

# Laboratorio de Programación II

## Ingeniería Técnica de Gestión, grupo A

### Facultad de Informática Universidad Complutense de Madrid

#### Curso 2005-2006

### Práctica 3: Sistema de Reconocimiento de Delincuentes<sup>1</sup>, v 3.0

Fecha límite de entrega: 20-3-2006

## 1. Introducción

Los objetivos de esta práctica son: mostrar nuevos conceptos de la orientación a objetos como la herencia, el polimorfismo y la vinculación dinámica; introducir la memoria dinámica como medio para implementar estructuras de tamaño variables en tiempo de ejecución; mostrar la utilización de ficheros como mecanismo de almacenamiento permanente de datos; y describir nuevos elementos del lenguaje de modelado UML (*Unified Modelling Language*). Los principales cambios sobre la práctica 2 son:

- **Orientación Objetos.** Implementación con clases y objetos del sistema que trabaja con el *Centro Europeo de Lucha contra la Delincuencia*. Introducción de una jerarquía de caras usando herencia.
- **Memoria dinámica para el contenedor de delincuentes.** Utilización de contenedores de tamaño variable implementados *ad hoc* con punteros para almacenar los delincuentes del sistema. Se trata de reemplazar los *arrays* empleados en la práctica 2 por listas doblemente enlazadas con memoria dinámica.
- **Ficheros para almacenamiento permanente.** Uso de ficheros en modo texto para salvar la información sobre los delincuentes entre ejecuciones.

La práctica se ejecutará con la opción *CodeGuard* del entorno activada para asegurar que no hay memoria dinámica sin liberar o se invocan métodos sobre áreas de memoria ya liberadas.

**No olvidéis incluir en todos los ficheros del programa vuestro nombre, grupo en esta asignatura y titulación.**

## 2. Desarrollo

Nuestros éxitos con las versiones anteriores del Sistema de Reconocimiento de Delincuentes (SRD) (ver prácticas 1 y 2) han cimentado nuestra posición como proveedores de servicios informáticos para el *Centro Europeo de Lucha contra la Delincuencia* (CELD). Fruto de las nuevas necesidades emergentes en el espacio de Schengen, el CELD ha mostrado su deseo de realizar una nueva versión de nuestro sistema. Por supuesto, nuestra empresa, con su modelo de desarrollo evolutivo en espiral y sus avanzadas técnicas de orientación a objetos, está preparada para el reto (y el dinero que le reportará).

Tras las correspondientes reuniones con los clientes se han descubierto varias mejoras a introducir sobre la versión previa. Estas se han plasmado en el Documento de Especificación de

---

<sup>1</sup> Idea original de la práctica cortesía de Jorge J. Gómez Sanz, Departamento de Sistemas Informáticos y Programación, UCM.

Requisitos correspondiente, que constituye la base de nuestro contrato con el CELD. Los requisitos son los siguientes:

- 1) *Parte 2.1.* Implementación orientada a objetos de un sistema con múltiples clases de caras, herederas todas ellas de la clase *Cara* de la práctica 2. Ver el epígrafe 2.1 para más detalles.
- 2) *Parte 2.2.* Implementación del contenedor de delincuentes como una lista doblemente enlazada de delincuentes con memoria dinámica. Ver el epígrafe 2.2 para más detalles.
- 3) *Parte 2.3.* Almacenamiento de la base de delincuentes en ficheros para su posterior recuperación en nuevas ejecuciones del sistema. Ver el epígrafe 2.3 para más detalles.
- 4) *Parte 2.4.* Fruto de las nuevas funcionalidades y de las conversaciones con los usuarios se han aprobado algunos cambios en el aspecto general de la interfaz, como es el uso de menús o la visualización de las fotos de los delincuentes. Ver el epígrafe 2.4 para más detalles.

Nota: Las clases que forman parte de la implementación de esta práctica se muestran a lo largo de los diagramas UML en las siguientes secciones. Presta atención a las indicaciones de qué métodos han de ser **virtual** y **const**. Las clases con sus atributos y métodos indicados son las mínimas que deberían aparecer. Puedes añadir nuevas clases, métodos y atributos. Tanto si introduces cambios en los elementos incluidos en esta especificación como si añades nuevos debes indicar el motivo por el que tú implementación es mejor.

