



VETERINARIA SIMPLE

Diseño e implementación

Descripción breve

En este proyecto llevarás adelante varios aspectos del diseño de software necesario para crear e implementar un programa de gestión muy básica de una veterinaria. El objetivo no es implementar algoritmos complejos de programación, sino comprender los conceptos esenciales sobre Clases, Herencia y Polimorfismo a través de un diseño de software que los aplique de forma práctica e ilustrativa y que a su vez funcione.

Vladimir Rodriguez
vladimir@kaedusoft.edu.uy

Proyecto Veterinaria Simple

Contenido

| | |
|--|----|
| 1. Requerimientos del programa..... | 2 |
| 1.1 Objetivo del proyecto..... | 2 |
| 1.2 Descripción del programa | 2 |
| 1.3 La interfaz gráfica | 3 |
| 2. Diseño del programa | 6 |
| 2.1 UArchivos | 7 |
| 2.1.1 Clases en UML | 8 |
| 2.1.2 Clase TArchivable | 9 |
| 2.1.3 Clase TListaArchivable | 11 |
| 2.1.4 Clase TNode..... | 12 |
| 2.1.5 Inicialización necesaria | 13 |
| 2.1.6 Diagrama final de UArchivos | 13 |
| 2.2 UAnimales | 14 |
| 2.2.1 Clase TEspecie | 14 |
| 2.2.2 Clase TAnimal | 15 |
| 2.2.3 Clase TMascota..... | 16 |
| 2.2.4 Diagrama final de UAnimales | 18 |
| 2.3 UPersonas y TPersona | 18 |
| 2.4 UControlador..... | 19 |
| 2.4.1 Interfaz IControlador | 19 |
| 2.4.2 Clase TControlador | 21 |
| 2.4.3 TStrings y TStringList..... | 21 |
| 2.4.4 Diagrama final de UControlador | 22 |
| 2.5 Diagrama final | 22 |
| 3. Entrega final | 24 |

1. Requerimientos del programa

En la cantidad de proyectos que he planteado a lo largo del curso he descrito siempre los requerimientos de cada programa, es decir, QUÉ DEBE HACER el software que se plantea implementar. El CÓMO estará a cargo del desarrollo del mismo, y esto siempre implica, como mínimo, lo siguiente:

1. El diseño del sistema
2. La elección de la tecnología (o tecnologías) a ser usada para implementarlo.
3. La implementación misma.

En este curso veremos aspectos básicos del diseño de software, cosa que ya he presentado en otras ocasiones, pero que de ahora en más se hará más notoria. Comenzaremos ahora a ver los requerimientos del programa que desarrollarás, cosa que ya habrás visto en el video de la clase, y luego pasaremos a ver el diseño del mismo, el cual deberás leer para poder pasar del esquema a un código Pascal funcional que cumpla con todo lo planteado.

1.1 Objetivo del proyecto

Lo principal en este proyecto es que entiendas y apliques todos los conceptos dados en esta unidad del curso, es decir, que puedas aplicar de forma conceptual y práctica la Herencia y el Polimorfismo a través de los componentes más esenciales de éstos: clases, clases abstractas, interfaces y unidades de Pascal.

De este modo, te daré en este documento todo el diseño del software con varios detalles extra, una interfaz gráfica ya programada, para que tú definas las clases que serán usadas en el sistema y las implementes a fin de lograr lo que se propone. Este proyecto requerirá que vuelvas a ver las clases de esta unidad, y quizá clases anteriores a ella, para repasar conceptos y poder aplicarlos en el desarrollo de tu sistema.

1.2 Descripción del programa

Haremos un pequeñísimo y super básico software de gestión de una veterinaria, pero no te dejes engañar, por más básico que resulte en su operabilidad, será bastante exigente en su implementación, inclusive al tener ya programada la interfaz gráfica de usuario.

En concreto, el programa permitirá registrar **especies** animales, como puede ser *Perro*, *Gato*, *Ave*, *Tortuga*, etc., y luego **razas** de estas especies, como por ejemplo *Siamés* (para un gato), *Pastor alemán* (para un perro), etc. Cada **raza**, por tanto, está asociada a una **especie** concreta.

Para cada raza luego se podrán registrar **mascotas**, como puede ser un gato llamado Firulaís de raza *Siamés*. Este registro debe solicitar los siguientes datos:

- ✓ **ID**: un identificador único para cada animal registrado, el cual se ingresa manualmente, no es generado automáticamente.
- ✓ **Nombre**: el nombre o mote de la mascota.
- ✓ **Edad**: la edad del animal.
- ✓ **Género**: simplemente macho o hembra.

- ✓ **Gestación:** si se trata de un animal **ovíparo** (pone huevos) o **vivíparo** (no pone huevos). Esto podría ligarse automáticamente a la especie o raza, pero se ingresará manualmente para facilitar la programación del software y enfatizar los aspectos más importantes de este proyecto: herencia y polimorfismo.
- ✓ **Especie:** justamente la especie de la mascota a registrar (es un gato, es un perro, etc.).
- ✓ **Raza:** una raza de las vinculadas a dicha especie.

Por cada mascota registrada en el sistema debe poder verse una **ficha** con los datos básicos, la cual se imprimirá de la siguiente manera:

| |
|------------------|
| ID: 568 |
| Especie: Ave |
| Raza: Loro común |
| Nombre: Pepe |
| Edad: 4 |
| Genero: Macho |
| Tipo: Ovíparo |
| Adoptado: NO |

Como se puede observar, aparece un campo en la ficha llamado **Adoptado**, que indica si la mascota ha sido adoptada por algún dueño o no. Esto se describirá superficialmente ya que no lo programaremos en este proyecto dado que complejiza la implementación del software sin aportar nada al objetivo de este proyecto. Por tanto, la función de “dar en adopción” una mascota no estará disponible.

Asimismo, se había pensado en implementar una opción de “dar de baja” una mascota, pero como tampoco aporta nada al objetivo del proyecto, quedará desechada. En futuros proyectos asumiremos sistemas de gran porte en donde tendrás que aplicar todo lo visto en el curso (y cuando digo todo es TODO).

El sistema permitirá también registrar **personas** con los siguientes datos:

- ✓ **DNI:** Un identificador único para la persona, ingresado manualmente.
- ✓ **Nombre:** El nombre de la persona.
- ✓ **Género:** Simplemente masculino o femenino.
- ✓ **Edad:** La edad.
- ✓ **Dirección:** La dirección del domicilio de la persona.

Para cada persona se llevaría el registro de mascotas en adopción y se daría la posibilidad de asignar nuevas mascotas. Como ya expresé, esto no lo haremos ahora por una cuestión de que te hará trabajar mucho sin aportar nada nuevo a lo que ya sabes. La idea de este proyecto es que puedas aprender *herencia y polimorfismo*, con lo cual ya tendrás bastante trabajo.

1.3 La interfaz gráfica

Como ya ha ocurrido en otros proyectos, recibirás la GUI ya programada y funcional, teniendo tú que hacer todo lo que en programación Web se conoce como **backend**, es decir, el funcionamiento interno de la aplicación. El **frontend** comenzará siendo la siguiente ventana:

Veterinaria

Archivo Mascotas Personas

Mascotas

Especies: Ave, Gato, Perro.
 Botón: Crear especie...

Razas: (Empty list)

Mascotas: (Empty table with columns ID, Nombre mascota)

Ficha: (Empty text area)

Registrar...
Dar en adopción...
Dar de baja

Personas

| Nombre | Edad | Genero | DNI | Dirección |
|------------------|------|-----------|----------|------------------|
| José | 23 | Masculino | 456 | Sarasa |
| Lorenzo Pepe | 20 | Masculino | 456789 | La Comarca |
| Lorena Cuello | 31 | Femenino | 46518746 | Calle 132 |
| Vladimir Rodrigu | 31 | Masculino | 43564708 | José Escobar 115 |

Mascotas: (Empty table with columns ID, Nombre)

Registrar...
Dar de baja
Salir

Como puedes observar está dividida en dos secciones horizontales: **Mascotas** y **Personas**. En la sección de mascotas tienes una lista de Especies, debajo de la cual está el botón **Crear especie...**, luego a su derecha está la lista de razas, la lista de mascotas y el visor de la ficha de cada mascota.

Al hacer clic sobre una especie concreta, la lista de razas se actualizará para mostrar justamente las razas de dicha especie:

Veterinaria

Archivo Mascotas Personas

Mascotas

Especies: Ave, **Gato**, Perro.
 Botón: Crear especie...

Razas: Gato común, Siames

Mascotas: (Empty table with columns ID, Nombre mascota)

Ficha: (Empty text area)

Registrar...
Dar en adopción...
Dar de baja

Personas

| Nombre | Edad | Genero | DNI | Dirección |
|------------------|------|-----------|----------|------------------|
| José | 23 | Masculino | 456 | Sarasa |
| Lorenzo Pepe | 20 | Masculino | 456789 | La Comarca |
| Lorena Cuello | 31 | Femenino | 46518746 | Calle 132 |
| Vladimir Rodrigu | 31 | Masculino | 43564708 | José Escobar 115 |

Mascotas: (Empty table with columns ID, Nombre)

Registrar...
Dar de baja
Salir

En este ejemplo se ha hecho clic sobre la especie **Gato**, y aparecieron dos razas: **Gato común** y **Siames**. Si se hace clic sobre una raza concreta, aparecerá entonces la lista de mascotas registradas con dicha raza:

Mascotas

Especies: Ave, **Gato**, Perro.
 Botón: Crear especie...

