Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI2692 – Laboratorio de Algoritmos y Estructuras II
Abril–Julio 2010

Proyecto: Variantes de Registros Académicos con Doble Indexación

(Semana 6 $\rightsquigarrow$ Semana 12)

El objetivo de este proyecto de laboratorio es el construir un par de variantes de implementación de un mismo tipo abstracto de datos (TAD). Además de las variantes del TAD, se requiere construir un cliente que permita manipular ambas variantes; una de las opciones del cliente permitirá hacer una pequeña comparación de rendimiento entre ellas. Toda la implementación será realizada en el lenguaje de programación Java, aunque las especificaciones seguirán siendo presentadas en el estilo GCL que hemos venido utilizando hasta el momento.

1 El tipo abstracto de datos (TAD) *Registro Académico*

El TAD a considerar en este proyecto permite manejar *registros académicos*, que manejan información de asignaturas ya cursadas por los estudiantes

En esta universidad cada uno de los estudiantes es identificado con un carné de tipo string, formado por siete caracteres numéricos (esto es, entre los caracteres '0' y '9'). Por simplicidad, asumiremos que el carné es fijo. No hay un máximo de asignaturas, pues la cantidad de asignaturas que un estudiante puede haber cursado previamente es ilimitada. Al igual que antes, cada asignatura es identificada mediante un código de tipo string, formado por dos caracteres alfabéticos en mayúscula (esto es, entre 'A' y 'Z') seguidos de cuatro caracteres numéricos (esto es, entre '0' y '9').

Por cada asignatura ya cursada por un estudiante, el registro académico almacenará la calificación obtenida por éste. A diferencia de otros sistemas académicos que Ud. pueda conocer, no se puede asociar más de una calificación a la misma asignatura; cada asignatura es cursada hasta su finalización por un estudiante a lo sumo una vez. La calificación es un número entre 1 y 20, ambos extremos incluidos, y no existe la noción de reprobación/aprobación; esto es, la calificación obtenida por un estudiante en un curso es definitiva, aunque ésta haya sido baja. Nunca se "repite" una asignatura.

A continuación presentaremos una especificación para el TAD de registros académicos, al cual llamaremos *RegAcad*. En ésta se modelan los cursos ya realizados mediante una *relación* entre los carnés de los estudiantes y los códigos de las asignaturas; por otra parte, el modelo contempla una *función* que indicará, para cada par de la relación anterior, la calificación correspondiente. (*Nota:* Como antes, el lenguaje matemático que utilizaremos para manejar el modelo abstracto de representación será, según se señala en la guía de lectura del curso de teoría (CI2612), el presentado en el capítulo 9 de [Morgan 94].)

Veamos entonces nuestra especificación con *modelo abstracto*:

Especificación $\mathbb{A}$ de TAD *RegAcad*

Modelo de Representación

var $cursadas : \text{String} \leftrightarrow \text{String}$
$\quad califs : \text{String} \times \text{String} \nrightarrow \text{int}$

Invariante de Representación

$$
\begin{array}{ll}
(\forall\, x \,:\, x \in \text{dom}(cursadas) \,:\, esCarne(x)\,) & \\
\quad \wedge & (\forall\, x \,:\, x \in \text{ran}(cursadas) \,:\, esCodigo(x)\,) \\
\quad \wedge & \text{dom}(califs) = cursadas \\
\quad \wedge & (\forall\, x \,:\, x \in \text{ran}(califs) \,:\, 1 \leqslant x \leqslant 20\,)
\end{array}
$$

Operaciones

$\vdots$

Fin TAD

Se utilizan las funciones auxiliares *esCarne* y *esCodigo* que verifican si un carné y un código de asignatura son correctos.

A continuación se define *esCarne*.

$$esCarne(s) \quad \equiv \quad (s.length() = 7) \;\wedge\; (\forall\, i : 0 \leqslant i < s.length() : \text{`0'} \leqslant s.i \leqslant \text{`9'})$$

a definición de *esCodigo* es la siguiente.

$$
\begin{array}{lll}
esCodigo(s) & \equiv & (s.length() = 6) \;\wedge\; (\forall\, i : 0 \leqslant i < 2 : \text{`A'} \leqslant s.i \leqslant \text{`Z'}) \\
& & \qquad\qquad \wedge\; (\forall\, i : 2 \leqslant i < 6 : \text{`0'} \leqslant s.i \leqslant \text{`9'})
\end{array}
$$

Se tiene que *length* es el método de clase String que devuelve la longitud de un objeto tipo String, y $s.i$ denota el subtring unitario formado con el i-ésimo caracter de s. Se recomienda revisar la documentación de la clase String para la obtención de subtrings.

Veamos ahora las operaciones de nuestro TAD.

Función *crearVacio*: Constructor para crear un nuevo registro académico sin ninguna información.

fun $crearVacio\,() \rightarrow RegAcad$
{ **Comportamiento Normal:**
Pre: *true*
Post: $crearVacio.cursadas = \varnothing$ }

La postcondición señala explícitamente que la relación de asignaturas cursadas del registro creado es vacía, e implícitamente señala, gracias al invariante de representación, que la función de calificaciones también debe ser vacía (esto es, la postcondición dada implica que también se cumple la igualdad $crearVacio.califs = \varnothing$).

Hay dos componentes de especificación el **Comportamiento Normal** y **Comportamiento Excepcional**. El **Comportamiento Normal** nos indica como es el comportamiento "normal" de la operación señalado por el par **Pre / Post** convencional. En cuanto al **Comportamiento Excepcional** nos permite señalar comportamientos "anormales" de la operación. El contenido de esta cláusula indica que si en el estado inicial se cumple la "precondición anormal", la operación debe terminar anormalmente

lanzando una excepción. La precondición convencional y la precondición para terminación anormal son disjuntas, y entre ambas se cubren todos los posibles casos.

Esta posibilidad de terminación anormal podemos manejarla en este proyecto gracias a que trabajaremos con el lenguaje Java, el cual permite utilizar excepciones.