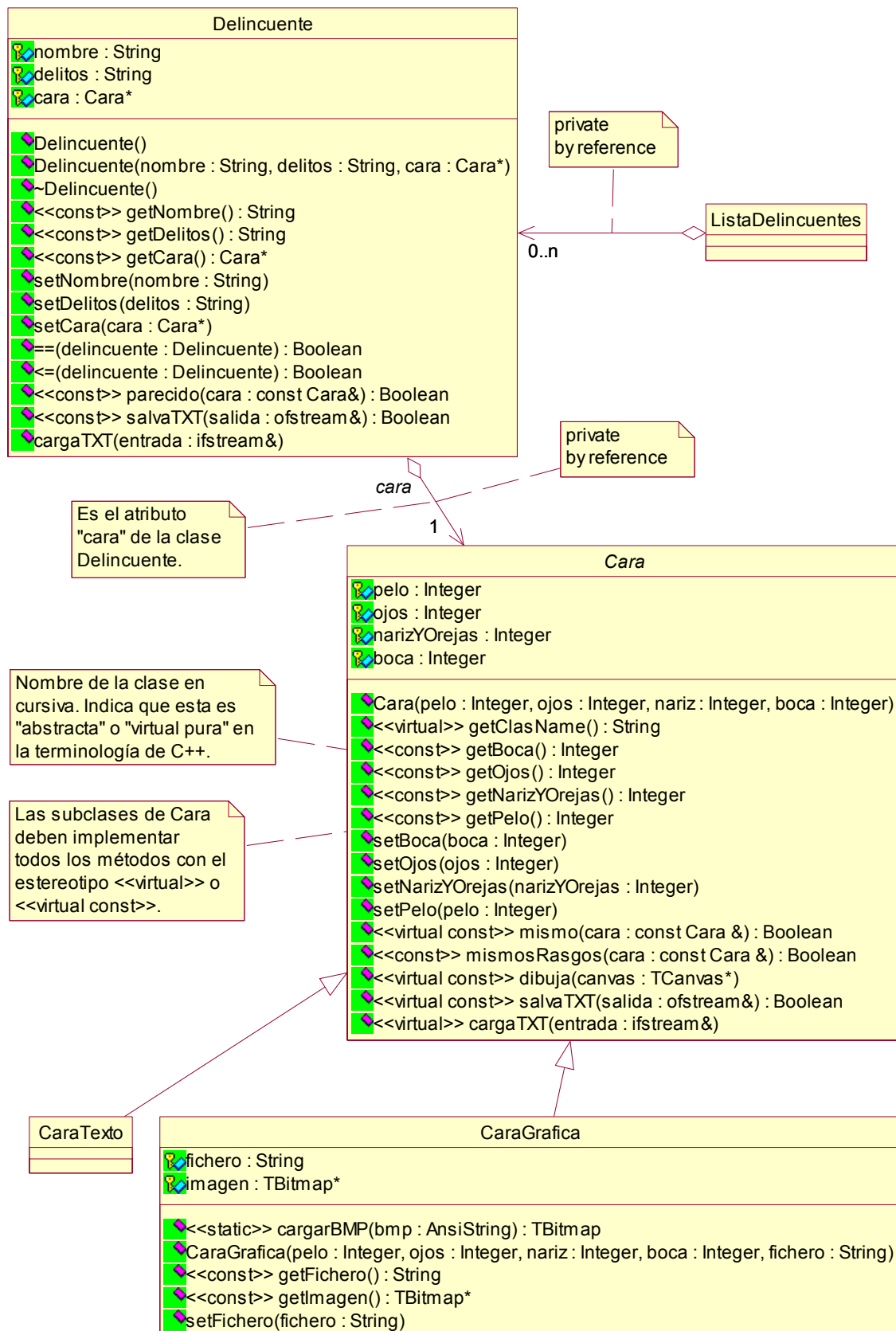
## 2.1. Jerarquía de clases de caras

La Ilustración 1 muestra las modificaciones en las clases que implementan los *Delincuentes* y *Caras* con respecto a la práctica 2. Los principales cambios son:

1. El atributo *cara* en la clase *Delincuente* es ahora un *Cara\**. Los métodos de la clase reciben y devuelven objetos *Cara* usando punteros.
2. La clase *Cara* tiene un nuevo método `public void dibuja(TCanvas* canvas)`. Este método dibuja la información de la cara sobre el *TCanvas* argumento.
3. Creación de las clases *CaraTexto* y *CaraGrafica*. Como parte de la evolución de nuestro sistema, el CELD quiere digitalizar la cara de los delincuentes. Éste será un proceso que ocurrirá gradualmente. Mientras que se concluye, el sistema ha de soportar caras donde sólo se conoce la descripción del criminal a través e sus rasgos (i.e. la clase *CaraTexto*) y otras donde además se incluye la fotografía del criminal (i.e. la clase *CaraGrafica*).

La clase *Cara* es una clase *virtual pura* de C++ (abstracta). Ello quiere decir que no se permite crear objetos de esta clase. Únicamente se introduce para definir la parte de común de interfaz y comportamiento para toda la jerarquía de caras. Fíjate que las subclases *CaraTexto* y *CaraGrafica* han de redefinir los métodos marcados como *virtual* en la clase *Cara*. *CaraTexto* tiene el mismo comportamiento de la clase *Cara* en las prácticas 1 y 2. *CaraGrafica* añade a la cara “textual” que hemos usado hasta ahora una imagen del delincuente. Esta imagen se encuentra almacenada en un fichero *bitmap* cuyo nombre se guarda en el atributo *fichero*. A fin de no tener que cargar la imagen cada vez que visualicemos el delincuente con esa cara, almacenaremos el *bitmap* cargado en el atributo *imagen* de *CaraGrafica*. Cuando se de valor al atributo *fichero* se aprovechara también para cargar el atributo *imagen*. La carga de *bitmaps* la realizaremos con el método estático de *CaraGrafica*:

```
Graphics::TBitmap* CaraGrafica::cargarBMP(AnsiString bmp){
    Graphics::TBitmap *BMP=&Graphics::TBitmap();
    BMP->LoadFromFile(Bmp);
    return BMP;
}
```