Razas: **Gato común**, Siames

Mascotas:

| ID | Nombre mascota |
|------|----------------|
| 0001 | Firulais |
| 0002 | Garfield |
| 0003 | Tom |

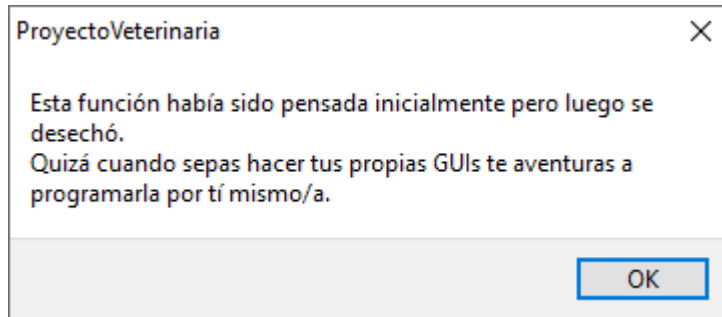
Ficha:

ID: 0001
Especie: Gato
Raza: Gato común
Nombre: Firulais
Edad: 10
Genero: Macho
Tipo: Ovíparo
Adoptado: NO

Registrar...
Dar en adopción...
Dar de baja

En este caso se seleccionó **Gato común** como raza, apareciendo tres mascotas: *Firulais*, *Garfield* y *Tom*. Al hacer clic sobre una de ellas se puede ver su ficha, resaltando el hecho de que quién registró a Firulais indicó que pone huevos.

Como verás, a la derecha del cuadro de texto donde se ve la ficha, hay tres botones: **Registrar...**, **Dar en adopción...** y **Dar de baja**. Solo funcionará **Registrar...**, ya que si haces clic en los otros dos solo verás el mensaje siguiente:



Al hacer clic en **Crear especie...** verás el siguiente cuadro de diálogo:

Allí tú podrás registrar nuevas especies animales y vincular razas nuevas, simplemente completando los datos y haciendo clic en **Registrar ahora**. No hay mucho más que hacer allí.

Si haces clic en **Registrar...**, aparecerá un cuadro pidiendo los datos de una nueva mascota, tal como ves aquí a tu izquierda. Allí solo tendrás que rellenar los campos y elegir las opciones que cada lista desplegable te muestra para luego hacer clic en **Registrar**. Si no hay errores habrás guardado una nueva mascota en el sistema.

Finalmente, en la sección de Personas, tendrás una tabla con los datos de todas las personas registradas en el sistema, pudiendo registrar nuevas:

| Personas | | | | | Mascotas | | |
|------------------|------|-----------|----------|------------------|----------|--------|--------------|
| Nombre | Edad | Genero | DNI | Dirección | ID | Nombre | |
| José | 23 | Masculino | 456 | Sarasa | | | Registrar... |
| Lorenzo Pepe | 20 | Masculino | 456789 | La Comarca | | | Dar de baja |
| Lorena Cuello | 31 | Femenino | 46518746 | Calle 132 | | | |
| Vladimir Rodrigu | 31 | Masculino | 43564708 | José Escobar 115 | | | Salir |

Si haces clic en alguna se mostrarían las mascotas adoptadas, pero como esta función no está disponible, pues no se verá nada. La función de **Dar de baja** tampoco estará disponible, por tanto solo te servirá hacer clic en **Registrar...** para ver el cuadro siguiente:

Registrar persona ✕

DNI:

Nombre:

Genero:

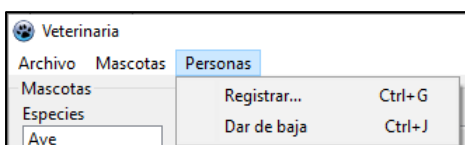
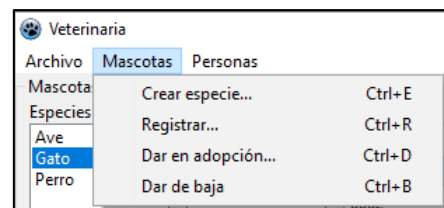
Edad:

Dirección:

Aquí solo completas los datos y haces clic en **Registrar** para que la nueva persona aparezca en la tabla y sea archivada por el sistema.

Finalmente tienes los menús **Archivo**, **Mascotas** y **Personas**. En el menú **Archivo** solo verás la opción de **Salir** del sistema, pero en el menú **Mascotas** tendrás las

mismas opciones que te brindan los botones de dicha sección en la ventana, es decir, **Crear especie**, **Registrar**, **Dar en adopción** y **Dar de baja**, tal como ves aquí a tu derecha. Además podrás ver los accesos rápidos de cada opción para utilizar el teclado si lo deseas.



De igual modo el menú **Personas** te permitirá **Registrar** nuevas personas y **Dar de baja** (si dicha función estuviera disponible).

Finalmente tendrás el botón **Salir** en la esquina inferior derecha de la ventana que, obviamente, cierra la aplicación. Esto también puedes hacerlo con la cruz, con el menú **Archivo** → **Salir**, o bien presionando **CTRL+S**.

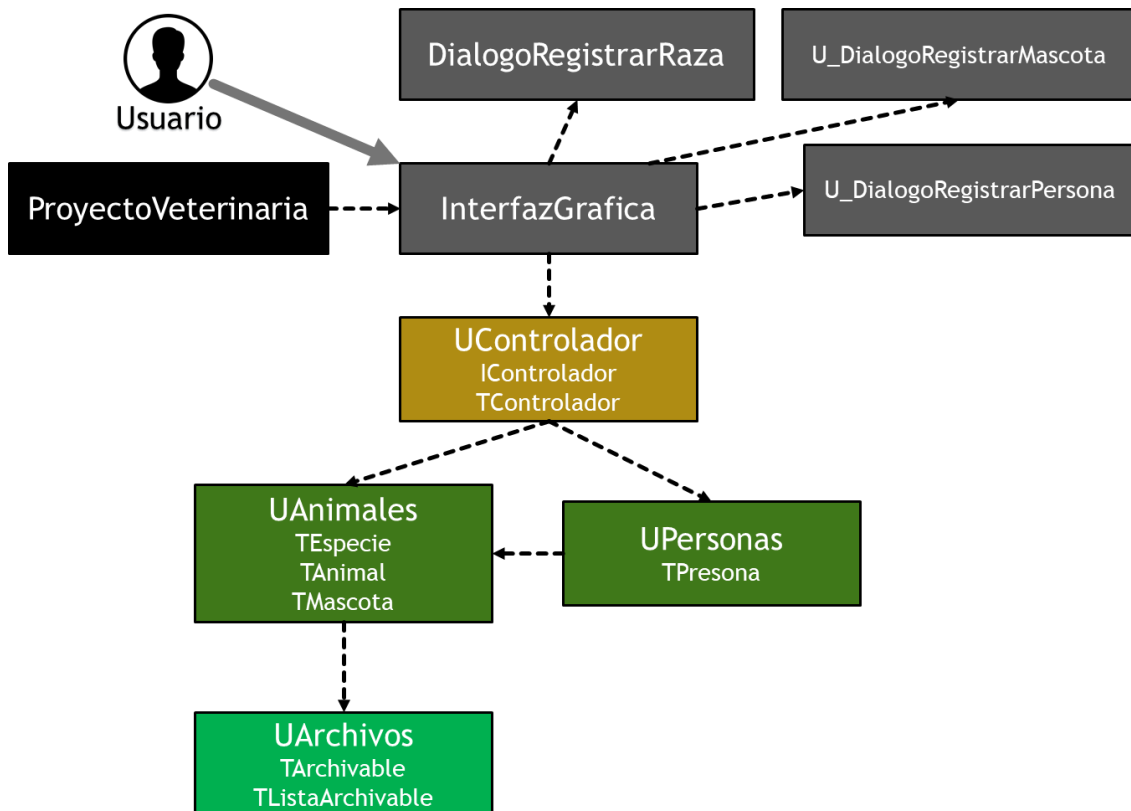
Cada registro hecho en el sistema para **Especies**, **Razas**, **Mascotas** y **Personas**, es guardado instantáneamente en los archivos del sistema, por tanto, la información estará disponible cuando vuelva a abrirse la aplicación. Esto lo veremos en detalle más adelante.

2. Diseño del programa

Como ya mencioné anteriormente, en esta oportunidad te daré el diseño del programa para que tú hagas la implementación siguiendo dicho diseño. El diagrama que te daré estará dado por el **Lenguaje Unificado de Modelado**, conocido como UML por sus siglas en inglés (*Unified Modeling Language*), el cual te iré explicando en este documento, así como en la clase designada a este proyecto. Es, por tanto, imprescindible que prestes suma atención a todo.

También recibirás, como es costumbre hasta ahora, el código fuente de la interfaz gráfica del programa, ya que aún no hemos visto cómo crear y utilizar componentes gráficos. Una vez aprendas esto, los proyectos que te pediré implicarán, por supuesto, que construyas las ventanas, cuadros, menús y demás, desde cero. Pero todo a su debido tiempo.

A continuación te daré un diagrama general de las dependencias del sistema, indicando en cada recuadro la unidad de Pascal a la que se hace referencia y, con letra más pequeña, las clases o interfaces que contienen. Este esquema es sumamente sencillo, con el objeto de que te hagas una idea clara de cómo está todo estructurado. Luego entraremos en detalles minuciosos:



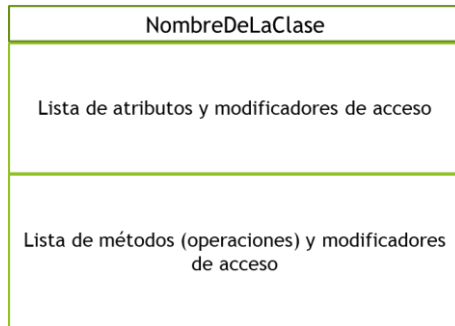
Puedes ver que el cuadro negro llamado **ProyectoVeterinaria** es el que contiene al programa principal, es decir, el archivo de pascal que inicia con la palabra PROGRAM. Este bloque es el encargado de iniciar la interfaz gráfica, que es con la que interactúa el usuario directamente, y dicha interfaz es la que interactúa con el resto del sistema. Por tanto, las unidades diagramadas en gris: InterfazGrafica, U_DialogoRegistrarMascota, U_DialogoRegistrarPersona y DialogoRegistrarRaza ya estarán programadas, no tendrás que hacer nada con ellas. Sin embargo, su existencia en el proyecto condiciona al resto de los componentes ya que tendrás que ceñirte al diseño para que todo encaje y funcione.

