Introducción a los Tipos Abstractos de Datos

- Introducción: Concepto de abstracción
- Abstracción funcional y abstracción de datos
- Construcción de tipos abstractos de datos
    - Especificación de un TAD
    - Implementación de un TAD

Introducción a los Tipos Abstractos de Datos (TAD) de FUOC está sujeta a una licencia de Reconocimiento-CompartirIgual 3.0 España de Creative Commons

Introducción: Concepto de Abstracción

La abstracción es un **proceso cognitivo** humano esencial para la comprensión de fenómenos o situaciones complejas que consiste en la categorización de elementos en grupos o clases de características similares. Cada una de las clases o grupo representa una abstracción, en virtud de la cual se destacan o ignoran determinadas características del grupo. La abstracción, por lo tanto, presenta dos caras complementarias:

- En primer lugar, considera o resalta algunos de los aspectos de los elementos en estudio, en concreto, los aspectos relevantes para el problema o situación que se desea resolver.
- En segundo lugar, ignora el resto de los detalles –no relevantes para la tarea en curso– de los elementos que se abstraen.

La abstracción permite estudiar un sistema complejo a diferentes niveles de detalle, es decir, la abstracción sigue un método *jerárquico*. El objetivo es poder representar y manejar sistemas complejos de manera más sencilla; para conseguirlo se suele realizar un proceso de abstracción en sentido descendente, lo que implica ir abstrayendo desde niveles más generales a niveles más detallados. El propio ser humano es un ejemplo de este tipo de abstracción: desde el nivel de los seres humanos en sociedades (sociología) al nivel de los constituyentes últimos de nuestro organismo (biología, química), pasando por el estudio de nuestros subsistemas (medicina).

Podemos encontrar otros muchos ejemplos en los que se utiliza la abstracción para tratar de disminuir la complejidad. Algunos de ellos pueden ser:

- Los sistemas de información en una empresa, desde su estructura departamental o geográfica hasta los procesos individuales que se realizan en cada punto.
- Un programa de ordenador es también un sistema complejo, ya que está compuesto de miles de instrucciones y cualquiera de ellas puede producir un error. Para disminuir la complejidad de los programas informáticos, se han llevado a cabo en toda la historia de la programación abstracciones como los lenguajes de alto nivel, que son abstracciones del lenguaje ensamblador o binario, y que permiten al desarrollador trabajar de un modo independiente de una máquina concreta.

El paso de un nivel a otro puede hacerse de un nivel de menor detalle (general) o mayor granularidad a otro de mayor detalle o granularidad más fina y viceversa, según el enfoque sea **descendente** o **ascendente**.

En resumen, podemos decir que la abstracción es un arma para la comprensión y resolución de sistemas o problemas complejos.

Así pues la evolución en la informática está basada en caminar hacia un grado creciente de abstracción. De hecho, los paradigmas de programación más modernos abstraen al programador de la secuencia concreta de instrucciones.

Abstracción Funcional y Abstracción de Datos

La abstracción produce como resultado el ocultamiento de la información, es decir, las entidades de uno de los niveles están compuestos internamente por entidades del nivel inferior, pero las ocultan. Podemos hablar de dos tipos de abstracción:

Introducción a los Tipos Abstractos de Datos (TAD) de FUOC está sujeta a una licencia de Reconocimiento-CompartirIgual 3.0 España de Creative Commons

- Abstracción funcional→ La abstracción funcional surge de la simple idea de crear procedimientos y funciones e invocarlos en diferentes partes del programa mediante un **nombre**. La parte que se ignora ("*la información que se oculta*") en este proceso es la de cómo el procedimiento o función realiza su tarea. La parte con la que nos quedamos es la signatura o interfaz: los parámetros de entrada y salida (y sus tipos), y la descripción de la tarea que realiza. Las ventajas de la abstracción funcional son las siguientes:
  - Generalización del concepto de operador→ Gracias a esta abstracción podemos definir operadores adicionales a los del lenguaje (por ejemplo, una función factorial), incluyendo la posibilidad de que dichos operadores no trabajen sobre tipos básicos del lenguaje (por ejemplo, determinantes o multiplicaciones de matrices).
  - Encapsulación y ocultamiento→ Haciendo que una secuencia de acciones esté oculta y sólo se haga visible desde el exterior su resultado global que se entrega a través de parámetros de salida bien definidos.
- Abstracción de datos → La primera aplicación de la abstracción a los datos está en los propios tipos de datos básicos de los lenguajes de programación. Un tipo boolean en un lenguaje como Pascal tiene una semántica bien definida, que abstrae la representación concreta en una máquina o sistema operativo determinado (que puede ser un bit o un valor entero). Así, un programa en un lenguaje de alto nivel puede ejecutarse en máquinas de arquitecturas diferentes con sólo recompilarlo. Las operaciones que pueden realizarse sobre un tipo boolean son las operaciones lógicas habituales, y, aunque puede que internamente sea un entero, el lenguaje no nos permite manipularlo como tal (por ejemplo, manipular un bit o sumar). En este sentido, lenguajes como C permiten "violar" la abstracción (y por tanto la semántica) de los tipos de datos.

