Tema II – Tipos Abstractos de Datos (TAD)

En el capítulo anterior establecimos que la abstracción es el proceso fundamental mediante el cual extraemos las características esenciales de un problema, ignorando los detalles superfluos. Ahora damos un paso crucial hacia la materialización de este concepto en el diseño de software: la definición formal de Tipos Abstractos de Datos (TAD). Si la abstracción forma parte del *pensamiento*, el TAD es su *expresión formal* dentro de la ingeniería de software.

Un TAD representa la cristalización de los principios de abstracción, encapsulamiento e interfaz que ya hemos estudiado. Es la herramienta que nos permite definir nuevos tipos de datos que no existen nativamente en el lenguaje de programación, pero que son esenciales para modelar información que necesitamos para resolver problemas específicos. A través de los TADs, transformamos ideas abstractas —como una cola de espera, un contador o un número complejo— en componentes software reutilizables, robustos y fáciles de mantener.

Este capítulo nos guiará a través de la definición, especificación e implementación de TADs, conectando continuamente estos conceptos con los fundamentos establecidos previamente sobre abstracción, caja negra y contrato.

1. Definición y esencia de los TADs

Un Tipo Abstracto de Dato (TAD) es una descripción de alto nivel de una colección de datos junto con el conjunto completo de operaciones que pueden realizarse sobre ellos. Define la estructura lógica y el comportamiento esperado mediante un contrato bien especificado, pero deliberadamente omite cualquier detalle sobre cómo se implementa internamente**.**

La esencia del TAD radica en su independencia de la implementación. Podemos concebir un TAD como una "caja negra" perfecta: sabemos exactamente qué funciones ofrece (su interfaz) y qué comportamiento debemos esperar de ellas (su contrato), pero nos es completamente opaco cómo produce internamente esos resultados. Esta separación entre *qué hace* y *cómo lo hace* es precisamente la materialización práctica del principio de abstracción.

Consideremos un TAD Cola. Su especificación nos dice que permite las siguientes operaciones:

- `enqueue(elemento)`: Añade un elemento al final de la cola
- `dequeue()`: Elimina y devuelve el elemento del frente de la cola
- `isEmpty()`: Consulta si la cola no contiene elementos
- `front()`: Consulta cuál es el primer elemento sin eliminarlo
- `back()`: Consulta cuál es el último elemento

Internamente, esta cola podría implementarse de múltiples formas: Usando un array con índices para marcar el frente y el final, empleando una lista enlazada con nodos que apuntan al siguiente elemento, usando dos pilas de manera creativa o empleando un buffer circular para optimizar el uso de memoria.

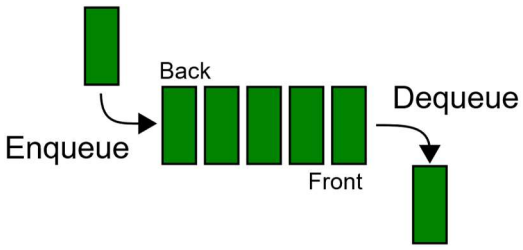

Ilustración 5. El TAD Cola

Lo más reseñable es que, a pesar de que existe tal amplitud en las posibilidades de implementación interna, desde la perspectiva del usuario del TAD todas estas implementaciones son intercambiables. El comportamiento externo es idéntico porque todas respetan el mismo contrato, lo cual ilustra perfectamente el poder del TAD: el programador que utiliza la cola puede concentrarse en *resolver su problema* (gestión de procesos, simulaciones, etc.) sin preocuparse por los detalles de implementación internos a la misma.

Las operaciones de un TAD pueden clasificarse según su propósito, lo que ayuda a diseñar interfaces completas y coherentes:

- Operaciones de Creación (Constructores): Permiten inicializar y obtener nuevos objetos del tipo que corresponde al TAD. En programación orientada a objetos se denominan constructores. Ejemplo: `crearColaVacia()`.
- Operaciones de Modificación (Mutadores): Alteran el estado interno del TAD, cambiando los valores almacenados o las relaciones entre ellos. Ejemplos: `enqueue()`, `dequeue()`.
- Operaciones de Consulta (Observadores): Extraen información del TAD sin modificarlo, lo cual proporciona una "ventana" al estado interno. Ejemplos: `front()`, `isEmpty()`.
- Operaciones propias del TAD: Funciones específicas que tienen sentido solo para ese tipo abstracto particularm y que por tanto no son aplicables a cualquier tipo. Ejemplos: el determinante de una matriz, la conjugación de un número complejo, o el cálculo de la distancia entre dos puntos en el espacio.

