Encapsulamiento y Abstracción

Un sistema es una colección de componentes que interactúan entre sí con un objetivo común. En la construcción de sistemas complejos el **encapsulamiento** es un mecanismo fundamental porque permite utilizar cada componente como una "caja negra", esto es, sabiendo **qué** hace sin saber **cómo** lo hace. Se reducen así las dependencias entre las diferentes componentes, cada una de las cuales es más fácil de entender, probar y modificar.

El concepto de encapsulamiento está ligado a toda actividad vinculada a la producción. Una máquina compuesta se construye conectando varias máquinas simples, de modo que la salida de cada una constituye la entrada de otra y en conjunto alcanzan un propósito. Un motor, por ejemplo, se fabrica a partir de dispositivos mecánicos y energéticos. Estos dispositivos pueden usarse para construir otro tipo de máquinas. Un motor a su vez, va a ser usado como una componente en la construcción de otras máquinas.

Cada componente es un **módulo** que puede considerarse individualmente o como parte del conjunto. Cuando un módulo se analiza como parte de un todo, es posible concentrarse en qué función realiza, sin considerar cómo fue construido. Cuando un módulo se analiza individualmente es posible hacer abstracción de los detalles del contexto en el que va a ser utilizado, para enfocarse en cómo lograr su propósito.

En el desarrollo de sistemas de software el concepto de encapsulamiento está fuertemente ligado al de **abstracción**. Analistas y diseñadoras elaboran un modelo a partir de un proceso de abstracción que especifica una colección de módulos relacionados entre sí. Los desarrolladores retienen en el código la estructura modular especificada en el diseño. Cada módulo puede ser verificado independientemente de los demás, antes de integrarse al sistema. Durante el mantenimiento, el encapsulamiento permite reducir el impacto de los cambios.

La construcción de cada módulo de software puede realizarse sin conocer el sistema al cual va a integrarse. El sistema completo se construye a partir de una colección de módulos, sin considerar cómo se implementó cada uno en particular.

Encapsulamiento en la programación orientada a objetos

La **estructura estática** de un sistema orientado a objetos queda establecida por una colección de clases relacionadas entre sí. El **modelo dinámico** es un entorno poblado de objetos comunicándose a través de mensajes. El comportamiento de cada objeto en respuesta a un mensaje, depende de la clase a la que pertenece.

El concepto de encapsulamiento está ligado tanto a las clases como a los objetos. Cada objeto de software interactúa con otros objetos del entorno a través de su **interfaz** y **oculta** su estado interno y los detalles que describen cómo actúa cuando recibe un mensaje. Así, si la estructura del estado interno se modifica, el cambio no afecta a los otros miembros del entorno. Solo las modificaciones en la interfaz pueden afectar a los demás objetos.

Para que un objeto de software oculte su estado interno, la clase que define sus atributos y comportamiento debe **esconder** la implementación. Cada clase es una caja negra, es decir, conoce **qué** hacen las demás clases del sistema, pero no **cómo** lo hacen.

En la programación orientada a objetos el **encapsulamiento**[1] es el mecanismo que permite **agrupar** en una clase los atributos y el comportamiento que caracterizan a sus instancias y **esconder** la representación de los datos y los algoritmos que modelan al comportamiento, de modo que solo sean visibles dentro de la clase.

El encapsulamiento brinda cierto nivel de independencia, cada clase puede ser construida y verificada antes de integrarse al sistema completo. Una clase puede modificar la representación de los datos o los algoritmos, sin afectar a sus clases clientes, en tanto mantenga su funcionalidad y sus responsabilidades.

Estructura de datos

Los atributos de una clase definen la **estructura de datos** que permite representar el estado interno de los objetos de esa clase en ejecución. La programación orientada a objetos propone esconder la representación de los datos, de modo que no sean visibles desde el exterior de la clase.

En Java los **modificadores de acceso** permiten establecer la visibilidad de los miembros de una clase. Los atributos se declaran privados y quedan así escondidos dentro de la clase.

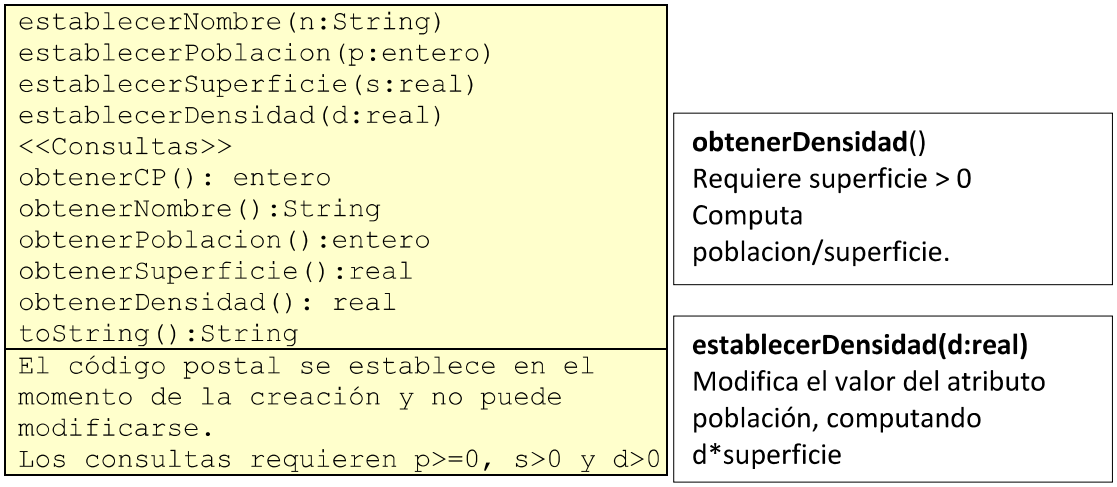
Caso de Estudio: Ciudad

*El Ministerio de Educación de la Provincia ofrece un Programa de Capacitación Docente para maestros de nivel Primario. Consideremos el problema de desarrollar un sistema para la gestión del Programa de Capacitación. El programa ofrece diferentes talleres para maestros. En cada **taller** se proponen **actividades** que luego los **maestros** desarrollan en la **escuela** en la que trabajan. El sistema de gestión requiere administrar la **inscripción** de los maestros en los talleres, el control de **asistencia**, la asignación de **capacitadores**, el registro de las **evaluaciones** de los maestros y las escuelas. Al Ministerio le interesa realizar un análisis comparativo del Programa de Capacitación en cada **ciudad** y en cada **barrio** en particular. El análisis considera la **población** de cada barrio y cada ciudad, la **densidad** de la ciudad, la **cantidad de alumnos** en escuelas públicas y en escuelas privadas, en cada barrio. El sistema de gestión para el Programa de Capacitación docente requiere entonces modelar al conjunto de ciudades, barrios, maestros, talleres, capacitadores, inscripciones, etc.*

El modelo completo va a incluir una clase para cada conjunto de entidades. A partir del modelo completo, es posible implementar cada clase en particular. El siguiente diagrama modela a la clase `Ciudad`:

Ciudad
<<Atributos de instancia>> CP: entero nombre:String poblacion : entero superficie: real
<<Constructor>> Ciudad (c : entero) <<Comandos>>

[1] Algunos autores utilizan los términos *encapsulamiento* y *oscurecimiento de información* para referenciar a dos conceptos relacionados pero diferentes. El primero agrupa atributos y servicios, el segundo esconde la implementación.



El diseñador decidió que es relevante incluir atributos para el código postal, el nombre, la población y la superficie. En la caracterización interesa también conocer la densidad de cada ciudad, pero el diseñador resolvió que la densidad no se mantenga como un atributo, sino que se calcule como el cociente entre la población y la superficie.

De acuerdo a esta propuesta, en el momento que se crea una ciudad se establece únicamente el código, que no puede ser luego modificado. La clase brinda una consulta para obtener cada atributo y una más para obtener la densidad. Para las clases cliente, una ciudad tiene una densidad, más allá de que se mantenga en el estado interno o se compute.

La implementación de la clase `Ciudad` es:

```
public class Ciudad {
//Atributos de instancia
private int CP;
private int poblacion;
private float superficie;
private String nombre;
//Constructor
public  Ciudad (int cod )
{ CP = cod; poblacion = 0 ; superficie = 0 ;}
//Comandos
public void establecerNombre (String n)
{  nombre = n ; }
public void establecerPoblacion (int p )
{  poblacion = p ; }
public void establecerSuperficie ( float s)
{  superficie = s ;}
public void establecerDensidad ( float d)
{  poblacion = (int) (d*superficie) ;}
//Consultas
public String obtenerNombre()
{ return nombre;}
public int obtenerCP ()
{ return CP; }
public int obtenerPoblacion ()
{ return poblacion; }
public float obtenerSuperficie ()
{ return superficie; }
```

```
public double densidad ()
{ return poblacion/superficie; }
}
```

Los atributos se declaran como privados, quedan así escondidos dentro de la clase. Las clases clientes de `Ciudad` no pueden acceder directamente al estado interno de sus instancias.

La clase brinda cuatro comandos que permiten modificar el estado interno de un objeto de clase `Ciudad`. Si la clase cliente usa el servicio `establecerDensidad()` no se modifica el valor de este atributo, porque de hecho no se mantiene explícitamente, sino que se computa un nuevo valor para el atributo `poblacion`. El cómputo requiere conversión de tipos.

La clase `Ciudad` asume que sus clientes cumplen con sus responsabilidades. Si se envía el mensaje `obtenerDensidad()` a un objeto, antes de asignarse un valor a superficie, la ejecución terminará anormalmente. Una clase **confiable**, debería prevenir situaciones de error, más allá de sus propias responsabilidades. El **manejo de excepciones** permite justamente considerar estos casos.

Consideremos que una vez implementado el sistema se resuelve modificar la representación interna de la clase `Ciudad`.

Ciudad
<<Atributos de instancia>> CP: entero nombre:String superficie : real densidad: real
<<Constructor>> Ciudad (c : entero) <<Comandos>> establecerNombre(n:String) establecerPoblacion(p:entero) establecerSuperficie(s:real) establecerDensidad(d:real) <<Consultas>> obtenerCP(): entero obtenerNombre():String obtenerPoblacion():entero obtenerSuperficie():real obtenerDensidad(): real toString():String
El código postal se establece en el momento de la creación y no puede modificarse. Las consultas requieren p>=0, s>0 y d>0

obtenerPoblación()
Computa la parte entera de la expresión densidad*superficie

establecerPoblación(p:real)
Requiere superficie > 0.
Modifica el valor del atributo densidad, computando p/superficie.

La implementación de la clase `Ciudad` es ahora:

```
public class Ciudad {
//Atributos de instancia
private int CP;
private float densidad;
private float superficie;
private String nombre;
//Constructor
public  Ciudad (int cod )
```

```
{ CP = cod; poblacion = 0 ; superficie = 0 ;}
//Comandos
public void establecerNombre (String n)
{  nombre = n ; }
public void establecerPoblacion (int p )
{/*Requiere superficie > 0 */
densidad = superficie / p ; }
public void establecerSuperficie ( float s)
{  superficie = s ;}
public void establecerDensidad ( float d)
{  poblacion = d ;}
//Consultas
public String obtenerNombre()
{ return nombre;}
public int obtenerCP ()
{ return CP; }
public int obtenerPoblacion ()
{ return (int) (superficie*densidad); }
public float obtenerSuperficie ()
{ return superficie; }
public double densidad ()
{ return densidad; }
}
```

Si los atributos fueran visibles fuera de la clase `Ciudad`, sus clientes podrían haber accedido directamente al atributo `poblacion`. Al cambiar el diseño de la clase proveedora, tendría que modificarse también cada una de las clases cliente. Siguiendo los lineamientos de la programación orientada a objetos, las clases clientes de `Ciudad` solo acceden a su interfaz, que no cambió aun cuando se modificaron los atributos.

