Diseño de TADs

Algoritmos y Estructuras de Datos

Diseño de TADs

Invariante de Representación

Función de Abstracción

Ejemplos

- Conjunto Acotado sobre Array de Bools
- Conjunto con Array de Naturales

¿Qué es un TAD?

- Un TAD (tipo abstracto de datos) es una abstracción que describe 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 postcondiciones.

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 se codifica una 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 ;-)

Múltiples Diseños de un TAD

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

```
TAD Punto {
      obs x: ℝ
      obs y: ℝ
}
```

```
Modulo PuntoImpl implementa Punto {
      rho: float
      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.
- En la implementación 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$< T >$ ¡¡TAMAÑO FIJO!! (no son seq ni conj)
  - Módulos de otros TADs
  - ¡y veremos muchísimas más!

Diseño de TADs

Invariante de Representación

Función de Abstracción

Ejemplos

- Conjunto Acotado sobre Array de Bools
- Conjunto con Array de Naturales

Invariante de Representación

```
Modulo PuntoImpl implementa Punto {
     rho: float
     theta: float
}
```

- ¿Cómo va a funcionar nuestro módulo en el almacenamiento de los ángulos?
  - Normalizado (entre 0 y 2π o entre $-\pi$ y π)
  - Desnormalizado (cualquier valor real)
- Podemos elegir cualquiera, pero tenemos que tener consistencia

No todos los posibles valores de las variables de estado representan un estado de Punto válido.

Tenemos restricciones

Invariante de Representación

Invariante de representación: es un predicado que nos indica qué conjuntos de valores son instancias válidas de la implementación.

- Se tiene que cumplir siempre al entrar y al salir de todas las operaciones
- Generalmente lo denotamos como InvRep
- Para cualquier `proc x()` del módulo, se tiene que poder verificar la siguiente tripla de Hoare:

  $\{InvRep(p)\} \quad procx(p, ...) \quad \{InvRep(p)\}$
- Además de las que involucran la pre y la post del proc
- Se escribe en lógica, haciendo referencia a la estructura de implementación

```
Modulo PuntoImpl implementa Punto {
       rho: float
       theta: float
       pred invRep(p': PuntoImpl)
              {−π <= p'.theta < π} }
```

Diseño de TADs

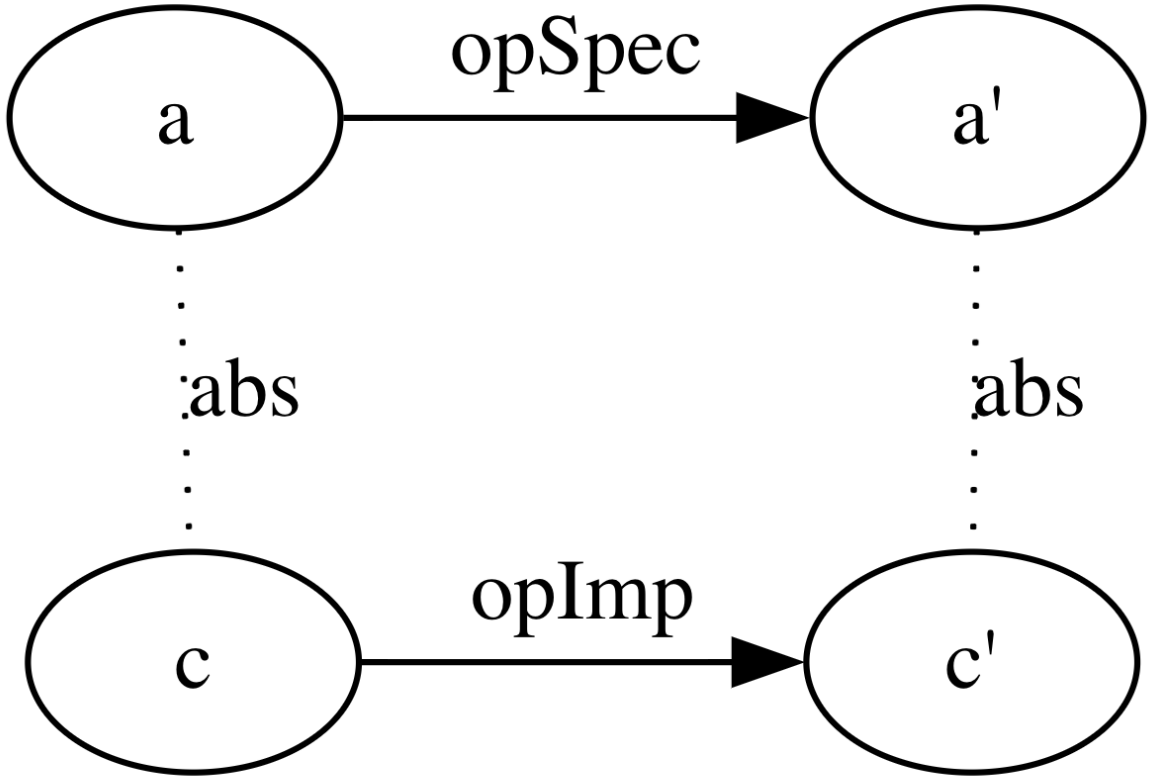
Invariante de Representación

Función de Abstracción

Ejemplos

- Conjunto Acotado sobre Array de Bools
- Conjunto con Array de Naturales

Correctitud



- Si la operación puede invocarse en el estado abstracto , también es ejecutable (y termina) en el estado concreto (de implementación)
- El diagrama de arriba conmuta (hay simulación por parte de la abstracción)

Función de Abstracción