2.1 UArchivos

Esta unidad contendrá lo necesario para manipular archivos en el sistema y poder así persistir los datos, modificarlos y recuperarlos cuando sea necesario. Deberás crear el archivo **UArchivos.pas** para definir dicha unidad, y dentro de ella declararás e implementarás lo que se describirá a continuación.

2.1.1 Clases en UML

Como ya indiqué anteriormente **UML** significa *Unified Modeling Language*, que se traduce como *Lenguaje Unificado de Modelado*, y es justamente una herramienta mundialmente reconocida para modelar sistemas informáticos a través de diagramas. UML se utiliza tanto para modelar los componentes de un sistema como el comportamiento y funcionamiento del mismo, sin importar el lenguaje de programación que se vaya a usar para el mismo.



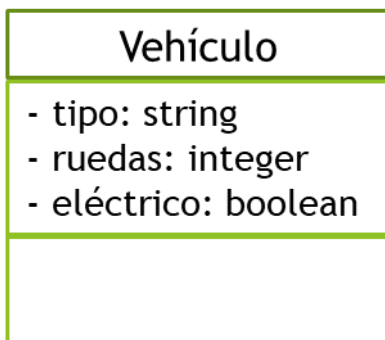
Para diagramar una clase en este lenguaje simplemente se ha de dibujar un rectángulo que se divide en tres partes:

1. Arriba del todo se coloca el nombre de la clase.
2. Luego se listan los atributos.
3. Finalmente se listan las operaciones.
4. Se puede agregar una cuarta sección.

Para listar los atributos se deben respetar las siguientes reglas básicas:

- Primero se coloca el modificador de acceso del atributo. Si es público se coloca un signo de más (+), si es privado un signo de menos (-) y si es protegido se utiliza la almohadilla o numeral (#). En nuestro caso particular, como Pascal posee el atributo **strict private** usaremos doble signo de menos (--) para ilustrarlo.
- Luego sigue el nombre del atributo y dos puntos.
- Finalmente se indica el tipo del atributo, tal como hacemos en Pascal.

Supongamos que nuestra clase se llama **Vehículo** y que la hemos diagramado así:

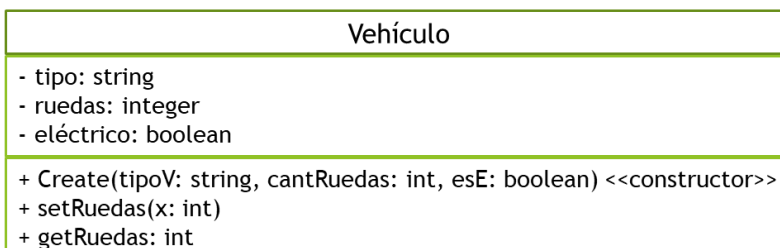


En este esquema la clase tiene tres atributos privados, **tipo** que es de tipo `string`, **ruedas** que es de tipo `integer` (en UML este tipo se suele indicar como `int` en vez de `integer`) y **eléctrico** que es de tipo `boolean`.

Sabemos que son privados porque a todos se les antepone el signo de menos (-). No se utiliza punto y coma para separar los atributos, y se coloca estrictamente uno debajo del otro. Ten presente que la idea de UML es siempre crear diagramas

que permitan esquematizar un proyecto. Tu trabajo es traducir ese esquema al lenguaje que se vaya a utilizar. Si todo está bien hecho, un proyecto UML debe poder ser implementado en cualquier lenguaje de programación que pueda soportar los requerimientos.

Veamos ahora cómo añadimos las operaciones de esta clase:



Fíjate que se han incluido 3 operaciones solamente para que comprendas cómo funciona UML. La primera es `Create`, el constructor. Al final se

indica con `<<constructor>>` que esta operación es, valga la redundancia, el constructor de la clase. Esto no es obligatorio, pero recomiendo hacerlo para que el diagrama quede mucho más claro.

Ahora fíjate en cómo se indican los parámetros: en primer lugar se indica el nombre del parámetro, dos puntos y luego el tipo de datos. Esto se hace para CADA argumento sin importar si se repite el tipo de datos, separándolos por coma simple. Por ejemplo:

- **Pascal:** `miOperacion(x, y, z: integer; nombre, apellido: string)`
- **UML:** `miOperacion(x: int, y: int, z: int, nombre: string, apellido: string)`

Esto es similar a como se declara una operación en Java o C++. Fíjate además que cambié el tipo `integer` por `int`. Además, si miras el dibujo, te darás cuenta de que no se utilizan las palabras **procedure** y **function** para determinar cuál operación es justamente un procedimiento o una función. Esto es porque en UML, al igual que en muchos lenguajes, todo son funciones, entendiéndose como procedimientos a aquellas funciones que no tienen un tipo de retorno o bien retornan el tipo `void` (que es el tipo nulo o vacío).

Si miras el ejemplo puedes ver que **setRuedas** es un procedimiento porque no hay ningún tipo de retorno indicado. También se podría haber escrito `setRuedas(x: int): void`, para indicar que es un procedimiento, pero esto en UML ha quedado en desuso.

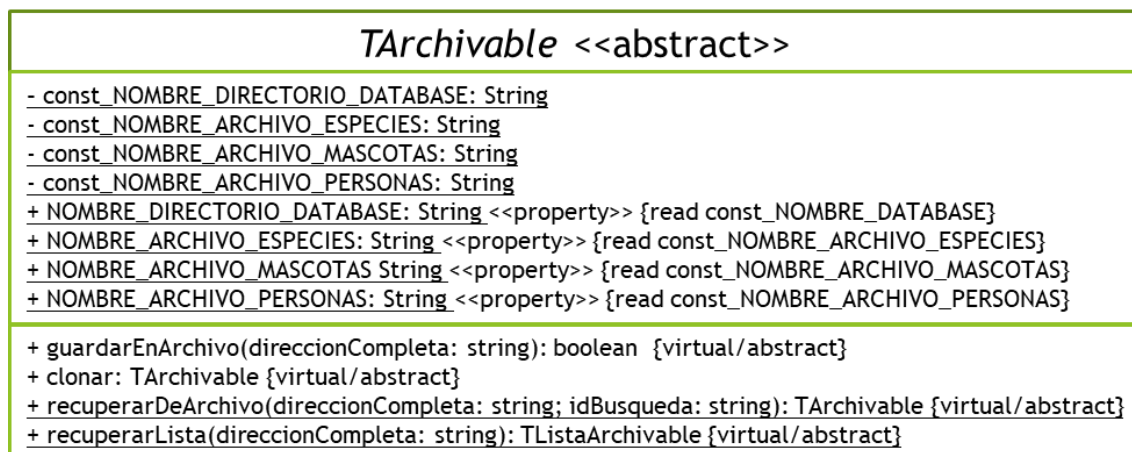
De esta manera tú tienes que ser capaz de mirar el dibujo de la clase **Vehiculo** y “traducirlo” al lenguaje de programación que estés utilizando en ese momento, sea Pascal, Java, PHP, C++, C# o cualquier otro que soporte el uso de clases y Programación Orientada a Objetos.

Algunas cuestiones más a tener en cuenta:

- Si un atributo u operación es estático/a se pone en subrayado.
- Si una operación es virtual se agrega al final el indicador `{virtual}`.
- Si una operación es abstracta se agrega al final el indicador `{abstract}`
- Si una clase es abstracta se pone su nombre en *cursiva* (itálica), o bien se agrega el indicador `<<abstract>>` luego del nombre.

2.1.2 Clase TArchivable

La primera clase que hemos de declarar en esta unidad es **TArchivable**, cuyo diagrama es:



Si prestas atención lo primero que se indica es que dicha clase es abstracta, lo cual quiere decir que no está pensada para crear objetos de ella y que puede contener métodos abstractos. Además, ya puedes ir sabiendo que el objetivo de esta clase es ser heredada por otras clases.

Fíjate ahora en los atributos: tienes un total de 8, 4 privados y 4 públicos. Además están en subrayado, así que son estáticos. Presta atención a los 4 atributos públicos, puedes ver que tienen el indicador `<<property>>` luego de su declaración, para indicarte que son propiedades de Pascal. Esto es algo que yo incluí porque realmente UML no soporta el modificador **property**, pero a los efectos de este proyecto ya puedes ver que dichos atributos son en realidad propiedades públicas. Al final de cada declaración tienes un indicador entre llaves, el cual muestra que estas propiedades son de solo lectura (read) y además te dice cuál atributo lee cada una en particular.





Veamos ahora las operaciones, que son todas públicas y además al final todas tienen el indicador **{virtual/abstract}**, indicándote que justamente son virtuales y abstractas. Además las últimas dos operaciones (`recuperarDeArchivo` y `recuperarLista`) son estáticas porque están subrayadas. Finalmente puedes ver que todas son funciones porque cada una tiene declarado un tipo de retorno. Entendiendo esto, por ejemplo, la declaración de `recuperarLista` en Pascal quedaría así:

```
class function recuperarLista(direccionCompleta: string): TListaArchivable; virtual; abstract;
```

Teniendo en cuenta toda la información que hay en el dibujo, tienes que escribir la clase **TArchivable** en Pascal, declarando todo de forma que cumpla con lo establecido por el diseño. No puedes agregar nada más a la clase, ni atributos ni operaciones. Si requirieras de algo extra para implementarla puedes agregarlo en el bloque **implementation** de la unidad.

Fíjate que esta operación en concreto retorna algo de tipo **TListaArchivable**, el cual no hemos declarado aún. Eso en Pascal te dará un problema que comentaré justo a continuación.