Esta clasificación guía no solo el diseño del TAD, sino también su documentación y uso, ayudando a los programadores a comprender rápidamente cómo interactuar con el nuevo tipo.

2. Especificación formal

La especificación de un TAD es el proceso mediante el cual definimos formalmente su contrato. Consiste en separar claramente el comportamiento esperado (qué debe hacer) de cualquier implementación concreta (cómo lo hace). Estamos, en esencia, definiendo cómo se comportará un nuevo tipo de datos, estableciendo las reglas del juego antes de escribir la primera línea de código.

Existen diferentes enfoques para especificar TADs, desde el uso de lenguaje natural (accesible pero potencialmente ambiguo) hasta especificaciones algebraicas (rigurosas y formales) o lenguajes de especificación especializados. En este nivel introductorio, nos centraremos en especificaciones en lenguaje natural estructurado, que constituyen un excelente punto de partida.

Una especificación completa en lenguaje natural suele contener los siguientes elementos:

- **Nombre del TAD:** Identificador claro y descriptivo.
- **Valores posibles:** El conjunto de estados válidos que puede tomar una instancia del TAD.
- **Operaciones:** Lista completa de operaciones disponibles, incluyendo para cada una:
  - Nombre y propósito
  - Parámetros de entrada y tipo de retorno
  - Precondiciones (condiciones que deben cumplirse antes de ejecutar la operación)
  - Postcondiciones (condiciones garantizadas después de ejecutar la operación)
- **Uso de otros tipos:** Dependencias con tipos de datos existentes.

Veamos algunos ejemplos prácticos de especificación.

Ejemplo 1: Semáforo Binario

- **Nombre:** `Semáforo`
- **Valores posibles:** `rojo`, `verde`
- **Operaciones:**
  - `abrir()`: Procedimiento que cambia el estado interno a `verde`

    - `cerrar()`: Procedimiento que cambia el estado interno a `rojo`
    - `estaAbierto()`: Función que devuelve `true` si el estado es `verde`, `false` si es `rojo`
- **Uso de otros tipos:** Se podría implementar usando un `boolean`, un `char`, o incluso un entero, pero esa decisión pertenece a la implementación, no a la especificación.

Ejemplo 2: Semáforo de 3 estados

- **Nombre:** `SemáforoCompleto`
- **Valores posibles:** `rojo`, `ámbar`, `verde`
- **Operaciones:**
    - `abrir()`: Procedimiento que cambia el estado interno a `verde`
    - `cerrar()`: Procedimiento que cambia el estado interno a `ámbar`, espera 3 segundos, luego cambia a `rojo`
    - `estaAbierto()`: Función que devuelve `true` si el estado es `verde`, `false` en caso contrario
- **Uso de otros tipos:** Como en el ejemplo anterior, se podría implementar usando un `boolean`, un `char`, o incluso un entero, decisión que pertenece a la implementación.

Observa cómo el comportamiento de `cerrar()` se ha hecho más complejo, pero la interfaz pública (los nombres de las operaciones) puede mantenerse idéntica. Esto demuestra cómo podemos evolucionar la implementación sin romper el contrato con los usuarios existentes.

Ejemplo 3: Contador simple

- **Nombre:** `Contador`
- **Valores posibles:** Enteros no negativos (0, 1, 2, ...)
- **Operaciones:**
    - `inicializar()`: Procedimiento que establece el valor interno a 0
    - `incrementar()`: Procedimiento que suma 1 al valor interno
    - `obtenerValor()`: Función que devuelve el valor interno actual
    - `decrementar()`: Procedimiento que resta 1 al valor interno. Esta operación tiene una precondición**:** El contador no debe estar en 0
- **Uso de otros tipos:** Enteros para el valor interno.