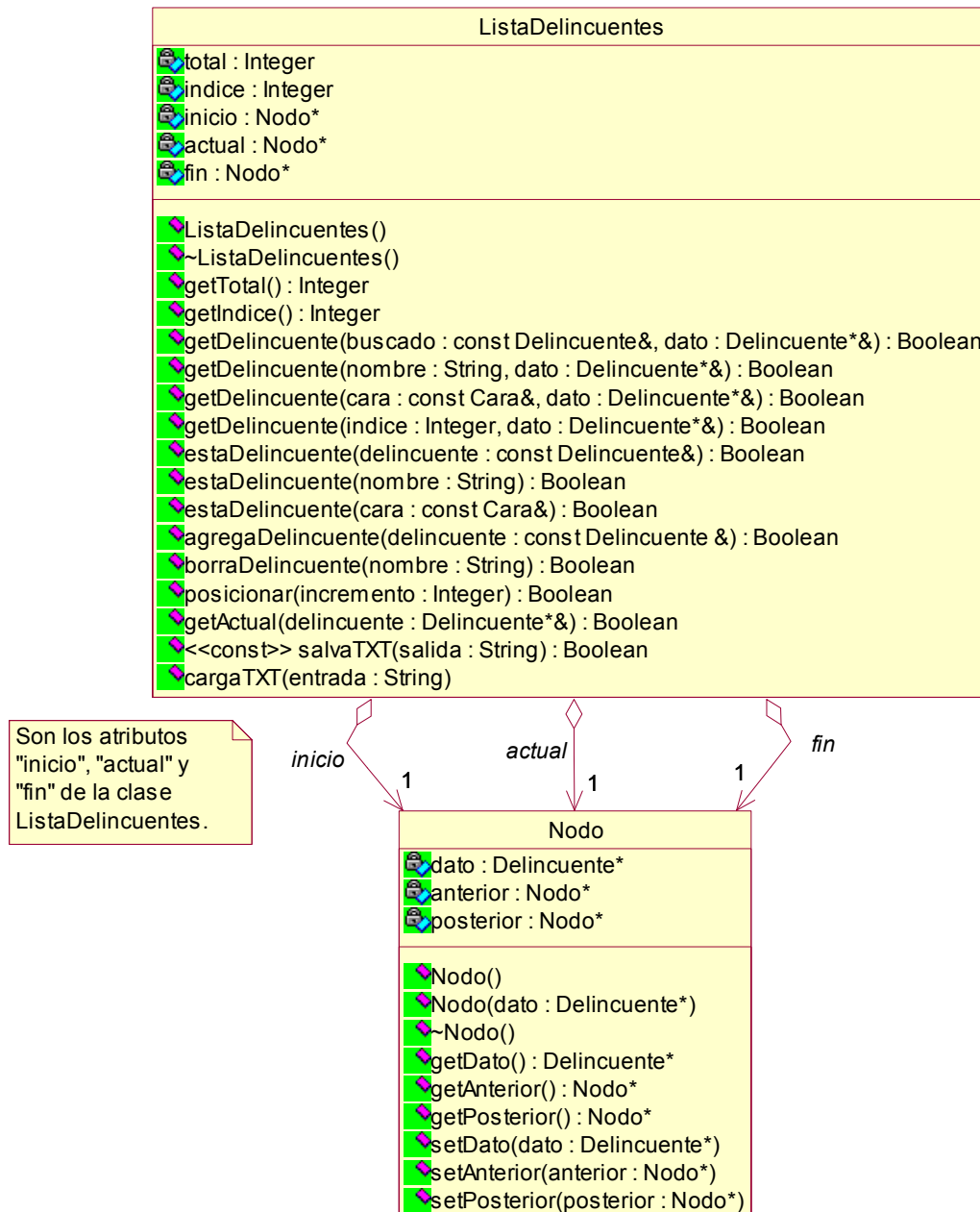
**Ilustración 1.** Diagrama de clases UML para la aplicación.

A propósito de las caras, fíjate que hemos separado en *Cara* las comparaciones en 2 métodos *mismo* y *mismosRasgos*. El método *mismo* considera todos los atributos de un objeto *Cara* (o de un objeto de una de sus subclases). Es decir, 2 caras son iguales si sus atributos *pelo*,

*ojos*, *narizYOrejas*, *boca* y *fichero* son iguales. Los 4 primeros atributos forman parte de cualquier cara, por lo que siempre pueden ser comparados. En el caso de sólo una de las caras sea e la clase *CaraGrafica* (por tanto con atributo *fichero*) se considerará que las caras son distintas. El método *mismosRasgos* es como el método *mismo* pero sólo compara los atributos *pelo*, *ojos*, *narizYOrejas* y *boca*.

En la clase *Delincuente* ahora el atributo *cara* es un *Cara\**. Fíjate en cómo son las signaturas de los métodos que manejan las caras. Las comparaciones entre delincuentes con los operadores relacionales (e.g. ==, <=) siguen considerando sólo el atributo *nombre*.

## 2.2. Lista doblemente enlazada



**Ilustración 2.** Diagrama de clases UML para la *ListaDelincuentes*.

El contenedor de delincuentes será en esta versión la clase *ListaDelincuentes*. Ésta es un recubrimiento de una lista implementada con punteros. Esta lista sustituye al *array* de la práctica 2. La Ilustración 2 muestra el correspondiente diagrama de clases.

