocar, de manera natural, situaciones de aprendizaje similares a las descritas.

Finalmente, cabe señalar que el docente debe involucrarse en el proceso promoviendo la generación de opciones y su discusión cuidadosa. No cabe la posición defensiva según la cual el docente es el poseedor del conocimiento relevante. Este es un elemento crucial en la gestación del clima apropiado en el aula, debe ser comprendido a fondo, tanto por el docente que debe asumir un papel de facilitador, como por el aprendiz, que debe comprometerse con su proceso de aprendizaje y no esperar pasivamente el suministro del conocimiento.

Bibliografía

Booch, Grady. *Análisis y diseño orientado a objetos con aplicaciones.* 2da. edición, editorial Addison-Wesley/Díaz de Santos, Delaware, 1996.

Gamma, E., Helm, R., Johnson, R. y Vlissides, J. *Design Patterns: Elements of reusable object-oriented software.* Editorial Addison-Wesley, Massachusetts,1995.

Hay, David. *Data Model Patterns (Conventions of Thought).* Dorset House Publishing, New York, 1996.

Johnson, Ralph. *Documenting Frameworks using Patterns.* Vancouver, OOPSLA'92.

Johnson-Laird, Phillip N. *Mental Models: towards a cognitive science of language, inference and consciousness.* Harvard University Press, Cambridge, Massachusetts, 1983.

Johnson-Laird, Phillip N. *Deduction.* Lawrence Erlabaum Associates Publishers, Hillsdale, 1991.

Liskov, Barbara & Guttag, John. *Abstraction and Specification in Program Development.* editorial del MIT, Massachusetts, 1986.

Macala, Randall R; Stuckey, Lynn D. & David, Gross. *Managing Domain-Specific, Product-Line Development.* IEEE Software, mayo 1996.

Rogoff, Barbara. *Apprenticeship in Learning (Cognitive development in social context).* Oxford Univ. Press, New York, Oxford, 1990.

Rumbaugh, James; Blaha, Michael; Premerlani, William; Eddy Frederick & Lorensen, William. *Modelado y diseño orientados a objetos (Metodología OMT).* 1era. en español, Prentice Hall, 1996.

Stroustrup, Bjarne. *El lenguaje de programación C++.* 2da edición. Editorial Addisson-Wesley, Delaware, 1993.