  Un paso más adelante en la abstracción de datos es el de los tipos definidos por el programador, como los enumerados, que permiten definir explícitamente el dominio de valores y dar un nombre a cada uno de ellos aunque internamente el compilador los maneje como un subrango de enteros. El problema es que se puede restringir el dominio de valores, pero no definir nuevas operaciones ni restringir las existentes, ya que estas vienen predefinidas por el lenguaje de programación.

  Los tipos estructurados son un paso adicional que ayuda a conseguir la **genericidad**, es decir, se define un tipo como un agregado de instancias de otro tipo. Este tipo subordinado se suele denominar *tipo parámetro*, y al agregado, *tipo parametrizado*:

  ```
  t = vector [límInferior..límSuperior] de t2
  ```

  donde `t2` es el tipo parámetro y `t` el tipo parametrizado.

  El problema de los tipos estructurados[1] es que el lenguaje no permite que el programador establezca cuáles son las operaciones válidas sobre el tipo. Por ejemplo, dado un tipo estructurado registro "fecha", nada nos impide realizar asignaciones incorrectas desde el punto de vista de la lógica de una

[1] Estamos definiendo un tipo estructurado desde un punto de vista sintáctico, sin dotarlo de una semántica. La utilización correcta del tipo estructurado de acuerdo con su significado es responsabilidad del programador.

Introducción a los Tipos Abstractos de Datos (TAD) de FUOC está sujeta a una licencia de Reconocimiento-CompartirIgual 3.0 España de Creative Commons

fecha, ni realizar operaciones sin sentido sobre este tipo, como las que se muestran a continuación sobre f1 y f2, ambas variables del tipo "fecha":

```
f1.Dia= 30
f1.Mes= 2
f2.Dia= 5 * f1.Mes
```

Para evitar este tipo de "incoherencias lógicas de significado" surgen los tipos abstractos de datos (TADs), que a continuación se describen.

Construcción de tipos abstractos de datos

Un tipo abstracto de datos (TAD) es una colección de propiedades y de operaciones que se definen mediante una especificación que es independiente de cualquier representación.

La abstracción se centra en la independencia de la representación. Esto permite al programador modificar la representación del TAD sin que esto afecte a su utilización.

Se suele considerar que un tipo abstracto de datos es un tipo de datos construido por el programador para resolver una determinada situación.

Para definir un TAD, el programador ha de comenzar por definir las operaciones que se pueden realizar con él, es decir, qué operaciones son relevantes y útiles para operar con las variables pertenecientes al mismo. Esto se conoce como establecer la interfaz del tipo. La interfaz permite al programador utilizar el tipo (*qué* se puede hacer frente a *cómo* está hecho).

Siguiendo con el ejemplo de las fechas, podemos indicar que las operaciones válidas sobre una fecha son, entre otras:

```
Crear (dia, mes, año: natural): fecha
Incrementar (fechaInicio: fecha; numDias: entero): fecha
Distancia (fechaInicio, fin: fecha): entero
ObtenerMes (f: fecha): natural
......
```

La representación del tipo fecha en el TAD es independiente y de hecho, puede variar sin que afecte a los programas que utilicen el tipo, ya que la especificación se mantiene. El hecho de que el programador desconozca la representación (ocultamiento) obliga a definir tanto un constructor para el tipo abstracto que permita asignar valores a una variable de ese tipo como procedimientos y funciones que permitan establecer y obtener los valores del tipo[2].

Una de las principales ventajas de los TADs es que permite construir programas modularizados y estructurados, ya que las tareas lógicas para un tipo se *encapsulan* en un módulo independiente, que se compila independientemente y cuya representación, como se ha dicho, puede variar sin que afecte al programa que los utiliza siempre y cuando no se modifique la interfaz.