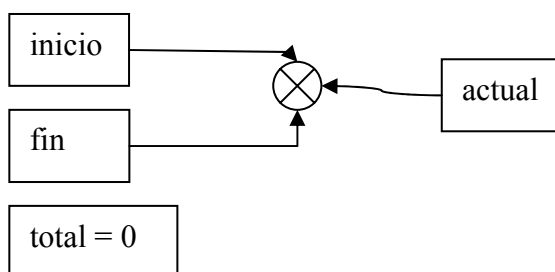
*ListaDelincentes* es una lista doblemente enlazada donde cada nodo conoce su anterior y siguiente en la lista. Un objeto *ListaDelincentes* contiene 3 punteros (tipo *Nodo\**) y 2 enteros. Los punteros apuntan al nodo inicial de la lista (atributo *inicio*), final (atributo *fin*) y al punto de interés o nodo actual de la lista (atributo *actual*). El atributo *total* indica el número de elementos en la lista. El atributo *indice* da el número de orden del nodo apuntado por *actual* en la lista. El valor de *indice* está en 0 y *total* -1.

La lista contiene los métodos habituales para buscar elementos, añadir y borrar. Estas operaciones, además de cambiar los punteros *inicio* y *fin* y el valor de *total*, han de actualizar *indice* y *actual* al último elemento manipulado. Al igual que en la práctica 2, el método *posicionar* funciona con incrementos (e.g. *incremento* = -1 indica mover *indice* y *actual* a la posición anterior) y altera el valor de *indice* y *actual*. El método *getActual* devuelve el dato del nodo apuntado por el atributo *actual*.

La clase *Nodo* tiene 3 atributos: *dato*, *anterior* y *posterior*. En *dato* se almacena el *Delincuente\** contenido en el objeto *Nodo*, mientras que *anterior* y *posterior* son punteros a los objetos *Nodo* anterior y posterior en la lista.

En las siguientes secciones dentro de este apartado se muestra como implementar las operaciones necesarias sobre la lista enlazada. Se trata de las operaciones para inicializar la lista, insertar y eliminar nodos de ella, buscar un dato y destruir la estructura. En las siguientes secciones, los elementos *inicio*, *actual*, *fin* y *total* son los atributos de la clase *ListaDelincentes*. Las actualizaciones del atributo *indice* no se muestran en los diagramas. El círculo con la X en su interior representa el valor especial *NULL* para un puntero, es decir, el puntero apunta explícitamente a **nada**. *Primero*, *Segundo*, *Tercero*, *Cuarto* y *Nuevo* son objetos de la clase *Nodo* de la Ilustración 2.

### 2.2.1. Inicialización

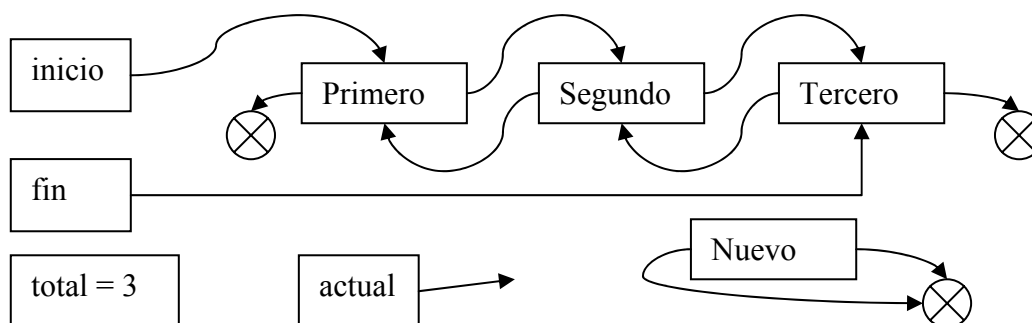


### 2.2.2. Inserción

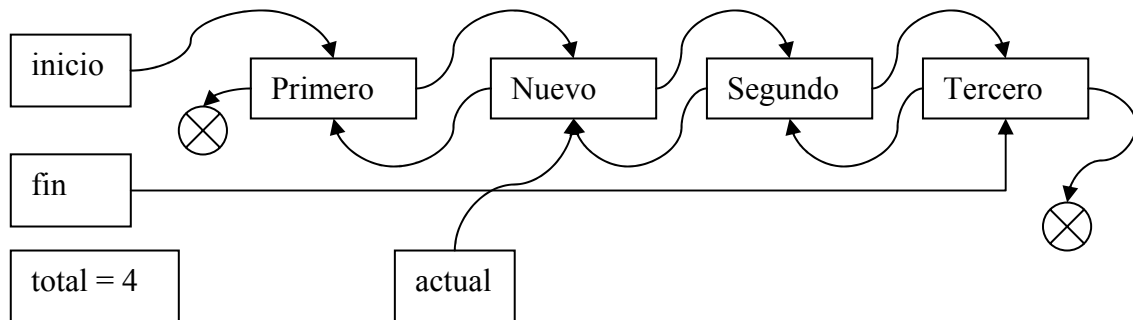
Antes de la inserción del nodo *Nuevo*.

*Primero*->*dato* <= *Segundo*->*dato* <= *Tercero*->*dato*.

El *Nodo* al que apunta *actual* antes de la inserción no es relevante.