```
TAD Punto {
      obs x: ℝ
      obs y: ℝ
}
```

```
Modulo PuntoImpl implementa Punto {
      rho: float
      theta: float
}
```

¿Cómo relacionamos el TAD con el módulo?

- **Función de abstracción:** nos va a indicar, dada una instancia de implementación, a qué instancia del TAD corresponde/representa, de qué instancia del TAD "es su abstracción"
- predAbs toma como parámetro una instancia del módulo y del TAD y evalua si son iguales
- Para definirla, se puede suponer que vale el invariante de representación

Función de Abstracción

A pesar de que se llama Función de Abstracción, lo vamos a escribir en formato de predicado:

```
Modulo PuntoImpl implementa Punto {
      rho: float
      theta: float

      pred invRep(p: PuntoImpl)
            {−π <= p.theta < π}

      pred predAbs(p: PuntoImpl, p':Punto)
            {p'.x = p.rho ∗ cos(p.theta)∧
             p'.y = p.rho ∗ sin(p.theta)}
}
```

Ahora sólo quedaría escribir los algoritmos.

Pero volvamos a los ejemplos de conjuntos que son más interesantes

Diseño de TADs

Invariante de Representación

Función de Abstracción

Ejemplos

- Conjunto Acotado sobre Array de Bools
- Conjunto con Array de Naturales

Conjunto Acotado sobre Array de Bools

```
TAD ConjAcotado<ℕ> {
  obs set: conj<ℕ>
  obs cota: ℕ

  proc conjVacío(in cota: ℕ):
ConjAcotado<ℕ>
    asegura{res.set = {} ∧ res.cota = cota}

  proc pertenece(in c: ConjAcotado<ℕ>,
in e: ℕ): bool
    asegura{res = true ↔ e ∈ c.set}

  proc agregar(inout c: ConjAcotado<ℕ>,
in e: ℕ)
    requiere{c = C0 ∧ e ≤ c.cota}
    asegura{c.set = C0.set ∪ ⟨e⟩}
}
```

```
Modulo ConjActadoSobreBitArray<Bool> implementa
ConjAcotado<ℕ> {
  elems: Array<Bool>

  proc conjVacío(in cota: Int): CASBA<Int>
    Creo un Array de cota elementos, todos en False

  proc pertenece(in c: CASBA, in e: Int): bool
    Me fijo el estado de la posición e.

  proc agregar(inout c: CASBA, in e: Int)
    Me fijo el estado de la posición e.
    Si no lo está (False), lo cambio.