---

[2] En programación orientada a objetos, a este tipo de métodos que permiten el acceso a atributos se les conoce como *getters* y *setters.*

Introducción a los Tipos Abstractos de Datos (TAD) de FUOC está sujeta a una licencia de Reconocimiento-CompartirIgual 3.0 España de Creative Commons

Los TAD tienen, por lo tanto, dos partes:

- Especificación → Refleja qué hace el tipo. Es necesaria para que el programador sepa cómo debe implementar el TAD. Suele coincidir con la interfaz, pero no es necesario que esto ocurra, ya que la interfaz es un concepto de programación por lo que puede añadir operaciones a la especificación (siempre que tengan justificación).
- Implementación → La implementación a su vez se compone de una interfaz pública, que permite al programador utilizar el TAD, y una implementación que puede variar siempre y cuando la interfaz se mantenga.

Especificación de un TAD

La especificación de los TADs suelen definirse de manera formal, según Aho, de la siguiente manera: *"Podemos pensar que un TAD es un modelo matemático con un conjunto de operaciones definido sobre ese modelo"*.

Mediante la especificación de un TAD se conocen las operaciones que el tipo ofrece y cómo actúan estas operaciones.

El proceso de definición (o construcción de la especificación) de un tipo abstracto conlleva los siguientes pasos:

- Definir las operaciones interesantes para el tipo (incluyendo los constructores del tipo, que nos permiten asignar valores a variables del tipo). A estas operaciones se las denomina *primitivas* en algunos textos.
- Hacer un análisis detallado de las signaturas de esas operaciones.
- Establecer precondiciones, postcondiciones e invariantes para las operaciones. Una precondición es un predicado lógico que debe cumplirse sobre los parámetros de entrada de una operación para que ésta pueda realizarse correctamente. Una postcondición es un predicado lógico que cumplen los datos de salida de una operación, luego expresa condiciones que afectan al resultado de la operación. Un invariante es un predicado lógico que indica los estados válidos para un objeto. Los invariantes pueden afectar a la representación del tipo o a las variables que intervienen en un bucle, dependiendo de si se trata del invariante de la representación o de un invariante de bucle.

La especificación de un TAD debe poseer cuatro propiedades:

- Precisión→ Que implica *decir sólo lo imprescindible*.
- Generalidad→ Que quiere decir que sea *adaptable a diferentes contextos*.
- Legibilidad→ Entendida como *facilidad de comprensión*.
- No-ambigüedad→ Que quiere decir que *no dé lugar a distintas interpretaciones*.

La especificación puede hacerse de manera formal o informal (en lenguaje natural[3]). La especificación formal debe ajustarse a alguna notación matemática.

---

[3] Con una especificación en lenguaje natural, puede ser difícil garantizar algunas de las propiedades antes mencionadas como, por ejemplo, la de no ambigüedad. En cualquier caso,

Introducción a los Tipos Abstractos de Datos (TAD) de FUOC está sujeta a una licencia de Reconocimiento-CompartirIgual 3.0 España de Creative Commons

A continuación se muestra la especificación de un TAD Natural. Pese a que se viene utilizado notación procedimiental en los ejemplos comentados hasta el momento, en este caso la especificación está realizada pensando que un número natural es un objeto de la clase Natural, y por lo tanto el acceso a sus operaciones se hace mediante la notación punto.

Tipo: Natural

Operación para comprobar si un natural es igual a cero: Esta operación devolverá verdadero si el natural que evalúa es igual a cero.

n: Natural; b: Booleano

{n= X}

b= n.esCero ()

{b≡(X=0)}

Entendiendo por X un valor natural.

Operación para hallar el sucesor de un número natural: Esta operación devuelve el natural incrementado en una unidad.

n_0: Natural; n1: Natural

{n_0= X}

n_1= n_0.Sucesor ()

{n_1= X + 1}

Operación para hallar el predecesor de un número natural: Para hallar el natural que precede a otro hay que tener en cuenta que el valor cero no tiene predecesor, ya que sería un número negativo y por tanto no natural.

n_0: Natural; n1: Natural

{n_0= X ^ n_0 ≠ 0}

n_1= n_0.Predecesor ()

{n_1= X - 1}

Operación que comprueba si dos números naturales son iguales: Esta operación devuelve el valor lógico verdadero si ambos valores naturales son iguales.

n_0: Natural; n1: Natural; b: Booleano

{n_0= X ^ n_1 = W}

b= n0. Igual (n1)

{ b≡ (X=W)}

Operación que averigua si un natural es mayor que otro: Esta operación devuelve el valor booleano verdadero si el primer valor natural es mayor que el segundo.

n_0: Natural; n1: Natural; b: Booleano

{n_0= X ^ n_1 = W}

b= n0. Mayor (n1)

{ b≡(X>W)}

---

la situación "ideal" sería una solución intermedia, donde se combinaría la especificación formal con la especificación en lenguaje natural.