Ejemplo 4: Contador con Límite

- **Nombre:** `ContadorLimitado`
- **Valores posibles:** Enteros desde 0 hasta un límite máximo L
- **Operaciones:**
    - `inicializar()`: Procedimiento que establece el valor interno a 0
    - `establecerLimite(L)`: Procedimiento que define el límite máximo
    - `incrementar(n)`: Procedimiento que suma n unidades al valor interno. Debe contemplar que, si al incrementar se supera el límite L, el contador vuelve a 0.
    - `obtenerValor()`: Función que devuelve el valor interno actual
    - `decrementar()`: Procedimiento que resta 1 al valor interno, teniendo en cuenta la precondición siguiente: El contador no debe estar en 0
- **Uso de otros tipos:** Enteros para el valor y el límite.

Estos ejemplos muestran cómo una especificación clara define completamente el comportamiento esperado, dejando abiertas múltiples posibilidades de implementación. También ilustran cómo podemos extender TADs existentes (de `Contador` a `ContadorLimitado`) manteniendo la compatibilidad cuando ello es deseable.

3. Implementación en Pascal

La implementación de los TADs en un lenguaje de programación conlleva la adaptación a las especificidades del lenguaje a emplear. De los varios conceptos importantes necesarios para implementar un TAD tal y como los hemos descrito, algunos Pascal no los soporta nativamente, requiriendo disciplina por parte del programador:

- **Privacidad (encapsulación):** Pascal distingue entre lo público (accesible desde fuera) y lo privado (detalles internos ocultos). En el nivel más restrictivo, el usuario externo no puede acceder directamente a la representación interna, solo a través de las operaciones públicas, pero Pascal no permite especificar, como Java, distintos niveles de privacidad.
- **Genericidad:** Capacidad de definir TADs parametrizados por tipos. Por ejemplo, una "Cola" genérica que pueda contener cualquier tipo de elemento (enteros, registros, etc.), especificando el tipo concreto al crear una instancia. Pascal estándar no soporta este concepto directamente.
- **Sobrecarga:** Posibilidad de tener múltiples subprogramas con el mismo nombre pero diferentes parámetros. Por ejemplo, `incrementar()` sin parámetros y `incrementar(n)` con un parámetro. Pascal no permite sobrecarga de funciones/procedimientos.

Dado que Pascal carece de soporte nativo para estos conceptos avanzados**,** la disciplina del programador será esencial para respetarlos**.** Así, debemos utilizar el TAD exclusivamente a través de su interfaz pública, nunca accediendo directamente a sus datos internos, aunque el lenguaje técnicamente lo permita.

Para implementar TADs en Pascal utilizamos unidades (UNITs)**,** que es el mecanismo del lenguaje para llevar a cabo la modularización y encapsulación de TADs. Una unidad en Pascal consta de dos partes claramente separadas:

- Interfaz Pública (Sección `INTERFACE`): Declara todo lo que será visible y accesible desde programas externos. Aquí definimos los tipos públicos, constantes, variables, y las cabeceras de los subprogramas que componen la interfaz del TAD.
- Implementación Privada (Sección `IMPLEMENTATION`): Contiene el código real de los subprogramas declarados en la interfaz, además de posibles tipos, constantes y variables auxiliares que son privados (no accesibles desde fuera).

El nombre de la unidad y el nombre del archivo que la contiene deben coincidir. Por ejemplo, la unidad `uContador` se guardaría en `uContador.pas`.

Veamos a continuación un ejemplo para un TAD sencillo: el contador. Su interfaz pública estaría almacenada en un archivo denominado `uContador.pas` y su contenido sería el siguiente:

```
unit ucontador;
interface
type
  contador = ^tcontador;  // tipo opaco: el usuario solo ve un puntero
  tcontador = record      // definición interna (invisible externamente)
    valor: integer;
  end;

// operaciones públicas del tad
procedure crear(var c: contador);
procedure incrementar(var c: contador);
function obtenervalor(c: contador): integer;
procedure destruir(var c: contador);
```

En cuanto a su implementación, que sería privada y por tanto no accesible a los usuarios del TAD, esta iría en el mismo archivo (`uContador.pas`) e inmediatamente a continuación de la interfaz. Sería la siguiente:

```
implementation

procedure crear(var c: contador);
begin
  new(c);           // reserva memoria dinámica
  c^.valor := 0;    // inicializa el valor interno
end;

procedure incrementar(var c: contador);
begin
  c^.valor := c^.valor + 1;
end;

function obtenervalor(c: contador): integer;
begin
  obtenervalor := c^.valor;
end;

procedure destruir(var c: contador);
begin
  dispose(c);       // libera la memoria dinámica
  c := nil;         // buen hábito: evitar punteros colgantes
end;

end.
```