Representaciones alternativas para estructuras lineales y homogéneas

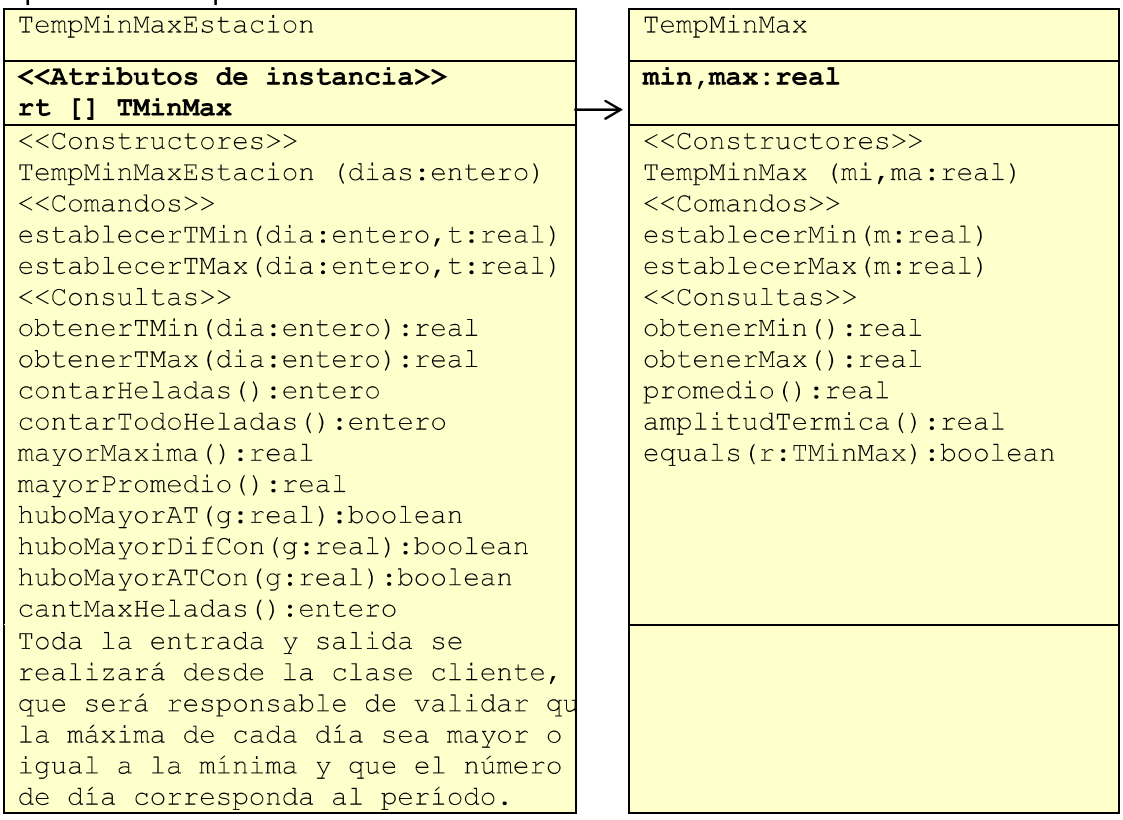
Cuando una clase modela una colección de elementos, por lo general existen diferentes estructuras alternativas para representar tanto a los elementos como a la colección misma. Si cada servicio de la clase mantiene la signatura y funcionalidad, los cambios en la representación de los datos no modifican la interfaz, aunque sí va a ser necesario modificar el código.

En este libro los casos de estudio que requieren representar colecciones de elementos pueden modelarse con estructuras lineales, homogéneas y ordenadas. Una estructura es homogénea si todos los elementos son del mismo tipo. En una estructura lineal cada elemento tiene un predecesor, excepto el primero, y un sucesor, excepto el último. La mayoría de los lenguajes de alto nivel permiten crear arreglos para definir estructuras homogéneas, lineales y ordenadas. En una estructura ordenada cada una de elemento puede accederse de acuerdo a su posición.

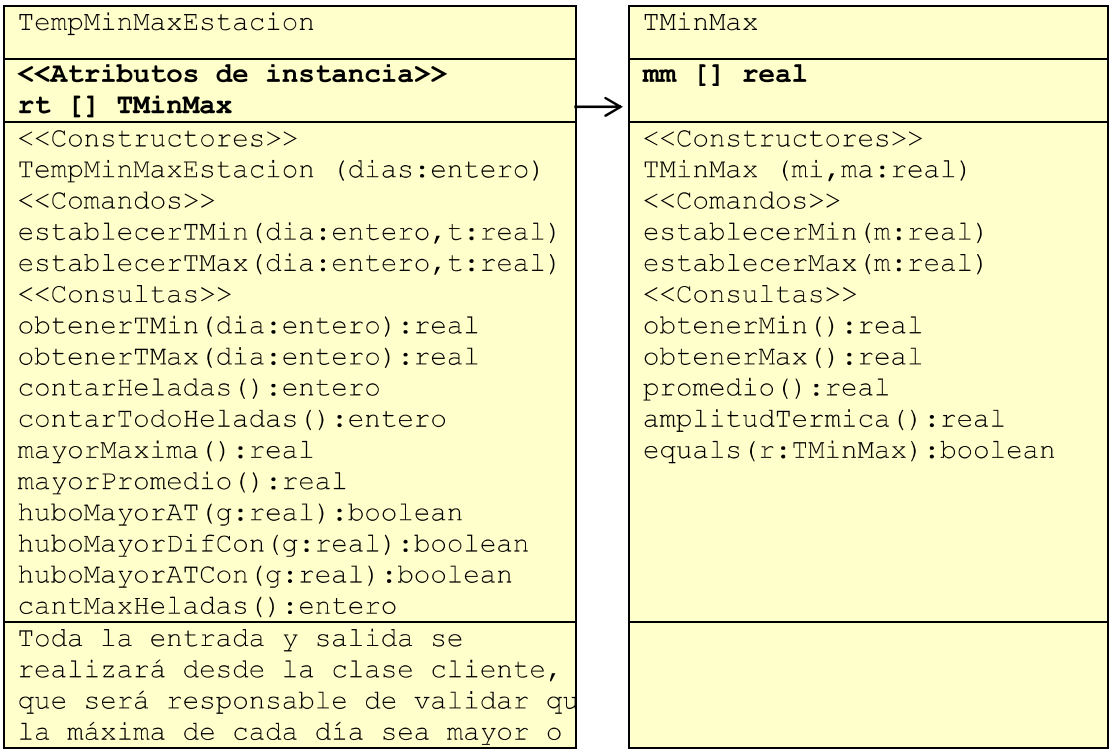
Caso de Estudio Estación Meteorológica

En una estación meteorológica se almacenan las temperaturas mínimas y máximas de cada día de un período y se computan algunos valores estadísticos. Antes de registrarse los valores se conoce la cantidad de días del período y todos los valores han sido almacenados en el momento que se computan estadísticas. El primer día del período se denota con el número 1 y así correlativamente, el último día del período corresponde al número n.

El siguiente diagrama especifica una **primera alternativa** de diseño, consistente con los requerimientos capturados:



Una **segunda alternativa** de diseño, para la misma especificación de requerimientos, puede ser:



igual a la mínima y que el número de día corresponda al período.	

Cada clase conforma un módulo o unidad de programación, a partir de los cuales se crean objetos y se les envían mensajes, sin conocer su implementación. El cambio en `TMinMax` no afecta `TempMinMaxEstacion`.

El **tercer diseño** alternativo es:

TempMinMaxEstacion
<<Atributos de Instancia>> **min [] real** **max [] real**
<<Constructores>> TempMinMaxEstacion (dias:entero) <<Comandos>> establecerTMin(dia:entero,t: real) establecerTMax(dia:entero,t: real) <<Consultas>> obtenerTMin(dia:entero):real obtenerTMax(dia:entero):real contarHeladas():entero contarTodoHeladas():entero mayorPromedio():real huboMayorAT(g:real):boolean huboMayorDifCon(g:real):boolean huboMayorATCon(g:real):boolean cantMaxHeladas():entero
Toda la entrada y salida se realizar desde la clase cliente, que será responsable de validar que la máxima de cada día sea mayor o igual a la mínima y que el número de día corresponda al período.

En las dos primeras alternativas las componentes de la colección son objetos de una clase especificada por el diseñador. Es decir, las componentes de la estructura de datos son también estructuras de datos. En la tercera alternativa, la clase `TempMinMaxEstación` esconde a dos estructuras de datos, cuyas componentes son valores elementales.

Cualquiera sea el diseño, las responsabilidades de la clase `TempMinMaxEstacion` son:

```
El primer día se referencia con 1.
Los servicios que reciben un parámetro que corresponde al día
requieren que este valor sea válido
La estructura está completa,  la cantidad de elementos se define en el
momento de la creación y es siempre mayor a 0
```

El propósito de cada servicio de la clase `TempMinMaxEstacion` es:

`contarHeladas()`: cuenta la cantidad de días del período en los que heló en algún momento del día.

`contarTodoHeladas()`:cuenta la cantidad de días del período en los que heló todo el día.

`mayorMaxima()`: computa la mayor temperatura máxima

`mayorPromedio()`: computa el mayor promedio entre la mínima y la máxima

`huboMayorAT(g:real)` : retorna true si y solo sí hubo al menos un día con amplitud térmica mayor a g.

`huboMayorATCon(g:real)`: retorna true sí y solo sí hubo al menos dos días consecutivos con amplitud térmica mayor a g.

`huboMayorATnCon(g:real,n:entero)`: retorna true sí y solo sí hubo al menos n días consecutivos con amplitud térmica mayor a g.

`huboMayorATNCon(g:real,n:entero)`: retorna true sí y solo sí hubo exactamente n días consecutivos con amplitud térmica mayor a g.

`huboMayorDifCon(g:real)`: retorna true sí y solo sí hubo al menos dos días consecutivos entre los cuales la diferencia absoluta entre las amplitudes térmicas fue mayor que g.

`cantMaxHeladas()`: retorna la duración en días del período más largo con heladas en días consecutivos.

La implementación de cada método requiere diseñar un algoritmo que va a quedar escondido en la clase. En el diseño de cada uno seguimos los lineamientos de la programación estructurada.

La representación de los datos es transparente para las clases relacionadas con `TempMinMaxEstacion`. De hecho, es posible elegir una de las tres alternativas propuestas y luego cambiarla por otra de las representaciones, sin afectar a las clases cliente, ya que no cambia la interfaz ni las responsabilidades.

Como el primer día del período se referencia con 1, si las temperaturas se almacenan en un arreglo va a ser necesario establecer un mapeo entre el día y el subíndice. Así, el día uno se almacena en la componente del arreglo que corresponde al subíndice 0, el día 2 en el subíndice 1 y así siguiendo.

Patrones de diseño de algoritmos

Con frecuencia los recorridos sobre las estructuras de datos responden a **patrones de algoritmos**, independientes del tipo de los elementos de la estructura. Por ejemplo, es habitual recorrer una estructura de datos lineal y homogénea para contar la cantidad de elementos que satisfacen una propiedad, el algoritmo puede ser entonces:

```
Algoritmo contar
DS contador
contador se inicializa en 0
para cada elemento de la estructura
  si el elemento cumple la propiedad
    incrementar el contador
```

El algoritmo `contar` describe el **comportamiento abstracto** de los servicios `contarHeladas` y `contarTodoHeladas`. El algoritmo puede **refinarse** en versiones más específicas:

```
Algoritmo cantHeladas
DS contador
contador se inicializa en 0
para cada día del período
  si temperatura minima <= 0
    incrementar el contador
```

```
Algoritmo cantTodoHeladas
DS contador
contador se inicializa en 0
para cada día del período
  si temperatura maxima <= 0
    incrementar el contador
```

Los algoritmos que implementan los servicios provistos por una clase, están **escondidos** dentro de la clase. Si se modifican, en tanto la signatura no cambie, la modificación es **transparente** para las clases cliente. Por ejemplo, el cómputo de la cantidad de componentes que verifican una propiedad dentro de un período, puede plantearse recursivamente:

Caso trivial: En un período vacío la cantidad de componentes que verifican una propiedad es 0.

Caso recursivo: En un período no vacío, la cantidad de componentes que verifican una propiedad es la cantidad de componentes que verifican la propiedad en el período que resulta de no considerar el último día, más 1 si el último día verifica la propiedad.

Caso recursivo: En un período no vacío, la cantidad de componentes que verifican una propiedad es la cantidad de componentes que verifican la propiedad en el período que resulta de no considerar el último día, si el último día NO verifica la propiedad.

Las clases relacionadas con `TempMinMaxEstacion` no *saben* si `contarHeladas` se implementa a partir de un algoritmo iterativo o un planteo recursivo.