}
```

- ¿Qué restricciones tenemos para esta representación de conjunto acotado?

  → Tenemos que hacer el invRep
- ¿Cómo se relaciona la implementación con el TAD?

  → Tenemos que hacer el predAbs

Conjunto Acotado sobre Array de Bools

Invariante de Representación:

- No tenemos restricciones importantes.
- Solo que las posiciones del Array estén definidas
- Más adelante vamos a dejar de pedir esto

```
pred invRep(c: CASBA){
```
$(\forall i : \mathbb{Z})(0 \leq i < \mathit{length}(c.\mathit{elems}) \rightarrow_L \mathit{def}(c.\mathit{elems}[i]))\}$

Función de Abstracción:

- Tenemos que relacionar la variable *elems* del módulo con los observadores del TAD

```
pred abs(c:CASBA, c':ConjuntoAcotado)
```
$\{$`c'.cota = length(c.elems)-1` $\wedge$
$(\forall n : \mathbb{N})($ `n`$\in$`c'.set` $\Leftrightarrow$
`(n`$\leq$`length(c.elems)-1` $\wedge_L$ `c.elems[n] = True)` $\}$

Y con esto ya podemos empezar a programar los procs

Conjunto Acotado sobre Array de Bools

```
TAD ConjAcotado<𝕀N> {
  obs set: conj<𝕀N>
  obs cota: 𝕀N

  proc conjVacío(in cota: 𝕀N):
ConjAcotado<𝕀N>
    asegura{res.set = {} ∧ res.cota = cota}

  proc pertenece(in c: ConjAcotado<𝕀N>,
in e: 𝕀N): bool
    asegura{res = true ↔ e ∈ c.set}

  proc agregar(inout c: ConjAcotado<𝕀N>,
in e: 𝕀N)
    requiere{c = C0 ∧ e ≤ c.cota}
    asegura{c.set = C0.set ∪ ⟨e⟩}
}
```

```
Modulo ConjActadoSobreBitArray<Bool> implementa
ConjAcotado<𝕀N> {
  elems: Array<Bool>

  pred invRep(c: CASBA){
    (∀i : ℤ)(0 ≤ i < length(c.elems) →L def(c.elems[i]))}

  pred abs(c:CASBA, c':ConjuntoAcotado)
    {c'.cota = length(c.elems)-1 ∧
    (∀n : ℕ)( n∈ c'.set ⇔
      (n≤length(c.elems)-1 ∧L c.elems[n] = true) }

  proc conjVacío(in cota: Int): CASBA
    res.elems = NewArray[cota+1][False];
    return res

  proc pertenece(in c: CASBA, in e: Int): bool
    if e<length(c.elems):
     res := c.elems[e]
    else:
     res := False
    return res

  proc agregar(inout c: CASBA, in e: Int)
    c.elems[e] := true
    return

}
```

- ¿Cómo impactan los requiere y asegura del TAD en los códigos del TAD?
- Cada proc del módulo tiene que cumplir con los requiere y asegura del TAD
- Y el invRep
- Para eso hay que "traducirlos" al "lenguaje" del módulo
- Se hace via la función de abstracción

Conjunto Acotado sobre Array de Bools

```
proc conjVacío(in cota: ℕ):
ConjAcotado<ℕ>
  asegura{res.set = {} ∧ res.cota = cota}
```

```
pred invRep(c: CASBA){
     (∀i : ℤ)(0 ≤ i < length(c.elems) →L def(c.elems[i]))}
pred abs(c:CASBA, c':ConjuntoAcotado)
     {c'.cota = length(c.elems)-1 ∧
     (∀n : ℕ)( n∈ c'.set ⇔
       (n≤length(c.elems)-1 ∧L c.elems[n] = true) }
```

```
proc conjVacío(in cota: Int): CASBA
      requiere {True}
      res.elems = NewArray[cota+1][False];
      return res
      asegura {length(res)= cota+1 ∧
      (∀i : ℤ)(0≤i<length(res) →L res[i]=False)}
```

Notar que es obvio que True ⇒ Wp(asegura, código) por semántica axiomática asumida del New de Array (crea un arreglo de length establecida y lo setea todo en el valor establecido). Eso sí, esto es caro en términos de tiempo: O(cota).

Conjunto Acotado sobre Array de Bools

