Tema 5: Abstracción

Índice

En el primer tema de la asignatura vimos que una de las características principales de los lenguajes de programación es que proporcionan herramientas para la construcción de abstracciones.

Una de los trabajos fundamentales del diseñador de software es analizar el problema a resover y definir abstracciones que sean útiles. En muchas ocasiones el proceso de abstracción conlleva un proceso de prueba y error en el que la abstracción se va construyendo mediante la generalización de unas primeras ideas demasiado concretas.

En este tema vamos a ver en primer lugar algunos ejemplos del proceso de generalización y después hablaremos del concepto de la barrera de abstracción.

1. Generalización de funciones

La generalización es un proceso mediante el que el diseñador de software detecta un un conjunto de funciones que implementan pequeñas variaciones de un patrón común con pequeñas variaciones y crea una única abstracción que implementa el patrón y engloba a todas las funciones. Para generalizar el patrón se define una función que permite hacer todo lo que hacían las funciones específicas, utilizando argumentos para determinar esas variaciones.

La generalización permite expresar el patrón general que se repite en todas las funciones concretas.

Quedará más claro cuando veamos algunos ejemplos.

1.1. Ejemplo 1

Supongamos las siguientes funciones específicas:

```
(define (cuadrado x)
   (* x x))

(define (cubo x)
   (* x x x))

(define (a-la-cuarta x)
   (* x x x x))
```

¿Cómo generalizaríamos esta idea? ¿Qué patrón común tienen todas estas funciones?

```
(define (elevado-a x y)
   (if (= y 0)
      1
      (* x (elevado-a x (- y 1)))))
```

Hemos generalizado las funciones específicas `(cuadrado x)`, `(cubo x)` y `(a-la-cuarta x)` mediante una función que utiliza como argumento adicional el número de veces que multiplicamos `x` por si mismo: `(elevado-a x y)`. Todas las funciones específicas las podemos definir concretando el argumento `y`:

- `(cuadrado x)` = `(elevado-a x 2)`
- `(cubo x)` = `(elevado-a x 3)`
- `(a-la-cuarta x)` = `(elevado-a x 4)`

Además, la función general permite cualquier otra concreción del patrón, no se limita a 2, 3 o 4.

1.2. Ejemplo 2

Veamos ahora un patrón en el que interviene una función.

Supongamos las siguientes funciones específicas:

```
(define (sum-cuadrado a b)
  (if (> a b)
      0
      (+ (cuadrado a) (sum-cuadrado (1+ a) b)) ))

(define (sum-cubo a b)
  (if (> a b)
      0
      (+ (cubo a) (sum-cubo (1+ a) b)) ))
```

La primera función implementa el sumatorio de la *función cuadrado* y la segunda el sumatorio de la *función cubo*.

¿Cuál es el patrón común a ambas? ¿Qué podemos extraer como argumento para generalizar el patrón común?

Ambas funciones realizan el sumatorio de *una función*. Lo podemos implementar así:

```
(define (sum f a b)
  (if (> a b)
      0
      (+ (f a) (sum f (1+ a) b)) ))
```

Las funciones específicas las podemos definir concretando el argumento `f`:

- `(sum-cuadrado a b)` = `(sum cuadrado a b)`b
- `(sum-cubo a b)` = `(sum cubo a b)`

1.3. Ejemplo 3

Veamos un último ejemplo.

Las funciones específicas:

```
(define (pares nums)
  (cond ((empty? nums) '())
        ((= (remainder (car nums) 2) 0)
            (append (list (car nums)) (pares (cdr nums))) )
        (else (pares (cdr nums)))))

(define (epals frase)
  (cond ((empty? frase) '())
        ((member? 'e (car frase))
            (append (list (car frase)) (epals (bf frase))) )
        (else (epals (cdr frase)))))

(define (nombres-propios frase)
  (cond ((empty? frase) '())
        ((member? (car frase) '(Ana Lucia Carlos Raquel))
            (append (list (car frase)) (nombres-propios (cdr frase)))
)
        (else (nombres-propios (cdr frase)))))
```

Su generalización:

```
(define (par x)
   (= (remainder x 2) 0))

(define (epal pal)
   (member? 'e pal))

(define (nombre-propio pal)
   (member? pal '(Ana Lucia Carlos Raquel)))

(define (filtra pred lista)
  (cond ((null? lista) '())
        ((pred (car lista))
            (append (list (car lista)) (filtra pred (cdr lista))))
        (else (filtra pred (cdr lista)))))
```

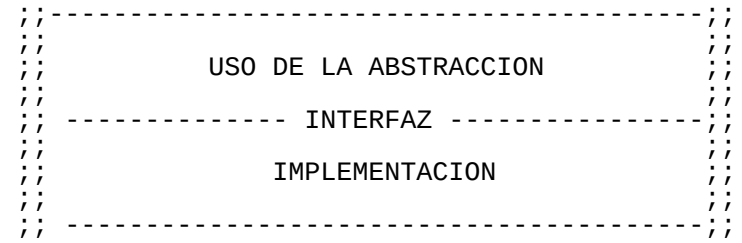
Las funciones específicas las podemos obtener concretando el argumento `pred`:

- `(pares nums)` = `(filtra par nums)`
- `(epals frase)` = `(filtra epal frase)`
- `(nombres-propios frase)` = `(filtra nombre-propio frase)`

2. Barreras de abstracción

Un concepto fundamental para la construcción de abstracciones es el concepto de *barrera de abstracción* o *interfaz*.

