

Tipo de Dato Abstracto

Estructura de Datos

Juan Carlos Carranza Acalco

Amilcar Serrano Ávila

Ingeniería en Sistemas Computacionales

**Instituto de Estudios Superiores del Pacífico Sur**

Tipo de Dato Abstracto

**Introducción - Concepto de TDA**

**Un Tipo de Dato Abstracto (TDA) es un modelo que define valores y las operaciones que se pueden realizan sobre ellos. Y se denomina abstracto ya que la intención es que quien lo utiliza, no necesita conocer los detalles de la representación interna o bien el cómo están implementadas las operaciones.**

**Es por esto una práctica que nos provee un grado de abstracción que permite desacoplar al código que usa un TDA de aquel código que lo implementa.**

**Veamos un ejemplo en pseudo código:**

**tipo Pantalla**

**operacion escribir (Pantalla, String)**

**operacion leer (Pantalla) -> String**

**Acá definimos un tipo Pantalla y dos operaciones sobre él.**

**La operación tiene un nombre y, como una función, un dominio e imagen. Se lee así**

**operacion nombre (parametro1, parametro2, ...) -> retorno**

**En nuestro caso simple, tenemos dos funciones: para escribir en pantalla, y para leer.**

**Fíjense que mediante este tipo, yo puedo escribir código que opere con la pantalla, sin conocer cómo realmente se escribe o lee de ésta.**

**TDA Diccionario**

**Veamos otro ejemplo.**

**Un Diccionario o Mapa (en software) es una abstracción que vincula un dato con otro. Así como en el diccionario de idioma, asocia una palabra con su definición, y podemos buscar en el a través de la palabra. El TDA Mapa/Diccionario permite tener muchas "entradas" y cada una de estas tendrá una "clave" y un "valor" asociado. Es interesante notar que el concepto de diccionario/mapa no define cómo se asociarán esos datos, es decir cómo se van a guardar internamente los valores. Entonces podríamos tener una implementación que usa dos listas paralelas, u otra que use un array de tuplas, etc. etc. Para el código que utiliza el Diccionario/Mapa es indiferente. Es algo así como un detalle de implementación.**

**Acá vemos un ejemplo muy simple donde un fragmento de código sabe recorrer un mapa e imprimir "clave=valor".**

**Este código puede funcionar tanto con un TDA implementado como un vector de tuplas, así como también con una implementación diferente que utilice dos vectores paralelos (uno para las claves y otro para los valores).**

**Veamos cómo sería la definición del TDA en nuestro pseudo lenguaje:**

**tipo Diccionario<Clave, Valor>**

**Tipos paramétricos (Tipos genéricos)**

**Algo que notamos aquí es que un Diccionario, como la mayoría de las colecciones o conjunto de elementos, podrían contener cualquier tipo de objetos. Al menos eso es lo que nos gustaría definir a nosotros acá. La mayoría de los lenguajes permiten actuales permite esto. Porque no queremos tener un DiccionarioDeStrings, DiccionarioDeInts, DiccionarioDeDoubles, etc, etc.. Si no, un único diccionario que permita ser utilizado para cualquier tipo.**

**Ahora, Diccionario ya es un tipo ! Como definiríamos un tipo que puede ser utilizado sobre otros tipos ?**

**A esto se le llama "tipos paramétricos", y se dice que el tipo Diccionario en este caso, es un tipo genérico.**

**Hay mucha más teoría sobre esto y la mayoría de los lenguajes modernos con checkeos de tipos en tiempo de compilación tienen algún tipo de implementación de esto. Acá pueden explorar más**

**Operaciones del Diccionario**

**Volviendo al ejemplo, decimos que el tipo Diccionario tiene dos tipos paramétricos, uno especifica el tipo de sus claves, y el otro el tipo de sus valores.**

**Pero eso no importa, ya lo veremos (verán) más adelante en otra materia.**

**Veamos entonces las operaciones que nos gustaría definir para el Diccionario.**

**operacion valor (Diccionario, Clave) -> Valor**

**operacion insertar (Diccionario, Clave, Valor) -> Diccionario**

**operacion entradas (Diccionario) -> Lista<Entrada<Clave,Valor>>**

**Permiten, dado un Diccionario, obtener el valor asociado a una clave, insertar una clave-valor, y obtener la lista de entradas que contengan todas las clave-valor.**

