UNIVERSIDAD DE EL SALVADOR
FACULTAD DE INGENIERIA Y ARQUITECTURA
ESCUELA DE INGENIERIA DE SISTEMAS INFORMATICOS
PROGRAMACION II

1. TIPOS ABSTRACTOS DE DATOS

Se suele decir que la "ciencia informática" o "ciencias de la computación" es la ciencia de la abstracción. Pero ¿Qué es exactamente la abstracción? Se puede pensar en el tamaño de un objeto sin conocer cómo está constituido ese objeto.

La abstracción es un mecanismo fundamental para la comprensión de fenómenos o situaciones que implican gran cantidad de detalles. Es considerada, como uno de los conceptos más potentes en el proceso de resolución de problemas.

Se entiende por abstracción la capacidad de manejar un objeto (tema o idea) como un concepto general, sin considerar la enorme cantidad de detalles que pueden estar asociados con dicho objeto Por ejemplo, se puede saber conducir un automóvil sin conocer el tipo del modelo o cómo está fabricado. La abstracción se utiliza para suprimir detalles irrelevantes, mientras se enfatiza en los relevantes o significativos.

El proceso de abstracción presenta dos aspectos complementarios:

1. Enfocarse en los aspectos más relevantes del objeto.
2. Ignorar aspectos irrelevantes del mismo (la irrelevancia depende del nivel de abstracción, ya que si se pasa a niveles más concretos, es posible que ciertos aspectos pasen a ser relevantes).

Se puede decir que la abstracción permite estudiar los fenómenos complejos siguiendo un método jerárquico, es decir, por sucesivos niveles de detalle. Generalmente, se sigue un sentido descendente, desde los niveles más generales a los niveles más concretos.

El beneficio principal de la abstracción es que facilita al programador pensar acerca del problema a resolver. Uno de los principios importantes del diseño de software es el de la abstracción y ocultación de la información.

Durante la década de los años setenta, se introdujo el concepto de tipo de abstracto de datos. La idea básica que subyace dentro de un tipo abstracto de datos es la separación del uso del tipo de datos de su implementación (las operaciones que actúan sobre los valores de esos datos). Aparece el concepto de especificación del tipo de dato que puede expresarse de modo independiente de su implementación.

Con los tipos abstractos de datos (TAD) se establece un nivel intermedio, donde se quiere modelar lo esencial de la realidad sin comprometerse con detalles de implementación. Es posible examinar los conceptos de *modularidad, abstracción de datos y objetos.* La modularidad es la posibilidad de dividir una aplicación en piezas más pequeñas llamadas módulos. *Abstracción de datos* es la técnica de inventar nuevos tipos de datos que sean más adecuados a una aplicación y, por consiguiente, facilitar la escritura del programa. La técnica de abstracción de datos es una técnica potente de propósito general que cuando se utiliza adecuadamente, puede producir programas más cortos, más legibles y flexibles. Los objetos combinan en una sola unidad *datos y funciones* que operan sobre esos datos.

El uso del paradigma de programación orientada a objetos para desarrollar software, además de su uso en la implementación de software, requiere que los desarrolladores traten con objetos en todas las fases del ciclo de vida del software. El objeto se representa en un primer nivel por los tipos abstractos de datos, TAD (Abstract Data Type, ADT).
Un tipo abstracto de dato define un tipo de dato aisladamente en términos de un tipo y un conjunto de operaciones sobre ese tipo. Cada operación se define por sus entradas y sus salidas. La definición de un TAD no especifica cómo se implementa el tipo de dato. Estos detalles están ocultos en el uso del TAD. El proceso de ocultación de los detalles de la implementación se conoce como **encapsulación**.

Con un TAD, los usuarios no están preocupados en *cómo* se hace la tarea sino *con qué* o *cómo* se puede hacer esta tarea. En otras palabras, el TAD consta de un conjunto de definiciones que permiten a los programadores utilizar las funciones mientras ocultan las implementaciones. Esta generalización de las operaciones con implementaciones no especificadas se conoce como **abstracción**. Se abstraen la esencia del proceso y se dejan ocultos los detalles de la implementación.

Una estructura de datos es la implementación física de un TAD. Cada operación asociada con el TAD se implementa por una o más subrutinas. El término “estructura de datos” se refiere, frecuentemente, a los datos almacenados en la memoria principal de la computadora, mientras que el término *estructura archivo* se refiere, normalmente, a la organización de los datos en un almacenamiento periférico, tal como una unidad de cinta, disco, CD, DVD o similar.

