

un grupo de pacientes que son atendidos en distintas clínicas de la ciudad. De cada paciente la aplicación debe almacenar un código numérico que es único (76527, por ejemplo), un nombre ("Mariana Chacón"), el nombre de la clínica a la cual fue remitido el paciente ("Clínica Reina Sofía"), la información médica del paciente ("Dolor de oído") y el sexo (Femenino). La interfaz de usuario del programa es la que se presenta en la figura 3.4.

Se espera que el programa sea capaz de agregar un paciente a la central, contemplando cuatro variantes posibles: (1) insertar al comienzo de la lista de pacientes, (2) insertar al final, (3) insertar después de otro paciente (dado su código) o (4) insertar antes de otro paciente (dado su código). El programa también debe permitir (5) eliminar un paciente, (6) consultar sus datos o (7) editar su información médica, para lo cual el usuario debe suministrar el respectivo código del paciente.

### 3. Caso de Estudio N° 1: Una Central de Pacientes

Un cliente nos pide que desarrollemos un programa de computador para administrar la información de

Por facilidad de implementación, no se pide que la información sea persistente.

Fig. 3.4 – Interfaz de usuario del caso de estudio



- En la parte izquierda de la ventana aparece la lista de pacientes registrados en la central. Al hacer clic sobre uno de ellos, se muestra la información que tiene registrada y se le permite al usuario editar la información médica.
- Con el primer botón (zona derecha de la ventana) se puede agregar un paciente a la central. Al oprimirlo, aparece un diálogo en el que se le pregunta al usuario la posición en la lista en la cual se quiere agregar el nuevo paciente, y luego se le pide toda la información de éste.
- Con el segundo botón se puede localizar a un paciente de la central dado su código. En la ventana que se le muestra al usuario (la cual se presenta más abajo) se puede editar su información médica.
- Con el tercer y último botón se puede eliminar un paciente dando su código.




- En esta ventana de diálogo aparece la información de un paciente. Los campos que tienen el nombre, el código, la clínica y el sexo del paciente no se pueden modificar.
- El único campo que se puede cambiar es el que contiene la información médica del paciente. Para que los cambios se hagan efectivos, se debe utilizar el botón **Registrar Cambios**.
- Para la selección del sexo del paciente se utiliza un componente gráfico que impide que se seleccionen las dos opciones al mismo tiempo.

3.1. **Objetivos de Aprendizaje**

¿Qué tenemos que hacer en el caso de estudio?	¿Qué tenemos que aprender o reforzar?
<ul style="list-style-type: none"><li>Insertar, eliminar y buscar un paciente en una estructura lineal enlazada.</li></ul>	<ul style="list-style-type: none"><li>Estudiar la algorítmica de manejo de las estructuras lineales enlazadas e identificar los patrones de algoritmo que permiten resolver problemas sobre dichas estructuras.</li></ul>
<ul style="list-style-type: none"><li>Presentar al usuario la información completa de un paciente en una nueva ventana.</li></ul>	<ul style="list-style-type: none"><li>Estudiar el componente <code>JDialog</code> del lenguaje de programación Java.</li></ul>
<ul style="list-style-type: none"><li>Permitir al usuario seleccionar una opción de un conjunto de opciones excluyentes.</li></ul>	<ul style="list-style-type: none"><li>Estudiar los componentes de interacción <code>JRadioButton</code> y <code>ButtonGroup</code> del lenguaje de programación Java.</li></ul>

3.2. **Comprensión de los Requerimientos**

**Tarea 1**



**Objetivo:** Entender el problema del caso de estudio.  
(1) Lea detenidamente el enunciado del caso de estudio e (2) identifique y complete la documentación de los siete requerimientos funcionales que allí aparecen.

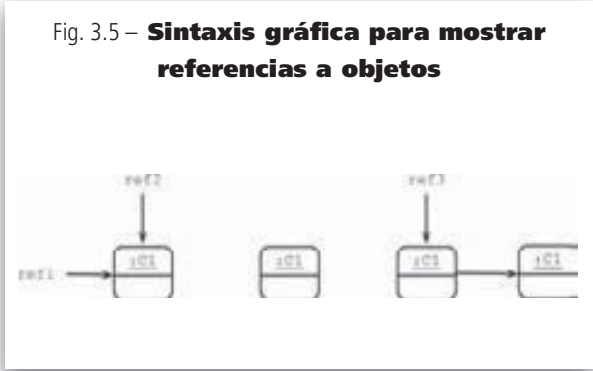
Requerimiento funcional 1	Nombre	R1 – Insertar un paciente al comienzo de la lista
	Resumen	
	Entrada	
	Resultado	

Requerimiento funcional 7	Nombre	R7 – Modificar la información médica de un paciente
	Resumen	
	Entrada	
	Resultado	

3.3. Referencias y Ciclo de Vida de los Objetos

Antes de abordar el tema de las estructuras enlazadas, es importante repasar la diferencia que existe entre un objeto y una referencia a éste. De igual forma, debemos definir claramente lo que quiere decir que un objeto deje de existir, y determinar el momento en el cual el espacio que el objeto ocupa en memoria es recuperado por el computador. Éstos son los dos temas que trataremos en esta sección.

Un objeto es una instancia de una clase y ocupa un espacio en la memoria del computador, en donde almacena físicamente los valores de sus atributos y asociaciones. La clase es la que define el espacio en memoria que necesita cada uno de sus objetos, puesto que conoce la cantidad de información que éstos deben almacenar. Una referencia es un nombre mediante el cual podemos señalar o indicar un objeto particular en la memoria del computador. Una variable de una clase `c1` declarada dentro de un método cualquiera no es otra cosa que una referencia temporal a un objeto de dicha clase. En la figura 3.5 presentamos una sintaxis gráfica que nos va a permitir mostrar referencias a objetos. Allí aparecen cuatro objetos de la clase `c1`. El primero de los objetos es señalado o apuntado por dos referencias llamadas `ref1` y `ref2`. El segundo de los objetos no tiene referencias. El tercer objeto está siendo referenciado por `ref3` y tiene una asociación a un objeto de la misma clase que se materializa como una referencia, la cual, en lugar de estar almacenada en una variable, está almacenada en un atributo.



Cuando usamos la sintaxis `ref1.m1()` estamos pidiendo la invocación del método `m1()` sobre el objeto que se encuentra referenciado por `ref1`. Es importante recordar que los objetos sólo se pueden acceder a través de referencias, lo que implica que si existe un objeto sin referencias hacia él, éste deja de ser utilizable dentro de un programa.

El **recolector de basura** (*garbage collector*) de Java es un proceso de la máquina virtual del lenguaje que recorre periódicamente la memoria del computador buscando y eliminando los objetos que no son referenciados desde ningún punto. Por esta razón, si queremos “destruir” un objeto porque ya no es útil dentro de un programa, sólo debemos asegurarnos de que no existe ninguna referencia hacia él. El resto del trabajo lo hace de manera automática el recolector de basura.

En el ejemplo 1 mostramos la manera como ciertas instrucciones de un programa afectan el estado de la memoria del computador.

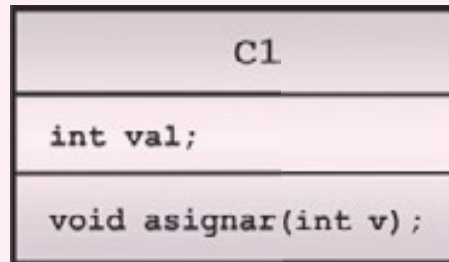
**Ejemplo 1**

**Objetivo:** Mostrar la manera como ciertas instrucciones del lenguaje Java manejan referencias y modifican el estado de los objetos.