Otro recorrido habitual sobre una estructura consiste en hallar el mayor elemento, de acuerdo a una relación de orden establecida. En este caso el algoritmo es:

```
Algoritmo mayor
DS mayor
mayor ← primer elemento
para cada elemento de la estructura a partir del segundo
  si elemento > mayor
    mayor ← elemento
```

El algoritmo propone un recorrido exhaustivo, todos los elementos del arreglo deben ser considerados para computar el mayor. En cada iteración se accede a un elemento particular, comenzando por el segundo. Nuevamente este patrón general es una **abstracción**, que puede refinarse para cada problema específico:

```
Algoritmo mayorPromedio
DS mayor
mayor ←promedio del primer día
para cada día del período
  si promedio del día > mayor
    mayor ← promedio del día
```

```
Algoritmo mayorMaxima
DS mayor
mayor ← máxima del primer día
para cada día del período
  si la máxima del día > mayor
    mayor ←máxima del día
```

Observemos que estamos utilizando una notación informal, en contraposición a la sintaxis rigurosa de un lenguaje de programación. Podemos usar el símbolo ← para denotar asignación o utilizar una notación verbal como “inicializar mayor con el promedio del primer día” o “asignar el promedio del primer día a mayor”

Para decidir si algún elemento de la estructura verifica una propiedad no es necesario hacer un recorrido exhaustivo:

```
Algoritmo verifica
```

```
DS verifica
verifica ← falso
para cada elemento de la estructura y mientras no verifica
  si el elemento actual verifica la propiedad
    verifica ← verdadero
```

Para el caso específico de decidir si la amplitud térmica fue mayor a `g` en algún día del período, el patrón se refina:

```
Algoritmo huboMayorAG
DE g
DS verifica
verifica ← falso
para cada día del período y mientras no verifica
  si la amplitud térmica del día > g
    verifica ← verdadero
```

El mismo patrón puede refinarse un poco más para modelar la solución de decidir si hubo dos días seguidos con amplitud térmica mayor a un valor g.

```
Algoritmo huboMayorATCon
DE g
DS verifica
verifica ← falso
para cada día del período excepto el último y mientras no verifica
  si la amplitud térmica del día > g y
     la amplitud térmica del día siguiente > g
    verifica ← verdadero
```

El algoritmo para decidir si hubo al menos n días consecutivos con temperaturas mayores a `g` implica nuevamente un recorrido no exhaustivo:

```
Algoritmo huboMayorATnCon
DE g,n
DS verifica
inicializa contador en 0
para cada día del período y mientras contador < n
  si la amplitud térmica del día > g
    incrementa el contador
  sino
    inicializa contador en 0
verifica es verdadero si contador = n
```

Si el problema plantea decidir si hubo exactamente `n` días que verifiquen la propiedad:

```
Algoritmo huboMayorATNCon
DE g,n
DS verifica
inicializa contador en 0
para cada día del período y mientras contador <= n
  si la amplitud térmica del día > g
    incrementa el contador
  sino
    inicializa contador en 0
verifica es verdadero si contador = n
```

Ejercicio: Diseñe algoritmos para `huboMayorDifCon(g:real)` y

`cantMaxHeladas()`.

Implementaciones alternativas

En los dos primeros diseños alternativos propuestos, las clases `TempMinMaxEstacion` y `TMinMax` están **asociadas**. En ambos casos la implementación parcial de `TempMinMaxEstacion` a partir de los algoritmos propuestos, será:

```
class TempMinMaxEstacion {
//Atributo de instancia
  private TMinMax [] rt;
//constructor
public TempMinMaxEstacion (int dias){
/*Cada elemento del arreglo representa un día del período*/
  rt = new TMinMax [dias];
  for (int dia = 0;dia < dias;dia++)
    rt[dia] = new TMinMax(0,0);}
//Comandos y Consultas triviales
/*Requiere que la clase Cliente haya controlado que max > min*/
public void establecerTempMin (int dia, float t){
  rt[dia-1].establecerMin(t);}
public void establecerTempMax (int dia,float t){
  rt[dia-1].establecerMax(t);}
public float obtenerTempMin (int dia){
  return rt[dia-1].obtenerMin();}
public float obtenerTempMax (int dia){
  return rt[dia-1].obtenerMax();}
// Consultas
public int cantDias(){
  return rt.length;}
public int cantHeladas(){
/*Calcula la cantidad de días con temperatura mínima menor a 0*/
  int cant=0;
  for (int dia=0;dia<cantDias();dia++)
    if (rt[dia].obtenerMin()<0) cant++;
  return cant;}
public float mayorPromedio(){
/*Computa el mayor promedio entre la maxima y la minima
Requiere que el periodo tenga al menos 1 día*/
  float mayor = rt[0].promedio();
  float m;
  for (int dia=1;dia<cantDias();dia++){
    m = rt[dia].promedio();
    if (m > mayor)
      mayor = m;}
  return mayor;}
}
```

Ejercicio: Complete la implementación de TempMinMaxEstacion.

La clase `TempMinMaxEstacion` se implementa sin conocer el código de `TMinMax`, solo conocemos la interfaz de esta clase, especificada en el diagrama. Para el primer diagrama de `TMinMax:`

TMinMax
min,max:real

La implementación de puede ser:

```
class TMinMax{
//Atributos de instancia
private float minima;
```

```
private float maxima;
//Constructor
public TMinMax (float mi,float ma){
//Requiere ma >= mi
  minima = mi;
  maxima = ma;}
//Comandos
public void establecerMin(float m){
  minima = m;}
public void establecerMax(float m){
  maxima = m;}
//Consultas
public float obtenerMin(){
  return minima;}
public float obtenerMax(){
  return maxima;}
public float promedio(){
  return (minima+maxima)/2;}
public float amplitudTermina(){
  return maxima-minima;}
public boolean equals(TMinMax r){
  return minima == r.obtenerMin() &&
         maxima == r.obtenerMax();}}
```

Consideremos ahora el **segundo diseño**, en el cual se modifica la clase `TMinMax`:

TMinMax
mm [] real

La implementación es:

```
class TMinMax{
//Atributos de instancia
private float [] mm;
//Constructor
public TMinMax (float mi,float ma){
//Requiere ma >= mi
  mm = new float [2];
  mm[0] = mi;
  mm[1] = ma;}
//Comandos
public void establecerMin(float m){
  mm[0] = m;}
public void establecerMax(float m){
  mm[1] = m;}
//Consultas
public float obtenerMin(){
  return mm[0];}
public float obtenerMax(){
  return mm[1];}
public float promedio(){
  return (mm[0]+mm[1])/2;}
public float amplitudTermina(){
  return mm[1]-mm[0];}
public boolean equals(TMinMax r){
  return mm[0] == r.obtenerMin() &&
         mm[1] == r.obtenerMax();}
}
```

Notemos que el cambio no tendría impacto en la clase `TempMinMaxEstacion`, ya que esta clase no accede directamente a los atributos de `TMinMax`.

Los atributos de instancia en la tercera alternativa de diseño son:

TempMinMaxEstacion
min [] real max [] real

En este caso sí es necesario modificar el código de `TempMinMaxEstacion`. La implementación parcial es entonces:

```
class TempMinMaxEstacion {
//Atributo de instancia
  private float [] min;
  private float [] max;
//constructor
public TempMinMaxEstacion (int dias){
//Cada elemento del arreglo representa un día del período
  min = new float [dias];
  max = new float [dias];}
//Comandos y Consultas triviales
// Requiere que la clase Cliente haya controlado que max > min
public void establecerTempMin (int dia, float t){
  min[dia-1] = t;}
public void establecerTempMax (int dia,float t){
  max [dia-1] = t;}
public float obtenerTempMin (int dia){
  return min[dia-1];}
public float obtenerTempMax (int dia){
  return max[dia-1];}
// Consultas
public int cantDias(){
  return min.length;}
public int cantHeladas(){
//Calcula la cantidad de días con temperatura mínima menor a 0
  int cant=0;
  for (int dia=0;dia<cantDias();dia++)
    if (min[dia]<0) cant++;
  return cant;}
public float mayorPromedio(){
/*Computa el mayor promedio entre la maxima y la minima
Requiere que el periodo tenga al menos 1 día*/
  float mayor = (min[0]+max[0])/2;
  float m;
  for (int dia=1;dia<cantDias();dia++){
    m = (min[dia]+max[dia])/2;
    if (m > mayor)
      mayor = m;}
  return mayor;}
}
```

Ejercicio: Completar la implementación de esta versión de `TempMinMaxEstación`.

A pesar del cambio de representación de los datos, las dos implementaciones corresponden a los mismos patrones de diseño para los algoritmos.

Encapsulamiento y Clases relacionadas

La modificación de la representación de los datos en una clase implica reescribir el código de algunos o incluso todos los servicios. Si los atributos están escondidos, el cambio en la representación es transparente para las clases asociadas, en tanto no cambie la función de cada servicio y las responsabilidades.

Las clases clientes de `TempMinMaxEstacion` solo pueden acceder a los atributos a través de los servicios provistos en la interfaz. Si cambia la representación, el cambio no impacta sobre las clases relacionadas.

En particular, es posible implementar una misma clase tester para verificar las dos versiones de `TempMinMaxEstación`. La clase `TestTempMinMaxEstacion` verifica el método `mayorPromedio` para tres casos de prueba establecidos con valores fijos.

```
class TestTempMinMaxEstacion {
public static void main(String[] args) {
/* Verifica el método mayorPromedio para tres casos de prueba
significativos*/
 TempMinMaxEstacion est;
 int cantD =7;
 est =genTempMinMaxEst() ;
 System.out.println("Muestra la estación  ");
 mostrarTempMinMaxEst(est) ;
 System.out.println(" El mayor promedio es "+est.mayorPromedio());
 // Caso de prueba: El mayor promedio se produjo en el primer día
  est.establecerTempMin(1,15);
  est.establecerTempMax(1,25);
  mostrarTempMinMaxEst(est) ;
  System.out.println(" El mayor promedio es "+est.mayorPromedio());
 // Caso de prueba: El mayor promedio se produjo el último día
  est.establecerTempMin(est.cantDias(),20);
  est.establecerTempMax(est.cantDias(),30);
  mostrarTempMinMaxEst(est) ;
  System.out.println(" El mayor promedio es "+est.mayorPromedio());}
public static TempMinMaxEstacion genTempMinMaxEst( ){
  TempMinMaxEstacion e;
  int cantD =7;
  e=new TempMinMaxEstacion(cantD);
  for (int dia= 1; dia<e.cantDias()-1; dia++){
    e.establecerTempMin(dia,-1);
    e.establecerTempMax(dia,dia+8);}
  e.establecerTempMin(3,5);
  e.establecerTempMax(3,15);
  e.establecerTempMin(e.cantDias(),5);
  e.establecerTempMax(e.cantDias(),11);
  return e;}
public static void mostrarTempMinMaxEst( TempMinMaxEstacion est){
 for (int dia=1; dia<=est.cantDias(); dia++)
     System.out.println(" "+est.obtenerTempMin(dia)+
                        " "+est.obtenerTempMax(dia) );}
}
```

Tanto si los valores están establecidos en la clase tester, son leídos por consola o desde un archivo, todos los datos están almacenados en el arreglo antes de que se realice cualquier procesamiento.

Ejercicio: Complete la clase tester con mensajes adecuados para probar los servicios provistos por `TempMinMaxEstacion`

Ejercicio: Modifique la clase tester para que los valores se lean de un archivo secuencial.

Ejercicio: Implemente el siguiente modelo que propone otra alternativa para modelar el mismo problema y verifique la clase con el mismo tester. La interfaz de la clase y las responsabilidades no se modifican.

TempMinMaxEstacion
minmax [] [] real

En la estación meteorológica la cantidad de componentes de la estructura de datos corresponde a la cantidad de días del período y se conoce en el momento que se crea el objeto de clase `TempMinMaxEstacion`. Las temperaturas de cada día se almacenan antes de que comience el procesamiento, es decir la estructura está completa. En muchas aplicaciones el procesamiento se realiza aun cuando la estructura no está completa, esto es, las consultas y modificaciones de los datos se entrelazan, como veremos más adelante, utilizando otro caso de estudio.

Abstracción y programación orientada a objetos

El análisis y diseño orientado a objetos demanda un proceso de abstracción a partir del cual se identifican los objetos del problema y se agrupan en clases. Una clase es una abstracción, un patrón que caracteriza los atributos y el comportamiento de un conjunto de objetos.

En la implementación, una clase que establece atributos y servicios define un **tipo de dato** a partir del cual es posible crear instancias. El conjunto de valores queda determinado por el tipo de los atributos, el conjunto de operaciones corresponde a los servicios provistos por la clase. Si la representación de los datos está escondida, la clase define un **tipo de dato abstracto (TDA)**.

La definición de tipos de datos abstractos es un recurso importante porque favorece la reusabilidad y permite reducir el impacto de los cambios. La modificación de una clase que define un tipo de dato abstracto, puede realizarse sin afectar a las clases que crean instancias del tipo.