Método *agregar*: Permite registrar en el parámetro implícito *this* un nuevo curso realizado por un estudiante. Recibe el carné e del estudiante, el código a de la asignatura cursada, y la calificación c a registrar.

meth *agregar* (**in** e : String ; **in** a : String ; **in** c : int)
{ **Comportamiento Normal:**
Pre: $esCarne(e) \ \wedge \ esCodigo(a) \ \wedge \ 1 \leqslant c \leqslant 20 \ \wedge \ (e, a) \notin this.cursadas$
Post: $this.califs = this_0.califs \cup \{ ((e, a), c) \}$ }
{ **Comportamiento Excepcional:**
$\neg esCarne(e) \ \leadsto \ ExcepcionEstudianteInvalido(e)$
$\neg\, esCodigo(a) \ \leadsto \ ExcepcionAsignaturaInvalida(a)$
$\neg (1 \leqslant c \leqslant 20) \ \leadsto \ ExcepcionCalificacionInvalida(c)$
$(esCodigo(a) \ \wedge \ 1 \leqslant c \leqslant 20$
$\wedge \ (e, a) \in this.cursadas) \ \leadsto \ ExcepcionCalificacionYaRegistrada(e, a)$ }

De nuevo note que la postcondición señala explícitamente el cambio que debe ser realizado sobre la función *this.califs*, y que, gracias al invariante de representación, esto señala implícitamente el cambio que debe ser realizado sobre la relación *this.cursadas* (esto es, que se le debe agregar el par (e, a)).

En cuanto al comportamiento excepcional, en esta operación se cuenta con varias posibilidades de terminación anormal, cada una con su precondición correspondiente. Observe que la precondición del comportamiento normal es disjunta de cada posible precondición del comportamiento excepcional, y que entre las dos se abarcan todos los casos posibles.

Método *eliminar*: Permite cancelar información de un curso asociado a un estudiante en el registro académico *this*. Recibe el carné e del estudiante afectado y el código a de la asignatura cursada.

meth *eliminar* (**in** e : String ; **in** a : String)
{ **Comportamiento Normal:**
Pre: $esCarne(e) \ \wedge \ esCodigo(a) \ \wedge \ (e, a) \in this.cursadas$
Post: $this.cursadas = this_0.cursadas - \{ (e, a) \}$ }
{ **Comportamiento Excepcional:**
$\neg\, esCarne(e) \ \leadsto \ ExcepcionEstudianteInvalido(e)$
$\neg\, esCodigo(a) \ \leadsto \ ExcepcionAsignaturaInvalida(a)$
$(esCarne(e) \ \wedge \ esCodigo(a) \ \wedge \ (e, a) \notin this.cursadas)$
$\leadsto \ ExcepcionCursoNoRegistrado(e, a)$ }

El cambio explícito indicado en la postcondición sobre *this.cursadas* de nuevo implica, gracias al invariante de representación, el cambio correspondiente a ser realizado sobre *this.califs* (esto es, quitarle el elemento $((e, a), this_0.califs(e, a))$).

De nuevo se tienen varias posibilidades de terminación anormal, todas ellas con precondición disjunta de la precondición convencional, y que cubren todo el universo.

Método *ListarAsignaturasPorEstudiante*: Lista todas las asignaturas, ordenadas por orden lexicográfico, ya cursadas por un estudiante dado e, señalando además las calificaciones obtenidas por éste. No

modifica el registro *this*.

meth *ListarAsignaturasPorEstudiante* (**in** e : String)
{ **Comportamiento Normal:**
Pre: $esCarne(e) \ \wedge \ (\exists\, a :: (e, a) \in this.cursadas)$
Post: $this = this_0 \ \wedge$ Está escrita en pantalla la secuencia, ordenada por asignatura y sin repeticiones, que contiene todos los pares del conjunto $\{ \ x \mid (e, x) \in this.cursadas \bullet (\ x \ , \ this.califs(e, x) \ ) \ \}$ }
{ **Comportamiento Excepcional:**
$\neg esCarne(e) \ \leadsto \ ExcepcionEstudianteInvalido(e)$
$esCarne(e) \ \wedge \ \neg(\exists\, a :: (e, a) \in this.cursadas) \ \leadsto$
$ExcepcionEstudianteNoRegistrado(e)$ }

Método *ListarEstudiantesPorAsignatura*: Dada una asignatura, se listan todos los estudiantes que cursan esa asignatura, ordenadas por orden lexicográfico, junto con su calificación correspondiente. No modifica el registro *this*.

meth *ListarEstudiantesPorAsignatura* (**in** a : String)
{ **Comportamiento Normal:**
Pre: $esCodigo(a) \ \wedge \ (\exists\, e :: (e, a) \in this.cursadas)$
Post: $this = this_0 \ \wedge$ Mostrar por pantalla la secuencia, ordenadas por el carné del estuduante, de todos los pares del conjunto $\{ \ e \mid (e, a) \in this.cursadas \bullet (\ e \ , \ this.califs(e, a) \ ) \ \}$ }
{ **Comportamiento Excepcional:**
$\neg \ esCodigo(a) \ \leadsto \ ExcepcionAsignaturaInvalida(a)$
$esCodigo(a) \ \wedge \ \neg \, (\exists\, e :: (e, a) \in this.cursadas) \ \leadsto$
$ExcepcionAsignaturaNoRegistrada(a)$ }

Método funcional *iteradorEstudianteAsignatura*: Devuelve un *iterador* que permite recorrer todos los elementos de información del registro académico *this*. No modifica *this*. En una sección posterior se explicará qué es un iterador; en este momento, basta con pensar en el iterador de nuestro interés como una secuencia de elementos de información académica.

fmeth *iteradorEstudianteAsignatura* ( ) $\rightarrow$ *Iterator* (*InfoAcad*)
{ **Comportamiento Normal:**
Pre: true
Post: $this = this_0 \ \wedge$ *iteradorEstudianteAsignatura* contiene la secuencia, ordenada por estudiante/asignatura y sin repeticiones, formada por todas las tripletas del conjunto $\{ \ x, y \mid (x, y) \in this.cursadas \bullet (\ x \ , \ y \ , \ this.califs(x, y) \ ) \ \}$ }

Debemos explicar qué es un *método funcional*. Así como un método, a secas, es un procedimiento con parámetro implícito de entrada-salida *this*, un método funcional es una función con parámetro implícito *this*. Esto es, se mantiene la noción del parámetro extra implícito, pero, a diferencia de los métodos a secas, por ser función se devuelve un valor resultado.