Veamos una descripción detallada de lo que debe hacer cada una de las operaciones de esta clase a fin de que puedas implementarlas:

-  **guardarEnArchivo**: Guarda el objeto actual en un archivo. Cada clase que hereda de **TArchivable** implementa esta operación a su manera. Se retorna **TRUE** si se pudo guardar la información, **FALSE** si no se pudo. El argumento `direccionCompleta` indica la dirección del archivo en el que se guardará la información, incluyendo el nombre y la extensión del mismo.
-  **clonar**: Crea una copia limpia del objeto y la retorna en un nuevo objeto.
-  **recuperarDeArchivo**: Busca en el archivo dado por `direccionCompleta` (incluye nombre y extensión) el registro identificado con `idBusqueda` y retorna un objeto con su información. Cada clase implementa esta operación a su manera. Si no se encuentra el registro se retorna **NIL**.
-  **recuperarLista**: Busca en la dirección dada (incluye nombre y extensión) todos los registros en el archivo, y retorna una lista con todos los objetos creados a partir de ellos. Cada clase implementa esta operación a su manera. Si no hay registros para leer, se retorna una lista vacía.

2.1.3 Clase TListaArchivable

La idea detrás de la clase **TArchivable** es definir un tipo de objetos que pueda ser guardado en un archivo, por tanto, tú puedes crear cualquier clase que quieras, hacer que herede de **TArchivable** para que implemente sus operaciones, y así podrás guardar esos objetos en archivos. Claro, deberías hacer algunos ajustes ya que esta clase en concreto está diseñada para este proyecto en particular y no fue pensada de forma genérica para todo uso, pero aquí aprenderás la idea, ya que en el mundo real usarás este tipo de clases por doquier.

La clase **TListaArchivable** provee de una colección de elementos *archivables* con formato de lista encadenada. Su diseño es el siguiente:

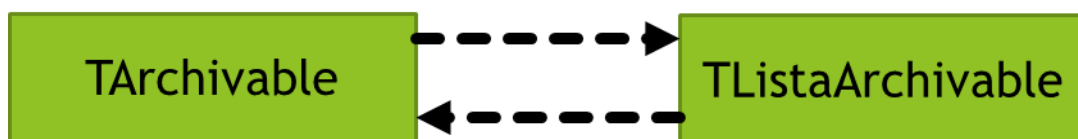
| TListaArchivable |
|--|
| <pre>-- atrNodo: TNode -- atrIterador: TNode + Vacia: boolean <<property>> {read esVacia} + Actual: TArchivable <<property>> {read get} + HaySiguiente: boolean <<property>> {read haySiguienteNodo} + Create() <<constructor>> + Destroy() <<destructor>> + agregar(const a: TArchivable) + existe(const a: TArchivable): boolean + quitar(const a: TArchivable) + esVacia: boolean + get: TArchivable + siguiente + haySiguienteNodo: boolean + reset</pre> |

Lo primero que podemos ver es que hay dos atributos estrictamente privados (**--**) y tres propiedades públicas de solo lectura.

Luego tenemos todas las operaciones públicas de la clase, que si prestas atención te darás cuenta de que implementan el TAD lista encadenada.

Verás que esta clase tiene elementos de tipo **TNode**, y esa

clase no ha sido definida aún, lo cual indicaré a continuación. Pero también es importante que veas que esta clase también tiene elementos de tipo **TArchivable**, que es la clase que definimos anteriormente. Por tanto, si miramos la dependencia de ambas clases tenemos esto:



Ambas clases son codependientes, por tanto para definir a **TArchivable** necesitas tener a **TListaArchivable**, y para definir a ésta necesitas a **TArchivable**, lo cual no es posible. En lenguajes como Java o C# basta con definir ambas clases vacías para solucionar este conflicto, pero en Pascal no, porque cada elemento solo puede contener tipos de elementos definidos previamente. Por tanto, si estás definiendo **TArchivable** necesitarás tener **TListaArchivable** ya definida, y viceversa. Para solucionar esto Pascal admite las declaraciones **forward** o adelantadas. En el caso de las clases debes hacerlo así:

```

TListaArchivable= class;

TArchivable= class
  Aquí defines toda la clase por completo.
end;

TListaArchivable= class
  Aquí defines toda la clase.
end;
  
```

Lo primero que haces entonces es definir la clase **TListaArchivable** vacía, tal como ves en la primera línea del código anterior. Con eso el tipo **TListaArchivable** estará disponible para que lo uses en **TArchivable**. No es necesario utilizar la palabra reservada `forward`. Luego defines la clase **TArchivable** completamente, con atributos, propiedades y operaciones, y luego vuelves a definir **TListaArchivable**, pero ahora en forma completa.

Quedan aún dos cosas por solucionar para lograr esta implementación:

1. Definir la clase interna **TNodo**.
2. Agregar un código de inicialización a la clase **TListaArchivable**.

Pero antes de ver estos dos puntos te dejo el detalle de todas las operaciones:

- ✚ **Create**: Crea una lista vacía.
- ✚ **Destroy**: Destruye la lista con todo su contenido.
- ✚ **Agregar**: Agrega un nodo al final de la lista.
- ✚ **Existe**: Retorna **TRUE** si existe el elemento a, **FALSE** si no.
- ✚ **Quitar**: Quita el elemento de la lista si éste existe. Se resetea el iterador.
- ✚ **esVacía**: Indica si la lista es vacía.
- ✚ **Get**: Retorna el nodo actual apuntado.
- ✚ **Siguiente**: Mueve el apuntador un lugar hacia adelante.
- ✚ **haySiguienteNodo**: Indica si hay más nodos por recorrer.
- ✚ **Reset**: Resetea el apuntador al inicio de la lista.

2.1.4 Clase TNodo

La clase **TListaArchivable** es una lista encadenada, y tal como viste en ejemplos anteriores en el curso, necesitas una clase interna para implementar los nodos de la lista. Esta clase interna tiene que ser privada, y la llamaremos **TNodo**.

| TNodo |
|--|
| <pre>-- atrContenido: TArchivable -- atrNext: Tnodo + Contenido: Tarchivable <<property>> {read getContenido/write setArchivable} + Next: TNodo <<property>> {read getNext/write setNext}</pre> |
| <pre>-- setArchivable(AValue: TArchivable) + Create(cont: Tarchivable) <<constructor>> + destructor Destroy() <<destructor>> + getContenido: Tarchivable + setNext(nod: TNodo) + getNext: Tnodo + setContenido(cont: TArchivable);</pre> |

TNodo define entonces a un nodo de la lista, que tendrá el contenido del mismo, un siguiente nodo (*next*) y luego definimos dos propiedades públicas que sirven para leer dichos datos, lo cual también se podrá hacer con los `getter` y `setter` correspondientes. Fíjate que los dos atributos privados son estrictamente privados, al igual que la operación

`setArchivable`. El resto son operaciones públicas. Allí puedes reconocer cuáles son funciones y cuáles procedimientos. Te dejo a continuación la descripción detallada de cada una:

- ✚ **setArchivable**: Operación privada que asigna contenido al nodo actual.
- ✚ **Create**: Crea un nuevo nodo.

- ✚ **Destroy**: Libera la memoria ocupada por el nodo.
- ✚ **GetContenido**: Retorna el contenido del nodo.
- ✚ **setNext**: Asigna un nuevo siguiente al nodo.
- ✚ **getNext**: Retorna el siguiente nodo.
- ✚ **serContenido**: Asigna un nuevo número al nodo.

2.1.5 Inicialización necesaria

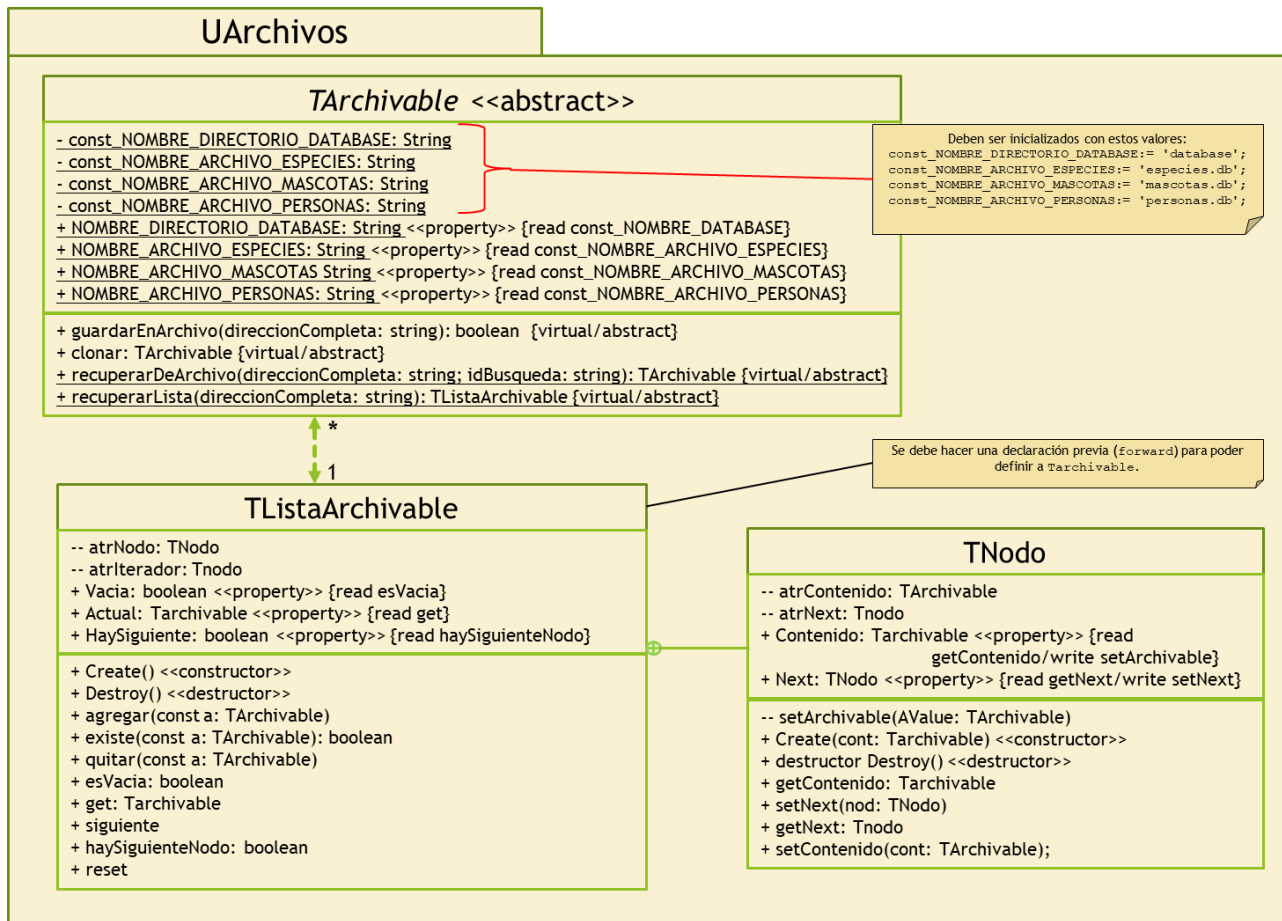
La unidad UArchivos, que es en donde declararás todo lo descrito en este apartado del documento, define a la clase TArchivable, la cual define 4 constantes estáticas que deben contener los siguientes valores al ser inicializadas:

