

Especificación de TADs

Qué es un TAD

- TAD quiere decir Tipo Abstracto de Datos
- ¿Qué es un Tipo Abstracto de Datos?
- Es un *tipo de datos* porque define un *conjunto de valores* y las *operaciones* que se pueden realizar sobre ellos
- Es *abstracto* ya que para utilizarlos, no se necesita conocer los detalles de la representación interna ni cómo están implementadas sus operaciones.
- No conocemos “la forma” de los valores
- Describe el “qué” y no el “cómo”
- Son una forma de *modularizar* a nivel de los datos
- ¿Qué TADs recuerdan de IP?

Qué es un TAD - Ejemplo

- El TAD *conjunto* es una abstracción de un conjunto matemático, que “contiene” “cosas” (todas del mismo tipo), sus “elementos”.
- Hay operaciones para agregar y sacar elementos y para ver si algo está o no (pertenece). Se puede saber cuántos elementos tiene.
- El conjunto no tiene en cuenta repetidos: si en un conjunto de números agregamos el 1, el 5 y otra vez el 1, la cantidad de elementos será 2.

Qué es un TAD - Otro ejemplo

- El TAD *punto 2D* es una abstracción de un punto en el plano cartesiano.
- Se puede describir a partir de sus coordenadas cartesianas (x, y) o polares (ρ, θ) .
- Tiene operaciones para moverlo, rotarlo sobre el eje o alejarlo del centro, etc.

¿Qué caracteriza a un TAD?

- Un TAD tiene *instancias* que pertenecen a su conjunto de valores
- Un TAD tiene operaciones
 - para *crear* una nueva instancia
 - para *calcular* valores a partir de una instancia
 - para *modificar* el estado

Observadores

- El estado de una *instancia* de un TAD lo describimos a través de *variables* o *funciones de estado* llamadas *observadores*
- Podemos usar todos los tipos de datos del lenguaje de especificación (int, real, seq<T>, conj<T>, etc.)
- En un instante de tiempo, el estado de una instancia del TAD estará dado por el estado de todos sus observadores

Observadores - Ejemplo

TAD punto 2D

- El estado del TAD punto 2D puede ser dado por:
 - variables de estado para las coordenadas cartesianas
 - obs x: real
 - obs y: real
 - o, variables de estado para las coordenadas polares
 - obs rho: real
 - obs theta: real
 - ¡Pero no ambas!
- ¿Podríamos tener un solo observador (por ejemplo, una sola coordenada)?
 - No nos serviría, porque no se puede describir un punto del plano mediante una sola coordenada. No nos alcanza.

Observadores

- El conjunto de observadores tiene que ser *completo*. Tenemos que poder observar todas las características *que nos interesan* de las instancias.
- A partir de los observadores se tiene que poder distinguir si dos instancias son distintas
- Todas las operaciones tienen que poder ser descritas a partir de los observadores

Observadores – Otro ejemplo

TAD conjunto

- El estado del TAD conjunto puede ser:
 - una variable de tipo `conj<T>` (el conjunto de nuestro lenguaje de especificación)

`obs elems: conj<T>`

(Formalmente, las variables de estado pueden considerarse también funciones como

`obs elems(c: Conjunto<T>): conj<T>`)

- O, una función que, dado un elemento, indique si está o no está presente en el conjunto y otra que nos indique la cantidad de elementos

`obs esta(e: T): bool`

Observadores

TAD conjunto

- OJO. Si usamos funciones como observadores, estas son funciones auxiliares de nuestro lenguaje de especificación, y por lo tanto
 - no pueden tener efectos colaterales ni modificar los parámetros
 - pueden usar tipos de nuestro lenguaje de especificación
 - pueden usar otros TADs
- Igualdad Observacional: decimos que dos instancias son iguales si todos sus observadores son iguales

Operaciones de un TAD

- Las operaciones del TAD indican qué se puede hacer con una instancia de un TAD
- Las especificamos con nuestro lenguaje de especificación
- Para indicar qué hacen, usamos precondiciones y postcondiciones (requiere y asegura)

```
proc agregar(inout c: conjunto<T>, in e: T)
    requiere ...
    asegura ...
```

- Para eso hablaremos del estado del TAD (o sea, del valor de sus observadores) antes y después de aplicar la operación.

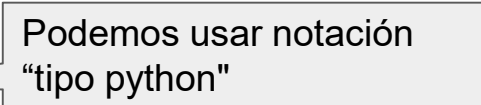
Operaciones de un TAD

```
TAD Conjunto<T> {  
    obs elems: conj<T>  
  
    proc agregar(inout c: conjunto<T>, in e: T)  
        asegura c == old(c) U {e}  
}
```

Para referirnos al valor inicial, usamos `old(c)`.
También podríamos usar c_0

Operaciones de un TAD

```
TAD Conjunto<T> {  
  obs esta(e: T): bool  
  
  proc agregar(inout c: conjunto<T>, in e: T)  
    asegura c.esta(e)  
    asegura forall e': T :: e' != e && old(c).esta(e') <==> c.esta(e')  
}
```



Podemos usar notación
"tipo python"

- Atención: para evitar la subespecificación, tenemos que describir el estado **completo** al salir de la función... Es decir, **cuánto valen TODOS los observadores**. (Pensar ejemplos de por qué sucede esto...)