El tipo resultado de nuestro método funcional, *Iterator* (*InfoAcad*) , está construido a partir del tipo *InfoAcad* y el tipo genérico *Iterator* (T) , a ser definidos en las dos próximas secciones.

El orden "por estudiante/asignatura" sobre las tripletas mencionado en la postcondición se refiere a que el carné estudiantil es tomado como clave primaria de ordenamiento y el código de asignatura

como clave secundaria. Esto es, el orden es determinado inicialmente por la codificación de los estudiantes y, cuando dos tripletas tengan el mismo estudiante, el orden es determinado por la asignatura. Formalmente:

$$(e0, a0, c0) < (e1, a1, c1) \quad \equiv \quad e0 < e1 \ \ \vee \ \ (e0 = e1 \ \ \wedge \ \ a0 < a1) \qquad .$$

Note que en esta definición el símbolo "<" ha sido sobrecargado: la primera ocurrencia de éste se refiere al orden en definición sobre las tripletas, la segunda ocurrencia y la tercera ocurrencia se refieren al orden lexicográfico sobre strings.

Método funcional *iteradorAsignaturaEstudiante*: Devuelve un *iterador* que permite recorrer todos los elementos de información del registro académico *this*, los cuales van a estar ordenados por asignatura/estudiante y sin repeticiones No modifica *this*.

fmeth *iteradorAsignaturaEstudiante* ( ) → *Iterator* (*InfoAcad*)
{ **Comportamiento Normal:**
Pre: true
Post: $this = this_0 \ \wedge$ *iteradorAsignaturaEstudiante* contiene la secuencia,
ordenada por asignatura/estudiante y sin repeticiones,
formada por todas las tripletas del conjunto
$\{ \ x, y \mid (x, y) \in this.cursadas \bullet (\ x \ , \ y \ , \ this.calif(x, y) \ ) \ \}$ }

El orden "por asignatura/estudiante" se refiere a que el código de asignatura es tomado como clave primaria de ordenamiento y el carné estudiantil como clave secundaria. Es decir, el orden es determinado inicialmente por la asignatura y, cuando dos tripletas tengan la misma asignatura el orden es determinado por el carné de los estudiantes. Esto es:

$$(e0, a0, c0) < (e1, a1, c1) \quad \equiv \quad a0 < a1 \ \ \vee \ \ (a0 = a1 \ \ \wedge \ \ e0 < e1) \qquad .$$

2 El tipo *Información Académica*

Este tipo simplemente representa una tripleta de información académica (e, a, c), siendo e un carné estudiantil, a un código de asignatura cursada por e, y c la calificación obtenida por e al haber cursado a. En nuestra extensión de notación GCL, bastaría con utilizar el constructor "record" de tipos concretos:

type *InfoAcad* = **record**
$\lceil$ *estud* : String
$|$ *asign* : String
$\lfloor$ *calif* : int
end

En Java, basta construir este registro, o "record", como una clase. Esta clase la llamamos *InfoAcad*, la cual va a tener tres campos privados. El primero es *estud*, que corresponde al carné de un estudiante, el segundo es *asign* que guarda el código de una asignatura y el tercero es *calif* almacena la calificación que obtuvo un estudiante en una asignatura. Estos campos se inicializan una vez que se crea un objeto *InfoAcad*, ya que el constructor de la clase recibe tres parámetros, el carné *estud*, el código *asign* y la calificación *calif*. Estos campos se hacen privados con el fin de evitar que sean accesados de manera directa desde un objeto de tipo *InfoAcad* ya sea para editarlos o para obtener su información. Esto es lo que se conoce como encapsulación de la información en Java. Para poder obtener el valor de estos campos se crean los métodos *obtenerEstudiante*, *obtenerAsignacion* y *obtenerCalificacion*. En consecuencia los usuarios de la clase *InfoAcad* van a poder tener acceso a los valores de los campos, pero los mismos no podrán ser editados.

```
public class InfoAcad {
    private String estud, asign;
    private int    calif;

    public InfoAcad(String estud, String asign, int calif){
        this.estud = estud;
        this.asign = asign;
        this.calif = calif;
    }

    public String obtenerEstudiante(){
        return this.estud;
    }

    public String obtenerAsignacion(){
        return this.assign;
    }

    public int obtenerCalificacion(){
        return this.calif;
    }
    ...
}
```

Al manejar *InfoAcad* como un tipo concreto, se están perdiendo chequeos de integridad de información, pues no se verifica la validez del carné del estudiante (formato especificado), ni del código de la asignatura (formato antes especificado), ni de la calificación (entre 1 y 20). Esto puede resolverse chequeando los valores de carné, asignatura y calificación antes de construir un objeto de tipo *InfoAcad*.

En nuestro caso, los elementos de información académica serán siempre provistos desde dentro del TAD *RegAcad*. Por lo tanto, la integridad de la información que se maneje en el tipo concreto *InfoAcad* vendrá garantizada por la integridad de la información contenida en un *RegAcad*. Sin embargo, si Ud. prefiere construir *InfoAcad* como un TAD en lugar de como un tipo concreto, puede hacerlo.

3 El TAD *Iterador*

Un *iterador*,en términos abstractos, simplemente provee operaciones para recorrer una secuencia de principio a fin. En sus posibles implementaciones concretas, la historia es diferente, pero de esto hablaremos en otro momento. Por ahora, simplemente veamos una especificación con modelo abstracto para el TAD de iteradores:

Especificación $\mathbb{A}$ de TAD *Iterator* (T)

Modelo de Representación

var $s : \text{seq } T$

Invariante de Representación

true

Operaciones

fmeth $hasNext\,() \rightarrow \text{boolean}$

{ **Comportamiento Normal:**
Pre: true
Post: $hasNext \equiv (this.s \neq \langle\,\rangle) \ \wedge \ this = this_0$ }

fmeth $next\,() \rightarrow T$

{ **Comportamiento Normal:**
Pre: $this.s \neq \langle\,\rangle$ }
Post: $next = \text{hd } this_0.s \ \wedge \ this.s = \text{tl } this_0.s$ }
{ **Comportamiento Excepcional:**
$this.s = \langle\,\rangle \ \rightsquigarrow \ ExcepcionNoHayElementos\,()$ }

Fin TAD

Note que el modelo abstracto es una secuencia. La notación y operaciones utilizadas para secuencias son, como hemos venido trabajando para modelos abstractos en general, las de [Morgan 94].