En este ejemplo utilizamos la clase C1, cuya declaración se encuentra a continuación. Luego, mostramos de manera incremental la forma como las instrucciones de un programa afectan el estado de la memoria del computador. Por último, presentamos algunas características que se deben tener en cuenta en el momento de comparar objetos y referencias.

```
public class C1
{
    private int val;

    public void asignar( int v )
    {
        val = v;
    }
}
```



Instrucción	Estado de la memoria
C1 v1 = new C1( );	
C1 v2 = v1;	
C1 v3 = null;	
v2.asignar( 5 ); v1 = v3;	
ArrayList a1 = new ArrayList( );	
a1.add( v2 );	
v1 = new C1( ); v1.asignar( 1 );	

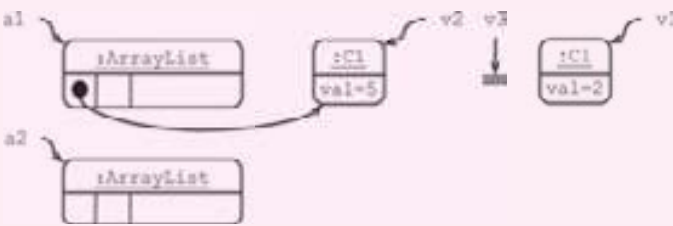
```
v1 = new C1( );
v1.asignar( 2 );
```



```
// Ejecución del recolector de
// basura de Java
```



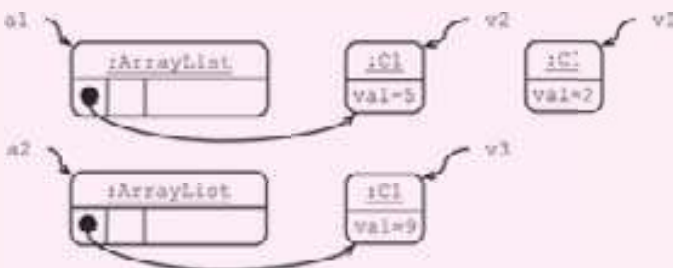
```
ArrayList a2 = new ArrayList( );
```



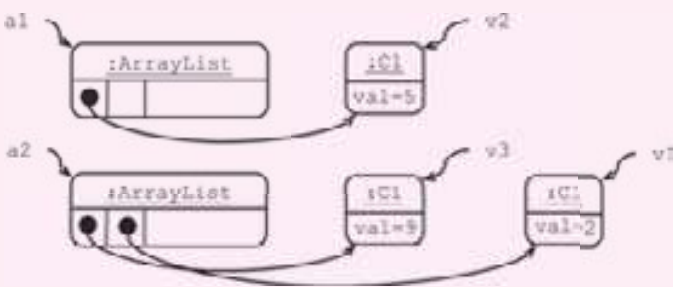
```
v3 = new C1( );
v3.asignar( 9 );
```



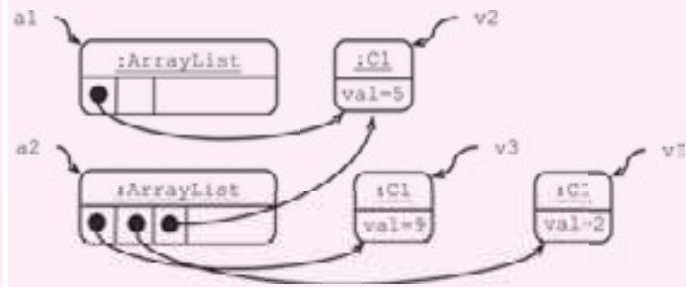
```
a2.add( v3 );
```



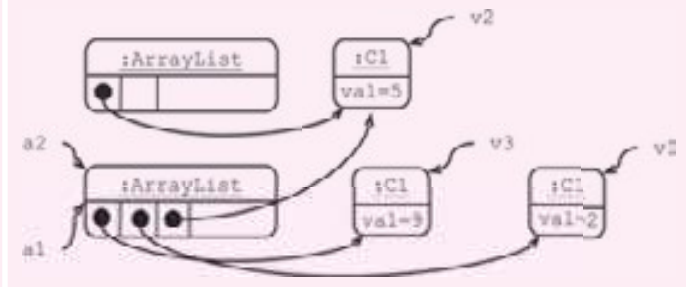
```
a2.add( v1 );
```



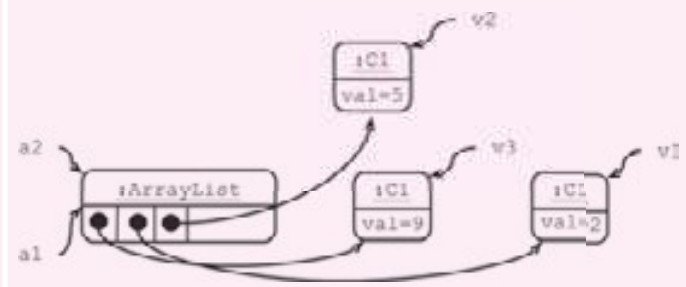
```
a2.add( v2 );
```



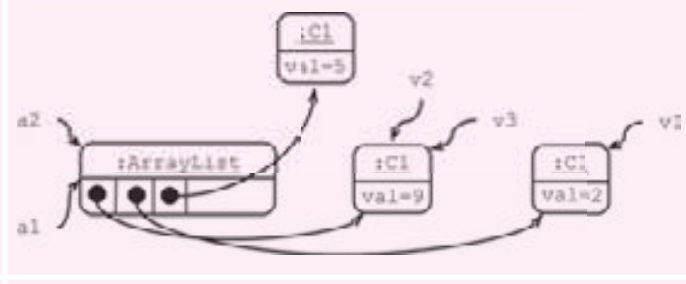
```
a1 = a2;
```



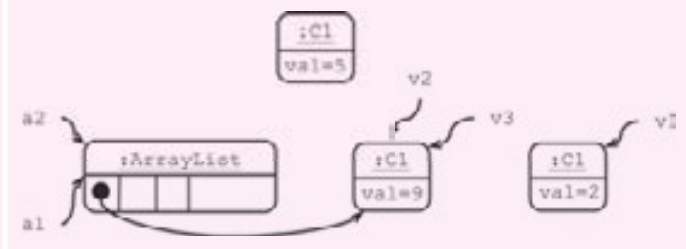
```
// Ejecución del recolector de  
// basura de Java
```



```
v2 = ( C1 )a2.get( 0 );
```



```
a2.remove( 2 );  
a2.remove( 1 );
```



<pre>// Ejecución del recolector de // basura de Java</pre>	
<pre>v1.asignar( 9 );</pre>	
Expresión	Explicación
Aprovechemos ahora el diagrama de objetos al que llegamos con la secuencia anterior de instrucciones para repasar la manera de comparar objetos. En la columna de la izquierda aparece una expresión y en la parte derecha su evaluación.	
<pre>v2 == v3</pre>	Verdadero. Cuando utilizamos el operador == entre referencias, estamos preguntando si físicamente están señalando el mismo objeto.
<pre>v1 == v2</pre>	Falso. Incluso si los objetos referenciados desde v1 y v2 tienen el mismo valor en su único atributo (9), el operador == sólo verifica si las dos referencias llegan al mismo objeto en memoria. Si queremos que la noción de igualdad sea más general, debemos implementar en la clase C1 el método equals(), tal como se muestra más adelante.
<pre>a1.get( 0 ) == v3</pre>	Verdadero. El método get() retorna una referencia al mismo objeto referenciado por v3.
<pre>a1 == a2</pre>	Verdadero. Ambas referencias llegan al mismo objeto de la clase ArrayList.