Una barrera de abstracción (o interfaz) define un conjunto de funciones que separan la especificación de la implementación. La funciones que estén *por encima* de la barrera de abstracción deben usar las funciones propuestas por esta barrera. De esta forma *se esconde la implementación* y se independiza las funciones que usan la abstracción de su implementación.



Este proceso recibe también el nombre de *encapsulación* y es fundamental para el diseño de software que sea *reusable* y *mantenible*.

La barrera de abstracción proporciona las siguientes ventajas:

- Define un conjunto de funciones estándar para acceder a un tipo de datos definido por el usuario.
- Permite modificar la implementación de lo que hay detrás de la barrera sin que el resto del sistema se vea afectado.

Un concepto íntimamente relacionado con el de barrera de abstracción es el de *tipos abstractos de datos*. Un tipo abstracto de datos se define mediante un conjunto de *constructores* que nos permiten definir nuevos datos de ese tipo y un conjunto de *selectores* y *operadores* que nos permiten acceder y modificar propiedades de esos datos.

La mayoría de lenguajes de programación de alto nivel tienen mecanismos para definir barreras de abstracción:

- Java: definición de clases, clases abstractas e interfaces
- C++: definición de clases y clases abstractas

Scheme no tiene estos mecanismos y lo tenemos que hacer "a mano". Vamos a ver unos par de ejemplos de construcción de tipos de datos utilizando un conjunto de funciones de Scheme que hacen de barrera de abstracción.

Por ejemplo, supongamos que estamos escribiendo un programa para jugar a las cartas y que en un momento dado necesitamos una función que calcule la puntuación total de una mano de cartas. Supongamos que escribimos el siguiente código:

```
(define (total mano)
```

```
  (if (equal? mano '())
      0
      (+ (butlast (car mano))
         (total (cdr mano)) )))

(total '(3o 10c 4b))
17
```

Esta definición de la función `total` es absolutamente críptica. ¿Cuál es el problema de fondo? Fíjate en que para entender cómo se calcula el valor total de la mano de cartas debemos entender cómo está implementada la mano de cartas. Si miramos en el ejemplo en el que se llama a la función total nos damos cuenta de que una mano de cartas es una lista de identificadores. Y también nos damos cuenta de que los identificadores están formados por un número que representa el valor de la carta y un carácter que representa su palo. Así, por ejemplo, `'12c` representa el 12 de copas o `'3o` representa el 3 de oros.

Ahora ya se puede entender un poco mejor el código: con `(car mano)` se obtiene el primer elemento de la lista `mano` y con `butlast` se quita el último carácter para quedarnos sólo con el número de la primera carta. Y después se hace una llamada recursiva para que se calcule el total del resto de la mano. Pero aún así el código sigue siendo poco legible, porque se incumple una regla fundamental de la programación:

Aviso:

REGLA DE PROGRAMACIÓN
Debemos usar la abstracción para crear tipos de datos. Un tipo de datos cumple dos funciones: acercar el programa al dominio que estamos programando y ocultar la implementación con una barrera de abstracción.

En este caso, el dominio es el juego de cartas. Podemos entonces usar la abstracción para definir las funciones `valor` y `palo` del tipo de dato carta:

```
(define (valor carta)
   (butlast carta))

(define (palo carta)
   (cond
      ((equal? (last carta) 'o) 'oros)
      ((equal? (last carta) 'c) 'copas)
      ((equal? (last carta) 'e) 'espadas)
      ((equal? (last carta) 'b) 'bastos)
      (else (error "carta mal definida:" carta))))
```

Y podemos definir las funciones una-carta, resto-cartas y mano-sin-cartas? del tipo de datos mano:

```
(define (una-carta mano)
   (car mano))

(define (resto-cartas mano)
   (cdr mano))

(define (mano-sin-cartas? mano)
   (equal? mano '())
```

Acabamos de crear una barrera de abstracción. Si usamos las funciones valor, palo, una-carta, resto-cartas y mano-sin-cartas? nos estamos acercando más al dominio del programa (juego de cartas) y no nos ligamos a una implementación concreta de nuestras estructuras de datos. La función total quedaría como sigue:

```
(define (total mano)
  (if (mano-sin-cartas? mano)
      0
      (+ (valor (una-carta mano))
         (total (resto-cartas mano)) )))
```

Esta versión es mucho más legible. Además, la ocultación de la información proporcionada por la barrera de abstracción de las funciones valor, mano-sin-cartas, una-carta y resto-cartas independiza la función total de la implementación del tipo de dato. Si por alguna razón quisiéramos modificar en el programa la representación de las cartas o de las manos no tendríamos que cambiar la función total.

