Programación II
Tipos Abstractos de Datos(TAD)

Definición

Un Tipo Abstracto de Datos es un conjunto de valores y de operaciones definidos mediante una especificación independiente de cualquier representación.

TAD = valores + operaciones

La manipulación de un TAD sólo depende de su especificación, nunca de su implementación.

Especificación / implementación

Dada una especificación de TAD hay muchas implementaciones válidas.
Un cambio de implementación de un TAD es transparente a los programas que lo utilizan

Reusabilidad

Se puede:

- Implementar un TAD's sólo a partir de la especificación, sin saber para qué se van a usar.
- Utilizar los TAD's sólo conociendo la especificación.
- Cambiar la implementación un TAD utilizado a otra más eficiente

Abstracción:

- Destacar los aspectos relevantes del objeto.
- Ignorar los aspectos irrelevantes del mismo

Abstracción funcional:

Crear procedimientos y funciones e invocarlos mediante un nombre donde se destaca qué hace la función y se ignora cómo lo hace. El usuario sólo necesita conocer la especificación de la abstracción (el qué) y puede ignorar el resto de los detalles (el cómo).

Ocultamiento de información:

Ocultar *decisiones de diseño* en un programa susceptible de cambios con la idea de proteger a otras partes del código si éstos se producen.

Proteger una decisión de diseño supone proporcionar una *interfaz* estable que proteja el resto del programa de la implementación (susceptible de cambios).

Encapsulamiento:

Mantener todas las características, habilidades y responsabilidades de un objeto por separado.

Especificar comportamientos esperados inesperados:

Una buena práctica de diseño es ser declarativo con los nombres (hacer lo que se dice) y ser coherente respecto de los efectos secundarios de nuestros.

Por ejemplo, avisar si tendremos aliasing. O si el Tad queda alterado de forma no deseable luego de cierta operación.

Invariante de representación

Definición: Afirmación sobre un objeto que debe ser cierta en todo momento.

Durante la implementación es necesario establecer que instancias van a representar estados válidos de TAD y cuáles no.

Durante la especificación solamente describimos la interfaz del TAD (operaciones), pero no decimos que otras cosas tienen que valer en todo momento.

Durante la implementación debemos garantizar que esas operaciones no corrompan el estado interno, es decir, que dichas operaciones no violen el invariante de representación.

Ejemplo1: Especificar el TAD Nat

Nat son los números naturales de matemática

Especificación:

```
Nat(Integer n){}        // Constructor.
void sumar(Nat n){}
```

Invariante de representación: $n \geq 0$

Emeplo2: Completar la implementación del TAD Fecha según la siguiente especificación

Asumir que los días van del 1 al 31

```
Fecha(Integer dia, Integer mes, Integer año){}        // Constructor.
void sumar(Fecha f) {}
Integer dia(){}
Integer mes(){}
Integer año(){}
```

Invariante de representación:

$31 \geq dia \geq 1$

$12 \geq mes \geq 1$

$9999 \geq año \geq 1$

NOTA

Todas las clases de un TAD deben implementar el método toString y un test que compruebe algún caso de uso.

```
public class Fecha {

	private int dia;
	private int mes;
	private int año;

	Fecha(int dia, int mes, int año){
		if (dia>31){
			throw new RuntimeException("El dia debe estar entre 1 y
			31");
		}
		//...
		this.dia = dia;
		this.mes = mes;
		//...
	}

	public int dia(){
		return dia;
	}

	public int mes(){
		return mes;
	}
```

```
public int año(){
		return año;
	}


public void sumar(Fecha fecha){
		dia = dia + fecha.dia;
		if (dia > 31){
			mes = mes + 1;
			dia = dia - 31;
		}
		//...
	}

	@Override
	public String toString(){
		return dia +"/"+ mes +"/"+ año;
	}

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		Fecha fecha1 = new Fecha(1,1,1);
		Fecha fecha2 = new Fecha(31,1,1);

		fecha1.sumar(fecha2);

		System.out.println(fecha1.toString());

	}

}
```

Verificación del Invariante

Algo que es un poco confuso sobre los invariantes es que a veces, estos dejan de cumplirse. Por ejemplo, en la mitad de sumar, `dia = dia + fecha.dia,` el invariante no se cumple.

Este tipo de violación al invariante es aceptable; de hecho, usualmente es imposible modificar un objeto sin violar un invariante al menos por un momento. Normalmente el requerimiento es que todo método que viole un invariable debe restablecerlo antes de terminar.

Tad's Paramétricos (Generics)

Como parte de la especificación se puede agregar un parámetro formal que determina sobre qué tipos de elementos va a operar el TAD.

La sintaxis en java para especificar un parámetro formal es agrando corchetes "<>" al final del nombre de cada clase del TAD.

Los parámetros formales se denominan variables de tipo, porque el dominio de dichas variables se compone de tipos de Java y tipos generados por los usuarios.

Ejemplos de tipos implementados en java
-Integer
-String

Ejemplo3: ArregloEstatico<T>

Queremos simular el comportamiento del arreglo estático de java "[]", que no soporta Generics, por un TAD que soporte Generics.