En algunos casos, nos interesa crear una copia de un objeto con los mismos valores del objeto original. Este proceso se denomina **clonación** y se logra implementando el método `clone()` en la respectiva clase. En

el ejemplo 2 mostramos la manera de extender la clase `C1` del ejemplo anterior, con un método de comparación y un método de clonación.

**Ejemplo 2**

**Objetivo:** Mostrar la manera de implementar un método de comparación y un método de clonación para la clase del ejemplo anterior.

En este ejemplo extendemos la clase C1 con un método que permite comparar dos instancias de la clase, teniendo en cuenta su estado interno. También mostramos la implementación de un método que permite clonar objetos.

```
public class C1
{
    private int val;

    public void asignar( int v )
    {
        val = v;
    }
    public boolean equals( C1 obj )
    {
        return val == obj.val;
    }

    public Object clone( )
    {
        C1 temp = new C1( );
        temp.val = val;
        return temp;
    }
}
```



Para implementar el método que compara dos objetos de la clase, teniendo en cuenta su estado interno, lo primero que debemos preguntarnos es cuándo se considera que dos objetos de la clase C1 son iguales. En nuestro ejemplo es simple, y es cuando tengan el mismo valor en su único atributo (val). Eso es lo que se refleja en la implementación del método equals(). Fíjese que los atributos del parámetro (obj) se pueden acceder directamente, puesto que dicho objeto pertenece a la misma clase que estamos definiendo.



El método de clonación debe crear una nueva instancia de la clase y pasarle el estado del objeto. En la signature de este método se debe declarar que retorna un objeto de la clase Object, algo que estudiaremos en el siguiente nivel. Aquí también accedemos directamente al atributo "val" del objeto "temp", sin necesidad de pasar por el método de asignación.



El uso de los métodos antes descritos se ilustra en el siguiente fragmento de programa:

```
C1 v1 = new C1( );
v1.asignar( 9 );
C1 v2 = ( C1 )v1.clone( );
if( v1.equals( v2 ) )
    ...
```



Note que al usar el método clone() debemos aplicar el operador de conversión ( C1 ) antes de hacer la asignación.

### 3.4. Del Análisis al Diseño

Al terminar la etapa de análisis de nuestro caso de estudio encontramos dos entidades: la central de pacientes y el paciente, con los atributos y asociaciones que aparecen en la figura 3.6. En dicho diagrama hay dos asociaciones con cardinalidad múltiple (pacientes y listaClinicas), para las cuales

es necesario tomar una decisión de diseño antes de iniciar la implementación. Lo demás corresponde a atributos de tipo simple o a objetos de la clase `String`, para los cuales no hay ninguna decisión adicional que tomar.

Hasta este momento, el invariante de las dos entidades identificadas en el análisis es el siguiente:



CentralPaciente:	Paciente:
<ul style="list-style-type: none"><li>Los códigos de los pacientes son únicos en la central</li></ul>	<ul style="list-style-type: none"><li>codigo &gt;= 0</li><li>nombre != null &amp;&amp; nombre != ""</li><li>clinica != null &amp;&amp; clinica != ""</li><li>informacionMedica != null</li><li>sexo == HOMBRE o sexo == MUJER</li></ul>

Fig. 3.6 – Diagrama de clases producto de la etapa de análisis

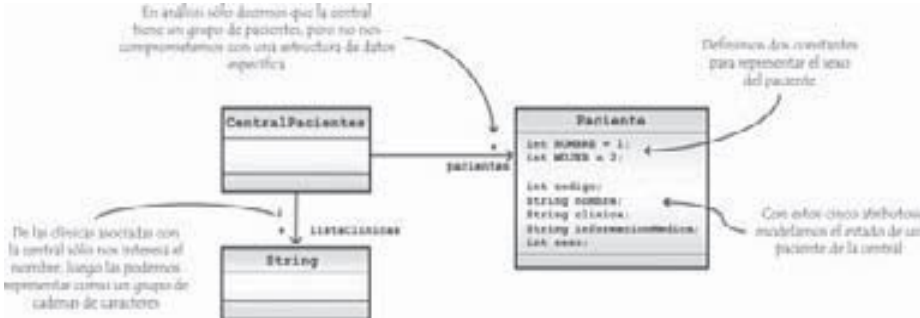
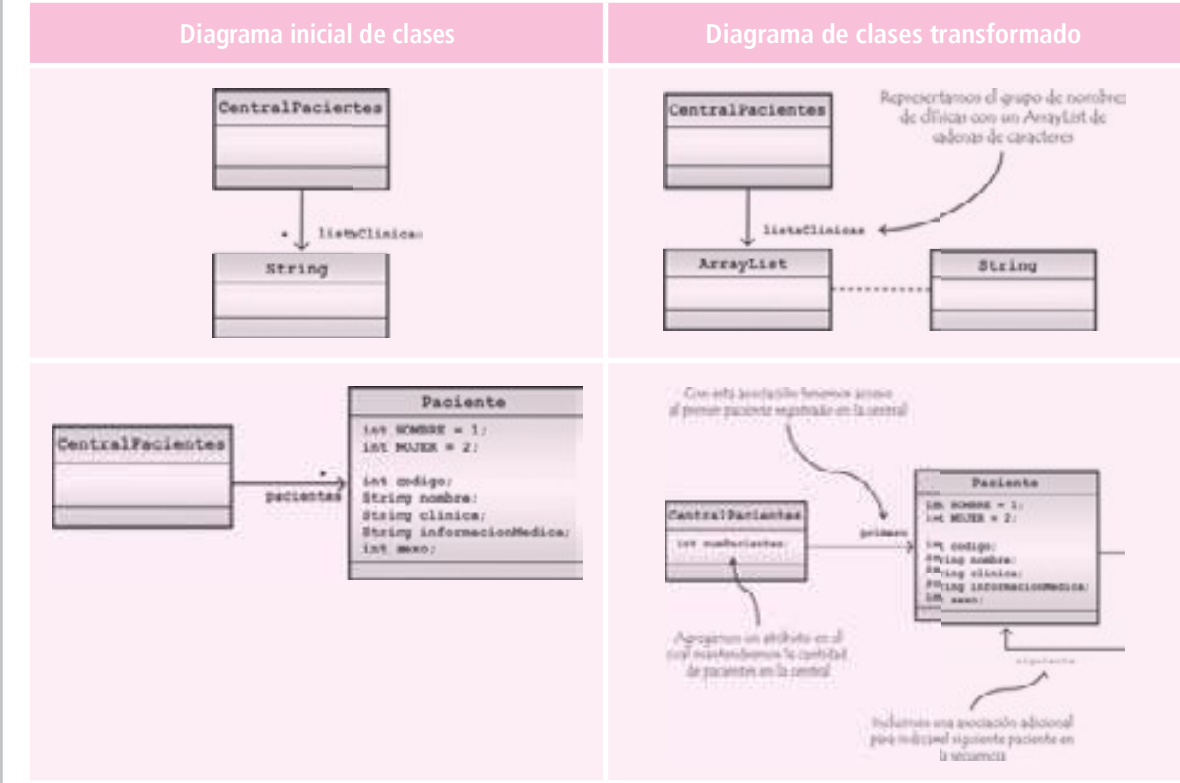


Fig. 3.7 – Transformación del diagrama de clases como parte de la etapa de diseño



El **diseño** es una etapa dentro del proceso de desarrollo de software en la cual transformamos el diagrama de clases resultado de la etapa de análisis con el fin de incorporar los requerimientos no funcionales definidos por el cliente (persistencia o distribución, por ejemplo), lo mismo que algunos criterios de calidad de la solución (eficiencia, claridad, etc.). Aquí, en particular, debemos definir las estructuras concretas de datos con las cuales vamos a representar los agrupamientos de objetos. Para el caso de estudio, las dos decisiones de diseño se pueden resumir en la figura 3.7.

