

Tipos Abstractos de Datos

Algoritmos y Estructuras de Datos

20/9/2023

Introducción

- Qué son los TADs? Qué recuerdan de intro?
- Por qué **tipos de datos**? Por qué **abstractos**?
- Qué TADs recuerdan de intro?

Introducción

- Vamos a definir **qué** hacen los TADs con nuestro lenguaje de especificación
 - Qué operaciones tienen
 - Qué efectos tienen esas operaciones (requiere/asegura)
- Más tarde vamos a diseñar módulos que definan **cómo** lo hacen
 - Eligiendo estructuras de datos
 - Escribiendo algoritmos

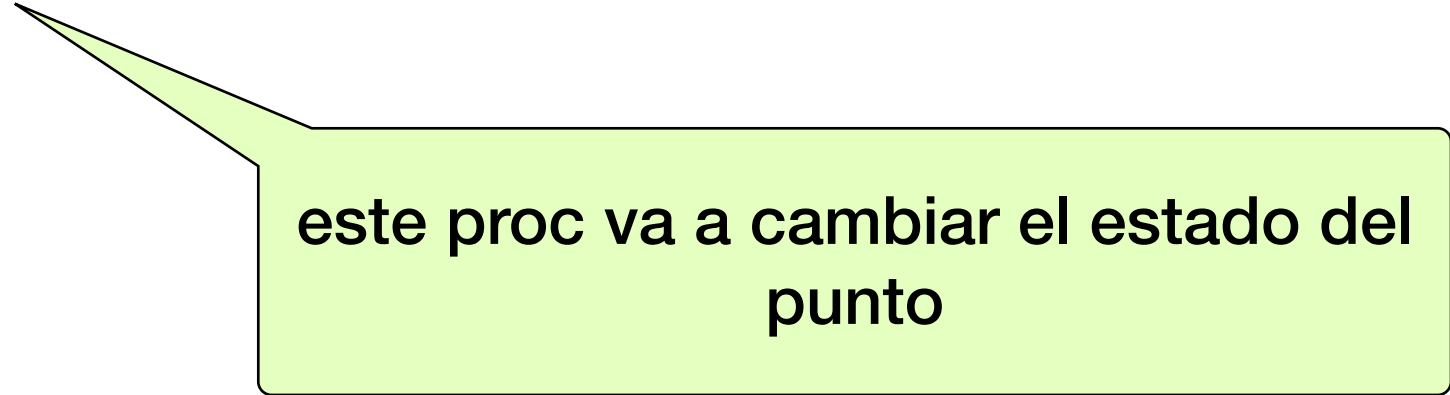
Introducción

- Un mismo TAD va a poder implementarse de diferentes maneras (**por qué? para qué?**)
- Un programa que usa el TAD va a poder elegir cualquier implementación y va a seguir andando
- En java: `interface` y `class`

Introducción

- Programamos en un lenguaje **imperativo**
- Las instancias de los TADs tienen **estado** y las operaciones pueden cambiar ese estado

```
proc mover(inout p: Punto, in deltaX: float, in deltaY: float)
```



este proc va a cambiar el estado del punto

Introducción

- Para facilitar la escritura vamos a permitir una sintaxis ascii de las especificaciones (ver apunte)
- Se pueden usar ambas

$$I : \{ (0 \leq i \leq |s|) \wedge_L res = true \leftrightarrow (\forall k : \mathbb{Z})(0 \leq k < i \rightarrow_L s[k] \neq 7) \}$$

`0 <= i <= |s| &&L res = true <==> forall k: int :: 0 <= k < i ==>L s[k] != 7`

LaTeX	ascii
T, F	<code>true, false</code>
$\wedge, \vee, \neg, \rightarrow, \leftrightarrow$	<code>&&, , !, ==>, <==></code>
$\wedge_L, \vee_L, \rightarrow_L$	<code>&&L, L, ==>L</code>

