UTN – FRTL – Paradigmas de Programación

Tipos Abstractos de Datos

1. Programación Estructurada

Fundamentalmente, la programación estructurada y la programación modular consisten en el uso de abstracción de procedimientos. La idea es meter un conjunto de instrucciones en un módulo, que tiene una interfaz claramente definida. Mediante la abstracción de procedimientos el programador separa claramente el cómo del qué: una subrutina contiene la lógica necesaria para ejecutar una tarea, y sus parámetros definen los objetos sobre los que trabaja.

Además, la programación estructurada requiere que el programador use las construcciones `if-else`, `while, for` y `switch`. Pero principalmente un programa estructurado es uno que no utiliza el ya bastante difamado `GOTO` para alterar la secuencia de ejecución del programa.

Los primeros lenguajes de programación (Fortran, Lisp, Basic y Cobol) no soportaban adecuadamente abstracción de procedimientos. Primero, no incorporaban las instrucciones de control mencionadas en el párrafo anterior. Segundo, aunque en ellos es posible definir argumentos para subrutinas, no es posible especificar sus tipos de datos, por lo que muchos errores de interfaz, que podrían ser detectados por el compilador, deben ser eliminados manualmente por el programador. Cuando estos lenguajes fueron definidos lo común era no especificar la interfaz entre módulos: hacerlo era visto por los programadores como un trabajo poco creativo y engorroso. Ellos se rehusaban a diseñar adecuadamente sus programas. La "documentación" era labor de funcionarios de poco sueldo, y se creía que lo más importante era escribir el código del programa.

Con el advenimiento de Algol, y luego de Pascal, poco a poco fue cambiando esta mala percepción sobre la especificación y documentación de programas. Con gran éxito se introdujo la verificación de tipos en los argumentos de cada rutina, como una herramienta para reducir el tiempo de desarrollo de un programa. Sin embargo, todavía no se reconocían las grandes ventajas de especificar procedimientos (funciones de librería).

En un procedimiento se oculta cómo se logra el objetivo. Un programador puede usar este procedimiento sin necesidad de conocer el código fuente de la rutina, con sólo disponer de una copia del código objeto producido al compilarla. Las bibliotecas de programas están formadas por procedimientos.

Un procedimiento realmente ayuda a partir un problema complejo en partes manejables, lo que garantiza no sólo una conclusión rápida del proceso de programación, sino que además permite reutilizar programas.

No basta con describir lo que el subprograma hace para que sea posible utilizarlo. También es necesario describir los datos con que trabaja cada procedimiento. Esto se logra mediante la abstracción de datos.

2. Tipos Abstractos de Datos

La programación que utiliza abstracción de datos se basa en el hecho de que en un programa se deben integrar y combinar los tipos básicos de datos, como números y caracteres, para formar estructuras de datos más complejas y así representar información dentro del computador. En general existe una fuerte relación entre todos los datos manipulados por un programa, por lo que es conveniente que esa relación esté claramente especificada y controlada, de forma que cada parte del programa "vea" sólo lo que necesita.

Esto último es muy importante para separar el programa en partes independientes, o módulos, evitando así que cambios en una parte produzcan errores en otras partes del programa. Por ejemplo, en un programa que usa varios arreglos y matrices para almacenar información, es frecuente que al aumentar el tamaño de una dimensión se olvide aumentar la de los demás arreglos, por lo que el mantenimiento del programa es más difícil. El objetivo perseguido al usar abstracción de datos es lograr aislar todas estas dependencias, de forma

que los cambios puedan ser hechos con un mínimo de esfuerzo y en una forma localizada. En nada ayuda tener que buscar por todo el programa en qué lugar debe hacerse cada cambio.

También es importante especificar mediante la abstracción de datos qué es cada estructura de datos. Una lista, por ejemplo, es una estructura de datos que tiene un comportamiento muy bien definido: pueden insertársele nuevos elementos, recorrérsela, encontrar el primer y último elemento, etc. Un programador que use el tipo de datos Lista no debe necesitar descubrir de nuevo ese concepto: simplemente debe poderlo importar de una biblioteca.

Al implementar la Lista como un Tipo Abstracto de Datos (**TAD**), el programador decide cuáles procedimientos se necesitan para manipular una lista, y define su interrelación. Un usuario del tipo de datos "lista" no necesitará entonces conocer cómo se interrelacionan (a nivel de implementación) los datos ni los procedimientos que manipulan listas pues le bastará usar las operaciones de la Lista para manejarla.