**Para eso se define otro tipo Entrada, bastante trivial:**

**tipo Entrada<Clave, Valor>**

**operacion clave (Entrada) -> Clave**

**operacion valor (Entrada) -> Valor**

**Y veamos cómo sería también en pseudo código un ejemplo de uso de este Diccionario:**

**Diccionario<String, int> edades**

**insertar(edades, "Alberto", 23)**

**insertar(edades, "Maria", 21)**

**insertar(edades, "Juan", 34)**

**insertar(edades, "Zoe", 5)**

**for entrada in entradas(edades) {**

**escribir(clave(entrada) + " " + valor(entrada))**

**}**

**Lo interesante de esto es que:**

**tenemos una definición de qué es un Diccionario**

**tenemos una aplicación (muy simple, pero podría ser más complicada) que crea un diccionario con edades de personas y las imprime en pantalle**

**Pero nunca definimos cómo se implementa un Diccionario !!! Cómo se guardan las clave-valor ? Cómo se inserta y se obtienen los valores para una clave ??**

**Justamente este es el objetivo de los Tipos de Dato Abstracto, poder tener código que utiliza un tipo (el qué) , sin estar acoplado al cómo está implementado.**

**Objetivo (Abstracción -> Flexibilidad)**

**El objetivo o la utilidad de la idea de usar TDA's es la de conseguir una mayor flexibilidad. Y ésto lo logramos mediante el concepto más general de abstracción. Como vimos en el ejemplo, el código que utiliza el TDA no conoce ni depende de la implementación de las operaciones.**

**La abstracción es probáblemente el concepto más general y más importante de la programación (o cualquier otra resolución de problemas).**

**En parte tiene que ver con la famosa idea del divide y conquista. Porque separamos los problemas, hacemos que diferentes partes de la aplicación se ocupen de una y solo una parte del problema. Así, si luego alguna de ellas cambia, las demás partes no se ven afectadas (o al menos el impacto no es tan grave).**

**Es por esto que obtenemos flexibilidad, ante el cambio.**

**Todo esto muy lindo en teoría. Para bajarlo más a detalle, necesitamos entender otros dos conceptos: Encapsulamiento y Ocultamiendo de la Información**

**Encapsulamiento**

**Como dijimos un TDA está compuesto por:**

**la definición de los valores**

**la definición de las operaciones que aplican sobre esos valores.**

**Y, sabemos que desacoplan el código cliente de los detalles de la implementación o representación interna (es decir el estado).**

**Ahora, para lograr el objetivo de la flexibilidad y luego facilitar los cambios, necesitaría cierta forma de agrupar las implementaciónes de las operaciones, y las del TDA, de forma de formar la idea de un componente, que luego sea fácil ser modificado/reemplazado.**

**A esto se lo llama encapsulamiento. Y el mecanismo concreto va a depender del paradigma en el que estemos trabajando, así como también del lenguaje en particular.**

**Por ejemplo, los lenguajes orientados a objetos, proveen naturalmente un mecanismo para el encapsulamiento, ya que un objeto es justamente una entidad que contiene tanto el estado como el comportamiento.**

**Vemos un ejemplo acá muy sencillo en pseudo código, utilizando el TDA Diccionario que estuvimos viendo antes:**

**tipo Diccionario<Clave, Valor> {**

**tiene Vector claves**

**tiene Vector valores**

**operacion valor(Clave clave) {**

**int indice = claves.indiceDe(clave)**

**retornar valores[indice]**

**}**

**operacion entradas(...)**

**operacion insertar(...)**

**...**

**}**

**Este diccionario define un tipo de objeto, como una entidad autocontenida. Estamos definiendo dos tipos de cosas:**

**Estado: a través de la keyword tiene definimos variables miembro del objeto. Algo así como vimos en los structs que tenían miembros (ojo, esto es una metáfora, de ninguna manera structs es = objetos).**

**Comportamiento: a través de la definición de operaciones. Una operación es una funcionalidad o responsabilidad del objeto. Algo que sabe hacer.**

**Veamos cómo se utilizaría esto:**

**Diccionario diccionarioEdades = crear Diccionario**

**diccionarioEdades.insertar("Alberto", 23)**

**diccionarioEdades.insertar("Maria", 21)**

**diccionarioEdades.insertar("Juan", 34)**

**diccionarioEdades.insertar("Zoe", 5)**

**Para tener un diccionario, necesitamos priméramente crearlo. Para eso usamos la palabra reservada crear que genera un nuevo objeto para un tipo dado (Diccionario en este caso).**