Definamos formalmente un TAD, un **tipo abstracto de datos** es una declaración de datos empaquetada junto con las operaciones que son significativas para el tipo de dato. En otras palabras, encapsulamos los datos y las operaciones sobre esos datos y ocultamos estos detalles a la vista del usuario.

Es posible observar que las operaciones de un TAD son de diferentes clases: algunas de ellas nos deben permitir crear objetos nuevos, otras determinar su estado, unas construir otros objetos a partir de algunos ya existentes, etc.

La implementación tradicional frente a los TAD

Según la clásica ecuación de Wirth:

Programa = Datos + Algoritmo

El enfoque tradicional se ciñe bastante bien a esta concepción. Con los TAD se identifican ciertas operaciones o partes del algoritmo que manipulan los datos. En la ecuación de Wirth la parte *Algoritmo* la podemos expresar como:

Algoritmo = Algoritmo de datos + Algoritmo de control

Se entiende como *Algoritmo de datos* a la parte del algoritmo encargada de manipular las estructuras de datos del problema, y *Algoritmo de control* a la parte restante (la que representa en sí el método de solución del problema, independiente hasta cierto punto de las estructuras de datos seleccionadas).

Entonces podemos reescribir:

Programa = Datos + Algoritmo de Datos + Algoritmos de Control

Colocando *Datos + Algoritmo de Datos* como *Implementación de TAD* se establece la siguiente ecuación:
Programa = Implementación del TAD + Algoritmo de Control

Describe el enfoque de desarrollo con Tipos Abstractos de Datos.
Ventajas de los tipos abstractos de datos.

Los tipos abstractos de datos proporcionan numerosos beneficios al programador, que se pueden resumir en los siguientes:

1. Permite una mejor conceptualización y modelización del mundo real. Mejora la representación y la comprensibilidad. Clarifica los objetos basados en estructuras y comportamientos comunes.
2. Mejora la robustez del sistema. Los tipos abstractos de datos permiten la comprobación de tipos para evitar errores de tipo en tiempo de ejecución.
3. Mejora el rendimiento (prestaciones). Para sistemas tipificados, el conocimiento de los objetos permite la optimización del tiempo de compilación.
4. Separa la implementación de la especificación. Permite la modificación y mejora de la implementación sin afectar a la interfaz pública del tipo abstracto de dato.
5. Permite la extensibilidad del sistema. Los componentes de software reutilizables son más fáciles de crear y mantener.
6. Recoge mejor la semántica del tipo. Los tipos abstractos de datos agrupan o localizan las operaciones y la representación de atributos.

Clasificación de las operaciones

Las operaciones las podemos clasificar de esta manera:

Operaciones para crear objetos

Iniciales: Se utilizan para crear objetos del TAD, en cuya creación no se requiere ningún objeto abstracto del mismo tipo.

Constructores: Utilizadas para crear objetos del TAD cuya creación depende de objetos del mismo tipo.

Operaciones para transformar objetos de TAD

Simplificadoras: Son operaciones cuyo codominio es el TAD que se define, pero que dan como resultado objetos que pueden ser descritos utilizando únicamente operaciones iniciales y constructoras.

Operaciones para analizar los elementos del TAD

Analizadoras: Son operaciones cuyo codominio no es el TAD que se define, sino otro ya conocido. El propósito de este tipo de operaciones es obtener información concerniente a cualquiera de los objetos de tipo abstracto.

Las operaciones simplificadoras y analizadoras inducen una noción de equivalencia entre los elementos de TAD que se define; si dos objetos abstractos son indistinguibles mediante operaciones simplificadoras y analizadoras, el observador debería concluir que se trata del mismo objeto.

Especificaciones de los TAD.

La especificación de un TAD consta de dos partes, la descripción matemática del conjunto de datos, y las operaciones definidas en ciertos elementos de ese conjunto de datos. El objetivo de la especificación es describir el comportamiento del TAD. La especificación del TAD puede tener un enfoque *informal*, en el que se describen los datos y las operaciones relacionadas en *lenguaje natural*. Un segundo enfoque semiformal, donde se combinan elementos del lenguaje natural y axiomas ó lenguaje de programación. Y un tercer enfoque, mas riguroso denominado especificación formal, a través del cual se suministra un conjunto de axiomas que describen las operaciones en su aspecto sintáctico y semántico.

La especificación informal consiste en dos partes:

- Detallar en los datos del tipo, los valores que pueden tomar.
- Describir las operaciones, relacionándolas con los datos.

El formato que emplea la especificación es el siguiente: primero se establece el nombre del TAD y los datos y que lo forman: *TAD nombre del tipo (valores y su descripción)*; Posteriormente se especifica cada una de las operaciones con sus argumentos, y una descripción funcional en lenguaje natural. *Operación (argumentos)*. Descripción funcional.