- TArchivable.const_NOMBRE_DIRECTORIO_DATABASE:= 'database'
- TArchivable.const_NOMBRE_ARCHIVO_ESPECIES:= 'especies.db'
- TArchivable.const_NOMBRE_ARCHIVO_MASCOTAS:= 'mascotas.db'
- TArchivable.const_NOMBRE_ARCHIVO_PERSONAS:= 'personas.db'

Estos atributos, que son privados dentro de la clase **TArchivable**, son estáticos, y como Pascal no nos permite definir constantes estáticas en una clase, lo que hice fue definir estas cuatro variables y luego cuatro propiedades de solo lectura, también estáticas, que permitirán obtener estos valores constantes siempre. Así que es importante inicializar dichos atributos privados, y eso lo harás en el bloque **initialization** de esta unidad.

2.1.6 Diagrama final de UArchivos

El diagrama final para las clases de esta unidad es el siguiente:




Este diagrama muestra la relación que hay entre todas las clases que componen a esta unidad, con el nivel de detalle que ya vimos por separado. En UML debes saber que:

- Una flecha punteada de una clase A a una clase B indica dependencia de A a B, o sea, A depende de B y por tanto B es independiente de A.
- Una flecha sólida de una clase A a una clase B indica que A hereda de B, por tanto B es clase base o padre de A, y A es clase derivada de B.

Si te fijas, entre `TArchivable` y `TListaArchivable` tienes una flecha punteada de dos puntas, y en cada una hay un ordinal:

↑ * Esta flecha indica la relación que hay entre ambas clases. El ordinal asterisco (*) indica, al igual que con las expresiones regulares, 0 o más elementos. Así se está diciendo que
↓ 1 1 objeto de `TListaArchivable` (fíjate que el 1 está en el lado de esta clase) tiene 0 o más elementos `TArchivable` (el asterisco está justo en el lado de `TArchivable`).

Luego pues ver entre `TListaArchivable` y `TNodo` la unión que ves aquí: 

Esta unión indica clase interna. La clase que tiene el círculo con el signo de más, en este caso `TListaArchivable`, es la clase padre, la que contiene a la clase interna, y la clase en el otro extremo de la línea recta es la clase interna, en este caso `TNodo`.

Finalmente encontrarás dos anotaciones en este diagrama:

Deben ser inicializados con estos valores:

```
const_NOMBRE_DIRECTORIO_DATABASE:= 'database';
const_NOMBRE_ARCHIVO_ESPECIES:= 'especies.db';
const_NOMBRE_ARCHIVO_MASCOTAS:= 'mascotas.db';
const_NOMBRE_ARCHIVO_PERSONAS:= 'personas.db';
```

Se debe hacer una declaración previa (*forward*) para poder definir a `Tarchivable`.

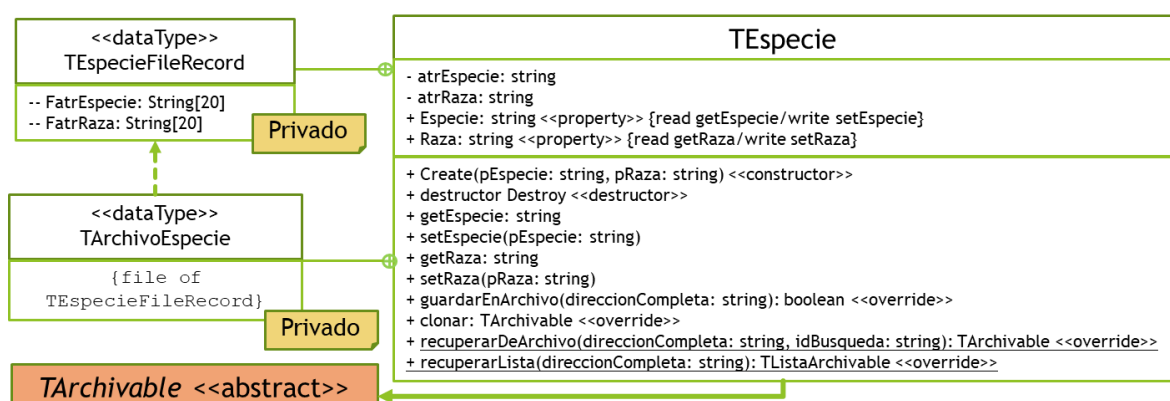
Las anotaciones se usan en UML para escribir aclaraciones que se consideren necesarias. Se ilustran con una hoja de punta doblada.

2.2 UAnimales


Esta unidad contendrá tres clases que harán uso de la unidad **UArchivos** y sus clases, heredándolas y extendiéndolas.

2.2.1 Clase `TEspecie`

Esta clase define lo básico sobre una especie animal, y está definida por el siguiente modelo:



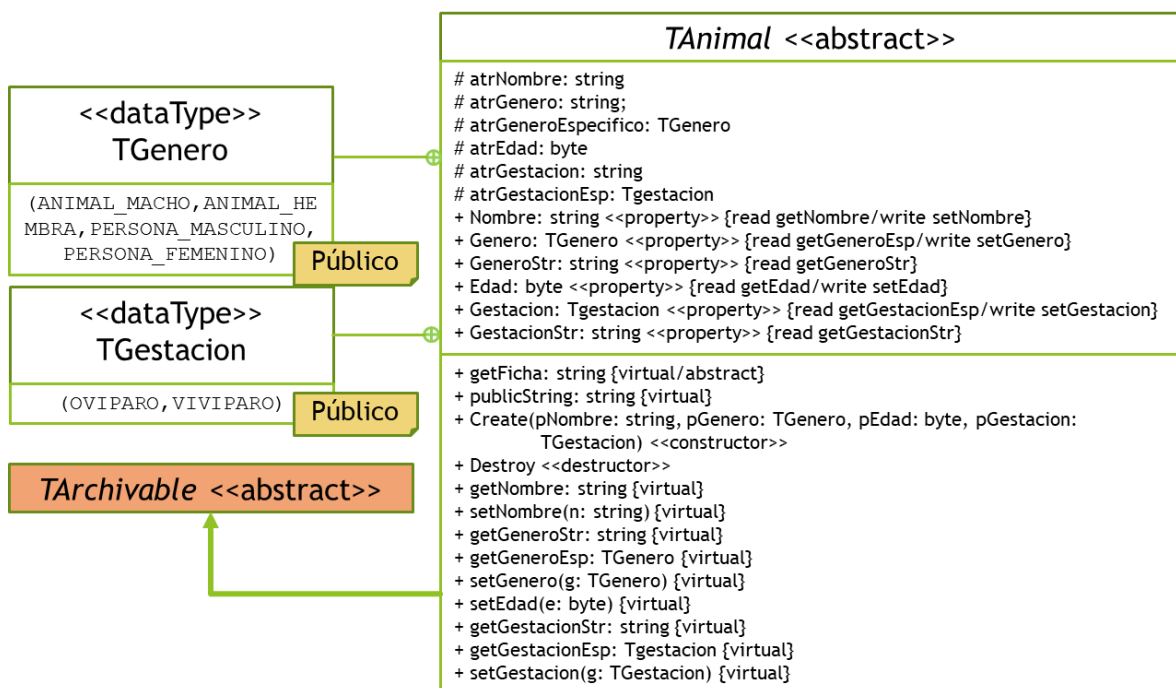
Veamos esto paso a paso. Primero que nada, ves por fuera **TArchivable**, la cual ya definimos en el modelo anterior. Esta clase está unida a **TEspecie** por una flecha sólida; si recuerdas, las flechas punteadas indican dependencia y la flecha sólida indica herencia (la herencia en sí misma es también dependencia ya que la clase hija depende de la clase base). De esta manera, **TEspecie** hereda de **TArchivable** por lo cual depende también de ella.

Antes de entrar en el detalle de **TEspecie**, fíjate que hay dos cuadros unidos a él por el símbolo de clase interna , pero no son clases ya que su nombre tiene el indicador `<<dataType>>`, indicando que se trata de un tipo de datos y no de una clase. Dichos tipos de datos son internos a **TEspecie**, y tal como indican las anotaciones anexadas son privados. En concreto en Pascal los representarás con registros. Además, ambos tipos están unidos por una flecha punteada que sale desde **TEspecieFileRecord** hacia **TArchivoEspecie**, indicando que este último depende del primero. En **TArchivoEspecie** puedes ver que se ha colocado su definición ya que es un tipo de archivos: `file of TEspeceFileRecord`.

Veamos ahora **TEspecie**, que en sí no es una clase complicada. Lo que hace definir dos atributos privados y dos propiedades públicas basadas en dichos atributos. Luego están sus operaciones públicas, en las cuales debes prestar atención a las 4 últimas que son las operaciones heredadas de **TArchivable**, por lo cual se utiliza `override` para indicar polimorfismo con sobreescritura. Si no se implementaran estas operaciones resultaría que **TEspecie** es abstracta porque estas operaciones son abstractas. Recuerda también que las dos últimas operaciones son estáticas.