Los procedimientos auxiliares como `valor` se denominan *selectores* (*selector*s) porque seleccionan un componente de un dato multi-valor.

Realmente, estamos violando la abstracción de datos cuando escribimos una mano de cartas como `'(3h 10c 4d)` porque eso asume que sabemos cómo se representan las cartas; esto es, como palabras en las que se combina el valor de la carta y una letra que representa el palo. Si queremos ir a más en la ocultación de la representación, necesitamos procedimientos *constructores* (*constructors*) además de los selectores:

```
(define (make-carta valor palo)
  (word valor (first palo)) )

(define make-mano list)

(total (make-mano (make-card 3 'oros)
                  (make-card 10 'copas)
                  (make-card 4 'bastos) ))
17
```

En el momento que usamos abstracciones de datos, podemos cambiar la implementación del tipo de datos sin afectar en absoluto a los programas que usan ese tipo de datos. Esto significa que podemos cambiar cómo representar una carta, por ejemplo, sin rescribir

:

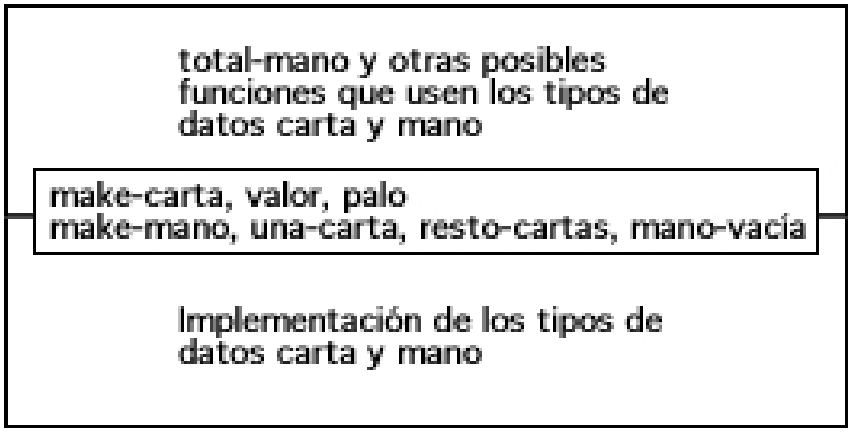
```
(define (make-carta valor palo)
  (cond ((equal? palo 'copas) valor)
        ((equal? palo 'oros) (+ valor 13))
        ((equal? palo 'bastos) (+ valor 26))
        ((equal? palo 'espadas) (+ valor 39))
        (else (error "palo incorrecto:" palo)) ))

(define (valor carta)
  (remainder carta 13))

(define (palo carta)
  (nth (truncate (/ (-1+ carta) 13)) '(copas oros bastos espadas)))
```

Hemos cambiado la *representación* interna, de forma que ahora la carta es únicamente un número entre 1 y 52, pero no hemos cambiado en absoluto la *conducta* (*behavior*) del programa. Podemos seguir llamando a `total` de la misma forma.

Podemos representar la barrera de abstracción con la siguiente figura:



A continuación podemos ver otro ejemplo: los números racionales.

```
(define (make-rat numer denom)
   (cons numer denom))

(define (numer rat)
   (car rat))

(define (denom rat)
   (cdr rat))

(define (add-rat x y)
```

```
   (make-rat (+ (* (numer x) (denom y))
                (* (numer y) (denom x)))
             (* (denom x) (denom y))))

(define (sub-rat x y)
   (make-rat (- (* (numer x) (denom y))
                (* (numer y) (denom x)))
             (* (denom x) (denom y))))

(define (mul-rat x y)
   (make-rat (* (numer x) (numer y))
             (* (denom x) (denom y))))

(define (div-rat x y)
   (make-rat (* (numer x) (denom y))
             (* (denom x) (numer y))))

(define (equal-rat? x y)
   (= (* (numer x) (denom y))
      (* (denom x) (numer y))))
```

Vemos que en el ejemplo hay dos niveles de funciones en el tipo de datos, por un lado se encuentran los selectores `numer` y `denom` que devuelven el numerador y el denominador del número racional. Y por otro lado se encuentran el resto de operaciones que usan estos selectores y que también son parte de la barrera de abstracción del tipo de datos.

3. Referencias

Para saber más de los temas que hemos tratado en esta clase puedes consultar las siguientes referencias:

- Structure and Interpretation of Computer Programs (http://mitpress.mit.edu/sicp/full-text/book/book.html) , Abelson y Sussman, MIT Press 1996 (capítulos 1.3 y 2.1). Disponible biblioteca politécnica (acceso al catálogo (http://gaudi.ua.es/uhtbin/boletin/285815))
- Encyclopedia of Computer Science (Wiley, 2000). Disponible en la biblioteca politécnica (POE R0/E/I/ENC/RAL). Consultar las entradas:
  - Abstract Data Type
  - Encapsulation
  - Information Hiding