LaTeX	ascii
$(\forall i : T)(P(i))$	<code>forall i: T :: P(i)</code>
$(\exists i : T)(P(i))$	<code>exists i: T :: P(i)</code>
$(\sum_{P(i)})(f(i))$	<code>sum i: T :: P(i) :: f(i)</code>
$(\sum_{i=0}^n)(f(i))$	<code>sum i: T :: lo <= i <hi :: f(i)</code>

Anatomía de un TAD

Primero le damos un nombre

TAD Punto {

- Si es genérico, le definimos los parámetros de tipo

TAD Conjunto<T>

TAD Diccionario<K, V>

}

Anatomía de un TAD

TAD Punto {

Ahora indicamos qué operaciones tiene

- Algunas van a crear nuevas instancias
- Otras van a devolver algún dato
- Otras van a modificar la instancia
- Por convención el primer parámetro va a ser la propia instancia (in o inout, según corresponda)

```
proc nuevoPunto(in x: float, in y: float): Punto
```

```
proc coordX(in p: Punto): float
```

```
proc coordY(in p: Punto): float
```

```
proc coordTheta(in p: Punto): float
```

```
proc coordRho(in p: Punto): float
```

```
proc mover(inout p: Punto, in deltaX: float, in deltaY: float)
```

```
}
```


Anatomía de un TAD

- Cómo explicamos qué hacen las operaciones?

Observadores

```
TAD Punto {  
  obs x: float  
  obs y: float  
  
  proc nuevoPunto(in x: float, in y: float): Punto  
  
  proc coordX(in p: Punto): float  
  
  proc coordY(in p: Punto): float  
  
  proc coordTheta(in p: Punto): float  
  
  proc coordRho(in p: Punto): float  
  
  proc mover(inout p: Punto, in deltaX: float, in deltaY: float)  
  
}
```

- Permiten distinguir instancias
- Permiten explicar las operaciones
- Puede haber diferentes alternativas

Observadores

- Los observadores nos dan una idea de cómo se puede identificar una instancia de un TAD
- Permiten explicar las operaciones

```
obs x: float  
obs y: float
```

```
proc mover(inout p: Punto, in deltaX: float, in deltaY: float)  
  asegura ... p.x == old(p).x + deltaX  $\wedge$  p.y == old(p).y + deltaY
```

Cuánto valen los observadores a la salida respecto de lo que valían a la entrada?

- Si no puedo explicar una operación es que me faltan observadores
- Si un observador no se usa en las operaciones, es que está de más
- Los observadores dependen de las operaciones

Observadores

- Los observadores permiten distinguir dos instancias del TAD
 - Si sus observadores son iguales, al aplicarle cualquier operación el resultado debería ser el mismo

```
proc nuevoPunto(in x: float, in y: float): Punto
proc coordX(in p: Punto): float
proc coordY(in p: Punto): float
proc coordTheta(in p: Punto): float
proc coordRho(in p: Punto): float
proc mover(inout p: Punto, in deltaX: float, in deltaY: float)
```

obs x: float

No alcanza, no puedo
calcular Y, Theta ni Rho

obs x: float
obs y: float

Parecería que está bien

obs rho: float
obs theta: float

También parecería que está
bien

obs x: float
obs y: float
obs rho: float
obs theta: float

Es redundante, sobran
cosas!

Observadores

- Qué podemos usar como observadores?

Se agrega

- bool
 - int
 - float (o real)
 - char
- seq<T>
 - tupla<T1, ..., Tn>
 - string
- conj<T>
 - dict<K, V>
 - struct<n1: T1, ..., nn: Tn>

conj<T>: conjunto de tipo T .

Operación	LaTeX	ascii
crear	$\{\}, \{x, y, z\}$	$\{\}, \{x, y, z\}$
tamaño	$ c , length(c)$	$ c , c.length$
pertenece	$i \in c$	$i \text{ in } c$
union	$c_1 \cup c_2$	$c1 + c2$
intersección	$c_1 \cap c_2$	$c1 * c2$
diferencia	$c_1 - c_2$	$c1 - c2$

dict<K, V>: diccionario que asocia claves de tipo K con valores de tipo V .