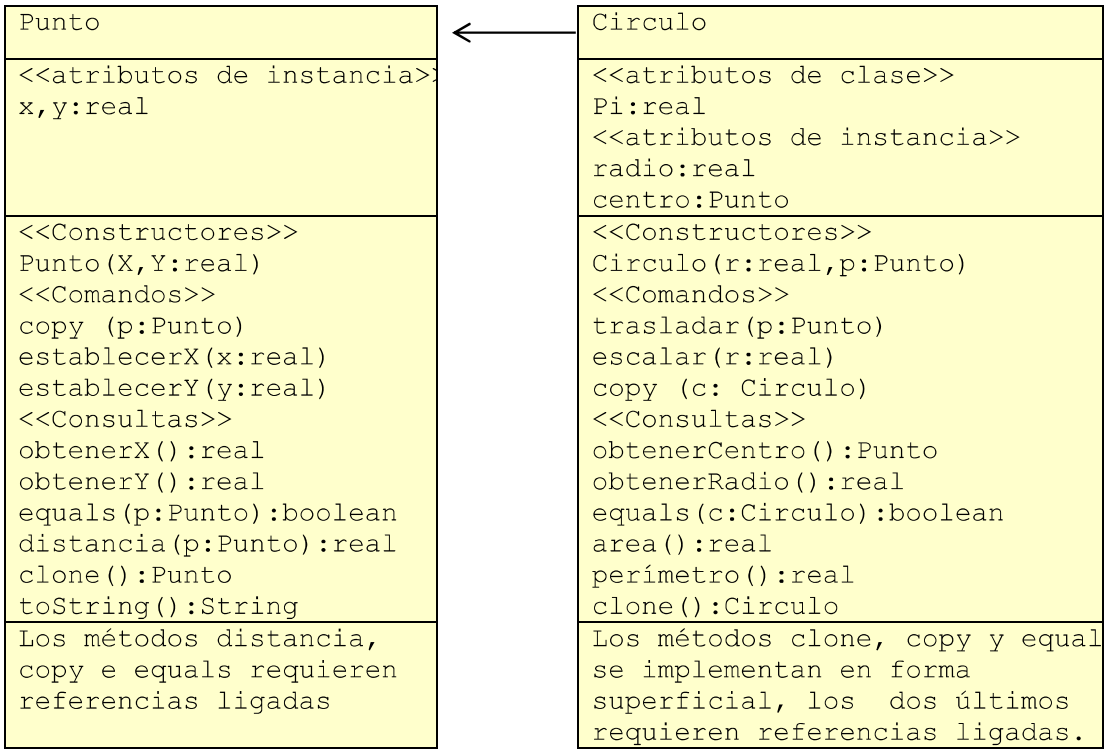
En Java la definición de clases y los modificadores de accesos, permiten que el programador defina nuevos tipos de datos abstractos a partir de los cuales se crean instancias. Todas las clases implementadas en los casos de estudio propuestos hasta el momento definen tipos de datos abstractos. A continuación presentamos varios casos de estudio, cada uno de los cuales define TDA asociados.

TDA y clases relacionadas

El siguiente caso de estudio ilustra como un cambio en la implementación del TDA `Circulo`, es transparente para las clases relacionadas por asociación o dependencia.

Caso de Estudio: Círculo y Punto

Un sistema de software didáctico de geometría para prescolar brinda facilidades para que los niños identifiquen y clasifiquen figuras geométricas. En el desarrollo del sistema el diseñador ha elaborado un diagrama con más de 50 clases, entre ellas Circulo y Punto, definidos como TDA. Implemente los TDA Circulo y Punto modelados en el siguiente diagrama:



Las clases `Circulo` y `Punto` están relacionadas por asociación. La clase `Circulo` tiene un atributo de clase `Punto`. El constructor de `Circulo` y el método `trasladar` reciben un parámetro de clase `Punto`. El método `obtenerCentro` retorna como resultado un objeto de clase `Punto`. Existe también una relación de dependencia entre `Circulo` y `Punto`.

La implementación de la clase `Punto` es:

```
class Punto{
//Atributos de Instancia
private float x,y;
//Constructores
public Punto (float x,float y){
  this.x = x;  this.y = y;}
public void establecerX (float x){
  this.x = x;}
public void establecerY (float y){
  this.y = y;}
public void copy (Punto p){
//Require p ligada
  x = p.obtenerX();
  y = p.obtenerY();}
//Consultas
public float obtenerX(){
  return x;}
public float obtenerY(){
```

```
  return y;}
public double distancia(Punto p) {
//Require p ligada
    float dx= x - p.obtenerX();
    float dy= y - p.obtenerY();
    return Math.sqrt(dx*dx+dy*dy);}
public  boolean equals (Punto p){
//Require p ligada
  return x==p.obtenerX() && y==p.obtenerY();}
public Punto clone(){
 return new Punto(x,y);}
public String toString(){
  return "("+x+","+y+")";}
}
```

La clase `Punto` define un tipo de dato abstracto a partir del cual es posible crear instancias. El conjunto de valores es el conjunto de pares de valores de tipo `float`. Cada servicio provisto por la clase corresponde a una operación provista por el tipo.

La clase `Circulo` mantiene un atributo de instancia del TDA `Punto`, como ilustra la siguiente implementación:

```
class Circulo {
//Atributos de clase
private static final float Pi= (float) 3.1415;
//Atributos de Instancia
private float radio;
private Punto centro;
//Constructor
public Circulo(float r, Punto p){
//Requiere p ligada
  radio = r;
  centro = p;}
//Comandos
public void trasladar(Punto p){
//Require p ligada
  centro = p;}
public void escalar(float r){
  radio = r;}
public void copy (Circulo c){
//Require c ligada
  radio = c.obtenerRadio();
  centro = c.obtenerCentro();}
//Consultas
public Punto obtenerCentro(){
  return centro;}
public float obtenerRadio(){
  return radio;}
public float perimetro(){
  return obtenerRadio()*2*Pi;}
public double area(){
  return Pi * obtenerRadio()* obtenerRadio();}
public boolean equals(Circulo c){
//Require c ligada
return obtenerRadio()== c.obtenerRadio() &&
```

```
        obtenerCentro() == c.obtenerCentro();}
public Circulo clone (){
  return new Circulo (obtenerRadio(),obtenerCentro());}
}
```

La clase `Circulo` también define un tipo de dato abstracto a partir del cual es posible crear instancias. La clase `Circulo` es cliente de la clase `Punto` y a su vez es proveedora de servicios para las clases que la usan.
Consideremos ahora el siguiente diseño alternativo para la clase *`Circulo`*:

Circulo
<<atributos de instancia>> centro:Punto punto:Punto
<<Constructores>> Circulo(r:real,p:Punto) <<Comandos>> trasladar(p:Punto) escalar(r:real) copy (c: Circulo) <<Consultas>> obtenerCentro():Punto obtenerRadio():real equals(c:Circulo):boolean area():real perímetro():real clone():Circulo

En este diseño, un objeto de clase `Circulo` se modela a través de dos atributos de instancia que corresponden al TDA `Punto`. Uno corresponde al centro y otro a un punto cualquiera de la circunferencia.

Observemos que el cambio en la representación no modifica la signatura de ninguno de los servicios. No se modificó la interfaz de la clase, aunque sí es necesario implementar de manera diferente varios de los servicios. Como los atributos están escondidos, el cambio no afecta a la clase `Punto`, proveedora de la clase `Circulo`, ni a las clases clientes de la clase `Circulo`.

El constructor de `Circulo` se implementa ahora como:

```
public Circulo (float r, Punto p){
//Require p ligada
  centro = p;
  punto = new Punto (centro.obtenerX(), centro.obtenerY()+r);}
```

El método `obtenerRadio()` se modifica porque el radio ya no es un atributo de la clase `Circulo`.

```
public float obtenerRadio(){
  return centro.distancia(punto);}
```

Las clases dependientes o asociadas a la clase `Circulo` no perciben el cambio de diseño ni de la implementación porque solo acceden a la interfaz de la clase proveedora.
Ejercicio

Complete la clase Circulo para este diseño alternativo, analizando qué métodos es necesario modificar.

Defina una única clase tester que permita verificar los servicios de las dos implementaciones.

TDA y estructuras homogéneas, lineales y con acceso directo

En la resolución de problemas de mediana y gran escala el diseño de las estructuras de datos es un tema relevante. Una estructura de datos es una colección de datos organizados de alguna manera. La organización está ligada a cómo van accederse las componentes.

Una estructura es homogénea si todas las componentes son del mismo tipo. En una estructura lineal cada componente tiene un predecesor, excepto el primero, y un sucesor, excepto el último. Una estructura tiene acceso directo si cada una de sus componentes puede accederse sin necesidad de acceder a las que la preceden o suceden.

La mayoría de los lenguajes de alto nivel permiten crear arreglos para definir estructuras homogéneas, lineales y con acceso directo.

Caso de Estudio TDA Matriz-Vector

Un sistema de software didáctico de álgebra elemental brinda facilidades para que los jóvenes operen con matrices y vectores. En el desarrollo del sistema el diseñador ha elaborado un diagrama con más de 50 clases, entre ellas Matriz y Vector, definidos como TDA. Implementar los TDA `Matriz` **y** `Vector` a partir del siguiente diagrama:

Matriz
mr [][] real
<<Constructor>> Matriz(nf,nc:entero) <<Comandos>> establecerElem (f,c:entero, elem:real) copy(m:Matriz) establecerIdentidad () invertirFilas(f1,f2:entero) xEscalar(r:real) <<Consultas>> existePos(f,c:entero):boolean obtenerNFil ():entero obtenerNCol ():entero obtenerElem (f,c:entero):real clone():Matriz equals(m:Matriz): boolean esCuadrada ():boolean esIdentidad():boolean esTriangularSuperior():boolean eSimetrica():boolean esRala():boolean cantElem (elem:real):entero mayorElemento ():real filaMayorElemento ():entero vectorMayores ():Vector suma (m:Matriz):Matriz producto(m:Matriz):Matriz transpuesta():Matriz

Vector
v [] real
<<Constructor>> Vector(n:entero) <<Comandos>> establecerElem (f:entero, elem:real) copy(m:Vector) xEscalar(r:real) <<Consultas>> existePos(f:entero):boolean obtenerN ():entero obtenerElem (f:entero):real clone():Vector equals(m:Vector): boolean mayorElemento ():real productoEscalar (m:Vector): real productoVectorial(m:Vector): Vector

Todos los servicios que reciben objetos como parámetros requieren referencias ligadas.	Todos los servicios que reciben objetos como parámetros requieren referencias ligadas.

La clase `Matriz` depende de la clase `Vector` porque el servicio `VectorMayores()` retorna un objeto de clase `Vector`.

`establecerElem (f,c:entero, elem:real):` requiere `0<=f <obtenerNFil()` y `0<=c <obtenerNCol()`

`establecerIdentidad ():` requiere `obtenerNCol()` igual a `obtenerNCol()`

`invertirFilas(f1,f2:entero):` requiere `0<=f1,f2 <obtenerNFil()`

`obtenerElem (f,c:entero):` requiere `0<=f <obtenerNFil()` y `0<=c <obtenerNCol()`

`esRala():` retorna verdadero si más de la mitad de los elementos son igual a 0

`vectorMayores ():` genera un objeto de clase `Vector` en el cual el elemento i es el mayor de la fila i de la matriz.

El método `xEscalar` implementa el siguiente algoritmo:

```
Algoritmo xEscalar
DE r
para cada posición i,j
   mr_{i,j} = mr_{i,j}*r
```

Observe que en el método `copy`, `m` es un objeto de clase `Matriz` y `mr` es un arreglo de dos dimensiones. Para acceder a los elementos de `m` se usan los servicios provistos por la clase, mientras que los elementos de `mr` se accedan con subíndices.

Un planteo recursivo para decidir si una matriz es la matriz identidad puede ser:

Caso trivial

Una matriz **mr** de 1x1 es la matriz identidad si su único elemento **mr**[0,0] es 1

Caso recursivo

Una matriz **mr** de nxn con n > 1 es la matriz identidad si

mr[i,n-1] = 0 para 0<= i < n-1

mr[n-1,j] = 0 para 0<= j < n-1

mr[n-1,n-1] = 1

y la matriz **mr'** de (n-1)x(n-1) es la matriz identidad

Propondremos la implementación de algunos de los servicios de la clase. Queda pendiente como ejercicio completar el código de `Matriz` y desarrollar completa la clase `Vector`.

```
class Matriz {
private float[][] mr;
// Constructor
public Matriz(int nfil,int ncol){
  mr=new float[nfil][ncol];}
//Consultas
public int obtenerNFil ()    {
  return mr.length; }
public int obtenerNCol ()    {
  return mr[0].length; }
public boolean existePos(int f, int c) {
  return (f>=0 && f < obtenerNFil()) &&
           (c>=0 && c < obtenerNCol());}
public float obtenerElem(int f,int c){
//Requiere f y c consistentes
  return mr[f][c];}
//Comandos
public void establecerElemento (int f,int c,
                  float x) {
//Requiere f y c consistentes
     mr[f][c]= x;}
public void establecerIdentidad () {
//Requiere que el número de filas sea igual al de columnas
    iniMatriz();
    for (int j=0;j<obtenerNCbol();j++)
      mr[j][j] = 1;    }
public void iniMatriz () {
  for (int i=0;i<obtenerNFil();i++)
    for (int j=0;j<obtenerNCol();j++)
       mr[i][j] = 0;    }
public void xEscalar (float r) {
//Multiplica cada elemento de la matriz por r
 for (int i=0;i<obtenerNFil();i++)
   for (int j=0;j<obtenerNCol();j++)
      mr[i][j] = mr[i][j] * r;}
public void copy (Matriz m) {
/*Crea un nuevo objeto para el atributo de instancia con el nro de
filas y columnas de m */
 mr = new float[m.obtenerNFil()][m.obtenerNCol()];
 for (int i=0;i<obtenerNFil();i++)  {
   for (int j=0;j<obtenerNCol();j++)
         mr[i][j] = m.obtenerElem(i,j); } }
public int cantElem (float ele)  {
    int cant = 0;
     for (int i=0;i< obtenerNFil();i++)
       for (int j=0;j< obtenerNCol();j++)
         if (mr[i][j] == ele) cant++ ;
    return cant;}
public boolean esCuadrada () {
   return (obtenerNFil() == obtenerNCol()) ; }
```

```
public boolean esIdent () {
  if (!esCuadrada())return false;
  else return esIdentidad(obtenerNFil());}
private boolean esIdentidad (int n) {
 if (n == 1)return (mr[0][0] == 1);
 else
   return (mr[n-1][n-1] == 1 &&
           esCeroFila(n-1) &&
           esCeroColumna(n-1)&&
           esIdentidad(n-1));}
}
```

La verificación de los TDA `Matriz` y `Vector` requiere como siempre establecer un conjunto de casos de prueba para cada operación provista por el tipo. Los servicios de la clase `Matriz` asumen que cada clase cliente cumple con los compromisos establecidos en el contrato. Cuando no basta que una solución sea correcta sino que se pretende que el sistema sea confiable, cada clase puede prevenir errores provocados por el incumplimiento de los compromisos de las clases clientes. Una manera de implementar el mecanismo de prevención es el manejo de excepciones.