Finalmente, el uso del TAD desde un programa Pascal sería algo como esto:

```
program pruebacontador;
uses ucontador;  // incluye la unidad con el tad
var
  micontador: contador;
begin
  // usamos solo las operaciones públicas
  crear(micontador);
  incrementar(micontador);
  incrementar(micontador);
  writeln('valor actual: ', obtenervalor(micontador));  // muestra 2
  destruir(micontador);
end.
```

4. Beneficios del uso de TADs

La adopción de TADs en el desarrollo de software produce beneficios sustanciales que impactan en todas las fases del ciclo de vida del software.

La primera es la **separación de responsabilidades**. El uso de TADs promueve una clara separación entre los siguientes roles:

- El programador del TAD, que se concentra en implementar correctamente las operaciones especificadas, optimizandolas para maximizar su eficiencia, robustez, etc.
- El programador cliente, que utiliza el TAD a través de su interfaz pública, concentrándose en la lógica de negocio de su aplicación sin preocuparse por los detalles internos del TAD.

Otro beneficio tangible es la **facilidad de mantenimiento**. Esta propiedad mejora porque cuando la implementación de un TAD necesita cambios (por optimización, corrección de errores, o nuevas características), el código cliente no necesita modificarse siempre que se mantenga inalterada la interfaz pública. Esto reduce mucho el riesgo de introducir errores al modificar sistemas complejos.

Por otra parte, se potencia la reutilización y la fiabilidad de los componentes software, algo muy deseable desde la perspectiva de la ingeniería del software. La existencia de TADs bien diseñados y probados que se convierten en **componentes reutilizables** y que pueden integrarse en múltiples proyectos potencia la construcción de sistemas complejos a partir de componentes ya existentes (en su mayoría, al menos). Dado que su interfaz es estable y su comportamiento está especificado formalmente, podemos confiar en ellos como "cajas negras" certificadas.

Por último, decir que el código que utiliza TADs resulta **más legible** y expresivo. En lugar de operaciones de bajo nivel sobre estructuras de datos primitivas, vemos operaciones de alto nivel que reflejan directamente el dominio del problema. Compara los dos códigos siguientes:

Sin TAD (bajo nivel)	Con TAD (alto nivel)
`arrayCola[final] := elemento;` `final := final + 1;`	`encolar(miCola, elemento);`

Como se aprecia, la versión con TAD es inmediatamente comprensible, mientras que la versión sin TAD requiere entender la representación interna específica.

Y finalmente, podemos citar el beneficio que supone la **abstracción progresiva**, ya que en el fondo la construcción de programas complejos se reduce a interconectar módulos TADs existentes y escribir una cantidad mínima de código "pegamento" específico del problema. Esta aproximación modular es escalable y gestionable incluso para sistemas de gran envergadura.

Resumen

Los Tipos Abstractos de Datos (TADs) representan la materialización práctica de los principios de abstracción estudiados en el capítulo anterior. A través de ellos, transformamos conceptos del dominio del problema en componentes software reutilizables, encapsulados y bien definidos.

La especificación formal (el "contrato") establece el comportamiento esperado, mientras que la implementación concreta (usando unidades en Pascal, clases en POO, etc.) proporciona la realización eficiente de ese comportamiento. Esta separación es poderosa: nos permite razonar sobre corrección a nivel abstracto, cambiar implementaciones para optimizar rendimiento, y construir sistemas complejos mediante la composición de componentes probados y fiables.

Al dominar el diseño e implementación de TADs, adquirimos una habilidad fundamental para cualquier ingeniero de software: la capacidad de crear abstracciones efectivas que capturen lo esencial de un problema, ignorando lo accesorio. Esta habilidad trascenderá el uso de Pascal o cualquier lenguaje específico, constituyendo una piedra angular del pensamiento computacional y el diseño de sistemas robustos y fáciles de mantener.

En los próximos capítulos, aplicaremos estos conceptos para construir estructuras de datos clásicas —pilas, colas, listas, árboles— cada una implementada como un TAD con su interfaz bien definida y múltiples posibles implementaciones. Los TADs serán nuestro marco unificador para entender, comparar y utilizar estas estructuras fundamentales.