Después de insertar *Nuevo* entre *Primero* y *Segundo*.  
 $\text{Primero} \rightarrow \text{dato} \leq \text{Nuevo} \rightarrow \text{dato} \leq \text{Segundo} \rightarrow \text{dato} \leq \text{Tercero} \rightarrow \text{dato}$ .



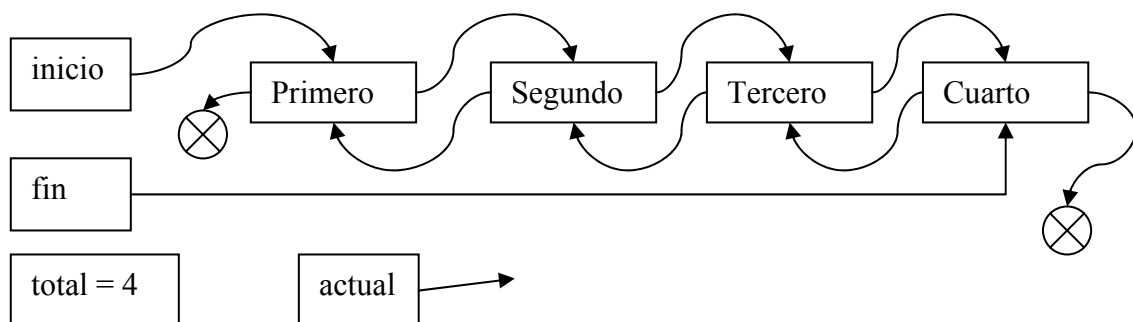
En los casos en que se inserta el elemento inicial de la lista (*total* pasa a ser 1) o bien en la primera o última posición, se requiere además actualizar los punteros *inicio* y/o *fin*.

### 2.2.3. Eliminación

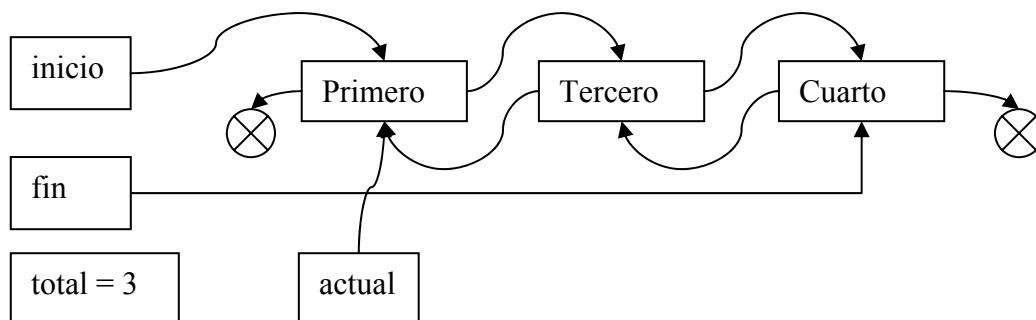
Antes de la eliminación del nodo *Segundo*.

$\text{Primero} \rightarrow \text{dato} \leq \text{Segundo} \rightarrow \text{dato} \leq \text{Tercero} \rightarrow \text{dato} \leq \text{Cuarto} \rightarrow \text{dato}$ .

El *Nodo* al que apunta *actual* antes de la inserción no es relevante.



Después de la eliminación del *Nodo Segundo*.  
 $\text{Primero} \rightarrow \text{dato} \leq \text{Tercero} \rightarrow \text{dato} \leq \text{Cuarto} \rightarrow \text{dato}$ .



Los casos en que se elimina el elemento final de la lista (*total* pasa a ser 0) o bien de la primera o última posición de la lista, requieren además actualizar los punteros *inicio* y/o *fin*.

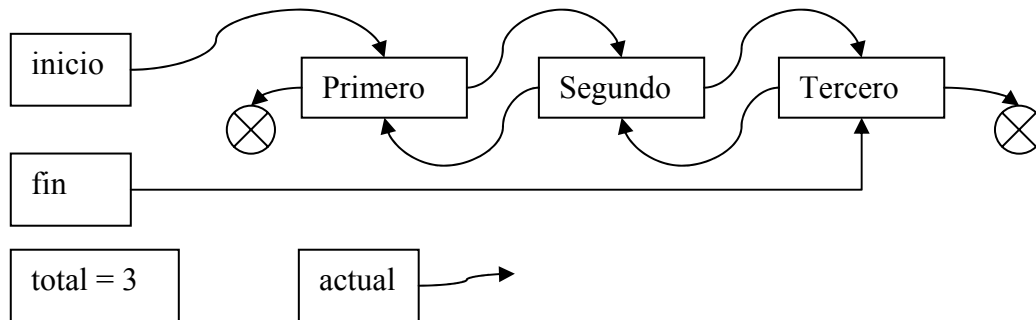
### 2.2.4. Buscar dato

Antes de la búsqueda de un *dato*.