Caso de Estudio TDA Racional MatrizRacional y **VectorRacional**

Implementar los TDA `Racional`, `MatrizRacional` **y** `VectorRacional` definidos a partir del siguiente diagrama:

MatrizRacional
mr [][] Racional
<<Constructor>> MatrizRacional(nf,nc:entero) <<Comandos>> establecerElem (f,c:entero, elem:Racional) copy(m:MatrizRacional) establecerIdentidad () invertirFilas(f1,f2:entero) xEscalar(r:Racional) <<Consultas>> existePos(f,c:entero):boolean obtenerNFil ():entero obtenerNCol ():entero obtenerElem (f,c:entero):Racional clone():MatrizRacional equals(m:MatrizRacional): boolean esCuadrada ():boolean esIdentidad():boolean esTriangularSuperior():boolean eSimetrica():boolean esRala():boolean cantElem (elem:Racional):entero mayorElemento ():Racional filaMayorElemento ():entero vectorMayores ():VectorRacional suma (m:MatrizRacional):MatrizRacional producto(m:MatrizRacional):

VectorRacional
v []Racional
<<Constructor>> VectorRacional(n:entero) <<Comandos>> establecerElem (f:entero, elem:Racional) copy(m:Racional) xRacional(r:Racional) <<Consultas>> existePos(f:entero): boolean obtenerN ():entero obtenerElem (f:entero):Racional clone():VectorRacional equals(m:VectorRacional): boolean mayorElemento ():Racional productoEscalar (m:VectorRacional): Racional productoVectorial(m:Vector): Racional

MatrizRacional transpuesta():MatrizRacional
Todos los servicios que reciben objetos como parámetros requieren referencias ligadas. Las búsqueda y comparaciones se hacen por equivalencia

Todos los servicios que reciben objetos como parámetros requieren referencias ligadas.

Racional
numerador:entero denominador:entero
<<Constructor>> Racional (n,d:entero) <<Comandos>> establecerNumerador(n:entero) establecerDenominador(d:entero) copy(r:Racional) <<Consultas>> obtenerNumerador ():entero obtenerDenominador ():entero equals(r:Racional):boolean suma(r:Racional):Racional resta(r:Racional):Racional producto(r:Racional):Racional cociente(r:Racional):Racional
El denominador es siempre positivo E constructor y establecerDenominador requieren d>0 Dos números racionales son iguales s representan un mismo valor.

equals(…)
Computa verdadero si el objeto que recibe el mensaje representa un número racional equivalente al parámetro

En la clase `MatrizRacional`:

`establecerElem (f,c:entero, elem:real):` requiere `0<=f <obtenerNFil()` y `0<=c <obtenerNCol()`

`establecerIdentidad ():` requiere `obtenerNCol()` igual a `obtenerNCol()`

`invertirFilas(f1,f2:entero):` requiere `0<=f1,f2 <obtenerNFil()`

`obtenerElem (f,c:entero):` requiere `0<=f <obtenerNFil()` y `0<=c <obtenerNCol()`

`esRala():` retorna verdadero si más de la mitad de los elementos representan el valor cero (con cualquier denominador positivo)

`vectorMayores ():` genera un objeto de clase `VectorRacional` en el cual el elemento i es el mayor de la fila i de la matriz.

La implementación parcial de `Racional` es:

```
class Racional {
//El denominador es siempre mayor a 0
private int numerador,denominador;
public Racional(int n,int d){
//Requiere d > 0
```

```
  numerador = n;
  denominador = d;}
public int obtenerNumerador(){
  return numerador;}
public int obtenerDenominador(){
  return denominador;}
public boolean equals(Racional r){
    return numerador*r.obtenerDenominador()==
           denominador*r.obtenerNumerador(); }
public Racional suma (Racional r){
  int n = numerador*r.obtenerDenominador()+
          r.obtenerNumerador()* denominador;
  int d = denominador*r.obtenerDenominador();
  Racional s = new Racional(n,d);
  return s;}
}
```

La implementación parcial de la clase `MatrizRacional` es:

```
class MatrizRacional {
private Racional[][] mr;
// Constructor
public MatrizRacional(int nfil,int ncol){
  mr=new Racional[nfil][ncol];}
//Consultas
public int obtenerNFil ()    {
  return mr.length; }
public int obtenerNCol ()    {
  return mr[0].length; }
public boolean existePos(int f, int c) {
  return (f>=0 && f < obtenerNFil()) &&
         (c>=0 && c < obtenerNCol());}
public Racional obtenerElem(int f,int c){
//Requiere f y c consistentes
  return mr[f][c];}
// Comandos
public void establecerElemento (int f,int c,
                  Racional x) {
//Requiere f y c consistentes
     mr[f][c]= x;}
public void establecerIdentidad () {
//Requiere que el número de filas sea igual al de columnas
    iniMatrizRacional();
    for (int j=0;j<obtenerNCol();j++)
      mr[j][j] = new Racional(1,1);    }
public void iniMatrizRacional () {
  for (int i=0;i<obtenerNFil();i++)
    for (int j=0;j<obtenerNCol();j++)
       mr[i][j] = new Racional(0,0);    }
public void xEscalar (Racional r) {
//Multiplica cada elemento por r
 for (int i=0;i<obtenerNFil();i++)
   for (int j=0;j<obtenerNCol();j++)
      mr[i][j] = mr[i][j].producto(r);}
public void copy (MatrizRacional m) {
/*Crea un nuevo objeto con el nro de filas y columnas de m y
referencias a los mismos elementos */
```

```
 mr = new Racional[m.obtenerNFil()][m.obtenerNCol()];
 for (int i=0;i<obtenerNFil();i++)  {
   for (int j=0;j<obtenerNCol();j++)
         mr[i][j] = m.obtenerElem(i,j); }}
public int cantElem (Racional ele)    {
/*Cuenta la cantidad de elementos equivalentes a ele que mantiene la
matriz */
    int cant = 0;
     for (int i=0;i< obtenerNFil();i++)
       for (int j=0;j< obtenerNCol();j++)
         if (mr[i][j].equals(ele)) cant++ ;
    return cant;}
public boolean esCuadrada () {
   return (obtenerNFil() == obtenerNCol()) ; }
public boolean esIdent () {
  if (!esCuadrada())
    return false;
  else return esIdentidad(obtenerNFil());}
private boolean esIdentidad (int n) {
 if (n == 1)
   return (mr[0][0].equals(new Racional(1,1)));
 else
   return (mr[n-1][n-1].equals(new Racional(1,1)) &&
           esCeroFila(n-1) &&
           esCeroColumna(n-1)&&
           esIdentidad(n-1));}
}
```

Ejercicios:

Completar el código de `MatrizRacional` *y desarrollar completa la clase* `VectorRacional`. *Complete la implementación de la clase Racional.*

Implemente nuevamente la clase Racional considerando que la sección de responsabilidades establece:

```
El denominador es siempre positivo El
constructor y establecerDenominador
requieren d>0
El constructor almacena la
representación irreductible de cada
número racional.
```

Implemente una única clase tester que permita verificar los servicios de las dos implementaciones de Racional.

TDA y estructuras parcialmente ocupadas

La abstracción permite proponer soluciones similares para problemas en apariencia muy diferentes. Esas similitudes van a permitir reusar diseño y código, favoreciendo la productividad.

En los casos de estudio que presentamos a continuación se define una clase que encapsula una estructura homogénea, lineal, con acceso directo y parcialmente ocupada. Es decir, el constructor crea una estructura para mantener cierta cantidad máxima de componentes. Desde la clase cliente, se insertan y eliminan componentes de acuerdo a funcionalidad

especificada en el diseño. Toda la entrada y salida se hace desde la clase cliente, de la clase que encapsula a la estructura de datos.

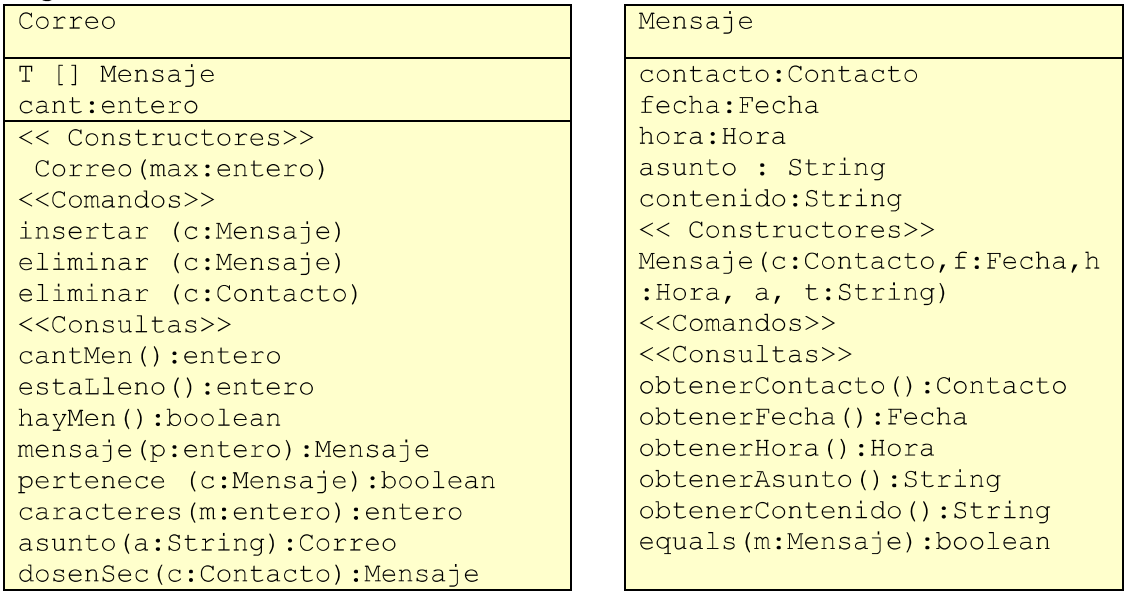
Utilizaremos el término **colección** para referirnos a una estructura con capacidad para mantener max elementos de un **tipo base** TB. En cada momento determinado solo n de los max elementos almacenan a un objeto de clase TB; más precisamente, referencian a un objeto del tipo base. Los n elementos ligados ocupan las primeras n posiciones de la estructura. De modo que las max-n componentes nulas, están comprimidas en las últimas posiciones del arreglo. La posición de cada elemento en una colección no tiene relevancia, de modo que una componente puede cambiar de posición. Por ejemplo, si un mensaje se asigna a una colección de mensajes de correo, pero su posición en la estructura no tiene significado en la aplicación.

Utilizaremos el término **tabla** para referirnos a una estructura con n elementos, cada uno de los cuales ocupa una posición que es significativa en la estructura. Por ejemplo, un micro se asigna a una unidad específica de un estacionamiento o un socio de un club tiene asignado un casillero particular que se identifica por su posición en la estructura que modela al conjunto de casilleros. Es decir las max-n componentes nulas pueden estar intercaladas con las componentes ligadas.

Caso de Estudio: Correo

Se desea mantener una secuencia de mensajes en la que pueden insertarse y eliminarse elementos, manteniendo siempre el orden de acuerdo en el cual se insertaron. También es posible recuperar un mensaje dada su posición en la secuencia, eliminar todos los mensajes de un contacto, contar cuántos mensajes tienen más de m caracteres en el contenido, generar una secuencia con todos los mensajes que corresponden a un mismo asunto y ordenar la colección de mensajes.