Introducción a los Tipos Abstractos de Datos (TAD) de FUOC está sujeta a una licencia de Reconocimiento-CompartirIgual 3.0 España de Creative Commons

Operación que averigua si un natural es menor que otro: Esta operación devuelve el valor booleano verdadero si el primer valor natural es menor que el segundo.

n_0: Natural; n1: Natural; b: Booleano

$\{n_0 = X \wedge n_1 = W\}$

b= n0. Menor (n1)

$\{ b \equiv (X < W)\}$

Operación que halla la suma de dos números naturales: El valor que devuelve esta operación es un natural igual a la suma de los dos valores de los naturales dados.

n_0= Natural; n_1= Natural; n_2= Natural

$\{n_0 = X \wedge n_1 = W \}$

$n_2 = n_0$.Suma (n_1)

$\{n_2 = X + W\}$

Operación que halla la resta de dos números naturales: El valor que devuelve esta operación es un natural igual a la resta de los valores de los naturales dados, teniendo en cuenta que no se puede llegar a efectuar si el resultado no es un natural.

n_0= Natural; n_1= Natural; n_2= Natural

$\{n_0 = X \wedge n_1 = W \wedge n_0 >= n_1\}$

$n_2 = n_0$.Resta (n_1)

$\{n_2 = X - W\}$

Implementación de un TAD

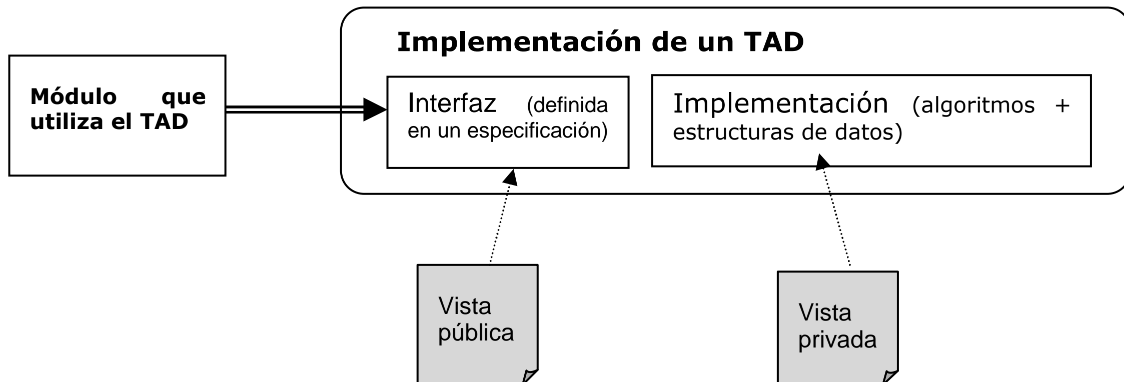
La implementación de un tipo abstracto de datos es un tipo de datos que consta de:

- Estructuras de datos propias (encapsuladas, invisibles para el usuario del tipo de datos). La elección de estas estructuras queda a la libertad del programador, que seleccionará las más eficientes o convenientes.
- Operaciones sobre esas estructuras propias (procedimientos y funciones). Dependerán de las estructuras internas seleccionadas.

Es muy importante resaltar la diferencia entre:

- Interfaces visibles, que deben diseñarse cuidadosamente, ya que no deberían cambiar a pesar de que la implementación cambie.
- Implementaciones encapsuladas, ocultas, que pueden cambiarse por otras mejores mientras no afecten a la interfaz.

Introducción a los Tipos Abstractos de Datos (TAD) de FUOC está sujeta a una licencia de Reconocimiento-CompartirIgual 3.0 España de Creative Commons



Para representar el modelo formal o matemático que define un TAD en un lenguaje de programación P, es decir, para **implementarlo** en P, tendremos que utilizar en último término los tipos de datos y operadores soportados por P. Al hacerlo, utilizamos **estructuras de datos**, que pueden definirse como colecciones de variables, posiblemente de diferentes tipos, conectadas de formas diversas.

Las estructuras de datos fundamentales dependen, por tanto, del lenguaje de programación. Por ejemplo, Pascal incorpora array, registro, fichero y conjunto. En otros lenguajes como Java, podemos considerar que las bibliotecas predefinidas proporcionan construcciones adicionales como tablas Hash, mientras que no soportan conjuntos, por ejemplo.

Introducción a los Tipos Abstractos de Datos (TAD) de FUOC está sujeta a una licencia de Reconocimiento-CompartirIgual 3.0 España de Creative Commons