Operación	LaTeX	ascii
crear	$\{\}, \{“juan” : 20, “diego” : 10\}$	$\{\}, \{“juan”: 20, “diego”: 10\}$
tamaño	$ d , length(d)$	$ d , d.length$
pertenece (hay clave)	$k \in d$	$k \text{ in } d$
valor	$d[k]$	$d[k]$

struct<campo1: T1, ..., campon: Tn>: tupla con nombres para los campos.

Operación	LaTeX	ascii
crear	$\langle x : 20, y : 10 \rangle$	$\langle x: 20, y: 10 \rangle$
campo	s_x, s_y	$s.x, s.y$

Observadores

- Qué podemos usar como observadores?

- bool
- int
- float (o real)
- char

- seq<T>
- tupla<T1, ..., Tn>
- string

- conj<T>
- dict<K, V>
- struct<n1: T1, ..., nn: Tn>

- Otros TADs

```
TAD Persona {  
  obs nombre: string  
  obs dirección: string  
  obs edad: int  
  
}
```

```
TAD Personal {  
  obs p: conj<Persona>  
}
```

- Funciones

```
TAD Conjunto<T> {  
  obs pertenece(e: T): bool  
}
```

Anatomía de un TAD

```
TAD Punto {  
  obs x: float  
  obs y: float  
  
  proc nuevoPunto(in x: float, in y: float): Punto  
    asegura res.x == x && res.y == y  
  
  proc coordX(in p: Punto): float  
    asegura res == p.x  
  
  proc coordY(in p: Punto): float  
    asegura res == p.y  
  
  proc coordTheta(in p: Punto): float  
    asegura res == safearctan(p.x, p.y)  
  
  proc coordRho(in p: Punto): float  
    asegura res == sqrt(p.x ** 2 + p.y ** 2)  
  
  proc mover(inout p: Punto, in deltaX: float, in deltaY: float)  
    asegura p.x == old(p).x + deltaX && p.y == old(p).y + deltaY  
  
  aux safearctan(x: float, y: float): float  
    if x == 0 then 0 else arctan(y/x) fi  
}
```

Podemos definir aux y pred

Anatomía de un TAD

- Hace falta un proc *coordX* si ya hay un observador *x*?

```
TAD Punto {  
  obs x: float  
  obs y: float  
  
  proc nuevoPunto(in x: float, in y: float): Punto  
    asegura res.x == x && res.y == y  
  
  proc coordX(in p: Punto): float  
    asegura res == p.x  
  
  proc coordY(in p: Punto): float  
    asegura res == p.y  
  
  proc coordTheta(in p: Punto): float  
    asegura res == safearctan(p.x, p.y)  
  
  proc coordRho(in p: Punto): float  
    asegura res == sqrt(p.x ** 2 + p.y ** 2)  
  
  proc mover(inout p: Punto, in deltaX: float, in deltaY: float)  
    asegura p.x == old(p).x + deltaX && p.y == old(p).y + deltaY  
  
  aux safearctan(x: float, y: float): float  
    if x == 0 then 0 else arctan(y/x) fi  
  
}
```

Preguntas??

Primer ejemplo: cola

```
TAD Cola<T> {
```

- Qué hace una cola?

```
}
```

Primer ejemplo: cola

```
TAD Cola<T> {
```

- Qué operaciones tiene?

```
}
```

Primer ejemplo: cola

```
TAD Cola<T> {
```

- Con qué se puede observar?

```
proc colaVacía(): Cola<T>
```

- `seq<T>`
- `conj<T>`
- `dict<K, V>`

```
proc vacia(in c: Cola<T>): bool
```

```
proc encolar(inout c: Cola<T>, in e: T)
```

```
proc desencolar(inout c: Cola<T>): T
```

```
}
```