El siguiente diagrama modela a la colección de mensajes de correo almacenados en un arreglo:



La estructura es homogénea, todas las componentes son objetos de clase `Mensaje`. Las clases `Correo` y `Mensaje` están asociadas. La clase `Correo` encapsula a un arreglo de elementos

de clase `Mensaje` y un entero que representa la cantidad de mensajes, es decir la cantidad de elementos del arreglo que mantienen referencias ligadas. Los elementos están *comprimidos*, todos los elementos nulos están al final.

Cuando se invoca el constructor de `Correo` se crea un arreglo de `max` elementos, inicialmente nulos.

`insertar (c:Mensaje)` Asigna el objeto `c` a la posición `cant` y aumenta el valor de `cant`, que mantiene en cada momento la cantidad de elementos ocupados en el arreglo y también la posición en la que se va a insertar el próximo mensaje de correo. Requiere que la clase cliente controle que la estructura no esté llena y que el mensaje no pertenezca al correo.

`eliminar (c:Mensaje)` Busca el mensaje con la misma identidad que `c` y si existe, arrastra los que siguen una posición, de modo que no queda una referencia nula entre otras ligadas y se conserva el orden en el que se insertaron los mensajes, actualiza `cant`. Si no existe el mensaje `c`, la colección no se modifica.

`eliminar (c:Contacto)` Elimina todos los mensajes del contacto `c` y comprime los que quedan. Actualiza `cant`. Si no existen mensajes del contacto `c`, la colección no se modifica.

`pertenece (c:Mensaje):boolean` retorna verdadero si existe un mensaje con la misma identidad que c.

`caracteres(m:entero):entero` computa la cantidad de mensajes con c caracteres en el contenido.

`asunto(a:String):Correo` genera una estructura con los mensajes que corresponden al asunto a

`dosenSec(c:Contacto):Mensaje` Dada una secuencia de mensajes $S=s_1,s_2,\ldots,s_{cant-1},s_{cant}$ la consulta `dosenSec` retorna $\mathbf{s_k}$ tal que s_k y s_{k-1} corresponden al mismo contacto c y no existe un j, j > k, tal que s_j y s_{j-1} corresponden al mismo contacto c, con 0<=k<`cant` y 0<=j<`cant`.

Observemos que los elementos pueden ser procesados, por ejemplo para calcular cuántos mensajes tienen más de `c` caracteres en el contenido, aun cuando la estructura no esté completa. Luego se insertan nuevos mensajes, se eliminan otros y se vuelve a procesar la estructura completa. La situación es diferente en otros problemas, como por ejemplo la estación meteorológica, en la cual se conoce la cantidad de elementos y la estructura está completa en el momento que se procesa.

La implementación parcial en Java de `Correo` es:

```
class  Correo {
//Atributos de Instancia
   private Mensaje[] T;
   private int cant;
/*Constructor
Crea una estructura con capacidad para max mensajes*/
public  Correo(int max) {
     T= new Mensaje [max];
     cant = 0;}
//Comandos
public void insertar  (Mensaje m) {
/*Asigna el mensaje m a la primera posición libre del arreglo, es
decir, cant. Aumenta el valor de cant. Requiere que la clase cliente
```

```
haya verificado que la colección no esté llena, m esté ligad y no
pertenezca a la colección*/
  T[cant++] = m;   }
public void eliminar  ( Mensaje m){
/* Busca un mensaje con la misma identidad que m en la secuencia de
mensajes, si lo encuentra arrastra los mensajes que siguen una
posición*/
  boolean esta = false; int i= 0;
  while (!esta && i <  cantMen())//Busca a m
     if (T [i] == m)
       esta = true;
     else
       i++;
  if (esta) {
    cant--;
    arrastrar(i);}}
private void arrastrar( int i){
/* arrastra todos los elementos una posición hacia arriba*/
   while (i < cantMen()){
     T[i] = T[i+1];
     i++;}
  T[i]=null;}
//Consultas
public int  cantMen () {
     return cant;}
public boolean estaLleno() {
     return cant == T.length;}
public boolean hayMen() {
     return cant > 0;}
public Mensaje mensaje(int p){
/*Retorna el mensaje que corresponde a la posición p, si p no es una
posición válida retorna nulo*/
   Mensaje m = null;
   if (p>=0 && p < cantMen()) m = T[p];
   return m;}
public boolean pertenece (Mensaje m){
/*Decide si algún elemento de la colección tiene la misma identidad
que m*/
  boolean esta = false;
  for (int i = 0; !esta && i <  cantMen() ; i++){
     esta = T[i] == m;}
  return esta;}
public int caracteres(int m){
/*Cuenta la cantidad de mensajes con más de m caracteres*/
int car =0;
String content;
for (int i = 0; i <  cantMen(); i++) {
    content = T[i].obtenerContenido();
    if (content.length() > m) car++;}
return car;
}
public Correo asunto(String a){
/*Genera un objeto de clase Correo solo con los objetos que
corresponden al asunto a*/
Correo n = new Correo(cantMen());
```

```
for (int i = 0; i <  cantMen() ; i++)
   if(T[i].obtenerAsunto().equals(a)){
     n.insertar(T[i]);}
return n;     }
}
```

Las clases que usan los servicios provistos por `Correo` no conocen su representación interna. La clase `Correo` usa los servicios provistos por la clase `Mensaje`, sin conocer su implementación.

La clase tester debería verificar cada servicio considerando casos significativos. Por ejemplo, para verificar el servicio `eliminar`, los casos de prueba pueden ser: eliminar el primer mensaje, el segundo, el anteúltimo, el último. También hay que considerar que el elemento a eliminar no pertenezca a la colección. La verificación, como siempre, asume que la clase cliente cumple con sus responsabilidades.

Dada un método en la clase tester que incluye las siguientes declaraciones:

```
Mensaje sms;
Correo col;
sms = new Mensaje(…);
col = new Correo(1000);
```

En la clase cliente, cada mensaje `insertar` debería controlar previamente que el mensaje no fue insertado previamente y que la estructura no está llena:

```
if (!col.pertenece(sms) && !col.estaLleno())
  col.insertar(sms);
```

Observemos que el control de pertenencia requiere recorrer la estructura completa para retornar false, que probablemente sea el caso más frecuente.
Ejercicio: Complete la implementación de la clase Correo e implementa las clases Mensaje y Contacto, considerando que un contacto tiene un nombre y una dirección de correo.

La clase `Correo` modela la colección de mensajes. El diseño puede ser parcialmente reusado para modelar otras colecciones, aunque la implementación va a ser diferente.

Caso de Estudio Inventario

Una dependencia municipal mantiene un inventario permanente de los bienes de uso, modelado de acuerdo al siguiente diagrama:

Inventario
T [] Articulo cant:entero
<< Constructores>> Inventario(max:entero) <<Comandos>> alta (c:Articulo):boolean baja (c:Articulo) depreciarRubro (r:entero,p:real) <<Consultas>> cantArt():entero estaLleno():entero recuperar (c:entero):Articulo pertenece (c:entero):boolean

Articulo
codigo:entero rubro:entero valor:real anio:entero
<<Constructor>> Articulo (c:entero,r:entero, v:entero,a:entero) <<Comandos>> depreciar(p:real) <<Consultas>> obtenerCodigo():entero obtenerRubro():entero obtenerValor():real obtenerAnio():entero equals(Articulo a):boolean

`pertenece (c:Articulo):boolean` `unRubro(r:entero):Inventario`	

Las clases `Inventario` y `Articulo` están asociadas. La clase `Inventario` encapsula a un arreglo de elementos de clase `Articulo` y un entero que representa la cantidad actual de artículos en el inventario.

Cuando se crea un objeto de clase `Inventario` se crea un arreglo de `max` elementos, inicialmente nulos. Cada vez que se utiliza el servicio `alta` aumenta el valor de `cant`, que mantiene en cada momento la cantidad de elementos ocupados en el arreglo y también la posición en la que se va a insertar el próximo elemento. Requiere que la clase cliente haya verificado que no existe un artículo con el mismo código. La clase asume la responsabilidad de controlar que la estructura no esté llena, si todos los elementos están ligados retorna false. Los elementos están "comprimidos", todos los elementos nulos están al final. Cuando se da de baja un elemento, el último se copia en su lugar, de modo que no se conserva el orden en el que se insertaron.

La implementación en Java de `Inventario` es:

```
class  Inventario {
//Atributos de Instancia
   private Articulo[] T;
   private int cant;
/*Constructor
Crea una Coleccion con capacidad para
max elementos*/
public  Inventario(int max) {
     T= new Articulo [max];
     cant = 0;}
//Comandos
public boolean alta  (Articulo elem) {
/*Inserta un elemento al final de la Coleccion, requiere que no exista
un artículo con el mismo código*/
  boolean existe=false;
  if (cant<T.length) {
       T[cant++] = elem;
       existe = true;}
  return existe;}
public void baja  ( Articulo c){
/*Busca un artículo equivalente a c y si existe copia el último en esa
posición*/
  boolean esta = false; int i= 0;
  while (!esta && i <  cantArt())
     if (T [i].equals(c)) esta = true;
     else i++;
  if (esta) {
    cant--;
    T[i] = T[cant];
    T [cant] = null;}}
public void depreciarRubro(int r, float p){
/*modifica el valor de cada artículo del rubro r decrementándolo de
acuerdo al porcentaje p.*/
float v;
for (int i = 0; i <  cantArt() ; i++)
```

```
   if(T[i].obtenerRubro() == r){
     v = T[i].obtenerValor();
     T[i].establecerValor(v*(1-p));}
}
//Consultas
public int  cantArt () {
     return cant;}
public boolean estaLleno() {
     return cant == T.length;}
public Articulo recuperar (int c){
//Retorna el artículo con el código dado
  boolean esta = false; int i;
  for (i = 0; !esta && i <  cantArt() ; i++){
     esta = T[i].obtenerCodigo() == c; }
  if (esta) return T[i];
  else return null;}
public boolean pertenece (int c){
/*Retorna verdadero si un elemento de la Coleccion tiene el código c*/
  boolean esta = false;
  for (int i = 0; !esta && i <  cantArt() ; i++)
     esta = T[i].obtenerCodigo() == c;
  return esta;}
public boolean pertenece (Articulo c){
/*Decide si algún elemento de la colección es equivalente a c*/
  boolean esta = false;
  for (int i = 0; !esta && i <  cantArt() ; i++)
     esta = T[i].equals(c);
  return esta;}
public Inventario unAnio(int a){
/*genera un objeto de clase Inventario solo con los objetos que
corresponden al año a. */
Inventario n = new Inventario(cantArt());
for (int i = 0; i <  cantArt() ; i++)
   if(T[i].obtenerAnio() == a){
     n.alta(T[i]);}}
return n;
}
```

Observemos que algunos servicios de `Inventario` tienen la misma funcionalidad y responsabilidades que los de `Correo`, aunque cambian los nombres. Ambas clases modelan a una colección de elementos homogéneos y brindan un comando para agregar un elemento o eliminar un elemento, aunque el comando para agregar tiene diferente responsabilidad en cada caso. Los servicios que realizan alguna forma de procesamiento son diferentes, porque dependen de la aplicación. Por supuesto podemos definir otras clases que compartan la funcionalidad de algunos servicios.