El modelo s representa la secuencia de elementos por recorrer, esto es, la secuencia de elementos sobre los cuales "se iterará". El método funcional *hasNext* sirve para indicar si aún hay elementos por recorrer y no altera el iterador, de allí que el resultado devuelto sea el valor booleano de la desigualdad $this.s \neq \langle\,\rangle$. Por otra parte, el método funcional *next* sirve para tomar el primer elemento del iterador y avanzar hacia el siguiente, de allí que el resultado devuelto sea el primer elemento o cabeza de la secuencia (el nombre de la operación "hd" se refiere a "head") y la secuencia sea alterada quitándole la cabeza y quedándose con el resto o cola (el nombre de la operación "tl" se refiere a "tail"). Otra formulación, equivalente, de la postcondición del método funcional *next* habría sido $\langle next \rangle \mathbin{+\!\!+} this.s = this_0.s$.

Así como escogimos el nombre "*Iterator*" para este TAD por convenciones del lenguaje Java, los nombres de los métodos y el nombre de la excepción utilizados también corresponden a Java.

Note que no hemos especificado función constructora para el tipo. Esto se debe a que se permite que cada implementación con modelo concreto determine qué tipo de constructor le conviene utilizar, en lugar de prefijarlo desde la especificación original con modelo abstracto.

La noción de iterador es tratada en el capítulo 6 de [Liskov & Guttag 01] de manera muy completa y clara. Se explica lo que son iteradores, cómo usarlos, cómo implementarlos en Java, e incluso define los invariantes de representación y relaciones de acoplamiento (funciones de abstracción en terminología Liskov) de los modelos concretos presentados como ejemplos contra un modelo abstracto como el dado por nosotros acá. Se recomienda fuertemente la lectura del referido capítulo; sin esta lectura es poco probable aprender a manejar los iteradores adecuadamente.

4 Las variantes de implementación

Como fue indicado inicialmente, se desea trabajar un par de variantes de implementación del TAD de registros académicos. En esta sección son descritas las dos variantes deseadas, especificando parcialmente

los modelos concretos correspondientes.

4.1 El primer modelo concreto: arreglos

En nuestra primera variante de implementación, el modelo concreto lo construiremos con arreglos.

Veamos la especificación con nuestro primer *modelo concreto*:

Especificación $\mathbb{B}$ de TAD *RegAcad* , **refinamiento de** $\mathbb{A}$

Modelo de Representación

var NIA, MAX : int
$esAsign$: array $[\,0\,..\,MAX\,)$ of InfoAcad
$asignEs$: array $[\,0\,..\,MAX\,)$ of InfoAcad

Fin TAD

Los arreglos *esAsign* y *asignEs* son de tipo *Información Académica*, conteniendo la información de cada uno de los estudiantes. La diferencia entre los dos arreglos consiste en que en el arreglo *esAsign*, los registros de información académica están ordenados por estudiante/asignatura, mientras que en el arreglo *asignEs* se ordenan por asignatura/estudiante. La variable *NIA* indica el número de registros de información académica almacenados en los arreglos. La variable *MAX* corresponde al tamaño del arreglo. Los registros de información académica están almacenados en las primeras *NIA* posiciones de los arreglos. Es decir, el segmento de los arreglos *esAsign* y *asignEs* con la información de los estudiantes será $[0 \ldots NIA)$, mientras que el segmento $[NIA \ldots MAX)$ será ignorado. Si al agregar información académica se tiene que se alcanza el tamaño máximo de los arreglos, entonces se debe duplicar el tamaño de los mismos y por consecuencia, el valor de variable *MAX*.

Invariante de Representación

$$
\begin{aligned}
& 0 \leqslant NIA \leqslant MAX \ \wedge \ MAX > 0 \\
& \wedge \ (\forall\, i \ : \ 0 \leqslant i < NIA \ : \ esCarne(esAsign[i].estud) \ \wedge \ esCodigo(esAsign[i].asign) \\
& \qquad\qquad \wedge \ 1 \leqslant esAsign[i].calif \leqslant 20\,) \\
& \wedge \ (\forall\, i,j \ : \ 0 \leqslant i < j < NIA \ : \ (\, esAsign[i].estud < esAsign[j].estud\,) \\
& \quad \vee \ (\, esAsign[i].estud = esAsign[j].estud \ \wedge \ esAsign[i].asign < esAsign[j].asign\,) \\
& \wedge \ (\forall\, i \ : \ 0 \leqslant i < NIA \ : \ esCarne(asignEs[i].estud) \ \wedge \ esCodigo(asignEs[i].asign) \\
& \qquad\qquad \wedge \ 1 \leqslant asignEs[i].calif \leqslant 20\,) \\
& \wedge \ (\forall\, i,j \ : \ 0 \leqslant i < j < NIA \ : \ (\, asignEs[i].asign < asignEs[j].asign\,) \\
& \quad \vee \ (\, asignEs[i].asign = asignEs[j].asign \ \wedge \ asignEs[i].estud < asignEs[j].estud\,)\,) \\
& \wedge \ \{\ i \ : \ 0 \leqslant i < NIA \ : \ esAsign[i]\ \} = \{\ i \ : \ 0 \leqslant i < NIA \ : \ asignEs[i]\ \}
\end{aligned}
$$

Usted deberá implementar el invariante de representación en JML.

En cuanto a la relación de acoplamiento tenemos que la relación *cursadas* debe terminar siendo el dominio de la función *califs*. Debido a esto sólo especificaremos detalladamente cómo se construye *califs* a partir del modelo concreto dado y a *cursadas* simplemente le exigiremos ser el dominio de tal construcción.