Ejemplo:

Realizar una especificación informal del TAD Conjunto con las operaciones: Crear*Conjunto, Esvacio, Añadir un elemento al conjunto, Pertenece un elemento al conjunto, Retirar un elemento del conjunto, Unión de dos conjuntos, Intersección de dos conjuntos e Inclusión de conjuntos*

Análisis del problema.

Definición del tipo de datos

TAD Conjunto (Especificación de elementos sin duplicidades pueden estar en cualquier orden, se usa para representar los conjuntos matemáticos de números enteros positivos con sus operaciones)

Descripción de las operaciones:

*CrearConjunto: . C*rea un conjunto sin elementos.

Añadir(Conjunto, elemento): Comprueba si el elemento forma parte del conjunto, en caso negativo es añadido. La función modifica al conjunto.

Retirar(Conjunto, elemento): En el caso de que el elemento pertenezca al conjunto es eliminado de este. La función modifica al conjunto.

Pertenece(Conjunto, elemento): Verifica si el elemento forma parte del conjunto, en cuyo caso devuelve *cierto*.

Esvacio(Conjunto): Verifica si el conjunto no tiene elementos, en cuyo caso devuelve *cierto*.

Cardinal(Conjunto): Devuelve el número de elementos del conjunto.

Union(Conjunto, Conjunto): Realiza la operación matemática de la unión de dos conjuntos. La operación devuelve un conjunto con los elementos comunes y no comunes a los dos argumentos.

Intersección(Conjunto, Conjunto): Realiza la inclusión matemática de la intersección de dos conjuntos. La operación devuelve un conjunto con los elementos comunes a los dos argumentos.

Inclusión(Conjunto, Conjunto): Verifica si el primer conjunto está incluido en el conjunto especificado en el segundo argumento, en cuyo caso devuelve *cierto.*

Resultado del análisis del problema:

Nombre del TAD:	***Descripción:***
Conjunto	Especificación de elementos sin duplicidades pueden estar en cualquier orden, se usa para representar los conjuntos matemáticos de números enteros positivos con sus operaciones.
Operaciones:	
CrearConjunto()	
Añadir(Conjunto, elemento)	
Retirar(Conjunto, elemento)	
Pertenece(Conjunto, elemento)	
Esvacio(Conjunto)	
Cardinal(Conjunto	
Union(Conjunto, Conjunto)	
Intersección(Conjunto, Conjunto)	
Inclusión(Conjunto, Conjunto)	

Diseño y/o especificación de las operaciones.

Existen varios métodos para especificar un TAD. El que se va a usar es semiformal y a grandes rasgos emplea la notación de C, pero la amplía cuando es necesario. Para ilustrar el concepto de un TAD y el método de especificación que se va a usar, considere el TAD Conjunto. (la notación empleada se describe más adelante).

/*Definición de valor */
Observe que se esta definiendo un nuevo tipo de datos formado por un conjunto de n enteros. Sin embargo, es posible escribir el número exacto de elementos sustituyendo la n por su respectivo valor.
Abstract typedef < integer, n > Conjunto
Condition Conjunto Є Numeros enteros positivos

/***Definición de operador***/
Es importante que se defina el objetivo de la operación a especificar. El diseñador debe de ampliar la especificación incluyendo información adicional que facilite posteriormente al programador la implementación de la operación.

La siguiente operación tiene como objetivo crear un conjunto vació para almacenar números enteros positivos.
abstract Conjunto CrearConjunto()
Conjunto MiConjunto = Null;
PostCondition CrearConjunto = MiConjunto;

Observe que la operación anterior permitirá crear conjuntos vacíos de números enteros cada vez que sea invocada.

/* El objetivo de la siguiente operación es identificar si un elemento dado pertenece al conjunto. El resultado de la operación es un valor lógico. Un valor de retorno 1 indica que el elemento ya existe en el conjunto. Un valor de 0 indica que el elemento no se encuentra en el conjunto.
Abstract int Pertenece (Conjunto MiConjunto, int Elemento)
Int i, Resultado=0;
Precondition MiConjunto != Ø y Elemento Є Números enteros positivos
PostCondition

Pertenece =
```
while MiConjunto[i] != Null
        If MiConjunto[i] == Elemento {
                Resultado=1; exit; // Rompe el ciclo}
return Resultado;
```

La operación añadir tiene como objetivo agregar nuevos elementos al conjunto. Para la especificación de esta operación, se debe tomar en cuenta que no es posible agregar elementos duplicados al conjunto.
abstract Conjunto Añadir(Conjunto MiConjunto, int Elemento)
int i=0;
PreCondition Elemento Є Números enteros positivos
PostCondition