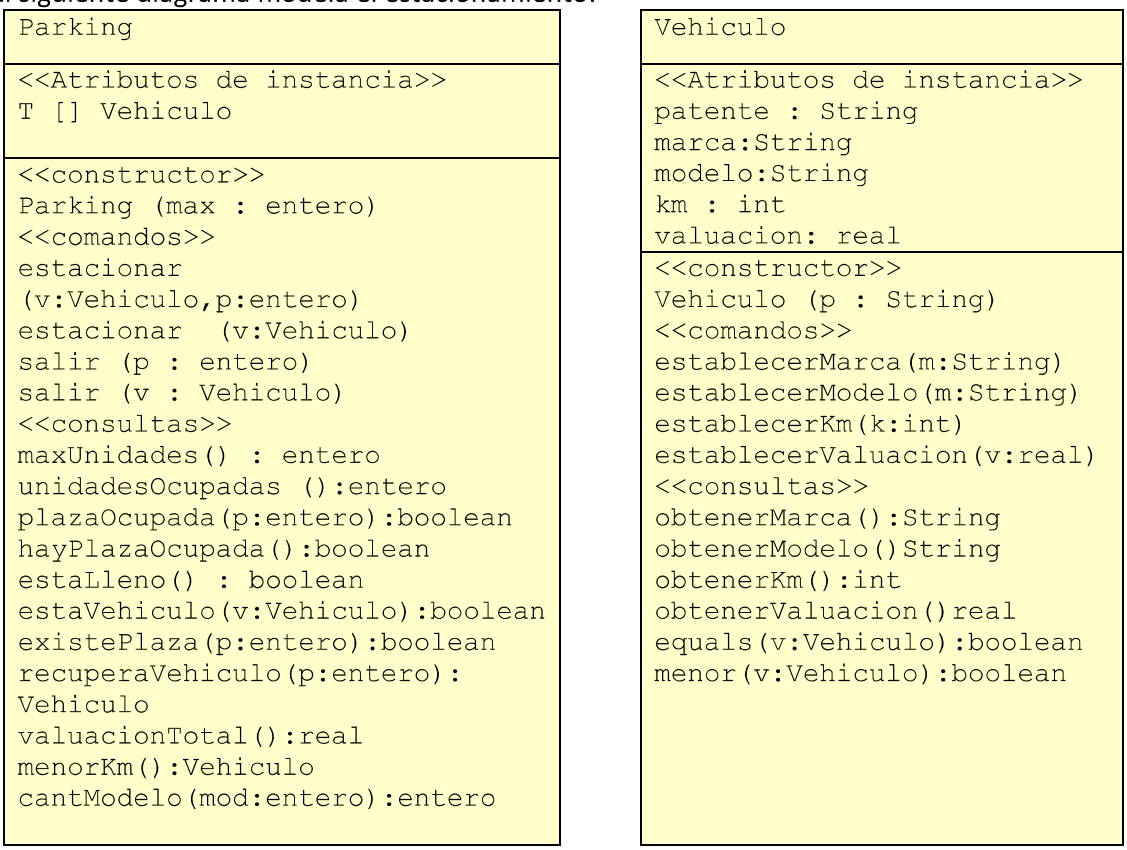
El siguiente caso de estudio define una clase que modela una tabla de elementos homogéneos. Mientras que las dos colecciones propuestas se caracterizan porque las componentes están comprimidas, utilizamos el nombre tabla para una estructura en la cual las componentes nulas y ligada pueden estar intercaladas.

Caso de Estudio Parking

Una concesionaria dispone de un estacionamiento tiene un conjunto de unidades. En un momento dado una plaza puede estar ocupada por un vehículo o libre. Las unidades libres y ocupadas están intercaladas.

El estacionamiento puede ser modelado por un arreglo en la cual el subíndice indica el número de plaza. Cuando llega un vehículo se inserta un vehículo en la tabla. La eliminación puede hacerse a partir del vehículo o del número de plaza.

El siguiente diagrama modela el estacionamiento:



`estacionar(unVeh:Vehiculo, p:entero)` asigna el vehículo `unVeh` a la plaza `p`, que requiere libre. Requiere además `0<=p<maxUnidades()`.

`salir(unVeh:Vehiculo)` busca un vehículo equivalente a `unVeh` y si está libera la plaza en la que está dicho vehículo

`salir(p: entero)` libera la plaza, requiere `0<=p<maxUnidades()`

`maxUnidades()`: retorna la cantidad de unidades del estacionamiento

`unidadesOcupadas()`: retorna la cantidad de unidades ocupadas

`plazaOcupada(int p)`: retorna true si la plaza p está ocupada, requiere `0<=p<maxUnidades()`

`hayPlazaOcupada()`: retorna true si hay al menos un vehículo en el estacionamiento

`estaLleno()`: retorna true si el estacionamiento no tiene unidades libres

`estaVehiculo(Vehiculo unVeh):` retorna true si una plaza del estacionamiento mantiene un vehículo equivalente a unVeh

`existePlaza (int p):` retorna true si `0<=p<maxUnidades()`

`recuperarVehiculo(int p):` retorna el vehículo de la plaza p, requiere `0<=p<maxUnidades()`

`valuacionTotal():` retorna la valuación total de todos los vehículos

`menorKm():Vehiculo :` retorna el vehículo con menor kilometraje

`cantModelo(mod:entero):` retorna la cantidad de vehículos del modelo mod.

Observemos que si la clase cliente envía el mensaje estacionar a un objeto de clase Parking y no se cumple `0<=p<maxUnidades()`, se produce un error de ejecución. Si en cambio `p` es válido pero no corresponde a una plaza libre, se produce un error de aplicación.

Las clases `Parking` y `Vehiculo` están asociadas. La clase `Parking` encapsula un arreglo de elementos de clase `Vehiculo`. La implementación en Java de `Parking` es:

```
class  Parking{
private Vehiculo[] T;
//Constructor
public Parking(int max){
   T = new Vehiculo[max];}
//Comandos
public void estacionar(Vehiculo unVeh, int p) {
/*Asigna el vehiculo a la plaza, requiere que la plaza exista*/
    T[p] = unVeh;}
public void salir(Vehiculo unVeh){
/*Busca en el estacionamiento un vehículo equivalente a unVeh y libera
la plaza*/
int i = 0; boolean esta=false;
while (i < T.length && !esta){
   if (T[i] != null)
     esta = unVeh.equals(T[i]) ;
   else
     i++;}
 if (esta)
   T[i] = null;}
public void salir(int p){
/* Elimina el elemento de la plaza p, requiere que p sea un plaza
válida*/
    T[p] = null;}
//Consultas
public int maxUnidades(){
  return T.length;}
public int unidadesOcupadas(){
/*Retorna la cantidad de unidades en las que hay un vehículo
asignado*/
  int cant=0;
  for (int i=0; i<=T.length;i++)
    if (T[i] != null)
      cant++;
  return cant;}
public boolean plazaOcupada(int p){
//Requiere p válida
```

```
   return T[p] != null;}
public boolean hayPlazaOcupada(){
int i = 0; boolean hay=false;
while (i < T.length && !hay){
   if (T[i] != null)
     hay = true ;
   i++;}
return hay;}
public boolean estaLleno(){
int i = 0; boolean hay=false;
while (i < T.length && !hay){
   if (T[i] == null)
     hay = true ;
   i++;}
return !hay;}
public boolean estaVehiculo(Vehiculo unVeh){
/*Decide si una plaza mantiene un vehículo equivalente a unVeh que
asume ligado*/
int i = 0; boolean esta=false;
while (i < T.length && !esta){
   if (T[i] != null)
     esta = unVeh.equals(T[i]) ;
   i++;}
return esta;}
public boolean existePlaza (int p) {
    return p >= 0 && p < T.length;}
public Vehiculo recuperarVehiculo(int p) {
// Requiere que p sea una posición válida
    return T[p];}
public Vehiculo menorKm() {
/*Asume que no hay vehículos con más de 500000 km
Puede retornar nulo si la tabla está vacía*/
  Vehiculo v=null; int menor = 500000;
  for (int i=0; i<=T.length;i++)
    if (T[i] != null &&  T[i].obtenerKm() < menor){
        v = T[i]; menor = v.obtenerKm();}
  return v;}
public int cantModelo(int mod) {
  int cant=0;
  for (int i=0; i<=T.length;i++)
    if (T[i] != null &&  T[i].obtenerModelo() == mod)
         cant++;
  return cant;}
public float valuacionTotal() {
  float v=0;
  for (int i=0; i<=T.length;i++)
    if (T[i] != null)
         v = v+T[i].obtenerValuacion();
  return v;}}
```

Observemos que los operandos no son conmutativos en la expresión:

```
(T[i] != null &&  T[i].obtenerModelo() = mod)
```

Y es necesario utilizar el operador con cortocircuito. Es decir, las siguientes expresiones computan error en ejecución si `T[i] == null`:

```
(T[i] != null &  T[i].obtenerModelo() = mod)
```

```
(T[i].obtenerModelo() = mod && T[i] != null)
```

Notemos también que no se ha asignado a la clase `Parking` ni a sus clientes la responsabilidad de controlar que la unidad está disponible antes de que se ejecute el comando `estacionar`. Si la componente está ligada, la nueva referencia se sobrescribe sobre la anterior.

La clase `Parking` utiliza un arreglo parcialmente ocupado para representar una tabla de elementos cada uno de los cuales ocupa una posición particular. Podemos definir una clase similar para modelar el conjunto de casilleros asignados a los socios de un club, la asignación de mozos a las mesas de un bar o cualquier otro problema en el cual cada elemento se asigna a una posición específica.

En algunos de estos problemas cada elemento puede estar asignado a una única posición. Por ejemplo, un vehículo solo estará asignado a una plaza del estacionamiento. En otros casos un mismo elemento puede estar ligado a varias posiciones de la tabla. Por ejemplo, un mismo mozo puede estar asignado a n mesas.

El caso de estudio que sigue se modela mediante una colección, pero los elementos están ordenados de acuerdo a un atributo.

Caso de Estudio Libreta de Contactos

Una libreta de contactos mantiene el nombre, número de teléfono móvil, número de teléfono fijo y email de un conjunto de personas u organizaciones.

La clase `Libreta_Contactos` encapsula una colección de elementos de clase Contacto, representada con un arreglo parcialmente ocupado. Los elementos se mantienen ordenados alfabéticamente por nombre y están comprimidos de modo que todas las posiciones libres están al final.

Libreta_Contactos
T [] Contacto cant:entero
<< Constructores>> LibretaContactos(max:entero) <<Comandos>> insertar(con:Contacto) eliminar(con:Contacto) <<Consultas>> cantContactos():entero estaLleno():entero pertenece(c:Contacto):boolean

Contacto
nombre:String nroMovil: String nroFijo:String email:String
<<Constructor>> Contacto (n:String) <<Comandos>> <<Consultas>> igualNombre(c: Contacto):boolean mayorNombre(c: Contacto):boolean

El comando insertar requiere que la clase cliente controle que la estructura no esté llena y no exista un contacto con el mismo nombre que el parámetro `con`. Si la libreta de contactos se mantiene ordenada por nombre, el servicio `insertar` no puede implementarse asignando el nuevo contacto a la primera posición libre. Para comprender cómo implementar el servicio insertar comencemos visualizando la libreta de contactos a través de una grilla con capacidad para 6 contactos:

Nombre	Número de Móvil	Número Fijo	Email

Consideremos que el primer contacto que se inserta corresponde a Davini Laura. Notemos que no interesan los otros atributos porque la libreta se ordena por nombre. Como la grilla está vacía el primer contacto ocupa la primera posición en la grilla:

Nombre	Número de Móvil	Número Fijo	email
Davini Laura	…	…	…

Si luego vamos a insertar Castro Ramón, para que la Libreta quede ordenada, debemos *arrastrar* a Davini Laura desde la posición 0 a la posición 1 y luego asignar el nuevo contacto a la posición 0:

Nombre	Número de Móvil	Número Fijo	email
Castro Ramón	…	…	…
Davini Laura	…	…	…

Si luego insertamos a Álvarez María, tenemos que arrastrar a los dos contactos anteriores una posición y luego asignar el nuevo contacto a 0:

Nombre	Número de Móvil	Número Fijo	email
Alvarez María	…	…	…
Castro Ramón	…	…	…
Davini Laura	…	…	…

Si luego insertamos a Funez María, como no hay ningún contacto mayor al nuevo, insertamos directamente en la última posición:

Nombre	Número de Móvil	Número Fijo	email
Alvarez María	…	…	…
Castro Ramón	…	…	…
Davini Laura	…	…	…
Funez María	…	…	…

Si luego insertamos un nuevo contacto con nombre Cepeda Pedro, arrastramos una posición a los dos contactos con nombres mayores al nuevo:

Nombre	Número de Móvil	Número Fijo	email
Alvarez María	…	…	…
Castro Ramón	…	…	…
Cepeda Pedro	…	…	…
Davini Laura	…	…	…
Funez María	…	…	…

La grilla nos permite visualizar la libreta de contactos de manera abstracta para mostrar cómo se inserta cada nuevo contacto. El diagrama de objetos, que también es una abstracción, es útil para graficar cómo se administra la memoria, pero no es una buena herramienta para diseñar el algoritmo insertar. El algoritmo para el servicio insertar puede ser entonces:

```
Algoritmo insertar
DE nuevo
   Buscar la posición del primer elemento mayor al nuevo
   Si existe
      Arrastrar todos los elementos desde esa posición hasta la última
   Asignar nuevo contacto a la posición encontrada
   Incrementar la cantidad de contactos
```

Si el nuevo elemento es mayor a todos, en particular si el nuevo elemento es el primero que se inserta, la posición encontrada es la primera libre.

El código del servicio `eliminar` también tiene que ser diseñado convenientemente, porque los elementos tienen que quedar ordenados y comprimidos. Si vamos a eliminar el contacto Cepeda Pedro, arrastramos una posición a los dos contactos con nombres mayores. En este caso la cantidad de contactos se decrementa en 1.