Relación de Acoplamiento

$$
\begin{aligned}
& califs \ = \ \{\ i \ : \ 0 \leqslant i < NIA \ : \ (\,(\, esAsign[i].estud\,,\, esAsign[i].asign\,)\,,\, esAsign[i].calif\,)\ \} \\
& \quad \wedge \\
& cursadas \ = \ \mathrm{dom}\,(califs)
\end{aligned}
$$

Debido a que en el invariante de representación se indica que el arreglo *esAsign*$[i]$ y el arreglo *asignEs*$[i]$ poseen los mismos elementos, se tiene que la relación de acoplamiento también se cumple para el arreglo *asignEs*$[i]$.

En cuanto a la construcción de este modelo concreto en Java, recuerde que el mecanismo de Java utilizado para construir TADs son las clases, pero igualmente son las clases el mecanismo de Java que puede ser utilizado para construir registros ("record"s). Para evitar confusión en cuanto a lo que se desea, y las posibles maneras de lograrlo en Java, indicamos explícitamente a continuación parte de la clase de Java que deberá ser construida:

```
public class RegAcadArreglos implements RegAcad {
    private int         NIA, MAX;
    private InfoAcad[]  esAsign;
    private InfoAcad[]  asignEs;
    ...
}
```

En relación con el atributo *MAX*, éste puede ser omitido si así Ud. lo prefiere, en vista que éste siempre deberá ser igual a las longitud de su correspondiente arreglo. (La longitud de un arreglo unidimensional `a` es `a.length` en Java.)

Una cuestión importante. Cada una de las posiciones de los arreglos *esAsign* y *asignEs deben hacer referencia a un mismo conjunto de objetos de tipo InfoAcad*. Se debe evitar duplicar información. En el siguiente ejemplo se muestra como la posición 3 del arreglo *esAsign* y la posición 1 del arreglo *asignEs*, hacen referencia a un mismo objeto de tipo *InfoAcad*.

```
...
InfoAcad a new InfoAcad("0696320", "ci2692", 16);
esAsign[3] = a;
asignEs[1] = a;
...
```

4.2 El segundo modelo concreto: árboles binarios de búsqueda

En la segunda variante de implementación, utilizaremos árboles binarios de búsqueda como modelo concreto. Éstos serán árboles binarios de nodo (esto es, árboles con información en los nodos internos mas no en las hojas) en los que cada nodo almacena un elemento de tipo información académica, *InfoAcad*.

Con nuestro segundo <u>modelo concreto</u>, la especificación es como sigue:

Especificación $\mathbb{C}$ de TAD *RegAcad* , **refinamiento de** $\mathbb{A}$

Modelo de Representación

var *eac* : *ArbolInfoAcad*
aec : *ArbolInfoAcad*

Fin TAD

siendo *ArbolInfoAcad* un tipo concreto correspondiente al siguiente tipo algebraico-libre:

freeType *ArbolInfoAcad* = <u>avac</u>
| <u>nodo</u> (InfoAcad , *ArbolInfoAcad* , *ArbolInfoAcad*) .

Los árbololes *eac*, *aec* reciben sus nombres de "*e*"studiante/"*a*"signatura/"*c*"alificación. Cada árbol binario contendrá un elemento de tipo información académica y los subárboles izquierdo y derecho usuales. El orden a ser respetado en la organización de la información dentro del árbol *eac* será el correspondiente a la relación de orden estudiante/asignatura. El árbol *aec* deberá estar organizado por asignatura/estudiante.

En el invariante de representación se deberá exigir que todos los elementos de información académica almacenados en los árboles *eac* y *aec* sean válidos y, además, exigir que *eac* y *aec* sean efectivamente árboles de búsqueda bajo el orden deseado. Vamos a mostrar el invariante sólo con el árbol *eac*, ya que es ánalogo para el árbol *aec*.

Veamos el invariante de representación:

Invariante de Representación

$$\begin{aligned} & \mathit{infoValida}\,(\mathit{eac}) \;\wedge\; \mathit{esArbolDeBusqEstAsig}\,(\mathit{eac}) \\ \wedge\; & \mathit{infoValida}\,(\mathit{aec}) \;\wedge\; \mathit{esArbolDeBusqAsigEst}\,(\mathit{aec}) \\ \wedge\; & \mathit{extraccionInfo}\,(\mathit{eac}) \;=\; \mathit{extraccionInfo}\,(\mathit{aec}) \end{aligned}$$

donde *infoValida*, *esArbolDeBusqEstAsig* y *esArbolDeBusqAsigEst* son funciones definidas inductivamente sobre la estructura de *ArbolInfoAcad*. La función *esArbolDeBusqEstAsig* determina si la estructura de entrada, es un árbol binario de búsqueda donde la información académica está ordenada por estudiante/asignatura. En la función *esArbolDeBusqAsigEst* la información académica está ordenada por asignatura/estudiante.

A continuación se definen *infoValida*, *esArbolDeBusqEstAsig* y *esArbolDeBusqAsigEst*.

$\mathit{infoValida} : \mathit{ArbolInfoAcad} \to \text{boolean}$

con cláusulas

$$\left[\begin{array}{l} \mathit{infoValida}(\underline{\text{avac}}) \;=\; \text{true} \quad , \\ \mathit{infoValida}(\underline{\text{nodo}}\,(\mathit{infoEstud}, \mathit{izq}, \mathit{der})) \;=\; \mathit{esCarne}(\mathit{infoEstud.estud}) \\ \qquad\qquad \wedge\; \mathit{esCodigo}\,(\mathit{infoEstud.asign}) \\ \qquad\qquad \wedge\; 1 \leqslant \mathit{infoEstud.calif} \leqslant 20 \\ \qquad\qquad \wedge\; \mathit{infoValida}\,(\mathit{izq}) \\ \qquad\qquad \wedge\; \mathit{infoValida}\,(\mathit{der}) \quad . \end{array}\right.$$