Por un lado, representaremos el agrupamiento de clínicas con un `ArrayList` de cadenas de caracteres, en el que vamos a almacenar el nombre de cada una de ellas. Por otro lado, utilizaremos una estructura encadenada para representar la lista de pacientes registrados en la central. Adicionalmente, vamos a agregar un atributo en la clase `CentralPacientes` con el número de pacientes que allí se encuentran.

Las dos clases se declaran en Java de la siguiente manera:

```
public class CentralPacientes
{
    // -----
    // Atributos
    // -----

    private Paciente primero;
    private int numPacientes;
    private ArrayList listaClinicas;
}
```

```
public class Paciente
{
    // -----
    // Constantes
    // -----

    public final static int HOMBRE = 1;
    public final static int MUJER = 2;

    // -----
    // Atributos
    // -----

    private int codigo;
    private String nombre;
    private String clinica;
    private String informacionMedica;
    private int sexo;
    private Paciente siguiente;
}
```

Extendemos ahora el invariante que habíamos calculado en el análisis, para incluir los elementos que aparecieron en la etapa de diseño:

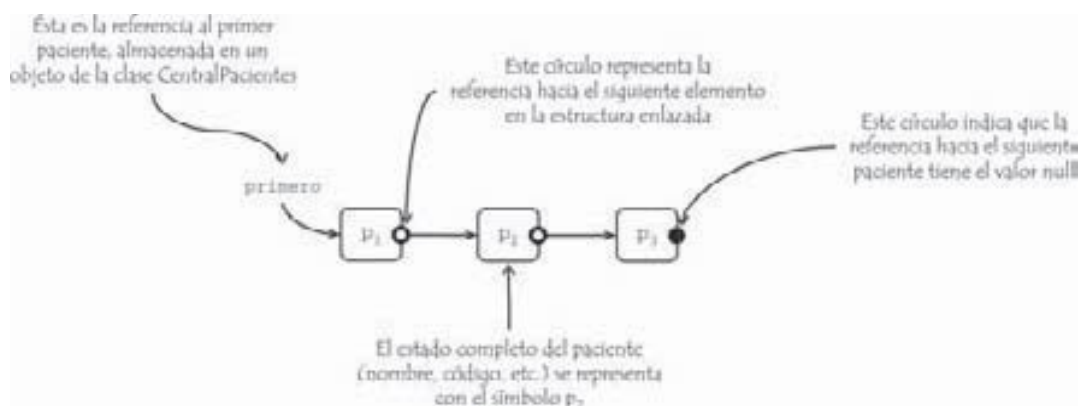
CentralPaciente:	Paciente:
<ul style="list-style-type: none"> <li>Los códigos de los pacientes son únicos en la central</li> <li><code>numPacientes == longitud de la lista de pacientes</code></li> <li><code>listaClinicas != null</code></li> </ul>	<ul style="list-style-type: none"> <li><code>codigo &gt;= 0</code></li> <li><code>nombre != null &amp;&amp; nombre != ""</code></li> <li><code>clinica != null &amp;&amp; clinica != ""</code></li> <li><code>informacionMedica != null</code></li> <li><code>sexo == HOMBRE o sexo == MUJER</code></li> </ul>

### 3.5. Estructuras Lineales Enlazadas

Vamos a introducir en esta sección una notación que corresponde a una simplificación de los diagramas de objetos de UML, para mostrar con ella los principales procesos que se tienen cuando se quiere manipular una estructura lineal sencillamente enlazada. Dicha notación se muestra en la figura 3.9, en donde apa-

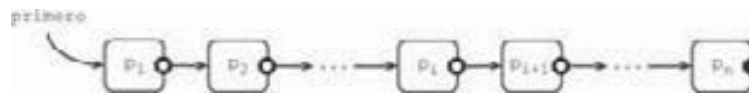
rece una central con tres pacientes registrados. Puesto que no estamos interesados en los valores exactos de los atributos de cada paciente, sino en el enlazamiento de la estructura, representamos todo el estado de cada paciente mediante un solo símbolo ( $p_1$ ,  $p_2$  y  $p_3$ ). La referencia al primer paciente la llamamos *primero*, que corresponde al nombre de la asociación que tiene la clase `CentralPacientes` hacia la estructura enlazada.

Fig. 3.9 – **Diagrama de objetos simplificado para representar la lista de pacientes**



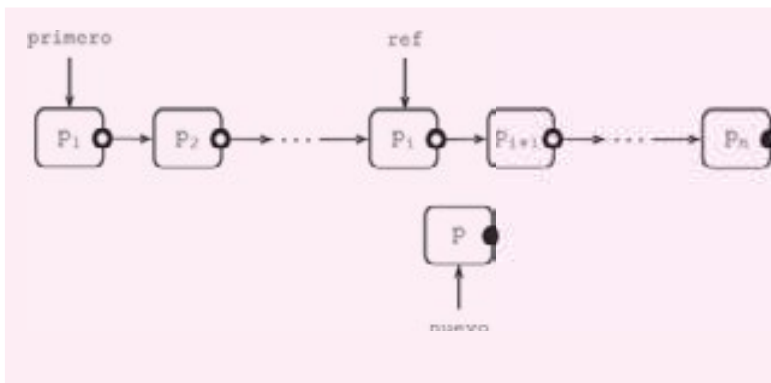
Para representar el caso general, en el cual en la lista hay  $n$  pacientes enlazados, utilizaremos la notación que se presenta en la figura 3.10.

Fig. 3.10 – **Caso general del diagrama de objetos simplificado**



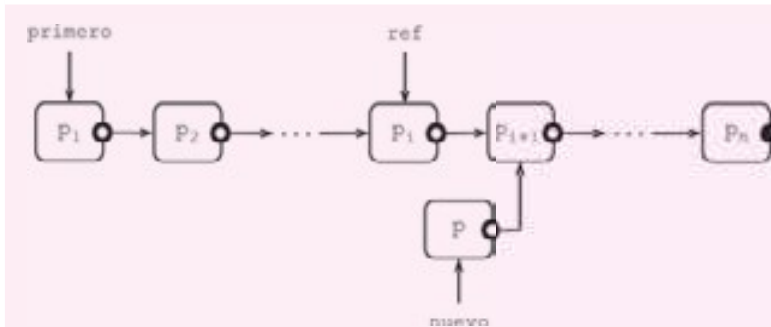
Existen tres procesos fundamentales de modificación de una estructura enlazada, los cuales se ilustran a continuación:

- Insertar un elemento después de otro del cual tenemos una referencia:

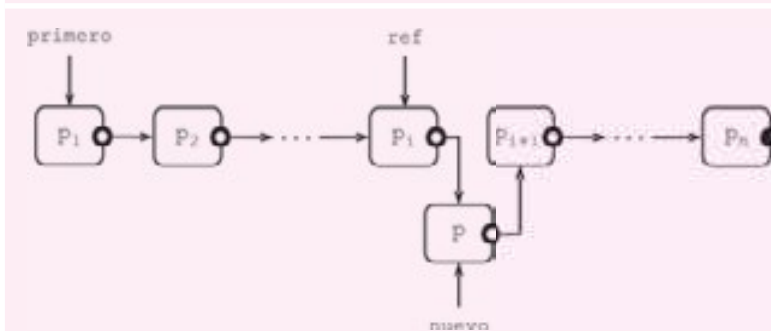


**Situación inicial:** tenemos una estructura enlazada, una referencia al punto de inserción (*ref*) y un nuevo elemento señalado por la referencia "*nuevo*".