Nombre	Número de Móvil	Número Fijo	Email
Alvarez María	…	…	…
Castro Ramón	…	…	…
Davini Laura	…	…	…
Funez María	…	…	…

El algoritmo para el servicio eliminar puede ser entonces:

```
Algoritmo eliminar
DE contacto
   Buscar la posición del contacto
   Si existe
      Arrastrar todos los elementos desde la última posición hasta la
      encontrada
     Decrementar la cantidad de contactos
```

Si el contacto no existe el comando eliminar no modifica a la estructura.

```
class Libreta_Contactos{
/*Mantiene una colección de contactos ordenada por nombre. */
//Atributos de instancia
private Contacto [] T;
private int cant;
//Constructor
public Libreta_Contactos(int max){
    T = new Contacto [max];}
//Comandos
public void insertar (Contacto nuevo){
//Requiere que la colección no esté llena
    int pos = posInsercion(nuevo,cant);
    arrastrarDsp (pos,cant-pos);
    T[pos] = nuevo;
    cant++; }
private void arrastrarDsp (int pos,int n){
    if (n > 0){
       T[pos+n] = T[pos+n-1];
       arrastrarDsp(pos,--n);       } }
public void eliminar(Contacto con){
/*Busca la posición de un contacto con el mismo nombre que con, si
existe lo elimina arrastrando los que le siguen una posición  */
    int pos = posElemento (con,cant);
    if (pos < cant){
      arrastrarAnt(pos,cant-pos-1);
      cant--;} }
private void arrastrarAnt(int pos,int n){
 if (n > 0){
       T[pos] = T[pos+1];
       arrastrarAnt(++pos,--n);        }}
//Consultas
public int cantContactos (){
  return cant;}
```

```
public boolean estaLleno (){
  return cant == T.length;}
public boolean pertenece(Contacto con){
return posElemento(con,cant) < cant;}
private int  posInsercion (Contacto con,int n){
/* Retornar la posición del primer elemento mayor a con, o 0
Caso trivial: Si la colección está vacía la posición de inserción es 0
Caso trivial: Si el nombre del último contacto de la colección es
menor al buscado, la posición de inserción es n
Caso Recursivo: Si el nombre del último contacto de la colección es
mayor al buscado, busca la posición entre los primeros n-1 contactos
de la colección*/
    int pos = 0;
    if (n > 0)
      if (con.mayorNombre(T[n-1].obtenerNombre()) ||
          con.igualNombre(T[n-1].obtenerNombre()))
        pos = n;
      else
        if (T[n-1].mayorNombre(con.obtenerNombre()))
          pos  = posInsercion (con,--n);
    return pos;}
private int  posElemento(Contacto con,int n){
/* retorna la posición del elemento, o cant
Caso trivial: Si la colección está vacía el elemento no está y retorna
cant
Caso trivial: Si el último elemento de la colección es menor al
buscado, el elemento no está y retorna cant
Caso trivial: Si el último elemento de la colección es el buscado, la
posición es n- 1
Caso Recursivo: Si el último elemento de la colección es mayor al
buscado, busca la posición en los primeros n-1 elementos de la
colección*/
    int pos=cant;
    if (n > 0)
      if (T[n-1].igualNombre(con.obtenerNombre()))
         pos = n-1;
      else
        if (T[n-1].mayorNombre(con.obtenerNombre()))
          pos  = posElemento (con,--n);
    return pos;}
public void test(){
  for (int i=0;i<cant;i++)
    System.out.println(T[i].obtenerNombre());}
}
```

El arreglo mantiene una colección homogénea de elementos. La colección se crea con capacidad para mantener `max` elementos pero inicialmente está vacía. En la colección se puede insertar un elemento en forma ordenada, eliminar un elemento manteniendo el orden y decidir si un elemento pertenece a la colección.

Mantener la estructura ordenada tiene un *costo* en ejecución, cada inserción requiere buscar la posición y arrastrar, en promedio, la mitad de los elementos. Como contrapartida las búsquedas son más eficientes, ya que si se busca un elemento y este no pertenece a la colección, no es necesario recorrerla todas. Si las búsquedas son más frecuentes que las inserciones, mantener la estructura ordenada puede ser una decisión acertada.

La eficiencia del algoritmo insertar puede mejorar si al mismo tiempo se busca la posición de inserción y se arrastran los elementos. Esta estrategia es posible dado que la clase cliente controla que hay al menos una posición libre y no existe un elemento con el mismo nombre que el nombre del contacto que se agrega.

```
Algoritmo insertar
DE nuevo
   Buscar desde la última posición cant-1 hasta 0, la posición pos del
   primer elemento menor al nuevo y arrastrar cada elemento de la
   posición i a la posición i+1
   Si existe Asignar nuevo contacto en pos+1
   Sino Asignar nuevo contacto en cant
   Incrementar cant
```

Si la responsabilidad de controlar que no haya dos contactos con el mismo nombre recae en la clase cliente, es necesario recorrer dos veces la estructura, aunque sea parcialmente, para insertar un nuevo elemento. La primera cuando la clase cliente envía el mensaje `pertenece`, la segunda cuando el comando `insertar` busca la posición de inserción. Una alternativa más eficiente es que la clase `Libreta_Contactos` asuma la responsabilidad de controlar que no haya repetidos y el comando `insertar` busque el contacto, si existe no lo inserta, si no existe, encontró un contacto con nombre mayor al parámetro y esa es la posición de inserción.

Ejercicios

- *Escriba una clase tester que verifique los servicios insertar y eliminar para una colección de 10 elementos y considerando casos de prueba significativos.*
- *Modifique la implementación de los métodos posInsertar y posElemento proponiendo soluciones iterativas*
- *Agregue un método intercalar (nueva:Libreta_Contactos) a la clase Libreta_Contactos que modifique el estado interno de la Libreta de contactos que recibe el mensaje intercalando ordenadamente los contactos de nueva. Puede utilizar un objeto auxiliar.*
- *Agregue un atributo de clase String llamado ciudad a la clase Contacto y analice cómo afecta el cambio a la clase Libreta_Contactos*
- *Agregue un método contactosCiudad (c:String):entero a la clase Libreta_Contactos que compute la cantidad de contactos de la ciudad c.*
- *Implemente el método insertar considerando que la responsabilidad de controlar la pertenencia es de la clase Libreta_Contactos.*

Cuando una colección de elementos está ordenada de acuerdo a un atributo y el necesario decidir si un elemento en particular pertenece a la colección, es posible aplicar una estrategia conocida como **búsqueda binaria**. La estrategia consiste en partir la estructura en mitades, considerando que el **elemento buscado** puede ser:

- Igual al que está en el medio
- Menor que el que está en el medio
- Mayor que el que está en el medio

De modo que podemos definir un algoritmo:

```
Algoritmo Búsqueda Binaria
```

```
DE elem
  si el elemento que está en el medio es el buscado
    EXISTE
  si hay un solo elemento y no es el buscado
     NO EXISTE
  sino
    si el elemento que está en el medio es menor al buscado
      Descartar la primera  mitad
      Buscar en la segunda mitad
    sino
      Descartar la segunda mitad
      Buscar en la primera mitad
```

Para nuestro caso de estudio específico podemos refinar el algoritmo:

```
Algoritmo Búsqueda Binaria
DE contacto
  si el contacto que está en el medio de la colección
     es el buscado
    EXISTE
  si hay un solo contacto y no es el buscado
     NO EXISTE
  sino
    si el contacto que está en el medio es menor al buscado
      Descartar la primera  mitad de la Libreta de contactos
      Buscar en la segunda mitad de la Libreta de contactos
    sino
      Descartar la segunda mitad de la Libreta de contactos
      Buscar en la primera mitad de la Libreta de contactos
```

El algoritmo puede refinarse todavía más considerando que los datos están en un arreglo:

```
Algoritmo BúsquedaBinaria
DE ini,fin,contacto
Mitad←(ini+fin)/2
si T_mitad = contacto
  EXISTE
sino
  si ini >= fin
     NO EXISTE
  sino
    si T_mitad < contacto
       BuscarBinaria mitad+1,fin,contacto
   sino
      BusquedaBinaria ini,mitad-1,contacto
```

Aunque esta versión sigue siendo abstracta, la traducción a un lenguaje de programación es más directa que en el caso anterior.

Ejercicio

- *Aplicar la estrategia descripta por el algoritmo a una grilla con 10 contactos considerando los siguientes casos: se busca el primer elemento, un elemento menor al primer elemento, el último elemento, un elemento mayor al último, un elemento que está en la estructura*

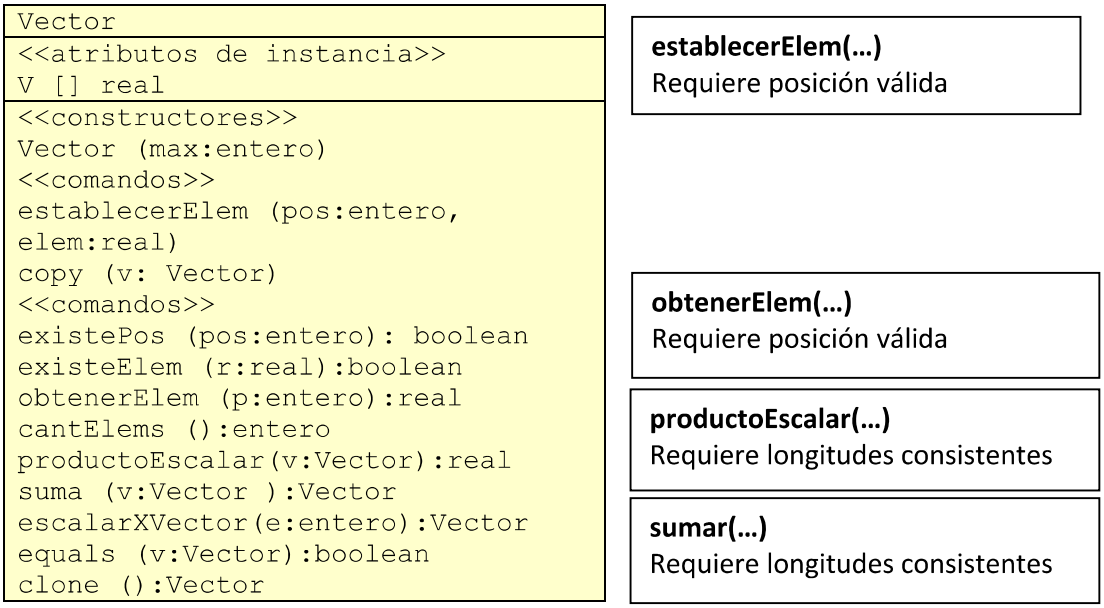
mayor al primero y menor al último, un elemento que NO está en la estructura y es mayor al primero y menor al último.

- *Implementar el método pertenece aplicando la estrategia de búsqueda binaria.*

Observemos que la recursividad permite modelar naturalmente esta estrategia de búsqueda, porque está especificada en forma recursiva.

Problemas Propuestos

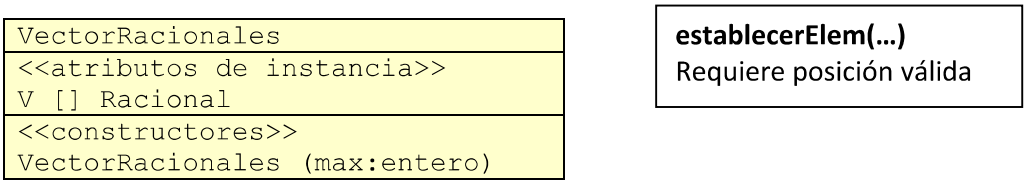
1. *Implemente el TDA Racional modelado en el siguiente diagrama:*
2. *Dado el siguiente diagrama que modela un TDA Vector*



a. *Implemente la clase Vector considerando que el cliente asume las posiciones dentro del vector de 1 a la cantidad de elementos del vector.*
b. *Implemente un Tester para la case vector, definiendo adecuadamente los casos de prueba.*
c. *Dibuje el diagrama de objetos para el siguiente segmento de instrucciones:*

```
Vector v ;
v= new Vector (7);
```

3. *Dado el siguiente diagrama que modela un TDA VectorRacionales*



```
<<comandos>>
establecerElem (pos:entero,
elem:Racional)
copy (v: Vector)
<<comandos>>
existePos (p:entero): boolean
existeElem (r:Racional):boolean
obtenerElem
(pos:entero):Racional
cantElems ():entero
productoEscalar(v:Vector):Racion
al
suma (v:Vector ):Vector
escalarXVector(e:entero):Vector
equals (v:Vector):boolean
clone ():Vector
```

obtenerElem(…)
Requiere posición válida

a. *Implemente la clase VectorRacionales en el diagrama considerando que **el cliente asume las posiciones dentro del vector de 1 a la cantidad de elementos del vector**. Implemente equals, copy y clone en profundidad.*

b. *Implemente un Tester para la case VectorRacionales, definiendo adecuadamente los casos de prueba.*

c. *Dibuje el diagrama de objetos para el siguiente segmento de instrucciones:*

```
Vector v ;
v= new Vector (7);
```