$\mathit{esArbolDeBusqEstAsig} : \mathit{ArbolInfoAcad} \to \text{boolean}$

con cláusulas

$$\left[\begin{array}{l} \mathit{esArbolDeBusqEstAsig}(\underline{\text{avac}}) \;=\; \text{true} \quad , \\ \mathit{esArbolDeBusqEstAsig}(\underline{\text{nodo}}(\mathit{infoEstud}, \mathit{izq}, \mathit{der})) = (\mathit{infoEstud.estud} < \mathit{minCarne}(\mathit{der}) \vee \\ \qquad\qquad (\mathit{infoEstud.estud} = \mathit{minCarne}(\mathit{der}) \wedge \\ \qquad\qquad \mathit{infoEstud.asign} < \mathit{minAsign}(\mathit{der}))) \\ \qquad\qquad \wedge\; \mathit{esArbolDeBusqEstAsig}(\mathit{der}) \\ \qquad\qquad \wedge\; (\mathit{infoEstud.estud} > \mathit{maxCarne}(\mathit{izq}) \vee \\ \qquad\qquad (\mathit{infoEstud.estud} = \mathit{maxCarne}(\mathit{izq}) \wedge \\ \qquad\qquad \mathit{infoEstud.asign} > \mathit{maxAsign}(\mathit{izq}))) \\ \qquad\qquad \wedge\; \mathit{esArbolDeBusqEstAsig}(\mathit{izq}) \end{array}\right.$$

$esArbolDeBusqAsigEst : ArbolInfoAcad \to \text{boolean}$

con cláusulas

$$\left\lceil \begin{array}{l} esArbolDeBusqAsigEst(\underline{\text{avac}}) = \text{true} \quad , \\ esArbolDeBusqAsigEst(\underline{\text{nodo}}(infoEstud, izq, der)) = (infoEstud.asign < minAsign(der) \vee \\ \qquad (infoEstud.asign = minAsign(der) \wedge \\ \qquad infoEstud.carne < minCarne(der))) \\ \qquad \wedge\ esArbolDeBusqAsigEst(der) \\ \qquad \wedge\ (infoEstud.asign > maxAsign(izq) \vee \\ \qquad (infoEstud.asign = maxAsign(izq) \wedge \\ \qquad infoEstud.carne > maxCarne(izq))) \\ \qquad \wedge\ esArbolDeBusqAsigEst(izq) \end{array} \right.$$

Se tiene que *infoEstud* es un elemento de tipo *InfoAcad*, las funciones *minCarne* y *maxCarne*, determinan el menor y mayor valor de carné que se encuentran en el árbol de búsqueda. De la misma manera las funciones *minAsign* y *maxAsign* tienen como fin encontrar el menor y mayor valor de asignatura del árbol de búsqueda.

La funcion *extraccionInfo* se define inductivamente como sigue:

$extraccionInfo : ArbolInfoAcad \to (\text{String} \times \text{String} \nrightarrow \text{int})$

con cláusulas

$$\left\lceil \begin{array}{l} extraccionInfo(\underline{\text{avac}}) = \varnothing \quad , \\ extraccionInfo(\underline{\text{nodo}}\,(infoEstud, izq, der)) = \{\ ((infoEstud.estud, infoEstud.asign), \\ \qquad infoEstud.calif\ )\ \} \ \cup\ extraccionInfo\,(izq) \\ \qquad \cup\ extraccionInfo\,(der) \quad . \end{array} \right.$$

Al igual que con el primer modelo, Ud. deberá implementar el invariante de representación en JML.

En cuanto a la relación de acoplamiento, la información correspondientes a la función *califs* se pueden obtener de los árboles *eac* y *aec*. En vista de que la relación *cursadas* debe terminar siendo el dominio de la función *califs*, sólo especificaremos detalladamente cómo construir *califs* a partir del modelo concreto dado, y a *cursadas* simplemente le exigiremos ser el dominio de tal construcción. Usaremos el árbol *eac*, pero la relación de acoplamiento también se debe cumplir para el árbol *aec*.

Relación de Acoplamiento

$$califs = extraccionInfo\,(eac) \quad \wedge \quad cursadas = \text{dom}\,(califs)$$

Para construir este modelo concreto en Java, sabiendo que no contamos con la posibilidad de definir directamente tipos algebraico-libres, debemos recurrir a las referencias/apuntadores de Java. Así, la referencia nula puede ser utilizada para representar el árbol vacío y una referencia no nula hacia un nodo para representar árboles no vacíos. Conviene además, dentro de nuestro proyecto, utilizar una estructura doblemente enlazada, por lo que cada nodo debe apuntar no sólo a sus dos hijos sino también a su progenitor. Veamos a continuación parte de la clase de Java que deberá ser construida para manejar este segundo modelo con árboles binarios de búsqueda doblemente enlazados:

```
public class RegAcadArboles implements RegAcad {
    private Nodo raizAEC;
    private Nodo raizEAC;

    private class Nodo {
        public InfoAcad infoEstud;
```

```
        public Nodo     izq, der, prog;
    }
    ...
}
```

Se tiene que el campo `raizEAC` es la raíz de un `árbol` en donde la información académica está ordenada por estudiante/asignatura. El campo `raizAEC` es la raíz de un `árbol` en donde la información académica está ordenada por asignatura/estudiante.

Al igual que en el modelo concreto con arreglos, los objetos de tipo información académica *InfoAcad* de los árboles *eac* y *aec*, deben hacer referencia a un mismo conjunto de objetos de tipo *InfoAcad*.

El hecho de que los atributos `izq`, `der` y `prog` sean de tipo `Nodo` facilitará el que Ud. realice implementaciones iterativas de las operaciones del TAD. Recuerde que la iteración es usualmente preferible a la recursión.

La implementación de los métodos *agregar* y *eliminar* en este modelo concreto debe ser iterativa. Ud. puede escoger si prefiere implementar los métodos *ListarAsignaturasPorEstudiante* y *ListarEstudiantesPorAsignatura* de manera iterativa o recursiva, justificando su elección.

