

# Diseño de TADs

## Repaso. Qué es un TAD?

- Un TAD (tipo abstracto de datos) es una abstracción que sirve para describir una parte de un problema
- Describe el qué y no el cómo
- Tiene estado
- Se manipula a través de operaciones, que describimos mediante un lenguaje de especificación (lógica) con pre y postcondición

# Diseño de TADs

- Un diseño de un TAD es una estructura de datos y una serie de algoritmos (en algún lenguaje de programación, real o simplificado) que nos indica *cómo* se representa *y* cómo se codifica una posible implementación del TAD
- Tendremos que elegir una *estructura* de representación con tipos *de datos*
- Tendremos que escribir *algoritmos* para todas las operaciones
- Los algoritmos deberán respetar la especificación del TAD
- Miren lo que está en *azul* en esta diapo ;-)

# Diseño de TADs

- ¡Puede haber muchos diseños para un TAD!
  - Porque dos personas lo pensaron de diferentes maneras
  - Porque hay requerimientos de eficiencia (memoria, tiempo de ejecución: [complejidad](#))
  - Perchè mi piace
- Mientras respeten la especificación, quien las use podrá elegir uno u otro diseño sin cambiar sus programas ([modularidad](#))

# Ocultando información

- Ventajas del ocultamiento, la abstracción y el encapsulamiento:
  - La implementación se puede cambiar y mejorar sin afectar su uso.
  - Ayuda a modularizar.
  - Facilita la comprensión.
  - Favorece el reuso.
  - Los módulos son más fáciles de entender.
  - Y de programar.
  - El sistema es más resistente a los cambios.



# Ocultando información

- Abstracción: “Abstraction is a process whereby we identify the important aspects of a phenomenon and ignore its details.” [Ghezzi et al, 1991]
- Information hiding: “The [...] decomposition was made using 'information hiding' [...] as a criterion. [...] Every module [...] is characterized by its knowledge of a design decision which it hides from all others. Its interface or definition was chosen to reveal as little as possible about its inner workings.” [Parnas, 1972b]
- “[...] the purpose of hiding is to make inaccessible certain details that should not affect other parts of a system.” [Ross et al, 1975]
- Encapsulamiento: “[...] A consumer has full visibility to the procedures offered by an object, and no visibility to its data. From a consumer's point of view, an object is a seamless capsule that offers a number of services, with no visibility as to how these services are implemented [...] The technical term for this is encapsulation.” [Cox, 1986]

# Diseño de un TAD

```
modulo PuntoImpl implementa Punto {  
    var rho: float  
    var theta: float  
}
```

Notar que especificamos con cartesianas y diseñamos con polares  
¿Podríamos hacer al revés?

- En la especificación nos referíamos a los valores del tipo a partir de los observadores. Aquí tenemos que definir los valores explícitamente a partir de una estructura.
- Los tipos de las variables de la estructura son tipos de implementación:

- int, float, char, ...
- tupla / struct (tupla con nombres)
- array<T>
- Módulos de otros TADs
- ¡y veremos muchísimas más!

OJO: son de tamaño fijo

Otro OJO, más grande: no usamos seq<T>, ni conj<T>

# El invariante de representación

```
modulo PuntoImpl implementa Punto {  
    var rho: float  
    var theta: float  
}
```

- ¡Momento! Tenemos (al menos) dos formas de almacenar el ángulo theta:
  - Normalizado (entre 0 y  $2\pi$ , o entre  $-\pi$  y  $\pi$ )
  - Desnormalizado (cualquier valor real)
- Vamos a elegir (por ahora) guardarlo normalizado



# El invariante de representación

- Entonces, no todos los posibles valores de las variables de estado representan un estado de Punto válido. Tenemos *restricciones*
- El *invariante de representación* es un predicado que nos indica qué conjuntos de valores son instancias válidas de la implementación.
- ¿Por qué invariante? Porque se tiene que cumplir siempre al entrar y al salir de todas las operaciones (similar (¡pero no igual!) al *invariante de ciclo*)
- Generalmente lo denotamos como `InvRep`
- Para cualquier operación del módulo, se tiene que poder verificar la siguiente tripla de Hoare:

$\{ \text{InvRep}(p') \} \text{ Operación}(p', \dots) \{ \text{InvRep}(p') \}$

Además de las que involucran la pre y la post de la operación...

## TAD punto: invariante de representación

- El invariante de representación se escribe en lógica (usando el lenguaje de especificación) haciendo referencia a la estructura de implementación.

```
modulo PuntoImpl implementa Punto {  
    var rho: float  
    var theta: float  
  
    pred InvRep(p': PuntoImpl) {  
        -pi <= p'.theta < pi  
    }  
}
```

## La función de abstracción

- Tenemos dos estructuras: el TAD y la implementación. ¿Cómo relacionamos ambas?
- La *función de abstracción* nos va a indicar, dada una instancia de implementación, a qué instancia del TAD corresponde, a qué instancia del TAD representa, qué instancia del TAD “es su abstracción”.
- La *función de abstracción* hace referencia a las variables de estado de la implementación y a los observadores del TAD (porque tiene que vincular unas con otros).
- Para definirla, se puede suponer que vale el invariante de representación.