Operaciones de un TAD

- **ATENCIÓN:** Los observadores son sólo para especificar. No son operaciones que se puedan usar en el código, en la implementación.
- Si queremos usarlas en el código tenemos que especificarlas también como operaciones...

```
TAD Conjunto<T> {  
    obs esta(e: T)
```

```
    proc pertenece(in c: Conjunto<T>, in e: T): bool  
        asegura res <==> c.esta(e)
```

```
}
```

recuerden que usamos *res*
para hablar del valor de retorno

TAD Conjunto

- Con observador elems

```
TAD Conjunto<T> {  
    obs elems: conj<T>
```

```
proc conjVacio(): Conjunto<T>  
    asegura res.elems == {}
```

```
proc agregar(inout c: Conjunto<T>, in e: T)  
    asegura c.elems == old(c).elems U {e}
```

```
proc sacar(inout c: Conjunto<T>, in e: T)  
    asegura c.elems == old(c).elems - {e}  
}
```

TAD Conjunto

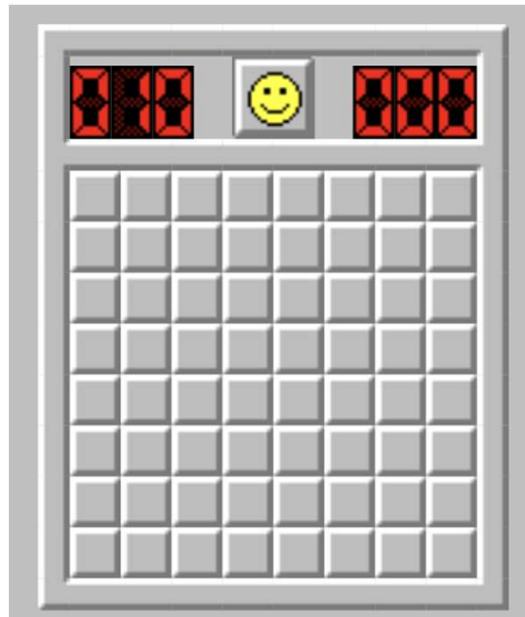
- Definición alternativa con observador esta

```
TAD Conjunto<T> {  
    obs esta(e: T): bool  
  
    proc conjVacio(): Conjunto<T>  
        asegura !exists e: T :: res.esta(e)  
  
    proc agregar(inout c: Conjunto<T>, in e: T)  
        asegura c.esta(e)  
        asegura forall e': T :: (e' != e) ==> (old(c).esta(e') <==> c.esta(e'))  
  
    proc sacar(inout c: Conjunto<T>, in e: T)  
        asegura !c.esta(e)  
        asegura forall e': T :: (e' != e) ==> (old(c).esta(e') <==> c.esta(e'))  
}
```


TAD Punto

```
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)  
  
    aux safearctan(x: float, y: float): float  
        if x == 0 then  $\pi/2$ *signo(y) else arctan(y/x)  
  
    proc mover(inout p: Punto, in deltaX: float, in deltaY: float)  
        asegura p.x == old(p).x + deltaX && p.y == old(p).y + deltaY  
}
```

Ejemplo: TAD Buscaminas



Ejemplo: TAD Buscaminas

- Principio: Descomponemos el problema en problemas más chicos
- Definimos TADs para los tipos más chicos y los componemos
 - TAD Tablero
 - TAD Juego

Ejemplo: TAD Buscaminas

```
TAD Tablero {
    obs minas: seq<seq<bool>>

    proc nuevoTablero(in filas: int, in cols: int): Tablero
        asegura |res.minas| == filas
        asegura forall i: int :: 0 <= i < filas ==>_L |res.minas[i]| == cols
        asegura forall i: int, j: int :: 0 <= i < filas && 0 <= j < cols
            ==>_L !res.minas[i][j]

    proc ponerMina(inout t: Tablero, in f: int, in c: int)
        asegura t.minas[f][c]
}
```

Ejemplo: TAD Buscaminas

```
TAD Juego {  
    obs tablero: Tablero  
    obs jugadas: seq<Pos>  
  
    proc nuevoJuego(in t: Tablero): Juego  
        asegura res.tablero == t  
  
    proc jugar(inout j: Juego, in p: Pos)  
        requiere !j.perdio() && !j.gano()  
        requiere !(p in j.jugadas)  
        asegura p in j.jugadas  
  
    pred perdio(j: Juego)  
        ...  
  
    pred gano(j: Juego)  
        ...  
}
```

Pos es tupla<int, int>