Queremos insertar en la lista el nuevo paciente, después del que se encuentra señalado por la referencia "*ref*".

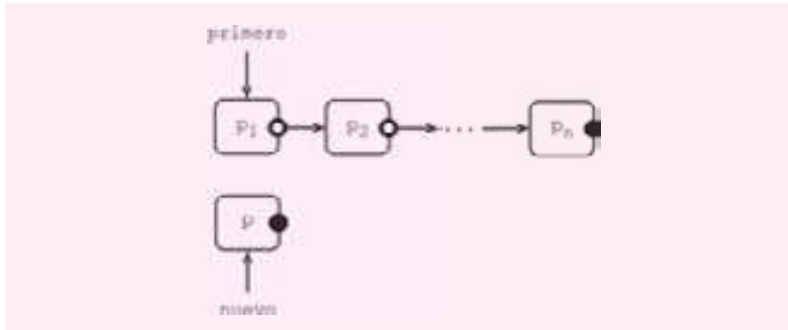


**Primer paso:** en el nuevo elemento, modificamos la referencia al siguiente paciente, de manera que señale al paciente que está en la posición "*i+1*".



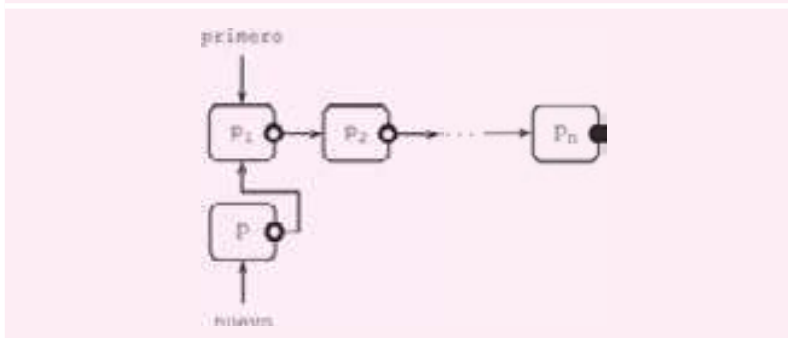
**Segundo paso:** modificamos la referencia entre el paciente "*i*" y el paciente "*i+1*" de manera que ahora se tenga en cuenta el nuevo elemento.

- Insertar un elemento como el primero de la secuencia:

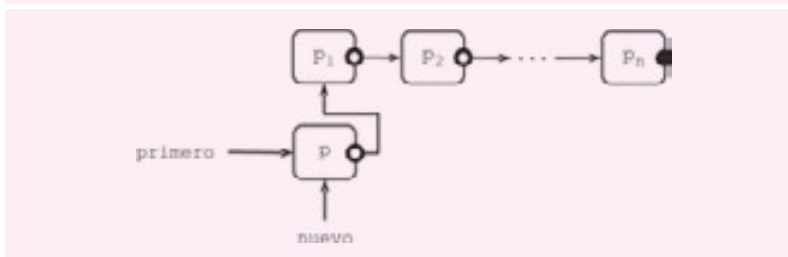


Situación inicial: tenemos una estructura enlazada y un nuevo elemento señalado por la referencia "nuevo".

Queremos agregar al nuevo paciente como el primero de la secuencia.

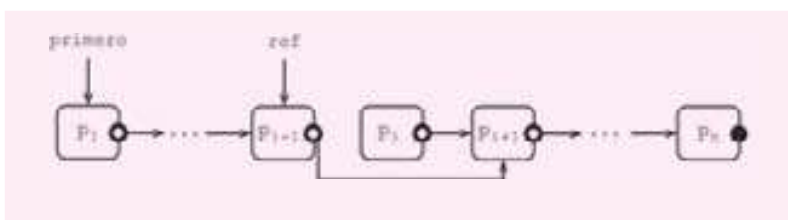


Primer paso: modificamos en el nuevo elemento la referencia al siguiente, de manera que señale al primer paciente.

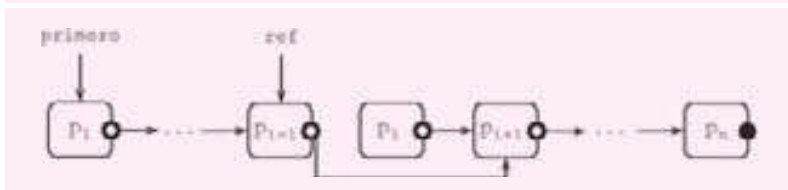


Segundo paso: modificamos la referencia "primero" para que ahora apunte al nuevo elemento.

- Eliminar un elemento, teniendo una referencia al anterior:



Situación inicial: tenemos una estructura enlazada y una referencia (`ref`) al elemento anterior al que se quiere eliminar de la estructura.



Primer paso: cambiamos la referencia al siguiente paciente, de manera que ahora apunte al paciente " $i+1$ ".

En una estructura enlazada se denomina **longitud** al número de elementos que tiene encadenados.

### 3.6. Algorítmica Básica








Pasemos ahora a desarrollar los métodos que implementan las operaciones básicas de manejo de una estructura enlazada. Comenzamos con las operaciones de localización y después abordaremos la problemática asociada con la inserción y supresión de elementos.

Vale la pena resaltar que no hay una teoría o unas instrucciones especiales para manejar este tipo de estructuras. Simplemente vamos a aprovechar la capacidad que tienen

los lenguajes de programación para manipular referencias a los objetos que se encuentran en su memoria.

#### 3.6.1. Localización de Elementos y Recorridos

Vamos a estudiar tres algoritmos de localización y un algoritmo de recorrido total, los cuales serán presentados a lo largo de los siguientes cuatro ejemplos. Supondremos para su implementación que en la clase `Paciente` están definidos los siguientes métodos:

Clase <code>Paciente</code> :	
<code>Paciente( int cod, String nom, String clin, String infoMed, int sex )</code>	 Éste es el constructor de la clase. Recibe toda la información del paciente y crea un nuevo objeto, con el atributo "siguiente" en null. En la precondition exige que la información suministrada sea válida.
<code>int darCodigo( )</code>	 Este método retorna el código del paciente.
<code>String darNombre( )</code>	 Este método retorna el nombre del paciente.
<code>String darClinica( )</code>	 Este método retorna el nombre de la clínica a la cual fue enviado el paciente.
<code>int darSexo( )</code>	 Este método retorna el sexo del paciente.
<code>String darInformacionMedica( )</code>	 Este método retorna la información médica del paciente.
<code>Paciente darSiguiente( )</code>	 Este método retorna el siguiente paciente de la estructura enlazada. Si es el último paciente de la secuencia, retorna null.

**Ejemplo 3**

**Objetivo:** Implementar el método de la clase `CentralPacientes` que permite localizar un paciente dado su código.

En este ejemplo presentamos el código del método e ilustramos gráficamente su funcionalidad.

```
public Paciente localizar( int codigo )
{
    Paciente actual = primero;

    while( actual != null &&
           actual.darCodigo( ) != codigo )
    {
        actual = actual.darSiguiente( );
    }

    return actual;
}
```

- Utilizamos una referencia auxiliar llamada "actual" para hacer el recorrido de la estructura. Dicha referencia comienza apuntando al primer elemento.
- El ciclo lo repetimos mientras que la referencia auxiliar sea distinta de null y mientras que no hayamos llegado al elemento que estamos buscando.
- El avance del ciclo consiste en mover la referencia al siguiente paciente de la lista.
- Al final retornamos el valor en el que termine la referencia auxiliar: si encontró un paciente con el código pedido, lo retorna. En caso contrario, retorna null.