4.3 Los iteradores concretos

Para cada uno de los dos modelos concretos descritos, Ud. debe proveer una implementación del TAD *Iterator*, según lo requerido por el TAD *RegAcad*. Estos iteradores concretos deben ser construidos, tal como lo indica el capítulo 6 de [Liskov & Guttag 01], como clases privadas internas de las clases `RegAcadArreglos` y `RegAcadArboles`. Cada una de esas implementaciones concretas proveerá el constructor que Ud. considere conveniente.

Se le recomienda que analice por qué el estar utilizando una estructura de doble enlace para los árboles le permite implementar su iterador sólo con una referencia a `Nodo`. Si la estructura fuese de enlace simple, esto es, con cada nodo apuntando sólo a sus hijos pero no a su progenitor, sería más complejo construir el iterador. Como ejercicio adicional, Ud. deberá analizar cómo habría podido construir el iterador si se hubiese tenido una estructura simplemente enlazada en lugar de doblemente enlazada. (*Ayuda*: Una solución a este problema es presentada en [Liskov & Guttag 01].)

5 El programa cliente

Contando con las dos implementaciones del TAD *RegAcad*, que en Java serán las dos clases antes descritas, cada una de las cuales implementa la interfaz `RegAcad`, Ud. montará un programa cliente de ellas. El cliente manejará una serie de registros académicos almacenados en un arreglo, cada uno de los cuales podrá corresponder a cualquiera de las dos implementaciones del TAD *RegAcad*.

El programa cliente preguntará al usuario cuántos registros académicos desea manejar, y luego entrará en una iteración de menú con las siguientes 11 opciones:

1. crear un registro vacío
2. agregar información académica
3. eliminar información académica
4. listar asignaturas/calificaciones de un estudiante

5. listar estudiantes/calificaciones de una asignatura
6. cargar un registro académico completo desde un archivo
7. listar toda la información contenida en un registro académico ordenada por estudiante/asignatura
8. listar toda la información contenida en un registro académico ordenada por asignatura/estudiante
9. procedimiento sorpresa1
10. procedimiento sorpresa2
11. salir del programa

En la 1ra. opción para la creación de registros académicos vacíos, se le debe preguntar con cuál variante de implementación desea que el nuevo registro sea manejado: arreglos o árboles binarios de búsqueda. Esto aplica igualmente para la 6ta. opción, la correspondiente a cargar un registro académico completo a partir de un archivo.

En las restantes 2da., 3ra., 4ta., 5ta., 7ma., 8va., 9na. y 10ma. opciones, al preguntar al usuario con cuál registro académico desea trabajar, éste deberá corresponder a uno previamente creado con la 1ra. o la 6ta. opción; en caso contrario se le indicará al usuario su error.

En la 6ta. opción, Ud. preguntará al usuario, tal como en todas las otras opciones, sobre cuál de los registros académicos se desea realizar la operación; también preguntará al usuario cuántos estudiantes deberán ser manejados en el nuevo registro y preguntará al usuario el nombre del archivo desde el que se desea realizar la carga. El archivo contendrá una información académica por línea en el formato

⟨CarnéEstudiantil⟩ ⟨CódigoAsignatura⟩ ⟨Calificación⟩

en el que el carné y el código, al igual que el código y la calificación, están separados por blancos. Asuma que el archivo de entrada no tiene errores. Esta 6ta. opción de carga de archivo deberá ser implementada, al igual que todas las otras opciones, con un procedimiento *cliente* del TAD *RegAcad*. Al TAD no se le deben agregar operaciones adicionales a las definidas inicialmente. Esto le permitirá a Ud. independizarse de con cuál de las implementaciones del TAD se está trabajando.

En la 7ma y 8va opción deberá ser implementada por Ud. con un procedimiento *cliente* de las operaciones del TAD *Iterator*, haciendo uso de sus propios iteradores.

Las operaciones 9 y 10 van a ser implementada por los profesores de laboratorio. En la opción 9na el método `sorpresa1` va a recibir como parámetro el iterador generado el método *iteradorEstudiante-Asignatura*. En la 10ma opción el método `sorpresa2` recibe como entrada el iterador que se obtiene del método *iteradorAsignaturaEstudiante*. Cuando se escoga alguna de estas dos opciones Ud. preguntará al usuario con cuál de los registros académicos se desea trabajar, verificará que éste haya sido creado previamente con la 1ra. o la 6ta. opción, generará el iterador que corresponda a la opción 9na o 10ma y lo enviará como parámetro al procedimiento `sorpresa1` o `sorpresa2` definidos en la clase `Sorpresa` que será provista por los profesores de laboratorio. Estos procedimientos, los cuales corresponden a métodos estáticos en Java, recibirán el iterador del tipo *Iterator* (*InfoAcad*), que corresponda a la opción y realizará algún procesamiento sobre la secuencia de elementos de información académica contenida en tal iterador. Estos procedimientos deberán funcionar, sin ser modificados, al ser integrados a las clases construidas por Ud. Tal integración funcionará correctamente si Ud. provee implementaciones adecuadas de los tipos *RegAcad*, *Iterator* e *InfoAcad*, ajustadas a las especificaciones dadas en este enunciado.

En relación con las excepciones que pueden ser lanzadas desde las operaciones de los TADs utilizados, no se debe permitir que éstas alcancen al usuario. Esto es, su programa cliente debe atrapar estas excepciones, las cuales señalan errores en la información provista a las operaciones de los TADs, y

procesar tales excepciones para presentar los errores amigablemente al usuario, sin permitir que el programa cliente aborte como consecuencia de estas excepciones.

6 Entrega de Avances y Entrega Final

Durante el desarrollo de este proyecto, Ud. realizará tres entregas: dos primeros avances parciales y una entrega final con todos los requerimientos.

Avance I - Lunes de semana 8

En este primer avance, Ud. trabajará sólo en parte la primera variante de implementación del TAD *RegAcad* con arreglos. Esto le permitirá familiarizarse con el problema y con la implementación de TADs en Java.