$\text{Primero} \rightarrow \text{dato} \leq \text{Segundo} \rightarrow \text{dato} \leq \text{Tercero} \rightarrow \text{dato}$

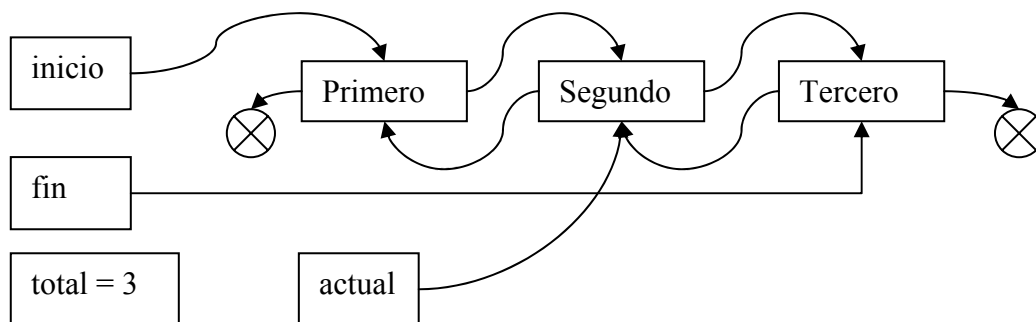
$dato = Segundo \rightarrow dato$

El *Nodo* al que apunta *actual* antes de la inserción no es relevante.



Después de la búsqueda de *dato*.

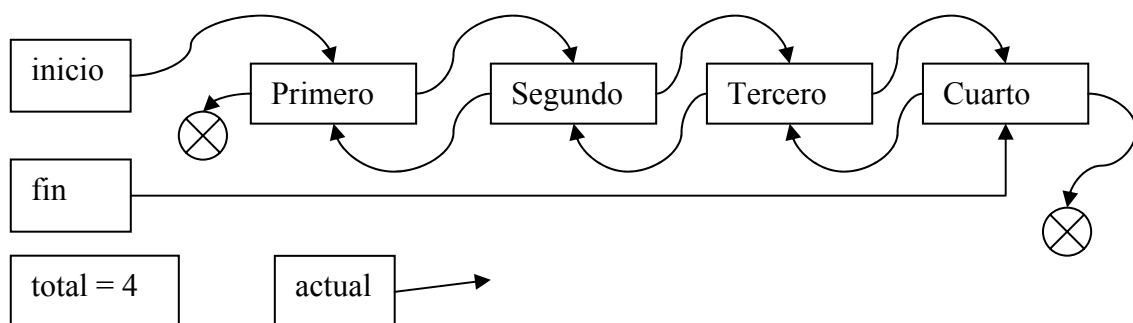
$Primero \rightarrow dato \leq Segundo \rightarrow dato \leq Tercero \rightarrow dato$ .



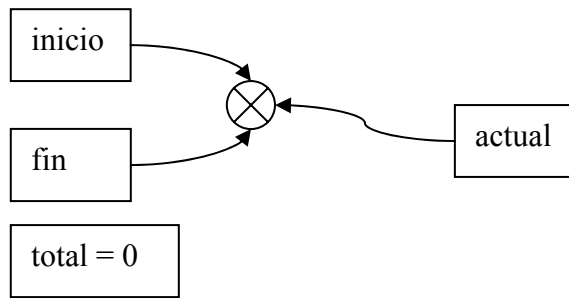
### 2.2.5. Destrucción

Antes de la destrucción de la lista.

El *Nodo* al que apunta *actual* antes de la destrucción no es relevante.



Después de destruir la lista.



Destruir la lista requiere recorrer uno por uno sus *Nodos* liberando el espacio asociado a ellos (*delete Nodo\**).

### 2.3. Almacenamiento permanente en ficheros

Los *Delincuentes* almacenados en la *ListaDelincuentes* serán salvados entre ejecuciones en ficheros de texto. Para esta funcionalidad utilizaremos los métodos *salvaTXT* y *cargaTXT* de las clases *ListaDelincuentes*, *Delincuente*, *Cara* y sus descendientes. Todos los métodos devuelven un booleano indicando si la operación se ha realizado con éxito.

- *Cara*

**bool** cargaTXT(ifstream& entrada);

Carga una cara con los datos contenidos en el *ifstream entrada*. Fíjate que la inicialización es diferente según la clase concreta de *Cara* considerada (*CaraTexto* o *CaraGrafica*). Fíjate que el valor del atributo *fichero* de una *CaraGrafica* puede contener espacios.

**bool** salvaTXT(ofstream& salida) **const**;

Salva el contenido del objeto *Cara* en el *ofstream salida*. En el caso de la clase *CaraTexto* se salvan los atributos *pelo*, *ojos*, *narizYOrejas* y *boca*. En el de *CaraGrafica* se salvan los mismos atributos que para un objeto de la clase *CaraTexto* y además el valor del atributo *fichero*. El primer dato que grabará una cara será el nombre de su clase (*CaraTexto* o *CaraGrafica*) para poder distinguirlas en la carga de datos.

- *Delincuente*

**bool** cargaTXT(ifstream& entrada);

Carga un *Delincuente* con los datos contenidos en el *ifstream entrada*. El *Delincuente* usa la indicación de la clase de su *cara* (ver las operaciones sobre ficheros con caras) para crear un objeto *CaraTexto* o *CaraGrafica* para su atributo *cara*. Fíjate que el valor del atributo *delitos* puede contener espacios.

**bool** salvaTXT(ofstream& salida) **const**;



Salva el contenido del objeto *Delincuente* en el *ofstream salida*.

- *ListaDelincuentes*

**bool** cargaTxt(string entrada);

Carga una *ListaDelincuentes* con los datos contenidos en el fichero de nombre *entrada*. Este método es el encargado de abrir y cerrar el *ifstream* que corresponde al nombre *entrada*.