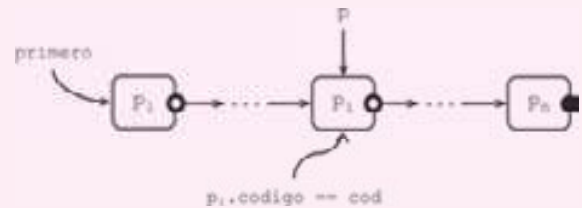
Inicial:



Llamado:

Paciente p = central.localizar( cod );

Final:

**Ejemplo 4**

**Objetivo:** Implementar el método de la clase `CentralPacientes` que permite localizar el último paciente de la estructura enlazada.

En este ejemplo presentamos el código del método e ilustramos gráficamente su funcionalidad.

```
public Paciente localizarUltimo( )
{
    Paciente actual = primero;

    if( actual != null )
    {
        while( actual.darSiguiente( ) != null )
        {
            actual = actual.darSiguiente( );
        }
    }

    return actual;
}
```

- Utilizamos de nuevo una referencia auxiliar llamada "actual". Dicha referencia comienza señalando al primer paciente de la estructura.
- Si la estructura se encuentra vacía (`primero == null`) el método retorna null.
- En el ciclo avanza elemento por elemento mientras no haya llegado a la última posición.

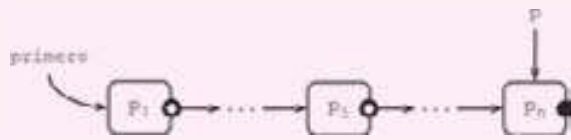
Inicial:



Llamado:

Paciente p = central.localizarUltimo( );

Final:



### Ejemplo 5



**Objetivo:** Implementar el método de la clase `CentralPacientes` que permite localizar al paciente que se encuentra antes de otro paciente del cual recibimos el código como parámetro.

En este ejemplo presentamos el código del método e ilustramos gráficamente su funcionalidad.

```

public Paciente localizarAnterior( int cod )
{
    Paciente anterior = null;

    Paciente actual = primero;

    while( actual != null &&
           actual.darCodigo( ) != cod )
    {
        anterior = actual;

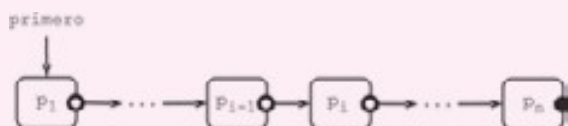
        actual = actual.darSiguiente( );
    }

    return actual != null ? anterior : null;
}

```

- Para este método utilizamos dos referencias auxiliares: una (actual), para indicar el elemento sobre el cual estamos haciendo el recorrido, y otra (anterior), que va una posición atrasada con respecto a la primera.
- Cuando la referencia "actual" llegue al paciente buscado, en la referencia "anterior" tendremos la respuesta del método.
- Dentro del ciclo avanzamos las dos referencias: "anterior" se mueve a la posición que tiene "actual" y "actual" avanza una posición sobre la secuencia de pacientes.

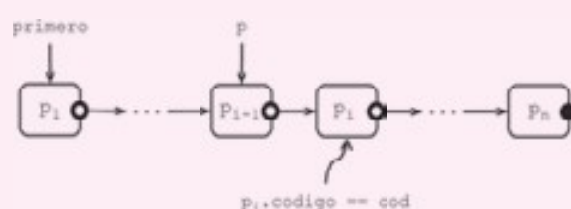
Inicial:



Llamado:

Paciente p = central.localizarAnterior(cod);

Final:





**Ejemplo 6**

**Objetivo:** Implementar el método de la clase `CentralPacientes` que calcula el número de pacientes de la estructura. Dentro del caso de estudio lo usaremos para verificar el invariante.

En este ejemplo presentamos el código del método y explicamos la estructura del algoritmo de recorrido total de este tipo de estructuras.

```
private int darLongitud( )
{
    Paciente actual = primero;

    int longitud = 0;

    while( actual != null )
    {
        longitud++;

        actual = actual.darSiguiente( );
    }

    return longitud;
}
```



Vamos a utilizar una referencia auxiliar (`actual`) para hacer el recorrido de la estructura. El ciclo termina cuando dicha referencia toma el valor `null`.



Declaramos también una variable de tipo entero (`longitud`), en la cual iremos acumulando el número de elementos recorridos.



Al final del ciclo, en la variable "`longitud`" tendremos el número total de pacientes registrados en la central.

A continuación planteamos, como tarea al lector, la implementación de cuatro métodos de localización y reco-

rrido, que corresponden a variantes de lo presentado en los ejemplos anteriores:

**Tarea 2**

**Objetivo:** Implementar algunos métodos de la clase `CentralPacientes` para practicar la algorítmica de localización y recorrido.

Desarrolle los cuatro métodos que se plantean a continuación.

```
public class CentralPacientes
{
    private Paciente primero;

    public int darPacientesEnClinica( String nomClinica )
    {
        // ...
    }

    // ...
}
```



Calcula el número de pacientes que fueron enviados a una misma clínica, cuyo nombre es dado como parámetro.

```
public Paciente darPacienteMenorCodigo( )  
{
```



Retorna el paciente con el menor código en la central. Si no hay ninguno, retorna null.

```
}
```

```
public boolean hayMasHombres( )  
{
```



Indica si hay más hombres que mujeres registrados en la central.

```
}
```

```
public Paciente darUltimaMujer( )  
{
```



Retorna la última mujer en la estructura. Si no hay ninguna retorna null.

```
}
```

```
}
```

### 3.6.2. Supresión de Elementos

Para eliminar un elemento de una estructura enlazada, vamos a dividir las responsabilidades entre

las clases `CentralPacientes` y `Paciente`, tal como se muestra en los siguientes ejemplos:

#### Ejemplo 7



**Objetivo:** Presentar los métodos de la clase `Paciente` que nos van a servir para eliminar un elemento de la lista.

En este ejemplo explicamos los métodos `cambiarSiguiente()` y `desconectarSiguiente()` de la clase `Paciente`.

```
public class Paciente
{
    // -----
    // Atributos
    // -----

    private Paciente siguiente;

    public void cambiarSiguiente( Paciente pac )
    {
        siguiente = pac;
    }

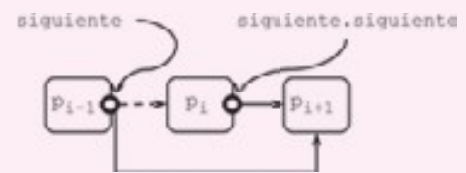
    public void desconectarSiguiente( )
    {
        siguiente = siguiente.siguiente;
    }
}
```



El primer método cambia la referencia que tiene al siguiente elemento y ahora señala al paciente que llega como parámetro.



El segundo método tiene como precondition que no es el último paciente de la lista. Para desencadenar el siguiente, se contenta con apuntar al que está después del que le sigue, tal como se muestra en la figura:



#### Ejemplo 8



**Objetivo:** Mostrar el método de la clase `CentralPacientes` que elimina un paciente de la central, dado su código.

En este ejemplo presentamos el método que elimina un elemento de una lista enlazada.

```
public class CentralPacientes
{
    // -----
    // Atributos
    // -----

    private Paciente primero;
    private int numPacientes;

    public void eliminarPaciente( int cod )
    {
        throws NoExisteException
        {
            if( primero == null )
                throw new NoExisteException( cod );
        }
    }
}
```



El método considera tres casos: (1) la lista está vacía, (2) es el primer elemento de la lista o (3) está en un punto intermedio de la lista (incluyendo el final).