Al usar abstracción de datos el programador define cómo puede comportarse cada una de las variables de su programa. O sea que además de usar abstracción de procedimientos para construir el programa modularmente, deben especificarse las operaciones válidas sobre los datos. El objetivo es programar módulos que definan nuevos tipos de datos, y además que incluyan varios procedimientos (operaciones) que permitan manipularlos. Este dúo procedimiento-dato es un Tipo Abstracto de Datos. El programador-usuario del TAD podrá manipular las variables del tipo de datos únicamente por medio de sus procedimientos asociados. En un sólo módulo se "encapsula" el nuevo tipo de datos junto con sus procedimientos.

Al hablar de TADs se dice que se usa Abstracción de Datos porque al definir un TAD el programador especifica cuáles son las estructuras de datos que su programa utiliza. Se dice que se usa Encapsulamiento de Datos porque junto al tipo de datos también se definen las rutinas que permitirán utilizarlo. Por medio del Ocultamiento de Datos se evita que un programador-usuario del Tipo de Datos Abstracto cambie de forma impropia el valor de una variable.

En C sin TADs:

```
struct Persona
{
        char direccion[20];
        char telefono[6];
        char nombre [35];
        char fechanac[9];
 };

int Edad (char fechanac []) // función
{
  /* cálculo de la edad*/
 }

int main()
{
Persona p;
int edad;
edad = Edad(p.fechanac);  /* se invoca a la función
Edad*/
}
```

En C++ con TADs:

```
struct Persona
{
        char direccion[20];
        char telefono[6];
        char nombre [35];
        char fechanac[9];

        int Edad (); // método
 };

int Persona::Edad ( )
{
  /* cálculo de la edad*/
 }

int main()
{
        Persona p;
        p.Edad();  /* "p recibe el mensaje Edad*/
}
```

Figura N° 2

Un lenguaje de programación soporta el uso de Abstracción de Datos si tiene construcciones sintácticas que le permiten al programador usar TADs cómodamente. Por ejemplo, C++ soporta encapsulamiento, pues le permite al programador definir un tipo de datos junto a sus operaciones, como se muestra en la Figura 2. Sin embargo C no soporta encapsulamiento.

El ejemplo de la Figura 2 muestra que al implementar un TAD deben definirse dos cosas: primero, los campos que se necesitan para almacenar los valores del dato, y segundo los procedimientos que permiten utilizarlo.

El asociar un tipo de datos con sus procedimientos en un módulo se conoce como encapsulamiento. En C++, el lenguaje incluye construcciones sintácticas que permiten encapsular un tipo de datos, también llamado una clase, con sus procedimientos. En este caso, los procedimientos se conocen por los nombres "operación", "procedimiento miembro" o "método".

La diferencia entre una operación, en el contexto de los TADs, y un método, en el contexto de un lenguaje que soporta encapsulamiento, es puramente sintáctica. En el ejemplo C++ de la Figura 2 la forma sintácticamente correcta de invocar al método `Edad()` de `Persona` es "`p.Edad;`" mientras que si se usa un lenguaje que no soporta encapsulamiento (como C) debe usarse la conocida forma sintáctica "`Edad(p);`".

La única diferencia entre las dos formas de invocar a la operación (o método) `Edad()` es el lugar en que la variable de tipo `Persona` debe escribirse. La primera es la forma "Orientada a los Objetos" porque la variable "`p`" aparece de primero, mientras que la segunda es orientada a los procedimientos porque lo primero que aparece es el nombre de la rutina. Es este el simple cambio sintáctico que, con demasiada pompa, ha dado el nombre a la Programación Orientada a los Objetos (**POO**). En el ejemplo de la Figura 2 la forma sintáctica "`p.Edad;`" se interpreta como si el objeto "`p`" es un ente activo en el programa, mientras que la forma "`Edad(p);`" es interpretada como si el trabajo en un programa lo realizan los procedimientos (que calculan), y no los objetos.

Encapsulación	**Procedimiento**	**Tipo de Dato**	**Mensaje**
Encapsulación	Procedimiento	Tipo de Dato	Mensaje
Abstracción	Subrutina	Objeto	Parámetro
Especificación	Rutina	Clase	Argumento
Diseño	Método	ADT	
Ocultamiento	Operación	Tipo Abstracto de Datos	

Figura N° 3: Tabla de sinónimos

Los computólogos hemos tenido la costumbre de crear muchos nuevos términos para viejos conceptos. Los términos estructura de datos, tipo de datos, tipo abstracto de datos, objeto, clase y variable son muy similares. Todos sirven básicamente para lo mismo: definir variables en un programa, o sea, instancias del tipo de datos (aunque tiene más prestigio hablar de "objetos" o de TADs, que de simples tipos).
La Figura 3 es una lista de los términos usados para referirse a los mismos conceptos (aunque no todos son sinónimos exactos).