Añadir =
```
If !Pertenece(MiConjunto, Elemento){
        /* identifica la primera posición vacía */
        while ( MiConjunto[i] != Ø)
                i=i+1;
        MiConjunto[i] = Elemento;
        }
return MiConjunto;
```

La operación Cardinal permite identificar el número de elementos que se encuentran dentro de un conjunto. Por tanto, el valor de retorno de la operación es un número entero.
abstract int Cardinal(Conjunto MiConjunto)
int NumElementos, i=0;
PreCondition MiConjunto != Ø
PostCondition

Cardinal =
```
while MiConjunto[i] != Null
        i=i+1;
NumElementos = i;
return NumElementos;
```

Se deja al estudiante la especificación de las operaciones restantes (retirar, esvacio, union, etc.) que se encuentran en el resultado del análisis del problema.

Un ejemplo de especificación adicional es el TAD RATIONAL, que corresponde al concepto matemático de un número racional. Un número racional es aquel que puede expresarse como el cociente de dos enteros. Las operaciones sobre números racionales que se definen son la creación de un número racional a partir de dos enteros, la adición, la multiplicación y una prueba de igualdad.

La siguiente es una especificación inicial de este TAD:

/* definición de valor */

abstract typedef <integer, integer> RATIONAL;
condition RATIONAL[1] != 0;

/* definición de operador */
La siguiente operación permite crear un número racional cada vez que es invocada.
abstract RATIONAL makerational (int Num, int Denom)
Precondition Denom != 0;
Postcondition makerational[0] = Num;
makerational[1] = Denom;

/*operación que permite sumar dos numeros racionales */
abstract RATIONAL add (RATIONAL a, RATIONAL b) /* written a + b */
Postcondition add[1] = a[1] * b[1] ;
add[0] = a[0] * b[1] + b[0] * a[1] ;

/*operación que permite multiplicar dos numeros racionales */
abstract RATIONAL mult (RATIONAL a, RATIONAL b) /* written a * b */
Postcondition mult[0] = a[0] * b[0] ;
mult[1] = a[1] * b[1] ;

Observe que la operación equal devolverá un valor de 1 en caso de que ambos racionales sean iguales y cero para racionales diferentes.

abstract int equal (RATIONAL a, RATIONAL b) /* written a == b */
Postcondition equal = (a[0] * b[1] == b[0] * a[1]);

Un TAD consta de dos partes: una definición de valor y una definición de operador. La definición de valor establece el conjunto de valores para el TAD y consta de dos partes: una cláusula de definición y una cláusula de condición. Por ejemplo, la definición de valor para el TAD RATIONAL establece que un valor RATIONAL tiene dos enteros, el segundo de los cuales no es igual a cero. Por supuesto, los dos enteros que forman un número racional son el numerador y el denominador.

Se emplea una notación de arreglo (paréntesis cuadrados) para indicar las partes de un tipo abstracto. Las palabras reservadas **abstract typedef** introducen una definición de valor, y la palabra reservada **condition** se usa para especificar las condiciones del tipo recientemente definido.

Inmediatamente después de la definición de valor viene la definición de operador. Cada operador esta definido como una función abstracta con tres partes: un encabezado, las condiciones previas opcionales y las condiciones posteriores.

Por ejemplo, la definición de operador del TAD RATIONAL, incluye las operaciones de creación (makerational), adición (add) y de multiplicación (mult), al igual que una prueba de igualdad (equal).

En la especificación de la multiplicación, se ve que contiene un encabezado y condiciones posteriores, pero no condiciones previas.

El encabezado de esta definición está en las dos primeras líneas, igual que en un encabezado de función de C. La palabra reservada abstract indica que no es una función de C, el comentario que empieza con la nueva palabra reservada written indica una forma alternativa de escribir la función.

La condición posterior (postcondition) especifica qué hace la operación. En una condición posterior, el nombre de la función (en este caso, mult) se usa para denotar el resultado de la operación. Por tanto, mult[0] representa el numerador del resultado y mult[1] el denominador. En este ejemplo, la condición posterior especifica que el numerador del resultado de una multiplicación racional sea

igual al producto entero de los numeradores de las dos entradas y que el denominador sea igual al producto entero de los dos denominadores.

La especificación para la igualdad (equal) es más importante y más compleja en lo conceptual. En general, cualquiera de los dos valores en un TAD son "iguales" si y sólo si los valores de sus componentes son iguales. Por lo regular, se supone que existe una operación de igualdad (y una de desigualdad) y que está definida de esa forma, por lo que no se requiere ninguna definición explícita de operador de igualdad.