En el primer caso, lanzamos una excepción informando que el paciente no fue encontrado.



En el segundo caso, cambiamos la referencia al primero y lo ponemos a apuntar al segundo de la lista.

```

        else if( cod == primero.darCodigo( ) )
        {
            primero = primero.darSiguiente( );
        }

        else
        {
            Paciente anterior = localizarAnterior( cod );
            if( anterior == null )
                throw new NoExisteException( cod );

            anterior.desconectarSiguiente( );
        }

        numPacientes--;
        verificarInvariante( );
    }
}

```



En el tercer caso, localizamos el paciente anterior al que queremos eliminar, utilizando el método `localizarAnterior()` desarrollado en el ejemplo 5. Una vez que ha sido localizado, procedemos a pedirle que desencadene al siguiente, utilizando para esto el método `desconectarSiguiente()` de la clase `Paciente`.



En el caso anterior, si el paciente con el código pedido no existe, lanzamos una excepción.



Finalmente, disminuimos en uno el número de pacientes de la lista (valor almacenado en el atributo `numPacientes`) y verificamos que el objeto continúe cumpliendo con el invariante.



En el sitio web puede localizar la herramienta llamada “Laboratorio de Estructuras de Datos” y resolver algunos de los retos que allí se plantean, como una manera de complementar el trabajo desarrollado hasta este punto del nivel.

En la siguiente tarea trabajaremos en algunos métodos cuyo objetivo es eliminar elementos de una estructura enlazada:

### Tarea 3



**Objetivo:** Implementar algunos métodos de la clase `CentralPacientes` para practicar la algorítmica de supresión de elementos.

Desarrolle los tres métodos que se plantean a continuación.

```

public class CentralPacientes
{
    private Paciente primero;

```

<pre>public void eliminarUltimo( ) {  }  </pre>	 Elimina al último paciente de la lista.
<pre>public void eliminarHombres( ) {  }  </pre>	 Elimina de la lista a todos los pacientes de sexo masculino.
<pre>public void eliminarMenorCodigo( ) {  }  </pre>	 Elimina de la lista al paciente con el menor código.

### 3.6.3. Inserción de Elementos

Para insertar un elemento en una estructura enlazada, vamos a dividir las responsabilidades entre las clases `CentralPacientes` y `Paciente`, de la manera como se muestra en los siguientes ejemplos:

#### Ejemplo 9



**Objetivo:** Presentar los métodos de la clase `Paciente` que nos van a servir para enlazar un nuevo elemento en una secuencia.

En este ejemplo explicamos el método `insertarDespues (paciente)` de la clase `Paciente`, el cual permite pedirle a un objeto de dicha clase que cambie su referencia al siguiente elemento.

```
public class Paciente
{
    // -----
    // Atributos
    // -----

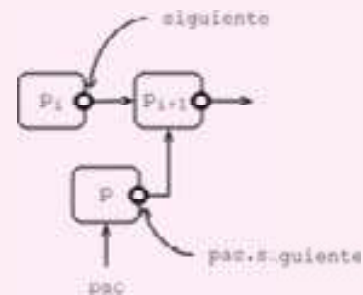
    private Paciente siguiente;

    public void insertarDespues( Paciente pac )
    {
        pac.siguiente = siguiente;

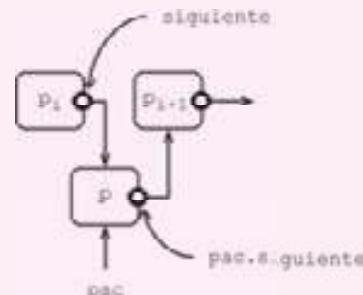
        siguiente = pac;
    }
}
```

Este método cambia la referencia al siguiente objeto de la estructura, de manera que ahora indique al paciente que llega como parámetro. Eso tiene como efecto que se inserta el nuevo paciente en la secuencia.

Con la primera instrucción hacemos que el nuevo paciente haga referencia al siguiente:



Con la segunda instrucción incluimos al nuevo paciente en la secuencia:



Ahora presentaremos los cuatro métodos de inserción que necesitamos para implementar los requerimientos funcionales del caso de estudio: un método que agregue un paciente como el primero de la lista (ejemplo 10), un método que agregue un paciente al final de la lista (ejemplo 11), un método que incluya

un nuevo paciente después de otro de la lista (ejemplo 12) y un método para insertar un nuevo paciente antes de otro de la lista (ejemplo 13). Con esos cuatro métodos completamos la algorítmica de inserción en estructuras enlazadas.

**Ejemplo 10**

**Objetivo:** Presentar el método de la clase `CentralPacientes` que inserta un nuevo paciente antes del primero de la lista.

La precondition del método garantiza que no hay ningún otro paciente en la central con el mismo código del paciente que se va a agregar.

```
public class CentralPacientes
{
    // -----
    // Atributos
    // -----

    private Paciente primero;
    private int numPacientes;

    public void agregarPacienteAlComienzo( Paciente pac )
    {
        if( primero == null )
        {
            primero = pac;
        }
        else
        {
            pac.cambiarSiguiente( primero );
            primero = pac;
        }

        numPacientes++;
        verificarInvariante( );
    }
}
```



El método contempla dos casos: si la lista de pacientes se encuentra vacía (`primero == null`) basta con cambiar el valor de la referencia que señala al primero, de manera que ahora apunte al nuevo paciente.



Si la lista no está vacía, agregamos el nuevo elemento en la primera posición.



Finalmente actualizamos el atributo que contiene el número de elementos de la secuencia y verificamos que la estructura cumpla el invariante.

**Ejemplo 11**

**Objetivo:** Presentar el método de la clase `CentralPacientes` que inserta un nuevo paciente al final de la lista.

La precondition del método garantiza que no hay ningún otro paciente en la central con el mismo código del paciente que se va a agregar.

```
public class CentralPacientes
{
    // -----
    // Atributos
    // -----

    private Paciente primero;
    private int numPacientes;

    public void agregarPacienteAlFinal( Paciente pac )
    {
        if( primero == null )
        {
            primero = pac;
        }
        else
        {
            Paciente p = localizarUltimo( );
            p.insertarDespues( pac );
        }

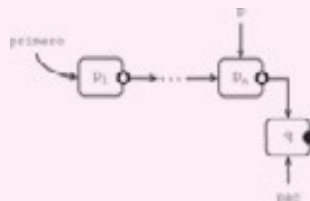
        numPacientes++;
        verificarInvariante( );
    }
}
```



Este método considera dos casos posibles: si la lista es vacía (`primero == null`), deja el nuevo nodo como el primero (y también el último, porque es el único paciente).



Si la lista no es vacía, utiliza el método que permite localizar el último elemento de la estructura (`localizarUltimo()`). Una vez que tenemos la referencia "p" señalando al último paciente, le pedimos que agregue al nuevo elemento como siguiente.