TAD ArregloEstatico<T> //T es una ***variable de tipo***

Notar que el TAD no es "ArregloEstatico", sino "ArregloEstatico<T>". El tipo de T se termina en tiempo de compilación.

Dominio(T) = TiposJava U TiposUsuairo

El ejemplo es interesante, porque como java no permite arreglos estáticos de elementos paramétricos (ej T []), construiremos uno:

```
public class ArregloEstatico<T> {
	ArrayList<T> datos;

public ArregloEstatico(int tamano){
	datos = new ArrayList<T>( );

	for (int i=0; i< tamano; i++){
		datos.add(null);
	}
}

public T leer(int indice){
	if (indice < datos.size())
		return datos.get(indice);
	else
		throw new RuntimeException("indice incorrecto");
}
```

```
public void escribir(T valor, int indice){
	if (indice < datos.size())
		datos.set(indice, valor);
	else
		throw new RuntimeException("indice incorrecto");

}
}


public class Test {

	public static void main(String[] args) {
		ArregloEstatico<String> a1 = new ArregloEstatico<String>(5);
		ArregloEstatico<Fecha> a2 = new ArregloEstatico<Fecha>(3);

		a1.escribir("hola0", 2);
		a1.escribir("hola1", 3);

		a2.escribir(new Fecha("01/01/2016"), 2);

		System.out.println(a1.leer(3));
		System.out.println(a2.leer(2).fecha);

	}

}
```

Problems @ Javadoc Declaration Console

<terminated> Test (6) [Java Application] C:\Program Files

```
hola1
01/01/2016
```

Cohesión y Acoplamiento[1] Otros aspectos importantes para el diseño de un TAD

El acoplamiento es el grado de conocimiento que dos TAD's (o clases) tienen que saber el uno del otro para poder funcionar.

La cohesión es el grado de agrupamiento (por funcionalidad y responsabilidad) que tienen los métodos de un TAD.

En los artículos de Stevens y Constantine, y en la industria del software en general, se recomienda tener:

-Poco acoplamiento

-Alta Cohesión

Veamos ejemplos donde esto no ocurre, y como mejorarlo:

Ejemplo4: Se desea modelar el juego Piedra Papel o Tijera para soportar más elementos y reglas que el juego original.

Diseño1
Única Clase

Clase PPT
ArrayList elementos
ArrayList reglas // son globales, todos las conocen
Void jugar(elemento1, elemento2)

Se puede observar que tenemos baja cohesión, porque todo el TAD tiene que conocer todos los problemas y estructuras de representación en todo momento.
Como resultado, modificar levemente el irep implica la modificación de todo el TAD

[1] Stevens, Myers y Constantine (1974) ; Yourdon y Constantine, 1979

Diseño2
Dos clases

Clase Elemento //única responsabilidad: Saber si le gana a otro

Clase PPT
ArrayList elementos
ArrayList reglas // son globales, todos las conocen
Void jugar(elemento1, elemento2)
-Le pregunta al elemento si le gana a otro

Bajo la cohesión levemente, pero veamos el acoplamiento:

Elemento tiene que saber cómo PPT implementó las reglas para saber cómo usarlas.

Tenemos alto acoplamiento.

Diseño3
Tres clases

Clase Regla // Solo sabe si a>b o si b>a

Clase Elemento
Lista reglas

Clase PPT
ArrayList elementos
Void jugar(elemento1, elemento2)
-Le pregunta al elemento si le gana a otro

Notar que la regla no debe forzar el irep (no puede pasar que a>b ^ b>a) porque si así lo hiciera tendríamos baja cohesión, es decir, una responsabilidad demasiado grande para una regla).

El que fuerza esta parte del irep es PPT

¿Podemos seguir generando más y más clases?

Sí, pero:

Diseño4
Cutro clases

Agregamos la clase jugador
...Que no tiene ninguna responsabilidad salvo saber que es el jugador1 o el jugador2

Entonces debemos parar este proceso cuando deja de haber responsabilidad para el nivel de abstracción del modelo que elegimos (ver modelo TAD Fecha sin fechas negativas por ejemplo).

Ok, pero que pasa si el día de mañana quiero un PPT multijugador, donde cada jugador pueda establecer una estrategia que el juego debe seguir.

No veremos este diseño en particular, pero queda claro que en este caso si conviene agregar una Clase jugador al diseño.

Ejemplo5: Diseñar el TAD Monopolio de la práctica, justificando el diseño elegido con los conceptos que acabamos de ver.

Recorrer una estructura de datos en Java

Hasta ahora vimos que podíamos recorrer un TAD o una estructura de datos de la siguiente manera

```
ArrayList<Integer> lista = new ArrayList();

lista.add(1);
lista.add(2);

for (int i=0; i < lista.size(); i++){
      System.out.println(lista.get(i));
}
Ejemplo1
```

```
Sin embargo, desde java 1.5 existe una manera más eficaz (siempre que sea
posible)

for (Integer elem: lista){
      System.out.println(elem);
}
Ejemplo2: Menor acoplamiento de código
```

De hacer lo mismo sin depender de que conozcamos como es la estructura en especial (menor acoplamiento de código).

Más adelante, cuando veamos objetos, veremos cómo hacer que un TAD soporte esta sintaxis.