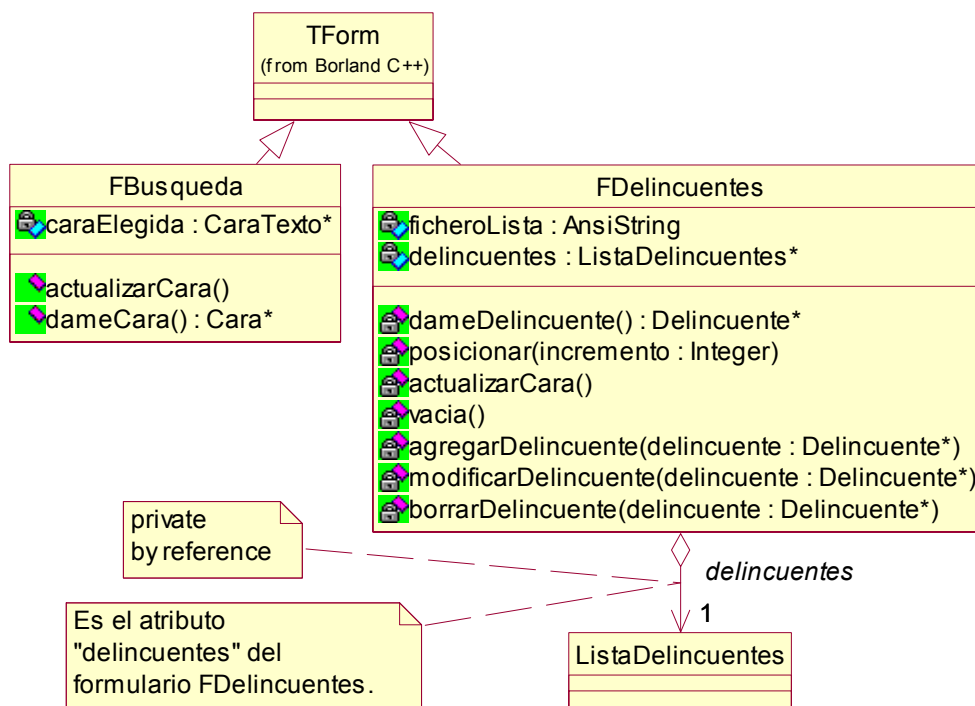
**bool** salvaTxt(string salida) **const**;

Salva los datos de una *ListaDelincuentes* en el fichero de nombre *salida*. Este método es el encargado de abrir y cerrar el *ofstream* que corresponde al nombre *salida*.

A continuación se muestra a modo de ejemplo un fichero de datos.

1	← Nombre del <i>Delincuente</i> 1
1 1	← Delitos del <i>Delincuente</i> 1
CaraTexto	← Clase de la cara del <i>Delincuente</i> 1
0 0 0 0	← Rasgos de la cara ( <i>pelo, ojos, nariz</i> Y <i>Orejas y boca</i> ) del <i>Delincuente</i> 1
2	← Nombre del <i>Delincuente</i> 2
22	← Delitos del <i>Delincuente</i> 2
CaraGrafica	← Clase de la cara del <i>Delincuente</i> 2
1 1 1 1	← Rasgos de la cara ( <i>pelo, ojos, nariz</i> Y <i>Orejas, boca y</i>
D:\WNT\Grano café.bmp	<i>fichero</i> ) del <i>Delincuente</i> 2

## 2.4. Rediseño de los formularios



**Ilustración 3.** Diagrama de clases UML para los formularios.

El nuevo formulario incluirá un menú de la aplicación para acceder a las funcionalidades de la misma acerca de la *ListaDelincuentes* y la operación con los ficheros. Además se incluyen algunos cambios en el aspecto de los formularios.

### 2.4.1. Menú

Añade un menú al formulario principal (componente *TMainMenu* de la paleta de componentes *Standard*). Éste será el menú principal de nuestra aplicación. A continuación se describen sus opciones.

La opción *Archivo* se despliegue en: *Nuevo*, barra de separación (*Caption* = "-"), *AbrirTxt*, *GuardarTxt*, *GuardarComoTxt*, barra de separación y *Salir*. Define los eventos asociados a las opciones *AbrirTxt* y *GuardarComoTxt* utilizando las cajas de diálogo *TOpenDialog* y *TSaveDialog* de la paleta *Dialogs*. Estas componentes tienen un método *Execute()* que muestra la caja de diálogo y devuelve un *boolean* indicando la forma en que se cerró (*CajaDlg->Execute()*), y una propiedad *Filename* de tipo *AnsiString* con el nombre completo del archivo seleccionado (*CajaDlg->FileName*). Declara otra variable privada en el formulario *FDelincuentes* donde guardar el nombre del archivo en curso. La opción *GuardarTxt* debe estar desactivada (propiedad *Enabled* a *false*) siempre que no haya un nombre de archivo en curso. La opción *Nuevo* deja al archivo en curso sin nombre.

Añade al menú una opción *Lista* que se despliegue en: *Nueva*, barra de separación (*Caption* = "-"), *Añadir*, *Modificar*, *Eliminar* y *Buscar*. *Nuevo* vacía la lista de delincuentes actual (atributo *almacen* de *FDelincuentes*). Las demás opciones tienen el mismo efecto que los botones del formulario, por lo que han de reutilizar las llamadas ya usadas por estos.