Sin embargo, para algunos tipos de datos, dos valores con componentes desiguales pueden considerarse iguales. En realidad, tal es el caso con los números racionales: por ejemplo, los números racionales ½, 2/4, 3/6 y 18/36 son todos iguales a pesar de la desigualdad de sus componentes. Dos números racionales se consideran iguales si sus componentes son iguales cuando los números se reducen a sus términos mínimos. Una manera de probar la igualdad racional es reducir los dos números a los términos mínimos y después probar la igualdad de los numeradores y denominadores. Otra forma de probar la igualdad, es verificar que los productos cruzados (esto es, el numerador de uno por el denominador de otro) son iguales.

Comprender que dos racionales pueden ser iguales incluso si son desiguales sus componentes, obliga a escribir de nuevo las condiciones posteriores para *makerational, add y mult.* Esto es, si

$$\frac{m0}{m1} = \frac{a0}{a1} * \frac{b0}{b1}$$

No es necesario que m0 sea igual a a0 * b0 ni que m1 sea igual a a1 * b1, sólo que m0 * a1 * b1 sea igual a m1 * a0 * b0.

Las secuencias como definiciones de valor.

Al desarrollar las especificaciones para diferentes tipos de datos, con frecuencia se usa notación de teoría de conjuntos para especificar el valor de un TAD. En particular, es útil usar la notación de secuencias matemáticas que se presentan a continuación.

Una secuencia es simplemente un conjunto de elementos ordenados. Una secuencia S se escribe en ocasiones como la enumeración de sus elementos, así:

$$S = < s_0, s_1, ..., s_{n-1} >$$

Si S contiene elementos, se dice que S tiene una longitud de n. Se supone la existencia de una función de longitud len, por lo que len(S) es la longitud de la secuencia S. También se suponen las funciones *first*(S), la cual regresa el valor del primer elemento de S, y *last*(S), que retorna el valor del último elemento de S. Hay una secuencia especial de longitud 0, llamada *nilseq*, que no contiene elementos. *first*(nilseq) y *last*(nilseq), no están definidos.

Se pretende definir un TAD stp1 cuyos valores son secuencias de elementos. Si las secuencias pueden ser de una longitud arbitraria y constar de elementos que sean todos del mismo tipo, tp, entonces stp1 puede definirse mediante

abstract typedef <<tp>> stp1;

O bien, tal vez si se pretendiera definir un TAD stp2, cuyos valores sean secuencias de longitud fija, con elementos de tipos específicos. En tal caso, se especificaría la definición.

abstract typedef<tp0, tp1, tp2, ..., tpn> stp2;

Por supuesto, si fuera necesario especificar una secuencia de longitud fija donde todos los elementos fueran del mismo tipo, entonces se podría escribir.

abstract typedef<<tp, n>> stp3;

Implementación de los TAD.

Cuando ya se tiene bien diseñado un TAD, el siguiente paso es decidir una implementación para el mismo. Esto implica escoger unas estructuras de datos para representar cada uno de los objetos abstractos y escribir una rutina (Procedimiento, función o método) es un lenguaje de programación, que simule el funcionamiento de cada una de las operaciones de acuerdo con su especificación. La selección de las estructuras de datos determina, en muchos casos, la complejidad del algoritmo que implementa una operación.

Los lenguajes convencionales, tales como C, permiten la definición de nuevos tipos y la declaración de funciones para realizar operaciones sobre objetos de los tipos. Sin embargo, tales lenguajes no permiten que los datos y las operaciones asociadas sean declarados juntos como una unidad y con un solo nombre. En los lenguajes en el que los módulos (TAD) se pueden implementar como una unidad, éstos reciben nombres distintos:

Turbo Pascal	unidad, objeto
Modula-2	módulo
Ada	paquete
C++	clase
Java	clase

En estos lenguajes se definen la *especificación* del TAD, que declara las operaciones y los datos ocultos al exterior, y la *implementación*, que muestra el código fuente de las operaciones y que permanece oculto al exterior del módulo.

En C no existe como tal una construcción del lenguaje para especificar un TAD. Sin embargo se puede agrupar la interfaz y la representación de los datos en un archivo de inclusión: *archivo.h*. La implementación de la interfaz, de la funciones se realiza en el correspondiente *archivo.c*. Los detalles de la codificación de las funciones quedan *ocultos* en el *archivo.c*

Ejercicios de programación:

1. Diseñe un TAD para los números imaginarios. Defina las operaciones crear, suma, resta, igual, menor que.