Primer ejemplo: cola

- pre/post

```
TAD Cola<T> {  
  obs s: seq<T>  
  
  proc colaVacía(): Cola<T>  
    asegura res.s == []  
  
  proc vacia(in c: Cola<T>): bool  
    asegura res <==> c.s == []  
  
  proc encolar(inout c: Cola<T>, in e: T)  
    asegura c.s == old(c).s + [e]  
  
  proc desencolar(inout c: Cola<T>): T  
    requiere c.s != []  
    asegura c.s == tail(old(c).s)  
    asegura res == head(old(c).s)  
}
```

Ejemplo: TAD VideoClub

- Las personas se asocian al VideoClub
- El VideoClub tiene un conjunto de películas y puede comprar pelis nuevas
- Los socios pueden alquilar películas y devolverlas

Ejemplo: TAD VideoClub

- Descomponemos el problema en problemas más chicos
- Definimos TADs para los tipos más chicos y los componemos
 - Pelicula
 - Socio
 - VideoClub

Ejemplo: TAD VideoClub

- Qué operaciones tiene el VideoClub?
 - alquilar
 - devolver
 - asociarse
 - comprarPeli

Ejemplo: TAD Videoclub

- Qué cosas observamos?
 - Clientes
 - Películas
 - Alquileres
- Qué usamos para cada una?
 - Clientes
 - No hay repetidos (cada persona puede estar una sola vez)
 - No nos importa el orden
 - Se identifica por DNI o Nombre completo

`conj<Persona>`, donde `Persona` es `string`

Ejemplo: TAD Videoclub

- Qué usamos para cada una?
 - Películas
 - No hay repetidos (para simplificar, vamos a modelar que cada película está una sola vez)
 - No nos importa el orden

`conj<Película>, donde Película es string`

- Y si cada película puede tener varias copias? Cómo harían?

Ejemplo: TAD Videoclub

- Alquileres
 - `dict<Persona, conj<Película>>`
 - **Por qué no `dict<Película, Persona>`?**
 - Podría ser, no hay un motivo

Ejemplo: TAD Videoclub

```
TAD VideoClub {  
  obs pelis: conj<Peli>  
  obs clientes: conj<Cliente>  
  obs alquileres: dict<Peli, Cliente>  
  
  proc nuevoVideoClub(): VideoClub  
    asegura res.pelis = {}  
    asegura res.clientes = {}  
    asegura res.alquileres = {}  
  
  proc asociarse(inout v: VideoClub, in cli: Cliente)  
    requiere !(cli in v.clientes)  
    asegura v.clientes == old(v).clientes + {cli}  
    asegura v.pelis == old(v).pelis && v.alquileres == old(v).alquileres  
  
  proc comprarPeli(inout v: VideoClub, in peli: Pelicula)  
    requiere !(peli in v.pelis)  
    asegura v.pelis == old(v).pelis + {peli}  
    asegura clientes == old(v).clientes && v.alquileres == old(v).alquileres  
  
  ...  
}
```

hay que decir qué pasa con
todos los observadores!

Ejemplo: TAD Videoclub

```
TAD VideoClub {
  obs pelis: conj<Peli>
  obs clientes: conj<Cliente>
  obs alquileres: dict<Peli, Cliente>

  proc alquilar(inout v: VideoClub, in cli: Cliente, in p: Peli)
    requiere cli in v.clientes && p in v.pelis
    requiere !(p in v.alquileres)
    asegura v.alquileres[p] == cli
    asegura forall p': Peli :: p' in old(v).alquileres ==>
      p' in v.alquileres &&L
      v.alquileres[p'] == old(v).alquileres[p']
    asegura forall p': Peli :: p' in v.alquileres && p' != p ==>
      p' in old(v).alquileres
    asegura v.pelis == old(v).pelis && v.clientes == old(v).clientes

  proc devolver(inout v: VideoClub, in p: Peli)
    requiere p in v.alquileres
    asegura !(p in v.alquileres)
    asegura forall p': Peli :: p' in old(v).alquileres && p' != p ==>
      p' in v.alquileres &&L
      v.alquileres[p'] == old(v).alquileres[p']
    asegura forall p': Peli :: p' in v.alquileres ==>
      p' in old(v).alquileres
    asegura v.pelis == old(v).pelis && v.clientes == old(v).clientes
```