**Ejemplo 12**






**Objetivo:** Presentar el método de la clase `CentralPacientes` que inserta un nuevo paciente después de un paciente que se encuentra en la lista y del cual recibimos su código como parámetro. La precondition del método garantiza que no hay ningún otro paciente en la central con el mismo código del paciente que se va a agregar.

```
public class CentralPacientes
{
    private Paciente primero;
    private int numPacientes;

    public void agregarPacienteDespuesDe( int cod,
                                           Paciente pac )
        throws NoExisteException
    {
        Paciente anterior = localizar( cod );

        if( anterior == null )
            throw new NoExisteException( cod );
        else
            anterior.insertarDespues( pac );

        numPacientes++;
        verificarInvariante( );
    }
}
```



-  El método recibe como parámetro el código del paciente después del cual debemos hacer la inserción y el nuevo paciente de la central.
-  Si no hay ningún paciente con dicho código, el método lanza una excepción.
-  El primer paso es localizar al paciente con el código "cod" dentro de la lista y dejar sobre él la referencia "anterior". Si no lo logra localizar, lanza la excepción.
-  Luego, le pide al paciente referenciado por la variable "anterior" que incluya al nuevo paciente dentro de la secuencia.
-  Finalmente, incrementa el atributo que contiene el número de pacientes y verifica el cumplimiento del invariante.

**Ejemplo 13**

**Objetivo:** Presentar el método de la clase `CentralPacientes` que inserta un nuevo paciente antes de un paciente que se encuentra en la lista y del cual recibimos su código como parámetro. La precondition del método garantiza que no hay ningún otro paciente en la central con el mismo código del paciente que se va a agregar.

```
public class CentralPacientes
{
    private Paciente primero;
    private int numPacientes;

    public void agregarPacienteAntesDe( int cod,
                                         Paciente pac )
        throws NoExisteException
    {
        if( primero == null )
            throw new NoExisteException( cod );
```

-  En este método debemos considerar tres casos distintos. En el primer caso, si la lista es vacía, lanzamos una excepción indicando que el paciente de código "cod" no existe.
-  En el segundo caso, sabiendo que la lista no es vacía, establecemos si el paciente de código "cod" es el primero, caso en el cual el nuevo paciente debe ser insertado en la primera posición.



```

else if( cod == primero.darCodigo( ) )
{
    pac.cambiarSiguiente( primero );
    primero = pac;
}
else
{
    Paciente anterior = localizarAnterior( cod );
    if( anterior == null )
        throw new NoExisteException( cod );

    anterior.insertarDespues( pac );
}
numPacientes++;
verificarInvariante( );
}
}

```



Para el caso general, utilizamos el método que nos permite localizar el anterior a un elemento (`localizarAnterior()`) y dejamos sobre dicho paciente la referencia "anterior". Luego, le pedimos a dicho elemento que incluya al nuevo paciente dentro de la secuencia.

En la siguiente tarea vamos a trabajar sobre algoritmos que modifican los enlazamientos de la estructura lineal de pacientes que manejamos en la central.

#### Tarea 4



**Objetivo:** Implementar algunos métodos de la clase `CentralPacientes` para practicar la algorítmica de modificación del enlazamiento en una estructura lineal.

Desarrolle los métodos que se plantean a continuación.

```

public class CentralPacientes
{
    private Paciente primero;

    public void invertir( )
    {

```



Invierte la lista de pacientes, dejando al primero de último, al segundo de penúltimo, etc.

```
}
```

```
public void pasarAlComienzoPaciente( int cod )  
                                   throws NoExisteException  
{
```

```
}
```

```
public void ordenar( )  
{
```

```
}
```



Mueve al comienzo de la lista al paciente que tiene el código "cod". Si no lo encuentra lanza una excepción.



Ordena la lista de pacientes de la central ascendentemente por código. Utiliza el algoritmo de selección.

```
public void pasarAlFinalPacientes( String nomClinica )
{

}

}
```



Mueve al final de la lista a todos los pacientes que fueron remitidos a una clínica, cuyo nombre llega como parámetro.

### 3.7. Patrones de Algoritmo

La mayoría de los algoritmos para manejar estructuras enlazadas corresponden a especializaciones de cuatro patrones distintos de recorrido y localización, los cuales se resumen a continuación. Las variantes corresponden a cálculos o a cambios en los enlaces que se hacen a medida que se avanza en el recorrido, o al terminar el mismo.

- Patrón de recorrido total:

El recorrido total es el patrón de algoritmo más simple que hay sobre estas estructuras. Consiste en pasar una vez sobre cada uno de los elementos de la secuencia, ya sea para contar el número de ellos o para calcular alguna propiedad del grupo. El esqueleto del algoritmo es el siguiente:

```
Nodo actual = primero;

while( actual != null )
{
    ...
    actual = actual.darSiguiente( );
}
```



Suponemos que los elementos de la estructura pertenecen a una clase llamada Nodo.



El esqueleto del patrón de recorrido total utiliza una referencia auxiliar (actual) que nos permite desplazarnos desde el primer elemento de la estructura (primero) hasta que se hayan recorrido todos los elementos enlazados (actual==null).



En cada ciclo se utiliza el método que permite avanzar hacia el siguiente nodo enlazado.

- Patrón de recorrido parcial – Localización del último elemento:

Es muy común en la manipulación de estructuras enlazadas que estemos interesados en localizar el último elemento de la secuencia. Para esto utilizamos el




patrón de recorrido parcial, con terminación al llegar al final de la secuencia. Fíjese que con este esqueleto no hacemos un recorrido total, puesto que siempre se queda el último elemento sin recorrer. Sólo sirve para posicionar una referencia sobre el último elemento.

```

if( actual != null )
{
    Nodo actual = primero;

    while( actual.darSiguiente( ) != null )
    {
        actual = actual.darSiguiente( );
    }
    ...
}

```

-  Suponemos de nuevo que los elementos de la estructura enlazada son objetos de la clase `Nodo`.
-  El esqueleto debe considerar aparte el caso en el cual la lista se encuentra vacía.
-  En el caso general, utilizamos una referencia auxiliar para hacer el recorrido. La salida del ciclo se hace cuando la referencia al siguiente sea nula (en ese momento la variable "actual" estará señalando el último elemento de la estructura).

- Patrón de recorrido parcial – Hasta que una condición se cumpla sobre un elemento:

Con este patrón de algoritmo podemos recorrer la




estructura hasta que detectemos que una condición se cumple sobre el elemento actual del recorrido. Con este esqueleto se resuelven problemas como el de localizar un valor en una estructura enlazada.

```

Nodo actual = primero;

while( actual != null && !condicion )
{
    actual = actual.darSiguiente( );
}
...

```

-  Suponemos de nuevo que los elementos de la estructura enlazada son objetos de la clase `Nodo`.
-  El esqueleto debe considerar aparte el caso en el cual la lista se encuentra vacía.
-  En el caso general, utilizamos una referencia auxiliar para hacer el recorrido. La salida del ciclo se hace cuando la referencia al siguiente sea nula (en ese momento la variable "actual" estará señalando al último elemento de la estructura).

- Patrón de recorrido parcial – Hasta que una condición se cumpla sobre el siguiente elemento:

Es muy frecuente encontrar algoritmos que utilizan este patrón, puesto que para hacer ciertas modifica-




ciones sobre una estructura simplemente enlazada es común tener que obtener una referencia al anterior elemento de uno que cumpla una propiedad. Piense, por ejemplo, en el problema de agregar un elemento antes de alguno que tenga un valor dado.

```

Nodo anterior = null;
Nodo actual = primero;

while( actual != null && !condicion )
{
    anterior = actual;
    actual = actual.darSiguiente( );
}
...

```

-  Este esqueleto utiliza dos referencias: una que tiene como valor el elemento actual del recorrido (actual) y otra que va una posición atrasada con respecto a la primera (anterior).
-  Al terminar el ciclo, en la variable "anterior" tenemos una referencia al elemento anterior al que cumple la propiedad.
-  Al igual que en el esqueleto anterior, la propiedad se representa con la variable "condicion", pero podría tratarse de cualquier expresión de tipo lógico.