2.2.2 Clase TAnimal

La clase **TAnimal** define justamente lo que es un animal en este proyecto, pero como no se usará ningún objeto de tipo **TAnimal**, sino que luego habrá objetos **TMascota** y **TPersona** (clases que detallaré luego) esta clase es abstracta. Veamos su diseño:



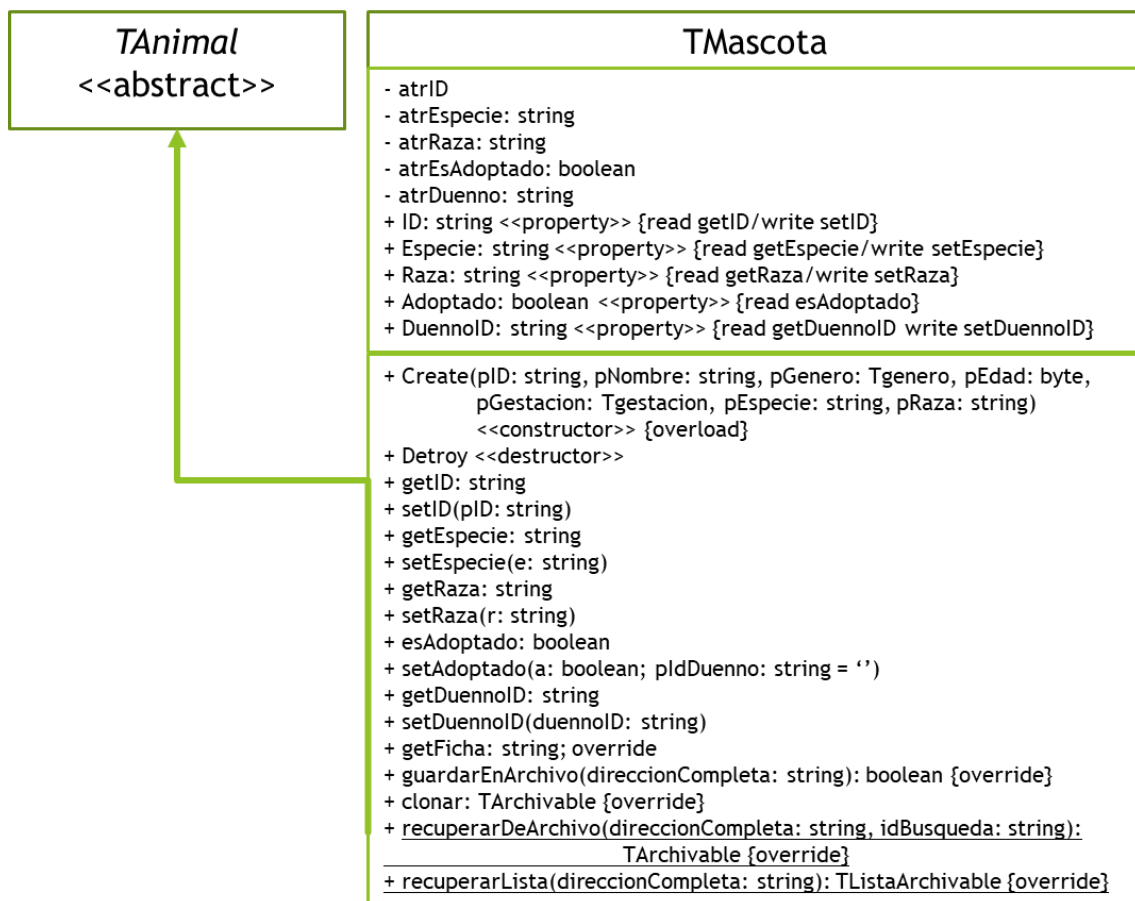
Primero que nada fijate en que hay dos tipos de datos internos a `TAnimal`, `TGenero` y `TGestacion`, ambos enumerados, independientes entre sí. Estos tipos son utilizados para definir atributos y propiedades de esta clase, tal como puedes ver en el modelo.

Esta clase está pensada para ser heredada, por tanto lo primero que ves respecto de esto es que los atributos no son privados, sino protegidos (**protected**), dados por el símbolo `#`. Esto es para que las clases hijas a `TAnimal` puedan acceder a estos atributos. Lo segundo que puedes ver respecto a que esta clase está pensada para ser heredada es que todas sus operaciones son virtuales y, en concreto, `getFicha` es abstracta. Las operaciones virtuales deben ser implementadas, pero `getFicha` no.

Es importante también que te percales de que esta clase hereda de **`TArchivable`**, que también es abstracta poseyendo 4 operaciones abstractas: **`guardarEnArchivo`**, **`clonar`**, **`recuperarDeArchivo`** y **`recuperarLista`**. Estas operaciones están presentes en `TAnimal`, pero si miras puedes darte cuenta de que no figuran en el diseño. Esto quiere decir que las estamos heredando manteniendo su abstracción, es decir, no las implementamos. Por esto es que **`TAnimal`** es abstracta (hereda operaciones abstractas que no implementa y además define su propia operación abstracta `getFicha`).

2.2.3 Clase `TMascota`

En este sistema habrá, como ya viste en su descripción, *especies de animales*, *mascotas* y *personas*. Veamos cómo se define una mascota para cumplir con los requisitos del sistema:



Primero que nada se indica que **TMascota** hereda de **TAnimal**, así que tenemos todos los atributos y operaciones que esta clase definía, más lo que agrega **TMascota**. Además, de entrada ya ves que **TMascota** no es abstracta, así que implementará todas las operaciones abstractas que hereda, tanto de **TAnimal** como de **TArchivable** (porque **TAnimal** hereda de **TArchivable**). Por este motivo puedes ver todas esas operaciones listadas en el diseño con el indicador `override`.

Veamos ahora la particularidad de la operación `setAdoptado`, cuyo encabezado Pascal te dejo completo a continuación:

```
procedure setAdoptado(a: boolean; pIdDuenno: string = '');;
```

El parámetro `pIdDuenno` es de tipo `string` y luego hay un signo de igual seguido de un valor `string`. Es posible en Pascal dar valores por defecto a los argumentos, por ejemplo:

```
function asignar(x: byte; verdad: boolean = false);
```

En este ejemplo, el argumento booleano `verdad` tiene un valor por defecto, `false`. Esto quiere decir que si al invocar a `asignar` no se le otorga valor a dicho argumento, automáticamente Pascal le dará el valor `false`. Por tanto, para invocar a `asignar` puedo omitir dicho parámetro, siendo estos dos ejemplos válidos:

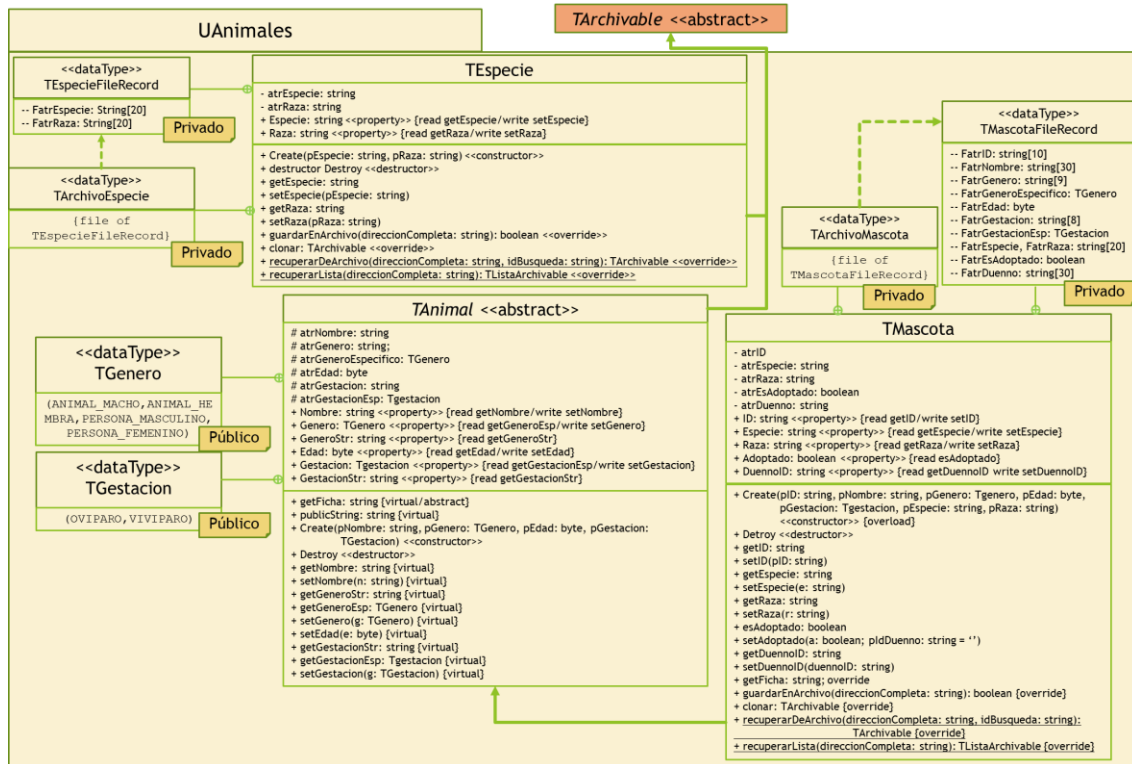
```
✓ asignar(10,true);  
✓ asignar(10);
```

Uno podría pensar que en el segundo ejemplo falta un argumento para invocar a `asignar`, pero realmente no es así, porque cuando un parámetro tiene un valor por defecto se convierte en un parámetro opcional. En nuestro caso, `setAdoptado` asigna por defecto el `string` vacío a `pIdDuenno` si no se pasa ningún valor al invocarla.