### 2.4.2. Modificación de los formularios

Observa que tanto en el formulario de búsqueda (ver Ilustración 4) como en el principal (ver Ilustración 5) no hay botones para actualizar el *Memo* donde se representa la cara. Al hacer *click* en los *RadioGroup* la representación se actualizará automáticamente.

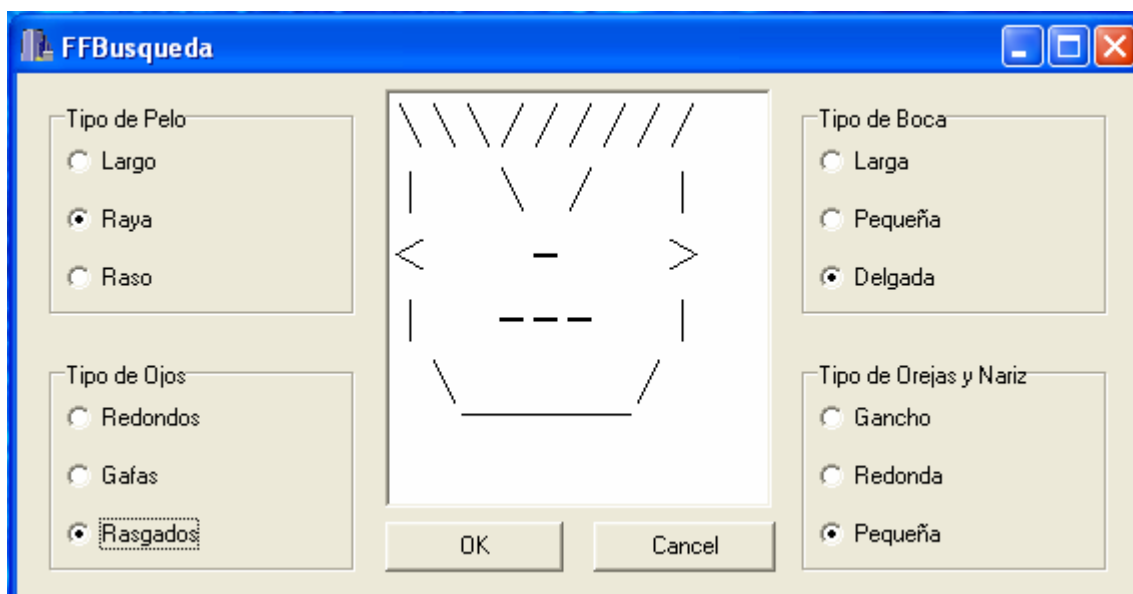


Ilustración 4. Formulario de búsqueda.

**Ilustración 5. Formulario principal.**

En el formulario principal, encima del *Memo* donde se representan los rasgos de la cara se ha colocado un componente *TImage* de la paleta *Additional*. En el componente *TImage* visualizaremos el atributo *imagen* de los objetos *CaraGráfica*. Recuerda que un *Delincuente* puede tener una *CaraTexto* o una *CaraGráfica*. Un objeto *TImage* tiene propiedades *Width* y *Height* que dan su ancho y alto respectivamente. También incluye una propiedad *canvas* de tipo *TCanvas\**. Para dibujar sobre el *canvas* usaremos los métodos:

- `canvas->Draw(a,b,bmp);` ← (a,b) esquina superior izquierda
- `canvas->TextOut(a,b,AnsiString);` ← (a,b) esquina superior izquierda

Define el método asociado al evento *OnClick* del *TImage*. En ese método usa una caja de diálogo *TOpenPictureDialog* de la paleta *Dialogs* para seleccionar la imagen de la cara del *Delincuente*. Este componente tiene un método *Execute()* que muestra la caja de diálogo y devuelve un *bool* que indica cómo se cerró (*CajaDlg->Execute()*), y una propiedad *Filename* de tipo *AnsiString* con el nombre completo del archivo seleccionado (*CajaDlg->FileName*).

Los botones del formulario tienen las siguientes funciones:

- *Nuevo*. Vacía la interfaz dejando un delincuente vacío. Los *RadioGroup* no tienen ninguna opción seleccionada y los campos que corresponden al nombre, los delitos, la imagen y el *Memo* están vacíos.
- *Añadir*. Añade el *Delincuente* en la interfaz si no hay ya otro *Delincuente* con ese nombre en la lista. Si lo hay lanza un aviso y no hace nada.
- *Modificar*. Modifica el *Delincuente* en la interfaz si ya hay un *Delincuente* con ese nombre en la lista. Si lo hay lanza un aviso y no hace nada.
- *Eliminar*. Elimina el *Delincuente* en la interfaz de la lista. Si no hay un *Delincuente* con el mismo nombre en la lista lanza un aviso y no hace nada.
- *Refrescar*. Recarga el *Delincuente* actual de la lista usando el método de *FDelincuentes* *posicionar(0)*.
- *<*. Muestra el anterior *Delincuente* de la lista. Si no hay ninguno advierte al usuario de que no es posible realizar la acción.
- *>*. Muestra el siguiente *Delincuente* de la lista. Si no hay ninguno advierte al usuario de que no es posible realizar la acción.