Específicamente, en este avance I, Ud. deberá entregar:

- Una versión parcial de la primera variante de implementación del TAD *RegAcad*, con las operaciones *crearVacio* (que en Java será un constructor), *agregar*, *ListarAsignaturasPorEstudiante* y *ListarEstudiantesPorAsignatura*. Esta versión ya deberá contar con la implementación de los invariantes de representación en JML.
- Una versión parcial del programa cliente, que tenga activas la 1ra. opción sólo para la variante de implementación con arreglos, y las opciones 2da., 4ta., 5ta., y 11va. del menú. Esto es, las opciones siguientes: crear un registro académico vacío con implementación de arreglos, agregar información académica, listar información académica de un estudiante, listar información de los estudiantes dada una asignatura y salir. Para esto, deberá bastar la versión parcial de la primera variante de implementación del TAD descrita en el punto anterior.
- En esta entrega las operaciones *no harán uso de excepciones*.

Avance II - Jueves de semana 10

En este segundo avance, Ud. deberá completar la primera variante de implementación del TAD *RegAcad*.

Específicamente, en este avance II, Ud. deberá entregar:

- Una versión de la primera variante del TAD *RegAcad* que debe estar ya completa, excepto por las opciones 9na y 10ma.
- Las operaciones de las opciones 4ta., 5ta., 7ma. y 8va. deberán hacer uso de iteradores.
- Una versión completa del programa cliente de la primera variante del TAD *RegAcad*, teniendo activas todas las opciones del menú excepto por la 9na y 10ma.
- Todas las operaciones que lo requieran deben hacer uso de excepciones.

Entrega Final - Lunes de semana 12

Esta última entrega incluye la versión final del programa cliente, con todas las opciones activas, lo cual requiere por supuesto que todas las variantes de implementación de todos los TADs hayan sido

completadas adecuadamente. El código de todas las clases e interfaces utilizadas en su programa debe ser entregado tanto en un CD como en papel, y también debe ser entregado un informe que describa el proceso de desarrollo y las preguntas adicionales que fueron dejadas en este enunciado.

En relación con el informe, éste deberá seguir un formato similar al que usualmente se da como guía en el laboratorio de Algoritmos y Estructuras I. Éste deberá contener las siguientes secciones:

- Portada, la cual debe contener:
  - Arriba a la izquierda: el nombre de la universidad, la carrera y la asignatura.
  - Centrado: el nombre del proyecto.
  - Abajo a la izquierda: nombre del profesor del laboratorio.
  - Abajo a la derecha: nombre y carné de los integrantes del equipo.
- Introducción:
  - Breve descripción del problema atacado en el proyecto.
  - Descripción del contenido del resto del informe.
- Diseño:
  - Decisiones de diseño tomadas por Ud. en relación con cualquier aspecto del proyecto, que Ud. considere conveniente explicar en el informe.
    - Un ejemplo de tales decisiones corresponde a haber optado por una implementación iterativa o recursiva de los métodos *ListarAsignaturasPorEstudiante* y *ListarEstudiantesPorAsignatura* en la variante de árboles del TAD *RegAcad*.
    - Otro ejemplo corresponde a la implementación de la función *esCodigo*: ¿Fue construida con un método estático o con un método convencional (esto es, dinámico o no estático)? ¿Fue duplicado este método en las dos variantes de implementación del TAD *RegAcad*? ¿Se decidió hacer uso de herencia para, en lugar de tener una interfaz `RegAcad`, tener una clase abstracta `RegAcad` con una única implementación de *esCodigo*, la cual es heredada por las dos variantes de implementación del TAD?
      Todas estas preguntas le están siendo planteadas para sugerirle que, en la entrega final, una vez que ya Ud. sabe de los mecanismos de herencia de Java, contemple la posibilidad de cambiar la interfaz `RegAcad` por una clase abstracta semi-implementada.
    - Un tercer ejemplo digno de comentar se refiere a si el doble enlace de los árboles de búsqueda contribuyó a simplificar la construcción del iterador correspondiente.
    - Cualquier otro ejemplo de decisiones similares de diseño tomadas por Ud...
- Detalles de implementación:
  - Peculiaridades relacionadas con el paso de parámetros en métodos de Java, que pudiesen diferir de lo diseñado o deseado por Ud.
  - Cualquier otro comentario referente a peculiaridades de Java.
- Estado actual:
  - Operatividad del programa: señalar si funciona perfectamente o no; en caso negativo, describir las anomalías.
  - Manual de operación: nombre de los archivos correspondientes a todas las clases e interfaces que conforman el programa, nombre del archivo a ejecutar (esto es: la clase con el programa principal), y modo de operar el programa.

- Conclusiones:
  - Experiencias adquiridas.
  - Dificultades encontradas durante el desarrollo.
  - Recomendaciones.
- Bibliografía: libros, revistas o cualquier otro recurso consultado (que puede incluso ser conversaciones o consultas con amigos o expertos).

  Una referencia bibliográfica debe incluir la siguiente información:
  - Libro: autores, nombre, editorial y año.
  - Artículo: autores, nombre, revista, volumen, número y año.
  - Página web: autores, nombre, URL, fecha de la última visita.
  - Conversaciones personales o consultas: interlocutor y fecha.

Esta entrega final será realizada el día lunes de la semana 12 *a las 3:00 pm.*, en la secretaría del Departamento de Computación y Tecnología de la Información (MYS216). En la fecha *y la hora* referidas, Ud. entregará el CD, el listado del programa y el informe dentro de un sobre debidamente identificado (nombre y carné de los integrantes del equipo, y nombre del proyecto). En la clase de laboratorio siguiente, el jueves de la semana 12, se le indicará si se requiere un proceso de revisión y prueba del proyecto por parte de su profesor de laboratorio con el equipo programador presente.

Es importante que el equipo trabaje de manera integrada. Es posible que durante la revisión del funcionamiento del programa se realice un breve interrogatorio *individual* a cada miembro del equipo.

7 Comentarios Finales

- Cualquier error que a posteriori sea hallado en este enunciado, así como cualquier otro tipo de observación adicional sobre el desarrollo del proyecto, serán publicados como fe de erratas en el wiki del curso.
- *Nota importante:* El proyecto es un trabajo en equipo de dos personas. Si bien lleva más de dos semanas, se considera similar a un examen en cuanto a que no puede haber intercambios contrarios a la ética con otros equipos.