# TAD punto: función de abstracción

- Como su nombre lo indica, es una función

```
aux FuncAbs(p': PuntoImpl): Punto {  
    p: Punto |  
    p.x == p'.rho * cos(p'.theta) &&  
    p.y == p'.rho * sin(p'.theta)  
}
```

- Pero también podemos escribirla como un predicado si nos resulta más cómodo. La siguiente formulación es equivalente a la anterior:

```
pred PredAbs(p': PuntoImpl, p: Punto) {  
    p.x == p'.rho * cos(p'.theta) &&  
    p.y == p'.rho * sin(p'.theta)  
}
```

# TAD punto: algoritmos

- ¡Listo! Sólo nos queda escribir los algoritmos:

```
proc mover(inout c': PuntoImpl, in deltaX: float, in deltaY: float)
{
    float nuevoX := c'.x + deltaX;
    float nuevoY := c'.y + deltaY;

    c'.rho := sqrt(nuevoX ** 2 + nuevoY ** 2);
    c'.theta := arctan(nuevoY / nuevoX); // aca hay un bug!!
}
```

- Reemplazamos el tipo abstracto por el de implementación

## TAD punto: algoritmos

- ¿Cómo sabemos que el algoritmo es correcto? Un avance:

```
{InvRep(c') && requiereTAD(FuncAbs(c'))}  
proc mover(inout c': PuntoImpl, in deltaX: float, in deltaY: float)  
    ...  
{InvRep(c') && aseguraTAD(FuncAbs(c'))}
```

- Ampliaremos...
- Mientras tanto, veamos otro ejemplo.

## TAD conjunto: diseño

- ¿Cómo diseñamos el conjunto?
- Podemos hacerlo con arrays
- Como son de tamaño fijo, con cada nuevo elemento que se agrega podríamos crear un nuevo array, copiar los elementos anteriores y el nuevo (¡MUY MALA IDEA! ¿Por qué?)
- Tener un entero que nos indique la cantidad de elementos del array que llevamos “usados”. Cuando el array se llena, ahí sí creamos uno nuevo más grande y copiamos el viejo.

# TAD conjunto: diseño

```
modulo ConjImpl<T> implementa Conjunto<T> {  
    var arr: array<T>  
    var largo: int  
}
```

- Tenemos que tomar una decisión más...
- ¿Qué pasa si agregamos al conjunto un mismo elemento dos veces?
  - Opción 1: buscamos en el arreglo, y si ya está, no lo insertamos. Llamamos a esta solución "arreglo sin repetidos"
  - Opción 2: lo agregamos directamente. Llamamos a esta solución "arreglo con repetidos"
  - ¿En qué podría afectarnos elegir una u otra?



# TAD conjunto: invariante de representación

- ¿Cómo sería el InvRep en cada caso?
- Con repetidos:

```
pred InvRep(c': ConjArrayRepe<T>)  
  {0 <= c'.largo <= c'.arr.Length}
```

- Sin repetidos:

```
pred InvRep(c': ConjArrayRepe<T>)  
  {0 <= c'.largo <= c'.arr.Length &&  
   forall i: int :: 0 <= i < c'.largo ==> apariciones(c', c'.arr[i]) == 1}
```

# TAD conjunto: función de abstracción

Con repetidos:

```
aux FunAbs(c': ConjImpl<T>): Conjunto<T>
{
    c: Conjunto<T> |
    forall e: T :: e in c.elems <==> e in c'.arr[0..c'.largo]
}
```

Sin repetidos:

```
aux FunAbs(c': ConjImpl<T>): Conjunto<T>
{
    c: Conjunto<T> |
    forall e: T :: e in c.elems <==> e in c'.arr[0..c'.largo]
}
```

# Bibliografía

- “Abstraction, Encapsulation, and Information Hiding”. By Edward V. Berard. The Object Agency.  
<http://www.tonymarston.net/php-mysql/abstraction.txt>
- [Parnas, 1972b] D.L. Parnas, “On the Criteria To Be Used in Decomposing Systems Into Modules,” Communications of the ACM, Vol. 5, No. 12, December 1972, pp. 1053-1058.
- [Ghezzi et al, 1991] C. Ghezzi, M. Jazayeri, and D. Mandrioli, Fundamentals of Software Engineering, Prentice-Hall, Englewood Cliffs, New Jersey, 1991.
- [Ross et al, 1975] D.T. Ross, J.B. Goodenough, and C.A. Irvine, “Software Engineering: Process, Principles, and Goals,” IEEE Computer, Vol. 8, No. 5, May 1975, pp. 17 - 27.
- [Cox, 1986] B.J. Cox, Object Oriented Programming: An Evolutionary Approach, Addison-Wesley, Reading, Massachusetts, 1986.
- [Hoare, 1972] C.A.R. Hoare. “Proof of correctness of Data Representation”. Acta Informatica 1(1), 1972.