Finalmente ten en cuenta que la operación `getFicha` debe retornar el `string` que es mostrado en la ventana del sistema cuando se selecciona una mascota. Por ejemplo:

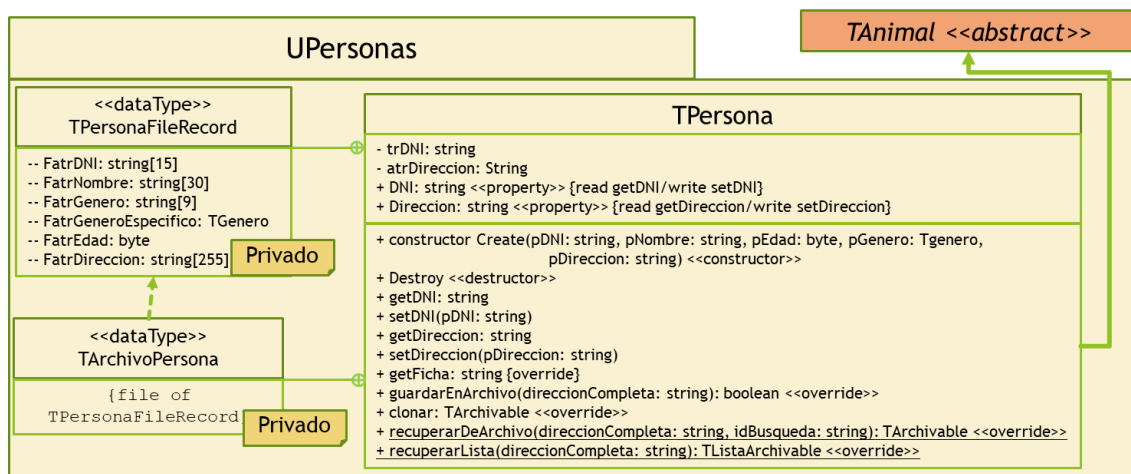
```
ID: 568  
Especie: Ave  
Raza: Loro común  
Nombre: Pepe  
Edad: 4  
Genero: Macho  
Tipo: Ovíparo  
Adoptado: NO
```

2.2.4 Diagrama final de UAnimales



2.3 UPersonas y TPersona

Esta unidad es la más simple de todas, ya que define una única clase que no es más compleja que TMascota o TEspece:



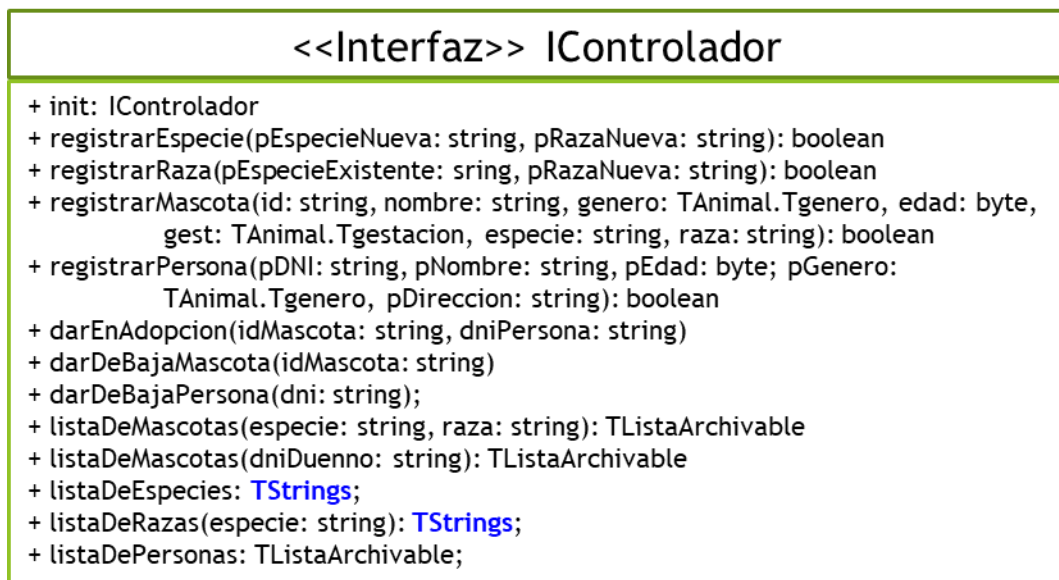
Tienes dos tipos internos privados, TPersonaFileRecord y TArchivoPersona que los usarás para poder implementar las operaciones de guardado heredadas de TArchivable. Luego tienes dos simples atributos y dos propiedades junto las operaciones básicas. El resto es todo heredado de TAnimal y ya está implementado, salvo getFicha que en esta implementación realmente no es usada, aunque la idea original era que retornara algo similar a getFicha en TMascota. Implementála como quieras.

2.4 UControlador

Esta unidad provee los servicios del sistema, es decir, las funcionalidades que éste brinda para poder ser ejecutado. Estos servicios serán consumidos por la interfaz gráfica de usuario. Además, el diseño es independiente de la implementación, y por tanto este modelo sirve para crear una nueva interfaz, como podría ser para una aplicación Web, o inclusive un programa que se utilice por consola mediante comandos.

2.4.1 Interfaz IControlador



Para lograr que los servicios del sistema sean reutilizables por cualquiera que quiera brindar un mecanismo para el usuario (otra GUI distinta, una aplicación Web, una aplicación Android, una consola, etc.) nos valdremos de una interfaz, que será la encargada de definir las operaciones y tipos de datos disponibles. Esta interfaz es llamada `IControlador` dada por este diseño:



Si te fijas, las interfaces en UML se presentan igual que las clases solo que se aclara mediante el indicador `<<interfaz>>` o en inglés `<<interface>>` que se trata justamente de una interfaz. En este caso particular no hay atributos definidos en `IControlador` y por tanto solo he listado las operaciones que esta provee.

Tu trabajo aquí es simplemente definir `IControlador` en Pascal con todo lo que ya aprendiste sobre interfaces.

A continuación te dejo el detalle de lo que debe hacer cada operación:

-  **init:** Inicializa todo lo necesario para comenzar a utilizar el sistema y retorna un objeto listo para invocar a las operaciones.
-  **registrarEspecie:** Registra una nueva especie vinculándola a la nueva raza creada. Si ya existe una especie con el nombre indicado entonces se registrará la raza nueva vinculada a dicha especie. En ambos casos se retorna `TRUE`. En caso de que ya exista la especie y la raza, vinculadas entre sí, no se registrará nada y se retornará `FALSE`.

- ✚ **registrarRaza:** Registra la nueva raza vinculada a la especie existente. Si no existe la especie indicada esta se registrará junto con la nueva raza invocando a `'registrarEspecie(string, string):boolean'` y se retornará `TRUE`. Si existe la especie indicada y no existe la raza, entonces se hará el registro vinculando la raza a la especie existente y se retornará `TRUE`. Si existe la especie y la raza, no se hará nada y se retornará `FALSE`.
- ✚ **registrarMascota:** Se registra una nueva mascota en el sistema. No debe existir una mascota con el `ID` indicado; en tal caso se retorna `FALSE`. Si el registro se hace de forma exitosa se retorna `TRUE`.
- ✚ **registrarPersona:** Se registra una nueva persona con el `DNI` y el resto de datos indicados. No debe existir una persona dicho `DNI` en el sistema; en tal caso se retorna `FALSE` y no se hace nada. Si no existe en el sistema una persona con el `DNI` indicado en `pDNI` se hará el registro y se retornará `TRUE`.
- ✚ **darEnAdopcion:** No debes implementarla. Deja un código vacío que compile.
- ✚ **darDeBajaMascota:** Ídem anterior.
- ✚ **darDeBajaPersona:** Ídem anterior.
- ✚ **listaDeMascotas:** Se retorna una lista con todas las Mascotas cuya especie y raza sean las indicadas. La lista estará vacía si no hay mascotas que cumplan con estas condiciones. Las mascotas retornadas serán copias de las que están en memoria, por tanto podrán luego ser eliminadas sin problemas ni repercusiones en la lista de mascotas del sistema. Las mascotas estarán en la lista como objetos `TArchivable`, por lo cual se requerirá casteo para verlas como `TMascota`.
- ✚ **listaDeMascotas:** Se retorna la lista de mascotas cuyo dueño sea la persona dada por el `DNI` indicado. Si no hay mascotas que cumplan esta condición la lista estará vacía. Las mascotas retornadas serán copias de las que están en memoria, por tanto podrán luego ser eliminadas sin problemas ni repercusiones en la lista de mascotas del sistema. Las mascotas estarán en la lista como objetos `TArchivable`, por lo cual se requerirá casteo para verlas como `TMascota`.
- ✚ **listaDeEspecies:** Se retorna una lista de las especies existentes en el sistema, sin que se repitan los nombres. La lista estará vacía si no hay especies registradas
- ✚ **listaDeRazas:** Se retorna una lista de razas en el sistema vinculadas a la especie indicada. Si la especie no existe, o bien, si no hay razas vinculadas a ella, la lista estará vacía.
- ✚ **listaDePersonas:** Se retorna una lista con todas las personas registradas en el sistema. Si no hay ninguna persona registrada, la lista resultante será vacía.

2.4.2 Clase TControlador

| TControlador <<singleton>> |
|---|
| <ul style="list-style-type: none"> - atrListaEspeciesStr: TStringList - atrListaRazasStr: TStringList - atrListaEspeciesObj: TListaArchivable - atrListaMascotas: TListaArchivable - atrListaPersonas: TListaArchivable - atrInstance: TControlador |
| <ul style="list-style-type: none"> - Create <<constructor>> + GetInstancia: TControlador + init: IControlador + registrarEspecie(pEspecieNueva: string, pRazaNueva: string): boolean + registrarRaza(pEspecieExistente: string, pRazaNueva: string): boolean + registrarMascota(id: string, nombre: string, genero: TAnimal.TGenero, edad: byte, gest: TAnimal.Tgestacion, especie: string, raza: string): boolean; + registrarPersona(pDNI: string, pNombre: string, pEdad: byte, pGenero: TAnimal.Tgenero, pDireccion: string): boolean + darEnAdopcion(idMascota: string, dniPersona: string) + darDeBajaMascota(idMascota: string) + darDeBajaPersona(dni: string) + listaDeMascotas(especie, raza: string): TListaArchivable + listaDeMascotas(dniDuenno: string): TListaArchivable + listaDeEspecies: TStrings + listaDeRazas(especie: string): TStrings + listaDePersonas: TListaArchivable |

Esta clase implementa a la interfaz **IControlador**, por tanto tiene todas las operaciones que ésta define y agrega además una nueva, pública y estática: **GetInstancia**. Además se especifica claramente que esta clase hace uso del patrón de diseño **singleton**, para que solo pueda existir una única instancia de ella. Por este motivo es que el constructor está definido como operación privada y no como público (Lazarus te dará una advertencia por esto, pero no le prestaremos atención).