**Luego asignamos eso a una variable diccionarioEdades y luego cualquier operación que querramos ejecutar sobre un diccionario la invocamos sobre el mismo objeto.**

**Como ven acá tanto estado como comportamiento están encapsulados. Están definidos en un solo lugar, bajo el tipo Diccionario.**

**Ocultamiento de la Información (Information Hiding)**

**Muchas veces se confunde el encapsulamiento con la ocultación de la información. La diferencia es sutil, pero importante.**

**Encapsular, es decir agrupar comportamiendo y estado, no necesariamente significa ocultar. En nuestro ejemplo anterior.**

**diccionarioEdades.claves[3] = "Jose"**

**Acá estamos accediendo al miembro claves del diccionario. A veces se dice de esto que estamos rompiendo el encapsulamiento. Lo que está sucediendo es que estamos accediendo a un detalle de implementación. Si luego queremos cambiar la implementación del Diccionario por otra que no use vectores paralelos, si no otra forma, éste código se va a ver afectado.**

**Entonces, acá aparece la idea de ocultamiento de la información.**

**Conceptualmente se refiere a la idea de ocultar detalles de implementación porque**

**son susceptibles a cambios**

**no queremos que impacten otras partes del sistema.**

**De nuevo, aquí dependerá del paradigma y del lenguaje el mecanismo a utilizar para lograr el ocultamiento. Hay lenguajes que permiten definir visibilidad a sus miembros.**

**Veamos como sería en nuestro pseudo código:**

**tipo Diccionario<Clave, Valor> {**

**tiene Vector claves privado**

**tiene Vector valores privado**

**...**

**}**

**Al decir privado estamos definiendo que estos miembros solo pueden ser vistos o accedidos desde las operaciones que pertenecen al Diccionario, y no desde "fuera".**

**Entonces el código anterior "diccionarioEdades.claves[3] = "Jose"" no compilaría.**

**Modularización**

**Por último, ya que estamos, vamos a vincular estos conceptos, con un tercero.**

**La idea de modularización se puede pensar como la de ocultamiento y encapsulamiento, pero a aplicadas a un nivel más amplio o general de la aplicación. No solo a una unidad, como vimos a un Tipo u objeto, si no a un conjunto de éstos.**

**Al agruparlos y dividirlos definimos módulos.**

**Por ejemplo:**

**Módulo de colecciones: provee un conjunto de TDA's para manipular conjuntos y estructuras de datos como listas, colas, diccionarios, etc.**

**En este caso a veces se conocen como librerías.**

**Módulo de interfaz de usuario: en mi aplicación podría separar la lógica del dominio de aquella que modela la interfaz de usuario.**

**En este caso se conoce como un módulo o componente de la aplicación.**

**...**

**Lo importante es que sirven para abstraer a otras partes de la aplicación de decisiones de diseño que podrían sufrir cambios.**

**La modularización es un tipo de Separation Of Concerns.**

**Resumiendo**

**Encapsulamiento**

**Agrupar estado y comportamiento. No necesariamente es ocultamiento de la información (information hiding).**

**En general se refiere a los elementos básicos del paradigma, unidades, como un Tipo, u objeto. (a diferencia de la idea de módulo)**

**Ocultamiento de la información**

**Separación entre detalles de implementación de la definición.**

**para tener menor impacto ante cambios**

**para "proteger" a las otras partes del sistema.**

**Modularización**

**Ocultar decisiones difíciles de diseño, o aquellas que podrían estar sujetas a cambios. (lleva a desacoplamiento)**

**Involucra el encapsulamiento y el ocultamiento de la información, pero a escalas mayores. En general un módulo no es una unidad, si no un conjunto de los elementos del paradigma (un conjunto de funciones, o un conjunto de clases, etc).**

**Desacoplamiento**

**Medida de (baja) dependencia entre dos partes/módulos.**

**Abstracción**

**Acá nos referimos a la idea de simplificar una operación o un concepto modelándola con los elementos del lenguaje (una función, o un objeto) de modo de:**

**poder reutilizarla en diferentes partes del programa, evitando duplicados**

**poder encapsular su comportamiento y ocultar la información, para lograr mayor flexibilidad.**

**Referencias**

**"Encapsulation is not information hiding"**