```
proc pertenece(in c: ConjAcotado<ℕ>,
in e: ℕ): bool
  asegura{res = true ↔ e ∈ c.set}
```

```
pred invRep(c: CASBA){
     (∀i : ℤ)(0 ≤ i < length(c.elems) →_L def(c.elems[i]))}
pred abs(c:CASBA, c':ConjuntoAcotado)
     {c'.cota = length(c.elems)-1 ∧
     (∀n : ℕ)( n∈ c'.set ⇔
       (n≤length(c.elems)-1 ∧_L c.elems[n] = true) }
```

```
proc pertenece(in c: CASBA, in e: Int): bool
        requiere { invRep(c) }
        if e<length(c.elems):
            res := c.elems[e]
        else:
            res := False
        return res
```

Tenemos que pasar el asegura del TAD al Módulo: `(res = True ↔ (e∈ c'.set))`
`asegura {invRep(c) ∧_L (res = True ↔ (e<length(c.elems)`
`∧_L c.elems[e]=True) )}`

Siempre vamos a utilizar el Abs para sacarnos todas las apariciones del TAD (c' en este caso) y reemplazarlas por lo que corresponda del módulo

Conjunto Acotado sobre Array de Bools

```
proc agregar(inout c: ConjAcotado<ℕ>,
in e: ℕ)
  requiere{c = C0 ∧ e ≤ c.cota}
  asegura{c.set = C0.set ∪ ⟨e⟩}
```

```
pred invRep(c: CASBA){
    (∀i : ℤ)(0 ≤ i < length(c.elems) →L def(c.elems[i]))}
pred abs(c:CASBA, c':ConjuntoAcotado)
    {c'.cota = length(c.elems)-1 ∧
    (∀n : ℕ)( n∈ c'.set ⇔
      (n≤length(c.elems)-1 ∧L c.elems[n] = true) }
```

```
proc agregar(inout c: CASBA, in e: Int)
      requiere { InvRep(c) ∧L e<length(bsbv.elems) ∧ c=C0 }
      c[e] := true
      return
      asegura {InvRep(c) ∧L (∀i : ℤ)(0 ≤ i < length(c.elems) →L
        (i ≠ e ∧ c.elems[i] = C0.elems[i]) ∨ (i = e ∧ c.elems = True)}
```

- Asumimos que los parámetros *in* no son modificados durante la ejecución
- Para asegurarlo formalmente habría que inicializar metavariables en el requiere y usarlas en el asegura

Algunas observaciones

En resumen:

- Escribir el invRep permite que nuestra estructura de representación no se rompa
- El invRep se tiene que cumplir siempre antes y despues de cada proc
- Escribir el predAbs nos permite conectar con el TAD
- En base al invRep, el predAbs, el TAD y nuestras implementaciones vamos a tener **requiere** y **asegura**
- Los **asegura** de los procs del módulo pueden ser más fuertes que los de la especificación.
- Van a reflejar las decisiones de diseño
- ¡Por esto es importante no sobreespecificar!

Conjunto con Array de Naturales

- ¿Qué pasa si el conjunto no está acotado?
- 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?)
- O 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.

Conjunto con Array de Naturales

```
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?

Conjunto con Array de Naturales - 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 ∧
(∀ i: int ) 0 ≤ i < c'.largo→L
apariciones(c', c'.arr[i]) == 1 }
```

Conjunto con Array de Naturales - Función de Abstracción

- Con repetidos:

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

- Sin repetidos:

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

- ¡Qué casualidad, son iguales!

Bibliografía

1 Berard, E. V. “Abstraction, Encapsulation, and Information Hiding”.

2 Parnas, D.L. “On the Criteria To Be Used in Decomposing Systems Into Modules”, 1972.

3 Ghezzi, C., Jazayeri, M., Mandrioli, D. “Fundamentals of Software Engineering”, 1991.

4 Ross, D.T., Goodenough, J.B., Irvine, C.A. “Software Engineering: Process, Principles, and Goals”, 1975.

5 Cox, B.J. “Object-Oriented Programming: An Evolutionary Approach”, 1986.