De todos los atributos que incluye esta clase, el que tiene que ver con el patrón **singleton** es **atrInstance**, que es estático. Si no sabes cómo implementar esto vuelve a ver la clase del curso en la cual hablo de este patrón de diseño específicamente.

NOTA: este patrón requiere que incluyas un bloque de inicialización en esta unidad en el cual inicialices el atributo **atrInstance** como **nil**.

2.4.3 TStrings y TStringList

Verás que las operaciones **listaDeEspecies** y **listaDeRazas** retornan objetos de tipo **TStrings**. También verás que los atributos de **TControlador**, **atrListaEspeciesStr** y **atrListaRazasStr** son de tipo **TStringList**.

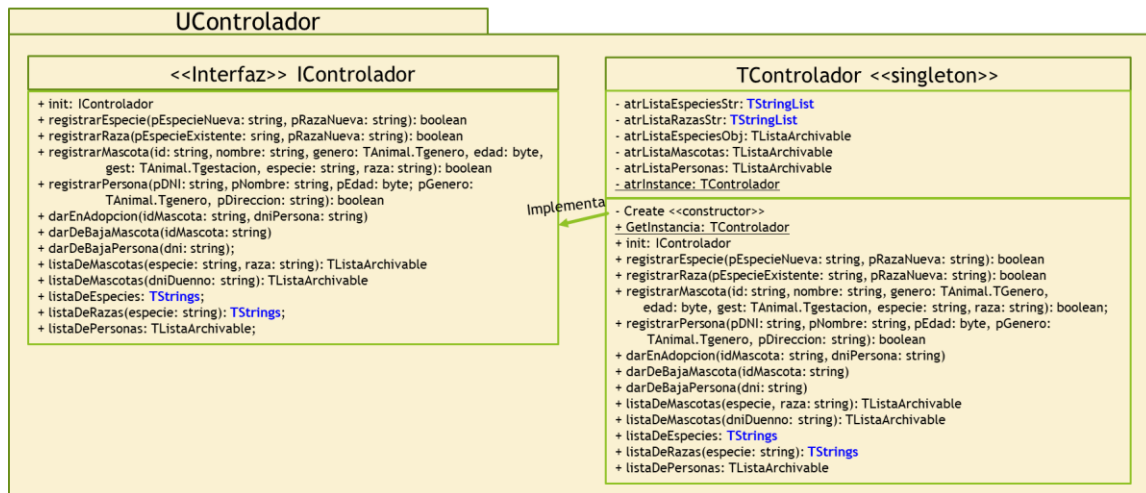
Estas dos clases están disponibles en la unidad **Classes**, por lo tanto deberás importarla para poder hacer uso de ellas, y no tienes más remedio que utilizarlas porque el diseño del sistema así lo requiere. Esto es porque los componentes gráficos de Lazarus utilizan **TStrings** y **TStringList** para mostrar texto. **TStringList** hereda de **TStrings**, lo cual debes saber.

Tú trabajarás con `TStringList` ya que retornar un objeto de este tipo implica retornar uno de tipo `TStrings`. Tu tarea es leer la documentación de estas clases para poder darles uso, ya que como programador/a en el mundo real te enfrentarás a esto constantemente: leer cómo funcionan ciertas herramientas necesarias para un proyecto.

Te dejo aquí un enlace por el cual puedes comenzar:

https://wiki.freepascal.org/TStringList-TStrings_Tutorial/es

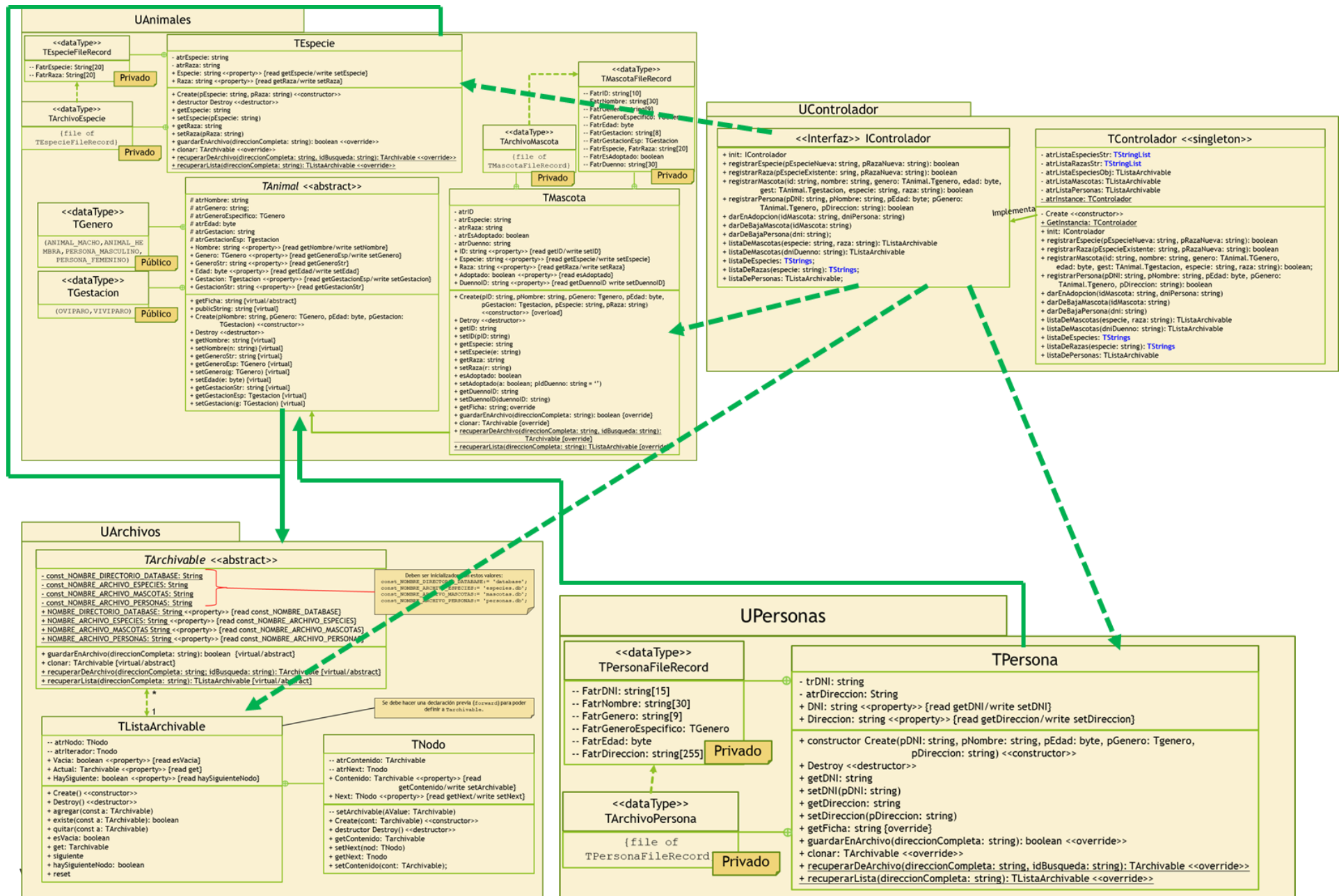
2.4.4 Diagrama final de UControlador



El único agregado apreciable al mirar el diagrama completo es que hay una flecha sólida que une a `IControlador` con `TControlador`, que en condiciones normales indica herencia, pero sobre la flecha se agrega la aclaración **implementa**, para indicar que justamente `TControlador` implementa la interfaz `IControlador`. Ya sabes que implementar una interfaz es similar a heredarla, ya que todo objeto de tipo `TControlador` también es de tipo `IControlador`.

2.5 Diagrama final

En la siguiente página te dejo una ilustración del diagrama completo del sistema sin incluir los componentes de la interfaz gráfica de usuario ya que esto está todo implementado de antemano. Verás que cuando se mira un diagrama UML completo resulta un tanto sobrecargado e intimidante, pero si lee detenidamente ayuda mucho a trabajar en el sistema. Normalmente estos diagramas siempre están acompañados de un documento de diseño, similar a esto que te he dado aquí, donde se detalla todo lo necesario para poder traducir el dibujo UML a un lenguaje de programación concreto (o varios).



3. Entrega final

Tu trabajo es tomar el diseño y las especificaciones que te he dado en este documento para implementar el sistema de forma que quede funcional. Te daré el código de la interfaz gráfica ya listo y tú deberás crear los archivos de las unidades correspondientes, agregarlos al proyecto y lograr que todo encaje. En concreto debes:

- Crear la unidad UArchivos
 - Crear e implementar TArchivable
 - Crear e implementar TListaArchivable junto con su clase interna TNode.
- Crear la unidad UAnimales
 - Crear e implementar TEspecie
 - Crear e implementar TAnimal
 - Crear e implementar TMascota
- Crear la unidad UPersonas
 - Crear e implementar TPersona
- Crear la unidad UControlador
 - Crear la interfaz IControlador
 - Crear e implementar TControlador

Está de más decir que debes ceñirte al diseño tal cual se ha descrito, respetando clases abstractas, herencia y polimorfismo, todo como se te ha indicado.

Son en total cuatro los archivos que debes enviar, aunque puedes enviar todo el proyecto completo. Lo ideal es que comprimas los archivos en formato ZIP, RAR o 7Z para enviar más fácilmente.

Envía todo a bedelia@kaedusoft.edu.uy con el asunto **PROYECTO VETERINARIA SIMPLE [NOMBRE] [APELLIDO]**. Ten en cuenta que podrás, como siempre, contactarme directamente al WhatsApp **+598 94 815 035**.