Entonces, una operación es una rutina que está asociada a un tipo de datos, y que permite manipularlo, examinarlo y cambiarlo. El término "método" es sinónimo de operación y, aunque es bastante confuso, es el utilizado en ambientes Smalltalk (aunque sí existe una buena razón para usar ese término, no la discutiremos).

Intrínsecamente ligado al concepto de encapsulamiento y abstracción está el concepto de ocultamiento de datos. Al ocultar datos, básicamente lo que se persigue es evitar que la representación usada para un TAD sea accesible a un programador-usuario. Por ejemplo, al implementar el TAD pila puede usarse una lista de nodos con punteros, o un arreglo. Si la representación del TAD es privada, entonces al usar una pila el programador no podría saber si la implementación que usa es una o la otra. El ocultamiento de datos es muy importante para lograr una mejor modularización, pues garantiza que el programador descuidado tenga

acceso controlado a los campos que conforman una instancia de un ADT, evitando así que destruya inadvertidamente alguna relación necesaria entre esos campos.

Tal vez pueda pensarse que los programadores, por antonomasia, no son "descuidados". Sin embargo, el ser "cuidadoso" requiere de un esfuerzo mental grande, a veces enorme, principalmente cuando se trabaja en grandes proyectos de programación. Si el compilador, la máquina computacional puede verificar las interfaces entre los módulos que comprenden un programa, ¿para qué cargar entonces al programador con esta innecesaria responsabilidad? Como se ha mencionado anteriormente, gran parte del éxito de Pascal como lenguaje radicaba en su capacidad de verificar tipos, que en realidad es una de esas actividades que podría hacerse manualmente. Al ser ésto responsabilidad de la máquina, se obtiene una significativa reducción del esfuerzo requerido para construir un programa, lo que decrece su costo y tiempo de programación.

El lector debe saber que la programación que usa abstracción de datos requiere de un mayor esfuerzo en el diseño de los programas, pues no basta simplemente con partir en subproblemas el problema a resolver, tal y como se hace al descomponer el diseño del programa de arriba hacia abajo (*top-down*). Cuando se programa de arriba hacia abajo el programador deja para después decisiones de diseño, pues cada nueva incógnita representa un nuevo procedimiento, en el que se oculta la respuesta correcta. Al final del diseño lo que queda es un grafo dirigido de dependencias de procedimientos. Desgraciadamente, en muchos casos la partición así definida no es la más adecuada para alcanzar los objetivos del programa.

Al usar abstracción de datos es necesario diseñar mejor el programa, con lo que se logra una mayor modularidad, que es una de las medidas de calidad en programación. El programa queda mejor modularizado que si hubiera sido hecho simplemente usando abstracción de procedimientos, pero el precio de este beneficio es que el programador puede tardar más en terminar su programa (más aún si no sabe bien lo que es un ADT). Un programa escrito sin usar abstracción de datos es más difícil de mantener, pues su estructura está oscurecida por la maraña de procedimientos que lo componen. Por eso al usar abstracción de datos se logra escribir programas más modulares.

La alta modularización alcanzada al usar TADs permite que el mantenimiento del programa sea mucho más simple. Sin embargo, en un ambiente de desarrollo de programas era difícil que esta nueva tecnología fuera aceptada, pues un administrador miope preferia que sus sistemas fueran programados muy rápidamente, aunque el posterior mantenimiento era más difícil. Con el correr del tiempo el mercado aprendió a apreciar la calidad de los programas, lo que obligó a los programadores a usar tecnologías nuevas que les permitan crear mejores productos a un precio más bajo.

Una de las ventajas más importantes de usar TADs es que un programador especializado puede dedicarse a depurar totalmente los procedimientos de un TAD específico. Esta persona se asegurará no sólo de que la implementación sea correcta, sino también de que sea muy eficiente. De esta manera un programador-usuario del TAD tiene acceso a rutinas de alta calidad y confiabilidad. Se evita así reprogramar los "trucos" que permiten implementar, eficientemente, una estructura de datos cada vez que se la necesite.

Precisamente el poder escoger entre varias alternativas de implementación es la razón principal para usar abstracción de datos: los detalles de implementación quedan totalmente ocultos dentro de las barreras del ADT, y es posible hacer modificaciones locales, generalmente para mejorar la eficiencia, que no tengan una repercusión global. Por eso el uso de abstracción de datos da dividendos cuando se le da mantenimiento a los programas. La abstracción de datos permite implementar programas que están mejor modularizados, y en la mayoría de los casos no implica desechar la programación estructurada y modular.